

Table of Contents

Introduction	1.1
什么是Elasticsearch	1.2
存入数据：文档与索引	1.2.1
取出信息：搜索与分析	1.2.2
扩展性与弹性	1.2.3
版本的新特性	1.3
开始使用Elasticsearch	1.4
配置Elasticsearch	1.5
Installing Elasticsearch	1.5.1
Configuring Elasticsearch	1.5.2
重要 Elasticsearch 配置	1.5.3
Important System Configuration	1.5.4
Bootstrap Checks	1.5.5
Bootstrap Checks for X-Pack	1.5.6
Starting Elasticsearch	1.5.7
Stopping Elasticsearch	1.5.8
Discovery and cluster formation	1.5.9
在集群中添加或移除节点	1.5.10
Full-cluster restart and rolling restart	1.5.11
Remote clusters	1.5.12
Set up X-Pack	1.5.13
Configuring X-Pack Java Clients	1.5.14
Plugins	1.5.15
升级Elasticsearch	1.6
索引模块	1.7
映射	1.8
动态映射(Dynamic mapping)	1.8.1
分词器详解	1.8.1.1
索引搜索分词	1.8.1.2
显式映射(Explicit mapping)	1.8.2
显式映射(Explicit mapping)	1.8.3
显式映射(Explicit mapping)	1.8.4
显式映射(Explicit mapping)	1.8.5
显式映射(Explicit mapping)	1.8.6
映射限制设置(Mapping limit settings)	1.8.7

显式映射(Explicit mapping)	1.8.8
文本分析	1.9
概述	1.9.1
概念	1.9.2
分词器详解	1.9.2.1
索引搜索分词	1.9.2.2
词干分析	1.9.2.3
标记图	1.9.2.4
配置文本分析	1.9.3
测试分词器	1.9.3.1
配置内置分词器	1.9.3.2
创建自定义分词器	1.9.3.3
指定分词器	1.9.3.4
内置分词器参考	1.9.4
标记化器参考	1.9.5
标记过滤器参考	1.9.6
字符过滤器参考	1.9.7
规范化器	1.9.8
索引模板	1.10
模拟多组件模板	1.10.1
数据流	1.11
预处理节点	1.12
搜索数据	1.13
折叠搜索结果	1.13.1
筛选搜索结果	1.13.2
高亮显示	1.13.3
长时间搜索(x-pack)	1.13.4
准即时搜索	1.13.5
搜索结果分页	1.13.6
Query DSL	1.14
查询筛选上下文	1.14.1
复合查询	1.14.2
布尔查询	1.14.2.1
增强查询	1.14.2.2
词干分析	1.14.2.3
标记图	1.14.2.4
标记图	1.14.2.5
全文本查询	1.14.3

间隔	1.14.3.1
匹配	1.14.3.2
匹配布尔前缀	1.14.3.3
匹配词组	1.14.3.4
标记图	1.14.3.5
地理查询	1.14.4
图形查询(x-pack)	1.14.5
连接查询	1.14.6
匹配全部查询	1.14.7
范围查询	1.14.8
特定查询	1.14.9
Distance feature	1.14.9.1
More like this	1.14.9.2
Percolate	1.14.9.3
Rank feature	1.14.9.4
Script	1.14.9.5
Script score	1.14.9.6
Wrapper	1.14.9.7
Pinned Query	1.14.9.8
术语级别查询	1.14.10
最小匹配查询	1.14.11
多术语重写查询	1.14.12
正则对称查询	1.14.13
聚合	1.15
EQL	1.16
SQL 接入	1.17
脚本	1.18
数据管理(x-pack)	1.19
ILM 管理索引生命周期(x-pack)	1.20
自动扩容(x-pack)	1.21
监控集群(x-pack)	1.22
冻结索引(x-pack)	1.23
汇总或转换数据	1.24
配置高可用集群	1.25
快照与恢复	1.26
集群安全	1.27
监视集群和索引事件(x-pack)	1.28
命令行工具	1.29

如何	1.30
通用建议	1.30.1
食谱	1.30.2
混合精确搜索和词干分析	1.30.2.1
获得一致的分数	1.30.2.2
将静态相关信号纳入分数	1.30.2.3
词语表	1.31
REST APIs	1.32
API conventions	1.32.1
Autoscaling APIs	1.32.2
cat APIs	1.32.3
Cluster APIs	1.32.4
Cross-cluster replication APIs	1.32.5
Data stream APIs(x-pack)	1.32.6
Document APIs	1.32.7
Reading and Writing documents	1.32.7.1
Index	1.32.7.2
Get	1.32.7.3
Delete	1.32.7.4
Delete by query	1.32.7.5
Update	1.32.7.6
Update by query	1.32.7.7
Multi get	1.32.7.8
Bulk	1.32.7.9
Reindex(中文)	1.32.7.10
Term vectors	1.32.7.11
Multi term vectors	1.32.7.12
?refresh	1.32.7.13
Optimistic concurrency control	1.32.7.14
Enrich APIs	1.32.8
Graph explore API(x-pack)	1.32.9
Find structure API(x-pack)	1.32.10
Index APIs	1.32.11
Add index alias	1.32.11.1
Analyze	1.32.11.2
Clear cache	1.32.11.3
Clone index	1.32.11.4
Close index	1.32.11.5

Create index	1.32.11.6
Delete index	1.32.11.7
Delete index alias	1.32.11.8
Delete component template	1.32.11.9
Delete index template	1.32.11.10
Delete index template (legacy)	1.32.11.11
Flush	1.32.11.12
Force merge	1.32.11.13
Freeze index	1.32.11.14
Get component template	1.32.11.15
Get field mapping	1.32.11.16
Get index	1.32.11.17
Get index alias	1.32.11.18
Get index settings	1.32.11.19
Get index template	1.32.11.20
Get index template (legacy)	1.32.11.21
Get mapping	1.32.11.22
Index alias exists	1.32.11.23
Index exists	1.32.11.24
Index recovery	1.32.11.25
Index segments	1.32.11.26
Index shard stores	1.32.11.27
Index stats	1.32.11.28
Index template exists (legacy)	1.32.11.29
Open index	1.32.11.30
Put index template(中文)	1.32.11.31
Put index template (legacy)	1.32.11.32
Put component template	1.32.11.33
Put mapping(中文)	1.32.11.34
Refresh	1.32.11.35
Rollover index	1.32.11.36
Shrink index	1.32.11.37
Simulate index	1.32.11.38
Simulate template	1.32.11.39
Split index	1.32.11.40
Synced flush	1.32.11.41
Type exists	1.32.11.42
Unfreeze index	1.32.11.43

Update index alias	1.32.11.44
Update index settings	1.32.11.45
Resolve index	1.32.11.46
List dangling indices	1.32.11.47
Import dangling index	1.32.11.48
Delete dangling index	1.32.11.49
Index lifecycle management APIs	1.32.12
Ingest APIs	1.32.13
Info API(x-pack)	1.32.14
Licensing APIs(x-pack)	1.32.15
Logstash APIs(x-pack)	1.32.16
Machine learning anomaly detection APIs(x-pack)	1.32.17
Machine learning data frame analytics APIs(x-pack)	1.32.18
Migration APIs(x-pack)	1.32.19
Reload search analyzers API(x-pack)	1.32.20
Repositories metering APIs(x-pack)	1.32.21
Rollup APIs(x-pack)	1.32.22
Search APIs	1.32.23
Searchable snapshots APIs(x-pack)	1.32.24
Security APIs(x-pack)	1.32.25
Snapshot and restore APIs(中文)	1.32.26
Snapshot lifecycle management APIs(x-pack)	1.32.27
Transform APIs(x-pack)	1.32.28
Usage API(x-pack)	1.32.29
Watcher APIs(x-pack)	1.32.30
版本迁移指导	1.33
发版记录	1.34
依赖与版本	1.35

Introduction

什么是Elasticsearch

你知道，为了搜索（和分析）

Elasticsearch是位于Elastic Stack核心的分布式搜索和分析引擎。Logstash和Beats有助于收集、聚合和丰富数据，并将其存储在Elasticsearch中。Kibana使您能够交互式地探索、可视化和共享对数据的见解，并管理和监视堆栈。Elasticsearch是索引、搜索和分析的神奇之处。

Elasticsearch为所有类型的数据提供近乎实时的搜索和分析。无论您拥有结构化或非结构化文本、数字数据或地理空间数据，Elasticsearch都可以以支持快速搜索的方式高效地存储和索引这些数据。您可以远远超越简单的数据检索和聚合信息来发现数据中的趋势和模式。而且，随着数据和查询量的增长，Elasticsearch的分布式特性使您的部署能够随着它无缝地增长。

虽然并非每个问题都是搜索问题，但Elasticsearch提供了在各种用例中处理数据的速度和灵活性：

- 向应用程序或网站添加搜索框
- 存储和分析日志、度量和安全事件数据
- 使用机器学习自动实时对数据行为进行建模
- 使用Elasticsearch作为存储引擎自动化业务工作流
- 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存储和处理遗传数据

我们一直对人们使用搜索的新奇方式感到惊讶。但是，无论您的用例与其中一个相似，还是您正在使用Elasticsearch来解决一个新问题，您在Elasticsearch中处理数据、文档和索引的方式都是相同的。

数据输入：文档和索引

Elasticsearch是一个分布式文档存储。Elasticsearch不将信息存储为列数据行，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，存储的文档会分布在整个群集中，并且可以从任何节点立即访问。

当一个文档被存储时，它会被编入索引，并且几乎可以在1秒内准实时地进行完全搜索。Elasticsearch使用一种称为倒排索引的数据结构，支持非常快速的全文搜索。倒排索引列出任何文档中出现的每个唯一单词，并标识每个单词出现的所有文档。

索引可以看作是文档的优化集合，每个文档都是字段的集合，字段是包含数据的键值对。默认情况下，Elasticsearch索引每个字段中的所有数据，每个索引字段都有一个专用的优化数据结构。例如，文本字段存储在倒排索引中，数字和地理字段存储在BKD树中。使用每个字段的数据结构来组合和返回搜索结果的能力是Elasticsearch如此快速的原因。

Elasticsearch还具有无模式的能力，这意味着可以对文档进行索引，而无需明确指定如何处理文档中可能出现的每个不同字段。启用动态映射后，Elasticsearch会自动检测并向索引中添加新字段。此默认行为使索引和浏览数据变得容易—只需开始索引文档，Elasticsearch就会检测布尔值、浮点值和整数值、日期和字符串，并将其映射到适当的Elasticsearch数据类型。

不过，最终，您比Elasticsearch更了解您的数据以及如何使用这些数据。您可以定义控制动态映射的规则，并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
- 执行特定语言的文本分析
- 为部分匹配优化字段
- 使用自定义日期格式
- 使用无法自动检测的数据类型，如 `geo_point` 和 `geo_shape`

为不同的目的以不同的方式索引同一字段通常是有用的。例如，您可能希望将字符串字段索引为全文搜索的文本字段和排序或聚合数据的关键字字段。或者，您可以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。

索引期间应用于全文字段的分析链也在搜索时使用。当您查询全文字段时，在索引中查找术语之前，查询文本将经历相同的分析。

信息输出：搜索和分析

虽然您可以将Elasticsearch用作文档存储和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构建在Apache Lucene搜索引擎库上的全套搜索功能。

Elasticsearch提供了一个简单、一致的REST API，用于管理集群、索引和搜索数据。出于测试目的，您可以轻松地直接从命令行或通过Kibana中的开发人员控制台提交请求。从您的应用程序中，您可以使用[Elasticsearch客户端](#)来选择您的语言：Java、JavaScript、Go、.NET、PHP、Perl、Python或Ruby。

搜索数据

Elasticsearch REST API支持结构化查询、全文查询和将两者结合起来的复杂查询。结构化查询类似于可以在SQL中构造的查询类型。例如，您可以搜索 employee 索引中的 gender 和 age 字段，并按 hire_date 字段对匹配项进行排序。全文查询将查找与查询字符串匹配的所有文档，并返回按相关性排序的文档它们与搜索词的匹配程度如何。

除了搜索单个术语外，还可以执行短语搜索、相似性搜索和前缀搜索，并获取自动完成建议。

是否有要搜索的地理空间或其他数字数据？Elasticsearch支持在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的综合JSON风格查询语言（[Query DSL](#)）访问所有这些搜索功能。您还可以构造[SQL-style queries](#)来搜索和聚合Elasticsearch内部的本机数据，JDBC和ODBC驱动程序使各种第三方应用程序能够通过SQL与Elasticsearch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。聚合让您能够回答以下问题，而不仅仅是找到众所周知的“干草堆里捞针”：

- 干草堆里针有多少根？
- 针的平均长度是多少？
- 按制造商细分的针的平均长度是多少？
- 在过去的六个月里，每一个干草堆里加了多少针？

您还可以使用聚合来回答更微妙的问题，例如：

- 你们最受欢迎的针制造商是什么？
- 是否有异常或异常的针束？

因为聚合利用了用于搜索的相同数据结构，所以它们的速度也非常快。这使您能够实时分析和可视化数据。您的报告和仪表盘会随着数据的更改而更新，以便您可以根据最新信息采取行动。

此外，聚合与搜索请求一起运行。您可以搜索文档，过滤结果，并在同一时间，对同一数据，在一个单一的请求执行分析。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号针的计数，还显示了与用户搜索条件匹配的70号针

的计数—例如，所有70号不粘绣花针。

但是等等，还有更多

想自动分析时间序列数据吗？您可以使用[机器学习](#)功能创建数据中正常行为的准确基线，并识别异常模式。通过机器学习，您可以检测：

- 与值、计数或频率的时间偏差有关的异常
- 统计稀有性
- 群体成员的不寻常行为

最好的部分呢？您可以这样做，而不必指定算法、模型或其他与数据科学相关的配置。

可伸缩性和弹性：集群、节点和分片

Elasticsearch始终可用，并可根据您的需要进行扩展。它是通过自然分配来实现的。您可以将服务器（节点）添加到集群以增加容量，Elasticsearch会自动将数据和查询负载分布到所有可用节点。无需大修应用程序，Elasticsearch知道如何平衡多节点群集以提供规模和高可用性。节点越多越好。

这是怎么回事？实际上，Elasticsearch索引只是一个或多个物理分片的逻辑分组，其中每个分片实际上是一个自包含索引。通过将索引中的文档分布在多个分片上，并将这些分片分布在多个节点上，Elasticsearch可以确保冗余，这既可以防止硬件故障，又可以在节点添加到集群时增加查询容量。随着集群的增长（或收缩），Elasticsearch会自动迁移分片以重新平衡集群。

分片有两种类型：主碎片和副本分片。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。副本提供了数据的冗余副本，以防止硬件故障，并增加了服务于读取请求（如搜索或检索文档）的容量。

索引中主分片的数量在创建索引时是固定的，但副本分片的数量可以随时更改，而不中断索引或查询操作。

这取决于

对于分片大小和为索引配置的主分片的数量，有许多性能方面的考虑和权衡。分片越多，维护这些索引的开销就越大。当Elasticsearch需要重新平衡集群时，分片大小越大，移动分片的时间就越长。

查询大量的小分片可以加快每个分片的处理速度，但是查询越多意味着开销就越大，因此查询较少数量的较大分片可能会更快。简言之...视情况而定。

作为起点：

- 目的将平均分片大小保持在几GB到几十GB之间。对于使用基于时间的数据的用例，通常可以看到20GB到40GB范围内的分片。
- 避免无数分片的问题。节点可以容纳的碎片数量与可用堆空间成比例。一般来说，每GB堆空间的碎片数应小于20。

为您的用例确定最佳配置的最佳方法是[使用您自己的数据和查询进行测试](#)。

万一发生灾难

出于性能原因，集群中的节点需要在同一网络上。在不同数据中心的节点之间平衡集群中的碎片需要很长时间。但是高可用性架构要求您避免将所有的鸡蛋放在一个篮子里。在一个位置发生重大停机的情况下，另一个位置的服务器需要能够接管。天衣无缝。答案是什么？跨群集复制（CCR）。

CCR提供了一种将索引从主集群自动同步到可以用作热备份的辅助远程集群的方法。如果主集群出现故障，则辅助集群可以接管。您还可以使用CCR创建辅助集群，以便在地理位置接近您的用户的情况下为读取请求提供服务。

跨群集复制是主动-被动的。主集群上的索引是活动的领导索引，处理所有写请求。复制到辅助群集的索引是只读的跟随者。

护理和保养

与任何企业系统一样，您需要工具来保护、管理和监控您的Elasticsearch集群。集成到Elasticsearch中的安全、监视和管理功能使您能够将[Kibana](#)用作管理集群的控制中心。[data rollups](#)和[index lifecycle management](#)等功能可帮助您随着时间的推移智能地管理数据。

7.9版本的新特性

开始使用Elasticsearch

配置Elasticsearch

重要Elasticsearch配置

Elasticsearch只需要很少的配置就可以开始使用，但是在生产中使用集群之前，必须考虑以下几点：

- 路径设置
- 群集名称设置
- 节点名称设置
- 网络主机设置
- 发现设置
- 堆大小设置
- JVM堆转储路径设置
- GC日志记录设置
- 临时目录设置
- JVM致命错误日志设置
- 群集备份

我们的[Elastic云](#)服务自动配置这些项目，使您的集群在默认情况下准备就绪。

路径设置

对于macOS `.tar.gz`，Linux `.tar.gz`，以及Windows `.zip` 安装，默认情况下，Elasticsearch将数据和日志写入 `$ES_HOME` 的相应 `data` 和 `log` 志子目录。但是，`$ES_HOME` 中的文件在升级过程中可能会被删除。

在生产中，我们强烈建议您设置`elasticsearch.yml`中的 `path.data` 以及 `path.logs`，把它们配置到去 `$ES_HOME` 以外的地方。

默认情况下，Docker、Debian、RPM、macOS Homebrew和Windows `.msi` 安装会将数据和日志写入 `$ES_HOME` 以外的位置。

支持的 `path.data` 以及 `path.logs` 的值因平台而异：

```
path:
  data: /var/data/elasticsearch
  logs: /var/log/elasticsearch
```

如果需要，可以在 `path.data` 中指定多个路径。Elasticsearch跨所有提供的路径存储节点的数据，但将每个分片的数据保持在同一路径上。

Elasticsearch不会在节点的数据路径上平衡碎片。单个路径中的高磁盘使用率可能会触发整个节点的高磁盘使用率水印。如果触发，Elasticsearch将不会向节点添加分片，即使节点的其他路径具有可用磁盘空间。如果您需要额外的磁盘空间，我们建议您添加一个新节点，而不是额外的数据路径。

```
path:
  data:
    - /mnt/elasticsearch_1
    - /mnt/elasticsearch_2
    - /mnt/elasticsearch_3
```

集群名称设置

节点只能在共享 `cluster.name` 与群集中的所有其他节点时加入群集。默认名称是 `elasticsearch`，但您应该将其更改为描述集群用途的适当名称。

```
cluster.name: logging-prod
```

不要在不同的环境中重用相同的集群名称。否则，节点可能会加入错误的集群。

节点名称设置

Elasticsearch使用 `node.name` 作为Elasticsearch特定实例的可读标识符。这个名称包含在许多API的响应中。节点名默认为Elasticsearch启动时机器的主机名，但可以在 `elasticsearch.yml` 中显式配置：

```
node.name: prod-data-2
```

网络主机设置

默认情况下，Elasticsearch仅绑定到环回地址，如 `127.0.0.1` 和 `[::1]`。这足以在单个服务器上运行一个或多个节点的集群进行开发和测试，但弹性生产集群必须包含其他服务器上的节点。有许多[网络设置](#)，但通常您需要配置的是 `network.host`：

```
network.host: 192.168.1.10
```

当您为 `network.host` 提供值时，Elasticsearch假设您正在从开发模式切换到生产模式，并将大量系统启动检查从警告升级为异常。看看[开发模式和生产模式](#)的区别。

发现和集群组成设置

在投入生产之前需要配置两个重要的发现和集群组成的设置，以便集群中的节点可以相互发现并选举Master节点。

`discovery.seed_hosts`

开箱即用，无需任何网络配置，Elasticsearch将绑定到可用的环回地址，并扫描本地端口 `9300` 到 `9305`，以便与同一服务器上运行的其他节点连接。这种行为提供了一种无需进行任何配置的自动集群体验。

如果要与其他主机上的节点组成集群，请使用[静态](#) `discovery.seed_hosts` 设置。此设置提供群集中其他Master候选且可能处于活动状态且可联系的节点的列表，以作为[发现过程](#)的种子。此设置接受群集中所有Master候选节点的 YAML 序列或地址数组。每个地址可以是IP地址，也可以是通过DNS解析为一个或多个IP地址的主机名。

```
discovery.seed_hosts:
  - 192.168.1.10:9300
  - 192.168.1.11 ①
  - seeds.mydomain.com ②
  - [0:0:0:0:ffff:c0a8:10c]:9301 ③
```

①端口是可选的，默認為 9300，但可以重寫。

②如果主機名解析為多個IP地址，則節點將嘗試在所有解析的地址上發現其他節點。

③IPv6地址必須用方括號括起來。

如果Master候選節點沒有固定的名稱或地址，請使用[備用主機提供程序](#)動態查找其地址。

cluster.initial_master_nodes

當您第一次啟動Elasticsearch群集時，[群集引導](#)步驟將確定在第一次選舉中計算其投票的Master候選節點集。在開發模式下，在沒有配置發現設置的情況下，此步驟由節點自己自動執行。

因為自動引導是[本質不安全](#)的，所以在生產模式下啟動新群集時，必須顯式列出在第一次選舉中應計算其投票的Master候選節點。您可以使
用 `cluster.initial_master_node` 設置。

在群集第一次成功形成之後，從每個節點的配置移除 `cluster.initial_master_nodes` 設置。重新啟動群集或將新節點添加到現有群集時不要使用此設置。

```
discovery.seed_hosts:
  - 192.168.1.10:9300
  - 192.168.1.11
  - seeds.mydomain.com
  - [0:0:0:0:ffff:c0a8:10c]:9301
cluster.initial_master_nodes: ①
  - master-node-a
  - master-node-b
  - master-node-c
```

①通過它們的屬性 `node.name` 标識初始主節點，默認為其主機名。確保 `cluster.initial_master_nodes` 與 `node.name` 一致。如果使用類似 `master-node-a.example.com` 這樣的完全限定域名（FQDN）作為節點名，則必須使用此列表中的FQDN。相反，如果 `node.name` 是沒有任何尾隨限定符的裸主機名，您還必須在 `cluster.initial_master_nodes` 中省略尾隨限定符。

請參閱[群集創建指引](#)以及[發現和群集形成設置](#)。

堆大小設置

默認情況下，Elasticsearch根據節點的角色和總內存自動調整JVM堆的大小。對於大多數生產環境，我們建議使用此默認大小調整。如果需要，可以通過手動設置JVM堆大小來覆蓋默認大小。

自动调整堆大小需要绑定的JDK，如果使用自定义JRE位置，则需要java14或更高版本的JRE。

在容器（如Docker）中运行时，总内存定义为容器可见的内存量，而不是主机上的总系统内存。

JVM堆转储路径设置

默认情况下，Elasticsearch将JVM配置为在出现内存不足异常时将堆转储到默认数据目录。在RPM和Debian包上，数据目录是/var/lib/elasticsearch。在Linux、MacOS和Windows发行版上，数据目录位于Elasticsearch安装的根目录下。

如果此路径不适合接收堆转储，请修改-XX:HeapDumpPath=。。。进入jvm.options选项：

如果指定一个目录，JVM将根据运行实例的PID为堆转储生成一个文件名。

如果指定固定文件名而不是目录，则当JVM需要对内存不足异常执行堆转储时，该文件必须不存在。否则，堆转储将失败。

GC日志设置

默认情况下，Elasticsearch启用垃圾收集（GC）日志。这些在中配置jvm.options选项并输出到与Elasticsearch日志相同的默认位置。默认配置每64 MB循环一次日志，最多可占用2 GB的磁盘空间。

您可以使用jep 158：统一JVM日志记录中描述的命令行选项重新配置JVM日志记录。除非更改默认值jvm.options选项文件，除您自己的设置外，还应用Elasticsearch默认配置。要禁用默认配置，首先通过提供-Xlog:禁用选项，然后提供自己的命令行选项。这将禁用所有JVM日志记录，因此请确保查看可用选项并启用所需的一切。

要查看原始JEP中未包含的其他选项，请参阅使用JVM统一日志框架启用日志记录。

举例说明

将默认GC日志输出位置更改为/opt/my app/gc.log日志通过创建\$ES\user\HOME\config/jvm.options选项博士/gc选项有一些示例选项：

临时目录设置

默认情况下，Elasticsearch使用一个私有临时目录，启动脚本在系统临时目录的正下方创建该目录。

在某些Linux发行版上，如果文件和目录最近未被访问，系统实用程序将从/tmp中清除它们。如果长时间不使用需要临时目录的功能，则此行为可能导致运行Elasticsearch时删除专用临时目录。如果随后使用需要此目录的功能，则删除专用临时目录会导致问题。

如果使用 `.deb` 或 `.rpm` 包安装 Elasticsearch 并在 `systemd` 下运行，则 Elasticsearch 使用的私有临时目录将从定期清理中排除。

如果你想经营公司的话 `.tar.gz` 如果要在 Linux 或 MacOS 上长期发布，请考虑为 Elasticsearch 创建一个专用的临时目录，该目录不在清除旧文件和目录的路径下。此目录应设置权限，以便只有运行 Elasticsearch 的用户才能访问它。然后，在启动 Elasticsearch 之前，将 `$ES_TMPDIR` 环境变量设置为指向此目录。

JVM致命错误日志设置

默认情况下，Elasticsearch 将 JVM 配置为将致命错误日志写入默认日志目录。在 RPM 和 Debian 软件包上，这个目录是 `/var/log/elasticsearch`。在 Linux、MacOS 和 Windows 发行版上，日志目录位于 Elasticsearch 安装的根目录下。

这些是 JVM 遇到致命错误（如分段错误）时生成的日志。如果此路径不适合接收日志，请修改 `-XX:ErrorFile=...` 进入 `Jvm.options` 选项。

群集备份

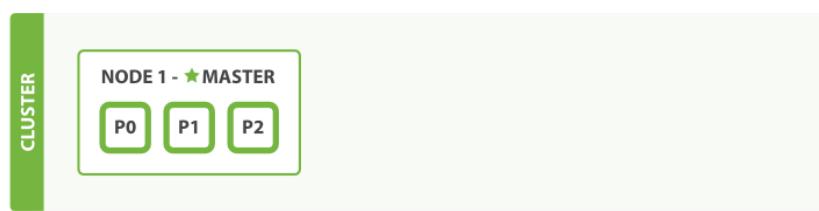
发生灾难时，快照可以防止永久性数据丢失。快照生命周期管理是对集群进行定期备份的最简单方法。有关详细信息，请参阅 [备份群集](#)。

备份集群的唯一可靠且受支持的方法是使用快照。您不能通过复制 Elasticsearch 群集节点的数据目录来备份该群集。不支持从文件系统级别备份还原任何数据的方法。如果您尝试从这样的备份中恢复群集，它可能会失败，并报告文件损坏、文件丢失或其他数据不一致，或者它似乎成功地悄悄丢失了一些数据。

在集群中添加和删除节点

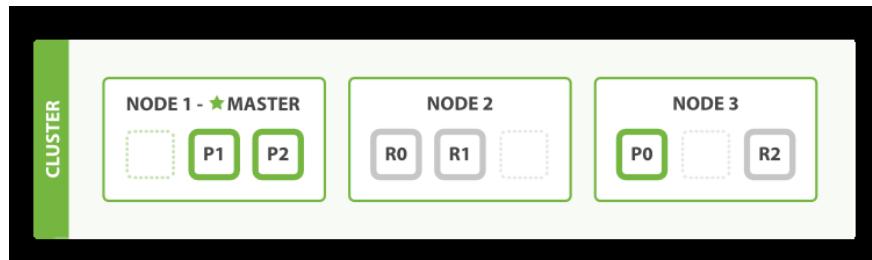
启动Elasticsearch实例时，即启动节点。Elasticsearch集群是一组具有相同 `cluster.name` 属性的节点。当节点加入或离开集群时，集群会自动重新组织自身，以便在可用节点之间均匀地分布数据。

如果您运行的是Elasticsearch的单个实例，那么您就拥有一个节点的集群。所有主分片都驻留在单个节点上。无法分配任何副本分片，因此群集状态保持为黄色。集群功能齐全，但一旦发生故障，可能会丢失数据。



将节点添加到集群以增加其容量和可靠性。默认情况下，节点既是数据节点，又有资格被选为控制集群的主节点。还可以为特定目的配置新节点，例如处理摄取请求。有关详细信息，请参见[节点](#)。

当您向集群添加更多节点时，它会自动分配副本分片。当所有主分片和副本分片都处于活动状态时，群集状态变为绿色。



您可以在本地计算机上运行多个节点，以试验多个节点组成的Elasticsearch集群的行为。要将节点添加到本地计算机上运行的群集，请执行以下操作：

1. 设置新的Elasticsearch实例。
2. 在 `Elasticsearch.yml` 中指定 `cluster.name` 来设置集群的名称。例如，要向 `logging-prod` 集群添加节点，请添加一行 `cluster.name: "logging prod"` 到 `Elasticsearch.yml`。
3. 启动Elasticsearch。节点自动发现并加入指定的群集。

要将节点添加到在多台计算机上运行的群集，还必须设置 `discovery.seed_hosts` 以便新节点可以发现其集群的其余部分。

有关发现和分片分配的更多信息，请参阅[发现和群集组成](#)以及[群集级别的分片分配和路由设置](#)。

Master候选节点

当添加或删除节点时，Elasticsearch通过自动更新群集的投票配置来保持最佳的容错级别，投票配置是一组Master候选节点，在做出选择新主机或提交新群集状态等决策时，这些节点的响应将被计数。

建议在一个集群中有少量固定数量的Master候选节点，并且只通过添加和删除非Master候选节点来上下扩展集群。然而，在某些情况下，可能需要向集群添加或从集群中移除一些Master候选节点。

添加Master候选节点

如果您希望将一些节点添加到集群中，简单配置新节点来找到现有集群并启动它们。如果合适，Elasticsearch会将新节点添加到投票配置中。

在Master选举期间或当加入现有形成的集群时，节点向主节点发送加入请求以便正式添加到集群。

移除Master候选节点

删除Master候选节点时，不要同时删除太多的节点。例如，如果当前有7个Master候选节点，并且您希望将其减少到3个，则不可能一次性停止其中4个节点：这样做将只剩下3个节点，这不到投票配置的一半，这意味着集群无法采取任何进一步的操作。

更准确地说，如果您同时关闭了一半或更多Master候选节点，那么集群通常将变得不可用。如果发生这种情况，则可以通过再次启动删除的节点使集群恢复上线。

只要集群中至少有三个Master候选节点，作为一般规则，最好一次删除一个节点，以便集群有足够的[时间自动调整投票配置](#)，并使容错级别适应新的节点集。

如果只剩下两个Master候选节点，那么两个节点都不能安全地删除，因为这两个节点都需要维持集群运作。要删除其中一个节点，必须首先通知Elasticsearch它不应是投票配置的一部分，而应将投票权授予另一个节点。然后可以使排除的节点脱机，而不阻止另一个节点取得进展。添加到投票配置排除列表的节点仍然正常工作，但Elasticsearch尝试将其从投票配置中删除，因此不再需要其投票。重要的是，Elasticsearch永远不会自动将投票排除列表上的节点移回投票配置。一旦排除的节点成功地从投票配置中自动重新配置出来，就可以安全地关闭它，而不会影响集群的主级可用性。可以使用[Voting configuration exclusions API](#) 将节点添加到投票配置排除列表。例如：

```
# Add node to voting configuration exclusions list and wait for the system
# to auto-reconfigure the node out of the voting configuration up to the
# default timeout of 30 seconds
POST /_cluster/voting_config_exclusions?node_names=node_name

# Add node to voting configuration exclusions list and wait for
# auto-reconfiguration up to one minute
POST /_cluster/voting_config_exclusions?node_names=node_name&timeout=1m
```

应该添加到排除列表中的节点通过`?node_names`查询参数指定，或者通过它们的永久节点ID使用`?node_ids`查询参数指定。如果对投票配置排除API的调用失败，您可以安全地重试。只有成功的响应才能保证节点实际上已从投票配置中删除，并且

不会恢复。如果所选的主节点被排除在投票配置之外，那么它将让位给另一个仍在投票配置中的Master候选节点（如果这样的节点可用）。

尽管投票配置排除API对于将两个节点缩减为一个节点集群最为有用，但也可以使用它同时删除多个Master候选节点。将多个节点添加到排除列表会使系统尝试从投票配置中自动重新配置所有这些节点，从而允许它们安全关闭，同时保持集群可用。在上面描述的示例中，将一个七个主节点集群缩小到只有三个主节点，您可以将四个节点添加到排除列表中，等待确认，然后同时关闭它们。

只有在短时间内从集群中移除至少一半Master候选节点时，才需要投票排除。在删除非Master候选节点时不需要它们，在删除不到一半Master候选节点时也不需要它们。

为节点添加排除将在投票配置排除列表中为该节点创建一个条目，该条目使系统自动尝试重配投票配置以删除该节点，并阻止该节点在删除后返回投票配置。当前排除列表存储在群集状态下，可以按如下方式进行检查：

```
GET /_cluster/state?filter_path=metadata.cluster_coordination.voting_config_ex
```

此列表的大小受 `cluster.max_voting_config_exclusions` 限制，默认为10。请参见[发现和集群形成设置](#)。由于投票配置排除是永久的并且数量有限，因此必须清除它们。通常，在集群上执行某些维护时会添加一个排除项，并且在维护完成时应该清除排除项。在正常操作中，群集不应具有投票配置排除。

如果某个节点由于要永久关闭而被排除在投票配置之外，则可以在关闭该节点并将其从集群中移除之后移除其排除。如果排除项是错误创建的，或者只是通过指定临时需要，则也可以清除排除项 `?wait_for_removal=false`。

```
# Wait for all the nodes with voting configuration exclusions to be removed from the cluster and then remove all the exclusions, allowing any node to return to the voting configuration in the future.
DELETE /_cluster/voting_config_exclusions

# Immediately remove all the voting configuration exclusions, allowing any node to return to the voting configuration in the future.
DELETE /_cluster/voting_config_exclusions?wait_for_removal=false
```

升级Elasticsearch

索引模块

映射

7.1 Dynamic mapping 动态映射

Elasticsearch最重要的功能之一就是它会尽量避开你的视线，让你尽快开始探索你的数据。要为文档编制索引，不必首先创建索引、定义映射类型和定义字段 - 只需为文档编制索引，索引、类型和字段将自动显示：

```
PUT data/_doc/1 ①
{ "count": 5 }
```

①创建 data 索引、_doc 映射类型和名为 count 且数据类型为 long 的字段。

自动检测和添加新字段称为 **动态映射**。动态映射规则可以通过以下方式进行自定义以满足您的目的：

动态字段映射

管理动态场检测的规则。

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，无论是自动创建的还是显式创建的。

8.2.1 分词器详解

分词器,无论是内置还是自定义,只是一个包,它包含三个较低级别的模块: 字符过滤器、标记器和标记过滤器。

内置分词器预先将这些模块打包到适合不同语言和文本类型的分词器中。Elasticsearch还公开了各个模块,以便可以组合它们来定义新的自定义分词器。

8.2.1.1 字符过滤器 Character filter

字符过滤器以字符流的形式接收原始文本,并通过添加、删除或更改字符来转换流。例如,可以使用字符过滤器将印度教阿拉伯数字(୧୨୩୫୦୬୮୯)转化为等值的阿拉伯拉丁数字(0123456789),或者从流中移除比如``这样的HTML元素

分词器可以有零个或多个字符过滤器,它们将按顺序应用。

8.2.1.2 标记器 Tokenizer

标记器接收字符流,将其分解为单个标记(通常是单个单词),然后输出标记流。例如,空格 whitespace 标记器只要看到空格,就会将文字分割成标记。它会转换文本 "Quick brown fox!" 为 [Quick, brown, fox!].

标记器还负责记录每个术语的顺序或位置以及术语所代表的原始单词的开始和结束字符偏移量。

分词器必须只有一个标记器。

8.2.1.3 标记过滤器 Token filter

标记过滤器接收标记流,可以添加、删除或更改标记。例如,小写 lowercase 标记筛选器将所有标记转换为小写,停止 stop 标记筛选器从标记流中删除类似 the 的常用词(停止词),同义词标记筛选器将同义词引入标记流。

不允许标记筛选器更改每个标记的位置或字符偏移量。

一个分词器可以有零个或多个标记过滤器,它们是按顺序应用的

8.2.2 索引和搜索分词

文本分析分两次进行：

索引时间点

对文档进行索引时，将分析任何 `text` 字段值。

搜索时间点

在 `text` 字段上运行全文本搜索时，将分析查询字符串（用户正在搜索的文本）。

搜索时间点也称为 **查询时间点**。

每次使用的分词器或分析规则集合也被称为**索引分词器**或**搜索分词器**。

8.2.2.1 索引和搜索分词器如何协同工作

在大多数情况下，索引和搜索时应该使用相同的分词器。这可以确保将字段的值和查询字符串更改为相同形式的标记。反过来，这将确保标记在搜索期间与预期匹配。

例子

使用 `text` 字段中的以下值索引为文档

```
The QUICK brown foxes jumped over the dog!
```

字段的索引分词器将值转换为标记并将其规范化。在这种情况下，每个标记代表一个单词：

```
[ quick, brown, fox, jump, over, dog ]
```

然后这些标记被索引。

稍后，用户将在同一 `text` 字段中搜索：

```
"Quick fox"
```

用户希望这次搜索能匹配到之前索引的句子，`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` VS `QUICK`
- `fox` VS `foxes`

为此，使用相同的分词器分析查询字符串。此分词器生成以下标记：

```
[ quick, fox ]
```

为了执行搜索，Elasticsearch 将这些查询字符串标记与文本字段中索引的标记进行比较。

标记	查询字符串	text 字段
qucik	x	x
brown		x
fox	x	x
j ump		x
over		x
dog		x

因为字段值与查询字符串的分词方法相同，所以它们创建了相同的标记。标记 quick 和 fox 是完全匹配的。这意味着搜索与包含 "The QUICK brown foxes jumped over the dog!" 的文档匹配，正如用户所期望的那样。

8.2.2.2 何时使用其他搜索分词器

虽然不常见，但有时在索引和搜索时使用不同的分词器是很重要的。要启用此功能，Elasticsearch允许您指定单独的搜索分词器。

通常，只有在对字段值和查询字符串使用相同形式的标记会创建意外或不相关的搜索匹配时，才应指定单独的搜索分词器。

例子

Elasticsearch用于创建只匹配以所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，但不能返回 taxi 或 bat。

一个文档将添加到搜索引擎的索引中；此文档 text 字段中包含一个这样的单词：

```
"Apple"
```

字段的索引分词器将值转换为标记并将其规范化。在这种情况下，每个标记表示单词的潜在前缀：

```
[ a, ap, app, appl, apple]
```

然后这些标记被索引。

稍后，用户将在同一 text 字段中搜索：

```
"appli"
```

用户希望此搜索只匹配以 appli 开头的单词，例如 appliance 或 application。搜索结果应该与 apple 不匹配。

但是，如果使用索引分词器分析此查询字符串，它将生成以下标记：

```
[ a, ap, app, appl, appli ]
```

当Elasticsearch将这些查询字符串标记与为 apple 索引的字符串标记进行比较时，它会找到几个匹配项。

标记	appli	apple
a	x	x
ap	x	x
app	x	x
appl	x	x
appli		x

这意味着搜索结果会错误地匹配 `apple`。不仅如此，它还可以匹配任何以 `a` 开头的单词。

要解决此问题，可以为 `text` 字段上使用的查询字符串指定不同的搜索分词器。

在这个例子里，您可以指定一个搜索分词器来生成单个标记而不是一组前缀：

```
[ appli ]
```

此查询字符串标记将只匹配以 `appli` 开头的单词的标记，这更好地符合用户的搜索期望。

7.2 Explicit mapping 显式映射

您对数据的了解比Elasticsearch所能猜到的要多，因此，虽然动态映射对于入门很有用，但在某个时候您可能需要指定自己的显式映射。

当[创建索引](#)和[向现有索引添加字段](#)时，可以创建字段映射。

使用显式映射创建索引

您可以使用[create index](#) API创建具有显式映射的新索引。

```
PUT /my-index-000001
{
  "mappings": {
    "properties": {
      "age": { "type": "integer" }, ①
      "email": { "type": "keyword" }, ②
      "name": { "type": "text" }     ③
    }
  }
}
```

① 创建 `age`，一个 `integer` 字段

② 创建 `email`，一个 `keyword` 字段

③ 创建 `name`，一个 `text` 字段

向现有映射添加字段

可以使用 [put mapping](#) API向现有索引添加一个或多个新字段。

下面的示例添加 `employee-id`，这是一个 `index` 映射参数值为 `false` 的 `keyword` 字段。这意味着 `employee-id` 字段的值被存储后，但没有索引或可供搜索。

```
PUT /my-index-000001/_mapping
{
  "properties": {
    "employee-id": {
      "type": "keyword",
      "index": false
    }
  }
}
```

更新字段的映射

除了支持的[映射参数](#)外，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索引的数据无效。

如果需要更改数据流备份索引中字段的映射，请参阅[更改数据流的映射和设置](#)。

如果需要更改字段在其他索引中的映射，请使用正确的映射创建一个新索引，并将数据[重新索引](#)到该索引中。

重命名字段将使已在旧字段名下索引的数据无效。而是添加一个 `alias` 字段来创建备用字段名。

查看索引的映射

您可以使用[get mapping API](#)来查看现有索引的映射。

```
GET /my-index-000001/_mapping
```

API 返回以下结果：

```
{
  "my-index-000001" : {
    "mappings" : {
      "properties" : {
        "age" : {
          "type" : "integer"
        },
        "email" : {
          "type" : "keyword"
        },
        "employee-id" : {
          "type" : "keyword",
          "index" : false
        },
        "name" : {
          "type" : "text"
        }
      }
    }
  }
}
```

查看特定字段的映射

如果只想查看一个或多个特定字段的映射，可以使用[get field mapping API](#)。

如果您不需要索引的完整映射，或者索引包含大量字段，这将非常有用。

以下请求检索 `employee-id` 字段的映射。

```
GET /my-index-000001/_mapping/field/employee-id
```

API 返回以下结果：

```
{
  "my-index-000001" : {
    "mappings" : {
      "employee-id" : {
        "full_name" : "employee-id",
        "mapping" : {
          "employee-id" : {
            "type" : "keyword",
            "index" : false
          }
        }
      }
    }
  }
}
```


映射限制设置

使用以下设置可以限制字段映射的数量（手动或动态创建）并防止文档导致映射爆炸：

index.mapping.total_fields.limit

索引中字段的最大数目。字段和对象映射以及字段别名都计入此限制。默认值为 `1000`。

设置此限制是为了防止映射和搜索变得过大。更高的值会导致性能下降和内存问题，特别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，用于限制查询中布尔子句的最大数目。

如果字段映射包含一组大的任意键，请考虑使用展平数据类型。

index.mapping.depth.limit

字段的最大深度，以内部对象的数量来度量。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 `nested` 映射的最大数目。`nested` 类型只应在特殊情况下使用，即需要相互独立地查询对象数组时。为了防止映射设计不当，此设置限制每个索引的唯一 `nested` 类型数。默认值为 `50`。

index.mapping.nested_objects.limit

单个文档可以跨所有嵌套类型包含的最大嵌套JSON对象数。当文档包含太多嵌套对象时，此限制有助于防止内存不足错误。默认值为 `10000`。

index.mapping.field_name_length.limit

设置字段名的最大长度。这个设置实际上并不能解决映射爆炸的问题，但是如果还想限制字段长度，它可能仍然很有用。通常不需要设置此设置。默认值是可以的，除非用户开始添加大量具有很长名称的字段。默认值为 `Long.MAX_VALUE`（无限制）。

8 Elasticsearch-Text analysis文本分析

文本分析 `Text analysis` 是转化非结构性文本(比如 邮件的内容、产品简介)到结构性格式的过程，这种转化将有助于搜索。

何时配置文本分析

Elasticsearch在索引、搜索 `text` 字段时会采用文本分析

如果索引不包含 `text` 字段，无需配置文本分析，可以跳过本部分

然而，如果使用 `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。你应该查看分析配置，如果要使用Elasticsearch执行以下操作：

- 创建搜索引擎
- 挖掘非结构化数据
- 对特定语言进行微调搜索
- 进行词典学或语言学研究

8.1 概述

文本分析使Elasticsearch能够执行全文本搜索，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`，您可能想要返回包含 `A quick brown fox jumps over the lazy dog` 的文档，您也可能还需要包含相关单词的文档，比如 `fast fox` 或 `foxes leap`。

8.1.1 标记化 Tokenization

分析通过 标记化 使全文本搜索成为可能：将文本分解成更小的块，称为 标记 tokens。在大多数情况下，这些标记是单独的单词。

如果您将短语 `the quick brown fox` 索引为单个字符串，并且用户搜索 `quick fox`，则它不被视为匹配项。但是，如果将短语标记化并分别索引每个单词，则可以单独搜寻查询字符串中的术语 terms。这意味着可以通过搜索 `quickfox`、`foxbrown` 或其他变体来匹配它们。

8.1.2 规范化 Normalization

标记化允许在单个术语上进行匹配，但每个标记仍然按字面匹配。这意味着：

- 搜索 `Quick` 不会匹配 `quick`，即使您可能希望其中一个术语与另一个匹配
- 虽然 `fox` 和 `foxes` 共享同一个词根，但是搜索 `foxes` 将与 `fox` 不匹配，反之亦然。
- 搜索 `jumps` 将不会匹配到 `leaps`。虽然它们不共享一个词根，但它们是同义词，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规范化为标准格式。这允许您匹配与搜索词不完全相同，但足够相似且仍然相关的标记。例如：

- `Quick` 可以小写为：`quick`。
- `foxes` 可以视为词干，也可以简化为它的词根：`fox`。
- `jump` 和 `leap` 是同义词，可以作为一个单词编制索引：`jump`。

为了确保搜索词与预期的这些词匹配，可以对查询字符串应用相同的标记化 tokenization 和规范化 normalization 规则。例如，搜索 `Foxes leap` 可以规范化为搜索 `fox jump`。

8.1.3 自定义文本分析 Customize text analysis

文本分析由 分词器 analyzer 执行，分词器是一组控制整个过程的规则。

Elasticsearch 包含一个默认的分词器，称为 标准分词器，它可以很好地在大多数用例中开箱即用。

如果您想定制您的搜索体验，您可以选择不同的 内置分词器，甚至可以配置自定义分词器。自定义分词器使您可以控制分析过程的每个步骤，包括：

- 在标记化之前对文本进行更改

- 如何将文本转换为标记
- 索引或搜索之前对标记的规范化进行更改

8.2 概念

本节解释了Elasticsearch中文本分析的基本概念。

- [分词器详解](#)
- [索引搜索分词](#)
- [词干提取](#)
- [标记图](#)

8.2.3 词干提取

词干分析 `Stemming` 是将一个词还原为词根形式的过程。这样可以确保在搜索期间单词的变体匹配。

例如，`walking` 和 `walked` 可以来源于同一个词根：`walk`。一旦词干有了词干，在搜索中两个词的出现都会匹配另一个词。

词干分析依赖于语言，但通常需要从单词中删除前缀和后缀。

在某些情况下，词干单词的词根形式可能不是真正的单词。例如，`jumping` 和 `jumpiness` 都可以归为 `jumpi`。虽然 `jumpi` 不是一个真正的英语单词，但对于搜索来说并不重要；如果一个单词的所有变体都被简化为同一个词根形式，它们就会正确匹配。

8.2.3.1 词干分析标记过滤器 Stemmer token filters

在Elasticsearch中，词干由词干分析标记过滤器处理。这些标记过滤器可以根据词干分析的方式进行分类：

- 算法词干分析器，根据一组规则对单词进行词干处理
- 字典词干分析器，通过在字典中查找单词进行词干处理

因为词干分析会改变标记，我们建议在索引和搜索分析期间使用相同的词干分析标记过滤器。

8.2.3.2 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还原为其根形式。例如，英语的算法词干分析器可能会从复数单词的末尾删除 `-s` 和 `-es` 后缀。

算法词干分析器有几个优点：

- 它们需要很少的设置，而且通常开箱即用。
- 他们占用内存很少。
- 它们通常比字典词干分析器快。

然而，大多数算法词干分析器只改变单词的现有文本。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分析：

- `stemmer`，它为几种语言提供算法词干分析，有些语言还带有附加变体。
- `kstem`，一个英语词干分析器，它结合了算法词干分析和内置字典。
- `porter_stem`，我们推荐的用于英语的算法词干分析器。
- `snowball`，它对几种语言使用基于 `Snowball` 的词干分析规则。

8.2.3.3 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，用字典中的词干单词替换非词干词的变体。

理论上，字典词干分析器非常适合于：

- 词干分析不规则的词
- 区分拼写相似但概念上不相关的单词，例如：
 - organ 和 organization
 - broker 和 broken

实际上，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

字典词干分析器的质量取决于字典。为了更好地工作，这些词典必须包含大量的单词，定期更新，并随语言趋势而变化。通常，当一本词典问世时，它是不完整的，有些词条已经过时了。

- 大小和性能

字典词干分析器必须将字典中的所有单词、前缀和后缀加载到内存中。这可能需要大量的 RAM。低质量的字典在删除前缀和后缀时也可能效率较低，这会显著降低词干分析过程的速度。

您可以使用 `hunspell` 标记过滤器来执行字典词干分析。

如果可以的话，我们建议在使用 `hunspell` 标记过滤器之前尝试一个针对您的语言的算法词干分析器。

8.2.3.4 控制词干分析

有时词干分析可以产生拼写相似但在概念上不相关的共享词根。例如，词干分析器可以将 `skies` 和 `skilling` 都简化为同一个词根：`ski`。

要防止这种情况并更好地控制词干分析，可以使用以下标记过滤器：

- `stemmer_override`，可用于定义词干分析特定标记的规则。
- `keyword_marker`，将指定标记记为关键字。关键字标记不会被后续的词干分析标记筛选器进行词干分析。
- `conditional`，可用于将标记记为关键字，类似于 `keyword_marker` 过滤器。

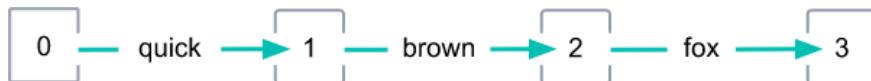
对于内置的语言分词器，您还可以使用 `stem_exclusion` 参数来指定一个不会被词干分析的单词列表。

8.2.4 标记图

当标记器将文本转换为标记流时，它还记录以下内容：

- 流中每个标记的 位置
- 位置长度， 标记跨越的位置数

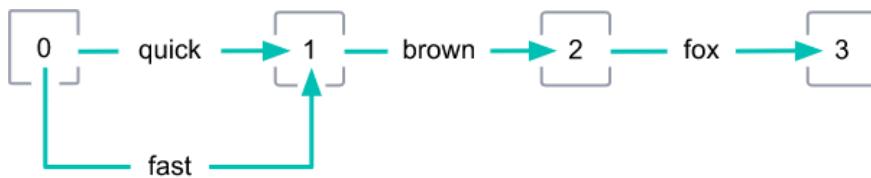
使用这些，可以为流创建一个称为标记图 token graph 的有向无环图。在标记图中，每个位置代表一个节点。每个标记表示一条边或一条弧，指向下一个位置。



8.2.4.1 同义词 Synonyms

一些标记过滤器可以向现有标记流添加新的标记，如同义词。这些同义词通常跨越与现有标记相同的位置。

在下图中，`quick` 及其同义词 `fast` 的位置都为0。它们跨越相同的位置。



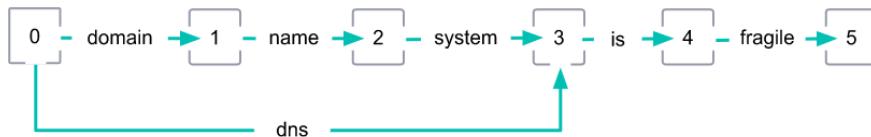
8.2.4.2 多位置标记

某些标记筛选器可以添加跨越多个位置的标记。它们可以包括多单词同义词的标记，例如使用“atm”作为“自动取款机”的同义词

但是，只有一些标记过滤器（称为图形标记过滤器）可以准确地记录多位置标记的位置长度。这些过滤器包括：

- `synonym_graph`
- `word_delimiter_graph`

在下图中，`domain name system` 及其同义词 `dns` 的位置都是0。但是，`dns` 的位置长度 `positionLength` 为 3 。图中的其他标记的默认位置长度为 1 。



8.2.4.3 使用标记图进行搜索

进行索引时会忽略 `positionLength` 属性，不支持包含多位置标记的标记图。

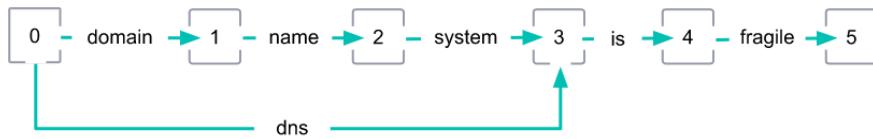
但是，查询（例如 `match` 或 `match_phrase` 查询）可以使用这些图从单个查询字符串生成多个子查询。

例子：

用户使用 `match_phrase` 查询运行以下短语搜索：

```
domain name system is fragile
```

在搜索分析期间，`dns`（`domain name system` 的同义词）被添加到查询字符串的标记流中。`dns` 标记的 `positionLength` 为 3。



`match_phrase` 查询使用这个图来生成了下列子查询短语：

```
dns is fragile
domain name system is fragile
```

这意味着这个查询将会匹配包含 `dns is fragile` 或 `domain name system is fragile` 的文档。

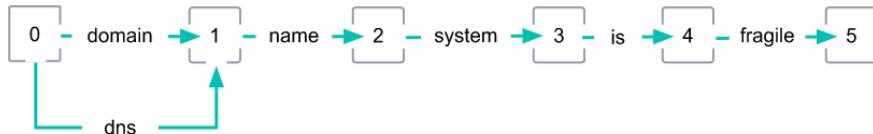
8.2.4.4 无效的标记图

以下标记筛选器可以添加跨越多个位置但只记录默认 `positionLength` 为 1 的标记：

- `synonym`
- `word_delimiter`

这意味着这些过滤器将为包含此类标记的流生成无效的标记图。

在下图中，`dns` 是 `domain name system` 的多位置同义词。但是，`dns` 的默认 `positionLength` 值为 1，这将导致一个无效的图形。



避免使用无效的标记图进行搜索。无效的图形可能会导致意外的搜索结果。

8.3 配置文本分析

8.3.1 测试分词器

analyze API 是查看分词器生成的术语的宝贵工具。可以在请求中内联指定内置分词器：

```
POST _analyze
{
  "analyzer": "whitespace",
  "text":     "The quick brown fox."
}
```

API返回如下结果：

```
{
  "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
    }
  ]
}
```

你也可以尝试组合：

- 标记器
- 零个或多个标记过滤器
- 零个或多个字符过滤器

```
POST _analyze
{
  "tokenizer": "standard",
  "filter":  [ "lowercase", "asciifolding" ],
  "text":     "Is this déjà vu?"
}
```

API返回如下结果：

```
{  
  "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  
    }  
  ]  
}
```

位置和字符偏移

从analyze API的输出可以看出，分词器不仅将单词转换为术语，还记录每个术语的顺序或相对位置（用于短语查询或单词邻近查询），以及原始文本中每个术语的开始和结束字符偏移量（用于突出显示搜索片段）。

或者，在特定索引上运行analyze API时，可以引用自定义分词器：

```

PUT my-index-000001
{
  "settings": {
    "analysis": {
      "analyzer": {
        "std_folded": { ①
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "asciifolding"
          ]
        }
      }
    },
    "mappings": {
      "properties": {
        "my_text": {
          "type": "text",
          "analyzer": "std_folded" ②
        }
      }
    }
  }
}

GET my-index-000001/_analyze ③
{
  "analyzer": "std_folded", ④
  "text": "Is this déjà vu?"
}

GET my-index-000001/_analyze
{
  "field": "my_text", ⑤
  "text": "Is this déjà vu?"
}

```

- ①定义一个名为 `std_folded` 的自定义分词器
- ② `my_text` 字段使用 `std_folded` 分词器
- ③要引用这个分词器，`analyze` API 必须指定索引名称
- ④通过名称引用到分词器
- ⑤通过 `my_text` 引用到分词器

API 返回如下结果：

```
{  
  "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  
    }  
  ]  
}
```

8.3.2 配置内置分词器

内置分词器可直接使用，无需任何配置。然而，其中一些支持配置选项来改变它们的行为。例如，可以将标准分析器配置为支持停词列表：

```
PUT my-index-000001
{
  "settings": {
    "analysis": {
      "analyzer": {
        "std_english": { ①
          "type": "standard",
          "stopwords": "_english_"
        }
      }
    },
    "mappings": {
      "properties": {
        "my_text": {
          "type": "text",
          "analyzer": "standard",
          "fields": {
            "english": {
              "type": "text",
              "analyzer": "std_english"
            }
          }
        }
      }
    }
  }
}

POST my-index-000001/_analyze
{
  "field": "my_text", ②
  "text": "The old brown cow"
}

POST my-index-000001/_analyze
{
  "field": "my_text.english", ③
  "text": "The old brown cow"
}
```

- ① 我们定义一个 `std_english` 分词器，它基于 `standard` 分词器创建，但是配置了需要被移除的预置英语停词列表
- ② `my_text` 字段直接使用 `standard` 分词器，无需任何配置。这个字段不会移除任何停词。结果术语为： [the, old, brown, cow]
- ③ `my_text.english` 字段使用 `std_english` 分词器，于是英语停词将会被移除，结果术语为： [old, brown, cow]

8.3.3 创建自定义分词器

8.3.3 创建自定义分词器

当内置分词器不能满足您的需要时，您可以创建一个自定义分词器，该分词器使用以下各项的适当组合：

- 零个或多个字符过滤器
- 标记器
- 零个或多个标记过滤器

8.3.3.1 配置

自定义分词器接受以下参数：

<code>tokenizer</code>	内置或自定义的标记器(必需)
<code>char_filter</code>	一组可选的内置或自定义字符过滤器的数组
<code>filter</code>	一组可选的内置或自定义标记过滤器的数组
<code>position_increment_gap</code>	当索引文本值数组时，Elasticsearch在一个值的最后一个项和下一个值的第一个项之间插入一个假“间隙”，以确保短语查询不匹配来自不同数组元素的两个词。默认为 <code>100</code> 。有关详细信息，请参见 <code>position_increment_gap</code> 。

8.3.3.2 配置范例

下面将展示一个组合了以下配置的例子：

Character Filter

- HTML Strip Character Filter

Tokenizer

- Standard Tokenizer

Token Filters

- Lowercase Token Filter
- ASCII-Folding Token Filter

```

PUT my-index-000001
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_custom_analyzer": {
          "type": "custom", ①
          "tokenizer": "standard",
          "char_filter": [
            "html_strip"
          ],
          "filter": [
            "lowercase",
            "asciifolding"
          ]
        }
      }
    }
  }
}

POST my-index-000001/_analyze
{
  "analyzer": "my_custom_analyzer",
  "text": "Is this <b>déjà vu</b>?"
}

```

- ①将 type 设置为 custom 告诉 Elasticsearch 我们正在定义一个自定义分词器。比较一下如何配置内置分词器： type 将被设置为内置分词器的名称，如 standard 或 simple。

上述例子将产生如下术语：

```
[ is, this, déjà, vu ]
```

上一个示例使用了默认配置的标记器、标记过滤器和字符过滤器，但是可以创建每个配置的版本并在自定义分词器中使用它们。

下面是一个更复杂的示例，它结合了以下内容：

Character Filter

- Mapping Character Filter, 配置后将替换 :) 为 _happy_ , :(为 _sad_

Tokenizer

- Pattern Tokenizer, 配置后将按照标点符号进行分割

Token Filters

- Lowercase Token Filter
- Stop Token Filter, 配置后将采用预配置的英文停词列表

下面是范例：

```

PUT my-index-000001
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_custom_analyzer": { ①
          "type": "custom",
          "char_filter": [
            "emoticons"
          ],
          "tokenizer": "punctuation",
          "filter": [
            "lowercase",
            "english_stop"
          ]
        }
      },
      "tokenizer": {
        "punctuation": { ②
          "type": "pattern",
          "pattern": "[ .,!?]"
        }
      },
      "char_filter": {
        "emoticons": { ③
          "type": "mapping",
          "mappings": [
            ":) => _happy_",
            ":(>_sad_"
          ]
        }
      },
      "filter": {
        "english_stop": { ④
          "type": "stop",
          "stopwords": "_english_"
        }
      }
    }
  }
}

POST my-index-000001/_analyze
{
  "analyzer": "my_custom_analyzer",
  "text": "I'm a :) person, and you?"
}

```

- ①配置这个索引的默认分词器为自定义的 `my_custom_analyzer`，这个分词器使用后续自定义的标记器、字符过滤器、标记过滤器
- ②定义一个自定义的 `punctuation` 标记器
- ③定义一个自定义的 `emoticons` 字符过滤器
- ④定义一个自定义的 `english_stop` 标记过滤器

上述例子将产生如下术语：

```
[ i' m, _happy_, person, you ]
```

8.3.4 指定分词器

8.4 内置分词器参考

8.5 标记化器参考

8.6 标记过滤器参考

8.7 字符過濾器參考

8.8 规范化器 Normalizers

规范化器与分词器类似，只是它们可能只发出一个标记。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一个子集。只允许基于每个字符工作的过滤器。例如，允许使用小写过滤器，但不允许使用需要将关键字作为一个整体来查看的词干过滤器。当前可在规范化器中使用的过滤器的列表如下：

```
arabic_normalization, ascii_folding, bengali_normalization, cjk_width,
decimal_digit, elision, german_normalization, hindi_normalization,
indic_normalization, lowercase, persian_normalization,
scandinavian_folding, serbian_normalization, sorani_normalization,
uppercase
```

Elasticsearch附带一个 `lowercase` 内置规范化器。对于其他形式的规范化，需要自定义配置。

自定义规范化器

自定义规范化器需要一组字符过滤器和标记过滤器的列表。

```
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"
      }
    }
  }
}
```

索引模板

本主题介绍Elasticsearch 7.8中引入的可组合索引模板。有关索引模板以前如何工作的信息，请参阅[旧版模板文档](#)。

索引模板是告诉Elasticsearch如何在创建索引时配置索引的一种方法。对于数据流，索引模板在创建时配置流的备份索引。模板是在创建索引之前配置的，当手动或通过索引文档创建索引时，模板设置将用作创建索引的基础。

有两种类型的模板，索引模板和[组件模板](#)。组件模板是可重用的构建块，用于配置映射、设置和别名。使用组件模板来构造索引模板，它们不会直接应用于一组索引。索引模板可以包含组件模板的集合，也可以直接指定设置、映射和别名。

如果新的数据流或索引与多个索引模板匹配，则使用具有最高优先级的索引模板。

Elasticsearch具有内置索引模板，每个模板的优先级为 100，用于以下索引模式：

- logs--
- metrics--
- synthetics--

[Elastic代理](#)使用这些模板来创建数据流。Fleet集成创建的索引模板使用类似的重叠索引模式，优先级高达 200。

如果您使用Fleet或Elastic代理，请为索引模板指定低于 100 的优先级，以避免覆盖这些模板。否则，为避免意外应用模板，请执行以下一个或多个操作：

- 要禁用所有内置索引和组件模板，请用[cluster update settings API](#)设置 `stack.templates.enabled` 为 `false`。
- 使用不重叠的索引模式。
- 为重叠模式的模板分配高于 200 的 `priority`。例如，如果您不使用 Fleet或Elastic代理，并且希望为 `logs-*` 索引模式创建一个模板，请将模板的优先级指定为 500。这将确保应用您的模板代替了 `logs-*--` 的内置模板。

当可组合模板与给定索引匹配时，它总是优先于旧模板。如果没有匹配的可组合模板，则遗留模板仍然可以匹配并应用。

如果索引是使用显式设置创建的，并且还与索引模板匹配，则创建索引请求中的设置优先于索引模板及其组件模板中指定的设置。

```
PUT _component_template/component_template1
{
  "template": {
    "mappings": {
      "properties": {
        "@timestamp": {
          "type": "date"
        }
      }
    }
  }
}

PUT _component_template/other_component_template
{
  "template": {
    "mappings": {
      "properties": {
        "ip_address": {
          "type": "ip"
        }
      }
    }
  }
}

PUT _index_template/template_1
{
  "index_patterns": ["te*", "bar*"],
  "template": {
    "settings": {
      "number_of_shards": 1
    },
    "mappings": {
      "properties": {
        "host_name": {
          "type": "keyword"
        },
        "created_at": {
          "type": "date",
          "format": "EEE MMM dd HH:mm:ss Z yyyy"
        }
      }
    },
    "aliases": {
      "mydata": { }
    }
  },
  "priority": 500,
  "composed_of": ["component_template1", "other_component_template"],
  "version": 3,
  "_meta": {
    "description": "my custom"
  }
}
```

模拟多组件模板

由于模板不仅可以由多个组件模板组成，还可以由索引模板本身组成，因此有两个模拟api来确定最终的索引设置。

要模拟将应用于特定索引名称的设置，请执行以下操作：

```
POST /_index_template/_simulate_index/my-index-000001
```

要模拟将从现有模板应用的设置，请执行以下操作：

```
POST /_index_template/_simulate/template_1
```

您还可以在模拟请求中指定模板定义。这使您能够在添加新模板之前验证设置是否按预期应用。

```
PUT /_component_template/ct1
{
  "template": {
    "settings": {
      "index.number_of_shards": 2
    }
  }
}

PUT /_component_template/ct2
{
  "template": {
    "settings": {
      "index.number_of_replicas": 0
    },
    "mappings": {
      "properties": {
        "@timestamp": {
          "type": "date"
        }
      }
    }
  }
}

POST /_index_template/_simulate
{
  "index_patterns": ["my*"],
  "template": {
    "settings": {
      "index.number_of_shards": 3
    },
    "composed_of": ["ct1", "ct2"]
  }
}
```

响应显示将应用于匹配索引的设置、映射和别名，以及其配置将被模拟模板体或更高优先级模板取代的任何重叠模板。

```
{  
  "template" : {  
    "settings" : {  
      "index" : {  
        "number_of_shards" : "3",     ①  
        "number_of_replicas" : "0"  
      }  
    },  
    "mappings" : {  
      "properties" : {  
        "@timestamp" : {  
          "type" : "date"           ②  
        }  
      }  
    },  
    "aliases" : { }  
  },  
  "overlapping" : [  
    {  
      "name" : "template_1",       ③  
      "index_patterns" : [  
        "my*"  
      ]  
    }  
  ]  
}
```

①来自模拟模板体的碎片数

② @timestamp 字段继承自 ct2 组件模板

③任何本应匹配但优先级较低的重叠模板

索引模板

索引模板

12 Search your data 搜索数据

搜索查询或查询是对Elasticsearch数据流或索引中的数据的信息的请求。

您可以将查询视为一个问题，以Elasticsearch理解的方式编写。根据您的数据，您可以使用查询来获得以下问题的答案：

- 服务器上的哪些进程响应时间超过500毫秒？
- 上周，我的网络上有哪些用户运行了regsvr32.exe？
- 我的网站上的哪些页面包含特定的单词或短语？

搜索由一个或多个查询组成，这些查询被合并并发送到Elasticsearch。与搜索的查询匹配的文档将在响应的命中或搜索结果中返回。

搜索还可能包含用于更好地处理查询的附加信息。例如，搜索可能仅限于特定索引或只返回特定数量的结果。

执行搜索

您可以使用[search API](#)来搜索和[聚合\(aggregate\)](#)存储在Elasticsearch数据流或索引中的数据。API的 `query` 请求体参数接受用[Query DSL](#)编写的查询。

以下请求使用[match](#)查询搜索 `my-index-000001`。此查询将匹配 `user.id` 的值为 `kimchy` 的文档。

```
GET /my-index-000001/_search
{
  "query": {
    "match": {
      "user.id": "kimchy"
    }
  }
}
```

API回调中返回匹配查询的前10个结果，存放在 `hits.hits` 属性中

```
{
  "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": "my-index-000001",
        "_type": "_doc",
        "_id": "kxFcnMByiguvud1Z8vC",
        "_score": 1.3862942,
        "_source": {
          "@timestamp": "2099-11-15T14:12:12",
          "http": {
            "request": {
              "method": "get"
            },
            "response": {
              "bytes": 1070000,
              "status_code": 200
            },
            "version": "1.1"
          },
          "message": "GET /search HTTP/1.1 200 1070000",
          "source": {
            "ip": "127.0.0.1"
          },
          "user": {
            "id": "kimchy"
          }
        }
      }
    ]
  }
}
```

常用搜索选项

可以使用以下选项自定义搜索。

Query DSL

Query DSL 支持多种查询类型，您可以混合和匹配以获得所需的结果。查询类型包括：

- [布尔\(Boolean\)查询](#)和其他[复合\(compound\)查询](#)，允许您根据多个条件组合查询并匹配结果
- 用于筛选和查找精确匹配项的[术语级\(Term-level\)查询](#)
- [全文\(Full text\)查询](#)，通常在搜索引擎中使用
- [地理\(Geo\)和空间\(spatial\)查询](#)

聚合(Aggregations)

您可以使用[搜索聚合\(search aggregations\)](#)来获取搜索结果的统计信息和其他分析。聚合可帮助您回答以下问题：

- 我的服务器的平均响应时间是多少？
- 我的网络中用户访问的最大IP地址是什么？
- 客户的总交易收入是多少？
- 搜索多个数据流和索引

搜索多个数据流和索引

可以使用逗号分隔值和类似grep的索引模式来搜索同一请求中的多个数据流和索引。你甚至可以从特定的索引中提升搜索结果。请参见[搜索多个数据流和索引](#)。

搜索结果分页

默认情况下，搜索只返回前10个匹配的命中结果。要检索更多或更少的文档，请参见[搜索结果分页](#)。

检索选定字段

搜索结果中的`hit.hits`属性包含每次命中的完整文档`_source`。若要仅检索`_source`或其他字段的子集，请参见[检索选定字段](#)。

对搜索结果排序

默认情况下，搜索结果按`_score`排序，这是一个衡量每个文档与查询匹配程度的相关性分数。要自定义这些分数的计算，请使用[script_score 查询](#)。要按其他字段值对搜索结果进行排序，请参见[对搜索结果排序](#)。

运行异步搜索

Elasticsearch搜索的目的是在大量数据上快速运行，通常在毫秒内返回结果。因此，默认情况下搜索是同步的。搜索请求在返回响应之前等待完整的结果。

但是，对于跨[冻结索引](#)或[多个集群](#)的搜索，完整的结果可能需要更长的时间。

为了避免长时间等待，可以运行异步搜索。[异步搜索](#)允许您检索长时间运行的搜索时先获得部分结果，并在以后获得完整的结果。

搜索超时

默认情况下，搜索请求不会超时。请求在返回响应之前等待完整的结果。

虽然[异步搜索](#)是为长时间运行的搜索而设计的，但您也可以使用`timeout`参数指定要等待搜索完成的持续时间。如果在此期间结束之前没有收到响应，则请求失败并返回错误。

```
GET /my-index-000001/_search
{
  "timeout": "2s",
  "query": {
    "match": {
      "user.id": "kimchy"
    }
  }
}
```

要为所有搜索请求设置群集范围的默认超时，请使用[集群设置API\(cluster settings API\)](#)配置 `search.default_search_timeout` 参数。如果请求中没有传递 `timeout` 参数，则使用此全局超时持续时间。如果全局搜索超时在搜索请求完成之前过期，则使用[任务取消\(task cancellation\)](#)来取消请求。这个 `search.default_search_timeout` 设置默认为 `-1`（无超时）。

搜索取消

可以使用[任务管理API\(task management API\)](#)取消搜索请求。当客户端的HTTP连接关闭时，Elasticsearch还会自动取消搜索请求。我们建议您将客户端设置为在搜索请求中止或超时时关闭HTTP连接。

跟踪总命中

通常，如果不访问所有匹配项，则无法准确计算总命中计数，这对于匹配大量文档的查询来说代价高昂。`track_total_hits` 参数允许您控制跟踪总命中数。考虑到通常有一个点击数下限就足够了，比如“至少有10000次命中”，默认设置为 `10000` 次。这意味着请求将精确计算命中总数，最多可达10000次。这是一个很好的权衡，以加快搜索速度，如果你不需要在某个阈值后的准确点击数。

当设置为 `true` 时，搜索响应将始终跟踪与查询精确匹配的点击数（例如。当 `track_total_hits` 设置为`true`时，`total.relation` 将始终等于 “`eq`”）。否则 “`total.relation`” 在搜索响应中的 “`total`” 对象中返回确定 “`total.value`” 应该被解释。“`gte`” 值表示 “`total.value`” 是与查询匹配的总命中数的下限，值 “`eq`” 表示 “`total.value`” 是准确的计数。

```
GET my-index-000001/_search
{
  "track_total_hits": true,
  "query": {
    "match": {
      "user.id": "elkbee"
    }
  }
}
```

返回：

```
{
  "_shards": ...
  "timed_out": false,
  "took": 100,
  "hits": {
    "max_score": 1.0,
    "total": {
      "value": 2048, ①
      "relation": "eq" ②
    },
    "hits": ...
  }
}
```

①与查询匹配的命中总数。

②计数准确（例如“eq”表示等于）。

也可以将 `track_total_hits` 设置为整数。例如，以下查询将精确跟踪匹配查询的总命中计数，最多100个文档：

```
GET my-index-000001/_search
{
  "track_total_hits": 100,
  "query": {
    "match": {
      "user.id": "elkbee"
    }
  }
}
```

响应中的 `hits.total.relation`，将指示 `hits.total.value` 中返回的值是否是准确的（“eq”）或总数的下限（“gte”）。

例如，以下响应：

```
{
  "_shards": ...
  "timed_out": false,
  "took": 30,
  "hits": {
    "max_score": 1.0,
    "total": {
      "value": 42,           ①
      "relation": "eq"       ②
    },
    "hits": ...
  }
}
```

①查询匹配到了42个文档

②计数是准确的（“eq”）

如果根本不需要跟踪命中总数，则可以通过将此选项设置为 `false` 来提高查询时间：

```
GET my-index-000001/_search
{
  "track_total_hits": false,
  "query": {
    "match": {
      "user.id": "elkbee"
    }
  }
}
```

返回：

```
{
  "_shards": ...,
  "timed_out": false,
  "took": 10,
  "hits": {           ①
    "max_score": 1.0,
    "hits": ...
  }
}
```

①命中总数是未知的

快速检查匹配的文档

如果您只想知道某个特定查询是否存在任意文档的匹配结果，我们可以指定该查询的 `size` 为 `0`，来表示我们对搜索的具体结果不感兴趣。还可以将 `terminate_after` 设置为 `1`，以指示只要找到第一个匹配的文档（每个分片），就可以终止查询执行。

```
GET /_search?q=user.id:elkbee&size=0&terminate_after=1
```

- `terminate_after` 始终在 `post_filter` 之后应用，并在分片上收集到足够的点击后停止查询和聚合执行。虽然聚合上的文档计数可能不会反映响应中的 `hits.total`，因为聚合是在后筛选之前应用的。

响应中将不包含任何命中，因为 `size` 设置为 `0`。这个 `hits.total` 也将等于 `0`，表示没有匹配的文档；大于 `0` 表示在查询提前终止时，至少有相同数量的文档与查询匹配。另外，如果查询提前终止，则在响应中将 `terminated_early` 标志设置为 `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 `took` 数值表示处理此请求所用的毫秒数，从节点接收到查询后快速开始，直到完成所有与搜索相关的工作，以及将上述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。

12.1 Collapse search results 折叠搜索结果

可以使用 `collapse` 参数根据字段值折叠搜索结果。通过为每个折叠键只选择排序靠前的文档来完成折叠。

例如，以下搜索结果按 `user.id` 折叠，并按照 `http.response.bytes` 排序

```
GET /my-index-000001/_search
{
  "query": {
    "match": {
      "message": "GET /search"
    }
  },
  "collapse": {
    "field": "user.id"          ①
  },
  "sort": [ "http.response.bytes" ], ②
  "from": 10                  ③
}
```

①使用 `user.id` 字段对结果集合进行折叠 ②通过 `http.response.bytes` 对结果进行排序 ③定义第一个折叠结果的偏差值

- 响应中的总命中数表示匹配的文档数而不折叠。不同组的总数未知。

展开折叠结果

也可以使用 `inner_hits` 选项展开每个折叠的顶部命中。

```
GET /my-index-000001/_search
{
  "query": {
    "match": {
      "message": "GET /search"
    }
  },
  "collapse": {
    "field": "user.id",    ①
    "inner_hits": {
      "name": "most_recent",   ②
      "size": 5,                ③
      "sort": [ { "@timestamp": "asc" } ] ④
    },
    "max_concurrent_group_searches": 4 ⑤
  },
  "sort": [ "http.response.bytes" ]
}
```

①使用 “`user.id`” 字段折叠结果集合 ②响应中用于内部命中部分的名称 ③每个折叠键要检索的 `inner_hits` ④如何在每个组内对文档进行排序 ⑤允许为每个组检索 `inner_hits` 的并发请求数

有关支持的选项和响应格式的完整列表，请参见[内部命中](#)。

也可以为每个折叠命中请求多个 `inner_hits`。当您想要获得折叠命中的多个表示时，这可能很有用。

```
GET /my-index-000001/_search
{
  "query": {
    "match": {
      "message": "GET /search"
    }
  },
  "collapse": {
    "field": "user.id",
    "inner_hits": [
      {
        "name": "largest_responses",
        "size": 3,
        "sort": [ "http.response.bytes" ]
      },
      {
        "name": "most_recent",
        "size": 3,
        "sort": [ { "@timestamp": "asc" } ]
      }
    ],
    "sort": [ "http.response.bytes" ]
  }
}
```

组的扩展是通过为每个 `inner_hit` 的请求的响应中返回的每个折叠的命中发送一个额外的查询来完成的。如果有太多的组和/或``inner_hit`请求，这会大大降低速度。

`max_concurrent_group_searches`请求参数可用于控制此阶段允许的最大并发搜索数。默认值基于数据节点的数量和默认的搜索线程池大小。

第二级折叠

第二级折叠也是支持的，并被应用于 `inner_hits`。

例如，以下搜索按以下方式折叠结果：`geo.country_name`。在每个 `geo.country_name` 中，内部命中通过 `user.id` 折叠。

```
GET /my-index-000001/_search
{
  "query": {
    "match": {
      "message": "GET /search"
    }
  },
  "collapse": {
    "field": "geo.country_name",
    "inner_hits": {
      "name": "by_location",
      "collapse": { "field": "user.id" },
      "size": 3
    }
  }
}
```

响应：

```
{
  ...
  "hits": [
    {
      "_index": "my-index-000001",
      "_type": "_doc",
      "_id": "9",
      "_score": ...,
      "_source": {...},
      "fields": { "geo": { "country_name": [ "UK" ] } },
      "inner_hits": {
        "by_location": {
          "hits": {
            ...
            "hits": [
              {
                ...
                "fields": { "user": "id": { [ "user124" ] } }
              },
              {
                ...
                "fields": { "user": "id": { [ "user589" ] } }
              },
              {
                ...
                "fields": { "user": "id": { [ "user001" ] } }
              }
            ]
          }
        }
      }
    },
    {
      "_index": "my-index-000001",
      "_type": "_doc",
      "_id": "1",
      "_score": ...,
      "_source": {...},
      "fields": { "geo": { "country_name": [ "Canada" ] } },
      "inner_hits": {
        "by_location": {
          "hits": {
            ...
            "hits": [
              {
                ...
                "fields": { "user": "id": { [ "user444" ] } }
              },
              {
                ...
                "fields": { "user": "id": { [ "user1111" ] } }
              },
              ...
              "fields": { "user": "id": { [ "user999" ] } }
            ]
          }
        }
      }
    },
    ...
  ]
}
```

- 第二级的折叠不支持 `inner_hits` 配置

12.2 Filter search results过滤搜索结果

可以使用两种方法筛选搜索结果：

- 使用带 `filter` 子句的布尔查询。搜索请求对搜索命中和 聚合 应用 布尔过滤器 (`boolean filter`)。
- 使用搜索API的 `post_filter` 参数。搜索请求仅对搜索命中应用 后期筛选，而不是聚合。您可以使用后过滤器根据更广泛的结果集计算聚合，然后进一步缩小结果范围。

您还可以在后过滤器之后重新扫描命中，以提高相关性并重新排序结果。

展开折叠结果

也可以使用 `inner_hits` 选项展开每个折叠的顶部命中。

```
GET /my-index-000001/_search
{
  "query": {
    "match": {
      "message": "GET /search"
    }
  },
  "collapse": {
    "field": "user.id", ①
    "inner_hits": {
      "name": "most_recent", ②
      "size": 5, ③
      "sort": [ { "@timestamp": "asc" } ] ④
    },
    "max_concurrent_group_searches": 4 ⑤
  },
  "sort": [ "http.response.bytes" ]
}
```

①使用 “`user.id`” 字段折叠结果集合 ②响应中用于内部命中部分的名称 ③每个折叠键要检索的 `inner_hits` ④如何在每个组内对文档进行排序 ⑤允许为每个组检索 `inner_hits` 的并发请求数

有关支持的选项和响应格式的完整列表，请参见[内部命中](#)。

也可以为每个折叠命中请求多个 `inner_hits`。当您想要获得折叠命中中的多个表示时，这可能很有用。

```

GET /my-index-000001/_search
{
  "query": {
    "match": {
      "message": "GET /search"
    }
  },
  "collapse": {
    "field": "user.id",
    "inner_hits": [
      {
        "name": "largest_responses",
        "size": 3,
        "sort": [ "http.response.bytes" ]
      },
      {
        "name": "most_recent",
        "size": 3,
        "sort": [ { "@timestamp": "asc" } ]
      }
    ],
    "sort": [ "http.response.bytes" ]
  }
}

```

组的扩展是通过为每个 `inner_hit` 的请求的响应中返回的每个折叠的命中发送一个额外的查询来完成的。如果有太多的组和/或``inner_hit``请求，这会大大降低速度。

`max_concurrent_group_searches` 请求参数可用于控制此阶段允许的最大并发搜索数。默认值基于数据节点的数量和默认的搜索线程池大小。

第二级折叠

第二级折叠也是支持的，并被应用于 `inner_hits`。

例如，以下搜索按以下方式折叠结果：`geo.country_name`。在每个 `geo.country_name` 中，内部命中通过 `user.id` 折叠。

```

GET /my-index-000001/_search
{
  "query": {
    "match": {
      "message": "GET /search"
    }
  },
  "collapse": {
    "field": "geo.country_name",
    "inner_hits": {
      "name": "by_location",
      "collapse": { "field": "user.id" },
      "size": 3
    }
  }
}

```

响应：

```
{
  ...
  "hits": [
    {
      "_index": "my-index-000001",
      "_type": "_doc",
      "_id": "9",
      "_score": ...,
      "_source": {...},
      "fields": { "geo": { "country_name": [ "UK" ] } },
      "inner_hits": {
        "by_location": {
          "hits": {
            ...
            "hits": [
              {
                ...
                "fields": { "user": "id": { [ "user124" ] } }
              },
              {
                ...
                "fields": { "user": "id": { [ "user589" ] } }
              },
              {
                ...
                "fields": { "user": "id": { [ "user001" ] } }
              }
            ]
          }
        }
      }
    },
    {
      "_index": "my-index-000001",
      "_type": "_doc",
      "_id": "1",
      "_score": ...,
      "_source": {...},
      "fields": { "geo": { "country_name": [ "Canada" ] } },
      "inner_hits": {
        "by_location": {
          "hits": {
            ...
            "hits": [
              {
                ...
                "fields": { "user": "id": { [ "user444" ] } }
              },
              {
                ...
                "fields": { "user": "id": { [ "user1111" ] } }
              },
              ...
              "fields": { "user": "id": { [ "user999" ] } }
            ]
          }
        }
      }
    },
    ...
  ]
}
```

- 第二级的折叠不支持 `inner_hits` 配置

12.3 Highlighting 高亮

高亮显示使您能够从搜索结果中的一个或多个字段中获取高亮显示的代码段，以便向用户显示查询匹配的位置。当您请求高亮显示时，响应为每个搜索命中包含一个额外的 `highlight` 元素，该元素包括突出显示的字段和突出显示的片段。

- 在提取要高亮显示的术语时，高亮显示不反映查询的布尔逻辑。因此，对于一些复杂的布尔查询（例如嵌套布尔查询、使用 `minimum_should_match` 的查询等），可能会高亮显示文档中与查询匹配不对应的部分。

高亮显示需要字段的实际内容。如果未存储字段（映射未将 `store` 设置为 `true`），则将加载实际的 `_source` 并从 `_source` 提取相关字段。

例如，要使用默认高亮对每个搜索命中获取 `content` 字段的高亮显示，请在请求体中包含一个指定 `content` 字段的 `highlight` 对象：

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

Elasticsearch 支持三种类型的高亮显示：`unified`、`plain`、`fvh`（快速矢量高亮）。你可以为每个字段指定你想要用的高亮显示 `type`。

Unified 高亮器

`unified` 高亮器使用 Lucene Unified 高亮。这个高亮将文本分成句子，并使用 BM25 算法对单个句子进行评分，就好像它们是语料库中的文档一样。它还支持精确短语和多术语（fuzzy、prefix、regex）高亮显示。这是默认的高亮显示。

Plain 高亮器

`plain` 高亮器使用标准的 Lucene 高亮。它试图通过理解短语查询中的单词重要性和任何单词定位标准来反映查询匹配逻辑。

- `plain` 高亮器最适合在单个字段中高亮显示简单查询匹配项。为了准确地反映查询逻辑，它创建了一个微小的内存索引，并通过 Lucene 的查询执行计划器重新运行原始查询条件，以获取当前文档的低级匹配信息。对于需要突出显示的每个字段和每个文档，都会重复此操作。如果您想突出显示具有复杂查询的大量文档中的许多字段，建议在 `postings` 或 `term_vector` 字段上使用 `unified` 高亮显示。

Fast vector 高亮器

`fvh` 高亮器使用Lucene快速矢量高亮。此高亮可用于映射中 `term_vector` 设置为 `with_positions_offsets` 的字段。

快速矢量高亮器：

- 可以用 `boundary_scanner` 定制。
- 需要将 `term_vector` 设置为 `with_positions_offsets`，这会增加索引的大小
- 可以将多个字段中的匹配项合并为一个结果。请参见 `matched_fields`
- 可以为不同位置的匹配分配不同的权重，这样就可以在高亮显示增强查询时，将短语匹配排序在术语匹配之上，从而增强术语匹配的短语匹配

设置高亮器类型

`type` 字段允许强制指定一个高亮器类型。可以配置的值有 `unified`, `plain`, `fvh`。以下是一个强制使用`plain`高亮器的范例。

```
{
  "query": {
    "match": { "user.id": "kimchy" }
  },
  "highlight": {
    "fields": {
      "comment": { "type": "plain" }
    }
  }
}
```

配置高亮标签

默认高亮显示会把高亮文本用 `` 和 `` 标签包住。可以配置 `pre_tags` 和 `post_tags` 进行控制，例如：

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

当使用快速矢量高亮器时，可以指定附加标签，并且按优先级排序

```
GET /_search
{
  "query" : {
    "match": { "user.id": "kimchy" }
  },
  "highlight" : {
    "pre_tags" : ["<tag1>", "<tag2>"],
    "post_tags" : ["</tag1>", "</tag2>"],
    "fields" : {
      "body" : {}
    }
  }
}
```

也可以使用内置的 `style` 标签架构

```
GET /_search
{
  "query" : {
    "match": { "user.id": "kimchy" }
  },
  "highlight" : {
    "tags_schema" : "styled",
    "fields" : {
      "comment" : {}
    }
  }
}
```

明确排序高亮字段

Elasticsearch按字段发送的顺序高亮显示字段，但根据JSON规范，对象是无序的。如果需要明确字段高亮显示的顺序，请将 `fields` 指定为数组：

```
GET /_search
{
  "highlight" : {
    "fields": [
      { "title": {} },
      { "text": {} }
    ]
  }
}
```

Elasticsearch内置的高亮显示工具都不关心字段高亮显示的顺序，但插件可能会。

unified 高亮器 使用范例

让我们更详细地了解一下 unified高亮 是如何工作的。

首先，我们创建一个包含文本字段 `content` 的索引，该索引将使用 `english` 分词器进行索引，并且将不使用偏移量或术语向量进行索引。

```
PUT test_index
{
  "mappings": {
    "properties": {
      "content": {
        "type": "text",
        "analyzer": "english"
      }
    }
  }
}
```

我们将下列文件加入索引：

```
PUT test_index/_doc/doc1
{
  "content" : "For you I'm only a fox like a hundred thousand other foxes."
}
```

然后我们执行下列带有高亮请求的查询：

```
GET test_index/_search
{
  "query": {
    "match_phrase" : {"content" : "only fox"}
  },
  "highlight": {
    "type" : "unified",
    "number_of_fragments" : 3,
    "fields": {
      "content": {}
    }
  }
}
```

在找到 doc1 作为此查询的命中后，此命中将传递给unified高亮显示器以高亮显示文档的 content 字段。由于 content 字段未使用偏移量或术语向量索引，因此将分析其原始字段值，并根据与查询匹配的术语构建内存索引：

```
{"token":"only","start_offset":12,"end_offset":16,"position":3},
 {"token":"fox","start_offset":19,"end_offset":22,"position":5},
 {"token":"fox","start_offset":53,"end_offset":58,"position":11},
 {"token":"only","start_offset":117,"end_offset":121,"position":24},
 {"token":"only","start_offset":159,"end_offset":163,"position":34},
 {"token":"fox","start_offset":164,"end_offset":167,"position":35}
```

我们的复杂短语查询将转换为span查询： `spanNear([text:only, text:fox], 0, true)`，这意味着我们要查找的术语“only”和“fox”之间的距离不超过0，并且按给定的顺序。span查询将针对内存中之前创建的索引运行，以查找以下匹配项：

```
{"term":"only", "start_offset":159, "end_offset":163},
 {"term":"fox", "start_offset":164, "end_offset":167}
```

在我们的示例中，只有一个匹配项，但可能有多个匹配项。给定匹配项，unified高亮显示器将字段的文本分成所谓的“段落”。每个段落必须至少包含一个匹配项。unified高亮显示器使用Java的 BreakIterator 确保每个段落都代表一个完整的句子，只要它不超过 fragment_size 。对于我们的示例，我们得到了一个具有以下属性的单独段落（这里仅显示属性的子集）：

```
Passage:  
  startOffset: 147  
  endOffset: 189  
  score: 3.7158387  
  matchStarts: [159, 164]  
  matchEnds: [163, 167]  
  numMatches: 2
```

注意一篇段落是如何被评分的，这个分数是用适合段落的BM25评分公式计算出来的。分数允许我们选择最好的评分段落，如果有段落比用户所要求的 `number_of_fragments` 多。如果用户要求，分数还允许我们按 `order: "score"` 对段落进行排序

最后一步，unified高亮器将从字段文本中提取对应于每个段落的字符串：

```
"I' ll be the only fox in the world for you."
```

将对此字符串中的所有匹配项使用标记和格式化，参照段落的 `matchStart` 和 `matchEnds` 信息：

```
I' ll be the <em>only</em> <em>fox</em> in the world for you.
```

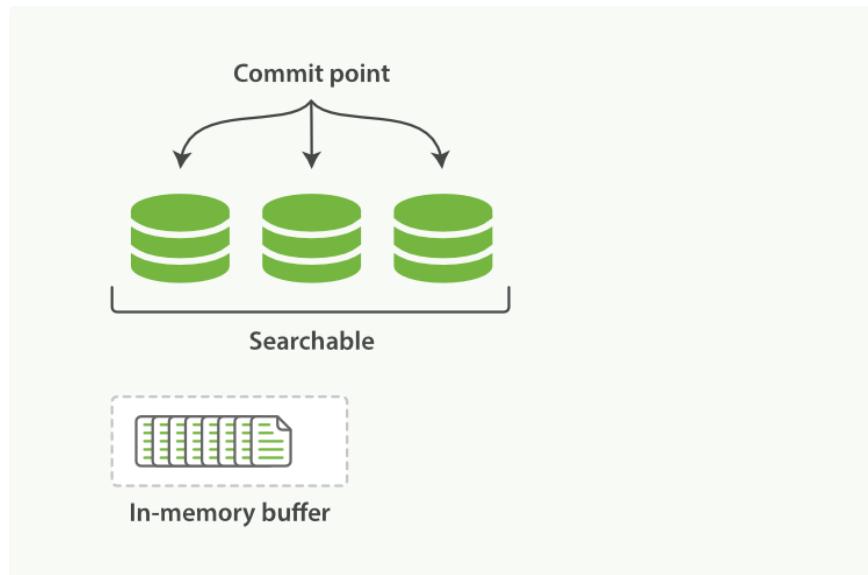
准实时搜索

对[文档和索引](#)的概述表明，当文档存储在Elasticsearch中时，它会被编入索引，并在1秒内几乎实时地完全搜索。什么定义了准实时搜索？

Elasticsearch所基于的Java库Lucene引入了每段搜索的概念。段类似于倒排索引，但Lucene中的*index*一词的意思是“段的集合加上一个提交点”。提交之后，一个新的段被添加到提交点，缓冲区被清除。

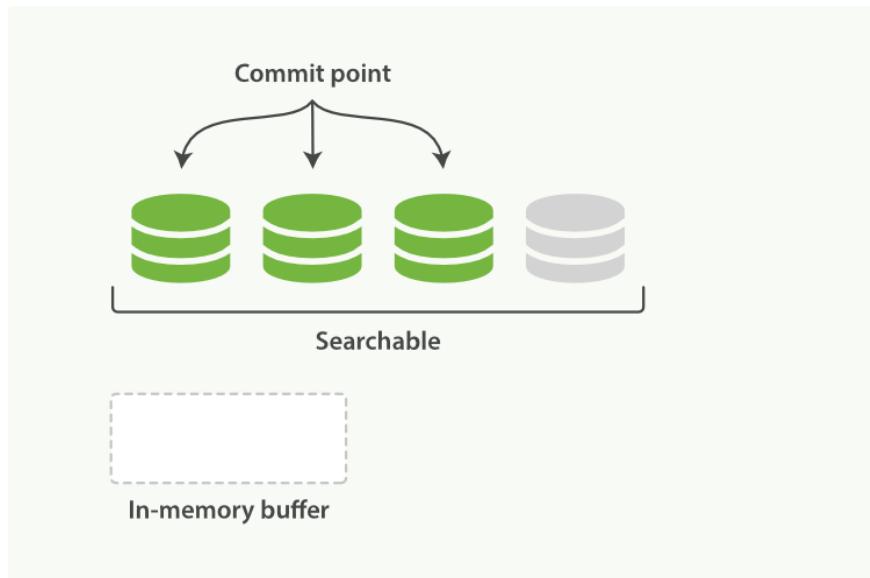
位于Elasticsearch和磁盘之间的是文件系统缓存。内存索引缓冲区（图1）中的文档被写入一个新段（图2）。新段首先写入文件系统缓存（这很便宜），然后才刷新到磁盘（这很昂贵）。但是，当一个文件在缓存中之后，它可以像其他任何文件一样被打开和读取。

图1. 在内存缓冲区中包含新文档的Lucene索引



Lucene允许编写和打开新的段，使它们包含的文档在不执行完全提交的情况下对搜索可见。这是一个比提交到磁盘轻得多的过程，并且可以经常执行而不会降低性能。

图2. 缓冲区内容被写入一个段，该段是可搜索的，但尚未提交



在Elasticsearch中，写入和打开新段的过程称为刷新。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。您可以通过以下方式控制刷新：

- 等待刷新间隔
- 设置[?refresh](#)选项
- 使用[Refresh API](#)显式完成刷新（`POST _refresh`）

默认情况下，Elasticsearch每秒定期刷新索引，但仅对在过去30秒内收到一个或多个搜索请求的索引进行刷新。这就是为什么我们说Elasticsearch具有近乎实时的搜索：文档更改对于搜索来说不是立即可见的，而是在这个时间范围内可见的。

可伸缩性和弹性：集群、节点和分片

Elasticsearch始终可用，并可根据您的需要进行扩展。它是通过自然分配来实现的。您可以将服务器（节点）添加到集群以增加容量，Elasticsearch会自动将数据和查询负载分布到所有可用节点。无需大修应用程序，Elasticsearch知道如何平衡多节点群集以提供规模和高可用性。节点越多越好。

这是怎么回事？实际上，Elasticsearch索引只是一个或多个物理分片的逻辑分组，其中每个分片实际上是一个自包含索引。通过将索引中的文档分布在多个分片上，并将这些分片分布在多个节点上，Elasticsearch可以确保冗余，这既可以防止硬件故障，又可以在节点添加到集群时增加查询容量。随着集群的增长（或收缩），Elasticsearch会自动迁移分片以重新平衡集群。

分片有两种类型：主碎片和副本分片。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。副本提供了数据的冗余副本，以防止硬件故障，并增加了服务于读取请求（如搜索或检索文档）的容量。

索引中主分片的数量在创建索引时是固定的，但副本分片的数量可以随时更改，而不中断索引或查询操作。

这取决于

对于分片大小和为索引配置的主分片的数量，有许多性能方面的考虑和权衡。分片越多，维护这些索引的开销就越大。当Elasticsearch需要重新平衡集群时，分片大小越大，移动分片的时间就越长。

查询大量的小分片可以加快每个分片的处理速度，但是查询越多意味着开销就越大，因此查询较少数量的较大分片可能会更快。简言之...视情况而定。

作为起点：

- 目的将平均分片大小保持在几GB到几十GB之间。对于使用基于时间的数据的用例，通常可以看到20GB到40GB范围内的分片。
- 避免无数分片的问题。节点可以容纳的碎片数量与可用堆空间成比例。一般来说，每GB堆空间的碎片数应小于20。

为您的用例确定最佳配置的最佳方法是[使用您自己的数据和查询进行测试](#)。

万一发生灾难

出于性能原因，集群中的节点需要在同一网络上。在不同数据中心的节点之间平衡集群中的碎片需要很长时间。但是高可用性架构要求您避免将所有的鸡蛋放在一个篮子里。在一个位置发生重大停机的情况下，另一个位置的服务器需要能够接管。天衣无缝。答案是什么？跨群集复制（CCR）。

CCR提供了一种将索引从主集群自动同步到可以用作热备份的辅助远程集群的方法。如果主集群出现故障，则辅助集群可以接管。您还可以使用CCR创建辅助集群，以便在地理位置接近您的用户的情况下为读取请求提供服务。

跨群集复制是主动-被动的。主集群上的索引是活动的领导索引，处理所有写请求。复制到辅助群集的索引是只读的跟随者。

护理和保养

与任何企业系统一样，您需要工具来保护、管理和监控您的Elasticsearch集群。集成到Elasticsearch中的安全、监视和管理功能使您能够将[Kibana](#)用作管理集群的控制中心。[data rollups](#)和[index lifecycle management](#)等功能可帮助您随着时间的推移智能地管理数据。

Query DSL

Elasticsearch提供了基于JSON的完整查询DSL（域特定语言）来定义查询。将查询DSL看作是一个AST（抽象语法树）查询，由两种类型的子句组成：

叶查询从句

叶查询从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。

复合查询从句

复合查询子句包装其他叶查询或复合查询，用于以逻辑方式组合多个查询（如 `bool` 或 `dis_max` 查询），或更改其行为（如 `constant_score` 查询）。

查询从句的行为不同，这取决于它们是在[查询上下文或筛选器上下文](#)中使用。

允许昂贵的查询

由于实现方式的不同，某些类型的查询通常执行速度较慢，这会影响集群的稳定性。这些查询可以分类如下：

- 需要进行线性扫描以识别匹配项的查询：
 - `script queries`
- 前期成本高的查询
 - `fuzzy queries` (`wildcard` 字段除外)
 - `regexp queries` (`wildcard` 字段除外)
 - `prefix queries` (`wildcard` 字段除外或没有 `index_prefixes`)
 - `wildcard queries` (`wildcard` 字段除外)
 - `range queries` on <>`text, text and keyword fields`
- `Joining queries`
- `deprecated geo shapes` 上的查询
- 每文档成本较高的查询：
 - `script score queries`
 - `percolate queries`

通过设置 `search.allow_expensive_queries` 设置为 `false` 可以阻止这些查询的执行（默认为 `true`）。

13.1 Query and filter context 查询和筛选上下文

相关性得分

默认情况下， Elasticsearch按相关性得分对匹配的搜索结果进行排序， 相关性得分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， 返回到search API的 `_score` 元数据字段中。 `_score` 越高， 文档就越相关。 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上下文中运行还是在筛选上下文中运行。

查询上下文

在查询上下文中， 查询子句回答“此文档与此查询子句的匹配程度如何？”， 除了决定文档是否匹配外， 查询子句还在 `_score` 元数据字段中计算相关性分数。

每当查询子句被传递给 `query` 参数（例如search API中的 `query` 参数）时， 查询上下文就生效。

筛选上下文

在筛选上下文中， 查询子句需要回答问题“此文档是否与此查询子句匹配？” 答案是简单的是或否 - 不计算分数。 筛选上下文主要用于筛选结构化数据， 例如。

- 这个 `timestamp` 是否属于2015年到2016年的范围？
- `status` 字段是否设置为 `published` ?

Elasticsearch会自动缓存常用筛选器， 以提高性能。

每当查询子句被传递给筛选器参数时， 筛选器上下文就生效， 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。

查询和筛选上下文示例

下面是在搜索API的查询和筛选上下文中使用的查询子句的示例。此查询将匹配满足以下所有条件的文档：

- `title` 字段包含文字 `search` 。
- `content` 字段包含单词 `elasticsearch` 。
- `status` 字段包含确切的文字 `published` 。
- `publish_date` 字段包含从2015年1月1日起的日期。

```

GET /_search
{
  "query": { ①
    "bool": { ②
      "must": [
        { "match": { "title": "Search" } },
        { "match": { "content": "Elasticsearch" } }
      ],
      "filter": [ ③
        { "term": { "status": "published" } },
        { "range": { "publish_date": { "gte": "2015-01-01" } } }
      ]
    }
  }
}

```

① query 参数表示查询上下文。

② bool 和两个 match 子句用于查询上下文，这意味着它们用于对每个文档的匹配程度进行评分。

③ filter 参数表示筛选上下文。它的 term 和 range 子句用于筛选上下文。它们会筛选掉不匹配的文档，但不会影响匹配文档的分数。

- 为查询上下文中的查询计算的分数表示为单精度浮点数；它们的有效位精度只有24位。超过有效位精度的分数计算将转换为精度损失的浮点值。
- 在查询上下文中使用查询子句来处理应该影响匹配文档得分的条件（即文档匹配的程度），并在筛选上下文中使用所有其他查询子句。

13.2 Compound queries 复合查询

复合查询包括其他复合查询或叶查询，以组合其结果和分数，更改其行为，或从查询切换到筛选上下文。

此组中的查询包括：

bool query 布尔查询

- 用于组合多个叶或复合查询子句（例如 `must`、`should`、`must_not` 或 `filter` 子句）的默认查询。`must` 和 `should` 子句的分数相加，匹配的子句越多越好，而 `must_not` 和 `filter` 子句在筛选器上下文中执行。

boosting query 增强查询

- 返回与 `positive` 查询匹配的文档，但同时减少与 `negative` 查询匹配的文档的分数。

constant_score query 常量分数查询

- 包装另一个查询，但在筛选器上下文中执行它的查询。所有匹配的文档都会得到相同的“常量” `_score`。

dis_max查询

- 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` 查询组合了所有匹配查询的分数，`dis_max` 查询使用单个最佳匹配查询子句的分数。

function_score query 功能评分查询

- 使用函数修改主查询返回的分数，以考虑流行度、最近度、距离或脚本实现的自定义算法等因素。

13.2.1 Boolean Query 布尔查询

一种查询，它匹配与其他查询的布尔组合相匹配的文档。bool查询对应到Lucene中的 BooleanQuery 。它是使用一个或多个布尔子句构建的，每个子句都有一个类型化的引用。相关类型为：

标记	描述
must	子句 (query) 必须出现在匹配的文档中，并且将对得分起作用
filter	子句 (query) 必须出现在匹配的文档中。但是不同于 must ，查询的分数将被忽略。Filter子句是在 Filter上下文 中执行的，这意味着评分被忽略，子句被考虑用于缓存。
should	子句 (query) 应该出现在匹配的文档中。
must_not	子句 (query) 不能出现在匹配的文档中。子句是在 Filter上下文 中执行的，这意味着评分被忽略，子句被考虑用于缓存。因为评分被忽略，所以返回所有文档为 0 分。

bool 查询采用了一种“匹配越多越好”的方法，因此每个匹配 must 或 should 子句的分数将被添加到一起，以提供每个文档的最终 _score 。

```
POST /_search
{
  "query": {
    "bool": {
      "must": [
        {"term": { "user.id": "kimchy" }}
      ],
      "filter": [
        {"term": { "tags": "production" }}
      ],
      "must_not": [
        {"range": {
          "age": { "gte": 10, "lte": 20 }
        }}
      ],
      "should": [
        { "term": { "tags": "env1" } },
        { "term": { "tags": "deployed" } }
      ],
      "minimum_should_match": 1,
      "boost": 1.0
    }
  }
}
```

使用最小匹配 minimum_should_match

您可以使用 minimum_should_match 参数指定返回的文档必须匹配的 should 子句的数目或百分比。

如果 bool 查询至少包含一个 should 子句，并且没有 must 或 filter 子句，则默认值为 1 。否则，默认值为 0 。

有关其他有效值, 请参阅 [minimum_should_match 参数](#)。

使用 `bool.filter` 算分

在 `filter` 元素下指定的查询对评分没有影响 - 分数返回为 `0`。分数只受指定的查询影响。例如, 以下三个查询都返回 `status` 字段包含术语 `active` 的所有文档。

第一个查询为所有文档分配 `0` 分, 因为没有指定评分查询:

```
GET _search
{
  "query": {
    "bool": {
      "filter": {
        "term": {
          "status": "active"
        }
      }
    }
  }
}
```

这个 `bool` 查询有一个 `match_all` 查询, 它为所有文档分配 `1.0` 的分数。

```
GET _search
{
  "query": {
    "bool": {
      "must": {
        "match_all": {}
      },
      "filter": {
        "term": {
          "status": "active"
        }
      }
    }
  }
}
```

这个 `constant_score` 查询的行为方式与上面的第二个示例完全相同。`constant_score` 查询为所有与筛选器匹配的文档分配 `1.0` 的分数。

```
GET _search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "status": "active"
        }
      }
    }
  }
}
```

命名查询

每个查询在其顶级定义中接受一个 `_name`。您可以使用命名查询来跟踪哪些查询与返回的文档匹配。如果使用命名查询，则响应将为每个命中包含 `matched_queries` 属性。

```
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"
        }
      }
    }
  }
}
```

13.2.2 Boosting query 增强查询

返回与 `positive` 查询匹配的文档，同时降低与 `negative` 查询匹配的文档的[相关性得分](#)。

您可以使用 `boosting` 查询降级某些文档，而不将其从搜索结果中排除。

示例请求

```
GET /_search
{
  "query": {
    "boosting": {
      "positive": {
        "term": {
          "text": "apple"
        }
      },
      "negative": {
        "term": {
          "text": "pie tart fruit crumble tree"
        }
      },
      "negative_boost": 0.5
    }
  }
}
```

`boosting` 的顶级参数

`positive`

- (必需，查询对象) 要运行的查询。所有返回的文档必须与此查询匹配。

`negative`

- (必选，查询对象) 用于降低匹配文档相关性得分的查询。

如果返回的文档与 `positive` 查询和此查询匹配，则 `boosting` 查询将按如下方式计算文档的最终相关性分数：

1. 从 `positive` 查询中获取原始的[相关性得分](#)。
- 2 将分数乘以 `negative_boost`。

`negative_boost`

(必选，浮点) 介于 `0` 和 `1.0` 之间的浮点数，用于降低与 `negative` 查询匹配的文档的[相关性得分](#)。

13.2.3 Constant score query 固定分数查询

包装[过滤器查询](#)并返回每个匹配文档，其[相关性得分](#)等于 `boost` 参数值。

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

constant_score 的顶级参数

filter

(必需，查询对象) 要运行的[筛选器查询](#)。所有返回的文档必须与此查询匹配。

筛选查询不计算[相关性分数](#)。为了提高性能，Elasticsearch会自动缓存常用的筛选器查询。

boost

(可选，float) 浮点数，用作与 `filter` 查询匹配的每个文档的常量[相关分数](#)。默认认为 `1.0`。

13.2.3 Disjunction max query

返回与一个或多个包装查询（称为查询子句或子句）匹配的文档。

如果返回的文档与多个查询子句匹配，`dis_max` 查询将为文档分配任何匹配子句中的最高相关性分数，加上任何其他匹配子查询的中断增量。

您可以使用 `dis_max` 在映射了不同`boost`因子的字段中搜索一个词语。

示例请求

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

`dis_max` 的顶级参数

queries

(必需，查询对象数组) 包含一个或多个查询子句。返回的文档必须与这些查询中的一个或多个匹配。如果一个文档匹配多个查询，Elasticsearch将使用最高的相关性分数。

tie_breaker

(可选，float) 介于0和1.0之间的浮点数，用于增加匹配多个查询子句的文档的相关性得分。默认为0.0。

对于在多个字段中包含同一术语的文档，可以使用`tie_breaker`值为其分配比仅在这些字段中最好的字段中包含该术语的文档更高的相关性分数，而不会将其与在多个字段中包含两个不同术语的更好情况相混淆。

如果文档匹配多个子句，`dis_max`查询将按以下方式计算文档的相关性分数：

从得分最高的匹配从句中获取相关性得分。

将任何其他匹配子句的得分乘以平局破坏者值。

把最高的分数加到相乘的分数上。

如果`tie_breaker`值大于0.0，则所有匹配子句都计算在内，但得分最高的子句最为重要。

13.1 Query and filter context 查询和筛选上下文

相关性得分

默认情况下， Elasticsearch按相关性得分对匹配的搜索结果进行排序， 相关性得分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， 返回到search API的 `_score` 元数据字段中。 `_score` 越高， 文档就越相关。 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上下文中运行还是在筛选上下文中运行。

查询上下文

在查询上下文中， 查询子句回答“此文档与此查询子句的匹配程度如何？”， 除了决定文档是否匹配外， 查询子句还在 `_score` 元数据字段中计算相关性分数。

每当查询子句被传递给 `query` 参数（例如search API中的 `query` 参数）时， 查询上下文就生效。

筛选上下文

在筛选上下文中， 查询子句需要回答问题“此文档是否与此查询子句匹配？” 答案是简单的是或否 - 不计算分数。 筛选上下文主要用于筛选结构化数据， 例如。

- 这个 `timestamp` 是否属于2015年到2016年的范围？
- `status` 字段是否设置为 `published` ?

Elasticsearch会自动缓存常用筛选器， 以提高性能。

每当查询子句被传递给筛选器参数时， 筛选器上下文就生效， 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。

查询和筛选上下文示例

下面是在搜索API的查询和筛选上下文中使用的查询子句的示例。此查询将匹配满足以下所有条件的文档：

- `title` 字段包含文字 `search` 。
- `content` 字段包含单词 `elasticsearch` 。
- `status` 字段包含确切的文字 `published` 。
- `publish_date` 字段包含从2015年1月1日起的日期。

```

GET /_search
{
  "query": { ①
    "bool": { ②
      "must": [
        { "match": { "title": "Search" } },
        { "match": { "content": "Elasticsearch" } }
      ],
      "filter": [ ③
        { "term": { "status": "published" } },
        { "range": { "publish_date": { "gte": "2015-01-01" } } }
      ]
    }
  }
}

```

① query 参数表示查询上下文。

② bool 和两个 match 子句用于查询上下文，这意味着它们用于对每个文档的匹配程度进行评分。

③ filter 参数表示筛选上下文。它的 term 和 range 子句用于筛选上下文。它们会筛选掉不匹配的文档，但不会影响匹配文档的分数。

- 为查询上下文中的查询计算的分数表示为单精度浮点数；它们的有效位精度只有24位。超过有效位精度的分数计算将转换为精度损失的浮点值。
- 在查询上下文中使用查询子句来处理应该影响匹配文档得分的条件（即文档匹配的程度），并在筛选上下文中使用所有其他查询子句。

13.3 全文本查询

全文本查询使您能够搜索被分词的文本字段，如电子邮件正文。查询字符串使用应用于字段索引期间的同一分词器处理。

此组中的查询包括：

`intervals` 查询

允许对匹配项的顺序和接近度进行细粒度控制的全文查询。

`match` 查询

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` 查询

创建一个 `bool` 查询，将每个词语作为 `term` 查询进行匹配，但最后一个术语作为 `prefix` 查询进行匹配

`match_phrase` 查询

与 `match` 查询类似，但用于匹配精确的短语或单词邻近匹配。

`match_phrase_prefix` 查询

与 `match_phrase` 查询类似，但对最后一个单词进行通配符搜索。

`multi_match` 查询

`match` 查询的多字段版本。

`common` 词语查询

一种更专门的查询，它更倾向于不常用的词。

`query_string` 查询

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` 查询

`query_string` 语法的一个更简单、更健壮的版本，适合直接向用户公开。

13.3.1 Intervals query 间隔查询

根据匹配项的顺序和接近程度返回文档。

`intervals` 查询使用匹配规则，这些规则是由一小部分定义构成的。然后将这些规则应用于指定 `field` 中的词语。

这些定义产生了最小间隔的序列，这些间隔跨越了文本体中的术语。这些间隔可以由父源进一步组合和过滤。

请求示例

以下 `intervals` 搜索返回包含 `my favorite food` 的文档，没有任何空隙，同时在 `my_text` 字段中包含 `hot water` 或 `cold porridge`。

此搜索将匹配值为 `my favorite food is cold porridge` 的 `my_text` 字段，而不会匹配 `when it's cold my favorite food is porridge`。

```
POST _search
{
  "query": {
    "intervals": {
      "my_text": {
        "all_of": [
          {
            "ordered": true,
            "intervals": [
              {
                "match": {
                  "query": "my favorite food",
                  "max_gaps": 0,
                  "ordered": true
                }
              },
              {
                "any_of": {
                  "intervals": [
                    {
                      "match": {
                        "query": "hot water"
                      }
                    },
                    {
                      "match": {
                        "query": "cold porridge"
                      }
                    }
                  ]
                }
              }
            ]
          }
        }
      }
    }
}
```

intervals 的顶级参数

(必需，规则对象) 要搜索的字段。

此参数的值是一个规则对象，用于根据匹配项、顺序和接近度匹配文档。

有效规则包括：

- `match`

- prefix
- wildcard
- fuzzy
- all_of
- any_of

match 规则参数

`match` 规则匹配分词后的文本。

`query`

(必需, 字符串) 您希望在提供的 `<field>` 中找到的文本。

`max_gaps`

(可选, 整数) 匹配词语之间的最大位置数。相距较远的词语不视为匹配项。默认为 `-1`。

如果未指定或设置为 `-1`, 则匹配项上没有宽度限制。如果设置为 `0`, 则词语必须相邻出现。

`ordered`

(可选, 布尔值) 如果为 `true`, 则匹配项必须按指定顺序出现。默认为 `false`。

`analyzer`

(可选, 字符串) 分词器(analyzer), 用于分词查询中的词语。默认为顶级 `<field>` 的分词器。

`filter`

(可选, 间隔筛选器(interval filter)规则对象) 可选的间隔筛选器。

`use_field`

(可选, 字符串) 如果指定, 则匹配来自此字段而不是顶级 `<field>` 的间隔。使用搜索分词器从此字段分析词语。这允许您跨多个字段进行搜索, 就像它们都是同一个字段一样; 例如, 您可以将相同的文本索引到带词干和未带词干的字段中, 并在未带词干的字段附近搜索带词干的标记。

prefix 规则参数

`prefix` 规则匹配以指定字符集开头的词语。此前缀最多可扩展到匹配128个词语。如果前缀匹配超过128个词, Elasticsearch将返回一个错误。可以在字段映射中使用`index prefixes`选项来避免此限制。

`prefix`

(必需, 字符串) 希望在顶级 `<field>` 中找到的词语的开头字符。

`analyzer`

(可选, 字符串) 用于规范化 `prefix` 的`analyzer`。默认为顶级 `<field>` 的分析器。

`use_field`

(可选, 字符串) 如果指定, 则匹配来自此字段而不是顶级 `<field>` 的间隔。

除非指定了单独的 `analyzer`，否则使用此字段中的搜索分析器对 `prefix` 进行规范化。

wildcard 规则参数

The wildcard rule matches terms using a wildcard pattern. This pattern can expand to match at most 128 terms. If the pattern matches more than 128 terms, Elasticsearch returns an error.

`pattern`

(Required, string) Wildcard pattern used to find matching terms.

This parameter supports two wildcard operators:

`?`, which matches any single character , *which can match zero or more characters, including an empty one* Avoid beginning patterns with `or ?`. This can increase the iterations needed to find matching terms and slow search performance.

`analyzer`

(Optional, string) analyzer used to normalize the pattern. Defaults to the top-level 's analyzer.

`use_field`

(Optional, string) If specified, match intervals from this field rather than the top-level .

The pattern is normalized using the search analyzer from this field, unless analyzer is specified separately.

fuzzy 规则参数

`fuzzy` 规则在 [Fuzziness](#) 定义的编辑距离内匹配与提供的词语相似的词语。如果模糊展开匹配超过128个词，Elasticsearch将返回一个错误。

`term`

(必选, 字符串) 要匹配的词语

`prefix_length`

(可选, 字符串) 创建扩展时保持不变的起始字符数。默认为 `0` 。

`transpositions`

(可选, 布尔值) 指示编辑是否包括两个相邻字符的换位 (`ab→ba`) 。默认为 `true` 。

`fuzziness`

(可选, 字符串) 允许匹配的最大编辑距离。有关有效值和更多信息，请参见 [Fuzziness](#) 。默认为 `auto` 。

`analyzer`

(可选, 字符串) 用于规范化 `term` 的`analyzer`。默认为顶级 `<field>` 的分词器。

`use_field`

(可选, 字符串) 如果指定, 则匹配来自此字段而不是顶级 `<field>` 的间隔。

除非单独指定了 `analyzer`, 否则使用此字段中的搜索分析器对 `term` 进行规范化。

all_of 规则参数

The `all_of` rule returns matches that span a combination of other rules.

`intervals`

(Required, array of rule objects) An array of rules to combine. All rules must produce a match in a document for the overall source to match.

`max_gaps`

(Optional, integer) Maximum number of positions between the matching terms. Intervals produced by the rules further apart than this are not considered matches. Defaults to -1.

If unspecified or set to -1, there is no width restriction on the match. If set to 0, the terms must appear next to each other.

`ordered`

(Optional, Boolean) If true, intervals produced by the rules should appear in the order in which they are specified. Defaults to false.

`filter`

(Optional, interval filter rule object) Rule used to filter returned intervals.

any_of 规则参数

`any_of` 规则返回由其任何子规则生成的间隔。

`intervals`

(必需, 规则对象数组) 要匹配的规则数组。

`filter`

(可选, 间隔筛选(interval filter)规则对象) 用于筛选返回间隔的规则。

filter 规则参数

筛选规则基于查询返回间隔。有关示例, 请参见过滤器示例。

`after`

(Optional, query object) Query used to return intervals that follow an interval from the filter rule.

`before`

(Optional, query object) Query used to return intervals that occur before an interval from the filter rule.

`contained_by`

(Optional, query object) Query used to return intervals contained by an interval from the filter rule.

`containing`

(Optional, query object) Query used to return intervals that contain an interval from the filter rule.

`not_contained_by`

(Optional, query object) Query used to return intervals that are not contained by an interval from the filter rule.

`not_containing`

(Optional, query object) Query used to return intervals that do not contain an interval from the filter rule.

`not_overlapping`

(Optional, query object) Query used to return intervals that do not overlap with an interval from the filter rule.

`overlapping`

(Optional, query object) Query used to return intervals that overlap with an interval from the filter rule.

`script`

(Optional, script object) Script used to return matching documents. This script must return a boolean value, true or false. See Script filters for an example.

过滤器示例

以下搜索包括 `filter` 规则。它返回的文档中，单词 `hot` 和 `porridge` 的位置在10个以内，而不包含单词 `salty`：

```
POST _search
{
  "query": {
    "intervals": {
      "my_text": {
        "match": {
          "query": "hot porridge",
          "max_gaps": 10,
          "filter": {
            "not_containing": {
              "match": {
                "query": "salty"
              }
            }
          }
        }
      }
    }
  }
}
```

脚本筛选器

可以使用脚本根据间隔的开始位置、结束位置和内部间隔计数来过滤间隔。以下 `filter` 脚本将 `interval` 变量与 `start`、`end` 和 `gaps` 方法一起使用：

```
POST _search
{
  "query": {
    "intervals": {
      "my_text": {
        "match": {
          "query": "hot porridge",
          "filter": {
            "script": {
              "source": "interval.start > 10 && interval.end < 20 && interval"
            }
          }
        }
      }
    }
  }
}
```

最小化

intervals查询总是最小化间隔，以确保查询可以在线性时间内运行。这有时会导致令人惊讶的结果，特别是在使用 max_gaps 限制或过滤器时。例如，以下面的查询为例，搜索短语 hot porridge 中包含的 salty

```
POST _search
{
  "query": {
    "intervals": {
      "my_text": {
        "match": {
          "query": "salty",
          "filter": {
            "contained_by": {
              "match": {
                "query": "hot porridge"
              }
            }
          }
        }
      }
    }
  }
}
```

此查询不会匹配到包含短语 hot porridge is salty porridge 的文档，因为间隔查询返回的匹配为 hot porridge，只涵盖了文档中的前两个词语，这些词语不重叠的间隔涵盖 salty。

另一个需要注意的限制是任何包含重叠子规则的规则的情况。特别是，如果其中一条规则是另一条规则的严格前缀，则较长的规则永远无法匹配，这在与max_gaps 结合使用时可能会引起意外。考虑以下查询，搜索紧跟其后的大或大坏，紧跟其后的wolf：

```
POST _search
{
  "query": {
    "intervals": {
      "my_text": {
        "all_of": {
          "intervals": [
            { "match": { "query": "the" } },
            { "any_of": {
              "intervals": [
                { "match": { "query": "big" } },
                { "match": { "query": "big bad" } }
              ] } },
            { "match": { "query": "wolf" } }
          ],
          "max_gaps": 0,
          "ordered": true
        }
      }
    }
  }
}
```

与直觉相反，这个查询不匹配的文档很大

wolf，因为中间的任何规则只会产生大的间隔-大坏的间隔比大坏的间隔长，同时从相同的位置开始，所以被最小化。在这些情况下，最好重写查询，以便在顶层显式地列出所有选项：

```
POST _search
{
  "query": {
    "intervals": {
      "my_text": {
        "any_of": [
          "intervals": [
            { "match": {
              "query": "the big bad wolf",
              "ordered": true,
              "max_gaps": 0 } },
            { "match": {
              "query": "the big wolf",
              "ordered": true,
              "max_gaps": 0 } }
          ]
        }
      }
    }
  }
}
```

13.2.1 Boolean Query 布尔查询

一种查询，它匹配与其他查询的布尔组合相匹配的文档。bool查询对应到Lucene中的 BooleanQuery 。它是使用一个或多个布尔子句构建的，每个子句都有一个类型化的引用。相关类型为：

标记	描述
must	子句 (query) 必须出现在匹配的文档中，并且将对得分起作用
filter	子句 (query) 必须出现在匹配的文档中。但是不同于 must ，查询的分数将被忽略。Filter子句是在 Filter上下文 中执行的，这意味着评分被忽略，子句被考虑用于缓存。
should	子句 (query) 应该出现在匹配的文档中。
must_not	子句 (query) 不能出现在匹配的文档中。子句是在 Filter上下文 中执行的，这意味着评分被忽略，子句被考虑用于缓存。因为评分被忽略，所以返回所有文档为 0 分。

bool 查询采用了一种“匹配越多越好”的方法，因此每个匹配 must 或 should 子句的分数将被添加到一起，以提供每个文档的最终 _score 。

```
POST /_search
{
  "query": {
    "bool": {
      "must": [
        {"term": { "user.id": "kimchy" }}
      ],
      "filter": [
        {"term": { "tags": "production" }}
      ],
      "must_not": [
        {"range": {
          "age": { "gte": 10, "lte": 20 }
        }}
      ],
      "should": [
        { "term": { "tags": "env1" } },
        { "term": { "tags": "deployed" } }
      ],
      "minimum_should_match": 1,
      "boost": 1.0
    }
  }
}
```

使用最小匹配 minimum_should_match

您可以使用 minimum_should_match 参数指定返回的文档必须匹配的 should 子句的数目或百分比。

如果 bool 查询至少包含一个 should 子句，并且没有 must 或 filter 子句，则默认值为 1 。否则，默认值为 0 。

有关其他有效值, 请参阅 [minimum_should_match 参数](#)。

使用 `bool.filter` 算分

在 `filter` 元素下指定的查询对评分没有影响 - 分数返回为 `0`。分数只受指定的查询影响。例如, 以下三个查询都返回 `status` 字段包含术语 `active` 的所有文档。

第一个查询为所有文档分配 `0` 分, 因为没有指定评分查询:

```
GET _search
{
  "query": {
    "bool": {
      "filter": {
        "term": {
          "status": "active"
        }
      }
    }
  }
}
```

这个 `bool` 查询有一个 `match_all` 查询, 它为所有文档分配 `1.0` 的分数。

```
GET _search
{
  "query": {
    "bool": {
      "must": {
        "match_all": {}
      },
      "filter": {
        "term": {
          "status": "active"
        }
      }
    }
  }
}
```

这个 `constant_score` 查询的行为方式与上面的第二个示例完全相同。`constant_score` 查询为所有与筛选器匹配的文档分配 `1.0` 的分数。

```
GET _search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "status": "active"
        }
      }
    }
  }
}
```

命名查询

每个查询在其顶级定义中接受一个 `_name`。您可以使用命名查询来跟踪哪些查询与返回的文档匹配。如果使用命名查询，则响应将为每个命中包含 `matched_queries` 属性。

```
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"
        }
      }
    }
  }
}
```

13.2.1 Boolean Query 布尔查询

一种查询，它匹配与其他查询的布尔组合相匹配的文档。bool查询对应到Lucene中的 BooleanQuery 。它是使用一个或多个布尔子句构建的，每个子句都有一个类型化的引用。相关类型为：

标记	描述
must	子句 (query) 必须出现在匹配的文档中，并且将对得分起作用
filter	子句 (query) 必须出现在匹配的文档中。但是不同于 must ，查询的分数将被忽略。Filter子句是在 Filter上下文 中执行的，这意味着评分被忽略，子句被考虑用于缓存。
should	子句 (query) 应该出现在匹配的文档中。
must_not	子句 (query) 不能出现在匹配的文档中。子句是在 Filter上下文 中执行的，这意味着评分被忽略，子句被考虑用于缓存。因为评分被忽略，所以返回所有文档为 0 分。

bool 查询采用了一种“匹配越多越好”的方法，因此每个匹配 must 或 should 子句的分数将被添加到一起，以提供每个文档的最终 _score 。

```
POST /_search
{
  "query": {
    "bool": {
      "must": [
        {"term": { "user.id": "kimchy" }},
      ],
      "filter": [
        {"term": { "tags": "production" }}
      ],
      "must_not": [
        {"range": {
          "age": { "gte": 10, "lte": 20 }
        }}
      ],
      "should": [
        { "term": { "tags": "env1" } },
        { "term": { "tags": "deployed" } }
      ],
      "minimum_should_match": 1,
      "boost": 1.0
    }
  }
}
```

使用最小匹配 minimum_should_match

您可以使用 minimum_should_match 参数指定返回的文档必须匹配的 should 子句的数目或百分比。

如果 bool 查询至少包含一个 should 子句，并且没有 must 或 filter 子句，则默认值为 1 。否则，默认值为 0 。

有关其他有效值, 请参阅 [minimum_should_match 参数](#)。

使用 `bool.filter` 算分

在 `filter` 元素下指定的查询对评分没有影响 - 分数返回为 `0`。分数只受指定的查询影响。例如, 以下三个查询都返回 `status` 字段包含术语 `active` 的所有文档。

第一个查询为所有文档分配 `0` 分, 因为没有指定评分查询:

```
GET _search
{
  "query": {
    "bool": {
      "filter": {
        "term": {
          "status": "active"
        }
      }
    }
  }
}
```

这个 `bool` 查询有一个 `match_all` 查询, 它为所有文档分配 `1.0` 的分数。

```
GET _search
{
  "query": {
    "bool": {
      "must": {
        "match_all": {}
      },
      "filter": {
        "term": {
          "status": "active"
        }
      }
    }
  }
}
```

这个 `constant_score` 查询的行为方式与上面的第二个示例完全相同。`constant_score` 查询为所有与筛选器匹配的文档分配 `1.0` 的分数。

```
GET _search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "status": "active"
        }
      }
    }
  }
}
```

命名查询

每个查询在其顶级定义中接受一个 `_name`。您可以使用命名查询来跟踪哪些查询与返回的文档匹配。如果使用命名查询，则响应将为每个命中包含 `matched_queries` 属性。

```
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"
        }
      }
    }
  }
}
```

匹配词组查询

`match_phrase` 查询分析文本并从分析的文本中创建 `phrase` 查询。例如：

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

短语查询以任意顺序将术语匹配到可配置的 `slop`（默认为 `0`）。换位术语的斜率为 `2`。

`analyzer` 可以设置为控制哪个分词器将对文本执行分词过程。它默认为字段显式映射定义或默认搜索分词器，例如：

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

如 `match_query` 中所述，此查询还接受 `zero_terms_query`。

13.1 Query and filter context 查询和筛选上下文

相关性得分

默认情况下，Elasticsearch按相关性得分对匹配的搜索结果进行排序，相关性得分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，返回到search API的`_score`元数据字段中。`_score`越高，文档就越相关。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上下文中运行还是在筛选上下文中运行。

查询上下文

在查询上下文中，查询子句回答“此文档与此查询子句的匹配程度如何？”，除了决定文档是否匹配外，查询子句还在`_score`元数据字段中计算相关性分数。

每当查询子句被传递给`query`参数（例如search API中的`query`参数）时，查询上下文就生效。

筛选上下文

在筛选上下文中，查询子句需要回答问题“此文档是否与此查询子句匹配？”答案是简单的是或否 - 不计算分数。筛选上下文主要用于筛选结构化数据，例如。

- 这个`timestamp`是否属于2015年到2016年的范围？
- `status`字段是否设置为`published`？

Elasticsearch会自动缓存常用筛选器，以提高性能。

每当查询子句被传递给筛选器参数时，筛选器上下文就生效，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。

查询和筛选上下文示例

下面是在搜索API的查询和筛选上下文中使用的查询子句的示例。此查询将匹配满足以下所有条件的文档：

- `title`字段包含文字`search`。
- `content`字段包含单词`elasticsearch`。
- `status`字段包含确切的文字`published`。
- `publish_date`字段包含从2015年1月1日起的日期。

```

GET /_search
{
  "query": { ①
    "bool": { ②
      "must": [
        { "match": { "title": "Search" } },
        { "match": { "content": "Elasticsearch" } }
      ],
      "filter": [ ③
        { "term": { "status": "published" } },
        { "range": { "publish_date": { "gte": "2015-01-01" } } }
      ]
    }
  }
}

```

① query 参数表示查询上下文。

② bool 和两个 match 子句用于查询上下文，这意味着它们用于对每个文档的匹配程度进行评分。

③ filter 参数表示筛选上下文。它的 term 和 range 子句用于筛选上下文。它们会筛选掉不匹配的文档，但不会影响匹配文档的分数。

- 为查询上下文中的查询计算的分数表示为单精度浮点数；它们的有效位精度只有24位。超过有效位精度的分数计算将转换为精度损失的浮点值。
- 在查询上下文中使用查询子句来处理应该影响匹配文档得分的条件（即文档匹配的程度），并在筛选上下文中使用所有其他查询子句。

Specialized queries 特定查询

This group contains queries which do not fit into the other groups:

distance_feature query

一种查询，根据动态计算的原始数据和文档的日期、日期和地理点字段之间的距离来计算分数。它能够有效地跳过非竞争性点击。

more_like_this query

此查询查找与指定文本、文档或文档集合相似的文档。

percolate query

此查询查找存储为与指定文档匹配的文档的查询。

rank_feature query

一种基于数字特征值计算分数的查询，能够有效地跳过非竞争性点击。

script query

This query allows a script to act as a filter. Also see the function_score query.

script_score query

一种允许用脚本修改子查询得分的查询。

wrapper query

接受json或yaml字符串形式的其他查询的查询。

pinned query

将选定文档提升到与给定查询匹配的其他文档之上的查询。

索引模板

如何...

Elasticsearch附带默认值，旨在提供良好的开箱即用体验。全文搜索、高亮显示、聚合和索引都应该在用户不必更改任何内容的情况下工作。

但是，一旦您更好地理解了如何使用Elasticsearch，您就可以进行一些优化来提高用例的性能。

本节提供了关于应该和不应该进行哪些更改的指导。

29.1 General recommendations通用建议

不返回大结果集

Elasticsearch被设计成一个搜索引擎，这使得它能够很好地获取匹配查询的头部文档。但是，对于属于数据库域的工作负载（例如检索与特定查询匹配的所有文档）来说，这并不是一个好方法。如果需要这样做，请确保使用[Scroll API](#)。

避免使用大型文档

鉴于默认 `http.max_content_length` 如果设置为100MB，Elasticsearch将拒绝索引任何大于该值的文档。您可能会决定增加该特定设置，但Lucene仍有大约2GB的限制。

即使不考虑硬限制，大型文档通常也不实用。大型文档在网络、内存使用和磁盘方面的压力更大，即使对于不请求 `_source` 的搜索请求也是如此，因为Elasticsearch在所有情况下都需要获取文档的 `_id`，而对于大型文档，由于文件系统缓存的工作方式，获取此字段的成本更高。索引此类文档可以使用相当于文档原始大小数倍的内存量。邻近搜索（例如短语查询）和[高亮显示](#)也变得更加昂贵，因为它们的成本直接取决于原始文档的大小。

有时，重新考虑信息单位应该是什么是有用的。例如，您希望使书籍可搜索的事实并不一定意味着文档应该由整本书组成。最好使用章节甚至段落作为文档，然后在这些文档中有一个属性来标识它们属于哪本书。这不仅避免了大型文档的问题，还使搜索体验更好。例如，如果用户搜索两个单词 `foo` 和 `bar`，那么跨不同章节的匹配可能非常差，而同一段落中的匹配可能很好。

29.2 Recipes 食谱

本节包括一些解决常见问题的方法：

- 混合精确搜索和词干分析
- 获得一致的分数
- 将静态相关信号纳入分数

29.2.1 Mixing exact search with stemming 混合精确搜索和词干分析

在构建搜索应用程序时，词干分析通常是必须的，因为查询 `skiing` 时需要匹配包含 `ski` 或 `skis` 的文档。但是如果用户想专门搜索 `skiing` 呢？执行此操作的典型方法是使用[多字段](#)，以便以两种不同的方式索引相同的内容：

```
PUT index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "english_exact": {
          "tokenizer": "standard",
          "filter": [
            "lowercase"
          ]
        }
      }
    },
    "mappings": {
      "properties": {
        "body": {
          "type": "text",
          "analyzer": "english",
          "fields": {
            "exact": {
              "type": "text",
              "analyzer": "english_exact"
            }
          }
        }
      }
    }
  }
}

PUT index/_doc/1
{
  "body": "Ski resort"
}

PUT index/_doc/2
{
  "body": "A pair of skis"
}

POST index/_refresh
```

使用这样的设置，在 `body` 上搜索 `ski` 将会返回相同的文档：

```
GET index/_search
{
  "query": {
    "simple_query_string": {
      "fields": [ "body" ],
      "query": "ski"
    }
  }
}
```

```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": {
      "value": 2,
      "relation": "eq"
    },
    "max_score": 0.18232156,
    "hits": [
      {
        "_index": "index",
        "_type": "_doc",
        "_id": "1",
        "_score": 0.18232156,
        "_source": {
          "body": "Ski resort"
        }
      },
      {
        "_index": "index",
        "_type": "_doc",
        "_id": "2",
        "_score": 0.18232156,
        "_source": {
          "body": "A pair of skis"
        }
      }
    ]
  }
}
```

另一方面，在 `body.exact` 上搜索 `ski`，将只返回文档 1，因为 `body.exact` 不执行词干分析。

```
GET index/_search
{
  "query": {
    "simple_query_string": {
      "fields": [ "body.exact" ],
      "query": "ski"
    }
  }
}
```

```
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": {
      "value": 1,
      "relation": "eq"
    },
    "max_score": 0.8025915,
    "hits": [
      {
        "_index": "index",
        "_type": "_doc",
        "_id": "1",
        "_score": 0.8025915,
        "_source": {
          "body": "Ski resort"
        }
      }
    ]
  }
}
```

这并不是一件容易向最终用户公开的事情，因为我们需要一种方法来确定他们是否在寻找一个精确的匹配，并相应地重定向到相应的字段。另外，如果只有部分查询需要精确匹配，而其他部分仍然需要考虑词干，该怎么办？

幸运的是，`query_string` 和 `simple_query_string` 查询有一个功能可以解决这个问题：`quote_field_suffix`。这会告诉Elasticsearch，出现在引号之间的单词将被重定向到另一个字段，请参见以下内容：

```
GET index/_search
{
  "query": {
    "simple_query_string": {
      "fields": [ "body" ],
      "quote_field_suffix": ".exact",
      "query": "\"ski\""
    }
  }
}
```

```
{  
    "took": 2,  
    "timed_out": false,  
    "_shards": {  
        "total": 1,  
        "successful": 1,  
        "skipped": 0,  
        "failed": 0  
    },  
    "hits": {  
        "total": {  
            "value": 1,  
            "relation": "eq"  
        },  
        "max_score": 0.8025915,  
        "hits": [  
            {  
                "_index": "index",  
                "_type": "_doc",  
                "_id": "1",  
                "_score": 0.8025915,  
                "_source": {  
                    "body": "Ski resort"  
                }  
            }  
        ]  
    }  
}
```

在上面的例子中，由于 `quote_field_suffix` 参数，所以在 `body.exact` 字段，因此只有文档 1 匹配。这允许用户根据自己的喜好将精确搜索与词干搜索混合使用。

- 如果在 `quote_field_suffix` 中传递的字段选项不存在，搜索将退回到使用查询字符串的默认字段。

29.2.2 获得一致的得分

Elasticsearch使用分片和备份进行操作，这一事实为获得好的打分带来了挑战。

分数不可复制

假设同一个用户连续两次运行同一个请求，而两次返回的文档顺序都不相同，这是一种相当糟糕的体验，不是吗？不幸的是，如果您有备份，这种情况可能会发生（`index.number_of_replicas` 大于0）。原因是Elasticsearch以循环方式选择查询应该转到的分片，因此如果您连续运行同一查询两次，它很可能会转到同一分片的不同副本。

为什么这是个问题？索引统计是分数的重要组成部分。由于被删除的文档，这些索引统计数据在同一个分片的副本中可能不同。如您所知，当删除或更新文档时，旧文档不会立即从索引中删除，它只是标记为已删除，并且只有在下次合并此旧文档所属的段时才会从磁盘中删除。然而，出于实际原因，在索引统计中会考虑这些删除的文件。因此，假设主分片刚刚完成了一个删除了大量已删除文档的大型合并，那么它的索引统计信息可能与副本（仍然有大量已删除文档）有大量的不同，因此得分也不同。

解决此问题的建议方法是使用一个字符串，将登录的用户（例如用户id或会话id）标识为[首选项](#)。这确保了给定用户的所有查询总是会命中相同的分片，因此查询之间的分数保持更一致。

这种解决方法还有另一个好处：当两个文档具有相同的分数时，默认情况下，它们将按其内部Lucene doc id（与 `_id` 无关）排序。但是，这些doc id在同一个分片的多个副本中可能是不同的。因此，通过总是命中同一个分片，我们将获得具有相同分数的文档的更一致的排序。

相关性看起来是错误的

如果您注意到具有相同内容的两个文档得到不同的分数，或者某个完全匹配的文档没有排在第一位，那么这个问题可能与分片有关。默认情况下，Elasticsearch使每个分片负责生成自己的分数。但是，由于索引统计信息对得分有重要影响，因此只有当分片具有类似的索引统计信息时，这种方法才能很好地工作。假设是，由于默认情况下文档均匀地路由到分片，那么索引统计数据应该非常相似，评分也会像预期的那样起作用。但是，如果您：

- 在索引时使用路由
- 查询多个索引
- 或者索引中的数据太少

然后，很有可能搜索请求中涉及的所有分片都没有类似的索引统计信息，相关性可能很差。

如果您有一个小的数据集，解决这个问题的最简单方法就是将所有内容索引到一个只有一个分片的索引中（`index.number_of_shards: 1`），这是默认值。然后所有文档的索引统计将是相同的，分数将是一致的。

否则，解决此问题的建议方法是使用 `dfs_query_then_fetch` 搜索类型。这将使 Elasticsearch 对所有涉及的分片执行一次初始往返，请求它们提供与查询相关的索引统计信息，然后协调节点将合并这些统计信息，并在请求分片执行 [查询](#) 阶段时将合并的统计信息与请求一起发送，以便分片可以使用这些全局统计信息而不是自己统计来做评分。

在大多数情况下，这种额外的往返旅行应该非常便宜。但是，如果您的查询包含大量字段/术语或模糊查询，请注意单独收集统计信息可能并不便宜，因为必须在术语词典中查找所有术语才能查找统计信息。

29.2.3 Incorporating static relevance signals into the score 将静态相关信号纳入分数

许多领域有静态信号，已知与相关性相关。例如，PageRank和url length是网页搜索的两个常用特性，用于独立于查询调整网页页面的得分。

有两个主要的查询允许将静态分数贡献与文本相关性相结合，例如用BM25计算：`- script_score query - rank_feature query`

例如，假设您有一个 pagerank 字段，希望将其与BM25分数相结合，以便最终分数等于 `score = bm25_score + pagerank / (10 + pagerank)`

使用 `script_score query` 的请求将会像这样：

```
GET index/_search
{
  "query": {
    "script_score": {
      "query": {
        "match": { "body": "elasticsearch" }
      },
      "script": {
        "source": "_score * saturation( doc['pagerank'].value, 10)" ①
      }
    }
  }
}
```

① `pagerank` 必须被映射为 Numeric

同时，使用 `rank_feature query` 的请求将会像这样：

```
GET _search
{
  "query": {
    "bool": {
      "must": [
        "match": { "body": "elasticsearch" }
      ],
      "should": [
        "rank_feature": {
          "field": "pagerank", ①
          "saturation": {
            "pivot": 10
          }
        }
      ]
    }
  }
}
```

① `pagerank` 必须被映射为 `rank_feature` 字段

虽然这两个选项都会返回相似的分数，但有一个权衡：`script_score` 提供了很大的灵活性，使您能够根据自己的喜好将文本相关性分数与静态信号相结合。另一方面，`rank_feature` 查询只公开了几种将静态信号合并到分数中的方法。但是，它

依赖于 `rank_feature` 和 `rank_features` 字段，这些字段以一种特殊的方式索引值，从而允许 `rank_feature` 查询 跳过非竞争性文档并更快地获得查询的头部匹配项。

REST API

Elasticsearch公开了UI组件使用的REST API，可以直接调用这些API来配置和访问Elasticsearch功能。

我们正致力于在本节中包含更多ElasticSearch API。某些内容可能尚未包括在内。

- [API conventions](#)
- [Autoscaling APIs](#)
- [cat APIs](#)
- [Cluster APIs](#)
- [Cross-cluster replication APIs](#)
- [Data stream APIs](#)
- [Document APIs](#)
- [Enrich APIs](#)
- [Graph explore API](#)
- [Find structure API](#)
- [Index APIs](#)
- [Index lifecycle management APIs](#)
- [Ingest APIs](#)
- [Info API](#)
- [Licensing APIs](#)
- [Logstash APIs](#)
- [Machine learning anomaly detection APIs](#)
- [Machine learning data frame analytics APIs](#)
- [Migration APIs](#)
- [Reload search analyzers API](#)
- [Repositories metering APIs](#)
- [Rollup APIs](#)
- [Search APIs](#)
- [Searchable snapshots APIs](#)
- [Security APIs](#)
- [Snapshot and restore APIs](#)
- [Snapshot lifecycle management APIs](#)
- [Transform APIs](#)
- [Usage API](#)
- [Watcher APIs](#)

Document APIs

本节首先简要介绍Elasticsearch的[数据复制模型](#)，然后详细介绍以下CRUD API：

单文档 APIs

- [Index](#)
- [Get](#)
- [Delete](#)
- [Update](#)

多文档 APIs

- [Multi get](#)
- [Bulk](#)
- [Delete by query](#)
- [Update by query](#)
- [Reindex](#)

所有CRUD API都是单索引API。`index` 参数接受单个索引名称或指向单个索引的 `alias`。

Reindex API

将文档从源复制到目标。

源和目标可以是任何预先存在的索引、索引别名或[数据流](#)。但是，源和目标必须不同。例如，不能将数据流重新索引到自身中。

重新索引要求为源中的所有文档启用[_source](#)。

在调用 `_reindex` 之前，应根据需要配置目标。重新索引不会从源或其关联模板复制设置。

必须提前配置映射、分片、计数、副本等。

```
POST _reindex
{
  "source": {
    "index": "my-index-000001"
  },
  "dest": {
    "index": "my-new-index-000001"
  }
}
```

请求

```
POST /_reindex
```

先决条件

- 如果启用了Elasticsearch安全功能，则您必须具有以下安全权限：
 - 源数据流、索引或索引别名的 `read` [索引权限](#)。
 - 目标数据流、索引或索引别名的 `write` 索引权限。
 - 若要使用 `reindex` API 请求自动创建数据流或索引，必须对目标数据流、索引或索引别名具有 `auto_configure`，`create_index`，或 `manage` 索引权限。
 - 如果从远程集群重新索引，则 `source.remote.user` 必须具有源数据流、索引或索引别名的 `monitor` [集群权限](#)和 `read` 索引权限。
- 如果从远程集群重新建立索引，则必须在 `elasticsearch.yml` 配置文件中的 `reindex.remote.whitelist` 选项显式配置远程主机。请参见[从远程重新索引](#)。
- 自动创建数据流需要启用数据流的匹配索引模板。请参见[设置数据流](#)。

说明

从源索引中提取[文档源](#)，并将文档索引到目标索引中。您可以将所有文档复制到目标索引，或重新索引文档的子集。

就像 `_update_by_query` 一样，`_reindex` 获取源的快照，但其目标必须不同，这样就不太可能发生版本冲突。`dest` 元素可以像 `index` API 一样配置，以控制乐观并发控制。省略 `version_type` 或将其设置为 `internal` 会导致 Elasticsearch 盲目地将文档转储到目标中，覆盖所有恰好具有相同 ID 的文档。

将 `version_type` 设置为 `external` 会导致 Elasticsearch 保留源中的 `version`，创建丢失的任何文档，并更新目标中版本比源中版本旧的任何文档。

将 `op_type` 设置为 `create` 会导致 `_reindex` 只在目标中创建缺少的文档。所有现有文档都将导致版本冲突。

因为数据流是 **只附加(append-only)** 的，所以对目标数据流的任何重新索引请求都必须具有 `op_type` 类型“`create`”。重新索引只能向目标数据流添加新文档。它无法更新目标数据流中的现有文档。

默认情况下，版本冲突会中止 `_reindex` 过程。若要在存在冲突时继续重新索引，请将“`conflicts`”请求主体参数设置为 `proceed`。在本例中，响应包含遇到的版本冲突的计数。请注意，其他错误类型的处理不受“`conflicts`”参数的影响。

异步运行reindex

如果请求包含 `wait_for_completion=false`，Elasticsearch 将执行一些飞行前检查，启动请求，并返回一个 `task`，您可以使用该任务取消或获取该任务的状态。

Elasticsearch 在 `.tasks/_doc/${taskId}` 以文档形式创建此任务的记录。完成任务后，应删除任务文档，以便 Elasticsearch 可以回收空间。

从多个源reindex

如果有许多源需要重新索引，通常最好一次重新索引一个源，而不是使用 `glob` 模式来选取多个源。这样，如果存在任何错误，可以通过删除部分完成的源并重新开始来恢复。它还使并行化过程相当简单：拆分源列表以重新索引，并行运行每个列表。

一次性 `bash` 脚本似乎可以很好地实现这一点：

```
for index in i1 i2 i3 i4 i5; do
  curl -HContent-Type:application/json -XPOST localhost:9200/_reindex?pretty -
    "source": {
      "index": "$index"
    },
    "dest": {
      "index": "$index-reindexed"
    }
  done
```

Index APIs

Index APIs 用于管理单个索引、索引设置、别名、映射和索引模板。

索引管理：

- 创建索引
- 删除索引
- 获取索引
- 索引已存在
- 关闭索引
- 开放索引
- 收缩指数
- 拆分索引
- 克隆索引
- 滚动指数
- 冻结索引
- 解冻索引
- 解析索引

映射管理：

- 放置映射
- 获取映射
- 获取字段映射
- 类型已存在

别名管理：

- 添加索引别名
- 删除索引别名
- 获取索引别名
- 存在索引别名
- 更新索引别名

索引设置：

- 更新索引设置
- 获取索引设置
- 分析

索引模板：

索引模板会自动将设置、映射和别名应用于新索引。它们通常用于为时间序列数据配置滚动索引，以确保每个新索引与前一个索引具有相同的配置。与数据流关联的索引模板配置其备份索引。有关详细信息，请参见索引模板。

- 放置索引模板
- 获取索引模板
- 删除索引模板
- 放置组件模板
- 获取组件模板
- 删除组件模板
- 模拟索引
- 模拟模板

监测：

- 索引统计信息
- 索引段
- 指数恢复
- 索引碎片存储

状态管理：

- 清除缓存
- 刷新
- 冲洗
- 同步刷新
- 强制合并

悬空索引：

- 列出悬挂索引
- 导入悬索引
- 删除悬挂索引

Put index template API

创建或更新索引模板。索引模板定义可以自动应用于新索引的设置、映射和别名。

```
PUT /_index_template/template_1
{
  "index_patterns" : ["te*"],
  "priority" : 1,
  "template": {
    "settings" : {
      "number_of_shards" : 2
    }
  }
}
```

请求

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

先决条件

如果已启用Elasticsearch安全功能，则必须具有
有 `manage_index_templates` 或 `manage 群集权限` 才能使用此API。

说明

Elasticsearch基于与索引名称匹配的通配符模式将模板应用于新索引。

索引模板在数据流或索引创建期间应用。对于数据流，在创建流的备份索引时应用这些设置和映射。

[创建索引](#) 请求中指定的设置和映射将覆盖索引模板中指定的任何设置或映射。

对索引模板的更改不会影响现有索引，包括数据流的现有备份索引。

索引模板中的注释

可以在索引模板中使用C语言风格的`/**/`块注释。除了开头的花括号之前，您可以在请求正文中的任何位置插入注释。

路径参数

<`index-template`>

(必需，字符串) 要创建的索引模板的名称。

请求参数

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`。

master_timeout

(可选, 时间单位) 等待连接到主节点的时间。如果在超时过期之前没有收到响应, 则请求失败并返回错误。默认为 30秒。

请求正文

index_patterns

(必需, 字符串数组) 用于在创建期间匹配数据流和索引名称的通配符 (*) 表达式数组。

Elasticsearch具有内置索引模板, 每个模板的优先级为100, 用于以下索引模式:

- logs--
- metrics--
- synthetics--

[Elastic代理](#)使用这些模板来创建数据流。Fleet集成创建的索引模板使用类似 的重叠索引模式, 优先级高达 200。

如果您使用Fleet或Elastic代理, 请为索引模板指定低于100的优先级, 以避免覆盖这些模板。否则, 为避免意外应用模板, 请执行以下一个或多个操作:

- 要禁用所有内置索引和组件模板, 请设置 `stack.templates.enabled` 已启用使用[群集更新设置API](#)设置为false。
- 使用不重叠的索引模式。
- 为重叠模式的模板分配高于200的优先级。例如, 如果您不使用Fleet或 Elastic代理, 并且希望为logs- 索引模式创建一个模板, 请将模板的优先级指定为500。这将确保应用您的模板, 而不是日志的内置模板 `logs-*`。`

composed_of

(可选, 字符串数组) 组件模板名称的有序列表。组件模板按指定的顺序合并, 这意味着最后指定的组件模板具有最高的优先级。有关示例, 请参见组合多个组件模板。

priority

(可选, 整数) 创建新数据流或索引时确定索引模板优先级的优先级。选择具有最高优先级的索引模板。如果未指定优先级, 则将模板视为优先级为0 (最低优先级)。Elasticsearch不会自动生成此号码。

version

(可选, 整数) 用于外部管理索引模板的版本号。Elasticsearch不会自动生成此号码。

_meta

(可选, 对象) 关于索引模板的可选用户元数据。可能有任何内容。Elasticsearch不会自动生成此地图。

Put mapping API

向现有数据流或索引添加新字段。您还可以使用put mapping API更改现有字段的搜索设置。

对于数据流，这些更改默认应用于所有备份索引。

```
PUT /my-index-000001/_mapping
{
  "properties": {
    "email": {
      "type": "keyword"
    }
  }
}
```

在7.0.0之前，映射定义包含类型名。尽管现在不赞成在请求中指定类型，但如果设置了请求参数 `include_type_name`，仍然可以提供类型。有关详细信息，请参阅[删除映射类型](#)。

请求

```
PUT /<target>/_mapping
PUT /_mapping
```

先决条件

- 如果启用了Elasticsearch安全功能，则必须对目标数据流、索引或索引别名具有 `manage 索引权限`。
- [在7.9中已弃用]如果请求以索引或索引别名为目标，您还可以使用 `create`，`create_doc`，`index` 或 `write` 权限更新其映射。

路径参数

`<target>`

(可选，字符串) 使用逗号分隔的数据流、索引和索引别名的列表，用于限制请求。支持通配符表达式（`*`）。

若要以群集中的所有数据流和索引为目标，请省略此参数或使用 `_all` 或 `*`。

请求参数

`allow_no_indices`

(可选, 布尔值) 如果为 `false`, 则如果任何通配符表达式、索引别名或 `_all` 值没有命中或只命中关闭的索引, 则请求将返回错误。即使请求命中了其他已开启的索引, 此行为也适用。例如, 如果有索引以 `foo` 开头, 而没有索引以 `bar` 开头, 则以 `foo*,bar*` 为目标的请求将返回错误。

默认为 `false`。

expand_wildcards

(可选, 字符串) 通配符表达式可以匹配的索引类型。如果请求可以以数据流为目标, 则此参数确定通配符表达式是否匹配隐藏的数据流。支持逗号分隔的值, 例如 `open,hidden`。有效值为:

- **all**

匹配任何数据流或索引, 包括隐藏的数据流或索引。

- **open**

匹配开放的、非隐藏的索引。也匹配任何非隐藏的数据流。

- **closed**

匹配封闭的非隐藏索引。也匹配任何非隐藏的数据流。数据流不能是关闭的。

- **hidden**

匹配隐藏数据流和隐藏索引。必须与 `open, closed` 或两者结合使用。

- **none**

不接受通配符表达式。

默认为 `open`。

include_type_name

[在7.0.0中已弃用] (可选, 布尔值) 如果为 `true`, 则映射体中应包含映射类型。

默认为 `false`。

ignore_unavailable

(可选, 布尔值) 如果为 `true`, 则响应中不包括缺失或关闭的索引。默认为 `false`。

master_timeout

(可选, 时间单位) 等待连接到主节点的时间。如果在超时过期之前没有收到响应, 则请求失败并返回错误。默认为 30秒。

timeout

(可选, [时间单位](#)) 等待响应的时间。如果在超时过期之前没有收到响应, 则请求失败并返回错误。默认为 30秒。

write_index_only

(可选, 布尔值) 如果为 true, 则映射仅应用于当前写入的目标索引。默认为 false。

请求正文

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](#)

有关现有字段, 请参见[更改现有字段的映射](#)。

范例

单目标示例

put mapping API需要一个现有的数据流或索引。下面的[create index](#) API请求创建了一个不带映射的 publications 索引。

```
PUT /publications
```

下面的put mapping API请求将 title (一个新的 text 字段) 添加到 publications 索引中。

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

多目标

put mapping API可以通过一个请求应用于多个数据流或索引。例如, 可以同时更新索引 my-index-000001 和索引 my-index-000002 的映射:

```
# Create the two indices
PUT /my-index-000001
PUT /my-index-000002

# Update both mappings
PUT /my-index-000001,my-index-000002/_mapping
{
  "properties": {
    "user": {
      "properties": {
        "name": {
          "type": "keyword"
        }
      }
    }
  }
}
```

向现有对象字段添加新属性

可以使用put mapping API向现有 `object` 字段添加新属性。要了解其工作原理，请尝试以下示例。

使用create index API创建一个包含 `name` 对象字段和内部 `first` 文本字段的索引。

```
PUT /my-index-000001
{
  "mappings": {
    "properties": {
      "name": {
        "properties": {
          "first": {
            "type": "text"
          }
        }
      }
    }
  }
}
```

使用put mapping API向 `name` 字段添加一个新的内部 `last` 文本字段。

```
PUT /my-index-000001/_mapping
{
  "properties": {
    "name": {
      "properties": {
        "last": {
          "type": "text"
        }
      }
    }
  }
}
```

向现有字段添加多个字段

[多字段](#)允许您以不同的方式索引同一字段。可以使用put mapping API更新 `fields` 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[create index API](#)创建包含 `city` 文本字段的索引。

```
PUT /my-index-000001
{
  "mappings": {
    "properties": {
      "city": {
        "type": "text"
      }
    }
  }
}
```

虽然文本字段对于全文搜索效果很好，但[keyword](#)字段不会被分词，对于排序或聚合可能效果更好。

使用[put mapping API](#)为 `city` 字段启用多字段。此请求添加用于排序的 `city.raw` 关键字多字段。

```
PUT /my-index-000001/_mapping
{
  "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](#)的文档指示是否可以使用[put mapping API](#)对现有字段进行更新。例如，您可以使用[put mapping 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[create index 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-index-000001
{
  "mappings": {
    "properties": {
      "user_id": {
        "type": "keyword",
        "ignore_above": 20
      }
    }
  }
}
```

使用[put mapping API](#)将 `ignore_above` 参数值更改为 `100`。

```
PUT /my-index-000001/_mapping
{
  "properties": {
    "user_id": {
      "type": "keyword",
      "ignore_above": 100
    }
  }
}
```

更改现有字段的映射

除了支持的[映射参数](#)外，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索引的数据无效。

如果需要更改数据流备份索引中字段的映射，请参阅[更改数据流的映射和设置](#)。

如果需要更改字段在其他索引中的映射，请使用正确的映射创建一个新索引，并将数据[重新索引](#)到该索引中。

要查看如何更改索引中现有字段的映射，请尝试以下示例。

使用[create index](#) API创建一个具有 `long` 字段类型的 `user_id` 字段的索引。

```
PUT /my-index-000001
{
  "mappings" : {
    "properties": {
      "user_id": {
        "type": "long"
      }
    }
  }
}
```

使用[index](#) API索引多个具有 `user_id` 字段值的文档。

```
POST /my-index-000001/_doc?refresh=wait_for
{
  "user_id" : 12345
}

POST /my-index-000001/_doc?refresh=wait_for
{
  "user_id" : 12346
}
```

要将 `user_id` 字段更改为 `keyword` 字段类型，请使用[create index](#) API创建具有正确映射的新索引。

```
PUT /my-new-index-000001
{
  "mappings" : {
    "properties": {
      "user_id": {
        "type": "keyword"
      }
    }
  }
}
```

使用[reindex API](#)将文档从旧索引复制到新索引。

```
POST /_reindex
{
  "source": {
    "index": "my-index-000001"
  },
  "dest": {
    "index": "my-new-index-000001"
  }
}
```

重命名字段

重命名字段将使已在旧字段名下索引的数据无效。应该添加一个 `alias` 字段来创建备用字段名。

例如，使用[create index API](#)创建一个带有 `user_identifier` 字段的索引。

```
PUT /my-index-000001
{
  "mappings": {
    "properties": {
      "user_identifier": {
        "type": "keyword"
      }
    }
  }
}
```

使用[put mapping API](#)为现有的 `user_identifier` 字段添加别名 `user_id` 字段。

```
PUT /my-index-000001/_mapping
{
  "properties": {
    "user_id": {
      "type": "alias",
      "path": "user_identifier"
    }
  }
}
```

Snapshot and restore API 快照和还原API

可以使用以下API设置快照存储库、管理快照备份以及将快照还原到正在运行的群集。

有关详细信息，请参阅[快照和还原](#)。

快照存储库管理API

- 放置快照存储库
- 验证快照存储库
- 获取快照存储库
- 删除快照存储库
- 清理快照存储库

快照管理API

- 创建快照
- 克隆快照
- 获取快照
- 获取快照状态
- 还原快照
- 删除快照