Elasticsearch7.6学习指南

官方参考资料：

https://www.elastic.co/guide/en/elasticsearch/reference/7.6/index.html

https://www.elastic.co/guide/en/kibana/7.6/index.html

https://www.elastic.co/guide/en/logstash/7.6/index.html

https://docs.spring.io/spring-data/elasticsearch/docs/4.0.9.RELEASE/reference/html/#preface

1 Elasticsearch简介

Elasticsearch是Elastic Stack核心的分布式搜索和分析引擎。Logstash和Beats有助于收集，聚合和丰富您的数据并将其存储在Elasticsearch中。使用Kibana，您可以交互式地探索，可视化和共享对数据的见解，并管理和监视堆栈。Elasticsearch是建立索引，搜索和分析的地方。

Elasticsearch为所有类型的数据提供实时搜索和分析。无论您是结构化文本还是非结构化文本，数字数据或地理空间数据，Elasticsearch都能以支持快速搜索的方式有效地对其进行存储和索引。您不仅可以进行简单的数据检索，还可以聚合信息来发现数据中的趋势和模式。随着数据和查询量的增长，Elasticsearch的分布式特性使您的部署可以随之无缝地增长。

尽管并非*每个*问题都是搜索问题，但是Elasticsearch都提供了速度和灵活性来处理各种用例中的数据：

- 将搜索框添加到应用或网站
- 存储和分析日志，指标和安全事件数据
- 使用机器学习自动实时建模数据行为
- 使用Elasticsearch作为存储引擎自动化业务工作流程
- 使用Elasticsearch作为地理信息系统（GIS）管理，集成和分析空间信息
- 使用Elasticsearch作为生物信息学研究工具来存储和处理遗传数据

人们不断使用搜索的新颖方式使我们不断感到惊讶。但是，无论您的用例与其中之一相似，还是正在使用Elasticsearch解决新问题，在Elasticsearch中处理数据，文档和索引的方式都是相同的。

1.1 数据输入：文档和索引

Elasticsearch是一个分布式文档存储。Elasticsearch不会将信息存储为列数据的行，而是存储已序列化为JSON文档的复杂数据结构。当集群中有多个Elasticsearch节点时，存储的文档将分布在集群中，并且可以从任何节点立即访问。

存储文档时，将在1秒钟内几乎实时地对其进行索引和完全搜索。Elasticsearch使用倒排索引的数据结构，该结构支持非常快速的全文本搜索。反向索引列出了出现在任何文档中的每个唯一单词，并标识了每个单词出现的所有文档。

索引可以认为是文档的优化集合，每个文档都是字段的集合，这些字段是包含数据的键值对。默认情况下，Elasticsearch对每个字段中的所有数据建立索引，并且每个索引字段都具有专用的优化数据结构。例如，文本字段存储在倒排索引中，数字字段和地理字段存储在BKD树中。使用按字段数据结构组合并返回搜索结果的能力使Elasticsearch如此之快。

Elasticsearch还具有无模式的能力，这意味着无需显式指定如何处理文档中可能出现的每个不同字段即可对文档建立索引。启用动态映射后，Elasticsearch自动检测并向索引添加新字段。这种默认行为使索引和浏览数据变得容易-只需开始建立索引文档，Elasticsearch就会检测布尔值，浮点数和整数值，日期和字符串并将其映射到适当的Elasticsearch数据类型。

但是，最终，您比Elasticsearch更了解您的数据以及如何使用它们。您可以定义规则来控制动态映射，也可以显式定义映射以完全控制字段的存储和索引方式。

定义自己的映射使您能够：

- 区分全文字符串字段和精确值字符串字段
- 执行特定于语言的文本分析
- 优化字段以进行部分匹配
- 使用自定义日期格式
- 使用无法自动检测到的数据类型，例如`geo_point`和`geo_shape`

为不同的目的以不同的方式对同一字段建立索引通常很有用。例如，您可能希望将一个字符串字段索引为全文搜索的文本字段和索引关键字，以便对数据进行排序或汇总。或者，您可能选择使用多个语言分析器来处理包含用户输入的字符串字段的内容。

在搜索时也会使用在索引期间应用于全文字段的分析链。当您查询全文字段时，对查询文本进行相同的分析，然后才能在索引中查找词语。

1.2 信息输出：搜索和分析

尽管您可以将Elasticsearch用作文档存储并检索文档及其元数据，但真正的强大之处在于能够轻松访问基于Apache Lucene搜索引擎库构建的全套搜索功能。

Elasticsearch提供了一个简单，一致的REST API，用于管理您的集群以及索引和搜索数据。为了进行测试，您可以轻松地直接从命令行或通过Kibana中的开发者控制台提交请求。在您的应用程序中，您可以使用Elasticsearch客户端作为您选择的语言：Java，JavaScript，Go，.NET，PHP，Perl，Python或Ruby。

搜索数据

Elasticsearch REST API支持结构化查询，全文查询和结合了两者的复杂查询。结构化查询类似于您可以在SQL中构造的查询类型。例如，您可以搜索索引中的`gender`和`age`字段，`employee`然后按`hire_date`字段对匹配项进行排序。全文查询会找到所有与查询字符串匹配的文档，并按*相关性对*它们进行归还（它们与您的搜索词的匹配程度如何）。

除了搜索单个词语外，您还可以执行短语搜索，相似性搜索和前缀搜索，并获得自动完成建议。

是否要搜索地理空间或其他数字数据？Elasticsearch在支持高性能地理和数字查询的优化数据结构中索引非文本数据。

您可以使用Elasticsearch全面的JSON样式查询语言（Query DSL）访问所有这些搜索功能。您还可以构造SQL样式的查询，以在Elasticsearch内部本地搜索和聚合数据，并且JDBC和ODBC驱动程序使范围广泛的第三方应用程序可以通过SQL与Elasticsearch进行交互。

分析数据

Elasticsearch聚合使您能够构建数据的复杂摘要，并深入了解关键指标，模式和趋势。通过汇总，您不仅可以找到谚语“大海捞针”，还可以回答以下问题：

- 大海捞针有多少根？
- 针的平均长度是多少？
- 针头的中位长度是多少，由制造商细分？
- 在过去六个月的每个月中，有多少根针被添加到干草堆中？

您还可以使用聚合来回答更细微的问题，例如：

- 您最受欢迎的针头制造商是哪些？

- 是否有异常或异常的针团?

由于聚合利用用于搜索的相同数据结构，因此它们也非常快。这使您可以实时分析和可视化数据。您的报告和仪表板会随着数据的更改而更新，因此您可以根据最新信息采取措施。

而且，聚合与搜索请求一起运行。您可以在单个请求中同时对相同数据搜索文档，过滤结果并执行分析。而且由于聚合是在特定搜索的上下文中计算的，因此您不仅显示了所有70针大小的针数，而且还显示了符合用户搜索条件的70针大小的针数-例如，所有尺寸的*不绣花*70针。

但是等等，还有更多

是否想自动分析您的时间序列数据？您可以使用机器学习功能在数据中创建正常行为的准确基准，并识别异常模式。通过机器学习，您可以检测到：

- 与值，计数或频率的时间偏差有关的异常
- 统计稀有度
- 人口成员的异常行为

最好的部分是？您无需指定算法，模型或其他与数据科学相关的配置即可执行此操作。

1.3 可扩展性和弹性

Elasticsearch旨在始终可用并根据您的需求扩展。它是通过自然分布来实现的。您可以将服务器（节点）添加到集群以增加容量，Elasticsearch会自动在所有可用节点之间分配数据和查询负载。无需大修您的应用程序，Elasticsearch知道如何平衡多节点集群以提供扩展性和高可用性。节点越多，越好。

这是如何运作的？在幕后，Elasticsearch索引实际上只是一个或多个物理碎片的逻辑分组，其中每个碎片实际上是一个独立的索引。通过将文档分布在多个分片中的索引中，然后将这些分片分布在多个节点中，Elasticsearch可以确保冗余，这既可以防止硬件故障，又可以在将节点添加到集群中时提高查询能力。随着集群的增长（或收缩），Elasticsearch会自动迁移碎片以重新平衡集群。

分片有两种类型：主数据库和副本数据库。索引中的每个文档都属于一个主分片。副本分片是主分片的副本。副本可提供数据的冗余副本，以防止硬件故障并提高处理读取请求（例如搜索或检索文档）的能力。

创建索引时，索引中主碎片的数量是固定的，但是副本碎片的数量可以随时更改，而不会中断索引或查询操作。

这取决于.....

在分片大小和为索引配置的主分片数量方面，存在许多性能方面的考虑和权衡取舍。碎片越多，维护这些索引的开销就越大。分片大小越大，当Elasticsearch需要重新平衡集群时，移动分片所需的时间就越长。

查询很多小的分片会使每个分片的处理速度更快，但是更多的查询意味着更多的开销，因此查询较小数量的大分片可能会更快。简而言之...要视情况而定。

作为起点：

- 旨在将平均分片大小保持在几GB到几十GB之间。对于具有基于时间的数据的用例，通常会看到20GB到40GB范围内的碎片。
- 避免庞大的碎片问题。节点可以容纳的分片数量与可用堆空间成比例。通常，每GB堆空间中的分片数量应少于20。

确定用例最佳配置的最佳方法是通过使用自己的数据和查询进行测试。

在灾难的情况下

出于性能原因，群集内的节点必须位于同一网络上。跨不同数据中心中的节点的群集中的碎片平衡太简单了。但是高可用性架构要求您避免将所有鸡蛋都放在一个篮子里。如果一个位置发生重大故障，则另一个位置的服务器需要能够接管。无缝地。答案？跨集群复制（CCR）。

CCR提供了一种方法，可以自动将索引从主群集同步到可以用作热备份的辅助远程群集。如果主群集出现故障，则辅助群集可以接管。您还可以使用CCR创建辅助群集，以接近地理位置的方式向您的用户提供读取请求。

跨集群复制是主动-被动的。主群集上的索引是活动的领导者索引，并处理所有写请求。复制到辅助群集的索引是只读关注者。

照顾和喂养

与任何企业系统一样，您需要工具来保护，管理和监视Elasticsearch集群。集成到Elasticsearch中的安全性，监视和管理功能使您可以将Kibana用作控制中心来管理集群。类似的特征数据汇总和指标生命周期管理可帮助您明智随着时间的推移管理您的数据。

2 Elasticsearch入门

准备好试一试Elasticsearch了吗，亲自看看如何使用REST APIs来存储、搜索和分析数据?

逐步完成本入门教程，以：

1. 启动并运行Elasticsearch集群
2. 为一些样本文件编制索引
3. 使用Elasticsearch查询语言搜索文档
4. 使用存储桶和指标聚合分析结果

需要更多内容吗?

查看Elasticsearch简介，学习词语，并了解Elasticsearch的工作原理。如果您已经熟悉Elasticsearch并想了解它如何与其余堆栈一起工作，则可能要跳到Elastic Stack教程以了解如何使用Elasticsearch，Kibana，Beats和Logstash。

2.1 运行Elasticsearch

要试用 Elasticsearch，您可以在 Elasticsearch Service上创建托管部署，或在您自己的 Linux、macOS 或 Windows 机器上设置多节点 Elasticsearch 集群。

在Elastic Cloud上运行

在Elasticsearch Service上创建部署时，该服务与Kibana和APM一起预配一个三节点Elasticsearch集群。

要创建部署：

1. 注册免费试用版，然后验证您的电子邮件地址。
2. 为您的帐户设置密码。
3. 单击**创建部署**。

创建部署后，就可以为一些文档建立索引了。

在Docker上运行

拉取镜像

```
docker pull docker.elastic.co/elasticsearch/elasticsearch:7.6.2
```

启动单节点集群

```
docker run -d \
-p 9200:9200 \
-p 9300:9300 \
-e "discovery.type=single-node" \
--name=elasticsearch \
docker.elastic.co/elasticsearch/elasticsearch:7.6.2
```

查看集群状态

使用cat health API验证集群是否正在运行。

cat API以比原始JSON更易于阅读的格式返回有关集群和索引的信息。

您可以通过向Elasticsearch REST API提交HTTP请求来直接与集群交互。

```
curl -XGET "localhost:9200/_cat/health?v&pretty"
```

```
[root@localhost elk]# curl -X GET "localhost:9200/_cat/health?v&pretty"
epoch      timestamp cluster       status node.total node.data shards pri
1662519525 02:58:45  elasticsearch green           1         1      9   9
[root@localhost elk]#
```

该响应应表明`elasticsearch`集群的状态为`green`并且它具有1个节点：

如果您仅运行单个Elasticsearch实例，则集群状态将保持黄色。单节点群集具有完整的功能，但是无法将数据复制到另一个节点以提供弹性。副本分片必须可用，群集状态为绿色。如果群集状态为红色，则某些数据不可用。

cURL的命令

对Elasticsearch的请求包含与任何HTTP请求相同的部分：

```
curl -X<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY_STRING>' -d '<BODY>'
```

本示例使用以下变量：

- VERB：适当的HTTP方式或动词。例如，`GET`，`POST`，`PUT`，`HEAD`，或`DELETE`。
- PROTOCOL：无论是`http`或`https`。如果您在Elasticsearch前面有一个HTTPS代理，或者您使用Elasticsearch安全功能来加密HTTP通信，请使用后者。
- HOST：Elasticsearch集群中任何节点的主机名。或者，`localhost`用于本地计算机上的节点。
- PORT：运行Elasticsearch HTTP服务的端口，默认为`9200`。
- PATH：API端点，可以包含多个组件，例如`_cluster/stats`或`_nodes/stats/jvm`。
- QUERY_STRING：任何可选的查询字符串参数。例如，`?pretty`将*漂亮地打印*JSON响应以使其更易于阅读。
- BODY：JSON编码的请求正文（如有必要）。

如果启用了Elasticsearch安全功能，则还必须提供有权运行API的有效用户名（和密码）。

例如，使用`-u`或`--u`cURL命令参数。有关运行每个API所需的安全特权的详细信息，请参阅REST API。

Elasticsearch使用HTTP状态代码（例如）响应每个API请求`200 OK`。除`HEAD`请求外，它还返回JSON编码的响应正文。

其他安装选项

从包安装Elasticsearch使您能够轻松地在本地安装和运行多个实例，以便您可以尝试一下。要运行一个实例，您可以在Docker容器中运行Elasticsearch，在Linux上使用DEB或RPM软件包安装Elasticsearch，在macOS上使用Homebrew进行安装，或者在Windows上使用MSI软件包安装程序进行安装。

有关更多信息，请参见*安装Elasticsearch*。

2.2 运行Kibana

Kibana是什么?

Kibana用于探索和可视化您的数据并管理Elastic Stack的所有内容。
无论您是普通用户还是管理员，Kibana通过提供三个关键功能使您的数据具有可操作性。

Kibana是：

- 一个开源分析和可视化平台。
  - 使用Kibana探索您的Elasticsearch数据，然后构建漂亮的视觉效果和仪表板。
- 用于管理Elastic Stack的UI。
  - 管理安全设置、分配用户角色、摄取快照、汇总数据等等——所有这些都是Kibana UI带来的便利。
- Elastic解决方案的集中式枢纽。
  - 从日志分析到文档发现再到SIEM，Kibana是访问这些功能和其他功能的门户。

在Docker上运行

拉取镜像

```
docker pull docker.elastic.co/kibana/kibana:7.6.2
```

启动服务

```
docker run -d \
--link 你的es容器的名称或容器ID:elasticsearch \
-p 5601:5601 \
--name=kibana \
docker.elastic.co/kibana/kibana:7.6.2
```

集群状态查看

通过浏览器访问你的Kibana，访问地址：http://ip:5601

进入控制台方式，如下图所示。

查看集群健康状态

```
GET /_cat/health?v
```

```
epoch      timestamp cluster       status node.total node.data shards pri
1662519593 02:59:53  elasticsearch green           1         1      9   9
```

查看节点状态

```
GET /_cat/nodes?v
```

```
ip         heap.percent ram.percent cpu load_1m load_5m load_15m node.role master name
172.18.0.2           56          94  10    0.12    0.16     0.30 dilm      *      3b533a0963d6
```

查看所有索引信息

```
GET /_cat/indices?v
```

```
health status index                           uuid                   pri rep docs.count
green  open   projectname-debug-2022.09.05    K0MSPddLQlG3OH9zKZL3kg   1   0         88
green  open   bank                           Hcbe2TM0TImNcw7md7Nm8A   1   0       1000
green  open   .kibana_task_manager_1          QPSUplpSRyqIoLxP00bH-Q   1   0          2
green  open   .apm-agent-configuration        iJ4xBRnTSvysZyVI5Vzg0g   1   0          0
green  open   projectname-error-2022.09.05    2Ed0EhHkTAKaRdN00wUEXg   1   0          1
green  open   .kibana_1                       Fn6_3WuDQaO4ovjRKNzPwA   1   0         32
green  open   projectname-business-2022.09.05 -W4TCqpjTXyZFpKCpAhCoA   1   0         19
green  open   customer                       7IXHSVkiTqSbfkl3kSOAXA   1   0          1
green  open   projectname-record-2022.09.05   Lo5pUjiASGyLVLaLrNtW5Q   1   0          3
```

2.3 核心概念

想要学好、用好Elasticsearch，首先要了解其核心概念、名词和属性。这就好比想要看懂地图，首先要知道地图里常用的标记符号一样。 Elasticsearch的核心概念有Node 、Cluster 、Shards 、Replicas 、Index、Type、Document、Settings、Mapping和Analyzer，其含义分别如下所示。

- Node：即节点。节点是组成Elasticsearch集群的基本服务单元，集群中的每个运行中的Elasticsearch服务器都可称之为节点。
- Cluster ：即集群。Elasticsearch的集群是由具有相同cluster.name （默认值为elasticsearch）的一个或多个Elasticsearch节点组成的，各个节点协同工作，共享数据。同一个集群内节点的名字不能重复，但集群名称一定要相同。 在实际使用Elasticsearch集群时，一般需要给集群起一个合适的名字来替代cluster.name的默认值。自定义集群名称的好处是，可以防止一个新启动的节点加入相同网络中的另一个同名的集群中。 在Elasticsearch集群中，节点的状态有Green、Yellow和Red三种，分别如下所述。
    - Green：绿色，表示节点运行状态为健康状态。所有的主分片和副本分片都可以正常工作。
    - Yellow：黄色，表示节点的运行状态为预警状态。所有的主分片都可以正常工作，但至少有一个副本分片是不能正常工作的。此时集群依然可以正常工作，但集群的高可用性在某种程度上被弱化。
    - Red：红色，表示集群无法正常使用。此时，集群中至少有一个分片的主分片及它的全部副本分片都不可正常工作。虽然集群的查询操作还可以进行，但是也只能返回部分数据（其他正常分片的数据可以返回），而分配到这个有问题分片上的写入请求将会报错，最终导致数据丢失。
- Shards：即分片。当索引的数据量太大时，受限于单个节点的内存、磁盘处理能力等，节点无法足够快地响应客户端的请求，此时需要将一个索引上的数据进行水平拆分。拆分出来的每个数据部分称之为一个分片。一般来说，每个分片都会放到不同的服务器上。
    - 当软件开发人员在一个设置有多分片的索引中写入数据时，是通过路由来确定具体写入哪个分片中的，因此在创建索引时需要指定分片的数量，并且分片的数量一旦确定就不能更改。
    - 当软件开发人员在查询索引时，需要在索引对应的多个分片上进行查询。Elasticsearch会把查询发送给每个相关的分片，并汇总各个分片的查询结果。对上层的应用程序而言，分片是透明的，即应用程序并不知道分片的存在。
    - 在Elasticsearch中，默认为一个索引创建5个主分片，并分别为每个主分片创建一个副本。
- Replicas：即备份，也可称之为副本。副本指的是对主分片的备份，这种备份是精确复制模式。每个主分片可以有零个或多个副本， 主分片和备份分片都可以对外提供数据查询服务。当构建索引进行写入操作时，首先在主分片上完成数据的索引，然后数据会从主分片分发到备份分片上进行索引。
    - 当主分片不可用时，Elasticsearch会在备份分片中选举出一个分片作为主分片，从而避免数据丢失。
    - 一方面，备份分片既可以提升Elasticsearch系统的高可用性能，又可以提升搜索时的并发性能；
    - 另一方面，如果备份分片数量设置得太多，在写操作时会增加数据同步的负担。
- Index：即索引。在Elasticsearch中，索引由一个和多个分片组成。在使用索引时，需要通过索引名称在集群内进行唯一标识。
- Type：即类别。类别指的是索引内部的逻辑分区，通过Type的名字在索引内进行唯一标识。在查询时如果没有该值，则表示需要在整个索引中查询。
- Document：即文档。索引中的每一条数据叫作一个文档，与关系数据库的使用方法类似，一条文档数据通过_id在Type内进行唯一标识。
- Settings：Settings是对集群中索引的定义信息，比如一个索引默认的分片数、副本数等。
- Mapping：Mapping表示中保存了定义索引中字段（Field）的存储类型、分词方式、是否存储等信息，有点类似于关系数据库（如MySQL）中的表结构信息。
    - 在Elasticsearch中，Mapping是可以动态识别的。如果没有特殊需求，则不需要手动创建Mapping，因为Elasticsearch会根据数据格式自动识别它的类型。
    - 当需要对某些字段添加特殊属性时，如定义使用其他分词器、是否分词、是否存储等，就需要手动设置Mapping了。一个索引的Mapping一旦创建，若已经存储了数据，就不可修改了。

- Analyzer：Analyzer表示的是字段分词方式的定义。
  - 一个Analyzer通常由一个Tokenizer和零到多个Filter组成。
  - 在Elasticsearch中， 默认的标准Analyzer包含一个标准的Tokenizer和三个Filter，即Standard Token Filter、Lower Case Token Filter和Stop Token Filter。

2.4 索引文档

集群启动并运行后，就可以为某些数据建立索引了。Elasticsearch有多种摄取选项，但最终它们都做同样的事情：将JSON文档放入Elasticsearch索引中。

添加索引文档

您可以通过一个简单的PUT请求直接执行此操作，该请求指定要添加文档的索引，唯一的文档ID以及`"field": "value"`请求正文中的一对或多对：

```
PUT /customer/_doc/1
{
  "name": "John Doe"
}
```

或cUrl

```
curl -X PUT "localhost:9200/customer/_doc/1?pretty" -H 'Content-Type: application/json' -d'
{
  "name": "John Doe"
}
'
```

如果该请求`customer`尚不存在，该请求将自动创建该索引，添加ID为`1`的新文档，并存储该`name`字段并为其建立索引。

由于这是一个新文档，因此响应显示该操作的结果是创建了该文档的版本`1`

```
{
  "_index" : "customer",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 0,
  "_primary_term" : 1
}
```

查看索引文档

可以从群集中的任何节点立即使用新文档。您可以使用指定其文档ID的GET请求检索它：

```
GET /customer/_doc/1
```

或cUrl

```
curl -X GET "localhost:9200/customer/_doc/1?pretty"
```

该响应表明找到了具有指定ID的文档，并显示了已索引的原始源字段。

```
{
  "_index" : "customer",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 1,
  "_seq_no" : 0,
  "_primary_term" : 1,
  "found" : true,
  "_source" : {
    "name" : "John Doe"
  }
}
```

修改索引文档

```
POST /customer/_update/1
{
  "doc": { "name": "Jane Doe abd" }
}
```

响应结果

```
{
  "_index" : "customer",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 2,
  "result" : "updated",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 1,
  "_primary_term" : 1
}
```

再次查看索引文档

```
GET /customer/_doc/1
```

响应结果

```
{
  "_index" : "customer",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 2,
  "_seq_no" : 1,
  "_primary_term" : 1,
  "found" : true,
  "_source" : {
    "name" : "Jane Doe abd"
  }
}
```

删除索引文档

```
DELETE /customer/_doc/1
```

响应结果

```
{
  "_index" : "customer",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 3,
  "result" : "deleted",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 2,
  "_primary_term" : 1
}
```

再次查看索引文档

```
GET /customer/_doc/1
```

响应结果

```
{
  "_index" : "customer",
  "_type" : "_doc",
  "_id" : "1",
  "found" : false
}
```

批量索引文档

如果您有很多要编制索引的文档，则可以使用批量API批量提交。使用批量处理批处理文档操作比单独提交请求要快得多，因为它可以最大程度地减少网络往返次数。

```
POST /customer/_bulk
{"index":{"_id":"1"}}
{"name": "come on 的喂" }
{"index":{"_id":"2"}}
{"name": "ABC" }
```

响应结果

```
{
  "took" : 57,
  "errors" : false,
  "items" : [
    {
      "index" : {
        "_index" : "customer",
        "_type" : "_doc",
        "_id" : "1",
        "_version" : 1,
        "result" : "created",
        "_shards" : {
          "total" : 1,
          "successful" : 1,
          "failed" : 0
        },
        "_seq_no" : 3,
        "_primary_term" : 1,
        "status" : 201
      }
    },
    {
      "index" : {
        "_index" : "customer",
        "_type" : "_doc",
        "_id" : "2",
        "_version" : 1,
        "result" : "created",
        "_shards" : {
          "total" : 1,
          "successful" : 1,
          "failed" : 0
        },
        "_seq_no" : 4,
        "_primary_term" : 1,
        "status" : 201
      }
    }
  ]
}
```

索引文件批量

最佳批处理大小取决于许多因素：文档大小和复杂性，索引编制和搜索负载以及群集可用的资源。

一个好的起点是批处理1,000至5,000个文档，总有效负载在5MB至15MB之间。

从那里，您可以尝试找到最佳位置。

要将一些数据导入Elasticsearch，您可以开始搜索和分析：

1. 下载 accounts.json 样本数据集。此随机生成的数据集中的文档代表具有以下信息的用户帐户：

```
{
    "account_number": 0,
    "balance": 16623,
    "firstname": "Bradshaw",
    "lastname": "Mckenzie",
    "age": 29,
    "gender": "F",
    "address": "244 Columbus Place",
    "employer": "Euron",
    "email": "bradshawmckenzie@euron.com",
    "city": "Hobucken",
    "state": "CO"
}
```

2. `bank` 使用以下 `_bulk` 请求将帐户数据索引到索引中：

```
# 保证account.json文件在你执行命令所在的目录，如果account.json下载不到，可以使用资源目录下面的文件
curl -H "Content-Type: application/json" -XPOST "localhost:9200/bank/_bulk?pretty&refresh" \
--data-binary "@accounts.json"
# 查看所有索引
curl "localhost:9200/_cat/indices?v"
```

响应表明成功索引了1,000个文档。

```
[root@localhost elk]# curl "localhost:9200/_cat/indices?v"
health status index                            uuid                   pri rep docs.count docs.deleted store.size pri.store.size
green  open   projectname-debug-2022.09.05     K0MSPddLQlG3OH9zKZL3kg   1   0         88            0     99.4kb         99.4kb
yellow open   bank                             Hcbe2TM0TImNcw7md7Nm8A   1   1       1000            0    436.7kb        436.7kb
green  open   .kibana_task_manager_1           QPSUplpSRyqIoLxP00bH-Q   1   0          2            0     31.6kb         31.6kb
green  open   .apm-agent-configuration         iJ4xBRnTSvysZyVI5Vzg0g   1   0          0            0       283b           283b
green  open   projectname-error-2022.09.05     2Ed0EhHkTAKaRdN00wUEXg   1   0          1            0     13.8kb         13.8kb
green  open   .kibana_1                        Fn6_3WuDQaO4ovjRKNzPwA   1   0         31            1     24.4kb         24.4kb
green  open   projectname-business-2022.09.05 -W4TCqpjTXyZFpKCpAhCoA   1   0         19            0     80.9kb         80.9kb
green  open   projectname-record-2022.09.05   Lo5pUjiASGyLVLaLrNtW5Q   1   0          3            0     43.3kb         43.3kb
yellow open   customer                         7IXHSVkiTqSbfkl3kSOAXA   1   1          1            0      3.4kb          3.4kb
```

2.5 开始搜索

将一些数据摄取到Elasticsearch索引后，您可以通过将请求发送到 `_search` 端点来对其进行搜索。

要访问全套搜索功能，请使用Elasticsearch Query DSL在请求正文中指定搜索条件。

您可以在请求URI中指定要搜索的索引的名称。

match_all

例如，以下请求检索`bank`按帐号排序的索引中的所有文档：

```
GET /bank/_search
{
  "query": { "match_all": {} },
  "sort": [
    { "account_number": "asc" }
  ]
}
```

或cUrl

```
curl -X GET "localhost:9200/bank/_search?pretty" -H 'Content-Type:
application/json' -d'
{
  "query": { "match_all": {} },
  "sort": [
    { "account_number": "asc" }
  ]
}
'
```

默认情况下，`hits`响应部分包含与搜索条件匹配的前10个文档：

```
{
  "took" : 21,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1000,
      "relation" : "eq"
    },
    "max_score" : null,
    "hits" : [
      {
        "_index" : "bank",
        "_type" : "_doc",
        "_id" : "0",
        "_score" : null,
        "_source" : {
          "account_number" : 0,
          "balance" : 16623,
          "firstname" : "Bradshaw",
          "lastname" : "Mckenzie",
          "age" : 29,
          "gender" : "F",
```

```
          "address" : "244 Columbus Place",
          "employer" : "Euron",
          "email" : "bradshawmckenzie@euron.com",
          "city" : "Hobucken",
          "state" : "CO"
        },
        "sort" : [0]
      },
       . . . .
    ]
  }
}
```

该响应还提供有关搜索请求的以下信息：

- `took`：Elasticsearch运行查询多长时间（以毫秒为单位）
- `timed_out`：搜索请求是否超时
- `_shards`：搜索了多少个分片以及成功、失败或跳过了多少个分片。
- `max_score`：找到的最相关文件的分数
- `hits.total.value`：找到了多少个匹配的文档
- `hits.sort`：文档的排序位置（不按相关性得分排序时）
- `hits._score`：文档的相关性得分（使用时不适用`match_all`）

from & size

每个搜索请求都是独立的：Elasticsearch在请求中不维护任何状态信息。要翻阅搜索结果，请在您的请求中指定`from`和`size`参数。

例如，以下请求的匹配数为10到19：

```
GET /bank/_search
{
  "query": { "match_all": {} },
  "sort": [
    { "account_number": "asc" }
  ],
  "from": 10,
  "size": 10
}
```

或cUrl

```
curl -X GET "localhost:9200/bank/_search?pretty" -H 'Content-Type:
application/json' -d'
{
  "query": { "match_all": {} },
  "sort": [
    { "account_number": "asc" }
  ],
  "from": 10,
  "size": 10
}
'
```

match

既然您已经了解了如何提交基本的搜索请求，则可以开始构建比有趣的查询`match_all`。

要在字段中搜索特定词语，可以使用`match`查询。

例如，以下请求搜索该`address`字段以查找地址包含`mill`或的客户`lane`：

```
GET /bank/_search
{
  "query": { "match": { "address": "mill lane" } }
}
```

或cUrl

```
curl -X GET "localhost:9200/bank/_search?pretty" -H 'Content-Type:
application/json' -d'
{
  "query": { "match": { "address": "mill lane" } }
}
'
```

match_phrase

要执行词组搜索而不是匹配单个词，请使用`match_phrase`代替`match`。

例如，以下请求仅匹配包含短语的地址`mill lane`：

```
GET /bank/_search
{
  "query": { "match_phrase": { "address": "mill lane" } }
}
```

或cUrl

```
curl -X GET "localhost:9200/bank/_search?pretty" -H 'Content-Type:
application/json' -d'
{
  "query": { "match_phrase": { "address": "mill lane" } }
}
'
```

bool

要构造更复杂的查询，可以使用`bool`查询来组合多个查询条件。

您可以根据需要（必须匹配），期望（应该匹配）或不期望（必须不匹配）指定条件。

例如，以下请求在`bank`索引中搜索属于40岁客户的帐户，但不包括居住在爱达荷州（ID）的任何人：

```
GET /bank/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "age": "40" } }
      ],
      "must_not": [
        { "match": { "state": "ID" } }
      ]
    }
  }
}
```

或cUrl

```
curl -X GET "localhost:9200/bank/_search?pretty" -H 'Content-Type:
application/json' -d'
{
  "query": {
    "bool": {
      "must": [
        { "match": { "age": "40" } }
      ],
      "must_not": [
        { "match": { "state": "ID" } }
      ]
    }
  }
}
'
```

布尔查询中的每个`must`，`should`和`must_not`元素称为查询子句。

文档满足每个条款`must`或`should`条款的标准的程度有助于文档的*相关性得分*。分数越高，文档就越符合您的搜索条件。

默认情况下，Elasticsearch返回按这些相关性分数排名的文档。

filter

`must_not`子句中的条件被视为*过滤器*。它会影响文档是否包含在结果中，但不会影响文档的评分方式。

您还可以显式指定任意过滤器，以基于结构化数据包括或排除文档。

例如，以下请求使用范围过滤器将结果限制为余额在20,000美元到30,000美元（含）之间的帐户。

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

```
          "balance": {
            "gte": 20000,
            "lte": 30000
          }
        }
      }
    }
  }
}
```

或cUrl

```
curl -X GET "localhost:9200/bank/_search?pretty" -H 'Content-Type:
application/json' -d'
{
  "query": {
    "bool": {
      "must": { "match_all": {} },
      "filter": {
        "range": {
          "balance": {
            "gte": 20000,
            "lte": 30000
          }
        }
      }
    }
  }
}
'
```

2.6 使用汇总分析结果

Elasticsearch聚合使您能够获取有关搜索结果的元信息，并回答诸如“德克萨斯州有多少个帐户持有人”之类的问题。或“田纳西州的平均帐户余额是多少？”您可以在一个请求中搜索文档，过滤命中并使用汇总分析结果。

terms

例如，以下请求使用`terms`汇总将`bank`索引中的所有帐户按状态分组，并按降序返回帐户数量最多的十个州：

```
GET /bank/_search
{
  "size": 0,
  "aggs": {
    "group_by_state": {
      "terms": {
        "field": "state.keyword"
      }
    }
  }
}
```

这个响应中的`buckets`是`state`字段的值。`doc_count`表示的是每个州帐户数量。

例如，您可以看到`ID`（爱达荷州）有27个帐户。

因为请求设置`size=0`，所以响应仅包含聚合结果。


```
{
  "took" : 33,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1000,
      "relation" : "eq"
    },
    "max_score" : null,
    "hits" : [ ]
  },
  "aggregations" : {
    "group_by_state" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 743,
      "buckets" : [
        {
          "key" : "TX",
          "doc_count" : 30
        },
        {
          "key" : "MD",
          "doc_count" : 28
        },
        {
          "key" : "ID",
          "doc_count" : 27
        },
        {
          "key" : "AL",
          "doc_count" : 25
        },
        {
          "key" : "ME",
          "doc_count" : 25
        },
        {
          "key" : "TN",
          "doc_count" : 25
        },
        {
          "key" : "WY",
          "doc_count" : 25
        },
```

```
        {
          "key" : "DC",
          "doc_count" : 24
        },
        {
          "key" : "MA",
          "doc_count" : 24
        },
        {
          "key" : "ND",
          "doc_count" : 24
        }
      ]
    }
  }
}
```

avg

您可以组合聚合以构建更复杂的数据汇总。

例如，以下请求将一个`avg`聚合嵌套在先前的`group_by_state`聚合中，以计算每个状态的平均帐户余额。

```
GET /bank/_search
{
  "size": 0,
  "aggs": {
    "group_by_state": {
      "terms": {
        "field": "state.keyword"
      },
      "aggs": {
        "average_balance": {
          "avg": {
            "field": "balance"
          }
        }
      }
    }
  }
}
```

order

您可以通过指定`terms`聚合内的顺序来使用嵌套聚合的结果进行排序，而不是按计数对结果进行排序：

```
GET /bank/_search
{
  "size": 0,
  "aggs": {
    "group_by_state": {
      "terms": {
        "field": "state.keyword",
        "order": {
```

```
          "average_balance": "desc"
        }
      },
      "aggs": {
        "average_balance": {
          "avg": {
            "field": "balance"
          }
        }
      }
    }
  }
}
```

除了这些基本的存储桶和指标聚合外，Elasticsearch还提供了专门的聚合，用于在多个字段上操作并分析特定类型的数据，例如日期，IP地址和地理数据。您还可以将单个聚合的结果馈送到管道聚合中，以进行进一步分析。

聚合提供的核心分析功能可启用高级功能，例如使用机器学习来检测异常。

2.7 API公约

Elasticsearch暴露了REST API，这些API被UI组件使用，可以直接调用，以配置和访问Elasticsearch功能。

除非另有规定，本章中列出的惯例可以适用于整个REST API。

- 通用选项
- 多索引
- 索引名称中的日期数学支持
- Cron表达式
- 基于URL的访问控制

通用选项

以下选项可应用于所有 REST API。

美化响应结果

将?pretty=true附加到任何请求时，返回的JSON将被格式化（仅用于调试！）。

另一种选择是设置?format=yaml这将导致结果以（有时）更易读的 yaml 格式返回。

人类可读的输出

统计信息以适合人类（例如“exists_time”：“1h”或“size”：“1kb”）和计算值（例如“exists_time_in_millis”：3600000或“size_in_bytes”：1024）的格式返回。可以通过将 ?human=false 添加到查询字符串来关闭人类可读的值。当统计结果被监控工具使用而不是供人类使用时，这是有道理的。人工标志的默认值为 false。

日期数学

大多数接受格式化日期值的参数，例如范围查询中的gt和lt，或日期范围聚合中的from和to。
表达式以锚定日期开始，可以是现在，也可以是以 || 结尾的日期字符串。

此锚定日期可以选择后跟一个或多个数学表达式：

- `+1h`：添加1小时

- `-1d`: 减少1天
- `/d`: 向下舍入到最近的一天

支持的时间单位与持续时间的时间单位所支持的时间单位不同。支持的单位是：

单位	描述
y	年
M	月
w	周
d	日
h	时（12小时制）
H	时（24小时制）
m	分
s	秒

假设现在是2001-01-01 12:00:00，一些例子是：

示例	解析结果
`now+1h`	`now` 以毫秒为单位增加一小时。解析为：2001-01-01 13:00:00
`now-1h`	`now` 以毫秒为单位减少一小时。解析为：2001-01-01 11:00:00
`now-1h/d`	`now` 以毫秒为单位减少一小时，向下舍入到 UTC 00:00。解析为：2001-01-01 00:00:00
`2001.02.01||+1M/d`	`2001-02-01` 以毫秒为单位增加一个月。解析为：2001-03-01 00:00:00

响应过滤

所有 REST API 都接受一个filter_path参数，该参数可用于减少Elasticsearch返回的响应。此参数采用逗号分隔的过滤器列表，用点表示法表示：

```
GET /_search?q=elasticsearch&filter_path=took,hits.hits._id,hits.hits._score
```

响应结果

```
{
  "took" : 63,
  "hits" : {
    "hits" : [
      {
        "_id" : "8yM6DYMBWrBWp6-PkUWn",
        "_score" : 2.8408504
      },
      {
```

```
        "_id" : "FSNtDYMBWrBWp6-PGkZi",
        "_score" : 2.8408504
      },
      {
        "_id" : "LCOADYMBWrBWp6-PsObj",
        "_score" : 2.8408504
      },
      {
        "_id" : "QyOEDYMBWrBWp6-P8EZ7",
        "_score" : 2.8408504
      },
      {
        "_id" : "EiNtDYMBWrBWp6-PGkZi",
        "_score" : 2.3114512
      },
      {
        "_id" : "KSOADYMBWrBWp6-PsObj",
        "_score" : 2.3114512
      },
      {
        "_id" : "RiOEDYMBWrBWp6-P8EZ7",
        "_score" : 2.3114512
      }
    ]
  }
}
```

它还支持`*`通配符来匹配任何字段或字段名称的一部分：

```
GET /_cluster/state?filter_path=metadata.indices.*.stat*
```

响应结果

```
{
  "metadata" : {
    "indices" : {
      "bank" : {
        "state" : "open"
      },
      "customer" : {
        "state" : "open"
      }
    }
  }
}
```

并且`**`通配符可用于在不知道字段的确切路径的情况下包含字段。例如，我们可以通过这个请求返回每个段的 Lucene 版本：

```
GET /_cluster/state?filter_path=routing_table.indices.**.state
```

响应结果

```
{
  "routing_table" : {
    "indices" : {
      "bank" : {
        "shards" : {
          "0" : [
            {
              "state" : "STARTED"
            }
          ]
        }
      }
    }
  }
}
```

也可以通过在过滤器前加上 `-` 来排除一个或多个字段：

```
GET /_count?filter_path=-_shards
```

响应结果

```
{
  "count" : 1154
}
```

为了获得更多控制，可以在同一个表达式中组合包含和排除过滤器。在这种情况下，将首先应用独占过滤器，然后使用包含过滤器再次过滤结果：

```
GET /_cluster/state?filter_path=metadata.indices.*.state,-metadata.indices.projectname-*
```

响应结果

```
{
  "metadata" : {
    "indices" : {
      ".kibana_task_manager_1" : {
        "state" : "open"
      },
      ".kibana_1" : {
        "state" : "open"
      },
      "bank" : {
        "state" : "open"
      },
      ".apm-agent-configuration" : {
        "state" : "open"
      },
      "customer" : {
        "state" : "open"
      }
    }
```

```
  }
}
```

注意Elasticsearch有时会直接返回一个字段的原始值，比如`_source`字段。如果你想过滤`_source`字段，你应该考虑将已经存在的`_source`参数与filter_path参数结合起来，像这样。

```
POST /library/book?refresh
{"title": "Book #1", "rating": 200.1}
POST /library/book?refresh
{"title": "Book #2", "rating": 1.7}
POST /library/book?refresh
{"title": "Book #3", "rating": 0.1}
GET /_search?filter_path=hits.hits._source&_source=title&sort=rating:desc
```

查询响应结果

```
{
  "hits" : {
    "hits" : [
      {
        "_source" : {
          "title" : "Book #1"
        }
      },
      {
        "_source" : {
          "title" : "Book #2"
        }
      },
      {
        "_source" : {
          "title" : "Book #3"
        }
      }
    ]
  }
}
```

扁平化设置

flat_settings 标志会影响设置列表的呈现。当 flat_settings 标志为true时，设置以扁平格式返回：

```
GET /customer/_settings?flat_settings=true
```

响应结果

```
{
  "customer" : {
    "settings" : {
      "index.creation_date" : "1662522350526",
      "index.number_of_replicas" : "0",
      "index.number_of_shards" : "1",
      "index.provided_name" : "customer",
      "index.uuid" : "ewbC61BtQWmvoZu_J-HYrg",
      "index.version.created" : "7060299"
    }
  }
}
```

时间单位

每当需要指定持续时间时，例如对于 timeout 参数，duration 必须指定单位，如2d表示2天。支持的单位有：

单位	描述
d	天
h	小时
m	分钟
s	秒
ms	毫秒
micros	微秒
nanos	纳秒

字节大小单位

每当需要指定数据的字节大小时，例如设置缓冲区大小参数时，该值必须指定单位，例如10kb表示10 KB。请注意，这些单位使用1024的幂，因此1kb表示1024字节。支持的单位有：

单位	描述
b	一字节
kb	千字节
mb	兆字节
gb	吉字节
tb	太字节
pb	拍字节

无单位数量

无单位数量意味着它们没有像“字节”或“赫兹”或“米”或“长吨”这样的“单位”。
如果其中一个数量很大，我们会将其打印为 10m 对应 10,000,000 或 7k 对应 7,000。当我们的意思是 87 时，我们仍然会打印 87。这些是支持的乘数：

单位	描述
k	千
m	兆
g	吉
t	太
p	拍

距离单位

在需要指定距离的地方（例如 Geo-distance 中的距离参数），如果未指定，则默认单位为米。距离可以用其他单位指定，例如“1km”或“2mi”（2 英里）。
完整的单位列表如下：

单位	描述
mi 或 miles	英里
yd 或 yards	码
ft 或 feet	英尺
in 或 inch	英寸
km 或 kilometers	千米
m 或 meters	米
cm 或 centimeters	厘米
mm 或 millimeters	毫米
NM ， nmi 或 nauticalmiles	纳米

多索引

大多数引用索引参数的 API 支持跨多个索引执行，使用简单的 test1、test2、test3 表示法（或 `_all` 表示所有索引）。它还支持通配符，例如：`test*` 或 `*test` 或 `te*t` 或 `*test*`，以及“排除”(-) 的能力，例如：`test*,-test3`。

所有多索引 API 都支持以下 url 查询字符串参数：

- `ignore_unavailable`：忽略不可用的。
    - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为 false。
- `allow_no_indices`：允许缺失或关闭索引。

    - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
    - 此参数也适用于指向缺失或关闭索引的索引别名。
- `expand_wildcards`：通配符扩展性。
    - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
        - `all`：扩展到开放和关闭的索引。
        - `open`：仅扩展至开放索引。
        - `closed`：仅扩展到关闭索引。
        - `none`：不接受通配符表达式。

    上述参数的默认设置取决于所使用的 API。

    一些多索引 API 还支持以下 url 查询字符串参数：

- `ignore_throttled`：
    - （可选，布尔值）如果为 `true`，则在限制时忽略具体、扩展或别名索引。

> 注意：单索引APIs（比如文档 API 和单索引别名 API）是不支持多索引的。

索引名称中的日期数学支持

日期数学索引名称解析使您能够搜索一系列时间序列索引，而不是搜索所有时间序列索引并过滤结果或维护别名。限制搜索的索引数量可以减少集群的负载并提高执行性能。例如，如果您在日常日志中搜索错误，则可以使用日期数学名称模板将搜索限制为过去两天。

几乎所有具有索引参数的 API 都支持索引参数值中的日期数学。

日期数学索引名称采用以下形式：

```
<static_name{date_math_expr{date_format|time_zone}}>
```

- static_name：是名称的静态文本部分。
- date_math_expr：是动态计算日期的动态日期数学表达式。
- date_format：是显示计算日期的可选格式。默认为 yyyy.MM.dd。
    - 格式应与 java-time 兼容，下面是参考链接：
    - https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html
- time_zone：是可选的时区。默认为 UTC。

> 请注意 date_format 中使用的小写字母和大写字母的用法。
>
> 例如：mm 表示小时的分钟，而 MM 表示一年中的月份。
>
> 同样，hh 表示 1-12 范围内的小时与 AM/PM 组合，而 HH 表示 0-23 24 小时范围内的小时。

日期数学表达式与区域设置无关。因此，不能使用公历以外的任何其他日历。

您必须将日期数学索引名称表达式括在尖括号内，并且所有特殊字符都应进行 URI 编码。例如：

```
# GET /<logstash-{now/d}>/_search
GET /%3Clogstash-%7Bnow%2Fd%7D%3E/_search
{
  "query" : {
    "match": {
      "test": "data"
    }
  }
}
```

用于日期舍入的特殊字符必须采用 URI 编码，如下所示：

特殊符号	URI编码
<	%3C
>	%3E
/	%2F
{	%7B
}	%7D
|	%7C
+	%2B
:	%3A
,	%2C

以下示例显示了不同形式的日期数学索引名称以及它们解析为的最终索引名称，假如当前时间是 2024 年 3 月 22 日中午 UTC。

表达式	解析结果
`<logstash-{now/d}>`	logstash-2024.03.22
`<logstash-{now/M}>`	logstash-2024.03.01
`<logstash-{now/M{yyyy.MM}}>`	logstash-2024.03
`<logstash-{now/M-1M{yyyy.MM}}>`	logstash-2024.02
`<logstash-{now/d{yyyy.MM.dd+12:00}}>`	logstash-2024.03.23

要在索引名称模板的静态部分中使用字符 { 和 }，请使用反斜杠 \ 对它们进行转义，例如：

`<elastic\{ON\}-{now/M}>` 解析为 `elastic{ON}-2024.03.01`

以下示例显示了搜索过去三天的 Logstash 索引的搜索请求，假设索引使用默认的 Logstash 索引名称格式 logstash-yyyy.MM.dd。

```
# GET /<logstash-{now/d-2d}>,<logstash-{now/d-1d}>,<logstash-{now/d}>/_search
GET /%3Clogstash-%7Bnow%2Fd-2d%7D%3E%2C%3Clogstash-%7Bnow%2Fd-1d%7D%3E%2C%3Clogstash-%7Bnow%2Fd%7D%3E/_search
{
  "query" : {
    "match": {
      "test": "data"
    }
  }
}
```

Cron表达式

一个Cron表达式格式如下：

```
<seconds> <minutes> <hours> <day_of_month> <month> <day_of_week> [year]
```

Elasticsearch 使用Quartz Job Scheduler中的cron解析器。有关编写Quartz cron表达式的更多信息，请参阅 Quartz CronTrigger教程。
所有时间表时间均采用协调世界时 (UTC)，不支持其他时区。

> 小提示：您可以使用 elasticsearch-croneval 命令行工具来验证您的 cron 表达式。

除年份外，所有元素都是必需的。有关允许的特殊字符的信息，请参阅Cron 特殊字符。

- `<seconds>`：秒
  - (必选) 有效值: `0-59` 和特殊字符 `,` `-` `*` `/`
- `<minutes>`：分
  - (必选) 有效值: `0-59` 和特殊字符 `,` `-` `*` `/`
- `<hours>`：时
  - (必选) 有效值: `0-23` 和特殊字符 `,` `-` `*` `/`
- `<day_of_month>`：日
  - (必选) 有效值: `1-31` 和特殊字符 `,` `-` `*` `/` `?` `L` `W`
- `<month>`：月
  - (必选) 有效值: `1-12`, `JAN-DEC`, `jan-dec`, 和特殊字符 `,` `-` `*` `/`
- `<day_of_week>`：一周中的第几天
  - (必选) 有效值: `1-7`, `SUN-SAT`, `sun-sat`, 和特殊字符 `,` `-` `*` `/` `?` `L` `#`
- `<year>`：年
  - (可选) 有效值: `1970-2099` 和特殊字符 `,` `-` `*` `/`

特殊字符：

- `*`
  - 选择字段的所有可能值。例如，小时字段中的`*`表示“每小时”
- `?`
  - 没有具体价值。当您不关心值是什么时使用。例如，如果您希望计划在每月的特定日期触发，但不关心恰好是星期几，您可以指定?在day_of_week字段中。
- `-`

    - 范围值. 用于分隔最小值和最大值。例如，如果您希望计划在上午9:00到下午5:00之间每小时触发一次，则可以在小时字段中指定9-17。
- `,`
    - 多个值。用于分隔一个字段的多个值。例如，如果您希望计划在每周二和周四触发，您可以在day_of_week字段中指定TUE,THU。
- `/`
    - 周期性增量。指定时间增量时用于分隔值。第一个值代表起点，第二个值代表区间。例如，如果您希望计划从整点开始每20分钟触发一次，您可以在分钟字段中指定0/20。同样，在day_of_month字段中指定1/5将从该月的第一天开始每5天触发一次。
- `L`
    - 最后的。
    - 在day_of_month字段中使用表示该月的最后一天- 1月的第31天，非闰年的2月的第28天，4月的第30 天，依此类推。
    - 在day_of_week字段中单独使用以代替7或SAT，或在一周中的特定一天之后选择该类型可以表示为在该月的最后一个星期几。例如，6L表示该月的最后一个星期五。
    - 您可以在day_of_month字段中指定LW以指定该月的最后一个工作日。
    - 在指定值的列表或范围时避免使用L选项，因为结果可能不是您所期望的。
- `W`
    - 工作日。用于指定距给定日期最近的工作日（周一至周五）。
    - 例如，如果您在day_of_month字段中指定15W，并且15日是星期六，则计划将在14日触发。如果15日是星期日，则时间表将在16日星期一触发。如果15日是星期二，则时间表将在15日星期二触发。
    - 但是，如果您将1W指定为day_of_month的值，并且1日是星期六，则计划将在3日星期一触发-它不会跳过月份边界。
    - 您可以在day_of_month字段中指定LW以指定该月的最后一个工作日。仅当day_of_month为一天时，您才能使用W选项——在指定范围或天数列表时它无效。
- `#`
    - 一个月中的第XXX天。在day_of_week字段中使用以指定该月的第XXX天。例如，如果您指定6#1，计划将在每月的第一个星期五触发。请注意，如果您指定3#5并且特定月份没有5个星期二，则该计划不会触发该月。

示例参考

表达式	**触发结果**
`0 5 9 * * ?`	在UTC时间，每天上午9:05触发
`0 5 9 * * ? 2024`	在UTC时间，2024这一年中每天上午9:05触发
`0 5 9 ? * MON-FRI`	在UTC时间，周一至周五每天上午9:05触发
`0 0-5 9 * * ?`	在UTC时间，每天从上午9:00开始到上午9:05结束，每分钟触发一次。
`0 0/15 9 * * ?`	在UTC时间，每天从上午9:00开始到上午9:45结束，每15分钟触发一次。
`0 5 9 1/3 * ?`	在UTC时间，从每月的第一天开始，每3天上午9:05触发一次。
`0 1 4 1 4 ?`	在UTC时间，每年4月1日凌晨4:01触发一次。
`0 0,30 9 ? 4 WED`	在UTC时间，每年4月的每周三，上午9:00和上午9:30触发。

表达式	触发结果
`0 5 9 15 * ?`	在UTC时间，在每个月的第15天上午9:05触发。
`0 5 9 15W * ?`	在UTC时间，在距每月15日最近的工作日上午9:05触发。
`0 5 9 ? * 6#1`	在UTC时间，在每个月的第一个星期五上午9:05触发。
`0 5 9 L * ?`	在UTC时间，在每月最后一天上午9:05触发。
`0 5 9 ? * 2L`	在UTC时间，在每个月最后一个星期一上午9:05触发。
`0 5 9 LW * ?`	在UTC时间，在每个月最后一个工作日上午9:05触发

基于URL的访问控制

许多用户使用具有基于URL的访问控制的代理来保护对Elasticsearch索引的访问。对于多搜索、多获取和批量请求，用户可以选择在URL中以及在请求正文中的每个单独请求上指定索引。这会使基于URL的访问控制具有挑战性。
要防止用户覆盖URL中指定的索引，请将此设置添加到elasticsearch.yml文件：

```
rest.action.multi.allow_explicit_index: false
```

默认值为 true，但当设置为 false 时，Elasticsearch 将拒绝在请求正文中指定了显式索引的请求。

3 索引

在前面章节我们通过索引文档接触到索引，但没有做深入的探讨，在这里我们开始详细的总结使用Rest APIs来操作索引。

3.1 索引管理

创建索引

您可以使用创建索引 API 将新索引添加到 Elasticsearch 集群。创建索引时，您可以指定以下内容：

- 索引设置
- 索引中字段的映射
- 索引别名

请求方式

```
PUT /<index>
```

路径参数

- `<index>`
  - (必选, 字符串) 你希望创建索引的名称。
  - 索引名称必须满足以下条件：
    - 仅允许小写
    - 不能包含 `\`, `/`, `*`, `?`, `"`, `<`, `>`, `|`, (空白字符), `,`, `#`
    - 7.0之前的索引可能包含冒号 (:)，但已弃用，7.0+将不再支持
    - 开始字符不能为： `-`, `_`, `+`

- 不能是 `.` 或 `..`
- 不能超过255个字节 (注意它是字节，所以多字节字符将更快地计入255限制)

查询参数

- `include_type_name`
  - （可选，布尔值）~~[7.0.0开始已弃用]~~如果为true，则映射主体中应包含映射类型。默认为false。
- `wait_for_active_shards`
  - （可选，字符串）在继续操作之前必须处于活动状态的分片副本数。设置为`all`或任何正整数，最多为索引中的分片总数 (number_of_replicas + 1)。默认值：1。
- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。
- `timeout`
  - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。

请求主体

- `aliases`
  - （可选，别名对象）包含索引的索引别名。请参阅更新索引别名。
- `mappings`
  - （可选，映射对象）索引中字段的映射。如果指定，此映射可以包括：
    - 字段名称
    - 字段数据类型
    - 映射参数
  - 请参阅映射。
- `settings`
  - （可选，索引设置对象）索引的配置选项。请参阅索引设置。

使用示例

创建的每个索引都可以具有与之关联的特定设置，在正文中定义：

```
PUT /twitter
{
    "settings" : {
        "index" : {
            "number_of_shards" : 3,
            "number_of_replicas" : 2
        }
    }
}
```

- number_of_shards：的默认值为 1
- number_of_replicas：的默认值为 1（即每个主分片一个副本）

或更简化

```
PUT /twitter
{
    "settings" : {
        "number_of_shards" : 3,
        "number_of_replicas" : 2
    }
}
```

创建索引 API 允许提供映射定义：

```
PUT /test
{
    "settings" : {
        "number_of_shards" : 1
    },
    "mappings" : {
        "properties" : {
            "field1" : { "type" : "text" }
        }
    }
}
```

创建索引 API 还允许提供一组别名：

```
PUT /test
{
    "aliases" : {
        "alias_1" : {},
        "alias_2" : {
            "filter" : {
                "term" : {"user" : "kimchy" }
            },
            "routing" : "kimchy"
        }
    }
}
```

默认情况下，索引创建只会在每个分片的主副本已启动或请求超时时才向客户端返回响应。索引创建响应将指示发生了什么：

```
{
    "acknowledged": true,
    "shards_acknowledged": true,
    "index": "test"
}
```

我们可以通过索引设置 `index.write.wait_for_active_shards` 更改默认只等待主分片启动（注意更改此设置也会影响所有后续写入操作的 `wait_for_active_shards` 值）：

```
PUT /test
{
    "settings": {
        "index.write.wait_for_active_shards": "2"
    }
}
```

或者通过请求参数`wait_for_active_shards`：

```
PUT /test?wait_for_active_shards=2
```

删除索引

请求方式

```
DELETE /<index>
```

路径参数

- `<index>`
  - （请求，字符串）要删除的索引的逗号分隔列表或通配符表达式。
  - 在此参数中，通配符表达式仅匹配开放的、具体的索引。
  - 您不能使用别名删除索引。
  - 要删除所有索引，请使用 `_all` 或 `*` 。
  - 要禁止使用 `_all` 或通配符表达式删除索引，请将 `action.destructive_requires_name` 集群设置更改为 `true` 。您可以在 elasticsearch.yml 文件中或使用集群更新设置API更新此设置。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `ignore_unavailable`
  - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为false。
- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为30s。
- `timeout`

- （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为30s。

使用示例

```
DELETE /twitter?ignore_unavailable=true
```

响应结果

```
{
  "acknowledged" : true
}
```

获取索引

请求方式

```
GET /<index>
```

路径参数

- `<index>`
  - （必需，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
  - 使用 `_all` 的值来检索集群中所有索引的信息。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `flat_settings`
  - （可选，布尔值）如果为true，则以扁平化格式返回设置。默认为false。
- `include_defaults`
  - （可选，字符串）如果为 true，则返回响应中的所有默认设置。默认为false。
- `include_type_name`
  - （可选，布尔值）~~[7.0.0开始已弃用]~~如果为true，则映射主体中应包含映射类型。默认为false。
- `ignore_unavailable`
  - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为false。

- `local`
  - （可选，布尔值）如果为 true，则请求仅从本地节点检索信息。默认为 false，这意味着从主节点检索信息。
- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为30s。

使用示例

```
GET /customer,twitter*?flat_settings=true
```

响应结果

```
{
  "customer" : {
    "aliases" : { },
    "mappings" : {
      "properties" : {
        "name" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        }
      }
    },
    "settings" : {
      "index.creation_date" : "1662522350526",
      "index.number_of_replicas" : "0",
      "index.number_of_shards" : "1",
      "index.provided_name" : "customer",
      "index.uuid" : "ewbC61BtQWmvoZu_J-HYrg",
      "index.version.created" : "7060299"
    }
  }
}
```

索引是否存在

用于检查索引是否存在。

请求方式

```
HEAD /<index>
```

路径参数

- `<index>`
  - （必需，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。

> 此参数不区分索引名称和别名，即如果存在具有该名称的别名，则也返回状态代码 200。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `ignore_unavailable`
  - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为false。
- `local`
  - （可选，布尔值）如果为 true，则请求仅从本地节点检索信息。默认为 false，这意味着从主节点检索信息。

响应代码

- `200`
  - 表示所有指定的索引或索引别名都存在。
- `404`
  - 表示一个或多个指定的索引或索引别名不存在。

使用示例

```
HEAD /twitter,customer?ignore_unavailable=true
```

响应结果

```
200 - OK
```

关闭索引

关闭的索引不能进行读/写操作，并且不具备打开状态索引的所有操作。

通过关闭索引，可以不必维护用于索引或搜索文档的内部数据结构，从而减少集群的开销。

在打开或关闭索引时，master 负责重新启动索引分片以响应索引的新状态，然后分片将经历正常的恢复过程。

打开/关闭索引的数据由集群自动复制，以确保始终安全地保留足够的分片副本。

您可以打开和关闭多个索引。如果请求明确引用缺少的索引，则会引发错误，可以使用 `ignore_unavailable=true` 参数禁用此行为。
所有索引都可以使用 `_all` 作为索引名称或指定标识所有索引的模式（例如 `*`）一次打开或关闭。

可以通过将配置文件中的 `action.destructive_requires_name` 标志设置为 `true` 来禁用通过通配符或 `_all` 识别索引。此设置也可以通过集群更新设置 API 进行更改。

关闭索引会消耗大量磁盘空间，这可能会导致托管环境出现问题。通过将 `cluster.indices.close.enable` 设置为 `false`，可以通过集群设置 API 禁用关闭索引。默认值为 `true`。

因为打开或关闭索引会分配其分片，所以索引创建时的 `wait_for_active_shards` 设置也适用于 `_open` 和 `_close` 索引操作。

请求方式

```
POST /<index>/_close
```

路径参数

- `<index>`
  - （必需，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
  - 要关闭所有索引，请使用 `_all` 或 `*`。要禁止使用 `_all` 或通配符表达式关闭索引，请将 `action.destructive_requires_name` 集群设置更改为 `true`。
  - 您可以在 `elasticsearch.yml` 文件中或使用集群更新设置API更新此设置。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `ignore_unavailable`
  - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为false。
- `wait_for_active_shards`
  - （可选，字符串）在继续操作之前必须处于活动状态的分片副本数。设置为`all`或任何正整数，最多为索引中的分片总数 (number_of_replicas + 1)。默认值：1。
- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为30s。

- `timeout`
    - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为30s。

使用示例

以下示例显示了如何关闭索引：

```
POST /my_index1,customer/_close?ignore_unavailable=true
```

响应结果

```
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "indices" : {
    "customer" : {
      "closed" : true
    }
  }
}
```

打开索引

请求方式

```
POST /<index>/_open
```

路径参数、查询参数与关闭索引一致，这里就不赘述了。

使用示例

关闭的索引可以像这样重新打开：

```
POST /my_index1,customer/_open?ignore_unavailable=true
```

响应结果

```
{
  "acknowledged" : true,
  "shards_acknowledged" : true
}
```

冻结索引

冻结索引在集群上几乎没有开销（除了在内存中维护其元数据）并且是只读的。

只读索引被阻止用于写入操作，例如 docs-index_ 或强制合并。

详见冻结索引

> 冻结索引将关闭索引并在同一个 API 调用中重新打开它。这会导致在短时间内无法分配主节点，并导致集群变红，直到再次分配主节点。将来可能会取消此限制。

请求方式

```
POST /<index>/_freeze
```

路径参数

- `<index>`
  - （必需，字符串）索引的标识符。

使用示例

以下示例冻结索引：

```
POST /my_index1,customer/_freeze?ignore_unavailable=true
```

响应结果

```
{
  "acknowledged" : true,
  "shards_acknowledged" : true
}
```

解冻索引

请求方式

```
POST /<index>/_unfreeze
```

路径参数

- `<index>`
  - （必需，字符串）索引的标识符。

使用示例

以下示例解冻索引：

```
POST /my_index1,customer/_unfreeze?ignore_unavailable=true
```

响应结果

```
{
  "acknowledged" : true,
  "shards_acknowledged" : true
}
```

3.2 映射管理

添加映射

将新字段添加到现有索引或更改现有字段的搜索设置。

请求方式

```
PUT /<index>/_mapping
PUT /_mapping
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
  - 要更新所有索引的映射，请省略此参数或使用 `_all` 值。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `include_type_name`
  - （可选，布尔值）~~[7.0.0开始已弃用]~~如果为true，则映射主体中应包含映射类型。默认为false。
- `ignore_unavailable`
  - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为false。
- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。
- `timeout`
  - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。

请求主体

- `properties`
  - （必需，映射对象）字段的映射。对于新字段，此映射可以包括：
    - 字段名称
    - 字段数据类型
    - 映射参数
  - 对于现有字段，请参阅更改现有字段的映射。

使用示例

索引设置示例

PUT `_mapping` API 需要现有索引。以下创建索引 API 请求创建没有映射的发布索引。

```
PUT /publications
```

以下PUT `_mapping` API 请求将标题（一个新的文本字段）添加到出版物索引。

```
PUT /publications/_mapping
{
  "properties": {
    "title":  { "type": "text"}
  }
}
```

多个索引

PUT `_mapping` API 可以通过单个请求应用于多个索引。例如，我们可以同时更新 twitter-1 和 twitter-2 映射：

```
PUT /twitter-1
PUT /twitter-2
PUT /twitter-1,twitter-2/_mapping
{
  "properties": {
    "user_name": {
      "type": "text"
    }
  }
}
```

注意：指定的索引 (twitter-1,twitter-2) 遵循多个索引名称和通配符格式。

现有对象字段添加新属性

您可以使用PUT `_mapping` API 将新属性添加到现有对象字段。要了解其工作原理，请尝试以下示例。

使用创建索引 API 创建一个带有名称对象字段和内部第一个文本字段的索引。

```
PUT /my_index
{
  "mappings": {
    "properties": {
      "name": {
        "properties": {
          "first": {
            "type": "text"
          }
        }
      }
    }
  }
}
```

使用PUT `_mapping` API 将新的内部最后文本字段添加到名称字段。

```
PUT /my_index/_mapping
{
  "properties": {
    "name": {
      "properties": {
        "last": {
          "type": "text"
        }
      }
    }
  }
}
```

使用GET `_mapping` API 来验证您的更改。

```
GET /my_index/_mapping
```

API 返回以下响应：

```
{
  "my_index" : {
    "mappings" : {
      "properties" : {
        "name" : {
          "properties" : {
            "first" : {
              "type" : "text"
            },
            "last" : {
              "type" : "text"
            }
          }
        }
      }
    }
  }
}
```

将多字段添加到现有字段

多字段允许您以不同方式索引同一字段。您可以使用PUT `_mapping` API来更新字段映射参数并为现有字段启用多字段。

要了解其工作原理，请尝试以下示例。

使用创建索引API创建带有城市文本字段的索引。

```
PUT /my_index1
{
  "mappings": {
    "properties": {
      "city": {
        "type": "text"
      }
    }
  }
}
```

虽然文本字段适用于全文搜索，但关键字字段不被分析，可能更适合排序或聚合。

使用PUT `_mapping` API为城市字段启用多字段。本次请求添加`city.raw`关键字多字段，可用于排序。

```
PUT /my_index1/_mapping
{
  "properties": {
    "city": {
      "type": "text",
      "fields": {
        "raw": {
          "type": "keyword"
        }
      }
    }
  }
}
```

使用GET `_mapping`API 来验证您的更改。

```
GET /my_index1/_mapping
```

API 返回以下响应：

```
{
  "my_index1" : {
    "mappings" : {
      "properties" : {
        "city" : {
          "type" : "text",
          "fields" : {
            "raw" : {
              "type" : "keyword"
            }
          }
        }
      }
    }
  }
}
```

更改现有字段支持的映射参数

每个映射参数的文档说明您是否可以使用添加映射API为现有字段更新它。

例如，您可以使用添加映射API来更新 `ignore_above` 参数。
要了解其工作原理，请尝试以下示例。

使用添加索引API创建包含user_id关键字字段的索引。 `user_id` 字段的 `ignore_above` 参数值为 `20`。

```
PUT /my_index2
{
  "mappings": {
    "properties": {
      "user_id": {
        "type": "keyword",
        "ignore_above": 20
      }
    }
  }
}
```

使用添加映射API将 `ignore_above` 参数值更改为 `100`。

```
PUT /my_index2/_mapping
{
  "properties": {
    "user_id": {
      "type": "keyword",
      "ignore_above": 100
    }
  }
}
```

使用GET `_mapping` API 来验证您的更改。

```
GET /my_index2/_mapping
```

API 返回以下响应：

```
{
  "my_index2" : {
    "mappings" : {
      "properties" : {
        "user_id" : {
          "type" : "keyword",
          "ignore_above" : 100
        }
      }
    }
  }
}
```

更改现有字段的映射

除支持的映射参数外，您不能更改现有字段的映射或字段类型。更改现有字段可能会使已编入索引的数据无效。

如果您需要更改字段的映射，请创建一个具有正确映射的新索引，并将您的数据重新索引到该索引中。

要了解其工作原理，请尝试以下示例。

使用添加索引API创建`user_id`字段为`long`字段类型的用户索引。

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

使用索引API来索引具有`user_id`字段值的多个文档。

```
POST /users/_doc?refresh=wait_for
{
    "user_id" : 12345
}

POST /users/_doc?refresh=wait_for
{
    "user_id" : 12346
}
```

要将 user_id 字段更改为关键字字段类型，请使用创建索引API创建具有正确映射的`new_users`索引。

```
PUT /new_users
{
  "mappings" : {
    "properties": {
      "user_id": {
        "type": "keyword"
      }
    }
  }
}
```

使用 reindex API 将文档从 users 索引复制到 new_users 索引。

```
POST /_reindex
{
  "source": {
    "index": "users"
  },
  "dest": {
    "index": "new_users"
  }
}
```

API 返回以下响应：

```
{
  "took" : 18,
  "timed_out" : false,
  "total" : 2,
  "updated" : 0,
  "created" : 2,
  "deleted" : 0,
  "batches" : 1,
  "version_conflicts" : 0,
  "noops" : 0,
  "retries" : {
    "bulk" : 0,
    "search" : 0
  },
  "throttled_millis" : 0,
  "requests_per_second" : -1.0,
  "throttled_until_millis" : 0,
  "failures" : [ ]
}
```

重命名字段

重命名字段将使已在旧字段名称下索引的数据无效。相反，添加别名字段以创建备用字段名称。

例如，使用添加索引API创建带有`user_identifier`字段的索引。

```
PUT /my_index3
{
  "mappings": {
    "properties": {
      "user_identifier": {
        "type": "keyword"
      }
    }
  }
}
```

使用添加映射API为现有的 user_identifier 字段添加 user_id 字段别名。

```
PUT /my_index3/_mapping
{
  "properties": {
    "user_id": {
      "type": "alias",
      "path": "user_identifier"
    }
  }
}
```

使用获取映射API来验证您的更改。

```
GET /my_index3/_mapping
```

响应结果

```
{
  "my_index3" : {
    "mappings" : {
      "properties" : {
        "user_id" : {
          "type" : "alias",
          "path" : "user_identifier"
        },
        "user_identifier" : {
          "type" : "keyword"
        }
      }
    }
  }
}
```

获取映射

检索集群中索引的映射定义。

请求方式

```
GET /_mapping
GET /<index>/_mapping
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。

- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `include_type_name`
  - （可选，布尔值）~~[7.0.0开始已弃用]~~如果为true，则映射主体中应包含映射类型。默认为false。
- `ignore_unavailable`
  - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为false。
- `local`
  - （可选，布尔值）如果为 true，则请求仅从本地节点检索信息。默认为 false，这意味着从主节点检索信息。
- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。

使用示例

获取映射API可用于通过一次调用获取多个索引。 API 的一般用法遵循以下语法：

`host:port/{index}/_mapping` 其中 `{index}` 可以接受逗号分隔的名称列表。

要获取所有索引的映射，您可以使用 `_all` 代替 `{index}`。以下是一些示例：

```
GET /my_index1,my_index2/_mapping
```

响应结果

```
{
  "my_index2" : {
    "mappings" : {
      "properties" : {
        "user_id" : {
          "type" : "keyword",
          "ignore_above" : 100
        }
      }
    }
  },
  "my_index1" : {
    "mappings" : {
      "properties" : {
        "city" : {
          "type" : "text",
          "fields" : {
            "raw" : {
              "type" : "keyword"
            }
```

```
          }
        }
      }
    }
  }
}
```

如果要获取所有索引和类型的映射，则以下两个示例是等效的：

```
GET /_all/_mapping
GET /_mapping
```

获取字段映射

检索一个或多个字段的映射定义。如果您不需要索引的完整映射或者您的索引包含大量字段，这将很有用。

请求方式

```
GET /_mapping/field/<field>
GET /<index>/_mapping/field/<field>
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
- `<field>`
  - （可选，字符串）用于限制返回信息的字段的逗号分隔列表或通配符表达式。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `include_type_name`
  - （可选，布尔值）~~[7.0.0开始已弃用]~~如果为true，则映射主体中应包含映射类型。默认为false。
- `ignore_unavailable`
  - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为false。
- `include_defaults`

- （可选，布尔值）如果为 true，则响应包括默认映射值。默认为false。

- `local`
  - （可选，布尔值）如果为 true，则请求仅从本地节点检索信息。默认为 false，这意味着从主节点检索信息。

使用示例

您可以在创建新索引时提供字段映射。以下创建索引 API 请求创建具有多个字段映射的发布索引。

```
PUT /publications
{
    "mappings": {
        "properties": {
            "id": { "type": "text" },
            "title":  { "type": "text"},
            "abstract": { "type": "text"},
            "author": {
                "properties": {
                    "id": { "type": "text" },
                    "name": { "type": "text" }
                }
            }
        }
    }
}
```

以下仅返回字段标题的映射：

```
GET publications/_mapping/field/title
```

响应结果

```
{
  "publications" : {
    "mappings" : {
      "title" : {
        "full_name" : "title",
        "mapping" : {
          "title" : {
            "type" : "text"
          }
        }
      }
    }
  }
}
```

指定字段

获取映射API允许您指定以逗号分隔的字段列表。

例如，要选择作者字段的id，您必须使用其全名author.id。

```
GET publications/_mapping/field/author.id,abstract,name
```

响应结果

```
{
  "publications" : {
    "mappings" : {
      "abstract" : {
        "full_name" : "abstract",
        "mapping" : {
          "abstract" : {
            "type" : "text"
          }
        }
      },
      "author.id" : {
        "full_name" : "author.id",
        "mapping" : {
          "id" : {
            "type" : "text"
          }
        }
      }
    }
  }
}
```

获取字段映射 API 还支持通配符表示法。

```
GET publications/_mapping/field/a*
```

响应结果

```
{
  "publications" : {
    "mappings" : {
      "author.name" : {
        "full_name" : "author.name",
        "mapping" : {
          "name" : {
            "type" : "text"
          }
        }
      },
      "abstract" : {
        "full_name" : "abstract",
        "mapping" : {
          "abstract" : {
            "type" : "text"
          }
        }
      },
      "author.id" : {
```

```
        "full_name" : "author.id",
        "mapping" : {
          "id" : {
            "type" : "text"
          }
        }
      }
    }
  }
}
```

多个索引和字段

获取字段映射 API 可用于通过一次调用从多个索引中获取多个字段的映射。 API 的一般用法遵循以下语法：

`host:port/<index>/_mapping/field/<field>` 其中 `<index>` 和 `<field>` 可以代表逗号分隔的名称列表或通配符。

要获取所有索引的映射，您可以将 `_all` 用于 `<index>`。以下是一些示例：

```
GET /twitter,kimchy/_mapping/field/message
GET /_all/_mapping/field/message,user.id
GET /_all/_mapping/field/*.id
```

3.3 别名管理

索引别名是用于引用一个或多个索引的逻辑名称。

大多数 Elasticsearch API 接受索引别名来代替索引名称。

Elasticsearch中的API在处理特定索引时接受索引名称，并在适用时接受多个索引。索引别名 API 允许使用名称为索引设置别名，所有 API 都会自动将别名转换为实际的索引名称。一个别名也可以映射到多个索引，当指定它时，别名会自动扩展为别名索引。别名也可以与搜索时自动应用的过滤器和路由值相关联。别名不能与索引同名。

添加别名

创建或更新索引别名。

请求方式

```
PUT /<index>/_alias/<alias>
POST /<index>/_alias/<alias>
PUT /<index>/_aliases/<alias>
POST /<index>/_aliases/<alias>
```

路径参数

- `<index>`
  - （必需，字符串）要添加到别名的索引名称的逗号分隔列表或通配符表达式。
  - 要将集群中的所有索引添加到别名，请使用 `_all` 值。
- `<alias>`
  - （必需，字符串）要创建或更新的索引别名的名称。

查询参数

- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。
- `timeout`
  - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。

请求主体

- `filter`
  - （必填，查询对象）用于限制索引别名的过滤查询。
  - 如果指定，索引别名仅适用于过滤器返回的文档。
  - 过滤查询用于限制索引别名。
- `routing`
  - （可选，字符串）用于将操作路由到特定分片的自定义路由值。

使用示例

以下请求为customer索引创建别名cus。

```
PUT /customer/_alias/cus
```

添加基于用户的别名
首先，使用user_id字段的映射创建索引users：

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

然后为特定用户user_12添加索引别名：

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

在索引创建期间添加别名
您可以使用创建索引API在索引创建期间添加索引别名。

```
PUT /logs_20302801
```

```
{
    "mappings" : {
        "properties" : {
            "year" : {"type" : "integer"}
        }
    },
    "aliases" : {
        "2030" : {
            "filter" : {
                "term" : {"year" : 2030 }
            }
        }
    }
}
```

删除别名

删除现有索引别名。

请求方式

```
DELETE /<index>/_alias/<alias>
DELETE /<index>/_aliases/<alias>
```

路径参数

- `<index>`
  - （必需，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
  - 要包含集群中的所有索引，请使用 `_all` 或 `*` 值。
- `<alias>`
  - （必需，字符串）用于限制请求的索引别名的逗号分隔列表或通配符表达式。
  - 要删除所有别名，请使用 `_all` 或 `*` 值。

查询参数

- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。
- `timeout`
  - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。

使用示例

下面示例演示删除多个别名

```
DELETE /customer,users/_alias/cus,cus1,user_12
```

获取别名

返回有关一个或多个索引别名的信息。

请求方式

```
GET /_alias
GET /_alias/<alias>
GET /<index>/_alias/<alias>
```

路径参数

- `<alias>`
  - （可选，字符串）用于限制请求的索引别名的逗号分隔列表或通配符表达式。
  - 要检索所有索引别名的信息，请使用值 `_all` 或 `*`。
- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `all`。
- `ignore_unavailable`
  - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为false。
- `local`
  - （可选，布尔值）如果为 true，则请求仅从本地节点检索信息。默认为 false，这意味着从主节点检索信息。

使用示例

获取索引的所有别名。

您可以在创建索引期间使用创建索引 API 请求添加索引别名。

以下创建索引API请求使用两个别名创建logs_20302802索引：

- `current_day`
- `2030`，仅返回 logs_20302802 索引中年份字段值为 2030 的文档

```
PUT /logs_20302802
{
  "aliases" : {
    "current_day" : {},
    "2030" : {
      "filter" : {
          "term" : {"year" : 2030 }
      }
    }
  }
}
```

以下获取索引别名API请求返回索引logs_20302802的所有别名：

```
GET /logs_20302802/_alias/*
```

响应结果

```
{
  "logs_20302802" : {
    "aliases" : {
      "2030" : {
        "filter" : {
          "term" : {
            "year" : 2030
          }
        }
      },
      "current_day" : { }
    }
  }
}
```

获取特定别名

以下索引别名 API 请求返回 2030 别名：

```
GET /_alias/2030
```

API 返回以下响应：

```
{
  "logs_20302801" : {
    "aliases" : {
      "2030" : {
        "filter" : {
          "term" : {
            "year" : 2030
          }
        }
      }
    }
  },
```

```
  "logs_20302802" : {
    "aliases" : {
      "2030" : {
        "filter" : {
          "term" : {
            "year" : 2030
          }
        }
      }
    }
  }
}
```

根据通配符获取别名

以下索引别名 API 请求返回以 20 开头的任何别名：

```
GET /_alias/20*
```

响应结果与上一个示例效果一致。

别名是否存在

检查是否存在索引别名。

请求方式

```
HEAD /_alias/<alias>
HEAD /<index>/_alias/<alias>
```

路径参数

- `<alias>`
  - （必需，字符串）用于限制请求的索引别名的逗号分隔列表或通配符表达式。
- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。

查询参数

- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `all`。
- `ignore_unavailable`
  - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为false。
- `local`
  - （可选，布尔值）如果为 true，则请求仅从本地节点检索信息。默认为 false，这意味着从主节点检索信息。

响应代码

- `200`
  - 表示所有指定的索引别名都存在。
- `404`
  - 表示一个或多个指定的索引别名不存在。

使用示例

执行下面示例

```
HEAD /_alias/2030
HEAD /_alias/20*
HEAD /logs_20302801/_alias/*
```

响应结果

```
# HEAD /_alias/2030
200 - OK
# HEAD /_alias/20*
200 - OK
# HEAD /logs_20302801/_alias/*
200 - OK
```

更新别名

用于添加或删除索引别名。

请求方式

```
POST /_aliases
```

查询参数

- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。
- `timeout`
  - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。

请求主体

- `actions`
  - （必需，操作数组）要执行的操作集。有效的行为包括：
    - `add`
      - 为索引添加别名。
    - `remove`
      - 从索引中删除别名。
    - `remove_index`

        - 删除索引或索引别名，比如删除索引API。
    - 您可以对别名对象执行这些操作。别名对象的有效参数包括：
        - `index`
            - （字符串）用于执行操作的索引名称的通配符表达式。
            - 如果未指定 `indices` 参数，则此参数是必需的。
        - `indices`
            - （数组）用于执行操作的索引名称数组。
            - 如果未指定 `index` 参数，则此参数是必需的。
        - `alias`
            - （字符串）要添加、删除或删除的索引别名的逗号分隔列表或通配符表达式。
            - 如果未指定 `aliases` 参数，则添加或删除操作需要此参数。
        - `aliases`
            - （字符串）要添加、删除或删除的索引别名的逗号分隔列表或通配符表达式。
            - 如果未指定`alias`参数，则添加或删除操作需要此参数。
        - `filter`
            - （可选，查询对象）用于限制索引别名的过滤查询。
            - 如果指定，索引别名仅适用于过滤器返回的文档。
        - `is_write_index`
            - （可选，布尔值）如果为真，则将该索引分配为别名的写入索引。默认为false。
            - 一个别名一次可以有一个写索引。
        - `routing`
            - （可选，字符串）用于将操作路由到特定分片的自定义路由值。
        - `index_routing`
            - （可选，字符串）用于别名索引操作的自定义路由值。
        - `search_routing`
            - （可选，字符串）用于别名搜索操作的自定义路由值。

使用示例

添加别名

以下请求将alias1别名添加到test1索引。

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

删除别名

以下请求删除 alias1 别名。

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

重命名别名

重命名别名是在同一个 API 中简单的删除然后添加操作。

```
POST /_aliases
{
    "actions" : [
        { "remove" : { "index" : "test1", "alias" : "alias1" } },
        { "add" : { "index" : "test1", "alias" : "alias2" } }
    ]
}
```

为多个索引添加别名

将别名与多个索引相关联只是几个添加操作：

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

可以使用索引数组语法为动作指定多个索引：

```
POST /_aliases
{
    "actions" : [
        { "add" : { "indices" : ["test1", "test2"], "alias" : "alias3" } }
    ]
}
```

要在一个操作中指定多个别名，相应的别名数组语法也存在。

对于上面的示例，模式匹配也可用于将别名与多个共享同一个名称的索引相关联：

```
POST /_aliases
{
    "actions" : [
        { "add" : { "index" : "test*", "alias" : "all_test_indices" } }
    ]
}
```

在这种情况下，别名是一个时间点别名，它将对所有匹配的当前索引进行分组，它不会随着与此模式匹配的新索引的添加/删除而自动更新。

过滤的别名

带有过滤器的别名提供了一种简单的方法来创建同一索引的不同“视图”。

过滤器可以使用 Query DSL 定义，并应用于所有使用此别名的 Search、Count、Delete By Query 等操作。

要创建过滤别名，首先我们需要确保映射中已经存在字段：

```
PUT /test3
{
  "mappings": {
    "properties": {
      "user" : {
        "type": "keyword"
      }
    }
  }
}
```

现在我们可以创建一个别名，对字段用户使用过滤器：

```
POST /_aliases
{
    "actions" : [
        {
            "add" : {
                "index" : "test3",
                "alias" : "alias3",
                "filter" : { "term" : { "user" : "kimchy" } }
            }
        }
    ]
}
```

路由

可以将路由值与别名相关联。此功能可以与过滤别名一起使用，以避免不必要的分片操作。

以下命令创建一个指向索引test的新别名alias1。创建alias1后，所有具有此别名的操作都会自动修改为使用值1 进行路由：

```
POST /_aliases
{
    "actions" : [
        {
            "add" : {
                "index" : "test",
                "alias" : "alias1",
                "routing" : "1"
            }
        }
    ]
}
```

也可以为搜索和索引操作指定不同的路由值：

```
POST /_aliases
{
    "actions" : [
        {
            "add" : {
                "index" : "test",
                "alias" : "alias2",
                "search_routing" : "1,2",
                "index_routing" : "2"
            }
        }
    ]
}
```

如上例所示，搜索路由可能包含多个以逗号分隔的值。索引路由只能包含一个值。

如果使用路由别名的搜索操作也有路由参数，则使用搜索别名路由和参数中指定的路由的交集。

例如，以下命令将使用“2”作为路由值：

```
GET /alias2/_search?q=user:kimchy&routing=2,3
```

写索引

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写入索引的一个索引。每个别名一次只能分配一个索引作为写入索引。如果没有指定写入索引并且有多个索引被别名引用，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入索引。

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

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

在此示例中，我们将别名 alias1 与 test1 和 test2 相关联，其中 test1 将是选择用于写入的索引。

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

索引到 `/alias1/_doc/1` 的新文档将像 `/test1/_doc/1` 一样被索引。

```
GET /test1/_doc/1
```

要交换哪个索引是别名的写入索引，可以利用别名API进行原子交换。交换不依赖于动作的顺序。

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

3.4 索引配置

更新配置

实时更改索引设置。

请求方式

```
PUT /<index>/_settings
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
  - 要更新所有索引的设置，请使用 `_all` 或排除此参数。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为 `false`。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `flat_settings`
  - （可选，布尔值）如果为`true`，则以扁平化格式返回设置。默认为`false`
- `ignore_unavailable`
  - （可选，布尔值）如果为 `true`，则响应中不包含缺失或关闭的索引。默认为`false`。
- `preserve_existing`
  - （可选，布尔值）如果为`true`，现有索引设置保持不变。默认为`false`。
- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。
- `timeout`
  - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。

请求主体

- `settings`
  - （可选，索引设置对象）索引的配置选项。请参阅索引设置。

使用示例

重置索引设置

要将设置恢复为默认值，请使用 `null`。例如：

```
PUT /twitter/_settings
{
    "index" : {
        "refresh_interval" : null
    }
}
```

可以在索引模块中找到可以在实时索引上动态更新的每个索引设置的列表。要保留现有设置不被更新，可以将 `preserve_existing` 请求参数设置为 `true`。

批量索引使用

例如，更新设置 API 可用于动态更改索引，使其对批量索引的性能更高，然后将其移动到更实时的索引状态。在开始批量索引之前，请使用：

```
PUT /twitter/_settings
{
    "index" : {
        "refresh_interval" : "-1"
    }
}
```

（另一个优化选项是在没有任何副本的情况下启动索引，然后才添加它们，但这实际上取决于用例）。然后，一旦完成批量索引，就可以更新设置（例如恢复默认值）：

```
PUT /twitter/_settings
{
    "index" : {
        "refresh_interval" : "1s"
    }
}
```

并且，应该调用强制合并：

```
POST /twitter/_forcemerge?max_num_segments=5
```

更新索引分析

您只能在关闭索引上定义新的分析器。
要添加分析器，您必须关闭索引、定义分析器并重新打开索引。

例如，以下命令将内容分析器添加到twitter：

```
POST /twitter/_close

PUT /twitter/_settings
{
  "analysis" : {
    "analyzer":{
      "content":{
```

```
        "type":"custom",
        "tokenizer":"whitespace"
      }
    }
  }
}

POST /twitter/_open
```

获取配置

返回索引的设置信息。

请求方式

```
GET /<index>/_settings
GET /<index>/_settings/<setting>
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
  - 使用 `_all` 的值来检索集群中所有索引的信息。
- `<setting>`
  - （可选，字符串）用于限制请求的设置名称的逗号分隔列表或通配符表达式。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 `true`，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为 `true`。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `all`。
- `flat_settings`
  - （可选，布尔值）如果为 `true`，则以扁平化格式返回设置。默认为 `false`
- `ignore_unavailable`
  - （可选，布尔值）如果为 `true`，则响应中不包含缺失或关闭的索引。默认为 `false`。
- `local`
  - （可选，布尔值）如果为 `true`，则请求仅从本地节点检索信息。默认为 `false`，这意味着从主节点检索信息。
- `master_timeout`

- （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。

使用示例

多个索引

获取设置API可用于通过一次调用获取多个索引的设置。要获取所有索引的设置，您可以将 `_all` 用于 `<index>`。还支持通配符表达式。以下是一些示例：

```
GET /twitter,kimchy/_settings
GET /_all/_settings
GET /log_2013_*/_settings
```

按名称过滤设置

可以使用通配符匹配过滤返回的设置，如下所示：

```
GET /log_2013_-*/_settings/index.number_*
```

文本分析器配置

对文本字符串执行分析并返回结果标记。

请求方式

```
GET /_analyze
POST /_analyze
GET /<index>/_analyze
POST /<index>/_analyze
```

路径参数

- `<index>`
  - （可选，字符串）用于派生分析器的索引。
  - 如果指定，分析器或 `<field>` 参数将覆盖此值。
  - 如果未指定分析器或字段，则分析API使用索引的默认分析器。
  - 如果未指定索引或索引没有默认分析器，则分析API使用标准分析器。

查询参数

- `analyzer`
  - （可选，字符串或自定义分析器对象）用于分析提供的文本的分析器。
  - 有关内置分析器的列表，请参阅内置分析器参考。您还可以提供自定义分析器。
  - 如果未指定此参数，则分析API使用字段映射中定义的分析器。
  - 如果未指定任何字段，则分析API使用索引的默认分析器。
  - 如果未指定索引，或者索引没有默认分析器，则分析 API 使用标准分析器。
- `attributes`
  - （可选，字符串数组）用于过滤解释参数输出的标记属性数组。
- `char_filter`
  - （可选，字符串数组）用于在分词器之前预处理字符的字符过滤器数组。
  - 有关字符过滤器的列表，请参阅字符过滤器参考。

- `explain`
  - （可选，布尔值）如果为`true`，则响应包括令牌属性和其他详细信息。默认为`false`。
- `field`
  - （可选，字符串）用于派生分析器的字段。要使用此参数，您必须指定一个索引。
  - 如果指定，则分析器参数将覆盖此值。
  - 如果未指定任何字段，则分析 API 使用索引的默认分析器。
  - 如果未指定索引或索引没有默认分析器，则分析 API 使用标准分析器。
- `filter`
  - （可选，字符串数组）用于在标记器之后应用的标记过滤器数组。
  - 有关令牌过滤器的列表，请参阅令牌过滤器参考。
- `normalizer`
  - （可选，字符串）用于将文本转换为单个标记的规范器。
  - 有关规范化器的列表，请参阅规范化器。
- `text`
  - （必需，字符串或字符串数组）要分析的文本。
  - 如果提供了字符串数组，则将其作为多值字段进行分析。
- `tokenizer`
  - （可选，字符串）用于将文本转换为标记的标记器。
  - 有关标记器列表，请参阅标记器参考。

使用示例

未指定索引

您可以将任何内置分析器应用于文本字符串，而无需指定索引。

```
GET /_analyze
{
  "analyzer" : "standard",
  "text" : "this is a test"
}
```

文本字符串数组

如果文本参数作为字符串数组提供，则将其作为多值字段进行分析。

```
GET /_analyze
{
  "analyzer" : "standard",
  "text" : ["this is a test", "the second text"]
}
```

定制分析器

您可以使用分析API来测试由标记器、标记过滤器和字符过滤器构建的自定义瞬态分析器。

令牌过滤器使用 filter 参数，下面的案例可以把词转换成小写：

```
GET /_analyze
{
  "tokenizer" : "keyword",
  "filter" : ["lowercase"],
  "text" : "this is a Test"
}
```

响应结果

```
{
  "tokens" : [
    {
      "token" : "this is a test",
      "start_offset" : 0,
      "end_offset" : 14,
      "type" : "word",
      "position" : 0
    }
  ]
}
```

下面案例可以去掉html标签

```
GET /_analyze
{
  "tokenizer" : "keyword",
  "filter" : ["lowercase"],
  "char_filter" : ["html_strip"],
  "text" : "this is a <b>Test</b>"
}
```

响应结果

```
{
  "tokens" : [
    {
      "token" : "this is a test",
      "start_offset" : 0,
      "end_offset" : 21,
      "type" : "word",
      "position" : 0
    }
  ]
}
```

可以在请求正文中指定自定义标记器、标记过滤器和字符过滤器，如下所示：

```
GET /_analyze
{
  "tokenizer" : "whitespace",
  "filter" : ["lowercase", {"type": "stop", "stopwords": ["a", "is", "this"]}],
  "text" : "this is a test"
}
```

可以过滤掉一些词汇

```
{
  "tokens" : [
    {
      "token" : "test",
      "start_offset" : 10,
      "end_offset" : 14,
      "type" : "word",
      "position" : 3
    }
  ]
}
```

指定索引

您还可以针对特定索引运行分析API：

```
GET /analyze_sample/_analyze
{
  "text" : "this is a test"
}
```

以上将使用与 `analyze_sample` 索引关联的默认索引分析器对“this is a test”文本进行分析。

还可以提供一个分析器来使用不同的分析器：

```
GET /analyze_sample/_analyze
{
  "analyzer" : "whitespace",
  "text" : "this is a test"
}
```

从字段映射派生分析器

分析器可以基于字段映射导出，例如：

```
GET /analyze_sample/_analyze
{
  "field" : "obj1.field1",
  "text" : "this is a test"
}
```

将根据 obj1.field1 映射中配置的分析器（如果不是，则使用默认索引分析器）进行分析。

规范化器

可以为关键字字段提供规范化器，规范化器与`analyze_sample`索引相关联。

```
GET /analyze_sample/_analyze
{
  "normalizer" : "my_normalizer",
  "text" : "BaR"
}
```

或者通过使用令牌过滤器和字符过滤器构建自定义瞬态规范器。

```
GET /_analyze
{
  "filter" : ["lowercase"],
  "text" : "BaR"
}
```

解释分析

如果您想获得更高级的详细信息，请将`explain`设置为`true`（默认为`false`）。

它将输出每个令牌的所有令牌属性。您可以通过设置属性选项过滤要输出的令牌属性。

```
GET /_analyze
{
  "tokenizer" : "standard",
  "filter" : ["snowball"],
  "text" : "detailed output",
  "explain" : true,
  "attributes" : ["keyword"]
}
```

> 将“关键字”设置为仅输出“关键字”属性

请求返回以下结果：

```
{
  "detail" : {
    "custom_analyzer" : true,
    "charfilters" : [ ],
    "tokenizer" : {
      "name" : "standard",
      "tokens" : [
        {
          "token" : "detailed",
          "start_offset" : 0,
          "end_offset" : 8,
          "type" : "<ALPHANUM>",
          "position" : 0
        },
        {
          "token" : "output",
          "start_offset" : 9,
          "end_offset" : 15,
```

```
          "type" : "<ALPHANUM>",
          "position" : 1
        }
      ]
    },
    "tokenfilters" : [
      {
        "name" : "snowball",
        "tokens" : [
          {
            "token" : "detail",
            "start_offset" : 0,
            "end_offset" : 8,
            "type" : "<ALPHANUM>",
            "position" : 0,
            "keyword" : false
          },
          {
            "token" : "output",
            "start_offset" : 9,
            "end_offset" : 15,
            "type" : "<ALPHANUM>",
            "position" : 1,
            "keyword" : false
          }
        ]
      }
    ]
  }
}
```

> 仅输出“关键字”属性，因为在请求中指定“属性”。

设置令牌限制

生成过多的令牌可能会导致节点内存不足。以下设置允许限制可以生成的令牌数量：

可以使用 `_analyze` API生成的最大令牌数，默认值为10000。

如果生成的令牌超过此限制，则会引发错误。

没有指定索引的 `_analyze` 端点将始终使用10000值作为限制。

此设置允许您控制特定索引的限制：

```
PUT /analyze_sample
{
  "settings" : {
    "index.analyze.max_token_count" : 20000
  }
}
```

3.5 索引模板

索引模板定义了您可以在创建新索引时自动应用的设置和映射。

Elasticsearch根据与索引名称匹配的索引模式将模板应用于新索引。

索引模板仅在索引创建期间应用。对索引模板的更改不会影响现有索引。

创建索引API请求中指定的设置和映射会覆盖索引模板中指定的任何设置或映射。

您可以在索引模板中使用C风格的 `/**/` 块注释。

您可以在请求正文中的任何位置包含注释，除了开始的大括号之前。

创建模板

创建或更新索引模板。

请求方式

```
PUT /_template/<index-template>
```

路径参数

- `<index-template>`
  - （必需，字符串）要创建的索引模板的名称。

查询参数

- `create`
  - （可选，布尔值）如果为 `true`，则此请求无法替换或更新现有索引模板。默认为 `false`。
- `order`
  - （可选，整数）如果索引匹配多个模板，Elasticsearch应用此模板的顺序。
  - 首先合并具有较低 `order` 值的模板。具有较高 `order` 值的模板稍后合并，覆盖具有较低 `order` 值的模板。
- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。

请求主体

- `index_patterns`
  - （必需，字符串数组）用于在创建期间匹配索引名称的通配符表达式数组。
- `aliases`
  - （可选，别名对象）包含索引的索引别名。
  - 请参阅索引别名
- `mappings`
  - （可选，映射对象）索引中字段的映射。如果指定，此映射可以包括：
    - 字段名称
    - 字段数据类型
    - 映射参数
  - 请参阅映射

- `settings`
  - （可选，索引设置对象）索引的配置选项。
  - 请参阅索引设置。
- `version`
  - （可选，整数）用于在外部管理索引模板的版本号。
  - 这个数字不是由 Elasticsearch 自动生成的。

使用示例

索引别名

您可以在索引模板中包含索引别名。

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

> 在索引创建期间，别名中的 `{index}` 占位符将替换为模板应用到的实际索引名称。

匹配多个模板的索引

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，首先应用较低的顺序，然后应用较高的顺序覆盖它们。

例如：

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

PUT /_template/template_2
{
```

```
    "index_patterns" : ["te*"],
    "order" : 1,
    "settings" : {
        "number_of_shards" : 1
    },
    "mappings" : {
        "_source" : { "enabled" : true }
    }
}
```

以上将禁用存储 `_source`，但对于以 te* 开头的索引，`_source` 仍将启用。请注意，对于映射，合并是"深度"的，这意味着可以在高阶模板上轻松 `添加/覆盖` 基于特定 `对象/属性` 的映射，而低阶模板提供基础。

> 具有相同顺序值的多个匹配模板将导致不确定的合并顺序。

模板版本控制

您可以使用 `version` 参数向索引模板添加可选的版本号。外部系统可以使用这些版本号来简化模板管理。

`version` 参数是完全可选的，不是由 Elasticsearch 自动生成的。

要取消设置版本，请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，您可以使用带有 `filter_path` 查询参数的获取索引模板 API 来仅返回版本号：

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

API 返回以下响应：

```
{
  "template_1" : {
    "version" : 123
  }
}
```

删除模板

使用删除索引模板API删除一个或多个索引模板

请求方式

```
DELETE /_template/<index-template>
```

路径参数

- `<index-template>`
  - （必需，字符串）用于限制请求的索引模板名称的逗号分隔列表或通配符表达式。

查询参数

- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。
- `timeout`
  - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 `30s`。

使用示例

```
DELETE /_template/template_1,template_2
```

获取模板

返回有关一个或多个索引模板的信息。

请求方式

```
GET /_template/<index-template>
```

路径参数

- `<index-template>`
  - （必需，字符串）用于限制请求的索引模板名称的逗号分隔列表或通配符表达式。
  - 要返回所有索引模板，请省略此参数或使用 `_all` 或 `*` 的值。

查询参数

- `flat_settings`
  - （可选，布尔值）如果为true，则以平面格式返回设置。
  - 默认为false。
- `local`
  - （可选，布尔值）如果为true，则请求仅从本地节点检索信息。
  - 默认为false，这意味着从主节点检索信息。
- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。
  - 如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 30s。

使用示例

获取多个模板

```
GET /_template/template_1,template_2
```

通过通配符获取

```
GET /_template/temp*
```

获取所有模板

```
GET /_template
```

模板是否存在

使用索引模板存在 API 来确定是否存在一个或多个索引模板。

请求方式

```
HEAD /_template/<index-template>
```

路径参数

- `<index-template>`
  - （必需，字符串）用于限制请求的索引模板名称的逗号分隔列表或通配符表达式。

查询参数

- `local`
  - （可选，布尔值）如果为true，则请求仅从本地节点检索信息。
  - 默认为false，这意味着从主节点检索信息。
- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。
  - 如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 30s。

响应代码

- `200`
  - 表示所有指定的索引模板都存在。
- `404`
  - 表示一个或多个指定的索引模板不存在。

使用示例

```
HEAD /_template/template_1
```

3.6 索引监控

索引统计

使用stats获取索引的高级聚合和统计信息。

默认情况下，返回的统计信息是索引级别的，包含`primaries`和`total`聚合。

`primaries`仅仅是主分片的值。

`total`是主分片和副本分片的累积值。

要获取分片级别的统计信息，请将`level`参数设置为`shards`。

> 提示：
>
> 当移动到另一个节点时，一个分片的分片级统计信息被清除。尽管分片不再是节点的一部分，但该节点保留了分片贡献的任何节点级统计信息。

请求方式

```
GET /<index>/_stats/<index-metric>
GET /<index>/_stats
GET /_stats
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
  - 要检索所有索引的统计信息，请使用 `_all` 或 `*` 值或省略此参数。
- `<index-metric>`
  - （可选，字符串）用于限制请求的以逗号分隔的指标列表。支持的指标是：
    - `_all`
      - 返回所有统计信息。
    - `completion`
      - 完成建议统计。
    - `docs`
      - 尚未合并的文档和已删除文档的数量。索引刷新会影响此统计信息。
    - `fielddata`
      - 现场数据统计。
    - `flush`
      - 刷新统计。
    - `get`
      - 获取统计信息，包括缺失的统计信息
    - `indexing`
      - 索引统计。
    - `merge`
      - 合并统计。
    - `query_cache`
      - 查询缓存统计信息。

        - `refresh`
            - 刷新统计。
        - `request_cache`
            - 分片请求缓存统计信息。
        - `search`
            - 搜索统计信息，包括建议统计信息。
            - 您可以通过添加额外的`group`参数来包含自定义组的统计信息（搜索操作可以与一个或多个组相关联）。
            - `groups`参数接受以逗号分隔的组名列表。
            - 使用 `_all` 返回所有组的统计信息。
        - `segments`
            - 所有开放段的内存使用情况。
            - 如果 `include_segment_file_sizes` 参数为 `true`，则该指标包括每个 Lucene 索引文件的聚合磁盘使用情况。
        - `store`
            - 以字节为单位的索引大小。
        - `suggest`
            - 建议者统计。
        - `translog`
            - Translog统计。
        - `warmer`
            - 更温暖的统计数据。

查询参数

- `expand_wildcards`
    - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
        - `all`：扩展到开放和关闭的索引。
        - `open`：仅扩展至开放索引。
        - `closed`：仅扩展到关闭索引。
        - `none`：不接受通配符表达式。
    - 默认值为 `open`。
- `fields`
    - （可选，字符串）要包含在统计信息中的字段的逗号分隔列表或通配符表达式。
    - 除非在 `completion_fields` 或 `fielddata_fields` 参数中提供了特定的字段列表，否则用作默认列表。
- `completion_fields`
    - （可选，字符串）要包含在 `fielddata` 和 `suggest` 统计信息中的字段的逗号分隔列表或通配符表达式。
- `fielddata_fields`
    - （可选，字符串）要包含在 `fielddata` 统计信息中的字段的逗号分隔列表或通配符表达式。
- `forbid_closed_indices`
    - （可选，布尔值）如果为`true`，则不会从关闭索引中收集统计信息。默认为`true`。
- `groups`

    - （可选，字符串）要包含在`search`统计信息中的搜索组的逗号分隔列表。
- `level`
    - （可选，字符串）指示是否在集群、索引或分片级别聚合统计信息。
    - 有效值为：
        - `cluster`
        - `indices`
        - `shards`
- `include_segment_file_sizes`
    - （可选，布尔值）如果为`true`，则调用报告每个 Lucene 索引文件的汇总磁盘使用情况（仅在请求段统计信息时适用）。
    - 默认为`false`。
- `include_unloaded_segments`
    - （可选，布尔值）如果为`true`，则响应包括来自未加载到内存中的段的信息。
    - 默认为`false`。

使用示例

统计多个索引

```
GET /index1,index2/_stats
```

统计所有索引

```
GET /_stats
```

以下请求仅返回所有索引的合并和刷新统计信息。

```
GET /_stats/merge,refresh
```

以下请求仅返回 group1 和 group2 搜索组的搜索统计信息。

```
GET /_stats/search?groups=group1,group2
```

索引分片

返回有关索引分片中 Lucene 段的低级信息。

请求方式

```
GET /<index>/_segments
GET /_segments
```

路径参数

- `<index>`
    - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `ignore_unavailable`
  - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为false。

响应主体

- `<segment>`
  - （字符串）段的名称，例如 _0。段名来源于段生成，内部用于在分片目录中创建文件名。
- `generation`
  - （整数）代号，例如 0。
  - Elasticsearch 会为每个写入的段递增此代号。 Elasticsearch 然后使用这个数字来派生段名称。
- `num_docs`
  - （整数）段中未删除文档的数量，例如 25。
  - 此数量基于 Lucene 文档，可能包括来自嵌套字段的文档。
- `deleted_docs`
  - （整数）segment中被删除文档的个数，比如0。
  - 这个数字是基于Lucene文档的。合并段时，Elasticsearch 会回收已删除的 Lucene 文档的磁盘空间。
- `size_in_bytes`
  - （整数）段使用的磁盘空间，例如 50kb。
- `memory_in_bytes`
  - （整数）存储在内存中用于高效搜索的段数据字节数，例如 1264。
  - 值 -1 表示 Elasticsearch 无法计算此数字。
- `committed`
  - （布尔值）如果为真，则将段同步到磁盘。同步的段可以在硬重启后继续存在。

  如果为 false，来自未提交段的数据也会存储在事务日志中，以便 Elasticsearch 能够在下一次启动时重播更改。
- `search`

  （布尔值）如果为真，则该段是可搜索的。

  如果为 false，则该段很可能已写入磁盘，但需要刷新才能进行搜索。
- `version`

（字符串）用于写入段的 Lucene 版本。

- `compound`

  （布尔值）如果为真，Lucene 将段中的所有文件合并到一个文件中以保存文件描述符。

- `attributes`

  （对象）包含有关是否启用高压缩的信息。

使用示例

获取指定索引的分片信息

```
GET /test/_segments
```

获取多个索引的分片信息

```
GET /test1,test2/_segments
```

获取所有索引的分片信息

```
GET /_segments
```

要添加可用于调试的附加信息，请使用`verbose`标志。

> 此功能处于技术预览阶段，可能会在未来版本中更改或删除。 Elastic 将尽最大努力解决任何问题，但技术预览版中的功能不受官方 GA 功能的支持 SLA 约束。

```
GET /test/_segments?verbose=true
```

索引恢复

使用索引恢复API获取有关正在进行和已完成的分片恢复的信息。

分片恢复是从主分片同步副本分片的过程。完成后，副本分片可用于搜索。

在以下过程中会自动进行恢复：

- 节点启动或失败。这种类型的恢复称为本地存储恢复。
- 主分片复制。
- 将分片重定位到同一集群中的不同节点。
- 快照还原。

请求方式

```
GET /<index>/_recovery
GET /_recovery
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
  - 使用`_all`的值来检索集群中所有索引的信息。

查询参数

- `active_only`
  - （可选，布尔值）如果为 true，则响应仅包括正在进行的分片恢复。默认为false。
- `detailed`
  - （可选，布尔值）如果为 true，则响应包含有关分片恢复的详细信息。默认为false。
- `index`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。

响应主体

- `id`
  - （整数）分片的 ID。
- `type`
  - （字符串）恢复类型。返回值包括：
    - `STORE`
      - 恢复与节点启动或故障有关。这种类型的恢复称为本地存储恢复。
    - `SNAPSHOT`
      - 恢复与快照恢复有关。
    - `REPLICA`
      - 恢复与主分片复制有关。
    - `RELOCATING`
      - 恢复与将分片重定位到同一集群中的不同节点有关。
- `STAGE`
  - （字符串）恢复阶段。返回值包括：
    - `DONE`
      - 完成阶段.
    - `FINALIZE`
      - 清理阶段.
    - `INDEX`
      - 读取索引元数据并将字节从源复制到目标。
    - `INIT`
      - 恢复尚未开始。
    - `START`
      - 开始恢复过程；打开索引以供使用。
    - `TRANSLOG`
      - 重放事务日志。
- `primary`
  - （布尔值）如果为true，则分片是主分片。
- `start_time`
  - （字符串）恢复开始的时间戳。
- `stop_time`
  - （字符串）恢复完成的时间戳。
- `total_time_in_millis`

    - （字符串）以毫秒为单位恢复分片的总时间。
- `source`
    - （对象）恢复源。这可以包括：
        - 存储库描述（如果恢复来自快照）
        - 源节点说明
- `target`
    - （对象）目标节点。
- `index`
    - （对象）有关物理索引恢复的统计信息。
- `translog`
    - （对象）关于 translog 恢复的统计信息。
- `start`
    - （对象）有关打开和启动索引的时间的统计信息。

使用示例

获取多个索引的恢复信息

```
GET /index1,index2/_recovery?human
```

获取所有索引的恢复信息

```
GET /_recovery?human
```

此响应包括有关恢复单个分片的单个索引的信息。恢复的源是快照存储库，恢复的目标是 `my_es_node` 节点。

响应还包括恢复的文件和字节的数量和百分比。

要获取恢复中的物理文件列表，请将`detailed`查询参数设置为 `true`。

```
GET _recovery?human&detailed=true
```

响应包括所有恢复的物理文件及其大小的列表。

响应还包括各个恢复阶段的时间（以毫秒为单位）：

- 索引检索
- 事务日志重播
- 索引开始时间

此响应表明恢复已完成。所有恢复，无论是正在进行的还是完成的，都保持在集群状态，并且可以随时报告。

要仅返回有关正在进行的恢复的信息，请将 `active_only` 查询参数设置为 `true`。

索引存储

使用索引分片存储API返回有关一个或多个索引中的副本分片的存储信息。

返回的信息包括：

- 每个副本分片所在的节点

- 每个副本分片的分配 ID
- 每个副本分片的唯一 ID
- 打开分片索引或早期失败时遇到的任何错误

默认情况下，API 仅返回未分配或具有一个或多个未分配副本分片的主分片的存储信息。

请求方式

```
GET /<index>/_shard_stores
GET /_shard_stores
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
  - 要检索集群中所有索引的信息，请使用 `_all` 或 `*` 值或省略此参数。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `ignore_unavailable`
  - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为false。
- `status`
  - （可选，字符串）用于限制请求的分片健康状态的逗号分隔列表。有效值包括：
    - `green`
      - 分配主分片和所有副本分片。
    - `yellow`
      - 一个或多个副本分片未分配。
    - `red`
      - 主分片未分配。
    - `all`
      - 返回所有碎片，无论健康状况如何。
  - 默认为`yellow`,`red`。

使用示例

获取特定索引的分片存储信息

```
GET /test/_shard_stores
```

获取多个索引的分片存储信息

```
GET /test1,test2/_shard_stores
```

获取所有索引的分片存储信息

```
GET /_shard_stores
```

您可以使用 `status` 查询参数来限制基于分片健康的返回信息。

以下请求仅返回分配的主分片和副本分片的信息。

```
GET /_shard_stores?status=green
```

返回结果示例

```
{
  "indices" : {
    "bank" : {
      "shards" : {
        "0" : {
          "stores" : [
            {
              "Qs_-QMXaRlqdq377HSt2sQ" : {
                "name" : "3b533a0963d6",
                "ephemeral_id" : "HAhumZpgQUuPhrHd-LdJXg",
                "transport_address" : "172.18.0.2:9300",
                "attributes" : {
                  "ml.machine_memory" : "3864649728",
                  "xpack.installed" : "true",
                  "ml.max_open_jobs" : "20"
                }
              },
              "allocation_id" : "tn92W3OvSg6ugi9Iql2idA",
              "allocation" : "primary",
               "store_exception" :  .........
            }
          ]
        }
      }
    }
  }
}
```

- bank.shards.0：是存储信息对应的存储编号
- bank.shards.0.stores：是分片所有副本的存储信息列表
- Qs_-QMXaRlqdq377HSt2sQ：是托管存储副本的节点信息，键是唯一的节点id。

- allocation_id：存储副本的分配id
- allocation：存储副本的状态，是否用作主副本、副本或根本不使用
- store_exception：打开分片索引或早期引擎故障时遇到的任何异常

3.7 状态管理

清除缓存

清除一个或多个索引的缓存。

请求方式

```
POST /<index>/_cache/clear
POST /_cache/clear
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `fielddata`
  - （可选，布尔值）如果为 true，则清除字段缓存。
  - 使用 fields 参数仅清除特定字段的缓存。
- `fields`
  - （可选，字符串）用于限制 fielddata 参数的字段名称的逗号分隔列表。
  - 默认为所有字段。
  - 此参数不支持对象或字段别名。
- `index`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表。
- `ignore_unavailable`
  - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为 `false`。
- `query`
  - （可选，布尔值）如果为 true，则清除查询缓存。
- `request`

- （可选，布尔值）如果为 true，则清除请求缓存。

使用示例

清除特定缓存

默认情况下，清除缓存API会清除所有缓存。您可以通过将以下查询参数设置为`true`来仅清除特定的缓存：

- `fielddata`：仅清除字段缓存
- `query`：仅清除查询缓存
- `request`：仅清除请求缓存

```
POST /twitter/_cache/clear?fielddata=true
POST /twitter/_cache/clear?query=true
POST /twitter/_cache/clear?request=true
```

清除特定字段的缓存

要仅清除特定字段的缓存，请使用 fields 查询参数。

```
POST /twitter/_cache/clear?fields=foo,bar
```

清除 foo 和 bar 字段的缓存

清除多个索引的缓存

```
POST /kimchy,elasticsearch/_cache/clear
```

清除所有索引的缓存

```
POST /_cache/clear
```

刷新

使用刷新 API 显式刷新一个或多个索引。刷新使自上次刷新以来对索引执行的所有操作都可用于搜索。

默认情况下，Elasticsearch 每秒定期刷新索引，但仅在最近 30 秒内收到一个或多个搜索请求的索引上。

您可以使用`index.refresh_interval`设置更改此默认间隔。

> **重要：**
>
> 刷新是资源密集型的。为确保良好的集群性能，我们建议等待 Elasticsearch 的定期刷新，而不是尽可能执行显式刷新。
> 如果您的应用程序工作流对文档进行索引，然后运行搜索以检索索引文档，我们建议使用索引API的`refresh=wait_for`查询参数选项。此选项可确保索引操作在运行搜索之前等待定期刷新。

请求方式

```
POST <index>/_refresh
GET <index>/_refresh
POST /_refresh
GET /_refresh
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
  - 要刷新集群中的所有索引，请省略此参数或使用 `_all` 或 `*` 的值。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `ignore_unavailable`
  - （可选，布尔值）如果为 `true`，则响应中不包含缺失或关闭的索引。默认为 `false`。

使用示例

刷新多个索引

```
POST /kimchy,elasticsearch/_refresh
```

刷新所有索引

```
POST /_refresh
```

冲洗

冲洗索引是确保当前仅存储在事务日志中的任何数据也永久存储在 Lucene 索引中的过程。重启时，Elasticsearch 会将事务日志中的所有未冲洗操作重播到 Lucene 索引中，以使其恢复到重启前的状态。

Elasticsearch 会根据需要自动触发冲洗，使用启发式算法来权衡未冲洗事务日志的大小与执行每次冲洗的成本。

一旦每个操作被冲洗，它就会永久存储在 Lucene 索引中。这可能意味着不需要在事务日志中维护它的额外副本，除非出于其他原因保留它。事务日志由多个文件组成，称为*generations*，一旦不再需要，Elasticsearch 将删除任何*generations*文件，从而释放磁盘空间。

也可以使用冲洗API触发一个或多个索引的冲洗，尽管用户很少需要直接调用此 API。如果您在索引某些文档后调用冲洗API，如果成功的响应则表明Elasticsearch在调用冲洗API之前索引的所有文档已经完成冲洗。

调用方式

```
POST /<index>/_flush
GET /<index>/_flush
POST /_flush
GET /_flush
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
  - 要冲洗集群中的所有索引，请省略此参数或使用 `_all` 或 `*` 的值。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `force`
  - （可选，布尔值）如果为 true，即使没有更改提交到索引，请求也会强制冲洗。默认为 true。
  - 您可以使用此参数来增加事务日志的生成编号。
  - 此参数被视为内部参数。
- `ignore_unavailable`
  - （可选，布尔值）如果为 `true`，则响应中不包含缺失或关闭的索引。默认为 `false`。
- `wait_if_ongoing`
  - （可选，布尔值）如果为 true，则冲洗操作会阻塞，直到另一个冲洗操作正在运行时执行。
  - 如果为 false，如果您在另一个冲洗操作运行时请求冲洗，Elasticsearch会返回错误。
  - 默认为true。

使用示例

冲洗特定索引

```
POST /kimchy/_flush
```

冲洗多个索引

```
POST /kimchy,elasticsearch/_flush
```

冲洗所有索引

```
POST /_flush
```

强制合并

使用强制合并API强制合并一个或多个索引的分片。合并通过将其中的一些合并在一起来减少每个分片中的段数，并释放已删除文档使用的空间。合并通常自动发生，但有时手动触发合并很有用。

> 警告：
>
> 仅在完成写入索引后才应针对索引调用强制合并。强制合并可能会导致生成非常大（>5GB）的段，如果您继续写入这样的索引，那么自动合并策略将永远不会考虑这些段以进行未来的合并，直到它们主要由已删除的文档组成。这可能会导致非常大的分片保留在索引中，从而导致磁盘使用量增加和搜索性能变差。

调用此API块，直到合并完成。如果客户端连接在完成之前丢失，则强制合并过程将在后台继续。任何强制合并相同索引的新请求也将阻塞，直到正在进行的强制合并完成。

强制合并API可以通过一次调用应用于多个索引，甚至可以应用于 `_all` 索引。每个节点一次执行一个分片的多索引操作。强制合并使正在合并的分片的存储量暂时增加，如果 `max_num_segments` 参数设置为 1，则其大小会增加一倍，因为所有分片都需要重写为新的。

请求方式

```
POST /<index>/_forcemerge
POST /_forcemerge
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。
  - 要合并集群中的所有索引，请省略此参数或使用 `_all` 或 `*` 的值。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。
    - `closed`：仅扩展到关闭索引。
    - `none`：不接受通配符表达式。
  - 默认值为 `open`。
- `flush`
  - （可选，布尔值）如果为 `true`，Elasticsearch 在强制合并后对索引执行冲洗。默认为 `true`。

- `ignore_unavailable`
  - （可选，布尔值）如果为 `true`，则响应中不包含缺失或关闭的索引。默认为 `false`。
- `max_num_segments`
  - （可选，整数）要合并到的分片数。要完全合并索引，请将其设置为 1。
  - 默认检查是否需要执行合并。如果是，则执行它。
- `only_expunge_deletes`
  - （可选，布尔值）如果为 `true`，则仅删除包含文档删除的分片。默认为 `false`。
  - 在 Lucene 中，不会从分片中删除文档；刚刚标记为已删除。在合并期间，将创建一个不包含这些文档删除的新分片。
  - 此参数不会覆盖 `index.merge.policy.expunge_deletes_allowed` 设置。

使用示例

强制合并特定索引

```
POST /twitter/_forcemerge
```

强制合并多个索引

```
POST /kimchy,elasticsearch/_forcemerge
```

强制合并所有索引

```
POST /_forcemerge
```

强制合并对于基于时间的索引很有用，尤其是在使用翻转时。在这些情况下，每个索引仅在特定时间段内接收索引流量。一旦索引不再接收写入，其分片可以强制合并到单个分片。

```
POST /logs-000001/_forcemerge?max_num_segments=1
```

这可能是一个好主意，因为单段分片有时可以使用更简单、更高效的数据结构来执行搜索。

3.8 文档管理

本节首先简要介绍Elasticsearch的**数据复制模型**，然后详细描述以下`CRUD API`：

读写文档

介绍

Elasticsearch中的每个索引都被划分为shard，每个shard可以有多个副本。这些副本称为副本组，在添加或删除文档时必须保持同步。如果我们不这样做，从一个副本读取将导致与从另一个副本读取到不同的结果。保持分片副本同步并提供读取服务的过程就是我们所说的**数据复制模型**。

Elasticsearch的数据复制模型基于主备模型，在微软研究院的PacificA论文中有很好的描述。该模型基于复制组中的单个副本作为主分片。其他副本称为副本分片。主节点充当所有索引操作的主要入口点。它负责验证它们并确保它们是正确的。主节点接受索引操作后，主节点还负责将该操作复制到其他副本。

本节的目的是对 Elasticsearch 复制模型进行高级概述，并讨论它对写入和读取操作之间的各种交互的影响。

基本写入模型

Elasticsearch中的每个索引操作首先使用路由解析到一个复制组，通常基于文档ID。一旦确定了复制组，操作就会在内部转发到该组的当前主分片。主分片负责验证操作并将其转发给其他副本。由于副本可以离线，因此不需要将主副本复制到所有副本。

相反，Elasticsearch维护一个应该接收操作的分片副本列表。该列表称为同步副本，由主节点维护。顾名思义，这些是一组"好的"分片副本，保证已处理所有已向用户确认的索引和删除操作。主分片负责维护这个不变量，因此必须将所有操作复制到该集合中的每个副本。

主分片遵循以下基本流程：

1. 验证传入操作并在结构上拒绝无效操作（例如：有一个对象字段需要一个数字，那么接收到非数字那么就是无效操作）。
2. 在本地执行操作，即索引或删除相关文档。这还将验证字段的内容并在需要时拒绝（例如：关键字值太长，无法在Lucene中建立索引）。
3. 将操作转发到当前同步副本集中的每个副本。如果有多个副本，这是并行完成的。
4. 一旦所有副本都成功执行了操作并响应了主副本，主副本就向客户端确认请求已成功完成。

故障处理

索引过程中可能会出现许多问题（比如：磁盘可能会损坏，节点之间可能会断开连接，或者某些配置错误可能会导致副本上的操作失败，尽管它在主节点上成功）。这些是不常见的但非常重要，必须对它们做出响应。

在主分片本身发生故障的情况下，托管主分片的节点将向主节点发送有关它的消息。索引操作将等待（最多 1 分钟，默认情况下）让主节点将其中一个副本提升为新的主分片。然后该操作将被转发到新的主分片进行处理。请注意，主节点还监视节点的健康状况，并可能决定主动降级主分片。这通常发生在持有主分片的节点因网络问题而与集群隔离时。有关更多详细信息，请参见此处。

一旦在主分片上成功执行操作，主分片在副本分片上执行时必须处理潜在的故障。这可能是由于副本上的实际故障或由于网络问题导致操作无法到达副本（或阻止副本响应）造成的。所有这些共享相同的最终结果：作为同步副本集一部分的副本错过了即将被确认的操作。为了避免违反不变量，主分片向主节点发送消息，请求从同步副本集中删除有问题的分片。只有在主节点确认删除分片后，主分片才会确认该操作。请注意，主节点还将指示另一个节点开始构建新的分片副本，以将系统恢复到健康状态。

在将操作转发到副本时，主分片将使用副本来验证它仍然是活动的主分片。如果主要由于网络分区（或长时间 GC）而被隔离，它可能会在意识到它已被降级之前继续处理传入的索引操作。来自旧主分片的操作将被副本拒绝。当主分片收到来自副本的响应，因为它不再是主分片而拒绝其请求时，它将联系主节点并得知它已被替换。然后将该操作路由到新的主分片。

> 如果没有副本会发生什么？
> 这是一个有效的场景，可能由于索引配置或仅仅因为所有副本都失败而发生。在这种情况下，主要是在没有任何外部验证的情况下处理操作，这可能看起来有问题。另一方面，主分片不能自己使其他分片失败，而是要求主分片代表它这样做。这意味着master知道primary是唯一一个好的副本。因此，我们保证 master 不会将任何其他（过时的）分片副本提升为新的主分片，并且索引到主分片的任何操作都不会丢失。当然，由于那时我们只使用数据的单个副本运行，物理硬件问题可能会导致数据丢失。有关一些缓解选项，请参阅活动分片。

基本读取模型

Elasticsearch中的读取可以是通过 ID 进行的非常轻量级的查找，也可以是具有复杂聚合的繁重搜索请求，这些请求会占用大量 CPU 资源。主备份模型的优点之一是它使所有分片副本保持相同（除了进行中的操作）。因此，单个同步副本足以满足读取请求。

当节点接收到读取请求时，该节点负责将其转发到持有相关分片的节点，整理响应，并响应客户端。我们称该节点为该请求的协调节点。基本流程如下：

1. 解析对相关分片的读取请求。请注意，由于大多数搜索将发送到一个或多个索引，因此它们通常需要从多个分片中读取，每个分片代表不同的数据子集。
2. 从分片复制组中选择每个相关分片的活动副本。这可以是主要的或副本。默认情况下，Elasticsearch 将简单地在分片副本之间进行循环。
3. 将分片级别的读取请求发送到选定的副本。
4. 结合结果并做出回应。请注意，在通过 ID 查找的情况下，只有一个分片是相关的，可以跳过此步骤。

分片失败

当一个分片无法响应读取请求时，协调节点会将请求发送到同一复制组中的另一个分片副本。重复失败可能导致没有可用的分片副本。

为确保快速响应，如果一个或多个分片失败，以下 API 将响应部分结果：

- 搜索
- 多重搜索
- 块
- 多获取

包含部分结果的响应仍会提供 `200 OK` HTTP状态代码。分片失败由响应头的 `timed_out` 和 `_shards` 字段指示。

一些简单的含义

这些基本流程中的每一个都决定了Elasticsearch作为读写系统的行为方式。此外，由于读取和写入请求可以同时执行，这两个基本流程相互影响。这有一些内在的含义：

- 高效读取
  - 在正常操作下，每个相关复制组都会执行一次读取操作。只有在失败的情况下，同一个分片的多个副本才会执行相同的搜索。
- 读取未确认
  - 由于主数据库首先在本地建立索引，然后复制请求，因此并发读取可能在确认更改之前已经看到更改。
- 默认两份
  - 该模型可以容错，同时仅维护两个数据副本。这与基于仲裁的系统形成对比，后者的容错最小副本数为 3。

失败

在失败的情况下，以下是可能的：

- 单个分片会减慢索引速度
  - 因为主分片在每次操作期间都会等待同步副本集中的所有副本，所以单个慢分片可以减慢整个复制组的速度。这就是我们为上面提到的读取效率付出的代价。当然，单个慢分片也会减慢路由到它的不幸搜索。
- 脏读
  - 隔离的主节点可以暴露不会被确认的写入。这是由于一个孤立的主节点只有在向其副本发送请求或与主节点联系时才会意识到它是孤立的。此时，该操作已被索引到主数据库中，并且可以通过并发读取来读取。 Elasticsearch 通过每秒 ping 一次主节点（默认情况下）并在不知道主节点时拒绝索引操作来降低这种风险。

冰山一角

本文档提供了 Elasticsearch 如何处理数据的高级概述。当然，引擎盖下还有很多事情要做。_primary_term、集群状态发布和主节点选举等都在保持该系统正常运行方面发挥着作用。本文档也不涵盖已知和重要的错误（关闭和打开）。我们认识到 GitHub 很难跟上。为了帮助人们掌握这些信息，我们在我们的网站上维护了一个专门的弹性页面。我们强烈建议您阅读它。

创建文档

您可以使用 `_doc` 或 `_create` 索引一个新的JSON文档。

使用 `_create` 保证文档仅在它不存在时才被索引。

要更新现有文档，您必须使用 `_doc` 资源。

请求方式

```
PUT /<index>/_doc/<_id>
POST /<index>/_doc/
PUT /<index>/_create/<_id>
POST /<index>/_create/<_id>
```

路径参数

- `<index>`
  - （必需，字符串）目标索引的名称。默认情况下，如果索引不存在，则会自动创建该索引。
- `<_id>`
  - （可选，字符串）文档的唯一标识符。如果您使用PUT请求，则为必需。使用POST请求时省略自动生成ID。

查询参数

- `if_seq_no`
  - （可选，整数）仅当文档具有此序列号时才执行操作。请参阅乐观并发控制。
- `if_primary_term`
  - （可选，整数）仅当文档具有此primary term时才执行操作。请参阅乐观并发控制。
- `op_type`
  - （可选，枚举）设置为仅在文档不存在时创建索引（如果不存在则放置）。
  - 如果具有指定 `_id` 的文档已经存在，则索引操作将失败。
  - 与使用 `<index>/_create` 端点相同。
  - 有效值：`index、create`。如果指定了文档 ID，则默认为 `index`。否则，它默认 `create`。
- `pipeline`
  - （可选，字符串）用于预处理传入文档的管道 ID。
- `refresh`
  - （可选，枚举）如果为 true，则 Elasticsearch 刷新受影响的分片以使此操作对搜索可见，如果 wait_for 则等待刷新以使此操作对搜索可见，如果为 false，则不执行任何刷新操作。
  - 有效值：`true、false、wait_for`。默认值：`false`。
- `routing`
  - （可选，字符串）以指定的主分片为目标。
- `master_timeout`

    - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 30s。
- `timeout`
    - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 30s。
- `version`
    - （可选，整数）并发控制的显式版本号。指定的版本必须与文档的当前版本匹配，请求才能成功。
- `version_type`
    - （可选，枚举）特定版本类型：`internal`，`external`，`external_gte`.
- `wait_for_active_shards`
    - （可选，字符串）在继续操作之前必须处于活动状态的分片副本数。设置为所有或任何正整数，最多为索引中的分片总数（`number_of_replicas+1`）。默认值：1，主分片。

请求主体

- `<field>`
    - （必需，字符串）请求正文包含文档数据的 JSON 源。

响应主体

- `_shards`
    - 提供有关索引操作的复制过程的信息。
- `_shards.total`
    - 指示应在多少个分片副本（主分片和副本分片）上执行索引操作。
- `_shards.successful`
    - 指示索引操作成功的分片副本数。索引操作成功时，success至少为1。
    - 当索引操作成功返回时，副本分片可能不会全部启动——默认情况下，只需要主分片。设置 `wait_for_active_shards` 以更改此默认行为。
- `_shards.failed`
    - 在副本分片上的索引操作失败的情况下包含与复制相关的错误的数组。 0 表示没有失败。
- `_index`
    - 添加文档的索引的名称。
- `_type`
    - 文档类型。 Elasticsearch 索引现在支持单一的文档类型，`_doc`。
- `_id`
    - 添加文档的唯一标识符。
- `_version`
    - 文档版本。每次更新文档时递增。
- `_seq_no`
    - 为索引操作分配给文档的序列号。序列号用于确保文档的旧版本不会覆盖新版本。
- `_primary_term`
    - 为索引操作分配给文档的_primary_term。
- `result`
    - 索引操作的结果，可能值为：`created` 或 `updated`。

关键描述

自动创建索引

如果指定的索引不存在，默认情况下索引操作会自动创建它并应用任何配置的索引模板。如果不存在映射，则索引操作会创建一个动态映射。默认情况下，如果需要，新字段和对象会自动添加到映射中。

自动索引创建由 `action.auto_create_index` 设置控制。此设置默认为 `true`，它允许自动创建任何索引。您可以修改此设置以明确允许或阻止自动创建与指定模式匹配的索引，或者将其设置为 `false` 以完全禁用自动索引创建。指定您希望允许的以逗号分隔的模式列表，或者在每个模式前加上 + 或 - 以指示是允许还是阻止它。指定列表时，默认行为是禁止。

```
PUT _cluster/settings
{
    "persistent": {
        "action.auto_create_index": "twitter,index10,-index1*,+ind*"
    }
}

PUT _cluster/settings
{
    "persistent": {
        "action.auto_create_index": "false"
    }
}

PUT _cluster/settings
{
    "persistent": {
        "action.auto_create_index": "true"
    }
}
```

- 第一个设置请求：允许自动创建名为 `twitter` 或 `index10` 的索引，阻止创建与模式 `index1*` 匹配的索引，并允许创建与 `ind*` 模式匹配的任何其他索引。模式按照指定的顺序匹配。
- 第二个设置请求：完全禁用自动索引创建。
- 第三个设置请求：允许自动创建任何索引。这是默认设置。

您可以通过使用 `_create` 资源或将 `op_type` 参数设置为 `create` 来强制执行创建操作。在这种情况下，如果索引中已存在具有指定 ID 的文档，则索引操作将失败。

自动创建文档ID

如果您在使用 POST 时不指定文档ID，则会自动将 `op_type` 设置为 `create`，并且索引操作会为文档生成唯一 ID。

```
POST twitter/_doc/
{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elasticsearch"
}
```

乐观并发控制

索引操作可以是有条件的，并且只有在对文档的最后修改分配了由 `if_seq_no` 和 `if_primary_term` 参数指定的序列号和_primary_term时才执行。如果检测到不匹配，该操作将导致 `VersionConflictException` 和状态码 `409`。

路由

默认情况下，分片放置?或路由?，是通过使用文档的id值的散列来控制。对于更明确的控制，可以使用路由参数在每个操作的基础上直接指定输入路由器使用的散列函数的值。例如：

```
POST twitter/_doc?routing=kimchy
{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elasticsearch"
}
```

在此示例中，文档根据提供的路由参数路由到分片：“kimchy”。
在设置显式映射时，还可以使用 `_routing` 字段来指导索引操作从文档本身中提取路由值。这确实需要额外的文档解析传递的（非常小的）成本。如果定义了 `_routing` 映射并将其设置为必需，则如果未提供或提取路由值，则索引操作将失败。

活动分片

为了提高写入系统的弹性，可以将索引操作配置为在继续操作之前等待一定数量的活动分片副本。如果所需数量的活动分片副本不可用，则写入操作必须等待并重试，直到所需分片副本已启动或发生超时。默认情况下，写入操作仅等待主分片处于活动状态后再继续（即 `wait_for_active_shards=1`）。可以通过设置 `index.write.wait_for_active_shards` 在索引设置中动态覆盖此默认值。要更改每个操作的此行为，可以使用 `wait_for_active_shards` 请求参数。

有效值是全部或任何正整数，最多为索引中每个分片配置的副本总数（即 `number_of_replicas + 1`）。指定负值或大于分片副本数的数字将引发错误。

例如，假设我们有一个由三个节点 A、B 和 C 组成的集群，我们创建一个索引 `index`，副本数设置为 3（导致 4 个分片副本，比节点数多一个）。如果我们尝试索引操作，默认情况下，该操作只会确保每个分片的主副本在继续之前可用。这意味着即使 B 和 C 出现故障，并且 A 托管主分片副本，索引操作仍将仅使用数据的一个副本进行。如果在请求中将 `wait_for_active_shards` 设置为 3（并且所有 3 个节点都已启动），那么索引操作将需要 3 个活动分片副本才能继续进行，因为集群中有 3 个活动节点，所以应该满足这一要求，每个节点持有分片的副本。但是，如果我们将 `wait_for_active_shards` 设置为 `all`（或设置为 4，相同），则索引操作将不会继续，因为我们在索引中没有每个 shard 的所有 4 个副本都处于活动状态。除非在集群中启动一个新节点来托管分片的第四个副本，否则该操作将超时。

需要注意的是，此设置大大降低了写入操作未写入所需数量的分片副本的机会，但并不能完全消除这种可能性，因为此检查发生在写入操作开始之前。一旦写入操作开始，复制仍然有可能在任意数量的分片副本上失败，但在主分片上仍然成功。写入操作响应的 `_shards` 部分显示复制 `succeeded/failed` 的分片副本数。

```
{
    "_shards" : {
        "total" : 2,
        "failed" : 0,
        "successful" : 2
    }
}
```

超时

执行索引操作时，分配给执行索引操作的主分片可能不可用。造成这种情况的一些原因可能是主分片当前正在从网关恢复或正在进行重定位。默认情况下，索引操作将在主分片上等待最多 1 分钟，然后才会失败并以错误响应。 timeout 参数可用于明确指定等待的时间。这是将其设置为 5 分钟的示例：

```
PUT twitter/_doc/1?timeout=5m
{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elasticsearch"
}
```

版本控制

每个索引文档都有一个版本号。默认情况下，使用从 1 开始的内部版本控制，每次更新都会增加，包括删除。可选地，版本号可以设置为外部值（例如，如果在数据库中维护）。要启用此功能，应将 `version_type` 设置为 `external`。提供的值必须是大于或等于 0 且小于 9.2e+18 左右的数字长整型值。

使用外部版本类型时，系统会检查传递给索引请求的版本号是否大于当前存储文档的版本。如果为 `true`，文档将被索引并使用新的版本号。如果提供的值小于或等于存储文档的版本号，则会发生版本冲突，索引操作将失败。例如：

```
PUT twitter/_doc/1?version=2&version_type=external
{
    "message" : "elasticsearch now has versioning support, double cool!"
}
```

> **提示：**
>
> 版本控制是完全实时的，不受搜索操作的近实时方面的影响。如果没有提供版本，则执行该操作而不进行任何版本检查。

在前面的示例中，操作将成功，因为提供的版本 2 高于当前文档版本 1。如果文档已经更新并且其版本设置为 2 或更高，则索引命令将失败并导致冲突（409 http 状态码）。

一个好的副作用是，只要使用源数据库中的版本号，就不需要维护由于源数据库更改而执行的异步索引操作的严格顺序。如果使用外部版本控制，即使是使用数据库中的数据更新 Elasticsearch 索引的简单案例也会得到简化，因为如果索引操作因任何原因出现乱序，则只会使用最新版本。

除了外部版本类型，Elasticsearch 还支持针对特定用例的其他类型：

- `internal`
    - 仅当给定版本与存储文档的版本相同时才索引文档。
- `external` 或 `external_gt`

- 仅当给定版本严格高于存储文档的版本或不存在现有文档时才索引文档。给定版本将用作新版本，并将与新文档一起存储。提供的版本必须是非负长整数。

- `external_gte`
  - 仅当给定版本等于或高于存储文档的版本时才索引文档。如果没有现有文档，则操作也会成功。给定版本将用作新版本，并将与新文档一起存储。提供的版本必须是非负长整数。

> 提示：
>
> external_gte 版本类型适用于特殊用例，应谨慎使用。如果使用不当，可能会导致数据丢失。还有另一种选择，强制，它已被弃用，因为它可能导致主分片和副本分片发散。

使用示例

将JSON文档插入到 `_id` 为 1 的 `twitter` 索引中：

```
PUT twitter/_doc/1
{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elasticsearch"
}
```

如果不存在具有该ID的文档，则使用`_create` 资源将文档索引到 `twitter` 索引中：

```
PUT twitter/_create/1
{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elasticsearch"
}
```

如果不存在具有该ID的文档，则将op_type参数设置为create以将文档索引到twitter索引中：

```
PUT twitter/_doc/1?op_type=create
{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elasticsearch"
}
```

获取文档

您使用 GET 从特定索引中检索文档及其源或存储字段。使用 HEAD 验证文档是否存在。您可以使用 `_source` 仅检索文档源或验证它是否存在。

请求方式

```
GET <index>/_doc/<_id>
HEAD <index>/_doc/<_id>
GET <index>/_source/<_id>
HEAD <index>/_source/<_id>
```

路径参数

- `<index>`
  - （必需，字符串）包含文档的索引的名称。
- `<_id>`
  - （必需，字符串）文档的唯一标识符。

查询参数

- `preference`
  - （可选，字符串）指定应在其上执行操作的节点或分片。默认随机。
- `realtime`
  - （可选，布尔值）如果为 true，则请求是实时的，而不是接近实时的。默认为true。
- `refresh`
  - （可选，布尔值）如果为 true，则请求在检索文档之前刷新相关分片。默认为false。
- `routing`
  - （可选，字符串）以指定的主分片为目标。
- `stored_fields`
  - （可选，布尔值）如果为 true，则检索存储在索引中的文档字段，而不是文档 `_source`。默认为false。
- `_source`
  - （可选，字符串）true 或 false 以返回或不返回 `_source` 字段，或要返回的字段列表。
- `_source_excludes`
  - （可选，字符串）要从返回的 `_source` 字段中排除的字段列表。
- `_source_includes`
  - （可选，字符串）要从 `_source` 字段中提取和返回的字段列表。
- `version`
  - （可选，整数）并发控制的显式版本号。指定的版本必须与文档的当前版本匹配，请求才能成功。
- `version_type`
  - （可选，枚举）特定版本类型： `internal`, `external`, `external_gte`。

响应主体

- `_index`
  - 文档所属索引的名称。
- `_type`
  - 文档类型。 Elasticsearch 索引现在支持单一的文档类型， `_doc`。
- `_id`
  - 文档的唯一标识符。
- `_version`
  - 文档版本。每次更新文档时递增。
- `_seq_no`
  - 为索引操作分配给文档的序列号。序列号用于确保文档的旧版本不会覆盖新版本。
- `_primary_term`

    - 为索引操作分配给文档的_primary_term。
- `found`
    - 指示文档是否存在：true 或 false。
- `_routing`
    - 显式路由（如果已设置）。
- `_source`
    - 如果 `found` 为真，则包含 JSON 格式的文档数据。如果 `_source` 参数设置为 `false` 或 `stored_fields` 参数设置为 `true`，则排除。
- `_fields`
    - 如果`stored_fields`参数设置为`true`并且`found`为`true`，则包含存储在索引中的文档字段。

关键描述

即时的

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）。如果请求存储字段（请参阅`stored_fields`参数）并且文档已更新但尚未刷新，则获取API将必须解析和分析源以提取存储字段。为了禁用实时获取，可以将`realtime`参数设置为`false`。

源过滤

默认情况下，获取操作会返回 `_source` 字段的内容，除非您使用了 `stored_fields` 参数或 `_source` 字段被禁用。您可以使用 `_source` 参数关闭 `_source` 检索：

```
GET /twitter/_doc/0?_source=false
```

如果您只需要 `_source` 中的一两个字段，请使用 `_source_includes` 或 `_source_excludes` 参数来包含或过滤掉特定字段。这对于部分检索可以节省网络开销的大型文档特别有用。这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 /twitter/_doc/0?_source_includes=*.id&_source_excludes=entities
```

如果您只想指定包含，则可以使用较短的符号：

```
GET /twitter/_doc/0?_source=*.id,retweeted
```

路由

如果在索引期间使用路由，则还需要指定路由值来检索文档。例如：

```
GET /twitter/_doc/2?routing=user1
```

此请求获取 id 为 2 的推文，但它是基于用户路由的。如果未指定正确的路由，则不会获取文档。

首选项

控制执行获取请求的分片副本的首选项。默认情况下，操作在分片副本之间是随机的。

首选项可以设置为：

- `_local`

    - 如果可能，该操作将更倾向于在本地分配的分片上执行。
- `自定义（字符串）值`
    - 自定义值将用于保证相同的分片将用于相同的自定义值。当在不同的刷新状态下击中不同的分片时，这有助于“跳跃值”。简单的做法，可以使用Web会话ID或用户名之类的。

刷新

可以将 refresh 参数设置为 true，以便在获取操作之前刷新相关分片并使其可搜索。应在仔细考虑并验证这不会导致系统负载过重（并减慢索引速度）之后将其设置为 true。

分散式

获取操作被散列到特定的分片id中。然后它被重定向到该分片id内的副本之一并返回结果。副本是主分片及其在该分片ID组中的副本。这意味着我们拥有的副本越多，我们将拥有更好的获取扩展。

版本控制支持

仅当文档的当前版本等于指定的版本时，您才可以使用 `version` 参数来检索文档。

在内部，Elasticsearch 已将旧文档标记为已删除并添加了一个全新的文档。文档的旧版本不会立即消失，尽管您将无法访问它。当您继续索引更多数据时，Elasticsearch 会在后台清理已删除的文档。

使用示例

从 twitter 索引中检索 _id 为 0 的 JSON 文档：

```
GET /twitter/_doc/0
```

检查是否存在 _id 为 0 的文档：

```
HEAD /twitter/_doc/0
```

Elasticsearch 返回状态码，如果文档存在则返回 `200 - OK`，如果不存在则返回 `404 - Not Found`。

使用 `<index>/_source/<id>` 资源仅获取文档的 `_source` 字段。例如：

```
GET /twitter/_source/1
```

您可以使用源过滤参数来控制 `_source` 的哪些部分被返回：

```
GET /twitter/_source/1/?_source_includes=*.id&_source_excludes=entities
```

您可以使用 `HEAD` 和 `_source` 端点来有效地测试文档 `_source` 是否存在。如果在映射中禁用了文档源，则该文档的源不可用。

```
HEAD /twitter/_source/1
```

使用 `stored_fields` 参数来指定要检索的存储字段集。任何未存储的请求字段都将被忽略。例如考虑以下映射：

```
PUT /twitter
{
   "mappings": {
```

```
        "properties": {
           "counter": {
              "type": "integer",
              "store": false
           },
           "tags": {
              "type": "keyword",
              "store": true
           }
        }
    }
}
```

现在我们可以添加一个文档：

```
PUT /twitter/_doc/1
{
    "counter" : 1,
    "tags" : ["red"]
}
```

然后尝试检索它：

```
GET /twitter/_doc/1?stored_fields=tags,counter
```

API 返回以下结果：

```
{
    "_index": "twitter",
    "_type": "_doc",
    "_id": "1",
    "_version": 1,
    "_seq_no" : 22,
    "_primary_term" : 1,
    "found": true,
    "fields": {
       "tags": [
          "red"
       ]
    }
}
```

从文档本身获取的字段值始终作为数组返回。由于未存储计数器字段，因此获取请求将忽略它。
您还可以检索元数据字段，例如 `_routing` 字段：

```
PUT /twitter/_doc/2?routing=user1
{
    "counter" : 1,
    "tags" : ["white"]
}
```

```
GET /twitter/_doc/2?routing=user1&stored_fields=tags,counter
```

API 返回以下结果：

```
{
   "_index": "twitter",
   "_type": "_doc",
   "_id": "2",
   "_version": 1,
   "_seq_no" : 13,
   "_primary_term" : 1,
   "_routing": "user1",
   "found": true,
   "fields": {
      "tags": [
         "white"
      ]
   }
}
```

使用 `stored_field` 选项只能检索叶字段。无法返回对象字段——如果指定，则请求失败。

删除文档

从指定索引中删除 JSON 文档。

请求方式

```
DELETE /<index>/_doc/<_id>
```

路径参数

- `<index>`
  - （必需，字符串）目标索引的名称。
- `<_id>`
  - （必需，字符串）文档的唯一标识符。

查询参数

- `if_seq_no`
  - （可选，整数）仅当文档具有此序列号时才执行操作。请参阅乐观并发控制。
- `if_primary_term`
  - （可选，整数）仅当文档具有此*primary_term时才执行操作。请参阅[乐观并发控制(https://www.elastic.co/guide/en/elasticsearch/reference/7.6/docs-index*.html#optimistic-concurrency-control-index)。
- `refresh`
  - （可选，枚举）如果为 true，则 Elasticsearch 刷新受影响的分片以使此操作对搜索可见，
  - 如果 wait_for 则等待刷新以使此操作对搜索可见，
  - 如果为 false，则不执行任何刷新操作。
  - 有效值：true、false、wait_for。默认值：false。
- `routing`
  - （可选，字符串）以指定的主分片为目标。
- `master_timeout`

    - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 30s。
- `timeout`
    - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 30s。
- `version`
    - （可选，整数）并发控制的显式版本号。指定的版本必须与文档的当前版本匹配，请求才能成功。
- `version_type`
    - （可选，枚举）特定版本类型：`internal`, `external`, `external_gte`。
- `wait_for_active_shards`
    - （可选，字符串）在继续操作之前必须处于活动状态的分片副本数。
    - 设置为所有或任何正整数，最多为索引中的分片总数 (number_of_replicas+1)。默认值：1。

使用示例

从 twitter 索引中删除 JSON 文档 1：

```
DELETE /twitter/_doc/1
```

如果在索引过程中使用了路由，还需要指定路由值来删除文档。
如果 `_routing` 映射设置为 required 且未指定路由值，则删除 API 将引发 RoutingMissingException 并拒绝请求。
例如：

```
DELETE /twitter/_doc/1?routing=kimchy
```

此请求会删除 id 为 1 的推文，但它是基于用户进行路由的。如果未指定正确的路由，则不会删除文档。

`timeout` 参数可用于明确指定等待的时间。这是将其设置为 5 分钟的示例：

```
DELETE /twitter/_doc/1?timeout=5m
```

更新文档

使您能够编写文档更新脚本。该脚本可以更新、删除或跳过修改文档。更新 API 还支持传递部分文档，该部分文档会合并到现有文档中。要完全替换现有文档，请使用添加索引文档API。

这个操作：

1. 从索引中获取文档。
2. 运行指定的脚本。
3. 索引结果。

该文档仍必须重新索引，但使用`update`消除了一些网络往返，也能够减少了获取和索引操作之间版本冲突的可能性。
必须启用 `_source` 字段才能使用更新。除了 `_source`，您还可以通过`ctx`映射访问以下变量：`_index`、`_type`、`_id`、`_version`、`_routing` 和 `_now`（当前时间戳）。

关于脚本请参考：https://www.elastic.co/guide/en/elasticsearch/reference/7.6/modules-scripting.html

请求方式

```
POST /<index>/_update/<_id>
```

路径参数

- `<index>`
  - （必需，字符串）目标索引的名称。默认情况下，如果索引不存在，则会自动创建该索引。
- `<_id>`
  - （必需，字符串）要更新的文档的唯一标识符。

查询参数

- `if_seq_no`
  - （可选，整数）仅当文档具有此序列号时才执行操作。请参阅乐观并发控制。
- `if_primary_term`
  - （可选，整数）仅当文档具有此*primary_term时才执行操作。请参阅[乐观并发控制(https://www.elastic.co/guide/en/elasticsearch/reference/7.6/docs-index*.html#optimistic-concurrency-control-index)。
- `lang`
  - （可选，字符串）脚本语言。默认值：`painless`。
- `refresh`
  - （可选，枚举）如果为 true，则 Elasticsearch 刷新受影响的分片以使此操作对搜索可见，
  - 如果 wait_for 则等待刷新以使此操作对搜索可见，
  - 如果为 false，则不执行任何刷新操作。
  - 有效值：true、false、wait_for。默认值：false。
- `retry_on_conflict`
  - （可选，整数）指定发生冲突时应重试操作的次数。默认值：0。
- `routing`
  - （可选，字符串）以指定的主分片为目标。
- `routing`
  - （可选，字符串）以指定的主分片为目标。
- `stored_fields`
  - （可选，布尔值）如果为 true，则检索存储在索引中的文档字段，而不是文档 `_source`。默认为false。
- `_source`
  - （可选，列表）设置为 false 以禁用源检索（默认值：true）。
  - 您还可以指定要检索的字段的逗号分隔列表。
- `_source_excludes`
  - （可选，字符串）要从返回的 `_source` 字段中排除的字段列表。
- `_source_includes`
  - （可选，字符串）要从 `_source` 字段中提取和返回的字段列表。
- `master_timeout`
  - （可选，时间单位）指定等待连接到主节点的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 30s。
- `timeout`

    - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为 30s。
- `wait_for_active_shards`
    - （可选，字符串）在继续操作之前必须处于活动状态的分片副本数。
    - 设置为所有或任何正整数，最多为索引中的分片总数 (number_of_replicas+1)。默认值：1。

使用示例

首先，让我们索引一个简单的文档：

```
PUT test/_doc/1
{
    "counter" : 1,
    "tags" : ["red"]
}
```

要增加计数器，您可以使用以下脚本提交更新请求：

```
POST test/_update/1
{
    "script" : {
        "source": "ctx._source.counter += params.count",
        "lang": "painless",
        "params" : {
            "count" : 4
        }
    }
}
```

同样，您可以使用和更新脚本将标签添加到标签列表中（这只是一个列表，因此即使存在标签也会添加标签）：

```
POST test/_update/1
{
    "script" : {
        "source": "ctx._source.tags.add(params.tag)",
        "lang": "painless",
        "params" : {
            "tag" : "blue"
        }
    }
}
```

您还可以从标签列表中删除标签。删除标签的Painless函数采用要remove的元素的数组索引。为避免可能的运行时错误，您首先需要确保标签存在。如果列表包含重复的标记，则此脚本仅删除一次出现。

```
POST test/_update/1
{
    "script" : {
        "source": "if (ctx._source.tags.contains(params.tag)) {
ctx._source.tags.remove(ctx._source.tags.indexOf(params.tag)) }",
        "lang": "painless",
        "params" : {
            "tag" : "blue"
        }
    }
}
```

您还可以在文档中添加和删除字段。例如，此脚本添加字段 new_field：

```
POST test/_update/1
{
    "script" : "ctx._source.new_field = 'value_of_new_field'"
}
```

相反，此脚本删除字段 new_field：

```
POST test/_update/1
{
    "script" : "ctx._source.remove('new_field')"
}
```

除了更新文档之外，您还可以更改从脚本中执行的操作。例如，如果 tags 字段包含green，此请求将删除 doc，否则它什么也不做（noop）：

```
POST test/_update/1
{
    "script" : {
        "source": "if (ctx._source.tags.contains(params.tag)) { ctx.op =
'delete' } else { ctx.op = 'none' }",
        "lang": "painless",
        "params" : {
            "tag" : "green"
        }
    }
}
```

以下部分更新为现有文档添加了一个新字段：

```
POST test/_update/1
{
    "doc" : {
        "name" : "new_name"
    }
}
```

如果同时指定了 `doc` 和 `script`，则忽略 `doc`。如果您指定脚本更新，请在脚本中包含要更新的字段。

默认情况下，不会更改任何内容的更新会检测到它们没有更改任何内容并返回“result”:“noop”：

```
POST test/_update/1
{
    "doc" : {
        "name" : "new_name"
    }
}
```

如果 name 的值已经是 new_name，则忽略更新请求，响应中的结果元素返回 noop：

```
{
  "_index" : "test",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 2,
  "result" : "noop",
  "_shards" : {
    "total" : 0,
    "successful" : 0,
    "failed" : 0
  },
  "_seq_no" : 8,
  "_primary_term" : 1
}
```

您可以通过设置“detect_noop”禁用此行为：false：

```
POST test/_update/1
{
    "doc" : {
        "name" : "new_name"
    },
    "detect_noop": false
}
```

如果文档不存在，则将 upsert 元素的内容作为新文档插入。如果文档存在，则执行脚本：

```
POST test/_update/1
{
    "script" : {
        "source": "ctx._source.counter += params.count",
        "lang": "painless",
        "params" : {
            "count" : 4
        }
    },
    "upsert" : {
        "counter" : 1
    }
}
```

要在文档存在与否都执行`script`，请将`scripted_upsert`设置为 true：

```
POST test/_update/3
{
    "scripted_upsert":true,
    "script" : {
        "source": "ctx._source.counter += params.count",
        "lang": "painless",
        "params" : {
            "count" : 4
        }
    },
    "upsert" : {
        "counter" : 1
    }
}
```

您可以将 `doc_as_upsert` 设置为 `true` 以使用 `doc` 的内容作为 `upsert` 值，当索引不存在的时候，也可以同时将 `doc` 部分添加 `upsert` 文档中：

```
POST test/_update/1
{
    "doc" : {
        "name" : "new_name"
    },
    "doc_as_upsert" : true
}
```

多文档获取

前面我们都是操作单个文档，有的时候我们希望操作多文档

您可以使用 mget 从一个或多个索引中检索多个文档。如果在请求 URI 中指定索引，则只需在请求正文中指定文档 ID。

请求方式

```
GET /_mget
GET /<index>/_mget
```

路径参数

- `<index>`
    - （可选，字符串）指定ids或docs数组中的文档未指定索引时，设定检索文档的索引名称。

查询参数

- `preference`
    - （可选，字符串）指定应在其上执行操作的节点或分片。默认随机。
- `realtime`
    - （可选，布尔值）如果为 true，则请求是实时的，而不是接近实时的。默认为true。
- `refresh`
    - （可选，布尔值）如果为 true，则请求在检索文档之前刷新相关分片。默认为false。
- `routing`

    - （可选，字符串）以指定的主分片为目标。
- `stored_fields`
    - （可选，布尔值）如果为 true，则检索存储在索引中的文档字段，而不是文档 `_source`。默认为false。
- `_source`
    - （可选，字符串）true 或 false 以返回或不返回 `_source` 字段，或要返回的字段列表。
- `_source_excludes`
    - （可选，字符串）要从返回的 `_source` 字段中排除的字段列表。
- `_source_includes`
    - （可选，字符串）要从 `_source` 字段中提取和返回的字段列表。

请求主体

- `docs`
    - （可选，数组）要检索的文档。
    - 如果请求 URI 中未指定索引，则为必需。
    - 您可以为每个文档指定以下属性：
        - `_id`
            - （必需，字符串）唯一的文档
        - `_index`
            - （可选，字符串）包含文档的索引。如果请求 URI 中未指定索引，则为必需。
        - `_routing`
            - （可选，字符串）文档所在的主分片的键。如果在索引期间使用路由，则需要。
        - `_source`
            - （可选，布尔值）如果为 false，则排除所有 `_source` 字段。默认为true。
        - `source_include`
            - （可选，数组）要从 `_source` 字段中提取和返回的字段。
        - `source_exclude`
            - （可选，数组）要从返回的 `_source` 字段中排除的字段。
        - `_stored_fields`
            - （可选，数组）要检索的存储字段。
- `ids`
    - （可选，数组）要检索的文档的ID。在请求URI中指定索引时允许。

响应主体

响应包括一个 docs 数组，其中包含按请求中指定的顺序排列的文档。返回文档的结构与获取索引文档返回的类似。如果获取特定文档失败，则会包含错误以代替文档。

使用示例

通过ID获取文档

如果您在请求URI中指定索引，则请求正文中只需要文档 ID：

```
GET /test/_mget
{
    "docs" : [
        {
            "_id" : "1"
        },
        {
            "_id" : "2"
        }
    ]
}
```

也可以通过IDS来获取

```
GET /test/_mget
{
    "ids" : ["1", "2"]
}
```

过滤源字段

默认情况下，为每个文档（如果已存储）返回 `_source` 字段。使用 `_source` 和 `_source_include` 或 `_source_exclude` 属性来过滤为特定文档返回的字段。

您可以在请求 URI 中包含 `_source`、`_source_includes` 和 `_source_excludes` 查询参数，以指定在没有每个文档说明时使用的默认值。

例如，以下请求将文档 1 的 `_source` 设置为 `false` 以完全排除源，从文档 2 中检索 `field3` 和 `field4`，并从文档 3 中检索 `user` 字段但过滤掉 `user.location` 字段。

```
GET /_mget
{
    "docs" : [
        {
            "_index" : "test",
            "_id" : "1",
            "_source" : false
        },
        {
            "_index" : "test",
            "_id" : "2",
            "_source" : ["field3", "field4"]
        },
        {
            "_index" : "test",
            "_id" : "3",
            "_source" : {
                "include": ["user"],
                "exclude": ["user.location"]
            }
        }
    ]
}
```

获取存储的字段

使用 `stored_fields` 属性来指定要检索的存储字段集。任何未存储的请求字段都将被忽略。您可以在请求 URI 中包含 `stored_fields` 查询参数，以指定在没有每个文档说明时使用的默认值。

例如，以下请求从文档 1 中检索 `field1` 和 `field2`，从文档 2 中检索 `field3` 和 `field4`：

```
GET /_mget
{
    "docs" : [
        {
            "_index" : "test",
            "_id" : "1",
            "stored_fields" : ["field1", "field2"]
        },
        {
            "_index" : "test",
            "_id" : "2",
            "stored_fields" : ["field3", "field4"]
        }
    ]
}
```

以下请求默认从所有文档中检索 field1 和 field2。这些默认字段为文档 1 返回，但被覆盖以返回文档 2 的 field3 和 field4。

```
GET /test/_mget?stored_fields=field1,field2
{
    "docs" : [
        {
            "_id" : "1"
        },
        {
            "_id" : "2",
            "stored_fields" : ["field3", "field4"]
        }
    ]
}
```

指定文档路由

如果在索引过程中使用了路由，则需要指定路由值来检索文档。比如下面的请求从路由键key1对应的shard中取出`test/_doc/2`，从路由键key2对应的shard中取出`test/_doc/1`。

```
GET /_mget?routing=key1
{
    "docs" : [
        {
            "_index" : "test",
            "_id" : "1",
            "routing" : "key2"
        },
        {
            "_index" : "test",
```

```
            "_id" : "2"
        }
    ]
}
```

4 映射

通过前面的索引的学习我们初步了解了如何利用rest api操作映射，但是不够细致，从这章开始我们来详细讨论映射。

4.1 相关概念

映射是定义文档及其包含的字段如何存储和索引的过程。例如，使用映射来定义：

- 哪些字符串字段应被视为全文字段。
- 哪些字段包含数字、日期或地理位置。
- 日期值的格式。
- 自定义规则来控制动态添加字段的映射。

映射定义具有：

- 元字段
  - 元字段用于自定义如何处理文档的相关元数据。
  - 元字段的示例包括文档的 `_index`、 `_id` 和 `_source` 字段。
- 字段或属性
  - 映射包含与文档相关的字段或属性列表。

在7.0.0之前，映射定义包含了一个类型名称。有关详细信息，请参阅删除映射类型。

字段数据类型

每个字段都有一个数据类型，可以是：

- 一个简单的类型，如文本(text)、关键字(keyword)、日期(date)、长整形(long)、双精度(double)、布尔值(boolean)或 ip。
- 一种支持 JSON 分层特性的类型，例如对象或嵌套。
- 或特殊类型，如 geo_point、geo_shape 或completion。

出于不同目的以不同方式索引同一字段通常很有用。

例如，可以将字符串字段索引为全文搜索的文本字段，以及用于排序或聚合的关键字字段。

或者，您可以使用标准分析器、英语分析器和法语分析器来索引字符串字段。

这就是多领域的目的。大多数数据类型通过 `fields` 参数支持多字段。

阻止映射爆炸增长的设置

在索引中定义太多字段会导致映射爆炸，这可能会导致内存不足错误和难以恢复的情况。这个问题可能比预期的更常见。例如，考虑这样一种情况，其中插入的每个新文档都会引入新字段。这在动态映射中很常见。每次文档包含新字段时，这些字段最终都会出现在索引的映射中。对于少量数据，这并不令人担忧，但随着映射的增长，它可能会成为一个问题。以下设置允许您限制可以手动或动态创建的字段映射的数量，以防止不良文档导致映射爆炸：

- `index.mapping.total_fields.limit`
  - 索引中的最大字段数。字段和对象映射以及字段别名计入此限制。默认值为1000。

> 该限制是为了防止映射和搜索变得过大。较高的值会导致性能下降和内存问题，尤其是在负载高或资源少的集群中。
> 如果您增加此设置，我们建议您也增加 `indices.query.bool.max_clause_count` 设置，该设置限制查询中布尔子句的最大数量。

- `index.mapping.depth.limit`
  - 字段的最大深度，以内部对象的数量来衡量。例如，如果所有字段都在根对象级别定义，则深度为1。如果有一个对象映射，则深度为2，以此类推。默认值为 20。
- `index.mapping.nested_fields.limit`
  - 索引中不同嵌套映射的最大数量，默认为50。
- `index.mapping.nested_objects.limit`
  - 单个文档中所有嵌套类型的最大嵌套JSON对象数，默认为10000。
- `index.mapping.field_name_length.limit`
  - 设置字段名称的最大长度。默认值为 Long.MAX_VALUE（无限制）。
  - 此设置并不是真正解决映射爆炸的问题，但如果您想限制字段长度，它可能仍然有用。
  - 通常不需要设置此设置。除非用户开始添加大量名称很长的字段，否则默认值是可以的。

动态映射

字段和映射类型在使用前不需要定义。多亏了动态映射，新的字段名称将自动添加，只需索引文档即可。新字段既可以添加到顶级映射类型，也可以添加到内部对象和嵌套字段。

可以配置动态映射规则以自定义用于新字段的映射。

显式映射

您对数据的了解比Elasticsearch猜测的要多，因此虽然动态映射对入门很有用，但在某些时候您会想要指定自己的显式映射。

您可以在创建索引并将字段添加到现有索引时创建字段映射。

您可以使用创建索引API创建具有显式映射的新索引。

```
PUT /my-index
{
  "mappings": {
    "properties": {
      "age":    { "type": "integer" },
      "email":  { "type": "keyword"  },
      "name":   { "type": "text"  }
    }
  }
}
```

您可以使用添加映射API将一个或多个新字段添加到现有索引。

以下示例添加了employee-id，这是一个索引映射参数值为false的关键字字段。这意味着employee-id字段的值已存储但未编入索引或可用于搜索。

```
PUT /my-index/_mapping
{
  "properties": {
    "employee-id": {
      "type": "keyword",
      "index": false
    }
  }
}
```

除支持的映射参数外，您不能更改现有字段的映射或字段类型。更改现有字段可能会使已编入索引的数据无效。
如果您需要更改字段的映射，请创建一个具有正确映射的新索引，并将您的数据重新索引到该索引中。

重命名字段将使已在旧字段名称下索引的数据无效。相反，添加别名字段以创建备用字段名称。

您可以使用获取映射API查看现有索引的映射。

```
GET /my-index/_mapping
```

API 返回以下响应：

```
{
  "my-index" : {
    "mappings" : {
      "properties" : {
        "age" : {
          "type" : "integer"
        },
        "email" : {
          "type" : "keyword"
        },
        "employee-id" : {
          "type" : "keyword",
          "index" : false
        },
        "name" : {
          "type" : "text"
        }
      }
    }
  }
}
```

如果您只想查看一个或多个特定字段的映射，您可以使用获取字段映射 API。
如果您不需要索引的完整映射或者您的索引包含大量字段，这将很有用。

以下请求检索employee-id 字段的映射。

```
GET /my-index/_mapping/field/employee-id
```

API 返回以下响应：

```
```

```
{
  "my-index" : {
    "mappings" : {
      "employee-id" : {
        "full_name" : "employee-id",
        "mapping" : {
          "employee-id" : {
            "type" : "keyword",
            "index" : false
          }
        }
      }
    }
  }
}
```

4.2 字段数据类型

Elasticsearch支持文档中字段的多种不同数据类型：

类型总览

核心数据类型

- **字符串**

  `text` and `keyword`

- **数字**

  `long`, `integer`, `short`, `byte`, `double`, `float`, `half_float`, `scaled_float`

- **日期**

  `date`

- **日期纳秒**

  `date_nanos`

- **布尔**

  `boolean`

- **二进制**

  `binary`

- **范围**

  `integer_range`, `float_range`, `long_range`, `double_range`, `date_range`, `ip_range`

复杂数据类型

- **对象**

  `object` 单个JSON对象

- **嵌套**

  `nested` 嵌套在 JSON 对象数组中

地理数据类型

- **地理点**

  `geo_point` 纬度/经度点

- **地理形状**

  `geo_shape` 用于多边形等复杂形状

特殊数据类型

- **IP**

  `ip` 用于IPv4和IPv6地址

- **完成数据类型**

  `completion` 提供自动完成建议

- **令牌计数**

  `token_count` 计算字符串中的标记数

- `mapper-murmur3`

  `murmur3` 在索引时计算值的哈希并将它们存储在索引中

- `mapper-annotated-text`

  `annotated-text` 用于索引包含特殊标记的文本（通常用于识别命名实体）

- **渗滤器**

  接受来自 query-dsl 的查询

- **连接**

  为同一索引中的文档定义父/子关系

- **Rank feature**

  记录数字特征以在查询时提高点击率。

- **Rank features**

  记录数字特征以在查询时提高点击率。

- **密集矢量**

  记录浮点值的密集向量。

- **稀疏向量**

  记录浮点值的稀疏向量。

- **键入即搜索**

  为查询优化的类似文本的字段，以实现“键入时完成”

- **别名**

  定义现有字段的别名。

- **扁平化**

  允许将整个 JSON 对象作为单个字段进行索引。

- **形状**

  `shape` 任意笛卡尔几何形状。

- **直方图**

`histogram` 百分位数聚合的预聚合数值的直方图。

数组

在 Elasticsearch 中，数组不需要专用的字段数据类型。默认情况下，任何字段都可以包含零个或多个值，但是，数组中的所有值必须是相同的数据类型。请参阅数组.

多字段

出于不同目的以不同方式索引同一字段通常很有用。例如，字符串字段可以映射为全文搜索的 `text` 字段，以及排序或聚合的 `keyword` 字段。或者，您可以使用标准分析器、英语分析器和法语分析器来索引文本字段。

这就是多字段的目的。大多数数据类型通过 `fields` 参数支持多字段。

字符串

`text`

用于索引全文值的字段，例如电子邮件正文或产品描述。对这些字段进行分析，也就是说，它们在被索引之前通过分析器将字符串转换为单个词语的列表。分析过程允许 Elasticsearch 在每个全文字段中搜索单个单词。文本字段不用于排序，也很少用于聚合（尽管重要的文本聚合是一个明显的例外）。

如果您需要索引结构化内容，例如电子邮件地址、主机名、状态代码或标签，您可能应该使用 `keyword` 字段。

下面是一个文本字段的映射示例：

```
PUT my_index
{
  "mappings": {
    "properties": {
      "full_name": {
        "type":  "text"
      }
    }
  }
}
```

将字段用作文本和关键字

有时同时拥有同一字段的全文 (text) 和关键字 (keyword) 版本很有用：一个用于全文搜索，另一个用于聚合和排序。这可以通过多字段来实现。

文本字段的参数

`text` 字段接受以下参数：：

字段	描述
`analyzer`	在索引时间和搜索时间都应该用于文本字段的分析器（除非被 search_analyzer 覆盖）。默认为默认索引分析器或 `standard` 分析器
`boost`	映射字段级查询时间提升。接受浮点数，默认为 1.0。

字段	描述
eager_global_ordinals	是否应该在刷新时急切地加载全局序数？接受true或false（默认）。对于经常用于（重要）词语聚合的字段，启用此功能是一个好主意。
fielddata	该字段可以使用内存中的字段数据进行排序、聚合或脚本吗？接受 true 或 false（默认）。
fielddata_frequency_filter	允许在启用 fielddata 时决定将哪些值加载到内存中的专属设置。默认情况下会加载所有值。
fields	多字段允许为不同目的以多种方式索引相同的字符串值，例如一个用于搜索的字段和一个用于排序和聚合的多字段，或者由不同的分析器分析相同的字符串值。
index	该字段应该是可搜索的吗？接受 true（默认）或 false。
index_options	哪些信息应存储在索引中，用于搜索和突出显示。默认为 positions。
index_prefixes	如果启用，2 到 5 个字符之间的词语前缀将被索引到单独的字段中。这允许前缀搜索以更大的索引为代价更有效地运行。
index_phrases	如果启用，两个词的组合（shingles）将被索引到一个单独的字段中。这允许以更大的索引为代价，更有效地运行精确的短语查询（no slop）。请注意，当停用词未被删除时，此方法效果最佳，因为包含停用词的短语将不使用辅助字段，并将回退到标准短语查询。接受 true 或 false （默认）。
norms	对查询进行评分时是否应考虑字段长度。接受 true 或 false （默认）。
position_increment_gap	应该在字符串数组的每个元素之间插入的伪词语位置的数量。默认为分析器上配置的 position_increment_gap，默认为 100。选择 100 是因为它可以防止具有相当大的斜率（小于 100）的短语查询跨字段值匹配词语。
store	字段值是否应与 _source 字段分开存储和检索。 接受true 或 false （默认）。
search_analyzer	在 text 字段上搜索时应使用的分析器。默认为 analyzer 设置。
search_quote_analyzer	遇到短语时应在搜索时使用的分析器。默认为 search_analyzer 设置。
similarity	应该使用哪种评分算法或相似度。默认为 BM25。
term_vector	是否应为该字段存储词语向量。默认为 no。
meta	关于字段的元数据。

`keyword`

用于索引结构化内容的字段，例如 ID、电子邮件地址、主机名、状态代码、邮政编码或标签。

它们通常用于过滤（查找所有发布状态的博客文章）、排序和聚合。关键字字段只能按其确切值进行搜索。

如果您需要索引全文内容，例如电子邮件正文或产品描述，您可能应该使用`text`字段。

以下是关键字字段的映射示例：

```
PUT my_index
{
  "mappings": {
    "properties": {
      "tags": {
        "type":  "keyword"
      }
    }
  }
}
```

关键字字段的参数

关键字字段接受以下参数：

字段	描述
`boost`	映射字段级查询时间提升。接受浮点数，默认为 1.0。
`doc_values`	该字段是否应该以列跨步的方式存储在磁盘上，以便以后可以用于排序、聚合或脚本？接受`true`（默认）或`false`。
`eager_global_ordinals`	是否应该在刷新时急切地加载全局序数？接受`true`或`false`（默认）。对于经常用于（重要）词语聚合的字段，启用此功能是一个好主意。
`fields`	多字段允许为不同目的以多种方式索引相同的字符串值，例如一个用于搜索的字段和一个用于排序和聚合的多字段，或者由不同的分析器分析相同的字符串值。
`ignore_above`	不要索引任何长于该值的字符串。默认为 2147483647，以便接受所有值。但是请注意，默认动态映射规则会创建一个子关键字字段，通过设置`ignore_above: 256`来覆盖此默认值。
`index`	该字段应该是可搜索的吗？接受`true`（默认）或`false`。
`index_options`	哪些信息应存储在索引中，用于搜索和突出显示。默认为`positions`。
`norms`	对查询进行评分时是否应考虑字段长度。接受`true`或`false`（默认）。

字段	描述
null_value	接受替换任何显式空值的字符串值。默认为 null，这意味着该字段被视为缺失。
store	字段值是否应与 _source 字段分开存储和检索。 接受 true 或 false （默认）。
similarity	应该使用哪种评分算法或相似度。默认为 BM25。
normalizer	如何在索引之前对关键字进行预处理。默认为 null，这意味着关键字保持原样。
split_queries_on_whitespace	为该字段构建查询时，全文查询是否应在空格上拆分输入。接受 true 或 false （默认）。
meta	关于字段的元数据。

> TIP:
>
> 从 2.x 导入的索引不支持keyword。相反，他们会尝试将关键字降级为字符串。这允许您将现代映射与旧映射合并。在升级到 6.x 之前必须重新创建长期存在的索引，但映射降级使您有机会按照自己的时间表进行重新创建。

数字类型

支持以下数字类型：

类型	描述
long	一个有符号的64位整数，最小值为 -2^63，最大值为 2^63-1。
integer	一个带符号的32位整数，最小值为 -2^31，最大值为 2^31-1。
short	一个带符号的16位整数，最小值为 -32,768，最大值为 32,767。
byte	一个带符号的8位整数，最小值为 -128，最大值为 127。
double	双精度 64 位 IEEE 754 浮点数，限制为有限值。
float	单精度 32 位 IEEE 754 浮点数，限制为有限值。
half_float	半精度 16 位 IEEE 754 浮点数，限制为有限值。
scaled_float	由 long 支持的浮点数，按固定的双倍缩放因子缩放。

下面是使用数字字段配置映射的示例：

```
PUT my_index
{
  "mappings": {
    "properties": {
      "number_of_bytes": {
        "type": "integer"
      },
      "time_in_seconds": {
```

```
        "type": "float"
      },
      "price": {
        "type": "scaled_float",
        "scaling_factor": 100
      }
    }
  }
}
```

> 提示
>
> double、float 和 half_float 类型认为 -0.0 和 +0.0 是不同的值。因此，在 -0.0 上进行词语查询将不匹配 +0.0，反之亦然。范围查询也是如此：如果上限为 -0.0，则 +0.0 将不匹配，如果下限为 +0.0，则 -0.0 将不匹配。

我应该使用哪种类型?

就整数类型（字节、短整数和长整数）而言，您应该选择足以满足您的用例的最小类型。这将有助于索引和搜索更有效。但是请注意，存储是根据存储的实际值进行优化的，因此选择一种类型而不是另一种类型不会对存储要求产生影响。

对于浮点类型，使用比例因子将浮点数据存储到整数中通常更有效，这就是 scaled_float 类型在幕后所做的。例如，一个 price 字段可以存储在 scaled_float 中， scaling_factor 为 100。所有 API 都可以像该字段存储为 double 一样工作，但在底层 Elasticsearch 将使用美分数量，price*100，这是一个整数。这对节省磁盘空间很有帮助，因为整数比浮点更容易压缩。 scaled_float 也可以用来交换磁盘空间的准确性。例如，假设您将 cpu 利用率跟踪为 0 和 1 之间的数字。cpu 利用率是 12.7% 还是 13% 通常并不重要，因此您可以使用 scaling_factor 为 100 的 scaled_float 来舍入 cpu 利用率到最接近的百分比以节省空间。

如果 scaled_float 不合适，那么您应该在浮点类型中选择足以满足用例的最小类型：double、float 和 half_float。这是一个比较这些类型的表格，以帮助做出决定。

类型	最小值	最大值	有效位/位数
`double`	`2^-1074`	`(2-2^-52)·2^1023`	`53` / `15.95`
`float`	`2^-149`	`(2-2^-23)·2^127`	`24` / `7.22`
`half_float`	`2^-24`	`65504`	`11` / `3.31`

数值字段的参数

数字类型接受以下参数：

参数	描述
`coerce`	尝试将字符串转换为数字并截断整数的分数。接受`true`（默认）和`false`。
`boost`	映射字段级查询时间提升。接受浮点数，默认为 1.0。
`doc_values`	该字段是否应该以列跨步的方式存储在磁盘上，以便以后可以用于排序、聚合或脚本？ 接受`true`（默认）和`false`。

参数	描述
ignore_malformed	如果为 true，格式错误的数字将被忽略。如果为 false（默认），则格式错误的数字会引发异常并拒绝整个文档。
index	该字段应该是可搜索的吗？接受 true（默认）和 false。
null_value	接受与替换任何显式空值的字段类型相同的数值。默认为 null，这意味着该字段被视为缺失。
store	字段值是否应与 _source 字段分开存储和检索。接受真或假（默认）。接受 true（和 false（默认）。
meta	关于字段的元数据。

scaled_float 的参数

scaled_float 接受一个附加参数：

scaling_factor

编码值时使用的比例因子。值将在索引时乘以该因子并四舍五入为最接近的长值。例如，缩放因子为 10 的 scaled_float 将在内部将 2.34 存储为 23，并且所有搜索时间操作（查询、聚合、排序）的行为就像文档的值为 2.3。 scaling_factor 的高值提高了准确性，但也增加了空间需求。此参数是必需的。

其他类型

其他类型请参考官方文档自行学习，在类型总览里面有对应的链接。

4.3 元字段

每个文档都有与之关联的元数据，例如 _index、映射 _type 和 _id 元字段。创建映射类型时，可以自定义其中一些元字段的行为。

标识元字段

字段名	描述
_index	文档所属的索引。
_type	文档的映射类型。
_id	文档的 ID。

文档源元字段

- _source
  - 表示文档正文的原始 JSON。
- _size
  - _source 字段的大小（以字节为单位），由 mapper-size 插件提供。

索引元字段

- _field_names
  - 文档中包含非空值的所有字段。

- `_ignored`
  - 由于`ignore_malformed`而在索引时被忽略的文档中的所有字段。

路由元字段

- `_routing`
  - 将文档路由到特定分片的自定义路由值。

其他元字段

- `_meta`
  - 应用程序特定的元数据。

`_field_names`

`_field_names`字段用于索引文档中包含除null之外的任何值的每个字段的名称。在使用`exists`查询时，使用此字段来查找具有或不具有特定字段的任何非空值的文档。

现在`_field_names`字段仅索引禁用了`doc_values`和`norms`的字段的名称。对于启用了`doc_values`或`norms`的字段，`exists`查询仍然可用，但不会使用`_field_names`字段。

`_ignored`

_ignored 字段索引并存储文档中每个字段的名称，这些字段因格式错误而被忽略，并且`ignore_malformed`已打开。

此字段可使用`term`、`terms`和`exists`查询进行搜索，并作为搜索命中的一部分返回。

例如，以下查询匹配具有一个或多个被忽略的字段的所有文档：

```
GET _search
{
  "query": {
    "exists": {
      "field": "_ignored"
    }
  }
}
```

类似地，以下查询查找在索引时忽略 @timestamp 字段的所有文档：

```
GET _search
{
  "query": {
    "term": {
      "_ignored": "@timestamp"
    }
  }
}
```

`_id`

每文档都有一个唯一标识它的 `_id`，它被编入索引，以便可以使用 GET API 或 ids 查询来查找文档。

_id 字段的值可在某些查询（term、terms、match、query_string、simple_query_string）中访问。

例如

```
PUT my_index/_doc/1
{
  "text": "Document with ID 1"
}

PUT my_index/_doc/2?refresh=true
{
  "text": "Document with ID 2"
}

GET my_index/_search
{
  "query": {
    "terms": {
      "_id": [ "1", "2" ]
    }
  }
}
```

_id 字段的值也可以在聚合或排序中访问，但不鼓励这样做，因为它需要在内存中加载大量数据。如果需要对 _id 字段进行排序或聚合，建议将 _id 字段的内容复制到另一个启用了 doc_values 的字段中。

_id 的大小限制为 512 字节，较大的值将被拒绝。

`_type`

在 6.0.0 中已弃用。查看 *Removal of mapping types*

每个被索引的文档都与一个_type和一 _id相关联。 _type字段已编入索引，以便快速按类型名称进行搜索。

_type字段的值可在查询、聚合、脚本和排序时访问：

```
PUT my_index/_doc/1?refresh=true
{
  "text": "Document with type 'doc'"
}

GET my_index/_search
{
  "query": {
    "term": {
      "_type": "_doc"
    }
  },
  "aggs": {
    "types": {
      "terms": {
```

```
        "field": "_type",
        "size": 10
      }
    }
  },
  "sort": [
    {
      "_type": {
        "order": "desc"
      }
    }
  ],
  "script_fields": {
    "type": {
      "script": {
        "lang": "painless",
        "source": "doc['_type']"
      }
    }
  }
}
```

_index

在跨多个索引执行查询时，有时需要添加与仅某些索引的文档相关联的查询子句。 _index 字段允许匹配文档被索引到的索引。它的值可在某些查询和聚合中以及在排序或编写脚本时访问：

```
PUT index_1/_doc/1
{
  "text": "Document in index 1"
}

PUT index_2/_doc/2?refresh=true
{
  "text": "Document in index 2"
}

GET index_1,index_2/_search
{
  "query": {
    "terms": {
      "_index": ["index_1", "index_2"]
    }
  },
  "aggs": {
    "indices": {
      "terms": {
        "field": "_index",
        "size": 10
      }
    }
  },
  "sort": [
    {
      "_index": {
```

```
        "order": "asc"
      }
    }
  ],
  "script_fields": {
    "index_name": {
      "script": {
        "lang": "painless",
        "source": "doc['_index']"
      }
    }
  }
}
```

_index 字段实际上是公开的 — 它没有作为真实字段添加到 Lucene 索引中。这意味着您可以在一个或多个词语查询（或任何重写为词语查询的查询，例如 match、query_string 或 simple_query_string 查询）以及前缀和通配符查询中使用 _index 字段。但是，它不支持正则表达式和模糊查询。

除了具体的索引名称之外，对 _index 字段的查询还接受索引别名。

其他元字段

其他元字段请参考官方文档。

4.4 映射参数

映射参数对某些或所有字段数据类型是通用的：

- `analyzer`
- `boost`
- `coerce`
- `copy_to`
- `doc_values`
- `dynamic`
- `eager_global_ordinals`
- `enabled`
- `fielddata`
- `fields`
- `format`
- `ignore_above`
- `ignore_malformed`
- `index_options`
- `index_phrases`
- `index_prefixes`
- `index`
- `meta`
- `normalizer`
- `norms`
- `null_value`
- `position_increment_gap`
- `properties`
- `search_analyzer`

- `similarity`
- `store`
- `term_vector`

`analyzer`

> 注意：只有`text`字段支持分析器映射参数。

analyzer参数指定在索引或搜索文本字段时用于文本分析的分析器。

除非使用 `search_analyzer` 映射参数覆盖，否则此分析器用于索引和搜索分析。请参阅指定分析器。

我们建议在生产中使用分析器之前首先对其进行测试。请参阅测试分析器。

`search_quote_analyzer`

`search_quote_analyzer` 设置允许您为短语指定分析器，这在处理禁用短语查询的停用词时特别有用。

要禁用短语的停用词，需要使用三个分析器设置的字段：

1. 用于索引所有词语（包括停用词）的 `analyzer` 设置
2. 将删除停用词的非短语查询的 `search_analyzer` 设置
3. 不会删除停用词的短语查询的 `search_quote_analyzer` 设置

```
PUT my_index
{
   "settings":{
      "analysis":{
         "analyzer":{
            "my_analyzer":{
               "type":"custom",
               "tokenizer":"standard",
               "filter":[
                  "lowercase"
               ]
            },
            "my_stop_analyzer":{
               "type":"custom",
               "tokenizer":"standard",
               "filter":[
                  "lowercase",
                  "english_stop"
               ]
            }
         },
         "filter":{
            "english_stop":{
               "type":"stop",
               "stopwords":"_english_"
            }
         }
      }
   },
   "mappings":{
```

```
      "properties":{
         "title": {
            "type":"text",
            "analyzer":"my_analyzer",
            "search_analyzer":"my_stop_analyzer",
            "search_quote_analyzer":"my_analyzer"
         }
      }
   }
}

PUT my_index/_doc/1
{
   "title":"The Quick Brown Fox"
}

PUT my_index/_doc/2
{
   "title":"A Quick Brown Fox"
}

GET my_index/_search
{
   "query":{
      "query_string":{
         "query":"\"the quick brown fox\""
      }
   }
}
```

- my_analyzer ： 它标记所有词语，包括停用词
- my_stop_analyzer ： 删除停用词
- analyzer设置：指向将在索引时使用的my_analyzer分析器
- search_analyzer设置：指向my_stop_analyzer并删除非短语查询的停用词
- search_quote_analyzer设置：指向my_analyzer分析器，并确保不从短语查询中删除停用词
- 由于查询包含在引号中，因此它被检测为短语查询，因此search_quote_analyzer启动并确保不从查询中删除停用词。然后my_analyzer分析器将返回以下标记[the, quick, brown, fox] 将匹配其中一个文档。同时，词语查询将使用my_stop_analyzer分析器进行分析，该分析器将过滤掉停用词。因此，搜索`The quick brown fox`或`A quick brown fox`将返回两个文档，因为两个文档都包含以下标记 [quick, brown, fox]。如果没有 search_quote_analyzer，就不可能对短语查询进行精确匹配，因为短语查询中的停用词将被删除，从而导致两个文档都匹配。

`boost`

个别字段可以自动提升 — 计入相关性分数 — 在查询时，提升参数如下：

```
PUT my_index
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "boost": 2
      },
```

```
      "content": {
        "type": "text"
      }
    }
  }
}
```

`title`字段上的匹配项的权重是`content`字段上的两倍，其默认提升为1.0。

提升仅适用于词语查询（不提升前缀、范围和模糊查询）。

您可以通过直接在查询中使用 boost 参数来达到相同的效果，例如以下查询（使用字段时间提升）：

```
POST _search
{
    "query": {
        "match" : {
            "title": {
                "query": "quick brown fox",
                "boost": 2
            }
        }
    }
}
```

> 在 5.0.0 中已弃用。
> 不推荐使用索引时间提升。相反，字段映射提升是在查询时应用的。对于 5.0.0 之前创建的索引，提升仍将在索引时应用。
> 为什么索引时间提升是一个坏主意
> 我们建议不要使用索引时间提升，原因如下：
>
> - 如果不重新索引所有文档，就无法更改索引时间提升值。
> - 每个查询都支持查询时间提升，从而达到相同的效果。不同之处在于您可以调整提升值而无需重新索引。
> - 索引时间提升作为标准的一部分存储，只有一个字节。这会降低字段长度归一化因子的分辨率，从而导致相关性计算质量降低。

`fields`

出于不同目的以不同方式索引同一字段通常很有用。这就是多字段的目的。例如，字符串字段可以映射为全文搜索的文本字段，以及排序或聚合的关键字字段：

```
PUT my_index
{
  "mappings": {
    "properties": {
      "city": {
        "type": "text",
        "fields": {
          "raw": {
            "type":  "keyword"
          }
        }
```

```
      }
    }
  }
}

PUT my_index/_doc/1
{
  "city": "New York"
}

PUT my_index/_doc/2
{
  "city": "York"
}

GET my_index/_search
{
  "query": {
    "match": {
      "city": "york"
    }
  },
  "sort": {
    "city.raw": "asc"
  },
  "aggs": {
    "Cities": {
      "terms": {
        "field": "city.raw"
      }
    }
  }
}
```

- city.raw 字段是 city 字段的关键字版本。
- city字段可用于全文搜索。
- city.raw 字段可用于排序和聚合

多字段的另一个用例是以不同方式分析同一字段以获得更好的相关性。例如，我们可以使用将文本分解为单词的标准分析器来索引一个字段，并再次使用将单词词干成词根形式的英语分析器来索引：

```
PUT my_index
{
  "mappings": {
    "properties": {
      "text": {
        "type": "text",
        "fields": {
          "english": {
            "type":     "text",
            "analyzer": "english"
          }
        }
      }
    }
```

```
  }
}

PUT my_index/_doc/1
{ "text": "quick brown fox" }

PUT my_index/_doc/2
{ "text": "quick brown foxes" }

GET my_index/_search
{
  "query": {
    "multi_match": {
      "query": "quick brown foxes",
      "fields": [
        "text",
        "text.english"
      ],
      "type": "most_fields"
    }
  }
}
```

- text字段使用标准分析器。
- text.english 字段使用英语分析器。
- 索引两份文档，一份使用 fox，另一份使用 foxes。
- 查询 text 和 text.english 字段并合并分数。

text字段在第一个文档中包含词语 fox，在第二个文档中包含 foxes。 text.english 字段包含两个文档的 fox，因为 foxes 的词干是 fox。

text字段的标准分析器和 text.english 字段的英语分析器也分析查询字符串。词干字段允许对 foxes 的查询也匹配仅包含 fox 的文档。这使我们能够匹配尽可能多的文档。通过查询未提取的文本字段，我们提高了与 foxes 完全匹配的文档的相关性分数。

`doc_values`

默认情况下，大多数字段都已编入索引，这使得它们可搜索。倒排索引允许查询在唯一排序的词语列表中查找搜索词语，并从中立即访问包含该词语的文档列表。

排序、聚合和访问脚本中的字段值需要不同的数据访问模式。我们需要能够查找文档并找到它在字段中的词语，而不是查找词语和查找文档。

文档值是在文档索引时构建的磁盘数据结构，这使得这种数据访问模式成为可能。它们存储与 `_source` 相同的值，但以面向列的方式存储，这对于排序和聚合更有效。几乎所有字段类型都支持 doc 值，但 text 和 annotated_text 字段除外。

所有支持 doc 值的字段都默认启用它们。如果您确定不需要对字段进行排序或聚合，或从脚本访问字段值，则可以禁用 doc 值以节省磁盘空间：

```
PUT my_index
{
  "mappings": {
    "properties": {
      "status_code": {
```

```
        "type": "keyword"
      },
      "session_id": {
        "type": "keyword",
        "doc_values": false
      }
    }
  }
}
```

status_code 字段默认启用 doc_values。
session_id 禁用了 doc_values，但仍然可以查询。

`fielddata`

默认情况下，大多数字段都已编入索引，这使得它们可搜索。然而，在脚本中排序、聚合和访问字段值需要与搜索不同的访问模式。

搜索需要回答“哪些文档包含这个词？”这个问题，而排序和聚合需要回答一个不同的问题：“这个文档的这个字段的值是什么？”。

对于这种数据访问模式，大多数字段可以使用索引时间、磁盘上的 `doc_values`，但文本字段不支持 `doc_values`。

相反，文本字段使用称为 `fielddata` 的查询时内存数据结构。此数据结构是在第一次将字段用于聚合、排序或在脚本中时按需构建的。它是通过从磁盘读取每个段的整个倒排索引、反转词语 ↔ 文档关系并将结果存储在内存中的 JVM 堆中来构建的。

默认情况下，在文本字段上禁用了 `fielddata`
fielddata会消耗大量堆空间，尤其是在加载高基数文本字段时。一旦 fielddata 被加载到堆中，它就会在段的生命周期内保持在那里。此外，加载字段数据是一个昂贵的过程，可能会导致用户遇到延迟问题。这就是为什么默认禁用 fielddata 的原因。
如果您尝试对文本字段上的脚本进行排序、聚合或访问值，您将看到以下异常：

```
Fielddata is disabled on text fields by default.  Set `fielddata=true` on
[`your_field_name`] in order to load  fielddata in memory by uninverting the
inverted index. Note that this can however use significant memory.
```

在启用 fielddata 之前，请考虑为什么要使用文本字段进行聚合、排序或在脚本中。这样做通常没有意义。

在索引之前分析文本字段，以便可以通过搜索 new 或 york 找到像 New York 这样的值。当您可能需要一个名为 New York 的存储桶时，此字段上的术语聚合将返回一个新存储桶和一个 york 存储桶。

相反，您应该有一个用于全文搜索的文本字段，以及一个为聚合启用 doc_values 的未分析关键字字段，如下所示：

```
PUT my_index
{
  "mappings": {
    "properties": {
      "my_field": {
        "type": "text",
        "fields": {
          "keyword": {
```

```
            "type": "keyword"
          }
        }
      }
    }
  }
}
```

使用 my_field 字段进行搜索。
使用 my_field.keyword 字段进行聚合、排序或在脚本中。

您可以使用添加映射API在现有文本字段上启用 fielddata，如下所示：

```
PUT my_index/_mapping
{
  "properties": {
    "my_field": {
      "type":     "text",
      "fielddata": true
    }
  }
}
```

您为 my_field 指定的映射应包含该字段的现有映射以及 fielddata 参数。

`fielddata_frequency_filter`

字段数据过滤可用于减少加载到内存中的词语数量，从而减少内存使用量。可以按频率过滤词语：

频率过滤器允许您仅加载文档频率介于`min`和`max`之间的词语，可以表示为绝对数字（当数字大于 1.0 时）或百分比（例如 0.01 是 1%，1.0 是 100 %）。频率按段计算。百分比基于具有字段值的文档数量，而不是细分中的所有文档。

通过使用`min_segment_size`指定段应包含的最小文档数，可以完全排除小段：

```
PUT my_index
{
  "mappings": {
    "properties": {
      "tag": {
        "type": "text",
        "fielddata": true,
        "fielddata_frequency_filter": {
          "min": 0.001,
          "max": 0.1,
          "min_segment_size": 500
        }
      }
    }
  }
}
```

其他映射参数

请参考官方文档查看。

4.5 动态映射

Elasticsearch 最重要的功能之一是它试图让您摆脱困境，让您尽快开始探索您的数据。要为文档编制索引，您不必首先创建索引、定义映射类型和定义字段 — 您只需索引文档，索引、类型和字段就会自动生效：

```
PUT data/_doc/1
{ "count": 5 }
```

创建data索引、_doc映射类型和一个名为count的字段，数据类型为long。

自动检测和添加新字段称为动态映射。可以通过以下方式自定义动态映射规则以适合您的目的：

- **动态字段映射**
  - 管理动态字段检测的规则。
- **动态模板**
  - 为动态添加的字段配置映射的自定义规则。

索引模板允许您为新索引配置默认映射、设置和别名，无论是自动创建还是显式创建。

动态字段映射

默认情况下，当在文档中找到以前未见过的字段时，Elasticsearch会将新字段添加到类型映射中。通过将 `dynamic` 参数设置为 `false`（忽略新字段）或 `strict`（如果遇到未知字段则抛出异常），可以在文档和对象级别禁用此行为。

假设启用了动态字段映射，则使用一些简单的规则来确定该字段应具有的数据类型：

JSON数据类型	Elasticsearch数据类型
`null`	没有添加任何字段。
`true` 或 `false`	`boolean`
浮点型数字	`float`
整形数字	`long`
对象	`object`
数组	取决于数组中的第一个非空值。
字符串	日期字段（如果值通过日期检测）、双精度或长字段（如果值通过数字检测）或带有关键字子字段的文本字段。

这些是唯一动态检测的字段数据类型。所有其他数据类型必须显式映射。
除了下面列出的选项之外，动态字段映射规则还可以使用 `dynamic_templates` 进一步自定义。

日期检测

如果启用 date_detection（默认），则检查新字符串字段以查看其内容是否与 dynamic_date_formats 中指定的任何日期模式匹配。如果找到匹配项，则会添加一个具有相应格式的新日期字段。

dynamic_date_formats 的默认值为：

`[ "strict_date_optional_time","yyyy/MM/dd HH:mm:ss Z||yyyy/MM/dd Z"]`

例如：

```
PUT my_index/_doc/1
{
  "create_date": "2015/09/02"
}

GET my_index/_mapping
```

禁用日期检测

可以通过将 date_detection 设置为 false 来禁用动态日期检测：

```
PUT my_index
{
  "mappings": {
    "date_detection": false
  }
}

PUT my_index/_doc/1
{
  "create": "2015/09/02"
}
```

自定义检测到的日期格式

或者，可以自定义 `dynamic_date_formats` 以支持您自己的日期格式：

```
PUT my_index
{
  "mappings": {
    "dynamic_date_formats": ["MM/dd/yyyy"]
  }
}

PUT my_index/_doc/1
{
  "create_date": "09/25/2015"
}
```

数值检测

虽然 JSON 支持本机浮点和整数数据类型，但某些应用程序或语言有时可能会将数字呈现为字符串。通常正确的解决方案是显式映射这些字段，但可以启用数字检测（默认情况下禁用）以自动执行此操作：

```
PUT my_index
{
  "mappings": {
    "numeric_detection": true
  }
}

PUT my_index/_doc/1
{
  "my_float":   "1.0",
  "my_integer": "1"
}
```

- my_float 字段作为浮点字段添加。
- my_integer 字段作为长字段添加。

动态模板

动态模板允许您定义可应用于基于以下动态添加的字段的自定义映射：

- Elasticsearch 检测到的数据类型，带有 match_mapping_type。
- 字段的名称，包括 match 和 unmatch 或 match_pattern。
- 字段的完整虚线路径，带有 path_match 和 path_unmatch。

原始字段名称 {name} 和检测到的数据类型 {dynamic_type} 模板变量可以在映射规范中用作占位符。

> 只有当字段包含具体值 — 不是 null 或空数组时，才会添加动态字段映射。这意味着如果在 dynamic_template 中使用 null_value 选项，它只会在第一个具有字段具体值的文档被索引后应用。

动态模板被指定为命名对象的数组：

```
"dynamic_templates": [
  {
    "my_template_name": {
      ...  match conditions ...
      "mapping": { ... }
    }
  },
  ...
]
```

- `my_template_name` 可以是任何字符串值。
- `match conditions` 可以包括以下任何一种：
  - match_mapping_type、match、match_pattern、unmatch、path_match、path_unmatch。
- `"mapping": { ... }` 匹配字段应使用的映射。

模板按顺序处理 — 第一个匹配的模板获胜。当通过 put 映射 API 放置新的动态模板时，所有现有模板都会被覆盖。这允许在最初添加动态模板后重新排序或删除它们。

匹配映射类型

match_mapping_type是 json 解析器检测到的数据类型。由于 JSON 不允许区分long与integer或double与float，因此它将始终选择更广泛的数据类型，即 long 用于整数，double 用于浮点数。

可以自动检测以下数据类型：

- 遇到 true 或 false 时的布尔值。
- 启用日期检测并找到与任何配置的日期格式匹配的字符串时的日期。
- double 用于带小数部分的数字。
- long 表示没有小数部分的数字。
- object表示对象，也称为哈希。
- string 用于字符串。

也可以用于匹配所有数据类型。

例如，如果我们想将所有整数字段映射为整数而不是长整数，并将所有字符串字段同时映射为文本和关键字，我们可以使用以下模板：

```
PUT my_index
{
  "mappings": {
    "dynamic_templates": [
      {
        "longs_as_strings": {
          "match_mapping_type": "string",
          "match":   "long_*",
          "unmatch": "*_text",
          "mapping": {
            "type": "long"
          }
        }
      }
    ]
  }
}

PUT my_index/_doc/1
{
  "long_num": "5",
  "long_text": "foo"
}
```

- long_num 字段映射为 long。
- long_text 字段使用默认字符串映射。

匹配模式

match_pattern 参数调整 match 参数的行为，使其支持字段名称上的完整 Java 正则表达式匹配，而不是简单的通配符，例如：

```
  "match_pattern": "regex",
  "match": "^profit_\d+$"
```

path_match & path_unmatch

path_match 和 path_unmatch 参数的工作方式与 match 和 unmatch 相同，但对字段的完整虚线路径进行操作，而不仅仅是最终名称，例如some_object.*.some_field。

此示例将 name 对象中任何字段的值复制到顶级 full_name 字段，middle字段除外：

```
PUT my_index
{
  "mappings": {
    "dynamic_templates": [
      {
        "full_name": {
          "path_match":   "name.*",
          "path_unmatch": "*.middle",
          "mapping": {
            "type":  "text",
            "copy_to":  "full_name"
          }
        }
      }
    ]
  }
}

PUT my_index/_doc/1
{
  "name": {
    "first":  "John",
    "middle": "Winston",
    "last":   "Lennon"
  }
}
```

请注意，除了叶字段之外，path_match 和 path_unmatch 参数还匹配对象路径。例如，索引以下文档将导致错误，因为 path_match 设置也匹配对象字段 name.title，它不能映射为文本：

```
PUT my_index/_doc/2
{
  "name": {
    "first":  "Paul",
    "last":   "McCartney",
    "title": {
      "value": "Sir",
      "category": "order of chivalry"
    }
  }
}
```

{name} 和 {dynamic_type}

{name} 和 {dynamic_type} 占位符在映射中被替换为字段名称和检测到的动态类型。以下示例将所有字符串字段设置为使用与字段同名的分析器，并禁用所有非字符串字段的doc_values：

```
PUT my_index
{
  "mappings": {
    "dynamic_templates": [
      {
        "named_analyzers": {
          "match_mapping_type": "string",
          "match": "*",
          "mapping": {
            "type": "text",
            "analyzer": "{name}"
          }
        }
      },
      {
        "no_doc_values": {
          "match_mapping_type":"*",
          "mapping": {
            "type": "{dynamic_type}",
            "doc_values": false
          }
        }
      }
    ]
  }
}

PUT my_index/_doc/1
{
  "english": "Some English text",
  "count":   5
}
```

- 使用英语分析器将 `english` 字段映射为字符串字段。
- `count` 字段映射为禁用 doc_values 的 `long` 字段。

模板示例

以下是一些可能有用的动态模板的示例：

结构化搜索

默认情况下，Elasticsearch 会将字符串字段映射为带有子关键字字段的文本字段。但是，如果您只是索引结构化内容并且对全文搜索不感兴趣，则可以使 Elasticsearch 仅将您的字段映射为“关键字”。请注意，这意味着为了搜索这些字段，您必须搜索与索引完全相同的值。

```
PUT my_index
{
  "mappings": {
    "dynamic_templates": [
```

```
      {
        "strings_as_keywords": {
          "match_mapping_type": "string",
          "mapping": {
            "type": "keyword"
          }
        }
      }
    ]
  }
}
```

字符串的纯文本映射

与前面的示例相反，如果您对字符串字段唯一关心的是全文搜索，并且如果您不打算在字符串字段上运行聚合、排序或精确搜索，您可以告诉 Elasticsearch仅将其映射为文本字段（这是 5.0 之前的默认行为）：

```
PUT my_index
{
  "mappings": {
    "dynamic_templates": [
      {
        "strings_as_text": {
          "match_mapping_type": "string",
          "mapping": {
            "type": "text"
          }
        }
      }
    ]
  }
}
```

禁用norms

`norms`是索引时间的评分因素。如果您不关心评分，例如，如果您从不按分数对文档进行排序，您可以禁用这些评分因素在索引中的存储并节省一些空间。

```
PUT my_index
{
  "mappings": {
    "dynamic_templates": [
      {
        "strings_as_keywords": {
          "match_mapping_type": "string",
          "mapping": {
            "type": "text",
            "norms": false,
            "fields": {
              "keyword": {
                "type": "keyword",
                "ignore_above": 256
              }
            }
```

```
          }
        }
      }
    ]
  }
}
```

该模板中出现的子关键字字段与动态映射的默认规则保持一致。当然，如果您不需要它们，因为您不需要对该字段执行精确搜索或聚合，您可以按照上一节中的说明将其删除。

时间序列

使用 Elasticsearch 进行时间序列分析时，通常会有许多数字字段，您经常会在这些字段上进行聚合，但从不进行过滤。在这种情况下，您可以禁用这些字段的索引以节省磁盘空间并可能获得一些索引速度：

```
PUT my_index
{
  "mappings": {
    "dynamic_templates": [
      {
        "unindexed_longs": {
          "match_mapping_type": "long",
          "mapping": {
            "type": "long",
            "index": false
          }
        }
      },
      {
        "unindexed_doubles": {
          "match_mapping_type": "double",
          "mapping": {
            "type": "float",
            "index": false
          }
        }
      }
    ]
  }
}
```

与默认的动态映射规则一样，双精度数映射为浮点数，通常足够准确，但需要一半的磁盘空间。

5 搜索

5.1 搜索API

大多数搜索API都是多索引的，除了解释API端点。

执行搜索时，Elasticsearch 将根据自适应副本选择公式选择数据的“最佳”副本。也可以通过提供路由参数来控制将搜索哪些分片。例如，索引推文时，路由值可以是用户名：

```
POST /twitter/_doc?routing=kimchy
{
    "user" : "kimchy",
    "post_date" : "2009-11-15T14:12:12",
    "message" : "trying out Elasticsearch"
}
```

在这种情况下，如果我们只想搜索特定用户的推文，我们可以将其指定为路由，从而使搜索只命中相关分片：

```
POST /twitter/_search?routing=kimchy
{
    "query": {
        "bool" : {
            "must" : {
                "query_string" : {
                    "query" : "some query string here"
                }
            },
            "filter" : {
                "term" : { "user" : "kimchy" }
            }
        }
    }
}
```

路由参数可以多值表示为逗号分隔的字符串。这将导致命中路由值匹配的相关分片。

默认情况下，Elasticsearch 将使用所谓的自适应副本选择。这允许协调节点根据一些标准将请求发送到被认为是“最佳”的副本：

- 协调节点与包含数据副本的节点之间过去请求的响应时间
- 过去的搜索请求在包含数据的节点上执行的时间
- 包含数据的节点上搜索线程池的队列大小

这可以通过将动态集群设置 `cluster.routing.use_adaptive_replica_selection` 从 `true` 更改为 `false` 来关闭：

```
PUT /_cluster/settings
{
    "transient": {
        "cluster.routing.use_adaptive_replica_selection": false
    }
}
```

如果关闭了自适应副本选择，则在所有数据副本（主副本和副本）之间以循环方式将搜索发送到索引/索引分片。

搜索可以与统计组相关联，统计组维护每个组的统计聚合。稍后可以专门使用索引统计API检索它。例如，这是一个搜索正文请求，它将请求与两个不同的组相关联：

```
POST /_search
{
    "query" : {
        "match_all" : {}
    },
    "stats" : ["group1", "group2"]
}
```

搜索

返回与请求中定义的查询匹配的搜索命中。

请求方式

```
GET /<index>/_search
GET /_search
POST /<index>/_search
POST /_search
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。

查询参数

- `allow_no_indices`
  - （可选，布尔值）如果为 true，则如果通配符表达式或 `_all` 值仅检索缺失或关闭的索引，则请求不会返回错误。
  - 此参数也适用于指向缺失或关闭索引的索引别名。
  - 默认为true。
- `allow_partial_search_results`
  - （可选，布尔值）如果为 true，则在请求超时或分片失败时返回部分结果。如果为 false，则返回没有部分结果的错误。默认为true。
  - 要覆盖此字段的默认值，请将 `search.default_allow_partial_results` 集群设置设置为 false。
- `batched_reduce_size`
  - （可选，整数）协调节点上一次应该减少的分片结果数。如果请求中的潜在分片数量可能很大，则应将此值用作保护机制以减少每个搜索请求的内存开销。默认为 `512`。
- `ccs_minimize_roundtrips`
  - （可选，布尔值）如果为 true，则在执行跨集群搜索 (CCS) 请求时，协调节点和远程集群之间的网络往返行程将被最小化。请参阅跨集群搜索如何处理网络延迟。
  - 默认为true。
- `docvalue_fields`
  - （可选，字符串）以逗号分隔的字段列表，作为每个命中的字段的 docvalue 表示形式返回。
- `expand_wildcards`
  - （可选，字符串）控制通配符表达式可以扩展的索引类型。有效值为：
    - `all`：扩展到开放和关闭的索引。
    - `open`：仅扩展至开放索引。

        - `closed`：仅扩展到关闭索引。
        - `none`：不接受通配符表达式。
    - 默认值为 `open`。
- `explain`
    - （可选，布尔值）如果为true，则返回有关分数计算的详细信息作为命中的一部分。默认为false。
- `from`
    - （可选，整数）起始文档偏移量。默认为 0。
- `ignore_throttled`
    - （可选，布尔值）如果为 true，则在限制时将忽略具体、扩展或别名索引。默认为false。
- `ignore_unavailable`
    - （可选，布尔值）如果为 true，则响应中不包含缺失或关闭的索引。默认为false。
- `max_concurrent_shard_requests`
    - （可选，整数）定义此搜索同时执行的每个节点的并发分片请求数。该值应该用于限制搜索对集群的影响，以限制并发分片请求的数量。默认为 5。
- `pre_filter_shard_size`
    - （可选，整数）定义一个阈值，如果搜索请求扩展到的分片数量超过阈值，则该阈值会根据查询重写来强制执行预过滤器往返以预过滤搜索分片。此过滤器往返可以显着限制分片的数量，例如，如果分片无法匹配基于其重写方法的任何文档，即。如果必须匹配日期过滤器，但分片边界和查询是不相交的。未指定时，如果满足以下任何条件，则执行预过滤阶段：
        - 该请求针对超过 128 个分片。
        - 该请求针对一个或多个只读索引。
        - 查询的主要排序以索引字段为目标。
- `preference`
    - （可选，字符串）指定应在其上执行操作的节点或分片。默认随机。
- `q`
    - （可选，字符串）以 Lucene 查询字符串语法进行查询。
    - 您可以使用 q 参数运行查询参数搜索。查询参数搜索不支持完整的 Elasticsearch Query DSL，但便于测试。
    - q 参数覆盖请求正文中的查询参数。如果同时指定了这两个参数，则不返回与查询请求正文参数匹配的文档。
- `request_cache`
    - （可选，布尔值）如果为 true，则为大小为 0 的请求启用搜索结果缓存。请参阅分片请求缓存。默认为索引级别设置。
- `rest_total_hits_as_int`
    - （可选，布尔值）指示 hits.total 是否应在剩余搜索响应中呈现为整数或对象。默认为false。
- `routing`
    - （可选，时间单位）指定应为滚动搜索维护索引的一致视图多长时间。
- `search_type`
    - （可选，字符串）搜索操作的类型。可用选项：
        - query_then_fetch
        - dfs_query_then_fetch
- `seq_no_primary_term`
    - （可选，布尔值）如果为true，则返回每个命中的最后修改的序列号和`_primary_term`。

- `size`
  - （可选，整数）定义要返回的命中数。默认为 10。
  - 您还可以使用 size 请求正文参数指定此值。如果同时指定了两个参数，则仅使用查询参数。
- `sort`
  - （可选，字符串）以逗号分隔的 `<field>:<direction>` 对列表。
- `_source`
  - （可选，布尔值）如果为 true，则响应包括匹配文档的 `_source` 匹配项。默认为true。
- `_source_excludes`
  - （可选，字符串）要从返回的 `_source` 字段中排除的字段列表。
- `_source_includes`
  - （可选，字符串）要从返回的 `_source` 字段中提取和返回的字段列表。
- `stats`
  - （可选，字符串）用于记录和统计目的的请求的特定`tag`。
- `stored_fields`
  - （可选，字符串）以逗号分隔的存储字段列表，作为命中的一部分返回。如果未指定任何字段，则响应中不包含存储的字段。
- `suggest_field`
  - （可选，字符串）指定用于建议的字段。
- `suggest_text`
  - （可选，字符串）应为其返回建议的源文本。
- `terminate_after`
  - （可选，整数）为每个分片收集的最大文档数，达到该数时查询执行将提前终止。
  - 默认为 0，它不会提前终止查询执行。
- `timeout`
  - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为无超时。
- `version`
  - （可选，布尔值）如果为true，则返回文档版本作为命中的一部分。默认为false。
- `track_scores`
  - （可选，布尔值）如果为 true，则计算并返回文档分数，即使分数不用于排序。默认为 false。
- `track_total_hits`
  - （可选，整数或布尔值）匹配查询以准确计数的命中数。默认为 10000。如果为true，则使用默认值。如果为 false，则响应不包括与查询匹配的总命中数。
- `typed_keys`
  - （可选，布尔值）如果为 true，则聚合和建议者名称将在响应中以它们各自的类型作为前缀。默认为true。

请求主体

> 提示：请求主体中的参数如果查询参数也支持那么优先级最高的查询参数

- `explain`
  - （可选，布尔值）如果为true，则返回有关分数计算的详细信息作为命中的一部分。默认为false。

- `from`
  - （可选，整数）起始文档偏移量。默认为 0。
- `query`
  - （可选，查询对象）使用查询DSL定义搜索定义。
- `seq_no_primary_term`
  - （可选，布尔值）如果为true，则返回每个命中的最后修改的序列号和`_primary_term`。
- `size`
  - （可选，整数）定义要返回的命中数。默认为 10。
- `terminate_after`
  - （可选，整数）为每个分片收集的最大文档数，达到该数时查询执行将提前终止。
  - 默认为 0，它不会提前终止查询执行。
- `timeout`
  - （可选，时间单位）指定等待响应的时间段。如果在超时到期之前没有收到响应，则请求失败并返回错误。默认为无超时。
- `version`
  - （可选，布尔值）如果为true，则返回文档版本作为命中的一部分。默认为false。

响应主体

- `took`
  - （整数）Elasticsearch 执行请求所用的毫秒数。
  - 该值是通过测量在协调节点上接收到请求与协调节点准备好发送响应的时间之间经过的时间来计算的。
  - 花费时间包括：
    - 协调节点和数据节点之间的通信时间
    - 请求在搜索线程池中花费的时间，排队等待执行
    - 实际执行时间
  - 花费时间不包括：
    - 将请求发送到 Elasticsearch 所需的时间
    - 序列化 JSON 响应所需的时间
    - 将响应发送给客户端所需的时间
- `timed_out`
  - (boolean) 如果为true，则请求在完成前超时；返回的结果可能是部分的或空的。
- `_shards`
  - (object) 包含用于请求的分片计数。
  - `_shards` 的属性：
    - `total`
      - （整数）需要查询的分片总数，包括未分配的分片。
    - `successful`
      - （整数）成功执行请求的分片数。
    - `skipped`
      - （整数）跳过请求的分片数，因为轻量级检查有助于意识到该分片上没有文档可能匹配。这通常发生在搜索请求包含范围过滤器并且分片仅具有超出该范围的值时。
    - `failed`

        - （整数）未能执行请求的分片数。请注意，未分配的分片将被视为既不成功也不失败。因此，失败+成功少于总数表明某些分片未分配。
- `hits`
    - (object) 包含返回的文档和元数据。
    - hits的属性：
        - `total`
            - （对象）有关返回文档数量的元数据。
            - `total` 的属性：
                - `value`
                    - （整数）返回文档的总数。
                - `relation`
                    - (string) 表示 value 参数中返回的文档数是准确的还是下限。
                    - relation的值：
                    - `eq`
                        - 准确的
                    - `gte`
                        - 下限，包括退回的文件
        - `max_score`
            - (float) 最高返回文档 _score。
            - 对于不按 _score 排序的请求，此值为空。
        - `hits`
            - （对象数组）返回的文档对象数组。
            - hits对象属性：
                - `_index`
                    - （字符串）包含返回文档的索引的名称。
                - `_type`
                    - （字符串）返回文档的映射类型。
                - `_id`
                    - （字符串）返回文档的唯一标识符。此 ID 仅在返回的索引中是唯一的。
                - `_score`
                    - (float) 正的 32 位浮点数，用于确定返回文档的相关性。
                - `_source`
                    - （对象）在索引时为文档传递的原始 JSON 正文。

使用示例

使用q查询参数搜索索引

```
GET /twitter/_search?q=user:kimchy
```

API 返回以下响应：

```
{
```

```
  "took" : 5,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 1.3862942,
    "hits" : [
      {
        "_index" : "twitter",
        "_type" : "_doc",
        "_id" : "0",
        "_score" : 1.3862942,
        "_source" : {
          "date" : "2009-11-15T14:12:12",
          "likes" : 0,
          "message" : "trying out Elasticsearch",
          "user" : "kimchy"
        }
      }
    ]
  }
}
```

使用 q 查询参数搜索多个索引

```
GET /kimchy,elasticsearch/_search?q=user:kimchy
```

使用 q 查询参数搜索所有索引

```
GET /_search?q=user:kimchy
```

或者，您可以在 `<index>` 参数中使用 `_all` 或 `*` 值。

```
GET /_all/_search?q=user:kimchy
GET /*/_search?q=user:kimchy
```

使用查询请求主体参数搜索索引

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

URI搜索

您可以使用查询参数直接在请求 URI 中定义您的搜索条件，而不是在请求正文中。请求 URI 搜索不支持完整的 Elasticsearch Query DSL，但便于测试。

请求方式

```
GET /<index>/_search?q=<parameter>
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。

使用示例

```
GET twitter/_search?q=user:kimchy
```

响应结果示例

```
{
    "timed_out": false,
    "took": 62,
    "_shards":{
        "total" : 1,
        "successful" : 1,
        "skipped" : 0,
        "failed" : 0
    },
    "hits":{
        "total" : {
            "value": 1,
            "relation": "eq"
        },
        "max_score": 1.3862942,
        "hits" : [
            {
                "_index" : "twitter",
                "_type" : "_doc",
                "_id" : "0",
                "_score": 1.3862942,
                "_source" : {
                    "user" : "kimchy",
                    "date" : "2009-11-15T14:12:12",
                    "message" : "trying out Elasticsearch",
                    "likes": 0
                }
            }
        ]
    }
}
```

请求主体搜索

可以使用搜索 DSL 执行搜索请求，该搜索 DSL 在其正文中包括查询 DSL。

请求方式

```
GET /<index>/_search
{
  "query": {<parameters>}
}
```

路径参数

- `<index>`
  - （可选，字符串）用于限制请求的索引名称的逗号分隔列表或通配符表达式。

快速检查任何匹配的文档

如果我们只想知道是否有任何文档与特定查询匹配，我们可以将大小设置为 0 以表示我们对搜索结果不感兴趣。我们还可以将 `terminate_after` 设置为 1 以指示只要找到第一个匹配文档（每个分片）就可以终止查询执行。

```
GET /_search?q=message:number&size=0&terminate_after=1
```

响应将不包含任何命中，因为大小设置为 0。 `hits.total` 将等于 0，表示没有匹配的文档，或者大于 0 表示至少有与查询匹配的文档当它被提前终止时。此外，如果查询提前终止，则 `terminate_early` 标志将在响应中设置为 true。

```
{
  "took": 3,
  "timed_out": false,
  "terminated_early": true,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped" : 0,
    "failed": 0
  },
  "hits": {
    "total" : {
        "value": 1,
        "relation": "eq"
    },
    "max_score": null,
    "hits": []
  }
}
```

响应中的花费时间包含此请求处理所花费的毫秒数，从节点接收到查询后快速开始，直到所有搜索相关工作完成并且在上述 JSON 返回给客户端之前。这意味着它包括在线程池中等待、在整个集群中执行分布式搜索以及收集所有结果所花费的时间。

单据值字段

允许为每个命中返回一个字段的文档值表示，例如：

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

docvalue_fields支持字段字符串也支持对象，同时还可以在对象中指定格式。

Doc value 字段可以在启用了 doc-values 的字段上工作，无论它们是否被存储

- 可以用作通配符，例如：

```
GET /_search
{
    "query" : {
        "match_all": {}
    },
    "docvalue_fields" : [
        {
            "field": "*_date_field",
            "format": "epoch_millis"
        }
    ]
}
```

请注意，如果 fields 参数指定没有 docvalues 的字段，它将尝试从 fielddata 缓存中加载值，从而导致该字段的词语被加载到内存（缓存），这将导致更多的内存消耗。

自定格式

虽然大多数字段不支持自定义格式，但其中一些支持：

- 日期字段可以采用任何日期格式。
- 数字 字段接受DecimalFormat模式.

默认情况下，字段的格式基于合理的配置，该配置取决于它们的映射： `long`，`double` 和其他数字字段被格式化为数字， `keyword` 字段被格式化为字符串， `date` 字段被格式化为配置的日期格式，等等。

> 就其本身而言，docvalue_fields不能用于加载嵌套对象中的字段 — 如果字段在其路径中包含嵌套对象，则不会为该 docvalue 字段返回任何数据。要访问嵌套字段，必须在inner_hits 块中使用docvalue_fields。

字段折叠

允许根据字段值折叠搜索结果。折叠是通过每个折叠键仅选择排序最靠前的文档来完成的。例如，下面的查询检索每个用户的最佳推文，并按喜欢的数量对其进行排序。

```
GET /twitter/_search
{
    "query": {
        "match": {
            "message": "elasticsearch"
        }
    },
    "collapse" : {
        "field" : "user"
    },
    "sort": ["likes"],
    "from": 10
}
```

> 响应中的总命中数表示没有折叠的匹配文档的数量。不同组的总数未知。

用于折叠的字段必须是单值 `keywords` 或 激活 `doc_values` 的数字字段。

展开折叠结果

也可以使用 `inner_hits` 选项展开每个折叠的热门命中。

```
GET /twitter/_search
{
    "query": {
        "match": {
            "message": "elasticsearch"
        }
    },
    "collapse" : {
        "field" : "user",
        "inner_hits": {
            "name": "last_tweets",
            "size": 5,
            "sort": [{ "date": "asc" }]
        },
        "max_concurrent_group_searches": 4
    },
    "sort": ["likes"]
}
```

- 使用“user”字段折叠结果集
- name属性用于响应中内部命中部分的名称（last_tweets）
- size属性表示每个折叠键要检索的 inner_hits 数
- sort属性对每个组内的文档进行排序
- max_concurrent_group_searches属性设置每组允许检索inner_hits的并发请求数

有关受支持选项的完整列表和响应格式，请参阅inner hits。
也可以为每个折叠的命中请求多个 inner_hits。当您想要获得折叠匹配的多个表示时，这可能很有用。

```
GET /twitter/_search
{
    "query": {
        "match": {
            "message": "elasticsearch"
        }
    },
    "collapse" : {
        "field" : "user",
        "inner_hits": [
            {
                "name": "most_liked",
                "size": 3,
                "sort": ["likes"]
            },
            {
                "name": "most_recent",
                "size": 3,
                "sort": [{ "date": "asc" }]
            }
        ]
    },
    "sort": ["likes"]
}
```

- 使用“user”字段折叠结果集
- 为用户返回三个最喜欢的推文
- 为用户返回三个最近的推文

from/size

可以使用 from 和 size 参数对结果进行分页。 from 参数定义了您要获取的第一个结果的偏移量。 size 参数允许您配置要返回的最大命中数。
虽然 from 和 size 可以设置为请求参数，但它们也可以在搜索正文中设置。默认为 0，大小默认为 10。

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

注意 from + size 不能超过 index.max_result_window 索引设置，默认为 10,000。

min_score

排除 _score 小于 min_score 中指定的最小值的文档：

```
GET /_search
{
    "min_score": 0.5,
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

请注意，大多数情况下，这没有多大意义，但它是为高级用例提供的。

命名查询

每个过滤器和查询都可以在其顶级定义中接受 _name。

```
GET /_search
{
    "query": {
        "bool" : {
            "should" : [
                {"match" : { "name.first" : {"query" : "shay", "_name" : "first"} }},
                {"match" : { "name.last" : {"query" : "banon", "_name" : "last"} }}
            ],
            "filter" : {
                "terms" : {
                    "name.last" : ["banon", "kimchy"],
                    "_name" : "test"
                }
            }
        }
    }
}
```

搜索响应将包含每个匹配的匹配查询。查询和过滤器的标记仅对 bool 查询有意义。

后置过滤器

在已经计算聚合之后，post_filter 将应用于搜索请求最后的搜索命中。它的目的最好用例子来解释：
假设您正在销售具有以下属性的衬衫：

```
PUT /shirts
{
    "mappings": {
        "properties": {
            "brand": { "type": "keyword"},
            "color": { "type": "keyword"},
            "model": { "type": "keyword"}
        }
    }
}

PUT /shirts/_doc/1?refresh
{
    "brand": "gucci",
```

```
    "color": "red",
    "model": "slim"
}
```

假设用户指定了两个过滤器：`color:red` 和 `brand:gucci`。

您只想在搜索结果中向他们展示Gucci制作的红色衬衫。通常，您会使用 bool 查询来执行此操作：

```
GET /shirts/_search
{
  "query": {
    "bool": {
      "filter": [
        { "term": { "color": "red"   }},
        { "term": { "brand": "gucci" }}
      ]
    }
  }
}
```

但是，您还想使用分面导航来显示用户可以单击的其他选项列表。也许您有一个`model`字段，允许用户将搜索结果限制为红色 Gucci T 恤或礼服衬衫。

这可以通过`terms`聚合来完成：

```
GET /shirts/_search
{
  "query": {
    "bool": {
      "filter": [
        { "term": { "color": "red"   }},
        { "term": { "brand": "gucci" }}
      ]
    }
  },
  "aggs": {
    "models": {
      "terms": { "field": "model" }
    }
  }
}
```

但也许您还想告诉用户有多少 Gucci 衬衫有其他颜色可供选择。如果您只是在颜色字段上添加`terms`聚合，则只会返回红色，因为您的查询仅返回 Gucci 的红色衬衫。
相反，您希望在聚合期间包含所有颜色的衬衫，然后仅将颜色过滤器应用于搜索结果。这是 post_filter 的目的：

```
{
  "query": {
    "bool": {
      "filter": {
        "term": { "brand": "gucci" }
      }
    }
```

```
  },
  "aggs": {
    "colors": {
      "terms": { "field": "color" }
    },
    "color_red": {
      "filter": {
        "term": { "color": "red" }
      },
      "aggs": {
        "models": {
          "terms": { "field": "model" }
        }
      }
    }
  },
  "post_filter": {
    "term": { "color": "red" }
  }
}
```

- 主查询现在查找Gucci的所有衬衫，无论颜色如何。
- colors 聚合返回Gucci衬衫的流行颜色。
- color_red聚合将模型子聚合限制为红色Gucci 衬衫。
- 最后，post_filter从搜索结果中删除除红色以外的颜色。

其他

请参参阅官方文档。

5.2 Query DSL

Elasticsearch 提供了基于 JSON 的完整 Query DSL（Domain Specific Language）来定义查询。将查询 DSL 视为查询的 AST（抽象语法树），由两种类型的子句组成：

- 叶查询子句
  - 叶查询子句在特定字段中查找特定值，例如 `match`，`term` 或 `range` 查询。这些查询可以自己使用。
- 复合查询子句
  - 复合查询子句包装其他叶或复合查询，并用于以逻辑方式组合多个查询（例如 `bool` 或 `dis_max` 查询），或改变它们的行为（例如 `constant_score` 查询）。

查询子句的行为不同，具体取决于它们的查询上下文或过滤器上下文。

查询和过滤上下文

相关性分数

默认情况下，Elasticsearch 按相关性分数对匹配的搜索结果进行排序，相关性分数衡量每个文档与查询的匹配程度。
相关性分数是一个正浮点数，在搜索 API 的 _score 元字段中返回。 _score 越高，文档越相关。虽然每种查询类型可以不同地计算相关性分数，但分数计算还取决于查询子句是在查询还是过滤上下文中运行。

查询上下文

在查询上下文中，查询子句回答“该文档与该查询子句匹配程度如何？”的问题。除了决定文档是否匹配之外，查询子句还计算 _score 元字段中的相关性分数。
只要将查询子句传递给查询参数（例如搜索 API 中的查询参数），查询上下文就会生效。

过滤上下文

在过滤器上下文中，查询子句回答“此文档是否匹配此查询子句？”的问题。答案很简单，是或否 — 不计算分数。过滤上下文主要用于过滤结构化数据，例如

- 这个时间戳是否在 2015 年到 2016 年的范围内？
- 状态字段是否设置为“已发布”？

Elasticsearch 会自动缓存常用的过滤器，以提高性能。
只要将查询子句传递给过滤器参数（例如 `bool` 查询中的 `filter` 或 `must_not` 参数、`constant_score` 查询中的 `filter` 参数或 `filter` 聚合），过滤器上下文就会生效。

使用示例

下面是在搜索 API 的查询和过滤上下文中使用的查询子句示例。此查询将匹配满足以下所有条件的文档：

- `title` 字段包含单词 `search`。
- `content` 字段包含单词 `elasticsearch`。
- `status` 字段包含 `published` 的确切字词。
- `publish_date` 字段包含从2015年1月1日开始的日期。

```
GET /_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title":   "search" }},
        { "match": { "content": "elasticsearch" }}
      ],
      "filter": [
        { "term":  { "status": "published" }},
        { "range": { "publish_date": { "gte": "2015-01-01" }}}
      ]
    }
  }
}
```

复合查询

复合查询包装其他复合查询或叶查询，以组合它们的结果和分数，改变它们的行为，或者从查询切换到过滤上下文。

该组中的查询是：

- `bool`

  用于组合多个叶或复合查询子句的默认查询，如 must、should、must_not 或 filter 子句。 must 和 should 子句的分数相结合 — 匹配的子句越多越好 — 而 must_not 和 filter 子句在过滤上下文中执行。

- `boosting`

  返回匹配`positive`查询的文档，但减少也匹配`negative`查询的文档的分数。

- `constant_score`

  包装另一个查询，但在过滤器上下文中执行它的查询。所有匹配的文档都被赋予相同的“常量”`_score`。

- `dis_max`

  接受多个查询并返回与任何查询子句匹配的任何文档的查询。 bool 查询结合了所有匹配查询的分数，而 dis_max 查询使用单个最佳匹配查询子句的分数。

- `function_score`

  使用函数修改主查询返回的分数，以考虑流行度、新近度、距离或使用脚本实现的自定义算法等因素。

详细使用查看对应的文档，点击超链接即可。

全文查询

全文查询使您能够搜索分析的文本字段，例如电子邮件正文。使用在索引期间应用于字段的相同分析器处理查询字符串。

该组中的查询是：

- `intervals`

  允许对匹配项的排序和接近度进行细粒度控制的全文查询。

- `match`

  用于执行全文查询的标准查询，包括模糊匹配和短语或邻近查询。

- `match_bool_prefix`

  创建一个`bool`查询，将每个`term`匹配为`term`查询，但最后一个`term`除外，它匹配为前缀查询

- `match_phrase`

  与`match`查询类似，但用于匹配精确短语或单词邻近匹配。

- `match_phrase_prefix`

  与`match_phrase`查询类似，但对最终单词进行通配符搜索。

- `multi_match`

  匹配查询的多字段。

- `common` **terms**

  一个更专业的查询，它更偏爱不常见的单词。

- `query_string`

  支持紧凑的 Lucene 查询字符串语法，允许您在单个查询字符串中指定 AND|OR|NOT 条件和多字段搜索。仅供专家用户使用。

- `simple_query_string`

  更简单、更健壮的`query_string`语法版本，适合直接向用户公开。

详细使用查看对应的文档，点击超链接即可。

匹配所有查询

最简单的查询，匹配所有文档，所有文档的 _score 均为 1.0。

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

_score 可以用 boost 参数改变：

```
GET /_search
{
    "query": {
        "match_all": { "boost" : 1.2 }
    }
}
```

这与 match_all 查询相反，它不匹配任何文档。

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

6 聚合

聚合框架有助于提供基于搜索查询的聚合数据。它基于称为聚合的简单构建块，可以组合这些块以构建复杂的数据摘要。

聚合可以看作是在一组文档上构建分析信息的工作单元。执行的上下文定义了该文档集是什么（例如，顶级聚合在搜索请求的已执行查询/过滤器的上下文中执行）。

有许多不同类型的聚合，每种都有自己的目的和输出。为了更好地理解这些类型，通常更容易将它们分为四个主要系列：

- **Bucketing**

  构建存储桶的聚合系列，其中每个存储桶都与一个键和一个文档标准相关联。执行聚合时，将对上下文中的每个文档评估所有存储桶标准，并且当标准匹配时，该文档被认为“fall in”相关存储桶中。在聚合过程结束时，我们将得到一个桶列表——每个桶都有一组“属于”它的文档。

- **Metric**

  跟踪和计算一组文档的指标的聚合。

- **Matrix**

  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系列。与度量和桶聚合不同，这个聚合系列还不支持脚本。

- **Pipeline**

  聚合其他聚合及其相关指标的输出的聚合

接下来是有趣的部分。由于每个存储桶有效地定义了一个文档集（属于该存储桶的所有文档），因此可以潜在地将存储桶级别的聚合关联起来，并且这些聚合将在该存储桶的上下文中执行。这就是聚合真正强大的地方：**聚合可以嵌套**！

> 提示：
>
> 分桶聚合可以有子聚合（分桶或度量）。将为它们的父聚合生成的桶计算子聚合。嵌套聚合的级别/深度没有硬性限制（可以将聚合嵌套在“父”聚合下，该聚合本身是另一个更高级别聚合的子聚合）。
>
> 聚合对数据的`double`表示进行操作。因此，在绝对值大于 `2^53` 的 long 上运行时，结果可能是近似的。

详细使用查看对应的文档，点击超链接即可。

聚合数据结构

以下代码片段捕获了聚合的基本结构：

```
"aggregations" : {
    "<aggregation_name>" : {
        "<aggregation_type>" : {
            <aggregation_body>
        }
        [,"meta" : {  [<meta_data_body>] } ]?
        [,"aggregations" : { [<sub_aggregation>]+ } ]?
    }
    [,"<aggregation_name_2>" : { ... } ]*
}
```

JSON中的聚合对象（也可以使用键aggs）保存要计算的聚合。每个聚合都与用户定义的逻辑名称相关联（例如，如果聚合计算平均价格，则将其命名为avg_price是有意义的）。这些逻辑名称也将用于唯一标识响应中的聚合。每个聚合都有一个特定的类型（上述代码片段中的 <aggregation_type> ）并且通常是命名聚合体中的第一个键。每种类型的聚合都定义了自己的主体，具体取决于聚合的性质（例如，特定字段上的 avg 聚合将定义计算平均值的字段）。在聚合类型定义的同一级别，可以选择定义一组附加聚合，尽管这仅在您定义的聚合具有分桶性质时才有意义。在这种情况下，您在桶聚合级别上定义的子聚合将为桶聚合构建的所有桶计算。例如，如果您在`range`聚合下定义了一组聚合，则将为定义的范围存储桶计算子聚合。

值数据源

一些聚合作用于从聚合文档中提取的值。通常，这些值将从使用聚合的字段键设置的特定文档字段中提取。也可以定义一个脚本来生成值（每个文档）。

当为聚合配置`field`和`script`设置时，脚本将被视为值脚本。普通脚本在文档级别进行评估（即脚本可以访问与文档关联的所有数据），而值脚本在值级别进行评估。在这种模式下，值是从配置的字段中提取的，并且脚本用于对这些值应用“转换”。

> 提示：
>
> 使用脚本时，还可以定义 lang 和 params 设置。前者定义了使用的脚本语言（假设在 Elasticsearch 中可以使用正确的语言，默认情况下或作为插件）。后者可以将脚本中的所有“动态”表达式定义为参数，从而使脚本在调用之间保持静态（这将确保在 Elasticsearch 中使用缓存的编译脚本）。

Elasticsearch 使用映射中字段的类型来确定如何运行聚合和格式化响应。然而，有两种情况下 Elasticsearch 无法识别这些信息：未映射的字段（例如，在跨多个索引的搜索请求的情况下，并且只有其中一些具有字段的映射）和纯脚本。对于这些情况，可以使用 value_type 选项为 Elasticsearch 提供提示，该选项接受以下值：string、long（适用于所有整数类型）、double（适用于所有小数类型，如 float 或 scaled_float）、date、ip和boolean。

7 索引分析

索引分析模块充当分析器的可配置注册表，可用于将字符串字段转换为单独的`terms`，这些`terms`是：

- 添加到倒排索引以使文档可搜索
- 由高级查询（例如匹配查询）用于生成搜索词。

有关配置详细信息，请参阅文本分析。

文本分析是将非结构化文本（如电子邮件正文或产品描述）转换为针对搜索优化的结构化格式的过程。

Elasticsearch在索引或搜索`text`字段时执行文本分析。

如果您的索引不包含`text`字段，则无需进一步设置；

但是，如果您使用`text`字段或文本搜索未按预期返回结果，则配置文本分析通常会有所帮助。如果您使用 Elasticsearch来执行以下操作，您还应该查看分析配置：

- 建立一个搜索引擎
- 挖掘非结构化数据
- 微调特定语言的搜索
- 进行词典或语言研究

7.1 文本分析概述

文本分析使Elasticsearch能够执行全文搜索，其中搜索返回所有相关结果，而不仅仅是精确匹配。

如果您搜索`Quick fox jumps`，您可能想要包含`A quick brown fox jumps over the lazy dog`的文档，并且您可能还想要包含相关词（例如`fast fox`或`foxes jump`）的文档。

标记化

分析法通过标记化使全文搜索成为可能：将文本分解成较小的块，称为标记。在大多数情况下，这些标记是单个词。

如果你把短语`the quick brown fox jumps`作为一个单一的字符串进行索引，而用户搜索`quick fox`，这并不被认为是一个匹配。然而，如果你将这个短语标记化，并对每个词单独建立索引，那么查询字符串中的短语就可以被单独查找。这意味着它们可以被`quick fox`，`fox brown`或其他变体的搜索所匹配。

标准化

标记化支持对单个短语进行匹配，但每个标记仍按字面意思匹配。

这表示：

- 搜索`Quick`不会快速匹配，即使您可能希望任一短语匹配另一个。
- 尽管`fox`和`foxes`共享相同的词根，但搜索`foxes`将不会匹配`fox`，反之亦然。
- 搜索`jumps`不会匹配`leaps`。虽然它们不共享词根，但它们是同义词并且具有相似的含义。

为了解决这些问题，文本分析可以将这些标记标准化为标准格式。这允许您匹配与搜索词不完全相同但足够相似以仍然相关的标记。例如：

- `Quick` 可以小写：`quick`。
- `foxes` 可以被词干或简化为它的词根：`fox`。
- `jump` 和 `leap` 是同义词，可以作为一个词进行索引：`jump`。

为确保搜索词与这些词按预期匹配，您可以将相同的标记化和标准化规则应用于查询字符串。例如，对 `Foxes leap` 的搜索可以标准化为对 `fox jump` 的搜索。

自定义文本分析

文本分析由分析器执行，分析器是一组管理整个过程的规则。
Elasticsearch包含一个默认分析器，称为标准分析器，它适用于大多数开箱即用的用例。
如果您想定制您的搜索体验，您可以选择不同的内置分析器，甚至配置自定义分析器。自定义分析器可让您控制分析过程的每个步骤，包括：

- 标记化之前对文本的更改
- 文本如何转换为标记
- 在索引或搜索之前对标记进行标准化更改

7.2 文本分析概念

本节解释Elasticsearch中文本分析的基本概念。

剖析分析器

分析器 — 无论是内置的还是自定义的 — 只是一个包，其中包含三个较低级别的构建块：字符过滤器、标记器和标记过滤器。
内置分析器将这些构建块预先打包到适用于不同语言和文本类型的分析器中。 Elasticsearch 还公开了各个构建块，以便可以将它们组合起来定义新的自定义分析器。

字符过滤器

字符过滤器将原始文本作为字符流接收，并可以通过添加、删除或更改字符来转换流。例如，字符过滤器可用于将印度-阿拉伯数字 (٠١٢٣٤٥٦٧٨٩) 转换为阿拉伯-拉丁等价物 (0123456789)，或从流中去除像 `<b>` 这样的 HTML元素。
分析器可能有零个或多个字符过滤器，它们按顺序应用。

分词器

分词器接收一个字符流，将其分解为单个令牌（通常是单个单词），并输出一个令牌流。例如，只要看到任何空格，空格分词器就会将文本 `"Quick brown fox!"` 为词组 `[Quick, brown, fox!]`。
分词器还负责记录每个词条的顺序或位置以及该词条所代表的原始词的开始和结束字符偏移量。
分析器必须只有一个分词器。

令牌过滤器

令牌过滤器接收令牌流并可以添加、删除或更改令牌。例如，小写令牌过滤器将所有标记转换为小写，停止令牌过滤器从令牌流中删除常用词（停止词），例如，同义词令牌过滤器将同义词引入令牌流。
令牌过滤器不允许更改每个令牌的位置或字符偏移量。
分析器可能有零个或多个令牌过滤器，它们按顺序应用。

索引和搜索分析

文本分析发生在两次：

- **Index time**

  当文档被索引时，任何文本字段值都会被分析。

- **Search time**

  在文本字段上运行全文搜索时，会分析查询字符串（用户正在搜索的文本）。

  搜索时间也称为查询时间。

每次使用的分析器或分析规则集分别称为索引分析器或搜索分析器。

索引和搜索分析器协同工作

在大多数情况下，应在索引和搜索时使用相同的分析器。这可确保字段的值和查询字符串更改为相同形式的标记。反过来，这可以确保令牌在搜索期间按预期匹配。

例如

文档在文本字段中使用以下值进行索引：

```
The QUICK brown foxes jumped over the dog!
```

该字段的索引分析器将值转换为令牌并将它们标准化。在这种情况下，每个令牌代表一个单词：

```
[ quick, brown, fox, jump, over, dog ]
```

然后将这些令牌编入索引。
稍后，用户在相同的文本字段中搜索：

```
"Quick fox"
```

用户希望此搜索匹配之前索引的句子 `The QUICK brown foxes jumped over the dog!` 。
但是，查询字符串不包含文档原始文本中使用的确切单词：

- quick与QUICK
- fox与foxes

考虑到这一点，使用相同的分析器分析查询字符串。该分析器产生以下令牌：

```
[ quick, fox ]
```

为了执行搜索，Elasticsearch将这些查询字符串令牌与文本字段中索引的令牌进行比较。

Token	Query string	`text` field
`quick`	X	X
`brown`		X
`fox`	X	X
`jump`		X

Token	Query string	`text` field
`over`		X
`dog`		X

因为字段值是查询字符串以相同的方式分析，所以它们创建了相似的标记。令牌 quick 和 fox 是完全匹配的。这意味着搜索匹配包含"The QUICK brown foxes jumped over the dog!"，正如用户所期望的那样。

何时使用不同的搜索分析器

虽然不太常见，但有时在索引和搜索时使用不同的分析器是有意义的。为了实现这一点，Elasticsearch 允许您指定一个单独的搜索分析器。
通常，仅当对字段值使用相同形式的令牌时才应指定单独的搜索分析器，并且查询字符串会创建意外或不相关的搜索匹配。
例如

Elasticsearch用于创建仅匹配以提供的前缀开头的单词的搜索引擎。例如，搜索 tr 应该返回 tram 或 trope，但绝不会返回 tax 或 bat。
一个文档被添加到搜索引擎的索引中；本文档在文本字段中包含一个这样的词：

```
"Apple"
```

该字段的索引分析器将值转换为标记并将它们标准化。在这种情况下，每个标记都代表该单词的潜在前缀：

```
[ a, ap, app, appl, apple]
```

然后将这些令牌编入索引。
稍后，用户在相同的文本字段中搜索：

```
"appli"
```

用户希望此搜索仅匹配以 appli 开头的单词，例如appliance或application。搜索不应匹配apple。
但是，如果使用索引分析器分析此查询字符串，它将产生以下令牌：

```
[ a, ap, app, appl, appli ]
```

当 Elasticsearch 将这些查询字符串标记与为苹果索引的令牌进行比较时，它会找到几个匹配项。

Token	`appli`	`apple`
`a`	X	X
`ap`	X	X
`app`	X	X
`appl`	X	X
`appli`		X

这意味着搜索将错误地匹配apple。不仅如此，它还能匹配任何以 a 开头的单词。
要解决此问题，您可以为文本字段上使用的查询字符串指定不同的搜索分析器。
在这种情况下，您可以指定一个生成单个标记而不是一组前缀的搜索分析器：

```
[ appli ]
```

此查询字符串标记只会匹配以 appli 开头的单词的标记，这更好地符合用户的搜索期望。

词干

词干提取是将单词简化为词根形式的过程。这确保了在搜索期间单词匹配的变体。
例如，walking 和 walk 可以被提取为同一个词根：walk。一旦词干化，任何一个词的出现都会在搜索中与另一个词匹配。
词干提取依赖于语言，但通常涉及从单词中删除前缀和后缀。
在某些情况下，词干词的词根形式可能不是真正的词。例如，`jumping` 和 `jumpiness` 都可以归结为 `jumpi`。虽然 `jumpi` 不是一个真正的英语单词，但搜索并不重要。如果一个单词的所有变体都简化为相同的词根形式，它们将正确匹配。

词干过滤器

在Elasticsearch中，词干提取由词干分析器标记过滤器处理。这些标记过滤器可以根据它们的词干方式进行分类：

- Algorithmic stemmers, 基于一组规则的词干
- Dictionary stemmers, 通过查找字典的词干

由于词干分析会更改标记，我们建议在索引和搜索分析期间使用相同的词干分析器标记过滤器。

算法词干分析器

算法词干分析器对每个单词应用一系列规则，以将其简化为词根形式。例如，英语的算法词干分析器可能会从复数词的末尾删除 -s 和 -es 前缀。

算法词干分析器有几个优点：

- 它们需要很少的设置，并且通常开箱即用。
- 他们使用很少的内存。
- 它们通常比字典词干分析器更快。

但是，大多数算法词干分析器只会更改单词的现有文本。这意味着它们可能无法很好地处理不包含其词根形式的不规则单词，例如：

- `be`, `are`, 和 `am`
- `mouse` 和 `mice`
- `foot` 和 `feet`

以下标记过滤器使用算法词干：

- `stemmer`, 它为多种语言提供了算法词干提取，其中一些具有额外的变体。
- `kstem`, 将算法词干与内置字典相结合的英语词干分析器。
- `porter_stem`, 我们推荐的英语算法词干分析器。
- `snowball`, 它对多种语言使用基于Snowball的词干规则。

字典词干分析器

字典词干分析器在提供的字典中查找单词，用字典中的词干词替换未词干的词变体。

从理论上讲，字典词干分析器非常适合：

- 词干不规则词
- 区分拼写相似但概念上不相关的单词，例如：
  - `organ` 与 `organization`
  - `broker` 与 `broken`

在实践中，算法词干分析器通常优于字典词干分析器。这是因为字典词干分析器有以下缺点：

- 字典质量
  - 字典词干分析器的好坏取决于它的字典。为了正常工作，这些词典必须包含大量单词，定期更新，并随着语言趋势而变化。通常，当字典可用时，它是不完整的，其中一些条目已经过时了。
- 词量和性能
  - 词典词干分析器必须将其词典中的所有单词、前缀和后缀加载到内存中。这可能会使用大量 RAM。低质量的字典在去除前缀和后缀时可能效率较低，这会显着减慢词干提取过程。

您可以使用 hunspell标记过滤器来执行字典词干提取。

控制词干

有时，词干可以产生拼写相似但概念上不相关的共享词根。例如，词干分析器可以将 skys 和 skiing 归结为同一个词根：ski。

为了防止这种情况并更好地控制词干，您可以使用以下标记过滤器：

- `stemmer_override`，它允许您定义用于词干特定标记的规则。
- `keyword_marker`，将指定的标记标记为关键字。关键字标记不会被后续的词干分析器标记过滤器提取。
- `conditional`，可用于将标记标记为关键字，类似于 `keyword_marker` 过滤器。

对于内置语言分析器，您还可以使用 `stem_exclusion` 参数来指定不会被词干的单词列表。

令牌图

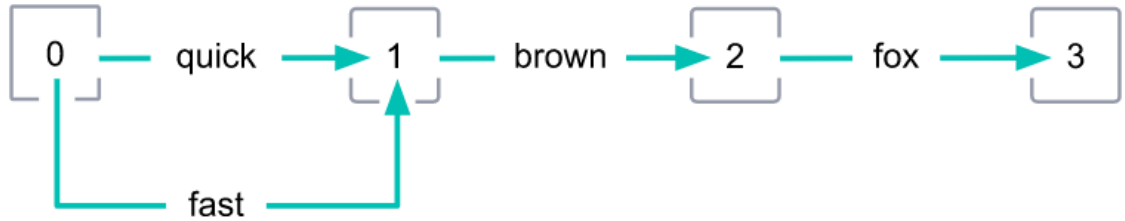
当标记器将文本转换为标记流时，它还会记录以下内容：

- 流中每个令牌的 `position`
- `positionLength`，一个令牌跨越的位置数

使用这些，您可以为流创建称为标记图的有向无环图。在令牌图中，每个位置代表一个节点。每个标记代表一条边或弧，指向下一个位置。

同义词

一些令牌过滤器可以将新令牌（如同义词）添加到现有令牌流中。这些同义词通常与现有令牌跨越相同的位置。
在下图中，quick 及其同义词 fast 的位置均为 0。它们跨越相同的位置。

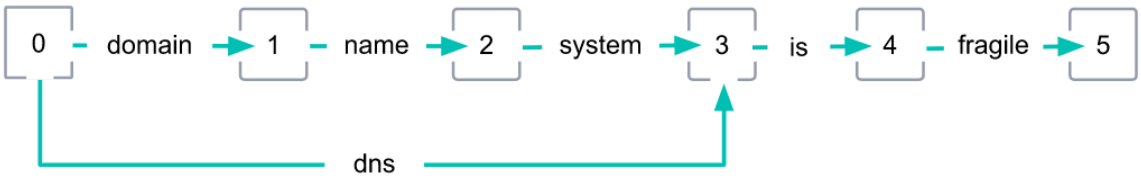


多位置令牌

一些令牌过滤器可以添加跨越多个位置的令牌。这些可以包括多词同义词的令牌，例如使用“atm”作为“automatic”的同义词。
但是，只有一些令牌过滤器，称为图令牌过滤器，准确记录了多位置令牌的 `positionLength`。此过滤器包括：

- `synonym_graph`
- `word_delimiter_graph`

在下图中，`domain name system` 及其同义词 `dns` 的位置均为 0。但是，dns 的 positionLength 为 3。图中其他令牌的默认 positionLength 为 1。



使用令牌图进行搜索

索引忽略 positionLength 属性并且不支持包含多位置令牌的令牌图。

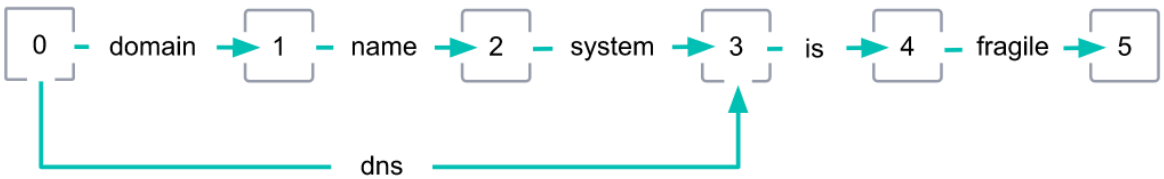
但是，查询（例如 match 或 match_phrase 查询）可以使用这些图从单个查询字符串生成多个子查询。

比如

用户使用 match_phrase 查询来搜索以下短语：

```
domain name system is fragile
```

在搜索分析过程中，将域名系统的同义词 dns 添加到查询字符串的令牌流中。 dns 令牌的 positionLength 为 3。



match_phrase 查询使用此图为以下短语生成子查询：

```
dns is fragile
domain name system is fragile
```

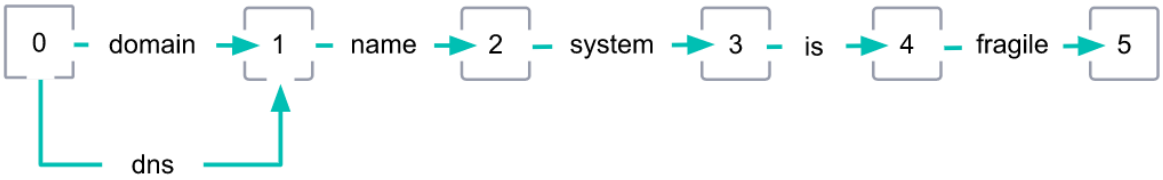
这意味着查询匹配包含`dns is fragile`或`domain name system is fragile`的文档。

无效的令牌图

以下标记过滤器可以添加跨越多个位置但仅记录默认位置长度为 1 的标记：

- `synonym`
- `word_delimiter`

这意味着这些过滤器将为包含此类令牌的流生成无效的令牌图。
下图中，dns是`domain name system`的多位同义词。但是，dns 的默认 positionLength 值为 1，导致图表无效。



避免使用无效的令牌图进行搜索。无效的图表可能会导致意外的搜索结果。

7.3 配置文本分析

默认情况下，Elasticsearch使用标准分析器进行所有文本分析。标准分析器为您提供对大多数自然语言和用例的开箱即用支持。如果您选择按原样使用标准分析仪，则无需进一步配置。
如果标准分析器不符合您的需求，请查看并测试Elasticsearch的其他内置分析器。内置分析器不需要配置，但有一些支持选项可用于调整其行为。例如，您可以使用要删除的自定义停用词列表配置标准分析器。
如果没有适合您需求的内置分析器，您可以测试并创建自定义分析器。定制分析器涉及选择和组合不同的分析器组件，让您更好地控制过程。

测试分析器

分析API是查看分析器生成的term的宝贵工具。可以在请求中内联指定内置分析器：

```
POST _analyze
{
  "analyzer": "whitespace",
  "text":     "The quick brown fox."
}
```

API 返回以下响应：

```
{
  "tokens": [
    {
      "token": "The",
      "start_offset": 0,
      "end_offset": 3,
      "type": "word",
      "position": 0
    },
```

```
    {
      "token": "quick",
      "start_offset": 4,
      "end_offset": 9,
      "type": "word",
      "position": 1
    },
    {
      "token": "brown",
      "start_offset": 10,
      "end_offset": 15,
      "type": "word",
      "position": 2
    },
    {
      "token": "fox.",
      "start_offset": 16,
      "end_offset": 20,
      "type": "word",
      "position": 3
    }
  ]
}
```

您还可以测试以下组合：

- 分词器
- 零个或多个令牌过滤器
- 零个或多个字符过滤器

```
POST _analyze
{
  "tokenizer": "standard",
  "filter":  [ "lowercase", "asciifolding" ],
  "text":      "Is this déja vu?"
}
```

API 返回以下响应：

```
{
  "tokens": [
    {
      "token": "is",
      "start_offset": 0,
      "end_offset": 2,
      "type": "<ALPHANUM>",
      "position": 0
    },
    {
      "token": "this",
      "start_offset": 3,
      "end_offset": 7,
      "type": "<ALPHANUM>",
      "position": 1
```

```
    },
    {
      "token": "deja",
      "start_offset": 8,
      "end_offset": 12,
      "type": "<ALPHANUM>",
      "position": 2
    },
    {
      "token": "vu",
      "start_offset": 13,
      "end_offset": 15,
      "type": "<ALPHANUM>",
      "position": 3
    }
  ]
}
```

> 从分析API的输出可以看出，分线器不仅将单词转换为词条，还记录了每个词条的顺序或相对位置（用于词组查询或词邻近查询），以及原始文本中的每个词语开始和结束字符的偏移量（用于突出显示搜索片段）。

或者，在特定索引上运行分析API时，可以参考自定义分析器：

```
PUT my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "std_folded": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "asciifolding"
          ]
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "my_text": {
        "type": "text",
        "analyzer": "std_folded"
      }
    }
  }
}

GET my_index/_analyze
{
  "analyzer": "std_folded",
  "text":     "Is this déjà vu?"
}
```

```
GET my_index/_analyze
{
  "field": "my_text",
  "text":  "Is this déjà vu?"
}
```

- 定义一个名为 `std_folded` 的自定义分析器。
- 字段 `my_text` 使用 `std_folded` 分析器。
- 要引用此分析器，分析API必须指定索引名称。
- 按名称引用分析器或使用my_text字段的分析器。

API返回以下响应：

```
{
  "tokens": [
    {
      "token": "is",
      "start_offset": 0,
      "end_offset": 2,
      "type": "<ALPHANUM>",
      "position": 0
    },
    {
      "token": "this",
      "start_offset": 3,
      "end_offset": 7,
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "deja",
      "start_offset": 8,
      "end_offset": 12,
      "type": "<ALPHANUM>",
      "position": 2
    },
    {
      "token": "vu",
      "start_offset": 13,
      "end_offset": 15,
      "type": "<ALPHANUM>",
      "position": 3
    }
  ]
}
```

配置内置分析器

内置分析器无需任何配置即可直接使用。但是，其中一些支持配置选项以改变其行为。

例如，可以将标准分析器配置为支持停用词列表：

```
PUT my_index
{
```

```
  "settings": {
    "analysis": {
      "analyzer": {
        "std_english": {
          "type":  "standard",
          "stopwords": "_english_"
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "my_text": {
        "type":  "text",
        "analyzer": "standard",
        "fields": {
          "english": {
            "type":  "text",
            "analyzer": "std_english"
          }
        }
      }
    }
  }
}

POST my_index/_analyze
{
  "field": "my_text",
  "text": "The old brown cow"
}

POST my_index/_analyze
{
  "field": "my_text.english",
  "text": "The old brown cow"
}
```

- 我们将`std_english`分析器定义为基于标准分析器，但配置为删除预定义的英语停用词列表。
- `my_text`字段直接使用标准分析器，无需任何配置。不会从此字段中删除停用词。
  - 结果是：`[ the, old, brown, cow ]`
- `my_text.english`字段使用`std_english`分析器，因此将删除英语停用词。
  - 结果是：`[ old, brown, cow ]`

创建自定义分析器

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，它使用以下适当组合：

- 零个或多个字符过滤器
- 一个分词器
- 零个或多个令牌过滤器

配置

自定义分析器接受以下参数：

参数	描述
`tokenizer`	内置或定制的分词器。（必需的）
`char_filter`	可选的内置或自定义字符过滤器数组。
`filter`	可选的内置或自定义令牌过滤器数组。
`position_increment_gap`	当索引一个文本值数组时，Elasticsearch在一个值的最后一个词和下一个值的第一个词之间插入一个假的“间隙”，以确保一个短语查询不匹配来自不同数组元素的两个词。默认为100。有关更多信息，请参见 `position_increment_gap`。

示例配置

这是一个结合了以下内容的示例：

字符过滤器

- HTML去除字符过滤器

分词器

- Standard Tokenizer

令牌过滤器

- 小写令牌过滤器
- ASCII折叠令牌过滤器

```
PUT my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_custom_analyzer": {
          "type": "custom",
          "tokenizer": "standard",
          "char_filter": [
            "html_strip"
          ],
          "filter": [
            "lowercase",
            "asciifolding"
          ]
        }
      }
    }
  }
}

POST my_index/_analyze
{
```

```
  "analyzer": "my_custom_analyzer",
  "text": "Is this <b>déjà vu</b>?"
}
```

将type设置为custom告诉Elasticsearch我们正在定义一个自定义分析器。

将此与如何配置内置分析器进行比较：`type`将设置为内置分析器的名称，如 `standard` 或 `simple`。

上面的示例产生以下词条：

```
[ is, this, deja, vu ]
```

这是一个更复杂的示例，它结合了以下内容：

字符过滤器

- 映射字符过滤器, 配置为将 `:)` 替换为 `_happy_` 和 `:(` 替换为 `_sad_`

分词器

- Pattern Tokenizer, 配置为在标点符号上拆分

令牌过滤器

- 小写令牌过滤器
- Stop Token Filter, 配置为使用预定义的英文停用词列表

```
PUT my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_custom_analyzer": {
          "type": "custom",
          "char_filter": [
            "emoticons"
          ],
          "tokenizer": "punctuation",
          "filter": [
            "lowercase",
            "english_stop"
          ]
        }
      },
      "tokenizer": {
        "punctuation": {
          "type": "pattern",
          "pattern": "[ .,!?]"
        }
      },
      "char_filter": {
        "emoticons": {
          "type": "mapping",
          "mappings": [
            ":) => _happy_",
            ":( => _sad_"
          ]
```

```
        }
      },
      "filter": {
        "english_stop": {
          "type": "stop",
          "stopwords": "_english_"
        }
      }
    }
  }
}

POST my_index/_analyze
{
  "analyzer": "my_custom_analyzer",
  "text": "I'm a :) person, and you?"
}
```

- 为索引分配一个默认的自定义分析器`my_custom_analyzer`。
  - 此分析器使用稍后在请求中定义的自定义标记器、字符过滤器和令牌过滤器。
- 定义自定义标点符号生成器。
- 定义自定义表情字符过滤器。
- 定义自定义`english_stop`令牌过滤器。

上面的示例产生以下词条：

```
[ i'm, _happy_, person, you ]
```

指定分析器

Elasticsearch提供了多种方法来指定内置或自定义分析器：

- 按`text`字段、索引或查询
- 对于索引或搜索时间

ES如何确定索引分析器

Elasticsearch通过依次检查以下参数来确定要使用的索引分析器：

1. 字段的分析器映射参数。请参阅为字段指定分析器。
2. `analysis.analyzer.default`索引设置。请参阅为索引指定默认分析器.

如果没有指定这些参数，则使用标准分析器标准分析器。

指定字段的分析器

映射索引时，您可以使用分析器映射参数为每个文本字段指定分析器。
以下创建索引API请求将空白分析器设置为标题字段的分析器。

```
PUT my_index
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "analyzer": "whitespace"
      }
    }
  }
}
```

指定索引的默认分析器

除了字段级分析器之外，您还可以设置备用分析器以使用 analysis.analyzer.default 设置。
以下创建索引 API 请求将简单分析器设置为 my_index 的备用分析器。

```
PUT my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "default": {
          "type": "simple"
        }
      }
    }
  }
}
```

ES如何确定搜索分析器

在大多数情况下，不需要指定不同的搜索分析器。这样做可能会对相关性产生负面影响并导致意外的搜索结果。

如果您选择指定单独的搜索分析器，我们建议您在部署到生产环境之前测试您的分析配置。

在搜索时，Elasticsearch通过依次检查以下参数来确定要使用的分析器：

1. 搜索查询中的分析器参数。请参阅为查询指定搜索分析器。
2. 字段的 `search_analyzer` 映射参数。请参阅为字段指定搜索分析器。
3. `analysis.analyzer.default_search` 索引设置。请参阅为索引指定默认搜索分析器。
4. 字段的分析器映射参数。请参阅为字段指定分析器。

如果没有指定这些参数，则使用标准分析器。

指定查询的搜索分析器

编写全文查询时，可以使用analyzer参数指定搜索分析器。如果提供，它将覆盖任何其他搜索分析器。
以下搜索API请求将停止分析器设置为匹配查询的搜索分析器。

```
GET my_index/_search
{
  "query": {
    "match": {
      "message": {
        "query": "Quick foxes",
        "analyzer": "stop"
      }
    }
  }
}
```

指定字段的搜索分析器

映射索引时，可以使用search_analyzer映射参数为每个文本字段指定搜索分析器。
如果提供了搜索分析器，则还必须使用分析器参数指定索引分析器。
以下创建索引API请求将简单分析器设置为标题字段的搜索分析器。

```
PUT my_index
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "analyzer": "whitespace",
        "search_analyzer": "simple"
      }
    }
  }
}
```

指定索引的默认搜索分析器

创建索引时，您可以使用`analysis.analyzer.default_search`设置默认搜索分析器。
如果提供了搜索分析器，则还必须使用`analysis.analyzer.default`设置指定默认索引分析器。
以下创建索引API请求将空白分析器设置为my_index索引的默认搜索分析器。

```
PUT my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "default": {
          "type": "simple"
        },
        "default_search": {
          "type": "whitespace"
        }
      }
    }
  }
}
```

7.4 标准化器

标准化器与分析器类似，只是它们只能发出一个标记。因此，它们没有标记器，只接受可用字符过滤器和标记过滤器的子集。只允许使用基于每个字符的过滤器。例如，允许使用小写过滤器，但不允许使用词干过滤器，它需要将关键字视为一个整体。当前可以在标准化器中使用的过滤器列表如下：`elision`，`german_normalization`，`hindi_normalization`，`indic_normalization`，`lowercase`，`persian_normalization`，`scandinavian_folding`，`serbian_normalization`，`sorani_normalization`，`uppercase`。

自定义标准化器

到目前为止，Elasticsearch还没有内置标准化器，因此获得标准化器的唯一方法是构建自定义标准化器。自定义标准化器采用字符过滤器列表和令牌过滤器列表。

```
PUT index
{
  "settings": {
    "analysis": {
      "char_filter": {
        "quote": {
          "type": "mapping",
          "mappings": [
            "« => \"",
            "» => \""
          ]
        }
      },
      "normalizer": {
        "my_normalizer": {
          "type": "custom",
          "char_filter": ["quote"],
          "filter": ["lowercase", "asciifolding"]
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "foo": {
        "type": "keyword",
        "normalizer": "my_normalizer"
      }
    }
  }
}
```

7.5 其他参考

- 内置分析器参考
- 分词器参考
- 令牌过滤器参考
- 字符过滤器参考

7.6 中文分词

`es`自带了一堆的分词器，比如`standard`、`whitespace`、`language(比如english)`等分词器，但是都对中文分词的效果不太好，此处安装第三方分词器`ik`，来实现分词。

插件地址：https://github.com/medcl/elasticsearch-analysis-ik/tree/v7.6.2

安装IK分词器

```
elasticsearch-plugin install https://github.com/medcl/elasticsearch-analysis-ik/releases/download/v7.6.2/elasticsearch-analysis-ik-7.6.2.zip
```

安装完成后重启elasticsearch

测试IK分词器

`ik`分词器提供了2种分词的模式

1. `ik_max_word`: 将需要分词的文本做最小粒度的拆分，尽量分更多的词。
2. `ik_smart`: 将需要分词的文本做最大粒度的拆分。

测试默认的分词效果

测试语句

```
GET _analyze
{
  "analyzer": "default",
  "text": ["我是小红，今年6岁，刚上幼儿园。"]
}
```

响应结果

```
{
  "tokens" : [
    {
      "token" : "我",
      "start_offset" : 0,
      "end_offset" : 1,
      "type" : "<IDEOGRAPHIC>",
      "position" : 0
    },
    {
      "token" : "是",
      "start_offset" : 1,
      "end_offset" : 2,
      "type" : "<IDEOGRAPHIC>",
      "position" : 1
    },
    {
      "token" : "小",
      "start_offset" : 2,
      "end_offset" : 3,
      "type" : "<IDEOGRAPHIC>",
      "position" : 2
```

```
    },
    {
      "token" : "红",
      "start_offset" : 3,
      "end_offset" : 4,
      "type" : "<IDEOGRAPHIC>",
      "position" : 3
    },
    ..............
  ]
}

```

可以看到默认的分词器，对中文的分词完全无法达到我们中文的分词的效果。

测试ik_max_word的分词

测试语句

```
GET _analyze
{
  "analyzer": "ik_max_word",
  "text": ["我是小红，今年6岁，刚上幼儿园。"]
}
```

响应结果

```
{
  "tokens" : [
    {
      "token" : "我",
      "start_offset" : 0,
      "end_offset" : 1,
      "type" : "CN_CHAR",
      "position" : 0
    },
    {
      "token" : "是",
      "start_offset" : 1,
      "end_offset" : 2,
      "type" : "CN_CHAR",
      "position" : 1
    },
    {
      "token" : "小红",
      "start_offset" : 2,
      "end_offset" : 4,
      "type" : "CN_WORD",
      "position" : 2
    },
    {
      "token" : "今年",
      "start_offset" : 5,
      "end_offset" : 7,
      "type" : "CN_WORD",
```

```
      "position" : 3
    },
    {
      "token" : "6",
      "start_offset" : 7,
      "end_offset" : 8,
      "type" : "ARABIC",
      "position" : 4
    },
    {
      "token" : "岁",
      "start_offset" : 8,
      "end_offset" : 9,
      "type" : "COUNT",
      "position" : 5
    },
    {
      "token" : "刚",
      "start_offset" : 10,
      "end_offset" : 11,
      "type" : "CN_CHAR",
      "position" : 6
    },
    {
      "token" : "上",
      "start_offset" : 11,
      "end_offset" : 12,
      "type" : "CN_CHAR",
      "position" : 7
    },
    {
      "token" : "幼儿园",
      "start_offset" : 12,
      "end_offset" : 15,
      "type" : "CN_WORD",
      "position" : 8
    },
    {
      "token" : "幼儿",
      "start_offset" : 12,
      "end_offset" : 14,
      "type" : "CN_WORD",
      "position" : 9
    },
    {
      "token" : "园",
      "start_offset" : 14,
      "end_offset" : 15,
      "type" : "CN_CHAR",
      "position" : 10
    }
  ]
}
```


可以看到基于`ik`分词可以达到我们需要的分词效果。

测试ik_smart的分词

测试语句

```
GET _analyze
{
  "analyzer": "ik_smart",
  "text": ["我是小红，今年6岁，刚上幼儿园。"]
}
```

响应结果

```
{
  "tokens" : [
    {
      "token" : "我",
      "start_offset" : 0,
      "end_offset" : 1,
      "type" : "CN_CHAR",
      "position" : 0
    },
    {
      "token" : "是",
      "start_offset" : 1,
      "end_offset" : 2,
      "type" : "CN_CHAR",
      "position" : 1
    },
    {
      "token" : "小红",
      "start_offset" : 2,
      "end_offset" : 4,
      "type" : "CN_WORD",
      "position" : 2
    },
    {
      "token" : "今年",
      "start_offset" : 5,
      "end_offset" : 7,
      "type" : "CN_WORD",
      "position" : 3
    },
    {
      "token" : "6岁",
      "start_offset" : 7,
      "end_offset" : 9,
      "type" : "TYPE_CQUAN",
      "position" : 4
    },
    {
      "token" : "刚",
      "start_offset" : 10,
      "end_offset" : 11,
      "type" : "CN_CHAR",
      "position" : 5
```

```
    },
    {
      "token" : "上",
      "start_offset" : 11,
      "end_offset" : 12,
      "type" : "CN_CHAR",
      "position" : 6
    },
    {
      "token" : "幼儿园",
      "start_offset" : 12,
      "end_offset" : 15,
      "type" : "CN_WORD",
      "position" : 7
    }
  ]
}
```

8 SpringBoot集成

项目元数据

- 项目仓库 - https://github.com/spring-projects/spring-data-elasticsearch
- API 文档 - https://docs.spring.io/spring-data/elasticsearch/docs/current/api/
- Bugtracker - https://jira.spring.io/browse/DATAES
- 发布存储库 - https://repo.spring.io/libs-release
- 里程碑存储库 - https://repo.spring.io/libs-milestone
- 快照存储库 - https://repo.spring.io/libs-snapshot

8.1 使用要求

版本

下表显示了Spring Data发布系列使用的 Elasticsearch版本和其中包含的Spring Data Elasticsearch 版本，以及引用该特定Spring Data发布系列的Spring Boot版本：

Spring Data Release Train	Spring Data Elasticsearch	Elasticsearch	Spring Boot
Neumann[1]	4.0.x[1]	7.6.2	2.3.x[1]
Moore	3.2.x	6.8.6	2.2.x
Lovelace	3.1.x	6.2.2	2.1.x
Kay[2]	3.0.x[2]	5.5.0	2.0.x[2]
Ingalls[2]	2.1.x[2]	2.4.0	1.5.x[2]

依赖

在spring boot 中使用还需要导入依赖库

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-elasticsearch</artifactId>
    <version>x.x.x.RELEASE</version>
</dependency>
```

8.2 Elasticsearch客户端

本章说明了受支持的Elasticsearch客户端实现的配置和使用。
Spring Data Elasticsearch在连接到单个Elasticsearch节点或集群的Elasticsearch客户端上运行。尽管Elasticsearch Client 可用于与集群一起工作，但使用 Spring Data Elasticsearch 的应用程序通常使用 Elasticsearch Operations 和 Elasticsearch Repositories 的更高级别抽象。

Transport Client

我们强烈建议使用High Level REST Client而不是TransportClient。

比如查看下面示例

```
@Configuration
public class TransportClientConfig extends ElasticsearchConfigurationSupport
{
    @Bean
    public Client elasticsearchClient() throws UnknownHostException {
        //TransportClient必须配置集群名称
        Settings settings = Settings.builder().put("cluster.name",
"elasticsearch").build();
        TransportClient client = new PreBuiltTransportClient(settings);
        //连接客户端的主机和端口
        client.addTransportAddress(new
TransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
        return client;
    }
    @Bean(name = { "elasticsearchOperations", "elasticsearchTemplate" })
    public ElasticsearchTemplate elasticsearchTemplate() throws
UnknownHostException {
        return new ElasticsearchTemplate(elasticsearchClient());
    }
}

//下面是测试使用代码片段
// ...
IndexRequest request = new IndexRequest("spring-data", "elasticsearch",
randomID())
    .source(someObject)
    .setRefreshPolicy(IMMEDIATE);
IndexResponse response = client.index(request);
```

High Level REST Client

`Java High Level REST Client`是Elasticsearch的默认客户端，它提供了对 `TransportClient` 的直接替代，因为它接受并返回完全相同的请求/响应对象，因此依赖于 Elasticsearch 核心项目。异步调用在客户端管理的线程池上进行操作，并且需要在请求完成时通知回调。

查看下面示例

```
@Configuration
public class RestClientConfig extends AbstractElasticsearchConfiguration {
    @Override
    @Bean
    public RestHighLevelClient elasticsearchClient() {
        //使用构建器提供集群地址、设置默认HttpHeaders或启用SSL
        final ClientConfiguration clientConfiguration =
ClientConfiguration.builder()
            .connectedTo("localhost:9200")
            .build();
        //创建RestHighLevelClient
        return RestClients.create(clientConfiguration).rest();
    }
}

//下面是测试使用代码片段
// ...
@Autowired
RestHighLevelClient highLevelClient;
//也可以获取lowLevelRest()客户端
RestClient lowLevelClient = highLevelClient.lowLevelClient();
// ...

IndexRequest request = new IndexRequest("spring-data")
    .source(singletonMap("feature", "high-level-rest-client"))
    .setRefreshPolicy(IMMEDIATE);
IndexResponse response = highLevelClient.index(request);
```

Reactive Client

ReactiveElasticsearchClient是一个基于WebClient的非官方驱动。它使用Elasticsearch核心项目提供的请求/响应对象。调用直接在响应式堆栈上操作，而不是将异步（线程池绑定）响应包装到响应式类型中。

使用示例

```
static class Config {
    @Bean
    ReactiveElasticsearchClient client() {
        //使用构建器提供集群地址、设置默认HttpHeaders或启用SSL
        ClientConfiguration clientConfiguration =
ClientConfiguration.builder()
            .connectedTo("localhost:9200", "localhost:9291")
            //在配置响应式客户端时，可以使用withWebClientConfigurer钩子来自定义
WebClient
            .withWebClientConfigurer(webClient -> {
```

```
                ExchangeStrategies exchangeStrategies =
ExchangeStrategies.builder()
                    .codecs(configurer -> configurer.defaultCodecs()
                            .maxInMemorySize(-1))
                    .build();
                return
webClient.mutate().exchangeStrategies(exchangeStrategies).build();
            })
            .build();
        return ReactiveRestClients.create(clientConfiguration);
    }
}

//下面是使用示例代码片段
// ...
Mono<IndexResponse> response = client.index(request ->
                                            request.index("spring-data")
                                            .type("elasticsearch")
                                            .id(randomID())
                                            .source(singletonMap("feature",
"reactive-client"))
                                            .setRefreshPolicy(IMMEDIATE);
                                           );
```

> ReactiveClient响应，特别是对于搜索操作，绑定到请求的from(offset)&size(limit)选项。

客户端配置

客户端行为可以通过允许设置SSL选项、连接和套接字超时、标头和其他参数的ClientConfiguration进行更改。

使用示例

```
//定义默认标题（如果需要自定义）
HttpHeaders httpHeaders = new HttpHeaders();
httpHeaders.add("some-header", "on every request") ;
//使用构建器提供集群地址、设置默认HttpHeaders或启用 SSL
ClientConfiguration clientConfiguration = ClientConfiguration.builder()
    //<必须> 提供集群地址
    .connectedTo("localhost:9200", "localhost:9291")
    //[可选] 启用 SSL
    .usingSsl()
    //[可选] 设置代理
    .withProxy("localhost:8888")
    //[可选] 设置路径前缀，主要用于在某些反向代理后面的不同集群时
    .withPathPrefix("ela")
    //设置连接超时。默认值为 10 秒。
    .withConnectTimeout(Duration.ofSeconds(5))
    //设置套接字超时。默认值为 5 秒。
    .withSocketTimeout(Duration.ofSeconds(3))
    //[可选] 设置默认标头
    .withDefaultHeaders(defaultHeaders)
    //添加基本身份验证
    .withBasicAuth(username, password)
```

```
    //可以指定一个Supplier<Header>函数，每次在请求发送到Elasticsearch之前都会调用该函数
    //例如，当前时间被写入标头中
    .withHeaders(() -> {
        HttpHeaders headers = new HttpHeaders();
        headers.add("currentTime", LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
        return headers;
    })
    . // ... other options
    .build();
```

> 如上例所示添加Header supplier允许注入可能随时间变化的标头，例如身份验证JWT令牌。如果在反应设置中使用它，则supplier功能不得阻塞！

客户端日志

要查看实际发送到服务器和从服务器接收的内容，需要打开传输级别的请求/响应日志记录，如下面的代码段所述。
启用传输层日志记录

```
<logger name="org.springframework.data.elasticsearch.client.WIRE" level="trace"/>
```

spring boot配置

```
logging:
  level:
    org.springframework.data.elasticsearch.client.WIRE: trace
```

> 当通过RestClients分别获得ReactiveRestClients时，上述内容适用于RestHighLevelClient和ReactiveElasticsearchClient，不适用于TransportClient。

8.3 Elasticsearch对象映射

Spring Data Elasticsearch对象映射是将 Java 对象（域实体）映射到存储在 Elasticsearch 中并返回的 JSON 表示的过程。

Spring Data Elasticsearch的早期版本使用基于Jackson的转换，Spring Data Elasticsearch 3.2.x 引入了元模型对象映射。从4.0版开始，仅使用元对象映射(MappingElasticsearchConverter)，不再使用基于Jackson的映射器。
移除基于Jackson的映射器的主要原因是：

- 需要使用@JsonFormat或@JsonInclude等注解来完成字段的自定义映射。当相同的对象用于不同的基于JSON的数据存储或通过基于JSON的API发送时，这通常会导致问题。
- 自定义字段类型和格式也需要存储到Elasticsearch索引映射中。基于Jackson的注解没有完全提供表示 Elasticsearch类型所需的所有信息。
- 字段不仅在从实体转换到实体时必须映射，而且在查询参数、返回数据和其他地方也必须映射。

现在使用MappingElasticsearchConverter涵盖了所有这些情况。

元模型对象映射

基于元模型的方法使用域类型信息来读取/写入Elasticsearch。这允许为特定域类型映射注册Converter实例。

映射注解概述

MappingElasticsearchConverter使用元数据来驱动对象到文档的映射。元数据取自可以注释的实体属性。
可以使用以下注解：

- `@Document`：应用于类级别以指示该类是映射到数据库的候选对象。最重要的属性是：
  - `indexName`：存储此实体的索引的名称。
  - `shards`：索引的分片数。
  - `replicas`：索引的副本数。
  - `refreshIntervall`：索引的刷新间隔。用于创建索引。默认值为“1s”。
  - `indexStoreType`：索引的索引存储类型。用于创建索引。默认值为“fs”。
  - `createIndex`：配置是否在存储库引导时创建索引。默认值为true。
  - `versionType`：版本管理的配置。默认值为*EXTERNAL*。
- `@Id`：应用于字段级别以标记用于标识目的的字段。
- `@Transient`：默认情况下，所有字段在存储或检索时都映射到文档，此注解用于排除该字段。
- `@PersistenceConstructor`：标记一个给定的构造函数——甚至是一个包保护的构造函数——在从数据库中实例化对象时使用。构造函数参数按名称映射到检索到的Document中的键值。
- `@Field`：应用于字段级别并定义字段的属性，大部分属性映射到各自的 Elasticsearch Mapping 定义（以下列表不完整，请查看注释 Javadoc 以获得完整参考）：
  - `name`：将在Elasticsearch文档中表示的字段名称，如果未设置，则使用Java字段名称。
  - `type`：段类型，可以是 *Text, Keyword, Long, Integer, Short, Byte, Double, Float, Half_Float, Scaled_Float, Date, Date_Nanos, Boolean, Binary, Integer_Range, Float_Range, Long_Range, Double_Range, Date_Range, Ip_Range, Object, Nested, Ip, TokenCount, Percolator, Flattened, Search_As_You_Type*. 请参阅Elasticsearch Mapping Types
  - `format` 和 `pattern` Date 类型的定义。 `format` 必须为日期类型定义。
  - `store`：标记原始字段值是否应存储在Elasticsearch中，默认值为 false。
  - `analyzer`，`searchAnalyzer`，`normalizer` 用于指定自定义分析器、搜索分析器和标准化器。
- `@GeoPoint`：将字段标记为 geo_point 数据类型。如果该字段是 GeoPoint 类的实例，则可以省略。

> 从TemporalAccessor派生的属性必须具有FieldType.Date类型的@Field注解，或者必须为此类型注册自定义转换器。
> 如果您使用自定义日期格式，则需要使用uuuu表示年份而不是yyyy。这是由于Elasticsearch7发生了变化。

映射元数据基础设施在一个独立的spring-data-commons项目中定义，该项目与技术无关。

映射规则

类型说明

映射使用嵌入在发送到服务器的文档中的类型说明来允许泛型类型映射。这些类型说明在文档中表示为`_class`属性，并针对每个聚合根写入。

比如

```
public class Person {
    @Id
    String id;
    String firstname;
    String lastname;
}
```

```
{
  "_class" : "com.example.Person",
  "id" : "cb7bef",
  "firstname" : "Sarah",
  "lastname" : "Connor"
}
```

默认情况下，域类型类名用于类型说明。

类型说明可以配置为保存自定义信息。使用@TypeAlias注解来执行此操作。

比如

```
@TypeAlias("human")
public class Person {
    @Id
    String id;
    // ...
}
```

```
{
  "_class" : "human",
  "id" : ...
}
```

> 除非属性类型是对象、接口或实际值类型与属性声明不匹配，否则不会为嵌套对象编写类型说明。

地理空间类型

像Point和GeoPoint这样的地理空间类型被转换为纬度/经度对。

比如

```
public class Address {
    String city, street;
    Point location;
}
```

```
{
  "city" : "Los Angeles",
  "street" : "2800 East Observatory Road",
  "location" : { "lat" : 34.118347, "lon" : -118.3026284 }
}
```

集合

对于集合内的值，在类型说明和自定义转换方面应用与聚合根相同的映射规则。

比如

```
public class Person {
     // ...
    List<Person> friends;
}
```

```
{
  // ...
  "friends" : [ { "firstname" : "Kyle", "lastname" : "Reese" } ]
}
```

Maps

对于Maps中的值，在类型说明和自定义转换方面应用与聚合根相同的映射规则。但是，映射键需要一个字符串才能由Elasticsearch处理。

比如

```
public class Person {
    // ...
    Map<String, Address> knownLocations;
}
```

```
{
  // ...
  "knownLocations" : {
    "arrivedAt" : {
       "city" : "Los Angeles",
       "street" : "2800 East Observatory Road",
       "location" : { "lat" : 34.118347, "lon" : -118.3026284 }
     }
  }
}
```

自定义转化

查看上一节中的配置ElasticsearchCustomConversions允许注册映射域和简单类型的特定规则。

比如

```
@Configuration
public class Config extends AbstractElasticsearchConfiguration {
    @Override
```

```
    public RestHighLevelClient elasticsearchClient() {
        return RestClients.create(ClientConfiguration.create("localhost:9200")).rest();
    }
    @Bean
    @Override
    public ElasticsearchCustomConversions elasticsearchCustomConversions() {
        return new ElasticsearchCustomConversions(
            //添加转换器实现
            Arrays.asList(new AddressToMap(), new MapToAddress()));
    }
    //设置用于将DomainType写入Elasticsearch的转换器
    @WritingConverter
    static class AddressToMap implements Converter<Address, Map<String, Object>> {
        @Override
        public Map<String, Object> convert(Address source) {
            LinkedHashMap<String, Object> target = new LinkedHashMap<>();
            target.put("ciudad", source.getCity());
            // ...
            return target;
        }
    }
    //设置用于从搜索结果中读取DomainType的转换器。
    @ReadingConverter
    static class MapToAddress implements Converter<Map<String, Object>, Address> {
        @Override
        public Address convert(Map<String, Object> source) {
            // ...
            return address;
        }
    }
}
```

```
{
  "ciudad" : "Los Angeles",
  "calle" : "2800 East Observatory Road",
  "localidad" : { "lat" : 34.118347, "lon" : -118.3026284 }
}
```

8.4 Elasticsearch操作

Spring Data Elasticsearch使用多个接口来定义可以针对Elasticsearch索引调用的操作（有关反应式接口的描述，请参阅反应式Elasticsearch操作）。

- `IndexOperations`定义索引级别的操作，例如创建或删除索引。
- `DocumentOperations`定义了基于id存储、更新和检索实体的操作。
- `SearchOperations`定义使用查询搜索多个实体的操作
- `ElasticsearchOperations`结合了`DocumentOperations`和`SearchOperations`接口。

这些接口对应于Elasticsearch API的结构。
接口的默认实现提供：

- 索引管理功能。

- 对域类型的读/写映射支持。
- 丰富的查询和标准api。
- 资源管理和异常翻译。

ElasticsearchTemplate

ElasticsearchTemplate是使用传输客户端的ElasticsearchOperations接口的实现。

使用示例

```
@Configuration
public class TransportClientConfig extends ElasticsearchConfigurationSupport
{
    //设置传输客户端。自 4.0 版起已弃用。
    @Bean
    public Client elasticsearchClient() throws UnknownHostException {
        Settings settings = Settings.builder().put("cluster.name",
"elasticsearch").build();
        TransportClient client = new PreBuiltTransportClient(settings);
        client.addTransportAddress(new
TransportAddress(InetAddress.getByName("127.0.0.1"), 9300));
        return client;
    }
    //创建ElasticsearchTemplate，同时提供名称、elasticsearchOperations 和
elasticsearchTemplate。
    @Bean(name = {"elasticsearchOperations", "elasticsearchTemplate"})
    public ElasticsearchTemplate elasticsearchTemplate() throws
UnknownHostException {
        return new ElasticsearchTemplate(elasticsearchClient());
    }
}
```

ElasticsearchRestTemplate

ElasticsearchRestTemplate是使用高级REST客户端的ElasticsearchOperations接口的实现。

使用示例

```
@Configuration
public class RestClientConfig extends AbstractElasticsearchConfiguration {
    //设置高级 REST 客户端。
    @Override
    public RestHighLevelClient elasticsearchClient() {
        return RestClients.create(ClientConfiguration.localhost()).rest();
    }
    //基类AbstractElasticsearchConfiguration已经提供了elasticsearchTemplate
}
```

使用示例

由于ElasticsearchTemplate和ElasticsearchRestTemplate都实现了ElasticsearchOperations接口，因此使用它们的代码没有什么不同。该示例展示了如何在Spring REST控制器中使用注入的ElasticsearchOperations实例。决定是使用TransportClient还是RestClient，是通过为相应的Bean提供上面显示的配置之一来做出的。
下面示例演示ElasticsearchOperations的使用

```java
@Document(indexName = "marvel")
public class Person {
    private @Id String id;
    private String name;
    private int age;
    // Getter/Setter omitted...
}
```

```java
@RestController
@RequestMapping("/")
public class TestController {
    private  ElasticsearchOperations elasticsearchOperations;
    public TestController(ElasticsearchOperations elasticsearchOperations) {
        //让Spring在构造函数中注入提供的ElasticsearchOperations bean。
        this.elasticsearchOperations = elasticsearchOperations;
    }
    //在Elasticsearch集群中存储一些实体。
    @PostMapping("/person")
    public String save(@RequestBody Person person) {
        IndexQuery indexQuery = new IndexQueryBuilder()
            .withId(person.getId().toString())
            .withObject(person)
            .build();
        String documentId = elasticsearchOperations.index(indexQuery);
        return documentId;
    }
    //通过id检索具有查询的实体。
    @GetMapping("/person/{id}")
    public Person findById(@PathVariable("id")  Long id) {
        Person person = elasticsearchOperations
            .queryForObject(GetQuery.getById(id.toString()), Person.class);
        return person;
    }
}
```

要查看ElasticsearchOperations的全部功能，请参阅API文档。

搜索结果类型

当使用DocumentOperations接口的方法检索文档时，只会返回找到的实体。使用SearchOperations接口的方法进行搜索时，每个实体都可以获得附加信息，例如找到的实体的分数或sortValues。

为了返回此信息，每个实体都包装在SearchHit对象中，该对象包含此实体特定的附加信息。这些SearchHit对象本身在SearchHits对象中返回，该对象还包含有关整个搜索的信息，例如maxScore或请求的聚合。现在可以使用以下类和接口：

`SearchHit<T>`

包含以下信息：

- ID
- 分数
- 排序值

- 突出显示字段
- 检索到的 `<T>` 类型实体

`SearchHits<T>`

包含以下信息：

- 总命中数
- 总关联命中数
- 最高分
- `SearchHit<T>` 对象的列表
- 返回的聚合

`SearchPage<T>`

- 定义一个包含 `SearchHits<T>` 元素的 Spring Data Page，可用于使用存储库方法进行分页访问。

`SearchScrollHits<T>`

- 由 ElasticsearchRestTemplate 中的低级滚动 API 函数返回，它使用 Elasticsearch 滚动 ID 丰富了 `SearchHits<T>`。

`SearchHitsIterator<T>`

- SearchOperations 接口的流函数返回的迭代器。

8.5 Elasticsearch Repositories

本章包括 Elasticsearch存储库实现的详细信息。

首先定义一个实体类

```
@Document(indexName="books")
class Book {
    @Id
    private String id;
    @Field(type = FieldType.Text)
    private String name;
    @Field(type = FieldType.Text)
    private String summary;
    @Field(type = FieldType.Integer)
    private Integer price;
    // getter/setter ...
}
```

查询方法

查询查找策略

Elasticsearch模块支持所有基本查询构建功能，如字符串查询、本机搜索查询、基于条件的查询或从方法名称派生。

声明的查询

从方法名称派生查询并不总是足够的和/或可能导致不可读的方法名称。在这种情况下，可能会使用@Query注解（请参阅使用@Query注解）。

查询创建

通常，Elasticsearch的查询创建机制按照查询方法中的描述工作。以下是Elasticsearch查询方法转换为的简短示例：
从方法名称创建查询

```
interface BookRepository extends Repository<Book, String> {
    List<Book> findByNameAndPrice(String name, Integer price);
}
```

上面的方法名会被翻译成下面的 Elasticsearch json 查询

```
{
    "query": {
        "bool" : {
            "must" : [
                { "query_string" : { "query" : "?", "fields" : [ "name" ] } },
                { "query_string" : { "query" : "?", "fields" : [ "price" ] } }
            ]
        }
    }
}
```

Elasticsearch支持的关键字列表如下所示。

Keyword	Sample	Elasticsearch Query String
`And`	`findByNameAndPrice`	`{ "query" : { "bool" : { "must" : [ { "query_string" : { "query" : "?", "fields" : [ "name" ] } }, { "query_string" : { "query" : "?", "fields" : [ "price" ] } } ] } }}`
`Or`	`findByNameOrPrice`	`{ "query" : { "bool" : { "should" : [ { "query_string" : { "query" : "?", "fields" : [ "name" ] } }, { "query_string" : { "query" : "?", "fields" : [ "price" ] } } ] } }}`

Keyword	Sample	Elasticsearch Query String
`Is`	`findByName`	`{ "query" : { "bool" : { "must" : [ { "query_string" : { "query" : "?", "fields" : [ "name" ] } } ] } }}`
`Not`	`findByNameNot`	`{ "query" : { "bool" : { "must_not" : [ { "query_string" : { "query" : "?", "fields" : [ "name" ] } } ] } }}`
`Between`	`findByPriceBetween`	`{ "query" : { "bool" : { "must" : [ {"range" : {"price" : {"from" : ?, "to" : ?, "include_lower" : true, "include_upper" : true } } } ] } }}`
`LessThan`	`findByPriceLessThan`	`{ "query" : { "bool" : { "must" : [ {"range" : {"price" : {"from" : null, "to" : ?, "include_lower" : true, "include_upper" : false } } } ] } }}`
`LessThanEqual`	`findByPriceLessThanEqual`	`{ "query" : { "bool" : { "must" : [ {"range" : {"price" : {"from" : null, "to" : ?, "include_lower" : true, "include_upper" : true } } } ] } }}`
`GreaterThan`	`findByPriceGreaterThan`	`{ "query" : { "bool" : { "must" : [ {"range" : {"price" : {"from" : ?, "to" : null, "include_lower" : false, "include_upper" : true } } } ] } }}`
`GreaterThanEqual`	`findByPriceGreaterThan`	`{ "query" : { "bool" : { "must" : [ {"range" : {"price" : {"from" : ?, "to" : null, "include_lower" : true, "include_upper" : true } } } ] } }}`
`Before`	`findByPriceBefore`	`{ "query" : { "bool" : { "must" : [ {"range" : {"price" : {"from" : null, "to" : ?, "include_lower" : true, "include_upper" : true } } } ] } }}`

Keyword	Sample	Elasticsearch Query String
`After`	`findByPriceAfter`	`{ "query" : { "bool" : { "must" : [ {"range" : {"price" : {"from" : ?, "to" : null, "include_lower" : true, "include_upper" : true } } } ] } }}`
`Like`	`findByNameLike`	`{ "query" : { "bool" : { "must" : [ { "query_string" : { "query" : "?*", "fields" : [ "name" ] }, "analyze_wildcard": true } ] } }}`
`StartingWith`	`findByNameStartingWith`	`{ "query" : { "bool" : { "must" : [ { "query_string" : { "query" : "?*", "fields" : [ "name" ] }, "analyze_wildcard": true } ] } }}`
`EndingWith`	`findByNameEndingWith`	`{ "query" : { "bool" : { "must" : [ { "query_string" : { "query" : "*?", "fields" : [ "name" ] }, "analyze_wildcard": true } ] } }}`
`Contains/Containing`	`findByNameContaining`	`{ "query" : { "bool" : { "must" : [ { "query_string" : { "query" : "*?*", "fields" : [ "name" ] }, "analyze_wildcard": true } ] } }}`
`In` (当注解为 FieldType.Keyword)	`findByNameIn(Collection<String>names)`	`{ "query" : { "bool" : { "must" : [ {"bool" : {"must" : [ {"terms" : {"name" : ["?","?"]}} ] } } ] } }}`
`In`	`findByNameIn(Collection<String>names)`	`{ "query": {"bool": {"must": [{"query_string": {"query": "\"?\" \"?\"", "fields": ["name"]}}]}}}`
`NotIn` (当注解 FieldType.Keyword)	`findByNameNotIn(Collection<String>names)`	`{ "query" : { "bool" : { "must" : [ {"bool" : {"must_not" : [ {"terms" : {"name" : ["?","?"]}} ] } } ] } }}`
`NotIn`	`findByNameNotIn(Collection<String>names)`	`{"query": {"bool": {"must": [{"query_string": {"query": "NOT(\"?\" \"?\")", "fields": ["name"]}}]}}}`

Keyword	Sample	Elasticsearch Query String
`Near`	`findByStoreNear`	`Not Supported Yet !`
`True`	`findByAvailableTrue`	`{ "query" : { "bool" : { "must" : [ { "query_string" : { "query" : "true", "fields" : [ "available" ] } } ] } }}`
`False`	`findByAvailableFalse`	`{ "query" : { "bool" : { "must" : [ { "query_string" : { "query" : "false", "fields" : [ "available" ] } } ] } }}`
`OrderBy`	`findByAvailableTrueOrderByNameDesc`	`{ "query" : { "bool" : { "must" : [ { "query_string" : { "query" : "true", "fields" : [ "available" ] } } ] } }, "sort":[{"name":{"order":"desc"}}] }`

方法返回类型

可以将存储库方法定义为具有以下返回类型以返回多个元素：

- `List<T>`
- `Stream<T>`
- `SearchHits<T>`
- `List<SearchHit<T>>`
- `Stream<SearchHit<T>>`
- `SearchPage<T>`

使用@Query注解

例如

```
interface BookRepository extends ElasticsearchRepository<Book, String> {
    @Query("{\"match\": {\"name\": {\"query\": \"?0\"}}}")
    Page<Book> findByName(String name,Pageable pageable);
}
```

设置为注解参数的字符串必须是有效的Elasticsearch JSON 查询。它将作为查询元素的值发送到 Easticsearch；例如，如果使用参数 John 调用该函数，它将产生以下查询体：

```
{
  "query": {
    "match": {
      "name": {
        "query": "John"
      }
    }
  }
}
```

存储库方法的注解

@Highlight

存储库方法上的@Highlight注解定义应包含返回的实体突出显示的哪些字段。要在Book的名称或摘要中搜索某些文本并突出显示找到的数据，可以使用以下存储库方法：

```
interface BookRepository extends Repository<Book, String> {
    @Highlight(fields = {
        @HighlightField(name = "name"),
        @HighlightField(name = "summary")
    })
    List<SearchHit<Book>> findByNameOrSummary(String text, String summary);
}
```

可以像上面一样定义多个要突出显示的字段，并且可以使用@HighlightParameters注解进一步自定义@Highlight和@HighlightField注解。检查Javadocs以获得可能的配置选项。
在搜索结果中，可以从 SearchHit 类中检索高亮数据。

基于注解的配置

Spring Data Elasticsearch 存储库支持可以通过 JavaConfig 使用注解激活。
比如

```
@Configuration
//EnableElasticsearchRepositories注解激活存储库支持。
//如果没有配置基本包，它将使用它所放置的配置类之一。
@EnableElasticsearchRepositories(
    basePackages = "org.springframework.data.elasticsearch.repositories"
)
static class Config {
    @Bean
    public ElasticsearchOperations elasticsearchTemplate() {
        // 使用 Elasticsearch Operations 一章中显示的配置之
        // 一提供一个名为 elasticsearchTemplate 的 ElasticsearchOperations 类型的 Bean。
    }
}

class ProductService {
    private ProductRepository repository;
    public ProductService(ProductRepository repository) {
        //让 Spring 将 Repository bean 注入到您的类中。
        this.repository = repository;
    }
    public Page<Product> findAvailableBookByName(String name, Pageable pageable) {
        return repository.findByAvailableTrueAndNameStartingWith(name, pageable);
    }
}
```

8.6 其他

请参考官方文档

8.7 综合示例

导入依赖

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.zeroone.star</groupId>
    <artifactId>project-es</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <properties>
        <java.version>1.8</java.version>
        <spring-boot.version>2.3.12.RELEASE</spring-boot.version>
        <hutool.version>5.8.3</hutool.version>
    </properties>
    <dependencies>
        <!-- web -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- lombok -->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <!-- hutool -->
        <dependency>
            <groupId>cn.hutool</groupId>
            <artifactId>hutool-all</artifactId>
        </dependency>
        <!-- spring data es -->
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-elasticsearch</artifactId>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <!-- spring boot -->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>${spring-boot.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
```

```xml
            <!-- hutool -->
            <dependency>
                <groupId>cn.hutool</groupId>
                <artifactId>hutool-all</artifactId>
                <version>${hutool.version}</version>
                <optional>true</optional>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>${spring-boot.version}</version>
                <executions>
                    <execution>
                        <id>repackage</id>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

修改项目配置文件

```yaml
spring:
  application:
    name: PROJECT-ES
server:
  port: 8080

logging:
  level:
    org.springframework.data.elasticsearch.client.WIRE: trace

es:
  # 格式：host:port,多个集群用,分割
  hosts: 192.168.220.128:9200
  # 超时配置
  timeout:
    # 连接超时
    connect: 10
    # 套接字超时
    socket: 5
```

创建ES配置类

```
/**
 * @Description ES配置
 * @Author 阿伟学长
 * @Copy &copy;01星球
 * @Address 01星球总部
 */
@Configuration
public class RestEsClientConfig extends AbstractElasticsearchConfiguration {
    @Value("${es.hosts}")
    private String[] hosts;
    @Value("${es.timeout.connect:10}")
    private long connTimeout;
    @Value("${es.timeout.socket:5}")
    private long socketTimeout;

    @Bean
    @Override
    public RestHighLevelClient elasticsearchClient() {
        //使用构建器提供集群地址、设置默认HttpHeaders或启用SSL
        ClientConfiguration clientConfiguration =
ClientConfiguration.builder()
                .connectedTo(hosts)
                .withConnectTimeout(Duration.ofSeconds(connTimeout))
                .withSocketTimeout(Duration.ofSeconds(socketTimeout))
                .build();
        //创建RestHighLevelClient
        return RestClients.create(clientConfiguration).rest();
    }
}
```

编写实体类

```
/**
 * @Description 定义Book实体类，并与Elasticsearch对象映射
 * @Author 阿伟学长
 * @Copy &copy;01星球
 * @Address 01星球总部
 */
@Data
@Document(indexName = "book")
public class Book {
    @Id
    @Field(type = FieldType.Text)
    private String id;
    @Field(analyzer = "ik_max_word")
    private String title;
    @Field(analyzer = "ik_max_word")
    private String author;
    @Field(type = FieldType.Double)
    private Double price;
    @Field(type = FieldType.Date, format = DateFormat.custom, pattern =
"yyyy-MM-dd")
```

```
    private Date createTime;
    @Field(type = FieldType.Date, format = DateFormat.custom, pattern = "yyyy-MM-dd")
    private Date updateTime;
}
```

编写ES存储操作

```
/**
 * @Description ES数据存储操作
 * @Author 阿伟学长
 * @Copy &copy;01星球
 * @Address 01星球总部
 */
public interface EsBookRepository extends ElasticsearchRepository<Book, String> {
    /**
     * 通过标题和作者查询书籍
     * @param title  标题
     * @param author 作者
     * @return 查询结果
     */
    List<Book> findByTitleOrAuthor(String title, String author);
    /**
     * 通过关键字查找书籍
     * @param keyword 关键字查找
     * @return 查询结果
     */
    @Highlight(fields = {
            @HighlightField(name = "title"),
            @HighlightField(name = "author")
    })
    @Query("{\"match\":{\"title\":\"?0\"}}")
    SearchHits<Book> find(String keyword);
}
```

定义服务接口与实现

```
/**
 * @Description 图书服务接口
 * @Author 阿伟学长
 * @Copy &copy;01星球
 * @Address 01星球总部
 */
public interface IBookService {
    /**
     * 保存图书
     * @param book 图书对象
     * @return 保存成功返回true
     */
    boolean saveBook(Book book);
    /**
     * 通过标题关键词查询书籍
     * @param keywords 关键词
```

```
     * @return
     */
    SearchHits<Book> findByTitle(String keywords);
    /**
     * 通过标题或作者关键词查询
     * @param keywords 关键词
     * @return
     */
    List<Book> findByTitleOrAuthor(String keywords);
}
```

```
/**
 * @Description 图书服务实现
 * @Author 阿伟学长
 * @Copy &copy;01星球
 * @Address 01星球总部
 */
@Slf4j
@Service
public class BookServiceImpl implements IBookService {
    @Resource
    EsBookRepository esBookRepository;
    @Override
    public boolean saveBook(Book book) {
        try {
            esBookRepository.save(book);
            return true;
        } catch (Exception e) {
            log.error(String.format("保存ES错误！%s", e.getMessage()));
        }
        return false;
    }
    @Override
    public SearchHits<Book> findByTitle(String keywords) {
        return esBookRepository.find(keywords);
    }
    @Override
    public List<Book> findByTitleOrAuthor(String keywords) {
        return esBookRepository.findByTitleOrAuthor(keywords, keywords);
    }
}
```

编写测试控制器

```
/**
 * @Description 测试请求类
 * @Author 阿伟学长
 * @Copy &copy;01星球
 * @Address 01星球总部
 */
@RestController
@RequestMapping("/book")
public class TestController {
    @Resource
```

```
    IBookService bookService;

    @PostMapping("/add")
    public Map<String,String> addBook(@RequestBody Book book){
        bookService.saveBook(book);
        Map<String,String> map = new HashMap<>(1);
        map.put("msg","ok");
        return map;
    }

    @GetMapping("/search1")
    public SearchHits<Book> search1(String key){
        return bookService.findByTitle(key);
    }

    @GetMapping("/search2")
    public List<Book> search2(String key){
        return bookService.findByTitleOrAuthor(key);
    }
}
```

请求测试

下面是IDEA测试请求工具脚本

```
### 测试添加书籍1
POST http://localhost:8080/book/add
Content-Type: application/json

{
  "id": 1,
  "title": "《微服务设计》",
  "author": "Sam Newman",
  "price": 69,
  "createTime":"2022-09-18"
}

### 测试添加书籍2
POST http://localhost:8080/book/add
Content-Type: application/json

{
  "id": 2,
  "title": "《微服务架构设计模式》",
  "author": "Chris Richardson",
  "price": 139,
  "createTime":"2022-09-18"
}

### 测试添加书籍3
POST http://localhost:8080/book/add
Content-Type: application/json

{
  "id": 3,
```

```
  "title": "《Microservices Recipes》",
  "author": "Eberhard Wolff",
  "price": 0,
  "createTime":"2022-09-18"
}

### 测试添加书籍4
POST http://localhost:8080/book/add
Content-Type: application/json

{
  "id": 4,
  "title": "《实现领域驱动设计》",
  "author": "Eberhard Wolff",
  "price": 99,
  "createTime":"2022-09-18"
}

### 测试添加书籍5
POST http://localhost:8080/book/add
Content-Type: application/json

{
  "id": 5,
  "title": "《数据密集型应用系统设计》",
  "author": "Eberhard Wolff",
  "price": 128,
  "createTime":"2022-09-18"
}

### 测试搜索1
GET http://localhost:8080/book/search1?key=设计

### 测试搜索2
GET http://localhost:8080/book/search2?key=Wolff
```

通过测试关系效果即可