Elasticsearch API Guide 中文版

Table of contents:

字段匹配 match
- operator
- minimum_should_match
- Fuzziness
- Zero terms query
- cutoff_frequency

多字段匹配 multi_match
- 提升字段权重
- 算分类型
  - 1. best_fields
  - 2. most_fields
  - 3. cross_fields
  - 4. phrase 和 phrase_prefix

短语查询 match_phrase
  - 核心参数slop： 位置距离容差值
  - 指定分析器

短语前缀查询 match_phrase_prefix

关键字匹配 term
- Term Query 关键词查询
- Terms Query 包含多个词
- Terms Set Query 集合类查询

范围匹配 range
- 日期字段的范围查询
- 日期字段范围查询 - Date Format
- 日期字段范围查询 - 指定时区

是否存在 exists
  - 查找具有空值的文档

前缀匹配 prefix

通配符匹配 wildcard

正则匹配 regexp
  - 支持算分
  - 您还可以使用特殊标志

模糊匹配 fuzzy
  - 参数

主键匹配 ids

跨度匹配 span
- Span Queries
  - Span Term Queries

    - Span Multi Term Queries
    - Span First Query
    - Span Near Query
    - Span Or Query
    - Span Not Query
    - Span Containing Query
    - Span Within Query
    - Span Field Masking Query
布尔查询 bool
- 使用 minimum_should_match
常量算分 constant_score
- filter
- boost
最佳匹配 dis_max
- 算分公式
子句加权 boosting
- 数据准备
- 1. 基础算分
- 2. 影响算分
- 解释
函数算分 function_score
- 单函数样例
- 组合多个函数
- 内置函数
    - script_score
    - weight
    - random_score
    - field_value_factor
    - decay
边界查询 geo_bounding_box
- 示例 查询位于指定矩形中的文档
    - 多种查询方式示例
    - type
- 参数说明
    - Ignore Unmapped
    - 精度说明
距离查询 geo_distance
- 例子 查询200km半径内的位置

基础函数聚合 avg,sum,max,min..

- Avg Aggregation 取平均值
- Sum Aggregation 取总和值
- Max Aggregation 取最大值
- Min Aggregation 取最小值
- Script
- Value Script
- Missing value

加权平均值 weighted_avg

- Weighted Avg Aggregation
- 示例
- Script
- Missing values
- 参数说明
  - weighted_avg
  - value
  - weight

基数聚合 cardinality

- Cardinality Aggregation
  - 精度控制
  - 近似值计数
  - 预计算 hash
  - Script
  - Missing value

地理边界聚合 geo_bounds

- Geo Bounds Aggregation

地理重心聚合 geo_centroid

- Geo Centroid Aggregation

百分比聚合 percentiles

- Percentiles Aggregation
- Keyed Response
- Script
- 百分比（通常）是近似值
- 压缩
- HDR直方图
- Missing Value

百分比排名聚合 percentile_ranks

- Percentile Ranks Aggregation

- Keyed Response
- Script
- HDR Histogram

脚本度量聚合 scripted_metric

- Scripted Metric Aggregation
- 允许的返回类型
- 脚本的Scope

统计聚合 stats

- Stats Aggregation
- Script
- Value Script
- Missing value

扩展统计聚合 extended_stats

- Extended Stats Aggregation
- 标准偏差界限
- Script
- Value Script
- Missing Value

排行榜聚合 top_hits

- Top Hits Aggregation
- Example
- 字段折叠示例 (Field collapse)
- 嵌套或反向嵌套聚合器中的top_hits支持

值计数聚合 value_count

- Value Count Aggregation
- Script

中值绝对偏差聚合 median_absolute_deviation

- Median Absolute Deviation Aggregation
- 实例
- 近似值 Approximation
- Script
- Missing value

关键词聚合 Terms

- Terms Aggregation
- Size
- Shard Size
- Order
- Minimum document count

- Script
- Value Script
- Filtering Values
  - Filtering Values 使用正则表达式
  - Filtering Values with exact values
  - Filtering Values with partitions
- 多字段 terms aggregation
- 收集模式
- Execution hint

直方图 histogram

- Histogram Aggregation
- 最小文档数
- Order
- Offset
- Response Format
- Missing value

日期直方图 date_histogram

- Date Histogram Aggregation
- 示例
- Timezone
- Offset
- Keyed Response
- Scripts
- Missing value
- Order
- 使用脚本按星期几聚合

自动间隔日期直方图 auto_date_histogram

- Auto-interval Date Histogram Aggregation
- Keys
- Intervals
- Time Zone
- Script
- Missing value

范围聚合 range

- Range Aggregation
- Keyed Response
- Script
- Value Script

- Sub Aggregations

日期范围聚合 date_range

- Date Range Aggregation
- Missing Values
- Date Format/Pattern
- Time zone
- Keyed Response

IP范围聚合 ip_range

- IP Range Aggregation
- Keyed

筛选聚合 filter

- Filter Aggregation

多筛选聚合 filters

- Filters Aggregation
- Anonymous filters
- Other Bucket

地理距离聚合 geo_distance

- Geo Distance Aggregation
- Keyed Response

GeoHash网格聚合 geohash_grid

- GeoHash grid Aggregation
- Low-precision 低精度请求
- High-precision 高精度请求
- 赤道处的单元尺寸

父聚合 parent

- Parent Aggregation

子聚合 children

- Children Aggregation

嵌套聚合 nested

- Nested Aggregation

反向嵌套聚合 reverse_nested

- Reverse nested Aggregation

复合聚合 composite

- Composite Aggregation
- Values source
- Terms
- Histogram
- Date Histogram

- Format
- Time Zone
- Mixing different values source
- Order
- Missing bucket
- Size
- After
- Sub-aggregations

全局聚合 global

- Global Aggregation

缺值聚合 Missing

- Missing Aggregation

采样器聚合 sampler

- Sampler Aggregation
- shard_size
- 局限性

多样化采样器聚集 diversified_sampler

- Diversified Sampler Aggregation
- 示例用例：
- Scripted example:
- shard_size
- max_docs_per_value
- execution_hint
- 局限性
- 有限的重复数据消除逻辑。
- geo/date字段没有专门的语法

邻接矩阵聚合 adjacency_matrix

- Adjacency Matrix Aggregation
- 使用
- 局限性

PipelineAgg介绍

- Pipeline Aggregation Overvew
  - Parent (父级)
  - Sibling（同级）
- buckets_path 语法
- Special Paths
- Dealing with dots in agg names
- Dealing with gaps in the data

- EWMA (Exponentially Weighted)
- Holt-Linear
- Holt-Winters
- "Cold Start"
- Additive Holt-Winters
- Multiplicative Holt-Winters
- Prediction
- Minimization

移动函数聚合 moving_fn

- Moving Function Aggregation
- 语法
- Custom user scripting
- Pre-built Functions
  - max
  - min
  - sum
  - stdDev
  - unweightedAvg
  - linearWeightedAvg
  - ewma
  - holt
  - holtWinters

矩阵统计聚合 matrix_stats

- Matrix Stats
- 多值字段
- Missing Values
- Script

使用分词器

- 简单测试
- 分析多个文本
- 过滤器处理
- 前置处理
- 自定义tokenizer和filter
- 在指定的索引上分析
- 分词组件概念
- Elasticsearch 内置的分词器
- 纳速云附加安装的分词器

Standard Analyzer

Elasticsearch API

Elasticsearch 的 RESTful API 使用 HTTP 作为传输协议，遵循以下标准：

- 使用 JSON 作为数据交换格式。
- 使用标准HTTP响应CODE。
- 所有的语言都可以使用 RESTful API，通过 9200 端口的与云端 Elasticsearch 进行通信。
- RESTful API调用是无状态的，您发出的每一个请求都与其他请求无关。

API 调试

- 本手册中的所有示例代码都可以直接粘贴到 Kinana 的 `开发工具` 运行，这是最便捷的调试方式。
- 若缺少环境 ➡️ 帮助您5秒创建 Elasticsearch/Kibana

SSL/TLS

Nasu Elasticsearch Serverless 采用基于 HTTPS and Basic Authentication 的身份验证，例如：

- 命令行方式

```
curl -XPOST 'https://router.nasuyun.com:9200/logs/_doc' -H 'Content-Type:
application/json' \
-u your_username:your_password  \
-d '
{
  "timestamp": "2018-01-24 12:34:56",
  "message": "User logged in",
  "user_id": 4,
```

```
  "admin": false
}'
```

- java方式

POM依赖

```
<dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch</artifactId>
    <version>6.8.23</version>
</dependency>
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-high-level-client</artifactId>
    <version>6.8.23</version>
</dependency>
```

```
private RestHighLevelClient client() {
    Header[] headers = new Header[]{new BasicHeader(HttpHeaders.CONTENT_TYPE,
"application/json")};
    final CredentialsProvider credentialsProvider = new BasicCredentialsProvider();
    credentialsProvider.setCredentials(AuthScope.ANY, new
UsernamePasswordCredentials(your_username, your_password));
    RestClientBuilder builder = RestClient.builder(HttpHost.create(server))
            .setDefaultHeaders(headers)
            .setHttpClientConfigCallback(httpClientBuilder ->
httpClientBuilder.setDefaultCredentialsProvider(credentialsProvider));
    return new RestHighLevelClient(builder);
}
```

索引写入 bulk

创建索引

创建一个索引，包含2个分片1份副本(settings)，指定了2个字段类型(mappings) 。

```
PUT greeting
{
    "settings" : {
      "number_of_shards" : 2,
      "number_of_replicas" : 1
    },
    "mappings" : {
        "_doc" : {
            "properties" : {
                "email" : { "type" : "keyword" },
                "message" : { "type" : "text" }
            }
        }
    }
}
```

写入索引（单条）

```
POST greeting/_doc
{
  "email" : "greeting@nasuyun.com",
  "message" : "hello world"
}
```

写入索引（指定ID）

```
POST greeting/_doc/1
{
  "email" : "greeting@nasuyun.com",
  "message" : "hello world"
}
```

Bulk 批量写入

```
POST greeting/_bulk
{ "index" : { "_type" : "_doc"} }
{ "email" : "g1@nasuyun.com","message":"hello1" }
{ "index" : { "_type" : "_doc"} }
{ "email" : "g2@nasuyun.com","message":"hello2" }
{ "index" : { "_type" : "_doc"} }
{ "email" : "g3@nasuyun.com","message":"hello3" }
{ "index" : { "_type" : "_doc"} }
{ "email" : "g4@nasuyun.com","message":"hello4" }
```

更复杂的一个例子，在body内指定 _index, _type , _id , 并增加了删除更新等操作。

```
POST _bulk
{ "index" : { "_index" : "test", "_type" : "_doc", "_id" : "1" } }
{ "field1" : "value1" }
{ "delete" : { "_index" : "test", "_type" : "_doc", "_id" : "2" } }
{ "create" : { "_index" : "test", "_type" : "_doc", "_id" : "3" } }
{ "field1" : "value3" }
{ "update" : {"_id" : "1", "_type" : "_doc", "_index" : "test"} }
{ "doc" : {"field2" : "value2"} }
```

获取文档 get

获取单个文档

```
GET greeting/_doc/1
```

返回

```
{
  "_index" : "greeting",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 1,
  "_seq_no" : 3,
  "_primary_term" : 1,
  "found" : true,
  "_source" : {
    "email" : "greeting@nasuyun.com",
    "message" : "hello world"
  }
}
```

获取多个文档

```
GET /_mget
{
    "docs" : [
        {
            "_index" : "greeting",
            "_type" : "_doc",
            "_id" : "1"
        },
        {
            "_index" : "greeting",
            "_type" : "_doc",
            "_id" : "2"
        }
    ]
}
```

返回


```
{
  "docs" : [
    {
      "_index" : "greeting",
      "_type" : "_doc",
      "_id" : "1",
      "_version" : 1,
      "_seq_no" : 3,
      "_primary_term" : 1,
      "found" : true,
      "_source" : {
        "email" : "greeting@nasuyun.com",
        "message" : "hello world"
      }
    },
    {
      "_index" : "greeting",
      "_type" : "_doc",
      "_id" : "2",
      "found" : false
    }
  ]
}
```


Realtime

默认情况下，获取 API 是实时的，不受索引刷新率的影响（当数据对搜索可见时）。如果文档已更新但尚未刷新，则获取 API 将就地发出刷新调用以使文档可见。这也将使自上次刷新以来更改的其他文档可见。为了禁用实时 GET，可以将 realtime 参数设置为 false。

```
GET greeting/_doc/1?realtime=false
```

Source 过滤

默认情况下，get 操作返回 _source 字段的内容，除非您使用了 stored_fields 参数或者 _source 字段被禁用。您可以使用 _source 参数关闭 _source 检索

```
GET greeting/_doc/1?_source=false
```

如果你只需要完整_source中的一两个字段，你可以使用_source_includes和_source_excludes参数来包含或过滤掉你需要的部分。这对于部分检索可以节省网络开销的大型文档尤其有用。两个参数都采用逗号分隔的字段列表或通配符表达式。例子

```
GET greeting/_doc/1?_source_includes=*message&_source_excludes=email
```

简单查询 query string

何处使用

1. 查询q参数方式

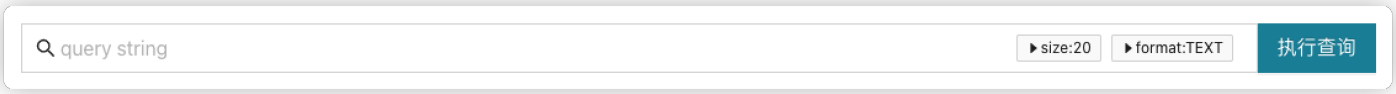
`GET /_search?q=user:kimchy`

`GET /kimchy,elasticsearch/_search?q=tag:wow`

2. 查询body方式

```
GET /_search
{
    "query": {
        "query_string" : {
            "query" : "this AND that OR thus"
        }
    }
}
```

3. 控制台数据集查询框采用 query string 可快速调试



基础用法

1. 关键词
最简单的关键词匹配

`hello`

2. 逻辑关系
带运算符的关键词匹配

`this AND that OR thus`

```
(new york city) OR (big apple)
```

3. 指定字段
其中status字段包含active

```
status:active
```

```
title:(quick OR brown)
```

```
title:(quick brown)
```

4. 包含短语
其中作者字段包含确切的短语“john smith”

```
author:"John Smith"
```

5. 通配符字段
其中任何字段 book.title、book.content 或 book.date 包含 quick 或 brown （注意我们需要如何用反斜杠转义 *）

```
book.\*:(quick brown)
```

6. 带关系运算的字段匹配

```
(content:this OR name:this) AND (content:that OR name:that)
```

7. 范围查找

```
date:[2012-01-01 TO 2012-12-31]
```

```
count:[1 TO 5]
```

```
count:[10 TO *]
```

```
date:{* TO 2012-01-01}
```

进阶用法

1. 关键词权重
如下例quick会在bm25算分的结果上乘以2， 因此会有更高的排名。

```
quick^2 fox
```

算分提升也可以应用于短语或组

```
"john smith"^2 (foo bar)^4
```

其他用法

1. 模糊查询
我们可以使用“模糊”运算符搜索与我们的搜索词相似但不完全相同的词：

```
quikc~ brwn~ foks~
```

2. 通配符匹配

```
*hello*world*
```

3. 正则表达式匹配

```
name:/joh?n(ath[oa]n)/
```

> **TIP**
> 使用提示： 以上三种用法需谨慎使用，在大数据量场景不会使用到倒排索引，因而产生极高的计算量及性能损耗。

查询参数 search body

From/Size

可以使用from和size参数对结果进行分页，from参数定义要获取的第一个结果的偏移量。size参数指定返回的最大文档数。

```
GET /_search
{
    "from" : 0, "size" : 10,
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

> `from + size`总和不能超过`10000`

Sort 排序

允许您在特定字段上添加一个或多个排序，每种类型也可以反转。

```
GET /my_index/_search
{
    "sort" : [
        { "post_date" : {"order" : "asc"}},
        "user",
        { "name" : "desc" },
        { "age" : "desc" },
        "_score"
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

参数	说明

参数	说明
asc	按升序排序
desc	按降序排序

Source filtering 源字段过滤

通过_source控制字段展示。

1. 关闭_source

```
GET /_search
{
    "_source": false,
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

2. 指定通配符只看部分字段

```
GET /_search
{
    "_source": "obj.*",
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

3. 数组方式

```
GET /_search
{
    "_source": [ "obj1.*", "obj2.*" ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

4. 包含与排除

```
GET /_search
{
    "_source": {
        "includes": [ "obj1.*", "obj2.*" ],
        "excludes": [ "*.description" ]
    },
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Doc value Fields

允许为每次命中返回字段的doc_value，例如：

```
GET /_search
{
    "query" : {
        "match_all": {}
    },
    "docvalue_fields" : [
        {
            "field": "my_ip_field",
            "format": "use_field_mapping"
        },
        {
            "field": "my_date_field",
            "format": "epoch_millis"
        }
    ]
}
```

同时也支持通配符模式

```
GET /_search
{
    "query" : {
        "match_all": {}
    },
    "docvalue_fields" : [
        {
```

```
            "field": "*field",
            "format": "use_field_mapping"
        }
    ]
}
```

Post filter 后置过滤

搜索结果过滤

```
GET /shirts/_search
{
  "query": {
    ...
  },
  "aggs": {
    ...
  },
  "post_filter": {
    "term": { "color": "red" }
  }
}
```

Rescoring 二次算分

二次算分：对query和post_filter阶段返回的Top-K结果执行第二次查询，原始查询和二次查询的分数线性组合，以生成每个文档的最终_score。

```
POST /_search
{
   "query" : {
      "match" : {
         "message" : {
            "operator" : "or",
            "query" : "the quick brown"
         }
      }
   },
   "rescore" : {
      "window_size" : 50,
      "query" : {
         "rescore_query" : {
```

```
            "match_phrase" : {
               "message" : {
                  "query" : "the quick brown",
                  "slop" : 2
               }
            }
         },
         "query_weight" : 0.7,
         "rescore_query_weight" : 1.2
      }
   }
}
```

可以使用score_mode控制分数的组合方式：

Score Mode	描述
total	添加原始分数和重新搜索查询分数。默认值。
multiply	将原始分数乘以重新搜索查询分数。用于函数查询重新搜索。
avg	平均原始分数和重新搜索查询分数。
max	取原始分数和重新搜索查询分数的最大值。
min	取原始分数和重新搜索查询分数的最小值。

乘法实例

```
POST /_search
{
   "query" : {
      "match" : {
         "message" : {
            "operator" : "or",
            "query" : "the quick brown"
         }
      }
   },
   "rescore" : [ {
      "window_size" : 100,
      "query" : {
         "rescore_query" : {
```

```
            "match_phrase" : {
               "message" : {
                  "query" : "the quick brown",
                  "slop" : 2
               }
            }
         },
         "query_weight" : 0.7,
         "rescore_query_weight" : 1.2
      }
   }, {
      "window_size" : 10,
      "query" : {
         "score_mode": "multiply",
         "rescore_query" : {
            "function_score" : {
               "script_score": {
                  "script": {
                    "source": "Math.log10(doc.likes.value + 2)"
                  }
               }
            }
         }
      }
   } ]
}
```

Explain 解释

解释查询的得分计算方式。

```
GET /_search
{
    "explain": true,
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

Profile 测量

Profile 提供了有关搜索请求中各个组件执行的详细耗时信息

> 注意，Profile API不测量网络延迟、搜索获取阶段花费的时间、请求在队列中花费的时间或在协调节点上合并shard响应时花费的时间。

```
GET /twitter/_search
{
  "profile": true,
  "query" : {
    "match" : { "message" : "some number" }
  }
}
```

Highlighting 高亮

示例1：指定`content`字段高亮

```
GET /_search
{
    "query" : {
        "match": { "content": "kimchy" }
    },
    "highlight" : {
        "fields" : {
            "content" : {}
        }
    }
}
```

示例2：所有字段高亮，设置tag。

```
GET /_search
{
    "query" : {
        "match": { "user": "kimchy" }
    },
    "highlight" : {
        "pre_tags" : ["<tag1>"],
        "post_tags" : ["</tag1>"],
        "fields" : {
            "_all" : {}
        }
    }
}
```

Elasticsearch支持三种高亮显示：：`unified`, `plain`, `fvh（快速矢量高亮显示）`。 更多示例

多个搜索请求 msearch

Multi Search API

_msearch API允许在同一API中执行多个搜索请求。

请求的格式类似于批量API格式，并使用换行分隔的JSON（NDJSON）格式。结构如下

```
header\n
body\n
header\n
body\n
```

> **TIP**
>
> 注意：数据的最后一行必须以换行符\n结尾。每个换行符前面都可以加上回车符\r。向该接口发送请求时，Content-Type头应设置为application/x-ndjson。

头部包括要搜索的index、search_type、preference、routing。正文包括典型的搜索正文请求（包括query, aggregations, from, size等）。下面是一个示例：

```
$ cat requests
{"index" : "test"}
{"query" : {"match_all" : {}}, "from" : 0, "size" : 10}
{"index" : "test", "search_type" : "dfs_query_then_fetch"}
{"query" : {"match_all" : {}}}
{}
{"query" : {"match_all" : {}}}

{"query" : {"match_all" : {}}}
{"search_type" : "dfs_query_then_fetch"}
{"query" : {"match_all" : {}}}
```

```
$ curl -H "Content-Type: application/x-ndjson" -XGET localhost:9200/_msearch --
data-binary "@requests"; echo
```

> **NOTE**
>
> 注意，上面包括一个`empty header`的示例（也可以只是没有任何内容）这是支持的。

响应返回一个响应数组，其中包含搜索响应和每个搜索请求的状态代码，这些搜索请求与原始多搜索请求中的顺序相匹配。如果该特定搜索请求完全失败，则将返回一个带有错误消息和相应状态代码的对象来代替实际的搜索响应。

接口还允许根据URI本身中的一个或多个索引进行搜索。例如：

```
GET twitter/_msearch
{}
{"query" : {"match_all" : {}}, "from" : 0, "size" : 10}
{}
{"query" : {"match_all" : {}}}
{"index" : "twitter2"}
{"query" : {"match_all" : {}}}
```

上面将针对所有未定义索引的请求执行twitter索引搜索，最后一个将针对twitter2索引执行。

search_type可以以类似的方式设置，以全局应用于所有搜索请求。

msearch的max_concurrent_searchs请求参数可用于控制多搜索api将执行的最大并发搜索数。

> **INFO**
>
> 请求参数max_concurrent_shard_requests可用于控制每个子搜索请求将执行的并发shard请求的最大数量。该参数应用于保护单个请求不使集群过载（例如，如果每个节点的shard数量较高，默认请求将命中集群中的所有索引，这可能会导致shard请求被拒绝）。此默认值基于集群中的数据节点数量，但最多为256个。在某些情况下，并行性无法通过并发请求实现，因此这种保护将导致性能不佳。例如，在预期并发搜索请求的数量非常少的环境中，将此值增加到更高的数量可能会有所帮助。

Template support

_msearch也提供了对模板的支持。按如下方式提交：

```
GET _msearch/template
{"index" : "twitter"}
```

```
{ "source" : "{ \"query\": { \"match\": { \"message\" : \"{{keywords}}\" } } } }",
"params": { "query_type": "match", "keywords": "some message" } }
{"index" : "twitter"}
{ "source" : "{ \"query\": { \"match_{{template}}\": {} } }", "params": {
"template": "all" } }
```

用于inline templates。

您还可以创建搜索模板

```
POST /_scripts/my_template_1
{
    "script": {
        "lang": "mustache",
        "source": {
            "query": {
                "match": {
                    "message": "{{query_string}}"
                }
            }
        }
    }
}
```

```
POST /_scripts/my_template_2
{
    "script": {
        "lang": "mustache",
        "source": {
            "query": {
                "term": {
                    "{{field}}": "{{value}}"
                }
            }
        }
    }
}
```

然后在_msearch中使用它们：

```
GET _msearch/template
{"index" : "main"}
{ "id": "my_template_1", "params": { "query_string": "some message" } }
```

```
{"index" : "main"}
{ "id": "my_template_2", "params": { "field": "user", "value": "test" } }
```

部分响应

为了确保快速响应，如果一个或多个shard失败，_msearch API将以部分结果进行响应。

滚动查询 scroll

Scroll API

Scroll API 通常用于翻页查询，其方式与在传统数据库中使用游标的方式大致相同，首先从第一次滚动查询获取结果及scroll_id。

> 参数 `scroll=1m` 保持scroll上下文1分钟

```
POST /twitter/_search?scroll=1m
{
    "size": 100,
    "query": {
        "match" : {
            "title" : "elasticsearch"
        }
    }
}
```

上述请求的结果包括一个_scroll_id，它应该传递给滚动API，以便检索下一批结果。

```
POST /_search/scroll
{
    "scroll" : "1m",
    "scroll_id" :
"DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAAD4WYm9laVYtZndUQlNsdDcwakFMNjU1QQ=="
}
```

query参数

```
POST /twitter/_search?scroll=1m
{
    "size": 100,
    "query": {
        "match" : {
            "title" : "elasticsearch"
        }
```

```
    }
}
```

排序参数

> **INFO**
>
> sort 会将结果集排序，但也会影响查询效率。

```
GET /_search?scroll=1m
{
  "sort": [
    "_doc"
  ]
}
```

清除scroll上下文

保持滚动打开是有代价的，因此，一旦不再使用滚动，应使用clear scroll API明确清除滚动

```
DELETE /_search/scroll
{
    "scroll_id" :
"DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAAD4WYm9laVYtZndUQlNsdDcwakFMNjU1QQ=="
}
```

清除多个scroll上下文

```
DELETE /_search/scroll
{
    "scroll_id" : [
      "DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAAD4WYm9laVYtZndUQlNsdDcwakFMNjU1QQ==",

"DnF1ZXJ5VGhlbkZldGNoBQAAAAAAAABFmtSWWRRWUJrU2o2ZExpSGJCVmQxYUEAAAAAAAAAAxZrUllkUVlC
    ]
}
```

清除所有scroll上下文

```
DELETE /_search/scroll/_all
```

Sliced Scroll

如果 scroll 查询返回的文档数量过多，可以把它们拆分成多个切片以便独立使用:

```
GET /twitter/_search?scroll=1m
{
    "slice": {
        "id": 0,
        "max": 2
    },
    "query": {
        "match" : {
            "title" : "elasticsearch"
        }
    }
}
GET /twitter/_search?scroll=1m
{
    "slice": {
        "id": 1,
        "max": 2
    },
    "query": {
        "match" : {
            "title" : "elasticsearch"
        }
    }
}
```

- id: 切片的 id
- max: 最大切片数量

上面的例子，第一个请求返回的是第一个切片（id : 0）的文档，第二个请求返回的是第二个切片的文档。因为我们设置了最大切片数量是 2 ，所以两个请求的结果等价于一次不切片的 scroll 查询结果。默认情况下，先在第一个分片（shard）上做切分，然后使用以下公式： slice(doc) = floorMod(hashCode(doc._uid), max) 在每个 shard 上执行切分。例如，如果 shard 的数量是 2 ，并且用户请求 4 slices ，那么 id 为 0 和 2 的 slice 会被分配给第一个 shard ，id 为 1 和 3 的 slice 会被分配给第二个 shard 。

每个 scroll 是独立的，可以像任何 scroll 请求一样进行并行处理。

> **NOTE**
>
> 如果 slices 的数量比 shards 的数量大，第一次调用时，slice filter 的速度会非常慢。它的复杂度时 O(n)，内存开销等于每个 slice N 位，其中 N 时 shard 中的文档总数。经过几次调用后，筛选器会被缓存，后续的调用会更快。但是仍需要限制并行执行的 sliced 查询的数量，以免内存激增。

为了完全避免此成本，可以使用另一个字段的 doc_values 来进行切片，但用户必须确保该字段具有以下属性：

该字段是数字类型 该字段启用了 doc_values 每个文档应当包含单个值。如果一份文档有指定字段的多个值，则使用第一个值 每个文档的值在创建文档时设置了之后不再更新，这可以确保每个切片获得确定的结果 字段的基数应当很高，这可以确保每个切片获得的文档数量大致相同

```
GET /twitter/_search?scroll=1m
{
    "slice": {
        "field": "date",
        "id": 0,
        "max": 10
    },
    "query": {
        "match" : {
            "title" : "elasticsearch"
        }
    }
}
```

> **NOTE**
>
> 默认情况下，每个 scroll 允许的最大切片数量时 1024。你可以更新索引设置中的 index.max_slices_per_scroll 来绕过此限制。

模板 template

ES中的模板概念有两处，我们加以区分 避免混淆：

- **dynamic_templates** 动态模板（字段级）：在索引的mapping内定义，如输入大量字段，在无法预期字段类型的情况下交由动态模板自动映射。
- **index_template** 索引模板（索引级）：由独立的 _template API 管理，允许您定义在创建新索引时自动应用的模板，你也可以在 `index_template` 内定义字段自动映射 `dynamic_templates`。

一、dynamic_templates

动态模板示例

```
PUT my_index
{
  "mappings": {
    "_doc": {
      "dynamic_templates": [
        {
          "strings": {
            "match_mapping_type": "string",
            "mapping": {
              "type": "keyword"
            }
          }
        }
      ],
      "properties": {
        "@timestamp": {
          "type": "date"
        }
      }
    }
  }
}
```

```
POST my_index/_doc
{
```

```
  "name":"jackma"
}
```

说明：my_index的新字段 `name` 将被映射为 `keyword`，而非默认的 `keyword and text`

> **NOTE**
>
> properties 内的字段定义优先级高于动态映射。

match_mapping_type 参数

- **string** 遇到字符串。
- **boolean** 遇到true或false时。
- **double** 遇到带有小数部分的数字。
- **long** 遇到没有小数部分的数字。
- **object** 遇到对象。
- **date** 当date_detection启用时，遇到匹配了配置的日期格式。

dynamic_templates 更多详细用法

二、index_template

索引模板允许您定义在创建新索引时自动应用的模板。这些模板包括`settings`和`mappings`，以及一个简单的`pattern`，用于控制模板是否应应用于新索引。

> **TIP**
>
> 模板仅在创建索引时应用。更改模板不会影响现有索引。使用创建索引API时，作为创建索引调用自定义的`settings/mappings`优先于模板中定义的任何匹配`settings/mappings`。

新建索引模板

```
PUT _template/template_1
{
  "index_patterns": ["te*", "bar*"],
  "settings": {
    "number_of_shards": 1
```

```
  },
  "mappings": {
    "_doc": {
      "_source": {
        "enabled": false
      },
      "properties": {
        "host_name": {
          "type": "keyword"
        },
        "created_at": {
          "type": "date",
          "format": "EEE MMM dd HH:mm:ss Z yyyy"
        }
      }
    }
  }
}
```

定义名为template_1的模板，模板模式为`te*`或`bar*`。设置和映射将应用于与`te*`或`bar*`模式匹配的任何索引名称。

也可以在索引模板中包含别名，如下所示

```
PUT _template/template_1
{
    "index_patterns" : ["te*"],
    "settings" : {
        "number_of_shards" : 1
    },
    "aliases" : {
        "alias1" : {},
        "alias2" : {
            "filter" : {
                "term" : {"user" : "kimchy" }
            },
            "routing" : "kimchy"
        },
        "{index}-alias" : {}
    }
}
```

索引模板内定义字段动态映射示例

```
{
  "index_patterns": [
    "log*"
  ],
  "order": 1000,
  "mappings": {
    "doc": {
      "dynamic_templates": [
        {
          "strings": {
            "match_mapping_type": "string",
            "mapping": {
              "type": "keyword"
            }
          }
        }
      ],
      "properties": {
        "@timestamp": {
          "type": "date"
        },
        "timestamp": {
          "type": "date"
        }
      }
    }
  },
  "settings": {
    "index": {
      "number_of_shards": "1",
      "number_of_replicas": "1"
    }
  }
}
```

删除模板

```
DELETE /_template/template_1
```

获取模板

根据模板名称或通配符获取模板

```
GET /_template/template_1
GET /_template/temp*
GET /_template/template_1,template_2
```

是否存在

```
HEAD _template/template_1
```

多个模板匹配

多个索引模板可能会匹配一个索引，在这种情况下，设置和映射都会合并到索引的最终配置中。可以使用order参数控制合并的顺序，先应用较低的顺序，然后再应用较高的顺序。例如

```
PUT /_template/template_1
{
    "index_patterns" : ["*"],
    "order" : 0,
    "settings" : {
        "number_of_shards" : 1
    },
    "mappings" : {
        "_doc" : {
            "_source" : { "enabled" : false }
        }
    }
}

PUT /_template/template_2
{
    "index_patterns" : ["te*"],
    "order" : 1,
    "settings" : {
        "number_of_shards" : 1
    },
    "mappings" : {
        "_doc" : {
            "_source" : { "enabled" : true }
        }
    }
}
```

模板版本

模板可以选择添加版本号（可以是任何整数值），以简化外部系统的模板管理。版本字段是完全可选的，仅用于模板的外部管理。要取消设置版本，只需替换模板而不指定模板。

```
PUT /_template/template_1
{
    "index_patterns" : ["*"],
    "order" : 0,
    "settings" : {
        "number_of_shards" : 1
    },
    "version": 123
}
```

要检查版本，可以使用filter_path筛选：

```
GET /_template/template_1?filter_path=*.version
```

索引别名 alias

Index Aliases

别名可以理解为指向多个索引的一个引用，类似于数据的视图，常见的使用场景

- 查询指向别名，使用别名任意切换后端的索引对比搜索效果。
- 一组按天翻滚的索引，写入别名指向当天索引，查询别名为所有索引。
- 字段发生变更，在数据重建后将别名切换到新索引。

> **TIP**
> 索引的别名让我们可以以视图的方式来操作多个索引，这个视图可是多个索引，也可是一个索引或索引的一部分。

新建别名

```
POST /_aliases
{
    "actions" : [
        { "add" : { "index" : "test1", "alias" : "alias1" } }
    ]
}
```

删除别名

```
POST /_aliases
{
    "actions" : [
        { "remove" : { "index" : "test1", "alias" : "alias1" } }
    ]
}
```

重命名别名

```
POST /_aliases
{
    "actions" : [
        { "remove" : { "index" : "test1", "alias" : "alias1" } },
        { "add" : { "index" : "test2", "alias" : "alias1" } }
    ]
}
```

多个添加别名操作

```
POST /_aliases
{
    "actions" : [
        { "add" : { "index" : "test1", "alias" : "alias1" } },
        { "add" : { "index" : "test2", "alias" : "alias1" } }
    ]
}
```

为同一个索引添加多个别名

```
# 数组方式
POST /_aliases
{
    "actions" : [
        { "add" : { "indices" : ["test1", "test2"], "alias" : "alias1" } }
    ]
}

# 通配符方式
POST /_aliases
{
    "actions" : [
        { "add" : { "index" : "test*", "alias" : "all_test_indices" } }
    ]
}
```

Filtered Aliases

带有filters的别名提供了创建同一索引的不同“视图”。可以使用查询DSL定义filter，并将其应用于具有此别名的所有Search、Count、Delete By Query和其他类似操作。

要创建Filtered Aliases，首先需要确保mapping中已经存在字段

```
PUT /test1
{
  "mappings": {
    "_doc": {
      "properties": {
        "user" : {
          "type": "keyword"
        }
      }
    }
  }
}
```

现在，我们可以创建一个对字段`user`使用过滤器的别名：

```
POST /_aliases
{
    "actions" : [
        {
            "add" : {
                "index" : "test1",
                "alias" : "alias2",
                "filter" : { "term" : { "user" : "kimchy" } }
            }
        }
    ]
}
```

Write Index

可以将别名指向的索引关联为写索引。指定后，针对指向多个索引的别名的所有索引和更新请求都将尝试解析为一个索引，即写索引。每次只能为每个别名分配一个索引作为写索引。如果未指定写入索引，并且别名引用了多个索引，则不允许写入。

可以使用别名API和索引创建API将与别名关联的索引指定为写索引。

```
POST /_aliases
{
    "actions" : [
        {
            "add" : {
                "index" : "test",
                "alias" : "alias1",
                "is_write_index" : true
            }
        },
        {
            "add" : {
                "index" : "test2",
                "alias" : "alias1"
            }
        }
    ]
}
```

在本例中，我们将别名alias1与test和test2相关联，其中test为写入的索引。

```
PUT /alias1/_doc/1
{
    "foo": "bar"
}
```

索引为/alias1/doc/1的新文档将被索引为/test/doc/1。

```
GET /test/_doc/1
```

要交换别名的写索引，可以利用别名API进行原子交换。

```
POST /_aliases
{
    "actions" : [
        {
            "add" : {
                "index" : "test",
                "alias" : "alias1",
                "is_write_index" : false
            }
        }, {
            "add" : {
```

```
                "index" : "test2",
                "alias" : "alias1",
                "is_write_index" : true
            }
        }
    ]
}
```

添加单个别名

`PUT /{index}/_alias/{name}`

参数	说明
index	别名引用的索引. 可以是 * 、 _all 、name1, name2, ...
name	别名名称
routing	（可选）与别名关联的routing
filter	（可选）与别名关联的filter

例子

单索引别名

`PUT /logs_201305/_alias/2013`

filter

```
PUT /users
{
    "mappings" : {
        "_doc" : {
            "properties" : {
                "user_id" : {"type" : "integer"}
            }
        }
    }
}
```

```
PUT /users/_alias/user_12
{
    "routing" : "12",
    "filter" : {
        "term" : {
            "user_id" : 12
        }
    }
}
```

在创建索引期间指定别名

```
PUT /logs_20162801
{
    "mappings" : {
        "_doc" : {
            "properties" : {
                "year" : {"type" : "integer"}
            }
        }
    },
    "aliases" : {
        "current_day" : {},
        "2016" : {
            "filter" : {
                "term" : {"year" : 2016 }
            }
        }
    }
}
```

删除索引别名

```
DELETE /logs_20162801/_alias/current_day
```

索引复制 reindex

Reindex

复制一个索引到另一个索引

> **重要**
>
> 1. 源索引必须启用 `_source`。
> 2. Reindex 不会复制源索引的设置。在运行_reindex操作之前，应该先创建目标索引，设置与源索引对等的settings、mappings。

```
POST _reindex
{
  "source": {
    "index": "twitter"
  },
  "dest": {
    "index": "new_twitter"
  }
}
```

version_type 参数（可选）

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来控制：

```
POST _reindex
{
  "source": {
    "index": "twitter"
  },
  "dest": {
    "index": "new_twitter",
    "version_type": "internal"
  }
}
```

- **internal**: 强制更新，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。
- **external**: 保留最新版本，只有当源索引的数据 version 比目标索引的数据 version 高的时候，才会去更新。

query 参数（可选）

通过query指定部分数据复制。

```
POST _reindex
{
  "source": {
    "index": "twitter",
    "type": "_doc",
    "query": {
      "term": {
        "user": "kimchy"
      }
    }
  },
  "dest": {
    "index": "new_twitter"
  }
}
```

多对一复制

从多个源索引复制到目标索引

```
POST _reindex
{
  "source": {
    "index": ["twitter", "blog"],
    "type": ["_doc", "post"]
  },
  "dest": {
    "index": "all_together",
    "type": "_doc"
  }
}
```

size

通过设置size来限制复制文档的数量

```
POST _reindex
{
  "size": 1,
  "source": {
    "index": "twitter"
  },
  "dest": {
    "index": "new_twitter"
  }
}
```

使用排序

排序会降低复制的效率，但在某些情况下，这是值得的。例如复制时间较近的文档。

```
POST _reindex
{
  "size": 10000,
  "source": {
    "index": "twitter",
    "sort": { "date": "desc" }
  },
  "dest": {
    "index": "new_twitter"
  }
}
```

简要查看 cat

cat count

查看文档总数

```
GET /_cat/count
```

cat indices

查看索引

```
GET /_cat/indices

# 前缀查询
GET /_cat/indices/twi*?v&s=index

# 存在未分配shard的索引
GET /_cat/indices?v&health=yellow
```

> **TIP**
> ?v 带上列名

cat shards

查看分片

```
GET _cat/shards

GET _cat/shards?h=index,shard,state,prirep,store.bytes,docs,segments.count,tier

GET _cat/shards/twitt*
```

TIP

?h 指定查看的列

cat segments

查看分片的段

```
GET /_cat/segments?v
```

cat aliases

查看别名

```
GET /_cat/aliases
```

cat templates

查看模板

```
GET /_cat/templates
```

字段匹配 match

最简单的查询，它匹配所有文档。

```
GET /_search
{
    "query": {
        "match_all": {}
    }
}
```

Match Query 最基础的一个例子，"this is a test" 经过分析过程，去除停用词 ["is"、 "a"], 提取关键词 ["this"、 "test"]以默认的 or 关系针对 message 字段进行匹配查询。

```
GET /_search
{
    "query": {
        "match" : {
            "message" : "this is a test"
        }
    }
}
```

INFO

1. MatchQuery 接受 text/numerics/date 类型的字段
2. 可以将 lenient 参数设置为 true 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。

operator

以 and 来控制逻辑关系（默认为 or）做更精准的匹配。

```
GET /_search
{
    "query": {
        "match" : {
```

```
            "message" : {
                "query" : "this is a test",
                "operator" : "and"
            }
        }
    }
}
```

minimum_should_match

minimum_should_match 最小匹配数。 配置参考

下例中关键词需匹配2次以上

```
GET /_search
{
    "query": {
        "match" : {
            "message" : {
                "query" : "hello world",
                "operator" : "or",
                "minimum_should_match": 2
            }
        }
    }
}
```

Fuzziness

fuzziness 允许不精确的模糊匹配。 配置参考

默认情况下允许模糊转置 (ab → ba)，但可以通过将 fuzzy_transpositions 设置为 false 来禁用。

```
GET /_search
{
    "query": {
        "match" : {
            "message" : {
                "query" : "this is a testt",
                "fuzziness": "AUTO"
            }
        }
```

```
    }
}
```

> **INFO**
>
> 注意，模糊匹配不适用于具有同义词的关键词

Zero terms query

如果使用的分析器（例如停用词过滤器）删除查询中的所有输入词，则默认行为是匹配不到任何文档。为了改变可以使用 zero_terms_query 选项，它接受 none（默认）并且 all 对应于 match_all 查询

```
GET /_search
{
    "query": {
        "match" : {
            "message" : {
                "query" : "to be or not to be",
                "operator" : "and",
                "zero_terms_query": "all"
            }
        }
    }
}
```

cutoff_frequency

cutoff_frequency 将查询字符串里的词项分为低频和高频两组。低频组（更重要的词项）组成大量查询条件，而高频组（次重要的词项）只会用来评分，而不参与匹配过程。通过对这两组词的区分处理，我们可以获取更高的检索性能提升。

以下面查询为例：

```
GET /_search
{
    "query": {
        "match" : {
            "message" : {
                "query" : "Shakespeare says, to be or not to be",
                "cutoff_frequency" : 0.01
```

```
            }
        }
    }
}
```

任何词项出现在文档中超过1%，被认为是高频词。上例中出现频率较多的停用词 to be or not to be 被划分为高频词，出现频率低的Shakespeare says为低频词 此查询会被重写为以下的 bool 查询:

```
{
  "bool": {
    "must": {
      "bool": {
        "should": [
          { "term": { "text": "Shakespeare" }},
          { "term": { "text": "says"  }}
        ]
      }
    },
    "should": {
      "bool": {
        "should": [
          { "term": { "text": "to" }},
          { "term": { "text": "be" }},
          { "term": { "text": "or" }},
          { "term": { "text": "not" }}
        ]
      }
    }
  }
}
```

搜索更重要的低频词，而高频词只用于提高算分，以获取更好的结果相关度。

多字段匹配 multi_match

MultiMatchQuery 是建立在 MatchQuery 之上以允许多字段查询：

```
GET /_search
{
  "query": {
    "multi_match" : {
      "query":    "this is a test",
      "fields": [ "subject", "message", "*_name" ]
    }
  }
}
```

> *_name 为前缀匹配的字段

提升字段权重

可以使用特殊字符 (^) 表示法提升字段权重：

```
GET /_search
{
  "query": {
    "multi_match" : {
      "query" : "this is a test",
      "fields" : [ "subject^3", "message" ]
    }
  }
}
```

> subject字段的重要性是message字段的三倍。

算分类型

multi_match查询内部执行的方式取决于type参数，可以设置为

- **best_fields** 查找与任何字段匹配的文档，算分来自最佳字段的分数。 **默认**

- **most_fields** 查找与任何字段匹配的文档，算分来自每个字段的分数总和。
- **cross_fields** 使用相同的分析器处理字段，就好像它们是一个大字段一样。查找任何字段中的每个单词。
- **phrase** 对每个字段运行 match_phrase 查询并使用最佳字段的分数。
- **phrase_prefix** 在每个字段上运行 match_phrase_prefix 查询并合并每个字段的分数。

1. best_fields

当您搜索在 **同一字段** 中最好找到多个词时，使用best_fields。例如，单个字段中的“brown fox”比一个字段中的“brown”和另一个字段中的“fox”更有意义。

best_fields 为每个字段生成匹配查询并将它们包装在 dis_max 查询中，以找到单个最佳匹配字段。例如这个查询：

```
GET /_search
{
  "query": {
    "multi_match" : {
      "query":      "brown fox",
      "type":       "best_fields",
      "fields":     [ "subject", "message" ],
      "tie_breaker": 0.3
    }
  }
}
```

将被执行为：

```
GET /_search
{
  "query": {
    "dis_max": {
      "queries": [
        { "match": { "subject": "brown fox" }},
        { "match": { "message": "brown fox" }}
      ],
      "tie_breaker": 0.3
    }
  }
}
```

tie_breaker 这个参数将其他匹配语句的评分也考虑其中：

```
总分 = (best_fields _score) + (tie_breaker * 其他字段 _score)
```

2. most_fields

most_fields 匹配多个字段返回的综合评分，和best_fields的区别是总分不再是最佳字段的匹配分，而是所有字段匹配分的总和。

```
GET /_search
{
  "query": {
    "multi_match" : {
      "query":      "quick brown fox",
      "type":       "most_fields",
      "fields":     [ "title", "title.original", "title.shingles" ]
    }
  }
}
```

等价于

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title":          "quick brown fox" }},
        { "match": { "title.original": "quick brown fox" }},
        { "match": { "title.shingles": "quick brown fox" }}
      ]
    }
  }
}
```

每个匹配子句的分数相加，然后除以匹配子句的数量。

3. cross_fields

跨字段匹配，与 best_fields 和 most_fields 使用面向字段的算分方式不同，cross_fields 是以面向词为主体，将所有字段当成一个大字段进行关键词的匹配度算分。

例如我们期望查找 **first_name:Will** 以及 **last_name: Smith** 匹配度最高的文档。

```
PUT person
{
    "mappings" : {
        "_doc" : {
            "properties" : {
                "first_name" : { "type" : "text" },
                "last_name" : { "type" : "text" }
            }
        }
    }
}


POST person/_bulk
{ "index" : { "_type" : "_doc"} }
{ "first_name" : "Will","last_name":"Smith" }
{ "index" : { "_type" : "_doc"} }
{ "first_name" : "tony","last_name":"Will Minth" }
{ "index" : { "_type" : "_doc"} }
{ "first_name" : "Will","last_name":"Smith world" }
{ "index" : { "_type" : "_doc"} }
{ "first_name" : "Will Minth","last_name":"tony" }

GET person/_search
{
  "query": {
    "multi_match" : {
      "query":      "Will Smith",
      "type":       "most_fields",
      "fields":     [ "first_name", "last_name" ]
    }
  }
}

# "Will Smith world" 相比 "tony Will Minth" 具备更高的评分
GET person/_search
{
  "query": {
    "multi_match" : {
      "query":      "Will Smith",
      "type":       "cross_fields",
      "fields":     [ "first_name", "last_name" ]
    }
  }
}
```

4. phrase 和 phrase_prefix

phrase 和 phrase_prefix 类型的行为类似 best_fields，但它们使用 match_phrase 或 match_phrase_prefix 查询而不是 match 查询。

```
GET /_search
{
  "query": {
    "multi_match" : {
      "query":      "quick brown f",
      "type":       "phrase_prefix",
      "fields":     [ "subject", "message" ]
    }
  }
}
```

等价于

```
GET /_search
{
  "query": {
    "dis_max": {
      "queries": [
        { "match_phrase_prefix": { "subject": "quick brown f" }},
        { "match_phrase_prefix": { "message": "quick brown f" }}
      ]
    }
  }
}
```

TIP

- 此外，接受匹配查询中的analyze、boost、lenient、zero_terms_query，以及匹配短语查询中解释的 slop。类型 phrase_prefix 还可接受 max_expansions。
- fuzziness 参数不能与 phrase 或 phrase_prefix 类型一起使用

短语查询 match_phrase

短语查询 match_phrase 会将检索关键词分词，并保证分词结果必须在被检索字段的分词中都包含，且顺序必须相同。

```
GET /_search
{
    "query": {
        "match_phrase" : {
            "message" : "this is a test"
        }
    }
}
```

核心参数slop： 位置距离容差值

```
{
    "query": {
        "match_phrase" : {
            "message" : "this is a test",
            "slop":1
        }
    }
}
```

更高的距离度可以容忍更多的匹配结果

指定分析器

可以设置分析器来控制哪个分析器将对文本执行分析过程 例如

```
GET /_search
{
    "query": {
        "match_phrase" : {
            "message" : {
                "query" : "this is a test",
                "analyzer" : "my_analyzer"
```

```
                }
            }
        }
    }
```

短语前缀查询 match_phrase_prefix

match_phrase_prefix 与 match_phrase 相同，只是它允许对文本中的最后一个词进行前缀匹配。例如：

```
GET /_search
{
    "query": {
        "match_phrase_prefix" : {
            "message" : "quick brown f"
        }
    }
}
```

它接受与match_phrase相同的参数。此外，它还接受一个 max_expansions 参数（默认为 50），该参数可以控制最后一项将扩展到多少后缀。强烈建议将其设置为可接受的值以控制查询的执行时间。例如

```
GET /_search
{
    "query": {
        "match_phrase_prefix" : {
            "message" : {
                "query" : "quick brown f",
                "max_expansions" : 10
            }
        }
    }
}
```

TIP

match_phrase_prefix 非常适合自动提示场景 AutoComplete,可让您快速检索“输入时搜索 ...”。

关键字匹配 term

Term Query 关键词查询

查找包含倒排索引中指定关键词的文档。例如：

```
GET /_search
{
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

可以指定 boost 参数来为该关键词提供比另一个查询更高的相关性分数，例如：

```
GET _search
{
  "query": {
    "bool": {
      "should": [
        {
          "term": {
            "status": {
              "value": "urgent",
              "boost": 2.0
            }
          }
        },
        {
          "term": {
            "status": "normal"
          }
        }
      ]
    }
  }
}
```

TIP

> urgent查询子句的提升为 2.0，这意味着它的重要性是普通查询子句的两倍。

Terms Query 包含多个词

```
GET /_search
{
    "query": {
        "terms" : { "user" : ["kimchy", "elasticsearch"]}
    }
}
```

Terms Set Query 集合类查询

返回至少一个或多个提供的关键词匹配的任何文档。由于terms未被分词 必须完全匹配。因此由最小应匹配字段控制或按最小应匹配脚本计算每个文档。

```
PUT /my-index
{
    "mappings": {
        "_doc": {
            "properties": {
                "required_matches": {
                    "type": "long"
                }
            }
        }
    }
}

PUT /my-index/_doc/1?refresh
{
    "codes": ["ghi", "jkl"],
    "required_matches": 2
}

PUT /my-index/_doc/2?refresh
{
    "codes": ["def", "ghi"],
    "required_matches": 2
}
```

```
GET /my-index/_search
{
    "query": {
        "terms_set": {
            "codes" : {
                "terms" : ["abc", "def", "ghi"],
                "minimum_should_match_field": "required_matches"
            }
        }
    }
}
```

范围匹配 range

查询指定范围内的文档。以下示例返回年龄在 10 到 20 之间的所有文档：

```
GET _search
{
    "query": {
        "range" : {
            "age" : {
                "gte" : 10,
                "lte" : 20
            }
        }
    }
}
```

范围查询接受以下参数：

操作	描述
gte	大于等于
gt	大于
lte	小于等于
lt	小于

日期字段的范围查询

在日期类型的字段上运行范围查询时，可以使用 Date Math 指定范围：

```
GET _search
{
    "query": {
        "range" : {
            "date" : {
```

```
                "gte" : "now-1d/d",
                "lt" :  "now/d"
            }
        }
    }
}
```

日期字段范围查询 - Date Format

默认情况下，格式化日期将使用日期字段上指定的格式进行解析，但可以通过将格式参数传递给范围查询来覆盖它：

```
GET _search
{
    "query": {
        "range" : {
            "born" : {
                "gte": "01/01/2012",
                "lte": "2013",
                "format": "dd/MM/yyyy||yyyy"
            }
        }
    }
}
```

日期字段范围查询 - 指定时区

```
GET _search
{
    "query": {
        "range" : {
            "timestamp" : {
                "gte": "2015-01-01T00:00:00",
                "lte": "now",
                "time_zone": "+01:00"
            }
        }
    }
}
```

是否存在 exists

返回在提供的字段中包含 null 或 [] 以外的值的文档。

```
GET /_search
{
    "query": {
        "exists": {
            "field": "user"
        }
    }
}
```

查找具有空值的文档

要在提供的字段中查找仅包含空值或 [] 的文档，请将 must_not 布尔查询与 exists 查询一起使用

以下搜索返回用户字段中仅包含空值或 [] 的文档。

```
GET /_search
{
    "query": {
        "bool": {
            "must_not": {
                "exists": {
                    "field": "user"
                }
            }
        }
    }
}
```

前缀匹配 prefix

匹配用户字段包含以 ki 开头的文档：

```
GET /_search
{ "query": {
    "prefix" : { "user" : "ki" }
  }
}
```

也可以提升算分：

```
GET /_search
{ "query": {
    "prefix" : { "user" :  { "value" : "ki", "boost" : 2.0 } }
  }
}
```

通配符匹配 wildcard

通配符是匹配一个或多个字符的占位符。例如，* 通配符匹配零个或多个字符。您可以将通配符运算符与其他字符组合以创建通配符模式。

```
GET /_search
{
    "query": {
        "wildcard": {
            "user": {
                "value": "ki*y",
                "boost": 1.0,
                "rewrite": "constant_score"
            }
        }
    }
}
```

参数说明

?, 匹配任何单个字符

*，可以匹配零个或多个字符，包括空字符

正则匹配 regexp

Regexp Query允许您使用正则表达式查询关键词。有关支持的正则表达式语言的详细信息，请参阅正则表达式语法。

> **CAUTION**
>
> 注意：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。匹配 *.之类的所有内容以及使用正向预查正则表达式都非常慢。如果可能，您应该尝试在正则表达式开始之前使用长前缀。像 .?+* 这样的通配符匹配器大多会降低性能。

```
GET /_search
{
    "query": {
        "regexp":{
            "name.first": "s.*y"
        }
    }
}
```

支持算分

```
GET /_search
{
    "query": {
        "regexp":{
            "name.first":{
                "value":"s.*y",
                "boost":1.2
            }
        }
    }
}
```

您还可以使用特殊标志

```
GET /_search
{
```

```
    "query": {
        "regexp":{
            "name.first": {
                "value": "s.*y",
                "flags" : "INTERSECTION|COMPLEMENT|EMPTY"
            }
        }
    }
}
```

支持的的Flag有 ALL（默认）、ANYSTRING、COMPLEMENT、EMPTY、INTERSECTION、INTERVAL 或 NONE。请查看 Lucene 文档了解它们的含义

正则表达式很危险，因为很容易意外地创建一个看起来无害的表达式，它需要指数数量的内部确定自动机状态（以及相应的 RAM 和 CPU）来执行 Lucene。 Lucene 使用 max_determinized_states 设置（默认为 10000）来防止这些。您可以提高此限制以允许执行更复杂的正则表达式。

```
GET /_search
{
    "query": {
        "regexp":{
            "name.first": {
                "value": "s.*y",
                "flags" : "INTERSECTION|COMPLEMENT|EMPTY",
                "max_determinized_states": 20000
            }
        }
    }
}
```

模糊匹配 fuzzy

模糊查询使用基于 Levenshtein 编辑距离的相似性。

模糊查询生成在模糊度中指定的最大编辑距离内的匹配词，然后检查词词典以找出那些生成的词中哪些实际存在于索引中。最终查询最多使用 max_expansions 个匹配项。

这是一个简单的例子

```
GET /_search
{
    "query": {
       "fuzzy" : { "user" : "ki" }
    }
}
```

或者使用更高级的设置：

```
GET /_search
{
    "query": {
        "fuzzy" : {
            "user" : {
                "value": "ki",
                "boost": 1.0,
                "fuzziness": 2,
                "prefix_length": 0,
                "max_expansions": 100
            }
        }
    }
}
```

参数

fuzziness

最大编辑距离。默认为自动。请参见模糊性。

prefix_length

不会被“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。

max_expansions

模糊查询将扩展到的最大术语数。默认为 50。

transpositions

是否支持模糊转置 (ab → ba)。默认为false。

> **TIP**
> 如果 prefix_length 设置为 0 且 max_expansions 设置为较大的数字，则此查询可能会非常繁重。它可能导致检查索引中的每个术语!

主键匹配 ids

根据其 ID 返回文档。此查询使用存储在 _id 字段中的文档 ID。

```
GET /_search
{
    "query": {
        "ids" : {
            "type" : "_doc",
            "values" : ["1", "4", "100"]
        }
    }
}
```

跨度匹配 span

Span Queries

跨度查询是 low-level 的位置查询，它提供对指定关键词的顺序和接近度的专门控制。

Span Term Queries

span_term 提供基础span query ， 通常和其他span查询组合使用。

```
GET /_search
{
    "query": {
        "span_term" : { "user" : "kimchy" }
    }
}
```

Span Multi Term Queries

span_multi 查询允许您将多术语查询（通配符、模糊、前缀、范围或正则表达式查询之一）包装为跨度查询，因此它可以嵌套。例子：

```
GET /_search
{
    "query": {
        "span_multi":{
            "match":{
                "prefix" : { "user" :  { "value" : "ki" } }
            }
        }
    }
}
```

Span First Query

匹配字段开头附近的跨度。这是一个例子：

```
GET /_search
{
    "query": {
        "span_first" : {
            "match" : {
                "span_term" : { "user" : "kimchy" }
            },
            "end" : 3
        }
    }
}
```

match 子句可以是任何其他 span query。 end 控制match中允许的最大结束位置。

Span Near Query

匹配彼此靠近的跨度。可以指定 slop，中间不匹配位置的最大数量，以及匹配是否需要按顺序进行。 这是一个例子：

```
{
    "query": {
        "span_near" : {
            "clauses" : [
                { "span_term" : { "field" : "value1" } },
                { "span_term" : { "field" : "value2" } },
                { "span_term" : { "field" : "value3" } }
            ],
            "slop" : 12,
            "in_order" : false
        }
    }
}
```

clauses 是一个或多个其他 span 类型查询的列表， slop 控制允许的中间不匹配位置的最大数量。

Span Or Query

匹配其跨度子句的并集 这是一个例子：

```
GET /_search
{
    "query": {
```

```
        "span_or" : {
            "clauses" : [
                { "span_term" : { "field" : "value1" } },
                { "span_term" : { "field" : "value2" } },
                { "span_term" : { "field" : "value3" } }
            ]
        }
    }
}
```

Span Not Query

删除与另一个SpanQuery重叠的匹配项，或在另一个SpanQuery之前（由参数pre控制）x标记内或之后（由参数post控制）y标记内的匹配项。span not查询映射到Lucene SpanNotQuery。下面是一个示例：

```
GET /_search
{
    "query": {
        "span_not" : {
            "include" : {
                "span_term" : { "field1" : "hoya" }
            },
            "exclude" : {
                "span_near" : {
                    "clauses" : [
                        { "span_term" : { "field1" : "la" } },
                        { "span_term" : { "field1" : "hoya" } }
                    ],
                    "slop" : 0,
                    "in_order" : true
                }
            }
        }
    }
}
```

include 和 exclude 子句可以是任何跨度类型的查询。 include 子句是匹配被过滤的span 查询，exclude 子句是匹配不能与返回的重叠的span 查询。

在上面的示例中，除了前面有 la 的文档外，所有包含术语 hoya 的文档都被过滤掉了。

参数选项：

参数	含义
pre	如果设置包含范围之前的令牌数量，则不能与排除范围重叠。默认值为0。
post	如果设置包含范围之后的令牌数量，则不能与排除范围重叠。默认值为0。
dist	如果设置，则包含范围内的令牌数量不能与排除范围重叠。相当于设置前后。

Span Containing Query

返回包含另一个span查询的匹配项。包含查询的span映射到Lucene SpanContainingQuery。下面是一个示例

```
GET /_search
{
    "query": {
        "span_containing" : {
            "little" : {
                "span_term" : { "field1" : "foo" }
            },
            "big" : {
                "span_near" : {
                    "clauses" : [
                        { "span_term" : { "field1" : "bar" } },
                        { "span_term" : { "field1" : "baz" } }
                    ],
                    "slop" : 5,
                    "in_order" : true
                }
            }
        }
    }
}
```

big和little子句可以是任何跨度类型的查询。返回从大到小的匹配跨度。

Span Within Query

返回包含在另一个span查询中的匹配项。查询中的span映射到Lucene SpanWithinQuery。下面是一个示例：

```
GET /_search
{
    "query": {
        "span_within" : {
            "little" : {
                "span_term" : { "field1" : "foo" }
            },
            "big" : {
                "span_near" : {
                    "clauses" : [
                        { "span_term" : { "field1" : "bar" } },
                        { "span_term" : { "field1" : "baz" } }
                    ],
                    "slop" : 5,
                    "in_order" : true
                }
            }
        }
    }
}
```

big 和 little 子句 可以是任何跨度类型的查询。返回包含在大范围内的小范围的匹配跨度。

Span Field Masking Query

Wrapper允许span查询通过隐藏其搜索字段来参与复合单字段span查询。span字段掩码查询映射到Lucene的SpanFieldMaskingQuery。这可用于支持span near或span或跨不同字段的查询，这通常是不允许的。 当使用多个分析器对同一内容进行索引时，跨度字段屏蔽查询与多字段结合使用非常有用。例如，我们可以用标准分析器将文本分解为单词，然后用英语分析器将单词词根形式。

例子

```
GET /_search
{
  "query": {
    "span_near": {
      "clauses": [
        {
          "span_term": {
            "text": "quick brown"
          }
        },
        {
          "field_masking_span": {
```

```
            "query": {
              "span_term": {
                "text.stems": "fox"
              }
            },
            "field": "text"
          }
        }
      ],
      "slop": 5,
      "in_order": false
    }
  }
}
```

TIP

注意：当span字段掩码查询返回掩码字段时，将使用提供的字段名称规范进行评分。这可能会导致意外的得分行为。

布尔查询 bool

布尔组合查询

```
POST _search
{
  "query": {
    "bool" : {
      "must" : {
        "term" : { "user" : "kimchy" }
      },
      "filter": {
        "term" : { "tag" : "tech" }
      },
      "must_not" : {
        "range" : {
          "age" : { "gte" : 10, "lte" : 20 }
        }
      },
      "should" : [
        { "term" : { "tag" : "wow" } },
        { "term" : { "tag" : "elasticsearch" } }
      ],
      "minimum_should_match" : 1,
      "boost" : 1.0
    }
  }
}
```

参数	说明
must	必须满足子句查询条件，子句参与算分并影响结果的排名。
filter	必须满足子句查询条件。与 must 不同的是，查询子句不参与算分，且有几率在缓存中执行以获取更快的执行速度。
should	应该满足的子句查询条件，仅用于影响算分。

参数	说明
must_not	查询子句不得出现在匹配文档中。子句在过滤器上下文中执行，这意味着评分被忽略并且子句有几率在缓存中执行。因为评分被忽略，所有文档的分数都为 0。

> **TIP**
>
> **Bool query in filter context**
>
> 如果查询在filter上下文中使用并且它具有 should 子句，则至少需要一个 should 子句才能匹配

bool 查询采用匹配越多越好的方法，因此每个匹配的 must 或 should 子句的分数将被加在一起以提供每个文档的最终 _score。

```
POST _search
{
  "query": {
    "bool" : {
      "must" : {
        "term" : { "user" : "kimchy" }
      },
      "filter": {
        "term" : { "tag" : "tech" }
      },
      "must_not" : {
        "range" : {
          "age" : { "gte" : 10, "lte" : 20 }
        }
      },
      "should" : [
        { "term" : { "tag" : "wow" } },
        { "term" : { "tag" : "elasticsearch" } }
      ],
      "minimum_should_match" : 1,
      "boost" : 1.0
    }
  }
}
```

使用 minimum_should_match

您可以使用 minimum_should_match 参数指定返回文档必须匹配的 should 子句的数量或百分比。

如果 bool 查询包含至少一个 should 子句并且没有 must 或 filter 子句，则默认值为 1。否则，默认值为 0。

常量算分 constant_score

Constant Score Query

包装过滤器查询并返回每个匹配的文档，其相关性得分等于 boost 参数值。

```
GET /_search
{
    "query": {
        "constant_score" : {
            "filter" : {
                "term" : { "user" : "kimchy"}
            },
            "boost" : 1.2
        }
    }
}
```

filter

- 过滤器查询，任何返回的文档都必须匹配此查询。
- 过滤器查询不计算相关性分数。为了提高性能，Elasticsearch 会自动缓存常用的过滤器查询

boost

用于匹配过滤器查询的每个文档的常量相关性分数，默认为 1.0。

最佳匹配 dis_max

算分公式

如果文档匹配多个子句，则 dis_max 查询计算文档的相关性分数如下：

`总分` = `最佳匹配子句分数` + `sum(其他匹配子句分数) * tie_breaker`

```
GET /_search
{
    "query": {
        "dis_max" : {
            "tie_breaker" : 0.7,
            "queries" : [
                { "term": { "title": "Quick pets" } },
                { "term": { "body": "Quick pets" } }
            ]
        }
    }
}
```

> 1. 从得分最高的匹配子句中获取相关性得分。
> 2. 将任何其他匹配子句的分数乘以 tie_breaker 值。
> 3. 将最高分加到相乘的分数上获得最后得分。

如果 tie_breaker 值大于 0.0，则所有匹配的子句都有效，但得分最高的子句最重要

参数	是否必选	类型	说明
queries	必选项	查询子句数组	包含一个或多个查询子句。如果一个文档匹配多个查询子句，dismax 会取最高的子句相关性分数。

参数	是否必选	类型	说明
tie_breaker	可选项	float （取值范围：0 到 1.0 之间的浮点数）	增益系数 - 用于增加最高分子句外的其他匹配子句的文档相关性分数。默认为 0.0。

子句加权 boosting

根据negative匹配子句影响positive匹配子句的分数。

数据准备

```
PUT /blogs
{
    "mappings" : {
        "_doc" : {
            "properties" : {
                "title" : { "type" : "text" },
                "body" : { "type" : "text" }
            }
        }
    }
}

PUT /blogs/_doc/1
{
  "title": "Quick brown rabbits",
  "body": "Brown rabbits are commonly seen."
}
```

1. 基础算分

```
GET blogs/_search
{
  "query": {
    "match": {
      "title": "rabbits"
    }
  }
}
```

基础的BM25分数为 **0.2876821**

2. 影响算分

```
GET blogs/_search
{
  "query": {
    "boosting": {
      "positive": {
        "match": {
          "title": "rabbits"
        }
      },
      "negative": {
        "match": {
          "body": "rabbits"
        }
      },
      "negative_boost": 1.2
    }
  }
}
```

分数为 0.2876821 * 1.2 = 0.3452185

解释

当negative满足匹配时，得分=negative_boost * positive的query分数，否则不影响。

用途：例如我们不需要使用 bool 查询中的“NOT”子句排除文档，而仅仅降低它们的分数用于影响排名。

参数	说明
positive	（必填项）用于执行匹配的查询子句
negative	（必填项）用于加权条件判断的查询子句
negative_boost	（必填项）加权系数，当negative查询匹配时，用于降低或提升positive匹配的文档相关性得分。

函数算分 function_score

单函数样例

```
GET /_search
{
    "query": {
        "function_score": {
            "query": { "match_all": {} },
            "boost": "5",
            "random_score": {},
            "boost_mode":"multiply"
        }
    }
}
```

> random_score 生成从 0 到但不包括 1 的均匀分布的浮点数，通常由于结果集打散。

组合多个函数

```
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
        }
    }
}
```

> boost : 对整体分数进行系数加权
> weight : 过滤子句评分权重

首先，每个文档都根据定义的函数进行评分。参数 score_mode 指定如何组合这些分数：

score_mode	说明
multiply	分数相乘（默认）
sum	分数求和
avg	分数取平均值
first	分数取匹配过滤器的第一个函数
max	取最高分
min	取最低分

内置函数

function_score 提供以下几种评分函数

- script_score
- weight
- random_score
- field_value_factor
- decay functions: gauss, linear, exp

script_score

script_score 函数允许您包装另一个查询并自定义它的评分，使用脚本表达式从文档中的其他数字字段值派生的计算（可选的）。这是一个简单的示例：

```
GET /_search
{
    "query": {
        "function_score": {
            "query": {
                "match": { "message": "elasticsearch" }
            },
            "script_score" : {
                "script" : {
                  "source": "Math.log(2 + doc['likes'].value)"
                }
            }
        }
    }
}
```

> **TIP**
>
> 在 Elasticsearch 中，所有文档得分都是正的 32 位浮点数。
> 如果 script_score 函数产生的分数精度更高，则会将其转换为最接近的 32 位浮点数。
> 同样，分数必须是非负数。否则，Elasticsearch 会返回一个错误。

脚本编译在缓存中进行以加快执行速度。如有必要可以考虑重用同一个脚本，并为其提供参数

```
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['likes'].value)"
                }
            }
        }
```

```
    }
}
```

weight

权重分数允许您将分数乘以提供的权重。

`"weight" : number`

random_score

如果您希望分数可重复，可以提供`seed`和`field`。然后，将根据该seed、所考虑文档的最小字段值和基于索引名称和shardid计算的salt来计算最终得分，以便具有相同值但存储在不同索引中的文档获得不同的得分。请注意，位于同一shard内且字段值相同的文档将获得相同的分数，因此通常需要使用对所有文档具有唯一值的字段。一个好的默认选择可能是使用`_seq_no`字段，其唯一的缺点是，如果文档被更新，分数将发生变化，因为更新操作也会更新_seq_n字段的值。

```
GET /_search
{
    "query": {
        "function_score": {
            "random_score": {
                "seed": 10,
                "field": "_seq_no"
            }
        }
    }
}
```

field_value_factor

field_value_factor 函数允许您使用文档中的字段来影响分数。它类似于使用 script_score 函数，但是，它避免了编写脚本的开销。如果用于多值字段，则只有该字段的第一个值用于计算。

假设你有一个数字类型字段 likes，并希望用这个字段影响文档的分数，这样做的例子如下

```
GET /_search
{
    "query": {
        "function_score": {
            "field_value_factor": {
```

```
                "field": "likes",
                "factor": 1.2,
                "modifier": "sqrt",
                "missing": 1
            }
        }
    }
}
```

这将转化为以下评分公式：`sqrt(1.2 * doc['likes'].value)`

field_value_factor 的函数选项：

参数	说明
field	要从文档中提取的字段。
factor	与字段值相乘的可选因子，默认为 1。
modifier	应用于字段值的修饰符可以是以下之一：none、log、log1p、log2p、ln、ln1p、ln2p、square、sqrt 或 reciprocal。默认为none。

Modifier 参数

Modifier	含义
none	不要对字段值应用任何乘数
log	取字段值的常用对数
log1p	字段值加1，取常用对数
log2p	字段值加2，取常用对数
ln	取字段值的自然对数
ln1p	字段值加1取自然对数

Modifier	含义
ln2p	字段值加2取自然对数
square	将字段值平方（乘以它本身）
sqrt	取字段值的平方根
reciprocal	倒数字段值，类似于 1/x ，其中 x 是字段值

> **TIP**
> field_value_score 函数产生的分数必须是非负数，否则将发出弃用警告。

> 注意，取 0 的 log() 或负数的平方根是非法操作，会抛出异常。一定要限制使用范围过滤器的字段值以避免这种情况，或使用 `log1p` 和 `ln1p`

decay

衰减函数（Decay Function）对于一个字段，它有一个理想的值，而字段实际的值越偏离这个理想值（无论是增大还是减小），就越不符合期望。

Decay Function 可用于数值、日期和地理位置类型。由以下属性组成：

- **原点（`origin`）**：该字段最理想的值，这个值可以得到满分（1.0）
- **偏移量（`offset`）**：与原点相差在偏移量之内的值也可以得到满分
- **衰减规模（`scale`）**：当值超出了原点到偏移量这段范围，它所得的分数就开始进行衰减了，衰减规模决定了这个分数衰减速度的快慢
- **衰减值（`decay`）**：该字段可以被接受的值（默认为 0.5），相当于一个分界点，具体的效果与衰减的模式有关

例如我们想要买一样东西：

- 它的理想价格是 50 元，这个值为原点
- 但是我们不可能非 50 元就不买，而是会划定一个可接受的价格范围，例如 45-55 元，±5 就为偏移量

- 当价格超出了可接受的范围，就会让人觉得越来越不值。如果价格是 70 元，评价可能是不太想买，而如果价格是 200 元，评价则会是不可能会买，这就是由衰减规模和衰减值所组成的一条衰减曲线

或者如果我们想租一套房：

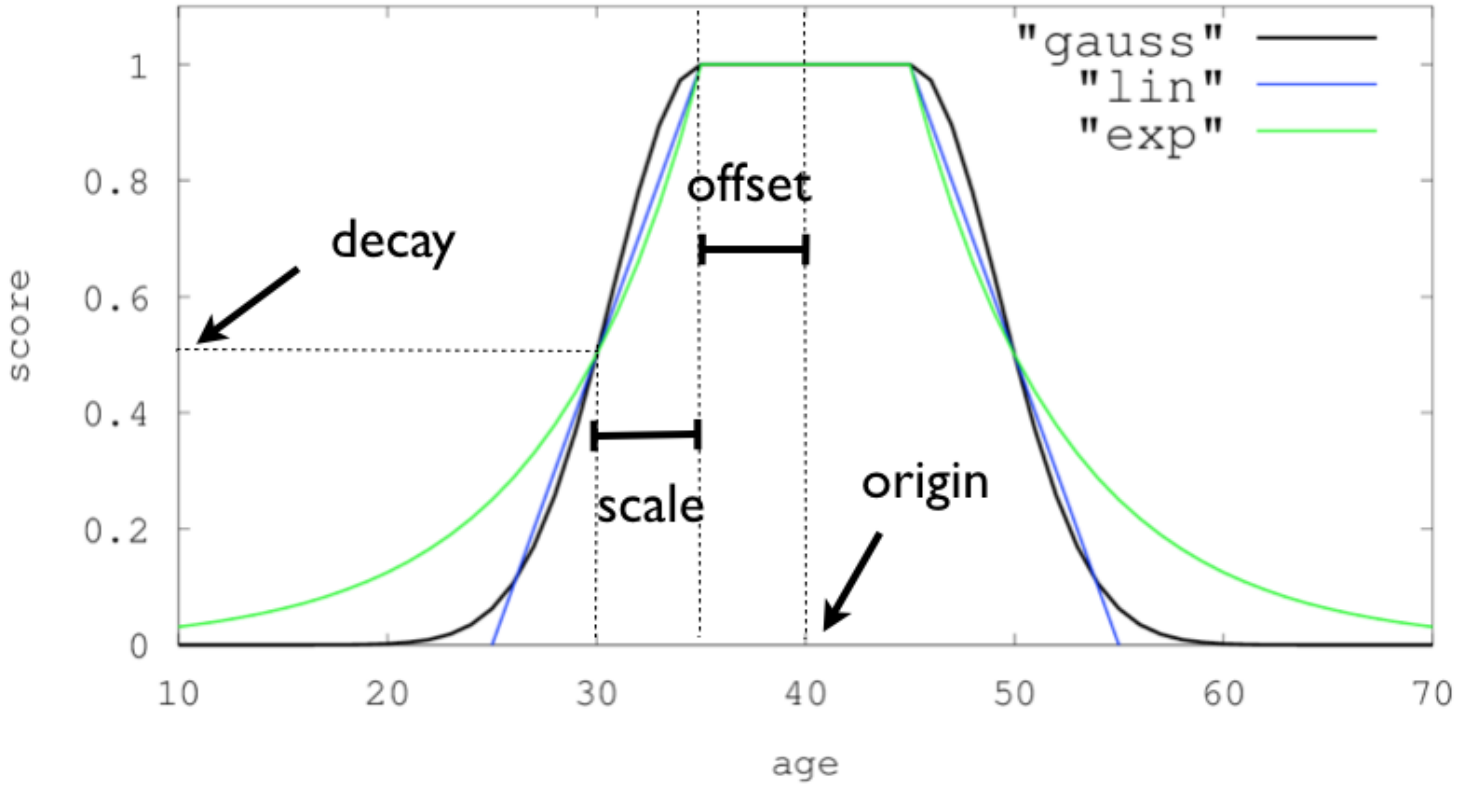
- 它的理想位置是公司附近，公司位置就是原点
- 如果离公司在 5km 以内，是我们可以接受的范围，在这个范围内我们不去考虑距离，而是更偏向于其他信息
- 当距离超过 5km 时，我们对这套房的评价就越来越低了，直到超出了某个范围就再也不会考虑了

使用样例 字段值介于 2013-09-12 和 2013-09-22 之间的文档的权重为 1.0，而距该日期 15 天的文档的权重为 0.5。

```
GET /_search
{
    "query": {
        "function_score": {
            "gauss": {
                "date": {
                      "origin": "2013-09-17",
                      "scale": "10d",
                      "offset": "5d",
                      "decay" : 0.5
                }
            }
        }
    }
}
```

DECAY_FUNCTION

衰减函数可以指定三种不同的模式：线性函数（linear）、以 e 为底的指数函数（Exp）和高斯函数（gauss），它们拥有不同的衰减曲线：



1. gauss

高斯函数，曲线特点：他的衰减速率是先缓慢，然后变快，最后又放缓。 计算公式：

$$S(doc) = \exp\left(-\frac{\max(0, |fieldvalue_{doc} - origin| - offset)^2}{2\sigma^2}\right)$$

计算以确保分数在距原点+-偏移量的距离范围内采用值衰减

$$\sigma^2 = -scale^2/(2 \cdot ln(decay))$$

2. exp

指数衰减，曲线特点：先剧烈衰减然后变缓。 计算公式：

$$S(doc) = \exp(\lambda \cdot \max(0, |fieldvalue_{doc} - origin| - offset))$$

再次计算参数以确保分数在距原点+-偏移量的距离范围内衰减值

$$\lambda = ln(decay)/scale$$

3. linear

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，计算公式：

$$S(doc) = \max\left(\frac{s - \max\left(0, |fieldvalue_{doc} - origin| - offset\right)}{s}, 0\right)$$

再次计算参数 s 以确保分数在距原点+-偏移量的距离范围内取值衰减

$$s = scale/(1.0 - decay))$$

多值字段 Multi-values fields

如果用于计算衰减的字段包含多个值，默认情况下选择最接近原点的值来确定距离。这可以通过设置 multi_value_mode 来改变。

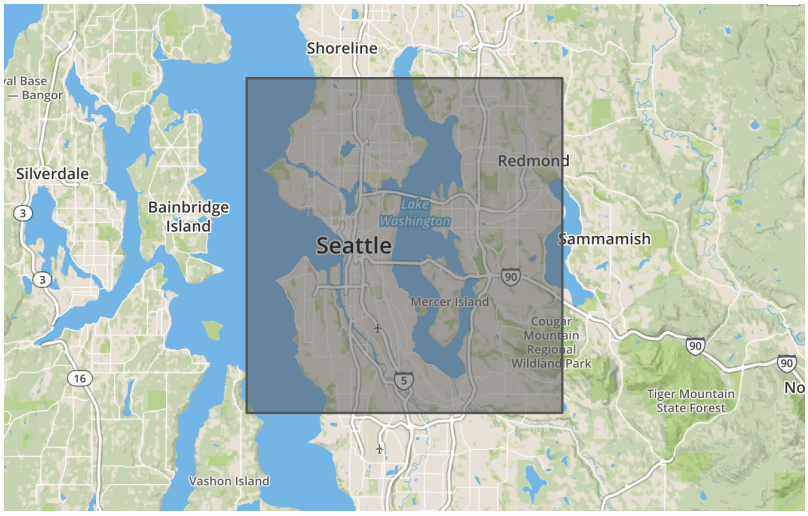
multi_value_mode	含义
min	距离取最小距离值
max	距离取最大距离值
avg	距离是所有距离的的平均值
sum	距离是所有距离的总和

例子

```
    "DECAY_FUNCTION": {
        "FIELD_NAME": {
              "origin": ...,
              "scale": ...
        },
        "multi_value_mode": "avg"
    }
```

边界查询 geo_bounding_box

查找地理点位于指定矩形中的文档。



示例 查询位于指定矩形中的文档

1. 输入 geo_point 类型的索引数据

```
PUT /my_locations
{
    "mappings": {
        "_doc": {
            "properties": {
                "pin": {
                    "properties": {
                        "location": {
                            "type": "geo_point"
                        }
                    }
                }
            }
        }
    }
}

PUT /my_locations/_doc/1
{
    "pin" : {
        "location" : {
```

```
            "lat" : 40.12,
            "lon" : -71.34
        }
    }
}
```

2. 查询位于指定矩形中的文档。

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_bounding_box" : {
                    "pin.location" : {
                        "top_left" : {
                            "lat" : 40.73,
                            "lon" : -74.1
                        },
                        "bottom_right" : {
                            "lat" : 40.01,
                            "lon" : -71.12
                        }
                    }
                }
            }
        }
    }
}
```

多种查询方式示例

1. properties 方式

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
```

```
                "geo_bounding_box" : {
                    "pin.location" : {
                        "top_left" : {
                            "lat" : 40.73,
                            "lon" : -74.1
                        },
                        "bottom_right" : {
                            "lat" : 40.01,
                            "lon" : -71.12
                        }
                    }
                }
            }
        }
    }
}
```

> top_left : 左上角
> bottom_right : 右下角

2. 数组方式

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_bounding_box" : {
                    "pin.location" : {
                        "top_left" : [-74.1, 40.73],
                        "bottom_right" : [-71.12, 40.01]
                    }
                }
            }
        }
    }
}
```

> Array 格式为 [经度, 纬度]

3. 字符串方式

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_bounding_box" : {
                    "pin.location" : {
                        "top_left" : "40.73, -74.1",
                        "bottom_right" : "40.01, -71.12"
                    }
                }
            }
        }
    }
}
```

> String 格式为 "经度, 纬度"

4. Well-Know Text (WKT)

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_bounding_box" : {
                    "pin.location" : {
                        "wkt" : "BBOX (-74.1, -71.12, 40.73, 40.01)"
                    }
                }
            }
        }
    }
}
```

> Well-Know Text (WKT)

54. Geo Hash

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_bounding_box" : {
                    "pin.location" : {
                        "top_left" : "dr5r9ydj2y73",
                        "bottom_right" : "drj7teegpus6"
                    }
                }
            }
        }
    }
}
```

> geo_hash

6. 顶点

边界框的顶点也可以使用简单的 top、left、bottom 和 right 来单独设置值。

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_bounding_box" : {
                    "pin.location" : {
                        "top" : 40.73,
                        "left" : -74.1,
                        "bottom" : 40.01,
                        "right" : -71.12
                    }
                }
            }
        }
    }
}
```

type

bounding box执行的类型默认设置为memory，即在内存中检查doc是否在bounding box范围内。在某些情况下，indexed 选项会执行得更快

> **TIP**
> 注意，在这种情况下不支持多值类的geo_point， 且geo_point的mapping中必须为indexed=true。

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_bounding_box" : {
                    "pin.location" : {
                        "top_left" : {
                            "lat" : 40.73,
                            "lon" : -74.1
                        },
                        "bottom_right" : {
                            "lat" : 40.10,
                            "lon" : -71.12
                        }
                    },
                    "type" : "indexed"
                }
            }
        }
    }
}
```

参数说明

Ignore Unmapped

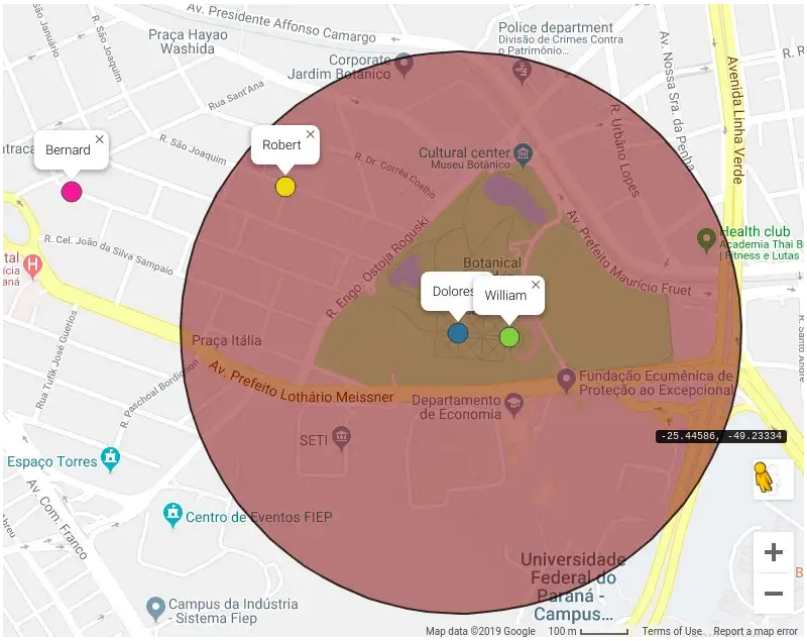
在查询具有不同mapping的多个索引时，ignore_unmapped 选项将忽略未映射的字段。当设置为 false（默认值）时，如果字段未映射，查询将抛出异常。

精度说明

地理点的精度有限，并且在索引时间内始终向下舍入。在查询期间，边界框的上边界向下舍入，而下边界向上舍入。因此，由于舍入误差，下边界上的点（边界框的底部和左边缘）可能无法进入边界框。同时，查询可能会选择上边界（顶部和右边缘）旁边的点，即使它们稍微位于边缘之外。纬度上的舍入误差应小于 4.20e-8 度，经度上的舍入误差应小于 8.39e-8 度，这意味着即使在赤道上也小于 1 厘米的误差。

距离查询 geo_distance

查找地理点位于一个中心点到指定距离范围内的文档。



例子 查询200km半径内的位置

```
PUT /my_locations
{
    "mappings": {
        "_doc": {
            "properties": {
                "pin": {
                    "properties": {
                        "location": {
                            "type": "geo_point"
                        }
                    }
                }
            }
        }
    }
}

PUT /my_locations/_doc/1
{
    "pin" : {
```

```
        "location" : {
            "lat" : 40.12,
            "lon" : -71.34
        }
    }
}
```

然后可以使用 geo_distance 过滤器执行以下简单查询：

```
GET /my_locations/_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_distance" : {
                    "distance" : "200km",
                    "pin.location" : {
                        "lat" : 40,
                        "lon" : -70
                    }
                }
            }
        }
    }
}
```

接受的查询格式

1. properties 方式

```
GET /my_locations/_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_distance" : {
                    "distance" : "12km",
```

```
                    "pin.location" : {
                        "lat" : 40,
                        "lon" : -70
                    }
                }
            }
        }
    }
}
```

2. 数组方式

```
GET /my_locations/_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_distance" : {
                    "distance" : "12km",
                    "pin.location" : [-70, 40]
                }
            }
        }
    }
}
```

3. 字符串方式

```
GET /my_locations/_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_distance" : {
                    "distance" : "12km",
                    "pin.location" : "40,-70"
                }
            }
```

```
        }
    }
}
```

4. Geo Hash方式

```
GET /my_locations/_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_distance" : {
                    "distance" : "12km",
                    "pin.location" : "drm3btev3e86"
                }
            }
        }
    }
}
```

参数说明

geo_distance

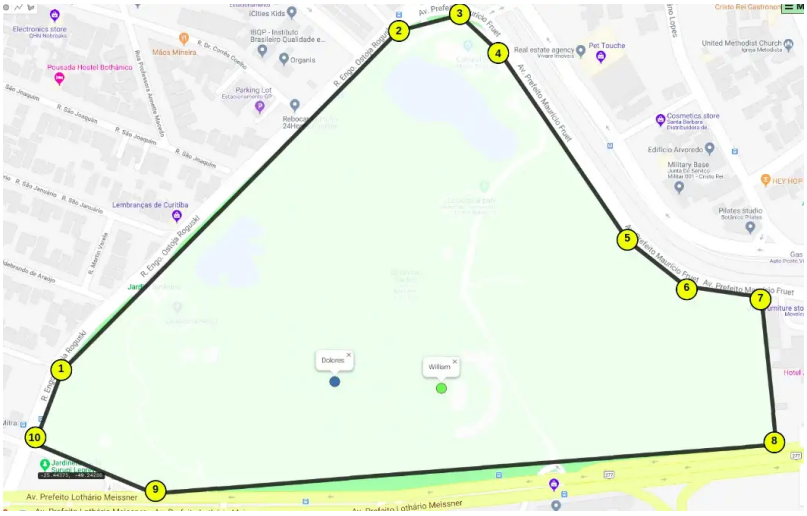
- **distance** 以指定位置为中心的圆的半径。落在这个圆圈中的点被认为是匹配的。可以用各种单位指定距离。请参见距离单位。
- **distance_type** 如何计算距离。可以是`arc`（默认）或`plane`平面（更快，但在长距离和靠近两极时不准确）。
- **_name** 用于标识查询的可选名称字段
- **validation_method** 设置为 `IGNORE_MALFORMED` 以接受具有无效纬度或经度的地理点，设置为 `COERCE` 以另外尝试并推断正确的坐标（默认为 `STRICT`）

Ignore Unmapped

在查询具有不同mapping的多个索引时，ignore_unmapped 选项将忽略未映射的字段。当设置为 false（默认值）时，如果字段未映射，查询将抛出异常。

多边形查询 geo_polygon

查找地理点位于指定多边形的文档。



简单样例

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_polygon" : {
                    "person.location" : {
                        "points" : [
                            {"lat" : 40, "lon" : -70},
                            {"lat" : 30, "lon" : -80},
                            {"lat" : 20, "lon" : -90}
                        ]
                    }
                }
            }
        }
    }
}
```

查询格式

1. 数组方式

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_polygon" : {
                    "person.location" : {
                        "points" : [
                            [-70, 40],
                            [-80, 30],
                            [-90, 20]
                        ]
                    }
                }
            }
        }
    }
}
```

2. 字符串方式

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
               "geo_polygon" : {
                    "person.location" : {
                        "points" : [
                            "40, -70",
                            "30, -80",
                            "20, -90"
                        ]
                    }
                }
```

```
            }
        }
    }
}
```

3. Geohash

```
GET /_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
               "geo_polygon" : {
                    "person.location" : {
                        "points" : [
                            "drn5x1g8cu2y",
                            "30, -80",
                            "20, -90"
                        ]
                    }
                }
            }
        }
    }
}
```

参数说明

Ignore Unmapped

在查询具有不同mapping的多个索引时，ignore_unmapped 选项将忽略未映射的字段。当设置为 false（默认值）时，如果字段未映射，查询将抛出异常。

形状查询 geo_shape

查找与指定的地理形状相交、包含或不相交的地理形状的文档。

示例1. 查询形状内的文档

```
# 写入geo_shape索引

PUT /example
{
    "mappings": {
        "_doc": {
            "properties": {
                "location": {
                    "type": "geo_shape"
                }
            }
        }
    }
}

POST /example/_doc?refresh
{
    "name": "Wind & Wetter, Berlin, Germany",
    "location": {
        "type": "point",
        "coordinates": [13.400544, 52.530286]
```

```
    }
}

# 查询在矩形内的所有文档

GET /example/_search
{
    "query":{
        "bool": {
            "must": {
                "match_all": {}
            },
            "filter": {
                "geo_shape": {
                    "location": {
                        "shape": {
                            "type": "envelope",
                            "coordinates" : [[13.0, 53.0], [14.0, 52.0]]
                        },
                        "relation": "within"
                    }
                }
            }
        }
    }
}

# 查询多边形内的所有文档

GET /example/_search
 {
    "query": {
        "bool": {
            "must": {
                "match_all": {}
            },
            "filter": {
                "geo_shape": {
                    "location": {
                        "shape": {
                            "type": "polygon",
                            "coordinates": [
                                [
                                    [48.854609,2.443977],
                                    [48.854609,2.442428],
                                    [48.855702,2.442198],
                                    [48.855702,2.444138],
                                    [48.854609,2.443977]
                                ]
```

```
                            ]
                        },
                        "relation": "within"
                    }
                }
            }
        }
    }
}
```

示例2. 查询与圆形相交的所有文档

```
 PUT /circle-example
{
    "mappings": {
        "_doc": {
            "properties": {
                "location": {
                    "type": "geo_shape",
                    "strategy": "recursive"
                }
            }
        }
    }
}

POST /circle-example/_doc?refresh
{
    "name": "Leting Sooh",
    "location": {
      "type" : "circle",
      "coordinates" : [70.0, 1.0],
      "radius" : "25mi"
    }
}

GET /circle-example/_search
{
    "query":{
        "bool": {
            "must": {
                "match_all": {}
            },
            "filter": {
                 "geo_shape": {
                    "location": {
```

```
                        "shape": {
                            "type": "circle",
                            "coordinates":  [70.3, 1.2] ,
                            "radius":"2km"
                        },
                        "relation": "INTERSECTS"
                    }
                }
            }
        }
    }
}
```

引用查询

查询时引用一个包含geo_shape的索引文档做更简洁的形状查询。

> 例如输入一个索引，设置为上海市的地理位置形状， 当我们查询上海市内的文档时可以直接引用它而不需要每次都提供它的坐标经纬度。

以下是一个示例：

```
PUT /shapes
{
    "mappings": {
        "_doc": {
            "properties": {
                "location": {
                    "type": "geo_shape"
                }
            }
        }
    }
}

PUT /shapes/_doc/deu
{
    "location": {
        "type": "envelope",
        "coordinates" : [[13.0, 53.0], [14.0, 52.0]]
    }
}

GET /example/_search
{
```

```
    "query": {
        "bool": {
            "filter": {
                "geo_shape": {
                    "location": {
                        "indexed_shape": {
                            "index": "shapes",
                            "type": "_doc",
                            "id": "deu",
                            "path": "location"
                        }
                    }
                }
            }
        }
    }
}
```

参数说明

Ignore Unmapped

在查询具有不同mapping的多个索引时，ignore_unmapped 选项将忽略未映射的字段。当设置为 false（默认值）时，如果字段未映射，查询将抛出异常。

recursive 位置与形状的空间关系

- INTERSECTS 相交
- DISJOINT 不相交
- WITHIN 内
- CONTAINS 包含

查找类似文档 more_like_this

More Like This 查找给定文档集“相似”的文档。

例1. 基础样例

例如，我们要求所有在 “title” 和 “description” 字段中包含类似于 “Once upon a time” 的文本的所有电影，将所选关键词的数量限制为 12。

```
GET /_search
{
    "query": {
        "more_like_this" : {
            "fields" : ["title", "description"],
            "like" : "Once upon a time",
            "min_term_freq" : 1,
            "max_query_terms" : 12
        }
    }
}
```

例2. 查询与存在的文档相似的文档

一个更复杂的用例包括将文本与索引中已存在的文档混合。在这种情况下，指定文档的语法类似于 Multi GET API 中使用的语法。

```
GET /_search
{
    "query": {
        "more_like_this" : {
            "fields" : ["title", "description"],
            "like" : [
            {
                "_index" : "imdb",
                "_type" : "movies",
                "_id" : "1"
            },
            {
                "_index" : "imdb",
```

```
                "_type" : "movies",
                "_id" : "2"
            },
            "and potentially some more text here as well"
            ],
            "min_term_freq" : 1,
            "max_query_terms" : 12
        }
    }
}
```

最后，可以混合一些文本，一组选定的文档，还可以提供索引中不一定存在的文档。

```
GET /_search
{
    "query": {
        "more_like_this" : {
            "fields" : ["name.first", "name.last"],
            "like" : [
            {
                "_index" : "marvel",
                "_type" : "quotes",
                "doc" : {
                    "name": {
                        "first": "Ben",
                        "last": "Grimm"
                    },
                    "_doc": "You got no idea what I'd... what I'd give to be
invisible."
                  }
            },
            {
                "_index" : "marvel",
                "_type" : "quotes",
                "_id" : "2"
            }
            ],
            "min_term_freq" : 1,
            "max_query_terms" : 12
        }
    }
}
```

参数	含义

参数	含义
fields	需要匹配的字段。
like	要匹配的文本。
unlike	unlike 参数与 like 结合使用，用于排除包含不喜欢的文本的文档。
min_term_freq	文档中词项的最低频率，默认是2，低于此频率的文档会被忽略。
max_query_terms	query中能包含的最大词项数目，默认为25。
min_doc_freq	最小的文档频率，默认为5。
max_doc_freq	最大文档频率。
min_word_length	单词的最小长度。
max_word_length	单词的最大长度。
stop_words	停用词列表。
analyzer	分词器。
minimum_should_match	文档应该匹配的最小单词数量，默认为query分词后词项的30%。
boost_terms	词项的权重。
include	是否把输入文档作为结果返回。
boost	整个query的权重，默认为1.0。

脚本查询 script_query

允许将脚本定义为查询的查询。它们通常用于filter子句中，例如：

```
GET /_search
{
    "query": {
        "bool" : {
            "filter" : {
                "script" : {
                    "script" : {
                        "source": "doc['num1'].value > 1",
                        "lang": "painless"
                    }
                }
            }
        }
    }
}
```

自定义参数

脚本被编译和缓存以加快执行速度。使用相同的脚本而使用不同的参数来执行查询，例如

```
GET /_search
{
    "query": {
        "bool" : {
            "filter" : {
                "script" : {
                    "script" : {
                        "source" : "doc['num1'].value > params.param1",
                        "lang"   : "painless",
                        "params" : {
                            "param1" : 5
                        }
                    }
                }
            }
        }
    }
}
```

反向检索 percolate

反向查询：将查询存储到索引中，然后通过Percolate API定义文档以检索这些查询。

使用场景

反向查询通常用于通知的场景

举例：用户订阅了对阿凡达电影感兴趣，当阿凡达电影上映时给该用户发送通知。

```
PUT /my-index
{
    "mappings": {
        "_doc": {
            "properties": {
                "message": {
                    "type": "text"
                },
                "query": {
                    "type": "percolator"
                }
            }
        }
    }
}

PUT /my-index/_doc/1?refresh
{
    "query" : {
        "match" : {
            "message" : "avatar"
        }
    }
}
```

执行反向查询，将文档文本与注册的query进行匹配：

```
GET /my-index/_search
{
    "query" : {
        "percolate" : {
```

```
            "field" : "query",
            "document" : {
                "message" : "avatar is offer"
            }
        }
    }
}
```

返回结果

```
{
  "took": 13,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped" : 0,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 0.5753642,
    "hits": [
      {
        "_index": "my-index",
        "_type": "_doc",
        "_id": "1",
        "_score": 0.5753642,
        "_source": {
          "query": {
            "match": {
              "message": "avatar"
            }
          }
        },
        "fields" : {
          "_percolator_document_slot" : [0]
        }
      }
    ]
  }
}
```

1. id 为 1 的查询与我们的文档匹配。
2. _percolator_document_slot 字段指示哪个文档与此查询匹配。

参数	含义
field	（必填项） 保存索引查询的过滤器类型的字段。
name	（可选项）如果指定了多个渗透查询，则用于 _percolator_document_slot 字段的后缀。这是一个可选参数。
document	正在渗透的文档的来源。
documents	与文档参数类似，但通过 json 数组接受多个文档。

Percolating 关闭算分以提升性能

```
GET /my-index/_search
{
    "query" : {
        "constant_score": {
            "filter": {
                "percolate" : {
                    "field" : "query",
                    "document" : {
                        "message" : "avatar is offer"
                    }
                }
            }
        }
    }
}
```

如果不需要计算分数，则应该将Percolating Query 包装在 Constant Score Query 或 Bool 查询的 filter 子句中，以获取查询性能的提升。

反向查找多个文档

percolate 查询可以使用索引的 percolator 查询同时匹配多个文档。在单个请求中渗透多个文档可以提高性能，因为查询只需要解析和匹配一次而不是多次。每个匹配的过滤器查询返回的 _percolator_document_slot 字段在同时过滤多个文档时很重要。它指示哪些文档与特定的过滤器查询匹配。这些数字与渗透查询中指定的文档数组中的slot相关

```
GET /my-index/_search
{
    "query" : {
        "percolate" : {
            "field" : "query",
            "documents" : [
                {
                    "message" : "avatar is offer"
                },
                {
                    "message" : "offer movies"
                },
                {
                    "message" : "Sparta is offer"
                }
            ]
        }
    }
}
```

返回

```
{
  "took": 13,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped" : 0,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 1.5606477,
    "hits": [
      {
        "_index": "my-index",
        "_type": "_doc",
        "_id": "1",
        "_score": 1.5606477,
        "_source": {
          "query": {
            "match": {
              "message": "bonsai tree"
            }
          }
```

```
        },
        "fields" : {
          "_percolator_document_slot" : [0, 1, 3]
        }
      }
    ]
  }
}
```

反向查找已存在的文档

```
PUT /my-index/_doc/2
{
  "message" : "A new bonsai tree in the office"
}
```

percolate 基于已存在的索引构建新搜索请求：

```
GET /my-index/_search
{
    "query" : {
        "percolate" : {
            "field": "query",
            "index" : "my-index",
            "type" : "_doc",
            "id" : "2",
            "version" : 1
        }
    }
}
```

> 版本是可选的，但在某些情况下很有用。我们可以确保我们正在尝试渗透我们刚刚索引的文档。在我们建立索引后可能会进行更改，如果是这种情况，搜索请求将因版本冲突错误而失败

反向查找与高亮

```
PUT /my-index/_doc/3?refresh
{
    "query" : {
        "match" : {
```

```
            "message" : "brown fox"
        }
    }
}

PUT /my-index/_doc/4?refresh
{
    "query" : {
        "match" : {
            "message" : "lazy dog"
        }
    }
}
```

执行高亮查询：

```
GET /my-index/_search
{
    "query" : {
        "percolate" : {
            "field": "query",
            "document" : {
                "message" : "The quick brown fox jumps over the lazy dog"
            }
        }
    },
    "highlight": {
      "fields": {
        "message": {}
      }
    }
}
```

返回

```
{
  "took": 7,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped" : 0,
    "failed": 0
  },
```

```
  "hits": {
    "total": 2,
    "max_score": 0.5753642,
    "hits": [
      {
        "_index": "my-index",
        "_type": "_doc",
        "_id": "4",
        "_score": 0.5753642,
        "_source": {
          "query": {
            "match": {
              "message": "lazy dog"
            }
          }
        },
        "highlight": {
          "message": [
            "The quick brown fox jumps over the <em>lazy</em> <em>dog</em>"
          ]
        },
        "fields" : {
          "_percolator_document_slot" : [0]
        }
      },
      {
        "_index": "my-index",
        "_type": "_doc",
        "_id": "3",
        "_score": 0.5753642,
        "_source": {
          "query": {
            "match": {
              "message": "brown fox"
            }
          }
        },
        "highlight": {
          "message": [
            "The quick <em>brown</em> <em>fox</em> jumps over the lazy dog"
          ]
        },
        "fields" : {
          "_percolator_document_slot" : [0]
        }
      }
    ]
  }
}
```

搜索请求中的查询不是高亮过滤结果，而是高亮过滤查询中定义的文档。 当同时过滤多个文档时（如下面的请求），高亮响应是不同的

```
GET /my-index/_search
{
    "query" : {
        "percolate" : {
            "field": "query",
            "documents" : [
                {
                    "message" : "bonsai tree"
                },
                {
                    "message" : "new tree"
                },
                {
                    "message" : "the office"
                },
                {
                    "message" : "office tree"
                }
            ]
        }
    },
    "highlight": {
      "fields": {
        "message": {}
      }
    }
}
```

略有不同的响应：

```
{
  "took": 13,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped" : 0,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 1.5606477,
```

```
    "hits": [
      {
        "_index": "my-index",
        "_type": "_doc",
        "_id": "1",
        "_score": 1.5606477,
        "_source": {
          "query": {
            "match": {
              "message": "bonsai tree"
            }
          }
        },
        "fields" : {
          "_percolator_document_slot" : [0, 1, 3]
        },
        "highlight" : {
          "0_message" : [
              "<em>bonsai</em> <em>tree</em>"
          ],
          "3_message" : [
              "office <em>tree</em>"
          ],
          "1_message" : [
              "new <em>tree</em>"
          ]
        }
      }
    ]
  }
}
```

指定多个反向查询

可以在单个搜索请求中指定多个反向查询

```
GET /my-index/_search
{
    "query" : {
        "bool" : {
            "should" : [
                {
                    "percolate" : {
                        "field" : "query",
                        "document" : {
                            "message" : "bonsai tree"
```

```
                        },
                        "name": "query1"
                    }
                },
                {
                    "percolate" : {
                        "field" : "query",
                        "document" : {
                            "message" : "tulip flower"
                        },
                        "name": "query2"
                    }
                }
            ]
        }
    }
}
```

name 参数将用于标识哪个 percolator document slots 属于哪个 percolate 查询。

_percolator_document_slot 字段名称将以 _name 参数中指定的内容为后缀。如果未指定，则将使用 field 参数，在这种情况下会导致歧义。

上面的搜索请求返回类似这样的响应

```
{
  "took": 13,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped" : 0,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 0.5753642,
    "hits": [
      {
        "_index": "my-index",
        "_type": "_doc",
        "_id": "1",
        "_score": 0.5753642,
        "_source": {
          "query": {
            "match": {
```

```
            "message": "bonsai tree"
          }
        }
      },
      "fields" : {
        "_percolator_document_slot_query1" : [0]
      }
    }
  ]
}
}
```

包装器查询 wrapper

接受任何其他查询作为 base64 编码字符串的查询。

```
GET /_search
{
    "query" : {
        "wrapper": {
            "query" : "eyJ0ZXJtIiA6IHsgInVzZXIiIDogIktpbWNoeSIgfX0="
        }
    }
}
```

> Base64 编码字符串：{"term" : { "user" : "Kimchy" }}

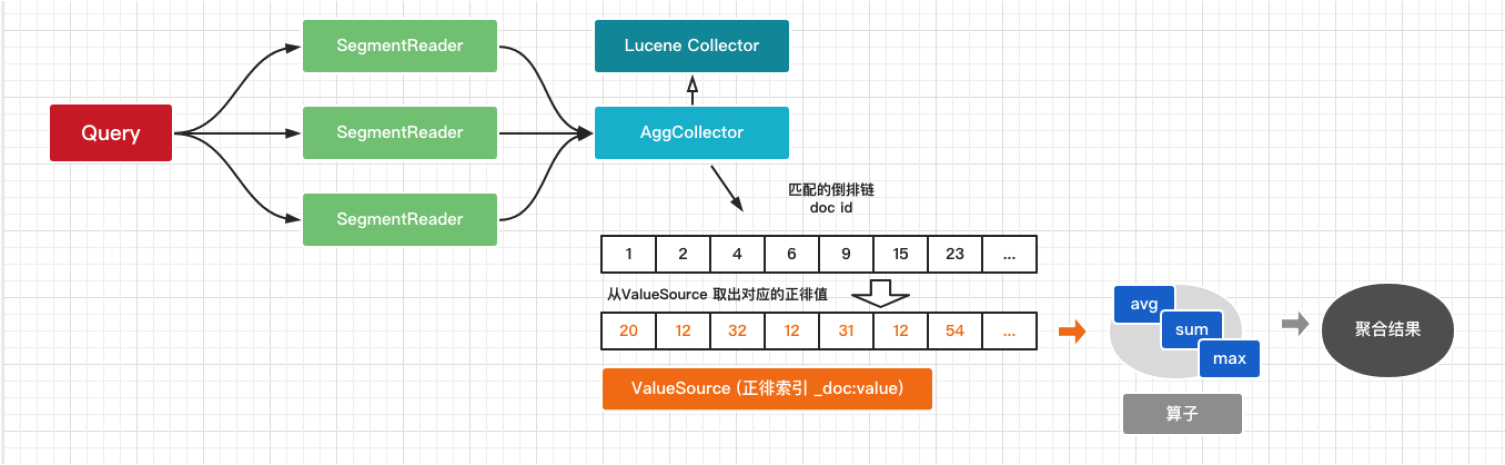
此查询在 Java high-level REST Client 更有用，它可以接受查询作为 json 格式的字符串。在这些情况下，可以将查询指定为 json 或 yaml 格式的string 或 query builder.

聚合概览

Aggregations

Elasticsearch提供了Aggregation聚合功能辅助我们进行数据的分析，原理是基于查询(Query)在倒排链的形成过程中增加了二次聚合计算过程。如下图

1. 执行一个Query。在检索过程中，底层的 Lucene Collector 会为匹配到的文档形成一条链式结构。doc1,doc2,doc4 ...
2. AggCollector作为一个 Filted Collector，根据取到的doc id从正排索引拿出字段值并丢给算子执行汇总计算。



了解了基础原理，我们便可以理解在使用聚合分析时：

1. 只有开启正排的索引才能进行聚合计算, doc_value=true
2. 聚合计算和Query的size无关，因为它发生在倒排检索阶段，通常我们可以设置size=0来优化性能。
3. 聚合计算的代价由Query匹配的文档数决定(倒排链越长代价越高,涉及到抓取和计算)，如计算量过大可以设置query 的 terminate_after 减少匹配文档数。
4. 聚合计算过程发生在算分的前一阶段，也就是和算分无关，通常我们可以通过 Constant Score Query 或 Filter 子句关闭算分，降低倒排查询代价。

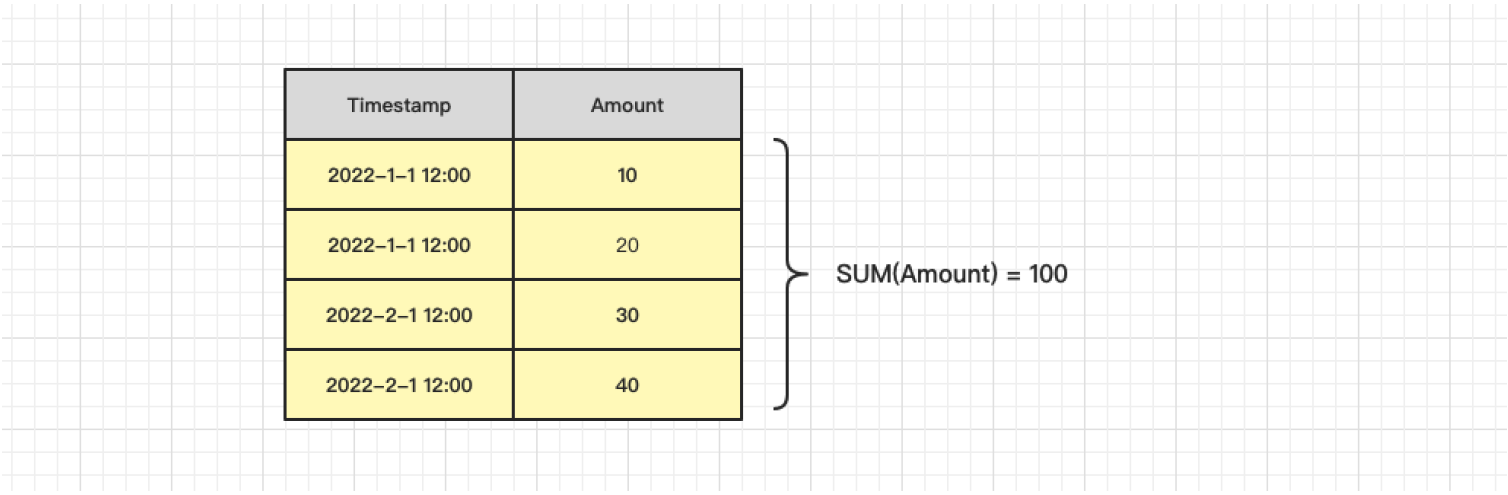
概念

- **Bucketing (桶聚合)** 满足特定条件的文档的集合。
- **Metric (指标聚合)** 对桶内的文档进行统计计算。

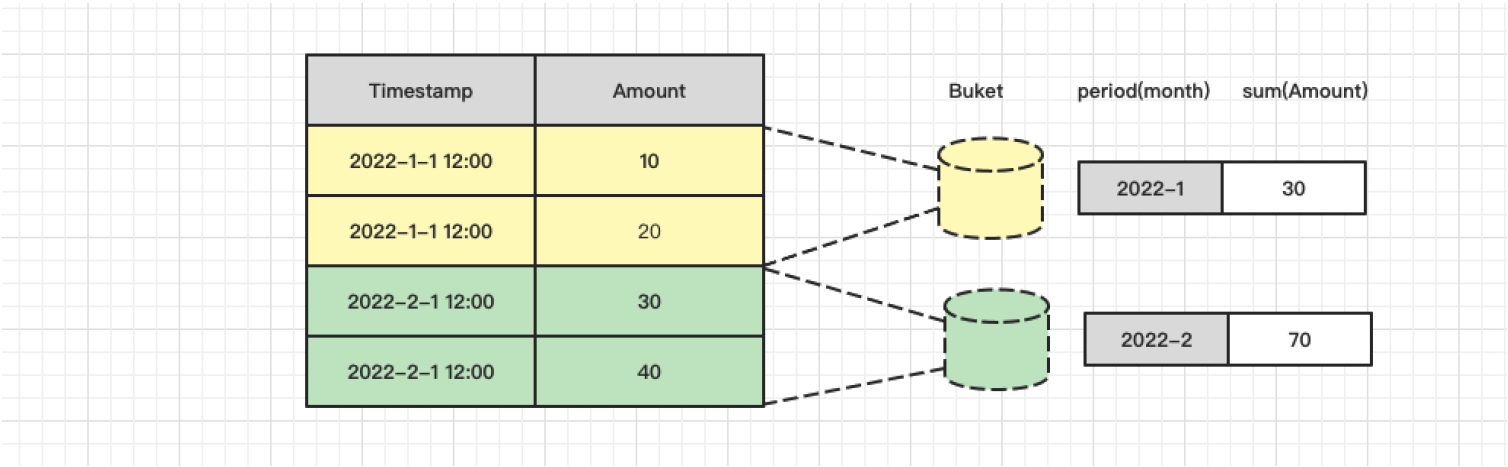
- **Pipeline (管道聚合)** 聚合其他聚合的输出及其相关度量的聚合。
- **Matrix (矩阵聚合)** 对多个字段进行操作并根据从请求的文档字段中提取的值生成矩阵结果的聚合族。

例如有一组商品的销售数据。

1. 指标聚合：统计全部的总金额。

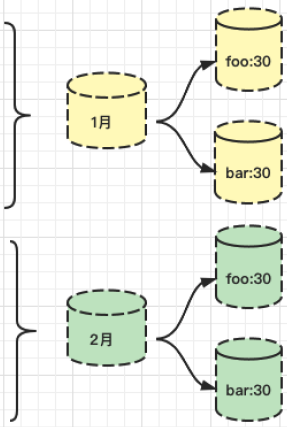


2. 桶聚合：增加按月汇总。



3. 管道聚合：基于按月的统计结果再按销售员的统计

Timestamp	Amount	sales
2022–1–1 12:00	10	foo
2022–1–2 12:00	20	foo
2022–1–3 12:00	10	bar
2022–1–4 12:00	20	bar
2022–2–1 12:00	10	foo
2022–2–2 12:00	20	foo
2022–2–3 12:00	10	bar
2022–2–4 12:00	20	bar



基础函数聚合 avg,sum,max,min..

最常用的基础函数算子 `avg、max、min、sum`， 属于单值度量聚合。

Avg Aggregation 取平均值

例如假设数据由代表学生考试成绩（介于0和100之间）的文件组成，我们可以用以下公式对他们的分数进行平均

```
POST /exams/_search?size=0
{
    "aggs" : {
        "avg_grade" : { "avg" : { "field" : "grade" } }
    }
}
```

Sum Aggregation 取总和值

假设数据由代表销售记录的文件组成，我们可以将所有帽子的销售价格加起来

```
POST /sales/_search?size=0
{
    "query" : {
        "constant_score" : {
            "filter" : {
                "match" : { "type" : "hat" }
            }
        }
    },
    "aggs" : {
        "hat_prices" : { "sum" : { "field" : "price" } }
    }
}
```

Max Aggregation 取最大值

计算所有文档的最大价格

```
POST /sales/_search?size=0
{
    "aggs" : {
        "max_price" : { "max" : { "field" : "price" } }
    }
}
```

Min Aggregation 取最小值

计算所有文档的最小价格

```
POST /sales/_search?size=0
{
    "aggs" : {
        "min_price" : { "min" : { "field" : "price" } }
    }
}
```

Script

根据脚本计算平均成绩：

```
POST /exams/_search?size=0
{
    "aggs" : {
        "avg_grade" : {
            "avg" : {
                "script" : {
                    "source" : "doc.grade.value"
                }
            }
        }
    }
}
```

脚本参数方式

```
```

```
POST /exams/_search?size=0
{
    "aggs" : {
        "avg_grade" : {
            "avg" : {
                "script" : {
                    "id": "my_script",
                    "params": {
                        "field": "grade"
                    }
                }
            }
        }
    }
}
```

Value Script

还可以使用_value从脚本中访问字段值。例如，这将计算所有帽子价格的平方：

```
POST /sales/_search?size=0
{
    "query" : {
        "constant_score" : {
            "filter" : {
                "match" : { "type" : "hat" }
            }
        }
    },
    "aggs" : {
        "square_hats" : {
            "sum" : {
                "field" : "price",
                "script" : {
                    "source": "_value * _value"
                }
            }
        }
    }
}
```

Missing value

缺失值处理，如下例 grade 字段中没有值的文档将与值为10的文档属于同一个存储桶。

```
POST /exams/_search?size=0
{
    "aggs" : {
        "grade_avg" : {
            "avg" : {
                "field" : "grade",
                "missing": 10
            }
        }
    }
}
```

加权平均值 weighted_avg

Weighted Avg Aggregation

单值度量聚合，用于计算从聚合文档中提取的数值的加权平均值。这些值可以从文档中的数字字段中提取。

当计算常规平均值时，每个数据点都有一个相等的“权重”……它对最终值的贡献相等。另一方面，加权平均值对每个数据点的权重不同。每个数据点对最终值的贡献量从文档中提取，或由脚本提供。

作为公式，加权平均值为 ∑(value * weight) / ∑(weight)

规则平均值可以被认为是加权平均值，其中每个值的隐式权重为1。

示例

如果我们的文档有一个包含0-100数字分数的“grade”字段和一个包含任意数字权重的“weight”字段，我们可以使用以下公式计算加权平均值：

```
POST /exams/_search
{
    "size": 0,
    "aggs" : {
        "weighted_grade": {
            "weighted_avg": {
                "value": {
                    "field": "grade"
                },
                "weight": {
                    "field": "weight"
                }
            }
        }
    }
}
```

返回的结果

```
{
    ...
    "aggregations": {
        "weighted_grade": {
            "value": 70.0
        }
    }
}
```

虽然每个字段允许多个值，但只允许一个权重。如果聚合遇到具有多个权重的文档（例如，权重字段是多值字段），它将抛出异常。如果出现这种情况，则需要为权重字段指定一个脚本，并使用该脚本将多个值组合成一个要使用的值。

此单个权重将独立应用于从值字段中提取的每个值。

此示例显示了如何使用单个权重对具有多个值的单个文档进行平均值计算

```
POST /exams/_doc?refresh
{
    "grade": [1, 2, 3],
    "weight": 2
}

POST /exams/_search
{
    "size": 0,
    "aggs" : {
        "weighted_grade": {
            "weighted_avg": {
                "value": {
                    "field": "grade"
                },
                "weight": {
                    "field": "weight"
                }
            }
        }
    }
}
```

三个值（1、2和3）将作为独立值包括在内，所有值的权重均为2：

```
{
    ...
    "aggregations": {
        "weighted_grade": {
            "value": 2.0
        }
    }
}
```

聚合返回2.0作为结果，这与我们手动计算时的预期一致：`（1*2）+（2*2）+（3*2））/（2+2+2）==2`

Script

值和权重都可以从脚本而不是字段中导出。作为一个简单的示例，下面将使用脚本为文档中的grade和weight 加1：

```
POST /exams/_search
{
    "size": 0,
    "aggs" : {
        "weighted_grade": {
            "weighted_avg": {
                "value": {
                    "script": "doc.grade.value + 1"
                },
                "weight": {
                    "script": "doc.weight.value + 1"
                }
            }
        }
    }
}
```

Missing values

缺少的参数定义了应如何处理缺少值的文档。值和权重的默认行为不同：

默认情况下，如果缺少值字段，则忽略该文档，聚合将转到下一个文档。如果缺少权重字段，则假设其权重为1（与正常平均值一样）。

这两个默认值都可以用缺少的参数重写：

```
POST /exams/_search
{
    "size": 0,
    "aggs" : {
        "weighted_grade": {
            "weighted_avg": {
                "value": {
                    "field": "grade",
                    "missing": 2
                },
                "weight": {
                    "field": "weight",
                    "missing": 3
                }
            }
        }
    }
}
```

参数说明

weighted_avg

参数	说明	缺省
value	提供值的字段或脚本的配置	必填
weight	提供权重的字段或脚本的配置	必填
format	数字响应格式化	可选
value_type	关于纯脚本或未映射字段的值的提示	可选

value

参数	说明	缺省
field	应从中提取值的字段	必填

参数	说明	缺省
missing	字段完全缺失时使用的值	可选

weight

参数	说明	缺省
field	应从中提取权重的字段	必填
missing	字段完全缺失时使用的权重	可选

基数聚合 cardinality

Cardinality Aggregation

计算不同值的近似计数的单值度量聚合。值可以从文档中的特定字段提取，也可以由脚本生成。

假设您正在为商店销售编制索引，并希望统计与查询匹配的已售出产品的唯一数量：

```
POST /sales/_search?size=0
{
    "aggs" : {
        "type_count" : {
            "cardinality" : {
                "field" : "type"
            }
        }
    }
}
```

返回结果

```
{
    ...
    "aggregations" : {
        "type_count" : {
            "value" : 3
        }
    }
}
```

精度控制

```
POST /sales/_search?size=0
{
    "aggs" : {
        "type_count" : {
            "cardinality" : {
                "field" : "_doc",
                "precision_threshold": 100
```

```
            }
        }
    }
}
```

precision_threshold选项允许以内存换取精度，并定义一个唯一的计数，低于该计数的计数将接近精度。高于此值，计数可能会变得更加模糊。支持的最大值为40000，高于此值的阈值将与40000的阈值具有相同的效果。默认值为3000。

近似值计数

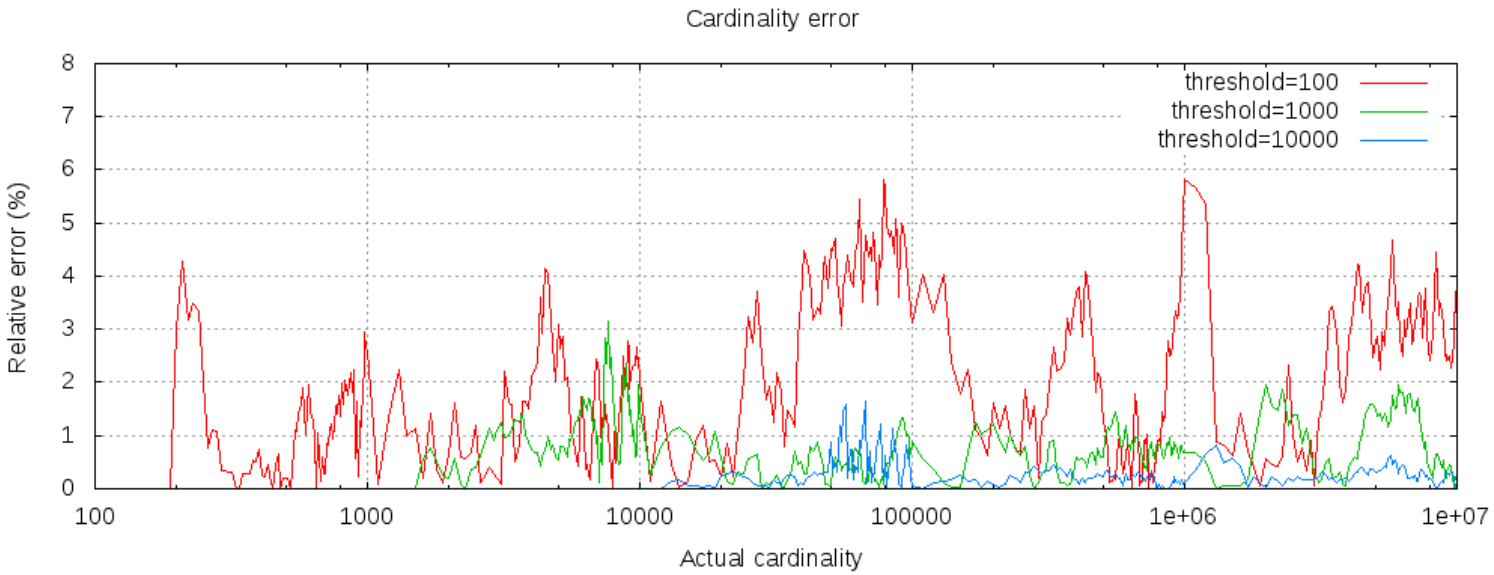
计算精确计数需要将值加载到哈希集并返回其大小。当处理高基数集和/或大值时，这不会扩展，因为所需的内存使用量和节点之间每个shard集的通信需要会占用集群的太多资源。

这种基数聚合基于HyperLogLog++算法，该算法基于具有一些有趣属性的值的哈希值进行计数：

- 可配置精度，其决定如何以内存换取精度，
- 在低基数集合上具有优异的精度，
- 固定内存使用量：无论是否存在数百或数十亿个唯一值，内存使用量仅取决于配置的精度。

对于精度阈值c，我们使用的实现需要大约c*8字节。

下表显示了阈值前后的误差变化情况



对于所有3个阈值，计数都精确到配置的阈值。虽然不能保证，但情况很可能是这样。实践中的准确性取决于所讨论的数据集。一般来说，大多数数据集显示出一致的良好准确性。还要注意的是，即使阈值低至100，即使在计算数百万项时，误差仍然很低（如上图所示，为1-6%）。

HyperLogLog++算法依赖于哈希值的前导零，数据集中哈希值的精确分布会影响基数的准确性。

还请注意，即使阈值低至100，错误仍然很低，即使在计算数百万项时也是如此。

预计算 hash

对于具有高基数的字符串字段，将字段值的哈希存储在索引中，然后对该字段运行基数聚合可能会更快。这可以通过从客户端提供哈希值来实现，也可以通过使用mapper-murmur3插件让Elasticsearch为您计算哈希值。

> 预计算散列通常只在非常大和/或高基数字段上有用，因为它节省了CPU和内存。然而，在数字字段上，哈希运算非常快，存储原始值所需的内存与存储哈希值所需内存相同或更少。对于低基数字符串字段也是如此，特别是考虑到这些字段进行了优化，以确保每个段的每个唯一值最多计算一次散列

Script

cardinality 度量支持脚本编写，但由于需要实时计算哈希值，因此性能会受到显著影响。

```
POST /sales/_search?size=0
{
    "aggs" : {
        "type_promoted_count" : {
            "cardinality" : {
                "script": {
                    "lang": "painless",
                    "source": "doc['type'].value + ' ' + doc['promoted'].value"
                }
            }
        }
    }
}
```

这将把脚本参数解释为具有painless语言且没有脚本参数的inline script。要使用存储的脚本，请使用以下语法：

```
POST /sales/_search?size=0
{
    "aggs" : {
        "type_promoted_count" : {
            "cardinality" : {
                "script" : {
```

```
                    "id": "my_script",
                    "params": {
                        "type_field": "_doc",
                        "promoted_field": "promoted"
                    }
                }
            }
        }
    }
}
```

Missing value

缺少的参数定义了应如何处理缺少值的文档。默认情况下，它们将被忽略，但也可以将它们视为具有值。

```
POST /sales/_search?size=0
{
    "aggs" : {
        "tag_cardinality" : {
            "cardinality" : {
                "field" : "tag",
                "missing": "N/A"
            }
        }
    }
}
```

标记字段中没有值的文档将与值为N/a的文档属于同一存储桶。

地理边界聚合 geo_bounds

Geo Bounds Aggregation

一种度量聚合，用于计算包含字段的所有geo_point值的边界框。

例子

```
PUT /museums
{
    "mappings": {
        "_doc": {
            "properties": {
                "location": {
                    "type": "geo_point"
                }
            }
        }
    }
}

POST /museums/_doc/_bulk?refresh
{"index":{"_id":1}}
{"location": "52.374081,4.912350", "name": "NEMO Science Museum"}
{"index":{"_id":2}}
{"location": "52.369219,4.901618", "name": "Museum Het Rembrandthuis"}
{"index":{"_id":3}}
{"location": "52.371667,4.914722", "name": "Nederlands Scheepvaartmuseum"}
{"index":{"_id":4}}
{"location": "51.222900,4.405200", "name": "Letterenhuis"}
{"index":{"_id":5}}
{"location": "48.861111,2.336389", "name": "Musée du Louvre"}
{"index":{"_id":6}}
{"location": "48.860000,2.327000", "name": "Musée d'Orsay"}

POST /museums/_search?size=0
{
    "query" : {
        "match" : { "name" : "musée" }
    },
    "aggs" : {
        "viewport" : {
            "geo_bounds" : {
```

```
                "field" : "location",
                "wrap_longitude" : true
            }
        }
    }
}
```

1. `geo_bounds`聚合指定用于获取边界的字段
2. `wraplongitude`是一个可选参数，指定是否允许边界框与国际日期线重叠。默认值为true

上面的聚合演示了如何计算业务类型为商店的所有文档的位置字段的边界框

响应结果

```
{
    ...
    "aggregations": {
        "viewport": {
            "bounds": {
                "top_left": {
                    "lat": 48.86111099738628,
                    "lon": 2.3269999679178
                },
                "bottom_right": {
                    "lat": 48.85999997612089,
                    "lon": 2.3363889567553997
                }
            }
        }
    }
}
```

地理重心聚合 geo_centroid

Geo Centroid Aggregation

从地理点数据类型字段的所有坐标值计算加权矩心(centroid)的度量聚合。

例子

```
PUT /museums
{
    "mappings": {
        "_doc": {
            "properties": {
                "location": {
                    "type": "geo_point"
                }
            }
        }
    }
}

POST /museums/_doc/_bulk?refresh
{"index":{"_id":1}}
{"location": "52.374081,4.912350", "city": "Amsterdam", "name": "NEMO Science
Museum"}
{"index":{"_id":2}}
{"location": "52.369219,4.901618", "city": "Amsterdam", "name": "Museum Het
Rembrandthuis"}
{"index":{"_id":3}}
{"location": "52.371667,4.914722", "city": "Amsterdam", "name": "Nederlands
Scheepvaartmuseum"}
{"index":{"_id":4}}
{"location": "51.222900,4.405200", "city": "Antwerp", "name": "Letterenhuis"}
{"index":{"_id":5}}
{"location": "48.861111,2.336389", "city": "Paris", "name": "Musée du Louvre"}
{"index":{"_id":6}}
{"location": "48.860000,2.327000", "city": "Paris", "name": "Musée d'Orsay"}

POST /museums/_search?size=0
{
    "aggs" : {
        "centroid" : {
            "geo_centroid" : {
```

```
                "field" : "location"
            }
        }
    }
}
```

上面的汇总演示了如何计算盗窃犯罪类型的所有文档的位置字段的形心

上述聚合的响应：

```
{
    ...
    "aggregations": {
        "centroid": {
            "location": {
                "lat": 51.00982963107526,
                "lon": 3.9662130922079086
            },
            "count": 6
        }
    }
}
```

当geo_centroid聚合作为其他桶聚合的子聚合组合时，它更有趣。

```
POST /museums/_search?size=0
{
    "aggs" : {
        "cities" : {
            "terms" : { "field" : "city.keyword" },
            "aggs" : {
                "centroid" : {
                    "geo_centroid" : { "field" : "location" }
                }
            }
        }
    }
}
```

上面的示例使用geo_centroid作为术语桶聚合的子聚合，用于查找每个城市博物馆的中心位置。

上述聚合的响应

```
{
    ...
    "aggregations": {
        "cities": {
            "sum_other_doc_count": 0,
            "doc_count_error_upper_bound": 0,
            "buckets": [
               {
                   "key": "Amsterdam",
                   "doc_count": 3,
                   "centroid": {
                      "location": {
                         "lat": 52.371655656024814,
                         "lon": 4.909563297405839
                      },
                      "count": 3
                   }
               },
               {
                   "key": "Paris",
                   "doc_count": 2,
                   "centroid": {
                      "location": {
                         "lat": 48.86055548675358,
                         "lon": 2.3316944623366
                      },
                      "count": 2
                   }
                },
                {
                    "key": "Antwerp",
                    "doc_count": 1,
                    "centroid": {
                       "location": {
                          "lat": 51.22289997059852,
                          "lon": 4.40519998781383
                       },
                       "count": 1
                    }
                 }
            ]
        }
    }
}
```

百分比聚合 percentiles

Percentiles Aggregation

多值度量聚合，计算从聚合文档中提取的数值的一个或多个百分位数。这些值可以从文档中的特定数字字段中提取，也可以由提供的脚本生成。

百分位数显示观察值的某个百分比出现的点。例如，第95百分位是大于观察值的95%的值。

百分比通常用于查找异常值。在正态分布中，0.13和99.87个百分位数代表与平均值的三个标准差。任何超出三个标准偏差的数据通常被视为异常。

当检索到一个百分位数范围时，可以使用它们来估计数据分布，并确定数据是否倾斜、双峰等。

假设您的数据包含网站加载时间。平均加载时间和中值加载时间对管理员来说并不太有用，最大值可能更有趣。

```
GET latency/_search
{
    "size": 0,
    "aggs" : {
        "load_time_outlier" : {
            "percentiles" : {
                "field" : "load_time"
            }
        }
    }
}
```

默认情况下，百分位数度量将生成一个百分位数范围：[1，5，25，50，75，95，99]。响应如下：

```
{
    ...

   "aggregations": {
      "load_time_outlier": {
         "values" : {
            "1.0": 5.0,
            "5.0": 25.0,
```

```
            "25.0": 165.0,
            "50.0": 445.0,
            "75.0": 725.0,
            "95.0": 945.0,
            "99.0": 985.0
         }
      }
   }
}
```

如您所见，聚合将返回默认范围内每个百分位数的计算值。如果我们假设响应时间以毫秒为单位，那么很明显网页的加载时间通常为10-725ms，但偶尔会达到945-985ms。

通常，管理员只对异常值感兴趣—极端百分位数。我们可以只指定感兴趣的百分比（请求的百分比必须是介于0-100之间的值）：

```
GET latency/_search
{
    "size": 0,
    "aggs" : {
        "load_time_outlier" : {
            "percentiles" : {
                "field" : "load_time",
                "percents" : [95, 99, 99.9]
            }
        }
    }
}
```

> 使用percents参数指定要计算的特定百分比

Keyed Response

默认情况下，keyed标志设置为true，它将一个唯一的字符串键与每个bucket相关联，并将范围作为哈希而不是数组返回。将键控标志设置为false将禁用此行为：

```
GET latency/_search
{
    "size": 0,
    "aggs": {
        "load_time_outlier": {
            "percentiles": {
```

```
                "field": "load_time",
                "keyed": false
            }
        }
    }
}
```

响应

```
{
    ...

    "aggregations": {
        "load_time_outlier": {
            "values": [
                {
                    "key": 1.0,
                    "value": 5.0
                },
                {
                    "key": 5.0,
                    "value": 25.0
                },
                {
                    "key": 25.0,
                    "value": 165.0
                },
                {
                    "key": 50.0,
                    "value": 445.0
                },
                {
                    "key": 75.0,
                    "value": 725.0
                },
                {
                    "key": 95.0,
                    "value": 945.0
                },
                {
                    "key": 99.0,
                    "value": 985.0
                }
            ]
        }
    }
}
```

Script

百分比度量支持脚本。例如，如果我们的加载时间以毫秒为单位，但我们希望百分位数以秒为单位计算，那么我们可以使用脚本动态转换它们：

```
GET latency/_search
{
    "size": 0,
    "aggs" : {
        "load_time_outlier" : {
            "percentiles" : {
                "script" : {
                    "lang": "painless",
                    "source": "doc['load_time'].value / params.timeUnit",
                    "params" : {
                        "timeUnit" : 1000
                    }
                }
            }
        }
    }
}
```

1. `field`参数替换为脚本参数，该参数使用脚本生成计算百分位数的值
2. 与任何其他脚本一样，脚本支持参数化输入

这将把脚本参数解释为具有painless script且没有脚本参数的inline script。要使用存储的脚本，请使用以下语法：

```
GET latency/_search
{
    "size": 0,
    "aggs" : {
        "load_time_outlier" : {
            "percentiles" : {
                "script" : {
                    "id": "my_script",
                    "params": {
                        "field": "load_time"
                    }
                }
            }
```

```
        }
    }
}
```

百分比（通常）是近似值

这里有许多不同的算法来计算百分位数。天真的实现只是将所有值存储在一个排序数组中。要找到第50个百分位数，只需找到my_array[count（my_array）*0.5]处的值。

很明显，幼稚的实现并不能扩展—排序数组随数据集中值的数量线性增长。为了计算Elasticsearch集群中潜在数十亿值的百分位数，需要计算近似百分位数。

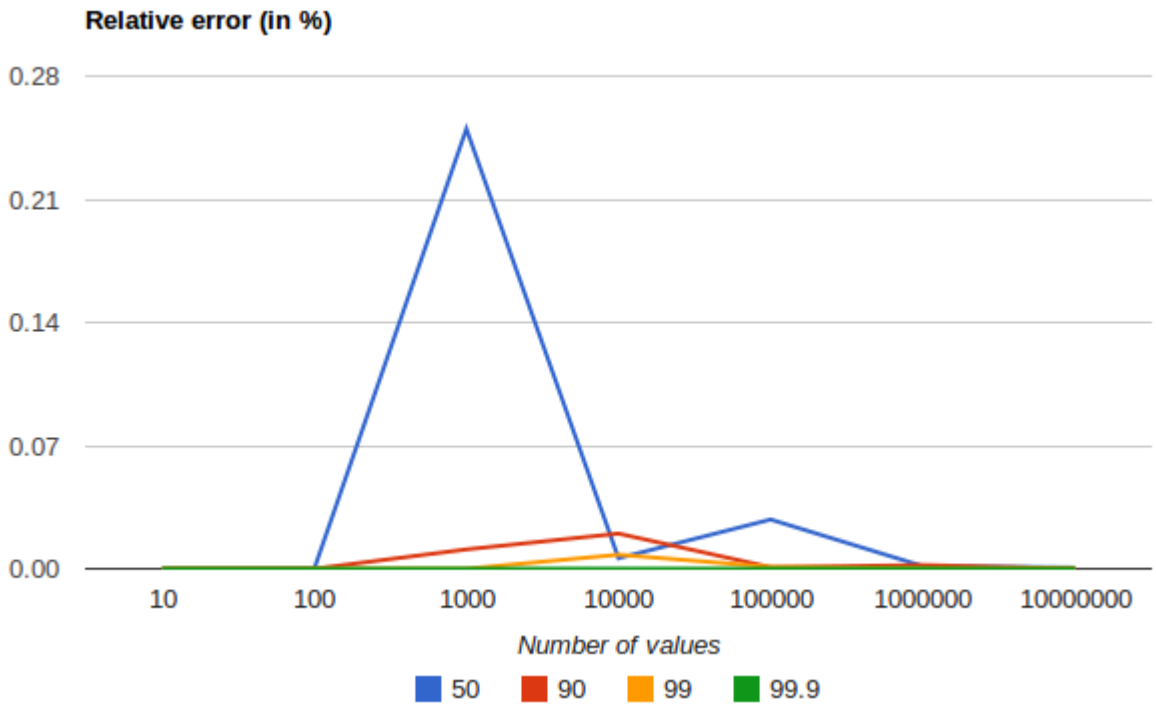
百分位数度量所使用的算法称为TDigest（由Ted Dunning在使用T位数计算精确分位数中介绍）。

在使用此度量时，需要记住以下几条准则：

精度与q（1-q）成正比。这意味着极端百分位数（例如99%）比不太极端的百分位数更准确，例如中值

对于小的值集，百分位数是高度准确的（如果数据足够小，则可能100%准确）。

作为a b中的值的数量



它显示了极端百分位数的精度如何更好。对于大量值，误差减小的原因是，大数定律使值的分布越来越均匀，t-digest tree可以更好地总结它。对于更为倾斜的分布，情况并非如此。

> 百分比聚合也是非确定性的。这意味着使用相同的数据可以得到稍微不同的结果。

压缩

近似算法必须平衡内存利用率和估计精度。可以使用压缩参数控制该平衡：

```
GET latency/_search
{
    "size": 0,
    "aggs" : {
        "load_time_outlier" : {
            "percentiles" : {
                "field" : "load_time",
                "tdigest": {
                  "compression" : 200
                }
            }
        }
    }
}
```

> Compression 控制内存使用和近似错误

TDigest算法使用许多“节点”来逼近百分位数—可用节点越多，与数据量成比例的准确性（和大内存占用）就越高。压缩参数将最大节点数限制为20*压缩。

因此，通过增加压缩值，可以以更多内存为代价提高百分位数的准确性。更大的压缩值也会使算法变慢，因为底层树数据结构的大小会增加，从而导致更昂贵的操作。默认压缩值为100。

一个“节点”大约使用32字节的内存，因此在最坏情况下（大量数据按顺序到达），默认设置将产生一个大约64KB大小的TDigest。实际上，数据往往更随机，TDigest将使用更少的内存。

HDR直方图

HDR直方图（High Dynamic Range Histogram，高动态范围直方图）是一种替代实现，在计算延迟测量的百分位数时非常有用，因为它可以比t摘要实现更快，但需要更大的内存占用。此实现保持固定的最坏情况百分比错误（指定为有效位数）。这意味着，如果在直方图中以1微秒到1小时（360000000微秒）的值记录数据，并将其设置为3个有效数字，则对于最长1毫秒的值，将保持1微秒的值分辨率，对于最大跟踪值（1小时），将保持3.6秒（或更高）的值分辨率。

通过在请求中指定方法参数，可以使用HDR直方图：

```
GET latency/_search
{
    "size": 0,
    "aggs" : {
        "load_time_outlier" : {
            "percentiles" : {
                "field" : "load_time",
                "percents" : [95, 99, 99.9],
                "hdr": {
                  "number_of_significant_value_digits" : 3
                }
            }
        }
    }
}
```

1. hdr对象表示应该使用hdr直方图来计算百分位数，并且可以在对象内指定此算法的特定设置

2. number_of_significant_value_digits指定直方图值的分辨率（有效位数）

HDR Histogram仅支持正值，如果传递负值，则会出错。如果值的范围未知，则使用HDR Histogram也不是一个好主意，因为这可能会导致高内存使用率。

Missing Value

缺少的参数定义了应如何处理缺少值的文档。默认情况下，它们将被忽略，但也可以将它们视为具有值。

```
GET latency/_search
{
    "size": 0,
    "aggs" : {
        "grade_percentiles" : {
            "percentiles" : {
                "field" : "grade",
                "missing": 10
            }
        }
    }
}
```

`grade`字段中没有值的文档将与值为10的文档属于同一个存储桶。

百分比排名聚合 percentile_ranks

Percentile Ranks Aggregation

多值度量聚合，计算从聚合文档中提取的数值的一个或多个百分位数排名。这些值可以从文档中的特定数字字段中提取，也可以由提供的脚本生成。

百分比排名显示低于某个值的观察值的百分比。例如，如果值大于或等于观察值的95%，则称其处于第95百分位。

假设您的数据包含网站加载时间。您可能有一个服务协议，即95%的页面加载在500ms内完成，99%的页面加载是在600ms内完成。

让我们看一下表示加载时间的百分比范围：

```
GET latency/_search
{
    "size": 0,
    "aggs" : {
        "load_time_ranks" : {
            "percentile_ranks" : {
                "field" : "load_time",
                "values" : [500, 600]
            }
        }
    }
}
```

返回结果

```
{
    ...

   "aggregations": {
      "load_time_ranks": {
         "values" : {
            "500.0": 90.01,
            "600.0": 100.0
         }
      }
```

```
    }
}
```

根据这些信息，您可以确定您达到了99%的load_time目标，但还没有达到95%的load time

Keyed Response

默认情况下，keyed标志设置为true将一个唯一的字符串键与每个bucket相关联，并将范围作为哈希而不是数组返回。将键控标志设置为false将禁用此行为：

```
GET latency/_search
{
    "size": 0,
    "aggs": {
        "load_time_ranks": {
            "percentile_ranks": {
                "field": "load_time",
                "values": [500, 600],
                "keyed": false
            }
        }
    }
}
```

响应

```
{
    ...

    "aggregations": {
        "load_time_ranks": {
            "values": [
                {
                    "key": 500.0,
                    "value": 90.01
                },
                {
                    "key": 600.0,
                    "value": 100.0
                }
            ]
        }
```

```
    }
}
```

Script

百分比等级度量支持脚本。例如，如果我们的加载时间以毫秒为单位，但我们希望以秒为单位指定值，那么我们可以使用脚本动态转换它们：

```
GET latency/_search
{
    "size": 0,
    "aggs" : {
        "load_time_ranks" : {
            "percentile_ranks" : {
                "values" : [500, 600],
                "script" : {
                    "lang": "painless",
                    "source": "doc['load_time'].value / params.timeUnit",
                    "params" : {
                        "timeUnit" : 1000
                    }
                }
            }
        }
    }
}
```

1. `field`参数替换为脚本参数，该参数使用脚本生成计算百分位等级的值
2. 与任何其他脚本一样，脚本支持参数化输入

这将把脚本参数解释为具有painless脚本语言且没有脚本参数的inline script。要使用存储的脚本，请使用以下语法：

```
GET latency/_search
{
    "size": 0,
    "aggs" : {
        "load_time_ranks" : {
            "percentile_ranks" : {
                "values" : [500, 600],
                "script" : {
                    "id": "my_script",
                    "params": {
```

```
                    "field": "load_time"
                }
            }
        }
    }
}
```

HDR Histogram

```
GET latency/_search
{
    "size": 0,
    "aggs" : {
        "load_time_ranks" : {
            "percentile_ranks" : {
                "field" : "load_time",
                "values" : [500, 600],
                "hdr": {
                  "number_of_significant_value_digits" : 3
                }
            }
        }
    }
}
```

脚本度量聚合 scripted_metric

Scripted Metric Aggregation

使用脚本执行以提供度量输出的度量聚合。

例子

```
POST ledger/_search?size=0
{
    "query" : {
        "match_all" : {}
    },
    "aggs": {
        "profit": {
            "scripted_metric": {
                "init_script" : "state.transactions = []",
                "map_script" : "state.transactions.add(doc.type.value == 'sale' ?
doc.amount.value : -1 * doc.amount.value)",
                "combine_script" : "double profit = 0; for (t in
state.transactions) { profit += t } return profit",
                "reduce_script" : "double profit = 0; for (a in states) { profit +=
a } return profit"
            }
        }
    }
}
```

> map_script是唯一必需的参数

上面的聚合演示了如何使用脚本聚合计算销售和成本交易的总利润。

上述聚合的响应：

```
{
    "took": 218,
    ...
    "aggregations": {
        "profit": {
            "value": 240.0
        }
```

```
    }
}
```

也可以使用存储的脚本指定上述示例，如下所示：

```
POST ledger/_search?size=0
{
    "aggs": {
        "profit": {
            "scripted_metric": {
                "init_script" : {
                    "id": "my_init_script"
                },
                "map_script" : {
                    "id": "my_map_script"
                },
                "combine_script" : {
                    "id": "my_combine_script"
                },
                "params": {
                    "field": "amount"
                },
                "reduce_script" : {
                    "id": "my_reduce_script"
                }
            }
        }
    }
}
```

> init、map和combine脚本的脚本参数必须在全局params对象中指定，以便在脚本之间共享。

允许的返回类型

虽然任何有效的脚本对象都可以在单个脚本中使用，但脚本必须仅返回以下类型或将其存储在状态对象中：

- primitive types
- String
- Map (仅包含此处列出的类型的键和值)
- Array (仅包含此处列出的类型的元素)

脚本的Scope

scripted metric aggregation在其执行的4个阶段使用脚本：

- init_script 在收集文件之前执行。允许聚合设置任何初始状态。
- map_script 每个收集的文档执行一次。这是唯一需要的脚本。如果未指定combine_script，则需要将生成的状态存储在状态对象中。
- combine_script 文档收集完成后，在每个shard上执行一次。允许聚合合并从每个shard返回的状态。如果未提供combine_script，则combine阶段将返回聚合变量。
- reduce_script 在所有shard返回结果后，在协调节点上执行一次。脚本提供了对变量状态的访问，变量状态是每个shard上combine_script结果的数组。如果未提供reduce_script，reduce阶段将返回状态变量。

统计聚合 stats

Stats Aggregation

多值度量聚合，用于计算从聚合文档中提取的数值的统计信息。这些值可以从文档中的特定数字字段中提取，也可以由提供的脚本生成。

返回的统计数据包括：min、max、sum、count和avg。

假设数据由代表学生考试成绩（0至100）的文件组成

```
POST /exams/_search?size=0
{
    "aggs" : {
        "grades_stats" : { "stats" : { "field" : "grade" } }
    }
}
```

上述汇总计算所有文档的成绩统计。聚合类型为stats，字段设置定义了要计算统计信息的文档的数字字段。上面将返回以下内容：

```
{
    ...

    "aggregations": {
        "grades_stats": {
            "count": 2,
            "min": 50.0,
            "max": 100.0,
            "avg": 75.0,
            "sum": 150.0
        }
    }
}
```

聚合的名称（grades_stats）也用作键，通过该键可以从返回的响应中检索聚合结果。

Script

```
POST /exams/_search?size=0
{
    "aggs" : {
        "grades_stats" : {
            "stats" : {
                "script" : {
                    "lang": "painless",
                    "source": "doc['grade'].value"
                }
            }
         }
    }
}
```

参数方式

```
POST /exams/_search?size=0
{
    "aggs" : {
        "grades_stats" : {
            "stats" : {
                "script" : {
                    "id": "my_script",
                    "params" : {
                        "field" : "grade"
                    }
                }
            }
        }
    }
}
```

Value Script

```
POST /exams/_search?size=0
{
    "aggs" : {
        "grades_stats" : {
            "stats" : {
                "field" : "grade",
                "script" : {
                    "lang": "painless",
                    "source": "_value * params.correction",
```

```
                    "params" : {
                        "correction" : 1.2
                    }
                }
            }
        }
    }
}
```

Missing value

```
POST /exams/_search?size=0
{
    "aggs" : {
        "grades_stats" : {
            "stats" : {
                "field" : "grade",
                "missing": 0
            }
        }
    }
}
```

`grade`字段中没有值的文档将与值为0的文档属于同一个存储桶。

扩展统计聚合 extended_stats

Extended Stats Aggregation

多值度量聚合，用于计算从聚合文档中提取的数值的统计信息。这些值可以从文档中的特定数字字段中提取，也可以由提供的脚本生成。

extended_stats聚合是stats聚合的扩展版本，其中添加了其他度量，如sum_of_squares、variance、std_deviation和std_deviation_bounds。

假设数据由代表学生考试成绩（0至100）的文件组成

```
GET /exams/_search
{
    "size": 0,
    "aggs" : {
        "grades_stats" : { "extended_stats" : { "field" : "grade" } }
    }
}
```

上述汇总计算所有文档的成绩统计。聚合类型为extended_stats，字段设置定义了要计算统计信息的文档的数字字段。上面将返回以下内容：

```
{
    ...

    "aggregations": {
        "grades_stats": {
           "count": 2,
           "min": 50.0,
           "max": 100.0,
           "avg": 75.0,
           "sum": 150.0,
           "sum_of_squares": 12500.0,
           "variance": 625.0,
           "std_deviation": 25.0,
           "std_deviation_bounds": {
            "upper": 125.0,
            "lower": 25.0
           }
```

```
        }
    }
}
```

聚合的名称（上面的grades_stats）也用作键，通过该键可以从返回的响应中检索聚合结果。

标准偏差界限

默认情况下，extended_stats度量将返回一个名为std_deviation_bounds的对象，该对象提供与平均值正负两个标准偏差的间隔。这是可视化数据差异的有用方法。如果您需要不同的边界，例如三个标准偏差，可以在请求中设置sigma：

```
GET /exams/_search
{
    "size": 0,
    "aggs" : {
        "grades_stats" : {
            "extended_stats" : {
                "field" : "grade",
                "sigma" : 3
            }
        }
    }
}
```

> sigma控制应显示多少与平均值+/-的标准偏差

sigma可以是任何非负双精度，这意味着您可以请求非整数值，例如1.5。值0是有效的，但只会返回上限和下限的平均值。

> **INFO**
>
> 标准偏差和界限要求正态性
> 默认情况下显示标准偏差及其边界，但它们并不总是适用于所有数据集。您的数据必须是正常分布的，才能使度量有意义。标准偏差背后的统计数据假设数据为正态分布，因此如果数据严重向左或向右倾斜，则返回的值将具有误导性

Script

根据脚本计算成绩统计：

```
GET /exams/_search
{
    "size": 0,
    "aggs" : {
        "grades_stats" : {
            "extended_stats" : {
                "script" : {
                    "source" : "doc['grade'].value",
                    "lang" : "painless"
                }
            }
        }
    }
}
```

这将把脚本参数解释为具有painless脚本且没有脚本参数的inline script。要使用存储的脚本，请使用以下语法：

```
GET /exams/_search
{
    "size": 0,
    "aggs" : {
        "grades_stats" : {
            "extended_stats" : {
                "script" : {
                    "id": "my_script",
                    "params": {
                        "field": "grade"
                    }
                }
            }
        }
    }
}
```

Value Script

结果发现，这场考试远远超出了学生的水平，需要进行成绩修正。我们可以使用value脚本获取新的统计数据：

```
GET /exams/_search
{
    "size": 0,
    "aggs" : {
        "grades_stats" : {
            "extended_stats" : {
                "field" : "grade",
                "script" : {
                    "lang" : "painless",
                    "source": "_value * params.correction",
                    "params" : {
                        "correction" : 1.2
                    }
                }
            }
        }
    }
}
```

Missing Value

缺少的参数定义了应如何处理缺少值的文档。默认情况下，它们将被忽略，但也可以将它们视为具有值。

```
GET /exams/_search
{
    "size": 0,
    "aggs" : {
        "grades_stats" : {
            "extended_stats" : {
                "field" : "grade",
                "missing": 0
            }
        }
    }
}
```

grade字段中没有值的文档将与值为0的文档属于同一个存储桶。

排行榜聚合 top_hits

Top Hits Aggregation

top_hits度量聚合器跟踪正在聚合的最相关文档。该聚合器旨在用作子聚合器，以便每个bucket可以聚合最匹配的文档。

top_hits聚合器可以有效地用于通过bucket聚合器按特定字段对结果集进行分组。一个或多个桶聚合器确定结果集被划分为哪些属性。

Example

在下面的示例中，我们按类型对销售进行分组，并按类型显示上一次销售。对于每次销售，源中只包含日期和价格字段。

```
POST /sales/_search?size=0
{
    "aggs": {
        "top_tags": {
            "terms": {
                "field": "type",
                "size": 3
            },
            "aggs": {
                "top_sales_hits": {
                    "top_hits": {
                        "sort": [
                            {
                                "date": {
                                    "order": "desc"
                                }
                            }
                        ],
                        "_source": {
                            "includes": [ "date", "price" ]
                        },
                        "size" : 1
                    }
                }
            }
```

```
        }
    }
}
```

返回

```
{
  ...
  "aggregations": {
    "top_tags": {
       "doc_count_error_upper_bound": 0,
       "sum_other_doc_count": 0,
       "buckets": [
          {
             "key": "hat",
             "doc_count": 3,
             "top_sales_hits": {
                "hits": {
                   "total": 3,
                   "max_score": null,
                   "hits": [
                      {
                         "_index": "sales",
                         "_type": "_doc",
                         "_id": "AVnNBmauCQpcRyxw6ChK",
                         "_source": {
                            "date": "2015/03/01 00:00:00",
                            "price": 200
                         },
                         "sort": [
                            1425168000000
                         ],
                         "_score": null
                      }
                   ]
                }
             }
          },
          {
             "key": "t-shirt",
             "doc_count": 3,
             "top_sales_hits": {
                "hits": {
                   "total": 3,
                   "max_score": null,
                   "hits": [
                      {
```

```
                        "_index": "sales",
                        "_type": "_doc",
                        "_id": "AVnNBmauCQpcRyxw6ChL",
                        "_source": {
                           "date": "2015/03/01 00:00:00",
                           "price": 175
                        },
                        "sort": [
                           1425168000000
                        ],
                        "_score": null
                     }
                  ]
               }
            }
         },
         {
            "key": "bag",
            "doc_count": 1,
            "top_sales_hits": {
               "hits": {
                  "total": 1,
                  "max_score": null,
                  "hits": [
                     {
                        "_index": "sales",
                        "_type": "_doc",
                        "_id": "AVnNBmatCQpcRyxw6ChH",
                        "_source": {
                           "date": "2015/01/01 00:00:00",
                           "price": 150
                        },
                        "sort": [
                           1420070400000
                        ],
                        "_score": null
                     }
                  ]
               }
            }
         }
      ]
   }
}
```


字段折叠示例 (Field collapse)

字段折叠或结果分组是一种功能，它在逻辑上将结果集分组，并按组返回顶部文档。组的排序由组中第一个文档的相关性确定。在Elasticsearch中，这可以通过将top_hits聚合器包装为子聚合器的bucket聚合器来实现。

在下面的示例中，我们搜索已爬网的网页。对于每个网页，我们存储网页所属的主体和域。通过在域字段上定义术语聚合器，我们可以按域对网页的结果集进行分组。然后将top_hits聚合器定义为子聚合器，以便每个bucket收集最匹配的命中数。

此外，还定义了一个最大聚合器，术语聚合器的顺序功能使用该最大聚合器按桶中最相关文档的相关性顺序返回桶。

```
POST /sales/_search
{
  "query": {
    "match": {
      "body": "elections"
    }
  },
  "aggs": {
    "top_sites": {
      "terms": {
        "field": "domain",
        "order": {
          "top_hit": "desc"
        }
      },
      "aggs": {
        "top_tags_hits": {
          "top_hits": {}
        },
        "top_hit" : {
          "max": {
            "script": {
              "source": "_score"
            }
          }
        }
      }
    }
  }
}
```

目前，需要最大（或最小）聚合器来确保术语聚合器中的桶是根据每个域最相关网页的得分排序的。不幸的是，top_hits聚合器还不能在术语聚合器的order选项中使用。

嵌套或反向嵌套聚合器中的top_hits支持

如果top_hits聚合器包装在嵌套或反向嵌套聚合器中，则返回嵌套命中。嵌套命中在某种意义上是隐藏的迷你文档，它们是常规文档的一部分，在映射中已配置了嵌套字段类型。如果top_hits聚合器被封装在嵌套或反向嵌套聚合器中，则它可以取消隐藏这些文档。阅读有关嵌套类型映射中嵌套的更多信息。

如果已经配置了嵌套类型，则单个文档实际上被索引为多个Lucene文档，并且它们共享相同的id。为了确定嵌套命中的标识，需要的不仅仅是id，因此嵌套命中也包括它们的嵌套标识。嵌套标识保存在搜索命中中的_nested字段下，并包括嵌套命中所属的数组字段中的数组字段和偏移量。偏移量从零开始。

让我们看看它是如何使用真实样本的。考虑以下映射

```
PUT /sales
{
    "mappings": {
        "_doc" : {
            "properties" : {
                "tags" : { "type" : "keyword" },
                "comments" : {
                    "type" : "nested",
                    "properties" : {
                        "username" : { "type" : "keyword" },
                        "comment" : { "type" : "text" }
                    }
                }
            }
        }
    }
}
```

comments 是一个数组，其中包含`product`对象下的嵌套文档。

添加一些文档

```
PUT /sales/_doc/1?refresh
{
    "tags": ["car", "auto"],
    "comments": [
        {"username": "baddriver007", "comment": "This car could have better
brakes"},
        {"username": "dr_who", "comment": "Where's the autopilot? Can't find it"},
        {"username": "ilovemotorbikes", "comment": "This car has two extra wheels"}
```

```
    ]
}
```

现在可以执行以下 top_hits 聚合（包含在嵌套聚合中）：

```
POST /sales/_search
{
    "query": {
        "term": { "tags": "car" }
    },
    "aggs": {
        "by_sale": {
            "nested" : {
                "path" : "comments"
            },
            "aggs": {
                "by_user": {
                    "terms": {
                        "field": "comments.username",
                        "size": 1
                    },
                    "aggs": {
                        "by_nested": {
                            "top_hits":{}
                        }
                    }
                }
            }
        }
    }
}
```

具有嵌套命中的热门命中响应片段，位于数组字段注释的第一个槽中：

```
{
  ...
  "aggregations": {
    "by_sale": {
      "by_user": {
        "buckets": [
          {
            "key": "baddriver007",
            "doc_count": 1,
            "by_nested": {
              "hits": {
```

```
                "total": 1,
                "max_score": 0.2876821,
                "hits": [
                  {
                    "_index": "sales",
                    "_type" : "_doc",
                    "_id": "1",
                    "_nested": {
                      "field": "comments",
                      "offset": 0
                    },
                    "_score": 0.2876821,
                    "_source": {
                      "comment": "This car could have better brakes",
                      "username": "baddriver007"
                    }
                  }
                ]
              }
            }
          }
          ...
        ]
      }
    }
  }
}
```

值计数聚合 value_count

Value Count Aggregation

单个值度量聚合，统计从聚合文档中提取的值的数量。这些值可以从文档中的特定字段中提取，也可以由提供的脚本生成。通常，此聚合器将与其他单值聚合一起使用。例如，当计算平均值时，人们可能会对计算平均值的数量感兴趣。

```
POST /sales/_search?size=0
{
    "aggs" : {
        "types_count" : { "value_count" : { "field" : "type" } }
    }
}
```

Script

```
POST /sales/_search?size=0
{
    "aggs" : {
        "type_count" : {
            "value_count" : {
                "script" : {
                    "source" : "doc['type'].value"
                }
            }
        }
    }
}
```

这将把脚本参数解释为具有painless script语言且没有脚本参数的inline script。要使用存储的脚本，请使用以下语法：

```
POST /sales/_search?size=0
{
    "aggs" : {
        "types_count" : {
```

```
            "value_count" : {
                "script" : {
                    "id": "my_script",
                    "params" : {
                        "field" : "type"
                    }
                }
            }
        }
    }
}
```

中值绝对偏差聚合 median_absolute_deviation

Median Absolute Deviation Aggregation

此单值聚合近似于其搜索结果的中值绝对偏差。

中值绝对偏差是变异性的度量。这是一个稳健的统计数据，这意味着它对于描述可能存在异常值或可能不正常分布的数据非常有用。对于此类数据，它可能比标准偏差更具描述性。

它被计算为每个数据点偏离整个样本中值的中值。也就是说，对于随机变量X，中值绝对偏差是中值（|median（X）-Xi|）。

实例

假设我们的数据代表一到五星级的产品评价。这种评论通常被概括为一种均值，这很容易理解，但并不能描述评论的可变性。估计中值绝对偏差可以深入了解评论之间的差异。

在这个例子中，我们有一个平均评级为3星的产品。让我们看看其评级的绝对偏差中值，以确定它们的变化程度

```
GET reviews/_search
{
  "size": 0,
  "aggs": {
    "review_average": {
      "avg": {
        "field": "rating"
      }
    },
    "review_variability": {
      "median_absolute_deviation": {
        "field": "rating"
      }
    }
  }
}
```

由此得出的绝对偏差中值2告诉我们，评级存在相当大的可变性。审查人员必须对此产品有不同的意见。

```
{
  ...
  "aggregations": {
    "review_average": {
      "value": 3.0
    },
    "review_variability": {
      "value": 2.0
    }
  }
}
```

近似值 Approximation

计算中值绝对偏差的简单实现将整个样本存储在内存中，因此该聚合计算的是近似值。它使用TDigest数据结构来近似样本中值和与样本中值的偏差中值。有关TDigests的近似特性的更多信息，请参阅百分比（通常）近似。

资源使用与TDigest分位数近似精度之间的权衡，以及因此该聚合的中值绝对偏差近似精度，由压缩参数控制。较高的压缩设置以较高的内存使用率为代价提供更精确的近似。

```
GET reviews/_search
{
  "size": 0,
  "aggs": {
    "review_variability": {
      "median_absolute_deviation": {
        "field": "rating",
        "compression": 100
      }
    }
  }
}
```

此聚合的默认压缩值为1000。在此压缩级别下，此聚合通常在准确结果的5%以内，但观察到的性能将取决于样本数据。

Script

此度量聚合支持脚本。在我们上面的例子中，产品评论的比例是一到五。如果我们想将它们修改为1到10的比例，我们可以使用脚本。

提供inline script，请执行以下操作：

```
GET reviews/_search
{
  "size": 0,
  "aggs": {
    "review_variability": {
      "median_absolute_deviation": {
        "script": {
          "lang": "painless",
          "source": "doc['rating'].value * params.scaleFactor",
          "params": {
            "scaleFactor": 2
          }
        }
      }
    }
  }
}
```

要提供存储的脚本，请执行以下操作：

```
GET reviews/_search
{
  "size": 0,
  "aggs": {
    "review_variability": {
      "median_absolute_deviation": {
        "script": {
          "id": "my_script",
          "params": {
            "field": "rating"
          }
        }
      }
    }
  }
}
```

Missing value

缺少的参数定义了应如何处理缺少值的文档。默认情况下，它们将被忽略，但也可以将它们视为具有值。

让我们乐观一点，假设一些评论家非常喜欢这款产品，以至于忘记给它打分。我们会给他们五星

```
GET reviews/_search
{
  "size": 0,
  "aggs": {
    "review_variability": {
      "median_absolute_deviation": {
        "field": "rating",
        "missing": 5
      }
    }
  }
}
```

关键词聚合 Terms

Terms Aggregation

基于多桶聚合，按字段值分桶聚合。比如性别有男、女，就会创建两个桶，分别存放男女的信息。默认会搜集doc_count的信息，即记录有多少男生，有多少女生，然后返回给客户端，这样就完成了一个terms统计。

> **INFO**
>
> terms agg 通常用于keyword字段，如果想运用在text字段，您需要启用fielddata 。

一个简单的示例

```
GET /_search
{
    "aggs" : {
        "genres" : {
            "terms" : { "field" : "genre" }
        }
    }
}
```

返回

```
{
    ...
    "aggregations" : {
        "genres" : {
            "doc_count_error_upper_bound": 0,
            "sum_other_doc_count": 0,
            "buckets" : [
                {
                    "key" : "electronic",
                    "doc_count" : 6
                },
                {
                    "key" : "rock",
                    "doc_count" : 3
```

```
                },
                {
                    "key" : "jazz",
                    "doc_count" : 2
                }
            ]
        }
    }
}
```

Size

默认情况下，terms agg将返回doc_count排序的前十个term的桶。可以通过设置size参数来更改此默认行为。

Shard Size

请求的`size`越大，结果就越准确，但计算最终结果的成本也越高（这两方面都是由于在shard级别上管理的优先级队列越大，以及节点和客户端之间的数据传输越大）。

可以通过shard_size参数设置协调节点向各个分片请求的term个数，然后在协调节点进行聚合，最后只返回size个term给到客户端，shard_size >= size，如果shard_size设置小于size，ES会自动将其设置为size，默认情况下shard_size建议设置为(1.5 * size + 10)。

Order

通过设置order参数，可以自定义桶的顺序。默认情况下，存储桶按其doc_count降序排列。

- 按桶的doc_count升序排序：

```
GET /_search
{
    "aggs" : {
        "genres" : {
            "terms" : {
                "field" : "genre",
                "order" : { "_count" : "asc" }
            }
        }
```

```
    }
}
```

- 按term的字母顺序升序排列存储桶：

```
GET /_search
{
    "aggs" : {
        "genres" : {
            "terms" : {
                "field" : "genre",
                "order" : { "_key" : "asc" }
            }
        }
    }
}
```

- 按单值度量子聚合对桶进行排序：

```
GET /_search
{
    "aggs" : {
        "genres" : {
            "terms" : {
                "field" : "genre",
                "order" : { "max_play_count" : "desc" }
            },
            "aggs" : {
                "max_play_count" : { "max" : { "field" : "play_count" } }
            }
        }
    }
}
```

- 按多值度量子聚合对桶进行排序：

```
GET /_search
{
    "aggs" : {
        "genres" : {
            "terms" : {
                "field" : "genre",
                "order" : { "playback_stats.max" : "desc" }
```

```
            },
            "aggs" : {
                "playback_stats" : { "stats" : { "field" : "play_count" } }
            }
        }
    }
}
```

- 根据层次结构中的“deeper”聚合对桶进行排序。

path必须按以下格式定义

```
AGG_SEPARATOR       =  '>' ;
METRIC_SEPARATOR    =  '.' ;
AGG_NAME            =  <the name of the aggregation> ;
METRIC              =  <the name of the metric (in case of multi-value metrics
aggregation)> ;
PATH                =  <AGG_NAME> [ <AGG_SEPARATOR>, <AGG_NAME> ]* [
<METRIC_SEPARATOR>, <METRIC> ] ;
```

```
GET /_search
{
    "aggs" : {
        "countries" : {
            "terms" : {
                "field" : "artist.country",
                "order" : { "rock>playback_stats.avg" : "desc" }
            },
            "aggs" : {
                "rock" : {
                    "filter" : { "term" : { "genre" :  "rock" }},
                    "aggs" : {
                        "playback_stats" : { "stats" : { "field" : "play_count" }}
                    }
                }
            }
        }
    }
}
```

以上将根据摇滚歌曲中的平均播放次数对艺术家的国家进行排序。

通过提供一系列订购标准，可以使用多个标准来订购存储桶，如下所示：

```
GET /_search
{
    "aggs" : {
        "countries" : {
            "terms" : {
                "field" : "artist.country",
                "order" : [ { "rock>playback_stats.avg" : "desc" }, { "_count" :
"desc" } ]
            },
            "aggs" : {
                "rock" : {
                    "filter" : { "term" : { "genre" : "rock" }},
                    "aggs" : {
                        "playback_stats" : { "stats" : { "field" : "play_count" }}
                    }
                }
            }
        }
    }
}
```

以上将根据摇滚歌曲中的平均播放次数，然后按其doc_count降序对艺术家的国家/地区进行排序。

Minimum document count

可以使用min_doc_count选项仅返回与配置的命中数匹配的项：

```
GET /_search
{
    "aggs" : {
        "tags" : {
            "terms" : {
                "field" : "tags",
                "min_doc_count": 10
            }
        }
    }
}
```

上述聚合将仅返回在10次或更多次命中中找到的标记。默认值为1。

Script

```
GET /_search
{
    "aggs" : {
        "genres" : {
            "terms" : {
                "script" : {
                    "source": "doc['genre'].value",
                    "lang": "painless"
                }
            }
        }
    }
}
```

这将把脚本参数解释为具有默认脚本语言且没有脚本参数的 inline script。要使用存储的脚本，请使用以下语法：

```
GET /_search
{
    "aggs" : {
        "genres" : {
            "terms" : {
                "script" : {
                    "id": "my_script",
                    "params": {
                        "field": "genre"
                    }
                }
            }
        }
    }
}
```

Value Script

```
GET /_search
{
    "aggs" : {
        "genres" : {
            "terms" : {
                "field" : "genre",
                "script" : {
                    "source" : "'Genre: ' +_value",
```

```
                    "lang" : "painless"
                }
            }
        }
    }
}
```

Filtering Values

可以过滤将为其创建桶的值。这可以使用基于正则表达式字符串或精确值数组的include和exclude参数来完成。此外，include子句可以使用分区表达式进行筛选。

Filtering Values 使用正则表达式

```
GET /_search
{
    "aggs" : {
        "tags" : {
            "terms" : {
                "field" : "tags",
                "include" : ".*sport.*",
                "exclude" : "water_.*"
            }
        }
    }
}
```

在上面的示例中，将为所有包含单词sport的标记创建bucket，但以water_开头的标记除外（因此不会聚合标记watersports）。include正则表达式将确定“允许”聚合的值，而exclude则确定不应聚合的值。当两者都被定义时，exclude具有优先权，这意味着首先计算include，然后才计算exclude。

Filtering Values with exact values

对于基于精确值的匹配，include和exclude参数可以简单地采用一个字符串数组来表示索引中的term：

```
GET /_search
{
    "aggs" : {
        "JapaneseCars" : {
             "terms" : {
                 "field" : "make",
```

```
                "include" : ["mazda", "honda"]
            }
        },
        "ActiveCarManufacturers" : {
            "terms" : {
                "field" : "make",
                "exclude" : ["rover", "jensen"]
            }
        }
    }
}
```

Filtering Values with partitions

有时，在单个请求/响应对中处理的唯一term太多，因此将分析分解为多个请求可能很有用。这可以通过在查询时将字段的值分组为多个分区并在每个请求中只处理一个分区来实现。考虑此请求，该请求正在查找最近未记录任何访问权限的帐户

```
GET /_search
{
   "size": 0,
   "aggs": {
      "expired_sessions": {
         "terms": {
            "field": "account_id",
            "include": {
               "partition": 0,
               "num_partitions": 20
            },
            "size": 10000,
            "order": {
               "last_access": "asc"
            }
         },
         "aggs": {
            "last_access": {
               "max": {
                  "field": "access_date"
               }
            }
         }
      }
   }
}
```

此请求正在查找客户帐户子集的上次登录访问日期，因为我们可能希望终止一些很久没有看到的客户帐户。num_partitions设置要求将唯一的account_id平均组织为二十个分区（0到19）。并且此请求中的分区设置过滤为仅考虑落入分区0的account_id。后续请求应请求分区1，然后请求分区2等以完成过期帐户分析。

注意，返回结果数的大小设置需要使用num_partitions进行调整。对于这个特定的帐户过期示例，平衡size和num_partitions值的过程如下：

- 使用基数聚合来估计唯一account_id值的总数
- 为num_partitions选择一个值，将数字从1）拆分为更易于管理的块
- 为每个分区的响应数量选择一个大小值
- 运行测试请求

如果出现断路器错误，我们试图在一个请求中做太多，必须增加num_partitions。如果请求成功，但按日期排序的测试响应中的最后一个帐户ID仍然是我们可能希望过期的帐户，那么我们可能缺少感兴趣的帐户，并且将我们的数字设置得太低。我们必须

- 增加size参数以每个分区返回更多结果（可能会占用大量内存）或
- 增加num_partitions以减少每个请求的帐户数（可能会增加总体处理时间，因为我们需要发出更多请求）

最终，这是在管理处理单个请求所需的Elasticsearch资源和客户端应用程序完成任务所必须发出的请求量之间的平衡。

多字段 terms aggregation

term aggregation不支持从同一文档中的多个字段收集term。原因是term agg本身不收集字符串term值，而是使用全局序数`global ordinals`生成字段中所有唯一值的列表。全局序数会导致重要的性能提升，这在多个字段中是不可能的。

有两种方法可用于跨多个字段执行term agg：

- **Script** 使用脚本从多个字段检索term。这将禁用全局序数优化，并将比从单个字段中收集term慢，但它为您提供了在搜索时实现此选项的灵活性
- **copy_to field** 如果提前知道要从两个或多个字段中收集term，那么在映射中使用copy_to在索引时创建一个新的专用字段，其中包含两个字段的值。您可以在这个字段上进行聚合，这将受益于全局序数优化。

收集模式

推迟子聚合的计算

对于具有许多唯一项和少量所需结果的字段，将子聚合的计算延迟到顶级父级标记被删除之前可能会更有效。通常，聚合树的所有分支在一次深度优先过程中展开，然后才进行任何修剪。在某些情况下，这可能非常浪费，并会影响内存限制。一个示例问题场景是查询电影数据库，查找10位最受欢迎的演员及其5位最常见的合作主演：

```
GET /_search
{
    "aggs" : {
        "actors" : {
             "terms" : {
                 "field" : "actors",
                 "size" : 10
             },
            "aggs" : {
               "costars" : {
                    "terms" : {
                        "field" : "actors",
                        "size" : 5
                    }
                }
            }
         }
    }
}
```

尽管参与者的数量可能相对较少，我们只需要50个结果桶，但在计算过程中，桶的组合爆炸-单个参与者可以产生n^2个桶，其中n是参与者的数量。明智的选择是首先确定10位最受欢迎的演员，然后再检查这10位演员的最佳合作演员。这种替代策略是我们所称的breadth_first收集模式，而不是depth_firs模式。

```
GET /_search
{
    "aggs" : {
        "actors" : {
             "terms" : {
                 "field" : "actors",
                 "size" : 10,
                 "collect_mode" : "breadth_first"
             },
            "aggs" : {
               "costars" : {
```

```
                    "terms" : {
                        "field" : "actors",
                        "size" : 5
                    }
                }
            }
        }
    }
}
```

当使用breadth_first模式时，属于最高存储桶的一组文档将被缓存以供后续回放，因此这样做会产生内存开销，这与匹配文档的数量成线性关系。请注意，在使用breadth_first设置时，order参数仍然可以用于引用子聚合中的数据-父聚合知道需要在调用任何其他子聚合之前先调用此子聚合。

Execution hint

执行term聚合有不同的机制：

- 通过直接使用字段值来聚合每个存储桶（map）的数据
- 通过使用字段的全局序数并为每个全局序数分配一个桶（global_ordinals）

Elasticsearch尝试使用合理的默认值，因此这通常不需要配置。

globalordinals是关键字字段的默认选项，它使用全局ordinals动态分配桶，因此内存使用量与聚合范围内的文档值的数量成线性关系。

只有当很少文档与查询匹配时，才应考虑map。否则，基于序数的执行模式明显更快。默认情况下，map仅在对脚本运行聚合时使用，因为它们没有序数。

```
GET /_search
{
    "aggs" : {
        "tags" : {
            "terms" : {
                "field" : "tags",
                "execution_hint": "map"
            }
        }
    }
}
```

直方图 histogram

Histogram Aggregation

直方图聚合是一个用于评估数值型或是数值范围型价值的文档的多桶（multi-bucket）聚合，它可以对参与聚合的值动态生成固定尺寸的桶。

比如，如果一些文档具有数值型字段“price”，我们可以配置聚合间隔为5（在价钱中可能为5元）来动态生成直方图统计。当聚合执行的时候，每个文档的price字段会参与估算，并且为四舍五入到最近的桶中。比如，如果一个文档的price字段值为32，桶的尺寸为5，并且字段值四舍五入后的值为30，那么这个文档就会归入跟30这个关键字关联的桶内。下面的算式可以精确的确定每个文档的归属桶（根据桶的关键字确定）。

```
bucket_key = Math.floor((value - offset) / interval) * interval + offset
```

> 注意：interval必须是是一个十进制的正数，同时offset必须是[0,interval)（一个大于等于0、小于interval的十进制数）之间的一个十进制的数。

以下代码片段根据产品的价格以50的间隔对产品进行“分类”：

```
POST /sales/_search?size=0
{
    "aggs" : {
        "prices" : {
            "histogram" : {
                "field" : "price",
                "interval" : 50
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
```

```
        "prices" : {
            "buckets": [
                {
                    "key": 0.0,
                    "doc_count": 1
                },
                {
                    "key": 50.0,
                    "doc_count": 1
                },
                {
                    "key": 100.0,
                    "doc_count": 0
                },
                {
                    "key": 150.0,
                    "doc_count": 2
                },
                {
                    "key": 200.0,
                    "doc_count": 3
                }
            ]
        }
    }
}
```

最小文档数

上面的响应表明，没有文档的价格在[100，150）的范围内。默认情况下，响应将用空桶填充直方图中的空白。由于min_doc_count设置，可以更改此设置并请求具有更高最小计数的桶：

```
POST /sales/_search?size=0
{
    "aggs" : {
        "prices" : {
            "histogram" : {
                "field" : "price",
                "interval" : 50,
                "min_doc_count" : 1
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "prices" : {
            "buckets": [
                {
                    "key": 0.0,
                    "doc_count": 1
                },
                {
                    "key": 50.0,
                    "doc_count": 1
                },
                {
                    "key": 150.0,
                    "doc_count": 2
                },
                {
                    "key": 200.0,
                    "doc_count": 3
                }
            ]
        }
    }
}
```

默认情况下，直方图返回数据本身范围内的所有存储桶，即具有最小值的文档（使用直方图）将确定最小存储桶（具有最小关键字的存储桶），而具有最大值的文档将确定最大存储桶（具有最高关键字的存储）。通常，当请求空桶时，这会导致混乱，特别是当数据也被过滤时。

要了解原因，我们来看一个示例：

假设您正在过滤请求以获取值介于0和500之间的所有文档，此外，您还希望使用间隔为50的直方图对每个价格的数据进行切片。您还指定“min_doc_count”：0，因为您希望获取所有桶，即使是空桶。如果所有产品（文档）的价格都高于100，那么您将得到的第一个桶将是以100为关键字的桶。这很令人困惑，因为很多时候，你也希望这些桶在0到100之间。

通过extended_bounds设置，您现在可以“强制”直方图聚合以特定的最小值开始构建桶，并继续构建最大值的桶（即使不再有文档）。只有当min_doc_count为0时，使用extended_bounds才有意义（如果min_doc-count大于0，则不会返回空桶）。

注意 extended_bounds不是过滤桶。意思是如果扩展边界，min大于从文档中提取的值，文档仍将决定第一个bucket是什么（extendedbounds.max和最后一个bucket也是如此）。对于过滤桶，应该使用适当的从/到设置将直方图聚合嵌套在范围过滤器聚合下。

例子：

```
POST /sales/_search?size=0
{
    "query" : {
        "constant_score" : { "filter": { "range" : { "price" : { "to" : "500" } } } }
}
    },
    "aggs" : {
        "prices" : {
            "histogram" : {
                "field" : "price",
                "interval" : 50,
                "extended_bounds" : {
                    "min" : 0,
                    "max" : 500
                }
            }
        }
    }
}
```

Order

默认情况下，返回的桶按关键字升序排序，但可以使用`order`控制顺序。

Offset

默认情况下，存储桶键以0开始，然后以偶数间隔步长继续，例如，如果间隔为10，则前三个存储桶（假设其中有数据）将为[0，10），[10，20），[20，30）。可以使用`offset`移动存储桶边界。

这可以用一个例子来最好地说明。如果有10个文档的值在5到14之间，则使用间隔10将生成两个桶，每个桶包含5个文档。如果使用额外的偏移量5，则只有一个包含所有10个文档的存储桶[5，15）。

Response Format

默认情况下，存储桶作为有序数组返回。也可以将响应请求为散列，而不是由桶键键入：

```
POST /sales/_search?size=0
{
    "aggs" : {
        "prices" : {
            "histogram" : {
                "field" : "price",
                "interval" : 50,
                "keyed" : true
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "prices": {
            "buckets": {
                "0.0": {
                    "key": 0.0,
                    "doc_count": 1
                },
                "50.0": {
                    "key": 50.0,
                    "doc_count": 1
                },
                "100.0": {
                    "key": 100.0,
                    "doc_count": 0
                },
                "150.0": {
                    "key": 150.0,
                    "doc_count": 2
                },
                "200.0": {
                    "key": 200.0,
                    "doc_count": 3
                }
            }
        }
    }
}
```

Missing value

```
POST /sales/_search?size=0
{
    "aggs" : {
        "quantity" : {
            "histogram" : {
                "field" : "quantity",
                "interval": 10,
                "missing": 0
            }
         }
    }
}
```

`quantity`字段中没有值的文档将与值为0的文档属于同一个存储桶。

日期直方图 date_histogram

Date Histogram Aggregation

这种多桶聚合类似于 Histogram Aggregation，但只能与日期值一起使用。这两个API的主要区别在于，这里可以使用日期/时间表达式指定间隔。

示例

请求一个月的时段间隔。

```
POST /sales/_search?size=0
{
    "aggs" : {
        "sales_over_time" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            }
        }
    }
}
```

您还可以使用时间单位分析支持的缩写来指定时间值。请注意，不支持分数时间值，但您可以通过转换到另一个时间单位（例如，可以将1.5h指定为90m）来解决此问题。

```
POST /sales/_search?size=0
{
    "aggs" : {
        "sales_over_time" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "90m"
            }
        }
    }
}
```

支持表达型日期格式模式

```
POST /sales/_search?size=0
{
    "aggs" : {
        "sales_over_time" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "1M",
                "format" : "yyyy-MM-dd"
            }
        }
    }
}
```

Timezone

```
GET my_index/_search?size=0
{
  "aggs": {
    "by_day": {
      "date_histogram": {
        "field":     "date",
        "interval":  "day",
        "time_zone": "-01:00"
      }
    }
  }
}
```

Offset

使用`offset`参数按指定的正（+）或负偏移（-）持续时间更改每个存储桶的起始值

```
PUT my_index/_doc/1?refresh
{
  "date": "2015-10-01T05:30:00Z"
}

PUT my_index/_doc/2?refresh
{
```

```
  "date": "2015-10-01T06:30:00Z"
}

GET my_index/_search?size=0
{
  "aggs": {
    "by_day": {
      "date_histogram": {
        "field":     "date",
        "interval":  "day",
        "offset":    "+6h"
      }
    }
  }
}
```

Keyed Response

将keyed标志设置为true会将一个唯一的字符串键与每个bucket相关联，并以哈希而非数组的形式返回范围：

```
POST /sales/_search?size=0
{
    "aggs" : {
        "sales_over_time" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "1M",
                "format" : "yyyy-MM-dd",
                "keyed": true
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "sales_over_time": {
            "buckets": {
                "2015-01-01": {
                    "key_as_string": "2015-01-01",
```

```
                    "key": 1420070400000,
                    "doc_count": 3
                },
                "2015-02-01": {
                    "key_as_string": "2015-02-01",
                    "key": 1422748800000,
                    "doc_count": 2
                },
                "2015-03-01": {
                    "key_as_string": "2015-03-01",
                    "key": 1425168000000,
                    "doc_count": 2
                }
            }
        }
    }
}
```

Scripts

与普通直方图一样，支持 document-level scripts 和 value-level scripts。您可以使用`order`设置控制返回的桶的顺序，并根据min_doc_count设置过滤返回的桶（默认情况下，返回第一个与文档匹配的桶和最后一个桶之间的所有桶）。此直方图还支持extended_bounds设置，该设置允许将直方图的边界扩展到数据本身之外。

Missing value

```
POST /sales/_search?size=0
{
    "aggs" : {
        "sale_date" : {
            "date_histogram" : {
                "field" : "date",
                "interval": "year",
                "missing": "2000/01/01"
            }
         }
    }
}
```

publish_date字段中没有值的文档将与值为2000-01-01的文档属于同一个存储桶。

Order

默认情况下，返回的桶按关键字升序排序，但您可以使用顺序设置控制顺序。此设置支持与 Terms Aggregation相同的order功能。

使用脚本按星期几聚合

```
POST /sales/_search?size=0
{
    "aggs": {
        "dayOfWeek": {
            "terms": {
                "script": {
                    "lang": "painless",
                    "source": "doc['date'].value.dayOfWeekEnum.value"
                }
            }
        }
    }
}
```

返回

```
{
  ...
  "aggregations": {
    "dayOfWeek": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "7",
          "doc_count": 4
        },
        {
          "key": "4",
          "doc_count": 3
        }
      ]
    }
  }
}
```

自动间隔日期直方图 auto_date_histogram

Auto-interval Date Histogram Aggregation

类似于“Date Histogram Aggregation”的多桶聚合，不同之处在于，提供了指示所需桶数的目标桶数，并自动选择桶的间隔以最佳实现该目标，而不是提供用作每个桶宽度的间隔。返回的桶数始终小于或等于此目标数。

`buckets`字段是可选的，如果未指定，则默认为10个桶。

请求10个buckets。

```
POST /sales/_search?size=0
{
    "aggs" : {
        "sales_over_time" : {
            "auto_date_histogram" : {
                "field" : "date",
                "buckets" : 10
            }
        }
    }
}
```

Keys

在内部，日期表示为64位数字，表示自纪元以来的时间戳（以毫秒为单位）。这些时间戳作为bucket key返回。key_as_string是使用format参数指定的格式转换为格式化日期字符串的相同时间戳：

```
POST /sales/_search?size=0
{
    "aggs" : {
        "sales_over_time" : {
            "auto_date_histogram" : {
                "field" : "date",
                "buckets" : 5,
                "format" : "yyyy-MM-dd"
            }
```

```
        }
    }
}
```

响应

```
{
    ...
    "aggregations": {
        "sales_over_time": {
            "buckets": [
                {
                    "key_as_string": "2015-01-01",
                    "key": 1420070400000,
                    "doc_count": 3
                },
                {
                    "key_as_string": "2015-02-01",
                    "key": 1422748800000,
                    "doc_count": 2
                },
                {
                    "key_as_string": "2015-03-01",
                    "key": 1425168000000,
                    "doc_count": 2
                }
            ],
            "interval": "1M"
        }
    }
}
```

Intervals

返回的桶的间隔是根据聚合收集的数据选择的，以便返回的桶数小于或等于请求的数量。

参数	说明
seconds	1、5、10和30的倍数
minutes	1、5、10和30的倍数

参数	说明
days	1和7的倍数
months	1和3的倍数
years	1、5、10、20、50和100的倍数

在最坏的情况下，如果每日存储桶的数量超过请求的存储桶数量，则返回的存储桶数将是请求存储桶数量的1/7。

Time Zone

日期时间以UTC格式存储在Elasticsearch中。默认情况下，所有分格和舍入也在UTC中完成。time_zone参数可用于指示分组应使用不同的时区。

考虑以下示例：

```
PUT my_index/log/1?refresh
{
  "date": "2015-10-01T00:30:00Z"
}

PUT my_index/log/2?refresh
{
  "date": "2015-10-01T01:30:00Z"
}

PUT my_index/log/3?refresh
{
  "date": "2015-10-01T02:30:00Z"
}

GET my_index/_search?size=0
{
  "aggs": {
    "by_day": {
      "auto_date_histogram": {
        "field":     "date",
        "buckets" : 3
      }
    }
```

```
  }
}
```

如果未指定时区，则使用UTC，从2015年10月1日UTC午夜开始返回三个1小时时段：

```
{
  ...
  "aggregations": {
    "by_day": {
      "buckets": [
        {
          "key_as_string": "2015-10-01T00:00:00.000Z",
          "key": 1443657600000,
          "doc_count": 1
        },
        {
          "key_as_string": "2015-10-01T01:00:00.000Z",
          "key": 1443661200000,
          "doc_count": 1
        },
        {
          "key_as_string": "2015-10-01T02:00:00.000Z",
          "key": 1443664800000,
          "doc_count": 1
        }
      ],
      "interval": "1h"
    }
  }
}
```

如果指定了timezone -01:00，则午夜从UTC午夜前一小时开始：

```
GET my_index/_search?size=0
{
  "aggs": {
    "by_day": {
      "auto_date_histogram": {
        "field":     "date",
        "buckets" : 3,
        "time_zone": "-01:00"
      }
    }
  }
}
```

现在仍然返回三个1小时的存储桶，但第一个存储桶将于2015年9月30日晚上11点开始，因为这是指定时区中存储桶的本地时间。

```
{
  ...
  "aggregations": {
    "by_day": {
      "buckets": [
        {
          "key_as_string": "2015-09-30T23:00:00.000-01:00",
          "key": 1443657600000,
          "doc_count": 1
        },
        {
          "key_as_string": "2015-10-01T00:00:00.000-01:00",
          "key": 1443661200000,
          "doc_count": 1
        },
        {
          "key_as_string": "2015-10-01T01:00:00.000-01:00",
          "key": 1443664800000,
          "doc_count": 1
        }
      ],
      "interval": "1h"
    }
  }
}
```

> **INFO**
>
> 当使用DST（夏时制）更改后的时区时，接近这些更改发生时刻的存储桶可能与相邻存储桶的大小略有不同。例如，考虑CET时区的夏令时开始：2016年3月27日凌晨2点，当地时间提前1小时至凌晨3点。如果聚合的结果是每日存储桶，则当天的存储桶将只保存23小时的数据，而不是其他存储桶通常的24小时。对于较短的时间间隔（例如12小时）也是如此。在这里，我们将在3月27日上午夏令时转换时只有一个11小时的桶。

Script

与普通的date_histogram一样，支持文档级脚本和值级脚本。但是，此聚合不支持min_doc_count、extended_bounds和order参数。

Missing value

```
POST /sales/_search?size=0
{
    "aggs" : {
        "sale_date" : {
             "auto_date_histogram" : {
                 "field" : "date",
                 "buckets": 10,
                 "missing": "2000/01/01"
             }
         }
    }
}
```

publish_date字段中没有值的文档将与值为2000-01-01的文档属于同一个存储桶。

范围聚合 range

Range Aggregation

基于多桶聚合，使用户能够定义一组范围，每个范围代表一个桶。

```
GET /_search
{
    "aggs" : {
        "price_ranges" : {
            "range" : {
                "field" : "price",
                "ranges" : [
                    { "to" : 100.0 },
                    { "from" : 100.0, "to" : 200.0 },
                    { "from" : 200.0 }
                ]
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "price_ranges" : {
            "buckets": [
                {
                    "key": "*-100.0",
                    "to": 100.0,
                    "doc_count": 2
                },
                {
                    "key": "100.0-200.0",
                    "from": 100.0,
                    "to": 200.0,
                    "doc_count": 2
                },
                {
```

```
                    "key": "200.0-*",
                    "from": 200.0,
                    "doc_count": 3
                }
            ]
        }
    }
}
```

Keyed Response

将keyed标志设置为true会将一个唯一的字符串键与每个bucket相关联，并将范围作为哈希而不是数组返回：

```
GET /_search
{
    "aggs" : {
        "price_ranges" : {
            "range" : {
                "field" : "price",
                "keyed" : true,
                "ranges" : [
                    { "to" : 100 },
                    { "from" : 100, "to" : 200 },
                    { "from" : 200 }
                ]
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "price_ranges" : {
            "buckets": {
                "*-100.0": {
                    "to": 100.0,
                    "doc_count": 2
                },
                "100.0-200.0": {
                    "from": 100.0,
```

```
                    "to": 200.0,
                    "doc_count": 2
                },
                "200.0-*": {
                    "from": 200.0,
                    "doc_count": 3
                }
            }
        }
    }
}
```

还可以自定义每个范围的key：

```
GET /_search
{
    "aggs" : {
        "price_ranges" : {
            "range" : {
                "field" : "price",
                "keyed" : true,
                "ranges" : [
                    { "key" : "cheap", "to" : 100 },
                    { "key" : "average", "from" : 100, "to" : 200 },
                    { "key" : "expensive", "from" : 200 }
                ]
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "price_ranges" : {
            "buckets": {
                "cheap": {
                    "to": 100.0,
                    "doc_count": 2
                },
                "average": {
                    "from": 100.0,
                    "to": 200.0,
```

```
                    "doc_count": 2
                },
                "expensive": {
                    "from": 200.0,
                    "doc_count": 3
                }
            }
        }
    }
}
```

Script

范围聚合接受脚本参数。此参数允许定义将在聚合执行期间执行的inline script。

```
GET /_search
{
    "aggs" : {
        "price_ranges" : {
            "range" : {
                "script" : {
                    "lang": "painless",
                    "source": "doc['price'].value"
                },
                "ranges" : [
                    { "to" : 100 },
                    { "from" : 100, "to" : 200 },
                    { "from" : 200 }
                ]
            }
        }
    }
}
```

也可以使用存储的脚本。下面是一个简单的存储脚本：

```
POST /_scripts/convert_currency
{
  "script": {
    "lang": "painless",
    "source": "doc[params.field].value * params.conversion_rate"
  }
}
```

这个新的存储脚本可以在范围聚合中使用，如下所示：

```
GET /_search
{
    "aggs" : {
        "price_ranges" : {
            "range" : {
                "script" : {
                    "id": "convert_currency",
                    "params": {
                        "field": "price",
                        "conversion_rate": 0.835526591
                    }
                },
                "ranges" : [
                    { "from" : 0, "to" : 100 },
                    { "from" : 100 }
                ]
            }
        }
    }
}
```

Value Script

让我们假设产品价格是美元，但我们希望得到的价格范围是欧元。我们可以使用价值脚本转换聚合之前的价格（假设转换率为0.8）

```
GET /sales/_search
{
    "aggs" : {
        "price_ranges" : {
            "range" : {
                "field" : "price",
                "script" : {
                    "source": "_value * params.conversion_rate",
                    "params" : {
                        "conversion_rate" : 0.8
                    }
                },
                "ranges" : [
                    { "to" : 35 },
                    { "from" : 35, "to" : 70 },
                    { "from" : 70 }
```

```
                ]
            }
        }
    }
}
```

Sub Aggregations

下面的示例不仅将文档“桶”到不同的桶，还计算每个价格范围内的价格统计

```
GET /_search
{
    "aggs" : {
        "price_ranges" : {
            "range" : {
                "field" : "price",
                "ranges" : [
                    { "to" : 100 },
                    { "from" : 100, "to" : 200 },
                    { "from" : 200 }
                ]
            },
            "aggs" : {
                "price_stats" : {
                    "stats" : { "field" : "price" }
                }
            }
        }
    }
}
```

返回

```
{
  ...
  "aggregations": {
    "price_ranges": {
      "buckets": [
        {
          "key": "*-100.0",
          "to": 100.0,
          "doc_count": 2,
          "price_stats": {
            "count": 2,
```

```
            "min": 10.0,
            "max": 50.0,
            "avg": 30.0,
            "sum": 60.0
          }
        },
        {
          "key": "100.0-200.0",
          "from": 100.0,
          "to": 200.0,
          "doc_count": 2,
          "price_stats": {
            "count": 2,
            "min": 150.0,
            "max": 175.0,
            "avg": 162.5,
            "sum": 325.0
          }
        },
        {
          "key": "200.0-*",
          "from": 200.0,
          "doc_count": 3,
          "price_stats": {
            "count": 3,
            "min": 200.0,
            "max": 200.0,
            "avg": 200.0,
            "sum": 600.0
          }
        }
      ]
    }
  }
}
```

如果子聚合也基于与范围聚合相同的Value source（如上面示例中的统计数据聚合），则可以省略其Value source定义。以下将返回与上述相同的响应：

```
GET /_search
{
    "aggs" : {
        "price_ranges" : {
            "range" : {
                "field" : "price",
                "ranges" : [
                    { "to" : 100 },
```

```
                    { "from" : 100, "to" : 200 },
                    { "from" : 200 }
                ]
            },
            "aggs" : {
                "price_stats" : {
                    "stats" : {}
                }
            }
        }
    }
}
```

我们不需要指定price，因为我们默认从父范围聚合“继承”它

日期范围聚合 date_range

Date Range Aggregation

专用于日期值的范围聚合。此聚合与正常范围聚合之间的主要区别在于，from和to值可以用Date Math表达式表示，还可以指定返回from和to响应字段的日期格式。

```
POST /sales/_search?size=0
{
    "aggs": {
        "range": {
            "date_range": {
                "field": "date",
                "format": "MM-yyyy",
                "ranges": [
                    { "to": "now-10M/M" },
                    { "from": "now-10M/M" }
                ]
            }
        }
    }
}
```

在上面的示例中，我们创建了两个范围桶，第一个“桶”为所有日期在10个月之前的文档，第二个“桶”为所有日期为10个月之后的文档

```
{
    ...
    "aggregations": {
        "range": {
            "buckets": [
                {
                    "to": 1.4436576E12,
                    "to_as_string": "10-2015",
                    "doc_count": 7,
                    "key": "*-10-2015"
                },
                {
                    "from": 1.4436576E12,
                    "from_as_string": "10-2015",
                    "doc_count": 0,
```

```
                    "key": "10-2015-*"
                }
            ]
        }
    }
}
```

Missing Values

缺失值处理

```
POST /sales/_search?size=0
{
   "aggs": {
       "range": {
           "date_range": {
               "field": "date",
               "missing": "1976/11/30",
               "ranges": [
                   {
                     "key": "Older",
                     "to": "2016/02/01"
                   },
                   {
                     "key": "Newer",
                     "from": "2016/02/01",
                     "to" : "now/d"
                   }
               ]
           }
       }
   }
}
```

> 日期字段中没有值的文档将被添加到“Older”存储桶中。

Date Format/Pattern

参照JodaDate 其定义如下：

符号	含义	介绍	示例值

符号	含义	介绍	示例值
G	era	text	AD
C	century of era (>=0)	number	20
Y	year of era (>=0)	year	1996
x	weekyear	year	1996
w	week of weekyear	number	27
e	day of week	number	2
E	day of week	text	Tuesday; Tue
y	year	year	1996
D	day of year	number	189
M	month of year	month	July; Jul; 07
d	day of month	number	10
a	halfday of day	text	PM
K	hour of halfday (0~11)	number	0
h	clockhour of halfday (1~12)	number	12
H	hour of day (0~23)	number	0
k	clockhour of day (1~24)	number	24
m	minute of hour	number	30
s	second of minute	number	55

符号	含义	介绍	示例值
S	fraction of second	millis	978
z	time zone	text	Pacific Standard Time; PST
Z	time zone offset/id	zone	-0800; -08:00; America/Los_Angeles
'	escape for text	delimiter	''

Time zone

```
POST /sales/_search?size=0
{
   "aggs": {
       "range": {
           "date_range": {
               "field": "date",
               "time_zone": "CET",
               "ranges": [
                  { "to": "2016/02/01" },
                  { "from": "2016/02/01", "to" : "now/d" },
                  { "from": "now/d" }
              ]
           }
       }
   }
}
```

Keyed Response

将keyed标志设置为true会将一个唯一的字符串键与每个bucket相关联，并将范围作为哈希而不是数组返回：

```
POST /sales/_search?size=0
{
    "aggs": {
        "range": {
            "date_range": {
```

```
                "field": "date",
                "format": "MM-yyy",
                "ranges": [
                    { "to": "now-10M/M" },
                    { "from": "now-10M/M" }
                ],
                "keyed": true
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "range": {
            "buckets": {
                "*-10-2015": {
                    "to": 1.4436576E12,
                    "to_as_string": "10-2015",
                    "doc_count": 7
                },
                "10-2015-*": {
                    "from": 1.4436576E12,
                    "from_as_string": "10-2015",
                    "doc_count": 0
                }
            }
        }
    }
}
```

还可以自定义每个范围的key：

```
POST /sales/_search?size=0
{
    "aggs": {
        "range": {
            "date_range": {
                "field": "date",
                "format": "MM-yyy",
                "ranges": [
                    { "from": "01-2015",  "to": "03-2015", "key": "quarter_01" },
```

```
                    { "from": "03-2015", "to": "06-2015", "key": "quarter_02" }
                ],
                "keyed": true
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "range": {
            "buckets": {
                "quarter_01": {
                    "from": 1.4200704E12,
                    "from_as_string": "01-2015",
                    "to": 1.425168E12,
                    "to_as_string": "03-2015",
                    "doc_count": 5
                },
                "quarter_02": {
                    "from": 1.425168E12,
                    "from_as_string": "03-2015",
                    "to": 1.4331168E12,
                    "to_as_string": "06-2015",
                    "doc_count": 2
                }
            }
        }
    }
}
```

IP范围聚合 ip_range

IP Range Aggregation

```
GET /ip_addresses/_search
{
    "size": 10,
    "aggs" : {
        "ip_ranges" : {
            "ip_range" : {
                "field" : "ip",
                "ranges" : [
                    { "to" : "10.0.0.5" },
                    { "from" : "10.0.0.5" }
                ]
            }
        }
    }
}
```

返回

```
{
    ...

    "aggregations": {
        "ip_ranges": {
            "buckets" : [
                {
                    "key": "*-10.0.0.5",
                    "to": "10.0.0.5",
                    "doc_count": 10
                },
                {
                    "key": "10.0.0.5-*",
                    "from": "10.0.0.5",
                    "doc_count": 260
                }
            ]
        }
    }
}
```

IP范围也可以定义为CIDR掩码：

```
GET /ip_addresses/_search
{
    "size": 0,
    "aggs" : {
        "ip_ranges" : {
            "ip_range" : {
                "field" : "ip",
                "ranges" : [
                    { "mask" : "10.0.0.0/25" },
                    { "mask" : "10.0.0.127/25" }
                ]
            }
        }
    }
}
```

返回

```
{
    ...

    "aggregations": {
        "ip_ranges": {
            "buckets": [
                {
                    "key": "10.0.0.0/25",
                    "from": "10.0.0.0",
                    "to": "10.0.0.128",
                    "doc_count": 128
                },
                {
                    "key": "10.0.0.127/25",
                    "from": "10.0.0.0",
                    "to": "10.0.0.128",
                    "doc_count": 128
                }
            ]
        }
    }
}
```

Keyed

将keyed标志设置为true会将一个唯一的字符串键与每个bucket相关联，并将范围作为哈希而不是数组返回：

```
GET /ip_addresses/_search
{
    "size": 0,
    "aggs": {
        "ip_ranges": {
            "ip_range": {
                "field": "ip",
                "ranges": [
                    { "to" : "10.0.0.5" },
                    { "from" : "10.0.0.5" }
                ],
                "keyed": true
            }
        }
    }
}
```

返回

```
{
    ...

    "aggregations": {
        "ip_ranges": {
            "buckets": {
                "*-10.0.0.5": {
                    "to": "10.0.0.5",
                    "doc_count": 10
                },
                "10.0.0.5-*": {
                    "from": "10.0.0.5",
                    "doc_count": 260
                }
            }
        }
    }
}
```

还可以自定义每个范围的key：

```
GET /ip_addresses/_search
{
    "size": 0,
    "aggs": {
        "ip_ranges": {
            "ip_range": {
                "field": "ip",
                "ranges": [
                    { "key": "infinity", "to" : "10.0.0.5" },
                    { "key": "and-beyond", "from" : "10.0.0.5" }
                ],
                "keyed": true
            }
        }
    }
}
```

返回

```
{
    ...

    "aggregations": {
        "ip_ranges": {
            "buckets": {
                "infinity": {
                    "to": "10.0.0.5",
                    "doc_count": 10
                },
                "and-beyond": {
                    "from": "10.0.0.5",
                    "doc_count": 260
                }
            }
        }
    }
}
```

筛选聚合 filter

Filter Aggregation

定义当前文档集上下文中与指定筛选器匹配的所有文档的单个存储桶。通常，这将用于将当前聚合上下文缩小到一组特定的文档。

```
POST /sales/_search?size=0
{
    "aggs" : {
        "t_shirts" : {
            "filter" : { "term": { "type": "t-shirt" } },
            "aggs" : {
                "avg_price" : { "avg" : { "field" : "price" } }
            }
        }
    }
}
```

在上面的例子中，我们计算了t恤类型的所有产品的平均价格。

```
{
    ...
    "aggregations" : {
        "t_shirts" : {
            "doc_count" : 3,
            "avg_price" : { "value" : 128.33333333333334 }
        }
    }
}
```

多筛选聚合 filters

Filters Aggregation

基于多桶聚合，其中每个存储桶都与一个`filter`关联。每个存储桶将收集与其关联筛选器匹配的所有文档。

```
PUT /logs/_doc/_bulk?refresh
{ "index" : { "_id" : 1 } }
{ "body" : "warning: page could not be rendered" }
{ "index" : { "_id" : 2 } }
{ "body" : "authentication error" }
{ "index" : { "_id" : 3 } }
{ "body" : "warning: connection timed out" }

GET logs/_search
{
  "size": 0,
  "aggs" : {
    "messages" : {
      "filters" : {
        "filters" : {
          "errors" :   { "match" : { "body" : "error"   }},
          "warnings" : { "match" : { "body" : "warning" }}
        }
      }
    }
  }
}
```

在上面的示例中，我们分析了日志消息。聚合将构建两个日志消息集合（桶），一个用于所有包含错误的消息，另一个用于包含警告的消息。

```
{
  "took": 9,
  "timed_out": false,
  "_shards": ...,
  "hits": ...,
  "aggregations": {
    "messages": {
      "buckets": {
```

```
        "errors": {
          "doc_count": 1
        },
        "warnings": {
          "doc_count": 2
        }
      }
    }
  }
}
```

Anonymous filters

`filters`字段也可以作为数组提供，如以下请求所示

```
GET logs/_search
{
  "size": 0,
  "aggs" : {
    "messages" : {
      "filters" : {
        "filters" : [
          { "match" : { "body" : "error"   }},
          { "match" : { "body" : "warning" }}
        ]
      }
    }
  }
}
```

过滤后的桶以请求中提供的相同顺序返回。该示例的响应如下：

```
{
  "took": 4,
  "timed_out": false,
  "_shards": ...,
  "hits": ...,
  "aggregations": {
    "messages": {
      "buckets": [
        {
          "doc_count": 1
        },
        {
```

```
          "doc_count": 2
        }
      ]
    }
  }
}
```

Other Bucket

other_bucket参数可以设置为向响应中添加一个bucket，该bucket将包含与任何给定过滤器都不匹配的所有文档。此参数的值可以如下所示：

- **false** 不计算其他存储桶
- **true** 如果正在使用命名过滤器，则返回另一个存储桶（默认情况下命名为*other*），如果正在使用匿名过滤器，则作为最后一个存储

下面的代码片段显示了一个响应，其中请求将另一个bucket命名为other_messages。

```
PUT logs/_doc/4?refresh
{
  "body": "info: user Bob logged out"
}

GET logs/_search
{
  "size": 0,
  "aggs" : {
    "messages" : {
      "filters" : {
        "other_bucket_key": "other_messages",
        "filters" : {
          "errors" :   { "match" : { "body" : "error"   }},
          "warnings" : { "match" : { "body" : "warning" }}
        }
      }
    }
  }
}
```

返回

```

```

```
{
  "took": 3,
  "timed_out": false,
  "_shards": ...,
  "hits": ...,
  "aggregations": {
    "messages": {
      "buckets": {
        "errors": {
          "doc_count": 1
        },
        "warnings": {
          "doc_count": 2
        },
        "other_messages": {
          "doc_count": 1
        }
      }
    }
  }
}
```

地理距离聚合 geo_distance

Geo Distance Aggregation

在geo_point字段上工作的多桶聚合，在概念上与范围聚合非常相似。用户可以定义源点和一组距离范围桶。聚合评估每个文档值与原点的距离，并根据范围确定其所属的桶（如果文档与原点之间的距离在桶的距离范围内，则文档属于该桶）。

```
PUT /museums
{
    "mappings": {
        "_doc": {
            "properties": {
                "location": {
                    "type": "geo_point"
                }
            }
        }
    }
}

POST /museums/_doc/_bulk?refresh
{"index":{"_id":1}}
{"location": "52.374081,4.912350", "name": "NEMO Science Museum"}
{"index":{"_id":2}}
{"location": "52.369219,4.901618", "name": "Museum Het Rembrandthuis"}
{"index":{"_id":3}}
{"location": "52.371667,4.914722", "name": "Nederlands Scheepvaartmuseum"}
{"index":{"_id":4}}
{"location": "51.222900,4.405200", "name": "Letterenhuis"}
{"index":{"_id":5}}
{"location": "48.861111,2.336389", "name": "Musée du Louvre"}
{"index":{"_id":6}}
{"location": "48.860000,2.327000", "name": "Musée d'Orsay"}

POST /museums/_search?size=0
{
    "aggs" : {
        "rings_around_amsterdam" : {
            "geo_distance" : {
                "field" : "location",
                "origin" : "52.3760, 4.894",
                "ranges" : [
```

```
                    { "to" : 100000 },
                    { "from" : 100000, "to" : 300000 },
                    { "from" : 300000 }
                ]
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "rings_around_amsterdam" : {
            "buckets": [
                {
                    "key": "*-100000.0",
                    "from": 0.0,
                    "to": 100000.0,
                    "doc_count": 3
                },
                {
                    "key": "100000.0-300000.0",
                    "from": 100000.0,
                    "to": 300000.0,
                    "doc_count": 1
                },
                {
                    "key": "300000.0-*",
                    "from": 300000.0,
                    "doc_count": 2
                }
            ]
        }
    }
}
```

`field` 必须是geo_point类型，也可以为一组geo_point字段。

`origin` 可以接受geo_point类型支持的所有格式：

- 对象格式： {“lat”：52.3760，“lon”：4.894} -这是最安全的格式，因为它是最明确的lat和lon值
- 字符串格式：“52.3760，4.894”-其中第一个数字是lat，第二个数字是lon

- 数组格式：[4.894，52.3760]-基于GeoJson标准，其中第一个数字是lon，第二个数字是lat

默认情况下，距离单位为m（米），但也可以接受：mi（英里）、in（英寸）、yd（码）、km（公里）、cm（厘米）、mm（毫米）。

```
POST /museums/_search?size=0
{
    "aggs" : {
        "rings" : {
            "geo_distance" : {
                "field" : "location",
                "origin" : "52.3760, 4.894",
                "unit" : "km",
                "ranges" : [
                    { "to" : 100 },
                    { "from" : 100, "to" : 300 },
                    { "from" : 300 }
                ]
            }
        }
    }
}
```

有两种距离计算模式：`arc` 圆弧（默认）和 `plane` 平面。圆弧计算最准确。`plane`最快但最不准确的。当您的搜索上下文为狭窄，并且跨越较小的地理区域（约5km）时，请考虑使用plane。plane将为跨越非常大区域的搜索（例如跨大陆搜索）返回更高的误差容限。可以使用distance_type参数设置距离计算类型：

```
POST /museums/_search?size=0
{
    "aggs" : {
        "rings" : {
            "geo_distance" : {
                "field" : "location",
                "origin" : "52.3760, 4.894",
                "unit" : "km",
                "distance_type" : "plane",
                "ranges" : [
                    { "to" : 100 },
                    { "from" : 100, "to" : 300 },
                    { "from" : 300 }
                ]
            }
        }
```

```
    }
}
```

Keyed Response

将keyed标志设置为true会将一个唯一的字符串键与每个bucket相关联，并将范围作为哈希而不是数组返回：

```
POST /museums/_search?size=0
{
    "aggs" : {
        "rings_around_amsterdam" : {
            "geo_distance" : {
                "field" : "location",
                "origin" : "52.3760, 4.894",
                "ranges" : [
                    { "to" : 100000 },
                    { "from" : 100000, "to" : 300000 },
                    { "from" : 300000 }
                ],
                "keyed": true
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "rings_around_amsterdam" : {
            "buckets": {
                "*-100000.0": {
                    "from": 0.0,
                    "to": 100000.0,
                    "doc_count": 3
                },
                "100000.0-300000.0": {
                    "from": 100000.0,
                    "to": 300000.0,
                    "doc_count": 1
                },
                "300000.0-*": {
```

```
                    "from": 300000.0,
                    "doc_count": 2
                }
            }
        }
    }
}
```

还可以自定义每个范围的key：

```
POST /museums/_search?size=0
{
    "aggs" : {
        "rings_around_amsterdam" : {
            "geo_distance" : {
                "field" : "location",
                "origin" : "52.3760, 4.894",
                "ranges" : [
                    { "to" : 100000, "key": "first_ring" },
                    { "from" : 100000, "to" : 300000, "key": "second_ring" },
                    { "from" : 300000, "key": "third_ring" }
                ],
                "keyed": true
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "rings_around_amsterdam" : {
            "buckets": {
                "first_ring": {
                    "from": 0.0,
                    "to": 100000.0,
                    "doc_count": 3
                },
                "second_ring": {
                    "from": 100000.0,
                    "to": 300000.0,
                    "doc_count": 1
                },
```

```
                "third_ring": {
                    "from": 300000.0,
                    "doc_count": 2
                }
            }
        }
    }
}
```

GeoHash网格聚合 geohash_grid

GeoHash grid Aggregation

一种多桶聚合，用于geo_point字段，并将点分组到表示网格中单元格的桶中。生成的网格可以是稀疏的，并且只包含具有匹配数据的单元格。每个单元格都使用具有用户可定义精度的geohash进行标记。

- 高精度geohash具有很长的字符串长度，表示只覆盖很小区域的单元格。
- 低精度geohash的字符串长度很短，表示每个单元格覆盖的区域很大。

此聚合中使用的Geohash可以在1到12之间选择精度。

> **INFO**
>
> 长度为12的最高精度geohash产生的单元覆盖面积不到一平方米，因此高精度请求在内存和结果size方面可能非常昂贵。请参阅下面的示例，了解如何在请求高级别的详细信息之前先将聚合过滤到较小的地理区域。

指定的字段必须是geo_point类型，并且它还可以包含一个geo_point字段数组，在这种情况下，聚合期间将考虑所有点。

Low-precision 低精度请求

```
PUT /museums
{
    "mappings": {
        "_doc": {
            "properties": {
                "location": {
                    "type": "geo_point"
                }
            }
        }
    }
}

POST /museums/_doc/_bulk?refresh
{"index":{"_id":1}}
```

```
{"location": "52.374081,4.912350", "name": "NEMO Science Museum"}
{"index":{"_id":2}}
{"location": "52.369219,4.901618", "name": "Museum Het Rembrandthuis"}
{"index":{"_id":3}}
{"location": "52.371667,4.914722", "name": "Nederlands Scheepvaartmuseum"}
{"index":{"_id":4}}
{"location": "51.222900,4.405200", "name": "Letterenhuis"}
{"index":{"_id":5}}
{"location": "48.861111,2.336389", "name": "Musée du Louvre"}
{"index":{"_id":6}}
{"location": "48.860000,2.327000", "name": "Musée d'Orsay"}

POST /museums/_search?size=0
{
    "aggregations" : {
        "large-grid" : {
            "geohash_grid" : {
                "field" : "location",
                "precision" : 3
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "large-grid": {
            "buckets": [
                {
                    "key": "u17",
                    "doc_count": 3
                },
                {
                    "key": "u09",
                    "doc_count": 2
                },
                {
                    "key": "u15",
                    "doc_count": 1
                }
            ]
        }
    }
}
```

High-precision 高精度请求

当请求详细的存储桶（通常用于显示“放大”地图）时，应该应用geo_bounding_box这样的过滤器来缩小主区域，否则可能会创建并返回数百万个桶。

```
POST /museums/_search?size=0
{
    "aggregations" : {
        "zoomed-in" : {
            "filter" : {
                "geo_bounding_box" : {
                    "location" : {
                        "top_left" : "52.4, 4.9",
                        "bottom_right" : "52.3, 5.0"
                    }
                }
            },
            "aggregations":{
                "zoom1":{
                    "geohash_grid" : {
                        "field": "location",
                        "precision": 8
                    }
                }
            }
        }
    }
}
```

geohash_grid聚合返回的作为bucket keys的geohash也可以用于“放大”，方法是使用一个可用的geohash库将它们转换为bounding boxes。例如，对于javascript，可以使用节点geohash库：

```
var geohash = require('ngeohash');

// bbox will contain [ 52.03125, 4.21875, 53.4375, 5.625 ]
//                   [   minlat,  minlon,  maxlat, maxlon]
var bbox = geohash.decode_bbox('u17');
```

赤道处的单元尺寸

下表显示了geohash的不同字符串长度覆盖的单元格的度量维度。单元尺寸随纬度而变化，因此该表适用于赤道的worst-case。

GeoHash 长度	面积 宽 x 高
1	5,009.4km x 4,992.6km
2	1,252.3km x 624.1kmm
3	156.5km x 156km
4	39.1km x 19.5km
5	4.9km x 4.9km
6	1.2km x 609.4m
7	152.9m x 152.4m
8	38.2m x 19m
9	4.8m x 4.8m
10	1.2m x 59.5cm
11	14.9cm x 14.9cm
12	3.7cm x 1.9cm

参数	说明
field	（必选）GeoPoints索引的字段

参数	说明
precision	（可选）用于在结果中定义 cells/buckets的geohash的字符串长度。默认值为5。精度可以根据上述整数精度级别定义。[1,12]以外的值将被拒绝。或者，精度等级可以通过“1km”、“10m”等距离度量来近似。计算精度级别时，单元格不会超过所需精度的指定大小（对角线）。当这将导致精度级别高于支持的12个级别时（例如，距离<5.6cm），该值将被拒绝
size	（可选）要返回的最大geohash桶数（默认为10000）。当对结果进行调整时，桶将根据其包含的文档量进行优先级排序。
shard_size	（可选）为了更准确地计算最终结果中返回的 top cells，聚合默认为从每个shard返回最大`max(10,(size x number-of-shards))`个桶。如果不希望使用这种启发式方法，则可以使用此参数覆盖每个shard中考虑的数量。

父聚合 parent

Parent Aggregation

一种特殊的单桶聚合，用于选择具有join字段中定义的指定类型的父文档。

此聚合只有一个选项：

- type： 必选项 child type。

例如，假设我们有一个问题和答案索引。应答类型在映射中具有以下`join`字段：

```
PUT parent_example
{
  "mappings": {
    "_doc": {
      "properties": {
        "join": {
          "type": "join",
          "relations": {
            "question": "answer"
          }
        }
      }
    }
  }
}
```

question文档包含tag字段，answer文档包含owner字段。使用父聚合，owner buckets可以在单个请求中映射到tag buckets，即使这两个字段存在于两种不同类型的文档中。

question文档示例

```
PUT parent_example/_doc/1
{
  "join": {
    "name": "question"
  },
  "body": "<p>I have Windows 2003 server and i bought a new Windows 2008
server...",
```

```
  "title": "Whats the best way to file transfer my site from server to a newer
one?",
  "tags": [
    "windows-server-2003",
    "windows-server-2008",
    "file-transfer"
  ]
}
```

answer 文档示例：

```
PUT parent_example/_doc/2?routing=1
{
  "join": {
    "name": "answer",
    "parent": "1"
  },
  "owner": {
    "location": "Norfolk, United Kingdom",
    "display_name": "Sam",
    "id": 48
  },
  "body": "<p>Unfortunately you're pretty much limited to FTP...",
  "creation_date": "2009-05-04T13:45:37.030"
}

PUT parent_example/_doc/3?routing=1&refresh
{
  "join": {
    "name": "answer",
    "parent": "1"
  },
  "owner": {
    "location": "Norfolk, United Kingdom",
    "display_name": "Troll",
    "id": 49
  },
  "body": "<p>Use Linux...",
  "creation_date": "2009-05-05T13:45:37.030"
}
```

可以构建以下请求，将两者连接在一起：

```
POST parent_example/_search?size=0
{
```

```
  "aggs": {
    "top-names": {
      "terms": {
        "field": "owner.display_name.keyword",
        "size": 10
      },
      "aggs": {
        "to-questions": {
          "parent": {
            "type" : "answer"
          },
          "aggs": {
            "top-tags": {
              "terms": {
                "field": "tags.keyword",
                "size": 10
              }
            }
          }
        }
      }
    }
  }
}
```

上面的示例返回 top answer owners 和每个owner的top question tags。

响应：

```
{
  "took": 9,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 3,
    "max_score": 0.0,
    "hits": []
  },
  "aggregations": {
    "top-names": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
```

```
      "buckets": [
        {
          "key": "Sam",
          "doc_count": 1,
          "to-questions": {
            "doc_count": 1,
            "top-tags": {
              "doc_count_error_upper_bound": 0,
              "sum_other_doc_count": 0,
              "buckets": [
                {
                  "key": "file-transfer",
                  "doc_count": 1
                },
                {
                  "key": "windows-server-2003",
                  "doc_count": 1
                },
                {
                  "key": "windows-server-2008",
                  "doc_count": 1
                }
              ]
            }
          }
        },
        {
          "key": "Troll",
          "doc_count": 1,
          "to-questions": {
            "doc_count": 1,
            "top-tags": {
              "doc_count_error_upper_bound": 0,
              "sum_other_doc_count": 0,
              "buckets": [
                {
                  "key": "file-transfer",
                  "doc_count": 1
                },
                {
                  "key": "windows-server-2003",
                  "doc_count": 1
                },
                {
                  "key": "windows-server-2008",
                  "doc_count": 1
                }
              ]
            }
```

```
            }
          }
        ]
      }
    }
}
```

子聚合 children

Children Aggregation

一种特殊的单桶聚合，用于选择具有`join`字段中定义的指定类型的子文档。

此聚合只有一个选项：

- type: child type。

例如，假设我们有一个questions和answers索引。answer type在mapping中具有以下json字段：

```
PUT child_example
{
  "mappings": {
    "_doc": {
      "properties": {
        "join": {
          "type": "join",
          "relations": {
            "question": "answer"
          }
        }
      }
    }
  }
}
```

questions文档包含tag字段，answer文档包含owner字段。通过子聚合，tag buckets桶可以在单个请求中映射到owner buckets，即使这两个字段存在于两种不同类型的文档中。

questions文档示例

```
PUT child_example/_doc/1
{
  "join": {
    "name": "question"
  },
  "body": "<p>I have Windows 2003 server and i bought a new Windows 2008
server...",
```

```
  "title": "Whats the best way to file transfer my site from server to a newer
one?",
  "tags": [
    "windows-server-2003",
    "windows-server-2008",
    "file-transfer"
  ]
}
```

回答文件示例：

```
PUT child_example/_doc/2?routing=1
{
  "join": {
    "name": "answer",
    "parent": "1"
  },
  "owner": {
    "location": "Norfolk, United Kingdom",
    "display_name": "Sam",
    "id": 48
  },
  "body": "<p>Unfortunately you're pretty much limited to FTP...",
  "creation_date": "2009-05-04T13:45:37.030"
}

PUT child_example/_doc/3?routing=1&refresh
{
  "join": {
    "name": "answer",
    "parent": "1"
  },
  "owner": {
    "location": "Norfolk, United Kingdom",
    "display_name": "Troll",
    "id": 49
  },
  "body": "<p>Use Linux...",
  "creation_date": "2009-05-05T13:45:37.030"
}
```

可以构建以下请求，将两者连接在一起：

```
POST child_example/_search?size=0
{
```

```
  "aggs": {
    "top-tags": {
      "terms": {
        "field": "tags.keyword",
        "size": 10
      },
      "aggs": {
        "to-answers": {
          "children": {
            "type" : "answer"
          },
          "aggs": {
            "top-names": {
              "terms": {
                "field": "owner.display_name.keyword",
                "size": 10
              }
            }
          }
        }
      }
    }
  }
}
```

> 类型指向带有名称 answer 的 type / mapping。

上面的示例返回 top question tags 和每个 top answer owners 的tag。

响应：

```
{
  "took": 25,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "skipped" : 0,
    "failed": 0
  },
  "hits": {
    "total": 3,
    "max_score": 0.0,
    "hits": []
  },
  "aggregations": {
    "top-tags": {
```

```
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "file-transfer",
          "doc_count": 1,
          "to-answers": {
            "doc_count": 2,
            "top-names": {
              "doc_count_error_upper_bound": 0,
              "sum_other_doc_count": 0,
              "buckets": [
                {
                  "key": "Sam",
                  "doc_count": 1
                },
                {
                  "key": "Troll",
                  "doc_count": 1
                }
              ]
            }
          }
        },
        {
          "key": "windows-server-2003",
          "doc_count": 1,
          "to-answers": {
            "doc_count": 2,
            "top-names": {
              "doc_count_error_upper_bound": 0,
              "sum_other_doc_count": 0,
              "buckets": [
                {
                  "key": "Sam",
                  "doc_count": 1
                },
                {
                  "key": "Troll",
                  "doc_count": 1
                }
              ]
            }
          }
        },
        {
          "key": "windows-server-2008",
          "doc_count": 1,
          "to-answers": {
```

```
            "doc_count": 2,
            "top-names": {
              "doc_count_error_upper_bound": 0,
              "sum_other_doc_count": 0,
              "buckets": [
                {
                  "key": "Sam",
                  "doc_count": 1
                },
                {
                  "key": "Troll",
                  "doc_count": 1
                }
              ]
            }
          }
        }
      ]
    }
  }
}
```

嵌套聚合 nested

Nested Aggregation

一种特殊的单桶聚合，支持聚合嵌套文档。

例如，假设我们有一个products索引，每个product都有resellers列表，每个reseller都有自己的product price。映射可能如下所示：

```
PUT /products
{
  "mappings": {
    "product" : {
        "properties" : {
            "resellers" : {
                "type" : "nested",
                "properties" : {
                    "reseller" : { "type" : "text" },
                    "price" : { "type" : "double" }
                }
            }
        }
    }
  }
}
```

> resellers 是一个包含嵌套文档的数组。

以下请求添加了一个有两个resellers的product：

```
PUT /products/_doc/0
{
  "name": "LED TV",
  "resellers": [
    {
      "reseller": "companyA",
      "price": 350
    },
    {
      "reseller": "companyB",
      "price": 500
```

```
    }
  ]
}
```

以下请求返回购买product的最低price：

```
GET /products/_search
{
    "query" : {
        "match" : { "name" : "led tv" }
    },
    "aggs" : {
        "resellers" : {
            "nested" : {
                "path" : "resellers"
            },
            "aggs" : {
                "min_price" : { "min" : { "field" : "resellers.price" } }
            }
        }
    }
}
```

如上所述， nested aggregation 需要`top level documents`中嵌套文档的`path`。然后可以在这些嵌套文档上定义任何类型的聚合。

返回：

```
{
  ...
  "aggregations": {
    "resellers": {
      "doc_count": 2,
      "min_price": {
        "value": 350
      }
    }
  }
}
```

反向嵌套聚合 reverse_nested

Reverse nested Aggregation

一种特殊的单桶聚合，支持从嵌套文档聚合父文档。实际上，此聚合可以脱离嵌套的块结构，并链接到其他嵌套结构或根文档，这允许嵌套其他聚合，而这些聚合不是嵌套聚合中嵌套对象的一部分。

反向嵌套聚合必须在嵌套聚合中定义。

选项：

- path-它定义了嵌套对象字段应该连接回的位置。默认值为空，这意味着它重新连接到 root / main document level。path不能包含对嵌套对象字段的引用，该字段位于reverse_nested所在的嵌套聚合的嵌套结构之外。

例如，假设我们有一个包含issues和comments的ticket system index。`comments`作为嵌套文档内联到`issues`文档中。映射可能如下所示：

```
PUT /issues
{
    "mappings": {
        "issue" : {
            "properties" : {
                "tags" : { "type" : "keyword" },
                "comments" : {
                    "type" : "nested",
                    "properties" : {
                        "username" : { "type" : "keyword" },
                        "comment" : { "type" : "text" }
                    }
                }
            }
        }
    }
}
```

以下聚合将返回已发表评论的顶级评论者的用户名，每个顶级评论者将返回用户评论的问题的顶级标签：

```
GET /issues/_search
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "comments": {
      "nested": {
        "path": "comments"
      },
      "aggs": {
        "top_usernames": {
          "terms": {
            "field": "comments.username"
          },
          "aggs": {
            "comment_to_issue": {
              "reverse_nested": {},
              "aggs": {
                "top_tags_per_comment": {
                  "terms": {
                    "field": "tags"
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

如上所述，反向嵌套聚合被放入嵌套聚合中，因为这是dsl中唯一可以使用反向嵌套聚合的地方。它的唯一目的是连接回嵌套结构中更高级别的父文档。

> reverse_nested 聚合，因为尚未定义path，所以它会连接root / main document level。如果mapping中定义了多个分层嵌套对象类型，则反向嵌套聚合可以通过path选项连接回不同的级别

返回

```
{
  "aggregations": {
    "comments": {
      "doc_count": 1,
```

```
      "top_usernames": {
        "doc_count_error_upper_bound" : 0,
        "sum_other_doc_count" : 0,
        "buckets": [
          {
            "key": "username_1",
            "doc_count": 1,
            "comment_to_issue": {
              "doc_count": 1,
              "top_tags_per_comment": {
                "doc_count_error_upper_bound" : 0,
                "sum_other_doc_count" : 0,
                "buckets": [
                  {
                    "key": "tag_1",
                    "doc_count": 1
                  }
                  ...
                ]
              }
            }
          }
          ...
        ]
      }
    }
  }
}
```

复合聚合 composite

Composite Aggregation

从不同来源创建复合桶的多桶聚合。

与其他多桶聚合不同，复合聚合可用于高效地对多级聚合中的所有桶进行分页。这种聚合提供了一种方式来流化特定聚合的所有桶，类似于滚动对文档所做的操作。

composite buckets 是根据为每个文档 extracted/created 的值的组合构建的，每个组合都被视为composite bucket。

例如，以下文件

```
{
    "keyword": ["foo", "bar"],
    "number": [23, 65, 76]
}
```

...当`keyword`和`number`用作聚合的value source时，将创建以下复合桶：

```
{ "keyword": "foo", "number": 23 }
{ "keyword": "foo", "number": 65 }
{ "keyword": "foo", "number": 76 }
{ "keyword": "bar", "number": 23 }
{ "keyword": "bar", "number": 65 }
{ "keyword": "bar", "number": 76 }
```

Values source

`sources`参数控制应用于构建复合存储桶的源。有三种不同类型的values source：

Terms

terms value source 相当于一个简单的`terms aggregation`。这些值是从字段或脚本中提取的，与`terms aggregation`完全相同。

例子

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
                "sources" : [
                    { "product": { "terms" : { "field": "product" } } }
                ]
            }
        }
     }
}
```

与terms aggregation类似，也可以使用脚本创建复合桶的值：

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
                "sources" : [
                    {
                        "product": {
                            "terms" : {
                                "script" : {
                                    "source": "doc['product'].value",
                                    "lang": "painless"
                                }
                            }
                        }
                    }
                ]
            }
        }
    }
}
```

Histogram

Histogram value source 可以应用于数值，以在数值上建立固定大小的间隔。interval参数定义如何转换数值。例如，设置为5的间隔将将任何数值转换为其最近的间隔，值101将转换为100，这是100和105之间的间隔的关键

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
                "sources" : [
                    { "histo": { "histogram" : { "field": "price", "interval": 5 }
} }
                ]
            }
        }
    }
}
```

这些值是从返回数值的数值字段或脚本生成的：

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
                "sources" : [
                    {
                        "histo": {
                            "histogram" : {
                                "interval": 5,
                                "script" : {
                                    "source": "doc['price'].value",
                                    "lang": "painless"
                                }
                            }
                        }
                    }
                ]
            }
        }
    }
}
```

Date Histogram

date_histogram与histogram value source相似，只是间隔由日期/时间表达式指定：

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
                "sources" : [
                    { "date": { "date_histogram" : { "field": "timestamp",
"interval": "1d" } } }
                ]
            }
        }
    }
}
```

上面的示例每天创建一个间隔，并将所有时间戳值转换为最近间隔的开始。间隔的可用表达式：`year, quarter, month, week, day, hour, minute, second`

时间值也可以通过时间单位解析支持的缩写来指定。请注意，不支持分数时间值，但您可以通过转换到另一个时间单位（例如，可以将1.5h指定为90m）来解决此问题。

Format

在内部，日期表示为64位数字，表示自纪元以来的时间戳（以毫秒为单位）。这些时间戳作为bucket key返回。可以使用format参数指定的格式返回格式化的日期字符串：

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
                "sources" : [
                    {
                        "date": {
                            "date_histogram" : {
                                "field": "timestamp",
                                "interval": "1d",
                                "format": "yyyy-MM-dd"
                            }
                        }
```

```
                    }
                ]
            }
        }
    }
}
```

Time Zone

日期时间以UTC格式存储在Elasticsearch中。time_zone参数可用于指示分组应使用不同的时区。

时区可以指定为ISO 8601 UTC偏移量（例如+01:00或-08:00），也可以指定为时区id，这是TZ数据库中使用的标识符，如America/Loss_Angeles。

Mixing different values source

sources参数接受value source的数组。可以混合不同的value来创建复合桶。例如：

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
                "sources" : [
                    { "date": { "date_histogram": { "field": "timestamp",
"interval": "1d" } } },
                    { "product": { "terms": {"field": "product" } } }
                ]
            }
        }
    }
}
```

这将从两个value source（date_histogram和terms）创建的值创建复合桶。每个存储桶由两个值组成，一个用于聚合中定义的每个value source。允许任何类型的组合，数组中的顺序保留在组合桶中。

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
```

```
                "sources" : [
                    { "shop": { "terms": {"field": "shop" } } },
                    { "product": { "terms": { "field": "product" } } },
                    { "date": { "date_histogram": { "field": "timestamp",
"interval": "1d" } } }
                ]
            }
        }
    }
}
```

Order

默认情况下，复合桶按其自然顺序排序。值按其值的升序排序。当请求多个value source时，对每个value source进行排序，将组合桶的第一个值与另一个组合桶的第二个值进行比较，如果它们相等，则组合桶中的下一个值将用于断开连接。这意味着复合桶[foo，100]被认为小于[foobar，0]，因为foo被认为小于foobar。通过直接在value source定义中将顺序设置为asc（默认值）或desc（降序），可以定义每个value source的排序方向。例如：

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
                "sources" : [
                    { "date": { "date_histogram": { "field": "timestamp",
"interval": "1d", "order": "desc" } } },
                    { "product": { "terms": {"field": "product", "order": "asc" } }
}
                ]
            }
        }
    }
}
```

...将在比较date_histogram源中的值时按降序对复合存储桶进行排序，在比较术语源中的数值时按升序对复合存储。

Missing bucket

默认情况下，忽略没有给定source value的文档。通过将missing_bucket设置为true（默认为false），可以将它们包含在响应中：

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
                "sources" : [
                    { "product_name": { "terms" : { "field": "product",
"missing_bucket": true } } }
                ]
            }
        }
     }
}
```

在上面的示例中， source product_name将为当product字段为空值的文档指定显示的`null`值，。`order`决定了空值应该排在第一位（升序，asc）还是最后一位（降序，desc）。

Size

可以设置size参数来定义应该返回多少复合桶。每个复合桶被视为单个桶，因此将大小设置为10将返回从value source创建的前10个复合桶。响应包含数组中每个复合存储桶的值，该数组包含从每个value source提取的值。

After

如果复合桶的数量太多（或未知），无法在单个响应中返回，则可以将检索拆分为多个请求。由于复合桶本质上是平的，因此请求的大小正好是将在响应中返回的复合桶的数量（假设它们至少是要返回的大小的复合桶）。如果应该检索所有复合桶，最好使用较小的大小（例如100或1000），然后使用after参数检索下一个结果。例如：

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
                "size": 2,
                "sources" : [
```

```
                    { "date": { "date_histogram": { "field": "timestamp",
"interval": "1d" } } },
                    { "product": { "terms": {"field": "product" } } }
                ]
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "my_buckets": {
            "after_key": {
                "date": 1494288000000,
                "product": "mad max"
            },
            "buckets": [
                {
                    "key": {
                        "date": 1494201600000,
                        "product": "rocky"
                    },
                    "doc_count": 1
                },
                {
                    "key": {
                        "date": 1494288000000,
                        "product": "mad max"
                    },
                    "doc_count": 2
                }
            ]
        }
    }
}
```

> after_key等于响应中返回的最后一个bucket，该bucket是管道聚合进行任何过滤之前返回的。如果管道聚合过滤/删除了所有桶，after_key将包含过滤前的最后一个桶。

after参数可用于检索上一轮返回的最后一个复合存储桶之后的复合存储桶。对于下面的示例，最后一个桶可以在after_key中找到，下一轮结果可以通过以下方式检索：

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
                "size": 2,
                 "sources" : [
                    { "date": { "date_histogram": { "field": "timestamp",
"interval": "1d", "order": "desc" } } },
                    { "product": { "terms": {"field": "product", "order": "asc" } }
}
                ],
                "after": { "date": 1494288000000, "product": "mad max" }
            }
        }
    }
}
```

Sub-aggregations

与任何多桶聚合一样，复合聚合可以包含子聚合。这些子聚合可用于计算其他存储桶或此父聚合创建的每个复合存储桶的统计信息。例如，以下示例计算每个复合存储桶的字段平均值：

```
GET /_search
{
    "aggs" : {
        "my_buckets": {
            "composite" : {
                 "sources" : [
                    { "date": { "date_histogram": { "field": "timestamp",
"interval": "1d", "order": "desc" } } },
                    { "product": { "terms": {"field": "product" } } }
                ]
            },
            "aggregations": {
                "the_avg": {
                    "avg": { "field": "price" }
                }
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "my_buckets": {
            "after_key": {
                "date": 1494201600000,
                "product": "rocky"
            },
            "buckets": [
                {
                    "key": {
                        "date": 1494460800000,
                        "product": "apocalypse now"
                    },
                    "doc_count": 1,
                    "the_avg": {
                        "value": 10.0
                    }
                },
                {
                    "key": {
                        "date": 1494374400000,
                        "product": "mad max"
                    },
                    "doc_count": 1,
                    "the_avg": {
                        "value": 27.0
                    }
                },
                {
                    "key": {
                        "date": 1494288000000,
                        "product" : "mad max"
                    },
                    "doc_count": 2,
                    "the_avg": {
                        "value": 22.5
                    }
                },
                {
                    "key": {
                        "date": 1494201600000,
                        "product": "rocky"
                    },
                    "doc_count": 1,
                    "the_avg": {
                        "value": 10.0
                    }
```

```
                }
            ]
        }
    }
}
```

全局聚合 global

Global Aggregation

Global Aggregation 是对所有的文档进行聚合，而不受查询条件的限制

> 全局聚合器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意义。

比如： 我们有50个文档，通过查询条件筛选之后存在10个文档，此时我想统计总共有多少个文档。是50个，因为global统计不受查询条件的限制。

```
POST /sales/_search?size=0
{
    "query" : {
        "match" : { "type" : "t-shirt" }
    },
    "aggs" : {
        "all_products" : {
            "global" : {},
            "aggs" : {
                "avg_price" : { "avg" : { "field" : "price" } }
            }
        },
        "t_shirts": { "avg" : { "field" : "price" } }
    }
}
```

上面的聚合演示了如何计算搜索上下文中所有文档的聚合（在本例中为avg_price），而不考虑query。

上述聚合的响应

```
{
    ...
    "aggregations" : {
        "all_products" : {
            "doc_count" : 7, (1)
            "avg_price" : {
                "value" : 140.71428571428572 (2)
            }
        },
```

```
        "t_shirts": {
            "value" : 128.33333333333334 (3)
        }
    }
}
```

- (1)聚合的文档数（在本例中，是搜索上下文中的所有文档）
- (2)指数中所有产品的平均价格
- (3)所有t恤的平均价格

缺值聚合 Missing

Missing Aggregation

基于字段数据的单桶聚合，创建当前文档集上下文中缺少字段值的所有文档的bucket（桶）（有效地，丢失了一个字段或配置了NULL值集），此聚合器通常与其他字段数据桶聚合器（例如范围）结合使用，以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。

```
POST /sales/_search?size=0
{
    "aggs" : {
        "products_without_a_price" : {
            "missing" : { "field" : "price" }
        }
    }
}
```

在上面的例子中，我们得到了没有价格的产品总数。

```
{
    ...
    "aggregations" : {
        "products_without_a_price" : {
            "doc_count" : 00
        }
    }
}
```

采样器聚合 sampler

Sampler Aggregation

一种 filtering aggregation 用于将任何子聚合的处理限制为得分最高的文档样本。

示例用例：

- 将分析的重点集中在高相关性匹配上，而不是低质量匹配的潜在长尾
- 降低聚合的运行成本，这些聚合仅使用样本（例如significant_terms）即可产生有用的结果

例子：

查询StackOverflow数据中的流行术语javascript或罕见术语kibana将匹配许多文档，其中大多数缺少kibana一词。为了将significant_terms聚合集中在得分最高的文档上，这些文档更有可能匹配查询中最有趣的部分，我们使用了一个示例。

```
POST /stackoverflow/_search?size=0
{
    "query": {
        "query_string": {
            "query": "tags:kibana OR tags:javascript"
        }
    },
    "aggs": {
        "sample": {
            "sampler": {
                "shard_size": 200
            },
            "aggs": {
                "keywords": {
                    "significant_terms": {
                        "field": "tags",
                        "exclude": ["kibana", "javascript"]
                    }
                }
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "sample": {
            "doc_count": 200,
            "keywords": {
                "doc_count": 200,
                "bg_count": 650,
                "buckets": [
                    {
                        "key": "elasticsearch",
                        "doc_count": 150,
                        "score": 1.078125,
                        "bg_count": 200
                    },
                    {
                        "key": "logstash",
                        "doc_count": 50,
                        "score": 0.5625,
                        "bg_count": 50
                    }
                ]
            }
        }
    }
}
```

如果没有sampler聚合，请求查询会考虑低质量匹配的完整“长尾”，因此识别不太重要的terms，如jquery和angular，而不是关注更具洞察力的Kibana-related terms。

```
POST /stackoverflow/_search?size=0
{
    "query": {
        "query_string": {
            "query": "tags:kibana OR tags:javascript"
        }
    },
    "aggs": {
             "low_quality_keywords": {
                "significant_terms": {
                    "field": "tags",
                    "size": 3,
                    "exclude":["kibana", "javascript"]
                }
```

```
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "low_quality_keywords": {
            "doc_count": 600,
            "bg_count": 650,
            "buckets": [
                {
                    "key": "angular",
                    "doc_count": 200,
                    "score": 0.02777,
                    "bg_count": 200
                },
                {
                    "key": "jquery",
                    "doc_count": 200,
                    "score": 0.02777,
                    "bg_count": 200
                },
                {
                    "key": "logstash",
                    "doc_count": 50,
                    "score": 0.0069,
                    "bg_count": 50
                }
            ]
        }
    }
}
```

shard_size

shard_size参数限制在每个shard上处理的样本中收集多少得分最高的文档。默认值为100。

局限性

不能嵌套在breadth_first aggregations下

作为基于质量的过滤器，采样器聚合需要访问为每个文档生成的相关性得分。因此，它不能嵌套在术语聚合下，该术语聚合将collect_mode从默认depth_first模式切换为breadth_firs模式，因为这会丢弃分数。在这种情况下，将抛出错误。

多样化采样器聚集 diversified_sampler

Diversified Sampler Aggregation

与sampler aggregation类似，这是一种 filtering aggregation，用于将任何子聚合的处理限制为得分最高的文档样本。diversified_sampler聚合增加了限制共享共同值（如“author”）的匹配数量的能力。

示例用例：

- 将分析的重点集中在高相关性匹配上，而不是低质量匹配的潜在长尾
- 通过确保不同来源内容的公平表示，消除分析中的偏见
- 降低聚合的运行成本，这些聚合仅使用样本（例如significant_terms）即可产生有用的结果

字段或脚本设置的选择用于提供用于重复数据消除的值，max_docs_per_value设置控制在共享一个公共值的任何一个shard上收集的文档的最大数量。max_docs_per_value的默认设置为1。

如果选择字段或脚本为单个文档生成多个值，则聚合将引发错误（由于效率问题，不支持使用多值字段进行重复数据消除）。

例子：

我们可能想看看哪些标签与StackOverflow论坛帖子上的#elasticsearch密切相关，但忽略了一些高产用户的影响，他们倾向于将#Kibana拼写为#Cabana。

```
POST /stackoverflow/_search?size=0
{
    "query": {
        "query_string": {
            "query": "tags:elasticsearch"
        }
    },
    "aggs": {
        "my_unbiased_sample": {
            "diversified_sampler": {
                "shard_size": 200,
                "field" : "author"
            },
            "aggs": {
```

```
                "keywords": {
                    "significant_terms": {
                        "field": "tags",
                        "exclude": ["elasticsearch"]
                    }
                }
            }
        }
    }
```

返回

```
{
    ...
    "aggregations": {
        "my_unbiased_sample": {
            "doc_count": 151,
            "keywords": {
                "doc_count": 151,
                "bg_count": 650,
                "buckets": [
                    {
                        "key": "kibana",
                        "doc_count": 150,
                        "score": 2.213,
                        "bg_count": 200
                    }
                ]
            }
        }
    }
}
```

1. 总共对151个文档进行了抽样.
2. significant_terms聚合的结果不会因任何一位作者的怪癖而扭曲，因为我们要求样本中任何一位作家最多发表一篇文章

Scripted example:

在这种情况下，我们可能希望在字段值的组合上实现多样化。我们可以使用脚本生成标记字段中多个值的哈希，以确保我们没有由相同重复的标记组合组成的样本。

```
POST /stackoverflow/_search?size=0
{
    "query": {
        "query_string": {
            "query": "tags:kibana"
        }
    },
    "aggs": {
        "my_unbiased_sample": {
            "diversified_sampler": {
                "shard_size": 200,
                "max_docs_per_value" : 3,
                "script" : {
                    "lang": "painless",
                    "source": "doc['tags'].hashCode()"
                }
            },
            "aggs": {
                "keywords": {
                    "significant_terms": {
                        "field": "tags",
                        "exclude": ["kibana"]
                    }
                }
            }
        }
    }
}
```

返回

```
{
    ...
    "aggregations": {
        "my_unbiased_sample": {
            "doc_count": 6,
            "keywords": {
                "doc_count": 6,
                "bg_count": 650,
                "buckets": [
                    {
                        "key": "logstash",
                        "doc_count": 3,
                        "score": 2.213,
                        "bg_count": 50
                    },
```

```
                    {
                        "key": "elasticsearch",
                        "doc_count": 3,
                        "score": 1.34,
                        "bg_count": 200
                    }
                ]
            }
        }
    }
}
```

shard_size

shard_size参数限制在每个shard上处理的样本中收集多少得分最高的文档。默认值为100。

max_docs_per_value

max_docs_per_value是一个可选参数，它限制每次选择重复数据消除值时允许的文档数。默认设置为“1”。

execution_hint

可选的execution_int设置会影响用于重复数据消除的值的管理。在执行重复数据消除时，每个选项最多可在内存中保存shard_size值，但保存的值类型可按如下方式控制：

- 直接保存字段值（map）
- 保存Lucene索引确定的字段序号（global_ordinals）
- 保存字段值的哈希值-可能发生哈希冲突（byteshash）

如果Lucene索引中有此信息，则默认设置为使用global_ordinals，如果没有，则返回到map。在某些情况下，bytes_hash设置可能会更快，但由于哈希冲突的可能性，在重复数据消除逻辑中引入了误报的可能性。请注意，如果执行提示不适用，Elasticsearch将忽略执行提示的选择，并且这些提示没有向后兼容性保证。

局限性

不能嵌套在breadth_first聚合下

作为一个基于质量的过滤器，diversified_sampler聚合需要访问为每个文档生成的相关性得分。因此，它不能嵌套在terms aggregation下，terms aggregation将collect_mode从默认depth_first模式切换为breadth_firs模式，因为这会丢弃分数。在这种情况下，将抛出错误。

有限的重复数据消除逻辑。

重复数据消除逻辑仅适用于shard级别，因此不会跨shard应用。

geo/date字段没有专门的语法

目前，定义多样化值的语法是通过选择字段或脚本来定义的-对于表示地理或日期单位，如“7d”（7天），没有添加语法糖。此支持可能会在以后的版本中添加，用户当前必须使用脚本创建这些类型的值。

邻接矩阵聚合 adjacency_matrix

Adjacency Matrix Aggregation

一个返回邻接矩阵形式的桶聚合，该请求提供一个名为filter表达式的集合，类似于filters聚合请求，响应中的每个桶表示交叉滤波器矩阵中的非空单元格。

给定名为A、B和C的filters，响应将返回具有以下名称的桶：

	A	B	C
A	A	A&B	A&C
B		B	B&C
C			C

使用由&号字符分隔的两个过滤器名称的组合来标记相交的桶例如A&C。

如果客户端希望使用除＆符号（默认值）的以外的分隔符字符串，则可以在请求中传递`separator`替换。

例子

```
PUT /emails/_doc/_bulk?refresh
{ "index" : { "_id" : 1 } }
{ "accounts" : ["hillary", "sidney"]}
{ "index" : { "_id" : 2 } }
{ "accounts" : ["hillary", "donald"]}
{ "index" : { "_id" : 3 } }
{ "accounts" : ["vladimir", "donald"]}

GET emails/_search
{
  "size": 0,
  "aggs" : {
    "interactions" : {
      "adjacency_matrix" : {
```

```
        "filters" : {
          "grpA" : { "terms" : { "accounts" : ["hillary", "sidney"] }},
          "grpB" : { "terms" : { "accounts" : ["donald", "mitt"] }},
          "grpC" : { "terms" : { "accounts" : ["vladimir", "nigel"] }}
        }
      }
    }
  }
}
```

在上面的示例中，我们分析电子邮件消息，以查看哪些个人组交换了消息。我们将分别获得每个组的计数，以及记录了交互的成对组的消息计数。

响应：

```
{
  "took": 9,
  "timed_out": false,
  "_shards": ...,
  "hits": ...,
  "aggregations": {
    "interactions": {
      "buckets": [
        {
          "key":"grpA",
          "doc_count": 2
        },
        {
          "key":"grpA&grpB",
          "doc_count": 1
        },
        {
          "key":"grpB",
          "doc_count": 2
        },
        {
          "key":"grpB&grpC",
          "doc_count": 1
        },
        {
          "key":"grpC",
          "doc_count": 1
        }
      ]
    }
  }
}
```

使用

这种聚合本身可以提供创建无向加权图所需的所有数据。然而，当与子聚合（如date_histogram）一起使用时，结果可以提供执行动态网络分析所需的额外级别的数据，在动态网络分析中，随着时间的推移检查交互变得重要。

局限性

对于N个过滤器，生成的桶矩阵可以是$N^2/2$，因此默认最大值为100个过滤器。可以使用索引更改此设置。max_ajacency_matrix_filters索引级别设置。

PipelineAgg介绍

Pipeline Aggregation Overvew

管道聚合：让上一步聚合的结果作为下一个聚合的输入，类似stream流的操作，有许多不同类型的管道聚合，每个聚合计算的信息与其他聚合不同，但这些类型可以分为两类：

Parent (父级)

父级聚合的输出提供了一组管道聚合，它可以计算新的存储桶或新的聚合以添加到现有存储桶中

Sibling（同级）

同级聚合的输出提供管道，并且能够计算与该同级聚合处于同一级别的新聚合。

> **INFO**
>
> 由于管道聚合仅添加到输出中，因此在链接管道聚合时，每个管道聚合的输出都将包含在最终输出中

buckets_path 语法

大多数管道聚合需要另一个聚合作为其输入。输入聚合通过buckets_path参数定义，该参数遵循特定格式：

```
AGG_SEPARATOR       =  '>' ;
METRIC_SEPARATOR    =  '.' ;
AGG_NAME            =  <the name of the aggregation> ;
METRIC              =  <the name of the metric (in case of multi-value metrics
aggregation)> ;
PATH                =  <AGG_NAME> [ <AGG_SEPARATOR>, <AGG_NAME> ]* [
<METRIC_SEPARATOR>, <METRIC> ] ;
```

例如，路径“my_bucket>my_stats.avg”将指向“my_stats”metric中的avg值，该metric包含在“my_bBucket”桶聚合中。

路径与管道聚合的位置是相对的；它们不是绝对路径，路径不能返回到聚合树的“上方”。例如，此移动平均值嵌入到date_histogram中，并引用`sibling` metric `the_sum`：

```
POST /_search
{
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"timestamp",
                "interval":"day"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "lemmings" }
                },
                "the_movavg":{
                    "moving_avg":{ "buckets_path": "the_sum" }
                }
            }
        }
    }
}
```

1. metric为“the_sum”
2. buckets_path通过相对路径“the_sum”引用metric

buckets_path也用于Sibling管道聚合，其中聚合是一系列桶的“next”，而不是嵌入在它内部。例如，max_bucket聚合使用buckets_path指定嵌入同级聚合中的metric：

```
POST /_search
{
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "sales": {
                    "sum": {
                        "field": "price"
                    }
                }
            }
        },
```

```
        "max_monthly_sales": {
            "max_bucket": {
                "buckets_path": "sales_per_month>sales"
            }
        }
    }
}
```

bucket_path指示此max_bucket聚合，我们希望取出sales_permonth日期直方图中的销售聚合的最大值。

Special Paths

buckets_path可以使用一个特殊的"_count"路径，而不是指向metric。这指示管道聚合使用文档计数作为其输入。例如，可以根据每个bucket的文档计数计算移动平均值。

```
POST /_search
{
    "aggs": {
        "my_date_histo": {
            "date_histogram": {
                "field":"timestamp",
                "interval":"day"
            },
            "aggs": {
                "the_movavg": {
                    "moving_avg": { "buckets_path": "_count" }
                }
            }
        }
    }
}
```

通过使用_count而不是metric名称，我们可以计算直方图中文档计数的移动平均值

buckets_path还可以使用"_bucket_count"和多桶聚合的路径来使用该聚合在管道聚合中返回的桶数，而不是metric。例如，这里可以使用bucket_selector来过滤不包含用于内部项聚合的桶的桶：

```
POST /sales/_search
{
  "size": 0,
  "aggs": {
    "histo": {
      "date_histogram": {
```

```
        "field": "date",
        "interval": "day"
      },
      "aggs": {
        "categories": {
          "terms": {
            "field": "category"
          }
        },
        "min_bucket_selector": {
          "bucket_selector": {
            "buckets_path": {
              "count": "categories._bucket_count"
            },
            "script": {
              "source": "params.count != 0"
            }
          }
        }
      }
    }
  }
```

通过使用_bucket_count而不是metric名称，我们可以过滤掉不包含类别聚合桶的历史桶

Dealing with dots in agg names

支持另一种语法来处理名称中有点的聚合或metric，例如99.9%。该指标可称为：

```
"buckets_path": "my_percentile[99.9]"
```

Dealing with gaps in the data

现实世界中的数据常常是嘈杂的，有时还存在差距—数据根本不存在的地方。出现这种情况的原因多种多样，最常见的是：

- 落入bucket的文档不包含必填字段
- 没有与一个或多个bucket的查询匹配的文档
- 正在计算的metric值无法生成值，可能是因为另一个从属bucket缺少值。某些管道聚合具有必须满足的特定要求（例如，导数无法计算第一个值的metric，因为没有以前的值，HoltWinters移动平均值需

要“预热”数据才能开始计算，等等）

Gap策略是一种机制，用于在遇到“gap”或丢失数据时通知管道聚合所需的行为。所有管道聚合都接受gap_policy参数。目前有两种差距政策可供选择：

skip

此选项将丢失的数据视为bucket不存在。它将跳过桶并使用下一个可用值继续计算。

insert_zeros

此选项将用零（0）替换缺失的值，管道聚合计算将照常进行。

管道桶聚合 bucket

语法

Avg Bucket Aggregation

平均值桶聚合

```
{
    "avg_bucket": {
        "buckets_path": "the_sum"
    }
}
```

Max Bucket Aggregation

最大值桶聚合

```
{
    "max_bucket": {
        "buckets_path": "the_sum"
    }
}
```

Min Bucket Aggregation

最小值桶聚合

```
{
    "min_bucket": {
        "buckets_path": "the_sum"
    }
}
```

Sum Bucket Aggregation

总和桶聚合

```
{
    "sum_bucket": {
        "buckets_path": "the_sum"
    }
}
```

Stats Bucket Aggregation

统计桶聚合

```
{
    "stats_bucket": {
        "buckets_path": "the_sum"
    }
}
```

Extended Stats Bucket Aggregation

扩展统计桶聚合，与stats_bucket聚合相比，该聚合提供了更多的统计信息（平方和、标准偏差等）。

```
{
    "extended_stats_bucket": {
        "buckets_path": "the_sum"
    }
}
```

Percentiles Bucket Aggregation

百分比桶聚合

```
{
    "percentiles_bucket": {
        "buckets_path": "the_sum"
    }
}
```

`xxx_bucket` 参数定义

参数名	描述	是否必须	默认值
buckets_path	桶的路径	必选	
gap_policy	在数据中发现差距时适用的策略	可选	skip
format	应用于此聚合的输出值的格式	可选	null

示例部分

avg_bucket

以下代码段计算每月总销售额的平均值：

```
POST /_search
{
  "size": 0,
  "aggs": {
    "sales_per_month": {
      "date_histogram": {
        "field": "date",
        "interval": "month"
      },
      "aggs": {
        "sales": {
          "sum": {
            "field": "price"
          }
        }
      }
    },
    "avg_monthly_sales": {
      "avg_bucket": {
        "buckets_path": "sales_per_month>sales"
      }
    }
  }
}
```

bucket_path指示这个avg_bucket聚合，我们希望在sales_permonth日期直方图中获得销售聚合的平均值。

返回


```
{
   "took": 11,
   "timed_out": false,
   "_shards": ...,
   "hits": ...,
   "aggregations": {
      "sales_per_month": {
         "buckets": [
            {
               "key_as_string": "2015/01/01 00:00:00",
               "key": 1420070400000,
               "doc_count": 3,
               "sales": {
                  "value": 550.0
               }
            },
            {
               "key_as_string": "2015/02/01 00:00:00",
               "key": 1422748800000,
               "doc_count": 2,
               "sales": {
                  "value": 60.0
               }
            },
            {
               "key_as_string": "2015/03/01 00:00:00",
               "key": 1425168000000,
               "doc_count": 2,
               "sales": {
                  "value": 375.0
               }
            }
         ]
      },
      "avg_monthly_sales": {
          "value": 328.33333333333333
      }
   }
}
```


stats_bucket

以下代码段计算每月销售额的统计数据：

```
POST /sales/_search
{
    "size": 0,
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "sales": {
                    "sum": {
                        "field": "price"
                    }
                }
            }
        },
        "stats_monthly_sales": {
            "stats_bucket": {
                "buckets_path": "sales_per_month>sales"
            }
        }
    }
}
```

返回

```
{
   "took": 11,
   "timed_out": false,
   "_shards": ...,
   "hits": ...,
   "aggregations": {
      "sales_per_month": {
         "buckets": [
            {
               "key_as_string": "2015/01/01 00:00:00",
               "key": 1420070400000,
               "doc_count": 3,
               "sales": {
                  "value": 550.0
               }
            },
            {
               "key_as_string": "2015/02/01 00:00:00",
               "key": 1422748800000,
```

```
               "doc_count": 2,
               "sales": {
                  "value": 60.0
               }
            },
            {
               "key_as_string": "2015/03/01 00:00:00",
               "key": 1425168000000,
               "doc_count": 2,
               "sales": {
                  "value": 375.0
               }
            }
         ]
      },
      "stats_monthly_sales": {
         "count": 3,
         "min": 60.0,
         "max": 550.0,
         "avg": 328.3333333333333,
         "sum": 985.0
      }
   }
}
```

extended_stats_bucket

下面的代码段计算了每月销售额桶的扩展统计数据：

```
POST /sales/_search
{
    "size": 0,
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "sales": {
                    "sum": {
                        "field": "price"
                    }
                }
            }
        },
```

```
        "stats_monthly_sales": {
            "extended_stats_bucket": {
                "buckets_path": "sales_per_month>sales"
            }
        }
    }
}
```

返回

```
{
   "took": 11,
   "timed_out": false,
   "_shards": ...,
   "hits": ...,
   "aggregations": {
      "sales_per_month": {
         "buckets": [
            {
               "key_as_string": "2015/01/01 00:00:00",
               "key": 1420070400000,
               "doc_count": 3,
               "sales": {
                  "value": 550.0
               }
            },
            {
               "key_as_string": "2015/02/01 00:00:00",
               "key": 1422748800000,
               "doc_count": 2,
               "sales": {
                  "value": 60.0
               }
            },
            {
               "key_as_string": "2015/03/01 00:00:00",
               "key": 1425168000000,
               "doc_count": 2,
               "sales": {
                  "value": 375.0
               }
            }
         ]
      },
      "stats_monthly_sales": {
         "count": 3,
         "min": 60.0,
```

```
          "max": 550.0,
          "avg": 328.3333333333333,
          "sum": 985.0,
          "sum_of_squares": 446725.0,
          "variance": 41105.55555555556,
          "std_deviation": 202.74505063146563,
          "std_deviation_bounds": {
            "upper": 733.8234345962646,
            "lower": -77.15676792959795
          }
        }
    }
}
```

percentiles_bucket

以下代码段计算每月总销售时段的百分比：

```
POST /sales/_search
{
    "size": 0,
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "sales": {
                    "sum": {
                        "field": "price"
                    }
                }
            }
        },
        "percentiles_monthly_sales": {
            "percentiles_bucket": {
                "buckets_path": "sales_per_month>sales",
                "percents": [ 25.0, 50.0, 75.0 ]
            }
        }
    }
}
```

管道桶排序 bucket_sort

Bucket Sort Aggregation

属于parent管道聚合，对其父多桶聚合的桶进行排序。可以指定零个或多个排序字段以及相应的排序顺序。每个桶可以根据其_key、_count或其子聚合进行排序。此外，可以设置from和size参数，以便截断结果桶。

语法

`bucket_sort` aggregation

```
{
    "bucket_sort": {
        "sort": [
            {"sort_field_1": {"order": "asc"}},
            {"sort_field_2": {"order": "desc"}},
            "sort_field_3"
        ],
        "from": 1,
        "size": 3
    }
}
```

bucket_sort 参数

参数	说明	缺省	默认值
sort	要排序的字段列表	可选	
from	位置前的桶将被截断	可选	0
size	要返回的桶数，默认为父聚合的所有桶。	可选	
gap_policy	在数据中发现差距时应用的策略	可选	skip

以下代码段按降序返回与总销售额最高的3个月相对应的存储桶：

```
POST /sales/_search
{
    "size": 0,
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "total_sales": {
                    "sum": {
                        "field": "price"
                    }
                },
                "sales_bucket_sort": {
                    "bucket_sort": {
                        "sort": [
                          {"total_sales": {"order": "desc"}}
                        ],
                        "size": 3
                    }
                }
            }
        }
    }
}
```

返回

```
   "took": 82,
   "timed_out": false,
   "_shards": ...,
   "hits": ...,
   "aggregations": {
      "sales_per_month": {
         "buckets": [
            {
               "key_as_string": "2015/01/01 00:00:00",
               "key": 1420070400000,
               "doc_count": 3,
               "total_sales": {
                  "value": 550.0
```

```
            }
         },
         {
            "key_as_string": "2015/03/01 00:00:00",
            "key": 1425168000000,
            "doc_count": 2,
            "total_sales": {
               "value": 375.0
            },
         },
         {
            "key_as_string": "2015/02/01 00:00:00",
            "key": 1422748800000,
            "doc_count": 2,
            "total_sales": {
               "value": 60.0
            },
         }
      ]
   }
}
```

截断而不排序

也可以使用此聚合来截断结果桶而不进行任何排序。为此，只需使用from和/或size参数而不指定排序。

下面的示例简单地截断结果，以便只返回第二个桶：

```
POST /sales/_search
{
    "size": 0,
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "bucket_truncate": {
                    "bucket_sort": {
                        "from": 1,
                        "size": 1
                    }
                }
```

```
            }
        }
    }
}
```

返回

```
{
   "took": 11,
   "timed_out": false,
   "_shards": ...,
   "hits": ...,
   "aggregations": {
      "sales_per_month": {
         "buckets": [
            {
               "key_as_string": "2015/02/01 00:00:00",
               "key": 1422748800000,
               "doc_count": 2
            }
         ]
      }
   }
}
```

管道桶选择器 bucket_selector

Bucket Selector Aggregation

属于parent管道聚合，它执行一个脚本，该脚本确定当前桶是否将保留在父多桶聚合中。指定的度量必须是数字，并且脚本必须返回布尔值。如果脚本语言是表达式，则允许数字返回值。在这种情况下，0.0将被计算为`false`，所有其他值将被计算成`true`。

语法

`bucket_selector` aggregation

```
{
    "bucket_selector": {
        "buckets_path": {
            "my_var1": "the_sum",
            "my_var2": "the_value_count"
        },
        "script": "params.my_var1 > params.my_var2"
    }
}
```

bucket_selector 参数

参数	说明	缺省	默认值
script	要为此聚合运行的脚本	必填	
buckets_path	脚本变量及其关联路径到我们希望用于变量的Bucket的映射	必填	
gap_policy	在数据中发现差距时应用的策略	可选	skip

以下片段仅保留当月总销售额超过200的桶：

```
POST /sales/_search
{
```

```
    "size": 0,
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "total_sales": {
                    "sum": {
                        "field": "price"
                    }
                },
                "sales_bucket_filter": {
                    "bucket_selector": {
                        "buckets_path": {
                          "totalSales": "total_sales"
                        },
                        "script": "params.totalSales > 200"
                    }
                }
            }
        }
    }
}
```

返回

```
{
   "took": 11,
   "timed_out": false,
   "_shards": ...,
   "hits": ...,
   "aggregations": {
      "sales_per_month": {
         "buckets": [
            {
               "key_as_string": "2015/01/01 00:00:00",
               "key": 1420070400000,
               "doc_count": 3,
               "total_sales": {
                  "value": 550.0
               }
            },
            {
               "key_as_string": "2015/03/01 00:00:00",
               "key": 1425168000000,
```

```
                "doc_count": 2,
                "total_sales": {
                    "value": 375.0
                },
            }
        ]
    }
  }
}
```

导数聚合 derivative

Derivative Aggregation

父管道聚合，用于计算父直方图（或date_histogram）聚合中指定度量的导数。指定的度量必须是数字，封闭直方图的min_doc_count必须设置为0（直方图聚合的默认值）。

语法

```
"derivative": {
  "buckets_path": "the_sum"
}
```

`derivative` 参数定义

参数名	描述	是否必须	默认值
buckets_path	我们希望为其查找派生的Bucket的路径	必选	
gap_policy	在数据中发现差距时应用的策略	可选	skip
format	应用于此聚合的输出值的格式	可选	null

First Order Derivative

以下代码段计算每月总销售额的导数：

```
POST /sales/_search
{
    "size": 0,
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
```

```
            },
            "aggs": {
                "sales": {
                    "sum": {
                        "field": "price"
                    }
                },
                "sales_deriv": {
                    "derivative": {
                        "buckets_path": "sales"
                    }
                }
            }
        }
    }
}
```

`buckets_path` 指定了 `derivative` 为`sales aggregation`的输出的派生

返回

```
{
   "took": 11,
   "timed_out": false,
   "_shards": ...,
   "hits": ...,
   "aggregations": {
      "sales_per_month": {
         "buckets": [
            {
               "key_as_string": "2015/01/01 00:00:00",
               "key": 1420070400000,
               "doc_count": 3,
               "sales": {
                  "value": 550.0
               }
            },
            {
               "key_as_string": "2015/02/01 00:00:00",
               "key": 1422748800000,
               "doc_count": 2,
               "sales": {
                  "value": 60.0
               },
               "sales_deriv": {
                  "value": -490.0
               }
```

```
            },
            {
               "key_as_string": "2015/03/01 00:00:00",
               "key": 1425168000000,
               "doc_count": 2,
               "sales": {
                  "value": 375.0
               },
               "sales_deriv": {
                  "value": 315.0
               }
            }
         ]
      }
   }
}
```

Second Order Derivative

Second Derivative可以通过 derivative agg 链接到另一个 derivative agg 的结果上来计算，如以下示例所示，该示例将计算每月总销售额的一阶导数和二阶导数：

```
POST /sales/_search
{
    "size": 0,
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "sales": {
                    "sum": {
                        "field": "price"
                    }
                },
                "sales_deriv": {
                    "derivative": {
                        "buckets_path": "sales"
                    }
                },
                "sales_2nd_deriv": {
                    "derivative": {
                        "buckets_path": "sales_deriv"
```

```
                    }
                }
            }
        }
    }
}
```

二阶导数的`buckets_path`指向一阶导数的名称

返回

```
{
   "took": 50,
   "timed_out": false,
   "_shards": ...,
   "hits": ...,
   "aggregations": {
      "sales_per_month": {
         "buckets": [
            {
               "key_as_string": "2015/01/01 00:00:00",
               "key": 1420070400000,
               "doc_count": 3,
               "sales": {
                  "value": 550.0
               }
            },
            {
               "key_as_string": "2015/02/01 00:00:00",
               "key": 1422748800000,
               "doc_count": 2,
               "sales": {
                  "value": 60.0
               },
               "sales_deriv": {
                  "value": -490.0
               }
            },
            {
               "key_as_string": "2015/03/01 00:00:00",
               "key": 1425168000000,
               "doc_count": 2,
               "sales": {
                  "value": 375.0
               },
               "sales_deriv": {
                  "value": 315.0
```

```
                },
                "sales_2nd_deriv": {
                   "value": 805.0
                }
             }
          ]
       }
    }
}
```

单位

导数聚合允许指定导数值的单位。这将在响应normalized_value中返回一个额外字段，该字段以所需的x轴单位报告导数值。在下例中，我们计算每月总销售额的导数，但要求以每天的销售额为单位计算销售额的导数：

```
POST /sales/_search
{
    "size": 0,
    "aggs" : {
        "sales_per_month" : {
            "date_histogram" : {
                "field" : "date",
                "interval" : "month"
            },
            "aggs": {
                "sales": {
                    "sum": {
                        "field": "price"
                    }
                },
                "sales_deriv": {
                    "derivative": {
                        "buckets_path": "sales",
                        "unit": "day"
                    }
                }
            }
        }
    }
}
```

`unit`指定用于导数计算的x轴的单位

返回


```
{
   "took": 50,
   "timed_out": false,
   "_shards": ...,
   "hits": ...,
   "aggregations": {
      "sales_per_month": {
         "buckets": [
            {
               "key_as_string": "2015/01/01 00:00:00",
               "key": 1420070400000,
               "doc_count": 3,
               "sales": {
                  "value": 550.0
               }
            },
            {
               "key_as_string": "2015/02/01 00:00:00",
               "key": 1422748800000,
               "doc_count": 2,
               "sales": {
                  "value": 60.0
               },
               "sales_deriv": {
                  "value": -490.0,
                  "normalized_value": -15.806451612903226
               }
            },
            {
               "key_as_string": "2015/03/01 00:00:00",
               "key": 1425168000000,
               "doc_count": 2,
               "sales": {
                  "value": 375.0
               },
               "sales_deriv": {
                  "value": 315.0,
                  "normalized_value": 11.25
               }
            }
         ]
      }
   }
}
```

1. `value`以每月的原始单位报告
2. `normalized_value`以每天所需的单位报告

时序差分聚合 serial_diff

Serial Differencing Aggregation

Serial Diff 的公式

$$f(x) = f(x_t) - f(x_{t-n})$$

> 其中n是使用的周期。

解释：用当前点减去上一个周期的对应点，然后得出一个新的时序图，周期为1相当于不对时间做归一化求导，它表示一个点到下一个点的变化。单周期差分聚合对移除常量和线性趋势是很有用的。

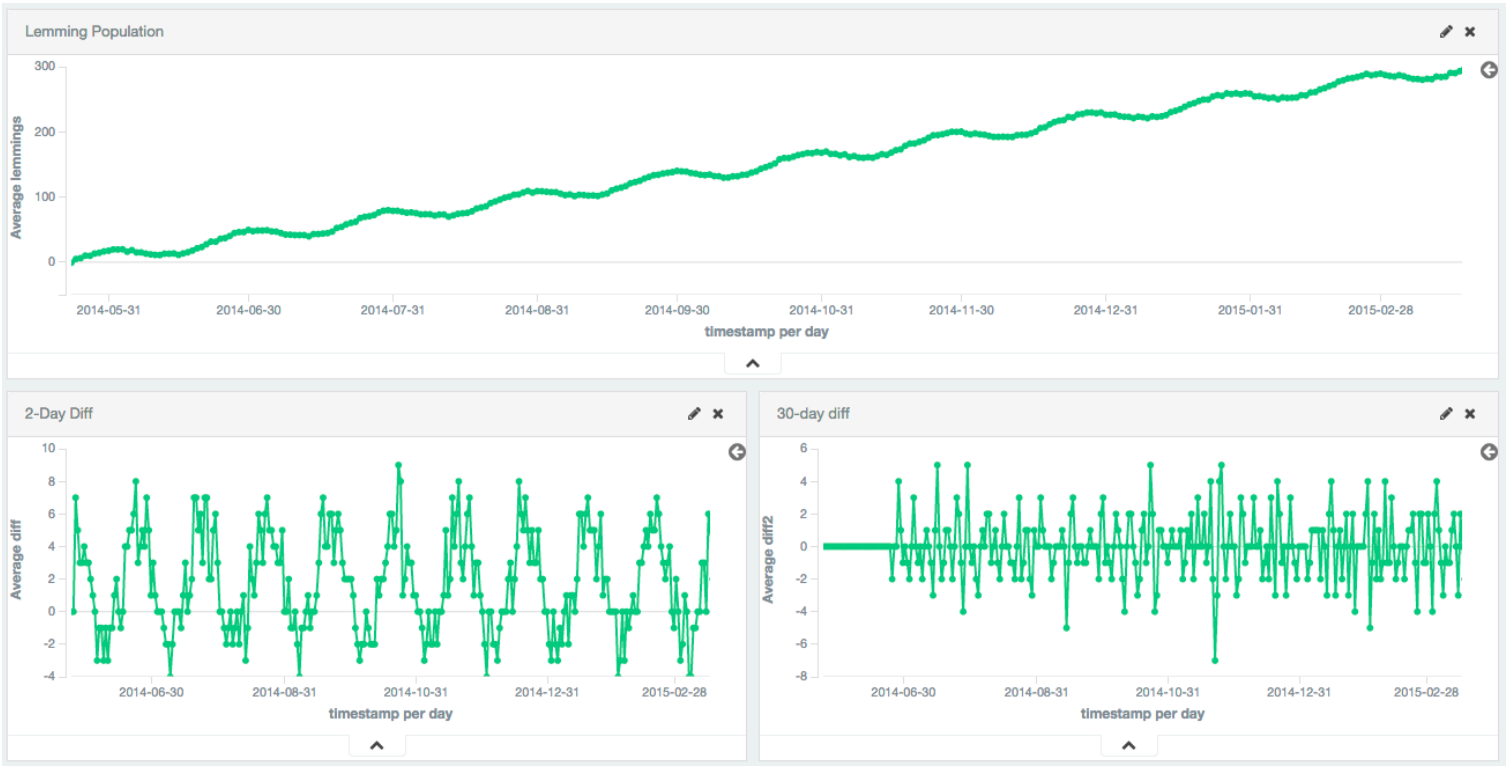
在本例中，通过计算第一个差异，我们对数据进行去趋势化（例如，去除恒定的线性趋势）。我们可以看到，数据变成了一个平稳的序列（例如，第一个差异随机分布在零附近，似乎没有表现出任何模式/行为）。转换表明数据集遵循随机游走；该值是先前的值+/-随机量。这种洞察力允许选择进一步的分析工具。

图15. 道琼斯指数用第一次差分绘制并固定

较大的周期可用于消除季节性/周期性行为。在这个例子中，旅鼠群是用正弦波+恒定线性趋势+随机噪声合成生成的。正弦波的周期为30天。

第一个差异消除了恒定趋势，只留下一个正弦波。然后将第30个差分应用于第一个差分以去除循环行为，留下一个可用于其他分析的平稳序列。

图16.用第1和第30个差值绘制的勒明斯数据



语法

```
{
    "serial_diff": {
        "buckets_path": "the_sum",
        "lag": "7"
    }
}
```

参数名	描述	是否必须	默认值
buckets_path	我们希望为其查找派生的Bucket的路径	必选	
lag	要从当前值中减去的历史存储段。	可选	1
gap_policy	在数据中发现差距时应用的策略	可选	skip
format	应用于此聚合的输出值的格式	可选	null

```
POST /_search
{
   "size": 0,
   "aggs": {
      "my_date_histo": {
         "date_histogram": {
            "field": "timestamp",
            "interval": "day"
         },
         "aggs": {
            "the_sum": {
               "sum": {
                  "field": "lemmings"
               }
            },
            "thirtieth_difference": {
               "serial_diff": {
                  "buckets_path": "the_sum",
                  "lag" : 30
               }
            }
         }
      }
   }
}
```

1. 在“timestamp”字段上构建名为“my_date_histo”的date_histogram，间隔为一天
2. 总和度量用于计算字段的总和。这可以是任何度量（总和、最小值、最大值等）
3. 最后，我们指定一个使用“the_sum”度量作为输入的serial_diff聚合。

通过首先在字段上指定直方图或日期直方图来构建序列差异。然后，您可以选择在直方图内添加常规度量，例如和。最后，serial_diff嵌入到直方图中。然后使用buckets_path参数“指向”直方图内的一个同级度量（有关buckets_path语法的描述，请参阅buckets_path语法

移动平均聚合 moving_avg

Moving Average Aggregation

给定一系列有序的数据，移动平均值聚合将在数据上滑动一个窗口，并显示该窗口的平均值。例如，给定数据[1，2，3，4，5，6，7，8，9，10]，我们可以计算窗口大小为5的简单移动平均值，如下所示：

- (1 + 2 + 3 + 4 + 5) / 5 = 3
- (2 + 3 + 4 + 5 + 6) / 5 = 4
- (3 + 4 + 5 + 6 + 7) / 5 = 5
- etc

`Moving averages` 是平滑连续数据的简单方法。移动平均值通常应用于基于时间的数据，如股价或服务器指标。平滑可用于消除高频波动或随机噪声，这使得较低频率的趋势更容易可视化，例如季节性。

语法

```
{
    "moving_avg": {
        "buckets_path": "the_sum",
        "model": "holt",
        "window": 5,
        "gap_policy": "insert_zeros",
        "settings": {
            "alpha": 0.8
        }
    }
}
```

`moving_avg` 参数

参数名	描述	是否必须	默认值
buckets_path	桶的路径	必选	

参数名	描述	是否必须	默认值
model	我们希望使用的移动平均加权模型	必选	simple
gap_policy	在数据中发现差距时应用的策略	可选	insert_zeros
window	在直方图上“滑动”的窗口大小。	可选	5
minimize	如果模型应在算法上最小化	可选	false
settings	特定于model的设置，内容因指定的model而异。	可选	

`movingavg agg`必须嵌入到`histogram`或`date_histogram` aggregation中。它们可以像任何其他`metric aggregation`一样嵌入：

```
OST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg":{
                    "moving_avg":{ "buckets_path": "the_sum" }
                }
            }
        }
    }
}
```

移动平均值是通过在字段上指定histogram或date_histogram来构建的 上述聚合的示例响应如下

```
{
   "took": 11,
   "timed_out": false,
   "_shards": ...,
```

```
    "hits": ...,
    "aggregations": {
       "my_date_histo": {
          "buckets": [
              {
                  "key_as_string": "2015/01/01 00:00:00",
                  "key": 1420070400000,
                  "doc_count": 3,
                  "the_sum": {
                     "value": 550.0
                  }
              },
              {
                  "key_as_string": "2015/02/01 00:00:00",
                  "key": 1422748800000,
                  "doc_count": 2,
                  "the_sum": {
                     "value": 60.0
                  },
                  "the_movavg": {
                     "value": 550.0
                  }
              },
              {
                  "key_as_string": "2015/03/01 00:00:00",
                  "key": 1425168000000,
                  "doc_count": 2,
                  "the_sum": {
                     "value": 375.0
                  },
                  "the_movavg": {
                     "value": 305.0
                  }
              }
          ]
       }
    }
}
```

Models

moving_avg聚合包括四个不同的移动平均“模型”。主要区别在于窗口中的值是如何加权的。随着数据点在窗口中变得“`older`”，它们的权重可能会有所不同。这将影响该窗口的最终平均值。

使用模型参数指定模型。某些`model`可能具有在`settings`参数中指定的可选配置。

Simple

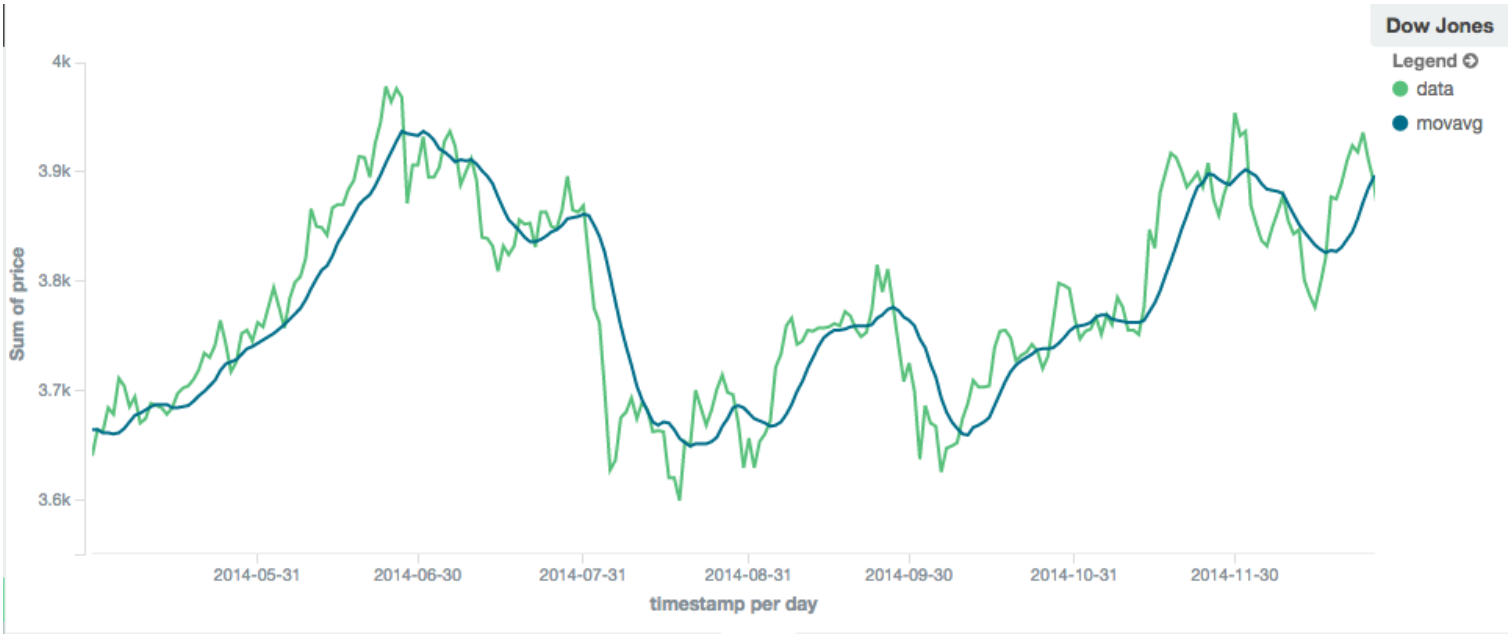
`simple model`计算窗口中所有值的总和，然后除以窗口的大小。它实际上是窗口的简单算术平均值。简单模型不执行任何时间相关加权，这意味着简单移动平均值的值往往“滞后”于实际数据。

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg":{
                    "moving_avg":{
                        "buckets_path": "the_sum",
                        "window" : 30,
                        "model" : "simple"
                    }
                }
            }
        }
    }
}
```

一个 `simple model` 的模型没有需要配置的特殊设置

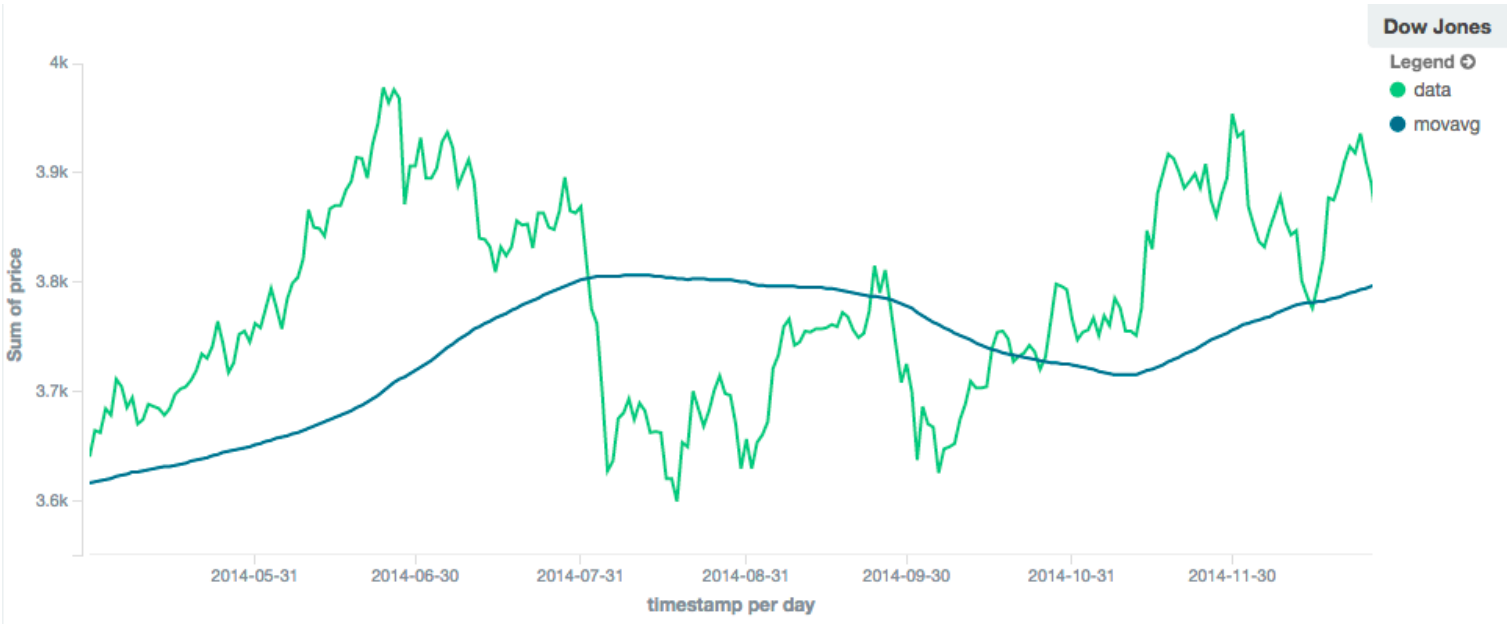
窗口大小可以改变移动平均线的行为。例如，一个小窗口（“窗口”：10）将密切跟踪数据，仅平滑小范围波动：

图1.窗口大小为10的移动平均线



相比之下，具有较大窗口（“窗口”：100）的`simple moving average`将平滑所有较高频率的波动，只留下低频的长期趋势。它也倾向于“滞后”于实际数据一大截：

图2.窗口大小为100的移动平均线



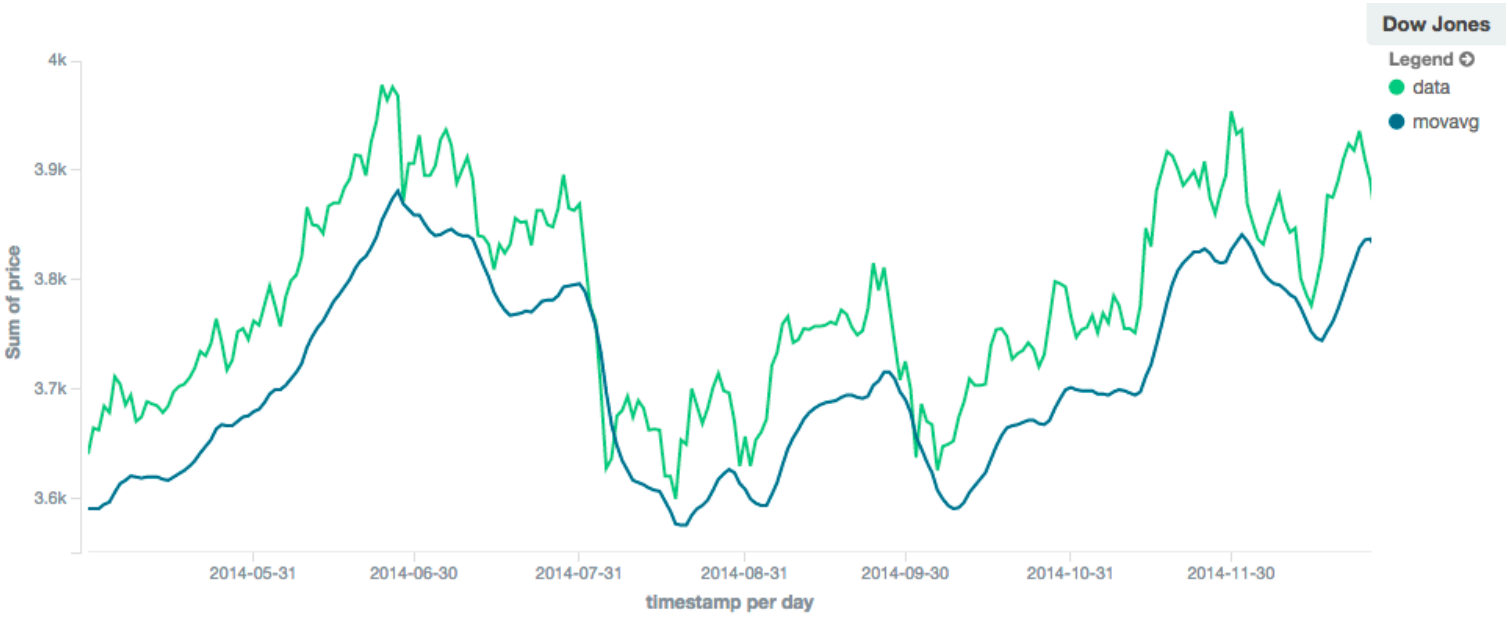
Linear

线性模型为序列中的点分配线性权重，使得“较旧”的数据点（例如，窗口开始处的数据点）对总平均值的贡献线性较小。线性加权有助于减少数据平均值之后的“滞后”，因为旧点的影响较小。

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg": {
                    "moving_avg":{
                        "buckets_path": "the_sum",
                        "window" : 30,
                        "model" : "linear"
                    }
                }
            }
        }
    }
}
```

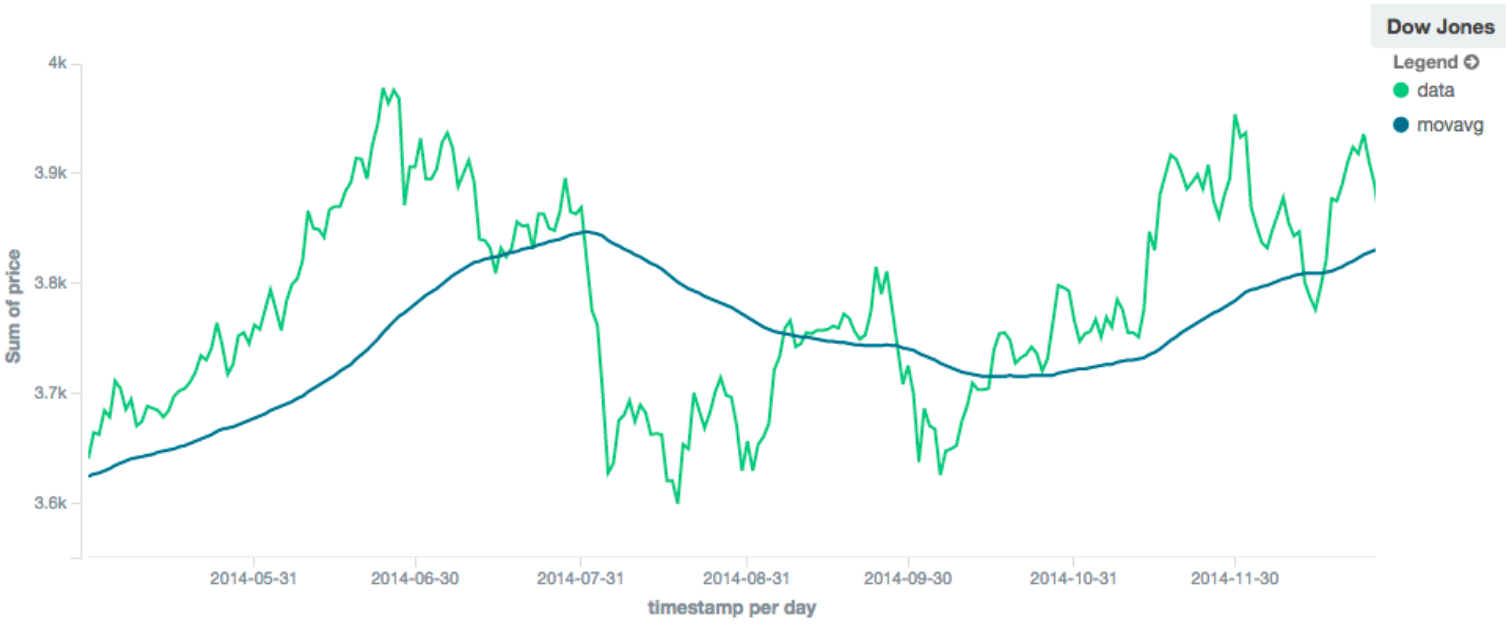
与简单模型一样，窗口大小可以改变移动平均线的行为。例如，一个小窗口（“窗口”：10）将密切跟踪数据，仅平滑小范围波动：

图3.窗口大小为10的线性移动平均值



相比之下，具有较大窗口（“窗口”：100）的线性移动平均线将平滑所有较高频率的波动，只留下低频的长期趋势。它也倾向于“滞后”于实际数据相当大的数量，尽管通常比简单模型要少：

图4.窗口大小为100的线性移动平均值



EWMA (Exponentially Weighted)

ewma模型（又称“单指数”）与线性模型相似，只是较旧的数据点变得不那么重要，而不是线性不那么重要。重要性衰减的速度可以通过alpha设置来控制。较小的值会使权重衰减缓慢，从而提供更大的平滑度，并考虑到窗口的较大部分。较大的值会使权重快速衰减，从而减少旧值对移动平均值的影响。这往往会使移动平均线更接近地跟踪数据，但平滑度较低。

alpha的默认值为0.3，该设置接受0-1（含0-1）之间的任何浮动。

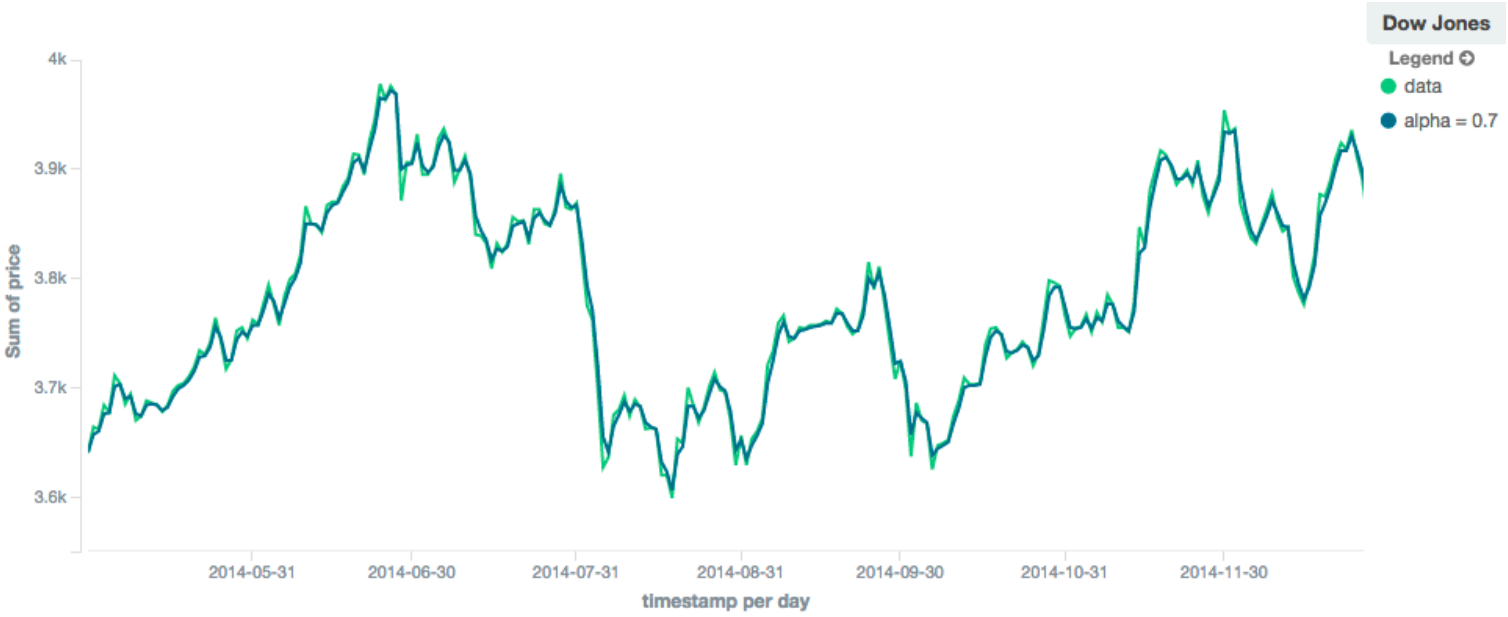
EWMA模型可以最小化

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
```

```
                    "sum":{ "field": "price" }
                },
                "the_movavg": {
                    "moving_avg":{
                        "buckets_path": "the_sum",
                        "window" : 30,
                        "model" : "ewma",
                        "settings" : {
                            "alpha" : 0.5
                        }
                    }
                }
            }
        }
    }
}
```

图5.EWMA，窗口大小为10，alpha=0.2

图6.EWMA，窗口大小为10，alpha=0.7



Holt-Linear

holt model （又名“双指数”）包含了跟踪数据趋势的第二个指数项。当数据具有潜在的线性趋势时，单指数表现不佳。双指数模型内部计算两个值：“level”和“trend”。

级别计算类似于ewma，是数据的指数加权视图。不同之处在于，使用了先前平滑的值而不是原始值，从而使其保持接近原始序列。趋势计算着眼于当前值和上次值之间的差异（例如，平滑数据的斜率或趋势）。趋势值也按指数加权。

值通过乘以水平和趋势分量产生。

alpha的默认值为0.3，beta为0.1。这些设置接受0-1（含0-1）之间的任何浮动。

Holt线性模型可以最小化

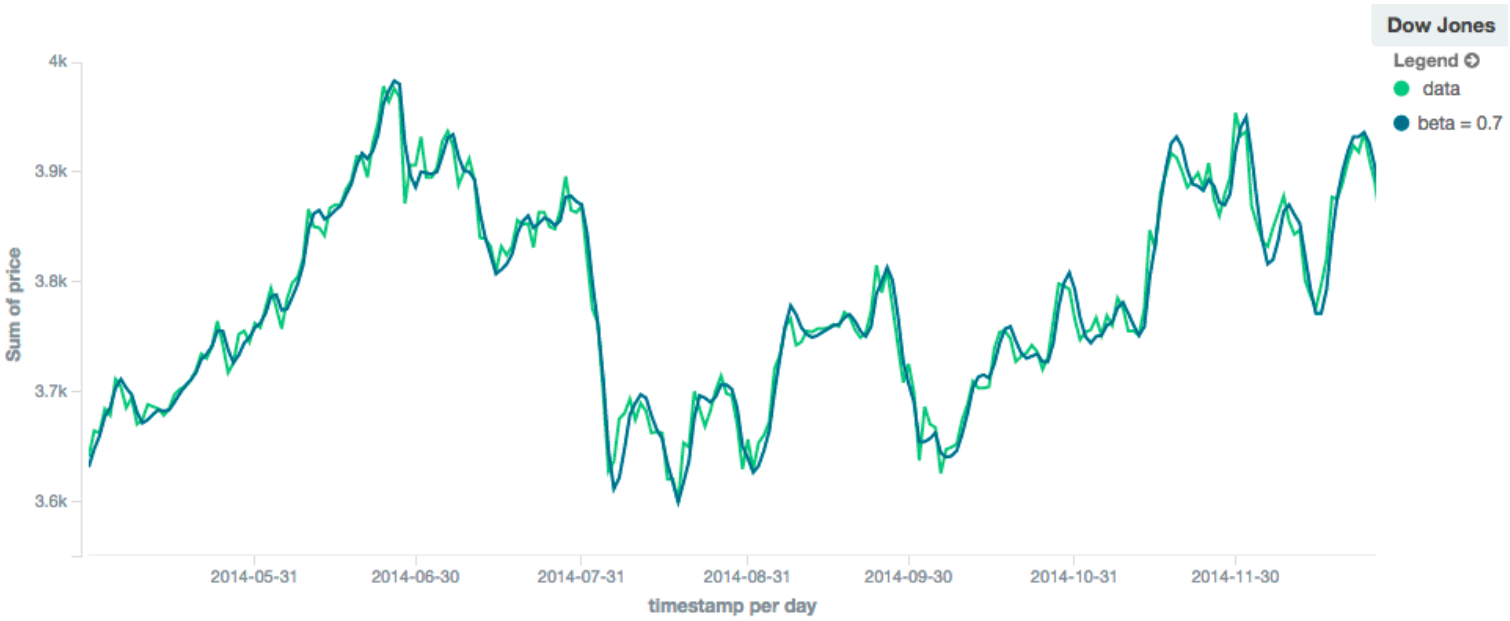
```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
```

```
                },
                "the_movavg": {
                    "moving_avg":{
                        "buckets_path": "the_sum",
                        "window" : 30,
                        "model" : "holt",
                        "settings" : {
                            "alpha" : 0.5,
                            "beta" : 0.5
                        }
                    }
                }
            }
        }
    }
}
```

实际上，alpha值在holt中的表现与ewma非常相似：较小的值产生更多的平滑和更多的滞后，而较大的值产生更紧密的跟踪和更少的滞后。beta的值通常很难看出。小值强调长期趋势（例如整个系列中的恒定线性趋势），而大值强调短期趋势。当您预测值时，这将变得更加明显。

图7.Holt线性移动平均值，窗口大小为100，alpha=0.5，beta=0.2

图8.Holt线性移动平均值，窗口大小为100，alpha=0.5，beta=0.7



Holt-Winters

holt_winters模型（又称“三指数”）包含第三个指数项，它跟踪数据的季节性方面。因此，这种聚合基于三个组成部分： "level", "trend"， "seasonality"。

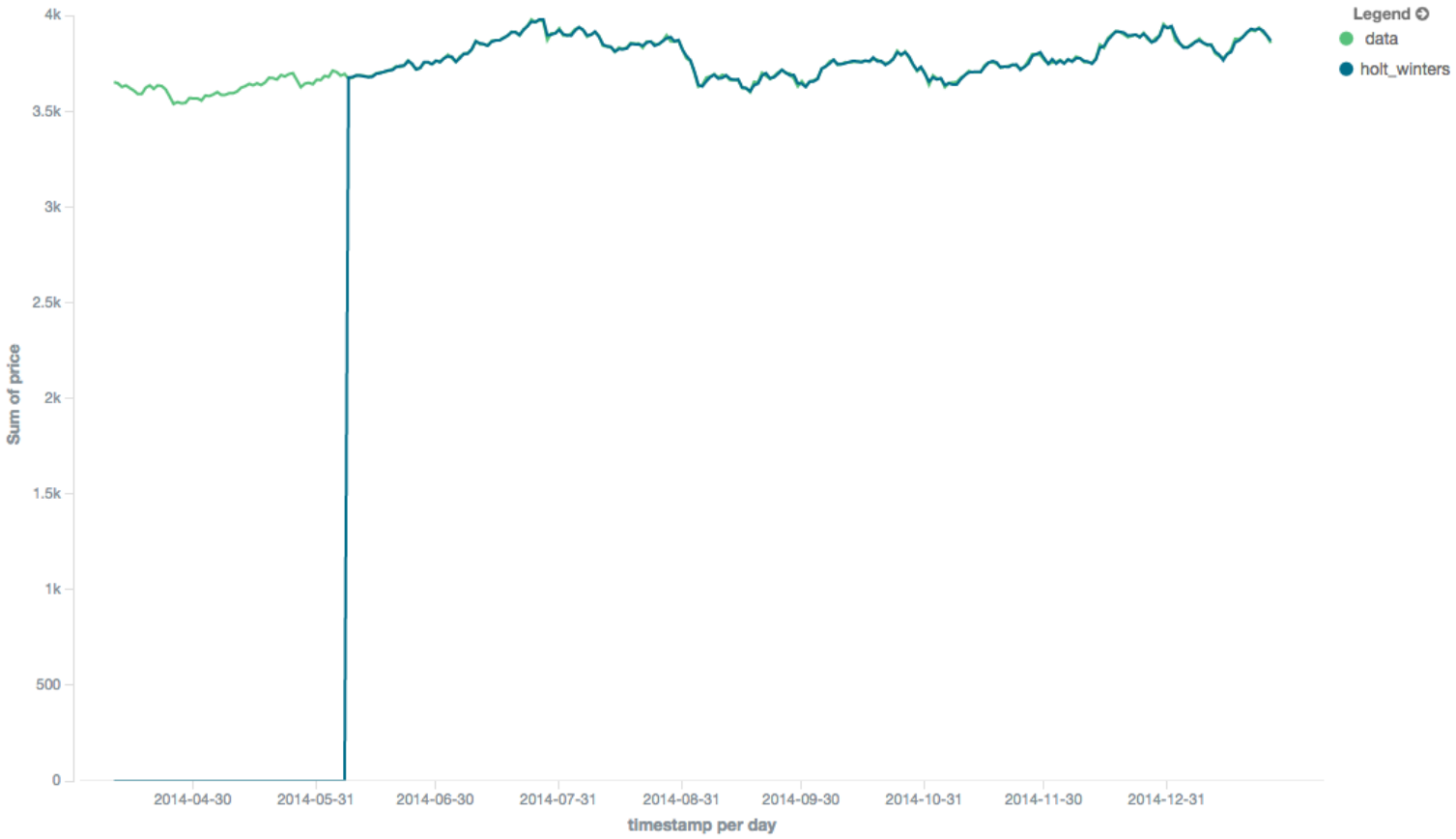
`level`和`trend`计算与holt相同。`seasonality`计算着眼于当前点和前一个周期点之间的差异。

Holt-Winters 需要比其他移动平均线多一点的handholding。您需要指定数据的“周期性”：例如，如果您的数据每7天有一次周期性趋势，您可以设置周期：7。同样，如果有月度趋势，您也可以将其设置为30。目前没有周期性检测，但这是为将来的增强而计划的。

"Cold Start"

不幸的是，由于Holt-Winters的性质，它需要两段时间的数据来“bootstrap”算法。这意味着你的窗口必须至少是你的周期的两倍。否则将引发异常。这也意味着Holt-Winters不会为前2*周期桶发出值；当前算法不进行反向预测。

图9.Holt-Winters显示了一个“冷”启动，其中没有发出任何值



因为“cold start”掩盖了移动平均线的样子，所以Holt-Winters的其余图像被截断，以不显示“冷开始”。请注意，这将始终出现在移动平均线的开头！

Additive Holt-Winters

默认情况下，添加季节性；也可以通过设置“type”：“add”来指定。当季节性影响增加到您的数据中时，首选此品种。E、 g.你可以简单地减去季节性影响，将数据“去季节化”为一个平稳的趋势。

alpha和gamma的默认值为0.3，而beta为0.1。这些设置接受0-1（含0-1）之间的任何浮动。句点的默认值为1。

可最小化加法Holt-Winters模型

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
```

```
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg": {
                    "moving_avg":{
                        "buckets_path": "the_sum",
                        "window" : 30,
                        "model" : "holt_winters",
                        "settings" : {
                            "type" : "add",
                            "alpha" : 0.5,
                            "beta" : 0.5,
                            "gamma" : 0.5,
                            "period" : 7
                        }
                    }
                }
            }
        }
    }
}
```

图10.Holt-Winters移动平均值，窗口大小为120，α=0.5，β=0.7，γ=0.3，周期=30

Multiplicative Holt-Winters

Multiplicative 通过设置“type”：“mult”来指定。当季节性影响与您的数据相乘时，首选该类型。

alpha和gamma的默认值为0.3，而beta为0.1。这些设置接受0-1（含0-1）之间的任何浮动。句点的默认值为1。

Multiplicative Holt-Winters模型可以最小化

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg": {
                    "moving_avg":{
                        "buckets_path": "the_sum",
                        "window" : 30,
                        "model" : "holt_winters",
                        "settings" : {
                            "type" : "mult",
                            "alpha" : 0.5,
                            "beta" : 0.5,
                            "gamma" : 0.5,
                            "period" : 7,
                            "pad" : true
                        }
                    }
                }
            }
        }
    }
}
```

Prediction

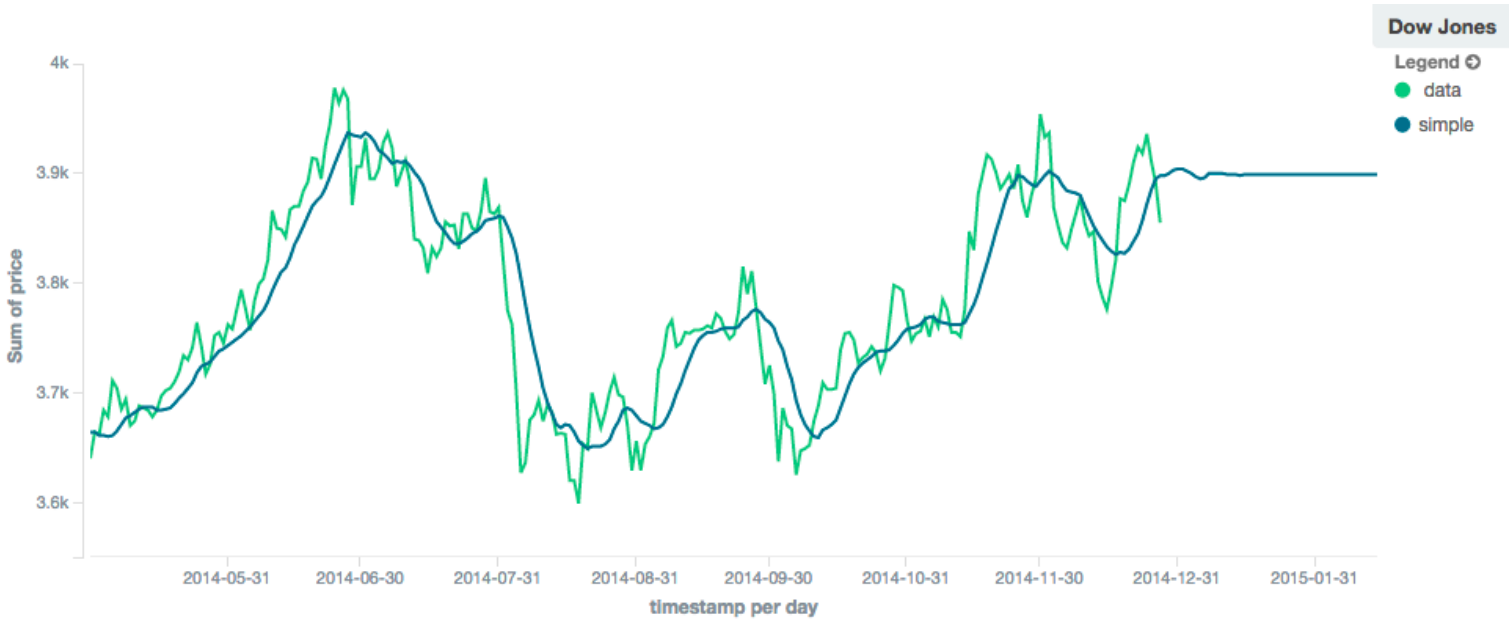
所有移动平均线模型都支持“预测”模式，该模式将尝试根据当前平滑的移动平均线推断未来。根据模型和参数，这些预测可能准确，也可能不准确。

通过向任何移动平均值聚合添加预测参数，指定要附加到系列末尾的预测数，可以启用预测。这些预测将以与您的存储桶相同的间隔进行

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg": {
                    "moving_avg":{
                        "buckets_path": "the_sum",
                        "window" : 30,
                        "model" : "simple",
                        "predict" : 10
                    }
                }
            }
        }
    }
}
```

simple、 linear 、ewma 模型都产生“flat”预测：它们基本上收敛于序列中最后一个值的平均值，产生flat：

图11.窗口大小为10的简单移动平均值，预测值=50

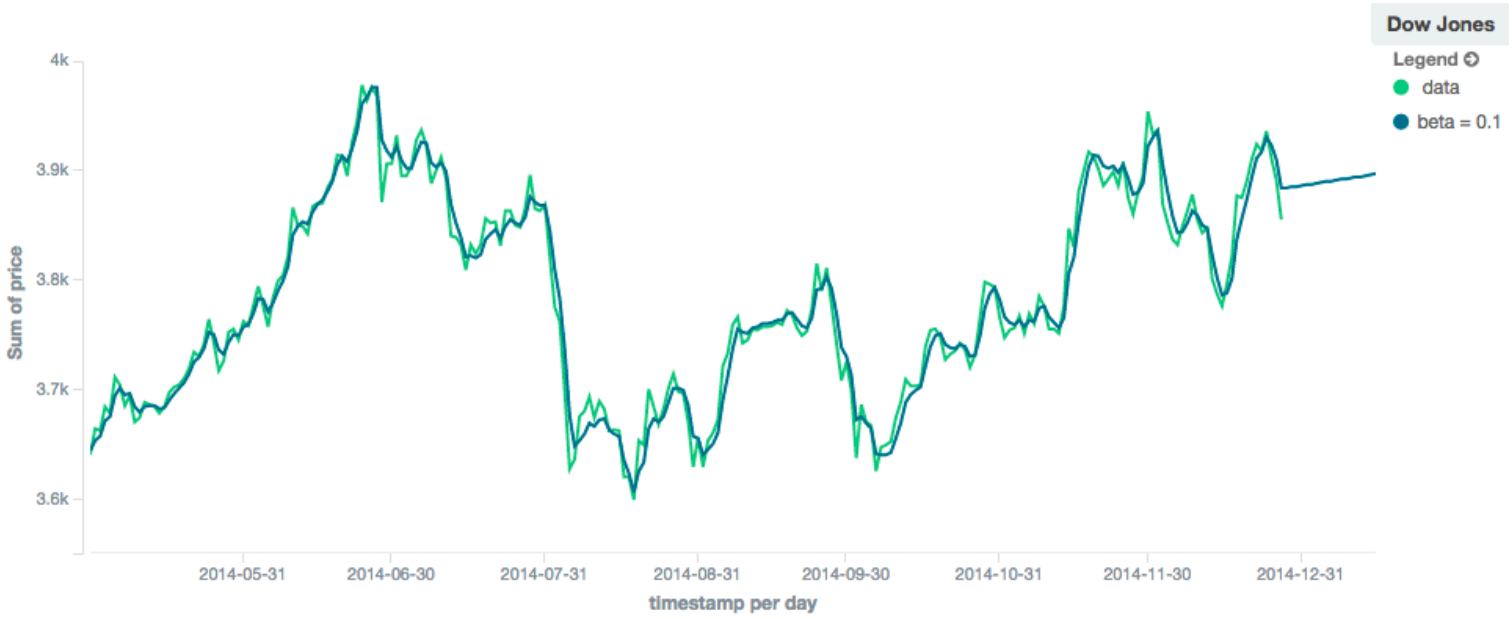


相比之下，霍尔特模型可以根据局部或全球恒定趋势进行推断。如果我们设置了一个高的贝塔值，我们可以根据局部恒定趋势进行推断（在这种情况下，预测是向下的，因为系列末尾的数据是向下的）：

图12.Holt线性移动平均值，窗口大小为100，预测值=20，α=0.5，β=0.8

相比之下，如果我们选择一个小的测试版，预测是基于全球恒定趋势。在这个系列中，全球趋势略为积极，因此预测出现了急u形转弯，并开始出现正斜率

图13.窗口大小为100的双指数移动平均值，预测值=20，alpha=0.5，beta=0.1



holt_inters模型具有提供最佳预测的潜力，因为它还将季节波动纳入模型：

图14.Holt-Winters移动平均值，窗口大小为120，预测值=25，α=0.8，β=0.2，γ=0.7，周期=30

Minimization

某些模型（EWMA、Holt Linear、Holt Winters）需要配置一个或多个参数。参数选择可能很棘手，有时不直观。此外，这些参数中的小偏差有时会对输出移动平均值产生剧烈影响。

因此，三个“可调”模型可以在算法上最小化。最小化是一个调整参数的过程，直到模型生成的预测与输出数据紧密匹配。最小化不是完全可靠的，并且可能会受到过度拟合的影响，但它通常比手动调整效果更好。

默认情况下，ewma和holt_liner禁用最小化，而holt_inters默认情况下启用最小化。最小化对Holt-Winters最有用，因为它有助于提高预测的准确性。EWMA和Holt Linear不是很好的预测工具，主要用于平滑数据，因此最小化在这些模型上不太有用。

通过最小化参数启用/禁用最小化

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg": {
                    "moving_avg":{
                        "buckets_path": "the_sum",
                        "model" : "holt_winters",
                        "window" : 30,
                        "minimize" : true,
                        "settings" : {
                            "period" : 7
                        }
                    }
                }
            }
        }
    }
}
```

启用时，最小化将找到alpha、beta和gamma的最佳值。用户仍应为窗口、时段和类型提供适当的值。

移动函数聚合 moving_fn

Moving Function Aggregation

给定一系列有序的数据，移动函数聚合将在数据上滑动一个窗口，并允许用户指定在每个数据窗口上执行的自定义脚本。为方便起见，预定义了许多常用函数，如min/max, moving averages ...

语法

```
{
    "moving_fn": {
        "buckets_path": "the_sum",
        "window": 10,
        "script": "MovingFunctions.min(values)"
    }
}
```

moving_fn聚合必须嵌入到直方图或日期直方图聚合中。它们可以像任何其他度量聚合一样嵌入：

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movfn": {
                    "moving_fn": {
                        "buckets_path": "the_sum",
                        "window": 10,
                        "script": "MovingFunctions.unweightedAvg(values)"
                    }
                }
            }
```

```
        }
    }
}
```

移动平均值是通过首先在字段上指定直方图或date_histogram来构建的。然后，您可以选择在直方图内添加数字度量，例如moving_fn被嵌入到直方图中。然后使用buckets_path参数“指向”直方图内的一个同级metric。

上述聚合的示例响应如下：

```
{
   "took": 11,
   "timed_out": false,
   "_shards": ...,
   "hits": ...,
   "aggregations": {
      "my_date_histo": {
         "buckets": [
            {
                "key_as_string": "2015/01/01 00:00:00",
                "key": 1420070400000,
                "doc_count": 3,
                "the_sum": {
                   "value": 550.0
                },
                "the_movfn": {
                   "value": null
                }
            },
            {
                "key_as_string": "2015/02/01 00:00:00",
                "key": 1422748800000,
                "doc_count": 2,
                "the_sum": {
                   "value": 60.0
                },
                "the_movfn": {
                   "value": 550.0
                }
            },
            {
                "key_as_string": "2015/03/01 00:00:00",
                "key": 1425168000000,
                "doc_count": 2,
                "the_sum": {
                   "value": 375.0
```

```
                },
                "the_movfn": {
                    "value": 305.0
                }
            }
        ]
    }
  }
}
```

Custom user scripting

移动函数聚合允许用户指定任意脚本来定义自定义逻辑。每次收集新的数据窗口时都会调用脚本。这些值在值变量中提供给脚本。然后脚本应该执行某种计算，并作为结果发出一个double。虽然允许NaN和+/-Inf，但不允许发出null。

例如，此脚本将只返回窗口中的第一个值，如果没有可用值，则返回NaN：

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg": {
                    "moving_fn": {
                        "buckets_path": "the_sum",
                        "window": 10,
                        "script": "return values.length > 0 ? values[0] :
Double.NaN"
                    }
                }
            }
        }
    }
}
```

Pre-built Functions

为了方便起见，已经预先构建了许多函数，这些函数在moving_fn脚本上下文中可用：

- max()
- min()
- sum()
- stdDev()
- unweightedAvg()
- linearWeightedAvg()
- ewma()
- holt()
- holtWinters()

这些函数可从MovingFunctions命名空间中获得。例如 `MovingFunctions.max()`

max

此函数接受一个doubles集合，并返回该窗口中的最大值。null和NaN值被忽略；最大值仅在实际值上计算。如果窗口为空，或所有值均为null/NaN，则返回NaN作为结果。

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_moving_max": {
                    "moving_fn": {
                        "buckets_path": "the_sum",
                        "window": 10,
                        "script": "MovingFunctions.max(values)"
                    }
                }
            }
```

```
        }
    }
}
```

min

此函数接受一个双精度集合，并返回该窗口中的最小值。null和NaN值被忽略；最小值仅在实际值上计算。如果窗口为空，或所有值均为null/NaN，则返回NaN作为结果。

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_moving_min": {
                    "moving_fn": {
                        "buckets_path": "the_sum",
                        "window": 10,
                        "script": "MovingFunctions.min(values)"
                    }
                }
            }
        }
    }
}
```

sum

此函数接受一个双精度集合，并返回该窗口中值的总和。null和NaN值被忽略；仅对实际值计算总和。如果窗口为空，或所有值均为空/NaN，则返回0.0作为结果。

```
POST /_search
{
    "size": 0,
    "aggs": {
```

```
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_moving_sum": {
                    "moving_fn": {
                        "buckets_path": "the_sum",
                        "window": 10,
                        "script": "MovingFunctions.sum(values)"
                    }
                }
            }
        }
    }
}
```

stdDev

此函数接受双精度和平均值的集合，然后返回该窗口中值的标准偏差。null和NaN值被忽略；仅对实际值计算总和。如果窗口为空，或所有值均为空/NaN，则返回0.0作为结果。

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_moving_sum": {
                    "moving_fn": {
                        "buckets_path": "the_sum",
                        "window": 10,
                        "script": "MovingFunctions.stdDev(values,
MovingFunctions.unweightedAvg(values))"
                    }
```

```
                }
            }
        }
    }
}
```

unweightedAvg

unweightedAvg函数计算窗口中所有值的总和，然后除以窗口的大小。它实际上是窗口的简单算术平均值。简单移动平均值不执行任何时间相关加权，这意味着来自简单移动平均的值往往“滞后”于实际数据。

null和NaN值被忽略；仅对实际值计算平均值。如果窗口为空，或所有值均为null/NaN，则返回NaN作为结果。这意味着平均计算中使用的计数是非空、非NaN值的计数。

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg": {
                    "moving_fn": {
                        "buckets_path": "the_sum",
                        "window": 10,
                        "script": "MovingFunctions.unweightedAvg(values)"
                    }
                }
            }
        }
    }
}
```

linearWeightedAvg

linearWeightedAvg函数为序列中的点分配一个线性权重，这样“较旧”的数据点（例如，窗口开始处的数据点）对总平均值的贡献线性较小。线性加权有助于减少数据平均值之后的“滞后”，因为旧点的影响较小。

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg": {
                    "moving_fn": {
                        "buckets_path": "the_sum",
                        "window": 10,
                        "script": "MovingFunctions.linearWeightedAvg(values)"
                    }
                }
            }
        }
    }
}
```

ewma

ewma函数（又称“单指数”）与linearMovAvg函数相似，只是较旧的数据点变得不那么重要，而不是线性地不那么重要。重要性衰减的速度可以通过alpha设置来控制。较小的值会使权重衰减缓慢，从而提供更大的平滑度，并考虑到窗口的较大部分。较大的值会使权重快速衰减，从而减少旧值对移动平均值的影响。这往往会使移动平均线更接近地跟踪数据，但平滑度较低。

null和NaN值被忽略；仅对实际值计算平均值。如果窗口为空，或所有值均为null/NaN，则返回NaN作为结果。这意味着平均计算中使用的计数是非空、非NaN值的计数。

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
```

```
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg": {
                    "moving_fn": {
                        "buckets_path": "the_sum",
                        "window": 10,
                        "script": "MovingFunctions.ewma(values, 0.3)"
                    }
                }
            }
        }
    }
}
```

holt

霍尔特函数（又名“双指数”）包含了跟踪数据趋势的第二个指数项。当数据具有潜在的线性趋势时，单指数表现不佳。双指数模型内部计算两个值：“水平”和“趋势”。

级别计算类似于ewma，是数据的指数加权视图。不同之处在于，使用了先前平滑的值而不是原始值，从而使其保持接近原始序列。趋势计算着眼于当前值和上次值之间的差异（例如，平滑数据的斜率或趋势）。趋势值也按指数加权。

值通过乘以水平和趋势分量产生。

null和NaN值被忽略；仅对实际值计算平均值。如果窗口为空，或所有值均为null/NaN，则返回NaN作为结果。这意味着平均计算中使用的计数是非空、非NaN值的计数。

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg": {
```

```
                    "moving_fn": {
                        "buckets_path": "the_sum",
                        "window": 10,
                        "script": "MovingFunctions.holt(values, 0.3, 0.1)"
                    }
                }
            }
        }
    }
}
```

holtWinters

holtWinters函数（又称“三指数”）包含第三个指数项，它跟踪数据的季节性方面。因此，这种聚合基于三个组成部分：“水平”、“趋势”和“季节性”。

水平和趋势计算与holt相同。季节性计算着眼于当前点和前一个周期点之间的差异。

霍尔特·温特斯（Holt Winters）需要比其他移动平均线多一点的握持。您需要指定数据的“周期性”：例如，如果您的数据每7天有一次周期性趋势，您可以将周期设置为7。同样，如果有月度趋势，您也可以将其设置为30。目前没有周期性检测，但这是为将来的增强而计划的。

null和NaN值被忽略；仅对实际值计算平均值。如果窗口为空，或所有值均为null/NaN，则返回NaN作为结果。这意味着平均计算中使用的计数是非空、非NaN值的计数。

```
POST /_search
{
    "size": 0,
    "aggs": {
        "my_date_histo":{
            "date_histogram":{
                "field":"date",
                "interval":"1M"
            },
            "aggs":{
                "the_sum":{
                    "sum":{ "field": "price" }
                },
                "the_movavg": {
                    "moving_fn": {
                        "buckets_path": "the_sum",
                        "window": 10,
                        "script": "if (values.length > 5*2)
{MovingFunctions.holtWinters(values, 0.3, 0.1, 0.1, 5, false)}"
```

```
                        }
                    }
                }
            }
        }
    }
}
```

矩阵统计聚合 matrix_stats

Matrix Stats

matrix_stats聚合是一个数字聚合，它计算一组文档字段的以下统计信息：

参数名	描述
count	计算中包含的每个字段样本数。
mean	每个字段的平均值
variance	每场测量样本与平均值的分布情况。
skewness	每场测量量化平均值周围的不对称分布。
kurtosis	每场测量量化分布的形状。
covariance	定量描述一个领域的变化如何与另一个领域相关联的矩阵。
correlation	协方差矩阵缩放到-1到1的范围（包括-1到1）。描述字段分布之间的关系。

下面的示例演示了使用矩阵统计数据来描述收入和贫困之间的关系。

```
GET /_search
{
    "aggs": {
        "statistics": {
            "matrix_stats": {
                "fields": ["poverty", "income"]
            }
        }
    }
}
```

聚合类型为matrix_stats，字段设置定义了用于计算统计信息的一组字段（作为数组）。上述请求返回以下响应：


```
{
    ...
    "aggregations": {
        "statistics": {
            "doc_count": 50,
            "fields": [{
                "name": "income",
                "count": 50,
                "mean": 51985.1,
                "variance": 7.383377037755103E7,
                "skewness": 0.5595114003506483,
                "kurtosis": 2.5692365287787124,
                "covariance": {
                    "income": 7.383377037755103E7,
                    "poverty": -21093.65836734694
                },
                "correlation": {
                    "income": 1.0,
                    "poverty": -0.8352655256272504
                }
            }, {
                "name": "poverty",
                "count": 50,
                "mean": 12.732000000000001,
                "variance": 8.637730612244896,
                "skewness": 0.4516049811903419,
                "kurtosis": 2.8615929677997767,
                "covariance": {
                    "income": -21093.65836734694,
                    "poverty": 8.637730612244896
                },
                "correlation": {
                    "income": -0.8352655256272504,
                    "poverty": 1.0
                }
            }]
        }
    }
}
```


doc_count字段表示统计计算中涉及的文档数量。

多值字段

matrix_stats聚合将每个文档字段视为独立样本。mode参数控制聚合将用于数组或多值字段的数组值。此参数可以采用以下参数之一：

参数名	描述
avg	（默认值）使用所有值的平均值。
min	选择最低值。
max	选择最高值。
sum	使用所有值的总和。
median	使用所有值的中值。

Missing Values

```
GET /_search
{
    "aggs": {
        "matrixstats": {
            "matrix_stats": {
                "fields": ["poverty", "income"],
                "missing": {"income" : 50000}
            }
        }
    }
}
```

`income`字段中没有值的文档将具有默认值50000。

Script

此聚合系列尚不支持脚本。

使用分词器

分词 就是通过文本分析（**Analysis**）将文本转换成一系列单词（**token**）的过程。 例如，字符串 “the quick Brown Foxes” 可以被分析成这些单词：`quick`、`Brown`、`fox`。分析后的单词将被编制进字段的倒排索引，以便我们在大块文本中高效地搜索单个单词。

注意：索引里只有 `text` 字段具备分词功能，默认采用 standard-analyzer 标准分词器。

简单测试

```
GET _analyze
{
  "analyzer" : "standard",
  "text" : "this is a test"
}
```

分析多个文本

```
GET _analyze
{
  "analyzer" : "standard",
  "text" : ["this is a test", "the second text"]
}
```

过滤器处理

```
GET _analyze
{
  "tokenizer" : "keyword",
  "filter" : ["lowercase"],
  "text" : "This is a test, baby"
}
```

- **keyword tokenizer** 将字符串视作一个整体不进行分词处理

- **filter** 将分词结果小写化

前置处理

```
GET _analyze
{
  "tokenizer" : "keyword",
  "filter" : ["lowercase"],
  "char_filter" : ["html_strip"],
  "text" : "this is a <b>test</b>"
}
```

- **char_filter** : `html_strip` 将输入文本去除html标签

自定义tokenizer和filter

添加排除停用词的一个例子

```
GET _analyze
{
  "tokenizer" : "whitespace",
  "filter" : ["lowercase", {"type": "stop", "stopwords": ["a", "is", "this"]}],
  "text" : "this is a test"
}
```

将返回唯一的单词 `test`

在指定的索引上分析

```
GET analyze_sample/_analyze
{
  "text" : "this is a test"
}
```

上面将使用与analyze_sample索引关联的默认`analyzer`对文本进行分析，也可以指定其他`analyzer`进行测试。

```
GET analyze_sample/_analyze
{
  "analyzer" : "whitespace",
  "text" : "this is a test"
}
```

分词组件概念

如上，一个分词器`Analyze`由以下几个组件`Character Filters`、`Tokenizer`、`Token Filters`组成，共同协助分词过程，这些过滤器可以组合起来为每个索引配置自定义分析器。

- **Character Filters**：原始文本处理，比如去除 html 标签
- **Tokenizer**：分词规则，比如按照空格切分
- **Token Filters**：将切分的单词进行再加工，比如大写转小写，删除停用词，增加同义语等

Elasticsearch 内置的分词器

- Standard Analyzer - 默认分词器，按词切分，小写处理
- Simple Analyzer - 按照非字母切分（符号被过滤），小写处理
- Stop Analyzer - 小写处理，停用词过滤（the ，a，is）
- Whitespace Analyzer - 按照空格切分，不转小写
- Keyword Analyzer - 不分词，直接将输入当做输出
- Pattern Analyzer - 正则表达式，默认 \W+
- Language Analyzer - 提供了 30 多种常见语言的分词器
- Custom Analyzer - 自定义分词器

纳速云附加安装的分词器

- IK Analyzer - IK中文分词器

Standard Analyzer

标准分词器（默认）：它提供基于语法的标记化（基于Unicode标准规定的Unicode文本分割算法），适用于大多数语言。

示例

```
POST _analyze
{
  "analyzer": "standard",
  "text": "The 2 QUICK Brown-Foxes jumped over the lazy dog's bone."
}
```

分词结果

```
[ the, 2, quick, brown, foxes, jumped, over, the, lazy, dog's, bone ]
```

配置

标准分词器接受以下参数：

- **max_token_length** 最大token长度。如果看到超过此长度的token，则以max_token_length间隔对其进行拆分。默认值为255。
- **stopwords** 预定义的停用词列表，如*english*或包含停止词列表的数组。默认为*none*。

配置示例

```
PUT my-index-000001
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_english_analyzer": {
          "type": "standard",
          "max_token_length": 5,
```

```
          "stopwords": "_english_"
        }
      }
    }
  }
}

POST my-index-000001/_analyze
{
  "analyzer": "my_english_analyzer",
  "text": "The 2 QUICK Brown-Foxes jumped over the lazy dog's bone."
}
```

上述示例产生以下分词结果：

```
[ 2, quick, brown, foxes, jumpe, d, over, lazy, dog's, bone ]
```

定义

`Standard Analyzer` 包括

- **Tokenizer**
  - Standard Tokenizer
- **Token Filters**
  - Lower Case Token Filter
  - Stop Token Filter (默认关闭)

自定义标准分析器，通常通过添加filter的方式增加自定义行为：

```
PUT /standard_example
{
  "settings": {
    "analysis": {
      "analyzer": {
        "rebuilt_standard": {
          "tokenizer": "standard",
          "filter": [
            "lowercase"
          ]
        }
      }
    }
```

```
    }
}
```

Simple Analyzer

每当遇到不是字母的字符时，简单分析器就会将文本拆分token。所有token均小写。

示例

```
POST _analyze
{
  "analyzer": "simple",
  "text": "org.elasticsearch.action.admin.indices.analyze.AnalyzeAction"
}
```

分词结果

```
[ org, elasticsearch, action, admin, indices, analyze, AnalyzeAction ]
```

参数

`simple analyzer` 不接受任何参数

定义

`Simple Analyzer` 包括

- **Tokenizer**
  - Lower Case Tokenizer

自定义简单分析器，通常通过添加自定义filter来处理结果。

```
PUT /simple_example
{
  "settings": {
    "analysis": {
      "analyzer": {
        "rebuilt_simple": {
```

```
          "tokenizer": "lowercase",
          "filter": [
            ## 添加你的filter
          ]
        }
      }
    }
  }
}
```

Stop Analyzer

停用词分析器在 `simple Analyzer` 的基础上增加了对排除停止词的支持。它默认使用_english_stop单词。

示例

```
POST _analyze
{
  "analyzer": "stop",
  "text": "The 2 QUICK Brown-Foxes jumped over the lazy dog's bone."
}
```

分词结果

```
[ quick, brown, foxes, jumped, over, lazy, dog, s, bone ]
```

配置

停用词分析器接受以下参数：

- **stopwords** 预定义的停止词列表，如*english*或包含停止词列表的数组。默认值为*english*。

参数示例

在此示例中，我们将停用词分析器配置为使用指定的单词列表作为停用单词：

```
PUT my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_stop_analyzer": {
          "type": "stop",
          "stopwords": ["the", "over"]
```

```
        }
      }
    }
  }
}

POST my_index/_analyze
{
  "analyzer": "my_stop_analyzer",
  "text": "The 2 QUICK Brown-Foxes jumped over the lazy dog's bone."
}
```

分词结果

```
[ quick, brown, foxes, jumped, lazy, dog, s, bone ]
```

定义

- **Tokenizer**
  - Lower Case Tokenizer
- **Token filters**
  - Stop Token Filter

自定义示例

```
PUT /stop_example
{
  "settings": {
    "analysis": {
      "filter": {
        "english_stop": {
          "type":       "stop",
          "stopwords":  "_english_"  (1)
        }
      },
      "analyzer": {
        "rebuilt_stop": {
          "tokenizer": "lowercase",
          "filter": [
            "english_stop"           (2)
          ]
        }
```

```
        }
      }
    }
}
```

1. 可以使用stopwwords参数覆盖默认的stopwword。
2. 您可以在english_stop之后添加任何token filter。

Whitespace Analyzer

每当遇到空白字符时，空格分析器都会将文本拆分为token。

示例

```
POST _analyze
{
  "analyzer": "whitespace",
  "text": "The 2 QUICK Brown-Foxes jumped over the lazy dog's bone."
}
```

分词结果

```
[ The, 2, QUICK, Brown-Foxes, jumped, over, the, lazy, dog's, bone. ]
```

参数

`whitespace analyzer` 不接受任何参数

定义

`Whitespace Analyzer` 包括

- **Tokenizer**
  - Whitespace Tokenizer

自定义空格分词器，通常通过添加自定义filter来增加结果处理。

```
PUT /whitespace_example
{
  "settings": {
    "analysis": {
      "analyzer": {
        "rebuilt_whitespace": {
```

```
          "tokenizer": "whitespace",
          "filter": [
            ## 添加你的filter
          ]
        }
      }
    }
  }
}
```

Keyword Analyzer

关键字分析器将整个输入字符串作为单个token返回

示例

```
POST _analyze
{
  "analyzer": "keyword",
  "text": "The 2 QUICK Brown-Foxes jumped over the lazy dog's bone."
}
```

分词结果

```
[ The 2 QUICK Brown-Foxes jumped over the lazy dog's bone. ]
```

参数

关键字分析器不可配置。

定义

- **Tokenizer**
  - Keyword Tokenizer

自定义示例

```
PUT /stop_example
PUT /keyword_example
{
  "settings": {
    "analysis": {
      "analyzer": {
        "rebuilt_keyword": {
          "tokenizer": "keyword",
```

```
            "filter": [          (1)
            ]
          }
        }
      }
    }
}
```

1. 您可以在此处添加任何令牌筛选器。

Pattern Analyzer

模式分析器使用正则表达式将文本拆分为token。正则表达式默认为\W+（所有非单词字符）。

> **谨防不理智的正则表达式**
>
> 模式分析器使用Java正则表达式，一个写得不好的正则表达式可能运行得非常慢，甚至引发StackOverflowError。

示例

```
POST _analyze
{
  "analyzer": "pattern",
  "text": "The 2 QUICK Brown-Foxes jumped over the lazy dog's bone."
}
```

分词结果

```
[ the, 2, quick, brown, foxes, jumped, over, the, lazy, dog, s, bone ]
```

配置

模式分析器接受以下参数：

- **pattern** Java正则表达式，默认为\W+。
- **flags** Java正则表达式标志。标志应以`|`分隔，例如“CASE_INSENSITIVE | COMMENTS”。
- **lowercase** 是否小写。默认为true。
- **stopwords** 预定义的停用词列表，如*english*或包含停止词列表的数组。默认为*none*。

配置参数示例

在本例中，我们将模式分析器配置为将电子邮件地址拆分为非单词字符或下划线（\W|_），并将结果小写：

```
PUT my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_email_analyzer": {
          "type":      "pattern",
          "pattern":   "\\W|_",
          "lowercase": true
        }
      }
    }
  }
}

POST my_index/_analyze
{
  "analyzer": "my_email_analyzer",
  "text": "John_Smith@foo-bar.com"
}
```

> **TIP**
>
> pattern 将模式指定为JSON字符串时，需要转义模式中的反斜杠。

分词结果

```
[ john, smith, foo, bar, com ]
```

CamelCase驼峰标记器

驼峰式拆分

```
PUT my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
```

```
        "camel": {
          "type": "pattern",
          "pattern": "([^\\p{L}\\d]+)|(?<=\\D)(?=\\d)|(?<=\\d)(?=\\D)|(?<=[\\p{L}&&
[^\\p{Lu}]])(?=\\p{Lu})|(?<=\\p{Lu})(?=\\p{Lu}[\\p{L}&&[^\\p{Lu}]])"
        }
      }
    }
  }
}

GET my_index/_analyze
{
  "analyzer": "camel",
  "text": "MooseX::FTPClass2_beta"
}
```

分词结果

```
[ moose, x, ftp, class, 2, beta ]
```

上面的正则表达式更容易理解为：

```
  ([^\p{L}\d]+)                 # swallow non letters and numbers,
| (?<=\D)(?=\d)                 # or non-number followed by number,
| (?<=\d)(?=\D)                 # or number followed by non-number,
| (?<=[ \p{L} && [^\p{Lu}]])    # or lower case
  (?=\p{Lu})                    #   followed by upper case,
| (?<=\p{Lu})                   # or upper case
  (?=\p{Lu}                     #   followed by upper case
    [\p{L}&&[^\p{Lu}]]          #   then lower case
  )
```

定义

- **Tokenizer**
  - Pattern Tokenizer
- **Token Filters**
  - Lower Case Token Filter
  - Stop Token Filter (disabled by default)

自定义示例

```
PUT /pattern_example
{
  "settings": {
    "analysis": {
      "tokenizer": {
        "split_on_non_word": {
          "type":       "pattern",
          "pattern":    "\\W+" (1)
        }
      },
      "analyzer": {
        "rebuilt_pattern": {
          "tokenizer": "split_on_non_word",
          "filter": [
            "lowercase"         (2)
          ]
        }
      }
    }
  }
}
```

1. 默认模式是\W+，它在非单词字符上拆分，您可以在此处更改它。
2. 您可以在小写后添加其他令牌筛选器。

Language Analyzers

语言分析器支持以下类型

`arabic, armenian, basque, bengali, brazilian, bulgarian, catalan, cjk, czech, danish, dutch, english, finnish, french, galician, german, greek, hindi, hungarian, indonesian, irish, italian, latvian, lithuanian, norwegian, persian, portuguese, romanian, russian, sorani, spanish, swedish, turkish, thai`.

> 阿拉伯语、亚美尼亚语、巴斯克语、孟加拉语、巴西语、保加利亚语、加泰罗尼亚语、捷克语、丹麦语、荷兰语、英语、芬兰语、法语、加利西亚语、德语、希腊语、印地语、匈牙利语、印度尼西亚语、爱尔兰语、意大利语、拉脱维亚语、立陶宛语、挪威语、波斯语、葡萄牙语、罗马尼亚语、俄语、索拉尼语、西班牙语、瑞典语、土耳其语、泰国语

配置语言分析器

- **Stopwords** 所有预压分析器都支持在配置中内部设置自定义 stopwords。
- **控制词干提取** `stem_exclusion` 参数允许您指定从词干中排除的单词数组。

arabic analyzer

阿拉伯语分析仪可以作为定制分析仪重新实现，如下所示：

```
PUT /arabic_example
{
  "settings": {
    "analysis": {
      "filter": {
        "arabic_stop": {
          "type":       "stop",
          "stopwords":  "_arabic_"
        },
        "arabic_keywords": {
          "type":       "keyword_marker",
          "keywords":   ["مثال"]
        },
        "arabic_stemmer": {
```

```
          "type":       "stemmer",
          "language":   "arabic"
        }
      },
      "analyzer": {
        "rebuilt_arabic": {
          "tokenizer":  "standard",
          "filter": [
            "lowercase",
            "decimal_digit",
            "arabic_stop",
            "arabic_normalization",
            "arabic_keywords",
            "arabic_stemmer"
          ]
        }
      }
    }
  }
}
```

更多详细配置

Fingerprint Analyzer

指纹分析器实现了OpenRefine项目使用的指纹识别算法来协助聚类

输入文本采用小写、标准化以删除扩展字符、排序、消除重复并连接到单个标记中。如果配置了停止词列表，停止词也将被删除。

示例

```
POST _analyze
{
  "analyzer": "fingerprint",
  "text": "Yes yes, Gödel said this sentence is consistent and."
}
```

分词结果

```
[ and consistent godel is said sentence this yes ]
```

参数

- **separator** 用于连接terms的字符。默认为空格。
- **max_output_size** 发出的最大token大小。默认值为255。大于此大小的token将被丢弃。
- **stopwords** 预定义的停用词列表，如*english*或包含停止词列表的数组。默认为*none*。

参数示例

在本例中，我们将模式分析器配置为将电子邮件地址拆分为非单词字符或下划线（\W|_），并将结果小写：

```
PUT my_index
{
  "settings": {
    "analysis": {
```

```
      "analyzer": {
        "my_fingerprint_analyzer": {
          "type": "fingerprint",
          "stopwords": "_english_"
        }
      }
    }
  }
}

POST my_index/_analyze
{
  "analyzer": "my_fingerprint_analyzer",
  "text": "Yes yes, Gödel said this sentence is consistent and."
}
```

分词结果

```
[ consistent godel said sentence yes ]
```

定义

- Tokenizer
  - Standard Tokenizer
- Token Filters (in order)
  - Lower Case Token Filter
  - ASCII Folding Token Filter
  - Stop Token Filter (disabled by default)
  - Fingerprint Token Filter

自定义示例

```
PUT /fingerprint_example
{
  "settings": {
    "analysis": {
      "analyzer": {
        "rebuilt_fingerprint": {
          "tokenizer": "standard",
          "filter": [
            "lowercase",
```

```
            "asciifolding",
            "fingerprint"
          ]
        }
      }
    }
  }
}
```

Custom Analyzer

当内置分析器不能满足您的需求时，您可以创建一个自定义分析器，采用适当的组合来构建：

- 零个或多个 `character filters`
- 1个 `tokenizer`
- 零个或多个 `token filters`.

配置

自定义分析器接受以下参数：

- **tokenizer** 内置或自定义标记器。（必需）
- **char_filter** 内置或自定义字符过滤器的可选阵列。
- **filter** 内置或自定义令牌过滤器的可选阵列。
- **position_increment_gap** 在对文本值数组进行索引时，Elasticsearch在一个值的最后一个项和下一个值中的第一个项之间插入一个假“间隙”，以确保短语查询不匹配来自不同数组元素的两个项。默认值为100。

参数示例

- Character Filter
  - HTML Strip Character Filter
- Tokenizer
  - Standard Tokenizer
- Token Filters
  - Lowercase Token Filter
  - ASCII-Folding Token Filter

```
PUT my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_custom_analyzer": {
```

```
          "type":      "custom",  (1)
          "tokenizer": "standard",
          "char_filter": [
            "html_strip"
          ],
          "filter": [
            "lowercase",
            "asciifolding"
          ]
        }
      }
    }
  }
}

POST my_index/_analyze
{
  "analyzer": "my_custom_analyzer",
  "text": "Is this <b>déjà vu</b>?"
}
```

分词结果

```
[ is, this, deja, vu ]
```

前面的示例使用了具有默认配置的标记化器、标记过滤器和字符过滤器，但可以创建每个的配置版本并在自定义分析器中使用它们。

下面是一个更复杂的示例，结合了以下内容：

- Character Filter
  - Mapping Character Filter, configured to replace :) with *happy* and :(with *sad*
- Tokenizer
  - Pattern Tokenizer, 配置为在标点符号上拆分
- Token Filters
  - Lowercase Token Filter
  - Stop Token Filter, 配置为使用预定义的英文停止词列表

```
PUT my_index
{
  "settings": {
    "analysis": {
```

```
      "analyzer": {
        "my_custom_analyzer": {  (1)
          "type": "custom",
          "char_filter": [
            "emoticons"
          ],
          "tokenizer": "punctuation",
          "filter": [
            "lowercase",
            "english_stop"
          ]
        }
      },
      "tokenizer": {
        "punctuation": {  (2)
          "type": "pattern",
          "pattern": "[ .,!?]"
        }
      },
      "char_filter": {
        "emoticons": {  (3)
          "type": "mapping",
          "mappings": [
            ":) => _happy_",
            ":( => _sad_"
          ]
        }
      },
      "filter": {
        "english_stop": { (4)
          "type": "stop",
          "stopwords": "_english_"
        }
      }
    }
  }
}

POST my_index/_analyze
{
  "analyzer": "my_custom_analyzer",
  "text":     "I'm a :) person, and you?"
}
```

1. 为索引分配默认自定义分析器my_custom_analysis。此分析器使用自定义标记器、字符过滤器和稍后在请求中定义的标记过滤器。
2. 定义自定义标点符号化器。

3. 定义自定义表情符号字符筛选器。

4. 定义自定义english_stop标记筛选器。

脚本 Scripting

使用scripting，您可以在Elasticsearch中自定义评分、自定义文本相关度、自定义过滤、自定义聚合析等。

内置scripting的语言支持

语言	沙盒	用途
painless	yes	通用
expression	yes	快速自定义排序
mustache	yes	templates

用法

```
  "script": {
    "lang":   "...",
    "source" | "id": "...",
    "params": { ... }
  }
```

- **lang**: 脚本使用的语言，默认为painless。
- **source**: 脚本本身，可以指定为 inline script 或已存储脚本的id。
- **params**: 传递到脚本中的参数。

例如，在搜索请求中使用以下脚本返回mapping中未定义的字段`script_feild`：

```
PUT my_index/_doc/1
{
  "my_field": 5
}

GET my_index/_search
{
```

```
  "script_fields": {
    "my_doubled_field": {
      "script": {
        "lang":   "expression",
        "source": "doc['my_field'] * multiplier",
        "params": {
          "multiplier": 2
        }
      }
    }
  }
}
```

访问文档字段和特殊变量

根据脚本的使用位置，它可以访问某些特殊变量和文档字段。

Update scripts

参数	说明
ctx._source	应用于文档的_source字段
ctx.op	应用于文档的操作：index或delete。
ctx._index	应用于文档元字段

搜索和聚合脚本

除了每个搜索命中执行一次的脚本字段外，搜索和聚合中使用的脚本将对可能匹配查询或聚合的每个文档执行一次。根据您拥有的文档数量，这可能意味着数百万或数十亿次的执行：这些脚本需要非常高效！

可以使用 docvalue、_source 或 stored fields 从脚本中访问字段值，下面将对每个字段进行解释

访问文档的_score

使用 function_score query、基于脚本的排序或聚合中使用的脚本可以访问_score变量，该变量表示文档的当前相关性得分。

以下是在function_score query中使用脚本来更改每个文档的相关性_score的示例：

```
PUT my_index/_doc/1?refresh
{
  "text": "quick brown fox",
  "popularity": 1
}

PUT my_index/_doc/2?refresh
{
```

```
  "text": "quick fox",
  "popularity": 5
}

GET my_index/_search
{
  "query": {
    "function_score": {
      "query": {
        "match": {
          "text": "quick brown fox"
        }
      },
      "script_score": {
        "script": {
          "lang": "expression",
          "source": "_score * doc['popularity']"
        }
      }
    }
  }
}
```

DocValue

从脚本中访问字段值最快最有效的方法是使用doc['field_name']语法，该语法从doc值中检索字段值。docvalue是一个列式存储，默认情况下，除 `analyzed text fields` 外，建议所有字段都启用。

```
PUT my_index/_doc/1?refresh
{
  "cost_price": 100
}

GET my_index/_search
{
  "script_fields": {
    "sales_price": {
      "script": {
        "lang":   "expression",
        "source": "doc['cost_price'] * markup",
        "params": {
          "markup": 0.2
        }
      }
    }
```

```
  }
}
```

> **TIP**
>
> docvalue只能返回“简单”字段值，如数字、日期、地理点、term等，如果字段是多值的，则只能返回这些值的数组，它无法返回JSON对象。

_source

可以使用_source.field_name语法访问文档_source。例如_source.name.first访问。

```
PUT my_index
{
  "mappings": {
    "_doc": {
      "properties": {
        "first_name": {
          "type": "text"
        },
        "last_name": {
          "type": "text"
        }
      }
    }
  }
}

PUT my_index/_doc/1?refresh
{
  "first_name": "Barry",
  "last_name": "White"
}

GET my_index/_search
{
  "script_fields": {
    "full_name": {
      "script": {
        "lang": "painless",
        "source": "params._source.first_name + ' ' + params._source.last_name"
      }
    }
  }
}
```

Stored fields

Stored fields一显式标记为“store”的字段：mapping中设置为true则可以使用_fields['field_name'].value或_fields[fields_name']语法访问：

```
PUT my_index
{
  "mappings": {
    "_doc": {
      "properties": {
        "full_name": {
          "type": "text",
          "store": true
        },
        "title": {
          "type": "text",
          "store": true
        }
      }
    }
  }
}

PUT my_index/_doc/1?refresh
{
  "full_name": "Alice Ball",
  "title": "Professor"
}

GET my_index/_search
{
  "script_fields": {
    "name_with_title": {
      "script": {
        "lang": "painless",
        "source": "params._fields['title'].value + ' ' +
params._fields['full_name'].value"
      }
    }
  }
}
```

Painless scripting language

Painless是一种简单、安全的脚本语言，专门设计用于Elasticsearch。它是Elasticsearch的默认脚本语言，可以安全地用于 inline and stored scripts。

您可以在Elasticsearch中可以使用的任何地方使用Painless脚本。Painless提供：

- 快速性能：Painless脚本的运行速度比其他脚本快几倍。
- 安全性：使用白名单来限制函数与字段的访问，避免了可能存在的安全隐患。
- 可选类型：变量和参数可以使用显式类型或`dynamic def`类型。
- 语法：扩展Java的语法，提供Groovy风格的脚本语言特性，使脚本更易于编写。
- 优化：专为Elasticsearch脚本设计。

Painless 语言规范

Lucene expressions language

Lucene的表达式将javascript表达式编译为字节码。它们是为高性能自定义排名和排序功能而设计的，默认情况下支持内联和存储脚本。

Performance

Expressions 旨在与高性能的 Lucene 代码。这种性能是由于与其他脚本引擎相比，每个文档的开销较低：表达式更“up-front”。 这允许非常快速的执行，甚至比您编写本机脚本还要快

语法

表达式支持javascript语法的子集。

表达式脚本中可用于访问的变量：

- 文档字段，例如 `doc['myfield'].value`
- 字段的变量和方法，例如 `doc['myfield'].empty`
- 传递到脚本中的参数，例如 `mymodifier`
- 当前文档的分数 `_score`（仅在`script_score`中使用时可用）

> **INFO**
> 您可以将表达式脚本用于 `script_score、script_fields、sort scripts、numeric aggregation scripts`, 只需将`lang`参数设置为`expression`即可。

Numeric field API

Expression	说明
doc['field_name'].value	字段的值
doc['field_name'].empty	一个布尔值，指示字段在文档中是否没有值

Expression	说明
doc['field_name'].length	文档长度
doc['field_name'].min()	此文档中字段的最小值
doc['field_name'].max()	此文档中字段的最大值
doc['field_name'].median()	此文档中字段的中值
doc['field_name'].avg()	此文档中字段的平均值
doc['field_name'].sum()	此文档中字段的总和

TIP

- 当文档完全缺少字段时，默认值将被视为0。您可以将其视为另一个值，例如 `doc['myfield'].empty ? 100 : doc['myfield'].value`
- 当文档具有多个字段值时，默认情况下返回最小值。您可以选择不同的值，例如 `doc['myfield'].sum()`。
- 当文档完全缺少字段时，默认值将被视为0。
- 布尔字段暴露为数字，true映射为1，false映射为0。例如：`doc['on_sale'].value ? doc['price'].value * 0.5 : doc['price'].value`

Date field API

Expression	说明
doc['field_name'].date.centuryOfEra	100年 (1-2920000)
doc['field_name'].date.dayOfMonth	Day (1-31), 例如 1 为当月的第一天.
doc['field_name'].date.dayOfWeek	一周中的某一天 (1-7) 例如 1 为星期一
doc['field_name'].date.dayOfYear	一年中的某一天 ， 例如 1 为1月1日

Expression	说明
doc['field_name'].date.era	纪元：公元前：0，公元前：1
doc['field_name'].date.monthOfYear	年内月份 (1-12)
doc['field_name'].date.hourOfDay	天内小时数 (0-23)
doc['field_name'].date.millisOfDay	天内的毫秒数（0-8639999）
doc['field_name'].date.minuteOfDay	天内分钟数 (0-1439).
doc['field_name'].date.secondOfDay	天内秒数 (0-86399).
doc['field_name'].date.minuteOfHour	小时内分钟数 (0-59).
doc['field_name'].date.secondOfMinute	分钟内秒数 (0-59).
doc['field_name'].date.millisOfSecond	秒内的毫秒数（0-999）
doc['field_name'].date.year	年 (-292000000 - 292000000).
doc['field_name'].date.yearOfCentury	世纪内年份 (1-100).
doc['field_name'].date.yearOfEra	时代内年份 (1-292000000).

以下示例显示日期字段date0和date1之间的年份差异：

```
doc['date1'].date.year - doc['date0'].date.year
```

geo_point field API

Expression	说明
doc['field_name'].empty	一个布尔值，指示字段在文档中是否没有值。

Expression	说明
doc['field_name'].lon	地理点的经度
doc['field_name'].lat	地理点的纬度

以下示例计算距离华盛顿特区的公里数：

```
haversin(38.9072, 77.0369, doc['field_name'].lat, doc['field_name'].lon)
```

在此示例中，坐标可以作为参数传递给脚本，例如基于用户的地理位置

Limitations

与其他脚本语言相比，存在一些限制：

- 只能访问数字、布尔值、日期和地理点字段
- Stored fields不可用

跨索引过滤 JoinedQuery

JoinedQuery 从引擎层面实现了跨索引过滤功能，用以简化 ES 在多个索引间的关系处理。

何时使用

需要通过主子表关联进行结果集过滤的场景

> Join基本上是基于共同属性的两组文档之间的semi-join，其中结果仅包含文档集合的一个属性。该连接用于过滤基于第二文档集的一个文档集，类似于SQL中的EXISTS算子。

```
POST ${主表}/_search
{
    "query": {
        "joined": {
            "query": {
                "match_all": {}  #主表原始Query
            },
            "join": [
                {
                    "${主表关联字段}": {
                        "indices": "${子表}",
                        "field": "${子表关联字段}",
                        "query": {
                            "term": {  #子表过滤Query
                                "field": "field_value"
                            }
                        }
                    }
                }
            ]
        }
    }
}
```

示例

准备一些测试数据

- 主表
  - score : 成绩表
- 子表
  - user : 学生表
  - subject : 科目表

注意： 关联外键类型需使用数值类型（integer或long)

```
-- 录入成绩表

POST score/_bulk
{ "index" : { "_type" : "_doc" } }
{ "score" : 95, "user_id": 1, "subject_id": 1  }
{ "index" : { "_type" : "_doc" } }
{ "score" : 66, "user_id": 1, "subject_id": 2  }
{ "index" : { "_type" : "_doc" } }
{ "score" : 88, "user_id": 2, "subject_id": 1  }
{ "index" : { "_type" : "_doc" } }
{ "score" : 100, "user_id": 2, "subject_id": 2  }

-- 录入学生表及科目表

PUT user
{
  "mappings": {
    "_doc": {
      "properties": {
        "id": {
          "type": "integer"
        },
        "name": {
          "type": "keyword"
        }
      }
    }
  }
}

PUT subject
{
  "mappings": {
    "_doc": {
      "properties": {
        "id": {
```

```
          "type": "integer"
        },
        "name": {
          "type": "keyword"
        }
      }
    }
  }
}

POST user/_doc
{
  "id": 1,
  "name": "jack"
}

POST user/_doc
{
  "id": 2,
  "name": "rose"
}

POST subject/_doc
{
  "id": 1,
  "name": "物理"
}

POST subject/_doc
{
  "id": 2,
  "name": "化学"
}
```

> 关联关系
> `score.user_id = user.id`
> `score.subject_id = subject.id`

示例1 - 查看 jack 的成绩表 (单表连接)

使用JoinedQuery跨索引过滤，功能类似如下sql

```
SELECT score.* FROM score
LEFT JOIN user
```

```
ON user.id=score.user_id
WHERE user.name='jack'
```

```
GET score/_search
{
    "query": {
        "joined": {
            "query": {
                "match_all": {}
            },
            "join": [{
                "user_id": {
                    "indices": "user",
                    "field": "id",
                    "query": {
                        "term" : { "name" : "jack" }
                    }
                }
            }]
        }
    }
}
```

示例2 -多子表连接

类似

```
SELECT score.* FROM score
LEFT JOIN user
ON user.id=score.user_id
LEFT JOIN subject
ON subject.id=score.subject_id
WHERE user.name='?' AND subject.name='?'
```

查看 jack 同学的物理成绩表

```
GET score/_search
{
    "query": {
        "joined": {
            "query": {
                "match_all": {}
            },
```

```
            "join": [
              {
                "user_id": {
                    "indices": "user",
                    "field": "id",
                    "query": {
                        "term" : { "name" : "jack" }
                    }
                }
              },
              {
                "subject_id": {
                    "indices": "subject",
                    "field": "id",
                    "query": {
                        "term" : { "name" : "物理" }
                    }
                }
              }
            ]
        }
    }
}
```

自定义排序 RankedQuery

RankedQuery 在原始的查询结果集上增加了二次自定义排位功能，可用于 BadCase 的快速处理或垂搜场景的结果集处理。

示例

准备一些测试数据，例如从1到10的序列

```
POST _bulk
{ "index" : { "_index" : "greeting","_type":"_doc", "_id" : "1" } }
{ "number" : 1 }
{ "index" : { "_index" : "greeting","_type":"_doc", "_id" : "2" } }
{ "number" : 2 }
{ "index" : { "_index" : "greeting","_type":"_doc", "_id" : "3" } }
{ "number" : 3 }
{ "index" : { "_index" : "greeting","_type":"_doc", "_id" : "4" } }
{ "number" : 4 }
{ "index" : { "_index" : "greeting","_type":"_doc", "_id" : "5" } }
{ "number" : 5 }
{ "index" : { "_index" : "greeting","_type":"_doc", "_id" : "6" } }
{ "number" : 6 }
{ "index" : { "_index" : "greeting","_type":"_doc", "_id" : "7" } }
{ "number" : 7 }
{ "index" : { "_index" : "greeting","_type":"_doc", "_id" : "8" } }
{ "number" : 8 }
{ "index" : { "_index" : "greeting","_type":"_doc", "_id" : "9" } }
{ "number" : 9 }
{ "index" : { "_index" : "greeting","_type":"_doc", "_id" : "10" } }
{ "number" : 10 }
```

执行简单查询

```
GET greeting/_search
{
  "query": {
    "match_all": {}
  }
}
```

结果集默认按_id排序

```
 "hits" : [
      {
        "_source" : {"number" : 1}
      },
      {
        "_source" : {"number" : 2}
      },
      {
        "_source" : {"number" : 3}
      },
      ...
```

1. 文档置顶

将 _id 为 7 和 8 的文档置顶

```
GET greeting/_search
{
  "query": {
    "ranked": {
      "query": {
        "match_all": {}
      },
      "rank": {
        "top": ["7","8"]
      }
    }
  }
}
```

结果集如下

```
 "hits" : [
      {
        "_source" : {"number" : 7}
      },
      {
        "_source" : {"number" : 8}
      },
      {
        "_source" : {"number" : 1}
```

```
      },
      {
        "_source" : {"number" : 2}
      },
      ...
```

2. 移除文档

将_id 为 1 和 2 的文档移除

```
GET greeting/_search
{
  "query": {
    "ranked": {
      "query": {
        "match_all": {}
      },
      "rank": {
        "block" : [ "1", "2" ]
      }
    }
  }
}
```

返回

```
 "hits" : [
      {
        "_source" : {"number" : 3}
      },
      {
        "_source" : {"number" : 4}
      },
      {
        "_source" : {"number" : 5}
      },
      ...
```

3. 指定文档位置

将 _id 为 1 的文档排在第 3 位 ， 将 _id 为 9 的文档排在第 2 位

```
GET greeting/_search
{
  "query": {
    "ranked": {
      "query": {
        "match_all": {}
      },
      "rank": {
        "pos": [ {"1" : 3}, {"9" : 2} ]
      }
    }
  }
}
```

返回

```
 "hits" : [
      {
        "_source" : {"number" : 2}
      },
      {
        "_source" : {"number" : 9}
      },
      {
        "_source" : {"number" : 1}
      },
      {
        "_source" : {"number" : 3}
      },
      ...
```

API

参数	说明	类型	默认值
top	指定_id的文档尽量置顶	string []	-
block	指定_id的文档在结果集中排除	string []	-
pos	精确指定文档排列位置，优先级高于 top	{ string : number } []	-

限制

- block 最多处理 1024 个文档。
- top+pos 合计最多处理 1024 个文档。

影响

- 被处理的文档将丧失原始分 _score。

存储层迁移 Store Tier

控制一个或多个索引的存储层冷热迁移。

> **TIP**
> 提交迁移前建议做一次索引的flush操作，减少增量数据拷贝过程，可大幅提升迁移效率。

API

迁移至热存储

`POST /{index}/_hot`

迁移至冷存储

`POST /{index}/_warm`

参数	说明	类型	默认值
index	索引名，支持单个索引、通配符 (*)或逗号分隔的索引列表。	string	-

Java High Level REST Client

Java High Level REST Client 至少需要Java 1.8，它的主要目标是封装Elasticsearch REST API 提供易于编写的代码实现。

脚手架

```
git@github.com:nasuyun/example-springboot.git

cd example-springboot

vi ./src/main/resources/application.properties # 替换成你的应用用户名密码

mvn clean package

java -jar ./target/example-springboot-0.0.1.jar
```

依赖

- **maven 依赖**

`pom.xml`

```
        <dependency>
            <groupId>org.elasticsearch</groupId>
            <artifactId>elasticsearch</artifactId>
            <version>6.8.23</version>
        </dependency>
        <dependency>
            <groupId>org.elasticsearch.client</groupId>
            <artifactId>elasticsearch-rest-high-level-client</artifactId>
            <version>6.8.23</version>
        </dependency>
```

- **gradle 依赖**

`build.gradle`

```
dependencies {
    compile 'org.elasticsearch:elasticsearch:6.8.23'
    compile 'org.elasticsearch.client:elasticsearch-rest-high-level-client:6.8.23'
}
```

初始化客户端

首选 我们需要创建一个 RestHighLevelClient 实例，然后使用该实例进行索引的增删改查操作。

```
final String server = "https://router.nasuyun.com:9200";
final String username = "your_app_username";
final String password = "your_app_password";

// 构建ES客户端
RestHighLevelClient client() {
    Header[] headers = new Header[]{new BasicHeader(HttpHeaders.CONTENT_TYPE,
"application/json")};
    final CredentialsProvider credentialsProvider = new BasicCredentialsProvider();
    credentialsProvider.setCredentials(AuthScope.ANY, new
UsernamePasswordCredentials(username, password));
    RestClientBuilder builder = RestClient.builder(HttpHost.create(server))
            .setDefaultHeaders(headers)
            .setHttpClientConfigCallback(httpClientBuilder ->
httpClientBuilder.setDefaultCredentialsProvider(credentialsProvider));
    return new RestHighLevelClient(builder);
}
```

正确的关闭客户端

在使用RestHighLevelClient进行相关操作后，我们需要根据场景正确的调用`client.close()`释放它的资源。

- **低频场景** 例如创建索引、删除索引、修改mapping等元数据操作， 应采用`用完即释放`的方式。

```
try (RestHighLevelClient client = client()) {
    // 操作
    client.dosomething ...
} catch (Exception e) {
    log.error("", e);
}
// try-with-resource 自动调用 client.clsoe()释放
```

- **高频场景** 例如持续高并发的写入或查询索引，应采用`连接复用`的方式

```
final RestHighLevelClient client;

// 在构造阶段创建RestHighLevelClient实例
Operator() {
    client = client();
}

void write() {
    bulk by client ...
}

void search() {
    search by client ...
}

// 当业务应用关闭时调用close
void close() {
    client.close();
}
```

> **TIP**
>
> **高频场景**：应避免频繁创建client导致socket端口耗尽。

索引操作

创建索引

```
try (RestHighLevelClient client = client()) {
    boolean exists = client.indices().exists(new GetIndexRequest(index),
RequestOptions.DEFAULT);
    if (exists == false) {
        CreateIndexRequest request = new CreateIndexRequest(index);
        request.settings(Settings.builder()
                .put("index.number_of_shards", 1)
                .put("index.number_of_replicas", 0)
        );
        Map<String, Object> message = new HashMap<>();
        message.put("type", "text");
        Map<String, Object> properties = new HashMap<>();
        properties.put("message", message);
        Map<String, Object> mapping = new HashMap<>();
        mapping.put("properties", properties);
        request.mapping(mapping);
        CreateIndexResponse createIndexResponse = client.indices().create(request,
RequestOptions.DEFAULT);
        log.info("create index[{}], result[{}]", createIndexResponse.index(),
createIndexResponse.isAcknowledged());
    }
} catch (Exception e) {
    log.error("", e);
}
```

删除索引

```
try (RestHighLevelClient client = client()) {
    boolean exists = client.indices().exists(new GetIndexRequest(index),
RequestOptions.DEFAULT);
    if (exists) {
        DeleteIndexRequest request = new DeleteIndexRequest(index);
        AcknowledgedResponse Response = client.indices().delete(request,
RequestOptions.DEFAULT);
    }
} catch (Exception e) {
```

```
    log.error("", e);
}
```

写入索引

同步方式

```
BulkRequest request = new BulkRequest();
request.add(new IndexRequest(index).type("_doc").source(XContentType.JSON,
"message", UUID.randomUUID().toString()));
request.add(new IndexRequest(index).type("_doc").source(XContentType.JSON,
"message", UUID.randomUUID().toString()));
BulkResponse bulkResponse = client.bulk(request, RequestOptions.DEFAULT);
```

BulkRequest 可选的参数

- **超时时间**

提供字符串或`TimeValue`方式设置bulk的等待超时时间

```
request.timeout(TimeValue.timeValueMinutes(2));
request.timeout("2m");
```

- **刷新策略**

```
request.setRefreshPolicy(WriteRequest.RefreshPolicy.WAIT_UNTIL);
```

策略	含义	默认
NONE	不刷新	默认
IMMEDIATE	立即强制刷新，此刷新策略不适用于生产环境高频率写入，常用于测试	
WAIT_UNTIL	写入请求在刷新完成前等待，刷新完成后结束。	

- **设置pipeline**

```
request.pipeline("pipelineId");
```

- **设置routing**

```
request.routing("routingId");
```

异步方式

```
client.bulkAsync(request, RequestOptions.DEFAULT, listener);
```

异步方法不会阻塞并立即返回。一旦完成，如果执行成功完成，将使用onResponse方法调用ActionListener；如果执行失败，则使用onFailure方法调用Action监听器。故障场景和预期异常与同步执行情况相同。

```
ActionListener<BulkResponse> listener = new ActionListener<BulkResponse>() {
    @Override
    public void onResponse(BulkResponse bulkResponse) {

    }

    @Override
    public void onFailure(Exception e) {

    }
};
```

Bulk 响应结果

```
for (BulkItemResponse bulkItemResponse : bulkResponse) {
    DocWriteResponse itemResponse = bulkItemResponse.getResponse();

    switch (bulkItemResponse.getOpType()) {
    case INDEX:
    case CREATE:
        IndexResponse indexResponse = (IndexResponse) itemResponse;
        break;
    case UPDATE:
        UpdateResponse updateResponse = (UpdateResponse) itemResponse;
```

```
            break;
        case DELETE:
            DeleteResponse deleteResponse = (DeleteResponse) itemResponse;
        }
    }
```

响应失败处理

- 至少有一个失败

```
if (bulkResponse.hasFailures()) {

}
```

- 失败条目处理

```
for (BulkItemResponse bulkItemResponse : bulkResponse) {
    if (bulkItemResponse.isFailed()) {
        BulkItemResponse.Failure failure = bulkItemResponse.getFailure();
    }
}
```

获取文档

Get 单个文档

```
GetRequest request = new GetRequest("index", "_doc", "1");
GetResponse getResponse = client.get(request, RequestOptions.DEFAULT);
String message = getResponse.getField("message").getValue();
```

Get 多个文档

```
MultiGetRequest request = new MultiGetRequest();
request.add(new MultiGetRequest.Item("index", "type", "example_id"));
request.add(new MultiGetRequest.Item("index", "type", "another_id"));
MultiGetResponse response = client.mget(request, RequestOptions.DEFAULT);
MultiGetItemResponse item = response.getResponses()[0];
String value = item.getResponse().getField("foo").getValue();
```

简单查询

简单的查询语句可以采用 RestHighLevelClient 拼SearchSourceBuilder方式。

```
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();
searchSourceBuilder.size(10);
SearchRequest searchRequest = new SearchRequest(index);
searchRequest.source(searchSourceBuilder);
SearchResponse searchResponse = client.search(searchRequest,
RequestOptions.DEFAULT);
```

响应结果 SearchRespone

通过执行搜索返回的SearchResponse提供了有关搜索执行本身的详细信息以及对返回文档的访问。首先，有关于请求执行本身的有用信息，如HTTP状态代码、执行时间或请求是否提前终止或超时：

```
RestStatus status = searchResponse.status();
TimeValue took = searchResponse.getTook();
Boolean terminatedEarly = searchResponse.isTerminatedEarly();
boolean timedOut = searchResponse.isTimedOut();
```

其次，SearchResponse还提供了分片总数以及成功分片与失败分片的统计信息，提供分片级别的执行信息。也可以通过ShardSearchFailures数组迭代处理可能的故障，如下例所示：

```
int totalShards = searchResponse.getTotalShards();
int successfulShards = searchResponse.getSuccessfulShards();
int failedShards = searchResponse.getFailedShards();
for (ShardSearchFailure failure : searchResponse.getShardFailures()) {
    // failures should be handled here
}
```

取出结果

```
SearchHits hits = searchResponse.getHits();
long totalHits = hits.getTotalHits();
float maxScore = hits.getMaxScore();
```

```
SearchHit[] searchHits = hits.getHits();
for (SearchHit hit : searchHits) {
    // do something with the SearchHit
}
```

SearchHit提供对基本信息的访问，如索引、类型、docId和每次搜索命中的分数：

```
String index = hit.getIndex();
String type = hit.getType();
String id = hit.getId();
float score = hit.getScore();
```

此外，它还允许您以简单的JSON字符串或键/值对映射的形式获取文档源。在此映射中，常规字段由字段名称设置关键帧，并包含字段值。多值字段作为对象列表返回，嵌套对象作为另一个键/值映射返回。这些情况需要进行相应处理：

```
String sourceAsString = hit.getSourceAsString();
Map<String, Object> sourceAsMap = hit.getSourceAsMap();
String documentTitle = (String) sourceAsMap.get("title");
List<Object> users = (List<Object>) sourceAsMap.get("user");
Map<String, Object> innerObject =
        (Map<String, Object>) sourceAsMap.get("innerObject");
```

高亮处理

如果请求，可以从结果中的每个SearchHit检索突出显示的文本片段。命中对象提供对HighlightField实例的字段名称映射的访问，每个实例包含一个或多个突出显示的文本片段

```
SearchHits hits = searchResponse.getHits();
for (SearchHit hit : hits.getHits()) {
    Map<String, HighlightField> highlightFields = hit.getHighlightFields();
    HighlightField highlight = highlightFields.get("title");
    Text[] fragments = highlight.fragments();
    String fragmentString = fragments[0].string();
}
```

取出聚合计算结果

通过Aggregations取出聚合对象，然后按名称获取聚合。

```
Aggregations aggregations = searchResponse.getAggregations();
Terms byCompanyAggregation = aggregations.get("by_company");
Bucket elasticBucket = byCompanyAggregation.getBucketByKey("Elastic");
Avg averageAge = elasticBucket.getAggregations().get("average_age");
double avg = averageAge.getValue();
```

复杂查询

涉及到复杂的查询语句，采用Java High Level REST Client 构建 SearchSourceBuilder 往往不那么直观，我们可以采用更高效的Low Level方式：

1. 在Kibana调试符合预期的查询语句。
2. 如果查询语句需要传入参数，利用占位符替换产生最终查询语句。
3. 调用 Java Low Level REST Client 调用获取查询结果，采用JSON工具类库解析结果集。

示例

一个稍显复杂的Query，存放在项目的资源目录下 `/resource/agent-network.json`

```
{
  "size": 0,
  "query": {
    "bool": {
      "filter": [
        {
          "range": {
            "@timestamp": {
              "gte": ${from},
              "lte": ${to},
              "format": "epoch_millis"
            }
          }
        },
        {
          "term": {
            "nginx.access.app_name": {
              "value": "${appname}"
            }
          }
        },
        {
          "exists": {
            "field":"nginx.access.user_agent.original"
          }
        }
      ]
    }
```

```
  },
  "aggs": {
    ...
  }
}
```

该示例带有3个参数 日期类型的 `from` `to` 和keyword类型的 `appname`

占位符填充参数值

```
public class TemplateFileUtils {
    public static String readFile(String classpathFile) {
        Resource resource = new ClassPathResource(classpathFile);
        try {
            return FileCopyUtils.copyToString(new
InputStreamReader(resource.getInputStream(), UTF_8));
        } catch (IOException e) {
            throw new RuntimeException("read file io error:" + classpathFile, e);
        }
    }

    public static String replaceHolder(String source, Map<String, Object> params) {
        StrSubstitutor sub = new StrSubstitutor(params, "${", "}");
        return sub.replace(source);
    }
}
```

> 该工具依赖 commons-lang3
>
> ```
> <dependency>
>     <groupId>org.apache.commons</groupId>
>     <artifactId>commons-lang3</artifactId>
> </dependency>
> ```

构造 Java Low Level REST Client

```
RestClientBuilder restClientBuilder() {
    Header[] headers = new Header[]{new BasicHeader(HttpHeaders.CONTENT_TYPE,
"application/json")};
    final CredentialsProvider credentialsProvider = new BasicCredentialsProvider();
    credentialsProvider.setCredentials(AuthScope.ANY, new
UsernamePasswordCredentials(username, password));
```

```
    RestClientBuilder builder = RestClient.builder(HttpHost.create(server))
            .setDefaultHeaders(headers)
            .setHttpClientConfigCallback(httpClientBuilder ->
httpClientBuilder.setDefaultCredentialsProvider(credentialsProvider));
    return builder;
}
```

调用查询

```
String template = TemplateFileUtils.readFile("agent-network.json");
Calendar c = Calendar.getInstance();
c.add(Calendar.HOUR, -hour);
Date from = c.getTime();
Date to = new Date();
Map<String, Object> params = Map.of(
        "from", from.getTime(),
        "to", to.getTime(),
        "appname", "foo");
String query = TemplateFileUtils.replaceHolder(template, params);

try (RestClient restClient = restClientBuilder.build()) {
    Request request = new Request("GET", "/_search");
    request.setJsonEntity(query);
    Response response = restClient.performRequest(request);
    ...
}
```

Python Client

连接测试

命令行方式测试连接性

```
➜  ~ python
Python 3.9.13 (main, May 24 2022, 21:28:31)
>>> from elasticsearch import Elasticsearch
>>> client = Elasticsearch(['https://router.nasuyun.com:9200'], http_auth=('应用用户名', '用户密码'))
>>> client.info()
{'name': 'xxx-xxx-xxx-xxx', 'cluster_name': 'free', 'cluster_uuid': 'xxxxxxx',
'version': {'number': '6.8.23', 'build_flavor': 'default', 'build_type': 'docker',
'build_hash': 'a6ca469', 'build_date': '2022-09-18T06:39:32.583Z',
'build_snapshot': False, 'lucene_version': '7.7.3',
'minimum_wire_compatibility_version': '5.6.0',
'minimum_index_compatibility_version': '5.0.0'}, 'tagline': 'You Know, for Search'}
```

创建索引文档

创建 index.py

index.py

```
from datetime import datetime
from elasticsearch import Elasticsearch
es = Elasticsearch(['https://router.nasuyun.com:9200'], http_auth=('应用用户名', '用户密码'))

doc = {
    'author': 'author_name',
    'text': 'Interesting content...',
    'timestamp': datetime.now(),
}
res = es.index(index="test-index", id=1, body=doc)
print(res['result'])
```

执行

```
python index.py
```

获取索引文档

get.py

```
from datetime import datetime
from elasticsearch import Elasticsearch
es = Elasticsearch(['https://router.nasuyun.com:9200'], http_auth=('应用用户名', '用户密码'))

res = es.get(index="test-index", id=1)
print(res['_source'])
```

搜索文档

search.py

```
from datetime import datetime
from elasticsearch import Elasticsearch
es = Elasticsearch(['https://router.nasuyun.com:9200'], http_auth=('应用用户名', '用户密码'))

res = es.search(index="test-index")
print("Got %s Hits:" % res['hits'])
```

返回

```
Got {'total': 1, 'max_score': 1.0, 'hits': [{'_index': 'test-index', '_type':
'_doc', '_id': '1', '_score': 1.0, '_source': {'author': 'author_name', 'text':
'Interesting content...', 'timestamp': '2023-02-11T20:37:03.494719'}}]} Hits:
```

Javascript Client

安装依赖

```
npm install @elastic/elasticsearch@6
npm install array.prototype.flatmap
```

连接

```
const { Client } = require('@elastic/elasticsearch')
const client = new Client({
  node: 'https://router.nasuyun.com:9200',
  auth: {
    username: '用户名称',
    password: '用户密码'
  }
})
```

写入索引

创建 bulk.js

bulk.js

```
'use strict'

require('array.prototype.flatmap').shim()
const { Client } = require('@elastic/elasticsearch')
const client = new Client({
  node: 'https://router.nasuyun.com:9200',
  auth: {
    username: '用户名称',
    password: '用户密码'
  }
})

 client.bulk({
```

```
      body: [
        { index:  { _index: 'test', _type: 'test', _id: 1 } },
        { title: 'foo' },
        { index:  { _index: 'test', _type: 'test', _id: 2 } },
        { title: 'bar' },
        { index:  { _index: 'test', _type: 'test', _id: 3 } },
        { title: 'zoo' },
      ]
    }, function (err, resp) {
        if(resp.errors) {
           console.log(JSON.stringify(resp, null, '\t'));
        }
    });
```

执行 bulk.js

```
node bulk.js
```

查询索引

创建 search.js

search.js

```
'use strict'

require('array.prototype.flatmap').shim()
const { Client } = require('@elastic/elasticsearch')
const client = new Client({
  node: 'https://router.nasuyun.com:9200',
  auth: {
    username: '用户名称',
    password: '用户密码'
  }
})

// callback API
client.search({
  index: 'test',
  body: {
    query: {
      match_all: {}
    }
```

```
  }
}, (err, result) => {
  console.log(result.body.hits)
  if (err) console.error(err)
})
```

执行及返回

```
node search.js

{
    "total": 3,
    "max_score": 1,
    "hits": [
        {
            "_index": "test",
            "_type": "test",
            "_id": "1",
            "_score": 1,
            "_source": {
                "title": "foo"
            }
        },
        {
            "_index": "test",
            "_type": "test",
            "_id": "2",
            "_score": 1,
            "_source": {
                "title": "bar"
            }
        },
        {
            "_index": "test",
            "_type": "test",
            "_id": "3",
            "_score": 1,
            "_source": {
                "title": "zoo"
            }
        }
    ]
}
```

Go Client

安装

可以安装go客户端的7.x版本（兼容纳速云Elasticsearch Serverless），请将软件包添加到`go.mod`文件中。

```
require github.com/elastic/go-elasticsearch/v7 7.16
```

连接示例

```
package main

import (
"log"
"github.com/elastic/go-elasticsearch/v7"
)


func main() {
cfg := elasticsearch.Config{
    Addresses: []string{
        "https://router.nasuyun.com:9200",
    },
    Username: <your_app_username>,
    Password: <your_app_password>,
}

es, err := elasticsearch.NewClient(cfg)
if err != nil {
  log.Fatalf("Error creating the client: %s", err)
}

res, err := es.Info()
if err != nil {
  log.Fatalf("Error getting response: %s", err)
}

defer res.Body.Close()
  log.Println(res)
```

```
}

func HttpExample(w http.ResponseWriter, r *http.Request) {
    ... # Client usage
}
```