

# 预备知识-Restful

## 起源

在没有前端分离概念之前，一个网站的完成总是“all in one”，在这个阶段，页面、数据、渲染全部在服务端完成，这样做的最大的弊端是后期维护，扩展极其痛苦，开发人员必须同时具备前后端知识。于是后来慢慢的兴起了前后端分离的思想：即后端负责数据编造，而前端则负责数据渲染，前端静态页面调用指定 api 获取到有固定格式的数据，再将数据展示出来，这样呈现给用户的就是一个“动态”的过程。

而关于 api 这部分的设计则成了一个问题。如何设计出一个便于理解，容易使用的 api 则成了一个问题，而所谓的 RESTful 就是用来规范我们的 API 的一种约束。

## REST

作为 REST，其实是 Representational State Transfer（表象层状态转变）三个单词的缩写，它由 Roy Fielding(Fielding 是 HTTP 协议（1.0 版和 1.1 版）的主要设计者、Apache 服务器软件的作者之一、Apache 基金会的第一任主席）于 2000 年论文中提出，他在论文中提到：“我这篇文章的写作目的，就是想在符合架构原理的前提下，理解和评估以网络为基础的应用软件的架构设计，得到一个功能强、性能好、适宜通信的架构。REST 指的是一组架构约束条件和原则。”如果一个架构符合 REST 的约束条件和原则，我们就称它为 RESTful 架构。

要理解 RESTful 架构，最好的方法就是去理解 Representational State Transfer 这个词组到底是什么意思，它的每一个词代表了什么涵义。如果你把这个名称搞懂了，也就不难体会 REST 是一种什么样的设计。

## 资源 ( Resources )

REST 的名称“表现层状态转化”中，省略了主语。“表现层”其实指的是“资源”（Resources）的“表现层”。

所谓“资源”，就是网络上的一个实体，或者说是网络上的一个具体信息。它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的实在。

要让一个资源可以被识别，需要有个唯一标识，在 Web 中这个唯一标识就是 URI(Uniform Resource Identifier)。

URI 既可以看成是资源的地址，也可以看成是资源的名称。如果某些信息没有使用 URI 来表示，那它就不能算是一个资源，只能算是资源的一些信息而已。

你可以用一个 URI(统一资源定位符)指向它，每种资源对应一个特定的 URI。要获取这个资源，访问它的 URI 就可以，因此 URI 就成了每一个资源的地址或独一无二的识别符。

所谓“上网”，就是与互联网上一系列的“资源”互动，调用它的 URI。

---

## URI 设计技巧

### 使用 \_ 或 - 来让 URI 可读性更好

曾经 Web 上的 URI 都是冰冷的数字或者无意义的字符串，但现在越来越多的网站使用\_或-来分隔一些单词，让 URI 看上去更为人性化。例如国内比较出名的开源中国社区，它上面的新闻地址就采用这种风格，如  
<http://www.oschina.net/news/38119/oschina-translate-reward-plan>。

### 使用 / 来表示资源的层级关系

例如上述/git/git/commit/e3af72cdafab5993d18fae056f87e1d675913d08 就表示了一个多级的资源，指的是 git 用户的 git 项目的某次提交记录，又例如/orders/2012/10 可以用来表示 2012 年 10 月的订单记录。

### 使用 ? 来过滤资源

很多人只是把?简单的当做是参数的传递，很容易造成 URI 过于复杂、难以理解。可以把?用于对资源的过滤，例如/git/git/pulls 用来表示 git 项目的所有推入请求，而/pulls?state=closed 用来表示 git 项目中已经关闭的推入请求，这种 URL 通常对应的是一些特定条件的查询结果或算法运算结果。

，或；可以用来表示同级资源的关系

有时候我们需要表示同级资源的关系时，可以使用, 或; 来进行分割。例如哪天 github 可以比较某个文件在随意两次提交记录之间的差异，或许可以使用 /git/git/block-sha1/sha1.h/compare/e3af72cdafab5993d18fae056f87e1d675913d08;b63e61bdf38e872d5215c07b264dcc16e4febca 作为 URI。不过，现在 github 是使用…来做这个事情的，例如/git/git/compare/master…next。

### URI 不应该包含动词

因为“资源”表示一种实体，所以应该是名词，URI 不应该有动词，动词应该放在 HTTP 协议中。

举例来说，某个 URI 是/posts/show/1，其中 show 是动词，这个 URI 就设计错了，正确的写法应该是/posts/1，然后用 GET 方法表示 show。

如果某些动作是 HTTP 动词表示不了的，你就应该把动作做成一种资源。比如网上汇款，从账户 1 向账户 2 汇款 500 元，错误的 URI 是：

POST /accounts/1/transfer/500/to/2

正确的写法是把动词 transfer 改成名词 transaction，资源不能是动词，但是可以是一种服务。

### URI 中不宜加入版本号

例如：

<http://www.example.com/app/1.0/foo>

<http://www.example.com/app/1.1/foo>

<http://www.example.com/app/2.0/foo>

---

因为不同的版本，可以理解成同一种资源的不同表现形式，所以应该采用同一个 URI。版本号可以在 HTTP 请求头信息的 Accept 字段中进行区分。

## 表现层 ( Representation )

“资源”是一种信息实体，它可以有多种外在表现形式。我们把“资源”具体呈现出来的形式，叫做它的“表现层” (Representation)。

比如，文本可以用 txt 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以采用二进制格式；图片可以用 JPG 格式表现，也可以用 PNG 格式表现。

URI 只代表资源的实体，不代表它的形式。严格地说，有些网址最后的 “.html”后缀名是不必要的，因为这个后缀名表示格式，属于“表现层”范畴，而 URI 应该只代表“资源”的位置。它的具体表现形式，应该在 HTTP 请求的头信息中用 Accept 和 Content-Type 字段指定，这两个字段才是对“表现层”的描述。

## 状态转化 ( State Transfer )

访问一个网站，就代表了客户端和服务器的一个互动过程。在这个过程中，势必涉及到数据和状态的变化。

互联网通信协议 HTTP 协议，是一个无状态协议。这意味着，所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生“状态转化” (State Transfer)。而这种转化是建立在表现层之上的，所以就是“表现层状态转化”。

客户端用到的手段，目前来说只能是 HTTP 协议。具体来说，就是 HTTP 协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：GET 用来获取资源，POST 用来新建资源（也可以用于更新资源），PUT 用来更新资源，DELETE 用来删除资源。GET、PUT 和 DELETE 请求都是幂等的，无论对资源操作多少次，结果总是一样的，POST 不是幂等的。

## 什么是 RESTful 架构

综合上面的解释，我们总结一下什么是 RESTful 架构：

1. 架构里，每一个 URI 代表一种资源；
2. 客户端和服务器之间，传递这种资源的某种表现层；
3. 客户端通过四个 HTTP 动词 (get、post、put、delete)，对服务器端资源进行操作，实现“表现层状态转化”。

注意：REST 架构风格并不是绑定在 HTTP 上，只不过目前 HTTP 是唯一与 REST 相关的实例。所以我们这里描述的 REST 也是通过 HTTP 实现的 REST。

## 辨析 URI、URL、URN

RFC 3986 中是这样说的：

---

A Uniform Resource Identifier (URI) 是一个紧凑的字符串用来标示抽象或物理资源。

一个 URI 可以进一步被分为定位符、名字或两者都是。术语“Uniform Resource Locator”(URL)是URI的子集，除了确定一个资源，还提供一种定位该资源的主要访问机制。

所以，URI = Universal Resource Identifier 统一资源标志符，包含 URL 和 URN，支持的协议有 http、https、ftp、mailto、magnet、telnet、data、file、nfs、gopher、ldap 等，java 还大量使用了一些非标准的定制模式，如 rmi、jar、jndi 和 doc，来实现各种不同用途。

URL = Universal Resource Locator 统一资源定位符，URL 唯一地标识一个资源在 Internet 上的位置。不管用什么方法表示，只要能定位一个资源，就叫 URL。

URN = Universal Resource Name 统一资源名称，URN 它命名资源但不指定如何定位资源，比如：只告诉你一个人的姓名，不告诉你这个人在哪。

对于一个资源来说，URN 就好比他的名字，而 URL 就好比是资源的街道住址。换句话说，URN 标识了一个资源项目，而 URL 则提供了一种找到他的方法。

比如同时指定基本的获取机制和网络位置。举个例子，[http://example.org/wiki/Main\\_Page](http://example.org/wiki/Main_Page)，指向了一个被识别为/wike/Main\_Page 的资源，这个资源的表现形式是 HTML 和相关的代码。而获取这个资源的方法是在网络中从一个名为 example.org 的主机上，通过 HTTP(Hypertext Transfer Protocol) 获得。

而 URN 则是一种在特定的名称空间中通过名字来标识资源的 URI。当讨论一种资源而不需要知道它的位置或者如何去获得它的时候，就可以使用 URN。例如，在 International Standard Book Number (ISBN) 系统中，\* ISBN 0-486-27557-4 用来指定莎士比亚的作品《罗密欧与朱丽叶》的一个特定版本。指示这一版本的 URN 是 urn:isbn:0-486-27557-4\*，但是如果想要获得这个版本的书，就需要知道它的位置，这样就必须知道它的 URL。

## 1、什么是 ELK

### 为什么需要 ELK

官网的说法：Elasticsearch 是一个开源的分布式 RESTful 搜索和分析引擎，能够解决越来越多不同的应用场景。

看一个应用场景，常见的 WEB 应用日志分析。一般我们会怎么做？

登录到每台服务器上，直接在日志文件中 grep、awk 就可以获得自己想要的信息。但在规模较大的场景中，此方法效率低下，面临问题包括日志量太大如何归档、文本搜索太慢怎么办、如何多维度查询。

---

这个时候我们希望集中化的日志管理，所有服务器上的日志收集汇总。常见解决思路是建立集中式日志收集系统，将所有节点上的日志统一收集，管理，访问。

这样对于大型系统来说，都是一个分布式部署的架构，不同的服务模块部署在不同的服务器上，问题出现时，大部分情况需要根据问题暴露的关键信息，定位到具体的服务器和服务模块，构建一套集中式日志系统，可以提高定位问题的效率。

一个完整的集中式日志系统，需要包含以下几个主要特点：

收集—能够采集多种来源的日志数据

传输—能够稳定的把日志数据传输到中央系统

存储—如何存储日志数据分析—可以支持

UI 分析警告—能够提供错误报告，监控机制

ELK 就是这样一套解决方案，并且都是开源软件，之间互相配合使用，完美衔接，高效的满足了很多场合的应用，而不仅仅是日志分析。

## 什么是 ELK

ELK 是三个开源软件的缩写，分别表示：Elasticsearch , Logstash, Kibana ，它们都是开源软件。

Elasticsearch 是个开源分布式搜索引擎，提供搜集、分析、存储数据三大功能。它的特点有：分布式，零配置，自动发现，索引自动分片，索引副本机制，restful 风格接口，多数据源，自动搜索负载等。

Logstash 主要是用来日志的搜集、分析、过滤日志的工具，支持大量的数据获取方式。一般工作方式为 c/s 架构，client 端安装在需要收集日志的主机上，server 端负责将收到的各节点日志进行过滤、修改等操作在一并发往 Elasticsearch 上去。

Kibana 也是一个开源和免费的工具，Kibana 可以为 Logstash 和 Elasticsearch 提供的日志分析友好的 Web 界面，可以帮助汇总、分析和搜索重要数据日志。

新增了一个 Beats 系列组件，它是一个轻量级的日志收集处理工具(Agent)，Beats 占用资源少，适合于在各个服务器上搜集日志或信息后传输给 Logstash。

加入 Beats 系列组件后，官方名称就变为了 Elastic Stack，产品如下：



**Elasticsearch**

**Kibana**

**Logstash**

**Beats**

# 安装，启动和 HelloWorld

## 环境和安装

本次课程我们的运行环境为 Linux，演示服务器操作系统版本情况如下：

```
[elk@xiangxue ~]$ lsb_release -a
LSB Version: :core-4.1-amd64:core-4.1-noarch
Distributor ID: CentOS
Description:    CentOS Linux release 7.7.1908 (Core)
Release:        7.7.1908
Codename:       Core
```

因为 Elastic Stack 中主要组件都是用 Java 写的，所以操作系统上还应该安装好 Java，因为本次我们将以 Elasticsearch 7 版本为主，所以，需要安装 JDK1.8 以上。

```
[elk@xiangxue ~]$ java -version
java version "1.8.0_77"
Java(TM) SE Runtime Environment (build 1.8.0_77-b03)
Java HotSpot(TM) 64-Bit Server VM (build 25.77-b03, mixed mode)
```

而 Elastic Stack 系列产品我们可以到 Elastic 的官网上去下载：

<https://www.elastic.co/cn/downloads>

The screenshot shows the Elastic Stack download page. At the top, there's a blue banner with the text '准备好了吗？我们开始吧！' (Are you ready? Let's start!). Below the banner, there are three main sections: 'Elasticsearch' (distributed, RESTful search and analysis), 'Kibana' (data visualization in the Elastic Stack), and 'Beats' (lightweight data collection, processing, and transmission). Each section has a small icon, a brief description, and two download options: '在 Elastic Cloud 上启用' (Enable on Elastic Cloud) and '下载' (Download). There are also '联系客服' (Contact Customer Support) and '免费试用' (Free Trial) buttons at the top right.

我们选择 7.7.0 版本为本次课程的讲授版本，从具体的下载页面可以看到 Elastic Stack 支持各种形式的安装。

---

Downloads:	<a href="#">WINDOWS shaasc</a>	<a href="#">MACOS shaasc</a>
	<a href="#">LINUX X86_64 shaasc</a>	<a href="#">LINUX AARCH64 shaasc</a>
	<a href="#">DEB X86_64 shaasc</a>	<a href="#">DEB AARCH64 shaasc</a>
	<a href="#">RPM X86_64 shaasc</a>	<a href="#">RPM AARCH64 shaasc</a>
	<a href="#">MSI (BETA) shaasc</a>	
Package Managers:	Install with <a href="#">yum, dnf, or zypper</a>	
	Install with <a href="#">apt-get</a>	
	Install with <a href="#">homebrew</a>	
Containers:	Run with <a href="#">Docker</a>	

我们选择以免安装的压缩包的形式。下载后上传到服务器解压缩即可运行。

```
elasticsearch-7.7.0-linux-x86_64.tar.gz  
kibana-7.7.0-linux-x86_64.tar.gz  
logstash-7.7.0.tar.gz
```

## 运行

Elasticsearch 默认不允许用 root 用户运行，会报错，而且从服务器安全的角度来说，也不应该以 root 用户来做日常工作，因此我们新建一个用户 elk 并以 elk 用户登录。

### *Elasticsearch*

用 tar -xvf 命令解压压缩包 `elasticsearch-7.7.0-linux-x86_64.tar.gz` 后进入 `elasticsearch-7.7.0` 文件夹中的 `bin` 文件夹，并执行命令 `./elasticsearch`。

待启动完成，

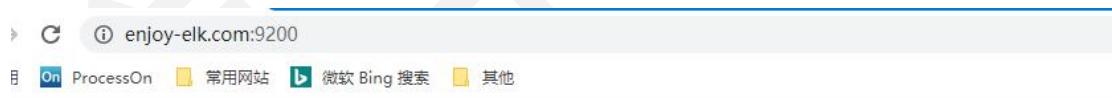
```
[elk@xiangxue bin]$ ./elasticsearch  
future versions of Elasticsearch will require Java 11; your Java version from  
es not meet this requirement  
future versions of Elasticsearch will require Java 11; your Java version from  
es not meet this requirement  
[2020-05-23T16:06:20,782][INFO ][o.e.e.NodeEnvironment      ] [xiangxue] using  
, net usable_space [38.6gb], net total_space [49gb], types [rootfs]  
[2020-05-23T16:06:20,783][INFO ][o.e.e.NodeEnvironment      ] [xiangxue] heap :  
object pointers [true]  
ilm-history-ilm-policy]  
[2020-05-23T16:06:37,969][INFO ][o.e.x.i.a.TransportPutLifecycleAction] [xiangxue] adding index lifecycle policy [  
ml-size-based-ilm-policy]  
[2020-05-23T16:06:38,125][INFO ][o.e.l.LicenseService      ] [xiangxue] license [59c2f23f-0de0-4450-9878-fce399932bf4] mode [basic] - valid  
[2020-05-23T16:06:38,127][INFO ][o.e.x.s.s.SecurityStatusChangeListener] [xiangxue] Active license is now [BASIC];  
Security is disabled  
[2020-05-23T16:06:38,133][INFO ][o.e.x.i.a.TransportPutLifecycleAction] [xiangxue] adding index lifecycle policy [  
slm-history-ilm-policy]
```

输入 curl <http://localhost:9200>

显示

```
[elk@xiangxue ~]$ curl http://localhost:9200
{
  "name" : "xiangxue",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "jxymaixiTja3kGNhI2FDvw",
  "version" : {
    "number" : "7.7.0",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "81ale9eda8e6183f5237786246f6dcfd26a10eaf",
    "build_date" : "2020-05-12T02:01:37.602180Z",
    "build_snapshot" : false,
    "lucene_version" : "8.5.1",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

表示我们的 Elasticsearch 运行成功了，我们试着在本地浏览器进行访问



却显示“拒绝了我们的连接请求”，因此我们还需要配置一下 Elasticsearch 以允许我们进行外网访问，进入 elasticseach-7.7.0 下的 config 目录，编辑 elasticsearch.yml 文件，删除 network.host 前的#字符，并写入服务器的地址并保存。

```
#----- Network -----  
#  
# Set the bind address to a specific IP (IPv4 or IPv6):  
#  
network.host: 172.18.194.140 ←  
#  
# Set a custom port for HTTP:  
#  
#http.port: 9200
```

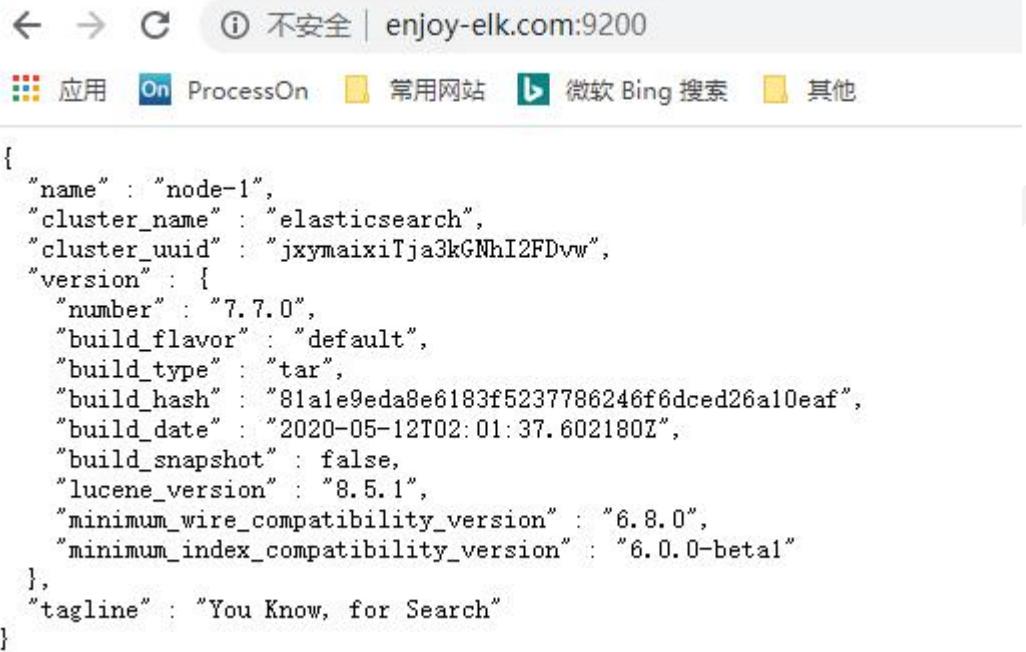
再次运行，但是这次却出了错

```
[2020-05-23T16:28:03,662][INFO ][o.e.b.BootstrapChecks    ] [xiangxue] bound or publishing to a non-loopback address, enforcing bootstrap checks  
ERROR: [1] bootstrap checks failed  
[1]: the default discovery settings are unsuitable for production use; at least one of [discovery.seed_hosts, discovery.seed_providers, cluster.initial_master_nodes] must be configured  
ERROR: Elasticsearch did not exit normally - check the logs at /home/elk/elk/zip/elasticsearch-7.7.0/logs/elasticsearch.log  
[2020-05-23T16:28:03,717][INFO ][o.e.n.Node               ] [xiangxue] stopping
```

从错误提示我们可以知道，还需对 Elasticsearch 配置做适当修改，重新编辑 `elasticsearch.yml` 文件，并做如下修改：

```
#----- Node -----  
#  
# Use a descriptive name for the node:  
#  
node.name: node-1  
#  
# Add custom attributes to the node:  
#  
#node.attr.rack: r1  
  
#----- Discovery -----  
#  
# Pass an initial list of hosts to perform discovery  
# The default list of hosts is ["127.0.0.1", "[::1"]  
#  
#discovery.seed_hosts: ["host1", "host2"]  
#  
# Bootstrap the cluster using an initial set of master nodes  
#  
cluster.initial_master_nodes: ["node-1"]  
#  
# For more information, consult the discovery and cluster sections of the Java API documentation.  
#
```

再次启动，并在浏览器中访问



The screenshot shows a browser window with the address bar containing "① 不安全 | enjoy-elk.com:9200". Below the address bar, there are several tabs: 应用 (Application), ProcessOn (highlighted in blue), 常用网站 (常用网站), 微软 Bing 搜索 (Microsoft Bing Search), and 其他 (Other). The main content area displays a JSON object representing the Elasticsearch version information:

```
{  
  "name": "node-1",  
  "cluster_name": "elasticsearch",  
  "cluster_uuid": "jxymaixiTja3kGNhI2FDvw",  
  "version": {  
    "number": "7.7.0",  
    "build_flavor": "default",  
    "build_type": "tar",  
    "build_hash": "81a1e9eda8e6183f5237786246f6dced26a10eaf",  
    "build_date": "2020-05-12T02:01:37.602180Z",  
    "build_snapshot": false,  
    "lucene_version": "8.5.1",  
    "minimum_wire_compatibility_version": "6.8.0",  
    "minimum_index_compatibility_version": "6.0.0-beta1"  
  },  
  "tagline": "You Know, for Search"  
}
```

显示 Elasticsearch 运行并访问成功！

我们知道，Elasticsearch 提供的是 restful 风格接口，我们用浏览器访问不是很方便，除非我们自行编程访问 restful 接口。这个时候，就可以用上 Kibana 了，它已经为我们提供了友好的 Web 界面，方便我们后面对 Elasticsearch 的学习。

接下来我们安装运行 Kibana。

## Kibana

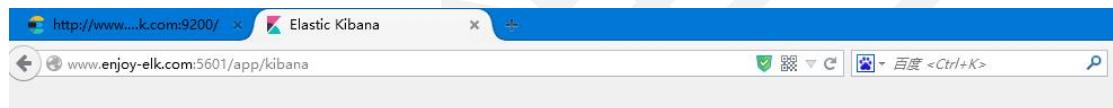
同样用 tar -xvf 命令解压压缩包，进入解压后的目录。为了方便我们的访问和连接 Elasticsearch，也需要进入 Kibana 的 config 目录对 Kibana 进行配置。

```
# To allow connections from remote users, set this parameter to a non-local IP.  
server.host: "172.18.194.140"  
  
# Enables you to specify a path to mount Kibana at if you are running behind a proxy.  
# Use the `server.rewriteBasePath` setting to tell Kibana if it should  
# rewrite requests it receives, and to prevent a deprecation warning at startup.  
# This setting cannot end in a slash.  
server.basePath: ""  
  
# Specifies whether Kibana should rewrite requests that are prefixed with  
# `server.basePath` or require that they are rewritten by your reverse proxy.  
# This setting was effectively always `false` before Kibana 6.3 and will  
# default to `true` starting in Kibana 7.0.  
server.rewriteBasePath: false  
  
# The maximum payload size in bytes for incoming server requests.  
server.maxPayloadBytes: 1048576  
  
# The Kibana server's name. This is used for display purposes.  
server.name: "your-hostname"  
  
# The URLs of the Elasticsearch instances to use for all your queries.  
elasticsearch.hosts: ["http://172.18.194.140:9200"]
```

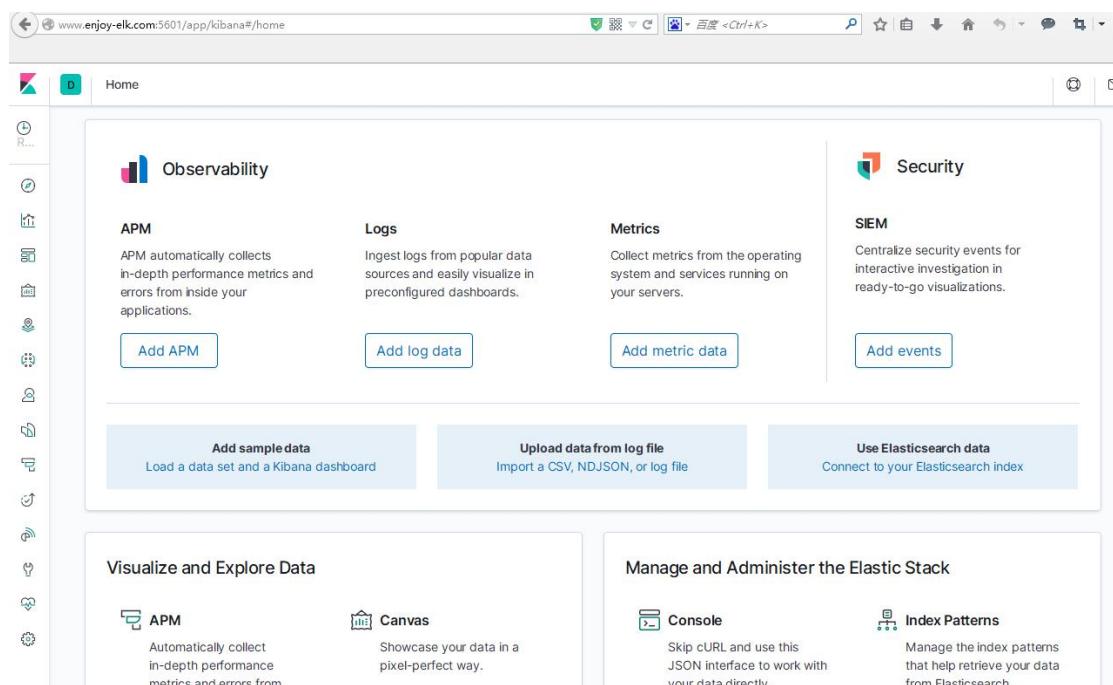
然后进入 Kibana 的 bin 目录运行 ./kibana， kibana 因为是用 node.js 编写的，所以启动和运行较慢，需要等待一段时间：

```
log [09:09:47.408] [warning][reporting] Enabling the Chromium sandbox provides an additional layer on.
log [09:09:49.481] [warning][reporting] Generating a random key for xpack.reporting.encryptionKey.
pending reports from failing on restart, please set xpack.reporting.encryptionKey in kibana.yml
log [09:09:49.487] [info][status][plugin:reporting@7.7.0] Status changed from uninitialized to green
log [09:09:49.541] [info][listening] Server running at http://172.18.194.140:5601
log [09:09:50.081] [info][server][Kibana][http] http server running at http://172.18.194.140:5601
```

从提示我们可以看出， kibana 的访问端口是 5601，我们依然在本地浏览器中访问：



等候几分钟以后，就会进入 kibana 的主界面。



The screenshot shows the main dashboard of the Elastic Kibana application. On the left, there is a sidebar with various icons for different features like APM, Logs, Metrics, Security, and more. The main area is divided into several cards:

- Observability**: Contains sections for APM (with a "Add APM" button), Logs (with a "Add log data" button), and Metrics (with a "Add metric data" button). It also has buttons for "Add sample data", "Upload data from log file", and "Use Elasticsearch data".
- Security**: Contains a section for SIEM (with a "Add events" button) and a brief description: "Centralize security events for interactive investigation in ready-to-go visualizations."
- Visualize and Explore Data**: Contains sections for APM (with a "Add APM" button) and Canvas (with a "Showcase your data in a pixel-perfect way" description).
- Manage and Administer the Elastic Stack**: Contains sections for Console (with a "Skip CURL and use this JSON interface to work with your data directly." description) and Index Patterns (with a "Manage the index patterns that help retrieve your data from Elasticsearch." description).

## TIPS : 如何检测系统中是否启动了 kibana ?

我们一般会用 ps -ef 来查询某个应用是否在 Linux 系统中启动, 比如 Elasticsearch, 我们用 ps -ef|grep java 或者 ps -ef|grep elasticsearch 均可

```
[elk@xiangxue config]$ ps -ef|grep java ←
elk      10786  2328  2 16:40 pts/0    00:00:57 /usr/local/java/jdk1.8.0_77/bin/java -Xshare:auto -Des.networkaddress.cache.ttl=60 -Des.networkaddress.cache.negative.ttl=10 -XX:+AlwaysPreTouch -Xss1m -Djava.awt.headless=true -Dfile.encoding=UTF-8 -Djna.nosys=true -XX:-OmitStackTraceInFastThrow -Dio.netty.noUnsafe=true -Dio.netty.noKeySetOptimization=true -Dio.netty.recycler.maxCapacityPerThread=0 -Dio.nettyallocator.numDirectArenas=0 -Dlog4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Djava.locale.providers=SPI,JRE -Xms1g -Xmx1g -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -Djava.io.tmpdir=/tmp/elasticsearch-1295776085275542248 -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=data -XX:ErrorFile=logs/hs_err_pid%p.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedTime -Xloggc:logs/gc.log -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=32 -XX:GCLogFileSize=64m -XX:MaxDirectMemorySize=536870912 -Des.path.home=/home/elk/zip/elasticsearch-7.7.0 -Des.path.conf=/home/elk/zip/elasticsearch-7.7.0/config -Des.distribution.flavor=default -Des.distribution.type=tar -Des.bundled_jdk=true -cp /home/elk/zip/elasticsearch-7.7.0/lib/* org.elasticsearch.bootstrap.Elasticsearch
elk      14458  6519  0 17:18 pts/2    00:00:00 grep --color=auto java
[elk@xiangxue config]$ ps -ef|grep elasticsearch ←
elk      10786  2328  2 16:40 pts/0    00:00:57 /usr/local/java/jdk1.8.0_77/bin/java -Xshare:auto -Des.networkaddress.cache.ttl=60 -Des.networkaddress.cache.negative.ttl=10 -XX:+AlwaysPreTouch -Xss1m -Djava.awt.headless=true -Dfile.encoding=UTF-8 -Djna.nosys=true -XX:-OmitStackTraceInFastThrow -Dio.netty.noUnsafe=true -Dio.netty.noKeySetOptimization=true -Dio.netty.recycler.maxCapacityPerThread=0 -Dio.nettyallocator.numDirectArenas=0 -Dlog4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Djava.locale.providers=SPI,JRE -Xms1g -Xmx1g -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -Djava.io.tmpdir=/tmp/elasticsearch-1295776085275542248 -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=data -XX:ErrorFile=logs/hs_err_pid%p.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedTime -Xloggc:logs/gc.log -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=32 -XX:GCLogFileSize=64m -XX:MaxDirectMemorySize=536870912 -Des.path.home=/home/elk/zip/elasticsearch-7.7.0 -Des.path.conf=/home/elk/zip/elasticsearch-7.7.0/config -Des.distribution.flavor=default -Des.distribution.type=tar -Des.bundled_jdk=true -cp /home/elk/zip/elasticsearch-7.7.0/lib/* org.elasticsearch.bootstrap.Elasticsearch
elk      10926  10786  0 16:40 pts/0    00:00:00 /home/elk/zip/elasticsearch-7.7.0/modules/x-pack-ml/platform/l
```

但是当我们尝试 ps -ef|grep kibana, 却是不行的

```
[elk@xiangxue config]$ ps -ef|grep kibana
elk      14808  6519  0 17:21 pts/2    00:00:00 grep --color=auto kibana
```

因为 kibana 是 node 写的, 所以 kibana 运行的时候是运行在 node 里面, 我们要查询的话, 只能 ps -ef|grep node

```
[elk@xiangxue config]$ ps -ef|grep node
polkitd   653  302  0 Mar21 ? 02:10:25 /usr/local/lib/erlang/erts-10.7/bin/beam.smp -W w -A 64 -MBas agef
fcbf -MHas ageffcbf -MBlmbcs 512 -MHlmbcs 512 -MMmcbs 30 -P 1048576 -t 5000000 -stbt db -zdbbl 128000 -K true -B i
-- -root /usr/local/lib/erlang -programe erl -- -home /var/lib/rabbitmq -- -pa /opt/rabbitmq/ebin -noshell -noinp
ut -s rabbit boot -sname rabbit@myRabbit -boot start_sasl -conf /etc/rabbitmq/rabbitmq.conf -conf_dix /var/lib/rab
bitmq/config -conf_script_dir /opt/rabbitmq/sbin -conf_schema_dir /var/lib/rabbitmq/schema -conf_advanced /etc/rab
bitmq/advanced.config -kernel inet_default_connect_options [{node=lay,true}] -sasl erlang_type error -sasl sasl_err
or logger tty -rabbit lager log_root "/var/log/rabbitmq" -rabbit lager default_file tty -rabbit lager upgrade_file
tty -rabbit feature_flags_file "/var/lib/rabbitmq/mnesia/rabbit@myRabbit-feature_flags" -rabbit enabled_plugins_f
ile "/etc/rabbitmq/enabled_plugins" -rabbit plugins_dir "/opt/rabbitmq/plugins" -rabbit plugins_expand_dir "/var/l
ib/rabbitmq/mnesia/rabbit@myRabbit-plugins-expand" -os_mon start_cpu_sup false -os_mon start_disksup false -os_mon
_start_memsup false -mnesia dir "/var/lib/rabbitmq/mnesia/rabbit@myRabbit" -ra data_dir "/var/lib/rabbitmq/mnesia/
rabbit@myRabbit/quorum" -kernel inet_dist_listen_min 25672 -kernel inet_dist_listen_max 25672 --
elk      13565  6477  6 17:09 pts/1    00:00:57 ../../node/bin/node ../../src/cli ←
elk      14953  6519  0 17:23 pts/2    00:00:00 grep --color=auto node
```

或者使用 netstat -tunlp|grep 5601

```
[elk@xiangxue config]$ netstat -tunlp|grep 5601
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp      0      0  172.18.194.140:5601    0.0.0.0:*
                                              LISTEN      13565/../../node/bin
```

因为我们的 kibana 开放的端口是 5601, 所以看到 5601 端口被 Listen (监听), 说明 kibana 启动成功。

附: netstat 参数说明:

-t (tcp)仅显示 tcp 相关选项

-u (udp)仅显示 udp 相关选项

-n 拒绝显示别名, 能显示数字的全部转化成数字。

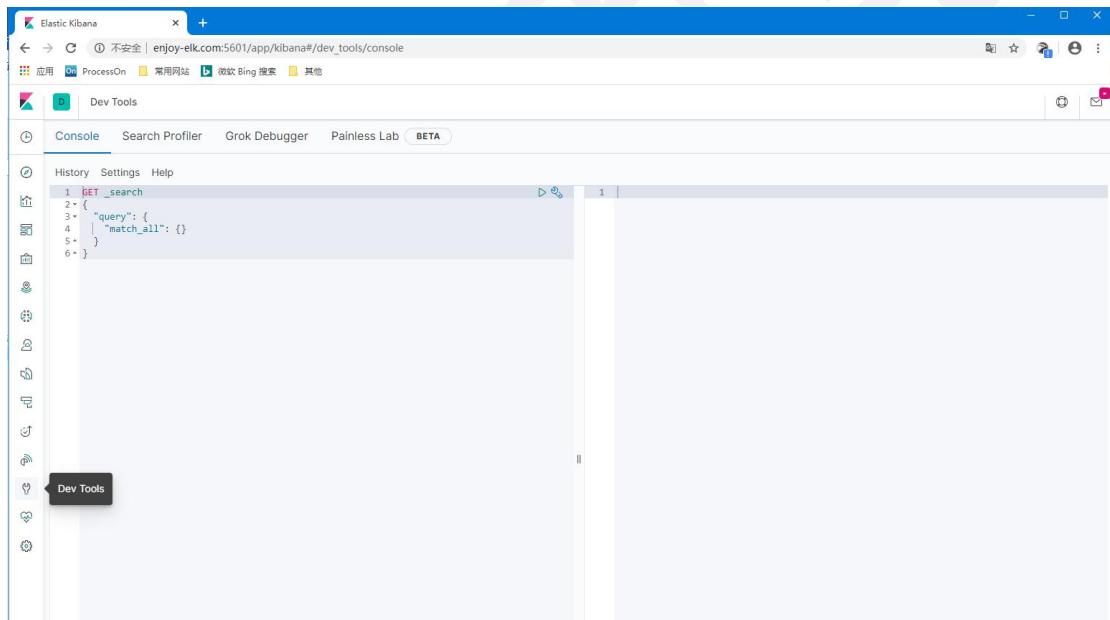
-l 仅列出有在 Listen (监听) 的服务状态

-p 显示建立相关链接的程序名

更多 netstat 的相关说明，自行查阅 Linux 手册。

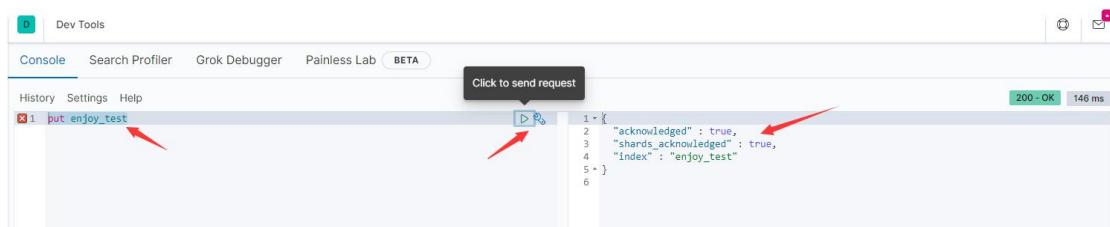
## HelloWorld

点击面板中的“Dev Tools”按钮，进入 Dev 工具，开始我们的 Elasticsearch 初次访问之旅。



## 创建索引

在左边命令窗口中输入 put enjoy\_test，并点击相关按钮 ，于是看到右边的结果窗口中 es 给我们返回了处理结果，表示我们在 es 中创建索引成功。



## 查看索引

要查看我们刚刚创建的索引，执行“get enjoy\_test”

```
① 1 get enjoy_test
  2+ {
  3   "enjoy_test": {
  4     "aliases": {},
  5     "mappings": {},
  6     "settings": {
  7       "index": {
  8         "creation_date": "1590227502669",
  9         "number_of_shards": "1",
 10        "number_of_replicas": "1",
 11        "uuid": "2ecFo5HKQJGnxgvBwTCTw",
 12        "version": {
 13          "created": "7070099"
 14        },
 15        "provided_name": "enjoy_test"
 16      }
 17    }
 18  }
```

## 添加文档

往这个索引中添加文档，执行：

```
PUT /enjoy_test/_doc/1
{
  "msg": "Hello World!"
}
```

```
PUT /enjoy_test/_doc/1
{
  "msg": "Hello World!"
}
```

② 1+ {
 2 "\_index": "enjoy\_test",
 3 "\_type": "\_doc",
 4 "\_id": "1",
 5 "\_version": 1,
 6 "result": "created",
 7 "\_shards": {
 8 "total": 2,
 9 "successful": 1,
 10 "failed": 0
 11 },
 12 "\_seq\_no": 0,
 13 "\_primary\_term": 1
 14 }

## 查看文档

查询我们刚刚加入的文档，执行“get /enjoy\_test/\_doc/1”

```
get /enjoy_test/_doc/1

```

③ 1+ {
 2 "\_index": "enjoy\_test",
 3 "\_type": "\_doc",
 4 "\_id": "1",
 5 "\_version": 1,
 6 "seq\_no": 0,
 7 "\_primary\_term": 1,
 8 "found": true,
 9 "source": {
 10 "msg": "Hello World!"
 11 }
 12 }

# Elasticsearch 基本原理和概念

从上面的例子中，我们看到了很多熟悉又陌生的概念，比如索引、文档等等。这些概念和我们平时看到的数据库里的比如索引有什么区别呢？

---

举个例子，现在我们要保存唐宋诗词，数据库中我们们会怎么设计？诗词表我们可能的设计如下：

朝代	作者	诗词年代	标题	诗词全文
唐	李白		静夜思	床前明月光，疑是地上霜。 举头望明月，低头思故乡。
宋	李清照		如梦令	常记溪亭日暮，沉醉不知归路，兴尽晚回舟，误入藕花深处。争渡，争渡，惊起一滩鸥鹭。
....	....	...	....	.....

要根据朝代或者作者寻找诗，都很简单，比如“select 诗词全文 from 诗词表 where 作者= ‘李白’ ”，如果数据很多，查询速度很慢，怎么办？我们可以在对应的查询字段上建立索引加速查询。

但是如果我们现在有个需求：要求找到包含“望”字的诗词怎么办？用

“select 诗词全文 from 诗词表 where 诗词全文 like ‘%望%’ ”，这个意味着要扫描库中的诗词全文字段，逐条比对，找出所有包含关键词“望”字的记录，。基本上，数据库中一般的 SQL 优化手段都是用不上的。数量少，大概性能还能接受，如果数据量稍微大点，就完全无法接受了，更何况在互联网这种海量数据的情况下呢？

怎么解决这个问题呢，用倒排索引。

## 倒排索引 Inverted index

比如现在有：

蜀道难（唐）李白 蜀道之难难于上青天，侧身西望长咨嗟。

静夜思（唐）李白 举头望明月，低头思故乡。

春台望（唐）李隆基 暇景属三春，高台聊四望。

鹤冲天(宋)柳永 黄金榜上，偶失龙头望。明代暂遗贤，如何向？未遂风云便，争不恣狂荡。何须论得丧？才子词人，自是白衣卿相。烟花巷陌，依约丹青屏障。幸有意中人，堪寻访。且恁偎红翠，风流事，平生畅。青春都一饷。忍把浮名，换了浅斟低唱！

都有望字，于是我们可以这么保存

序号	关键字	蜀道难	静夜思	春台望	鹤冲天
1	望	有	有	有	有

---

如果查哪个诗词中包含上，怎么办，上述的表格可以继续填入新的记录

序号	关键字	蜀道难	静夜思	春台望	鹤冲天
1	望	有	有	有	有
2	上	有			有

其实，上述诗词的中每个字都可以作为关键字，然后建立关键字和文档之间的对应关系，也就是标识关键字被哪些文档包含。

所以，倒排索引就是，将文档中包含的关键字全部提取处理，然后再将关键字和文档之间的对应关系保存起来，最后再对关键字本身做索引排序。用户在检索某一个关键字时，先对关键字的索引进行查找，再通过关键字与文档的对应关系找到所在文档。

在存储在关系型数据库中的数据，需要我们事先分析将数据拆分为不同的字段，而在 es 这类的存储中，需要应用程序根据规则自动提取关键字，并形成对应关系。

这些预先提取的关键字，在全文检索领域一般被称为 term（词项），文档的词项提取在 es 中被称为文档分析，这是全文检索很核心的过程，必须要区分哪些是词项，哪些不是，比如很多场景下，apple 和 apples 是同一个东西，望和看其实是同一个动作。

## Elasticsearch 基本概念

Elasticsearch 中比较关键的基本概念有索引、文档、映射、映射类型、文档字段概念，为了方便理解，可以和关系数据库中的相关概念进行个比对：

关系数据库	ES
库	索引 (Index)
表	映射类型 (Mapping Type)
数据行	文档 (Document)
字段	文档字段 (Field)
表结构	映射 (Mapping)

## Elasticsearch 索引

Elasticsearch 索引是映射类型的容器。一个 Elasticsearch 索引非常像关系型世界的数据库，是独立的大量文档集合。

当然在底层，肯定用到了倒排索引，最基本的结构就是“keyword”和“Posting List”， Posting list 就是一个 int 的数组，存储了所有符合某个 term 的文档 id。

---

另外，这个倒排索引相比特定词项出现过的文档列表，会包含更多其它信息。它会保存每一个词项出现过的文档总数，在对应的文档中一个具体词项出现的总次数，词项在文档中的顺序，每个文档的长度，所有文档的平均长度等等相关信息。

## 文档 (Document)

文档是 es 中所有可搜索数据的最小单位，比如日志文件中的日志项、一部电影的具体信息等等。

文档会被序列化 JSON 格式保存到 ElasticSearch 中，JSON 对象由字段组成，每个字段都有对象的字段类型（字符串，数值，布尔，日期，二进制，范围类型）。同时每个文档都有一个 Unique ID，可以自己指定 ID，或者通过 ElasticSearch 自动生成。

所以严格来说，es 中存储的文档是一种半结构化的数据。

## 映射

映射 (mapping) 定义了每个字段的类型、字段所使用的分词器等。

```
get /enjoy_test/_mapping
```



```
1 {  
2   "enjoy_test" : {  
3     "mappings" : {  
4       "properties" : {  
5         "msg" : {  
6           "type" : "text",  
7           "fields" : {  
8             "keyword" : {  
9               "type" : "keyword",  
10              "ignore_above" : 256  
11            }  
12          }  
13        }  
14      }  
15    }  
16  }  
17}  
18}
```

可以显式映射，由我们在索引映射中进行预先定义；也可以动态映射，在添加文档的时候，由 es 自动添加到索引，这个过程不需要事先在索引进行字段数据类型匹配等等，es 会自己推断数据类型。

既然说到了字段类型，当然就离不开字段的数据类型了。

## 文档字段

文档中的一个字段 field 就相当于关系型数据库中的一列 column，那么它肯定有数据类型，es 提供的数据类型包括至少有：

### 数据类型

#### 核心数据类型

---

# 字符串类型：string，字符串类还可被分为 text 和 keyword 类型，如果我们让 es 自动映射数据，那么 es 会把字符串定义为 text，并且还加了一个 keyword 类型字段。

text 文本数据类型，用于索引全文值的字段。使用文本数据类型的字段，它们会被分词，在索引之前将字符串转换为单个术语的列表(倒排索引)，分词过程允许 ES 搜索每个全文字段中的单个单词。什么情况适合使用 text，只要不具备唯一性的字符串一般都可以使用 text。

keyword，关键字数据类型，用于索引结构化内容的字段。使用 keyword 类型的字段，其不会被分析，给什么值就原封不动地按照这个值索引，所以关键字字段只能按其确切值进行搜索。什么情况下使用 keyword，具有唯一性的字符串，例如：电子邮件地址、MAC 地址、身份证号、状态代码...等等。

# 数字型数据类型：long、integer、short、byte、double、float

# 日期类型：date

# 布尔类型：boolean

### 复杂数据类型

# 数组：无需专门的数据类型

# 对象数据类型：单独的 JSON 对象

# 嵌套数据类型：nested，关于 JSON 对象的数组

### 地理数据类型：

# 地理点数据类型

# 地理形状数据类型

### 专门数据类型：

# IPv4 数据类型

# 单词计数数据类型 token\_count

我们结合前面的映射来看看：

创建一个新的索引：put /open-soft

### 显式映射：

```
put /open-soft/_mapping
```

```
{
```

```
  "properties" : {
```

```
    "corp" : {
```

```
      "type" : "text"
```

```
    },
```

```
    "lang" : {
```

```
      "type" : "text"
```

```
    },
```

```
    "name" : {
```

```
        "type" : "text"
    }
}
}
```

索引或者说入库一个文档，注意这个文档的字段，比我们显示映射的字段要多个 star 字段：

```
put /open-soft/_doc/1
{
  "name": "Apache Hadoop",
  "lang": "Java",
  "corp": "Apache",
  "stars":200
}
```

通过 get /open-soft/\_mapping，我们可以看到 es 自动帮我们新增了 stars 这个字段。

修改映射，增加一个新的字段：

```
put /open-soft/_mapping
{
  "properties" : {
    "year" :{
      "type" : "integer"
    }
  }
}
```

## 数组

不需要特殊配置，一个字段如果被配置为基本数据类型，就是天生支持数组类型的。任何字段都可以有 0 个或多个值，但是在一个数组中数据类型必须一样。

比如：

```
put /open-soft/_doc/2
{
  "name": ["Apache Activemq","Activemq Artemis"],
  "lang": "Java",
  "corp": "Apache",
  "stars": [500,200]
}
```

是没问题的，但是如果：

```
put /open-soft/_doc/3
{
  "name": ["Apache Kafka"],
  "lang": "Java",
  "corp": "Apache",
  "stars": [500, "kafka"]
}
```

则会出错。

## 对象

JSON 文档是有层次结构的，一个文档可能包含其他文档，如果一个文档包含其他文档，那么该文档值是对象类型，其数据类型是对象。当然 Elasticsearch 中是没有所谓对象类型的，比如：

```
put /open-soft/_doc/object
{
  "name": ["Apache ShardingSphere"],
  "lang": "Java",
  "corp": "JingDong",
  "stars": 400,
  "address": {
    "city": "BeiJing",
    "country": "亦庄"
  }
}
```

```
get /open-soft/_mapping
1 ▶ { "open-soft" : {
2 ▶   "mappings" : {
3 ▶     "properties" : {
4 ▶       "address" : {
5 ▶         "properties" : {
6 ▶           "city" : {
7 ▶             "type" : "text",
8 ▶             "fields" : {
9 ▶               "keyword" : {
10 ▶                 "type" : "keyword",
11 ▶                 "ignore_above" : 256
12 ▶               }
13 ▶             }
14 ▶           }
15 ▶         },
16 ▶         "country" : {
17 ▶           "type" : "text",
18 ▶           "fields" : {
19 ▶             "keyword" : {
20 ▶               "type" : "keyword",
21 ▶               "ignore_above" : 256
22 ▶             }
23 ▶           }
24 ▶         }
25 ▶       },
26 ▶     },
27 ▶   "corp" : {
28 ▶     "type" : "text"
29 ▶   }
30 ▶ }
```

---

对象类型可以在定义索引的映射关系时进行指定。

## 多数据类型

如果说数组允许你使用同一个设置索引多项数据，那么多数据类型允许使用不同的设置，对同一项数据索引多次。带来的好处就是可以同一文本有多种不同的索引方式，比如一个字符串类型的字段，可以使用 `text` 类型做全文检索，使用 `keyword` 类型做聚合和排序。我们可以看到 es 的动态映射生成的字段类型里，往往字符串类型都使用了多数据类型。当然，我们一样也可以自己定义：

```
put /open-soft/_mapping
{
  "properties": {
    "name": {
      "type": "text",
      "fields": {
        "raw": {
          "type": "keyword"
        },
        "length": {
          "type": "token_count",
          "analyzer": "standard"
        }
      }
    }
  }
}
```

在上面的代码里，我们使用`"fields"`就把 `name` 字段扩充为多字段类型，为 `name` 新增了两个子字段 `raw` 和 `length`，`raw` 设置类型为 `keyword`，`length` 设置类型为 `token_count`，告诉 es 这个字段在保存还需要做词频统计。

通过 `fields` 字段设置的子字段 `raw` 和 `length`，在我们添加文档时，并不需要单独设置值，他们 `name` 共享相同的值，只是 es 会以不同的方式处理字段值。同样在检索文档的时候，它们也不会显示在结果中，所以它们一般都是在检索中以查询条件的形式出现，以减少检索时的性能开销。

## 字段参数

在上面的代码里出现了 `analyzer` 这个词，这是什么？这个叫字段参数，和 `type` 一样，可以用来对字段进行配置。常用的字段参数和作用如下：

---

### **analyzer**

指定分词器。elasticsearch 是一款支持全文检索的分布式存储系统，对于 `text` 类型的字段，首先会使用分词器进行分词，然后将分词后的词根一个一个存储在倒排索引中，后续查询主要是针对词根的搜索。

`analyzer` 该参数可以在每个查询、每个字段、每个索引中使用，其优先级如下（越靠前越优先）：

- 1、字段上定义的分词器
- 2、索引配置中定义的分词器
- 3、默认分词器(standard)

### **normalizer**

规范化，主要针对 `keyword` 类型，在索引该字段或查询字段之前，可以先对原始数据进行一些简单的处理，然后再将处理后的结果当成一个词根存入倒排索引中，默认为 `null`，比如：

```
PUT index
{
  "settings": {
    "analysis": {
      "normalizer": {
        "my_normalizer": {
          "type": "custom",
          "char_filter": [],
          "filter": ["lowercase", "asciifolding"]           // 1
        }
      }
    }
  },
  "mappings": {
    "_doc": {
      "properties": {
        "foo": {
          "type": "keyword",
          "normalizer": "my_normalizer"                  // 2
        }
      }
    }
  }
}
```

---

```
}
```

代码 1：首先在 `settings` 中的 `analysis` 属性中定义 `normalizer`。

代码 2：设置标准化过滤器，示例中的处理器为小写、`asciifolding`。

代码 3：在定义映射时，如果字段类型为 `keyword`，可以使用 `normalizer` 引用定义好的 `normalizer`

### **boost**

权重值，可以提升在查询时的权重，对查询相关性有直接的影响，其默认值为 1.0。其影响范围为词根查询(`team query`)，对前缀、范围查询。5.0 版本后已废止。

### **coerce**

数据不总是我们想要的，由于在转换 `JSON body` 为真正 `JSON` 的时候，整型数字 5 有可能会被写成字符串 "5" 或者浮点数 5.0，这个参数可以将数值不合法的部分去除。默认为 `true`。

例如：将字符串会被强制转换为整数、浮点数被强制转换为整数。

例如存在如下字段类型：

```
"number_one": {  
    "type": "integer"  
}
```

声明 `number_one` 字段的类型为数字类型，那是否允许接收 “6” 字符串形式的数据呢？因为在 `JSON` 中，“6” 用来赋给 `int` 类型的字段，也是能接受的，默认 `coerce` 为 `true`，表示允许这种赋值，但如果 `coerce` 设置为 `false`，此时 `es` 只能接受不带双引号的数字，如果在 `coerce=false` 时，将“6”赋值给 `number_one` 时会抛出类型不匹配异常。

### **copy\_to**

`copy_to` 参数允许您创建自定义的 `_all` 字段。换句话说，多个字段的值可以复制到一个字段中。

例如，`first_name` 和 `last_name` 字段可以复制到 `full_name` 字段如下：

```
PUT my_index  
{  
    "mappings": {  
        "_doc": {  
            "properties": {  
                "first_name": {  
                    "type": "text",  
                    "copy_to": "full_name"  
                },  
                "last_name": {  
                    "type": "text",  
                    "copy_to": "full_name"  
                }  
            }  
        }  
    }  
}
```

---

```

        "type": "text",
        "copy_to": "full_name"
    },
    "full_name": {
        "type": "text"
    }
}
}
}
}

```

表示字段 `full_name` 的值来自 `first_name + last_name`。

关于 `copy_to` 重点说明：

- 1、字段的复制是原始值。
- 2、同一个字段可以复制到多个字段，写法如下：“`copy_to": [ "field_1", "field_2" ]`

### *doc\_values*

`Doc values` 的存在是因为倒排索引只对某些操作是高效的。倒排索引的优势 在于查找包含某个项的文档，而对于从另外一个方向的相反操作并不高效，即：确定哪些项是否存在单个文档里，聚合需要这种次级的访问模式。

对于以下倒排索引：

Term	Doc_1	Doc_2	Doc_3
brown	X	X	
dog	X		X
dogs		X	X
fox	X		X
foxes		X	
in		X	
jumped	X		X
lazy	X	X	
leap		X	
over	X	X	X
quick	X	X	X
summer		X	
the	X		X

如果我们想要获得所有包含 `brown` 的文档的词的完整列表，倒排索引是根据项来排序的，所以我们首先在词项列表中找到 `brown`，然后扫描所有列，找到包含 `brown` 的文档。我们可以快速看到 `Doc_1` 和 `Doc_2` 包含 `brown` 这个 `token`。

---

然后，对于聚合部分，我们需要找到 Doc\_1 和 Doc\_2 里所有唯一的词项。用倒排索引做这件事情代价很高：我们会迭代索引里的每个词项并收集 Doc\_1 和 Doc\_2 列里面 token。这很慢而且难以扩展：随着词项和文档的数量增加，执行时间也会增加。

Doc values 通过转置两者间的关系来解决这个问题。倒排索引将词项映射到包含它们的文档，doc values 将文档映射到它们包含的词项：

Doc	Terms
<hr/>	
---	
Doc_1	brown, dog, fox, jumped, lazy, over, quick, the
Doc_2	brown, dogs, foxes, in, lazy, leap, over, quick, summer
Doc_3	dog, dogs, fox, jumped, over, quick, the
<hr/>	
---	

当数据被转置之后，想要收集到 Doc\_1 和 Doc\_2 的唯一 token 会非常容易。获得每个文档行，获取所有的词项，然后求两个集合的并集。

doc\_values 缺省是 true，即是开启的，并且只适用于非 text 类型的字段。

### **dynamic**

是否允许动态的隐式增加字段。在执行 index api 或更新文档 API 时，对于 \_source 字段中包含一些原先未定义的字段采取的措施，根据 dynamic 的取值，会进行不同的操作：

true，默认值，表示新的字段会加入到类型映射中。

false，新的字段会被忽略，即不会存入 source 字段中，即不会存储新字段，也无法通过新字段进行查询。

strict，会显示抛出异常，需要先使用 put mapping api 先显示增加字段映射。

### **enabled**

是否建立索引，默认情况下为 true，es 会尝试为你索引所有的字段，但有时候某些类型的字段，无需建立索引，只是用来存储数据即可。也就是说，Elasticsearch 默认会索引所有的字段，enabled 设为 false 的字段，elasticsearch 会跳过字段内容，该字段只能从 source 中获取，但是不可搜。只有映射类型(type)和 object 类型的字段可以设置 enabled 属性。

### **eager\_global\_ordinals**

表示是否提前加载全局顺序号。Global ordinals 是一个建立在 doc values 和 fielddata 基础上的数据结构，它为每一个精确词按照字母顺序维护递增的编号。每一个精确词都有一个独一无二的编号 并且 精确词 A 小于精确词 B 的编号。Global ordinals 只支持 keyword 和 text 型字段，在 keyword 字段中，默认是启用的 而在 text 型字段中 只有 fielddata 和相关属性开启的状态下才是可用的。

## **fielddata**

为了解决排序与聚合, elasticsearch 提供了 doc\_values 属性来支持列式存储, 但 doc\_values 不支持 text 字段类型。因为 text 字段是需要先分析 (分词), 会影响 doc\_values 列式存储的性能。

es 为了支持 text 字段高效排序与聚合, 引入了一种新的数据结构(fielddata), 使用内存进行存储。默认构建时机为第一次聚合查询、排序操作时构建, 主要存储倒排索引中的词根与文档的映射关系, 聚合, 排序操作在内存中执行。因此 fielddata 需要消耗大量的 JVM 堆内存。一旦 fielddata 加载到内存后, 它将永久存在。

通常情况下, 加载 fielddata 是一个昂贵的操作, 故默认情况下, text 字段的字段默认是不开启 fielddata 机制。在使用 fielddata 之前请慎重考虑为什么要开启 fielddata。

## **format**

在 JSON 文档中, 日期表示为字符串。Elasticsearch 使用一组预先配置的格式来识别和解析这些字符串, 并将其解析为 long 类型的数值(毫秒), 支持自定义格式, 也有内置格式。

比如:

```
PUT my_index
{
  "mappings": {
    "_doc": {
      "properties": {
        "date": {
          "type": "date",
          "format": "yyyy-MM-dd HH:mm:ss"
        }
      }
    }
  }
}
```

elasticsearch 为我们内置了大量的格式, 如下:

**epoch\_millis**

时间戳, 单位, 毫秒, 范围受限于 Java Long.MIN\_VALUE 和 Long.MAX\_VALUE。

**epoch\_second**

时间戳, 单位, 秒, 范围受限于 Java 的限制 Long.MIN\_VALUE 并 Long.MAX\_VALUE 除以 1000 (一秒中的毫秒数)。

**date\_optional\_time** 或者 **strict\_date\_optional\_time**

日期必填, 时间可选, 其支持的格式如下:

---

```
date-opt-time = date-element ['T' [time-element] [offset]]
date-element = std-date-element | ord-date-element | week-date-element
std-date-element = yyyy ['-' MM ['- dd]]
ord-date-element = yyyy ['- DDD]
week-date-element = xxxx '-W' ww ['- e]
time-element = HH [minute-element] | [fraction]
minute-element = ':' mm [second-element] | [fraction]
second-element = ':' ss [fraction]

比如"yyyy-MM-dd"、"yyyyMMdd"、"yyyyMMddHHmmss"、
"yyyy-MM-ddTHH:mm:ss"、"yyyy-MM-ddTHH:mm:ss.SSS"、
"yyyy-MM-ddTHH:mm:ss.SSSZ"格式，不支持常用的"yyyy-MM-dd HH:mm:ss"等格式。
注意，"T"和"Z"是固定的字符。
```

**tips:** 如果看到“`strict_`”前缀的日期格式要求，表示`date_optional_time`的严格级别，这个严格指的是年份、月份、天必须分别以4位、2位、2位表示，不足两位的话第一位需用0补齐。

#### `basic_date`

其格式表达式为：`yyyyMMdd`

#### `basic_date_time`

其格式表达式为：`yyyyMMdd' T' HHmmss.SSSZ`

#### `basic_date_time_no_millis`

其格式表达式为：`yyyyMMdd' T' HHmmssZ`

#### `basic_ordinal_date`

4位数的年 + 3位(day of year)，其格式字符串为`yyyyDDD`

#### `basic_ordinal_date_time`

其格式字符串为`yyyyDDD' T' HHmmss.SSSZ`

#### `basic_ordinal_date_time_no_millis`

其格式字符串为`yyyyDDD' T' HHmmssZ`

#### `basic_time`

其格式字符串为`HHmmss.SSSZ`

#### `basic_time_no_millis`

其格式字符串为`HHmmssZ`

#### `basic_t_time`

其格式字符串为`' T' HHmmss.SSSZ`

#### `basic_t_time_no_millis`

其格式字符串为`' T' HHmmssZ`

#### `basic_week_date`

---

其格式字符串为 xxxx' W' wwe, 4 为年 , 然后用' W' ,2 位 week of year  
(所在年里周序号) 1 位 day of week。

basic\_week\_date\_time

其格式字符串为 xxxx' W' wwe' T' HH:mm:ss.SSSZ.

basic\_week\_date\_time\_no\_millis

其格式字符串为 xxxx' W' wwe' T' HH:mm:ssZ.

date

其格式字符串为 yyyy-MM-dd

date\_hour

其格式字符串为 yyyy-MM-dd' T' HH

date\_hour\_minute

其格式字符串为 yyyy-MM-dd' T' HH:mm

date\_hour\_minute\_second

其格式字符串为 yyyy-MM-dd' T' HH:mm:ss

date\_hour\_minute\_second\_fraction

其格式字符串为 yyyy-MM-dd' T' HH:mm:ss.SSS

date\_hour\_minute\_second\_millis

其格式字符串为 yyyy-MM-dd' T' HH:mm:ss.SSS

date\_time

其格式字符串为 yyyy-MM-dd' T' HH:mm:ss.SSS

date\_time\_no\_millis

其格式字符串为 yyyy-MM-dd' T' HH:mm:ss

hour

其格式字符串为 HH

hour\_minute

其格式字符串为 HH:mm

hour\_minute\_second

其格式字符串为 HH:mm:ss

hour\_minute\_second\_fraction

其格式字符串为 HH:mm:ss.SSS

hour\_minute\_second\_millis

其格式字符串为 HH:mm:ss.SSS

ordinal\_date

其格式字符串为 yyyy-DDD,其中 DDD 为 day of year。

ordinal\_date\_time

其格式字符串为 yyyy-DDD 'T' HH:mm:ss.SSSZZ,其中 DDD 为 day of year。

---

`ordinal_date_time_no_millis`

其格式字符串为 `yyyy-DDD 'T' HH:mm:ssZZ`

`time`

其格式字符串为 `HH:mm:ss.SSSZZ`

`time_no_millis`

其格式字符串为 `HH:mm:ssZZ`

`t_time`

其格式字符串为 `'T' HH:mm:ss.SSSZZ`

`t_time_no_millis`

其格式字符串为 `'T' HH:mm:ssZZ`

`week_date`

其格式字符串为 `xxxx-'W' ww-e`, 4 位年份, `ww` 表示 week of year, `e` 表示 day of week。

`week_date_time`

其格式字符串为 `xxxx-'W' ww-e 'T' HH:mm:ss.SSSZZ`

`week_date_time_no_millis`

其格式字符串为 `xxxx-'W' ww-e 'T' HH:mm:ssZZ`

`weekyear`

其格式字符串为 `xxxx`

`weekyear_week`

其格式字符串为 `xxxx-'W' ww`, 其中 `ww` 为 week of year。

`weekyear_week_day`

其格式字符串为 `xxxx-'W' ww-e`, 其中 `ww` 为 week of year, `e` 为 day of week。

`year`

其格式字符串为 `yyyy`

`year_month`

其格式字符串为 `yyyy-MM`

`year_month_day`

其格式字符串为 `yyyy-MM-dd`

### `ignore_above`

`ignore_above` 用于指定字段索引和存储的长度最大值, 超过最大值的会被忽略。

### `ignore_malformed`

`ignore_malformed` 可以忽略不规则数据, 对于 `login` 字段, 有人可能填写的是 `date` 类型, 也有人填写的是邮件格式。给一个字段索引不合适的数据类型发

---

生异常，导致整个文档索引失败。如果 `ignore_malformed` 参数设为 `true`，异常会被忽略，出异常的字段不会被索引，其它字段正常索引。

### **index**

`index` 属性指定字段是否索引，不索引也就不可搜索，取值可以为 `true` 或者 `false`，缺省为 `true`。

### **index\_options**

`index_options` 控制索引时存储哪些信息到倒排索引中，，用于搜索和突出显示目的。

`docs` 只存储文档编号

`freqs` 存储文档编号和词项频率。

`positions` 文档编号、词项频率、词项的位置被存储

`offsets` 文档编号、词项频率、词项的位置、词项开始和结束的字符位置都被存储。

### **fields**

`fields` 可以让同一文本有多种不同的索引方式，比如一个 `String` 类型的字段，可以使用 `text` 类型做全文检索，使用 `keyword` 类型做聚合和排序。

### **norms**

`norms` 参数用于标准化文档，以便查询时计算文档的相关性。`norms` 虽然对评分有用，但是会消耗较多的磁盘空间，如果不需要对某个字段进行评分，最好不要开启 `norms`。

### **null\_value**

一般来说值为 `null` 的字段不索引也不可以搜索，`null_value` 参数可以让值为 `null` 的字段显式的可索引、可搜索。

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "status_code": {
          "type": "keyword",
          "null_value": "NULL"
        }
      }
    }
  }
}
```

---

```
}

PUT my_index/my_type/1
{
    "status_code": null
}

PUT my_index/my_type/2
{
    "status_code": []
}

GET my_index/_search
{
    "query": {
        "term": {
            "status_code": "NULL"
        }
    }
}
```

文档 1 可以被搜索到，因为 status\_code 的值为 null，文档 2 不可以被搜索到，因为 status\_code 为空数组，但是不是 null。

### ***position\_increment\_gap***

文本数组元素之间位置信息添加的额外值。

举例，一个字段的值为数组类型: "names": [ "John Abraham", "Lincoln Smith"]

为了区别第一个字段和第二个字段，Abraham 和 Lincoln 在索引中有一个间距，默认是 100。例子如下，这是查询” Abraham Lincoln” 是查不到的：

```
PUT my_index/groups/1
{
    "names": [ "John Abraham", "Lincoln Smith"]
}
```

```
GET my_index/groups/_search
{
    "query": {
        "match_phrase": {
```

```
        "names": {  
            "query": "Abraham Lincoln"  
        }  
    }  
}  
}
```

指定间距大于 100 可以查询到：

```
GET my_index/groups/_search  
{  
    "query": {  
        "match_phrase": {  
            "names": {  
                "query": "Abraham Lincoln",  
                "slop": 101  
            }  
        }  
    }  
}
```

想要调整这个值，在 mapping 中通过 position\_increment\_gap 参数指定间距即可。

### **properties**

Object 或者 nested 类型，下面还有嵌套类型，可以通过 properties 参数指定。比如：

```
PUT my_index  
{  
    "mappings": {  
        "my_type": {  
            "properties": {  
                "manager": {  
                    "properties": {  
                        "age": { "type": "integer" },  
                        "name": { "type": "text" }  
                    }  
                },  
            },  
        },  
    },
```

```
    "employees": {  
        "type": "nested",  
        "properties": {  
            "age": { "type": "integer" },  
            "name": { "type": "text" }  
        }  
    }  
}  
}  
}  
}
```

对应的文档结构:

```
PUT my_index/my_type/1  
{  
    "region": "US",  
    "manager": {  
        "name": "Alice White",  
        "age": 30  
    },  
    "employees": [  
        {  
            "name": "John Smith",  
            "age": 34  
        },  
        {  
            "name": "Peter Brown",  
            "age": 26  
        }  
    ]  
}
```

### ***search\_analyzer***

通常，在索引时和搜索时应用相同的分析器，以确保查询中的术语与反向索引中的术语具有相同的格式，如果想要在搜索时使用与存储时不同的分词器，则使用 `search_analyzer` 属性指定，通常用于 ES 实现即时搜索(`edge_ngram`)。

### ***similarity***

指定相似度算法，其可选值：

---

## BM25

当前版本的默认值，使用 BM25 算法。

### classic

使用 TF/IDF 算法，曾经是 es,lucene 的默认相似度算法。

### boolean

一个简单的布尔相似度，当不需要全文排序时使用，并且分数应该只基于查询条件是否匹配。布尔相似度为术语提供了一个与它们的查询 **boost** 相等的分数。

### **store**

默认情况下，字段值被索引以使其可搜索，但它们不存储。这意味着可以查询字段，但无法检索原始字段值。通常这并不重要。字段值已经是 `_source` 字段的一部分，该字段默认存储。如果您只想检索单个字段或几个字段的值，而不是整个 `_source`，那么这可以通过字段过滤上下文 `source filtering context` 来实现。

在某些情况下，存储字段是有意义的。例如，如果您有一个包含标题、日期和非常大的内容字段的文档，您可能只想检索标题和日期，而不需要从大型 `_source` 字段中提取这些字段，可以将标题和日期字段的 `store` 定义为 `true`。

### **term\_vector**

`Term vectors` 包含分析过程产生的索引词信息，包括：

索引词列表

每个索引词的位置（或顺序）

索引词在原始字符串中的原始位置中的开始和结束位置的偏移量。

`term vectors` 会被存储，索引它可以作为一个特使的文档返回。

`term_vector` 可取值：

`no`

不存储 `term_vector` 信息， 默认值。

`yes`

只存储字段中的值。

`with_positions`

存储字段中的值与位置信息。

`with_offsets`

存储字段中的值、偏移量

`with_positions_offsets`

存储字段中的值、位置、偏移量信息。

## 元字段 meta-fields

一个文档根据我们定义的业务字段保存有数据之外，它还包含了元数据字段 (`meta-fields`)。元字段不需要用户定义，在任一文档中都存在，有点类似于数据库的表结构数据。在名称上有个显著的特征，都是以下划线 “`_`” 开头。

---

我们可以看看: `get /open-soft/_doc/1`

大体分为五种类型: 身份(标识)元数据、索引元数据、文档元数据、路由元数据以及其他类型的元数据,当然不是每个文档这些元字段都有的。

## 身份(标识)元数据

`_index`: 文档所属索引,自动被索引,可被查询,聚合,排序使用,或者脚本里访问

`_type`: 文档所属类型,自动被索引,可被查询,聚合,排序使用,或者脚本里访问

`_id`: 文档的唯一标识,建索引时候传入,不被索引,可通过`_uid`被查询,脚本里使用,不能参与聚合或排序

`_uid`: 由`_type`和`_id`字段组成,自动被索引,可被查询,聚合,排序使用,或者脚本里访问,6.0.0版本后已废止。

## 索引元数据

`_all`: 自动组合所有的字段值,以空格分割,可以指定分词词索引,但是整个值不被存储,所以此字段仅仅能被搜索,不能获取到具体的值。6.0.0版本后已废止。

`_field_names`: 索引了每个字段的名字,可以包含`null`值,可以通过`exists`查询或`missing`查询方法来校验特定的字段

## 文档元数据

`_source`: 一个`doc`的原生的`json`数据,不会被索引,用于获取提取字段值,启动此字段,索引体积会变大,如果既想使用此字段又想兼顾索引体积,可以开启索引压缩。

`_source`是可以被禁用的,不过禁用之后部分功能不再支持,这些功能包括:

部分`update api`、运行时高亮搜索结果

索引重建、修改`mapping`以及分词、索引升级

`debug`查询或者聚合语句

索引自动修复

`_size`: 整个`_source`字段的字节数大小,需要单独安装一个`mapper-size`插件才能展示。

## 路由元数据

`_routing`: 一个`doc`可以被路由到指定的`shard`上。

## 其他

`_meta`: 一般用来存储应用相关的元信息。

---

例如：

```
put /open-soft/_mapping
{
  "_meta": {
    "class": "cn.enjoyedu.User",
    "version": {"min": "1.0", "max": "1.3"}
  }
}
```

## 管理 Elasticsearch 索引和文档

在 es 中，索引和文档是 REST 接口操作的最基本资源，所以对索引和文档的管理也是我们必须要知道的。索引一般是以索引名称出现在 REST 请求操作的资源路径上，而文档是以文档 ID 为标识出现在资源路径上。映射类型 \_doc 也可以认为是一种资源，但在 es7 中废除了映射类型，所以可以 \_doc 也视为一种接口。

### 索引的管理

在前面的学习中我们已经知道，GET 用来获取资源，PUT 用来更新资源，DELETE 用来删除资源。所以对索引，GET 用来查看索引，PUT 用来创建索引，DELETE 用来删除索引，还有一个 HEAD 请求，用来检验索引是否存在。除此之外，对索引的管理还有

#### 列出所有索引

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

#### 关闭索引和打开

```
POST /open-soft/_close
```

除了删除索引，还可以选择关闭它们。如果关闭了一个索引，就无法通过 Elasticsearch 来读取和写入其中的数据，直到再次打开它。

在现实世界中，最好永久地保存应用日志，以防要查看很久之前的信息。另一方面，在 Elasticsearch 中存放大量数据需要增加资源。对于这种使用案例，关闭旧的索引非常有意义。你可能并不需要那些数据，但是也不想删除它们。

一旦索引被关闭，它在 Elasticsearch 内存中唯一的痕迹是其元数据，如名字以及分片的位置。如果有足够的磁盘空间，而且也不确定是否需要在那个数据中再次搜索，关闭索引要比删除索引更好。关闭它们会让你非常安心，永远可以重新打开被关闭的索引，然后在其中再次搜索。

```
重新打开 POST /open-soft/_open
```

---

## 配置索引

通过 `settings` 参数配置索引，索引的所有配置项都以 “`index`” 开头。索引的管理分为静态设置和动态设置两种。

### 静态设置

只能在索引创建时或在状态为 `closed index`（闭合索引）上设置，主要配置索引主分片、压缩编码、路由等相关信息

`index.number_of_shards` 主分片数，默认为 5. 只能在创建索引时设置，不能修改

`index.shard.check_on_startup` 是否应在索引打开前检查分片是否损坏，当检查到分片损坏将禁止分片被打开。`false`: 默认值; `checksum`: 检查物理损坏; `true`: 检查物理和逻辑损坏，这将消耗大量内存和 CPU; `fix`: 检查物理和逻辑损坏。有损坏的分片将被集群自动删除，这可能导致数据丢失

`index.routing_partition_size` 自定义路由值可以转发的目的分片数。默认为 1，只能在索引创建时设置。此值必须小于 `index.number_of_shards`

`index.codec` 默认使用 LZ4 压缩方式存储数据，也可以设置为 `best_compression`，它使用 DEFLATE 方式以牺牲字段存储性能为代价来获得更高的压缩比例。

如：

```
put test1{
  "settings":{
    "index.number_of_shards":3,
    "index.codec":"best_compression"
  }
}
```

### 动态设置

通过接口 “`_settings`” 进行，同时查询配置也通过这个接口进行，比如：

```
get _settings
get /open-soft/_settings
get /open-soft,test1/_settings
```

配置索引则通过：

```
put test1/_settings
{
  "refresh_interval":"2s"
}
```

常用的配置参数如下：

---

index.number\_of\_replicas 每个主分片的副本数。默认为 1

index.auto\_expand\_replicas 基于可用节点的数量自动分配副本数量,默认为 false (即禁用此功能)

index.refresh\_interval 执行刷新操作的频率。默认为 1s。可以设置为 -1 以禁用刷新。

index.max\_result\_window 用于索引搜索的 from+size 的最大值。默认为 10000

index.blocks.read\_only 设置为 true 使索引和索引元数据为只读, false 为允许写入和元数据更改。

index.blocks.read 设置为 true 可禁用对索引的读取操作

index.blocks.write 设置为 true 可禁用对索引的写入操作

index.blocks.metadata 设置为 true 可禁用索引元数据的读取和写入

index.max\_refresh\_listeners 索引的每个分片上可用的最大刷新侦听器数

index.max\_docvalue\_fields\_search 一次查询最多包含开启 doc\_values 字段的个数, 默认为 100

index.max\_script\_fields 查询中允许的最大 script\_fields 数量。默认为 32。

index.max\_terms\_count 可以在 terms 查询中使用的术语的最大数量。默认为 65536。

index.routing.allocation.enable 控制索引分片分配。All(所有分片)、primaries (主分片)、new\_primaries (新创建分片)、none (不分片)

index.routing.rebalance.enable 索引的分片重新平衡机制。all、primaries、replicas、none

index.gc\_deletes 文档删除后 (删除后版本号) 还可以存活的周期, 默认为 60s

index.max\_regex\_length 用于正在表达式查询(regex query)正在表达式长度, 默认为 1000

## 配置映射

通过 \_mapping 接口进行, 在我们前面的章节中, 已经展示过了。

get /open-soft/\_mapping

或者只看某个字段的属性:

get /open-soft/\_mapping/field/lang

修改映射, 当然就是通过 put 或者 post 方法了。但是要注意, 已经存在的映射只能添加字段或者字段的多类型。但是字段创建后就不能删除, 大多数参数也不能修改, 可以改的是 ignore\_above。所以设计索引时要做好规划, 至少初始时的必要字段要规划好。

---

## 文档的管理

### 增加文档

增加文档，我们在前面的章节已经知道了，比如：

```
put /open-soft/_doc/1
{
  "name": "Apache Hadoop",
  "lang": "Java",
  "corp": "Apache",
  "stars":200
}
```

如果增加文档时，在 Elasticsearch 中如果有相同 ID 的文档存在，则更新此文档，比如执行

```
put /open-soft/_doc/1
{
  "name": "Apache Hadoop2",
  "lang": "Java8",
  "corp": "Apache",
  "stars":300
}
```

则会发现已有文档的内容被更新了。

### 文档的 id

当创建文档的时候，如果不指定 ID，系统会自动创建 ID。自动生成的 ID 是一个不会重复的随机数。使用 GUID 算法，可以保证在分布式环境下，不同节点同一时间创建的\_id 一定是不冲突的。比如：

```
post /open-soft/_doc
{
  "message": "Hello"
}
```

```
post /open-soft/_doc
{
  "message": "Hello"
}
```

```
1 ↴ {
2   "_index" : "open-soft",
3   "_type" : "doc",
4   "_id" : "74NujnIBCUuqSEPfrJTX", // This ID is highlighted with a red box
5   "_version" : 1,
6   "result" : "created",
7   "_shards" : {
8     "total" : 2,
9     "successful" : 1,
10    "failed" : 0
11   },
12   "_seq_no" : 6,
13   "_primary_term" : 1
14 }
```

## 查询文档

```
get /open-soft/_doc/1
```

## 更新文档

前面我们用 `put` 方法更新了已经存在的文档，但是可以看见他是整体更新文档，如果我们要更新文档中的某个字段怎么办？需要使用 `_update` 接口。

```
post /open-soft/_update/1/
{
  "doc": {
    "year": 2016
  }
}
```

如果文档中存在 `year` 字段，更新 `year` 字段的值，如果不存在 `year` 字段，则会新增 `year` 字段，并将值设为 2016。

`update` 接口在文档不存在时提示错误，如果希望在文档不存在时创建文档，则可以在请求中添加 `upsert` 参数或 `doc_as_upsert` 参数，例如：

```
POST /open-soft/_update/5
{
  "doc": {
    "year": "2020"
  },
  "upsert": {
    "name": "Enjoyedu Framework",
    "corp": "enjoyedu"
  }
}
```

或

```
POST /open-soft/_update/6
{
  "doc": {
    "year": "2020"
  },
  "doc_as_upsert": true
}
```

`upsert` 参数定义了创建新文档使用的文档内容，而 `doc_as_upsert` 参数的含义是直接使用 `doc` 参数中的内容作为创建文档时使用的文档内容。

## 删除文档

```
delete /open-soft/_doc/1
```

# 数据检索和分析

为了方便我们学习，我们导入 `kibana` 为我们提供的范例数据。

The screenshot shows the Kibana Observability interface. It has four main sections: APM, Logs, Metrics, and Security. Each section contains a brief description and a 'Add [Section]' button. At the bottom, there are three buttons: 'Add sample data' (with a red arrow pointing to it), 'Upload data from log file', and 'Use Elasticsearch data'. Below these buttons are three cards: 'Sample eCommerce Data', 'Sample flight data', and 'Sample web logs', each with its own description and an 'Add data' button.

The screenshot shows the 'Add data' interface in Kibana. It features tabs for All, Logs, Metrics, SIEM, and Sample data. The Sample data tab is selected. Below it are three cards: 'Sample eCommerce Data', 'Sample flight data', and 'Sample web logs', each with its own description and an 'Add data' button. Red arrows point from the 'Add data' buttons in the first two sections to their respective sample data cards.

---

目前为止，我们已经探索了如何将数据放入 Elasticsearch，现在来讨论下如何将数据从 Elasticsearch 中拿出来，那就是通过搜索。毕竟，如果不能搜索数据，那么将其放入搜索引擎的意义又何在呢？幸运的是，Elasticsearch 提供了丰富的接口来搜索数据，涵盖了 Lucene 所有的搜索功能。因为 Elasticsearch 允许构建搜索请求的格式很灵活，请求的构建有无限的可能性。要了解哪些查询和过滤器的组合适用于你的数据，最佳的方式就是进行实验，因此不要害怕在项目的数据上尝试这些组合，这样才能弄清哪些更适合你的需求。

## \_search 接口

所有的 REST 搜索请求使用 \_search 接口，既可以是 GET 请求，也可以是 POST 请求，也可以通过在搜索 URL 中指定索引来限制范围。

\_search 接口有两种请求方法，一种是基于 URI 的请求方式，另一种是基于请求体的方式，无论哪种，他们执行的语法都是基于 DSL（ES 为我们定义的查询语言，基于 JSON 的查询语言），只是形式上不同。我们会基于请求体的方式来学习。比如说：

```
get kibana_sample_data_flights/_search
{
  "query": {
    "match_all": {}
  }
}
```

或

```
get kibana_sample_data_flights/_search
{
  "query": {
    "match_none": {}
  }
}
```

当然上面的查询没什么太多的用处，因为他们分别代表匹配所有和全不匹配。

所以我们经常要使用各种语法来进行查询，一旦选择了要搜索的索引，就需要配置搜索请求中最为重要的模块。这些模块涉及文档返回的数量，选择最佳的文档返回，以及配置不希望哪些文档出现在结果中等等。

- **query**-这是搜索请求中最重要的组成部分，它配置了基于评分返回的最佳文档，也包括了你不希望返回哪些文档。
- **size**-代表了返回文档的数量。
- **from**-和 **size** 一起使用，**from** 用于分页操作。需要注意的是，为了确定第 2 页的 10 项结果，Elasticsearch 必须要计算前 20 个结果。如果结果集合不断增加，获取某些靠后的翻页将会成为代价高昂的操作。

---

■ `_source` 指定 `_source` 字段如何返回。默认是返回完整的 `_source` 字段。通过配置 `_source`, 将过滤返回的字段。如果索引的文档很大, 而且无须结果中的全部内容, 就使用这个功能。请注意, 如果想使用它, 就不能在索引映射中关闭 `_source` 字段。

■ `sort` 默认的排序是基于文档的得分。如果并不关心得分, 或者期望许多文档的得分相同, 添加额外的 `sort` 将帮助你控制哪些文档被返回。

## 结果起始和页面大小

命名适宜的 `from` 和 `size` 字段, 用于指定结果的开始点, 以及每“页”结果的数量。举个例子, 如果发送的 `from` 值是 7, `size` 值是 5, 那么 Elasticsearch 将返回第 8、9、10、11 和 12 项结果(由于 `from` 参数是从 0 开始, 指定 7 就是从第 8 项结果开始)。如果没有发送这两个参数, Elasticsearch 默认从第一项结果开始(第 0 项结果), 在回复中返回 10 项结果。

例如

```
get kibana_sample_data_flights/_search
{
  "from":100,
  "size":20,
  "query":{
    "term":{
      "DestCountry":"CN"
    }
  }
}
```

但是注意, `from` 与 `size` 的和不能超过 `index.max_result_window` 这个索引配置项设置的值。默认情况下这个配置项的值为 10000, 所以如果要查询 10000 条以后的文档, 就必须要增加这个配置值。例如, 要检索第 10000 条开始的 200 条数据, 这个参数的值必须要大于 10200, 否则将会抛出类似“Result window is too large”的异常。

由此可见, Elasticsearch 在使用 `from` 和 `size` 处理分页问题时会将所有数据全部取出来, 然后再截取用户指定范围的数据返回。所以在查询非常靠后的数据时, 即使使用了 `from` 和 `size` 定义的分页机制依然有内存溢出的可能, 而 `max_result_window` 设置的 10000 条则是对 Elasticsearch 的一种保护机制。

那么 Elasticsearch 为什么要这么设计呢?首先, 在互联网时代的数据检索应该通过相似度算法, 提高检索结果与用户期望的附和度, 而不应该让用户在检索结果中自己挑选满意的数据。以互联网搜索为例, 用户在浏览搜索结果时很少会看到第 3 页以后的内容。假如用户在翻到第 10000 条数据时还没有找到需要的结果, 那么他对这个搜索引擎一定会非常失望。

---

## \_source 参数

元字段 `_source` 中存储了文档的原始数据。如果请求中没有指定 `_source`, Elasticsearch 默认返回整个 `_source`, 或者如果 `_source` 没有存储, 那么就只返回匹配文档的元数据:`_id`、`_type`、`_index` 和 `_score`。

例如:

```
get kibana_sample_data_flights/_search
{
  "query": {
    "match_all": {}
  },
  "_source": ["OriginCountry", "DestCountry"]
}
```

你不仅可以返回字段列表,还可以指定通配符。例如,如果想同时返回"DestCountry"和"DestWeather"字段,可以这样配置`_source: "Dest*"`。也可以使用通配字符串的数组来指定多个通配符,例如`_source:["Origin*", "* Weather"]`。

```
get kibana_sample_data_flights/_search
{
  "query": {
    "match_all": {}
  },
  "_source": ["Origin*", "*Weather"]
}
```

不仅可以指定哪些字段需要返回,还可以指定哪些字段无须返回。比如:

```
get kibana_sample_data_flights/_search
{
  "_source": {
    "includes": ["*.lon", "*.lat"],
    "excludes": "DestLocation.*"
  }
}
```

## 排序

大多搜索最后涉及的元素都是结果的排序(`sort`)。如果没有指定 `sort` 排序选项, Elasticsearch 返回匹配的文档的时候,按照 `_score` 取值的降序来排列,这样最为相关的(得分最高的)文档就会排名在前。为了对字段进行升序或降序排列,

---

指定映射的数组，而不是字段的数组。通过在 `sort` 中指定字段列表或者是字段映射，可以在任意数量的字段上进行排序。

例如：

```
get kibana_sample_data_flights/_search
{
  "from":100,
  "size":20,
  "query":{
    "match_all":{}
  },
  "_source":["Origin*","*Weather"],
  "sort":[{"DistanceKilometers":"asc"}, {"FlightNum":"desc"}]
}
```

## 检索

目前为止所进行的几乎所有搜索请求虽然有些条件限制，但是限制条件主要是在规定 ES 返回的查询结果中哪些字段返回，哪些字段不返回，对于哪些文档返回，哪些文档不返回，其实没有多少约束。真正如何更精确的找到我们需要的文档呢？这就需要我们需要使用带条件的搜索，主要包括两类，基于词项的搜索和基于全文的搜索。

### 基于词项的搜索

#### *term 查询*

对词项做精确匹配，数值、日期等等，如：

```
get kibana_sample_data_flights/_search
{
  "query":{
    "term":{
      "dayOfWeek":3
    }
  }
}
```

对于字符串而言，字符串的精确匹配是指字符的大小写，字符的数量和位置都是相同的，词条（`term`）查询使用字符的完全匹配方式进行文本搜索，词条查

询不会分析 (analyze) 查询字符串，给定的字段必须完全匹配词条查询中指定的字符串。比如：

```
get kibana_sample_data_flights/_search
{
  "query": {
    "term": {
      "OriginCityName": "Frankfurt am Main"
    }
  }
}
```

The screenshot shows a terminal window with two panes. The left pane displays the Elasticsearch query:

```
1 get kibana_sample_data_flights/_search
2 {
3   "query": {
4     "term": {
5       "OriginCityName": "Frankfurt am Main"
6     }
7   }
8 }
```

The right pane shows the JSON response from Elasticsearch:

```
1 {
2   "took" : 0,
3   "timed_out" : false,
4   "_shards" : {
5     "total" : 1,
6     "successful" : 1,
7     "skipped" : 0,
8     "failed" : 0
9   },
10  "hits" : {
11    "total" : {
12      "value" : 146,
13      "relation" : "eq"
14    },
15    "max_score" : 4.490284,
16    "hits" : [
17      {
18        "_index" : "kibana_sample_data_flights",
19        "_type" : "_doc",
20        "_id" : "RoOnjnIBCUuqSEPfd5rg",
21        "_score" : 4.490284,
22        "_source" : {
23          "FlightNum" : "9HV9SWR",
24          "DestCountry" : "AU",
25          "OriginWeather" : "Sunny",
26          "OriginCityName" : "Frankfurt am Main",
27          "AvgTicketPrice" : 841.2656419677076,
28          "DistanceMiles" : 10247.856675613455,
29          "FlightDelay" : false,
30        }
31      }
32    ]
33  }
34}
```

但是如果我们执行

```
get kibana_sample_data_flights/_search
{
  "query": {
    "term": {
      "OriginCityName": "Frankfurt"
    }
  }
}
```

结果却是

```

get kibana_sample_data_flights/_search
{
  "query": {
    "term": {
      "OriginCityName": "Frankfurt"
    }
  }
}

```

```

1 ↴ {
2   "took" : 0,
3   "timed_out" : false,
4 ↴   "_shards" : {
5     "total" : 1,
6     "successful" : 1,
7     "skipped" : 0,
8     "failed" : 0
9   },
10  "hits" : {
11    "total" : {
12      "value" : 0,
13      "relation" : "eq"
14    },
15    "max_score" : null,
16    "hits" : [ ]
17  }
18 }

```

因此可以把 term 查询理解为 SQL 语句中 where 条件的等于号。

### terms 查询

可以把 terms 查询理解为 SQL 语句中 where 条件的 in 操作符:

```

get kibana_sample_data_flights/_search
{
  "query": {
    "terms": {
      "OriginCityName": ["Frankfurt am Main", "Cape Town"]
    }
  }
}

```

Elasticsearch 在 terms 查询中还支持跨索引查询，这类似于关系型数据库中的一对多或多对多关系。比如，用户与文章之间就是一对多关系，可以在用户索引中存储文章编号的数组以建立这种对应关系，而将文章的实际内容保存在文章索引中(当然也可以在文章中保存用户 ID)。如果想将 ID 为 1 的用户发表的所有文章都找出来，在文章索引中查询时为

```

POST /articles/_search
{
  "query": {
    "terms": {
      "_id": {
        "index": "users",
        "id": 1,
        "path": "articles"
      }
    }
  }
}

```

```
    }
}
}
```

在上面的例子中，`terms` 要匹配的字段是 `id`，但匹配值则来自于另一个索引。这里用到了 `index`、`id` 和 `path` 三个参数，它们分别代表要引用的索引、文档 ID 和字段路径。在上面的例子中，先会到 `users` 索引中去找 `id` 为 1 的文档，然后取出 `articles` 字段的值与 `articles` 索引里的 `_id` 做对比，这样就将用户 1 的所有文章都取出来了。

## range 查询和exists 查询

`range` 查询和过滤器的含义是不言而喻的，它们查询介于一定范围之内的值，适用于数字、日期甚至是字符串。

为了使用范围查询，需要指定某个字段的上界和下界值。例如：

```
get kibana_sample_data_flights/_search
{
  "query": {
    "range": {
      "FlightDelayMin": {
        "gte": 100,
        "lte": 200
      }
    }
  }
}
```

可以查询出延误时间在 100~200 之间的航班。其中：

`gte`: 大于等于 (greater than and equal)

`gt`: 大于 (greater than)

`lte`: 小于等于 (less than and equal)

`lt`: 小于 (less than)

`boost`: 相关性评分 (后面的章节会讲到相关性评分)

`exists` 查询检索字段值不为空的的文档，无论其值是多少，在查询中通过 `field` 字段设置检查非空的字段名称，只能有一个。

## prefix 查询

`prefix` 查询允许你根据给定的前缀来搜索词条，这里前缀在同样搜索之前是没有经过分析的。例如：

```
get kibana_sample_data_flights/_search
```

```
{  
  "query":{  
    "prefix":{  
      "DestCountry":"C"  
    }  
  }  
}
```

找到航班目的国家中所有以 C 开头的文档。

### wildcard 查询和 regexp 查询

wildcard 查询就是通配符查询。

使用字符串可以让 Elasticsearch 使用\*通配符替代任何数量的字符(也可以不含)或者是使用?通配符替代单个字符。

例如，有 5 个单词：“bacon” “barn” “ban” 和 “baboon” “bam”，

“ba\*n”的查询会匹配“bacon” “barn” “ban” 和 “baboon”,这是因为\*号可以匹配任何字符序列，而查询“ba?n” 只会匹配“barn”，因为?任何时候都需要匹配一个单独字符。

也可以混合使用多个\*和?字符来匹配更为复杂的通配模板，比如 f\*f?x 就可以匹配 firefox。

```
get kibana_sample_data_flights/_search  
{  
  "query":{  
    "wildcard":{  
      "Dest": "*Marco*"  
    }  
  }  
}
```

}使用这种查询时，需要注意的是 wildcard 查询不像 match 等其他查询那样轻量级。查询词条中越早出现通配符(\*或者?)，Elasticsearch 就需要做更多的工作来进行匹配。例如，对于查询词条“h\*”，Elasticsearch 必须匹配所有以“h”开头的词条。如果词条是“hi\*”，Elasticsearch 只需搜索所有“hi”开头的词条，这是“h”开头的词条集合的子集，规模更小。考虑到额外开支和性能问题，在实际生产环境中使用 wildcard 查询之前，需要先考虑清楚，并且尽量不要让通配符出现在查询条件的第一位。

当然 Elasticsearch 也支持正则 regexp 查询，比如

```
get kibana_sample_data_flights/_search  
{  
  "query":{
```

```
"regexp":{  
    "字段名":"正则表达式"  
}  
}  
}
```

## 文本分析

词条 (term) 查询和全文 (fulltext) 查询最大的不同之处是：全文查询首先分析 (Analyze) 查询字符串，使用默认的分析器分解成一系列的分词，term1, term2, termN，然后从索引中搜索是否有文档包含这些分词中的一个或多个。

所以，在基于全文的检索里，ElasticSearch 引擎会先分析 (analyze) 查询字符串，将其拆分成小写的分词，只要已分析的字段中包含词条的任意一个，或全部包含，就匹配查询条件，返回该文档；如果不包含任意一个分词，表示没有任何文档匹配查询条件。

这里就牵涉到了 ES 里很重要的概念，文本分析，当然对应非 **text** 类型字段来说，本身不存在文本数据词项提取的问题，所以没有文本分析的问题。

### 什么分析

分析( analysis )是在文档被发送并加入倒排索引之前，Elasticsearch 在其主体上进行的操作。在文档被加入索引之前，Elasticsearch 让每个被分析字段经过一系列的处理步骤。

- 字符过滤--使用字符过滤器转变字符。
- 文本切分为分词---将文本切分为单个或多个分词。
- 分词过滤---使用分词过滤器转变每个分词。
- 分词索引--将这些分词存储到索引中。

比如有段话 “I like ELK, it include Elasticsearch&Logstash&Kibana” ，分析以后的分词为 : i like elk it include elasticsearch logstash kibana

### 字符过滤

Elasticsearch 首先运行字符过滤器 (char filter) 。这些过滤器将特定的字符序列转变为其他的字符序列。这个可以用于将 HTML 从文本中剥离，或者是将任意数量的字符转化为其他字符(也许是将“ I love u 2 ”这种缩写的短消息纠正为“ I love you too ”)。

在 “I like ELK.....” 的例子里使用特定的过滤器将 “&” 替换为 “and” 。

### 切分为分词

在应用了字符过滤器之后，文本需要被分割为可以操作的片段。底层的 Lucene 是不会对大块的字符串数据进行操作。相反，它处理的是被称为分词 ( token)的数据。

---

分词是从文本片段生成的，可能会产生任意数量(甚至是 0)的分词。例如，在英文中一个通用的分词器是标准分词器，它根据空格、换行和破折号等其他字符，将文本分割为分词。在我们的例子里，这种行为表现为将字符串 “I like ELK, it include Elasticsearch&Logstash&Kibana” 分解为分词 I like ELK it include Elasticsearch and Logstash Kibana。

## 分词过滤器

一旦文本块被转换为分词，Elasticsearch 将会对每个分词运用分词过滤器 (token filter)。这些分词过滤器可以将一个分词作为输入，然后根据需要进行修改，添加或者是删除。最为有用的和常用的分词过滤器是小写分词过滤器，它将输入的分词变为小写，确保在搜索词条“nosql”的时候，可以发现关于“NoSq”的聚会。分词可以经过多于 1 个的分词过滤器，每个过滤器对分词进行不同的操作，将数据塑造为最佳的形式，便于之后的索引。

在上面的例子，有 3 种分词过滤器：第 1 个将分词转为小写，第 2 个删除停用词（停止词）“and”，第三个将词条“tools”作为“technologies”的同义词进行添加。

## 分词索引

当分词经历了零个或者多个分词过滤器，它们将被发送到 Lucene 进行文档的索引。这些分词组成了第 1 章所讨论的倒排索引。

## 分析器

所有这些不同的部分，组成了一个分析器(analyzer)，它可以定义为零个或多个字符过滤器、1 个分词器、零个或多个分词过滤器。Elasticsearch 中提供了很多预定义的分析器。我们可以直接使用它们而无须构建自己的分析器。

## 配置分析器

### \_analyze 接口

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

可以使用 \_analyze API 来测试 analyzer 如何解析我们的字符串的，在我们下面的学习中，我们会经常用到这个接口来测试。

### 分词综述

因为文本分词会发生在两个地方：创建索引：当索引文档字符类型为 text 时，在建立索引时将会对该字段进行分词；搜索：当对一个 text 类型的字段进行全文检索时，会对用户输入的文本进行分词。

所以这两个地方都可以对分词进行配置。

---

## 创建索引时

ES 将按照下面顺序来确定使用哪个分词器：

1、先判断字段是否有设置分词器，如果有，则使用字段属性上的分词器设置；

2、如果设置了 `analysis.analyzer.default`，则使用该设置的分词器；

3、如果上面两个都未设置，则使用默认的 `standard` 分词器。

## 设置索引默认分词器

`PUT test`

```
{  
  "settings": {  
    "analysis": {  
      "analyzer": {  
        "default": {  
          "type": "simple"  
        }  
      }  
    }  
  }  
}
```

还可以为索引配置内置分词器，并修改内置的部分选项修改它的行为：

`put test`

```
{  
  "settings": {  
    "analysis": {  
      "analyzer": {  
        "my_analyzer": {  
          "type": "standard",  
          "stopwords": ["the", "a", "an", "this", "is"]  
        }  
      }  
    }  
  }  
}
```

## 如何为字段指定内置分词器

`put test`

```
{
```

---

```
"mappings":{  
    "properties": {  
        "title":{  
            "type":"text",  
            "analyzer":"standard",  
            "search_analyzer":"simple"  
        }  
    }  
}
```

甚至还可以自定义分词器。

我们综合来看看分词的设置，并且通过\_analyzer 接口来测试分词的效果：

```
PUT /my_index  
{  
    "settings": {  
        "analysis": {  
            "analyzer": {  
                "std_english": {  
                    "type": "standard",  
                    "stopwords": "_english_"  
                }  
            }  
        }  
    },  
    "mappings": {  
        "properties": {  
            "my_text": {  
                "type": "text",  
                "analyzer": "standard",  
                "fields": {  
                    "english": {  
                        "type": "text",  
                        "analyzer": "std_english"  
                    }  
                }  
            }  
        }  
    }  
}
```

```
        }
    }
}
}
```

我们首先，在索引 my\_index 中配置了一个分析器 std\_english，std\_english 中使用了内置分析器 standard，并将 standard 的停止词模式改为英语模式 \_english\_（缺省是没有的），对字段 my\_text 配置为多数据类型，分别使用了两种分析器，standard 和 std\_english。

```
POST /my_index/_analyze
{
  "field": "my_text",
  "text": "The old brown cow"
}
```

```
POST /my_index/_analyze
{
  "field": "my_text.english",
  "text": "The old brown cow"
}
```



```
POST /my_index/_analyze?pretty
{
  "field": "my_text",
  "text": "The old brown cow"
}

1 ↴ [
2 ↴   "tokens" : [
3 ↴     {
4 ↴       "token" : "the",
5 ↴       "start_offset" : 0,
6 ↴       "end_offset" : 3,
7 ↴       "type" : "<ALPHANUM>",
8 ↴       "position" : 0
9 ↴     },
10 ↴     {
11 ↴       "token" : "old",
12 ↴       "start_offset" : 4,
13 ↴       "end_offset" : 7,
14 ↴       "type" : "<ALPHANUM>",
15 ↴       "position" : 1
16 ↴     },
17 ↴     {
18 ↴       "token" : "brown",
19 ↴       "start_offset" : 8,
20 ↴       "end_offset" : 13,
21 ↴       "type" : "<ALPHANUM>",
22 ↴       "position" : 2
23 ↴     },
24 ↴     {
25 ↴       "token" : "cow",
26 ↴       "start_offset" : 14,
27 ↴       "end_offset" : 17,
28 ↴       "type" : "<ALPHANUM>",
29 ↴       "position" : 3
30 ↴     }
31 ↴   ]
32 ↴ }
33 ↴ ]
```

```

POST /my_index/_analyze?pretty
{
  "field": "my_text.english",
  "text": "The old brown cow"
}
1  [
2   "tokens" : [
3    {
4      "token" : "old",
5      "start_offset" : 4,
6      "end_offset" : 7,
7      "type" : "<ALPHANUM>",
8      "position" : 1
9    },
10   {
11     "token" : "brown",
12     "start_offset" : 8,
13     "end_offset" : 13,
14     "type" : "<ALPHANUM>",
15     "position" : 2
16   },
17   {
18     "token" : "cow",
19     "start_offset" : 14,
20     "end_offset" : 17,
21     "type" : "<ALPHANUM>",
22     "position" : 3
23   }
24 ]
25 ]

```

通过上述运行我们可以看到，分析器 std\_english 中的 The 被删除，而 standard 中的并没有。这是因为 my\_text.english 配置了单独的停止词。

#### 文档搜索时

文档搜索时使用的分析器有一点复杂，它依次从如下参数中如果查找文档分析器，如果都没有设置则使用 standard 分析器：

- 1、搜索时指定 analyzer 参数
- 2、创建索引时指定字段的 search\_analyzer 属性
- 3、创建索引时字段指定的 analyzer 属性
- 4、创建索引时 setting 里指定的 analysis.analyzer.default\_search
- 5、如果都没有设置则使用 standard 分析器

比如：

#### 搜索时指定 analyzer 查询参数

```

GET my_index/_search
{
  "query": {
    "match": {
      "message": {
        "query": "Quick foxes",
        "analyzer": "stop"
      }
    }
  }
}

```

#### 指定字段的 analyzer 和 search\_analyzer

```
PUT my_index
```

```
{  
  "mappings": {  
    "properties": {  
      "title": {  
        "type": "text",  
        "analyzer": "whitespace",  
        "search_analyzer": "simple"  
      }  
    }  
  }  
}
```

指定索引的默认搜索分词器

PUT my\_index

```
{  
  "settings": {  
    "analysis": {  
      "analyzer": {  
        "default": {  
          "type": "simple"  
        },  
        "default_search": {  
          "type": "whitespace"  
        }  
      }  
    }  
  }  
}
```

## 内置分析器

前面说过，每个被分析字段经过一系列的处理步骤：

字符过滤--使用字符过滤器转变字符。

文本切分为分词---将文本切分为单个或多个分词。

分词过滤---使用分词过滤器转变每个分词。

每个分析器基本上都要包含上面三个步骤至少一个。其中字符过滤器可以为 0 个，也可以为多个，分词器则必须，但是也只能有一个，分词过滤器可以为 0 个，也可以为多个。

---

Elasticsearch 已经为我们内置了很多的字符过滤器、分词器和分词过滤器，以及分析器。不过常用的就是那么几个。

#### 字符过滤器 (Character filters)

字符过滤器种类不多。elasticsearch 只提供了三种字符过滤器：

##### HTML 字符过滤器 (HTML Strip Char Filter)

从文本中去除 HTML 元素。

##### POST \_analyze

```
{  
  "tokenizer": "keyword",  
  "char_filter": ["html_strip"],  
  "text": "<p>I'm so <b>happy</b>!</p>"  
}
```

##### 映射字符过滤器 (Mapping Char Filter)

接收键值的映射，每当遇到与键相同的字符串时，它就用该键关联的值替换它们。

```
PUT pattern_test4  
{  
  "settings": {  
    "analysis": {  
      "analyzer": {  
        "my_analyzer": {  
          "tokenizer": "keyword",  
          "char_filter": ["my_char_filter"]  
        }  
      },  
      "char_filter": {  
        "my_char_filter": {  
          "type": "mapping",  
          "mappings": ["James => 666", "13 号 => 888"]  
        }  
      }  
    }  
  }  
}
```

上例中，我们自定义了一个分析器，其内的分词器使用关键字分词器，字符过滤器则是自定制的，将字符中的 James 替换为 666，13 号替换为 888。

---

```
POST pattern_test4/_analyze
{
    "analyzer": "my_analyzer",
    "text": "James 热爱 13 号，可惜后来 13 号结婚了"
}
```

#### 模式替换过滤器 (*Pattern Replace Char Filter*)

使用正则表达式匹配并替换字符串中的字符。但要小心你写的糟糕的正则表达式。因为这可能导致性能变慢！

比如：

```
POST _analyze
{
    "analyzer": "standard",
    "text": "My credit card is 123-456-789"
}
```

这样分词，会导致 123-456-789 被分为 123 456 789，但是我们希望 123-456-789 是一个整体，可以使用模式替换过滤器，替换掉“-”。

```
PUT pattern_test5
{
    "settings": {
        "analysis": {
            "analyzer": {
                "my_analyzer": {
                    "tokenizer": "standard",
                    "char_filter": [
                        "my_char_filter"
                    ]
                }
            },
            "char_filter": {
                "my_char_filter": {
                    "type": "pattern_replace",
                    "pattern": "(\\d+)-(?=\\d)",
                    "replacement": "$1_"
                }
            }
        }
    }
}
```

```
        }
    }

POST pattern_test5/_analyze
{
    "analyzer": "my_analyzer",
    "text": "My credit card is 123-456-789"
}
```

把数字中间的“-”替换为下划线“\_”，这样的话可以让“123-456-789”作为一个整体，而不至于被分成 123 456 789。

## 分词器 (Tokenizer)

### 1. 标准分词器(standard)

标准分词器( standard tokenizer) 是一个基于语法的分词器，对于大多数欧洲语言来说是不错的。它还处理了 Unicode 文本的切分。它也移除了逗号和句号这样的标点符号。

“I have, potatoes.”切分后的分词分别是“ I” 、“ have” 和“ potatoes”。

### 2. 关键词分词器(keyword)

关键词分词器( keyword tokenizer )是- -种简单的分词器，将整个文本作为单个的分词，提供给分词过滤器。只想应用分词过滤器，而不做任何分词操作时，它可能非常有用。

'Hi, there.' 唯一的分词是 Hi, there。

### 3. 字母分词器(letter)

字母分词器根据非字母的符号,将文本切分成分词。例如，对于句子“Hi,there.”分词是 Hi 和 there,因为逗号、空格和句号都不是字母:

'Hi, there. '分词是 Hi 和 there。

### 4. 小写分词器(lowercase)

小写分词器( lowercase tokenizer)结合了常规的字母分词器和小写分词过滤器(如你所想，它将整个分词转化为小写)的行为。通过 1 个单独的分词器来实现的主要原因是，2 次进行两项操作会获得更好的性能。

'Hi, there.'分词是 hi 和 there。

### 5. 空白分词器 whitespace

空白分词器( whitespace tokenizer )通过空白来分隔不同的分词，空白包括空格、制表符、换行等。请注意，这种分词器不会删除任何标点符号，所以文本“Hi, there.” 的分词。

'Hi, there. '分词是 Hi, 和 there..。

## 6. 模式分词器(pattern)

模式分词器( pattern tokenizer)允许指定一个任意的模式，将文本切分为分词。被指定的模式应该匹配间隔符号。例如，可以创建一个定制分析器，它在出现文本“.-.”的地方将分词断开。

## 7. UAX URL 电子邮件分词器(uax\_url\_email)

在处理英语单词的时候，标准分词器是非常好的选择。但是，当下存在不少以网站地址和电子邮件地址结束的文本。标准分析器可能在你未注意的地方对其进行切分。例如，有一个电子邮件地址的样本 `john.smith@example.com`,用标准分词器分析它，切分后：

`'john.smith@example.com'`

分词是 `john.smith` 和 `example.com`。

它同样将 URL 切分为不同的部分：

`'http://example.com?q=foo'` 分词是 `http`、`example.com`、`q` 和 `foo`。

UAX URL 电子邮件分词器( UAX URL email tokenizer )将电子邮件和 URL 都作为单独的分词进行保留。

## 8. 路径层次分词器(path\_hierarchy)

路径层次分词器( path hierarchy tokenizer )允许以特定的方式索引文件系统的路径，这样在搜索时，共享同样路径的文件将被作为结果返回。例如，假设有一个文件名想要索引，看上去是

这样的(`/usr/local/var/log/elasticsearch.log`)。路径层次分词器将其切分为：

`' /usr/local/var/1og/elasticsearch. log'`

分词是 `/usr`、`/usr/local`、`/usr/local/var`、`/usr/local/var/ log` 和 `/usr/local/var/ log/elasticsearch.log`。

这意味着，一个用户查询时，和上述文件共享同样路径层次(名字也是如此)的文件也会被匹配上。查询 `" /usr/local/var/log/es.log"` 时，它和

`" /usr/local/var/log/elasticsearch.log"` 拥有同样的分词，因此它也会被作为结果返回。

## 分词过滤器 (Token filters)

### 1. 标准分词过滤器 (standard)

不要认为标准分词过滤器( standard token filter )进行了什么复杂的计算，实际上它什么事情也没做。

### 2. 小写分词过滤器 (lowercase)

小写分词过滤器( lowercase token filter)只是做了这件事：将任何经过的分词转换为小写。这应该非常简单也易于理解。

### 3. 长度分词过滤器 (length)

长度分词过滤器(length token filter)将长度超出最短和最长限制范围的单词过滤掉。举个例子，如果将 `min` 设置为 `2`，并将 `max` 设置为 `8`，任何小于 `2` 个字符和任何大于 `8` 个字符的分词将会被移除。

#### 4. 停用词分词过滤器 (stop)

停用词分词过滤器(stop token filter)将停用词从分词流中移除。对于英文而言，这意味着停用词列表中的所有分词都将会被完全移除。用户也可以为这个过滤器指定一个待移除 单词的列表。

什么是停用词？

停用词是指在信息检索中，为节省存储空间和提高搜索效率，在处理自然语言数据（或文本）之前或之后会自动过滤掉某些字或词，这些字或词即被称为 Stop Words（停用词）。

停用词(Stop Words)大致可分为如下两类：

1、使用十分广泛，甚至是过于频繁的一些单词。比如英文的“i”、“is”、“what”，中文的“我”、“就”之类词几乎在每个文档上均会出现，查询这样的词搜索引擎就无法保证能够给出真正相关的搜索结果，难于缩小搜索范围提高搜索结果的准确性，同时还会降低搜索的效率。因此，在真正的工作中，Google 和百度等搜索引擎会忽略掉特定的常用词，在搜索的时候，如果我们使用了太多的停用词，也同样有可能无法得到非常精确的结果，甚至是可能大量毫不相关的搜索结果。

2、文本中出现频率很高，但实际意义又不大的词。这一类主要包括了语气助词、副词、介词、连词等，通常自身并无明确意义，只有将其放入一个完整的句子中才有一定作用的词语。如常见的“的”、“在”、“和”、“接着”之类。

下面是英文的默认停用词列表：

a, an, and, are, as, at, be, but, by, for, if, in, into, is, it, no, not, of, on, or; such, that, the, their;, then, there, these, they, this, to, was, will, with

系统内置的停止词如下：

种语言中常见的停止词。这些内置的停止词如下：

\_arabic\_, -armenian\_, \_basque , \_bengali\_1, \_brazilian , \_bulgarian\_-,  
catalan\_, \_czech\_-, \_danish\_-, \_dutch\_ , \_english\_ , \_finnish\_ , \_french\_-, \_galician\_-,  
german\_-, \_greek\_-, \_hungarian\_-, \_indonesian\_ , \_irish\_-, \_italian\_-,  
\_latvian\_-, \_norwegian\_-, \_persian\_ , \_portuguese\_-, \_romanian\_-, \_russian\_-,  
sorani\_-, \_spanish\_-, \_swedish\_-, \_thai\_ , \_turkish\_-

#### 5. 截断分词过滤器、修剪分词过滤器和限制分词数量过滤器

下面 3 个分词过滤器，通过某种方式限制分词流。

■ 截断分词过滤器(**truncate token filter**)允许你通过定制配置中的 length 参数，截断超过一定长度的分词。默认截断多于 10 个字符的部分。

■ 修剪分词过滤器(**trim token filter**)删除 1 个分词中的所有空白部分。例如，分词" foo "将被转变为分词 foo。

■ 限制分词数量分词过滤器(**limit token count token filter**)限制了某个字段可包含分词的最大数量。例如，如果创建了一个定制的分词数量过滤器，限制是 8，那么分词流中只有前 8 个分词会被索引。这个设置使用 max\_token\_count 参数，默认是 1(只有 1 个分词会被索引)。

---

## 常用内置分析器

### 1. 标准分析器

当没有指定分析器的时候，标准分析器( `standardanalyzer`)是文本的默认分析器。它综合了对大多欧洲语言来说合理的默认模块，它没有字符过滤器，包括标准分词器、小写转换分词过滤器和停用词分词过滤器（默认为`_none_`，也就是不去除停止词）。这里只需要记住，如果不为某个字段指定分析器，那么该字段就会使用标准分析器。可配置的参数如下：

`max_token_length`, 默认值 255, 表示词项最大长度，超过这个长度将按该长度分为多个词项

`stopwords`, 默认值`_none_`, 表示分析器使用的停止词数组，可使用内置停止词列表，比如`_english_`等

`stopwords_path` 停止词文件路径

### 2. 简单分析器

简单分析器( `simple analyzer`)就是那么简单！它只使用了小写转换分词器，这意味着在非字母处进行分词，并将分词自动转变为小写。这个分析器对于亚洲语言来说效果不佳，因为亚洲语言不是根据空白来分词，所以请仅仅针对欧洲语言使用它。

### 3. 空白分析器

空白分析器( `whitespace analyzer` )什么事情都不做，只是根据空白将文本切分为若干分词。

### 4. 停用词分析器

停用词分析器( `stop analyzer` )和简单分析器的行为很相像，只是在分词流中额外地过滤了停用词。

### 5. 关键词分析器

关键词分析器( `keyword analyzer` )将整个字段当作一个单独的分词。

### 6. 模式分析器

模板分析器( `pattern analyzer` )允许你指定一个分词切分的模式。但是，由于可能无论如何都要指定模式，通常更有意义的做法是使用定制分析器，组合现有的模式分词器和所需的分词过滤器。

### 7. 雪球分析器

雪球分析器( `snowball analyzer` )除了使用标准的分词器和分词过滤器(和标准分析器一样)，也使用了小写分词过滤器和停用词过滤器。它还使用了雪球词干器对文本进行词干提取。

## 试试自定义分析器

业务需求如下：

去除所有的 HTML 标签

将 & 替换成 and，使用一个自定义的 mapping 字符过滤器

---

使用 `standard` 分词器分割单词  
使用 `lowercase` 分词过滤器将词转为小写  
用 `stop` 分词过滤器去除一些自定义停用词。

PUT pattern\_custom

```
{  
  "settings": {  
    "analysis": {  
      "analyzer": {  
        "my_analyzer": {  
          "char_filter": [  
            "html_strip",  
            "&_to_and"  
          ],  
          "filter": [  
            "lowercase",  
            "my_stopwords"  
          ],  
          "tokenizer": "standard",  
          "type": "custom"  
        }  
      },  
      "char_filter": {  
        "&_to_and": {  
          "mappings": [  
            "&=>and"  
          ],  
          "type": "mapping"  
        }  
      },  
      "filter": {  
        "my_stopwords": {  
          "stopwords": [  
            "king",  
            "james"  
          ],  
          "type": "stop"  
        }  
      }  
    }  
  }  
}
```

```

        "type": "stop"
    }
}
}
}

```

```

POST pattern_custom/_analyze
{
  "analyzer": "my_analyzer",
  "text": "<br> I & Lison & king & James are handsome<br>"
}

```

```

1 POST pattern_custom/_analyze
2 {
3   "analyzer": "my_analyzer",
4   "text": "<br> I & Lison & king & James are handsome
      <br>"
5 }

1  {
2   "tokens" : [
3     {
4       "token" : "i",
5       "start_offset" : 5,
6       "end_offset" : 6,
7       "type" : "<ALPHANUM>",
8       "position" : 0
9     },
10    {
11      "token" : "and",
12      "start_offset" : 7,
13      "end_offset" : 8,
14      "type" : "<ALPHANUM>",
15      "position" : 1
16    },
17    {
18      "token" : "lison",
19      "start_offset" : 9,
20      "end_offset" : 14,
21      "type" : "<ALPHANUM>",
22      "position" : 2
23    },
24    {
25      "token" : "and",
26      "start_offset" : 15,
27      "end_offset" : 16,
28      "type" : "<ALPHANUM>",
29      "position" : 3
30    },
31    {
32      "token" : "James",
33      "start_offset" : 19,
34      "end_offset" : 24,
35      "type" : "<ALPHANUM>",
36      "position" : 4
37    },
38    {
39      "token" : "handsome",
40      "start_offset" : 22,
41      "end_offset" : 28,
42      "type" : "<ALPHANUM>",
43      "position" : 5
44    }
45  ]
46}

```

## 中文分析器

上面的分析器基本都是针对英文的，对中文的处理不是太好，比如：

```

post _analyze
{
  "analyzer": "standard",
  "text": "中央广播电视台推出《跨越24年的牵挂》，讲述习近平
          总书记宁夏看脱贫的故事"
}

```

---

分析后的结果是：

```
{  
  "tokens" : [  
    {  
      "token" : "中",  
      "start_offset" : 0,  
      "end_offset" : 1,  
      "type" : "<IDEOGRAPHIC>",  
      "position" : 0  
    },  
    {  
      "token" : "央",  
      "start_offset" : 1,  
      "end_offset" : 2,  
      "type" : "<IDEOGRAPHIC>",  
      "position" : 1  
    },  
    {  
      "token" : "广",  
      "start_offset" : 2,  
      "end_offset" : 3,  
      "type" : "<IDEOGRAPHIC>",  
      "position" : 2  
    },  
    {  
      "token" : "播",  
      "start_offset" : 3,  
      "end_offset" : 4,  
      "type" : "<IDEOGRAPHIC>",  
      "position" : 3  
    }  
  ]  
}
```

Standard 分析器把中文语句拆分为一个个的汉字，并不太适合。这时候，就需要中文分析器。

中文分析器有很多，例如 cjk, ik 等等，我们选用比较有名的 ik 作为我们的中文分析器。

### 安装

进入 elasticsearch 目录下的 plugins 目录，并执行

```
./elasticsearch-plugin install
```

<https://github.com/medcl/elasticsearch-analysis-ik/releases/download/v7.7.0/elasticsearch-analysis-ik-7.7.0.zip>

```
[elk@xiangxue bin]$ ./elasticsearch-plugin install https://github.com/medcl/elasticsearch-analysis-ik/releases/download/v7.7.0/elasticsearch-analysis-ik-7.7.0.zip  
future versions of Elasticsearch will require Java 11; your Java version from [/usr/local/java/jdk1.8.0_77/jre] does not meet this requirement  
-> Installing https://github.com/medcl/elasticsearch-analysis-ik/releases/download/v7.7.0/elasticsearch-analysis-ik-7.7.0.zip  
-> Downloading https://github.com/medcl/elasticsearch-analysis-ik/releases/download/v7.7.0/elasticsearch-analysis-ik-7.7.0.zip  
[=====] 100% 起  
@     WARNING: plugin requires additional permissions @  
@     * java.net.SocketPermission * connect,resolve  
See http://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html  
for descriptions of what these permissions allow and the associated risks.  
Continue with installation? [y/N]y  
-> Installed analysis-ik
```

如果询问你“Continue with installation?”，当然继续进行。

安装完成后，必须重启 elasticsearch。

## 使用

IK 分词器有两种分词效果，一种是 `ik_max_word`（最大分词）和 `ik_smart`（最小分词）

`ik_max_word`: 会将文本做最细粒度的拆分，比如会将“中华人民共和国国歌”拆分为“中华人民共和国,中华人民,中华,华人,人民共和国,人民,人,民,共和国,共和,和,国,国,国歌”，会穷尽各种可能的组合；

`ik_smart`: 会做最粗粒度的拆分，比如会将“中华人民共和国国歌”拆分为“中华人民共和国,国歌”。

使用方式和一般的分析器没有什么差别。比如，我们对“中央广播电视台总台推出《跨越 24 年的牵挂》，讲述习近平总书记宁夏看脱贫的故事”进行分词。

```
post _analyze
{
  "analyzer": "ik_max_word",
  "text": "中央广播电视台总台推出《跨越24年的牵挂》，讲述习近平
          总书记宁夏看脱贫的故事"
}
```

```
{
  "tokens": [
    {
      "token": "中央",
      "start_offset": 0,
      "end_offset": 2,
      "type": "CN_WORD",
      "position": 0
    },
    {
      "token": "广播",
      "start_offset": 2,
      "end_offset": 4,
      "type": "CN_WORD",
      "position": 1
    },
    {
      "token": "电视",
      "start_offset": 4,
      "end_offset": 6,
      "type": "CN_WORD",
      "position": 2
    },
    {
      "token": "总台",
      "start_offset": 6,
      "end_offset": 8,
      "type": "CN_WORD",
      "position": 3
    },
    {
      "token": "推出",
      "start_offset": 8,
      "end_offset": 10,
      "type": "CN_WORD",
      "position": 3
    },
    {
      "token": "跨越",
      "start_offset": 11,
      "end_offset": 13,
      "type": "CN_WORD",
      "position": 4
    }
  ]
}
```

```
post _analyze
{
  "analyzer": "ik_smart",
  "text": "中央广播电视台总台推出《跨越24年的牵挂》，讲述习近平
          总书记宁夏看脱贫的故事"
}
```

---

## 基于全文的搜索

了解了文本分析以后，就可以学习基于全文的搜索了，这里就需要用到 `match` 系列查询。

### ***match* 查询**

比如说：

```
"query":{  
  "match":{  
    "elk":"Elasticsearch Logstash Kibana"  
  }  
}
```

查询字符串是“`Elasticsearch Logstash Kibana`”，被分析器分词之后，产生三个小写的单词：`elasticsearch logstash kibana`，然后根据分析的结果构造一个布尔查询，默认情况下，引擎内部执行的查询逻辑是：只要 `elk` 字段值中包含有任意一个关键字 `elasticsearch` 或 `logstash` 或 `kibana`，那么返回该文档，相对于的伪代码是：

```
if(  
  doc.elk.contains(elasticsearch)  
  || doc.elk.contains(logstash)  
  || doc.elk.contains(kibana)  
)  
return doc;
```

匹配查询的行为受到两个参数的控制：

`operator`: 表示单个字段如何匹配查询条件的分词

`minimum_should_match`: 表示字段匹配的数量

通过调整 `operator` 和 `minimum_should_match` 属性值，控制匹配查询的逻辑条件，进而控制引擎返回的结果。默认情况下 `operator` 的值是 `or`，在构造查询时设置分词之间的逻辑运算符，如果设置为 `and`，那么引擎内部执行的查询逻辑是：

```
if(  
  doc.elk.contains(elasticsearch)  
  && doc.elk.contains(logstash)  
  && doc.elk.contains(kibana)  
)  
return doc;
```

对于 `minimum_should_match` 属性值，默认值是 `1`，如果设置其值为 `2`，表示分词必须匹配查询条件的数量为 `2`，这意味着，只要文档的 `elk` 字段包含任意

---

两个关键字，就满足查询条件，但是如果文档中只有 1 个关键字，这个文档就不满足条件。比如：

```
POST /kibana_sample_data_logs/_search
{
  "query": {
    "match": {
      "message": "firefox chrome"
    }
  }
}
```

检索包含 firefox 或 chrome 的文档，如果改为：

```
POST /kibana_sample_data_logs/_search
{
  "query": {
    "match": {
      "message": {
        "query": "firefox chrome",
        "operator": "and"
      }
    }
  }
}
```

则不会有任何文档返回，因为没有文档的 message 字段既包含 firefox 又包含 chrome。同样：

```
POST /kibana_sample_data_logs/_search
{
  "query": {
    "match": {
      "message": {
        "query": "firefox chrome",
        "minimum_should_match": 2
      }
    }
  }
}
```

---

也不会任何文档返回，原因也是一样的，因为没有文档的 message 字段既包含 firefox 又包含 chrome。

## ***multi\_match* 查询**

多个字段上执行匹配相同的查询，叫做“multi\_match”查询。比如：

```
POST /kibana_sample_data_flights/_search
```

```
{  
  "query": {  
    "multi_match": {  
      "query": "AT",  
      "fields": ["DestCountry", "OriginCountry"]  
    }  
  }  
}
```

请求将同时检索文档中 DestCountry 和 OriginCountry 这两个字段，只要有一个字段包含 AT 词项该文档就满足查询条件。

## ***match\_phrase* 查询**

当你希望寻找邻近的单词时，match\_phrase 查询可以帮你达到目的。比如：

假设我们要找到 title 字段包含这么一段文本 “quick brown fox” 的文档，然后我们用

```
GET /my_index/my_type/_search  
{  
  "query": {  
    "match_phrase": {  
      "title": "quick brown fox"  
    }  
  }  
}
```

match\_phrase 查询首先解析查询字符串来产生一个词条列表。然后会搜索所有的词条，但只保留包含了所有搜索词条的文档，并且词条的位置要邻接。

但是对于

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

```
        "match_phrase": {  
            "title": "quick fox"  
        }  
    }  
}
```

这个查询不会匹配我们的任何文档，因为没有文档含有邻接在一起的 quick 和 fox 词条。也就是说，匹配的文档必须满足：

- 1、quick、brown 和 fox 必须全部出现在 title 字段中。
- 2、brown 的位置必须比 quick 的位置大 1。
- 3、fox 的位置必须比 quick 的位置大 2。

如果以上的任何一个条件没有被满足，那么文档就不能被匹配。

精确短语(Exact-phrase)匹配也许太过于严格了。也许我们希望含有"quick brown fox"的文档也能够匹配"quick fox"查询，即使位置并不是完全相等的。

我们可以在短语匹配使用 slop 参数来引入一些灵活性：

```
GET /my_index/my_type/_search  
{  
    "query": {  
        "match_phrase": {  
            "title": {  
                "query": "quick fox",  
                "slop": 1  
            }  
        }  
    }  
}
```

slop 参数缺省为 0，它告诉 match\_phrase 查询词条能够最远相隔多远时仍然将文档视为匹配。相隔多远的意思是，你需要移动一个词条多少次来让查询和文档匹配？比如这样一段文本：hello world, java is very good, spark is also very good.

使用 match\_phrase 搜索 java spark 搜不到

如果我们指定了 slop，那么就允许 java spark 进行移动，来尝试与 doc 进行匹配

```
java      is      very      good      spark      is  
  
java      spark  
java      --> spark  
java                  --> spark
```

---

```
java          --> spark
```

上面展示了，当固定第一个 term 的时候，后面的 teram 经过移动直到匹配上搜索词的经过这个移动的次数就是 slop，实际例子如下：

```
POST /kibana_sample_data_logs/_search
{
  "query": {
    "match_phrase": {
      "message": "firefox 6.0a1"
    }
  }
}
```

### ***match\_phrase\_prefix* 查询**

被称为基于前缀的短语匹配，比如：

```
{
  "match_phrase_prefix": {
    "brand": "johnnie walker bl"
  }
}
```

这种查询的行为与 `match_phrase` 查询一致，不同的是它将查询字符串的最后一个词作为前缀使用，换句话说，可以将之前的例子看成如下这样：

johnnie

跟着 walker

跟着以 bl 开始的词

或者可以干脆理解为：

"johnnie walker bl\*"

与 `match_phrase` 一样，它也可以接受 `slop` 参数（参照 `slop` ）让相对词序位置不那么严格：

```
{
  "match_phrase_prefix": {
    "brand": {
      "query": "walker johnnie bl",
      "slop": 10
    }
  }
}
```

---

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

可以通过设置 `max_expansions` 参数来限制前缀扩展的影响，一个合理的值是是 50，这也是系统默认的值：

```
{  
  "match_phrase_prefix": {  
    "brand": {  
      "query": "johnnie walker bl",  
      "max_expansions": 50  
    }  
  }  
}
```

实际例子：

```
POST /kibana_sample_data_logs/_search  
{  
  "query": {  
    "match_phrase_prefix": {  
      "message": "firefox 6.0"  
    }  
  }  
}
```

## 模糊查询、纠错与提示器

### 编辑距离算法

在 Elasticsearch 基于全文的查询中，除了与短语相关的查询以外，其余查询都包含有一个名为 `fuzziness` 的参数用于支持模糊查询。Elasticsearch 支持的模糊查询与 SQL 语言中模糊查询还不一样，SQL 的模糊查询使用 “% keyword%”的形式，效果是查询字段值中包含 `keyword` 的记录。Elasticsearch 支持的模糊查询比这个要强大得多，它可以根据一个拼写错误的词项匹配正确的结果，例如根据 `firefix` 匹配 `firefox`。在自然语言处理领域，两个词项之间的差异通常称为距离或编辑距离，距离的大小用于说明两个词项之间差异的大小。计算词项

编辑距离的算法有多种，在 Elasticsearch 中主要使用 Levenshtein 和 NGram 两种。其他与此相关的算法也都是在这两种算法基础上进行的改造，基本思想都是一致的。所以理解这两个算法的核心思想是学习这部分内容的关键。

---

## Levenshtein 与 NGram

Levenshtein 算法是前苏联数学家 Vladimir Levenshtein 在 1965 年开发的一套算法，这个算法可以对两个字符串的差异程度做量化。量化结果是一个正整数，反映的是一个字符串变成另一个字符串最少需要多少次的处理。由于 Levenshtein 算法是最为普遍接受的编辑距离算法，所以在很多文献中如果没有特殊说明编辑距离算法就是指 Levenshtein 算法。

在 Levenshtein 算法中定义了三种字符操作，即替换、插入和删除，后来又由其他科学家补充了一个换位操作。在转换过程中，每执行一次操作编辑距离就加 1，编辑距离越大越能说明两个字符串之间的差距大。

比如从 firefix 到 firefox 需要将 “i” 替换成 “o”，所以编辑距离为 1；而从 fax 到 fair 则需要将 “x” 替换为 “i” 并在结尾处插入 “r”，所以编辑距离为 2。显然在编辑距离相同的情况下，单词越长错误与正确就越接近。比如编辑距离同样为 2 的情况下，从 fax 到 fair 与从 elascsearxh 到 elasticsearch，后者 elasteseash 是由拼写错误引起的可能性就更大。所以编辑距离这种量化标准一般还需要与单词长度结合起来考虑，在一些极端情况下编辑距离还应该设置为 0，比如像 at、on 这类长度只有 2 的短单词。

NGram 一般是指 N 个连续的字符，具体的字符个数被定义为 NGram 的 size。size 为 1 的 NGram 称为 Unigram，size 为 2 时称为 Bigram，而 size 为 3 时则称为 Trigram。如果 NGram 处理的单元不是字符而是单词，一般称之为 Shingle。使用 NGram 计算编辑距离的基本思路是让字符串分解为 NGram，然后比较分解后共有 NGram 的数量。假设有 a、b 两个字符串，则 NGram 距离的具体运算公式为

$$ngram(a) + ngram(b) - 2 * ngram(a) \cap ngram(b)$$

式中，ngram(a) 和 ngram(b) 代表 a、b 两个字符串 NGram 的数量；ngram(a) ∩ ngram(b) 则是两者共有 NGram 的数量。

例如按 Bigram 处理 firefix 和 firefox 两个单词，分别为“fi, ir, re, ef, fi, ix”和“fi, ir, re, ef, fi, ox”。那么两个字符串的 Bigram 个数都为 6，而共有 Bigram 为 4，则最终 NGram 距离为  $6+6-2\times4=4$ 。

在应用上，Levenshtein 算法更多地应用于对单个词项的模糊查询上，而 NGram 则应用于多词项匹配中。Elasticsearch 同时应用了两种算法。

## 模糊查询

返回包含与搜索字词相似的字词文档；为了找到相似的术语，fuzzy 查询将在指定的编辑距离内创建一组搜索词的所有可能的变体或扩展。查询然后返回每个扩展的完全匹配。

比如：

```
get kibana_sample_data_logs/_search
{
  "query": {
    "fuzzy": {
      "message": {
        "value": "firefix",
```

```
        "fuzziness": "1"
    }
}
}
}
```

我们想找到文档中 `message` 字段包含 `firefox`, 而查询条件中给出的是 `firefix`, 因为两者的编辑距离为 1, 所以包含 `firefox` 的文档依然可以找到, 但是, 如果使用 `firefit`, 因为编辑距离为 2, 则不会找到任何文档。

相关的参数有:

`value`, 必填项, 希望在 `field` 中找到的术语

`fuzziness`, 选填项, 匹配允许的最大编辑距离; 可以被设置为 “0”, “1”, “2” 或 “auto”。 “auto” 是推荐的选项, 它会根据查询词的长度定义距离。

`max_expansions`, 选填项, 创建的最大变体项, 默认为 50。应该避免使用较大的值, 尤其是当 `prefix_length` 参数值为 0 时, 如果过多, 会影响查找性能。

`prefix_length`, 选填项, 创建扩展时保留不变的开始字符数。默认为 0

`transpositions`, 选填项, 指示编辑是否包括两个相邻字符串的转置(`ab`→`ba`)。默认为 `true`。

### 纠错与提示器

纠错是在用户提交了错误的词项时给出正确词项的提示, 而输入提示则是在用户输入关键字时给出智能提示, 甚至可以将用户未输入完的内容自动补全。大多数互联网搜索引擎都同时支持纠错和提示的功能, 比如在用户提交了错误的搜索关键字时会提示:“ 你是不是想查找.... ”.而在用户输入搜索关键字时还能自动弹出提示框将用户可能要输入的内容全都列出来供用户选择。

Elasticsearch 也同时支持纠错与提示功能, 由于这两个功能从实现的角度来说并没有本质区别, 所以它们都由一种被称为提示器或建议器(`Suggester`)的特殊检索实现。由于输入提示需要在用户输入的同时给出提示词, 所以这种功能要求速度必须快, 否则就失去了提示的意义。在实现上, 输入提示是由单独的提示器完成。而在使用上, 提示器则是通过检索接口 `_search` 的一个参数设置, 例如:

```
POST /kibana_sample_data_logs/_search?filter_path=suggest
{
  "suggest": {
    "msg-suggest": {
      "text": "firefit chrom",
      "term": {
        "field": "message"
      }
    }
  }
}
```

```
}
```

在示例中，`search` 接口的 `suggest` 参数中定义了一个提示 `msg-suggest`，并通过 `text` 参数给出需要提示的内容。另一个参数 `term` 实际上是一种提示器的名称，它会分析 `text` 参数中的字符串并提取词项，再根据 Levenshtein 算法找到满足编辑距离的提示词项。所以在返回结果中会包含一个 `suguggest` 字段，其中列举了依照 `term` 提示器找到的提示词项：

```
{
  "suggest": {
    "msg-suggest": [
      {
        "text": "firefit",
        "offset": 0,
        "length": 7,
        "options": [
          {
            "text": "firefox",
            "score": 0.71428573,
            "freq": 5362
          }
        ]
      },
      {
        "text": "chrom",
        "offset": 8,
        "length": 5,
        "options": [
          {
            "text": "chrome",
            "score": 0.8,
            "freq": 4702
          }
        ]
      }
    ]
  }
}
```

Elasticsearch 提供了三种提示器，它们在本质上都是基于编辑距离算法。下面就来看看这些提示器如何使用。

#### `term` 提示器

在示例中使用的提示器就是 `term` 提示器，这种提示器默认使用的算法是称为 `internal` 的编辑距离算法。`internal` 算法本质上就是 Levenshtein 算法，但根据 Elasticsearch 索引特征做了一些优化而效率更高，可以通过 `string_distance` 参数更改算法。

`term` 提示器使用的编辑距离可通过 `max_edits` 参数设置，默认值为 2。

#### `phrase` 提示器

`terms` 会将需要提示的文本拆分成词项，然后对每一个词项做单独的提示，而 `phrase` 提示器则会使用整个文本内容做提示。所以在 `phrase` 提示器的返回结果中，不会看到一个词项一个词项的提示，而是针对整个短语的提示。但从使用的角度来看几乎是一样的，例如：

```
POST /kibana_sample_data_logs/_search
```

```
{
```

```
  "suggest":{
```

```
"msg-suggest":{

    "text": "firefix with chrime",

    "phrase":{

        "field": "message"

    }

}

}

}

}

"suggest" : {
    "msg-suggest" : [
        {
            "text" : "firefix with chrime",
            "offset" : 0,
            "length" : 19,
            "options" : [
                {
                    "text" : "firefox with chrome",
                    "score" : 7.092034E-5
                },
                {
                    "text" : "firefox white chrime",
                    "score" : 2.326709E-6
                },
                {
                    "text" : "firefox wiki chrime",
                    "score" : 2.326709E-6
                },
                {
                    "text" : "firefix white chrome",
                    "score" : 2.170999E-6
                },
                {
                    "text" : "firefix wiki chrome",
                    "score" : 2.170999E-6
                }
            ]
        }
    ]
}
```

但不要被 `phrase` 提示器返回结果欺骗，这个提示器在执行时也会对需要提示的文本内容做词项分析，然后再通过 `NGram` 算法计算整个短语的编辑距离。所以本质上来说，`phrase` 提示是基于 `term` 提示器的提示器，同时使用了 `Levenshtein` 和 `NGram` 算法。

*completion* 提示器

**completion** 提示器一般应用于输入提示和自动补全，也就是在用户输入的同时给出提示或补全未输入内容。这就要求 **completion** 提示器必须在用户输入结束前快速地给出提示，所以这个提示器在性能上做了优化以达到快速检索的目的。

首先要求提示词产生的字段为 `completion` 类型，这是一种专门为 `completion` 提示器而设计的字段类型，它会在内存中创建特殊的数据结构以满足快速生成提示词的要求。例如在示例中创建了 `articles` 索引，并向其中添加了 1 份文档：

PUT articles

{

"mappings":{

"properties": {

```
        "author":{  
            "type": "keyword"  
        },  
        "content":{  
            "type": "text"  
        },  
        "suggestions":{  
            "type": "completion"  
        }  
    }  
}  
}
```

POST articles/\_doc/

```
{  
    "author":"taylor",  
    "content":"an introduction of elastic stack and elasticsearch",  
    "suggestions":{  
        "input":["elastic stack", "elasticsearch"],  
        "weight":10  
    }  
}
```

POST articles/\_doc/

```
{  
    "author":"taylor",  
    "content":"an introduction of elastic stack and elasticsearch",  
    "suggestions": [  
        {"input":"elasticsearch", "weight":30},  
        {"input":"elastic stack", "weight":1}  
    ]  
}
```

在向 completion 类型的字段添加内容时可以使用两个参数，`input` 参数设置字段实际保存的提示词；而 `weight` 参数则设置了这些提示词的权重，权重越高它在返回的提示词中越靠前。在示例 5-29 中给出了两种设置提示词权重的方式，第一种是将一组提示词的权重设置为统一值，另一种则是分开设置它们的权重值。

---

需要注意的是，completion 类型字段保存的提示词是不会分析词项的，比如示例 5-29 中的“elastic stack”并不会拆分成两个提示词，而是以整体出现在提示词列表中。

completion 提示器专门用于输入提示或补全，它根据用户已经输入的内容提示完整词项，所以在 completion 提示器中没有 text 参数而是使用 prefix 参数。例如：

```
POST articles/_search
{
  "_source": "suggest",
  "suggest": {
    "article_suggestion": {
      "prefix": "ela",
      "completion": {
        "field": "suggestions"
      }
    }
  }
}
```

总结一下，term 和 phrase 提示器主要用于纠错，term 提示器用于对单个词项的纠错而 phrase 提示器则主要针对短语做纠错。completion 提示器是专门用于输入提示和自动补全的提示器，在使用上依赖前缀产生提示并且速度更快。

## 相关性检索和组合查询

在全文检索中，检索结果与查询条件的相关性是一个极为重要的问题，优秀的全文检索引擎应该将那些与查询条件相关性高的文档排在最前面。想象一下。如果满足查询条件的文档成千上万，让用户在这些文档中再找出自己最满意的那一条，这无异于再做一次人工检索。用户一般很少会有耐心在检索结果中翻到第 3 页，所以处理好检索结果的相关性对于检索引擎来说至关重要。Google 公司就是因为发明了 Page Rank 算法，巧妙地解决了网页检索结果的相关性问题，才在众多搜索公司中迅速崛起。

相关性问题有两方面问题要解决，一是如何评价单个查询条件的相关性，二是如何将多个查询条件的相关性组合起来。

### 相关性评分

全文检索与数据库查询的一个显著区别，就是它并不一定会根据查询条件做完全精确的匹配。除了模糊查询以外，全文检索还会根据查询条件给文档的相关性打分并排序，将那些与查询条件相关性高的文档排在最前面。相关性(Relevance)或相似性(Similarity)是指两个事物间相互关联的程度，在检索领域特

---

指检索请求与检索结果之间的相关程度。在 Elasticsearch 返回的每条结果中都会包含一个 `_score` 字段，这个字段的值就是当前文档匹配检索请求的相关性评分，我们也可以称为相关度。

解决相关性问题的核心是计算相关度的算法和模型，相关度算法和模型是全文检索引擎最重要的技术之一。相关度算法和相关度模型并非完全相同的概念，相关度模型可以认为是具有相同理论基础的算法集合。所以在实际应用时都是指定到具体的相关度算法，相关度模型则是从理论层面对相关度算法的归类。

## 相关度模型

Elasticsearch 支持多种相关度算法，它们通过类型名称来标识，包括 `boolean`、`BM25`、`DFR` 等等很多。这些算法分别归属于几种不同的理论模型，它们是布尔模型、向量空间模型、概率模型、语言模型等。

### 布尔模型

布尔模型( Boolean Model)是最简单的相关度模型，最终的相关度只有 1 或 0 两种。如果检索中包含多个查询条件，则查询条件之间的相关度组合方式取决于它们之间的逻辑运算符，即以逻辑运算中的与、或、非组合评分。文档的最终评分为 1 时会被添加到检索结果中，而评分为 0 时则不会出现在检索结果中。这与使用 SQL 语句查询数据库有些类似，完全根据查询条件决定结果，非此即彼。在 Elasticsearch 支持的相关度算法中，`boolean` 算法即采用布尔模型，有些地方也部分地采用了布尔模型。

### 向量空间模型

向量空间模型(Vector Space Model)组合多个相关度时采用的是基于向量的算法。在向量空间模型中，多个查询条件的相关度以向量的形式表示。向量实际上就是包含多个数的一维数组，例如[1, 2, 3, 4, 5, 6]就是一个 6 维向量，其中每个数字都代表一个查询条件的相关度。文档对于 n 个查询条件会形成一个 n 维的向量空间，如果定义 1 个查询条件最佳匹配的 n 维向量，那么与这个向量越接近则相关度越高。从向量的角度来看，就是两个向量之间的夹角越小相关度越高，所以 n 个相关度的组合就转换为向量之间夹角的计算。

简单理解，可以只考虑一个二维向量，也就是查询条件只有两个，这样就可以将两个相关度映射到二维坐标图的 X 轴和 Y 轴上。假设两个查询条件权重相同，那么最佳匹配值就可以设置为[1, 1]。如果某文档匹配了第一个条件，部分地匹配了第二个条件，则该文档的向量值为[1, 0.2]。将这两个向量绘制在二维坐标图中，就得到了它们的夹角。

对于多维向量来说，线性代数提供了余弦相似度算法，专门用于计算两个多维向量的夹角。

注：余弦相似性通过测量两个向量的夹角的余弦值来度量它们之间的相似性。  
0 度角的余弦值是 1，而其他任何角度的余弦值都不大于 1；并且其最小值是 -1。  
从而两个向量之间的角度的余弦值确定两个向量是否大致指向相同的方向。两个向量有相同的指向时，余弦相似度的值为 1；两个向量夹角为 90° 时，余弦相似

---

度的值为 0；两个向量指向完全相反的方向时，余弦相似度的值为 -1。这结果是与向量的长度无关的，仅仅与向量的指向方向相关。余弦相似度通常用于正空间，因此给出的值为 -1 到 1 之间。

注意这上下界对任何维度的向量空间中都适用，而且余弦相似性最常用于高维正空间。例如在信息检索中，每个词项被赋予不同的维度，而一个维度由一个向量表示，其各个维度上的值对应于该词项在文档中出现的频率。余弦相似度因此可以给出两篇文档在其主题方面的相似度。

## 概率模型

概率模型是基于概率论构建的模型，BM25、DFR、DFI 都属于概率模型中的一种实现算法，背后有着非常严谨的概率理论依据。以其中最为流行的 BM25 为例，它背后的概率理论是贝叶斯定理，而这个定理在许多领域中都有广泛的应用。

BM25 法将检索出来的文档(D)分为相关文档(R)和不相关文档(NR)两类，使用  $P(R|D)$  代表文档属于相关文档的概率，而使用  $P(NR|D)$  表示文档属于不相关文档的概率，则当  $P(R|D) > P(NR|D)$  时认为这个文档与用户查询相关。根据贝叶斯公式将  $P(R|D) > P(NR|D)$  转换为对某个比值的计算。在此基础上再进行一些转换，就可以得到不同的相关度算法。

## 语言模型

语言模型最早并不是应用于全文检索领域，而是应用于语音识别、机器翻译、拼写检查等领域。在全文检索中，语言模型为每个文档建立不同的计算模型，用以判断由文档生成某一查询条件的概率是多少，而这个概率的值就可以认为是相关度。可见，语言模型的与其他检索模型正好相反，其他检索模型都是从查询条件查找满足条件的文档，而语言模型则是根据文档推断可能的查询条件。

## TF/IDF

对于一篇几百字几千字的文章，如何生成足以准确表示该文章的特征向量呢？就像论文一样，摘要、关键词毫无疑问就是全篇最核心的内容，因此，我们要设法提取一篇文档的关键词，并对每个关键词计算其对应的特征权值，从而形成特征向量。这里涉及一个非常简单但又相当强大的算法，即 TF-IDF 算法。

TF/IDF 实际上两个影响相关度的因素，即 TF 和 IDF。其中，TF 是词项频率简称词频，指一个词项在当前文档中出现的次数，而 IDF 则是逆向文档频率，指词项在所有文档中出现的次数。

Elasticsearch 提供的几种算法中都或多或少有 TF/IDF 的思想，例如 BM25 算法虽然是通过概率论推导而来，但最终的计算公式与 TF/IDF 在本质上也是一致的。

TF/IDF 算法的核心思想是 TF 越高则相关度越高，而 IDF 越高相关度越低。TF 对相关度的影响比较容易理解，但 IDF 为什么会在词项出现次数多的时候反而相关度低呢？

---

举例来说，如果使用“elasticsearch 全文检索”两个词项做检索，文档中“elasticsearch”出现次数高的文档比“全文检索”出现次数高的文档相关度要高。这是因为“elasticsearch”是专业性比较强的词汇，更加专有，它在其他文档中出现的次数会比较少，也就是 IDF 低，而“全文检索”虽然也是专业性词汇，但它覆盖的面要比“elasticsearch”更广泛，所以它在其他文档中出现的次数会比较高，也就是 IDF 高。换句话说，介绍 Elasticsearch 的文章大概率会提到全文检索。但介绍全文检索的文章则不一定会提到 Elasticsearch，比如一篇介绍 MongoDB 的文章大概率会提到全文检索，但是这样的文章与“elasticsearch 全文检索”的相关度不高。

可见，在使用 TF/IDF 计算评分时必须要用到词项在文档中出现的频率，即词频。默认情况下文档 text 类型字段在编入索引时都会记录词频。

Elasticsearch 中的 classic 算法实际上是使用 Lucene 的实用评分函数( Practical Scoring Function)，这个评分函数结合了布尔模型、TF/IDF 和向量空间模型来共同计算分值。该算法是早期 Elasticsearch 运算相关度的算法，现在已经改为 BM25 了。

## BM25

BM25 是 Best Match25 的简写，由于最早应用于一个名为 Okapi 的系统中，所以很多文献中也称之为 Okapi BM25。BM25 算法被认为是当今最先进的相关度算法之一，Elasticsearch 文档字段的默认相关度算法就是采用 BM25，它属于概率模型，依据贝叶斯公式，经过一系列的严格推导以后，得出了一个关于 IDF 的公式

$$IDF(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5},$$

同时在这个基础上，最终的公式上加入了对 TF、当前文档的长度、词频饱和度、长度归一化等因素的考虑

$$Score(Q, d) = \sum_i^n IDF(q_i) \cdot \frac{f_i \cdot (k_1 + 1)}{f_i + k_1 \cdot (1 - b + b \cdot \frac{dl}{avgdl})}$$

## 词频饱和度

所谓词频饱和度指的是当词频超过一定数量之后，它对相关度的影响将趋于饱和。换句话说，词频 10 次的相关度比词频 1 次的分值要大很多，但 100 次与 10 次之间差距就不会那么明显了。在 BM25 算法中，控制词频饱和度的参数是  $k_1$ ，默认值为 1.2。参数  $k_1$  的值越小词频对相关度的影响就会越快趋于饱和，而值越大词频饱和度变化越慢。

---

举例来说，如果将  $k_1$ ，设置为 1，词频达到 10 时就会趋于饱和；而当  $k_1$  设置为 100 时词频在 100 时才会趋于饱和。一般来说  $k_1$  的取值范围为[1.2, 2.0]

## 长度归一化

一般来说，查询条件中的词项出现在较短的文本中，比出现在较长的文本中对结果的相关性影响更大。

举例来说，如果一篇文章的标题中包含 `elasticsearch`，那么这篇文章是专门介绍 `elasticsearch` 的可能性比只在文章内容中出现 `elasticsearch` 的可能要高很多。但这种比较其实是建立在两个不同的字段上，而在实际检索时往往是针对相同的字段做比较。

比如在两篇文章的标题中都出现了 `elasticsearch`，那么哪一篇文章的相关度更高呢？

`BM25` 针对这种情况对文本长度做了所谓的归一化处理，即考虑当前文档字段的文本长度与所有文档的字段平均长度的比值，而这个比值就是长度归一化因素。

为了控制长度归一化对相关度的影响，在长度归一化中加了一个控制参数  $b$ 。这个值的取值范围为[0.0, 1.0]，取值 0.0 时会禁用归一化，而取值 1.0 则会完全启用归一化，默认值为 0.75。

## 相关度解释

相关度算法可通过 `text` 或 `keyword` 类型字段的 `similarity` 参数修改，也就是说相关度算法不针对整个文档而是针对单个字段，它的默认值是 `BM25`。

当然 Elasticsearch 相关度评分比这里介绍的内容要复杂得多，可以通过在查询时添加 `explain` 参数查看评分解释。例如：

```
POST /kibana_sample_data_logs/_search
{
  "query": {
    "match": {
      "message": "chrome"
    }
  },
  "explain": true
}
```

除此之外，通过 `_explain` 接口也可以实现类似的功能，不同的是 `_explain` 接口查看的是单个文档与检索条件的相关度评分解释。例如：

```
POST /kibana_sample_data_logs/_explain/ni-ao3IBOcvNJ8V9P3JN
{
  "query": {
```

```
"match":{  
    "message": "chrome"  
}  
}  
}
```

我们可以大致了解下 Elasticsearch 是如何进行评分的：



```
:  
"details" : [  
    {  
        "value" : 1.1307781,  
        "description" : "score(freq=1.0), computed as boost * idf * tf from:",  
        "details" : [  
            {  
                "value" : 2.2,  
                "description" : "boost",  
                "details" : [ ]  
            },  
            {  
                "value" : 1.0963058,  
                "description" : "idf, computed as log(1 + (N - n + 0.5) / (n + 0.5))  
from:",  
                "details" : [  
                    {  
                        "value" : 4702,  
                        "description" : "n, number of documents containing term",  
                        "details" : [ ]  
                    },  
                    {  
                        "value" : 14074,  
                        "description" : "N, total number of documents with field",  
                        "details" : [ ]  
                    }  
                ]  
            },  
            {  
                "value" : 0.46883816,  
                "description" : "tf, computed as freq / (freq + k1 * (1 - b + b * dl /  
avdl)) from:"  
            }  
        ]  
    }  
]
```

## 相关度权重

在一些情况下需要将某些字段的相关度权重提升，以增加这些字段对检索结果相关性评分的影响。比如，同时使用对文章标题 `tile` 字段和文章内容 `content` 字段做检索，`tile` 字段化相关性评分中的权重应该比 `content` 字段高一些，这时就可以将 `tile` 字段的相关度评分权重提高。

所以相关度权重提升一般都是在多个查询条件时设置。提升相关度权重有多种办法，下面分别来看一下。

### `boost` 参数

`boost` 参数可以在创建索引时直接设置给字段，也可以在执行检索时动态更改。如果不做更改，`boost` 参数的默认值为 1。但并非所有类型的字段都可以设置 `boost`，在创建索引时设置 `boost` 参数并不是一个好的方法，因为这个参数在

---

索引创建以后就不能再更改而降低了灵活性，所以在 Elasticsearch 版本 5 中就已经被废止。

所以更好的方式是检索时提升查询条件的相关度权重，几乎前面介绍的所有 DSL 查询都支持通过 `boost` 参数设置查询条件的相关度权重。例如：

```
POST /kibana_sample_data_flights/_search
{
  "query": {
    "range": {
      "AvgTicketPrice": {
        "gte": 1000,
        "lte": 1200,
        "boost": 2
      }
    }
  }
}
```

通过 `boost` 参数设置查询条件的相关度权重主要用于多查询条件时调整每个查询条件在相关度计算中的权重。

## 组合查询与相关度组合

相关性问题不仅要解决单个查询条件的相关度计算，还要考虑如何将多个查询条件产生的相关度组合起来。而相关度组合问题主要出现在组合查询中，所以接下来我们就要了解下组合查询及相关度组合问题。

组合查询可以将通过某种逻辑将子查询组合起来，实现对多个字段与多个查询条件的任意组合。组合查询组合的子查询不仅可以是基于词项或基于全文的子查询，也可以是另一个组合查询。

单纯从组合查询的使用上来看，组合查询并不复杂，复杂的是组合多个子查询相关度的逻辑，这也是它们的核心区别之一。

### bool 组合查询

`bool` 组合查询将一组布尔类型子句组合起来，形成个大的布尔条件。通过 SQL 语言查询数据时，如果一条数据不满足 `where` 子句的查询条件，这条记录将不会作为结果返回。

但 Elasticsearch 的 `bool` 组合查询则不同，在它的子句中，一些子句的确会决定文档是否会作为结果返回，而另一些子句则不决定文档是否可以作为结果，但会影响到结果的相关度。

`bool` 组合查询可用的布尔类型子句包括 `must`、`filter`、`should` 和 `must_not` 四种，它们接收参数值的类型为数组。

---

**must** 查询结果中必须要包含的内容，影响相关度

**filter** 查询结果中必须要包含的内容，不会影响相关度

**should** 查询结果非必须包含项，包含了会提高分数，影响相关度

**must\_not** 查询结果中不能包含的内容，不会影响相关度

可见，**filter** 和 **must\_not** 单纯只用于过滤文档，而它们对文档相关度没有任何影响。换句话说，这两种子句对查询结果的排序没有作用。在这四种子句中，**should** 子句的情况有些复杂。首先它的执行结果影响相关度，但在是否过滤结果上则取决于上下文。当 **should** 子句与 **must** 子句或 **filter** 子句同时出现在子句中时，**should** 子句将不会过滤结果。也就是说，在这种情况下，即使 **should** 子句不满足，结果也会返回。例如：

```
POST /kibana_sample_data_logs/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": { "message": "firefox" } }
      ],
      "should": [
        {"term": { "geo. src": "CN" }},
        {"term": { "geo. dest": "CN" }}
      ]
    }
  }
}
```

只有 **message** 字段包含 **firefox** 词项的日志文档才会被返回，而 **geo** 的 **src** 字段和 **dest** 字段是否为 **CN** 只影响相关度，当然相关度越高的肯定排在前面，可以通过下面的例子观察到：

```
POST /kibana_sample_data_logs/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": { "message": "firefox" } }
      ],
      "should": [
        {"term": { "geo.src": "CN" }},
        {"term": { "geo.dest": "CN" }}
      ]
    }
  }
}
```

```
        ],
    },
},
"sort": [
{
    "_score": {
        "order": "asc"
    }
}
]
}
```

但是如果在查询条件中将 must 子句删除，那么 should 子句就至少要满足有一条。should 子句需要满足的个数由 query 的 minimum\_should\_match 参数决定，默认情况下它的值为 1。

布尔查询在计算相关性得分时，采取了匹配越多分值越高的策略。由于 filter 和 must\_not 不参与分值运算，所以文档的最后得分是 must 和 should 子句的相关性分值相加后返回给用户。同时还可以试试：

```
POST /kibana_sample_data_logs/_search
{
    "query": {
        "bool": {
            "must": [
                {"match": { "message": "firefox" } }
            ],
            "should": [
                {"term": { "geo. src": "CN" }},
                {"term": { "geo. dest": "CN" }}
            ],
            "filter": {
                "term": {
                    "extension": "zip"
                }
            }
        }
    }
}
```

---

可以发现，最终返回的结果中又过滤了一次，extension 的值为 zip 的才会返回。

### dis\_max 组合查询

dis\_max 查询也是一种组合查询，只是它在计算相关性度时与 bool 查询不同。dis\_max 查询在计算相关性分值时，会在子查询中取最大相关性分值为最终相关性分值结果，而忽略其他子查询的相关性得分。dis\_max 查询通过 queries 参数接收对象的数组。例如：

```
POST /kibana_sample_data_logs/_search
{
  "query": {
    "dis_max": {
      "queries": [
        { "match": { "message": "firefox"} },
        {"term": {"geo. src": "CN"} },
        {"term": {"geo. dest": "CN"} }
      ]
    }
  }
}
```

在多数情况下，完全不考虑其他字段的相关度可能并不合适，所以可以使用 tie\_breaker 参数设置其他字段参与相关度运算的系数。这个系数会在运算最终相关度时乘以其他字段的相关度，再加上最大得分就得到最终的相关度了。所以一般来说，tie\_breaker 应该小于 1，默认值为 0。例如在示例的返回结果中，即使文档 message 和 geo 字段都满足查询条件它也不一定会排在最前面。可添加 tie\_breaker 参数并设置为 0.7。

```
POST /kibana_sample_data_logs/_search
{
  "query": {
    "dis_max": {
      "queries": [
        { "match": { "message": "firefox"} },
        {"term": {"geo. src": "CN"} },
        {"term": {"geo. dest": "CN"} }
      ],
      "tie_breaker": 0.7
    }
  }
}
```

```
    }
}
```

在添加了 `tie_breaker` 参数后，相关度非最高值字段在参与最终相关度结果时的权重就降低为 0.7。但它们对结果排序会产生影响，完全满足条件的文档将排在结果最前面。

### constant\_score 查询

`constant_score` 查询返回结果中文档的相关度为固定值，这个固定值由 `boost` 参数设置，默认值为 1.0。`constant score` 查询只有两个参数 `filter` 和 `boost`，前者与 `bool` 组合查询中的 `filter` 完全相同，仅用于过滤结果而不影响分值。

```
POST /kibana_sample_data_logs/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "match": {
          "geo.src": "CN"
        }
      },
      "boost": 1.3
    }
  }
}
```

由于示例中通过 `boost` 参数设置了相关度，所以满足查询条件文档的 `score` 值将都是 1.3，`match_all` 查询也可以当成是一种特殊类型的 `constant_score` 查询，它会返回索引中所有文档，而每个文档的相关度都是 1.0。它的作用和 `boost` 参数类似，组合查询时调整子查询在相关度计算中的权重。

### boosting 查询

`boosting` 查询通过 `positive` 子句设置满足条件的文档，这类似于 `bool` 查询中的 `must` 子句，只有满足 `positive` 条件的文档才会被返回。`boosting` 查询通过 `negative` 子句设置需要排除文档的条件，这类似于 `bool` 查询中的 `must_not` 子句。但与 `bool` 查询不同的是，`boosting` 查询不会将满足 `negative` 条件的文档从返回结果中排除，而只是会拉低它们的相关性

```
POST /kibana_sample_data_logs/_search
{
  "query": {
    "boosting": {

```

```
        "positive": {"term": { "geo.src": "CN" } },
        "negative": {"term": {"geo.dest": "CN" } },
        "negative_boost": 0.2
    }
},
"sort": [{"_score": "asc"}]
}
```

在示例中，参数 `negative_boost` 设置了一个系数，当满足 `negative` 条件时相关度会乘以这个系数作为最终分值，所以这个值应该小于 1 而大于等于 0。如果 `geo_src` 为 CN 的文档相关度为 1.6，那么 `geo.Dest` 字段也是 CN 的文档相关度就需要再乘以 0.2，所以最终相关度为 0.32。

## function\_score 查询

`function_score` 查询提供了一组计算查询结果相关度的不同函数，通过为查询条件定义不同打分函数实现文档检索相关性的自定义打分机制。查询条件通过 `function_score` 的 `query` 参数设置，而使用的打分函数则使用 `functions` 参数设置例如：

```
POST /kibana_sample_data_logs/_search
{
  "query": {
    "function_score": {
      "query": {
        "query_string": {
          "fields": [ "message" ],
          "query": "(firefox 6.0a1) OR (chrome 11.0.696.50)"
        }
      },
      "functions": [
        {"weight": 2 },
        {"random_score": {} }
      ],
      "score_mode": "max",
      "boost_mode": "avg"
    }
  }
}
```

---

`function_score` 查询在运算相关度时，首先会通过 `functions` 指定的打分函数算出每份文档的得分。如果指定了多个打分函数，它们打分的结果会根据 `score_mode` 参数定义的模式组合起来。

以示例为例，`functions` 参数定义了两个打分函数，`random_score` 函数会在 0-1 之间产生一个随机数，而 `weight` 函数则会以指定的值为相关性分值。由于 `score_mode` 参数设置的值为 `max`，即从所有评分函数运算结果中取最大值，而 `weight` 值为 2，它将永远大于 `random_score` 产生的值，所以评分函数最终给出的分值也将永远是 2。

`score_mode` 包括以下几个选项 `multiply`、`sum`、`avg`、`first`、`max`、`min`，通过名称很容易判断它们的含义，分别是在所有评分函数的运算结果中取它们的乘积、和、平均值、首个值、最大值和最小值。

打分函数运算的相关性评分会与 `query` 参数中查询条件的相关度组合起来，组合的方式通过 `boost_mode` 参数指定，它的默认值与 `score_mode` 一样都是 `multiply`。`boost_mode` 参数的可选值与 `score_mode` 也基本一致，但没有 `first` 而多了一个 `replace`，代表使用评分函数计算结果代替查询分值。

可见 `function_score` 是一种在运算相关度上非常灵活的组合查询，这种灵活性主要体现在它提供了一组打分函数，以及组合这些打分函数的灵活方式。打分函数包括 `weight`、`random_score`、`field_value_factor` 以及衰减函数等等。在这些函数中下面再来简单介绍一下其他打分函数。

### field\_value\_factor 函数

`field_value_factor` 函数在计算相关度时允许加入某一字段作为干扰因子，比如：

```
POST /kibana_sample_data_flights/_search
{
  "query": {
    "function_score": {
      "query": {
        "bool": {
          "must": [
            {"match": { "OriginCountry": "CN" } },
            {"match": { "DestCountry": "US" } }
          ]
        }
      },
      "field_value_factor": {
        "field": "AvgTicketPrice",
        "factor": 0.001,
        "modifier": "reciprocal",
      }
    }
  }
}
```

```
        "missing":1000
    }
}
}
}
```

这个示例实际上是将相关度按票价由低到高的次序做了权重的提升，票价越低最终的相关度越高。这相当于是找出所有从中国到美国的航班，并按票价由低到高的次序排序。

在示例中，`field_value_factor` 打分函数通过 `field` 参数设置了干扰字段为 `AvgTicketPrice`，而 `factor` 则是为干扰字段设置的调整因子，它会与字段值相乘后再参与接下来的运算。`Missing` 表示如果字段值是丢失的，默认被使用的值。`modifier` 参数就有些复杂了，它代表了干扰字段与调整因子相乘的结果如何参与相关度运算。在示例中给出的是 `reciprocal`，代表取倒数  $1/x$ 。

`modifier` 可用运算方法除了 `reciprocal` 以外还有很多：

`none`:不做其他运算；

`log`:取对数

`log1p`:加 1 后再取对数，目的是为了防止字段值为 0-1 时计算结果为负数

`log2p`:加 2 后再取对数

`ln`:自然对数

`ln1p`: 加 1 后取自然对数

`ln1p`: 加 2 后取自然对数

`square`:取平方

`sqrt`: 取平方根

`Reciprocal`: 代表取倒数。

## 衰减函数

衰减函数是一组通过递减方式计算相关度的函数，它们会从指定的原始点开始对相关度做衰减，离原始点距离越远相关度就越低。衰减函数中的原始点是指某字段的具体值，由于要计算其他文档与该字段值的距离，所以要求衰减函数原始点的字段类型必须是数值、日期或地理坐标中的一种。

举例来说，如果在 2019 年 3 月 25 日前后系统运行出现异常，所以对这个日期前后的数据比较感兴趣，就可以按如下形式发送请求：

```
POST kibana_sample_data_flights/_search
```

```
{
```

```
  "query": {
```

```
    "function_score": {
```

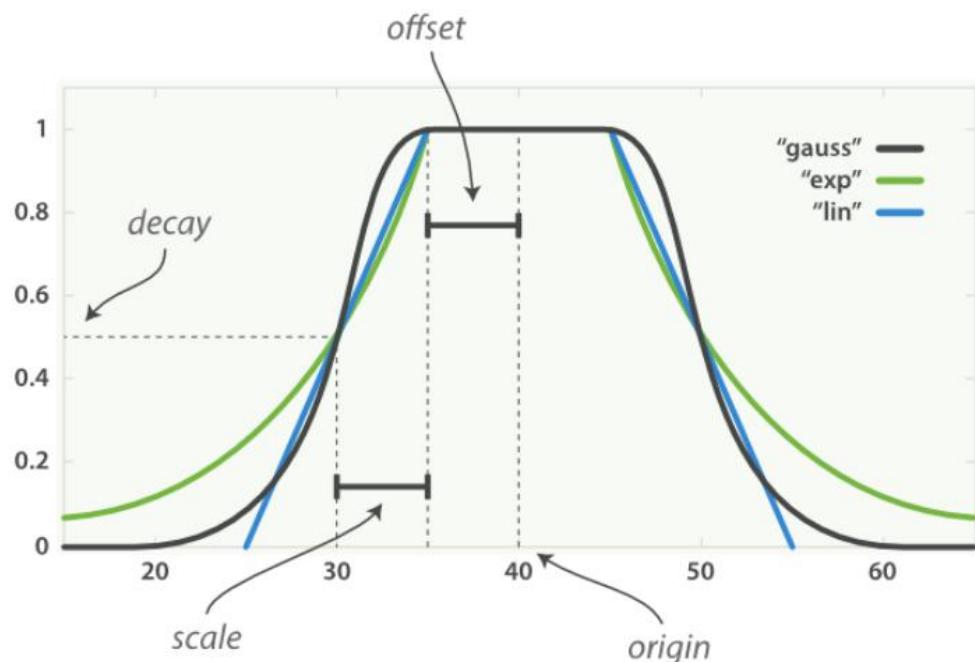
```
      "query": {
```

```
        "match":{
```

```
        "OriginCityName": "Beijing"
    }
},
"gauss": {
    "timestamp": {
        "origin": "2019-03-25",
        "scale": "7d",
        "offset": "1d",
        "decay": 0.3
    }
}
}
}
```

在示例中使用的衰减函数为高斯函数，定义原始点使用的字段为 `timestamp`，而具体的原始点则通过 `origin` 参数定义在了 2019 年 3 月 25 日。`offset` 参数定义了在 1 天的范围内相关度不衰减，也就是说 2019 年 3 月 24-26 日相关度不衰减。`scale` 参数和 `decay` 参数则共同决定了衰减的程度，前者定义了衰减的跨度范围，而后者则定义了衰减多少。以示例中的设置为例，代表的含义是 7 天后的文档相关度衰减至 0.3 倍。

在衰减函数除了高斯函数 `gauss` 以外，还有线性函数 `linear` 和指数函数 `exp` 两种。它们在使用上与高斯函数完全相同。如果将这几种衰减函数以图形画出来就会发现，它们在衰减的平滑度上有着比较明显的区别。



## 单查询条件下的相关度组合

组合查询一般由多个查询条件组成，所以在计算相关度时都要考虑以何种方式组合相关度。而很多的查询都只针对一个字段设置查询条件，所以只有相关度权重提升问题而没有相关度组合问题。但有些查询，比如 `multi_match` 查询可以针对多个字段设置查询条件，所以它们在计算相关度时也需要考虑组合多个相关度的问题。

`multi_match` 查询具有一个 `type` 参数，用于指定针对多字段检索时的执行逻辑及相关度组合方法。`type` 参数有 5 个可选值，即 `best_fields`、`most_fields`、`cross_fields`、`phrase` 和 `phrase_prefix`。例如：

```
POST /kibana_sample_data_flights/_search
{
  "query": {
    "multi_match": {
      "query": "CN",
      "fields": [ "originCountry^2", "DestCountry" ],
      "type": "best_fields"
    }
  }
}
```

### `best_fields`、`phrase` 与 `phrase_prefix` 类型

`best_fields` 类型在执行时会将与字段匹配的文档都检索出来，但在计算相关度时会取得分最高的作为整个查询的相关度。

例如在示例中，第一个查询通过“`OriginCountry^2`”的形式将 `OriginCountry` 字段的相关度权重提升到 2，所以这个字段相关度会高于 `DestCountry` 字段。

在 `best_fields` 类型下执行检索时，`DestCountry` 字段对最终相关度就不会再有影响。通过查看返回结果也可以看到，`OriginCountry` 字段为 `CN` 的文档相关度都相同，即使 `DestCountry` 字段也是 `CN`，文档的相关度也不会提升。

`best_fields` 适用于用户希望匹配条件全部出现在一个字段中的情况，比如在文章标题和文章内容中同时检索 `elasticseareh` 和 `logstash` 时，如果在文章标题或是文章内容中同时出现了两个词项，该文章在相关度就会高于其他文章。比如说：

有两份文档：

```
文档 1: {
  "title": "Quick brown rabbits",
  "body": "Brown rabbits are commonly seen."
}
```

---

```
文档 2: {
    "title": "Keeping pets healthy",
    "body": "My quick brown fox eats rabbits on a regular basis."
}
```

用户输入了"brown fox"来搜索文档，文档 2 匹配的更好一些，因为它的 body 字段同时包含了用户寻找的两个单词。

大家是否还记得前面的 `dis_max` 查询（`dis_max` 查询在计算相关性分值时，会在子查询中取最大相关性分值为最终相关性分值结果，而忽略其他子查询的相关性得分），在相关度计算上 `dis_max` 查询是不是与 `best_fields` 类型很像？事实上，`best_fields` 类型的查询在执行时会转化为 `dis_max` 查询，例如示例在执行时会转化为

```
POST /kibana_sample_data_flights/_search
{
  "query": {
    "dis_max": {
      "queries": [
        {"match": {"OriginCountry": {"query": "CN", "boost": 2}}},
        {"match": {"DestCountry": "CN"}}
      ]
    }
  }
}
```

`dis_max` 有一个参数 `tie_breaker`，可以设置非最高值相关度参与最终相关度运算的系数，`multi_match` 中使用 `best_fields` 类型时也可以使用这个参数。

`phrase` 与 `phrase_prefix` 类型在执行逻辑上与 `best_fields` 完全相同，只是在转换为 `dis_max` 时 `queries` 查询中的子查询会使用 `phrase` 或 `phrase_prefix` 而不是 `match`。

## most\_fields 类型

`most_fields` 类型在计算相关度时会将所有相关度累加起来，然后再除以相关度的个数以用到它们的平均值作为最终的相关度。

还是以 `best_fields` 中的那个示例检索为例，如果将 `type` 替换为 `most_fields`，

```
POST /kibana_sample_data_flights/_search
{
  "query": {
    "multi_match": {
      "query": "CN",
      "type": "most_fields"
    }
  }
}
```

```
        "fields": [ "originCountry^2", "DestCountry"],  
        "type": "most_fields"  
    }  
}  
}
```

它会将 `OriginCountry` 和 `DestCountry` 两个字段匹配 CN 时计算出的相关度累加，然后再用累加和除以 2 作为最终的相关度。所以只有当两个字段都匹配了 CN，最终的相关度才会更高。这在效果上相当于将出发地和目的地都是中国的文档排在了最前面，所以适用于希望检索出多个字段中同时都包含相同词项的检索。在实现上，`most_fields` 类型的查询会被转化为 `bool` 查询的 `should` 子句，上面的示例实际被转化为：

```
POST /kibana_sample_data_flights/_search  
{  
  "query": {  
    "bool": {  
      "should": [  
        {"match": {"OriginCountry": {"query": "CN", "boost": 2}}},  
        {"match": {"DestCountry": "CN"} }  
      ]  
    }  
  }  
}
```

### [cross\\_fields 类型](#)

如果查询条件中设置了多个词项，`best_fields` 类型和 `most_fields` 类型都支持通过 `operator` 参数设置词项之间的逻辑关系，即 `and` 和 `or`。但它们在设置 `operator` 时是针对字段级别的而不是针对词项级别的，来看一个例子：

```
POST /kibana_sample_data_logs/_search  
{  
  "query": {  
    "multi_match": {  
      "query": "firefox success",  
      "fields": [ "message", "tags"],  
      "type": "best_fields",  
      "operator": "and"  
    }  
  }  
}
```

```
}
```

示例设置的查询条件为 `firefox` 和 `success` 两个词项，而匹配字段也是两个 `messape` 和 `tags`。当 `operator` 设置为 `and` 时，在 `best_fields` 类型下这意味着两个字段中需要至少有一个同时包含 `firefox` 和 `success` 两个词项，而这样的日志文档并不存在。

而在 `cross_fields` 类型则可以理解为将文档的中 `message` 和 `tags` 字段组合为一个大的字段，然后在大的字段中搜索词条 `firefox` 和 `success`。这样的话只要 `firefox` 和 `success` 在大字段中均出现过，就认为这个文档符合条件。

## 聚集查询

聚集查询(Aggregation)提供了针对多条文档的统计运算功能，它不是针对文档本身内容的检索，而是要将它们聚合到一起运算某些方面的特征值。

聚集查询与 SQL 语言中的聚集函数非常像，聚集函数在 Elasticsearch 中相当于是聚集查询的一种聚集类型。比如在 SQL 中的 `avg` 函数用于求字段平均值，而在 Elasticsearch 中要实现相同的功能可以使用 `avg` 聚集类型。

聚集查询也是通过`_search` 接口执行，只是在执行聚集查询时使用的参数是 `aggregations` 或 `aggs`。所以`_search` 接口可以执行两种类型的查询，1 种是通过 `query` 参数执行 DSL，另一种则是通过 `aggregations` 执行聚集查询。

这两种查询方式还可以放在一起使用，执行逻辑是先通过 DSL 检索满足查询条件的文档，然后再使用聚集查询对 DSL 检索结果做聚集运算这一规则适用于本章列举的所有聚集查询。聚集查询有着比较规整的请求结构，具体格式如下：

```
"aggregations/aggs" : {  
    "<聚集名称>": {  
        "<聚集类型>": {  
            <聚集体>  
        }  
        .....  
        [ , "aggregations/aggs" : ( [ <子聚集>]+ ) ]  
    }  
    [ "<聚集名称>" : (...) ]*  
}
```

`aggregations` 和 `aggs` 都是`_search` 的参数，其中 `aggs` 是 `aggregations` 的简写。每一个聚集查询都需要定义一个聚集名称，并归属于某种聚集类型。聚集名称是用户自定义的，而聚集类型则是由 Elasticsearch 预先定义好。

聚集名称会在返回结果中标识聚集结果，而聚集类型则决定了聚集将如何运算。比如前面提到的 `avg` 就是一种聚集类型。这里要特别强调的是，聚集中可

---

以再包含子聚集。子聚集位于父聚集的名称中，与聚集类型同级，所以子聚集的运算都是在父聚集的环境中运算。Elasticsearch 对于子聚集的深度没有限制，所以理论上说可以包含无限深度的子聚集。

聚集类型总体上被分为四种大类型，即指标聚集(Metrics Aggregation)、桶型(Bucket Aggregation)、管道聚集(Pipeline Aggregation) 和矩阵聚集(Matrix Aggregation)。

指标聚集是根据文档字段所包含的值运算某些统计特征值，如平均值、总和等，它们的结果一般都包含一个或多个数值，前面提到的 `avg` 聚集就是指标聚集的 1 种。桶型聚集根据一定的分组标准将文档归到不同的组中，这些分组在 Elasticsearch 中被称为桶(Bucket)，桶型聚集与 SQL 中 `group by` 的作用类似，一般会与指标聚集嵌套使用。管道聚集可以理解为聚集结果的再聚集，它一般以另一个聚集结果作为输入，然后在此基础上再做聚集。矩阵聚集是 Elasticsearch 中的新功能，由于是针对多字段做多种运算，所以形成的结果类似于矩阵。

## 指标聚集

指标聚集是根据文档中某一字段做聚集运算，比如计算所有产品销量总和、平均值等等。指标聚集的结果可能是单个值，这种指标聚集称为单值指标聚集；也可能是多个值，称为多值指标聚集。

### 平均值聚集

平均值聚集是根据文档中数值类型字段计算平均值的聚集查询，包括 `avg` 聚集和 `weighted_avg` 聚集两种类型。`avg` 聚集直接取字段值后计算平均值，而 `weighted_avg` 聚集则会在计算平均值时添加不同的权重。

#### `avg` 聚集

`avg` 聚集计算平均值，例如在示例中计算航班的平均延误时间：

```
POST /kibana_sample_data_flights/_search ?filter_path=aggregations
{
  "aggs": {
    "delay_avg": {
      "avg": {"field": "FlightDelayMin"}
    }
  }
}
```

在示例 7-2 中使用了请求参数 `filter_path` 将返回结果的其他字段过滤掉了，否则查询的结果中将包含 `kibana_sample_data_flights` 索引中的文档。加了 `filter_path` 之后返果为：

```
{  
  "aggregations": {  
    "delay_avg": {  
      "value": 47.33517114633586  
    }  
  }  
}
```

在返回结果中，`aggregations` 是关键字，代表这是聚集查询的结果。其中的 `delay_avg` 则是在聚集查询中定义的聚集名称，`value` 是聚集运算的结果。

在示例中运算航班延误时间时会将所有文档都包含进来做计算，如果只想其中一部分文档参与运算则可以使用 `query` 参数以 DSL 的形式定义查询条件。

例如在示例中就是只计算了飞往中国的航班平均延误时间：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations  
{  
  "query": {  
    "match": {  
      "DestCountry": "CN"  
    }  
  },  
  "aggs": {  
    "delay_avg": {  
      "avg": {"field": "FlightDelayMin"}  
    }  
  }  
}
```

在示例中请求 `_search` 接口时，同时使用了 `query` 与 `aggs` 参数。在执行检索时会先通过 `query` 条件过滤文档，然后再在符合条件的文档中运算平均值聚集。

`weighted_avg` 无非就是根据某些条件对进行聚集的数据进行加权运算，和 `avg` 聚集没有本质差别。

## 计数聚集与极值聚集

计数聚集用于统计字段值的数量，而极值聚集则是查找字段的极大值和极小值。

### 计数聚集

`value_count` 聚集和 `cardinality` 聚集可以归入计数聚集中，前者用于统计从字段中取值的总数，而后者则用于统计不重复数值的总数。例如：

---

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "aggs": {
    "country_code": {
      "cardinality": {"field": "DestCountry"}
    },
    "total_country": {
      "value_count": {"field": "DestCountry"}
    }
  }
}
```

在示例中，`cardinality` 聚集统计了 `DestCountry` 字段非重复值的数量，类似于 SQL 中的 `distinct`。`value_count` 聚集则统计了 `DestCountry` 字段所有返回值的数量，类似于 SQL 中的 `count`。

需要注意的是，`cardinality` 聚集的算法使用极小内存实现统计结果的基本准确。所以 `cardinality` 在数据量极大的情况下是不能保证完全准确的。

## 极值聚集

极值聚集是在文档中提取某一字段最大值或最小值的聚集，包括 `max` 聚集和 `min` 聚集。

例如：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "aggs": {
    "max_price": {
      "max": {"field": "AvgTicketPrice"}
    },
    "min_price": {
      "min": {"field": "AvgTicketPrice"}
    }
  }
}
```

上面例子聚集有 `max_price` 和 `min_price` 两个，它们分别计算了机票价格的最大的最小值。

## 统计聚集

统计聚集是一个多值指标聚集，也就是返回结果中会包含多个值，都是一些与统计相生的数据。统计聚集包含 `stats` 聚集和 `extended_stats` 聚集两种，前者返回的统计数据是一些比较基本的数值，而后者则包含一些比较专业的统计数值。

### `stats` 聚集

`stats` 聚集返回结果中包括字段的最小值(`min`)、最大值(`max`)、总和(`sum`)、数量(`coun`) 及平均值(`avg`) 五项内容。例如，在示例 7-9 中对机票价格做统计：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "aggs": {
    "price_stats": {
      "stats": { "field": "AvgTicketPrice" }
    }
  }
}
```

在示例中，`stats` 聚集使用 `field` 参数指定参与统计运算的字段为 `AvgTicketPrice`。

### `extended_stats` 聚集

`extended_stats` 聚集增加了几项统计数据，这包括平方和、方差、标准方差和标准方差偏移量。从使用的角度来看，`extended_stats` 聚集与 `stats` 聚集完全相同，只是聚集类型不同。

## 百分位聚集

百分位聚集根据文档字段值统计字段值按百分比的分布情况，包括 `percentiles` 聚集和 `percentile_ranks` 两种。前者统计的是百分比与值的对应关系，而后者正好相反统计值与百分比的对应关系。

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "aggs": {
    "price_percentile": {
      "percentiles": {"field": "AvgTicketPrice", "percents": [25,50,75,100]}
    },
    "price_percentile_rank": {
      "percentile_ranks": {"field": "AvgTicketPrice", "values": [600,1200]}
    }
}
```

```
    }
}
```

percentiles 聚集通过 `percents` 参数设置组百分比，然后按值由小到大的顺序划分不同区间，每个区间对应一个百分比。percentile\_ranks 聚集则通过 `value` 参数设置组值，然后根据这些值分别计算落在不同值区间的百分比。

以示例返回的结果为例：

```
{
  "aggregations" : {
    "price_percentile_rank" : {
      "values" : {
        "600.0" : 45.39892372745635,
        "1200.0" : 100.0
      }
    },
    "price_percentile" : {
      "values" : {
        "25.0" : 410.01103863927534,
        "50.0" : 640.362667150751,
        "75.0" : 842.2134903196303,
        "100.0" : 1199.72900390625
      }
    }
  }
}
```

在 percentiles 返回结果 `price_percentile` 中，“`"25.0" : 410.0127977258341`”代表的含义是 25% 的机票价格都小于 `410.0127977258341`，其他以此类推。在 percentile\_ranks 返回结果 `price_percentile_rank` 中，`600.0": 45.39892372745635`”代表的含义是 600.0 以下的机票占总机票价格的百分比为 45.39892372745635%。

## 使用范围分桶

如果使用 SQL 语言类比，桶型聚集与 SQL 语句中的 `group by` 子句极为相似。桶型聚集(Bucket Aggregation)是 Elasticsearch 官方对这种聚集的叫法，它起的作用是根据条件对文档进行分组。

可以将这里的桶理解为分组的容器，每个桶都与一个分组标准相关联，满足这个分组标准的文档会落桶中。所以在默认情况下，桶型聚集会根据分组标准返回所有分组，同时还会通过 `doc_count` 字段返回每一桶中的文档数量。

由于单纯使用桶型聚集只返回桶内文档数量，意义并不大，所以多数情况下都是将桶型聚集与指标聚集以父子关系的形式组合在一起使用。桶型聚集作为父聚集起到分组的作用。而指标聚集则以子聚集的形式出现在桶型聚集中，起到分组统计的作用。比如将用户按性别分组，然后统计他们的平均年龄。

---

按返回桶的数量来看，桶型聚集可以分为单桶聚集和多桶聚集。在多桶聚集中，有些桶的数量是固定的。而有些桶的数量则是在运算时动态决定。由于桶聚集基本都是将所有桶一次返回，返回了过多的通会影响性能，所以单个请求允许返回的最大桶数受 `search.max_bucket` 参数限制。

这个参数在 7.0 之前的版本中默认值为 -1，代表无上限。但在 Elasticsearch 版本 7 中，这个参数的默认值已经更改为 10000 所以在做桶型聚集时要先做好数据验证，防止桶数量过多影响性能。

桶型聚集的种类非常多，我们分别来讲解。

## 数值范围

`range`、`date_range` 与 `ip_range` 这三种类型的聚集都用于根据字段的值范围内对文档分桶，字段值在同一范围内的文档归入同一桶中。每个值范围都可通过 `from` 和 `to` 参数指定，范围包含 `from` 值但不包含 `to` 值，用数学方法表示就是  $[from, to)$ 。在设置范围时，可以设置一个也可以设置多个，范围之间并非一定要连续，可以有间隔也可以有重叠。

### `range` 聚集

`range` 聚集使用 `ranges` 参数设置多个数值范围，使用 `field` 参数指定 1 个数值类型的字段。`range` 聚集在执行时会将该字段在不同范围内的文档数量统计出来，并在返回结果的 `doc_count` 字段中展示出来。例如统计航班不同范围内的票价数量，可以按示例的方式发送请求：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "aggs": {
    "price_ranges": {
      "range": {
        "field": "AvgTicketPrice",
        "ranges": [
          {"to": 300},
          {"from": 300, "to": 600},
          {"from": 600, "to": 900},
          {"to": 900}
        ]
      }
    }
  }
}
```

---

在返回结果中，每个范围都会包含一个 `key` 字段，代表了这个范围的标识，它的基本格式是“`<from>-<to>`”。如果觉得返回的这种 `key` 格式不容易理解，可以通过在 `range` 聚集的请求中添加 `keyed` 和 `key` 参数定制返回结果的 `key` 字段值。其中 `keyed` 是 `range` 的参数，用于标识是否使用 `key` 标识范围，所以为布尔类型，`key` 参数则是与 `from`、`to` 参数同级的参数，用于定义返回结果中的 `key` 字段值。

## date\_range 聚集

`date_range` 聚集与 `range` 聚集类似，只见范围和字段的类型为日期而非数值。  
`date_range` 聚集的范围指定也是通过 `ranges` 参数设置，具体的范围也是使用`<from>-<to>`两个参数，并且可以使用 `keyed` 和 `key` 定义返回结果的标识。  
`date_range` 聚集多了个指定日期格式的参数 `format`，可以用于指定 `from` 和 `to` 的日期格式。例如，

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "aggs": {
    "mar_flights": {
      "date_range": {
        "field": "timestamp",
        "ranges": [
          {"from": "2019-03-01", "to": "2019-03-30"}
        ],
        "format": "yyyy-MM-dd"
      }
    }
  }
}
```

## ip\_range 聚集

`ip_range` 聚集根据 `ip` 类型的字段统计落在指定 IP 范围的文档数量，使用的聚集类型名称为 `ip_range`。例如，统计了两个 IP 地址范围的文档数量：

```
POST /kibana_sample_data_logs/_search?filter_path=aggregations
{
  "aggs": {
    "local": {
      "ip_range": {
        "field": "clientip",
        "ranges": [
          {"from": "192.168.1.1", "to": "192.168.1.100"} // 第一个IP范围
        ]
      }
    }
  }
}
```

```
        {"from": "157.4.77.0", "to": "157.4.77.255"},  
        {"from": "105.32.127.0", "to": "105.32.127.255"}  
    ]  
}  
}  
}  
}
```

## 间隔范围

`histogram`、`date_histogram` 与 `auto_date_histogram` 这三种聚集与上一节中使用数值定义范围的聚集很像，也是统计落在某一范围内的文档数量。但与数值范围聚集不同的是，这三类座集统计范围由固定的间隔定义，也就是范围的结束值和起始值的差值是固定的。

### `histogram` 聚集

`histogram` 聚集以数值为间隔定义数值范围，字段值具有相同范围的文档将落入同桶中。例如示例以 100 为间隔做分桶，可以通过返回结果的 `doc_count` 字段获取票价在每个区间的文档数量：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations  
{  
  "aggs": {  
    "price_histo": {  
      "histogram": {  
        "field": "AvgTicketPrice",  
        "interval": 100,  
        "offset": 50,  
        "keyed": false,  
        "order": {  
          "_count": "asc"  
        }  
      }  
    }  
  }  
}
```

其中，`interval` 参数用于指定数值间隔必须为正值，而 `offset` 参数则代表起始数值的偏移量，必须位于 $[0, interval)$  范围内。`order` 参数用于指定排序字段和

---

顺序，可选字段为 `_key` 和 `_count`。当 `keyed` 参数设置为 `true` 时，返回结果中每个桶会有一个标识，标识的计算公式为

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

## **date\_histogram 聚集**

`date_histogra` 聚集以时间为间隔定义日期范围，字段值具有相同日期范围的文档将落入同一桶中。同样，返回结果中也会包含每个间隔范围内的文档数量 `doc_count`。例如统计每月航班数量：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "aggs": {
    "month flights": {
      "date_histogram": {"field": "timestamp", "interval": "month"}
    }
  }
}
```

在示例使用参数 `interval` 指定时间间隔为 `month`, 即按月划分范围。时间可以是还有：

- 毫秒： 1ms 10ms
- 秒： second/1s 10s
- 分钟： minute/1m 10m
- 小时： hour/1h 2h
- 天： day 2d 不支持
- 星期： week/1w 不支持
- 月： month/1M 不支持
- 季度： quarter/1q 不支持
- 年： year/1y 不支持

不过这个参数将要过期，替代的是 `fixed_interval` 和 `calendar_interval`，可以参考这个页面，有详细说明：

<https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket-datehistogram-aggregation.html>

简单来说，`calendar_interval` 支持 1 年、1 季度、1 月、1 周、1 天、1 小时、1 分钟，不支持时间为复数，`fixed_interval` 支持的时间单位为天、小时、分、秒、毫秒，允许复数，例如“`fixed_interval`”：“`30d`”，表示为 30 天。

## **auto\_date\_histogram 聚集**

前述两种聚集都是指定间隔的具体值是多少，然后再根据间隔值返回每一桶中满足条件的文档数。最终会有多少桶取决于两个条件，即间隔值和字段值在

---

所有文档中的实际跨度。反过来，如果预先指定需要返回多少个桶，那么间隔值也可以通过桶的数量以及字段值跨度共同确定。`auto_date_histogram` 聚集就是这样一种聚集，它不是指定时间间隔值，而是指定需要返回桶的数量。例如在示例中定义需要返回 10 个时间段的桶：

```
POST /kibana_sample_data_flights/_search?size=0
{
  "aggs": {
    "age_group": {
      "auto_date_histogram": {
        "field": "timestamp",
        "buckets": 10
      }
    }
  }
}
```

参数 `field` 设置通过哪一个字段做时间分隔，而参数 `buckets` 则指明了需要返回多少个桶。默认情况下，`buckets` 的数量为 10。需要注意的是，`buckets` 只是设置了期望返回桶的数量，但实际返回桶的数量可能等于也可能小于 `buckets` 设置的值。例如示例的请求中期望 10 个桶，但实际可能只返回 6 个桶。

`auto_date_histogram` 聚集在返回结果中还提供了一个 `interval` 字段，用于说明实际采用的间隔时间。从实现的角度来说，不精确匹配 `buckets` 数量也有利于提升检索的性能。

## 子聚集 (聚集嵌套)

前面介绍的桶型聚集，大部分都只是返回满足聚集条件的文档数量。在实际应用中，如果需要桶型聚集与 SQL 中的 `group by` 具有相同的意义，用于将文档分桶后计算各桶特定指标值，比如根据用户性别分组，然后分别求他们的平均年龄。Elasticsearch 这样的功能通过子聚集 (聚集嵌套) 来实现。例如，示例中的请求就是先按月对从中国起飞的航班做了分桶，然后又通过聚集嵌套计算每月平均延误时间：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "query": {
    "term": {"OriginCountry": "CN"}
  },
  "aggs": {
    "date_price_histogram": {
      "date_histogram": {
        "field": "timestamp",
        "interval": "month"
      },
      "aggs": {
        "avg_delay": {
          "avg": {
            "script": {
              "source": "params._source.duration_in_millis / 1000.0"
            }
          }
        }
      }
    }
  }
}
```

```
        "date_histogram": {
            "field": "timestamp",
            "interval": "month"
        },
        "aggs": {
            "avg_price": {
                "avg": {
                    "field": "FlightDelayMin"
                }
            }
        }
    }
}
```

在示例中，`_search` 接口共使用了两个参数，`query` 参数以 `term` 查询条件将所有 `OriginCountry` 字段是 `CN` 的文档筛选出来参与聚集运算。

`aggs` 参数则定义了一个名称为 `data_price_histogram` 的桶型聚集，这个聚集内部又嵌套了个名称为 `avg_price` 的聚集。由于 `price` 这个聚集位于 `data_price histogram` 中，所以它会使用这个聚集的分桶结果做运算而不会针对所有文档。所以，最终的效果就是将按月计算从中国出发航班的平均延误时间，

使用嵌套聚集时要注意，嵌套聚集应该位于父聚集名称下而与聚集类型同级，并且需要通过参数再次声明。如果与父聚集一样位于 `aggs` 参数下，那么这两个聚集就是平级而非嵌套聚集。

## 使用词项分桶

使用字段值范围分桶主要针对结构化数据，比如年龄、IP 地址等等。但对于字符串类型的字段来说，使用值范围来分桶显然是不合适的。由于字符串类型字段在编入索引时会通过分析器生成词项，所以字符串类型字段的分桶一般通过词项实现。使用词项实现分桶的聚集，包括 `terms`、`significant_terms` 和 `significant_text` 聚集。由于使用词项分桶需要加载所有词项数据，所以它们在执行速度上都会比较慢。为了提升性能，Elasticsearch 提供了 `sampler` 和 `diversified_sampler` 聚集，可通过缩小样本数量减少运算量。

### terms 聚集

`terms` 聚集根据文档字段中的词项做分桶，所有包含同一词项的文档将被归入同一桶中，聚集结果中包含字段中的词项及其词频，在默认情况下还会根据词频排序，所以 `terms` 聚集也可用于热词展示，由于 `terms` 聚集在统计词项的词频

---

数据时需要打开它的 `fielddata` 机制。`fielddata` 机制对内存消耗较大且有导致内存溢出的可能，所以 `terms` 聚集一般针对 `keyword` 非 `text` 类型。

*Fielddata:* 其实根据倒排索引反向出来的一个正排索引，即 `document` 到 `term` 的映射。

只要我们针对需要分词的字段设置了 `fielddata`，就可以使用该字段进行聚合，排序等。我们设置为 `true` 之后，在索引期间，就会以列式存储在内存中。为什么存在于内存呢，因为按照 `term` 聚合，需要执行更加复杂的算法和操作，如果基于磁盘或者 OS 缓存，性能会比较差。

`fielddata` 堆内存要求很高，如果数据量太大，对于 JVM 来说存在一定的挑战，也就是对 ES 带来巨大的压力。所以 `doc_value` 的出现我们可以使用磁盘存储，他同样是和 `fielddata` 一样的数据结构，在倒排索引基础上反向出来的正排索引，并且是预先构建，即在建倒排索引的时候，就会创建 `doc values`，这会消耗额外的存储空间，但是对于 JVM 的内存需求就会减少。总体来看，`DocValues` 只是比 `fielddata` 慢一点，大概 10-25%，则带来了更多的稳定性。

`cardinality` 聚集可以统计字段中不重复词项的数量，而 `terms` 聚集则可以将这些词项全部展示出来。与 `cardinality` 聚集一样，`terms` 聚集统计出来的词频也不能保证完全精确。例如：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "aggs": {
    "country_terms": {
      "terms": {
        "field": "DestCountry",
        "size": 10
      }
    },
    "country_terms_count": {
      "cardinality": {
        "field": "DestCountry"
      }
    }
  }
}
```

在示例中定义了两个聚集，由于它们都是定义在 `aggs` 下，所以不是嵌套聚集。`terms` 聚集的 `field` 参数定义了提取词项的字段为 `DestCountry`，它的词项在返回结果中会按词频由高到低依次展示，词频会在返回结果的 `doc_count` 字段中展示。另一个参数 `size` 则指定了只返回 10 个词项，这相当把 `DestCountry` 字段中词频前 10 名检索出来。

---

## significant\_terms 聚集

terms 聚集统计在字段中的词项及其词频，聚集结果会按各词项总的词频排序，并将出现次数最多的词项排在最前面，这非常适合做推荐及热词类的应用。但按词频总数不一定可能是总是正确的选择，在一些检索条件已知的情况下，一些词频总数比较低的词项反而是更合适的推荐热词。

举例来说，假设在 10000 篇技术类文章的内容中提到 Elasticsearch 有 200 篇，占比为 2%；但在文章标题含有 NoSQL 的 1000 篇文章中，文章内容提到 Elasticsearch 的为 180 篇，占比为 18%。这种占比显著的提升，说明在文章标题含有 NoSQL 的条件下，Elasticsearch 变得更为重要。换句话说，如果一个词项在某个文档子集中与在文档全集中相比发生了非常显著的变化，就说明这个词项在这个文档子集中是更为重要的词项。

significant\_terms 聚集就是针对上述情况的一种聚集查询，它将文档和词项分为前景集 Foreground Set 和背景集(Background Set)。前景集对应一个文档子集，而背景集则对应文档全集。significant\_terms 聚集根据 query 指定前景集，运算 field 参数指定字段中的词项在前景集和背景集中的词频总数，并在结果的 doc\_count 和 bg\_count 中保存它们。例如：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "query": {
    "term": {
      "OriginCountry": {"value": "IE"}
    }
  },
  "aggs": {
    "dest": {
      "significant_terms": {
        "field": "DestCountry"
      }
    }
  }
}
```

在示例中，query 参数使用 DSL 指定了前景集为出发国家为 IE (即爱尔兰)的航班，而聚集查询中则使用 significant\_terms 统计到达国家的前景集词频和背景集词频。来看下返回结果：

```
{  
  "aggregations": {  
    "dest": {  
      "doc_count": 119,  
      "bg_count": 13059,  
      "buckets": [  
        {  
          "key": "GB",  
          "doc_count": 12,  
          "score": 0.19491470110905626,  
          "bg_count": 449  
        },  
        {  
          "key": "KR",  
          "doc_count": 7,  
          "score": 0.15232998092035055,  
          "bg_count": 214  
        },  
        {  
          "key": "PE",  
          "doc_count": 1,  
          "score": 0.15232998092035055,  
          "bg_count": 214  
        }  
      ]  
    }  
  }  
}
```

在返回结果中，前景集文档数量为 119，背景集文档数量为 13059。

在 buckets 返回的所有词项中，国家编码为 GB 的航班排在第一位。它在前景集中的词频为 12,占比约为 10% (12/119); 而在背景集中的词频为 449, 占比约为 3. 4% (445/13059)。

词项 **GB** 在前景集中的占比是背景集中的 3 倍左右，发生了显著变化，所以在这个前景集中 **GB** 可以被视为热词而排在第一位。**GB** 代表的国家是英国，从爱尔兰出发去英国的航班比较多想来也是合情合理的。

除了按示例方式使用 `query` 参数指定前景集以外，还可以将 `terms` 聚集与 `significant_terms` 聚集结合起来使用，这样可以一次性列出一个字段的所有前景集的热词。例如：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations  
{  
  "aggs": {  
    "orgin_dest": {  
      "terms": {  
        "  
      "field": "OriginCountry"  
    },  
    "aggs": {  
      "dest": {  
        "terms": {  
          "field": "DestCountry"  
        }  
      }  
    }  
  }  
}
```

```
        "significant_terms": {
          "field": "DestCountry"
        }
      }
    }
  }
}
```

在示例中，使用 terms 聚集将 OriginCountry 字段的词项全部查询出来做前景集，然后再与 significant\_terms 聚集起查询它们的热词。

## significant\_text 聚集

如果参与 significant\_terms 聚集的字段为 text 类型，那么需要将字段的 fielddata 机制开启，否则在执行时会返回异常信息。significant\_text 聚集与 significant\_terms 聚集的作用类型，但不需要开启字段的 fielddata 机制，所以可以把它当成是种专门为 text 类型字段设计的 significant\_terms 聚集。例如在 kibana\_sample\_data\_logs 中，message 字段即为 text 类型，如果想在这个字段上做词项分析就需要使用 significant\_terms 聚集：

```
POST /kibana_sample_data_logs/_search?filter_path=aggregations
{
  "query": {
    "term": {
      "response": {
        "value": "200"
      }
    },
    "aggs": {
      "agent_term": {
        "significant_text": {
          "field": "message"
        }
      }
    }
}
```

在示例中，前景集为响应状态码 response 为 200 的日志，significant\_text 聚集则查看在这个前景集下 message 字段中出现异常热度的词项。返回结果片段：

```
{  
  "aggregations" : {  
    "agent_term" : {  
      "doc_count" : 12832,  
      "bg_count" : 14074,  
      "buckets" : [  
        {  
          "key" : "200",  
          "doc_count" : 12832,  
          "score" : 0.09559395920909229,  
          "bg_count" : 12846  
        },  
        {  
          "key" : "beats",  
          "doc_count" : 3460,  
          "score" : 0.004397033874757019,  
          "bg_count" : 3734  
        },  
        {  
          "key" : "x86_64",  
          "doc_count" : 6124,  
          "score" : 0.0037046431288825427,  
          "bg_count" : 6665  
        },  
        {  
          "key" : "20110421",  
          "doc_count" : 4931,  
          "score" : 0.003315842203821136,  
          "bg_count" : 5362  
        },  
        {  
          "key" : "6.0a1",  
          "doc_count" : 4931,  
          "score" : 0.003315842203821136,  
          "bg_count" : 5362  
        }  
      ]  
    }  
  }  
}
```

通过展示的返回结果可以看出，排在第一位的词项 200 在前景集和背景集中的数量是一样的，这说明 message 中完整地记录了 200 状态码；而排在第二位的词项 beats 前景集和背景集分别为 3462 和 3732。这说明请求 “/beats” 地址的成功率要远高于其他地址。

`significant_text` 聚集之所以不需要开启 `fielddata` 机制是因为它会在检索时对 `text` 字段重新做分析，所以 `significant_text` 聚集在执行时速度比其他聚集要慢很多。如果希望提升执行效率，则可以使用 `sampler` 聚集通过减少前景集的样本数量降低运算量。

## 样本聚集

`sampler` 聚集的作用是限定其内部嵌套聚集在运算时采用的样本数量。`sampler` 提取样本时会按文档检索的相似度排序，按相似度分值由高到低的顺序提取。例如：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations  
{  
  "query": {  
    "term": {  
      "OriginCountry": {  
        "value": "IE"  
      }  
    }  
  }  
}
```

```
        }
    },
},
"aggs": {
    "sample_data": {
        "sampler": {
            "shard_size": 100
        },
        "aggs": {
            "dest_country": {
                "significant_terms": {
                    "field": "DestCountry"
                }
            }
        }
    }
}
}
```

在示例中共定义了 sample\_data 和 dest\_country 两个聚集，其中 dest\_country 是 sample\_data 聚集的子聚集或嵌套聚集，因此 dest\_country 在运算时就只从分片上取一部分样本做运算。sampler 聚集的 shard\_size 就是定义了每个分片上提取样本的数量，这些样本会根据 DSL 查询结果的相似度得分由高到低的顺序提取。

执行后会发现，这次目的地最热的目的地国家由 GB 变成了 KR, 这就是样本范围缩小导致的数据失真。为了降低样本减少对结果准确性的影响，需要将些重复的数据从样本中剔除。换句话说就是样本更加分散，加大样本数据的多样性。Elasticsearch 提供的 diversified\_sampler 聚集提供了样本多样性的能力，它提供了 field 或 script 两个参数用于去除样本中可能重复的数据。由于相同航班的票价可能是相同的，所以可以将票价相同的航班从样本中剔除以加大样本的多样性，例如：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
    "query": {
        "term": {
            "OriginCountry": {
                "value": "IE"
            }
        }
    }
}
```

```
        },
      },
      "aggs": {
        "sample_data": {
          "diversified_sampler": {
            "shard_size": 100,
            "field": "AvgTicketPrice"
          },
          "aggs": {
            "dest_country": {
              "significant_terms": {
                "field": "DestCountry"
              }
            }
          }
        }
      }
    }
```

`diversified_sampler` 通过 `field` 参数设置了 `AvgTicketPrice` 字段，这样在返回结果中 GB 就又重新回到了第一位。

## 单桶聚集

前面介绍的桶型聚集都是多桶型聚集，本节主要介绍单桶聚集

单桶聚集在返回结果中只会形成一个桶，它们都有比较特定的应用场景。在 Elasticsearch 中，单桶聚集主要包括 `filter`, `global`, `missing` 等几种类型。另外还有一种 `filters` 聚集，它虽然属于多桶聚集，但与 `filter` 聚集很接近，所以放到一起说明。

## 过滤器聚集

过滤器聚集通过定义一个或多个过滤器来区分桶，满足过滤器条件的文档将落入这个过滤器形成的桶中。过滤器聚集分为单桶和多桶两种，对应的聚集类型自然就是 `filter` 和 `filters`。

本来看 `filter` 桶型聚集，它属于单桶型聚集。一般会同时嵌套一个指标聚集，用于在过滤后的文档范围内计算指标，例如：

```
POST /kibana_sample_data_flights/_search?size=0&filter_path=aggregations
{
  "aggs": {
```

```
"origin_cn": {  
    "filter": {  
        "term": {  
            "OriginCountry": "CN"  
        }  
    },  
    "aggs": {  
        "cn_ticket_price": {  
            "avg": {  
                "field": "AvgTicketPrice"  
            }  
        }  
    },  
    "avg_price": {  
        "avg": {  
            "field": "AvgTicketPrice"  
        }  
    }  
}
```

在示例中一共定义了 3 个聚集，最外层是两个聚集，最后聚集为嵌套聚集，`origin_cn` 聚集为单过滤器的桶型聚集，它将所有 `OriginCountry` 为 CN 的文档归入一桶。

`origin_cn` 桶型聚集嵌套了 `cn_ticket_price` 指标聚集，它的作用是计算当前桶内文档 `AvgTicketPrice` 字段的平均值。另一个外层聚集 `avg_price` 虽然也是计算 `AvgTicketPrice` 字段的平均值，但它计算的是所有文档的平均值。实际上，使用 `query` 与 `agg` 结合起来也能实现类似的功能，区别在于过滤器不会做相似度计算，所以效率更高一些也更灵活一些。

多过滤器与单过滤器的作用类似，只是包含有多个过滤器，所以会形成多个桶。多过滤器型聚集使用 `filters` 参数接收过滤条件的数组，一般也是与指标聚集一同使用。例如使用两个过滤器计算从中国、美国出发的航班平均机票价格：

```
POST /kibana_sample_data_flights/_search?size=0&filter_path=aggregations  
{  
    "aggs": {  
        "origin_cn_us": {  
            "filters": {
```

---

```
"filters": [
  {
    "term": {
      "OriginCountry": "CN"
    }
  },
  {
    "term": {
      "OriginCountry": "US"
    }
  }
],
"aggs": {
  "avg_price": {
    "avg": {
      "field": "AvgTicketPrice"
    }
  }
}
}
```

## global 聚集

**global** 桶型聚集也是一种单桶型聚集，它的作用是把索引中所有文档归入一个桶中。这种桶型聚集看似没有什么价值，但当 **global** 桶型聚集与 **query** 结合起来使用时，它不会受 **query** 定义的查询条件影响，最终形成的桶中仍然包含所有文档。**global** 聚集在使用上非常简单，没有任何参数，例如：

```
POST /kibana_sample_data_flights/_search?size=0&filter_path=aggregations
{
  "query": {
    "term": {
      "Carrier": {
        "value": "Kibana Airlines"
      }
    }
  }
}
```

```
        },
      },
      "aggs":{
        "kibana_avg_delay":{
          "avg":{
            "field": "FlightDelayMin"
          }
        },
        "all flights":{
          "global":{},
          "aggs":{
            "all_avg_delay":{
              "avg":{
                "field": "FlightDelayMin"
              }
            }
          }
        }
      }
    }
  }
}
```

在示例中 query 使用 term 查询将航空公司为 “Kibana Airline”的文档都检索出来，而 kibana \_avg delay 定义的平均值聚集会将它们延误时间的平均值计算出来。但另个 all\_fights 聚集由于使用了 global 聚集所以在嵌套的 all\_avg\_delay 聚集中计算出来的是所有航班延误时间的平均值。

## missing 聚集

missing 聚集同样也是一种单桶型聚集，它的作用是将某一字段缺失的文档归入一桶。

missing 聚集使用 field 参数定义要检查缺失的字段名称，例如：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
```

```
{
  "aggs":{
    "no_price":{
      "missing":{
        "field": "AvgTicketPrice"
      }
    }
  }
}
```

```
    }
}
}
```

示例将 kibana\_sample\_data\_flights 中缺失 AvgTicketPrice 字段的文档归入一桶，通过返回结果的 doc\_count 查询数量也可以与指标聚集做嵌套，计算这些文档的某一指标值。

## 聚集组合

有两种比较特殊的多桶型聚集，它们是 composite 聚集和 adjacency\_matrix 聚集。这两种聚集是以组合不同条件的形式形成新桶，只是在组合的方法和组件的条件上存在着明显差异。

composite 聚集可以将不同类型的聚集组合到一起，它会从不同的聚集中提取数据，并以笛卡尔乘积的形式组合它们，而每一个组合就会形成一个新桶。例如想查看平均票价与机场天气的对应关系，可以这样：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "aggs": {
    "price_weather": {
      "composite": {
        "sources": [
          {"avg_price": {"histogram": {"field": "AvgTicketPrice",
          "interval": 500}}},
          {"weather": {"terms": {"field": "OriginWeather"}}}
        ]
      }
    }
  }
}
```

在示例中，composite 聚集中通过 sources 参数定义了两个需要组合的子聚集。第一个聚集 avg\_price 是一个针对 AvgTicketPrice 以 500 为间隔的 histogram 聚集，第二个则聚集 weather 则一个针对 OriginWeather 的 terms 聚集。sources 参数中还可以定义更多的聚集，它们会以笛卡儿乘积的形式组合起来。

在返回结果中除了由各聚集组合形成的桶以外，还有一个 after\_key 字段，

```
"aggregations" : {  
    "price_weather" : {  
        "after_key" : {  
            "avg_price" : 500.0,  
            "weather" : "Cloudy"  
        },  
        "buckets" : [  
            {
```

它包含自前聚集结果中最后一个结果的 key。所以请求下一页聚集结果就可以通过 after 和 size 参数值定，例如：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations  
{  
    "aggs":{  
        "price_weather":{  
            "composite":{  
                "after":{  
                    "avg_price":500.0,  
                    "weather":"Cloudy"  
                },  
                "sources": [  
                    {"avg_price": {"histogram": {"field": "AvgTicketPrice",  
"interval":500}}},  
                    {"weather": {"terms": {"field": "OriginWeather"}}}  
                ]  
            }  
        }  
    }  
}
```

adjacency\_matrix 又叫邻接矩阵，是图论中的概念，描述顶点之间的相邻关系， adjacency\_matrix 聚集因为牵涉到这些概念，略过，感兴趣的的同学可以自行研究。

## 管道聚集

管道聚集不是直接从索引中读取文档，而是在其他聚集的基础上再进行聚集运算。所以管道聚集可以理解为是在聚集结果上再次做聚集运算，比如求聚集结果中多个桶中某一指标的平均值、最大值等。要实现这样的目的，管道聚集都会包含一个名为 buckets\_path 的参数，用于指定访问其他桶中指标值的路径。

buckets\_path 参数的值由三部分组成，即聚集名称、指标名称和分隔符。

---

聚集名称与聚集名称之间的分隔符是“>”，而聚集名称与指标名称之间的分隔符使用“.”。

按管道聚集运算来源分类，管道聚集可以分为基于父聚集结果和基于兄弟聚集结果两类。前者使用父聚集的结果并将运算结果添加到父聚集结果中，后者则使用兄弟聚集的结果并且结果会展示在自己的聚集结果中。

## 基于兄弟聚集

基于兄弟聚集的管道聚集包括 avg\_bucket、max\_bucket、min\_bucket、sum\_bucket、stats\_bucket、extended\_stats\_bucket、percentiles\_bucket 七种。如果将它们名称中的 bucket 去除，它们就与本章前面介绍的部分指标聚集同名了。事实上，它们不仅在名称上接近，而且在功能上也类似，只是聚集运算的范围由整个文档变成了另一个聚集结果。以 avg\_bucket 为例，它的作用是计算兄弟聚集结果中某一指标的平均值：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "aggs": {
    "carriers": {
      "terms": {
        "field": "Carrier",
        "size": 10
      },
      "aggs": {
        "carrier_stat": {
          "stats": {
            "field": "AvgTicketPrice"
          }
        }
      }
    },
    "all_stat": {
      "avg_bucket": {
        "buckets_path": "carriers>carrier_stat.avg"
      }
    }
  }
}
```

---

例中，最外层包含有两个名称分别为 `carriers` 和 `all_stat` 的聚集，这两个聚集就是兄弟关系。

`carries` 聚集是一个针对 `Carrier` 字段的 `terms` 聚集，`Carrier` 字段保存的是航班承运航空公司，所以这个聚集的作用是按航空公司对航班分桶。在这个聚集中嵌套了一个名为 `carrier_stat` 的聚集，它是一个针对 `AvgTicketPrice` 字段的 `stats` 聚集，会按桶计算票价的最小值、平均值等统计数据。

`all_stat` 聚集则是一个 `avg_bucket` 管道聚集，在它的 `Path` 参数中指定了运算平均值的路径 "`carriers>carrier_stat.avg`"，即从兄弟聚集中查找 `carrier_stat` 指标聚集，然后再用其中的 `avg` 字段参与平均值计算。所以 `all_stat` 这计算出来的是四个航空公司的平均票价的平均值，实际上就是所有航班的平均票价。

尽管示例针对 `avg_bucket` 管道聚集的检索，但使用其余六种基于兄弟的管道聚集的关键字值替换后，它们就变成了另一种合法的管道聚集了。

## 基于父聚集

基于父聚集的管道聚集包括 `moving_avg`、`moving_fn`、`bucket_script`、`bucket_selector`、`bucket_sort`、`derivative`、`cumulative_sum`、`serial_diff` 八种。

### 滑动窗口

`moving_avg` 和 `moving_fn` 这两种管道聚集的运算机制相同，都是基于滑动窗口( Siding Window)算法对父聚集的结果做新的聚集运算。滑动窗口算法使用一个具有固定宽度的窗口滑过一组数据，在滑动的过程中对落在窗口内的数据做运算。`moving_avg` 管道聚集是对落在窗口内的父聚集结果做平均值运算，而 `moving_fn` 管道聚集则可以对落在窗口内的父聚集结果做各种自定义的运算。由于 `moving_avg` 管道可以使用 `moving_fn` 管道聚集实现，所以 `moving_avg` 在 Elasticsearch 版本 6.4.0 中已经被废止。

由于使用滑动窗口运算时每次移动 1 个位置，这就要求 `moving_avg` 和 `moving_fn` 所在聚集桶与桶间隔必须固定，所以这两种管道聚集只能在 `histogram` 和 `date_histogram` 聚集中使用：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "aggs": {
    "day_price": {
      "date_histogram": {
        "field": "timestamp",
        "interval": "day"
      },
      "aggs": {
        "avg_price": {
          "avg": {
            "script": {
              "source": "params._source['AvgTicketPrice']"
            }
          }
        }
      }
    }
  }
}
```

```
        "field": "AvgTicketPrice"
    }
},
"smooth_price": {
    "moving_fn": {
        "buckets_path": "avg_price",
        "window": 10,
        "script": "MovingFunctions.unweightedAvg(values)"
    }
}
}
}
}
```

在示例中，最外层的父聚集 `day_price` 是 1 个 `date_histogam` 框型聚集，它根据文档的 `timestamp` 字段按天将文档分桶。`day_price` 聚集包含 `avg_price` 和 `smooth_price` 两个子聚集，其中 `avg_price` 聚集是一个求 `AvgTicketPrice` 字段在 1 个桶内平均值的 `avg` 聚集，而 `smooth_price` 则是一个使用滑动窗口做平均值平滑的管道聚集，窗口宽度由参数 `window` 设置为 10，默认值为 5。

通过返回结果比较 `avg_price` 与 `smooth_price` 就会发现，后者由于经过了滑动窗口运算，数据变化要平滑得多。

`moving_fn` 聚集包含一个用于指定运算脚本的 `script` 参数，在脚本中可以通过 `values` 访问 `buckets_path` 参数指定的指标值。`moving_fn` 还内置了一个 `MovingFunctions` 类，包括多个运算函数：

```
max()
min()
sum()
stdDev() 标准偏差
unweightedAvg() 无加权平均值
linearWeightedAvg() 线性加权移动平均值
ewma() 指数加权移动平均值
holt() 二次指数加权移动平均值
holtWinters() 三次指数加权移动平均值
```

## 单桶运算

目前我们学习的管道聚集会对父聚集结果中落在窗口内的多个桶做聚集运算，而 `bucket_script`、`bucket_selector`、`bucket_sort` 这三个管道聚集则会针对

---

父聚集结果中的每一个桶做单独的运算。其中，`bucket_script`会对每个桶执行一段脚本，运算结果会添加到父聚集的结果中，`bucket_selector`同样也是执行一段脚本，但它执行的结果一定是布尔类型，并且决定当前桶是否出现在父聚集的结果中；`bucket_sort`则根据每桶中的具体指标值决定桶的次序。下面通过示例来说明这三种管道聚集的具体用法：

```
POST /kibana_sample_data_flights/_search?filter_path=aggregations
{
  "aggs": {
    "date_price_diff": {
      "date_histogram": {
        "field": "timestamp",
        "fixed_interval": "1d"
      },
      "aggs": {
        "stat_price_day": {
          "stats": {"field": "AvgTicketPrice"}
        },
        "diff": {
          "bucket_script": {
            "buckets_path": {
              "max_price": "stat_price_day.max",
              "min_price": "stat_price_day.min"
            },
            "script": "params.max_price - params.min_price"
          }
        },
        "gt990": {
          "bucket_selector": {
            "buckets_path": {
              "max_price": "stat_price_day.max",
              "min_price": "stat_price_day.min"
            },
            "script": "params.max_price - params.min_price > 990"
          }
        },
        "sort_by": {
      
```

```
        "bucket_sort": {
            "sort": [
                {"diff": {"order": "desc"}}
            ]
        }
    }
}
}
```

在示例中同时应用这三种管道聚集，它们的聚集名称分别为 diff、gt990 和 sort\_by。最外层的 date\_price\_diff 聚集是一个以天为固定间隔的 date\_histogram 聚集，其中又嵌套了包括上述三个聚集在内的子聚集。

其中，stat\_price\_day 是一个根据 AvgTicketPrice 字段生成统计数据的 stats 聚集。diff 是一个 bucket\_script 管道聚集，它的作用是向最终聚集结果中添加代表最大值与最小值之差的 diff 字段。它通过 buckets\_path 定义了两个参数 max\_price 和 min\_price，并在 script 参数中通过脚本计算了这两个值的差作为最终结果，而这个结果将出现在整个聚集结果中。

gt990 是一个 bucket\_selector 管道聚集，它的作用是筛选哪些桶可以出现在最终的聚集结果中。它也在 buckets\_path 中定义了相同的参数，不同的是它的 script 参数运算的不是差值，而是差值是否大于 990，即 “`params.max_price - params.min_price > 990`”。如果差值大于 990 即运算结果为 true，那么当前桶将被选取到结果中，否则当前桶将不能在结果中出现。

sort\_by 是一个 bucket\_sort 管道聚集，它的作用是给最终的聚集结果排序。它通过 sort 参数接收一组排序对象，在示例中是使用 diff 聚集的结果按倒序排序。

所以示例整体的运算效果就是将那些票价最大值与最小值大于 990 的桶选取出来，并在桶中添加 diff 字段保存最大值与最小值的差值，并按 diff 字段值降序排列。

## 矩阵聚集

前面几节介绍的聚集都是针对一个字段做聚集运算，而矩阵聚集则是针对多个字段做多种聚集运算，因此产生的结果是一个矩阵。该类型的聚集比较简单，目前只支持一种针对多个字段做统计运算的矩阵聚集。这个聚集目前还不稳定，我们不予了解。

## 父子关系

---

Elasticsearch 中的父子关系是单个索引内部文档与文档之间的一种关系，父文档与子文档同属一个索引并通过父文档 `id` 建立联系，类似于关系型数据库中单表内部行与行之间的自关联，比如有层级关系的部门表中记录之间的关联。

## join 类型

在 Elasticsearch 中并没有外键的概念，文档之间的父子关系通过给索引定义 `join` 类型字段实现。例如创建一个员工索引 `employees`, 定义个 `join` 类型的 `management` 字段用于确定员工之间的管理与被管理关系:

```
PUT employees
{
  "mappings": {
    "properties": {
      "management": {
        "type": "join",
        "relations": {
          "manager": "member"
        }
      }
    }
  }
}
```

在示例中，`management` 字段的数据类型被定义为 `join`, 同时在该字段的 `relations` 参数中定义父子关系为 `manager` 与 `member`, 其中 `manager` 为父而 `member` 为子，它们的名称可由用户自定义。文档在父子关系中的地位，是在添加文档时通过 `join` 类型字段指定的。还是以 `employees` 索引为例，在向 `employees` 索引中添加父文档时，应该将 `mangement` 字段设置为 `manager`; 而添加子文档时则应该设置为 `member`。具体如下:

```
PUT /employees/_doc/1
{
  "name": "tom",
  "management": {
    "name": "manager"
  }
}
```

```
PUT /employees/_doc/2?routing=1
{
```

```
"name" : "smith",
"management":{
    "name" : "member",
    "parent": "1"
}
```

```
PUT /employees/_doc/3?routing=1
{
    "name" : "john",
    "management":{
        "name" : "member",
        "parent": "1"
    }
}
```

在示例中，编号为 1 的文档其 `management` 字段通过 `name` 参数设置为 `manager`, 即在索引定义父子关系中处于父文档的地位，而编号为 2 和 3 的文档其 `management` 字段则通过 `name` 参数设置为 `member`, 并通过 `parent` 参数指定了它的父文档为编号 1 的文档。

在使用父子关系时，要求父子文档必须要映射到同一分片中，所以在添加子文档时 `routing` 参数是必须要设置的。显然父子文档在同一分片可以提升在检索时的性能，可在父子关系中使用的查询方法有 `has_child`、`has_parent` 和 `parent_id` 查询，还有 `parent` 和 `children` 两种聚集。

## has\_child 查询

`has_child` 查询是根据子文档检索父文档的一种方法，它先根据查询条件将满足条件的子文档检索出来，在最终的结果中会返回具有这些子文档的父文档。例如，如果想检索 `smith` 的经理是谁，可以：

```
POST /employees/_search
{
    "query": {
        "has_child": {
            "type": "member",
            "query": {
                "match": {
                    "name": "smith"
                }
            }
        }
    }
}
```

```
        }
    }
}
}
```

在示例中，`has_child` 查询的 `type` 参数需要设置为父子关系中子文档的名称 `member`，这样 `has_child` 查询父子关系时就限定在这种类型中检索；`query` 参数则设置了查询子文档的条件，即名称为 `smith`。最终结果会根据 `smith` 所在文档，通过 `member` 对应的父子关系检索它的父文档。

## has\_parent 查询

`has_parent` 查询与 `has_child` 查询正好相反，是通过父文档检索子文档的一种方法。在执行流程上，`has_parent` 查询先将满足查询条件的父文档检索出来，但在最终返回的结果中展示的是具有这些父文档的子文档。例如，如果想查看 `tom` 的所有下属，可以按示例请求：

```
POST /employees/_search
{
  "query": {
    "has_parent": {
      "parent_type": "manager",
      "query": {
        "match": {
          "name": "tom"
        }
      }
    }
  }
}
```

`has_parent` 查询在结构上与 `has_child` 查询基本相同，只是在指定父子关系时使用的参数

是 `parent_type` 而不是 `type`。

## parent\_id 查询

`parent_id` 查询与 `has_parent` 查询的作用相似，都是根据父文档检索子文档。不同的是，`has_parent` 可以通过 `query` 参数设置不同的查询条件；而 `parent_id` 查询则只能通过父文档 `id` 做检索。例如，查询 `id` 为 1 的子文档：

```
POST /employees/_search
{
}
```

---

```
"query": {
  "parent_id": {
    "type": "member",
    "id": 1
  }
}
```

## children 聚集

如果想通过父文档检索与其关联的所有子文档就可以使用 `children` 聚集。同样以 `employees` 索引为例, 如果想要查看 `tom` 的所有下属就可以按示例的方式检索:

```
POST /employees/_search?filter_path=aggregations
{
  "query": {
    "term": {
      "name": "tom"
    }
  },
  "aggs": {
    "members": {
      "children": {
        "type": "member"
      },
      "aggs": {
        "member_name": {
          "terms": {
            "field": "name.keyword",
            "size": 10
          }
        }
      }
    }
  }
}
```

---

在示例中，`query` 参数设置了父文档的查询条件，即名称字段 `name` 为 `tom` 的文档，而聚集查询 `members` 中则使用了 `children` 聚集将它的子文档检索出来，同时还使用了一个嵌套聚集 `member_name` 将子文档 `name` 字段的词项全部展示出来了。

## parent 聚集

`parent` 聚集与 `children` 聚集正好相反，它是根据子文档查找父文档，`parent` 聚集在 Elasticsearch 版本 6.6 以后才支持。例如通过 `name` 字段为 `smith` 的文档，在找该文档的父文档：

```
POST /employees/_search?filter_path=aggregations
{
  "query": {
    "match": {
      "name": "smith"
    }
  },
  "aggs": {
    "who_is_manager": {
      "parent": {
        "type": "member"
      },
      "aggs": {
        "manager_name": {
          "terms": {
            "field": "name.keyword",
            "size": 10
          }
        }
      }
    }
  }
}
```

## 嵌套类型

前面所说的对象类型虽然可按 JSON 对象格式保存结构化的对象数据，但由于 Lucene 并不支持对象类型，所以 Elasticsearch 在存储这种类型的字段时会将它们平铺为单个属性。例如：

---

```
PUT colleges/_doc/1
{
  "address": {
    "country": "CN",
    "city": "BJ"
  },
  "age": 10
}
```

在示例中的 `colleges` 文档，`address` 字段会被平铺为 `address.country` 和 `address.city` 两个字段存储。这种平铺存储的方案在存储单个对象时没有什么问题，但如果在存储数组时会丢失单个对象内部字段的匹配关系。例如：

```
PUT colleges/_doc/2
{
  "address": [
    {
      "country": "CN",
      "city": "BJ"
    },
    {
      "country": "US",
      "city": "NY"
    }
  ],
  "age": 10
}
```

示例中的 `colleges` 文档在实际存储时，会被拆解为“`address.country": ["CN.US"]`” 和 `address.city": ["BJ", "NY"]`” 两个数组字段。这样一来，单个对象内部，`country` 字段和 `city` 字段之间的匹配关系就丢失了。换句话说，使用 `CN` 与 `NY` 作为共同条件检索的文档时，上述文档也会被检索出来，这在逻辑上就出现了错误：

```
POST colleges/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": { "address.country": "CN"}},
        {"match": { "address.city": "NY"}}
      ]
    }
  }
}
```

```
        ]
    }
}
}
```

在示例中使用了 `bool` 组合查询，要求 `country` 字段为 `CN` 而 `city` 字段为 `NY`。这样的文档显然并不存在，但由于数组中的对象被平铺为两个独立的数组字段，文档仍然会被检索出来。

## nested 类型

为了解决对象类型在数组中丢失内部字段之间匹配关系的问题，Elasticsearch 提供了一种特殊的对象类型 `nested`。这种类型会为数组中的每一个对象创建一个单独的文档，以保存对象的字段信息并使它们可检索。由于这类文档并不直接可见，而是藏置在父文档之中，所以这类文档可以称为为隐式文档或嵌入文档。还是以 `colleges` 索引为例，我们把原有的索引删除，将它的 `address` 字段设置为 `nested` 类型：

```
PUT colleges
{
  "mappings": {
    "properties": {
      "address": {
        "type": "nested"
      },
      "age": {
        "type": "integer"
      }
    }
  }
}
```

然后重新存入文档 1 和 2，当字段被设置为 `nested` 类型后，再使用原来查询中的 `bool` 组合查询就不能检索出来了。这是因为对 `nested` 类型字段的检索实际上是对隐式文档的检索，在检索时必须要将检索路由到隐式文档上，所以必须使用专门的检索方法。也就是说，现在即使将原来查询中的查询条件设置为 `CN` 和 `BJ` 也不会检索出结果。

`nested` 类型字段可使用的检索方法包括 DSL 的 `nested` 查询，还有聚集查询中的 `nested` 和 `reverse_nested` 两种聚集。

---

## nested 查询

nested 查询只能针对 nested 类型字段，需要通过 path 参数指定 nested 类型字段的路径，而在 query 参数中则包含了针对隐式文档的具体查询条件。例如：

```
POST /colleges/_search
{
  "query": {
    "nested": {
      "path": "address",
      "query": {
        "bool": {
          "must": [
            {"match": {"address.country": "CN"}},
            {"match": {"address.city": "NY"}}
          ]
        }
      }
    }
  }
}
```

在示例中再次使用 CN 与 NY 共同作为查询条件，但由于使用 nested 类型后会将数组中的对象转换成隐式文档，所以在 nested 查询中将不会有文档返回了。将条件更换为 CN 和 BJ，则有文档返回。

## nested 聚集

nested 聚集是一个单桶聚集，也是通过 path 参数指定 nested 字段的路径，包含在 path 指定路径中的隐式文档都将落入桶中。所以 nested 字段保存数组的长度就是单个文档落入桶中的文档数量，而整个文档落入桶中的数量就是所有文档 nested 字段数组长度的总和。有了 nested 聚集，就可以针对 nested 数组中的对象做各种聚集运算，例如：

```
POST /colleges/_search?filter_path=aggregations
{
  "aggs": {
    "nested_address": {
      "nested": {
        "path": "address"
      },
    }
  }
}
```

---

```
"aggs":{  
    "city_names": {  
        "terms":{  
            "field": "address.city.keyword",  
            "size": 10  
        }  
    }  
}  
}  
}
```

在示例中，`nested_address` 是一个 `nested` 聚集的名称，它会将 `address` 字段的隐式文档归入一个桶中。而嵌套在 `nested_address` 聚集中 `city_names` 聚集则会在这个桶中再做 `terms` 聚集运算，这样就将对象中 `city` 字段所有的词项枚举出来了。

## reverse\_nested 聚集

`reverse_nested` 聚集用于在隐式文档中对父文档做聚集，所以这种聚集必须作为 `nested` 聚集的嵌套聚集使用。例如：

```
POST /colleges/_search?filter_path=aggregations  
{  
    "aggs": {  
        "nested address": {  
            "nested":{  
                "path": "address"  
            },  
            "aggs":{  
                "city names":{  
                    "terms":{  
                        "field": "address.city.keyword",  
                        "size": 10  
                    },  
                    "aggs": {  
                        "avg_age_in_city":{  
                            "reverse_nested": {}  
                        },  
                        "aggs": {  
                            "size": 10  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```
        "avg_age": {
          "avg": {"field": "age"}
        }
      }
    }
  }
}
```

在示例中, `city_names` 聚集也是将隐式文档中 `city` 字段的词项全部聚集出来。不同的是在这个聚集中还嵌套了一个名为 `avg_age_in_city` 的聚集, 这个聚集就是个 `reverse_nested` 聚集。它会在隐式文档中将 `city` 字段具有相同词项的文档归入一个桶中, 而 `avg_age_in_city` 聚集嵌套的另外一个名为 `avg_age` 的聚集, 它会把落入这个桶中文档的 `age` 字段的平均值计算出来。所以从总体上来看, 这个聚集的作用就是将在同一城市中大学的平均校龄计算出来。

## 使用 SQL 语言

Elasticsearch 在 Basic 授权中支持以 SQL 语句的形式检索文档, SQL 语句在执行时会被翻译为 DSL 执行。从语法的角度来看, Elasticsearch 中的 SQL 语句与 RDBMS 中的 SQL 语句基本一致, 所以对于有数据库编程基础的人来说大大降低了使用 Elasticsearch 的学习成本。

Elasticsearch 提供了多种执行 SQL 语句的方法, 可使用类似 `_search` 样的 REST 接口执行也可以通过命令行执行。它甚至还提供了 JDBC 和 ODBC 驱动来执行 SQL 语句, 但 JDBC 和 ODBC 属于 Platinum (白金版) 授权需要付费, 所以本小节将只介绍 `_sql` 接口。

### sql 接口

在早期版本中, Elasticsearch 执行 SQL 的 REST 接口为 `_xpack/sql`, 但在版本 7 以后这个接口已经被废止而推荐使用 `_sql` 接口。例如:

```
POST _sql?format=txt
{
  "query": """
select DestCountry, OriginCountry,AvgTicketPrice
from kibana_sample_data_flights
where Carrier = 'Kibana Airlines'
order by AvgTicketPrice desc
  """
```

.....

}

在示例中，\_sql 接口通过 query 参数接收 SQL 语句，而 SQL 语句也包含有 select、from、where、orderby 等子句。\_sql 接口的 URL 请求参数 format 定义了返回结果格式。比如在示例中定义了返回结果格式为 txt。除了 txt 以外，\_sql 接口还支持 csv、json、tsv、yaml 等等格式。

示例中的请求会将所有航空公司为 Kibana Airlines 的航班文档检索出来，并以文本表格的形式返回。

1	DestCountry	OriginCountry	AvgTicketPrice
2			
3	US	PR	[1199.109130859375]
4	JP	IN	[1196.7706298828125]
5	AR	CO	[1195.72509765625]
6	IT	CA	[1195.3363037109375]
7	KR	NL	[1194.945556640625]
8	JP	CO	[1194.386962890625]
9	CN	PE	[1193.750244140625]
10	JP	IT	[1189.6920166015625]
11	IT	GB	[1189.5384521484375]
12	IT	EC	[1188.65576171875]
13	AU	IT	[1188.0474853515625]
14	IN	AE	[1188.0367431640625]
15	CH	IT	[1187.61083984375]
16	RU	US	[1186.8880615234375]
17	PL	CO	[1185.43701171875]
18	PR	US	[1182.4656982421875]
19	DE	CO	[1182.4119873046875]
20	CA	DK	[1181.982177734375]
21	AT	CA	[1181.8358154296875]
22	CA	NL	[1181.77001953125]
23	AT	ZA	[1180.7469482421875]
24	IT	US	[1178.8028564453125]
25	RU	FI	[1176.0587158203125]

对于总量比较大的 SOL 查询，sql 接口还支持以游标的形式实现分页。当\_sql 接口的请求参数中添加了 fetch\_size 参数，sql 接口在返回结果时就会根据 fetch\_size 参数设置的大小返回相应的条数，并在返回结果中添加游标标识。具体来说，当请求\_sql 接口时设置的 forma 为 json 时，返回结果中会包含 cursor 属性；而其他情况下则会在响应中添加 Cursor 报头。例如还是执行示例中的 SOL，但是加入分页支持：

```
POST _sql?format=json
{
  "query": """
    select DestCountry, OriginCountry,AvgTicketPrice
    from kibana_sample_data_flights
    where Carrier = 'Kibana Airlines'
    order by AvgTicketPrice desc
    """,
  "fetch_size":10
}
```

在示例的请求中，为了能够在返回结果中直接看到 cursor 值，我们将 format 设置为 json，可以看到：

```
[ ],
  "cursor" : "k8OvAwFaAXN4RkdsdVkyeDFaR1ZmWTI5dWRHVjRkRjkxZFds
  a0RYRjFaWEo1UVc1a1JtVjBZMmdCRkhJeGJWbDNXRWxDTVZKMl
  owOXpha2MwV0RWTEFBQUFBQUFCY1EwV1dIZ3daMFZWVld4U1NtbE1s
  VXBTWVdaRFN6RmlVUT09
  ////w8DAWYLRGVzdENvdW50cnkBC0Rlc3RDb3VudHJ5AQdrZXl3b3
  JkAQAAWYNT3JpZ2luQ291bnRyeQENT3JpZ2luQ291bnRyeQEHa2V5
  d29yZAEEAAFmDkF2Z1RpY2tIdFByaWNIAQ5BdmdUaWNrZXRQcm1jZQ
  EFZmxvYXQAAAABBw=="
}
```

```
POST _sql?format=json
{
  "cursor": "k8OvAwFaAXN4RkdsdVkyeDFaR1ZmWTI5dWRHVjRkRjkxZFds
  a0RYRjFaWEo1UVc1a1JtVjBZMmdCRkdWV2JWZDNXRWxDTVZKMlowOXpha2R4U0RVM0F
  BQUFBQUFCYk5jV1dIZ3daMFZWVld4U1NtbEISVXBTVdaRFN6RmlVUT09////w8DA
  WYLRGVzdENvdW50cnkBC0Rlc3RDb3VudHJ5AQdrZXl3b3JkAQAAWYNT3JpZ2luQ29
  1bnRyeQENT3JpZ2luQ291bnRyeQEHa2V5d29yZAEEAAFmDkF2Z1RpY2tIdFByaWNIA
  Q5BdmdUaWNrZXRQcm1jZQEFZmxvYXQAAAABBw=="
}
```

而在上面的请求中，参数 `cursor` 就是第一个请求返回结果中的 `cursor` 值，反复执行请求，Elasticsearch 就会将第一次请求的全部内容以每次 10 个的数量全部迭代出来。在请求完所有数据后，应该使用 `_sql/close` 接口将游标关闭以释放资源。

```
POST _sql/close
{
  "cursor": "k8OvAwFaAXN4RkdsdVkyeDFaR1ZmWTI5dWRHVjRkRjkxZFds
  a0RYRjFaWEo1UVc1a1JtVjBZMmdCRkdWV2JWZDNXRWxDTVZKMlowOXpha2R4U0RVM0F
  BQUFBQUFCYk5jV1dIZ3daMFZWVld4U1NtbEISVXBTVdaRFN6RmlVUT09////w8DA
  WYLRGVzdENvdW50cnkBC0Rlc3RDb3VudHJ5AQdrZXl3b3JkAQAAWYNT3JpZ2luQ29
  1bnRyeQENT3JpZ2luQ291bnRyeQEHa2V5d29yZAEEAAFmDkF2Z1RpY2tIdFByaWNIA
  Q5BdmdUaWNrZXRQcm1jZQEFZmxvYXQAAAABBw=="
}
```

除了 `fetch_size` 以外还有些可以在 `_sql` 接口请求体中使用的参数，如下：

`query` 需要执行的 SQL 语句，必须要设置的参数

`fetch_size` 默认 1000，每次返回的行数

`filter` 默认 `none`，使用 DSL 设置过滤器

`request_timeout` 默认 90s，请求超时时间

`page_timeout` 默认 45s，分页超时时间

`tume_zone` 默认 Z，时区

`field_multi_value_leniency` 默认 `false`，如果一个字段返回多个值时是否忽略

---

在这些参数中，filter 可以使用 DSL 对文档做过滤，支持 DSL 中介绍的所有查询条件。query 中的 SQL 语句在翻译为 DSL 后，会与 filter 中的 DSL 查询语句共同组合到 bool 查询中。其中 SQL 语句生成的 DSL 将出现在 must 子句，而 filter 中的 DSL 则出现在 filter 子句中。来想要查看 SQL 语句翻译后的 DSL，可以使用 \_sql/translate 执行相同的请求，在返回结果中就可以看到翻译后的 DSL 了。

## SQL 语法

Elasticsearch 支持传统关系型数据库 SQL 语句中的查询语句，但并不支持 DML、DCL 句。换句话说，它只支持 SELECT 语句，不支持 INSERT、UPDATE、DELETE 语句”。SELECT 语句以外，Elasticsearch 还支持 DESCRIBE 和 SHOW 语句。

### SELECT 语句

SELECT 语句用于查询文档，基本语法格式如示例 8-29 所示：

```
SELECT select_expr,  
      [ FROM table_name ]  
      [ WHERE condition ]  
      [ GROUP BY grouping_element ]  
      [ HAVING condition ]  
      [ ORDER BY expression [ASC|DESC] ]  
      [ LIMIT[ count]]]
```

通过示例可以看出，Elasticsearch 的 SELECT 语句跟普通 SQL 几乎没有什么区别，支持 SELECT、FROM、WHERE、GROUP BY、HAVING、ORDER BY 及 LIMIT 子句。

SELECT 子句中可以使用星号或文档字段名称列表，FROM 子句则指定要检索的索引名称，而 WHERE 子句则设定了检索的条件。一般的 SQL 查询使用这三个子句就足够了，而 GROUP BY 和 HAVING 子句则用于分组，ORDER BY 子句用于排序，而 LIMIT 一般则可以用于分页。和传统 SQL 语句非常接近。

### DESCRIBE 语句

DESCRIBE 语句用于查看一个索引的基础信息，在返回结果中一般会包含 column、type、mapping 三个列，分别对应文档的字段名称、传统数据库类型及文档字段中的类型。例如要查看索引的基本信息：

```
POST _sql?format=txt  
{  
  "query":"describe kibana_sample_data_flights"  
}
```

---

1	column	type	mapping
2			
3	AvgTicketPrice	REAL	float
4	Cancelled	BOOLEAN	boolean
5	Carrier	VARCHAR	keyword
6	Dest	VARCHAR	keyword
7	DestAirportID	VARCHAR	keyword
8	DestCityName	VARCHAR	keyword
9	DestCountry	VARCHAR	keyword
10	DestLocation	GEOOMETRY	geo_point
11	DestRegion	VARCHAR	keyword
12	DestWeather	VARCHAR	keyword
13	DistanceKilometers	REAL	float
14	DistanceMiles	REAL	float
15	FlightDelay	BOOLEAN	boolean
16	FlightDelayMin	INTEGER	integer

## SHOW 语句

SHOW 语句包括三种形式，即 SHOW COLUMNS、SHOW FUNCTIONS 和 SHOW TABLES。

SHOW COLUMNS 用于查看一个索引中的字段情况，它的作用与 DESCRIBE 语句完全一样，其甚至连返回结果都是一样的。

SHOW FUNCTIONS 用于返回在 Elasticsearch SQL 中支持的所有函数，返回结果中包括 MIN、MAX、COUNT 等常用的聚集函数。

最后，SHOW TABLES 用看 Elasticsearch 中所有的索引。

```
POST _sql?format=txt
{
  "query": "show columns in kibana_sample_data_flights"
}
```

```
POST _sql?format=txt
{
  "query": "show functions"
}
```

```
POST _sql?format=txt
{
  "query": "show tables"
}
```

这三种形式都支持使用 LIKE 子句过滤返回结果，LIKE 子句在用法上与 SQL 语句中的 LIKE 类似。例如，“show functions like 'a%'”将只返回以 a 开头的函数。

## 操作符与函数

Elasticsearch SQL 中支持的操作符与函数有 100 多种，这些操作符大多与普通 SQL 语言一致，所以这里只介绍一些与普通 SQL 语句不一样的地方。

先来看一下比较操作符。一般等于比较在 SQL 中使用等号 “=” ，这在 ElasticsearchSQL 中也成立。但是 ElasticsearchSQL 还引入了另一个等号比较 “<=>” ，这种等号可以在左值为 null 时不出现异常。



```
POST _sql?format=txt
{
  "query": "select null = 'elastic'"
}

POST _sql?format=txt
{
  "query": "select null <=> 'elastic'"
}
```

LIKE 操作符，在 LIKE 子句中可以使用%代表任意多个字符，而使用下划线 \_ 代表单个字符。Elasticsearch SQL 不仅支持 LIKE 子句，还支持通过 RLIKE 子句以正则表达式的形式做匹配，这大大扩展了 SQL 语句模糊匹配的能力。

尽管使用 LIKE 和 RLIKE 可以实现模糊匹配，但它离全文检索还差得很远。SQL 语句的 WHERE 子句一般都是使用字段整体值做比较，而没有使用词项做匹配的能力。为此 Elasticsearch SQL 提供了 MATCH 和 QUERY 两个函数，以实现在 SQL 做全文检索。例如在示例 8-33 中的两个请求分别使用 match 和 query 函数，它们的作用都是检索 DestCountry 字段为 CN 的文档：

```
POST _sql?format=txt
{
  "query": """
    select DestCountry, OriginCountry,AvgTicketPrice,score()
    from kibana_sample_data_flights
    where match(DestCountry,'CN')
    """
}

POST _sql?format=txt
{
  "query": """
    select DestCountry, OriginCountry,AvgTicketPrice,score()
    from kibana_sample_data_flights
    where query('DestCountry:CN')
    """
}
```

在示例中的两个请求的 select 子句中都使用了 SCORE 函数，它的作用是获取检索的相关度评分值。

---

Elasticsearch SQL 支持传统 SQL 中的聚集函数，这包括 MAX、MIN、AVG、COUNT、SUM 等。同时，它还支持一些 Elasticsearch 特有的聚集函数，这些聚集函数与 Elasticsearch 聚集查询相对应。这包括 FIRST/FIRST\_VALUE 和 LAST/LAST VALUE，可用于查看某个字段首个和最后一个非空值；PERCENTILE 和 PERCENTILE RANK 用于百分位聚集，KURTOSIS、SKEWNESS、STDDEV\_POP、SUM\_OF\_SQUARES 和 VAR\_POP 可用于运算其他统计聚集。除了以上这些函数和操作符，Elasticsearch SQL 还定义了一组用于日期、数值以及字符串运算的函数。

# 与 Spring 的集成

## 综述

目前常见的 Elasticsearch Java API 有四类 client 连接方式：

**TransportClient**（不推荐），Elasticsearch 原生的 api，TransportClient 可以支持 2.x、5.x 版本，TransportClient 将会在 Elasticsearch 7.0 弃用并在 8.0 中完成删除。

**RestClient**，ES 官方推荐使用。

**Jest**（不推荐），是 Java 社区开发的，是 Elasticsearch 的 Java Http Rest 客户端。

**Spring Data Elasticsearch**，与 Spring 生态对接，可以在 web 系统中整合到 Spring 中使用，与 SpringBoot、SpringData 版本容易冲突，而且往往很难跟上 Elasticsearch 版本的更新，比如 SpringBoot 目前的 2.3.1.RELEASE，所支持 Elasticsearch 7.6.2。

从使用上来说，Spring Data 的使命是给各种数据访问提供统一的编程接口，不管是关系型数据库（如 MySQL），还是非关系数据库（如 Redis），或者类似 Elasticsearch 这样的索引数据库。从而简化开发人员的代码，提高开发效率，也就是说，Spring Data 想要把对任何数据的访问都抽象为类似接口，这就导致了 Spring Data Elasticsearch 在基本查询上没问题，但是复杂查询（模糊、通配符、match 查询、聚集查询等）就显得力不从心了，此时，我们还是只能使用原生查询。

所以本次课程中，我们把精力放在 REST Client 上，Java REST Client 有 Low Level 和 High Level 两种：

**Java Low Level REST Client**：使用该客户端需要将 HTTP 请求的 body 手动拼成 JSON 格式，HTTP 响应也必须将返回的 JSON 数据手动封装成对象，使用上更为原始。

**Java High Level REST Client**：该客户端基于 Low Level 客户端实现，提供 API 解决 Low Level 客户端需要手动转换数据格式的问题。

ES 的官网已经提供了非常详尽的 API 参考手册，参见

<https://www.elastic.co/guide/en/elasticsearch/client/java-rest/current/index.html>

# 使用

## Java Low Level REST Client

具体代码在模块 es-low-level 下

### Maven

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-client</artifactId>
    <version>7.7.0</version>
</dependency>
```

### 代码

因为 Java Low Level REST Client 用法比较原始，在实际工作中用的比较少，所以大概了解下用法即可，对应的类是 TestEsLowSdk：

#### 1、创建访问客户端

```
/* 初始化 RestClient，填写es服务器的hostName 和 port*/
RestClient restClient = RestClient.builder(new HttpHost(
    |     hostname: "120.24.108.136", port: 9200, scheme: "http")).build();

/* 或者使用RestClientBuilder也可以创建客户端
RestClientBuilder builder = RestClient.builder(
    new HttpHost("120.24.108.136", 9200, "http"));
RestClient restClient2 = builder.build();
*/
```

#### 2、配置访问 es 的请求

```
/*提供谓词和终节点以及可选查询字符串参数和
 / org.apache.http.HttpEntity对象中包含的请求主体来发送请求*/
Map<String, String> params = Collections.emptyMap();
String jsonString = "{" +
    "\"msg\":\"Java Low Level REST Client\""+
    "}";
System.out.println(jsonString);

/*设置Content-Type请求头,
 / 以便Elasticsearch可以正确解析内容。*/
HttpEntity entity = new NStringEntity(jsonString, ContentType.APPLICATION_JSON);
Request request = new Request(method: "PUT", endpoint: "/enjoy_test/_doc/5");
request.addParameters(params);
request.setEntity(entity);
```

#### 3、发送请求并接收应答

```
Response response = restClient.performRequest(request);
System.out.println(response);
System.out.println(EntityUtils.toString(response.getEntity()));
/*关闭客户端*/
restClient.close();
```

## Java High Level REST Client

具体代码在模块 es-high-level 下

### Maven

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-high-level-client</artifactId>
    <version>7.7.0</version>
</dependency>
```

### 代码

对应的类是 TestEsHighSdk

1、创建访问客户端

```
// 初始化 RestClientBuilder, 填写es服务器的hostName 和 port
RestClientBuilder builder =
    RestClient.builder(new HttpHost("120.24.108.136",
        port: 9200, scheme: "http"));

// 由Low Level Client构造High Level Client
RestHighLevelClient client = new RestHighLevelClient(builder);
```

2、创建索引

```
/*-----创建索引-----*/
CreateIndexRequest createIndexRequest = new CreateIndexRequest(indexName);

/*这里可以对索引进行配置
request.settings(Settings.builder()
    .put("index.number_of_shards", 3)
    .put("index.number_of_replicas", 2)
);
*/

/*这里可以对索引的字段进行配置
Map<String, Object> message = new HashMap<>();
message.put("type", "text");
Map<String, Object> properties = new HashMap<>();
properties.put("message", message);
Map<String, Object> mapping = new HashMap<>();
mapping.put("properties", properties);
request.mapping(mapping);
*/
/* 创建索引 */
CreateIndexResponse createIndexResponse =
    client.indices().create(createIndexRequest, RequestOptions.DEFAULT);

System.out.println(createIndexResponse.index());
System.out.println(createIndexResponse.isAcknowledged());
```

---

使用 CreateIndexRequest 进行索引创建，如果想要对索引进行静态配置，可以使用 request.settings。

配置索引的映射有好几种方式，比如我们想配置的索引映射是：

```
put / high_sdk/_mapping
{
  "properties" : {
    "message" : { "type" : "text" }
  }
}
```

可以采用：

```
request.mapping(
  "\n" +
  "  \"properties\": {\n" +
  "    \"message\": {\n" +
  "      \"type\": \"text\"\n" +
  "    }\n" +
  "  }\n",
  XContentType.JSON);
```

也可以采用：

```
Map<String, Object> message = new HashMap<>();
message.put("type", "text");
Map<String, Object> properties = new HashMap<>();
properties.put("message", message);
Map<String, Object> mapping = new HashMap<>();
mapping.put("properties", properties);
request.mapping(mapping);
```

还可以采用：

```
XContentBuilder builder = XContentFactory.jsonBuilder();
builder.startObject();
{
  builder.startObject("properties");
  {
    builder.startObject("message");
    {
      builder.field("type", "text");
    }
  }
}
```

```
        }
        builder.endObject();
    }
    builder.endObject();
}
builder.endObject();
request.mapping(builder);
```

然后通过 `client.indices().create` 方法将请求发送给 es 即可，es 的应答将通过 `CreateIndexResponse` 返回给我们。

### 3、索引(保存)文档

基本上和创建索引的思路是一样的

```
IndexRequest indexRequest = new IndexRequest(indexName);
indexRequest.id(docId);
```

使用 `IndexRequest` 进行文档的索引，相关的请求体可以通过多种方法生成

```
/*可以手动拼接请求体字符串
String jsonString = "{" +
    "\"user\":\"mark\"," +
    "\"postDate\":\"2013-01-30\"," +
    "\"message\":\"Go Elasticsearch\""+
    "}";
indexRequest.source(jsonString, XContentType.JSON);
*/
/* 也可以用map的方式
Map<String, Object> jsonMap = new HashMap<>();
jsonMap.put("user", "mark");
jsonMap.put("postDate", new Date());
jsonMap.put("message", "Go Elasticsearch");
indexRequest.source(jsonMap);
*/
XContentBuilder xContentBuilder = XContentFactory.jsonBuilder();
xContentBuilder.startObject();
{
    xContentBuilder.field(name: "user", value: "mark");
    xContentBuilder.timeField(name: "postDate", new Date());
    xContentBuilder.field(name: "message", value: "Go Elasticsearch");
}
xContentBuilder.endObject();
indexRequest.source(xContentBuilder);
```

然后通过 `client.index` 方法将请求发送给 es 即可，es 的应答将通过 `CreateIndexResponse` 返回给我们。

### 4、查询文档

使用 `GetRequest` 进行查询，通过 `client.get` 方法将请求发送给 es 即可，es 的应答将通过 `GetResponse` 返回给我们。

```
/*-----查询文档-----*/
GetRequest getRequest = new GetRequest(indexName,docId);
GetResponse getResponse = client.get(getRequest, RequestOptions.DEFAULT);
System.out.println("获取的文档所属索引是："+getResponse.getIndex());
System.out.println("获取的文档id是："+getResponse.getId());
if (getResponse.exists()) {
    System.out.println("获取的文档是："+getResponse.getSourceAsString());
    //Map<String, Object> sourceAsMap = getResponse.getSourceAsMap();
    //byte[] sourceAsBytes = getResponse.getSourceAsBytes();
} else {
    System.out.println("文档不存在！");
}
```

更多的与检索相关的类或者方法，我们将在和 `SpringBoot` 的集成这个章节看到。

## 和 `SpringBoot` 的集成

具体代码在模块 `springboot-highlevel` 下

### Maven

```
<dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch</artifactId>
    <version>7.7.0</version>
</dependency>

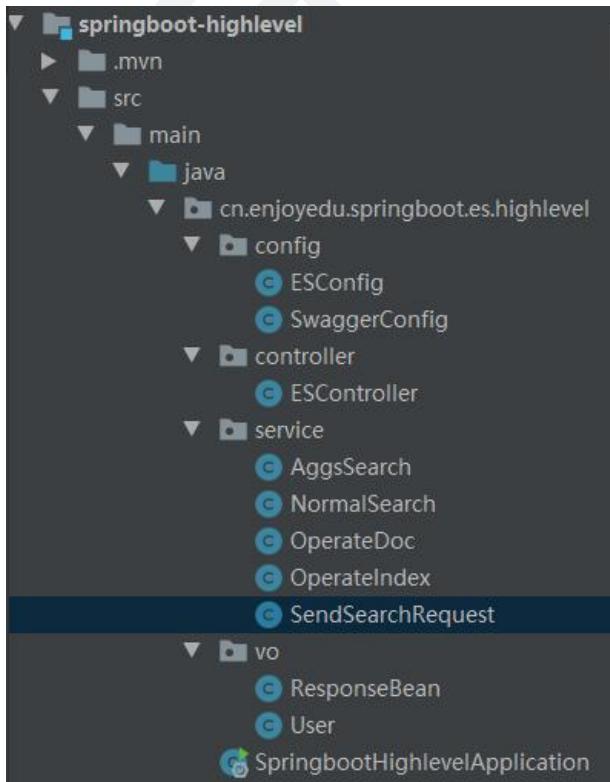
<!-- https://mvnrepository.com/artifact/org.elasticsearch.client/elasticsearch-rest-h
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-high-level-client</artifactId>
    <version>7.7.0</version>
</dependency>
```

为方便我们的使用，我们还会引入一些其他的组件，比如 `fastjson` 和 `Swagger2`

```
<!-- fastjson-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.54</version>
</dependency>
<!-- Swagger2 -->
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.7.0</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.7.0</version>
</dependency>
```

## 代码

代码目录结构如下：



比较关键的类是：

AggsSearch: 聚集搜索服务类

NormalSearch: 一般搜索服务类

OperateDoc: 操作文档的服务

OperateIndex: 操作索引的服务

SendSearchRequest: 通用的发送 Search 请求，并处理响应结果的服务。

其中主要包括了索引的管理、文档的管理、\_search 接口基本用法、基于词项的查询、基于全文的查询、基于全文的模糊查询、组合查询、聚集查询的相关范例，详情参阅相关代码。

## Elasticsearch 的集群

我们知道了 Elasticsearch 能够做什么，接下来我们将见识 Elasticsearch 另一个很强的----扩展能力，也就是，Elasticsearch 如何能够处理更多的索引和搜索请求，或者是更快地处理索引和搜索请求。在处理百万级甚至数十亿级的文档时，扩展性是一个非常重要的因素。没有了某种形式的扩展，在单一的 Elasticsearch 运行实例或节点( node)上就无法一直支持规模持续增大的流量。Elasticsearch 很容易扩展。所以我们来了解 Elasticsearch 所拥有的扩展能力，以及如何使用这些特性来给予 Elasticsearch 更多的性能、更多的可靠性。

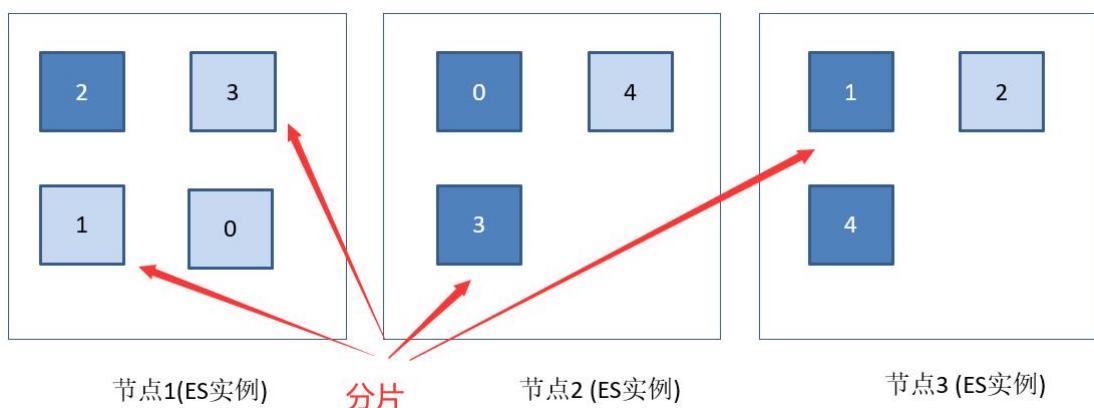
### 理解物理设计

#### 节点和分片

为了有个全局的理解，我们首先要知道 Elasticsearch 索引创建的时候，究竟发生了什么，理解数据在物理是如何上组织的？

默认情况下，每个索引由 5 个主要分片组成，而每份主要分片又有一个副本，一共 10 份分片。副本分片对于可靠性和搜索性能很有益处。技术上而言，一份分片是一个目录中的文件，Lucene 用这些文件存储索引数据。分片也是 Elasticsearch 将数据从一个节点迁移到另一个节点的最小单位。

一个节点是一个 Elasticsearch 的实例。在服务器上启动 Elasticsearch 之后，你就拥有了一个节点。如果在另一台服务器上启动 Elasticsearch，这就是另一个节点。甚至可以通过启动多个 Elasticsearch 进程，在同一台服务器上拥有多个节点。



---

多个节点可以加入同一个集群。在多节点的集群上，同样的数据可以在多台服务器上传播。这有助于性能，因为 Elasticsearch 有了更多的资源；这同样有助于稳定性，如果每份分片至少有 1 个副本分片，那么任何一个节点都可以宕机，而 Elasticsearch 依然可以进行服务，并返回所有数据。

对于使用 Elasticsearch 的应用程序，集群中有 1 个还是多个节点都是透明的。默认情况下，可以连接集群中的任一节点并访问完整的数据集，就好像集群只有单独的一个节点。

尽管集群对于性能和稳定性都有好处，但它也有缺点：必须确定节点之间能够足够快速地通信，并且不会产生脑裂。为了解决这个问题，我们后面会来讨论。

一份分片是 Lucene 的索引：一个包含倒排索引的文件目录。倒排索引的结构使得 Elasticsearch 在不扫描所有文档的情况下，就能告诉你哪些文档包含特定的词条( 单词)。

#### *Elasticsearch 索引和 Lucene 索引的对比：*

*Elasticsearch 索引被分解为多块: 分片。一份分片是一个 Lucene 的索引，所以一个 Elasticsearch 的索引由多个 Lucene 的索引组成。*

一个分片是一个 Lucene 索引（一个倒排索引）。它默认存储原始文档的内容，再加上一些额外的信息，如词条字典和词频，这些都能帮助到搜索。

词条字典将每个词条和包含该词条的文档映射起来。搜索的时候，Elasticsearch 没有必要为了某个词条扫描所有的文档，而是根据这个字典快速地识别匹配的文档。

词频使得 Elasticsearch 可以快速地获取某篇文档中某个词条出现的次数。这对于计算结果的相关性得分非常重要。例如，如果搜索 “denver”，包含多个 “denver” 的文档通常更为相关。Elasticsearch 将给它们更高的得分，让它们出现在结果列表的更前面。

接下来看看主分片和副本分片的细节，以及它们是如何在 Elasticsearch 集群中分配的。

## 附：ES 中的节点类型

### **Master-eligible nodes 与 Master node**

每个节点启动后，默认就是一个 *Master eligible* 节点，可以通过设置 *node.master:false* 来改变，*Master-eligible* 节点可以参加选主流程，成为 *Master* 节点，每个节点都保存了集群的状态，但只有 *Master* 节点才能修改集群的状态信息，主节点主要负责集群方面的轻量级的动作，比如：创建或删除索引，跟踪集群中的节点，决定分片分配到哪一个节点，在集群再平衡的过程中，如何在节点间移动数据等。

### **Data Node**

可以保存数据的节点，叫做 *Data Node*。负责保存分片数据。在数据扩展上起到了至关重要的作用，每个节点启动后，默认就是一个 *Data Node* 节点，可以通过设置 *node.data:false* 来改变。

### **Ingest Node**

---

可以在文档建立索引之前设置一些 *ingest pipeline* 的预处理逻辑，来丰富和转换文档。每个节点默认启动就是 *Ingest Node*，可用通过 *node.ingest = false* 来禁用。

#### **Coordinating Node**

*Coordinating Node* 负责接收 *Client* 的请求，将请求分发到合适的节点，最终把结果汇集到一起，每个节点默认都起到了 *Coordinating Node* 的职责，当然如果把 *master*、*Data*、*Ingest* 全部禁用，那这个节点就仅是 *Coordinating Node* 节点了。

#### **Machine Learning Node**

用于机器学习处理的节点

## **主分片和副本分片**

分片可以是主分片，也可以是副本分片，其中副本分片是主分片的完整副本，副本分片可以用于搜索，或者是在原有主分片丢失后成为新的主分片。主分片是权威数据，写过程先写主分片，成功后再写副本分片。

*Elasticsearch* 索引由一个或多个主分片以及零个或多个副本分片构成。副本分片可以在运行的时候进行添加和移除，而主分片不可以。

可以在任何时候改变每个分片的副本分片的数量，因为副本分片总是可以被创建和移除。这并不适用于索引划分为主分片的数量，在创建索引之前，你必须决定主分片的数量。请记住，过少的分片将限制可扩展性，但是过多的分片会影响性能。默认设置主分片的数量是 5 份。

#### **复习**

索引主分片的设置：

```
put test1{
  "settings":{
    "index.number_of_shards":3,
    "index.codec":"best_compression"
  }
}
```

索引副本分片的设置：

```
put test1/_settings
{
  "index.number_of_replicas":2
}
```

## **在集群中分发分片**

最简单的 *Elasticsearch* 集群只有一个节点：一台机器运行着一个 *Elasticsearch* 进程。我们安装并启动了 *Elasticsearch* 之后，就已经建立了一个拥有单节点的集群。随着越来越多的节点被添加到同一个集群中，现有的分片将在

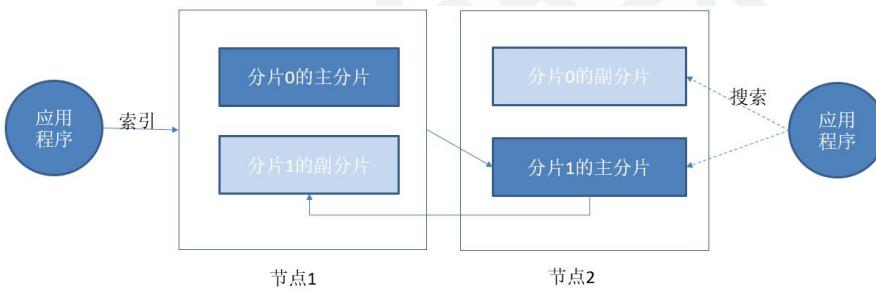
---

所有的节点中自动进行负载均衡。因此，在那些分片上的索引和搜索请求都可以从额外增加的节点中获益。以这种方式进行扩展（在节点中加入更多节点）被称为水平扩展。此方式增加更多节点，然后请求被分发到这些节点上，工作负载就被分摊了。

水平扩展的另一个替代方案是垂直扩展，这种方式为 Elasticsearch 的节点增加更多硬件资源，可能是为虚拟机分配更多处理器，或是为物理机增加更多的内存。尽管垂直扩展几乎每次都能提升性能，它并非总是可行的或经济的。

## 分布式索引和搜索

在多个节点的多个分片上是如何进行索引和搜索的呢？



### 当索引一篇文档时

默认情况下，当索引一篇文档的时候，系统首先根据文档 ID 的散列值选择一个主分片，并将文档发送到该主分片。这份主分片可能位于另一个节点，不过对于应用程序这一点是透明的。

默认地，文档在分片中均匀分布：对于每篇文档，分片是通过其 ID 字符串的散列决定的。每份分片拥有相同的散列范围，接收新文档的机会均等。

一旦目标主分片确定，接受请求的节点将文档转发到该主分片所在的节点。随后，索引操作在该主分片的所有副本分片中进行。在所有可用副本分片完成文档的索引后，索引命令就会成功返回。

这使得副本分片和主分片之间保持数据的同步。数据同步使得副本分片可以服务于搜索请求，并在原有主分片无法访问时自动升级为主分片。

### 搜索索引时

当搜索一个索引时，Elasticsearch 需要在该索引的完整分片集合中进行查找。这些分片可以是主分片，也可以是副本分片，原因是对应的主分片和副本分片通常包含一样的文档。Elasticsearch 在索引的主分片和副本分片中进行搜索请求的负载均衡，使得副本分片对于搜索性能和容错都有所帮助。

在搜索的时候，接受请求的节点将请求转发到一组包含所有数据的分片。Elasticsearch 使用 round-robin 的轮询机制选择可用的分片（主分片或副本分片），并将搜索请求转发过去，Elasticsearch 然后从这些分片收集结果，将其聚集为单一的回复，然后将回复返回给客户端应用程序。在默认情况下，搜索请求通过 round-robin 轮询机制选中主分片和副本分片。

# 向集群中加入节点

## 单机集群（伪集群）

创建 Elasticsearch 集群的第一步，是为单个节点加入另一个节点(或多个节点)，组成节点的集群。

没有加入第二个节点前：

[http://xxxxxx:9200/\\_cluster/state/master\\_node,nodes?pretty](http://xxxxxx:9200/_cluster/state/master_node,nodes?pretty)



The screenshot shows a browser window with the URL `http://xxxxxx:9200/_cluster/state/master_node,nodes?pretty`. The page displays a JSON response representing a single-node cluster. The response includes fields like `cluster_name`, `cluster_uuid`, `master_node`, and a `nodes` array containing one node object. The node object has fields such as `name`, `ephemeral_id`, `transport_address`, and `attributes` (including `ml.machine_memory`, `xpack.installed`, `transform.node`, and `ml.max_open_jobs`). The JSON is formatted with indentation and line breaks for readability.

```
{
  "cluster_name": "my-elk",
  "cluster_uuid": "fLDRp_X7T1G3SnICdJyISw",
  "master_node": "Xx0gEUU1QJiHEJRafCK1bQ",
  "nodes": [
    {
      "Xx0gEUU1QJiHEJRafCK1bQ": {
        "name": "node-1",
        "ephemeral_id": "C9vFB7PhTW2dd2Ub5f3cHA",
        "transport_address": "172.18.194.140:9300",
        "attributes": {
          "ml.machine_memory": "8051073024",
          "xpack.installed": "true",
          "transform.node": "true",
          "ml.max_open_jobs": "20"
        }
      }
    }
  ]
}
```

如何加入第二个节点：

1、解压缩压缩包 `elasticsearch-7.7.0-linux-x86_64.tar.gz` 到另外一个目录，比如 `elasticsearch-2`，并且保证这个 `es` 中不包含任何数据；

2、修改配置文件

```
cluster.name: my-elk,
```

将 `cluster.name` 改为和第一个节点一样；

```
node.name: node-2,
```

给第二个节点配置独立的名字；

```
network.host: 172.18.194.140
```

`network.host` 改为本机地址

```
# Pass an initial list of hosts to perform
# The default list of hosts is ["127.0.0.1"]
#
discovery.seed_hosts: ["172.18.194.140"],
```

将 `discovery.seed_hosts` 改为本机地址

如果第一个节点有插件，则也需要安装同样的插件。

然后启动第二个节点即可。

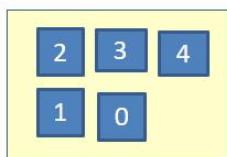
现在有了第二个 Elasticsearch 节点加入了集群，可以再次

`http://xxxxxx:9200/_cluster/state/master_node,nodes?pretty`

```
{  
  "cluster_name" : "my-elk",  
  "cluster_uuid" : "fLDRp_X7T1G3SnICdJyISw",  
  "master_node" : "Xx0gEUU1QJiHEJRafCK1bQ",  
  "nodes" : [  
    "L41_c_RsRHqcj1_h-H_2Dw" : {  
      "name" : "node-2",  
      "ephemeral_id" : "fAecYTz-RECqK9DZPgMHQA",  
      "transport_address" : "172.18.194.140:9301",  
      "attributes" : {  
        "ml.machine_memory" : "8051073024",  
        "ml.max_open_jobs" : "20",  
        "xpack.installed" : "true",  
        "transform.node" : "true"  
      }  
    },  
    "Xx0gEUU1QJiHEJRafCK1bQ" : {  
      "name" : "node-1",  
      "ephemeral_id" : "C9vFB7PhTW2dd2Ub5f3cHA",  
      "transport_address" : "172.18.194.140:9300",  
      "attributes" : {  
        "ml.machine_memory" : "8051073024",  
        "xpack.installed" : "true",  
        "transform.node" : "true",  
        "ml.max_open_jobs" : "20"  
      }  
    }  
  ]  
}
```

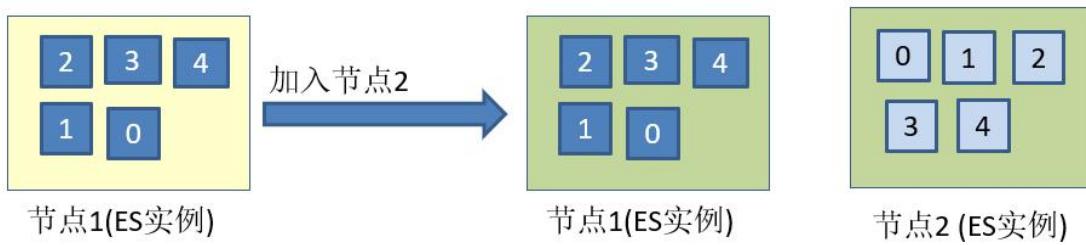
## 新增节点上的分片是如何运作

那么新增节点上的分片是如何运作的呢？看看向集群增加一个节点前后，索引发生了些什么。在左端，`test` 索引的主分片全部分配到节点 `Node1`，而副本分片“分配没有地方分配”。在这种状态下，集群是黄色的，因为所有的主分片有了安家之处，但是副本分片还没有。

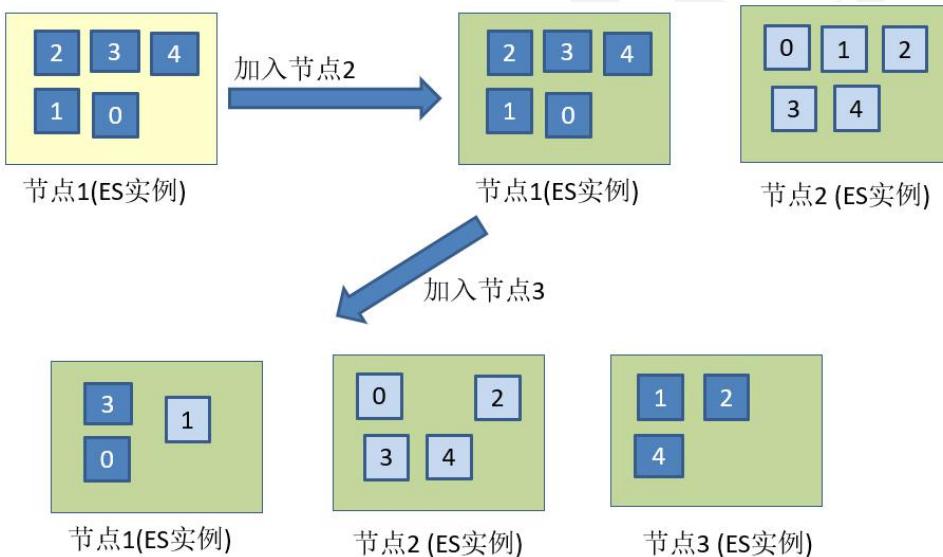


节点1(ES实例)

一旦第二个节点加入，尚未分配的副本分片就会分配到新的节点 `Node2`，这使得集群变为了绿色的状态。



当另一个节点加入的时候，Elasticsearch 会自动地尝试将分片在所有节点上进行均匀分配。



如果更多的节点加入集群，Elasticsearch 将试图在所有的节点上均匀地配置分片数量，这样每个新加入的节点都能够通过部分数据(以分片的形式)来分担负载。

将节点加入 Elasticsearch 集群带来了大量的好处，主要的收益是高可用性和提升的性能。当副本分片是激活状态时(默认情况下是激活的),如果无法找到主分片, Elasticsearch 会自动地将一个对应的副本分片升级为主分片。这样，即使失去了索引主分片所在的节点，仍然可以访问副本分片上的数据。数据分布在多个节点上同样提升了性能，原因是主分片和副本分片都可以处理搜索和获取结果的请求。如此扩展还为整体集群增加了更多的内存，所以如果过于消耗内存的搜索和聚集运行了太长时间或致使集群耗尽了内存，那么加入更多的节点总是一个处理更多更复杂操作的便捷方式。

## 发现其他 ES 节点

集群的第二个节点是如何发现第一个节点、并自动地加入集群的，或者在集群中有更多的节点的情况下，如何知道？

Elasticsearch 7.0 中引入的新集群协调子系统来处理这些事，采用的是单播机制，这种机制需要已知节点的列表来进行连接。

---

单播发现( unicast discovery )让 Elasticsearch 连接一系列的主机，并试图发现更多关于集群的信息。使用单播时，我们告诉 Elasticsearch 集群中其他节点的 IP 地址以及(可选的)端口或端口范围。

在 `elasticsearch.yml` 中通过 `discovery.seed_hosts` 配置种子地址列表，这样每个节点在启动时发现和加入集群的步骤就是：

- 1、去连接种子地址列表中的主机，如果发现某一个 Node 是 Master Eligible Node，那么该 Master Eligible Node 会共享它知道的 Master Eligible Node，这些共享的 Master Eligible Node 也会作为种子地址的一部分继续去试探；
- 2、直到找到某一个 seed address 对应的是 Master Node 为止；
- 3、如果第二步没有找到任何满足条件的 Node，ES 会默认每隔 1 秒后去重新尝试寻找，默认为 1 秒
- 4、重复第三步操作直到找到满足条件为止，也就是直到最终发现集群中的主节点，会发出一个加入请求给主节点
- 5、获得整个集群的状态信息。

为什么实例需要知道集群状态信息？例如，搜索必须被路由到所有正确的分片，以确保搜索结果是准确的。在索引或删除某些文档时，必须更新每个副本。每个客户端请求都必须从接收它的节点转发到能够处理它的节点。每个节点都了解集群的概况，这样它们就可以执行搜索、索引和其他协调活动。

`discovery.seed_hosts` 中的节点地址列表，可以包括集群中部分或者全部集群节点，但是建议无论怎样都应该包含集群中 `Master-eligible nodes` 节点的部分或者全部。

## 选举主节点

一旦集群中的节点发现了彼此，它们会协商谁将成为主节点。一个集群有一个稳定的主节点是非常重要的，主节点是唯一一个能够更新集群状态的节点。主节点一次处理一个群集状态更新，应用所需的更改并将更新的群集状态发布到群集中的所有其他节点。

Elasticsearch 认为所有的节点都有资格成为主节点，除非某个节点的 `node.master` 选项设置为 `false`，而 `node.master` 在不做配置的情况下，缺省为 `true`。

如果完全使用默认配置启动新安装的 Elasticsearch 节点，它们会自动查找在同一主机上运行的其他节点，并在几秒钟内形成集群。在生产环境或其他分布式环境中还不够健壮。现在还存在一些风险：节点可能无法及时发现彼此，可能会形成两个或多个独立的集群。从 Elasticsearch 7.0 开始，如果你想要启动一个全新的集群，并且集群在多台主机上都有节点，那么你必须指定该集群在第一次选举中应该使用的一组符合主节点条件的节点作为选举配置。这就是所谓的集群引导，也可以称为集群自举，只在首次形成集群时才需要。

`cluster.initial_master_nodes` 这个参数就是用来设置一系列符合主节点条件的节点的主机名或 IP 地址来进行集群自举。集群形成后，不再需要此设置，并且会忽略它，也就是说，这个属性就只是在集群首次启动时有用。

---

在向集群添加新的符合主节点条件的节点时不再需要任何特殊的仪式，只需配置新节点，让它们可以发现已有集群，并启动它们。当有新节点加入时，集群将会自动地调整选举配置。

在主节点被选举出来之后，它会建立内部的 ping 机制来确保每个节点在集群中保持活跃和健康，这被称为错误识别( **fault detection**), 有两个故障检测进程在集群的生命周期中一直运行。一个是主节点的，ping 集群中所有的其他节点，检查他们是否活着。另一种是每个节点都 ping 主节点，确认主节点是否仍在运行或者是否需要重新启动选举程序。

在 Elasticsearch7 以前的版本中，为了预防集群产生脑裂( **split brain**)的问题，Elasticsearch 6.x 及之前的版本使用了一个叫作 **Zen Discovery** 的集群协调子系统。往往会将 `discovery.zen.minimum_master_nodes` 设置为集群节点数除以 2 再加上 1。例如，3 个节点的集群 `discovery.zen.minimum_master_nodes` 要设置为 2，而对于 14 个节点的集群，最好将其设置为 8。

在 Elasticsearch 7 以后里重新设计并重建了的集群协调子系统，移除 `minimum_master_nodes` 参数，转而由集群自主控制。

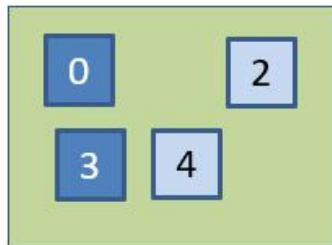
#### 什么是 Elasticsearch 的脑裂？

脑裂这个词描述了这样的场景：(通常是在重负荷或网络存在问题的情况下)Elasticsearch 集群中一个或多个节点失去了和主节点的通信，开始选举新的主节点，并且继续处理请求。这个时候，可能有两个不同的 Elasticsearch 集群相互独立地运行着，这就是“脑裂”一词的由来，因为单一的集群已经分裂成了两个不同的部分，和左右大脑类似。为了防止这种情况的发生，Elasticsearch 7 以前版本你需要根据集群节点的数量来设置 `discovery.zen.minimum_master_nodes`。如果节点的数量不变，将其设置为集群节点的总数；否则将节点数除以 2 并加 1 是一个不错的选择，因为这就意味着如果一个或多个节点失去了和其他节点的通信，它们无法选举新的主节点来形成新集群，因为对于它们不能获得所需的节点(可成为主节点的节点)数量(超过一半)。

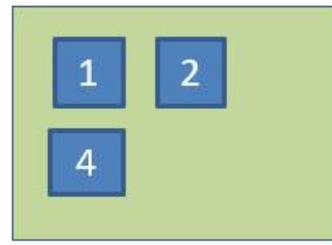
## 删除集群中的节点

添加节点是扩展的好方法，但是如果 Elasticsearch 集群中的一个节点掉线了或者被停机了，那又会发生什么呢？这里使用图 9-2 中 3 个节点的集群为例，其中包含了测试的索引，5 个主分片和每个主分片对应的 1 个副本分片都分布在这 3 个节点上。

我们假设节点 1 宕机了，那么在节点 1 的 3 个分片怎么办？Elasticsearch 所做的第 1 件事情是自动地将节点 2 上的 0 和 3 副本分片转为主分片。



节点2 (ES实例)

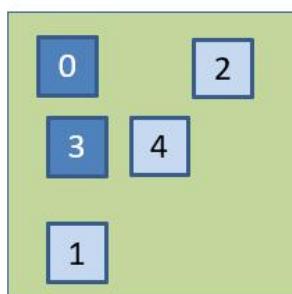


节点3 (ES实例)

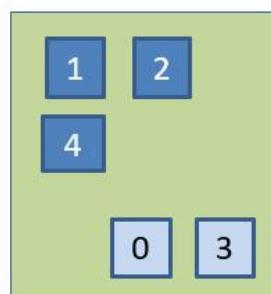
这是由于索引操作会首先更新主分片，所以 Elasticsearch 要尽力使索引的主分片正常运作。

注意 Elasticsearch 可以选择任一个副本分片并将其转变为主分片。只是在本例中每个主分片仅有一个副本分片可供选择，就是节点 Node2 上的副本分片。

副本分片变为主分片之后，集群就会变为黄色的状态，这意味着某些副本分片尚未分配到某个节点。Elasticsearch 下一步需要创建更多的副本分片来保持索引的高可用性。由于所有的主分片现在都是可用的，节点 2 上 0 和 3 主分片的数据会复制到节点 3 上作为副本分片，而节点 3 上 1 主分片的数据会复制到节点 2。



节点2 (ES实例)



节点3 (ES实例)

一旦副本分片被重新创建，并用于弥补损失的节点，那么集群将重新回归绿色的状态，全部的主分片及其副本分片都分配到了某个节点。请记住，在这个时间段内，整个集群都是可用于搜索和索引的，因为实际上没有丢失数据。

如果失去的节点多于 1 个，或者某个没有副本的主分片丢失了，那么集群就会变为红色的状态，这意味着某些数据永远地丢失了，你需要让集群重连拥有丢失数据的节点，或者对丢失的数据重新建立索引。

就副本分片的数量而言，你需要理解自己愿意承担多少风险，这一点非常重要。有 1 份副本分片意味着集群可以缺失 1 个节点而不丢失数据。如果有 2 个副本分片，可以缺失 2 个节点而不丢失数据，以此类推。所以，确保你选择了合适的副本数量。备份你的索引永远是个不错的主意。

## 停用节点

当节点宕机时，让 Elasticsearch 自动地创建新副本分片是很棒的选择。可是，当集群进行例行维护的时候，你总是希望关闭某个包含数据的节点，而同时不让集群进入黄色的状态。也许硬件过于老旧，或者处理的请求流量有所下降，总之

---

你不再需要这么多节点了。可以通过杀掉 Java 进程来停止节点，然后让 Elasticsearch 将数据恢复到其他节点，但是如果你的索引没有副本分片的时候怎么办？这意味着，如果不预先将数据转移，关闭节点就会让你丢失数据！

值得庆幸的是，Elasticsearch 有一种停用节点(`decommission`)的方式，告诉集群不要再分配任何分片到某个或 1 组节点上。在 3 个节点的例子中，假设节点 1、节点 2 和节点 3 的 IP 地址分别是 192.168.1.10、192.168.1.11 和 192.168.1.12。如果你想关闭节点 1 的同时保持集群为绿色状态，可以先停用节点，这个操作会将待停用节点上的所有分片转移到集群中的其他节点。系统通过集群设置的临时修改，来为你实现节点的停用，

```
PUT _cluster/settings
{
  "transient": {
    "cluster.routing.allocation.exclude._name": "node-1"
  }
}
```

或者

```
PUT _cluster/settings
{
  "transient": {
    "cluster.routing.allocation.exclude._ip": "192.168.1.10"
  }
}
```

一旦运行了这个命令，Elasticsearch 将待停用节点上的全部分片开始转移到集群中的其他节点上。

该过程可以重复，每次停止一个你想关闭的节点，或者也可以使用一个通过逗号分隔的 IP 地址列表，一次停止多个节点。请记住，集群中的其他节点必须有足够的磁盘和内存来处理分片的分配，所以在停止多个节点之前，做出相应的计划来确保你有足够的资源。

## 扩展策略

将节点加入集群以增加性能，看上去很简单，但是稍微做些计划会使得你在获取集群最佳性能的这条道路上走得更远。

Elasticsearch 的使用方式各有各的不同，所以需要根据如何索引和搜索数据，为集群选择最佳的配置。通常来说，规划生产环境的 Elasticsearch 集群至少需要考虑：过度分片、将数据切分为索引和分片。

---

## 过度分片

让我们从过度分片开始说起。过度分片( over-sharding )是指你有意地为索引创建大量分片，用于未来增加节点的过程。假设我们已经创建了拥有单一分片、无副本分片的索引。但是，在增加了另外一个节点之后又会发生什么？

我们将得不到增加集群节点所带来的任何好处了。由于全部的索引和查询负载仍然是由拥有单一分片的节点所处理，所以即使增加了一个节点你也无法进行扩展。

因为分片是 Elasticsearch 所能移动的最小单位，所以确保你至少拥有和集群节点一样多的主分片总是个好主意。如果现在有一个 5 个节点、11 个主分片的集群，那么当你需要加入更多的节点来处理额外的请求时，就有成长的空间。使用同样的例子，如果你突然需要多于 11 个的节点，就不能在所有的节点中分发主分片，因为节点的数量将会超出分片的数量。

怎么办？创建一个有 10000 个主分片的索引？一开始的时候，这看上去是个好主意，但是 Elasticsearch 管理每个分片都隐含着额外的开销。每个分片都是完整的 Lucene 索引，它需要为索引的每个分段创建一些文件描述符，增加相应的内存开销。如果索引有过多的活跃分片，可能会占用了本来支撑性能的内存，或者触及机器文件描述符或内存的极限。对数据的压缩上也会有影响。

值得注意的是，没有对所有案例适用的完美分片索引比例。Elasticsearch 选择的默认设置是 5 个分片，对于普通的用例是不错的主意，但是考虑你的规划在将来是如何增长(或缩减)所建分片的数量，这总是很件重要的事情。

不要忘记：一旦包含某些数量分片的索引被创建，其主分片的数量永远是不能改变的。

### Elasticsearch 索引能处理多大的数据

单一索引的极限取决于存储索引的机器之类型、你准备如何处理数据以及索引备份了多少副本。

如何评估 ES 中的数据量是否合适呢？有几个参考值：

1、ES 官方推荐分片的大小是 **20G - 40G**，最大不能超过 **50G**。  
2、每个节点上可以存储的分片数量与可用的堆内存大小成正比关系，但是 Elasticsearch 并未强制规定固定限值。这里有一个很好的经验法则：确保对于节点上已配置的每个 **GB**，将分片数量保持在 **20** 以下。如果某个节点拥有 **3GB** 的堆内存，那最多可有 **60** 个分片，那么有三个机器的集群，ES 可用总堆内存是 **9GB**，则最多是 **180** 个分片，注意这个数据是包含了主副分片的。但是在此限值范围内，设置的分片数量越少，效果就越好。

3、通常来说，一个 Lucene 索引(也就是一个 Elasticsearch 分片)不能处理多于 **21** 亿篇文档，或者多于 **2740** 亿的唯一词条，但是在达到这个极限之前，你可能就已经没有足够的磁盘空间了。

举例：三个机器的集群，总内存是 **9GB**，准备 **1 主 2 副**，支持的总主分片数量最大不宜超过 **60** 个分片。

## 将数据切分为索引和分片

现在还没有方法让我们增加或者减少某个索引中的主分片数量,但是你总是可以对数据进行规划,让其横跨多个索引。这是另一种完全合理的切分数据的方式。

比如说以地理位置创建索引和分片,你可以为西藏索引创建 2 个主分片,而为上海索引创建 10 个主分片,或者可以将数据以日期来分段,为数据按年份创建索引: 2019、 2020 和 2021 等。以这种方式将数据分段,对于搜索同样有所帮助,因为分段将恰当的数据放在恰当的位置。如果顾客只希望搜索 2019 年和 2020 年的活动或分组,你只需要搜索相应的索引,而不是整个数据集中检索。

使用索引进行规划的另一种方式是别名。别名( alias )就像指向某个索引或一组索引的指针。别名也允许你随时修改其所指向的索引。对于数据按照语义的方式来切分,这一点非常有用。你可以创建一个别名称为去年,指向 2019,当 2021 年 1 月 1 日到来,就可以将这个别名指向 2020 年的索引。

当索引基于日期的信息时(就像日志文件),这项技术是很常用的,如此一来数据就可以按照每月、每周、每日等日期来分段,而每次分段过时的时候,“当前”的别名永远可用来指向应该被搜索的数据,而无须修改待搜索的索引之名称。此外,别名拥有惊人的灵活性,而且几乎没有额外负载,所以值得尝试。

## 检查集群的健康状态

### 常用

集群的健康 API 接口提供了一个方便但略有粗糙的概览,包括集群、索引和分片的整体健康状况。这通常是发现和诊断集群中常见问题的第一步。

```
curl -XGET http://xxxxxx:9200/_cluster/health?pretty
```

从这个答复的表明信息,我们可以推断出很多关于集群整体健康状态的信息。

```
{
  "cluster_name": "my-elk",
  "status": "green",
  "timed_out": false,
  "number_of_nodes": 2,
  "number_of_data_nodes": 2,
  "active_primary_shards": 17,
  "active_shards": 34,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 0,
  "delayed_unassigned_shards": 0,
  "number_of_pending_tasks": 0,
  "number_of_in_flight_fetch": 0,
  "task_max_waiting_in_queue_millis": 0,
  "active_shards_percent_as_number": 100.0
}
```

我们一般比较关注的是:

---

**"cluster\_name"** :

集群名称

**"status"** :

正常情况下，Elasticsearch 集群健康状态分为三种：

**green** 最健康得状态，说明所有的分片包括备份都可用；这种情况 Elasticsearch 集群所有的主分片和副本分片都已分配，Elasticsearch 集群是 100% 可用的。

**yellow** 基本的分片可用，但是备份不可用（或者是没有备份）；这种情况 Elasticsearch 集群所有的主分片已经分片了，但至少还有一个副本是缺失的。不会有数据丢失，所以搜索结果依然是完整的。不过，你的高可用性在某种程度上被弱化。如果更多的分片消失，你就会丢数据了。把 yellow 想象成一个需要及时调查的警告。

**red** 部分的分片可用，表明分片有一部分损坏。此时执行查询部分数据仍然可以查到，遇到这种情况，还是赶快解决比较好；这种情况 Elasticsearch 集群至少一个主分片（以及它的全部副本）都在缺失中。这意味着你在缺少数据：搜索只能返回部分数据，而分配到这个分片上的写入请求会返回一个异常。

**"timed\_out"** :

是否有超时

**"number\_of\_nodes"** :

集群中的节点数量

**"number\_of\_data\_nodes"** :

数据节点数

**"active\_primary\_shards"** :

指出你集群中的主分片数量

**"active\_shards"** :

所有索引的\_所有\_分片的汇总值，即包括副本分片。

**"relocating\_shards"** :

大于 0 表示 Elasticsearch 正在集群内移动数据的分片，来提升负载均衡和故障转移。这通常发生在添加新节点、重启失效的节点或者删除节点的时候，因此出现了这种临时的现象。

**"initializing\_shards"** :

当用户刚刚创建一个新的索引或者重启一个节点的时候，这个数值会大于 0

**"unassigned\_shards"** :

这个值大于 0 的最常见原因是有尚未分配的副本分片。在开发环境中，这个问题很普遍，因为单节点的集群其索引默认有 5 个分片和 1 个副本分片。这种情况下，由于无多余节点来分配副本分片，因此还有 5 个未分配的副本分片。

**"active\_shards\_percent\_as\_number"** :

集群分片的可用性百分比，如果为 0 则表示不可用。

---

集群健康 API 提供了更多的细粒度的操作，允许用户进一步地诊断问题。在这个例子中，可以通过添加 `level` 参数，深入了解哪些索引受到了分片未配置的影响。

比如：

[http://xxxxxx:9200/\\_cluster/health?level=indices&pretty=true](http://xxxxxx:9200/_cluster/health?level=indices&pretty=true)

除此之外与集群相关的 API 还有：

查看集群健康状态接口(`_cluster/health`)

查看集群状况接口(`_cluster/state`)

查看集群统计信息接口(`_cluster/stats`)

查看集群挂起的任务接口(`_cluster/pending_tasks`)

集群重新路由操作(`_cluster/reroute`)

更新集群设置(`_cluster/settings`)

节点状态(`_nodes/stats`)

节点信息(`_nodes`)

节点的热线程(`_nodes/hot_threads`)

关闭节点(`_nodes/_master/_shutdown`)

## 使用`_cat` API

有的时候可读性很重要，我们就需要更方便的`_cat` API 接口。这个`_cat` API 提供了很有帮助的诊断和调试工具，将数据以更好的可读性打印出来。

`_cat` API 有很多特性，它们对于调试集群的各个方面都是很有帮助的。你可以运行 `curl 'localhost:9200/_cat'` 来查看所支持的`_cat` API 接口的完整清单。

`allocation`-展示分配到每个节点的分片数量。

`count`-统计整个集群或索引中文档的数量。

如： `GET _cat/count/employees?v`

`health`--展示集群的健康状态。

`indices`-展示现有索引的信息。

`master`--显示目前被选为主节点的节点。

`nodes`--显示集群中所有节点的不同信息。

`recovery`--显示集群中正在进行的分片恢复状态。

`shards`-展示集群中分片的数量、大小和名字。

`plugins`--展示已安装插件的信息。

## 路由

我们已经知道了文档是如何通过分片形式来定位的。这个过程被称为路由 (`routing`) 文档。为了让你更好地回忆，这里重温一下： 当 Elasticsearch 散列文

---

档的 ID 时就会发生文档的路由，来决定文档应该索引到哪个分片中，这可以由你指定也可以让 Elasticsearch 生成。

但是这个路由完全可以我们手动指定。

索引的时候，Elasticsearch 也允许你手动地指定文档的路由，使用父子关系实际上就是这种操作，因为子文档必须要和父文档在同一个分片。

## 为什么使用路由

假设你有一个 100 个分片的索引。当一个请求在集群上执行时会发生什么呢？

1. 这个搜索的请求会被发送到一个节点
2. 接收到这个请求的节点，将这个查询转到这个索引的每个分片上（可能是主分片，也可能是副本分片）
3. 每个分片执行这个搜索查询并返回结果
4. 结果在通道节点上合并、排序并返回给用户

因为默认情况下，Elasticsearch 使用文档的 ID（类似于关系数据库中的自增 ID），如果插入数据量比较大，文档会平均的分布于所有的分片上，这导致了 Elasticsearch 不能确定文档的位置，

所以它必须将这个请求转到所有的 N 个分片上去执行。这种操作会给集群带来负担，增大了网络的开销；

那么如何确定请求在哪个分片上执行呢？Elasticsearch 提供了一个 API 接口，告诉我们一个搜索请求在哪些节点和分片上执行。

### [search\\_shards API](#)

比如我们创建一个两分片的索引：

```
put open-soft-shard
{
  "settings":{
    "index.number_of_shards":2
  }
}
```

并放入数据：

```
put /open-soft-shard/_doc/1
{
  "name": "Apache Hadoop",
  "lang": "Java",
  "corp": "Apache",
  "stars":200
}
```

---

```
put /open-soft-shard/_doc/2
{
  "name": ["Apache Activemq", "Activemq Artemis"],
  "lang": "Java",
  "corp": "Apache",
  "stars": [500, 200]
}

put /open-soft-shard/_doc/3
{
  "name": ["Apache Kafka"],
  "lang": "Java",
  "corp": "Apache",
  "stars": [500, 400]
}

put /open-soft-shard/_doc/object
{
  "name": ["Apache ShardingSphere"],
  "lang": "Java",
  "corp": "JingDong",
  "stars": 400,
  "address": {
    "city": "BeiJing",
    "country": "亦庄"
  }
}
```

使用搜索分片( `search shards` ) 的 API 接口来查看请求将在哪些分片上执行。

```
GET /open-soft-shard/_search_shards?pretty
```

可以看到，会搜索全部的两个分片。

当我们

```
GET /open-soft-shard/_search_shards?pretty&routing=1
```

即使在索引中只有两个分片，当指定路由值 1 的时候，只有分片 `shard 0` 会被搜索。对于搜索需要查找的数据，有效地切除了一半的数据量！

所以当处理拥有大量分片的索引时，路由会很有价值，当然对于 Elasticsearch 的常规使用它并不是必需的。

## 配置路由

路由也可以不使用文档的 ID，而是定制的数值进行散列。通过指定 URL 中的 `routing` 查询参数，系统将使用这个值进行散列，而不是 ID。

```
PUT /employees/_doc/2?routing=rountkey
{
  .....
}
```

在这个例子中，`rountkey` 这个由我们自己输入的值决定文档属于哪个分片的散列值，而不是文档的 ID 值 2。

由上可知，自定义路由的方式非常简单，只需要在插入数据的时候指定路由的 `key` 即可。虽然使用简单，但有细节需要注意。我们来看看：

1、先创建一个名为 `study_route` 的索引，该索引有 2 个 shard，0 个副本

```
PUT study_route/
{
  "settings": {
    "number_of_shards": 2,
    "number_of_replicas": 0
  }
}
```

2：查看 shard

```
GET _cat/shards/study_route?v
```

index	shard	prirep	state	docs	store	ip	node
study_route	1	p	STARTED	0	208b	172.18.194.140	node-1
study_route	0	p	STARTED	0	208b	172.18.194.140	node-2

3：插入第 1 条数据

```
PUT study_route/_doc/a?refresh
{
  "data": "A"
}
```

4：查看 shard

```
GET _cat/shards/study_route?v
```

index	shard	prirep	state	docs	store	ip	node
study_route	1	p	STARTED	0	208b	172.18.194.140	node-1
study_route	0	p	STARTED	1	3.4kb	172.18.194.140	node-2

5: 插入第 2 条数据

```
PUT study_route/_doc/b?refresh
{
    "data": "B"
}
```

6: 查看 shard

```
GET _cat/shards/study_route?v
```

index	shard	prirep	state	docs	store	ip	node
study_route	1	p	STARTED	1	3.4kb	172.18.194.140	node-1
study_route	0	p	STARTED	1	3.4kb	172.18.194.140	node-2

7: 查看此时索引里面的数据

```
GET study_route/_search
```

这个例子比较简单，先创建了一个拥有 2 个 shard, 0 个副本（为了方便观察）的索引 `study_route`。创建完之后查看两个 shard 的信息，此时 shard 为空，里面没有任何文档（`docs` 列为 0）。接着我们插入了两条数据，每次插完之后，都检查 shard 的变化。通过对比可以发现 `docid=a` 的第一条数据写入了 0 号 shard, `docid=b` 的第二条数据写入了 1 号 shard。

接着，我们指定 `routing`, 看看有什么变化。

8: 插入第 3 条数据

```
PUT study_route/_doc/c?routing=key1&refresh
{
    "data": "C"
}
```

9: 查看 shard

```
GET _cat/shards/study_route?v
```

index	shard	prirep	state	docs	store	ip	node
study_route	1	p	STARTED	1	3.5kb	172.18.194.140	node-1
study_route	0	p	STARTED	2	7.1kb	172.18.194.140	node-2

10: 查看索引数据

```
GET study_route/_search
```

我们又插入了 1 条 `docid=c` 的新数据，但这次我们指定了路由，路由的值是一个字符串"key1". 通过查看 shard 信息，能看出这条数据路由到了 0 号 shard。也就是说用"key1"做路由时，文档会写入到 0 号 shard。

接着我们使用该路由再插入两条数据，但这两条数据的 `docid` 分别为之前使用过的 "a" 和 "b"，最终结果会是什么样？

11: 插入 `docid=a` 的数据，并指定 `routing=key1`

```
PUT study_route/_doc/a?routing=key1&refresh
```

```
{  
  "data": "A with routing key1"  
}
```

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

es 的返回信息表明文档 a 是 updated

## 12: 查看 shard

```
GET _cat/shards/study_route?v
```

index	shard	prirep	state	docs	store	ip	node
study_route	1	p	STARTED	1	3.5kb	172.18.194.140	node-1
study_route	0	p	STARTED	2	10.8kb	172.18.194.140	node-2

## 13: 查询索引

```
GET study_route/_search
```

```
    "hits" : [
      {
        "_index" : "study_route",
        "_type" : "_doc",
        "_id" : "c",
        "_score" : 1.0,
        "_routing" : "key1",
        "_source" : {
          "data" : "C"
        }
      },
      {
        "_index" : "study_route",
        "_type" : "_doc",
        "_id" : "a",
        "_score" : 1.0,
        "_routing" : "key1",
        "_source" : {
          "data" : "A with routing key1"
        }
      },
      {
        "_index" : "study_route",
        "_type" : "_doc",
        "_id" : "b",
        "_score" : 1.0,
        "_source" : {
          "data" : "B"
        }
      }
    ]
}
```

14: 插入 docid=b 的数据, 使用 key1 作为路由字段的值

```
PUT study_route/_doc/b?routing=key1&refresh
```

```
{
  "data": "B with routing key1"
}
```

```
{
  "_index" : "study_route",
  "_type" : "_doc",
  "_id" : "b",
  "_version" : 1,
  "result" : "created", ↑
  "forced_refresh" : true,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 3,
  "_primary_term" : 1
}
```

es 返回的信息变成了 created

15: 查看 shard 信息

```
GET _cat/shards/study_route?v
```

index	shard	prirep	state	docs	store	ip	node
study_route	1	p	STARTED	1	3.5kb	172.18.194.140	node-1
study_route	0	p	STARTED	3	14.5kb	172.18.194.140	node-2

## 16: 查询索引内容

GET study\_route/\_search

```
{
  "_index": "study_route",
  "_type": "_doc",
  "_id": "a",
  "_score": 1.0,
  "_routing": "key1",
  "_source": {
    "data": "A with routing key1"
  }
},
{
  "_index": "study_route",
  "_type": "_doc",
  "_id": "b",  
↑
  "_score": 1.0,  
↑
  "_routing": "key1",  
↑
  "_source": {
    "data": "B with routing key1"
  }
},
{
  "_index": "study_route",
  "_type": "_doc",
  "_id": "b",  
↑
  "_score": 1.0,  
↑
  "_source": {
    "data": "B"
  }
}
```

两个 id 为 b 的文档，其中一个比另一个多了一个字段“\_routing”

和步骤 11 插入 docid=a 的那条数据相比，这次这个有些不同，我们来分析一下。步骤 11 中插入 docid=a 时，es 返回的是 updated，也就是更新了步骤 2 中插入的 docid 为 a 的数据，步骤 12 和 13 中查询的结果也能看出，并没有新增数据，route\_test 中还是只有 3 条数据。

而步骤 14 插入 docid=b 的数据时，es 返回的是 created，也就是新增了一条数据，而不是 updated 原来 docid 为 b 的数据，步骤 15 和 16 的确也能看出多了一条数据，现在有 4 条数据。而且从步骤 16 查询的结果来看，有两条 docid 为 b 的数据，但一个有 routing，一个没有。而且也能分析出有 routing 的在 0 号 shard 上面，没有的那个在 1 号 shard 上。

这个就是我们自定义 routing 后会导致的一个问题：docid 不再全局唯一。ES shard 的实质是 Lucene 的索引，所以其实每个 shard 都是一个功能完善的倒排索引。ES 能保证 docid 全局唯一是采用 docid 作为路由，所以同样的 docid 肯定会路由到同一个 shard 上面，如果出现 docid 重复，就会 update 或者抛异常，从而保证了集群内 docid 唯一标识一个 doc。但如果我们将 docid 替换为 routing，那就不能保证了，如果用户还需要 docid 的全局唯一性，那就只能自己保证了。因为 docid 不再全局唯一，所以 doc 的增删改查 API 就可能产生问题，比如下面的查询：

---

```
GET study_route/_doc/b
GET study_route/_doc/b?routing=key1
```

上面两个查询，虽然指定的 docid 都是 b，但返回的结果是不一样的。所以，如果自定义了 routing 字段的话，一般 doc 的增删改查接口都要加上 routing 参数以保证一致性。

为此，ES 在 mapping 中提供了一个选项，可以强制检查 doc 的增删改查接口是否加了 routing 参数，如果没有加，就会报错。设置方式如下：

```
PUT <索引名>/
{
  "settings": {
    "number_of_shards": 2,
    "number_of_replicas": 0
  },
  "mappings": {
    "_routing": {
      "required": true    // 设置为 true，则强制检查；false 则不检查，默认认为 false
    }
  }
}
```

比如：

```
PUT study_route1/
{
  "settings": {
    "number_of_shards": 2,
    "number_of_replicas": 0
  },
  "mappings": {
    "_routing": {
      "required": true
    }
  }
}

// 写入一条数据
PUT study_route1/_doc/b?routing=key1
```

```
{  
    "data": "b with routing"  
}  
  
// 以下的增删改查都会报错  
GET study_route1/_doc/b  
PUT study_route1/_doc/b  
{  
    "data": "B"  
}  
DELETE study_route1/_doc/b  
// 错误信息
```

```
{  
    "error": {  
        "root_cause": [  
            {  
                "type": "routing_missing_exception",  
                "reason": "routing is required for [study_route1]/[_doc]/[b]",  
                "index_uuid": "_na_",  
                "index": "study_route1"  
            },  
            {  
                "type": "routing_missing_exception",  
                "reason": "routing is required for [study_route1]/[_doc]/[b]",  
                "index_uuid": "_na_",  
                "index": "study_route1"  
            }  
        ],  
        "status": 400  
    }  
}
```

前面我们说过很多时候自定义路由是为了减少查询时扫描 shard 的个数，从而提高查询效率。默认查询接口会搜索所有的 shard，但也可以指定 routing 字段，这样就只会查询 routing 计算出来的 shard，提高查询速度。

使用方式也非常简单，只需在查询语句上面指定 routing 即可，允许指定多个：

```
-- 查询所有分区  
GET study_route/_search  
{  
    "query": {  
        "match": {  
            "data": "b"  
        }  
    }  
}
```

---

```
        }
    }
-- 查询指定分区
GET study_route/_search?routing=key1
{
  "query": {
    "match": {
      "data": "b"
    }
  }
}
```

## 索引别名

别名，有点类似数据库的视图，别名一般都会和一些过滤条件相结合，可以做到即使是同一个索引上，让不同人看到不同的数据。别名的访问接口是`_alias`。

### 创建索引添加别名

```
PUT /index_alias
{
  "aliases": {
    "index1": {
      "filter": {
        "term": {
          "name": "Apache"
        }
      }
    }
  }
}
```

### 为已有索引添加别名

```
put /open-soft/_alias/open-soft-alias
{
  "filter":
```

```
{  
    "match":{  
        "lang":"Java8"  
    }  
}
```

## 查询指定索引的别名

get /open-soft/\_alias

## 删除别名

delete /open-soft/\_alias/open-soft-alias

## 查询别名

get \_cat/aliases

get 索引名/\_alias, 看索引所有的别名, 如: GET /open-soft/\_alias

get 别名, 看别名关联索引的详情, 如: GET open-soft-alias

## 通过别名查询

别名的使用和索引差不多, 比如:

GET open-soft-alias/\_search

## \_aliases 接口

同时\_aliases 接口可以做批量操作, 比如通过\_aliases 接口将一个别名关联多个索引:

```
POST /_aliases  
{  
    "actions": [  
        {  
            "add": {  
                "index": "my_index_1",  
                "alias": "my_index_alias"  
            }  
        },  
        {  
            "add": {  
                "index": "my_index_2",  
                "alias": "my_index_alias"  
            }  
        }  
    ]  
}
```

```
        }
    }
]
}
```

或者对于同一个 index，我们给不同人看到不同的数据，如 my\_index 有个字段是 team，team 字段记录了该数据是那个 team 的。team 之间的数据是不可见的。

```
POST /_aliases
{
  "actions": [
    {
      "add": {
        "index": "my_index",
        "alias": "my_index_teamA_alias",
        "filter": {
          "term": {
            "team": "teamA"
          }
        }
      }
    },
    {
      "add": {
        "index": "my_index",
        "alias": "my_index_teamB_alias",
        "filter": {
          "term": {
            "team": "teamB"
          }
        }
      }
    },
    {
      "add": {
        "index": "my_index",
        "alias": "my_index_team_alias"
      }
    }
  ]
}
```

```
        }
    }
]
}
```

只要有可能，尽量使用别名，推荐为 Elasticsearch 的每个索引都使用别名，因为在未来重建索引的时候，别名会赋予你更多的灵活性。假设一开始创建的索引只有一个主分片，之后你又决定为索引扩容。如果为原有的索引使用的是别名，现在你可以修改别名让其指向额外创建的新索引，而无须修改被搜索的索引之名称(假设一开始你就为搜索使用了别名)。

另一个有用的特性是，在不同的索引中创建窗口。比如，如果为数据创建了每日索引，你可能期望一个滑动窗口涵盖过去一周的数据，别名就称为 last-7-days。然后，每天创建新的每日索引时，你可以将其加入别名，同时停用或者删除第 8 天前的旧索引。

别名还能提供另一个特性，那就是路由。不过，在谈论使用别名进行路由之前，我们先来讲述一下通常情况下路由是如何使用的。

## 结合路由和别名

在之前的章节中，我们已经了解到别名是索引之上的抽象，非常强大和灵活。假设别名指向一个单独的索引，那么它们也可以和路由一起使用，在查询或索引的时候自动地使用路由值。如：

```
POST /_aliases
{
  "actions": [
    {
      "add": {
        "index": "open-soft",
        "alias": "my_index_alias",
        "filter": {"match": {"lang": "Java"}},
        "routing": "AKey"
      }
    },
    {
      "add": {
        "index": "open-soft",
        "alias": "my_index_alias2",
        "filter": {"match": {"lang": "Java"}},
        "routing": "BKey"
      }
    }
  ]
}
```

```
        }
    ]
}

或

PUT /open-soft/_alias/open-soft-alias2
{
  "filter": {
    "match": {
      "lang": "Java"
    }
  },
  "routing": "CKey"
}
```

## \_rollover 接口

\_rollover 接口用于根据一系列条件将别名指向一个新的索引，这些条件包括存续时间、文档数量和存储容量等。这与日志文件使用的文件滚动类似，文件滚动是通过不断创建新文件并滚动旧文件来保证日志文件不会过于庞大，而\_rollover 接口则是通过不断将别名指向新的索引以保证索引容量不会过大。但这种别名滚动并不会自动完成，需要主动调用\_rollover 接口。

别名滚动的条件可通过 conditions 参数设置，包括 max\_age、max\_docs 和 max\_size 等三个子参数。例如，创建一个索引 logs-1 并分配别名 logs，然后调用 logs 别名的\_rollover 接口设置别名滚动条件，如：

```
PUT /logs-1
{
  "aliases": {
    "logs": {}
  }
}
```

```
POST /logs/_rollover
{
  "conditions": {
    "max_age": "14d",
    "max_docs": 10000,
    "max_size": "4gb"
```

```
    }
}
```

在示例中，`logs` 别名指向 `logs-1` 索引，最大存活周期为 14 天，最大文档数量 10000 条，最大存储容量 4GB。因为 `logs-1` 索引刚刚创建，存活时间、文档数量和存储容量都不满足条件，所以使用示例的请求不会对 `logs` 别名产生任何影响。这通过请求返回的结果也可以看到：

```
{
  "acknowledged": false,
  "shards_acknowledged": false,
  "old_index": "logs-1",
  "new_index": "logs-000002",
  "rolled_over": false,
  "dry_run": false,
  "conditions": [
    "[max_size: 4gb]": false,
    "[max_docs: 10000)": false,
    "[max_age: 14d]": false
  ]
}
```

从返回结果的 `conditions` 属性来看，三个条件匹配的结果都是 `false`，所以不会触发索引滚动。如果想体验别名滚动的效果，可以将 `max_age` 设置为 1s 再调用上面的请求。之后通过“`GET _cat/indices`”接口就会发现有新的索引 `logs-000002` 产生，再分别查看这两个索引就会发现，`logs-1` 的别名已经被清空，而 `logs-000002` 的别名中则已经添加了 `logs`。新索引的命名规则在原索引名称数字的基础上加 1，并且将数值长度补 0 满足 6 位。

所以使用 `_rollover` 接口时，要求索引名称必须以数字结尾，数字与前缀之间使用连接线“-”连接。

由于 `_rollover` 接口在滚动新索引时，会将别名与原索引的关联取消，所以通过别名再想查找已经编入索引的文档就不可能了。为了保证原文档可检索，可以通过别名 `is_write_index` 参数保留索引与别名的关系。当使用 `is_write_index` 参数设置了哪一个索引为写索引时，`_rollover` 接口滚动别名指向索引时将不会取消别名与原索引之间的关系。它会将原索引的 `is_write_index` 参数设置为 `false`，并将新索引的 `is_write_index` 参数设置为 `true`。

例如在创建 `logs-4` 时添加参数所示：

```
PUT /logs-4
{
  "aliases": {
    "logs4": {
      "is_write_index": true
    }
  }
}
```

再执行示例中的请求时

```
POST /logs4/_rollover
```

```
{
  "conditions": {
    "max_age": "1s",
    "max_docs": 10000,
    "max_size": "4gb"
  }
}
```

GET \_cat/aliases?v

alias	index	filter	routing.index	routing.search
is_write_index		-	-	-
logs	logs-1	-	-	-
.kibana_task_manager	.kibana_task_manager_1	-	-	-
index1	index_alias	*	-	-
Logs4	logs-000005	-	-	true
logs	logs-2	-	-	-
.kibana	.kibana_1	-	-	-
logs4	logs-4	-	-	false
my_index_alias	open-soft	*	AKey	AKey
my_index_alias2	open-soft	*	BKey	BKey
open-soft-alias	open-soft	*	-	-
open-soft-alias2	open-soft	*	CKey	CKey
logs3	logs-3	-	-	-

会发现 logs-4 的 is\_write\_index 参数被设置为 false, 而新生成索引 logs-000005 的 is\_write\_index 参数则为 true。在两者的别名列表中都包含有 logs, 可以继续通过 logs 别名对原索引进行查询。

## Elasticsearch 接口拾遗

### 容量控制

#### \_split 接口

\_split 接口可以在新索引中将每个主分片分裂为两个或更多分片，所以使用 split 扩容时分片总量都是成倍增加而不能逐个增加。使用 split 接口分裂分片虽然会创建新的索引，但新索引中的数据只是通过文件系统硬连接到新索引中，所以并不存在数据复制过程。而扩容的分片又是在本地分裂，所以不存在不同节点间网络传输数据的开销，所以 split 扩容效率相对其他方案来说还是比较高的。

\_split 接口做动态打容需要预先设置索引的 number\_of\_routing\_shards 参数，Elasticsearch 向分片散列文档采用一致性哈希算法，这个参数实际上设置了索引分片散列空间。所以分裂后分片数量必须是 number\_of\_routing\_shards 的因数，同时是 number\_of\_shards 的倍数。

例如，设置 number\_of\_routing\_shards 为 12, number\_of\_shards 为 2，则分片再分裂存在 2->4->12、2->6->12 和 2->12 三种可能的扩容路径。分裂后分片数

---

量可通过`_split`接口的`index.number_of_shards`参数设置，数量必须满足前述整数倍的要求。上面讲解的这些规则比较抽象，下面通过创建一个具体示例来看一下如何通过`split`接口扩容索引。

首先创建一个索引`employee`,将它的主分片数量`number_of_shards`设置为2,散列空间`number.of.routing shards`设置为12。然后，通过将索引的`blocks.write`参数设置为`true`,将索引设置为只读，这是因为使用`split`接口要求索引必须为只读。最后调用`split`接口将`employee`索引的分片分裂到新索引`splited_employee`中，`index.number_of_shards`参数设置为4，即分裂为4个分片。如示例所示：

```
DELETE employee

PUT employee
{
  "settings": {
    "number_of_shards":2,
    "number_of_routing_shards":12
  }
}
PUT /employee/_settings
{
  "blocks.write": true
}
PUT /employee/_split/splited_employee
{
  "settings": {
    "index.number_of_shards":4,
    "index.number_of_replicas":1,
    "index.blocks.write":false
  },
  "aliases": {
    "stu":{}
  }
}
```

在执行成功后，可调用“`GET _cat/shards`”查看分片。在返回结果中可以看到`employee`索引共有4个分片，即2个主分片和2个副本分片;新索引`splited_employee`会有8个分片，即4个主分片和4个副本分片。`split`接口在创建新索引的同时，会将原索引的配置也一同设置到新索引中。所以`index.blocks.write`参数也会一同被复制过来，但这可能并不是我们想要的。

---

所以在分裂分片的同时支持通过 `aliases` 和 `settings` 设置新索引的别名和配置，所以可以在分裂分片的同时将 `index.blocks.write` 参数覆盖。在示例中就将这个参数覆盖了，同时还添加了新的别名 `stu`。另外，还可以在地址中添加 `copy_settings = false` 参数禁止从源索引中复制配置，但这个参数在版本 8 中有可能被废止，所以添加这个参数会收到警告。

使用 `split` 接口成功分裂分片后，原索引并不会被自动删除。通过原索引和新索引都可以查看到相同的文档数据，原索引是否删除应根据业务需要具体判断。

## \_shrink 接口

与 `_split` 接口相反，`_shrink` 接口用于缩减索引分片。尽管它们在逻辑上正好相反，但它们在应用时的规则基本上是一致的。比如，`shrink` 接口在缩减索引分片数量时也要求原始分片数量必须是缩减后分片数量的整数倍。例如原始分片数量为 12，则可以按 12->6->3->1 的路径缩减，也可以按 12->4->2->1 的路径缩减。在调用 `_shrink` 接口前要满足两个条件，第一个条件与 `_split` 接口类似，就是要求索引在缩容期间必须只读；第二个条件有些特殊，就是要求索引所有分片（包括副本分片）都要复制一份存储在同一节点，并且要求健康状态为 `green`，这可以通过 `routing.allocation.require._name` 指定节点名称实现。如果想要查看节点名称，可调用“`GET _nodes`”接口查看所有集群节点。

与 `_split` 接口类似，索引在缩减后的具体分片数量可通过 `shrink` 接口的 `index.number_of_shards` 参数设置。但它的值必须与原始分片数量保持整数比例关系，如果不设置该参数将直接缩减为 1 个分片。如示例所示，缩减后索引分片数量为 2，同时还清除了两项配置：

```
PUT /splited_employee/_settings
{
  "settings": {
    "blocks.write": true
    "routing.allocation.require._name": "TIAN-PC"
  }
}
```

```
PUT /splited_employee/_shrink/shrinked_employee?copy_settings=true
{
  "settings": {
    "index.routing.allocation.require.name": null,
    "index.blocks.write": null,
    "index.number_of_shards": 2
  }
}
```

---

同样地，使用 `shrink` 接口缩容后会创建新索引 `shrunk_employee`，原索引和新索引都可以查询到相同的文档数据。

## \_reindex 接口

尽管 `_split` 接口和 `shrink` 接口可以对索引分片数量做扩容和缩容，但在分片数量上有倍数要求，并且分片总量受散列空间(即 `number_of_routing_shards` 参数)的限制。如果索引容量超出了散列空间或者有其他特殊要求，则可以按新需求创建新的索引。`Elasticsearch` 提供的 `_reindex` 接口支持将文档从一个索引重新索引到另一个索引中。但显然重新索引在性能上的开销要比 `_split` 和 `shrink` 大，所以尽量不要使用这种办法。`_reindex` 接口需要两个参数 `source` 和 `dest`，前者指明文档来源索引而后者则指明了文档添加的新索引。例如在示例中是将 `users` 索引中的文档添加到 `users_copy` 索引中：

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

需要注意的是，在重新索引时，不会将原索引的配置信息复制到新索引中。如果事先没有指定索引配置，重新索引时将根据默认配置创建索引及映射。另外，使用 `reindex` 接口必须将索引的 `_source` 字段开启。

## 缓存相关

为了提升数据检索时的性能，`Elasticsearch` 为索引提供了三种缓存。

第一种缓存称为节点查询缓存(`Node Query Cache`)，负责存储节点查询结果。节点查询缓存是节点级别的，一个节点只有一个缓存，同一节点上的分片共享同一缓存。在默认情况下，节点查询缓存是开启的，可通过索引 `index.queries.cache.enabled` 参数关闭。节点查询缓存默认使用节点内存的 10%作为缓存容量上限，可通过 `indices.queries.cache_size` 更改，这个参数是节点的配置而非索引配置。

第二种缓存被称为分片请求缓存(`Shard Request Cache`)，负责存储分片接收到的查询结果。分片请求缓存不会缓存查询结果的 `hits` 字段，也就是具体的文档内容，它一般只缓存聚集查询的相关结果。在默认情况下，分片请求缓存也是开启的，通过索引 `index.requests.cache.enable` 参数关闭。另一种关闭该缓存的办法，是在调用 `search` 接口时添加 `request._cache = false` 参数。

分片请求缓存使用的键是作为查询条件 JSON 字符串，所以如果查询条件 JSON 串完全相同，文档的查询几乎可以达到实时。但由于 JSON 属性之间并没有

---

次序要求，这意味着即使 JSON 描述的是同一个对象，只要它们属性的次序不同就不能在缓存中命中数据。这点在使用时需要格外注意。

最后一种缓存就是 `text` 类型字段在开启 `fielddata` 机制后使用的缓存，它会将 `text` 类型字段提取的所有词项全部加载到内存中，以提高使用该字段做排序和聚集运算的效率。由于 `fielddata` 是 `text` 类型对文档值机制的代替，所以这种缓存机制天然就是开启的且不能关闭。但可通过 `indices. Fielddata.cache. size` 设置这个缓存的容量，默认情况下该缓存没有容量上限。

缓存的引入使得文档检索性能得到了提升，但缓存一般会带来两个主要问题：一是如何保证缓存数据与实际数据的一致；另一个问题是 当缓存容量超出时如何清理缓存。

数据一致性问题，`Elasticsearch` 是通过让缓存与索引刷新频率保持一致实现的。还记得索引是准实时的吗？索引默认情况下会以每秒 1 次的频率将文档编入索引，`Elasticsearch` 会在索引更新的同时让缓存也失效，这就保证了索引数据与缓存数据的一致性。缓存数据容量问题则是通过 LRU 的方式，将最近最少使用的缓存条目清除。同时，`Elasticsearch` 还提供了一个 `cache` 接口用于主动清理缓存。

## [\\_refresh 接口](#)

`_refresh` 接口用于主动刷新一个或多个索引，将已经添加的文档编入索引以使它们在检索时可见。在调用该接口时，可以直接调用或与一个或多个索引起使用， 还可以使用`_all` 刷新所有索引。

`GET 索引 1/_refresh`

`POST _refresh`

`GET all/_refresh`

`POST 索引 1, 索引 2/_refresh`

事实上，除了使用 `refresh` 接口主动刷新索引外，也可以在操作文档时通过 `refresh` 参数刷新索引。

## [\\_cache 接口](#)

`_cache` 接口用于主动清理缓存，在调用该接口时需要在`_cache` 后附加关键字 `clear`。

`_cache` 接口可以清理所有缓存，也可以清理某一索引甚至某一字段的缓存，还可以只清理某种类型的缓存。例如：

`POST /索引 1/_cache /clear?query=true`

`POST /索引 1, 索引 2/_cache /clear?request= true`

`POST /索引 1/_cache/clear?fielddata =true&fields = notes`

在示例中，`query`、`request`、`fielddata` 参数分别对应于不同的缓存类型，而 `fields` 参数则用于定义清理哪一个字段的缓存。

## 查看运行状态

除此上述接口以外，Elasticsearch 还提供了一组用于查看索引及分片运行情况的接口，包括`_stat`、`_shard_stores` 和 `_segments` 等。由于它们往往在性能分析时使用。

### `_stats` 接口

`_stats` 接口用于查看索引上不同操作的统计数据，可以直接请求也可以与索引名称一起使用。`_stats` 接口返回的统计数据非常多，如果只对其中某一组统计数据感兴趣，可以在`_stats` 接口后附加统计名称。例如以下对`stats` 接口的调用都是正确的：

```
GET _stats  
GET _stats/store  
GET kibana_sample_data_flights/_stats  
GET kibana_sample_data_flights/_stats/fielddata
```

在`stats` 接口中可以使用的统计名称及它们的含义见下面，它们在返回结果中的含义与此相同。

```
docs 文档总量  
store 索引存储量  
indexing 索引文档的统计  
get GET 查询文档的统计  
search _search 接口检索文档的统计  
segments 分段内存的统计  
completion completion 统计  
freelddata fielddata 机制使用情况统计  
flush flush 统计  
merge merge 统计  
request_cache request_cache 统计  
refresh refresh 统计  
warmer warmer 统计  
translog translog 统计
```

### `_segments` 接口

`_shard_stores` 接口用于查询索引分片存储情况，而`segments` 接口则用于查看底层 Lucene 的分段情况。这两个接口都只能通过`GET` 方法请求，同时都可以针对一个或多个索引，例如：

```
GET _shard_stores
```

---

```
GET /kibana_sample_data_flights/_shard_stores
```

```
GET _segments
```

```
GET /kibana_sample_data_flights/_segments/
```

## 其他检索接口

我们前面的检索实际上都是围绕着 `search` 接口，但实际上 Elasticsearch 还提供了许多与文档检索有关的接口。比如，如果想要查看索引中满足条件的文档数量可以使用 `_count` 接口，如果想要执行一组检索可以使用 `msearch` 接口，

### `_count` 接口

Elasticsearch 提供了查看文档总数的 `_count` 接口，可通过 `GET` 或 `POST` 方法请求该接口。在请求该接口的路径上，可以添加索引、映射类型，以限定统计文档数量的范围。所以在示例中对 `count` 接口的请求都是正确的：

```
GET _count
POST /kibana_sample_data_logs/_count
{
  "query": {
    "match": {
      "message": "chrome"
    }
  }
}
```

### `_msearch` 接口

`_msearch` 接口，可以在一次接口调用中执行多次查询，可以使用 `GET` 或 `POST` 方法请求。请求体每两行为一组视为一个查询，第一行为查询头包含 `index`、`search_type`、`preference` 和 `routing` 等基本信息，第二行为查询体包含具体要检索的内容如 `query`、`aggregation` 等。例如：

```
POST /kibana_sample_data_flights/_msearch
{}
{"query": {"match_all": {}}, "from": 0, "size": 10 }
{}
{"query": {"match_all": {}}}
{"index": "kibana_sample_data_logs" }
{"query": {"match_all": {}}}
```

---

在示例中包含了 3 个查询，前两个查询的查询头都是空的，所以默认在请求路径中指定的 kibana\_sample\_data\_flights 索引中查询；最后一个则在查询头 中指定了索引为 kibana\_sample\_data\_logs，所以会在 kibana\_sample\_data\_logs 索引中查询。

## \_validate 接口

\_validate 接口用于在不执行查询的情况下，评估一个查询是否合法可执行，这通常用于验证执行开销比较高的查询。\_validate 接口可通过 GET 或 POST 方法请求，请求路径中必须要包含 \_validate/query，也可以在路径中添加索引名称以限定查询执行的范围。类似 \_search 接口。示例：

```
POST _validate/query
{
  "query": {
    "range": {
      "AvgTicketPrice": {
        "gte": 1000,
        "lte": 1500
      }
    }
  }
}
```

## \_field\_caps 接口

\_field\_caps 接口用于查看某一字段支持的功能，主要包括字段是否可检索以及是否可聚集等。需要查看的字段可以通过 URI 参数 fields 设置，虽然可以使用 GET 或 POST 方法请求。在请求地址中，还可以添加索引名称以限定查询范围。例如：

```
GET _field_caps?fields=AvgTicketPrice
POST /kibana_sample_data_logs/_field_caps?fields=message,agent
```

## 批量接口

批量操作除了 \_msearch 接口以外，还有 \_bulk 接口和 \_mget 接口。

### \_bulk

如果需要批量地对 Elasticsearch 中的文档进行操作，可以使用 \_bulk 接口执行以提升效率和性能。\_bulk 接口一组请求体，请求体一般每两个一组，对应一种对文档的操作；第一个请求体代表操作文档的类型，而第二个请求体则代表操作文档所需要的参数。但对于某些不需要参数的文档操作来说，则可能只有一个请求体。

---

操作类型包括 index、create、 delete、 update 等，其中 index 和 create 都代表创建文档，区别在于当要创建的文档存在时，create 会失败而 index 则可以变为更新； delete 和 update 则分别代表删除和更新文档。例如：

```
POST _bulk
{
  "index": {"_index": "students", "_id": "10"}
  {
    "name": "smith"
  }
  {
    "delete": {"_index": "test", "_id": "5"}
  }
  {
    "create": {"_index": "test", "_id": "11"}
  }
  {
    "age": 30, "name": "Candy"
  }
  {
    "update": {"_id": "1", "_index": "students" }
  }
  {
    "doc": {"age": "20"}}
```

## \_mget

一条一条的查询，比如说要查询 100 条数据，那么就要发送 100 次网络请求，这个开销还是很大的，如果批量查询的话，查询 100 条数据，就只要发送 1 次网络请求，网络请求的性能开销缩减 100 倍。

比如一般的查询是： get /open-soft/\_doc/1

批量查询则是：

```
GET /_mget
{
  "docs": [
    {
      "_index": "open-soft",
      "_id": 1
    },
    {
      "_index": "open-soft-2",
      "_id": 2
    }
  ]
}
```

# Kibana

Kibana 在整个 Elastic Stack 家族中起到数据可视化的作用，也就是通过图、表、统计等方式将复杂的数据以更直观的形式展示出来。由于 Kibana 运行于

---

Elasticsearch 基础之上，所以可以将 Kibana 视为 Elasticsearch 的用户图形界面 ( Graphic User Interface, GUI) 。

因为 Kibana 中可视化对象很多，我们挑选几个典型的进行讲解。

## 文档发现( Discover)

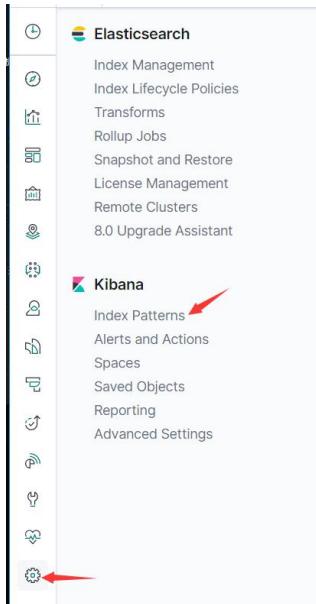
本章主要介绍 Kibana 导航栏中的第一个功能文档发现( Discover)，它提供了交互式检索文档的接口，用户可以在这里提交查询条件、设置过滤器并查看检索结果。在文档发现中的查询条件还可以保存起来，这些保存起来的查询条件称为查询对象，可以在文档可视化和仪表盘功能中使用。

## 索引模式

在使用文档发现功能检索文档之前，首先要告诉 Kibana 要检索 Elasticsearch 的哪些索引，这在 Kibana 中是通过定义索引模式来实现的。没有被索引模式包含进来的索引不能在文档发现、文档可视化和仪表盘等功能中使用，如何创建索引模式？

### 创建索引模式

索引模式是一种对 Elasticsearch 中索引的模式匹配，以定义哪些索引将被包含到这个模式中。它以索引名称为基础，可以匹配单个索引也可以使用星号 “\*” 匹配多个索引。例如在导入飞行记录样例数据时，Kibana 会创建一个名为 `kibana_sample_data_flights` 的索引模式。索引模式的管理功能位于 Management 菜单中。



单击 Index Patterns 链接将进入索引模式管理界面。这个界面列出了所有索引模式。

要创建新的索引模式，单击右上角的 Create index pattern 按钮，进入索引模式创建界面

Index patterns [?](#)

Search...

[Pattern ↑](#)

<a href="#">kibana_sample_data_flights</a>	Default
<a href="#">kibana_sample_data_logs</a>	
<a href="#">pattern_test4</a>	

Rows per page: 10 [▼](#)

< 1 >

创建索引模式分为两步，第一步需要在“Define index pattern”输入框中给出匹配索引名称的模式。模式可以直接使用索引名称，也可以使用星号“\*”匹配任意字符。在输入模式的同时，Kibana 会在输入框下动态地将匹配索引模式的所有索引列出来，用户可以实时查看索引模式是否满足要求。单击 Next Step 按钮进入下一步。

### Step 1 of 2: Define index pattern

Index pattern

[kibana\\_sample\\_data\\_\\*](#)

You can use a \* as a wildcard in your index pattern.  
You can't use spaces or the characters \, /, ?, ", <, >, |.

✓ Success! Your index pattern matches 2 indices.

[kibana\\_sample\\_data\\_flights](#)

[kibana\\_sample\\_data\\_logs](#)

Rows per page: 10 [▼](#)

[Next step](#)

创建索引模式第二步是设置索引模式的一些配置信息，这包括添加时间过滤器和为索引模式指定 ID。如果模式匹配的索引中包含有时间类型的字段，在这一页将会包含一个设置时间过滤器的选项，用户可以选择使用哪一个字段作为过滤条件；否则页面将提示不包含时间类型的字段而不会有时间过滤器选项。

在时间过滤器下，还有一个链接“Show advanced options”，单击这个链接会打开设置索引模式 ID 的输入框。默认情况下，Kibana 会给索引模式自动生成一个 ID。如果想自己定义索引模式 ID，可以在这个输入框中输入 ID。

### Step 2 of 2: Configure settings

You've defined [kibana\\_sample\\_data\\_\\*](#) as your index pattern. Now you can specify some settings before we create it.

Time Filter field name [Refresh](#)

timestamp [▼](#)

The Time Filter will use this field to filter your data by time.  
You can choose not to have a time field, but you will not be able to narrow down your data by a time range.

[Hide advanced options](#)

Custom index pattern ID

custom-index-pattern-id

Kibana will provide a unique identifier for each index pattern. If you do not want to use this unique ID, enter a custom one.

[Back](#)

[Create index pattern](#)

单击 Create index pattern 链接，Kibana 将创建这个索引模式，并跳入管理模式字段界面。同时在创建完索引模式后，在数据发现界面中就可以选择这些索引模式了。如图所示。

## Index patterns [?](#)

Search...  
Pattern ↑

- kibana\_sample\_data\_\* **Default**
- kibana\_sample\_data\_flights
- kibana\_sample\_data\_logs
- pattern\_test4

## 管理模式字段

在管理模式的页面中选择一个索引模式后，右侧页面中会列出索引模式中的字段信息，包括字段名称、类型、是否可检索等。在每个字段后面都有一个修改按钮，可修改字段值的格式等信息，如图所示。

★ kibana\_sample\_data\_\*

Time Filter field name: timestamp Default

This page lists every field in the **kibana\_sample\_data\_\*** index and the field's associated core type as recorded by Elasticsearch. To change a field type, use the Elasticsearch [Mapping API](#).

Name	Type	Format	Searchable	Aggregatable	Excluded
@timestamp	date		●	●	✎
AvgTicketPrice	number		●	●	✎
Cancelled	boolean		●	●	✎
Carrier	string		●	●	✎
Dest	string		●	●	✎
DestAirportID	string		●	●	✎

除了字段以外，在管理字段的界面中还包括脚本字段(Scripted fields)、源过滤器(Source filters)两个标签页。脚本字段是通过脚本在运行时动态添加到索引模式中的字段，而源过滤器则用于从源文档中过滤字段。

脚本字段并不是真实地存在于索引中，而是根据其他字段值运算而来。给索引模式添加脚本字段需要在索引模式创建后，在管理索引模式的界面中在“Scripted Field”标签页中处理。例如，Kibana 提供的样例索引模式 `kibana_sample_data_flights` 中就包含一个脚本字段。

### Scripted fields

You can use scripted fields in visualizations and display them in your documents. However, you cannot search scripted fields.

Add scripted field

Name	Lang	Script	Format
hour_of_day	painless	doc['timestamp'].value .hourOfDay	Number

源过滤器用于从源文档中过滤字段，被过滤的字段将不会在文档发现和仪表盘中展示。源过滤器在定义时可以使用字段名称精确匹配字段，也可以使用星号匹配多个字段。

在索引模式管理界面的右上角有 三个按钮，它们的作用分别是设置默认索引模式、刷新字段列表和删除索引模式。当设置索引模式为默认时，它将在文档发现中成为默认索引模式。

## 发现文档

文档发现就是要将满足条件的文档检索出来，Kibana 提供了多种方式设置查询条件。

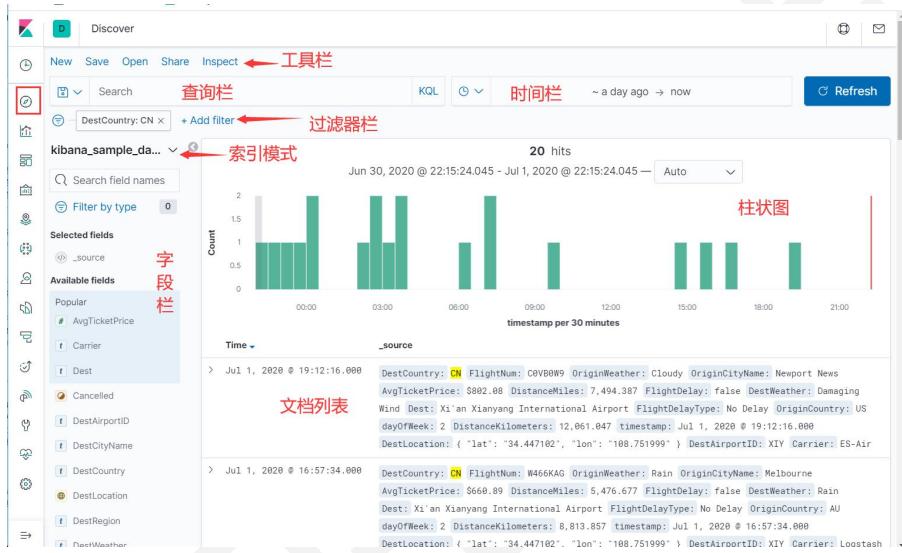
这包括通过时间范围过滤文档、使用过滤器过滤文档，还可以通过 Lucene 或 KQL 查询语言过滤文档。无论使用哪一种方式过滤文档，它们最终都会以 DSL 查询语言的形式传递给底层的 Elasticsearch。先来看看时间范围和过滤器这两种方式，它们最终会以 must 子句的形式组合进 bool 查询。

如果想要查看 Kibana 最终生成的请求，可单击工具栏中的 Inspect 按钮，在弹出窗口中选择 Request 标签页查看。

The screenshot shows the Kibana interface with the 'Inspect' button highlighted in red. A modal window titled 'Request' is open, also with its tab highlighted in red. The modal displays the generated Elasticsearch query:

```
{ "version": true, "size": 500, "sort": [ { "timestamp": { "order": "desc", "unmapped_type": "boolean" } } ] }
```

## 数据发现界面结构



首先，在数据发现界面最上面靠左侧有一排包含了特定功能的按钮，如 New、Save、Open 等，本书后续章节将这一栏称为工具栏，它们用于实现对文档发现的创建、保存等基础管理功能。

接下来，在工具栏下面有一个输入框可键入查询条件检索文档，我们把这个输入栏称为为查询栏。查询栏接收 Elasticsearch 查询语言 DSL,或者 Kibana 内置查询语言 KQL。

在查询栏下侧称为过滤器栏，在没有定义过滤器时该栏只有一个 Add a filter 链接，而添加了过滤器后会有相应的过滤器定义展示出来。

在查询栏右侧是时间栏，这一栏的作用可以设置检索文档的时间范围，还可以设置检索文档的刷新频率。所以查询栏、过滤器栏和时间栏的主要作用都是精确地限定检索文档的范围。

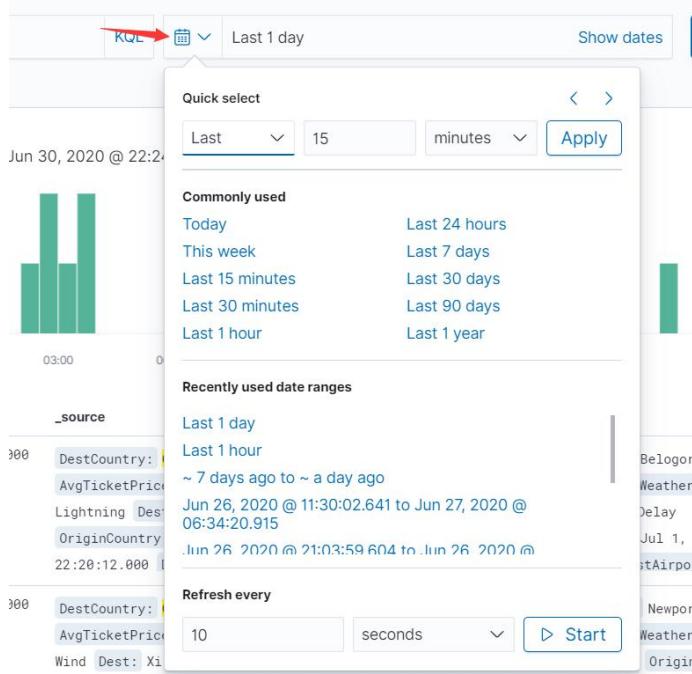
再往下可以分为左右两栏，左侧分别列出了索引模式、已选择字段和可选择字段；而右侧上部显示了满足过滤条件文档的柱状图，下部则将相应的文档数据展示出来了。在了解了文档发现界面的基本结构后就可以在其中浏览文档了，但有时会发现文档不存在，这时就需要更新文档的时间范围和刷新频率了。

## 使用时间过滤文档

在前面曾介绍过，如果索引包含时间类型的字段，则在创建索引模式的第二步中可以为索引模式添加时间过滤器。

如果在创建索引模式时添加了时间过滤器，那么在文档发现中就会看到设置时间过滤器的按钮。通常这个过滤器的默认值为"Last 15minutes",位于文档发现界面的时间栏上。

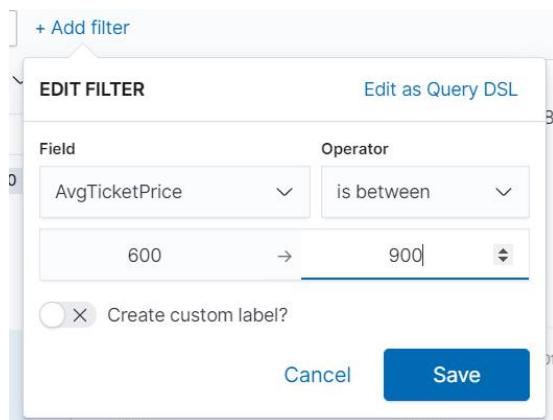
时间栏上有一个日历图形的按钮，单击它会弹出选择时间范围的界面。时间范围可以设置为一个相对时间范围，例如最近 24 小时、前两周、前两个月等；也可以是一个绝对时间范围，例如从 2019-4-1 至 2019-5-10，如图所示。



在图中，弹出时间范围窗口的最下面还有一个可以设置刷新频率的输入框。在这个输入框中可以按秒、分、小时为单位，输入页面刷新的时间间隔。设置好后单击后面的 Start 按钮，页面就会按设置好的时间间隔自动刷新数据。

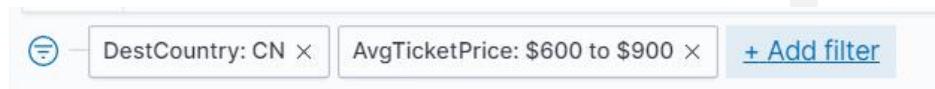
## 自定义过滤器

通过时间范围过滤文档只能通过在创建索引模式时指定的时间字段过滤文档，如果想通过其他字段对文档做过滤就必须要借助过滤器了。为文档添加过滤器可通过过滤器栏的 **Add filter** 按钮完成，单击这个按钮会弹出 **EDIT FILTER** 对话框，如图所示。

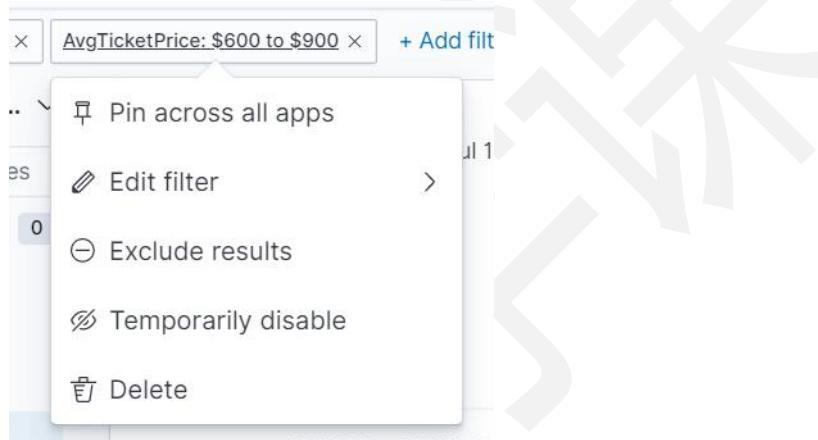


在 **EDIT FILTER** 对话框中，可通过 **Field** 下拉列表选择字段，然后通过 **Operator** 和 **Value**

下拉列表选择操作符和具体值。例如设置一个根据平均票价介于“600”到“900”之间的条件过滤航班文档，首先要在 **Field** 下拉列表中选择 **AvgTicketPrice** 字段，然后在 **Operator** 下拉列表中选择 “**is between**”, 最后在弹出的 **From** 和 **To** 输入框中输入 “600” 和 “900”，单击 **Save** 按钮就完成了过滤器的设置。



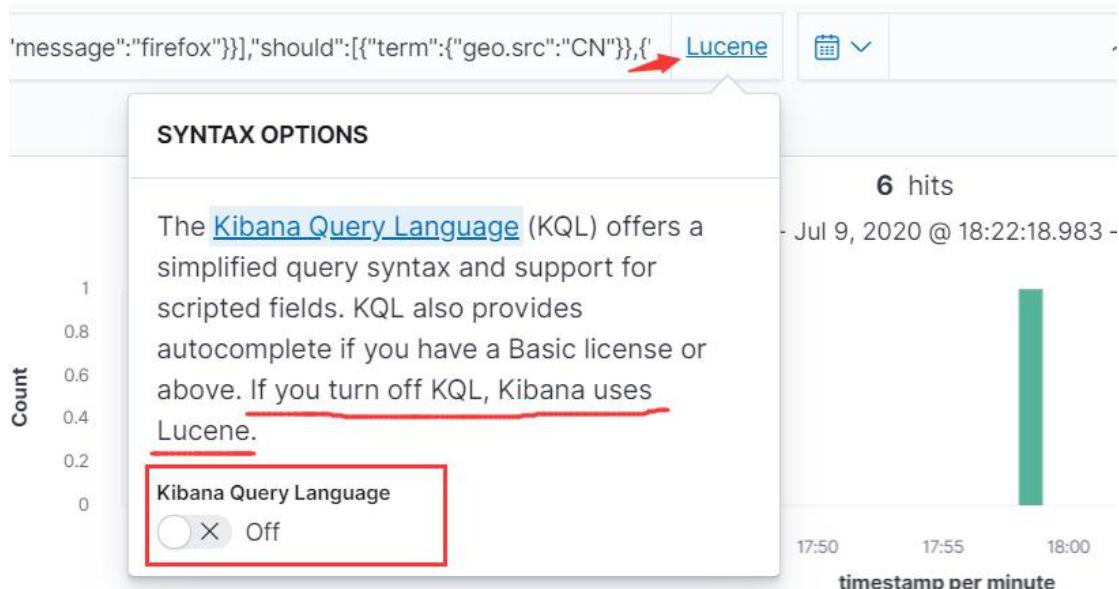
新创建出来的过滤器会出现在过滤器栏上，如果需要组合多个字段过滤文档，可以多次单击 **Addfilter** 创建多个过滤器。由于过滤器设置的查询条件最终会被放置在 **bool** 组合查询的 **must** 子句中，所以多个过滤器之间的逻辑关系是与。点击这些出现在过滤器栏上的过滤器时，会弹出一个对话框，包括对过滤器编辑、删除、启用以及逆向过滤等功能。如图所示。



## 使用查询语言

时间范围和过滤器设置的查询条件都是以逻辑与的形式组合在一起的，如果需要设置更复杂的查询条件就需要在查询栏中输入查询条件以检索文档。目前 Kibana 文档发现中支持 **Lucene** 和 **KQL** (**Kibana Query Language**)两种查询语言，前者可以认为就是 Elasticsearch 中的 **DSL**，而后者则是 Kibana 提供的一种新查询语言。

要切换查询语言，只需



要查询，在输入栏中输入查询语言即可，比如：{"bool": {"must": [{"match": {"message": "firefox"} }], "should": [{"term": { "geo.src": "CN"}}, {"term": { "geo.dest": "CN"} }]}}



*tips:*

还记得这个查询条件的含义吗？

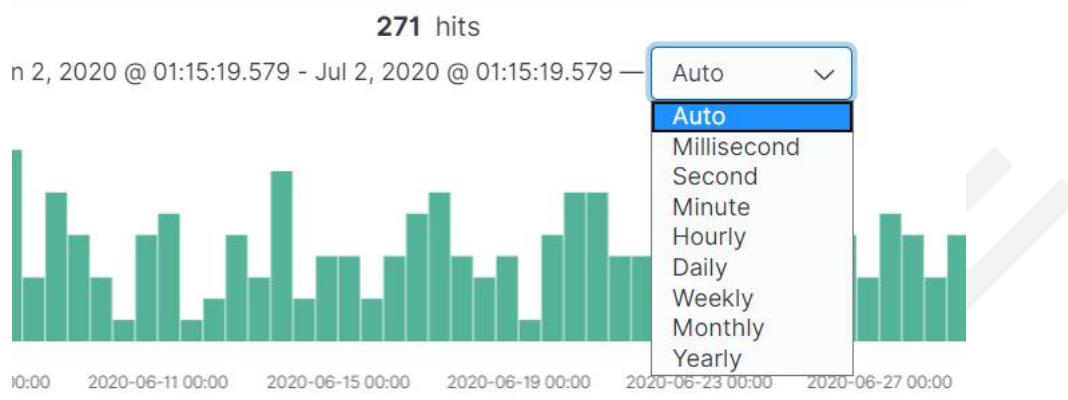
只有 `message` 字段包含 `firefox` 词项的日志文档才会被返回，而 `geo` 的 `src` 字段和 `dest` 字段是否为 `CN` 只影响相关度，当然相关度越高的肯定排在前面。

## 文档展示与字段过滤

通过时间范围、过滤器和查询语言发现的文档最终会以柱状图和表格两种形式展示

### 柱状图

文档发现会根据保存的查询条件刷新页面。根据查询条件生成的柱状图会以时间为 X 轴，而以文档数量为 Y 轴。当鼠标指针悬停于单个柱子上时，还会弹出当前柱子代表数据的具体说明，包括时间范围、文档数量等。在默认情况下，柱状图中各个柱子之间的时间间隔会根据时间范围自动匹配。在柱状图栏右上侧也提供了一个下拉列表修改时间间隔，如图所示。



柱状图还有另外一个功能，它可以通过鼠标划选的方式选择文档发现的时间范围，这对于通过柱状图精细查看某一时间段数据来说非常方便。当鼠标指针悬停于柱状图上时，指针会变为十字星状。按下鼠标左键选择一个范围做拖拽，会弹出提示框，并显示满足条件的文档数量和时间范围。选择到合适的时间范围后放开鼠标，Kibana 会根据划定的范围自动将时间范围设置好。

## 文档展示

默认情况下文档栏只包含两列，一列是 `Time`, 另一列是 `_source`。`Time` 列仅在索引模式设置了时间过滤器时才会有，它会展示时间过滤器中指定的时间字段值。

`_source` 列则显示源文档，也就是在使用 DSL 查询时返回的 `_source` 字段值。文档栏展示哪些列可通过左侧的字段栏来定制，字段栏分为“**Selected fields**”和“**Available fields**”两个区域。

“**Selected fields**”区域列出了将在文档展示栏展示的所有字段，默认情况下只有一个 `_source` 字段；而“**Available fields**”区域则列出了文档所有可以展示的字段。“**Available fields**”区域列出的字段受到索引模式的源文档过滤器影响。

当鼠标指针悬停在“**Available fields**”列表中的字段上时，字段后面将会出现“**add**”按钮。单击这个按钮，字段就会被添加到“**Selected fields**”列表中，同时文档展示栏中展示的字段也会随之变化。一旦明确指定了需要展示的字段，`source` 字段就不会再出现在文档展示栏中。反过来，当鼠标指针悬停在

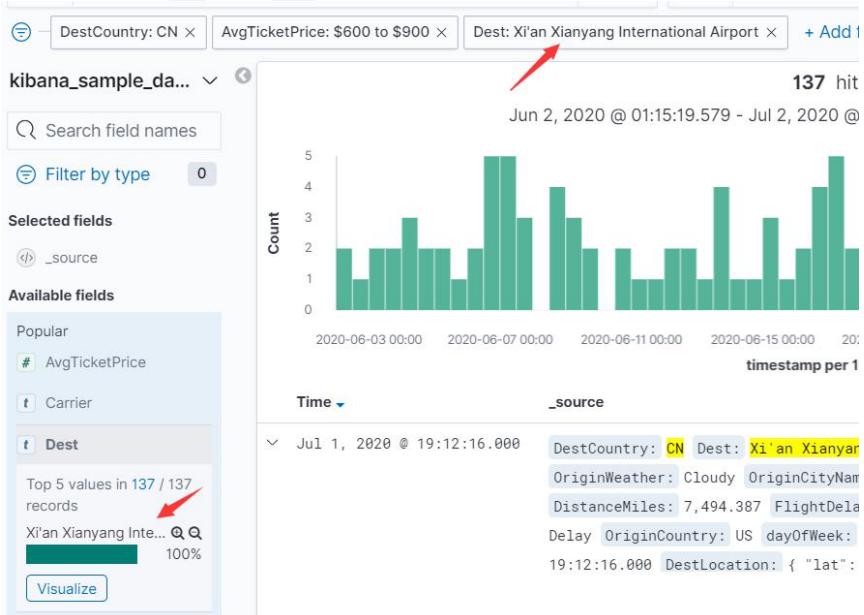
“**Selected fields**”字段上时，字段后面将会出现“**remove**”按钮。单击这个按钮，字段将从“**Selected fields**”列表中删除并出现在“**Available fields**”列表中，而文档展示栏中也不会再展示这个字段的内容。

## 添加过滤器

除了可以通过 `add` 和 `remove` 按钮设置展示字段外，单击任何一个字段都会列出字段中出现的热门词项。这些词项是根据它们词频的排名，取前五位而形成的词项列表。例如单击 `DestCountry` 字段会列出五个词项 `IT`、`US`、`CA`、`CN` 和 `JP`，在每个词项下面还有它们出现的百分比，这代表了航班的五大热门目的地国家。如图所示。



热门词项后面有一对类似于放大和缩小的图标，它们用于根据词项过滤文档。本书将称类似放大图标的按钮为包含按钮，而称类似缩小图标的按钮为排除按钮。这两个按钮在数据发现界面上的其他地方也会出现，作用是包含或排除词项值。具体来说，单击包含按钮会将当前字段包含该词项的文档过滤出来，而点击排除按钮则会将当前字段不包含该词项的文档过滤出来。单击这两个按钮会形成不同的过滤器，它们会自动出现在过滤器栏中。例如，选择 DestCountry 是 CN，而 Cancelled 为 false 的所有文档，会形成两个过滤器如图所示。



最后再来看一下文档栏。当鼠标指针悬停在每条文档的字段值上时，同样会出现包含按钮和排除按钮，它们也可以用于根据字段与词项生成过滤器。在每条文档前还有一个展开/收起开关，点击展开文档，可以看到当前文档的全部详细。这包括所有字段以及它们的值，还有对字段的操作等，如图所示。

The screenshot shows a detailed view of a flight document. At the top, there's a timestamp: Jul 1, 2020 @ 19:12:16.000. Below it, the document fields are listed:

	Value
DestCountry	CN
Dest	Xi'an Xianyang International Airport
FlightNum	C8VB0W9
OriginWeather	Cloudy
OriginCityName	Newport News
AvgTicketPrice	\$802.08
DistanceMiles	7,494.387
FlightDelay	false
DestWeather	Damaging Wind
FlightDelay1	Delay
OriginCountry	US
dayOfWeek	2
DistanceKilometers	12,061.047
timestamp	Jul 19:12:16.000
DestLocation	{ "lat": "34.447102", "lon": "108.751999" }
DestAirportID	

Below the table, there are three buttons: 'Expanded document' (with a file icon), 'View surrounding documents' (with a list icon), and 'View single document' (with a magnifying glass icon). At the bottom of the table section, there are two tabs: 'Table' (selected) and 'JSON'. Under the 'Table' tab, there's a 'Filter out value' button. The table rows are as follows:

#	AvgTicketPrice	\$802.08
<input checked="" type="radio"/> Cancelled	false	
<input type="radio"/> Carrier	ES-Air	
<input checked="" type="checkbox"/> Dest	Xi'an Xianyang International Airport	(highlighted)
<input type="checkbox"/> DestAirportID	XIY	
<input type="checkbox"/> DestCityName	Xi'an	
<input type="checkbox"/> DestCountry	CN	

除了包含和排除按钮以外，还有两个按钮。 按钮的作用是从文档栏的列表中添加或删除字段的开关，而 按钮则会添加一个字段存在性的过滤器。

## 文档的可视化

Kibana 可视化功能以图表形式展示 Elasticsearch 中的文档数据，能够让用户以最直观的形式了解数据变化的趋势、峰谷值或形成对比。这些图表根据查询条件从索引中提取文档，查询条件可以在文档可视化界面中定制，也可以使用在文档发现中保存的查询对象。文档可视化生成的图表也可以保存，本书后续章节称这些保存起来的可视化图表为可视化对象。在进入文档可视化界面时将会列出 Kibana 中所有的可视化对象，如果已经将 Kibana 提供的样例数据导入，则在这个页面上将列出几十个不同的可视化对象。可以逐一点开查看这些可视化对象，它们是学习文档可视化的好素材。在列表上方有搜索框，可根据名称搜索文档可视化对象，如图所示。

Title	Type
[Flights] Airline Carrier	Pie
[Flights] Airport Connections (Hover Over Airport)	Vega
[Flights] Average Ticket Price	Metric
[Flights] Controls	Controls
[Flights] Delay Buckets	Vertical Bar
[Flights] Delay Type	Area
[Flights] Delays & Cancellations	TSVB
[Flights] Destination Weather	Tag Cloud

在搜索框上面面有一个按钮，点击这个按钮会展示出所有可选的文档可视化类型，创建自定义的可视化对象就是从这里开始。文档可视化类型很多，包括折线图、饼图、仪表、表格等。因为很多，我们只挑选几个有代表性的进行学习。

New Visualization

Filter


+ Create visualization

## 二维坐标图

二维坐标图基于二维直角坐标系的 X/Y 轴绘制数据，包括折线图、面积图和柱状图三种。在二维直角坐标系中，X 轴也称横轴是自变量，而 Y 轴也称纵轴是因变量，Y 总是随 X 变化而变化。以折线图为例，它反映的就是 Y 随 X 变化的趋势。比如速度体现了距离随时间变化的趋势，那么 X 轴就应该是时间，而 Y 轴则是距离。

在 Kibana 中，二维坐标图的 Y 轴一般是一个或多个指标聚集，比如平均值、总数、极值等。而 X 轴则是桶型聚集，比如词项聚集、范围聚集等。所以指标聚集、桶型聚集的知识是学习可视化功能的基础。

## 面积图

下面使用 `kibana_sample_data_flights` 索引模式，创建个面积图来反映飞行距离与机票价格之间的关系。首先要确定飞行距离与机票价格哪一个是 X 轴，哪一个是 Y 轴。从数学角度来说，飞行距离是决定机票 价格的一个重要因素，所以飞行距离是自变量而机票价格则是因变量。从聚集查询的角度来说，机票平均价格使用指标聚集运算比较合适，而飞行距离使用桶型聚集更合适。所以从这两个角度来分析都可以使用飞行距离作为 X 轴，而使用机票价格作为 Y 轴。当然反过来也可以，但从逻辑上看比较奇怪。

在 `kibana_sample_data_flights` 中，`DistanceKilometers` 字段代表航班飞行距离，而 `AvgTicketPrice` 代表机票价格。接下来就来开始创建第 1 个可视化对象，飞行距离与机票价格 关系的面积图。先在新建可视化对象中选择第 2 个图形 `Area`，进入选择索引模式的界面，如图所示。

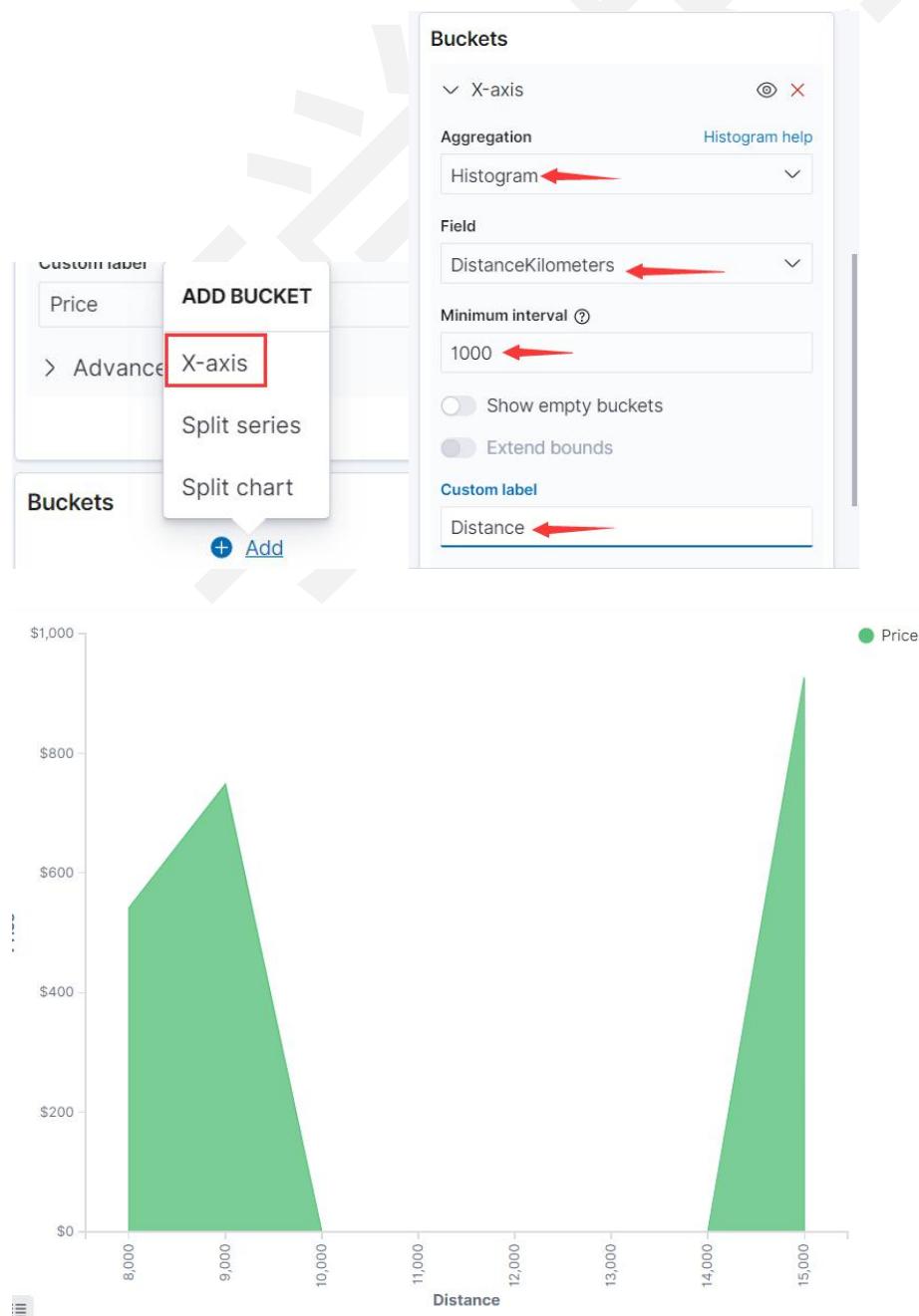
The screenshot shows the Kibana interface for creating a new visualization. On the left, there's a sidebar with icons for 'Lens', 'Area' (which is highlighted with a red box), 'Controls', and 'Data Table'. The main area has a title 'New Visualization' and a search bar labeled 'Search...'. Below the search bar is a list of index patterns: '[Flights] Flight Log', 'kibana\_sample\_data\_\*', 'kibana\_sample\_data\_flights' (which is also highlighted with a red box), 'kibana\_sample\_data\_logs', and 'pattern\_test4'. The 'kibana\_sample\_data\_flights' entry is currently selected.

单击 `kibana_sample_data_flights` 进入文档可视化创建界面。先在 Metrics 下面将 Y 轴设置为机票平均价格，即取 `AvgTicketPrice` 字段的平均值。单击 Y-Axis 展开表单，将 Aggregation 的默认聚集类型 Count 修改为 Average。接下来在 Field 选项中，选择字段为 `AvgTicketPrice`。表单中 Custom Label 用于定制 Y 轴的文字说明，比如输入 “Price”

This screenshot shows the 'Metrics & axes' configuration panel for the 'Area' visualization. It is under the 'Data' tab. The 'Metrics' section is expanded, showing the 'Y-axis' settings. Under 'Aggregation', 'Average' is selected. Under 'Field', 'AvgTicketPrice' is chosen. Under 'Custom label', the text 'Price' is entered. There are also 'Advanced' and 'Panel settings' tabs at the top right.

接下来再在 Buckets 下面将 X 轴设置为飞行距离, 即按 DistanceKilometers 字段的值分桶。首先点击 X Axis 展开表单, 将 Aggregation 下的聚集类型设置为 Histogram 接下来在 Field 选项中, 选择字段为 DistanceKilometers;Minimum Interval 则设置为 1000。 表单中 Custom Label 用于定制 X 轴的文字说明, 可输入 Distance。

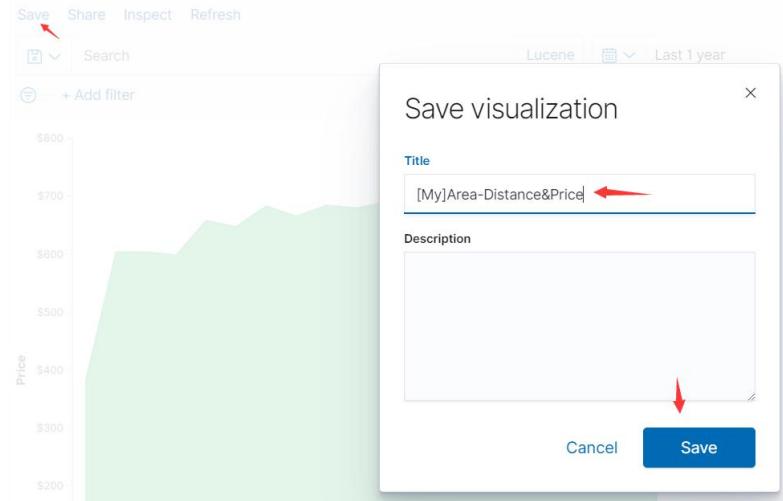
X 轴和 Y 轴这样设置的目的是将文档按 DistanceKilometers 字段以 1000km 为最小分隔单元划分成不同的组, 然后再对每一组文档的机票价格求平均值, 最后将它们绘制在界面上。设置结束后, 单击“Update”按钮, 右侧的图形就会将数据绘制出来, 如图所示



如果生成的面积图比较奇怪或者数据量比较少, 这极有可能是时间范围取得太小, 可将时间范围设置为 “This year” 以提取所有文档绘图。



创建好的可视化对象可以保存起来，单击工具栏的 **Save** 按钮，将面积图保存为“[My]Area-Distance&Price”。保存好的可视化对象，可以在进入可视化功能界面中看到。



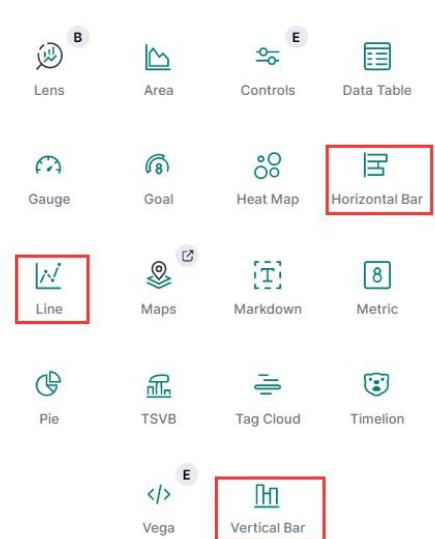
## 折线图和柱状图

由于折线图、柱状图与面积图对数据的需求完全一样，所以它们之间可以很容易做切换。在配置栏上点击 **Metrics&Axis** 链接，可以对指标和坐标做配置。如图所示。

The screenshot shows the 'Metrics & axes' configuration page. At the top, there are three tabs: 'Data', 'Metrics & axes' (which is highlighted with a red box), and 'Panel settings'. The main area is titled 'Metrics'. It contains a 'Value axis' dropdown set to 'LeftAxis-1'. Below it is a 'Chart type' section with a dropdown showing 'Area' (selected) and other options like 'Line', 'Area', 'Bar', and 'Stacked'. To the right of this is a 'Mode' dropdown set to 'Stacked'. Further down are sections for 'Y-axes' (containing 'LeftAxis-1 Price') and 'X-axis' (with a 'Position' dropdown set to 'Bottom').

Metrics&Axis 配置页分为 Metrics、Y-Axis、X-Axis 三部分，分别用于配置指标、Y 轴和 X 轴。在 Metrics 下有四个选项可以配置，其中的 Chart Type 有三个选项 line、area 和 bar，可以切换二维坐标图的类型。可自行选择切换类型，然后按“Update”按钮查看效果。

事实上在创建可视化对象时，可供选择的可视化对象类型中有单独的折线图和柱状图。要想创建保存独立的折线图或柱状图，可以在创建时选择 Line、Horizontal Bar 或 VerticalBar。由于它们创建的过程与面积图基本相同。

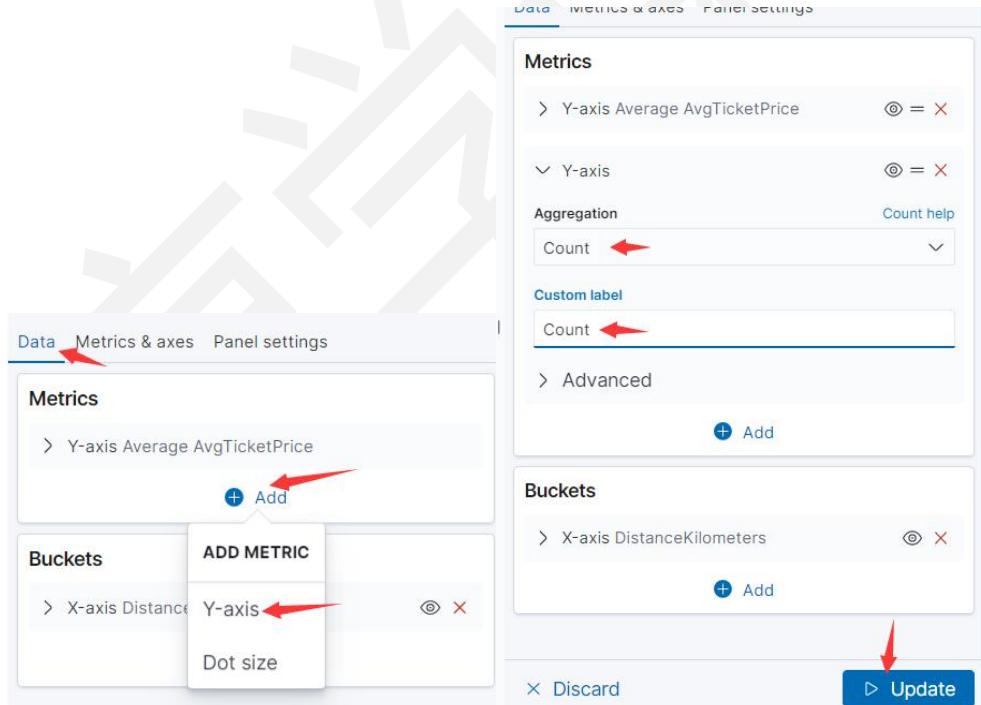


## 指标叠加

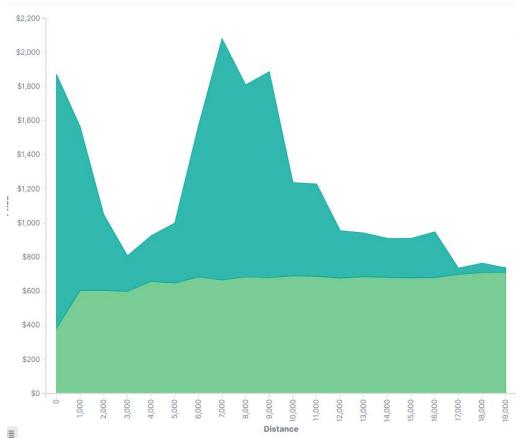
指标叠加是创建多个指标聚集，并且在一张二维坐标图中绘制出来，直观的表现就是在二维坐标中有多个图共享相同的坐标。下面就在飞行距离与机票价格的基础上，给可视化对象再添加一个指标，以体现不同飞行距离的航班数量。

重新打开 “[My]Area-Distance&Price”, 将它另存为 “[My]Area-Distance&Price & Count”。

在 Metrics 下面单击 Add 按钮为 Y 轴增加新的指标。在弹出的 Select metrics type 中选择 Y- Axis, 再选择指标聚集类型为 Count, 并为坐标添加说明标签 Count。如图所示：



设置好之后 Update, 会看到在面积图中出现了两种不同颜色的面积图，如图所示：

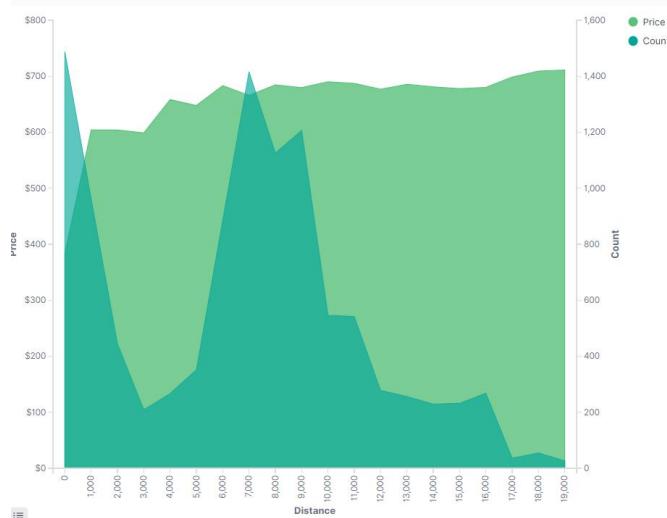
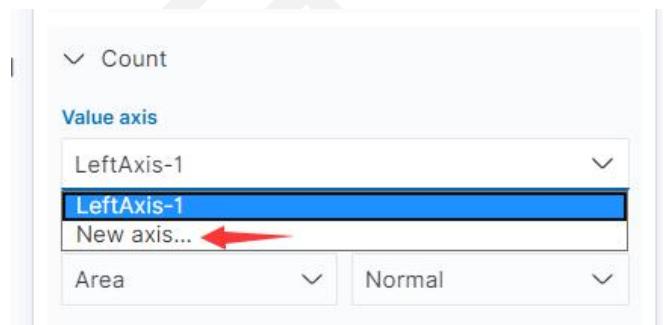


在默认情况下，叠加指标之间不会出现交叉，但可通过设置 Mode 参数修改。将配置页面切换回 Metrics&Axes 会看到 Metrics 在原来 Price 的基础上多了一个 Count。在 Mode 参数的下拉列表中分别选择 normal 看看。



在多指标的情况下每一个指标的设置都是独立的，可以分别给每个指标选择不同的图形，因此可以在一个图形中同时使用折线图、面积图和柱状图来表现不同的指标。

叠加指标时多个指标可以共享一个 Y 轴，也可以在 Metrics& Axes 配置页中设置图像使用多个 Y 轴。在 Metrics& Axes 配置页中将 Count 指标展开，在“Value Axis”选项中选择“New Axis”，这时在下面的 Y-Axes 中就会多出来一个新的 Y 轴，如图所示：



如果将两个指标的 Y 轴分列两边，则图形的 Mode 就只能是 **normal** 了，即使选择 **stacked** 也不会再起作用了。

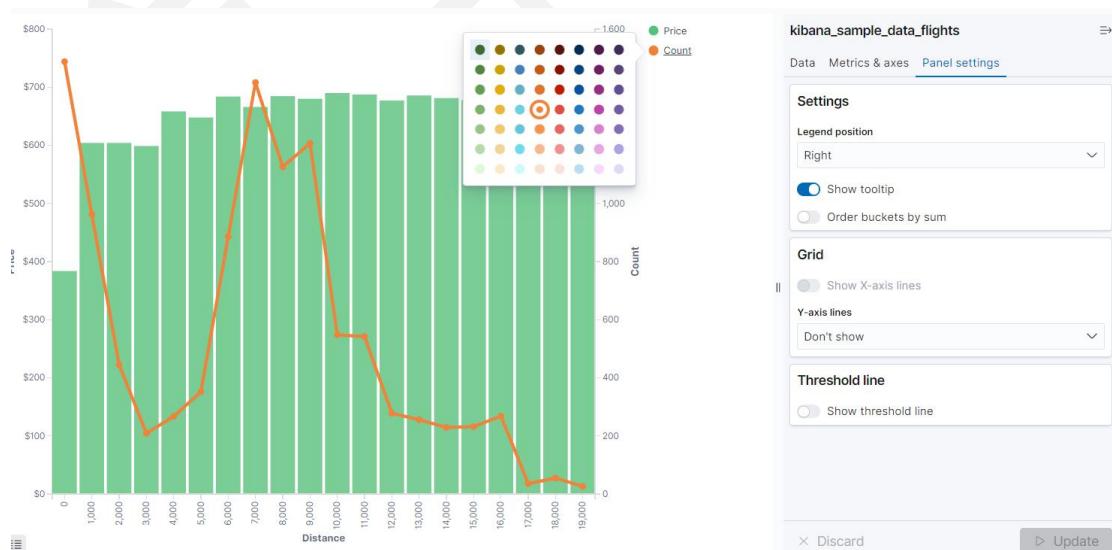
在配置栏最后一个标签页是“**Panel Settings**”，可以设置图例位置、提示、网格线等等配置。图侧默认情况下位于坐标图形的右上侧，它展示了图形中不同颜色图形对应的信息。不仅如此，单击这个图例还可以配置图形颜色，如图所示。

在图 10-11 中，**Price** 指标被设置为折线图，而 **Count** 则被设置为柱状图。同时，**Count** 指标通过图例修改为浅兰色，并通过 **Grid** 参数绘制了 Y 轴的网格线。

“**Panel Settings**”其余参数中，**Legend position** 用于设置图例位置，包括 **Top**、**Left**、**Right**、**Bottom** 四个选项，**Show tooltip** 则是图形提示信息的开关，当开启时鼠标指针悬停在图形拐点上时会出现提示信息，不过目前这个参数不起什么作用。

最后将这个可视化对象另存为 “[My]Line-Bar-Distance&Price & Count”。

它非常直观地体现了飞行距离与票价的关系，同时还体现了飞行距离在数量上的分布情况。



## 圆形与弧形图

二维坐标图是中规中矩的统计图形，它们可以严谨地体现出自变量与因变量之间的关系和趋势。在实际应用中，圆形与弧形也经常用来展示统计数据，但它们体现的往往是部分与整体之间的关系。

在 Kibana 中也包含了几种圆形或弧形的可视化对象，它们是饼图(Pie)、目标(Goal) 和仪表(Gauge)，本小节就来介绍这三种可视化对象。

### 饼图

饼图使用圆或圆环表示整个数据空间，并以扇面代表数据空间的某一部分，所以饼图在展示部分与整体的关系时非常直观。目标和仪表是两个不同的可视化对象，它们会将某一指标值的范围绘制在一个表盘中，并以指针或进度条的形

式显示该指标在仪表中的实际值。这两种对象一般用于监控某系统的运行状态，体现该系统在某一项指标上的健康状态。

饼图有两项变量需要定义，一是饼图要区分多少个扇面，另一个就是每个扇面有多大。这两个变量在 Kibana 中由桶型聚集和指标聚集来决定，所以创建 Kibana 饼图也同样包含 Metrics 和 Buckets 两个配置项。其中，Metrics 只能定义一个指标聚集，决定扇面大小；而 Buckets 则可以定义多个，决定扇面有多少个。

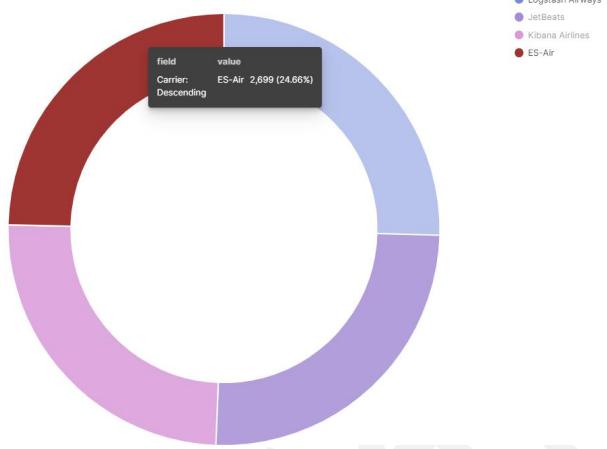
下面说明如何配置 Kibana 饼图。在创建可视化对象窗口中选择 Pie 并选择 kibana\_sample\_data\_flights 索引模式，进入创建饼图的配置界面。



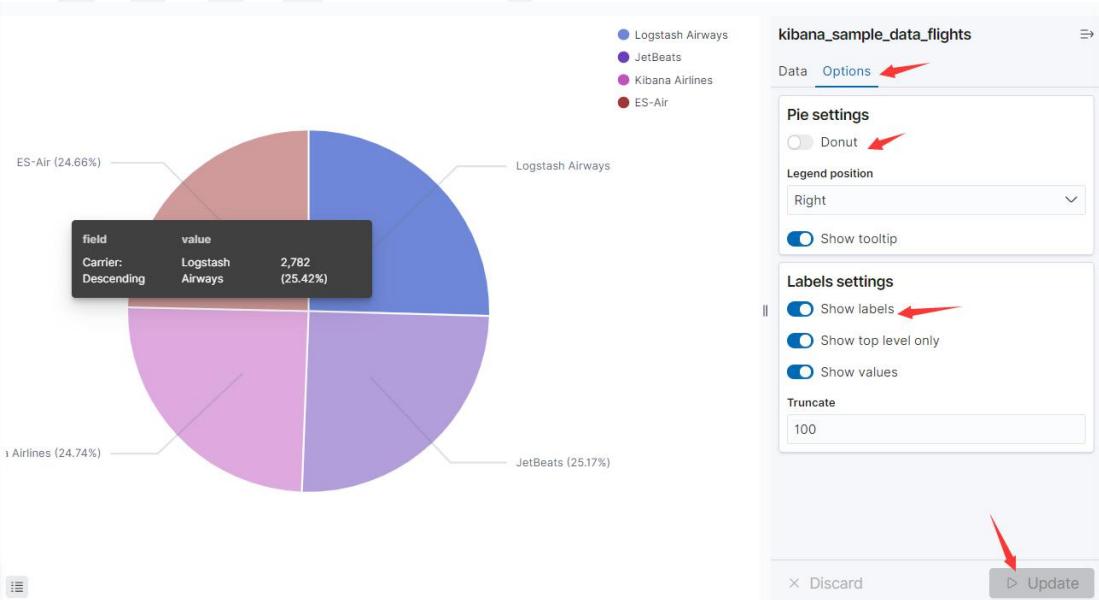
Pie

在 Metrics 配置项下单击 Slice Size 按钮展开表单，选择指标聚集为 Count，并在CustomLabel 中填写标识为“Count”。接下来选择 Buckets 下面的 Split Slices，并选择聚集为 Carrier 字段的 Terms 框型聚集。

The screenshot shows the Kibana visualization configuration interface for a Pie chart. The left panel displays the 'Metrics' section, where 'Slice size' is set to 'Count' under 'Aggregation'. The right panel shows the 'Buckets' configuration. Under 'Split slices', the 'Aggregation' dropdown is set to 'Terms' (indicated by a red arrow), and the 'Field' dropdown is set to 'Carrier' (also indicated by a red arrow). The 'Order by' section shows 'Metric: Count' with 'Descending' order and a size of 5. At the bottom, there are 'Discard' and 'Update' buttons, with 'Update' highlighted by a red arrow.



默认情况下，Kibana 饼图为圆环形，这可以通过在 Options 界面中修改 Donut 开关更改。勾选 Donut 选项时绘制的图形为圆环，而取消勾选 Donut 则绘制的图形为圆形。除了设置饼图的类型，还可以设置扇面的标识，即勾选 Show Labels 可以开启每个扇面的标识信息。最后，通过 Legend Position 下拉列表可以设置图例的位置，可选项包括 left、right、top、bottom，饼图被绘制为圆形，并包含了标识信息。将这个饼图保存为 “[My] Pie- Carrier&Count”。



## 饼图叠加

在添加桶型聚集时有两个选项，一个 is Split Slices，而另一个则是 Split Chart。与折线图类似，Split Chart 会以分割 X 轴或 Y 轴的形式添加子桶型，但只能添加一次且必须要先于 Split Slices 添加。上一小节的示例中选择了 Split Slices，这时如果再通过 Add sub buckets 添加子桶型时就不能再添加 Split Chart 了。

先将 “[My] Pie- Carrier&Count”另存为 “[My]Pie.Split.Chart-Carier&Count”。单击桶型聚集后面的叉子按钮删除原来的桶型聚集，

Buckets

Add

Buckets

> Split slices Carrier: Descending ⚙️ X

+ Add

ADD BUCKET

Split slices

Split chart

再以 Split Chart 方式添加 Carrier 字段的 Terms 聚集。然后再单击 Add subbuckets 添加子桶型，这时就只有一个 Split Slices 选项了，将子桶型设置为 DestCountry 的 Terms 聚集。

Buckets

> Split chart Carrier: Descendi... ⚙️ X

+ Add

Sub aggregation Terms

Field DestCountry

Order by Metric: Count

Order Descending Size 5

Group other values in separate bucket

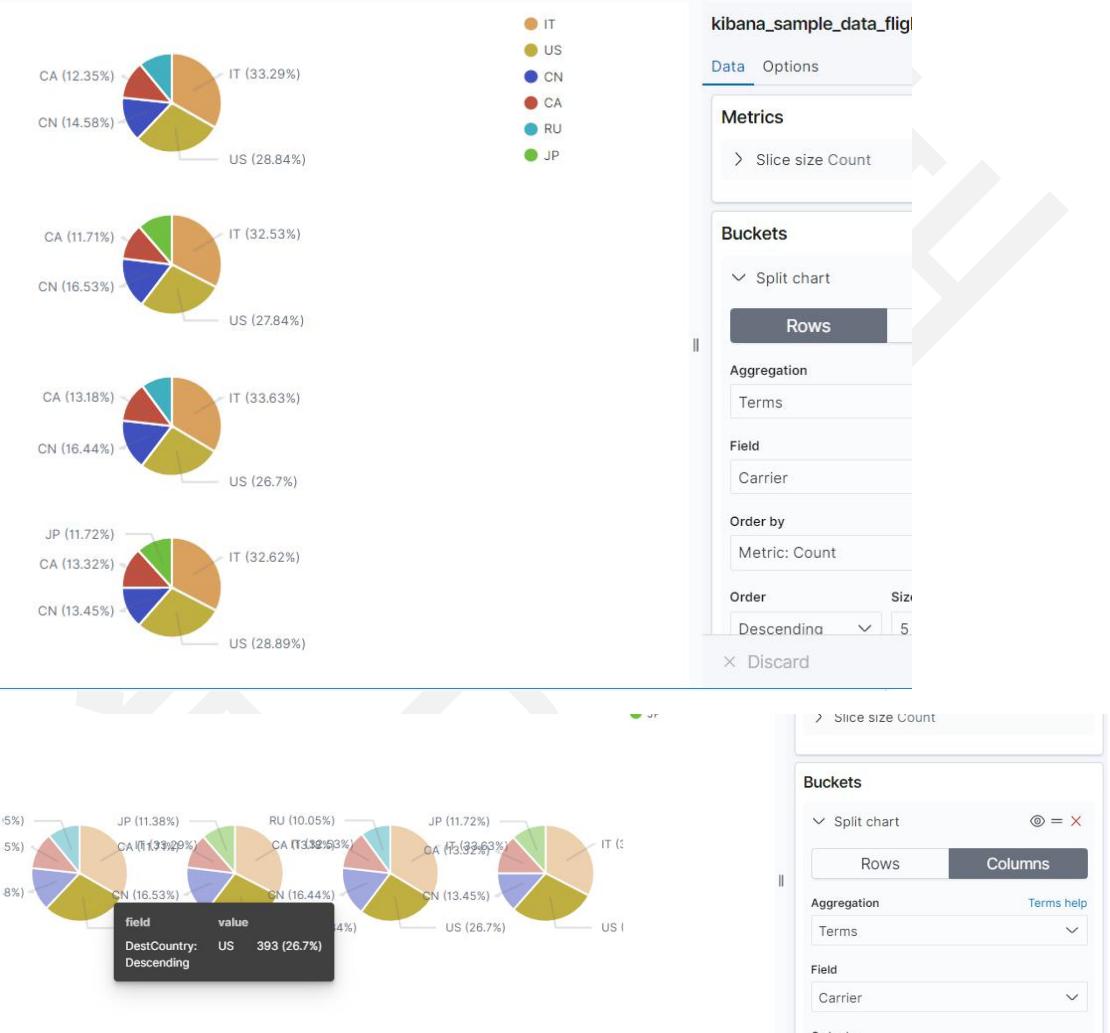
ADD SUB-BUCKET

Split slices

Split chart

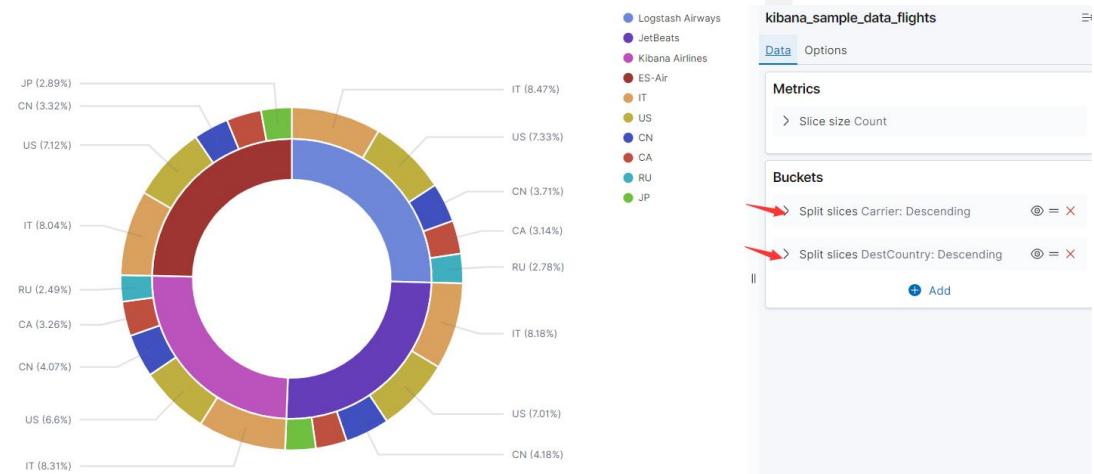
Discard Update

饼图的 Split Chart 也有 Rows 和 Columns 两个按钮，选择 Rows 时按行分割 Y 轴，而选择 Columns 则按列分割 X 轴。



上所示的饼图先按不同的航空公司(Carrier) 分割坐标，所以每一个饼图就代表一个航空公司。在每一个饼图中， 每个扇面代表的是航班的目的地国家占比。

与 Split Chart 只能添加一次不同，Split Slices 可以添加多次子桶型。新添加进来的子桶型，会对父桶型中每一个桶再次分桶， 展示出来的饼图是同心环或同心圆。将上面的可视化对象另存为 “[My] Pie. Split.Slices-Carrier&Count” ，将所有的桶型聚集删除后再使用两个 Split Slices 添加 Carrier 和 DestCountry 字段的 Terms 聚集，最终的效果如图所示。



在上图中使用两个圆环代表两个桶型聚集，内侧圆环分为 4 个扇面代表 4 个航空公司;而外侧圆环又将每个扇面划分为 10 个更小的扇面，代表 10 个不同的目的地国家。从这个饼图可以清楚地看出 4 个航空公司的航班数量占比，而每家航空公司在每个国家的市场份额也一目了然。

## 目标图

目标可视化对象体现的是指标值距离设定目标值之间的差距，实际值越高越接近目标值说明系统运行越好。比如，设定一个年度销售额目标，通过目标可视化对象就可以体现出当前实际销售额与目标销售额之间的对应关系。同样，仪表可视化对象也可以体现指标值在仪表中的实际位置，但体现的往往是系统某一项指标是否处于安全区间。因此，仪表会给不同的数值区间赋予不同的颜色，值越高颜色越深。目标和仪表一般使用弧形，也可以配置为圆环形状。

比如我们设置 **Kibana Airlines** 航空公司每月航班数量的目标为 3000 次，来生成一个目标可视化对象。由于这次要生成的可视化对象要过滤航空公司，所以在配置目标可视化对象前要先配置检索。在创建可视化对象中选择 **Goal** 对象，然后选择 **kibana\_sample\_data\_flights** 索引模式。



Goal

进入 **Goal** 对象配置界面后，先给可视化对象添加过滤器。单击 **Add a filter** 并设置过滤器为 **Carrier is Kibana Airlines**，单击 **Save** 按钮保存过滤器。

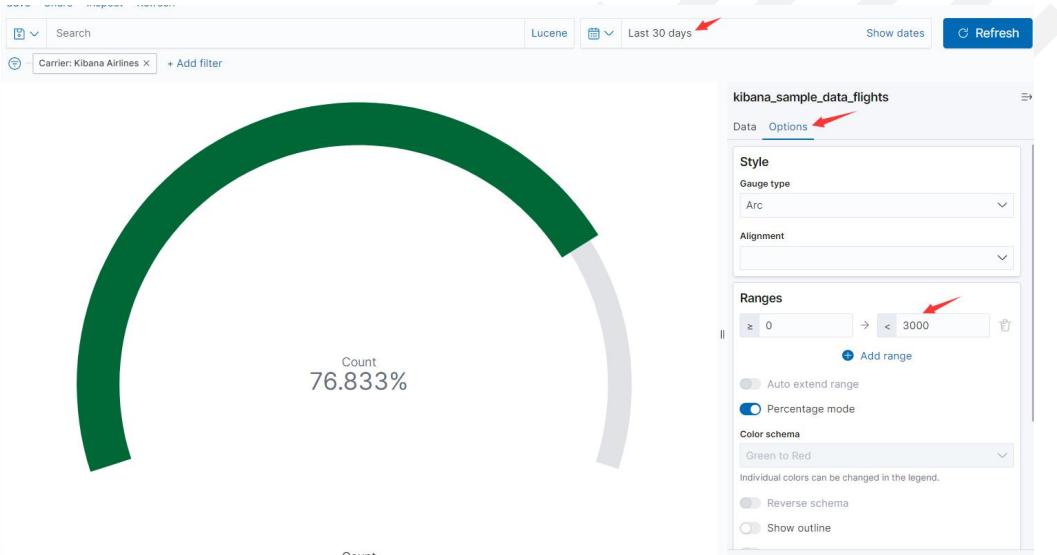
+ Add filter

**EDIT FILTER** Edit as Query DSL

Field	Operator
Carrier	is
Value	
Kibana Airlines	

同时还要注意，由于图形展示的目标是月度航班数量，所以要将时间范围设置为月。

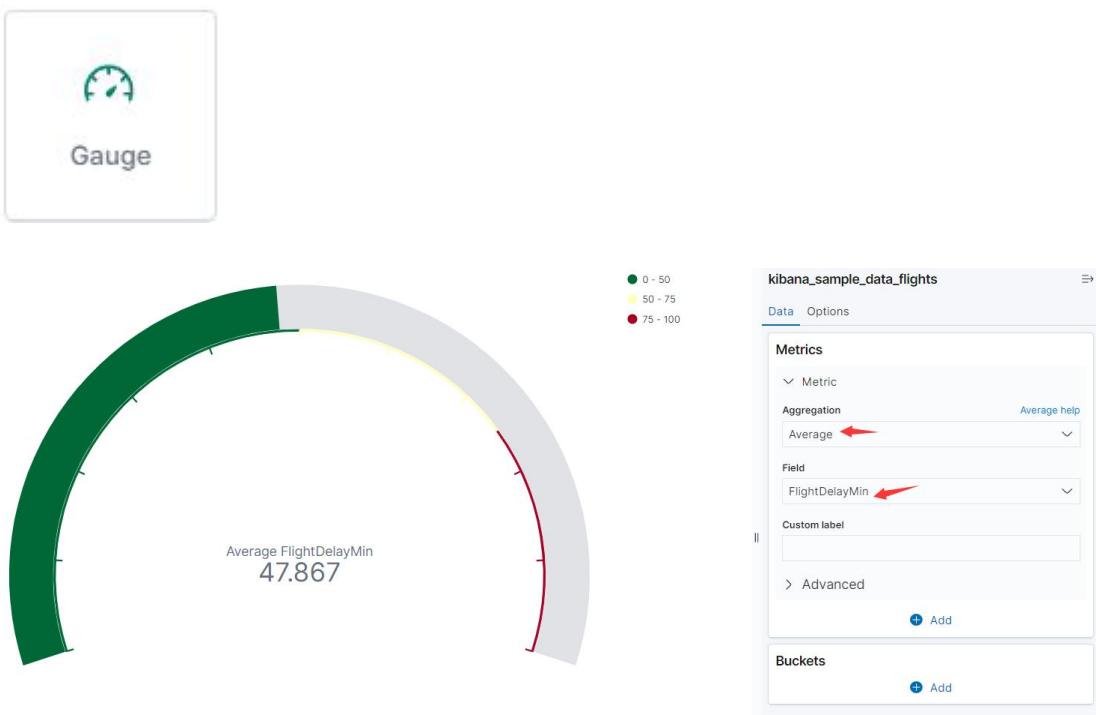
接下来，在 Options 界面中设置目标值为 3000 次。找到 Ranges 选项，将“To”设置为“3000”，这时目标对象会按 3000 目标来计算百分比，如图所示：



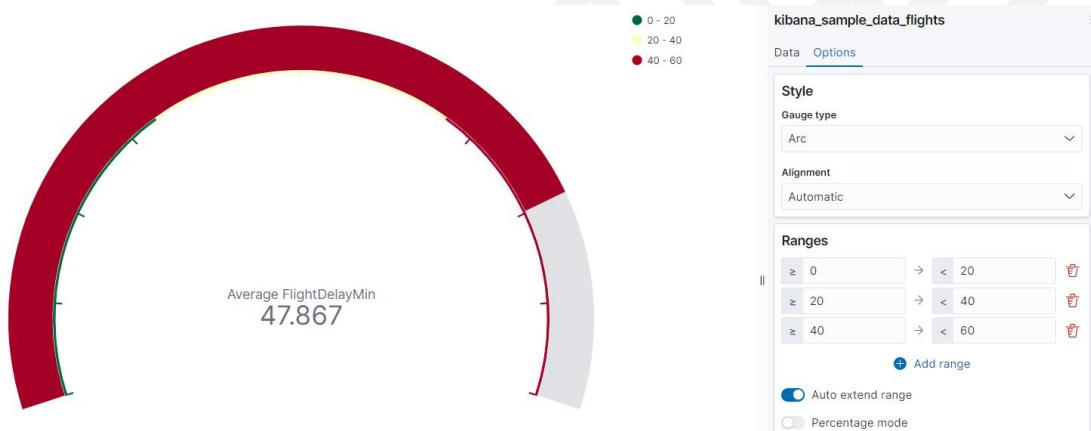
将它保存为 “[My] Goal- Count”。Ranges 选项可以设置为多个，生成的目标对象会根据实际值选择不同颜色，但这对于目标对象来说意义并不大。。

## 仪表图

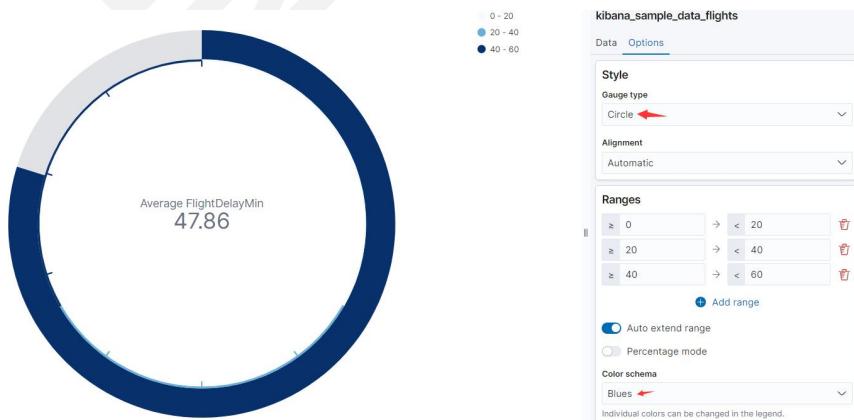
以航空公司的航班延误时间为指标，生成一个航班延误时间的仪表。选择 Gauge 对象及 kibana\_sample\_data\_flights 索引模式后，在配置页面上将指标聚集设置为 FlightDelayMin 的 Average 聚集。



在仪表中看到平均延误时间为 **47.867**，且标识该值的进度条为绿色。这是因为 Kibana 默认将整个取值范围设置为 100，并且设置了 0~50、50~75 和 75~100 三个区间，它们对应的颜色分别为绿色、黄色和红色。可以到 Options 配置页面，将延误取值范围修改为 1 个小时，并设置 0~20、20~40 和 40~60 三个区间：



修改范围区间后，由于当前平均延误时间已经达到第三个区间，进度条变为红色。有关颜色的变化，可以通过修改“Color Options”变更。另外在 Options 上还有其他一些 选项可以配置目标和仪表对象，比如可以通过“Gauge Type”将它们的图像设置为完整的圆形。



修改完成后，将可视化对象保存为 “[My] Gauge- Delay”。

## 热力图

热度原本是指某一事物的冷热程度，但在互联网和大数据时代它越来越多地代表某种资源的使用频度，资源使用得越多，它的热度也就越高。比如在搜索引擎中的热词，在地图中展示的交通流量、热点区域等。Kibana 有几种与热度相关的可视化对象。比如热力图、标签云等，我们来了解下热力图。

热力图通常是在一个可区分为不同区域的图形上，以不同颜色标识某一指标值在不同区域的高低分布情况。比如在地图的不同区域以不同颜色标识交通流量、人口分布等情况，交通流量越大、人口密度越高它们所在区域的热度也就越高，区域对应的颜色也就越深。

Kibana 热力图与此类似，但它并不是绘制在地图上，而是绘制在二维坐标上。与前面介绍的几种二维坐标图不同，Kibana 热力图的 X 轴和 Y 轴都是桶型聚集。它们相互交叉形成一个可以用热度值标识的二维矩阵，而热度值则由另一个独立的指标聚集来定义。

所以，在 Kibana 热力图中至少要定义三个聚集，即 X 轴的桶型聚集、Y 轴的桶型聚集和热度值对应的指标聚集。Kibana 在绘制热力图时，会根据 X 轴和 Y 轴对应的两个桶型聚集，分别计算它们所对应的指标聚集，然后再在热力图矩阵中以不同的颜色绘制出来。下面还是以飞行距离与机票价格的对应关系为基础，并添加航空公司这一桶型聚集，来看一下飞行距离、航空公司在机票价格上形成的热力图。



首先在创建文档可视化的弹出窗口中选择 **Heat Map**，并在接下来的索引模式中选择 `kibana_sample_data_fights`。热力图的配置栏也包含 **Metrics** 和 **Buckets** 两个配置项，点击 **Metrics** 下的 **Value** 按钮，将热度对应的指标配置为 `AvgTicketPrice` 的平均值聚集：

A screenshot of the Kibana Metrics configuration panel. It has tabs for 'Data' (selected) and 'Options'. Under the 'Metrics' section, there is a dropdown menu for 'Value' which is currently set to 'Average'. A red arrow points to the 'Average' option. Below this, there is a 'Field' dropdown containing 'AvgTicketPrice', also with a red arrow pointing to it. There is also a 'Custom label' input field.

接下来再来配置 X 轴和 Y 轴的桶型聚集。在 **Buckets** 下先选择 **X-Axis**，将 X 轴设置为 `DistanceKilometers` 字段的 `Histogram` 桶型聚集，`Minimum Interval` 设置为 1000。

## Buckets

X-axis

Aggregation: Histogram (arrow)

Field: DistanceKilometers (arrow)

Minimum interval: 1000 (arrow)

这之后再点击 Add sub- buckets

ADD SUB-BUCKET

X-axis

Y-axis (arrow)

Split chart

+ Add

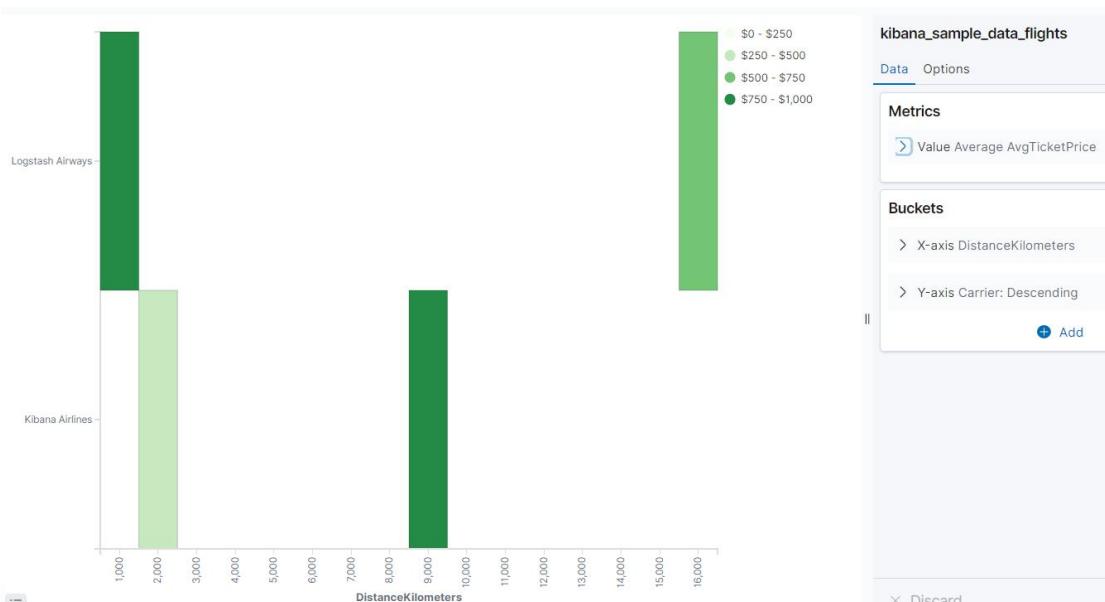
Y-axis

Sub aggregation: Terms

Field: Carrier

Order by: Metric: Average AvgTicketPrice

选择 Y- Axis 设置 Y 轴的桶型聚集为 Carrier 字段的 Terms 聚集:



这个热力图看上去不是那么经典，这主要是因为指标值的范围比较集中，所以绘制出来的热力图层次感并不强。但在热力图配置的 Options 页面，可以设置包括热度颜色种类、指标数量或范围等内容，通过这些配置可以让热力图看起来更能反映实际情况。

---

设置结束后单击 Save 按钮，以 "[My] Heat Map-Distance& Carrier&Price" 为名称保存热力图对象。

## 表格

尽管表格在文档可视化中的应用越来越少，但它在展示具体数值时具有一定优势，所以在一些特定场景下表格是不可或缺的可视化元素。

Kibana 中表格对应的可视化对象为 Data Table。



表格由行和列组成，是一种通用的展现关系型数据的形式，所以要想绘制表格必须先确定行和列分别由什么来确定。在 Kibana 的表格中，行由桶型聚集决定，而列则由指标聚集决定。

下面来创建一个表格，并以航空公司为行分别统计它们的航班数量、平均票价、平均飞行距离等，这将形成 4 行 3 列的表格(不包含表头)。在创建可视化对象页面选择 Data Table，然后选择 kibana\_sample\_data\_fights 索引模式进入表格配置界面。表格配置栏包含 Metrics 和 Buckets 两栏，Metric 相当于列可以设置多个，Buckets 相当于行也可以设置多个。

先添加三个指标聚集，分别为 Count 聚集、AvgTicketPrice 字段的 Average 聚集和 DistanceKilometers 字段的 Average 聚集，

A screenshot of the Kibana Metrics configuration interface. It shows three items listed under 'Metrics': 'Metric Count' (with a red 'X' next to the remove button), 'Metric Average AvgTicketPrice' (with a red 'X'), and 'Metric Average DistanceKilometers' (with a red 'X'). Each item has a small icon to its left.

Buckets 设置为 Carrier 的 Terms 聚集。

A screenshot of the Kibana Buckets configuration interface. It shows a section titled 'ADD SUB-BUCKET' with two options: 'Split rows' (which has a red arrow pointing to it) and 'Split table'. Below this, there's a table with two rows. The first row has columns for 'Aggregation' (set to 'Terms') and 'Field' (set to 'Carrier'). The second row has a 'Terms help' button. There are also 'Split rows' and 'Split table' buttons at the bottom of the table area.

设置完成生成图所示的表格：

Carrier: Descending	Count	Average AvgTicketPrice	Average DistanceKilometers
Kibana Airlines	558	\$655.2	7,127.635
Logstash Airways	557	\$621.25	7,215.139
JetBeats	523	\$624.08	7,139.108
ES-Air	521	\$641.08	7,122.974

如果想给表格增加更多列，可以在 Metrics 配置项中继续添加指标聚集。表格可按每一列排序，当鼠标指针悬停于每一行的表头时还会出现排除和包含按钮。单击它们可以在可视化对象中添加过滤器，表格中的数据也会跟着发生变化。将这个表格保存为 “[My]Table-Carrier”。

在添加 Buckets 时，有“Split Rows”和“Split Table”两个选择。“Split Rows”在添加子桶型时会将父桶型按行分割，后者则会直接将表格分割成多个。为上述表格再添加一个 DestCountry 的 Terms 聚集，在使用“Split Rows”形成的表格如图所示：

“Split Table”会将表格分割成多个，而且也是只可以添加一次。同样是为表格添加一个 DestCountry 的 Terms 聚集，但这次使用“Split Table”，如图所示。

The screenshot shows two tables side-by-side. The left table is titled "IT: DestCountry: Descending" and the right table is titled "kibana\_sample\_data\_flights". Both tables have columns: Carrier, Count, Average AvgTicketPrice, and Average DistanceKilometers. The right panel shows the configuration for these tables, including sections for Metrics and Buckets.

IT: DestCountry: Descending			
Carrier: Descending	Count	Average AvgTicketPrice	Average DistanceKilometers
ES-Air	106	\$579.97	5,851.603
JetBeats	89	\$582.47	5,944.811
Kibana Airlines	110	\$611.63	6,195.722
Logstash Airways	95	\$622.92	6,149.416

US: DestCountry: Descending			
Carrier: Descending	Count	Average AvgTicketPrice	Average DistanceKilometers
Logstash Airways	88	\$578.67	7,205.714
ES-Air	77	\$602.38	8,015.239
JetBeats	74	\$614.46	7,823.053
Kibana Airlines	78	\$619.18	7,607.905

**kibana\_sample\_data\_flights**

Data Options

**Metrics**

- > Metric Count  =
- > Metric Average AvgTicketPrice  =
- > Metric Average DistanceKilometers  =

**Buckets**

- > Split rows Carrier: Descending  =
- > Split table DestCountry: Descending  =

默认情况下表格在展示时会分页，在 Options 标签页中可以设置每页展示数据的数量，默认值为 10。

The screenshot shows the "Options" tab selected. It has a field labeled "Max rows per page" with a value of "10".

## 仪表盘

仪表盘(Dashboard)是 Kibana 提供的综合展示数据的功能，在 Kibana 中保存的可视化对象可以在仪表盘中组合起来共同展示。

仪表盘是位于导航栏的第三个功能，如果已经导入了 Kibana 样例数据，进入仪表盘界面后会看到已经保存的仪表盘对象，如图所示：

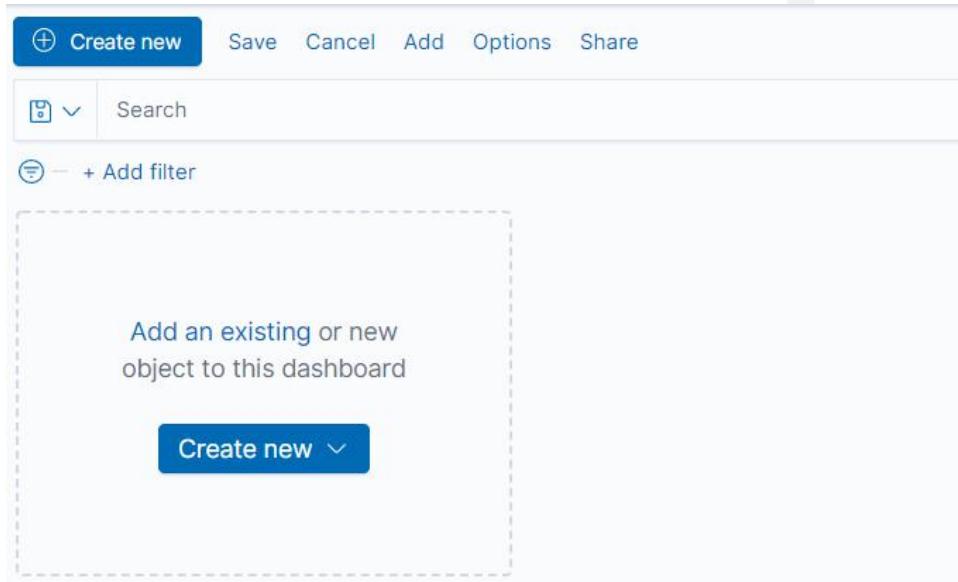
The screenshot shows the "Dashboards" section. It includes a search bar and a list of saved dashboards:

- Title
- [Flights] Global Flight Dashboard
- [Logs] Web Traffic

图中展示的仪表盘对象是学习仪表盘对象非常好的素材，可自行查看学习。如果想要创建新的仪表盘对象，可单击 Create new dashboard 按钮。

事实上，如果我们根据前面的章节里保存了查询对象和可视化对象，创建一个仪表盘还是非常简单的。

单击 Create dashboard 按钮进入创建仪表盘界面，仪表盘界面与文档发现、文档可视化界面类似，也包含有工具栏、查询栏、时间栏和过滤器栏，只是工具栏中的按钮有些不同。



创建仪表盘的过程很简单，单击工具栏中的 **Add** 按钮或者是在展示区域中的 **Add** 链接会弹出 **Add Panels** 对话框，其中包含了已经保存的可视化对象和查询对象列表。用户可在其中挑选希望加入仪表盘的对象，也可以单击 **Add new Visualization** 按钮创建可视化对象，如图所示。

Add panels

Search... Sort Types 4 Create new Add panels

Flights Airline Carrier  
Flights Airport Connections (Hover Over Airport)  
Flights Average Ticket Price  
Flights Controls  
Flights Delay Buckets  
Flights Delay Type  
Flights Delays & Cancellations  
Flights Destination Weather  
Flights Flight Cancellations  
Flights Flight Count and Average Ticket Price

My Line-Bar-Distance&Price &Count  
My Area-Distance&Price &Count  
My Line-Distance&Price  
My Area-Distance&Price  
My Gauge- Delay  
My Pie. Split.Slices-Carrier&Count  
My Pie.Split.Chart-Carier&Count  
My Pie- Carrier&Count  
My Heat Map-Distance& Carrier&Price  
My Table-Carrier

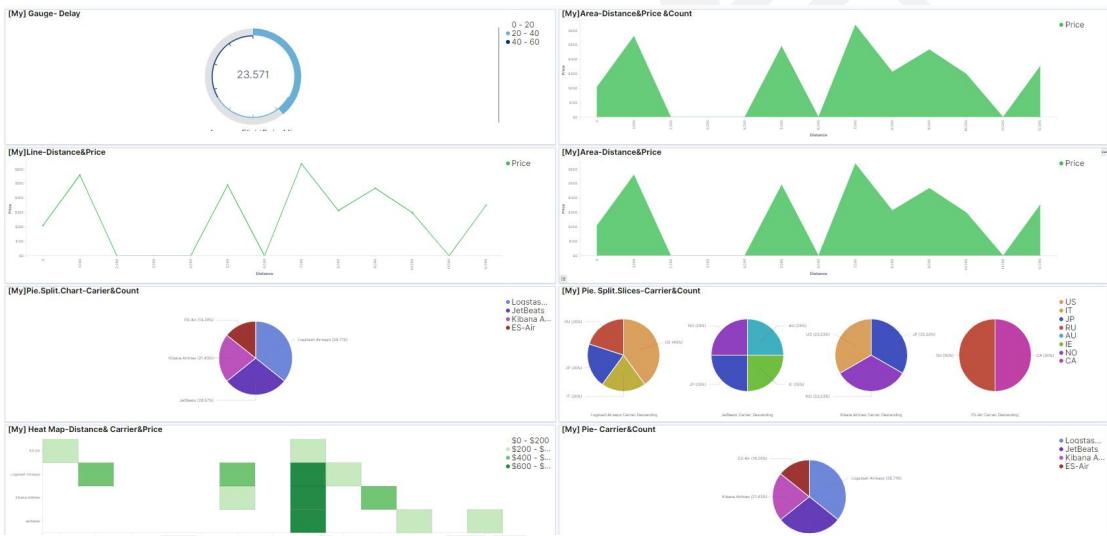
Rows per page: 10 1 2 3 4 5 >

在 **Add Panels** 对话框中还有一个查询栏，输入“**My**”关键字搜索我们前面保存的可视化对象，如图所示。

单击某个可视化对象名称，控件就被添加到仪表盘中了。按相同的方式选择所有感兴趣的可视化对象将它们依次添加到仪表盘中，它们将按添加顺序依次出现在仪表盘中。在每一个面板的右上角都有一个设置按钮 ，单击这个按钮会弹出一个选项对话框，通过这个对话框的选项可以实现对每个面板的定制。在每一个面板的右下角还有一个 形状的按钮，可通过这个按钮调节面板大小。

另外，使用鼠标左键按住面板的标题部分还可以在仪表盘中移动面板，调整面板在仪表盘中的位置。

仅仅通过鼠标进行几下简单的点击，一个像模像样的仪表盘就展现出来了。



仪表盘也可以一样保存起来，单击 **Save** 按钮，在弹出对话框中将它保存为“[My] Dashboard-Flights”。

## LogStash

**LogStash** 是一个类似实时流水线的开源数据传输引擎，它像一个两头连接不同数据源的数据传输管道，将数据实时地从一个数据源传输到另一个数据源中。在数据传输的过程中，**LogStash** 还可以对数据进行清洗、加工和整理，使数据在到达目的地时直接可用或接近可用，为更复杂的数据分析、处理以及可视化做准备。

既然需要将数据搬运到指定的地点，为什么不在数据产生时就将数据写到需要的地方呢？这个问题可以从以下几个方面理解。首先，许多数据在产生时并不支持直接写入到除本地文件以外的其他数据源。比如大多数第三方软件在运行中产生的日志，都以文本形式写到本地文件中。其次，在分布式环境下许多数据都分散在不同容器甚至不同机器上，而在处理这些数据时往往需要将数据收集到一起统一处理。最后，即使软件支持将数据写入到指定的地点，但随着人们对数据理解的深入和技术的诞生又会有新的数据分析需求出现，总会有一些接入需求是原生软件无法满足的。综上，**LogStash** 的核心价值就在于它将业务系统与数据处理系统隔离开来，屏蔽了各自系统变化对彼此的影响，使系统之间的依赖降低并可独自进化发展。

**LogStash** 可以从多个数据源提取数据，然后再清洗过滤并发送到指定的目标数据源。目标数据源也可以是多个，而且只要修改 **LogStash** 管道配置就可以轻松扩展数据的源头和目标。这在实际应用中非常有价值，尤其是在提取或发送的数据源发生变更时更为明显。比如原来只将数据提取到 **Elasticsearch** 中做检索，但现在需要将它们同时传给 **Spark** 做实时分析。如果事先没有使用 **LogStash** 就必须设计新代码向 **Spark** 发送数据，而如果预先使用了 **LogStash** 则只需要在管道配置中添加新的输出配置。这极大增强了数据传输的灵活性。

## 安装 Logstash

Logstash 是基于 Java 生态圈的语言开发的，所以安装 Logstash 之前需要先安装 JDK。Logstash 7 目前只支持 Java8 或 Java 11，所以安装前要检查安装 JDK 的版本是否正确。Logstash 提供了 DEB 和 RPM 安装包，还提供了 tar.gz 和 zip 压缩包，也可以直接通过 Docker 启动 Logstash。安装过程比较简单，直接解压缩文件是最简单的办法。

```
[elk@android0318 teaching]$ tar -xvf logstash-7.7.0.tar.gz
logstash-7.7.0/
logstash-7.7.0/tools/
logstash-7.7.0/bin/
logstash-7.7.0/config/
logstash-7.7.0/logstash-core/
logstash-7.7.0/logstash-core-plugin-api/
logstash-7.7.0/CONTRIBUTORS
logstash-7.7.0/NOTICE.TXT
logstash-7.7.0/lib/
```

```
[elk@android0318 teaching]$ ls
elasticsearch-2                               kibana-7.7.0-linux-x86_64           logstash-7.7.0.tar.gz
elasticsearch-7.7.0                           kibana-7.7.0-linux-x86_64.tar.gz
elasticsearch-7.7.0-linux-x86_64.tar.gz        logstash-7.7.0
```

## 启动 Logstash

Logstash 的启动命令位于安装路径的 bin 目录中，直接运行 logstash 不行，需要按如下方式提供参数：

```
./logstash -e "input {stdin {}} output {stdout{}}"
```

启动时应注意：-e 参数后要使用双引号。如果在命令行启动日志中看到“Successfully started Logstash API endpoint l:port= >9600”，就证明启动成功。

```
[2020-07-06T23:07:38,189] [INFO ] [logstash.agent]      ] Successfully started Logstash API endpoint {:p=>9600}
```

在上面的命令行中，-e 代表输入配置字符串，定义了一个标准输入插件(即 stdin)和一个标准输出插件(即 stdout)，意思就是从命令行提取输入，并在命令行直接将提取的数据输出。如果想要更换输入或输出，只要将 input 或 output 中的插件名称更改即可，这充分体现了 Logstash 管道配置的灵活性。按示例启动 Logstash，命令行将会等待输入，键入“Hello, World!”后会在命令行返回结果如下：

```
/home/elk/elk/teaching/logstash-7.7.0/vendor/bundle/jruby/2.5.0/gems/awesome_print-1.7.0/lib/awesome_print
/formatters/base_formatter.rb:31: warning: constant ::Fixnum is deprecated
{
    "host" => "xiangxue",
    "@timestamp" => 2020-07-06T15:10:28.835Z,
    "@version" => "1",
    "message" => "Hello World!"
```

在默认情况下，stdout 输出插件的编解码器为 rubydebug，所以输出内容中包含了版本、时间等信息，其中 message 属性包含的就是在命令行输入的内容。试着将输出插件的编码器更换为 plain 或 line，则输入的结果将会发生变化：

```
./logstash -e "input {stdin {}} output {stdout{codec => plain}}"
```

---

```
Hello World!
2020-07-06T15:19:31.338Z xiangxue Hello World!
```

## 连接 Elasticsearch

**Logstash** 基于插件开发和应用，包括输入、过滤器和输出三大类插件。输入插件指定了数据来源，过滤器插件则对数据做过滤清洗，而输出插件则指定了数据将被传输到哪里。在示例 1-5 中启动 **Logstash** 时，是通过命令行并使用-e 参数传入了配置字符串，指定了标准输入 **stdin** 插件和标准输出 **stdout** 插件。但在实际应用中，通常都是使用配置文件指定插件。

配置文件的语法形式与命令行相同，要使用的插件是通过插件名称来指定。例如，想要向 **Elasticsearch** 中发送数据，则应该使用名称为 **elasticsearch** 的输出插件。在 **Logstash** 安装路径下的 **config** 目录中，有一个名为 **logstash-sample.conf** 的文件，提供了配置插件的参考。

这个文件配置的输入插件为 **beats**,输出插件为 **elasticsearch**。复制这个文件并重命名为 **std\_es.conf** ( 也可以直接创建空文件), 下面通过修改这个文件配置一个从命令行提取输入，并传输到 **Elasticsearch** 的 **Logstash** 实例。按如下内容修改 **std\_es.conf** 文件的输入输出插件:

```
input {
    stdin {
    }

}

output {
    elasticsearch {
        hosts => ["http://172.18.194.140:9200"]
        index => "mystdin"
        #user => "elastic"
        #password => "changeme"
    }
}
```

在上面的配置中，**elasticsearch** 输出插件中的 **hosts** 参数指定了 **Elasticsearch** 地址和端口，**index** 参数则指定了存储的索引。**Elasticsearch** 索引不需要预先创建，但要保证它启动的地址和端口与配置文件中的一致。按如下方式启动 **Logstash**:

```
logstash -f ./config/std_es.conf
```

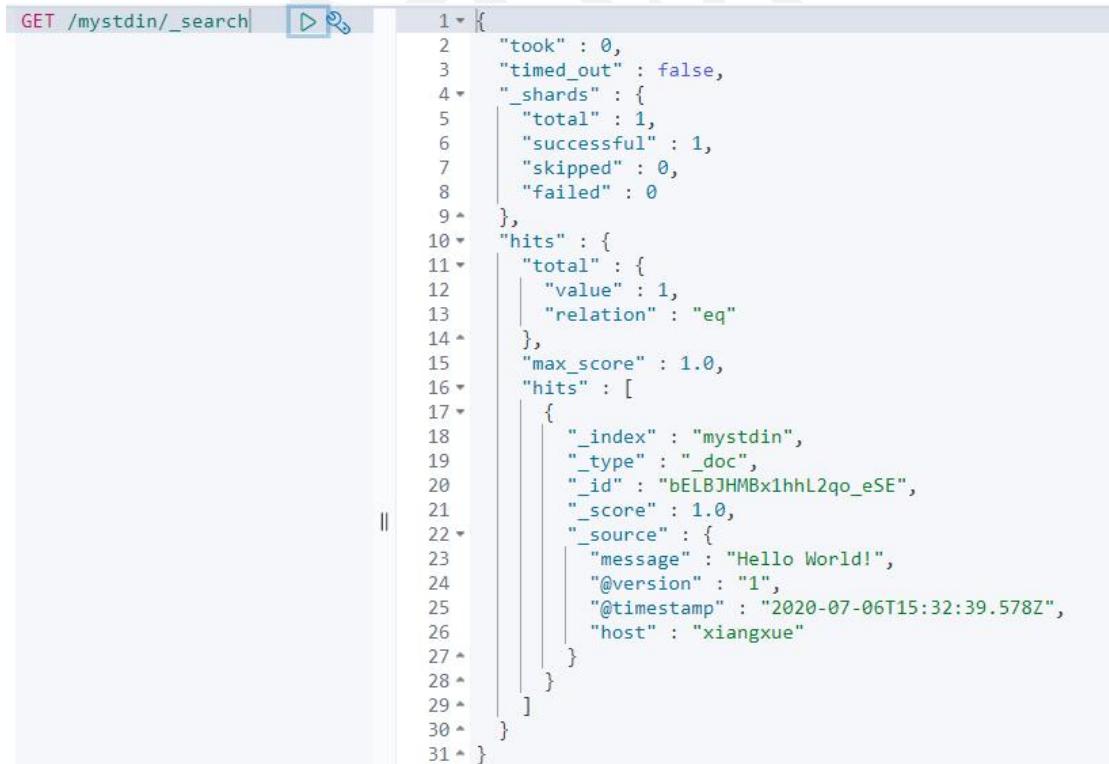
其中，-f 参数后面指定了配置文件的路径。启动后如果没有异常，可在命令行窗口中输入"Hello, World!" 等任意字符串，**Logstash** 将把输入内容提取到指

定的 Elasticsearch 服务中。通过 Kibana 开发工具输入 GET \_cat/indices，可以看到 mystdin 这个索引已经创建出来了。



```
1 GET _cat/indices ▶
2
3 1 green open high_sdk_test
4 2 green open pattern_custom
5 3 green open students
6 4 green open study_routel
7 5 green open logs_1
8 6 green open test_es
9 7 green open logs_2
10 8 green open kibana_sample_data_flights
11 9 green open test1
12 10 green open logs_3
13 11 green open test-user
14 12 green open logs_4
15 13 green open high_sdk
16 14 green open mystdin
17 15 green open .apm-custom-link
18 16 green open .monitoring-es-7-2020.07.01
19 17 green open .kibana_task_manager_1
20 18 green open .monitoring-es-7-2020.07.02
21 19
22 20
23 21
24 22
25 23
26 24
27 25
28 26
29 27
30 28
31 29
```

输入 GET /mystdin/\_search，则返回结果为



```
1 GET /mystdin/_search ▶
2
3 {
4   "took": 0,
5   "timed_out": false,
6   "_shards": {
7     "total": 1,
8     "successful": 1,
9     "skipped": 0,
10    "failed": 0
11  },
12  "hits": {
13    "total": {
14      "value": 1,
15      "relation": "eq"
16    },
17    "max_score": 1.0,
18    "hits": [
19      {
20        "_index": "mystdin",
21        "_type": "_doc",
22        "_id": "bELBJHMBx1hhL2qo_eSE",
23        "_score": 1.0,
24        "_source": {
25          "message": "Hello World!",
26          "@version": "1",
27          "@timestamp": "2020-07-06T15:32:39.578Z",
28          "host": "xiangxue"
29        }
30      }
31    ]
32  }
33 }
```

通过上图可以看到，Elasticsearch 为输入创建了一个 mystdin 索引，在命令行输入的“Hello, World!”已经被索引，在返回结果的 \_source 字段上可以看到文档内容。

## Logstash 体系结构

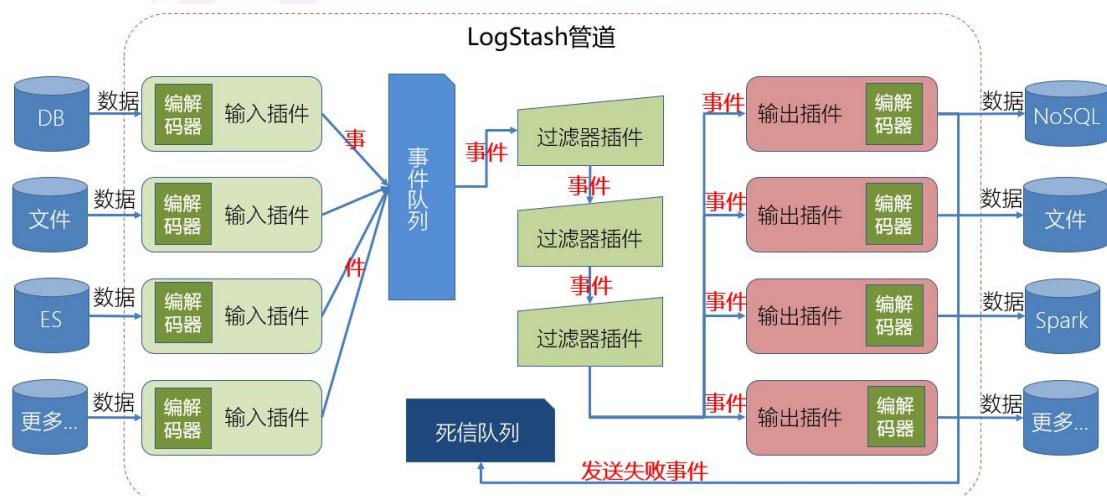
Logstash 数据传输管道所具备的流水线特征，体现在数据传输过程分为三个阶段--输入、过滤和输出。这三个阶段按顺序依次相连，像一个加工数据的流水线。在实现上，它们分别由三种类型的插件实现，即输入插件、过滤器插件和输出插件，并可通过修改配置文件实现快速插拔。除了这三种类型的插件以外，还有种称为编解码器 (Codec) 的插件。编解码器插件用于在数据进入和离开管道时对数据做解码和编码，所以它一般都是与具体的输入插件或输出插件结合起来使用。

事件(Event) 是 Logstash 中另一个比较重要的概念，它是对 Logstash 处理数据的一种面向对象的抽象。如果将 Logstash 比喻为管道，那么事件就是流淌在管道中的水流。事件由输入插件在读入数据时产生，不同输入插件产生事件的属性并不完全相同，但其中一定会包含有读入的原始数据。过滤器插件会对这些事件做进一步处理，处理的方式主要体现在对事件属性的访问、添加和修改。最后，输出插件会将事件转换为目标数据源的数据格式并将它们存储到目标数据源中。

在 Logstash 管道中，每个输入插件都是在独立的线程中读取数据并产生事件。但输入插件并不会将事件直接传递给过滤器插件或输出插件，而是将事件写入到一个队列中。队列可以是基于内存的，也可以是基于硬盘的可持久化队列。Logstash 管道会预先生成一组工作线程，这些线程会从队列中取出事件并在过滤器中处理，然后再通过输出插件将事件输出。事件队列可以协调输入插件与过滤器插件、输出插件的处理能力，使得 Logstash 具备了一定的消峰填谷能力。

除了输入插件使用事件队列与过滤器插件、输出插件交互以外，Logstash 还在输出插件与目标数据源之间提供了一个死信队列(Dead Letter Queue)。死信队列会在目标数据源没有成功接收到数据时，将失败的事件写入到队列中以供后续做进一步处理。

输入插件、过滤器插件、输出插件、编解码器插件以及事件、队列等组件，共同协作形成了整个 Logstash 的管道功能，它们构成 Logstash 的体系结构，如图所示。



## 插件

插件(Plugin)通过简单的配置就可接入系统并增强系统功能。Logstash 提供了丰富的输入、过滤器、输出和编解码器插件，它们也都是通过配置接入系统并增强 Logstash 在某一方面的功能。

输入插件的作用是使数据进入管道并生成数据传输事件，过滤器插件则对进入管道的事件进行修改、清洗等预处理，而输出插件则将过滤器处理好的事件发送到目标数据源。输入插件和输出插件都可以使用编解码器在数据进入或退出管道时对数据进行编码或解码，使数据以用户期望的格式进入或退出管道。

---

插件开发都很便捷，一般都会提供统一接口作为开发框架，开发人员只要遵从接口规范并编写逻辑就可以开发出新的插件。**Logstash** 插件使用 **Ruby** 语言开发，感兴趣或者有需求的同学可以自行了解下。

### **logstash-plugin 命令**

**Logstash** 官方提供的插件并非全部绑定在 **Logstash** 中，有一部分插件需要用户在使用时手工安装，比如 **log4j** 输入插件在默认情况下就没有安装。**Logstash** 提供了一条插件管理的命令 **logstash-plugin**，可以用于查看、安装、更新或删除插件。除了 **Logstash** 官方提供的插件以外，用户还可以根据需要自定义插件，这些插件也需要使用 **logstash-plugin** 命令管理。有关 **logstash-plugin** 的具体使用方法，可通过执行 **logstash-plugin -help** 查看帮助。

```
[elk@android0318 bin]$ ./logstash-plugin --help
Usage:
  bin/logstash-plugin [OPTIONS] SUBCOMMAND [ARG] ...
  Parameters:
    SUBCOMMAND          subcommand
    [ARG] ...           subcommand arguments
  Subcommands:
    list               List all installed Logstash plugins
    install            Install a Logstash plugin
    remove             Remove a Logstash plugin
    update             Update a plugin
    pack               Package currently installed plugins, Deprecated: Please use prepare-offline-pack instead
    unpack             Unpack packaged plugins, Deprecated: Please use prepare-offline-pack instead
    generate           Create the foundation for a new plugin
    uninstall          Uninstall a plugin. Deprecated: Please use remove instead
    prepare-offline-pack Create an archive of specified plugins to use for offline installation
```

### **插件配置**

**Logstash** 插件的可插拔性，除了体现在可通过 **logstash-plugin** 命令方便地安装与删除以外，还体现在已安装插件的使用和关闭仅需要通过简单的配置即可实现。

**Logstash** 配置文件分为两类，一类是用于设置 **Logstash** 自身的配置文件，主要是设置 **Logstash** 启动、执行等基本信息的配置文件。这类配置文件位于 **Logstash** 安装路径的 **config** 目录中，包括 **logstash.yml**、**pipelines.yml**、**jvm.options**、**log4j.properties** 等文件。在这些配置文件中，**logstash.yml** 文件是核心配置文件，采用的语法格式为 **YAML**；其余文件分别用于配置 **JVM** 和 **log4j**。

```
jvm.options
log4j2.properties
logstash-sample.conf
logstash.yml
pipelines.yml
startup.options
std_es.conf
```

另一类配置就是设置 **Logstash** 管道的配置文件了，它们用于设置 **Logstash** 如何应用这几种插件组成管道，是应用 **Logstash** 最核心的内容之一。

---

最简单的配置 **Logstash** 管道的方式就是我们前面展示的方式，通过 **logstash** 命令参数 **-e** 配置管道。再有一种方式就是通过 **-f** 参数指定管道配置文件的路径，以配置文件的形式设置 **Logstash** 管道。**Logstash** 在启动时会自动扫描目录，并加载所有以 **.conf** 为扩展名的配置文件，所以用户可以将 **Logstash** 的管道配置文件放置在这个目录中。

**Logstash** 管道配置的语法格式不是 **YAML**，它的语法格式是更接近 **Ruby** 哈希类型的初始化格式。

## 事件

**Logstash** 事件由一组属性组成，包括数据本身、事件产生时间、版本等。不同输入插件产生事件的属性各不相同，这些事件属性可以在过滤器插件和输出插件中访问、添加、修改或是删除。

由于事件本身由输入插件产生，所以在输入插件中不能访问事件及其属性。这里所说的对事件及其属性的访问是指在 **Logstash** 管道配置中的访问，比如过滤器插件配置中根据事件属性执行不同的过滤等。

在事件属性中有一个比较特殊的属性 **tags**，它的类型为数组，包含了插件在处理事件过程中为事件打上所有标签。比如插件在处理事件中发生了异常，一般都会为事件添加一个异常标签。标签本身就是一个字符串，没有特别的限制。事件的标签一开始的时候都是空的，只有插件为事件打上了新标签，这个属性才会出现在输出中。所以，总体来说可以认为事件是一组属性和标签的集合。

## 访问事件属性

在管道配置中访问事件属性的最基本语法形式是使用方括号和属性名，例如 **[ name ]** 代表访问事件的 **name** 属性；如果要访问的属性为嵌套属性，则需要使用多层次的路径，如 **[ parent ][ child ]** 相当于访问 **parent. child** 属性；如果要访问的属性为事件的最顶层属性则可以省略方括号。

事件中有哪些属性取决于输入插件的类型，但有一些事件属性几乎在所有事件中都有，比如 **@version**、**@ timestamp** 和 **@ metadata** 等。这类属性大多以 **@** 开头，可以认为是事件的元属性。其中，**@version** 代表了事件的版本，**@timestamp** 是事件的时间戳。

**@metadata** 与前述两个属性不同，它在默认情况下是一个空的散列表（**Hash table**）。**@ metadata** 最重要的特征一般不会在最终的输出中出现，即使在插件中向这个散列表中添加了内容也是如此。（只有当输出插件使用 **rubydebug** 解码器，并且将它的 **metadata** 参数设置为 **true**，**@ metadata** 属性才会在输出中显示）：

```
code => "event.set('[@metadata][hello], 'world' )"
```

**@ metadata** 属性设计的目的并不是为了给输出添加数据，而是为了方便插件做数据运算。它类似于一个共享的存储空间，在过滤器插件和输出插件中都可以访问。因为在实际业务有一些运算结果是需要在插件间共享而又不需要在最终结果中输出，这时就可以将前一插件处理的中间结果存储在 **@ metadata** 中。

---

除了使用方括号访问事件属性以外，在插件参数中还可以通过“`%{ field_name}`”的形式访问事件属性，其中 `field.name` 就是前述的方括号加属性名的形式。例如，在文件输出插件中将日志根据级别写入到不同的文件中：

```
output {  
    file{  
        path => "/var/log/%{loglevel}. log"  
    }  
}
```

在上面的例子中，`loglevel` 是当前事件的一个属性名称，`file` 输出插件的 `path` 参数通过`%{ loglevel}`的形式读取这个属性的值，并以它的值为日志文件名称。

## 事件 API

事件 API 主要在一些支持 Ruby 脚本的插件中使用等。这些插件一般都有一个 `code` 参数接收并运行 Ruby 脚本，在这些脚本中就可以使用 `event` 内置对象访问、修改或者删除事件属性。

## 队列

在互联网时代，许多活动或突发事件会导致应用访问量在某一时间点瞬间呈几何式增长。在这种情况下，应用产生的数据也会在瞬间爆发，而类似 Logstash 这样的数据管道要搬运的数据也会突然增加。如果没有应对这种瞬间数据爆炸的机制，轻则导致应用数据丢失，重则直接导致系统崩溃，甚至引发雪崩效应将其他应用一并带垮。

应对瞬间流量爆炸的通用机制是使用队列，将瞬间流量先缓存起来再交由后台系统处理。后台系统能处理多少就从队列中取出多少，从而避免了因流量爆炸导致的系统崩溃。

Logstash 输入插件对接的事件队列实际上就是应对瞬间流量爆炸、提高系统可用性的机制，它利用队列先进先出的机制平滑事件流量的峰谷，起到了削峰填谷的重要作用。除了输入插件使用的事件队列，输出插件还有一个死信队列。这个队列将会保存输出插件没有成功发送出去的事件，它的作用不是削峰填谷而是容错。

## 持久化队列

Logstash 输入插件默认使用基于内存的事件队列，这就意味中如果 Logstash 因为意外崩溃，队列中未处理的事件将全部丢失。不仅如此，基于内存的队列容量小且不可通过配置扩大容量，所以它能起到的缓冲作用也就非常有限。为了应对内存队列的这些问题，可以将事件队列配置为基于硬盘存储的持久化队列( Persistent Queue)。

持久化队列将输入插件发送过来的事件存储在硬盘中，只有过滤器插件或输出插件确认已经处理了事件，持久化队列才会将事件从队列中删除。当 Logstash

---

因意外崩溃后重启，它会从持久化队列中将未处理的事件取出处理，所以使用了持久化队列的 **Logstash** 可以保证事件至少被处理一次。

如果想要开启 **Logstash** 持久化队列，只要在 `logstash.yml` 文件中将 `queue.type` 参数设置为 `persisted` 即可，它的默认值是 `memory`。当开启了持久化队列后，队列数据默认存储在 **Logstash** 数据文件路径的 `queue` 目录中。数据文件路径默认是在 **Logstash** 安装路径的 `data` 目录，这个路径可以通过 `path.data` 参数修改。持久化队列的存储路径则可以通过参数 `path.queue` 修改，它的默认值是  `${path.data} / queue`。

尽管持久化队列将事件存储在硬盘上，但由于硬盘空间也不是无限的，所以需要根据应用实际需求配置持久化队列的容量大小。**Logstash** 持久化队列容量可通过事件数量和存储空间大小两种方式来控制，在默认情况下 **Logstash** 持久化队列容量为 1024MB 即 1CB，而事件数量则没有设置上限。当持久化队列达到了容量上限，**Logstash** 会通过控制输入插件产生事件的频率来防止队列溢出，或者拒绝再接收输入事件直到队列有空闲空间。持久化队列事件数量容量可通过 `queue.max.events` 修改，而存储空间容量则可通过 `queue.max.bytes` 来修改。

事实上在许多高访问量的应用中，单纯使用 **Logstash** 内部队列的机制还是远远不够的。许多应用会在 **Logstash** 接收数据前部署专业的消息队列，以避免瞬间流量对后台系统造成冲击。这就是人们常说的 **MQ (Message Queue)**，比如 **Kafka**、**RocketMQ** 等。这些专业的消息队列具有千万级别的数据缓存能力，从而可以保护后续应用避免被流量压跨。所以在 **Logstash** 的输入插件中也提供了一些对接 **MQ** 的输入插件，比如 `kafka`、`rabbitmq` 等。

## 死信队列

**Logstash** 输入插件的事件队列位于输入插件与其他插件之间，而死信队列则位于输出插件与目标数据源之间。如果 **Logstash** 处理某一事件失败，事件将被写入到死信队列中。

**Logstash** 死信队列以文件的形式存储在硬盘中，为失败事件提供了采取补救措施的可能。死信队列并不是 **Logstash** 中特有的概念，在许多分布式组件中都采用了死信队列的设计思想。由于死信队列的英文名称为 **Dead Letter Queue**，所以在很多文献中经常将它简写为 **DLQ**。

**Logstash** 在目标数据源返回 400 或 404 响应状态码时认为事件失败，而支持这种逻辑的目标数据源只有 **Elasticsearch**。所以 **Logstash** 死信队列目前只支持目标数据源为 **Elasticsearch** 的输出插件，并且在默认情况下死信队列是关闭的。开启死信队列的方式与持久化队列类似，也是在 `logstash.yml` 文件中配置，参数名为 `dead_letter_queue.enable`。死信队列默认存储在 **Logstash** 数据路径下的 `dead_letter_queue` 目录中，可通过 `path.dead_letter._queue` 参数修改。

死信队列同样也有容量上限，默认值为 1024MB，可通过 `dead_letter_queue.max.bytes` 参数修改。

虽然死信队列可以缓存一定数量的错误事件，但当容量超过上限时它们还是会删除，所以依然需要通过某种机制处理这些事件。**Logstash** 为此专门提

---

供了一种死信队列输入插件，它可以将死信队列中的事件读取出来并传输至另一个管道中处理。

## 管道配置

**Logstash** 安装路径下的 **bin** 目录包含了一些重要的命令，启动 **Logstash** 管道就是通过其中的 **logstash** 命令实现。**logstash** 命令在执行时会读取管道配置，然后根据管道配置初始化管道。管道配置可通过命令行-**e** 参数以字符串的形式提供，也可以通过命令行-**f** 参数以文件形式提供。前者定义管道配置的方式称为配置字符串(**Config String**) 后者则是配置文件，这两种方式只能任选其一而不能同时使用。

在 **logstash. yml** 配置文件中，也包含了 **config. string** 和 **path. config** 两个参数，分别用于以上述两种方式配置默认管道。

**logstash** 命令与 **logstash. yml** 配置文件之间还有许多这样用于配置管道的共享参数，下面就先来看看这些配置参数。

## 主管道配置

**Logstash** 启动后会优先加载在 **logstash. yml** 文件中配置的管道，并且为这个管道指定惟一标识 **main**,我们可以称这个管道为主管道。无论是通过-**e** 参数以配置字符串的形式创建管道，

还是通过-**f** 参数以配置文件的形式创建管道，它们在默认情况下都是在修改主管道的配置。既然有主管道存在，那么是不是还可以配置其他管道呢?答案是肯定的。事实上 **Logstash** 可以在一个进程中配置多个管道，不同的是主管道是在 **logstash. yml** 文件中配置，而其他管道则是在 **pipeline. yml** 文件中配置。但是 **pipeline. yml** 中配置的管道与主管道互斥，如果在 **logstash. yml** 文件中或是使用 **logstash** 命令行的-**e**、-**f** 等参数配置了主管道，那么 **pipeline. yml** 文件中配置的其他管道就会被忽略。只有主管道缺失，**logstash** 命令才会尝试通过 **pipeline. yml** 文件初始化其他管道。无论是主管道还是其他管道，它们的配置参数都是相同的。在 **logstash. yml** 文件中主管道配置参数范例如下：

```
pipeline.id: main  
pipeline.workers: 2  
pipeline.batch.size: 125  
pipeline.batch.delay: 50  
pipeline.unsafe_shutdown: false
```

其中：

**pipeline. id** 用于设置管道的 ID。

而 **pipeline. workers** 则用于设置并发处理事件的线程数量，默认情况下与所在主机 CPU 核数相同

**pipeline.batch.size**: 每个工作线程每批处理事件的数量

**pipeline.batch.delay**: 工作线程处理事件不足时的超时时间

**pipeline.unsafe\_shutdown**: 如果还有未处理完的事件，是否立即退出

---

**Logstash** 处理事件并不是来一个处理一个，而是先缓存 125 个事件或超过 50ms 后再统一处理。由于使用了缓存机制，所以当 **Logstash** 管道因意外崩溃时会丢失已缓存事件。**pipeline.unsafe\_shutdown** 参数用于设置在 **Logstash** 正常退出时，如果还有未处理事件是否强制退出。在默认情况下，**Logstash** 会将未处理完的事件全部处理完再退出。如果将这个参数设置为 true, 会因为强制退出而导致事件丢失。同时，这个参数也解决不了因意外崩溃而导致缓存事件丢失的问题。

## 单管道配置

一个 **Logstsh** 实例可以运行一个管道也可以运行多个管道，它们相互之间可以不受任何影响。一般来说，如果只需要运行一个管道可以使用 **logstash.yml** 配置的主管道；而在需要运行多个管道时才会使用 **pipeline.yml** 配置。无论是单管道还是多管道，对于其中某一具体管道的配置格式完全一致。这种格式不仅在配置文件中有效，在使用字符串配置管道时也是有效的。

### 语法格式

管道配置由 3 个配置项组成，分别用于配置输入插件、过滤器插件和输出插件，如下面的示例所示。尽管输入插件、过滤器插件和输出插件在处理数据时按顺序进行，但在配置文件中它们的顺序并不重要，而且它们也都不是必要配置。所有插件配置都必须要在这三个配置项中，插件是哪种类型就应该放置在哪一个配置项中。每一个配置项可以放多个插件配置，并可以通过一些参数设置这些插件的特性。

```
input {  
    .....  
}  
filter {  
    .....  
}  
output {  
    .....  
}
```

一个插件的具体配置由插件名称和一个紧跟其后的大括号组成，大括号中包含了这个插件的具体配置信息。虽然编解码器也是一种插件类型，但由于只能与输入或输出插件结合使用，所以编解码器只能在输入或输出插件的 **codec** 参数中出现。例如，在示例 12-7 中就配置了一个 **stdin** 输入插件、一个 **aggregate** 过滤器插件和一个 **stdout** 输出插件。其中，**stdin** 使用的编解码器 **multiline** 也是一个插件，它的配置也遵从插件配置格式，由编解码器插件名称和大括号组成。在编解码器 **muliline** 的大括号中配置了 **pattern** 和 **what** 两个参数，而在过滤器插件 **aggregate** 的大括号中则配置了 **task.\_id** 和 **code** 两个参数。

```
input {
```

```
stdin{
    codec => multiline {
        pattern => "^\\s"
        what => "previous"
    }
}
filter {
    aggregate {
        task id => "%{message}"
        code => "event.set ('val', event.get('val')==nil?11:22)"
    }
}
output {stdout { }}
```

插件参数的格式与插件本身的配置在格式上有些相同，但又有着比较明显的区别。插件配置在插件名称与大括号之间是空格，而插件参数则是使用“`=>`”关联起来的键值对。其中，键是插件参数名称，而值就是参数值。这种格式与 Ruby 哈希类型的初始化一样，所以本质上来说每个插件的配置都是对 Ruby 哈希类型的初始化。

管道配置的这种格式中还可以添加注释，注释以“`#`”开头，可以位于文件的任意位置。在 Logstash 安装路径的 config 目录中，有一个 logstash-sample.conf 的样例配置文件，可以在配置管道时参考。

## 多管道配置

Logstash 应用多管道通常是由于数据传输或处理的流程不同，使得不同输入事件不能共享同一管道。假如 Logstash 进程运行的宿主机处理能力超出一个管道的需求，如果想充分利用宿主机的处理能力也可以配置多管道。当然在这种情况下，也可以通过在宿主机上启动多个 Logstash 实例的方式，充分挖掘宿主机的处理能力。但多进程单管道的 Logstash 比单进程多管道的 Logstash 占用资源会更多，单进程多管道的 Logstash 在这种情况下就显现出它的意义了。

多管道虽然共享同- Logstash 进程，但它们的事件队列和死信队列是分离的。在使用多管道的情况下，要分析清楚每个管道的实际负载，并以此为依据为每个管道分配合理的计算资源，起码每个管道的工作线程数量 `pipeline.workers` 应该与工作负载成正比。

配置单进程多管道的 Logstash，是通过 `pipeline.yml` 配置文件实现的。

在启动 Logstash 时，如果没有设置主管道信息，Logstash 读取 `pipeline.yml` 文件以加载管道配置信息。`pipeline.yml` 通过 `config.string` 或 `path.config` 参数以配置字符串或配置文件的形式为每个管道设置独立的配置信息，还可以设置包括事件队列、管道工作线程等多种配置信息。

---

`pipeline.yml` 文件也采用 YAML 语法格式，它使用 YAML 中的列表语法来定义多个管道，在每个列表项中通过键值对的形式来定义管道的具体配置。默认情况下，`logstash.yml` 没有配置主管道，而 `pipeline.yml` 文件的内容也全部注释了，所以直接无参运行 `logstash` 时会报 `pipeline.yml` 文件为空的错误。配置范例如下：

```
- pipeline.id: test
  pipeline.workers: 1
  pipeline.batch.size: 1
  config.string: "input { generator {} } filter { sleep { time => 1 } } output
{stdout { .. } }"
- pipeline.id: another_test
  queue.type: persisted
  path.config: "/tmp/logstash/*.config"
```

配置了两组管道，它们的唯一标识分别是 `test` 和 `another_test`。`test` 管道使用 `config sting` 直接设置了管道的插件信息，而 `another_test` 则通过 `path.config` 指定了管道配置文件的路径。可见，`pipeline.yml` 配置多管道时使用的参数与 `logstash.yml` 中完全一致，它还可以使用配置队列和死信队列的参数，只是这些参数都只对当前管道有效。

## 编解码器插件

在 Logstash 管道中传递是事件，所以输入和输出插件都存在数据与事件的编码或解码工作。为了能够复用编码和解码功能，Logstash 将它们提取出来抽象到一个统一的组件中，这就是 Logstash 的编解码器插件。由于编解码器在输入和输出时处理数据，所以编解码器可以在任意输入插件或输出插件中通过 `codec` 参数指定。

### plain 插件

Logstash 版本 7 中一共提供了 20 多种官方编解码器插件，在这些编解码器插件中最为常用的是 `plain` 插件，有超过八成的输入和输出插件默认使用的编解码器插件就是 `plain`。

`plain` 编解码器用于处理纯文本数据，在编码或解码过程中没有定义明确的分隔符用于区分事件。与之相较，`line` 编解码器则明确定义了以换行符作为区分事件的分隔符，每读入行文本就会生成一个独立的事件，而每输出一个事件也会在输出文本的结尾添加换行符。`line` 编解码器符合人们处理文本内容的习惯，所以也是种比较常用的编解码器。除了 `plain` 和 `line` 编解码器以外，`json` 编解码器是另一种较为常用的编解码器，主要用于处理 JSON 格式的文本数据。

尽管 `plain` 编解码器非常重要，但它在使用上却非常简单，只有 `charset` 和 `format` 两个参数。其中，`charset` 用于设置文本内容采用的字符集，默认值为 `UTF-8`；而 `format` 则用于定义输出格式，只能用于输出插件中。`format` 参数定义输出格式时，采用的是 “`%{}`” 形式，例如：

```
input {
```

```

stdin {codec => line }

}

output{
    stdout{
        codec => plain {
            format => "The message is % {[message]}\nIt's
from %{[host]}\n"
        }
    }
}

```

在示例中，**Logstash** 在标准输入(即命令行)等待输入，并将输入内容按**format**

参数定义的格式以两行打印在标准输出中。

## line 编解码器

以行为单位对文本数据进行解码时，每-行都有可能成为一个独立事件；而在编码时，每个事件输出结尾都会添加换行符。以行为单位的编解码器有 **line** 和 **multiline** 两种，前者以单行为输入事件，而后者则根据一定条件将多行组装成一个事件。

**multiline** 编解码器也是以行为编解码的单元，但在它编解码的事件中可能会包含一行也有可能会包含多行文本。**multiline** 编解码器对于处理日志中出现异常信息时非常有用，它能将异常合并到同一个日志事件中。例如，在下面中所示的一段日志文本，从是一个 **Java** 异常打印出来的堆栈信息，在逻辑上它们都应该归属于 **ERROR** 日志：

```

2019-06-20 20:04:25,290 ERROR [http-nio-8090-exec-1] o.c.s.f.c.TestExceptionController [TestExceptionController.java:26] error: null
java.lang.NullPointerException: null
at org.chench.springboot.controller.TestExceptionController.test(TestExceptionController.java:24) ~[classes/:na]
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0_181]
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) ~[na:1.8.0_181]
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[na:1.8.0_181]
at java.lang.reflect.Method.invoke(Method.java:498) ~[na:1.8.0_181]
at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:209)
[spring-web-5.0.6.RELEASE.jar:5.0.6.RELEASE]
at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:136)
[spring-web-5.0.6.RELEASE.jar:5.0.6.RELEASE]
at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:102)
[spring-webmvc-5.0.6.RELEASE.jar:5.0.6.RELEASE]
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:877)
[spring-webmvc-5.0.6.RELEASE.jar:5.0.6.RELEASE]
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:783)
[spring-webmvc-5.0.6.RELEASE.jar:5.0.6.RELEASE]
at org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:87)
[spring-webmvc-5.0.6.RELEASE.jar:5.0.6.RELEASE]
at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:991) [spring-webmvc-5.0.6.RELEASE.jar:5.0.6.RELEASE]
at org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:925) [spring-webmvc-5.0.6.RELEASE.jar:5.0.6.RELEASE]
at org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:974) [spring-webmvc-5.0.6.RELEASE.jar:5.0.6.RELEASE]
at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:866) [spring-webmvc-5.0.6.RELEASE.jar:5.0.6.RELEASE]
at javax.servlet.http.HttpServlet.service(HttpServlet.java:635) [tomcat-embed-core-8.5.31.jar:8.5.31]
at org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:851) [spring-webmvc-5.0.6.RELEASE.jar:5.0.6.RELEASE]
at javax.servlet.http.HttpServlet.service(HttpServlet.java:742) [tomcat-embed-core-8.5.31.jar:8.5.31]

```

一般来说，正常日志都以方括号“[”开头，而当有异常写入到日志中时，异常信息不会按日志统一格式打印。**multiline** 提供了几个可以设置如何合并多行的参数，比如说：

```

codec => multiline {
    pattern => "\[ "
    negate => true
}

```

```
    what => "previous"  
}
```

**pattern** 参数设置了一个正则表达式，代表的含义是以方括号 “[” 开头的文本。**pattern** 参数与 **negate** 参数共同决定什么的文本需要被合并，**negate** 参数用于设置对 **pattern** 定义的正则表达式取反。所以上面定义这两个参数的含义就是所有不满足 “^[” 正则表达式的行都需要被合并，而合并到哪里则由参数 **what** 来决定。**what** 参数有 **previous** 和 **next** 两个可选值，前者将需要合并行与前一个事件合并，而后者则会将需要合并行与后一个事件合并。

## json 编解码器

**json** 文本的编解码器有两种，一种叫 **json** 编解码器，另一种叫 **json\_lines** 编解码器。在输入时从 JSON 格式的数据中将消息解析出来生成事件，而在输出时将事件编码为 JSON 格式的数据。如果 **json** 编解码器在解析 JSON 时出错，**Logstash** 会使用 **plain** 编解码器将 JSON 视为纯文件编入事件，同时会在事件中添加 **\_jsonparsefailure** 标签。

**json** 编解码器只有一个参数 **charset**, 用于设置 JSON 数据字符集，默认为 **UTF-8**。

**json\_lines** 编解码器也用于处理基于 JSON 的文本格式，但它处理的是一种被称为 **JSONLines** 的特殊 JSON 格式。这种格式由多个独立的 JSON 行组成，每一行都是有效的 JSON 值。

**json\_lines** 可用于存储每行一个 JSON 的结构化数据，适用于 **unix** 风格的文本处理工具和 **shell** 管道。有关 **JSON Lines** 的相关说明请参考 **JSON Lines** 官方网站 <http://jsonlines.org/> 。

**json\_lines** 编解码器包括 **charset** 和 **delimiter** 两个参数，前者用于设置 JSON 数据字符集，默认为 **UTF-8**; 而后者则用于设置 **JSON Lines** 格式中的行分隔符。

## 序列化编解码器

在分布式应用中，为了将对象在网络中传输通常需要将对象转换为可传输的文本或二进制字节流。这种将对象转换为文本或字节的过程就称为序列化，而将文本或字节还原为对象的过程则称为反序列化。由于存在读取序列化数据格式的需求，**Logstash** 也提供了两种针对不同序列化协议的编解码器，它们就是 **avro** 和 **protobuf**。

**Avro** 最初是 **Hadoop** 的一个子项目，负责 **Hadoop** 中数据序列化与反序列化，现在已经发展为 **Apache** 顶级开源项目。**Avro** 需要通过 **Schema** 文件定义对象格式，然后再依照 **Schema** 文件定义做序列化和反序列化。**Logstash** 的 **avro** 编解码器也是基于 **Schema** 文件，所以在定义 **avro** 编解码器时必须要指定的参数是 **schema\_uri**, 用于定义 **Schema** 文件的存储路径。

**Protobuf** 是 **Google** 定义的另一种序列化协议，与其他序列化协议相比 **Protobuf** 更小更快也更简单，所以在很多领域都得到了广泛应用。**Logstash** 的 **protobuf** 编解码器默认并没有安装，需要使用 “**logstash-plugin install**

---

`lgstash-codec-protobuf`" 安装。`protobuf` 编解码器有两个参数，一个是 `class_name` 用于指定 `.proto` 文件中定义类的名称，另一个是 `include_path` 用于定义 `.proto` 文件的存储路径。

## 输入输出插件

`Logstash` 管道可以配置多个输入插件，这可以将不同源头的数据整合起来做统一处理。在分布式系统中，数据都分散在不同的容器或不同的物理机上，每份数据往往又不完整，需要类似 `Logstash` 这样的工具将数据收集起来。比如在微服务环境下，日志文件就分散在不同机器上，即使是单个请求的日志也有可能分散在多台机器上。如果不将日志收集起来，就无法查看一个业务处理的完整日志。

`Logstash` 管道也可以配置多个输出插件，每个输出插件代表一种对数据处理的业务需求。比如对日志数据存档就可以使用 `s3` 输出插件，将日志数据归档到 `S3` 存储服务上；还可以使用 `elasticsearch` 输出插件，将数据索引到 `Elasticsearch` 中以便快速检索等。在业务系统创建之初，人们对于数据究竟会产生什么样的价值并不清楚。但随着人们对于业务系统理解的深入，对数据处理的新需求就会迸发出来。面对新需求，只要为 `Logstash` 管道添加新的输出插件就能立即与新的数据处理需求对接起来，而对已有数据处理业务又不会产生任何影响。到目前为止，`Logstash` 对于常见的数据处理需求都可以很好的对接，这包括数据归档存储、数据分析处理、数据监控报警等。

`Logstash` 官方提供的输入插件与输出插件都有 50 多种，而在这共计 100 多种的插件中每一种插件又有不同的配置参数，全部都了解意义不大。这些插件基本上都会成对出现，大多数输入插件也会在输出插件中出现。我们挑选几个有代表性的来学习一下。

## stdin 输入插件和 stdout 插件

两个最简单的插件，即 `stdin` 输入插件和 `stdout` 插件。这两个插件分别代表标准输入和标准输出，也就是命令行控制台。由于它们比较简单，所以一般不需要做任何配置就可以直接使用。

`stdin` 只有一组通用参数，这些参数不仅对 `stdin` 有效，对其他输入插件也有效。

名称	参数类型	默认值	用处
<code>add_field</code>	<code>hash</code>	{}	向事件中添加属性
<code>codec</code>	<code>codec</code>	<code>line</code>	编解码器
<code>enable_metric</code>	<code>boolean</code>	<code>true</code>	是否开启指标日志
<code>id</code>	<code>string</code>	无	插件 ID, 未设置时自动生成
<code>tags</code>	<code>array</code>	无	向事件中添加标签
<code>type</code>	<code>string</code>	无	向事件添加 <code>type</code> 属性

同样的，`stdout` 也是只有通用参数，这些参数也是对所有输出插件都有效。

名称	参数类型	默认值	用处
<code>codec</code>	<code>codec</code>	<code>rubydebug</code>	编解码器
<code>enable_metric</code>	<code>boolean</code>	<code>true</code>	是否开启指标日志

---

id	string	无	插件 ID,未设置时自动生成
----	--------	---	----------------

## elasticsearch 插件

Elasticsearch 既可以作为 Logstash 的输入也可以作为输出，但在多数情况下 Elasticsearch 都是作为 Logstash 的输出使用。

elasticsearch 输出插件使用 hosts 参数设置连接 Elasticsearch 实例的地址，hosts 参数可以接收多个 Elasticsearch 实例地址，Logstash 在发送事件时会在这些实例上做负载均衡。如果连接的 Elasticsearch 与 Logstash 在同一台主机且端口为 9200，那么这个参数可以省略。

---

### elasticsearch 插件连接相关部分参数列表

名称	参数类型	默认值	用处
Hosts	uri	[//127.0.0.1]	ES 所在主机地址, 可包括端口号
path	string	无	访问 Elasticsearch 的 HTTP 路径
bulk_path	string	无	执行 bulk 请求的路径
proxy	uri	无	HTTP 代理
parameters	hash	无	URL 参数
pipeline	string	nil	Elasticsearch ingest 管道
user	string	无	Elasticsearch 用户名
password	password	无	Elasticsearch 密码
timeout	number	60	连接超时时间
pool_max	number	1000	连接最大数
retry_max_interval	number	64	重连时间间隔的上限, 单位为秒(s)
retry_initial_interval	number	2	请求失败后重连的初始时间间隔,
之后每次重连后加倍时间间隔, 单位为秒(s)			
retry_on_conflict	number	1	冲突时重试次数
resurrect_delay	number	5	连接端点死机后重连的时间延迟,
单位为秒			
sniffing	boolean	false	感知 Elasticsearch 集群节点变化
smiffing_delay	number	5	感知 Elasticsearch 集群节点变化的时间间隔, 单位为秒
smiffing_path	string	无	感知 Elasticsearch 集群节点变化的路径

elasticsearch 输出插件将 Logstash 事件转换为对 Elasticsearch 索引的操作, 默认情况下插件会将事件编入名称格式为 “ logstash- %{+ YYYY.MM.dd}” 的索引中, 可使用 index 参数修改索引名称格式。

除了编入索引以外, elasticsearch 输出插件还支持根据 ID 删除文档、更新文档等操作, 具体可以通过 action 参数配置。action 可选参数包括 index、delete、create 和 update 几种, 其中 index 和 create 都用于向索引中编入文档, 区别在于 create 在文档存在时会报错, 而 index 则会使用新文档替换原文档。delete 和 update 则用于删除和更新文档, 所以在使用 delete、update 和 create 时需要通过 document\_id 参数指定文档 ID。此外, update 在更新时如果文档 ID 不存在, 那么可以使用 upsert 参数设置添加新文档的内容。例如:

```
output{
    elasticsearch {
        document_id => "% {message}"
        action = > "update"
```

```
        upsert => "{\"message\":\"id didn't exist\"}"  
    }  
}  
}
```

**message** 属性的内容作为文档 ID 值，并根据这个 ID 值使用当前文档更新原文档。如果文档不存在则使用 **upsert** 参数设置的内容添加文档，**upsert** 参数要求为字符串类型，其中属性要使用双引号括起来。除了 **upsert** 以外，还可以将 **doc\_as\_upsert** 参数设置为 **true**，则在文档不存在时会将当前文档添加到索引。**elasticsearch** 插件还有很多与 Elasticsearch 索引、模板、路由等相关的参数。

#### elasticsearch 输出插件部分参数列表

名称	参数类型	默认值	用处
Index	string	无	事件输出的索引，默认值为 logstash-%{+YYYY.MM.dd}
action	string	index	Elasticsearch 操作行为，可选值 index、delete、create、updateupsert
upsert	string	无	upsert 内容
doc_as_upsert	boolean	false	执行 upsert 操作，即在文档 ID 不存在时创建文档
document_id	string	无	文档 ID
document_type	string	无	文档映射类型
parent	string	nil	父文档 ID
routing	string	无	分片路由

Elasticsearch 作为 Logstash 输入源时，Logstash 默认会通过 DSL 从索引 "logstash-\*" 中读取所有文档，一旦读取完毕 Logstash 就会自动退出。输入插件使用的 DSL 通过 **query** 参数设置，参数值与 Elasticsearch 的 **\_search** 接口参数相同。

## 文件插件

面向文件的输入插件从文件中读取数据并转换为事件传入 Logstash 管道，而面向文件的输出插件则将 Logstash 管道事件转换为文本写入到文件中。面向文件的输入输出插件名称都是 **file**，它们都需要通过 **path** 参数设置文件路径。

不同的是，文件输入插件的 **path** 参数类型是数组，所以可以指定组文件作为输入源头。

而文件输出插件的 **path** 参数则为字符串，只能设置一个文件名称。此外，文件输入插件的 **path** 参数只能是绝对路径而不能使用相对路径，并且在路径中可使用 “\*” 和 “\*\*” 这样的通配符。

而文件输出插件则可以使用相对路径，并且还可以在路径中使用 “%{}” 的形式动态设置文件路径或名称。这也是文件输出插件形成滚动文件的方式，例如使用 “path=>/logs/log-%{+YYYY-MM-dd}.log” 配置输出插件就可以每日生成一个文件。

---

## 事件属性

文件输入插件产生的输入事件中包含一个名为 **path** 的属性，它记录了事件实际来自哪个文件。在配置文件输入插件时，如果在读取路径配置中使用了通配符或使用数组指定了多个读取路径时，**path** 属性可以明确事件实际来源的文件。换句话说，**path** 属性记录的是某个文件的具体路径和名称而不会包含通配符。

默认情况下，文件输入插件以一行结束标识一个输入事件，**message** 属性中会包含换行符前面的所有内容。当然也可以通过将插件的编解码器设置为 **multiline**，将多行数据归为一个输入事件。文件输入插件默认也以换行符 “\n” 作为行结束标识。

## 读取模式

文件输入插件默认会从文件结尾处读取文件的新行，这种模式被称为尾部读取模式(**Tail Mode**)。如果 **Logstash** 启动后被读取文件没有新行产生，则不会有新的事件输出。

除了尾部读取模式以外，文件输入插件也可配置为从文件开头读取，这种模式称为全部读取模式(**Read Mode**)。也就是说，文件输入插件支持两种读取模式，尾部读取模式和全部读取模式，可以通过设置文件输入插件的参数 **mode** 以修改读取模式。

例如可以将示例中读取模式设置为 **read**，这样只要被读取文件不为空，**Logstash** 启动后就会有输出：

```
input {  
    file {  
        path => "xxxx/xxxxx/logstash-plain.log"  
        mode => "read"  
    }  
}
```

在尾部读取模式下，文件被看作是一个永不终止的内容流，所以 **EOF EndOfFile** 标识对于文件输入插件来说没有特别意义。如果使用过 **Linux** 中的 **tail** 命令，文件输入插件的尾部读取模式与 **tail -OF** 极为相似。

尾部读取模式非常适合实时搬运日志文件数据，只要有新日志产生它就会将变化内容读取出来。而在全部读取模式下文件输入插件将每个文件视为具有有限行的完整文件，如果读取到 **EOF** 就意味着文件已经结束，文件输入插件会关闭文件并将其置于“未监视”状态以释放文件。

参数 **file\_completed action** 可以设置文件读取结束后的释放行为，可选值为 **delete**、**log** 和 **log\_and\_delete**，默认行为是删除文件即 **delete**。如果选择 **log** 或 **log\_and\_delete**，还需要使用 **file\_completed log\_path** 参数指定一个日志文件及其路径，文件输入插件会将已经读取完的文件及其路径写入到这个日志文件中。对于 **log\_and\_delete** 来说，则会先写日志再将文件删除。需要特别注意的是，文件输入插件在写这个日志文件时，并不负责对文件的滚动，所以该日志文件可能会变得非常大。

---

## 多文件

当 `path` 参数配置的文件是多个时，文件输入插件会同时监控这些文件。当文件大小发生了变化，文件输入插件会将这些文件激活并打开。

文件输入插件感知文件变化的方法是每隔 1 秒钟轮询所有被监控文件，这个时间间隔由参数 `stat_interval` 控制。显然增加 `stat_interval` 的值会减少访问文件的次数，但会增加用户感知文件变化的时间延迟。

除了感知已知文件的变化以外，文件输入插件也会感知到新文件的加入。新加入文件的感知时间间隔由 `discover_interval` 参数来控制，它设置的值是 `stat_interval` 的倍数，所以实际发现新文件的时间间隔是 `discover_interval * stat_interval`。默认情况下 `discover_interval` 为 15s，而 `stat_interval` 为 1s，所以发现新文件的最大时间间隔为 15s。如果被监控文件变化非常频繁，可以将 `stat_interval` 值缩小以减少数据延迟。

在轮询被监控文件时，轮询文件的顺序也是有讲究的。在 Logstash 的早期版本中，轮询的顺序由操作系统返回文件的顺序决定。

新版本中这个顺序由两个参数共同决定，一个是 `file_sort_by` 参数，定义根据文件哪一个属性排序；另一个是 `file_sort_direction` 参数，定义文件轮询是按升序还是按降序。

`file_sort_by` 有两个可选值 `last_modified` 和 `path`，前者按文件最后修改时间排序，是默认值；后者是按文件路径和名称排序。`file_sort_direction` 也有两个可选值，`asc` 代表升序，`desc` 代表降序，默认值为 `asc`。所以在默认情况下，轮询是按照文件最后修改时间的升序依次进行的，也就是最先修改的数据将会被优先感知到并处理。

文件输入插件同时打开文件的数量也有上限，默认情况下文件输入插件可同时打开文件的数量是 4095 个，这个值可以通过 `max_open_files` 参数来修改。设置了同时打开文件的最大值，可以防止打开文件过多而影响系统性能或者将系统拖垮。如果需要打开的文件数量很多，又不能将 `max_open_files` 设置过大，可以使用 `close_older` 将长时间无变化的文件关闭以释放打开文件的数量空间。`close_older` 设置了文件未被读取的时间间隔，达到这个标准后文件即被关闭，默认值为 1 小时。文件在关闭后如果又发生了变化，它将重新进入队列中等待打开。由于在全部读取模式下文件读取完毕后就会被释放，所以 `close_older` 参数仅对尾部读取模式有效。

文件输入插件在打开文件后，会按前面讲的顺序依次读取所有文件，直到读取完所有文件。在默认情况下，它会先将一个文件需要处理的内容全部读取完再读下一个文件。但这个行为可以通过更改 `file_chunk_size` 和 `file_chunk_count` 来修改，它可以使插件每次只读取文件一部分内容后就读取下一个文件，这对于快速将全部文件的变化反馈给用户非常有用。

其中，`file_hunk_size` 用于设置每次读取数据块的大小，默认值为 32768 即 32kB；`file_chunk_count` 则用于设置处理每个文件时读取多少个数据块，默认值为 4611686018427387903，所以相当于读取没有上限。一般只需要将 `file_chunk_count` 值缩小，而无需修改 `file_chunk_size` 值。

---

除此之外，还有一个 `start_position` 参数，设定文件开始读取位置，仅对 `tail` 模式有效。

在配置项中如果遇到了与时长有关的，有一类类特殊的数据类型叫 `string_duration`，它用于以字符串的形式定义时间范围。`string_duration` 的基本格式为 `[number][string]`，即由数字、空格和字符串组成，中间的空格可加也可以不加，而字符串代表时间单位。例如 `45s` 和 `45 s` 都代表 45 秒，`s` 代表的时间单位为秒。事件单位支持周 (`w week weeks`)、天 (`d day days`)、小时 (`h hour hours`)、分钟 (`m min minute minutes`)、秒 (`s sec second seconds`)、毫秒 (`ms msec msecs`)、微秒 (`us usec usecs`)。

## 文件输出插件

相对于文件输入插件来说，文件输出插件就简单多了。默认情况下，它会根据 `path` 参数指定的路径创建文件，并将事件转换为 JSON 格式写入文件的一行。如果文件中已经有数据，则新事件会附加到文件结尾。当然也可以通过修改 `write_behavior` 参数为 `overwrite`，覆盖文件中已存在的数据。

## 更多插件

### 面向关系型数据库的插件

面向关系型数据库的插件用于从关系型数据库中读取或写入数据，它们的名称是 `jdbc`。

`Logstash` 官方只提供了 `jdbc` 输入插件，而输出插件则属于社区开发插件。

`jdbc` 输入插件可以定期执行 SQL 查询数据库，也可以只执行一次 SQL。每读入数据库表中一行会生成一个输入事件，行中每一列会成为事件的一个属性。`jdbc` 输入插件可用于结构化数据的大数据分析和处理，可以将关系型数据库中的数据传输到类似 `Elasticsearch` 这样的 NoSQL 数据库以方便海量数据检索或存档，或传输给 `Spark`、`Storm` 这样的大数据分析框架做实时分析处理。

JDBC 连接数据库需要指定 JDBC 驱动程序、JDBC URL 以及数据库用户名和密码，`jdbc` 输入插件也不例外。除了连接数据库，还要为 `jdbc` 输入插件定义要执行的 SQL 语句。

### 面向消息中间件的插件

`Logstash` 虽然内置了队列对输入事件做缓冲，但面对瞬间突发流量时仍然存在出错甚至崩溃的可能。同样地，如果 `Logstash` 给目标数据源发送数据量过大，也存在着压垮目标数据源的可能。所以在一些流量比较大的互联网应用中，基本上都会视情况在 `Logstash` 之前或之后部署分布式消息中间件，以达到削峰填谷、保护应用的作用。除了削峰填谷，消息中间件还可以作为一种适配器来使用。有些应用可以向消息中间件发送消息，但却没有 `Logstash` 的相关输入插件。这时就

---

可以让 Logstash 从消息中间件中订阅该应用的消息，以间接的方式从应用中获取数据。

Logstash 面向消息中间件的插件主要是 kafka、redis 和 rabbitmq 三个插件，stomp 和 jms 插件虽然也是基于消息的协议，但它们并未在 Logstash 的默认绑定中，需要自行安装。

## 面向通信协议的插件

Logstash 面向通信协议的插件通过标准通信协议与外部数据源交互，支持包括 TCP、UDP、HTTP、IMAP 等广泛应用的协议。由于这些插件从通信协议层面支持数据传输，可以支持基于这些协议对外提供服务的应用，因此极大地扩展了 Logstash 的应用范围。以 Elasticsearch 为例，它对外提供的 REST 接口基于 HTTP，所以尽管 Logstash 提供了 Elasticsearch 输入和输出插件，但依然可以通过 HTTP 相应的插件实现数据输入和输出的功能。

## 执行命令插件

据采集问题，但依然有可能存在一些特殊情况需要使用特定命令或脚本处理数据。Logstash 提供了两个可执行命令或脚本的插件 exec 和 pipe，它们都同时具有输入和输出插件。

exec 输入插件将执行命令的结果转换为事件传入 Logstash 管道，而 exec 输出插件则针对每一个事件执行一条命令，所以它们都必须要通过 command 参数指定要执行的命令。

pipe 插件基于操作系统管道机制实现数据的输入和输出，但正因为使用了管道而不能在 Windows 系统中使用 pipe 插件。pipe 同时具备输入插件和输出插件。

## 开发测试插件

Logstash 提供了两种可以自动生成事件的输入插件，即 generator 和 heartbeat 输入插件，可以使用它们测试配置是否正常或测试组件的性能指标。

generator 插件默认会不间断地生成输入事件，事件包含的文本内容为“Hello world!”。事件的生成次数可由参数 count 设置，默认值 0 代表不限次数。事件的文本内容可以通过 message 或 lines 参数定义。

heartbeat 插件与 generator 插件类似，也是自动生成包含特定文本内容的输入事件。不同的是，heartbeat 插件的输入事件会按定的时间间隔产生。事件生成的事件间隔默认值为 60s，可使用 interval 参数修改。

## 过滤器插件

一个 Logstash 管道可以配置多个过滤器插件，每个过滤器插件的职责各不相同。它们可以对原始数据做添加、删除、修改等基本的转换操作，也可以对原始文本数据做结构化处理，还可以对数据做一致性校验、清除错误数据等。这相当于数据处理与分析领域中的数据清洗，是数据处理与分析中极为重要的一环。多个过滤器会组装成过滤器的职责链，事件经过一个过滤器处理后会传递给下一个过滤器，过滤器处理事件的顺序与它们在配置文件中的配置次序一致。

---

Logstash 官方提供了 40 多种过滤器插件，我们会了解下常用插件。

各种过滤器插件有一些通用参数，这些参数对于比如 `add_field` 和 `add_tag` 参数，它们可以在过滤器成功执行后向事件中添加属性或添加标签。而 `remove_field` 和 `remove_tag` 参数则刚好相反，它们会在过滤器成功执行后删除属性或标签。

#### 过滤器通用参数

参数名	类型	默认值	说明
<code>add_field</code> 属性，可使用 <code>%{}</code> 的格式	hash	{}	过滤器执行成功后向事件中添加属性
<code>add_tag</code> 标签，可使用 <code>%{}</code> 的格式	array	[]	过滤器执行成功后向事件中添加标签
<code>enable_metric</code>	boolean	true	是否开启指标日志
<code>id</code>	string	无	过滤器 ID，未指定时由 Logstash 自动分配
<code>periodic_flush</code>	boolean	false	调用过滤器 <code>flush</code> 方法的时间间隔
<code>remove_field</code> 属性，可使用 <code>%{}</code> 的格式	array	[]	过滤器执行成功后从事件中删除属性
<code>remove_tag</code> 标签，可使用 <code>%{}</code> 的格式	array	[]	过滤器执行成功后从事件中删除标签

这些参数在各种过滤器中的含义基本相同。

## 全文数据结构化过滤器

全文数据是典型的非结构化数据。但有些全文数据具有一定的格式规范，它们比较接近半结构化数据，但相比于半结构化数据又粗糙一些。在这种类型的文本中，比较典型的例子就是系统或应用输出的文本日志。这些文本日志往往都是按一定的次序包含时间、线程、日志级别、日志信息等内容，所以在处理上非常适合使用正则表达式做解析，并从中提取出结构化数据以方便对数据做进一步分析。

Logstash 过滤器插件中，`grok` 过滤器和 `dissect` 过滤器都可以通过定义模式文本中提取数据。不同的是 `grok` 过滤器使用的是正则表达式定义模式，而 `dissect` 过滤器则是使用“`%{}`”的格式定义了一套独立的模式规则。从使用上来说，`grok` 过滤器由于使用了正则表达式而功能强大，但也使得它在处理文本时的性能比较低；而 `dissect` 虽然功能简单，但性能更为出众，适用于文本格式规则比较简单文本。

### grok 过滤器

`grok` 过滤器在 Logstash 过滤器插件中应该是最为常用的文本结构化过滤器，对于一些中间件日志文本的处理来说，`grok` 过滤器几乎是必须要选择的过滤器插件。`grok` 过滤器根据一组预定义或自定义的模式去匹配文本，然后再将匹配的结果提取出来存储在事件属性上，后续过滤器插件就可以在此基础上做进一步处理。

---

## 预定义模式

grok 过滤器预定义了大约 120 种模式，它们使用一组由大写字母组成的模式名称来唯一标识。常用的模式名称包括 NUMBER (数字)、WORD ( 单词)、SPACE (空格)、DATA(任意字符)等：

NUMBER (?%|BASE10NUM| ) 十进制数, BASE10NUM 的简写  
BASE10NUM (?<! [0-9. +-])(?>[ +-]?(?:(?:[0-9] +(?:\\. [0-9] +)?) |(?:\\. [0-9]+)))  
十进制数, 整型或浮点型数值

WORD 单词

SPACE 空格

NOTSPACE 非空格

DATA 任意字符,非贪婪模式匹配

GREEDYDATA 任意字符,贪婪模式匹配

TIMESTAMP\_IS08601 ISO8601 定义的日期格式, 即 yyyy- MM- dd'T'HH:mm:ss:SSSZZ

LOGLEVEL 日志级别

由于这些预定义的模式很多, 大家可通过如下地址查看这些模式的列表和详细定义:

<https://github.com/logstash-plugins/logstash-patterns-core/tree/master/patterns>

如果使用 grok 过滤器预定义模式定义文本提取规则, 需要遵从一定的语法规则。具体来说, 它的基本语法格式为 “% { SYNTAX: SEMANTIC}” ,其中 SYNTAX 为模式名称, SEMANTIC 为存储匹配内容的事件属性名称。如果想匹配一段数字并希望将匹配出来的数字存储到事件的 age 属性上, 则按上述语法规则应该写为 "% { NUMBER:age}"。在这段提取规则中就使用到了前面的预定义模式 NUMBER。

## 自定义模式

如果 grok 过滤器预定义的模式不能满足要求, 用户也可以根据 grok 过滤器指定的语法直接使用正则表达式定义模式。自定义模式的语法格式为 “(<属性名>模式定义)” , 其中 “<属性名>” 指定了模式匹配后存储属性的名称, 注意这里的尖括号是语法的一部分故而不能省略。 “<属性名>” 后面的 “模式定义” 代表使用正则表达式定义的模式。例如, 义邮政编码的模式为 6 位数字, 并希望提取后存到 postcode 属性上, 则使用 grok 自定义模式表示的提取规则为 “(<postcode>\d{6})”。

### match 参数

grok 过滤器最重要的一个参数是 match, 它接收一个由事件属性名称及其文本提取规则为键值对的哈希结构。grok 过滤器会根据 match 中的键读取相应的事件属性, 然后再根据键对应的提取规则从事件属性中提取数据。下面通过一个简单的例子来说明 match 参数的用法。

以 Logstash 默认日志的格式为例, Logstash 每条日志开头都包括三个方括号, 方括号中的内容依次为时间、日志级别、Logger 名称等, 在这三个方括号后面

---

跟的是日志具体信息。具体格式可以到 Logstash 安装路径下的 logs 目录中查看 logstash-plain.log 文件，这里摘取其中几行为例：

```
[2020-07-08T16:49:42,666][INFO ][logstash.agent] Successfully started Logstash API endpoint {:port=>9600}
[2020-07-08T16:49:47,558][INFO ][logstash.runner] Logstash shut down.
[2020-07-08T16:55:56,950][WARN ][logstash.config.source.multilocal] Ignoring the 'pipelines.yml' file because modules or command line options
are specified.
[2020-07-08T16:55:57,078][INFO ][logstash.runner] Starting Logstash {"logstash.version"=>"7.7.0"}
```

由于 Logstash 日志遵从一定的格式规范，所以如果使用 grok 过滤器的 match 参数定义提取规则，就可以将时间、日志级别、Logger 名称和日志内容提取出来。提取规则如示例所示：

```
filter{
  grok {
    match =>
      message =>
        "\[%{TIMESTAMP_ISO8601:time}\]\[%{LOGLEVEL:level}%{SPACE}\]\[%{NOTSPACE:lo
        gger}%{SPACE}\]\%{SPACE}%{GREEDYDATA:msg}"
    }
  }
}
```

在示例中，match 参数的类型为散列，其中只包含有一个条目。这个条目以 message 属性为键，而以对该属性的提取规则为值。在 message 属性的提取规则中，使用了多个预定义模式。比如其中的“%{LOGLEVEL:level}”就是使用 LOGLEVEL 预定义模式匹配文本，并将匹配结果添加到事件的 level 属性。

在默认情况下，grok 插件在模式匹配失败时会给事件打上\_grokparsefailure 标签。除了 match 参数以外，grok 还有很多参数，大家可以自行查询相关网址。

## dissect 过滤器

dissect 过滤器是另一种可以实现文本结构化的过滤器，它使用用户自定义的分隔符切分文本，然后再将切分后的文本存储到事件属性上。

dissect 过滤器使用多个切分单元(Dissection)定义切分规则，切分单元之间可以有一个或多个字符，这些字符就是切分文本的分隔符。定义一个切分单元的格式为“%{field}”，其中的 field 在一般情况下是事件的属性名称。

例如在“%{field1}/%{field2}”中共定义了两个切分单元，由于符号“/”位于两个切分单元之间，所以将被认为是切分文本的分隔符。如果切分规则中包含有三个以上的切分单元，则在切分规则中就可以指定多个分隔符了。

### mapping 参数

dissect 过滤器使用 mapping 参数定义切分文本的方式，mapping 参数也是哈希类型。与 match 参数类似，mapping 参数的键为需要切分的属性名称，而值则为切分规则。下面以 Logstash 日志存档文件名称为例，说明如何使用 mapping 参数定义切分规则。Logstash 日志存档文件名称有着非常固定的格式，比如

```
-rw-rw-r-- 1 elk elk 1899 Jul 7 02:12 logstash-plain-2020-07-06-1.log.gz
-rw-rw-r-- 1 elk elk 117 Jul 8 16:49 logstash-plain-2020-07-07-1.log.gz
```

---

假设输入事件的 message 属性保存的文本就是 Logstash 存档日志文件名称，使用 dissect 过滤器做切分就可以定义如下：

```
filter {
    dissect{
        mapping => {
            message =>
"%{component}-%{log_type}-%{year}-%{month}-%{date}-%{index}.%{extension}.%{co
mpress}"
        }
    }
}
```

在示例中，message 属性定义的切分规则中就包含有两种分隔符“-”和“.”，dissect 过滤器会按分隔符的顺序依次切分文本。切分后的文本会存储到事件属性上，在后续过滤器中，这些属性就可以参与其他处理了。

总的来看，dissect 过滤器虽然不及 grok 过滤器强大，但针对简单文本的处理却也足够了。

## 处理半结构化文本

半结构化文本一般都有固定的格式要求，它们往往都有通用的格式规范，比如 JSON、XML、CSV 等。同样通过可以解析半结构化文本的过滤器插件，它们同样会将解析的结果存储在事件属性上。

### json 过滤器

json 过滤器从属性中读取 JSON 字符串，然后按 JSON 语法解析后将 JSON 属性添加到事件属性上。所以使用 json 插件有一个必须要设置的参数 source，它是用来指定插件从哪一个事件属性中读取 JSON 字符串，例如：

```
filter{
    json {source =>message"}
}
```

### xml 过滤器

类似地，xml 过滤器用于解析 XML 格式的文本，过滤器读取 XML 原始文本的事件属性由 source 参数配置，XML 解析结果存储的事件属性由 target 参数配置。例如：

```
filter(
    xml {
        source => message
        target => doc
    }
)
```

```
    }  
}
```

在示例中，配置的 `xml` 过滤器会从事件的 `message` 属性读取 XML 文本，然后将文本解析后存储到 `doc` 属性上。`xml` 过滤器在解析 XML 时会使用 XML 标签名称作为事件属性名称，而将标签体的文本作为事件属性的值。但如果标签体中嵌套了子标签或者标签本身还带有属性，那么事件的属性就会以散列类型代表这个标签的解析结果。对于标签属性，它们将会以属性名称和值为键值对成为散列结构的一个条目，而标签体中的非标签文本将会以 `content` 为键形成条目。比如：

```
<root>  
  <student id="1">tom</student>  
  <grade>3</grade>  
</root>
```

那么解析后就成为：

```
"doc" => {  
  "student" => {  
    "content" => " tom",  
    "id" => "1"  
  },  
  "grade" => "3"  
}
```

默认情况下，`xml` 过滤器为了防止同名标签覆盖问题，会将每个标签都解析为数组类型。

## csv 过滤器

CSV 是 Comma Separated Values 的简写，即逗号分隔值。它是一种数据格式，以这种格式保存的文件通常扩展名为 CSV。这种格式以纯文本形式表示表格数据，数据的每一行都是一条记录，每个记录又由一个或多个字段组成，用逗号分隔。一些关系型数据库的客户端就提供了导出 CSV 格式数据的功能。

`csv` 过滤器就是根据这种格式将它们解析出来，并将解析结果存储到不同的属性名上。

默认情况下，属性名由 `csv` 过滤器自动生成，名称格式类似于 `column1`、`column2`；属性名也可以通过 `columns` 参数以数组的形式定义。此外，在 CSV 格式中第一行也可以用于表示列名，如示例 14-12 中第行所示。如果希望使用 CSV 文本中定义的列名，则需要通过 `autodetect_column_names` 参数显示地设置首行为列名。

## kv 过滤器

`kv` 过滤器插件用于解析格式为 "`key = value`" 的键值对，多个键值对可以使用空格连接起来，使用 `field_split` 或 `field_split_pattern` 可以定制连接键值对的分

---

隔符。kv 过滤器默认会从 message 属性中取值并解析，解析后的结果会存储在事件的根元素上。例如"username =root&password = root"的分隔符为“&”，比如：

```
message =>"username = root&password = root "
```

则：

```
filter {  
    kv {field_split => "&"}  
}
```

## 事件聚集过滤器

Logstash 提供的 aggregate 过滤器可以将多个输入事件整合到一起，形成一个新的、带有总结性质的输入事件。例如，假设某个业务请求由多个服务处理，每个服务都会生成一些与请求相关的数据。如果需要从这些数据中提取部分信息做整合就可以使用 aggregate 过滤器，比如从每个服务的日志中提取单个服务处理时间以计算整个请求处理了多少时间。

为了将相关事件整合到一起，aggregate 过滤器需要解决两个问题。首先，由于需要将一组输入事件整合起来做处理，所以 aggregate 过滤器需要有办法区分输入事件的相关性；其次，由于需要跨事件做数据处理和运算，所以 aggregate 过滤器需要一个能在多个输入事件之间共享数据的存储空间。

aggregate 过滤器通过任务的概念组合相关事件，并通过 map 对象在事件之间共享数据。

# 日志采集系统构建实战

## 1、首先确保服务器已经运行了 ES 集群

```
[elk@android0318 config]$ ps -ef|grep elasticsearch|grep java  
elk      1551      1  1 Jul04 ?        01:21:45 /usr/local/java/jdk1.8.0_77/bin/java -Xshare:auto -Des.netw  
rkaddress.cache.negative.ttl=10 -XX:+AlwaysPreTouch -Xss1m -Djava.awt.headless=true -Dfile.encoding=UTF-8 -  
eInFastThrow -Dio.netty.noUnsafe=true -Dio.netty.noKeySetOptimization=true -Dio.netty.recycler.maxCapacityI  
mDirectArenas=0 -Dlog4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Djava.locale.providers=SPI,JRE  
epGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -Djava.io.tmpdir=/tmp/elasti  
csearch-7.0.0/config -Dlog4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Djava.locale.providers=SPI,JRE  
epGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -Djava.io.tmpdir=/tmp/elasti  
csearch-7.0.0/config -Des.path.home=/home/elk/elk/teaching/elasticsearch-7.0.0 -Des.path.c  
onfig=/tmp/elk/elk/teaching/elasticsearch-7.0.0/config -Des.bundled_jdk=true -cp /  
h-7.0.0/lib/* org.elasticsearch.bootstrap.Elasticsearch  
elk      1776      1  0 Jul04 ?        00:35:06 /usr/local/java/jdk1.8.0_77/bin/java -Xshare:auto -Des.netw  
rkaddress.cache.negative.ttl=10 -XX:+AlwaysPreTouch -Xss1m -Djava.awt.headless=true -Dfile.encoding=UTF-8 -  
eInFastThrow -Dio.netty.noUnsafe=true -Dio.netty.noKeySetOptimization=true -Dio.netty.recycler.maxCapacityI  
mDirectArenas=0 -Dlog4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Djava.locale.providers=SPI,JRE  
epGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -Djava.io.tmpdir=/tmp/elasti  
csearch-7.0.0/config -Dlog4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Djava.locale.providers=SPI,JRE  
epGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -Djava.io.tmpdir=/tmp/elasti  
csearch-7.0.0/config -Des.path.home=/home/elk/elk/teaching/elasticsearch-2/elasticsearch-  
7.0.0/config -Des.bundled_jdk=true -cp /  
h-7.0.0/lib/* org.elasticsearch.bootstrap.Elasticsearch
```

```

GET _cluster/state/master_node, nodes
PUT /logs-4
{
  "aliases": {
    "logs4": {
      "is_write_index": true
    }
  }
}

POST /logs4/_rollover
{
  "conditions": {
    "max_age": "is",
    "max_docs": 10000,
    "max_size": "4gb"
  }
}

GET logs-1
GET logs-00002
GET logs-00003

GET _cat/indices?v
GET /es-log-2020.07.08/_search

```

## 2、确保服务器上已经安装部署了 logstash

```

[elk@android0318 teaching]$ ls -ll
total 742064
drwxrwxr-x 3 elk elk 4096 Jun 27 22:05 elasticsearch-2
drwxr-xr-x 10 elk elk 4096 Jun 9 20:38 elasticsearch-7.7.0
-rwxrwxr-x 1 elk elk 314430566 Jun 9 15:54 elasticsearch-7.7.0-linux-x86_64.tar.gz
drwxrwxr-x 13 elk elk 4096 Jun 9 20:44 kibana-7.7.0-linux-x86_64
-rwxrwxr-x 1 elk elk 278962064 Jun 9 15:54 kibana-7.7.0-linux-x86_64.tar.gz
drwxr-xr-x 13 elk elk 4096 Jul 6 23:07 logstash-7.7.0
-rwxrwxr-x 1 elk elk 166451553 Jun 9 15:54 logstash-7.7.0.tar.gz

```

## 3、在 Logstash 安装路径下的 config 目录中，新建一个 conf 文件，取名为 es\_log.conf，并且填入以下内容：

```

input {
  file {
    path => "/home/elk/elk/teaching/elasticsearch-7.7.0/logs/my-elk.log"
    start_position => "beginning"
    codec => multiline {
      pattern => "^\\["
      negate => true
      what => "previous"
    }
  }
}

output {
  elasticsearch {
    hosts => ["http://172.18.194.140:9200"]
    index => "es-log-%{+YYYY.MM.dd}"
    #user => "elastic"
    #password => "changeme"
  }
  stdout{}
}

```

```

input {
  file {
    path => "/home/elk/elk/teaching/elasticsearch-7.7.0/logs/my-elk.log"
    start_position => "beginning"
    codec => multiline {
      pattern => "^\\["
      negate => true
    }
  }
}

```

```
        what => "previous"
    }
}
}

output {
    elasticsearch {
        hosts => ["http://172.18.194.140:9200"]
        index => "es-log-%{+YYYY.MM.dd}"
    }
    stdout{}
}
```

#### 4、进入 Logstash 的 bin 目录执行

```
./logstash -f ..//config/es_log.conf
```

#### 5、控制台会输出：

```
[2020-07-08T17:38:04,341] [INFO ] [logstash.agent]           Successfully started Logstash API endpoint {:port=>9600}
/home/elk/elk/teaching/logstash-7.7.0/vendor/bundle/jruby/2.5.0/gems/awesome_print-1.7.0/lib/awesome_print/formatters/base_formatter.rb:31: warning: constant ::Fixnum is deprecated
{
  "@timestamp" => 2020-07-08T09:38:04.646Z,
  "@version" => "1",
  "host" => "xiangxue",
  "path" => "/home/elk/elk/teaching/elasticsearch-7.7.0/logs/my-elk.log",
  "message" => "[2020-07-08T00:30:00,003][INFO ][o.e.x.m.a.TransportDeleteExpiredDataAction] [node-1] Deleting expired data"
}
{
  "@timestamp" => 2020-07-08T09:38:04.647Z,
  "@version" => "1",
  "host" => "xiangxue",
  "path" => "/home/elk/elk/teaching/elasticsearch-7.7.0/logs/my-elk.log",
  "message" => "[2020-07-08T00:30:00,028][INFO ][o.e.x.m.a.TransportDeleteExpiredDataAction] [node-1] Completed deletion of expired ML data"
}
```

#### 6、我们到浏览器查询：

健康状态	索引	uuid	pri	rep	docs.count	docs.deleted	store.size	pri.store.size
green	open	high_sdk_test	1Vc7tv asRWieGhLb59w4XQ	1	1	1	0	8.8kb
green	open	pattern_custom	zuakPIa6tFuvcc6z7xMDHQ	1	1	0	0	416b
green	open	students	MqPQohC6RGCx bZM2v+j-wug	1	1	1	0	7.1kb
green	open	study_route1	6BwyiZokSanc4Co2pLP6KA	2	0	1	0	9.1kb
green	open	logs-1	RNR6hNOYSgaIvJ7iYYDbA	1	1	0	0	416b
green	open	test_es	j6p32vcgkSwaBtjhNtNCw	1	1	0	0	416b
green	open	kibana_sample_data_flights	1oPiSR-RR-SixhsN3Irwfw	1	1	13059	0	12.4mb
green	open	test1	C2D0YiHYRgagZdu03xx7A	1	1	0	0	416b
green	open	logs-4	aCSBzajsORSaOMpjUeVcA	1	1	0	0	416b
green	open	test-user	sPXOr5p_RduMdzuNEYGm5g	1	1	8	4	47.1kb
green	open	high_sdk	6dxoXZSgReSOPfGeCM1MSg	1	1	1	0	9.1kb
green	open	mystdin	almtfaV4qTT6H17y_81vEq	1	1	1	0	10.2kb
green	open	.apm-custom-link	1InBD4_AT556sfgj7o_XpA	1	1	0	0	416b
green	open	.monitoring-es-7-2020.07.01	XWP8chMnUDTc1PyzmfxFxWmQ	1	1	153093	248508	182.2mb
green	open	.kibana_task_manager_1	D2GrMLN7Q0qeX1VfAWRhvKA	1	1	5	2	112.2kb
green	open	.monitoring-es-7-2020.07.02	nYQphRFhQYCY3Hm_ZnXsaA	1	1	294659	157808	299mb
green	open	.monitoring-es-7-2020.07.03	EbM1dc0X56qK02aBrAzSrzQ	1	1	207657	20580	201.2mb
green	open	.monitoring-es-7-2020.07.04	H7nvj1MsSeZPZKuKcbBWRw	1	1	148611	253834	161.1mb
green	open	.monitoring-es-7-2020.07.05	8clt65wsbHgHN436P7Pq	1	1	415396	50650	373.2mb
green	open	.monitoring-es-7-2020.07.06	aEPGcTeySxKmy46jmC3B0A	1	1	444043	105616	420.8mb
green	open	.monitoring-es-7-2020.07.07	Zc0gCOIRSAWIukkZqAr0Qw	1	1	453713	121624	425.9mb
green	open	enjoy_test	vyUCvaaEsdupMOXYevZD1w	1	1	3	0	21.3kb
green	open	.monitoring-es-7-2020.07.08	g3964kaGS-GenYCFCH9u1Q	1	1	190932	338908	422.4mb
green	open	pattern_test4	o81bkFUVSmrpMuICtktmnzg	1	1	2	0	15.2kb
green	open	.monitoring-kibana-7-2020.07.04	ktyLTUc801G4mrKMB624mw	1	1	3386	0	1.4mb
green	open	.monitoring-kibana-7-2020.07.03	iPZvoQDMTQyQD9Hz96RQQ	1	1	5764	0	2.3mb
green	open	.monitoring-kibana-7-2020.07.06	KT7q_1VEQWKTJ-9SN7wvf	1	1	8629	0	3.5mb
green	open	.monitoring-kibana-7-2020.07.05	o8LvrYRpQCqAwtpErcD8BxA	1	1	8639	0	3.5mb
green	open	test	jc5A0HfHQIC1CHL_DR9d0w	1	1	1	1	8.4kb
green	open	.monitoring-kibana-7-2020.07.02	MvxHeVWCSeasKpZJ2p9K1A	1	1	8639	0	3.6mb
green	open	open-soft	7MkwPOUTNufthaRtEQxTA	1	1	4	0	16.2kb
green	open	.monitoring-kibana-7-2020.07.01	J4AtqfNLQmdLR1VApUWnqA	1	1	4781	0	2.1mb
green	open	.apm-agent-configuration	RfrV5PbTSyRyp5I8vPCxW	1	1	0	0	416b
green	open	colleges	JFLduTz8S1Gle0nvoql-g	1	1	5	0	18.2kb
green	open	.monitoring-kibana-7-2020.07.08	K_j_7MbgrTlwazSn41KpUA	1	1	467	0	947.6kb
green	open	.monitoring-kibana-7-2020.07.07	1YEFtP02qwm8gbC7jf6w	1	1	8639	0	3.4mb
green	open	.kibana_1	a8c7Cvya95sASADRLLd4bg5w	1	1	658	21	1mb
green	open	es_controller_test	x_tUAHLL7t_eyl15DGHGfA1A	1	1	0	0	416b
green	open	route_test	8ztrW11PTO-RYbeBM1qkoA	1	1	1	0	7.7kb
green	open	open-soft-shard	IuFoBnHFTo-Py706Vytng	2	1	4	0	42.6kb
green	open	kibana_sample_data_logs	No0_4Af_TAm6h01xN9vJQ	1	1	14074	0	23.1mb
green	open	study_route	JNG1434T01-LdMjaa1fgtw	2	0	4	0	19.3kb
green	open	employees	tdERz1xf0-17cyNv-JxZw	1	1	3	0	17.8kb
green	open	logs-000005	VZ5y_HzJrBwgNF01jqyBgQ	1	1	0	0	416b
green	open	articles	nJnlif1CP7110CVT15rHEjw	1	1	1	0	10.6kb
green	open	logs-000003	hxHKR7N9Rqu3PmmGLukIA	1	1	0	0	416b
green	open	es-log-2020.07.08	IfnpdfTdiTA-3G-4Ns-CtzQ	1	1	15	0	53.8kb
green	open	logs-000002	OJ_fzrZRR-0AX3yqr_uubA	1	1	0	0	416b

发现 ES 中确实创建了一个索引 es-log-2020.07.08

## 7、在 ES 中查询这个索引的内容

8、仔细检查存入的日志内容，发现日志信息是作为整体存入 message 字段的：

```
"message" : "[2020-07-08T08:00:01,020][INFO ][o.e.c.r.a.AllocationService] [node-1]
Cluster health status changed from [YELLOW] to [GREEN] (reason: [shards started [[
.monitoring-kibana-7-2020.07.08][0]]])."
```

有没有办法存入的更细粒些呢？

9、仔细分析 es 的日志，呈现了一定的结构化特征：

```
[2020-07-09T00:30:00,000][INFO ][o.e.x.m.MlDailyMaintenanceService] [node-1] triggering scheduled [ML] maintenance tasks
```

总是：“[时间戳][日志级别][输出信息的类名][节点名]具体的日志信息

”这种格式，所以我们完全可以考虑使用过滤器插件对文本进行分析后再存入 es，怎么分析？可以用 grok 过滤器。

10、在 Logstash 安装路径下的 config 目录中，新建一个 conf 文件，取名为 log\_grok.conf，并且填入以下内容：

```
input {
    file {
        path => "/home/elk/elk/teaching/elasticsearch-7.7.0/logs/my-elk.log"
        start_position => "beginning"
        codec => multiline {
            pattern => "^["
            negate => true
            what => "previous"
        }
    }
}

filter{
    grok {
        match => {
            message => "\[%{TIMESTAMP_ISO8601:time}\]\[%{LOGLEVEL:level}%{SPACE}\]\[%{NOTSPACE:loggerclass}%{SPACE}\]%
            SPACE\[%{DATA:nodename}\] %{SPACE}%{GREEDYDATA:msg}"
        }
    }
}

output {
    stdout{}
}
```

```
input {
    file {
        path => "/home/elk/elk/teaching/elasticsearch-7.7.0/logs/my-elk.log"
        start_position => "beginning"
        codec => multiline {
            pattern => "^["
            negate => true
            what => "previous"
        }
    }
}

filter{
    grok {
        match => {
```

```
        }
```

```

    message =>
"\[%{TIMESTAMP_ISO8601:time}\]\[%{LOGLEVEL:level}%{SPACE}\]\[%{NOT
SPACE:loggerclass}%{SPACE}\]\%{SPACE}\]\[%{DATA:nodename}\]\%{SPACE}
%\{GREEDYDATA:msg\}"
}

}

}

output {
    stdout{}
}

```

11、如果担心自己写的 conf 有语法问题，可以执行  
`./logstash -f xxxxxxx.conf -t` 检查 conf，当然只能检查语法，不能检查诸如正则表达式是否正确这类问题。

```
[elk@android0318 bin]$ ./logstash -f ../config/log_grok.conf -t
Sending Logstash logs to /home/elk/elk/teaching/logstash-7.7.0/logs which is now configured via log4j2.properties
[2020-07-09T17:42:10,368][WARN ][logstash.config.source.multilocal] Ignoring the 'pipelines.yml' file because modules or command line options are specified
[2020-07-09T17:42:11,839][INFO ][org.reflections.Reflections] Reflections took 42 ms to scan 1 urls, producing 2
1 keys and 41 values
Configuration OK
[2020-07-09T17:42:13,457][INFO ][logstash.runner] ] Using config.test_and_exit mode. Config Validation Result: OK. Exiting Logstash
```

12、执行`./logstash -f ../config/log_grok.conf`，如果发现程序没有输出，有可能是 elk 的日志文件已经被处理过，logstash 不会重复处理，这时可以到 logstash 的`/data/plugins/inputs/file` 目录下，删除`.sinceedb` 文件（这个文件是个隐藏文件），再执行。

```
[elk@android0318 file]$ pwd
/home/elk/elk/teaching/logstash-7.7.0/data/plugins/inputs/file
[elk@android0318 file]$ ls -al
total 12
drwxrwxr-x 2 elk elk 4096 Jul  9 16:59 .
drwxrwxr-x 3 elk elk 4096 Jul  8 17:38 ..
-rw-rw-r-- 1 elk elk 101 Jul  9 18:01 .sinceedb 6d0dfacc8bae86f72bfef82de20ddb51
[elk@android0318 file]$ more .sinceedb 6d0dfacc8bae86f72bfef82de20ddb51
2127220 0 64769 478941 1594279883.0212429 /home/elk/elk/teaching/elasticsearch-7.7.0/logs/my-elk.log
[elk@android0318 file]$ rm .sinceedb_6d0dfacc8bae86f72bfef82de20ddb51
[elk@android0318 file]$ ls -al
total 8
drwxrwxr-x 2 elk elk 4096 Jul  9 18:04 .
drwxrwxr-x 3 elk elk 4096 Jul  8 17:38 ..
```

```
{
    "@timestamp" => 2020-07-09T10:06:51.063Z,
    "@version" => "1",
    "loggerclass" => "o.e.c.m.MetaDataMappingService",
    "message" => "[2020-07-09T15:31:21,457][INFO ][o.e.c.m.MetaDataMappingService] [node-1]
0.07.09/3lZZxfyQSc-P1nE9gED60Q] update_mapping [_doc]",
    "path" => "/home/elk/elk/teaching/elasticsearch-7.7.0/logs/my-elk.log",
    "host" => "xiangxue",
    "time" => "2020-07-09T15:31:21,457",
    "msg" => "[es-log-text-2020.07.09/3lZZxfyQSc-P1nE9gED60Q] update_mapping [_doc]",
    "nodename" => "node-1",
    "level" => "INFO"
```

可以看见，每条日志已经成功的被解析了。

13、现在考虑如何存入 es，按照原来的

```

elasticsearch {
    hosts => ["http://172.18.194.140:9200"]
    index => "es-log-%{+YYYY.MM.dd}"
    #user => "elastic"
    #password => "changeme"
}

```

是不行的，必须还要做点改变，怎么改？在 config 目录下创建一个 es\_template.json

```

[elk@android0318 config]$ more es_template.json
{
  "template": "es-log-text-%{+YYYY.MM.dd}",
  "settings": {
    "index.refresh_interval": "1s"
  },
  "mappings": {
    "properties": {
      "time": {
        "type": "date"
      },
      "level": {
        "type": "keyword"
      },
      "loggerclass": {
        "type": "keyword"
      },
      "nodename": {
        "type": "keyword"
      },
      "msg": {
        "type": "text"
      },
      "message": {
        "type": "text"
      }
    }
  }
}

```

定义好索引的 mapping（映射），注意要和我们对日志的分解一一对应。

14、创建一个新的 conf 文件，取名 es\_log\_grok.conf，内容如下：

```

input {
  file {
    path => "/home/elk/elk/teaching/elasticsearch-7.7.0/logs/my-elk.log"
    start_position => "beginning"
    codec => multiline {
      pattern => "\^["
      negate => true
      what => "previous"
    }
  }
}

filter{
  grok {
    match => {
      message => "\[%{TIMESTAMP_ISO8601:time}\] \[%{LOGLEVEL:level}\] \[%{SPACE}%{DATA:nodename}\] \[%{GREEDYDATA:msg}\]"
    }
  }
}

output {
  elasticsearch {
    hosts => ["http://172.18.194.140:9200"]
    index => "es-log-text-%{+YYYY.MM.dd}"
    template_name => "es_template"
    template => "/home/elk/elk/teaching/logstash-7.7.0/config"
  }
  stdout{}
}

```

---

```
input {
  file {
    path => "/home/elk/elk/teaching/elasticsearch-7.7.0/logs/my-elk.log"
    start_position => "beginning"
    codec => multiline {
      pattern => "^\\["
      negate => true
      what => "previous"
    }
  }
}

filter{
  grok {
    match => {
      message =>
"\[%{TIMESTAMP_ISO8601:time}\]\[%{LOGLEVEL:level}%{SPACE}\]\[%{NOT
SPACE:loggerclass}%{SPACE}\]\%{SPACE}\]\%{DATA:nodename}\]\%{SPACE}
%\{GREEDYDATA:msg\}"
    }
  }
}

output {
  elasticsearch {
    hosts => ["http://172.18.194.140:9200"]
    index => "es-log-text-%{+YYYY.MM.dd}"
    template_name => "es_template*"
    template => "/home/elk/elk/teaching/logstash-7.7.0/config"
  }
  stdout{}
}
```

主要是在 `elasticsearch` 插件中增加了对刚才新增的 `json` 文件的读取。

15、执行 `./logstash -f ./config/es_log_grok.conf`, 会看到控制台的输出, 同时在浏览器中, 也会看到解析后的日志:

```

GET /es-log-text-2020.07.09/_search
GET /es-log-2020.07.08/_search

PUT /logs-4
{
  "aliases": {
    "logs4": {
      "is_write_index": true
    }
  }
}

POST /logs4/_rollover
{
  "conditions": {
    "max_age": "1s",
    "max_docs": 10000,
    "max_size": "4gb"
  }
}

GET logs-1
GET logs-000002
GET logs-000003

GET _cat/indices?v
GET /es-log-2020.07.08/_search

POST _reindex
{
  "source": {
    "index": "logs-1"
  },
  "size": 100000
}

12   "value" : 135,
13   "relation" : "eq"
14 },
15   "max_score" : 1.0,
16   "hits" : [
17     {
18       "index" : "es-log-text-2020.07.09",
19       "type" : "doc",
20       "id" : "gIVBmHMBx1hhL2qoac15",
21       "score" : 1.0,
22       "source" : {
23         "host" : "xiangxue",
24         "time" : "2020-07-09T00:30:00,000",
25         "path" : "/home/elk/elk/teaching/elasticsearch-7.7.0/logs/my-elk.log",
26         "level" : "INFO",
27         "loggerclass" : "o.e.x.m.MlDailyMaintenanceService",
28         "nodename" : "node-1",
29         "msg" : "triggering scheduled [ML] maintenance tasks",
30         "message" : "[2020-07-09T00:30:00,000][INFO ][o.e.x.m.MlDailyMaintenanceService] [node-1]
31           triggering scheduled [ML] maintenance tasks",
32         "@timestamp" : "2020-07-09T07:31:20.084Z",
33         "@version" : "1"
34     },
35   },
36   {
37     "index" : "es-log-text-2020.07.09",
38     "type" : "doc",
39     "id" : "jVJBmHMBx1hhL2qoas2X",
40     "score" : 1.0,
41     "source" : {
42       "loggerclass" : "o.e.t.TcpTransport",
43       "msg" : "exception caught on transport Layer [Netty4TcpChannel{LocalAddress=/172.18.194
44 .140:9300, remoteAddress=/101.133.136.80:55420}], closing connection
io.netty.handler.codec.DecoderException: java.io.StreamCorruptedException: invalid internal
transport message format, got (0,c,0,0)
at io.netty.handler.codec.BvtToMessageDecoder.callDecode(BvtToMessageDecoder.java:468) ~[netty

```

# Elasticsearch 性能优化指引

## 合并请求

为了获得更快的索引速度，你能做的一项优化是通过批量 API，一次发送多个命令进行操作。这个操作将节省网络来回的开销，并产生更大的索引吞吐量。一个单独的批量可以接受任何索引操作。

## 优化 Lucene 分段的处理

一旦 Elasticsearch 接收到了应用所发送的文档，它会将其索引到内存中称为分段( segments )的倒排索引。这些分段会不时地写入磁盘。这些分段是不能改变的，只能被删除，这是为了操作系统更好地缓存它们。另外，较大的分段会定期从较小的分段创建而来，用于优化倒排索引，使搜索更快。

有很多调节的方式来影响每一个环节中 Elasticsearch 对于这些分段的处理，根据你的使用场景来配置这些，常常会带来意义重大的性能提升。本节将讨论这些调优的方式，可以将它们分为以下 3 类。

■ 刷新(refresh)和冲刷(flush)的频率——刷新会让 Elasticsearch 重新打开索引，让新建的文档可用于搜索。冲刷是将索引的数据从内存写入磁盘。从性能的角度来看，刷新和冲刷操作都是非常消耗资源的，所以为你的应用正确地配置它们是十分重要的。

■ 合并的策略——Lucene ( Elasticsearch 也是如此)将数据存储在不可变的一组文件中，也就是分段中。随着索引的数据越来越多，系统会创建更多的分段。由于在过多的分段中搜索是很慢的，因此在后台小分段会被合并为较大的分段，保持分段的数量可控。不过，合并也是十分消耗性能的，对于 I/O 子系统尤其如此。你可以调节合并的策略，来确定合并多久发生一次，而且分段应该合并到多大。

---

■ 存储和存储限流—— Elasticsearch 调节每秒写入的字节数, 来限制合并对于 I/O 系统的影响。根据硬件和应用, 你可以调整这个限制。还有一些其他的选项告诉 Elasticsearch 如何使用存储。例如, 可以选择只在内存中存放索引。

## 刷新和冲刷的阈值

Elasticsearch 通常被称为近实时(或准实时)系统。这是因为搜索不是经常运行于最新的索引数据之上(这个数据是实时的), 但是很接近了。打上近实时的标签是因为通常 Elasticsearch 保持了某个时间点索引打开的快照, 所以多个搜索会命中同一个文件并重用相同的缓存。在这个时间段中, 新建的文档对于那些搜索是不可见的, 直到你再次刷新。

刷新, 正如其名, 是在某个时间点刷新索引的快照, 这样你的搜索就可以命中新索引的数据。这是其优点。其不足是每次刷新都会影响性能:某些缓存将失效, 拖慢搜索请求, 而且重新打开索引的过程本身也需要一些处理能力, 拖慢了索引的建立。

## 何时刷新

默认的行为是每秒自动地刷新每份索引。你可以修改其设置, 改变每份索引的刷新间隔, 这个是在运行时完成的。例如, 下面的命令将自动刷新的间隔设置为了 5 秒。

```
put test1
{
  "settings":{
    "index.refresh_interval ":"5s"
  }
}
```

当增加 `refresh_interval` 的值时, 你将获得更大的索引吞吐量, 因为花费在刷新上的系统资源更少了。

或者你也可以将 `refresh_interval` 设置为 -1, 彻底关闭自动刷新并依赖手动刷新。这对于索引只是定期批量变化的应用非常有效, 如产品和库存每晚更新的零售供应链。索引的吞吐量是非常重要的, 因为你总想快速地进行更新, 但是数据刷新不一定是最主要的, 因为无论如何都不可能获得完全实时的更新。所以每晚你可以关闭自动刷新, 进行批量的 `bulk` 索引和更新, 完成后再进行手动刷新。

为了实现手动刷新, 访问待刷新索引的 `refresh` 端点。

## 何时冲刷

对于 Elasticsearch 而言, 刷新的过程和内存分段写入磁盘的过程是相互独立的。实际上, 数据首先索引到内存中, 经过一次刷新后, Elasticsearch 也会地搜索相应的内存分段。将内存中的分段提交到磁盘上的 Lucene 索引的过程, 被称为冲刷(`flush`), 无论分段是否能被搜到, 冲刷都会发生。

---

为了确保某个节点宕机或分片移动位置的时候，内存数据不会丢失，Elasticsearch 将使用事物日志来跟踪尚未冲刷的索引操作。除了将内存分段提交到磁盘，冲刷还会清理事物日志，

满足下列条件之一就会触发冲刷操作。

- 内存缓冲区已满。
- 自上次冲刷后超过了一定的时间。
- 事物日志达到了一定的阈值。

为了控制冲刷发生的频率，你需要调整控制这 3 个条件的设置。

内存缓冲区的大小在 `elasticsearch.yml` 配置文件中定义，通过 `indices.memory.index_buffer_size` 来设置。这个设置控制了整个节点的缓冲区，其值可以是全部 JVM 堆内存的百分比，如 10%，也可以是 100 MB 这样的固定值。

事物日志的设置是具体到索引上的，而且同时控制了触动冲刷的规模(通过 `index.translog.flush_threshold_size`)和多数索引设置一样,你可以在运行时修改它们。

```
put test1
{
  "settings":{
    "index.translog.flush_threshold_size": "500mb"
  }
}
```

当冲刷发生的时候，它会在磁盘上创建一个或多个分段。执行一个查询的时候, Elasticsearch(通过 Lucene )查看所有的分段，然后将结果合并到一个整体的分片中。

关于分段，这里需要记住的关键点是你需要搜索的分段越多,搜索的速度就越慢。为了防止分段的数量失去控制，Elasticsearch (也是通过 Lucene )在后台将多组较小的分段合并为较大的分段。

## JVM 堆和操作系统缓存

如果 Elasticsearch 没有足够的堆来完成一个操作，它将抛出一个 `out-of-memory` 的异常，很快该节点就会宕机，并被移出集群。这会给其他的节点带来额外的负载,因为系统需要复制和重新分配分配，以恢复到初始配置所需的状态。由于节点通常是相同的配置，额外的负载很可能使得至少另一个节点耗尽内存，这种多米诺骨牌效应会拖垮整个集群。

当 JVM 堆的资源很紧张的时候，即使在日志中没有看见 `out-of-memory` 的异常，节点还是可能变得没有响应。这可能是因为，内存不够迫使垃圾回收器( GC) 运行得更久或者更频繁来释放空闲的内存。由于 GC 消耗了更多的 CPU 资源，节点花费在服务请求甚至是应答主节点 ping 的计算能力就更少了，最后导致节点被移出集群。

---

当垃圾回收消耗了过多的 CPU 时间，工程师总是试图发现些神奇的 JVM 设置来解决所有的麻烦。多数情况下，这不是解决问题的最佳途径，因为过度的垃圾回收只是 Elasticsearch 需要更多堆内存的征兆。

尽管增加堆的大小是很明显的解决方案，这并非总是可能的。增加数据节点同样如此。然而，你可以参考些技巧来降低堆的使用。

- 减小索引缓冲区的大小。
- 减少查询缓存的大小。
- 减少搜索和聚集请求中 size 参数的值(对于聚集，还需要考虑到 shard size)。
- 如果必须处理大规模的数据，你可以增加一些非数据节点和非主节点来扮演客户端的角色。它们将负责聚合每个分片的搜索结果，以及聚集操作。

最后，Elasticsearch 使用另一种缓存类型来回避 Java 垃圾回收的方式。新的对象分配到名为新生代的空间。如果新对象存活得足够久，或者太多的新对象分配到新生代导致其被占满，这些新对象就会被“提升”到老年代。尤其是在聚集的时候，需要遍历大量的文档并创建很多可能被重用的对象，后一种填满新生代的情况就会出现。

通常，你希望将这些可能被聚集重用的对象提升到老年代，而不是新生代填满后恰巧放到老年代的一些随机的、临时的对象。为了实现这个目标，Elasticsearch 实现了一个页面缓存回收器( `PageCacheRecycler` )，其中被聚集所使用的大数组被保留下，不会被垃圾回收。默认的页面缓存是整个堆的 10%，某些情况下这个值可能太大(例如，你有 30 GB 的堆内存，页面缓存就有 3GB 了)。可以设置 `elasticsearch.yml` 中的 `cache . recycler . page . limit . heap` 来控制该缓存的大小。

不过，有的时候还是需要调优 JVM 的设置(尽管默认已经很好了)，例如，你的内存基本够用，但是当一些罕见的长时间 GC 停顿产生的时候，集群还是会遇到麻烦。有一些设置让垃圾回收发生得更频繁，但是停顿时间更短，降低些整体的吞吐量来换取更少的延迟：

增加幸存者区( `Survivor` )的空间(降低-`XX:SurvivorRatio`)或者是整个新生代在堆中的占比(降低-`XX:NewRatio`)。你可以监控不同的年代来判断这些是否需要。提升到老年代需要花费更长的等待时间，而更富裕的空间使得新生代的垃圾回收有较多的时间来清理临时存在的对象：避免它们被提升到老年代。但是，将新生代设置得过大，将使得新生代的垃圾回收过于频繁而且变得效率低下，这是因为生存周期更长的对象需要在两个幸存者空间来复制。

理想的堆大小：遵循“一半”原则

在不知道堆的实际使用情况时，经验法则是将节点内存的一半分配给 Elasticsearch，但是不要超过 32GB。这个“一半”法则通常给出了堆大小和系统缓存之间良好的均衡点。

如果可以监控实际的堆使用情况，一个好的堆大小就是足够容纳常规的使用，外加可预期的高峰冲击。内存使用的高峰可能会发生。例如，某些人决定在拥有许多唯一词条的分析字段上，对所有结果运行一次 `terms` 聚集。这强迫 Elasticsearch 将所有的词条加载到内存中用于统计。如果你不知道会有怎样的高峰冲击，经验法则同样是一半：将堆大小设置为比常规高出 50%。

# ELK 相关面试题

## ELK 是什么？

ELK 其实并不是一款软件，而是一整套解决方案，是三个软件产品的首字母缩写

Elasticsearch: 负责日志检索和储存

Logstash: 负责日志的收集和分析、处理

Kibana: 负责日志的可视化

这三款软件都是开源软件，通常是配合使用，而且又先后归于 Elastic.co 公司名下，故被简称为 ELK。加入 Beats 系列组件后，官方名称就变为了 Elastic Stack

## ELK 能做什么？

ELK 组件在海量日志系统的运维中，可用于解决：

分布式日志数据集中式查询和管理、系统监控，包含系统硬件和应用各个组件的监控、故障排查、安全信息和事件管理、报表功能等等

## 简要概述 Elasticsearch ?

ElasticSearch 是一个基于 Lucene 的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于 RESTful API 的 web 接口。

Elasticsearch 是用 Java 开发的，并作为 Apache 许可条款下的开放源码发布，是当前流行的企业级搜索引擎。设计用于云计算中，能够达到实时搜索，稳定，可靠，快速，安装使用方便

## Elasticsearch 主要特点

1. 实时分析
  2. 分布式实时文件存储，并将每一个字段都编入索引
  3. 文档导向，所有的对象全部是文档
  4. 高可用性，易扩展，支持集群(Cluster)、分片和复制(Shards 和 Replicas)
- 接口友好，支持 JSON

## ES 相关概念

Node: 装有一个 ES 服务器的节点。

Cluster: 有多个 Node 组成的集群

Document: 一个可被搜索的基础信息单元

Index: 拥有相似特征的文档的集合

Type: 一个索引中可以定义一种或多种类型

Filed: 是 ES 的最小单位，相当于数据的某一列

Shards: 索引的分片，每一个分片就是一个 Shard

Replicas: 索引的拷贝

## ES 与关系型数据库的对比

关系数据库	ES
库	索引 (Index)
表	映射类型 (Mapping Type)
数据行	文档 (Document)
字段	文档字段 (Field)
表结构	映射 (Mapping)

## 什么是分词器

分词是将文本转换成一系列单词 (Term or Token) 的过程，也可以叫文本分析，在 ES 里面称为 Analysis

分词器是 ES 中专门处理分词的组件，英文为 Analyzer，它的组成如下：

Character Filters: 针对原始文本进行处理，比如去除 html 标签

Tokenizer: 将原始文本按照一定规则切分为单词

Token Filters: 针对 Tokenizer 处理的单词进行再加工，比如转小写、删除或增新等处理

ES 提供了一个可以测试分词的 API 接口，方便验证分词效果，endpoint 是 \_analyze

ES 也提供了很多内置的分析器。

## elasticsearch 的倒排索引是什么？

正排索引是以文档的 ID 为关键字，表中记录文档中每个字的位置信息，查找时扫描表中每个文档中字的信息直到找出所有包含查询关键字的文档。

而倒排索引，是通过分词策略，形成了词和文章的映射关系表，这种词典+映射表即为倒排索引。

有了倒排索引，就能实现  $O(1)$  时间复杂度的效率检索文章了，极大的提高了检索效率。

所以总的来说，正排索引是从文档到关键字的映射（已知文档求关键字），倒排索引是从关键字到文档的映射（已知关键字求文档）。

## Elasticsearch 是如何实现 Master 选举的？

采用 Bully 算法，它假定所有节点都有一个唯一的 ID，使用该 ID 对节点进行排序。Elasticsearch 的选主是 ZenDiscovery 模块负责的，主要包含 Ping (节

---

点之间通过这个 **RPC** 来发现彼此) 和 **Unicast** (单播模块包含一个主机列表以控制哪些节点需要 ping 通) 这两部分;

对所有可以成为 **master** 的节点(`node.master: true`)根据 `nodeId` 字典排序, 每次选举每个节点都把自己所知道节点排一次序, 然后选出第一个(第 0 位)节点, 暂且认为它是 **master** 节点。

如果对某个节点的投票数达到一定的值(可以成为 **master** 节点数  $n/2+1$ ) 并且该节点自己也选举自己, 那这个节点就是 **master**。否则重新选举一直到满足上述条件。

补充: **master** 节点的职责主要包括集群、节点和索引的管理, 不负责文档级别的管理; **data** 节点可以关闭 **http** 功能。

7.X 之后的 ES, 采用一种新的选主算法, 实际上是 **Raft** 的实现, 但并非严格按照 **Raft** 论文实现, 而是做了一些调整。**Raft** 是工程上使用较为广泛分布式共识协议, 是多个节点对某个事情达成一致的看法, 即使是在部分节点故障、网络延时、网络分区的情况下。

## Elasticsearch 如何避免脑裂?

ES 集群中的节点(比如共 20 个), 其中的 10 个选了一个 **master**, 另外 10 个选了另一个 **master**, 怎么办?

当集群 **master** 候选数量不小于 3 个时, 可以通过设置最少投票通过数量 (`discovery.zen.minimum_master_nodes`) 超过所有候选节点一半以上来解决脑裂问题;

当候选数量为两个时, 只能修改为唯一的一个 **master** 候选, 其他作为 **data** 节点, 避免脑裂问题。

在 Elasticsearch 7.0 里重新设计并重建了的集群协调子系统, 移除 `minimum_master_nodes` 参数, 转而由集群自主控制。

## 详细描述一下 Elasticsearch 索引文档的过程

协调节点默认使用文档 ID 参与计算(也支持通过 **routing**), 以便为路由提供合适的分片。

`shard = hash(document_id) % (num_of_primary_shards)`

当分片所在的节点接收到来自协调节点的请求后, 会将请求写入到 **Memory Buffer**, 然后定时(默认是每隔 1 秒)写入到 **Filesystem Cache**, 这个从 **Memory Buffer** 到 **Filesystem Cache** 的过程就叫做 **refresh**;

当然在某些情况下, 存在 **Memory Buffer** 和 **Filesystem Cache** 的数据可能会丢失, ES 是通过 **translog** 的机制来保证数据的可靠性的。其实现机制是接收到请求后, 同时也会写入到 **translog** 中, 当 **Filesystem cache** 中的数据写入到磁盘中时, 才会清除掉, 这个过程叫做 **flush**;

在 **flush** 过程中, 内存中的缓冲将被清除, 内容被写入一个新段, 段的 **fsync** 将创建一个新的提交点, 并将内容刷新到磁盘, 旧的 **translog** 将被删除并开始一个新的 **translog**。

---

`flush` 触发的时机是定时触发（默认 30 分钟）或者 `translog` 变得太大（默认为 512M）时；

## 详细描述一下 Elasticsearch 搜索的过程？

搜索拆解为“`query then fetch`”两个阶段。

`query` 阶段的目的：定位到位置，但不取。

步骤拆解如下：

1) 假设一个索引数据有 5 主+1 副本 共 10 分片，一次请求会命中（主或者副本分片中）的一个。

2) 每个分片在本地进行查询，结果返回到本地有序的优先队列中。

3) 第 2) 步骤的结果发送到协调节点，协调节点产生一个全局的排序列表。

`fetch` 阶段的目的：取数据。

路由节点获取所有文档，返回给客户端。