

ElasticSearch 课程教案

1、搜索的介绍

搜索是指搜寻检索，指代使用一定手段来检索到我们自己需要的信息，包括从文件当中检索，百度当中检索，网站内部搜索等等

2、全文检索的介绍

1、全文检索的需求介绍

首先我们谈几个公司，如雷贯耳的：百度、谷歌、维基百科；这些公司都有一个相似性就是门户网站，可以提供我们通过关键字搜索，然后快速的检索出我们想要的信息；

【网页百度展示】

比如我们检索传智播客，百度后台就会按照这个关键字进行查找（里面有搜索库，以及爬虫库），然后按照权重来进行从上到下的排序，给我们高亮的展示出现

【京东或者淘宝展示】

随便搜索东西，就会高精度的展示我们想要的；就会根据关键词进行海量数据的快速的检索

比如我们查找：“护手霜”，那么这期间内部会经过大体的：1、分词（护手，手霜，护等）2、根据这些词去海量的数据中检索 3、然后根据权重把检索出来的信息进行排序展示给我们

【传统做法】

那么对于一般的公司，初期是没有那么多数据的，所以很多公司更倾向于使用传统的数据库：mysql；比如我们要查找关键字“传智播客”，那么查询的方式大概就是：`select * from table where field like '%传智播客%'`；但是随着业务发展，数据会不断的膨胀，那么问题就来了；mysql 单表查询能力即便经过了优化，它的极限也就是400W左右的数据量。而且还会经常出现查询超时的现象；

然后很多公司开始对数据库进行横向和纵向的扩容，开始进行数据库表的“拆分”：横向拆分和纵向拆分；但是即便这样操作，仍然会出现很多问题，比如：

- 1、数据库会出现单点故障问题，于是先天主从复制关系，于是增加了运维成本
- 2、因为对表的拆分，增加了后期维护的难度，同样也是增加了运维成本
- 3、即便做了大量的维护，但对于大数据的检索操作，依然很慢，完全达不到期望值

于是出现了 **lucene**，全文检索的工具。但是 **lucene** 对外暴露出的可用接口对于开发人员来说，操作是非常的复杂，而且没有效率的；于是在 **lucene** 的基础上进一步的封装，有了一个叫做 **solr** 的高性能分布式检索服务框架，但是，**solr** 有一个致命的缺点就是：在建立索引期间，**solr** 的搜索能力会极度下降，这就在一定程度上造成了 **solr** 在实时索引上效率并不高；

最后，出现了一个叫做 **elasticsearch** 的框架，同样是以 **lucene** 为基础，并且吸收了前两代的教训而开发出的分布式多用户能力的全文搜索引擎，并且 **elasticsearch** 是基于 **RESTful web** 接口进行发布的，那么这就意味着，我们开发人员操作起来更方便快捷；同时 **es** 拓展节点方便，可用于存储和检索海量数据，接近实时搜索能力，自动发现节点、副本机制保障可用性

2、非结构化数据查找方法

1：顺序扫描法(Serial Scanning)

所谓顺序扫描，比如要找内容包含某一个字符串的文件，就是一个文档一个文档的看，对于每一个文档，从头看到尾，如果此文档包含此字符串，则此文档为我们要找的文件，接着看下一个文件，直到扫描完所有的文件。如利用 **windows** 的搜索也可以搜索文件内容，只是相当的慢。

2：全文检索(Full-text Search)

将非结构化数据中的一部分信息提取出来，重新组织，使其变得有一定结构，然后对此有一定结构的数据进行搜索，从而达到搜索相对较快的目的。这部分**从非结构化数据中提取出的然后重新组织的信息，我们称之为索引。**

例如：字典。字典的拼音表和部首检字表就相当于字典的索引，对每一个字的解释是非结构化的，如果字典没有音节表和部首检字表，在茫茫辞海中找一个字只能顺序扫描。然而字的某些信息可以提取出来进行结构化处理，比如读音，就比较结构化，分声母和韵母，分别只有几种可以一一列举，于是将读音拿出来按一定的顺序排列，每一项读音都指向此字的详细解释的页数。我们搜索时按结构化的拼音搜到读音，然后按其指向的页数，便可找到我们的非结构化数据——也即对字的解释。

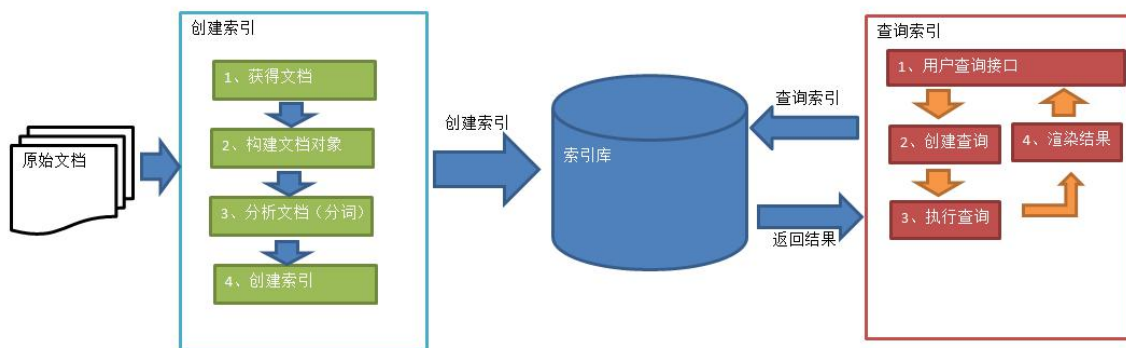
这种先建立索引，再对索引进行搜索的过程就叫全文检索(Full-text Search)。
虽然创建索引的过程也是非常耗时的，但是索引一旦创建就可以多次使用，全文检索主要处理的是查询，所以耗时间创建索引是值得的。

3、如何实现全文检索

可以使用 Lucene 实现全文检索。Lucene 是 apache 下的一个开放源代码的全文检索引擎工具包（提供了 Jar 包，实现全文检索的类库）。它提供了完整的查询引擎和索引引擎，部分文本分析引擎。Lucene 的目的是为软件开发人员提供一个简单易用的工具包，以方便地在目标系统中实现全文检索的功能。

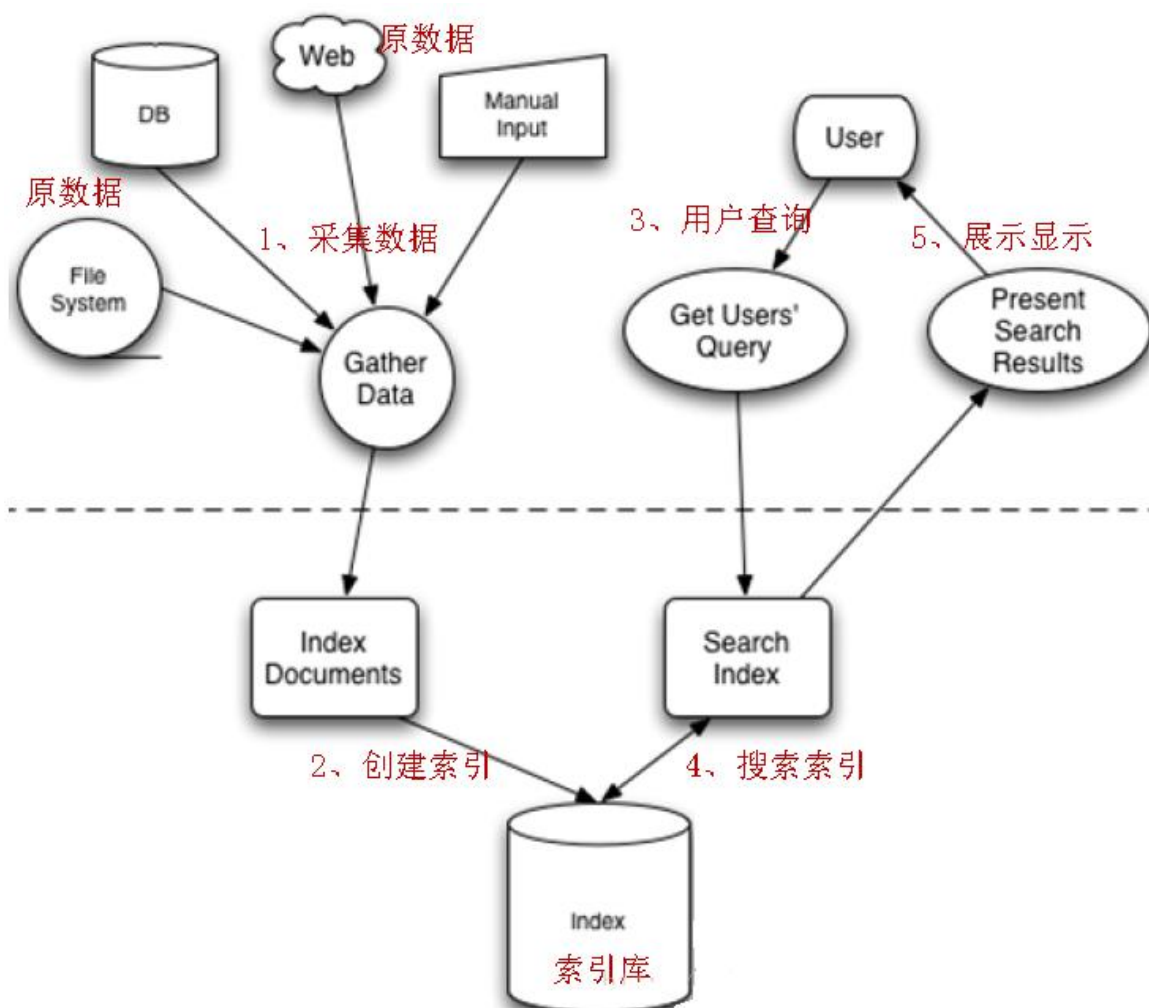
注意：Lucene 只是一个引擎，只是一个工具包，如果使用 Lucene 开发全文检索功能，要记住 **Lucene 是不能单独运行的**。

4、lucene 实现全文检索流程



1. 绿色表示索引过程，对要搜索的原始内容进行索引构建一个索引库，索引过程包括：确定原始内容即要搜索的内容→采集文档→创建文档→分析文档→索引文档。
2. 红色表示搜索过程，从索引库中搜索内容，搜索过程包括：用户通过搜索界面→创建查询→执行搜索，从索引库搜索→渲染搜索结果。

从上面了解到的知识点也可看出，索引和搜索流程图也可表示为：



总结：全文检索过程分为索引、搜索两个过程：

- 索引

1. 从关系数据库中、互联网上、文件系统采集源数据(要搜索的目标信息)，源数据的来源是很广泛的。
2. 将源数据采集到一个统一的地方，要创建**索引**，将索引创建到一个索引库（文件系统）中，从源数据库中提取关键信息，从关键信息中抽取一个**词**，词和源数据是有关联的。也即创建索引时，词和源数

据有关联，索引库中记录了这个关联，如果找到了词就说明找到了源数据（http 的网页、pdf 电子书等.....）。

- 搜索

1. 用户执行搜索（全文检索）编写查询关键字。
2. 从索引库中搜索索引，根据**查询关键字搜索索引库中的一个一个词**。
3. 展示搜索的结果。

5、全文检索框架介绍

市面上全文检索的框架很多，较早期的一个框架就是 lucene，基本上所有的全文检索的工作都交给 lucene 来实现，但是 lucene 最大的弊端就是 API 太原生，没有经过任何封装，不太好使用。所以后来出现一个叫做 solr 的框架，它也是基于 lucene 进行改造封装和包装，将服务端单独提取出来，客户端进行请求即可。

另外一个框架就是大名鼎鼎的 elasticsearch 了，es 也是一个基于 lucene 打造的全文检索的框架，且一经推出就迅速被市场认可，市场占有率越来越多，现在首选的全文检索的框架基本就是 ES 了。

3、ELK 日志协议栈

1、ELK 协议栈基本介绍

1、集中式日志系统

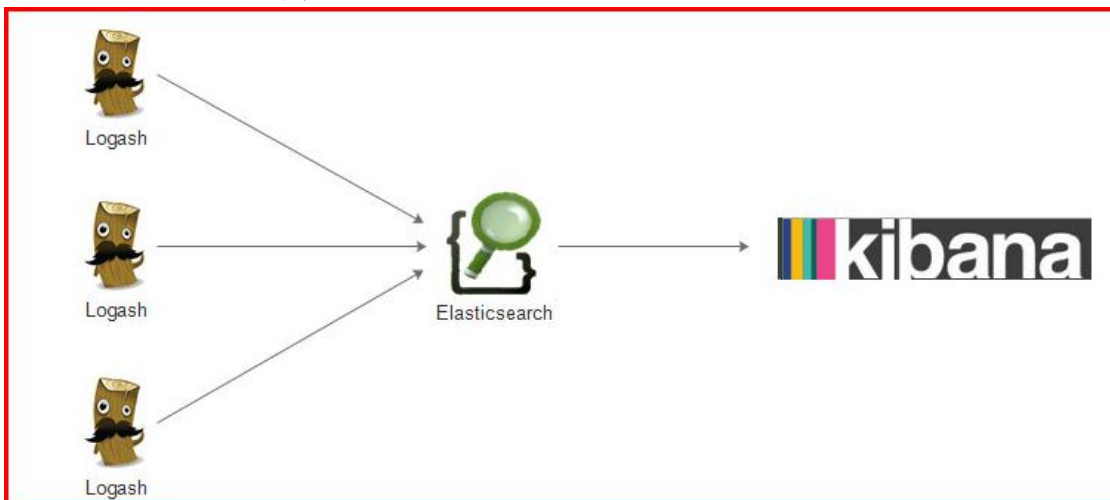
日志，对于任何系统来说都是及其重要的组成部分。在计算机系统里面，更是如此。但是由于现在的计算机系统大多比较复杂，很多系统都不是在一个地方，甚至都是跨国界的；即使是在一个地方的系统，也有不同的来源，比如，操作系统，应用服务，业务逻辑等等。他们都在不停产生各种各样的日志数据。根据不完全统计，我们全球每天大约要产生 2EB 的数据。

面对如此海量的数据，又是分布在各个不同地方，如果我们需要去查找一些重要的信息，难道还是使用传统的方法，去登陆到一台台机器上查看？看来传统的工具和方法已经显得非常笨拙和低效了。于是，一些聪明人就提出了建立一套集中式的方法，把不同来源的数据集中整合到一个地方。

一个完整的集中式日志系统，是离不开以下几个主要特点的。

- 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

2、ELK 协议栈介绍及体系结构



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，Logstash 和 Kibana。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 协议栈。

- Elasticsearch

Elasticsearch 是一个实时的分布式搜索和分析引擎，它可以用于全文搜索，结构化搜索以及分析。它是一个建立在全文搜索引擎 Apache Lucene 基础上的搜索引擎，使用 Java 语言编写。

主要特点

- 实时分析
- 分布式实时文件存储，并将**每一个字段**都编入索引

- 文档导向，所有的对象全部是文档
- 高可用性，易扩展，支持集群(Cluster)、分片和复制(Shards 和 Replicas)。见图 2 和图 3
- 接口友好，支持 JSON

- Logstash

Logstash 是一个具有实时渠道能力的数据收集引擎。使用 JRuby 语言编写。其作者是世界著名的运维工程师乔丹西塞 (JordanSissel)。

主要特点

- 几乎可以访问任何数据
- 可以和多种外部应用结合
- 支持弹性扩展

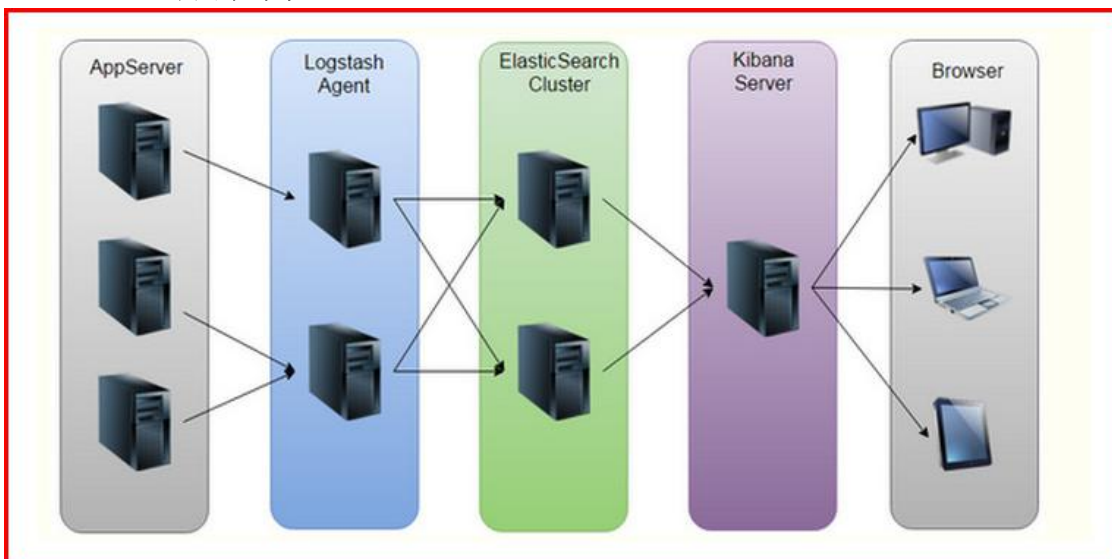
它由三个主要部分组成

- Shipper—发送日志数据
- Broker—收集数据，缺省内置 Redis
- Indexer—数据写入

- Kibana

Kibana 是一款基于 Apache 开源协议，使用 JavaScript 语言编写，为 Elasticsearch 提供分析和可视化的 Web 平台。它可以在 Elasticsearch 的索引中查找，交互数据，并生成各种维度的表图。

3、Elk 整体架构



4、参考文档

ELK 官网: <https://www.elastic.co/>

ELK 官网文档: <https://www.elastic.co/guide/index.html>

ELK 中文手册: <https://www.elastic.co/guide/cn/elasticsearch/guide/current/index.html>

ELK 中文社区: <https://elasticsearch.cn/>

4、Elasticsearch 介绍

1、什么是 Elasticsearch

Elasticsearch, 简称为 es, es 是一个开源的高扩展的分布式全文检索引擎, 它可以近乎实时的存储、检索数据; 本身扩展性很好, 可以扩展到上百台服务器, 处理 PB 级别的数据。es 也使用 Java 开发并使用 Lucene 作为其核心来实现所有索引和搜索的功能, 但是它的目的是通过简单的 RESTful API 来隐藏 Lucene 的复杂性, 从而让全文搜索变得简单。

北京市昌平区建材城西路金燕龙办公楼一层 电话: 400-618-9090

2、ElasticSearch 使用案例

- 2013 年初，GitHub 抛弃了 Solr，采取 ElasticSearch 来做 PB 级的搜索。“GitHub 使用 ElasticSearch 搜索 20TB 的数据，包括 13 亿文件和 1300 亿行代码”
- 维基百科：启动以 elasticsearch 为基础的核心搜索架构
- SoundCloud：“SoundCloud 使用 ElasticSearch 为 1.8 亿用户提供即时而精准的音乐搜索服务”
- 百度：百度目前广泛使用 ElasticSearch 作为文本数据分析，采集百度所有服务器上的各类指标数据及用户自定义数据，通过对各种数据进行多维分析展示，辅助定位分析实例异常或业务层面异常。目前覆盖百度内部 20 多个业务线（包括 casio、云分析、网盟、预测、文库、直达号、钱包、风控等），单集群最大 100 台机器，200 个 ES 节点，每天导入 30TB+数据
- 新浪使用 ES 分析处理 32 亿条实时日志
- 阿里使用 ES 构建挖财自己的日志采集和分析体系

3、ElasticSearch 对比 Solr

- Solr 利用 Zookeeper 进行分布式管理，而 Elasticsearch 自身带有分布式协调管理功能；
- Solr 支持更多格式的数据，而 Elasticsearch 仅支持 json 文件格式；
- Solr 官方提供的功能更多，而 Elasticsearch 本身更侧重于核心功能，高级功能多有第三方插件提供；
- Solr 在传统的搜索应用中表现好于 Elasticsearch，但在处理实时搜索应用时效率明显低于 Elasticsearch

4、ElasticSearch 架构图以及基本概念(术语)

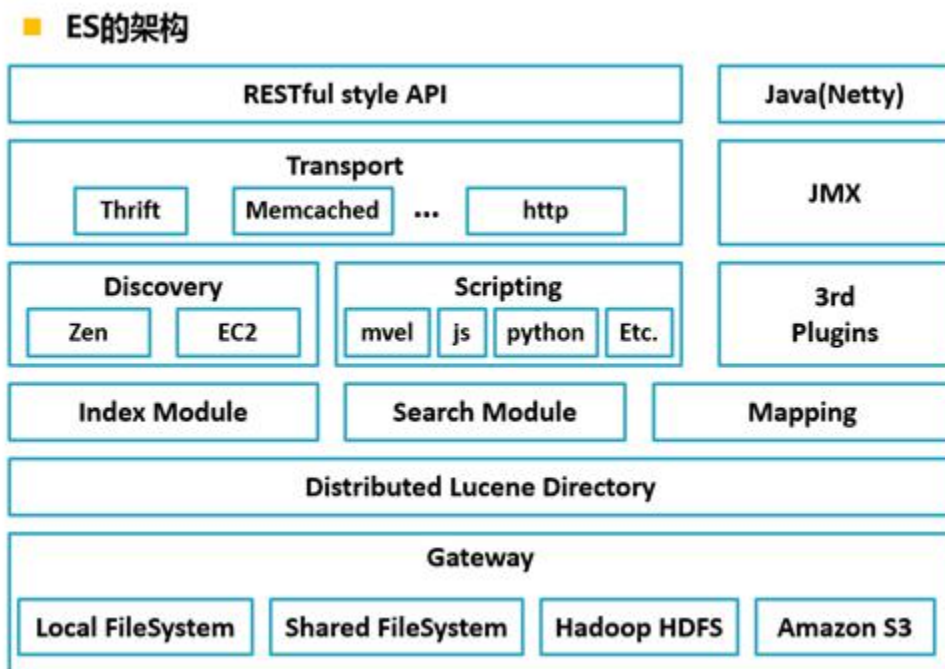
1、es 概述

Elasticsearch 是面向文档(document oriented)的，这意味着它可以存储整个对象或文档(document)。然而它不仅仅是存储，还会索引(index)每个文档的内容使之可以被搜索。在 Elasticsearch 中，你可以对文档（而非成行成列的数据）进行索引、搜索、排序、过滤。

Elasticsearch 比传统关系型数据库如下：

Relational DB -> Databases -> Tables -> Rows -> Columns
Elasticsearch -> Indices -> Types -> Documents -> Fields

2、ES 架构模块



Gateway 是 ES 用来存储索引的文件系统，支持多种类型。

Gateway 的上层是一个分布式的 lucene 框架。

Lucene 之上是 ES 的模块，包括：索引模块、搜索模块、映射解析模块等

ES 模块之上是 Discovery、Scripting 和第三方插件。

Discovery 是 ES 的节点发现模块，不同机器上的 ES 节点要组成集群需要进行消息通信，集群内部需要选举 master 节点，这些工作都是由 Discovery 模块完成。支持多种发现机制，如 Zen 、EC2、gce、Azure。

Scripting 用来支持在查询语句中插入 javascript、python 等脚本语言，scripting 模块负责解析这些脚本，使用脚本语句性能稍低。ES 也支持多种第三方插件。

再上层是 ES 的传输模块和 JMX.传输模块支持多种传输协议,如 Thrift、memecached、http，默认使用 http。JMX 是 java 的管理框架，用来管理 ES 应用。

最上层是 ES 提供给用户的接口，可以通过 RESTful 接口和 ES 集群进行交互。

3、Elasticsearch 核心概念

1、索引 index

一个索引就是一个拥有几分相似特征的文档的集合。比如说，你可以有一个客户数据的索引，另一个产品目录的索引，还有一个订单数据的索引。一个索引由一个名字来标识（必须全部是小写字母的），并且当我们要对对应于这个索引中的文档进行索引、搜索、更新和删除的时候，都要使用到这个名字。在一个集群中，可以定义任意多的索引。

2、类型 type

在一个索引中，你可以定义一种或多种类型。一个类型是你的索引的一个逻辑上的分类/分区，其语义完全由你来定。通常，会为具有一组共同字段的文档定义一个类型。比如说，我们假设你运营一个博客平台并且将你所有的数据存储到一个索引中。在这个索引中，你可以为用户数据定义一个类型，为博客数据定义另一个类型，当然，也可以为评论数据定义另一个类型。

3、字段 Field

相当于是数据表的字段，对文档数据根据不同属性进行的分类标识

4、映射 mapping

mapping 是处理数据的方式和规则方面做一些限制，如某个字段的数据类型、默认值、分析器、是否被索引等等，这些都是映射里面可以设置的，其它就是处理 es 里面数据的一些使用规则设置也叫做映射，按着最优规则处理数据对性能提高很大，因此才需要建立映射，并且需要思考如何建立映射才能对性能更好。

5、文档 document

一个文档是一个可被索引的基础信息单元。比如，你可以拥有某一个客户的文档，某一个产品的一个文档，当然，也可以拥有某个订单的一个文档。文档以 JSON

(Javascript Object Notation) 格式来表示，而 JSON 是一个到处存在的互联网数据交互格式。

在一个 index/type 里面，你可以存储任意多的文档。注意，尽管一个文档，物理上存在于一个索引之中，文档必须被索引/赋予一个索引的 type。

6、集群 cluster

一个集群就是由一个或多个节点组织在一起，它们共同持有整个的数据，并一起提供索引和搜索功能。一个集群由一个唯一的名字标识，这个名字默认就是“elasticsearch”。这个名字是重要的，因为一个节点只能通过指定某个集群的名字，来加入这个集群

7、节点 node

一个节点是集群中的一个服务器，作为集群的一部分，它存储数据，参与集群的索引和搜索功能。和集群类似，一个节点也是由一个名字来标识的，默认情况下，这个名字是一个随机的漫威漫画角色的名字，这个名字会在启动的时候赋予节点。这个名字对于管理工作来说挺重要的，因为在这个管理过程中，你会去确定网络中的哪些服务器对应于 Elasticsearch 集群中的哪些节点。

一个节点可以通过配置集群名称的方式来加入一个指定的集群。默认情况下，每个节点都会被安排加入到一个叫做“elasticsearch”的集群中，这意味着，如果你在你的网络中启动了若干个节点，并假定它们能够相互发现彼此，它们将会自动地形成并加入到一个叫做“elasticsearch”的集群中。

在一个集群里，只要你想，可以拥有任意多个节点。而且，如果当前你的网络中没有运行任何 Elasticsearch 节点，这时启动一个节点，会默认创建并加入一个叫做“elasticsearch”的集群。

8、分片和复制 shards&replicas

一个索引可以存储超出单个结点硬件限制的大量数据。比如，一个具有 10 亿文档的索引占据 1TB 的磁盘空间，而任一节点都没有这样大的磁盘空间；或者单个节点处理搜索请求，响应太慢。为了解决这个问题，Elasticsearch 提供了将索引划分成多份的能力，这些份就叫做分片。当你创建一个索引的时候，你可以指定你想要的分片的数量。每个分片本身也是一个功能完善并且独立的“索引”，这个“索引”可以被放置到集群中的任何节点上。分片很重要，主要有两方面的原因： 1) 允许你水

平分/扩展你的内容容量。 2) 允许你在分片（潜在地，位于多个节点上）之上进行分布式的、并行的操作，进而提高性能/吞吐量。

至于一个分片怎样分布，它的文档怎样聚合回搜索请求，是完全由 Elasticsearch 管理的，对于作为用户的你来说，这些都是透明的。

在一个网络/云的环境里，失败随时都可能发生，在某个分片/节点不知怎么的就处于离线状态，或者由于任何原因消失了，这种情况下，有一个故障转移机制是非常有用并且是强烈推荐的。为此目的，Elasticsearch 允许你创建分片的一份或多份拷贝，这些拷贝叫做复制分片，或者直接叫复制。

复制之所以重要，有两个主要原因：在分片/节点失败的情况下，提供了高可用性。因为这个原因，注意到复制分片从不与原/主要（original/primary）分片置于同一节点上是非常重要的。扩展你的搜索量/吞吐量，因为搜索可以在所有的复制上并行运行。总之，每个索引可以被分成多个分片。一个索引也可以被复制 0 次（意思是没有复制）或多次。一旦复制了，每个索引就有了主分片（作为复制源的原来的分片）和复制分片（主分片的拷贝）之别。分片和复制的数量可以在索引创建的时候指定。在索引创建之后，你可以在任何时候动态地改变复制的数量，但是你事后不能改变分片的数量。

默认情况下，Elasticsearch 中的每个索引被分片 5 个主分片和 1 个复制，这意味着，如果你的集群中至少有两个节点，你的索引将会有 5 个主分片和另外 5 个复制分片（1 个完全拷贝），这样的话每个索引总共就有 10 个分片。

5、es 的集群部署

第一步：创建普通用户

注意：ES 不能使用 root 用户来启动，必须使用普通用户来安装启动。这里我们创建一个普通用户以及定义一些常规目录用于存放我们的数据文件以及安装包等

创建一个 es 专门的用户（**必须**）

#使用 root 用户在三台机器执行以下命令

```
useradd es
mkdir -p /export/servers/es
chown -R es:es /export/servers/es
passwd es
```

第二步：为普通用户 es 添加 sudo 权限

为了让普通用户有更大的操作权限，我们一般都会给普通用户设置 sudo 权限，方便普通用户的操作

三台机器使用 root 用户执行 visudo 命令然后为 es 用户添加权限

```
visudo
```

es	ALL=(ALL)	ALL
----	-----------	-----

第三步：下载并上传压缩包，然后解压

将 es 的安装包下载并上传到 node01 服务器的/home/es 路径下，然后进行解压

使用 es 用户来执行以下操作（断开连接 linux 的工具，然后重新使用 es 用户连接上三台 linux 服务器）

node01 服务器使用 es 用户执行以下命令

```
cd /home/es/
wget
https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-6.7.0.
tar.gz
tar -zxvf elasticsearch-6.7.0.tar.gz -C /export/servers/es/
```

第四步：修改配置文件

修改 elasticsearch.yml

node01 服务器使用 es 用户来修改配置文件

```
cd /export/servers/es/elasticsearch-6.7.0/config
mkdir -p /export/servers/es/elasticsearch-6.7.0/logs
```

```
mkdir -p /export/servers/es/elasticsearch-6.7.0/datas
rm -rf elasticsearch.yml
vim elasticsearch.yml
```

```
cluster.name: myes
node.name: node01
path.data: /export/servers/es/elasticsearch-6.7.0/datas
path.logs: /export/servers/es/elasticsearch-6.7.0/logs
network.host: 192.168.227.10
http.port: 9200
discovery.zen.ping.unicast.hosts: ["node01", "node02", "node03"]
bootstrap.system_call_filter: false
bootstrap.memory_lock: false
http.cors.enabled: true
http.cors.allow-origin: "*"
```

修改 jvm.option

修改 jvm.option 配置文件，调整 jvm 堆内存大小

node01 使用 **es** 用户执行以下命令调整 jvm 堆内存大小，每个人根据自己服务器的内存大小来进行调整

```
cd /export/servers/es/elasticsearch-6.7.0/config
vim jvm.options
```

```
-Xms2g
-Xmx2g
```


第五步：将安装包分发到其他服务器上面

node01 使用 es 用户将安装包分发到其他服务器上面去

```
cd /export/servers/es/  
scp -r elasticsearch-6.7.0/ node02:$PWD  
scp -r elasticsearch-6.7.0/ node03:$PWD
```

第六步：node02 与 node03 修改 es 配置文件

node02 与 node03 也需要修改 es 配置文件

node02 使用 es 用户执行以下命令修改 es 配置文件

```
cd /export/servers/es/elasticsearch-6.7.0/config  
vim elasticsearch.yml
```

```
cluster.name: myes  
node.name: node02  
path.data: /export/servers/es/elasticsearch-6.7.0/datas  
path.logs: /export/servers/es/elasticsearch-6.7.0/logs  
network.host: 192.168.227.20  
http.port: 9200  
discovery.zen.ping.unicast.hosts: ["node01", "node02", "node03"]  
bootstrap.system_call_filter: false  
bootstrap.memory_lock: false  
http.cors.enabled: true  
http.cors.allow-origin: ""
```

node03 使用 es 用户执行以下命令修改配置文件

```
cd /export/servers/es/elasticsearch-6.0.0/config/  
vim elasticsearch.yml
```

```
cluster.name: myes
```

```
node.name: node03
path.data: /export/servers/es/elasticsearch-6.7.0/datas
path.logs: /export/servers/es/elasticsearch-6.7.0/logs
network.host: 192.168.227.30
http.port: 9200
discovery.zen.ping.unicast.hosts: ["node01", "node02", "node03"]
bootstrap.system_call_filter: false
bootstrap.memory_lock: false
http.cors.enabled: true
http.cors.allow-origin: "*"
```

第七步：修改系统配置，解决启动时候的问题

由于现在使用普通用户来安装 es 服务，且 es 服务对服务器的资源要求比较多，包括内存大小，线程数等。所以我们需要给普通用户解开资源的束缚

解决启动问题一：普通用户打开文件的最大数限制

问题错误信息描述：

```
max file descriptors [4096] for elasticsearch process likely too low,
increase to at least [65536]
```

ES 因为需要大量的创建索引文件，需要大量的打开系统的文件，所以我们需要解除 linux 系统当中打开文件最大数目的限制，不然 ES 启动就会抛错

三台机器使用 es 用户执行以下命令解除打开文件数据的限制

```
sudo vi /etc/security/limits.conf
```

添加如下内容：注意*不要去掉了

```
* soft nfile 65536
* hard nfile 131072
* soft nproc 2048
* hard nproc 4096
```

解决启动问题二：普通用户启动线程数限制

问题错误信息描述

max number of threads [1024] for user [es] likely too low, increase to at least [4096]

修改普通用户可以创建的最大线程数

max number of threads [1024] for user [es] likely too low, increase to at least [4096]

原因：无法创建本地线程问题,用户最大可创建线程数太小

解决方案：修改 90-nproc.conf 配置文件。

三台机器使用 es 用户执行以下命令修改配置文件

```
sudo vi /etc/security/limits.d/90-nproc.conf
```

找到如下内容：

```
* soft nproc 1024
#修改为
* soft nproc 4096
```

解决启动问题三：普通用户调大虚拟内存

错误信息描述：

max virtual memory areas vm.max_map_count [65530] likely too low, increase to at least [262144]

调大系统的虚拟内存

原因：最大虚拟内存太小

每次启动机器都手动执行下。

三台机器执行以下命令，注意每次启动 ES 之前都要执行

```
sudo sysctl -w vm.max_map_count=262144
```

备注：以上三个问题解决完成之后，重新连接 secureCRT 或者重新连接 xshell 生效需要保存、退出、重新登录 xshell 才可生效。

第八步：启动 ES 服务

三台机器使用 es 用户执行以下命令启动 es 服务

```
nohup /export/servers/es/elasticsearch-6.7.0/bin/elasticsearch >/dev/null  
&
```

启动成功之后 jsp 即可看到 es 的服务进程，并且访问页面

```
http://node01:9200/?pretty
```

能够看到 es 启动之后的一些信息

注意：如果哪一台机器服务启动失败，那么就到哪一台机器的

```
/export/servers/es/elasticsearch-6.7.0/logs
```

这个路径下面去查看错误日志

6、node01 服务器安装 elasticsearch-head 插件

由于 es 服务启动之后，访问界面比较丑陋，为了更好的查看索引库当中的信息，我们可以通过安装 elasticsearch-head 这个插件来实现，这个插件可以更方便快捷的看到 es 的管理界面

1 、node01 机器安装 nodejs

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。

Node.js 是一个 Javascript 运行环境(runtime environment)，发布于 2009 年 5 月，由 Ryan Dahl 开发，实质是对 Chrome V8 引擎进行了封装。Node.js 不是一个 JavaScript 框架，不同于 CakePHP、Django、Rails。Node.js 更不是浏览器端的库，不能与 jQuery、

ExtJS 相提并论。Node.js 是一个让 JavaScript 运行在服务端的开发平台，它让 JavaScript 成为与 PHP、Python、Perl、Ruby 等服务端语言平起平坐的脚本语言。

安装步骤参考：<https://www.cnblogs.com/kevingrace/p/8990169.html>

第一步：下载安装包

node01 机器执行以下命令下载安装包，然后进行解压

```
cd /home/es  
tar -zxvf node-v8.1.0-linux-x64.tar.gz -C /export/servers/es/
```

第二步：创建软连接

node01 执行以下命令创建软连接

```
sudo ln -s  
/export/servers/es/node-v8.1.0-linux-x64/lib/node_modules/npm/bin/npm-cl  
i.js /usr/local/bin/npm  
  
sudo ln -s /export/servers/es/node-v8.1.0-linux-x64/bin/node  
/usr/local/bin/node
```

第三步：修改环境变量

node01 服务器添加环境变量

```
sudo vim /etc/profile
```

```
export NODE_HOME=/export/servers/es/node-v8.1.0-linux-x64  
export PATH=$NODE_HOME/bin:$PATH
```

修改完环境变量使用 source 生效

```
source /etc/profile
```

第四步：验证安装成功

node01 执行以下命令验证安装生效

```
node -v  
npm -v
```

```
[root@node01 softwares]# node -v
v8.1.0
[root@node01 softwares]# npm -v
5.0.3
[root@node01 softwares]#
```

2 、node01 机器安装 elasticsearch-head 插件

elasticsearch-head 这个插件是 es 提供的一个用于图形化界面查看的一个插件工具，可以安装上这个插件之后，通过这个插件来实现我们通过浏览器查看 es 当中的数据

安装 elasticsearch-head 这个插件这里提供两种方式进行安装，第一种方式就是自己下载源码包进行编译，耗时比较长，网络较差的情况下，基本上不可能安装成功

第二种方式就是直接使用我已经编译好的安装包，进行修改配置即可

1、第一种方式：在线安装 elasticsearch-head 插件（网速慢，不推荐）

这里选择 node01 进行安装

第一步：在线安装必须依赖包

```
# 初始化目录
cd /export/servers/es
# 安装GCC
sudo yum install -y gcc-c++ make git
```

第二步：从 git 上面克隆编译包并进行安装

```
cd /export/servers/es
git clone https://github.com/mobz/elasticsearch-head.git
# 进入安装目录
cd /export/servers/es/elasticsearch-head
# intall 才会有 node-modules
npm install
```

```
[es@node02 elasticsearch-head]$ npm install
npm WARN deprecated coffee-script@1.10.0: CoffeeScript on NPM has moved to "coffeescript" (no hyphen)
npm WARN deprecated http2@3.3.7: Use the built-in module in node 9.0.0 or newer, instead
npm WARN prefer global coffee-script@1.10.0 should be installed with -g

> phantomjs-prebuilt@2.1.16 install /export/servers/es/elasticsearch-head/node_modules/phantomjs-prebuilt
> node install.js

PhantomJS not found on PATH
Downloading https://github.com/Medium/phantomjs/releases/download/v2.1.1/phantomjs-2.1.1-linux-x86_64.tar.bz2
Saving to /tmp/phantomjs/phantomjs-2.1.1-linux-x86_64.tar.bz2
Receiving...
[=====] 4%
□ 发送文本到当前Xshell窗口的全部会话
```

以下进度信息

npm WARN notice [SECURITY] lodash has the following vulnerability: 1 low.
Go here for more details:

npm WARN notice [SECURITY] debug has the following vulnerability: 1 low.
Go here for more details:

<https://nodesecurity.io/advisories?search=debug&version=0.7.4> - Run
`npm i npm@latest -g` to upgrade your npm version, and then `npm audit`
to get more info.

npm ERR! Unexpected end of input at 1:2096

npm ERR!
7c1a1bc21c976bb49f3ea", "tarball": "https://registry.npmjs.org/safer-bu

npm ERR!
^

npm ERR! A complete log of this run can be found in:

npm ERR!
/home/es/.npm/_logs/2018-11-27T14_35_39_453Z-debug.log

以上错误可以不用管。

第三步、node01 机器修改 Gruntfile.js

第一台机器修改 Gruntfile.js 这个文件

```
cd /export/servers/es/elasticsearch-head
vim Gruntfile.js
```

找到以下代码：

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090

添加一行: hostname: '192.168.52.100',

```
connect: {
    server: {
        options: {
            hostname: '192.168.52.100',
            port: 9100,
            base: '.',
            keepalive: true
        }
    }
}
```

第四步、node01 机器修改 app.js

第一台机器修改 app.js

```
cd /export/servers/es/elasticsearch-head/_site
```

```
vim app.js
```

```
(function( app, i18n ) {
    var ui = app.ns("ui");
    var services = app.ns("services");

    app.App = ui.AbstractWidget.extend({
        defaults: {
            base_uri: null
        },
        init: function(parent) {
            this._super();
            this.prefs = services.Preferences.instance();
            this.base_uri = this.config.base_uri || this.prefs.get("app-base_uri") || "http://node01:9200";
            if( this.base_uri.charAt( this.base_uri.length - 1 ) !== "/" ) {
                // XHR request fails if the URL is not ending with a "/"
                this.base_uri += "/";
            }
            if( this.config.auth_user ) {
                var credentials = window.btoa( this.config.auth_user + ":" + this.config.auth_password );
                $.ajaxSetup({
                    headers: {
                        "Authorization": "Basic " + credentials
                    }
                });
            }
            this.cluster = new services.Cluster({ base_uri: this.base_uri });
        }
    });
})( app, i18n );
```

更改前: http://localhost:9200

更改后: http://node01:9200

2、第二种方式：直接使用提供的编译之后的源码包解压之后修改配置文件即可（强烈推荐）

第一步：上传压缩包到/home/es 路径下去

将我们的压缩包 `elasticsearch-head-compile-after.tar.gz` 上传到 node01 机器的 `/home/es` 路径下面去

第二步：解压安装包

node01 执行以下命令解压安装包

```
cd /home/es/  
tar -zxvf elasticsearch-head-compile-after.tar.gz -C /export/servers/es/
```

第三步、node01 机器修改 Gruntfile.js

修改 `Gruntfile.js` 这个文件

```
cd /export/servers/es/elasticsearch-head  
vim Gruntfile.js
```

找到以下代码：

添加一行： `hostname: '192.168.227.10'`,

```
connect: {  
    server: {  
        options: {  
            hostname: '192.168.227.10',  
            port: 9100,  
            base: '.',  
            keepalive: true  
        }  
    }  
}
```

第四步、node01 机器修改 app.js

第一台机器修改 app.js

```
cd /export/servers/es/elasticsearch-head/_site
```

```
vim app.js
```

```
(function( app, i18n ) {  
    var ui = app.ns("ui");  
    var services = app.ns("services");  
    app.App = ui.AbstractWidget.extend({  
        defaults: {  
            base_uri: null  
        },  
        init: function(parent) {  
            this._super();  
            this.prefs = services.Preferences.instance();  
            this.base_uri = this.config.base_uri || this.prefs.get("app-base_uri") || "http://node01:9200";  
            if( this.base_uri.charAt( this.base_uri.length - 1 ) !== "/" ) {  
                // XHR request fails if the URL is not ending with a "/"  
                this.base_uri += "/";  
            }  
            if( this.config.auth_user ) {  
                var credentials = window.btoa( this.config.auth_user + ":" + this.config.auth_password );  
                $.ajaxSetup({  
                    headers: {  
                        "Authorization": "Basic " + credentials  
                    }  
                });  
            }  
            this.cluster = new services.Cluster({ base_uri: this.base_uri });  
        }  
    });  
})
```

更改前: http://localhost:9200

更改后: http://node01:9200

3、node01 机器启动 head 服务

node01 启动 elasticsearch-head 插件

```
cd /export/servers/es/elasticsearch-head/node_modules/grunt/bin/
```

进程前台启动命令

```
./grunt server
```

进程后台启动命令

```
nohup ./grunt server >/dev/null 2>&1 &
```

```
Running "connect:server" (connect) task  
Waiting forever...  
Started connect web server on http://192.168.52.100:9100
```

如何停止: elasticsearch-head 进程

执行以下命令找到 elasticsearch-head 的插件进程，然后使用 kill -9 杀死进程即可

```
netstat -nltp | grep 9100
```

```
kill -9 8328
```

```
[es@node01 bin]$ netstat -nltp | grep 9100
(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      0 192.168.52.100:9100  0.0.0.0:*        LISTEN      8328/grunt
[es@node01 bin]$ kill -9 8328
```

4、访问 elasticsearch-head 界面

打开 Google Chrome 访问

```
http://192.168.100.100:9100/
```

7、node01 服务器安装 Kibana

kibana 的基本介绍

Kibana 是一个开源的分析和可视化平台，设计用于和 Elasticsearch 一起工作。你用 Kibana 来搜索，查看，并和存储在 Elasticsearch 索引中的数据进行交互。你可以轻松地执行高级数据分析，并且以各种图标、表格和地图的形式可视化数据。Kibana 使得理解大量数据变得很容易。它简单的、基于浏览器的界面使你能够快速创建和共享动态仪表板，实时显示 Elasticsearch 查询的变化。

接着使用我们的 es 用户在 node01 服务器上面来实现我们的 kibana 的安装部署

第一步：下载资源上传服务器并解压

node01 服务器使用 es 用户执行以下命令来下载安装包并解压

```
cd /home/es
```

在线下载

```
tar -zxvf kibana-6.7.0-linux-x86_64.tar.gz -C /export/servers/es/
```

第二步：修改配置文件

node01 服务器使用 es 用户执行以下命令来修改配置文件

```
cd /export/servers/es/kibana-6.7.0-linux-x86_64/config/  
vi kibana.yml
```

配置内容如下：

```
server.host: "node01"  
elasticsearch.hosts: ["http://node01:9200"]
```

第三步：启动服务

node01 服务器使用 es 用户执行以下命令启动 kibana 服务

```
cd /export/servers/es/kibana-6.7.0-linux-x86_64  
nohup bin/kibana >/dev/null 2>&1 &
```

如何停止 kibana 进程：停止 kibana 服务进程

查看进程号

```
ps -ef | grep node
```

然后使用 kill -9 杀死进程即可

第四步：浏览器访问

浏览器地址访问 kibana 服务

```
http://node01:5601
```

8、使用 kibana 管理索引

curl 是利用 URL 语法在命令行方式下工作的开源文件传输工具，使用 curl 可以实现常见的 get/post 请求。简单的认为是可以在命令行下面访问 url 的一个工具。在 centos 的默认库里面是有 curl 工具的，如果没有请 yum 安装即可。

curl

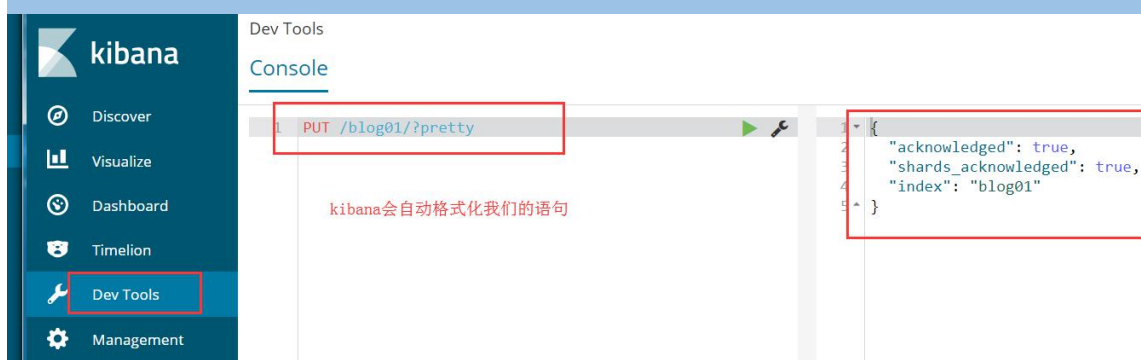
-X 指定 http 的请求方法 有 HEAD GET POST PUT DELETE
-d 指定要传输的数据
-H 指定 http 请求头信息

1、使用 Xput 创建索引

1、创建索引

在我们的 kibana 的 dev tools 当中执行以下语句

```
curl -XPUT http://node01:9200/blog01/?pretty
```



2、插入文档

前面的命令使用 PUT 动词将一个文档添加到 /article(文档类型)，并为该文档分配 ID 为 1。URL 路径显示为 index/doctype/ID（索引/文档类型/ID）。

```
curl -XPUT http://node01:9200/blog01/article/1?pretty -d '{"id": "1", "title": "What is Lucene"}'
```

问题：Content-Type header [application/x-www-form-urlencoded] is not supported

解决：

```
curl -XPUT http://node01:9200/blog01/article/1?pretty -d '{"id": "1", "title": "What is Lucene"}' -H "Content-Type: application/json"
```

原因：

此原因时由于 ES 增加了安全机制， 进行严格的内容类型检查， 严格检查内容类型也可以作为防止跨站点请求伪造攻击的一层保护。 [官网解释](#)

http.content_type.required

3、查询文档

```
curl -XGET http://node01:9200/blog01/article/1?pretty
```

问题: Content-Type header [application/x-www-form-urlencoded] is not supported

解决:

```
curl -XPUT http://node01:9200/blog01/article/1?pretty -d '{"id": "1", "title": "What is lucene"}' -H "Content-Type: application/json"
```

```
curl -XGET http://node01:9200/blog01/article/1?pretty -H "Content-Type: application/json"
```

4、更新文档

```
curl -XPUT http://node01:9200/blog01/article/1?pretty -d '{"id": "1", "title": " What is elasticsearch"}'
```

问题: Content-Type header [application/x-www-form-urlencoded] is not supported

解决:

```
curl -XPUT http://node01:9200/blog01/article/1?pretty -d '{"id": "1", "title": " What is elasticsearch"}' -H "Content-Type: application/json"
```

5、搜索文档

```
curl -XGET "http://node01:9200/blog01/article/_search?q=title:elasticsearch"
```

问题: Content-Type header [application/x-www-form-urlencoded] is not supported

解决:

```
curl -XGET "http://node01:9200/blog01/article/_search?q=title:'elasticsearch'&pretty" -H "Content-Type: application/json"
```

6、删除文档

```
curl -XDELETE "http://node01:9200/blog01/article/1?pretty"
```

7、删除索引

```
curl -XDELETE http://node01:9200/blog01?pretty
```


2、返回值说明

1、Hits

返回结果中最重要的部分是 **hits**，它包含 **total** 字段来表示匹配到的文档总数，并且一个 **hits** 数组包含所查询结果的前十个文档。

在 **hits** 数组中每个结果包含文档的 **_index**、**_type**、**_id**，加上 **_source** 字段。这意味着我们可以直接从返回的搜索结果中使用整个文档。这不像其他的搜索引擎，仅仅返回文档的 **ID**，需要你单独去获取文档。

每个结果还有一个 **_score**，它衡量了文档与查询的匹配程度。默认情况下，首先返回最相关的文档结果，就是说，返回的文档是按照 **_score** 降序排列的。在这个例子中，我们没有指定任何查询，故所有的文档具有相同的相关性，因此对所有的结果而言 **1** 是中性的 **_score**。

max_score 值是与查询所匹配文档的 **_score** 的最大值。

2、took

took 值告诉我们执行整个搜索请求耗费了多少毫秒

3、Shard

_shards 部分 告诉我们在查询中参与分片的总数，以及这些分片成功了多少个失败了多少个。正常情况下我们不希望分片失败，但是分片失败是可能发生的。

如果我们遭遇到一种灾难级别的故障，在这个故障中丢失了相同分片的原始数据和副本，那么对这个分片将没有可用副本来对搜索请求作出响应。假若这样，**Elasticsearch** 将报告这个分片是失败的，但是会继续返回剩余分片的结果。

4、timeout

timed_out 值告诉我们查询是否超时。默认情况下，搜索请求不会超时。如果低响应时间比完成结果更重要，你可以指定 **timeout** 为 **10** 或者 **10ms**（10 毫秒），或者 **1s**（1 秒）：

GET /_search?timeout=10ms

在请求超时之前，**Elasticsearch** 将会返回已经成功从每个分片获取的结果。

3、花式查询

在 kibana 提供的界面上进行操作。

```
POST /school/student/_bulk
{ "index": { "_id": 1 } }
{ "name": "liubei", "age": 20, "sex": "boy", "birth": "1996-01-02",
  "about": "i like diaocan he girl" }
{ "index": { "_id": 2 } }
{ "name": "guanyu", "age": 21, "sex": "boy", "birth": "1995-01-02",
  "about": "i like diaocan" }
```

```
{ "index": { "_id": 3 }}
{ "name" : "zhangfei", "age" : 18 , "sex": "boy", "birth":
"1998-01-02" , "about": "i like travel" }
{ "index": { "_id": 4 }}
{ "name" : "diaocan", "age" : 20 , "sex": "girl", "birth":
"1996-01-02" , "about": "i like travel and sport" }
{ "index": { "_id": 5 }}
{ "name" : "panjinlian", "age" : 25 , "sex": "girl", "birth":
"1991-01-02" , "about": "i like travel and wusong" }
{ "index": { "_id": 6 }}
{ "name" : "caocao", "age" : 30 , "sex": "boy", "birth": "1988-01-02" ,
"about": "i like xiaoqiao" }
{ "index": { "_id": 7 }}
{ "name" : "zhaoyun", "age" : 31 , "sex": "boy", "birth":
"1997-01-02" , "about": "i like travel and music" }
{ "index": { "_id": 8 }}
{ "name" : "xiaoqiao", "age" : 18 , "sex": "girl", "birth":
"1998-01-02" , "about": "i like caocao" }
{ "index": { "_id": 9 }}
{ "name" : "daqiao", "age" : 20 , "sex": "girl", "birth":
"1996-01-02" , "about": "i like travel and history" }
```

1、使用 match_all 做查询

```
GET /school/student/_search?pretty
{
  "query": {
    "match_all": {}
  }
}
```

问题：通过 match_all 匹配后，会把所有的数据检索出来，但是往往真正的业务需求并非要找全部的数据，而是检索出自己想要的；并且对于 es 集群来说，直接检索全部的数据，很容易造成 GC 现象。所以，我们要学会如何进行高效的检索数据

2、通过关键字段进行查询

```
GET /school/student/_search?pretty
{
  "query": {
    "match": {"about": "travel"}
  }
}
```

如果此时想查询喜欢旅游的，并且不能是男孩的，怎么办？

【这种方式是错误的，因为一个 match 下，不能出现多个字段值[match] query doesn't support multiple fields】，需要使用复合查询

```
1 GET /school/student/_search?pretty
2 {
3   "query": {
4     "match": {
5       "about": "travel",
6       "sex": "girl"
7     }
8   }
9 }

1 {
2   "error": {
3     "root_cause": [
4       {
5         "type": "parsing_exception",
6         "reason": "[match] query doesn't support multiple fields, found [about] and [sex]",
7         "line": 5,
8         "col": 19
9       }
10    ],
11    "type": "parsing_exception",
12    "reason": "[match] query doesn't support multiple fields, found [about] and [sex]",
13    "line": 5,
14    "col": 19
15  },
16  "status": 400
17 }
```

3、bool 的复合查询

当出现多个查询语句组合的时候，可以用 bool 来包含。bool 合并聚包含：must，must_not 或者 should， should 表示 or 的意思

例子：查询非男性中喜欢旅行的人

```
GET /school/student/_search?pretty
{
  "query": {
    "bool": {
      "must": { "match": {"about": "travel"}},
      "must_not": {"match": {"sex": "boy"}}
    }
  }
}
```

4、bool 的复合查询中的 should

should 表示可有可无的（如果 should 匹配到了就展示，否则就不展示）

例子：

查询喜欢旅行的，如果有男性的则显示，否则不显示

```
GET /school/student/_search?pretty
{
  "query": {
    "bool": {
      "must": { "match": {"about": "travel"}},
      "should": {"match": {"sex": "boy"}}
    }
  }
}
```

5、term 匹配 词条查询

使用 term 进行精确匹配（比如数字，日期，布尔值或 not_analyzed 的字符串(未经分析的文本数据类型)）

语法

```
{ "term": { "age": 20 } }
```

```
{ "term": { "date": "2018-04-01" } }
```

```
{ "term": { "sex": "boy" } }
```

```
{ "term": { "about": "trivel" } }
```

例子：

查询喜欢旅行的

```
GET /school/student/_search?pretty
{
  "query": {
    "bool": {
      "must": { "term": { "about": "travel" } },
      "should": { "term": { "sex": "boy" } }
    }
  }
}
```

6、使用 terms 匹配多个值

```
GET /school/student/_search?pretty
{
  "query": {
    "bool": {
      "must": { "terms": { "about": ["travel", "history"] } }
    }
  }
}
```

term 主要是用于精确的过滤比如说：“我爱你”

在 match 下面匹配可以为包含：我、爱、你、我爱等等的解析器

在 term 语法下面就精准匹配到：“我爱你”

7、Range 过滤

Range 过滤允许我们按照指定的范围查找一些数据：操作范围：gt::大于，gae::大于等于,lt::小于，lte::小于等于

例子：

查找出大于 20 岁，小于等于 25 岁的学生

```
GET /school/student/_search?pretty
{
  "query": {
    "range": {
      "age": {"gt":20,"lte":25}
    }
  }
}
```

8、exists 和 missing 过滤

exists 和 missing 过滤可以找到文档中是否包含某个字段或者是没有某个字段

例子：

查找字段中包含 age 的文档

```
GET /school/student/_search?pretty
{
  "query": {
    "exists": {
      "field": "age"
    }
  }
}
```

9、bool 的多条件过滤

用 bool 也可以像之前 match 一样来过滤多行条件：

must :: 多个查询条件的完全匹配,相当于 and 。

must_not :: 多个查询条件的相反匹配，相当于 not 。

should :: 至少有一个查询条件匹配，相当于 or

例子：

过滤出 about 字段包含 travel 并且年龄大于 20 岁小于 30 岁的同学

```
GET /school/student/_search?pretty
{
  "query": {
    "bool": {
      "must": [
        { "term": {
          "about": {
            "value": "travel"
          }
        } },
        { "range": {
          "age": {
            "gte": 20,
            "lte": 30
          }
        } }
      ]
    }
  }
}
```

10、查询与过滤条件合并

通常复杂的查询语句，我们也要配合过滤语句来实现缓存，用 **filter** 语句就可以来实现

例子：

查询出喜欢旅行的，并且年龄是 20 岁的文档

```
GET /school/student/_search?pretty
{
  "query": {
    "bool": {
      "must": { "match": { "about": "travel" } },
      "filter": [ { "term": { "age": 20 } } ]
    }
  }
}
```

4、索引映射（mappings）管理

1、为什么要映射

elasticsearch 中的文档等价于 java 中的对象，那么在 java 对象中有字段（比如 string、int、long 等），同理在 elasticsearch 索引中的具体字段也是有类型的。

```
PUT /document/article/1
{
  "title" : "elasticsearchshi 是什么",
  "author" : "zhangsan",
  "titleScore" : 60
}
```

这种操作并没有指定字段类型，那么 elasticsearch 会自动根据数据类型的格式识别字段的类型；查看索引字段类型：GET /document/article/_mapping。可以发现 titleScore 的类型是 long。

The screenshot shows an Elasticsearch client interface. On the left, a PUT request is entered: `PUT /document/article/1` with a JSON body: `{ "title": "elasticsearchshi是什么", "author": "zhangsan", "titleScore": 60 }`. Below this, the GET request `GET /document/article/_mapping` is entered. On the right, the resulting mapping is displayed in a tree view. The mapping for the `article` type includes properties for `author` (text) and `title` (text). The `titleScore` property is highlighted with a red box, showing its type as `long`.

```
1 {
2   "document": {
3     "mappings": {
4       "article": {
5         "properties": {
6           "author": {
7             "type": "text",
8             "fields": {
9               "keyword": {
10                "type": "keyword",
11                "ignore_above": 256
12              }
13            }
14          },
15          "title": {
16            "type": "text",
17            "fields": {
18              "keyword": {
19                "type": "keyword",
20                "ignore_above": 256
21              }
22            }
23          },
24          "titleScore": {
25            "type": "long"
26          }
27        }
28      }
29    }
30  }
31 }
```

然后在插入一条数据：

```
PUT /document/article/2
{
```



```
"title" : "elasticsearch 是什么",  
"author" : "zhangsan",  
"titleScore" : 66.666  
}
```

查询数据：GET /document/article/2

我们会发现 es 能存入，并没有报错（注意），这其实是一个问题，因为如果后期 elasticsearch 对接 java 的时候，我们会写一个类对数据做封装，比如：

```
class Article{  
private String title;  
private String author;  
private String titleScore // 《什么类型合适》？如果使用 Long 类型，那么后面肯定会有数据格式转换的异常 doubleLong  
}
```

所以，我们如果能提前知道字段类型，那么最好使用 mapping 的映射管理，提前指定字段的类型，防止后续的程序问题；

```
DELETE document  
PUT document  
{  
  "mappings": {  
    "article" : {  
      "properties": {  
        {  
          "title" : {"type": "text"} ,  
          "author" : {"type": "text"} ,  
          "titleScore" : {"type": "double"}  
        }  
      }  
    }  
  }  
}  
get document/article/_mapping
```

2、基本命令

```
DELETE school  
PUT school  
{  
  "mappings": {  
    "logs" : {  
      "properties": {"messages" : {"type": "text"}}  
    }  
  }  
}
```

```
}
```

添加索引：school，文档类型类 logs，索引字段为 message，字段的类型为 text

GET /school/_mapping/logs



```
PUT school
{
  "mappings": {
    "logs": {
      "properties": {"messages": {"type": "text"}}
    }
  }
}

GET /school/_mapping/logs
{
  "school": {
    "mappings": {
      "logs": {
        "properties": {
          "messages": {
            "type": "text"
          }
        }
      }
    }
  }
}
```

继续添加字段

```
POST /school/_mapping/logs
{
  "properties": {"number": {"type": "text"}}
}

GET /school/_mapping/logs
```



```
PUT school
{
  "mappings": {
    "logs": {
      "properties": {"messages": {"type": "text"}}
    }
  }
}

GET /school/_mapping/logs
POST /school/_mapping/logs
{
  "properties": {"number": {"type": "text"}}
}

GET /school/_mapping/logs
{
  "school": {
    "mappings": {
      "logs": {
        "properties": {
          "messages": {
            "type": "text"
          },
          "number": {
            "type": "text"
          }
        }
      }
    }
  }
}
```

3、获取映射字段

语法：

GET /{index}/_mapping/{type}/field/{field}

GET /school/_mapping/logs/field/number

```
# GET /{index}/_mapping/{type}/field/{field}
GET /school/_mapping/logs/field/number
```

```
1 | {
2 |   "school": {
3 |     "mappings": {
4 |       "logs": {
5 |         "number": {
6 |           "full_name": "number",
7 |           "mapping": {
8 |             "number": {
9 |               "type": "text"
10 |             }
11 |           }
12 |         }
13 |       }
14 |     }
15 |   }
```

5、索引库配置管理（settings）

1、索引库配置

所谓的 **settings** 就是用来修改索引分片和副本数的；

比如有的重要索引，副本数很少甚至没有副本，那么我们可以通过 **setting** 来添加副本数

```
DELETE document
PUT document
{
  "mappings": {
    "article" : {
      "properties": {
        "title" : {"type": "text"},
        "author" : {"type": "text"},
        "titleScore" : {"type": "double"}
      }
    }
  }
}
GET /document/_settings
```

```
1 DELETE document
2 PUT document
3 {
4   "mappings": {
5     "article": {
6       "properties": {
7         "title": {"type": "text"},
8         "author": {"type": "text"},
9         "titleScore": {"type": "double"}
10      }
11    }
12  }
13 }
14 GET /document/_settings
```

```
1 {
2   "document": {
3     "settings": {
4       "index": {
5         "creation_date": "1538221578346",
6         "number_of_shards": "5",
7         "number_of_replicas": "1",
8         "uuid": "LzWu3hxvRjK5SQ_SHep0WQ",
9         "version": {
10          "created": "6000099"
11        },
12        "provided_name": "document"
13      }
14    }
15  }
16 }
```

可以看到当前的副本数是 1，那么为了提高容错性，我们可以把副本数改成 2：

```
PUT /document/_settings
{
  "number_of_replicas": 2
}
```

```
PUT /document/_settings
{
  "number_of_replicas": 2
}
GET /document/_settings
```

```
1 # PUT /document/_settings
2 { }
3
4
5
6 # GET /document/_settings
7 {
8   "document": {
9     "settings": {
10      "index": {
11        "creation_date": "1538221578346",
12        "number_of_shards": "5",
13        "number_of_replicas": "2",
14        "uuid": "LzWu3hxvRjK5SQ_SHep0WQ",
15        "version": {
16          "created": "6000099"
17        },
18        "provided_name": "document"
19      }
20    }
21  }
22 }
```

副本可以改，分片不能改

```
PUT /document/_settings
{
  "number_of_shards": 3
}
```



2、零停机重新索引数据

实际生产，对于文档的操作，偶尔会遇到这种问题：

某一个字段的类型不符合后期的业务了，但是当前的索引已经创建了，我们知道 es 在字段的 mapping 建立后就不可再次修改 mapping 的值。

1、新建索引库 articles1，并添加数据

```
DELETE articles1
PUT articles1
{
  "settings":{
    "number_of_shards":3,
    "number_of_replicas":1
  },
  "mappings":{
    "article":{
      "dynamic":"strict",
      "properties":{
        "id":{"type": "text", "store": true},
        "title":{"type": "text","store": true},
        "readCounts":{"type": "integer","store": true},
        "times": {"type": "text", "index": false}
      }
    }
  }
}

PUT articles1/article/1
{
  "id" : "1",
  "title" : "世界1",
  "readCounts" : 2 ,
  "times" : "2018-05-01"
```

```
}  
  
get articles1/article/1
```

2、新建索引库 articles2

```
DELETE articles2  
PUT articles2  
{  
  "settings":{  
    "number_of_shards":5,  
    "number_of_replicas":1  
  },  
  "mappings":{  
    "article":{  
      "dynamic":"strict",  
      "properties":{  
        "id":{"type": "text", "store": true},  
        "title":{"type": "text", "store": true},  
        "readCounts":{"type": "integer", "store": true},  
        "times": {"type": "date", "index": false}  
      }  
    }  
  }  
}  
  
GET articles2/article/1
```

3、拷贝数据并验证

```
POST _reindex  
{  
  "source": {  
    "index": "articles1"  
  },  
  "dest": {  
    "index": "articles2"  
  }  
}  
  
GET articles2/article/1
```

9、分页解决方案

1、导入数据

```
DELETE us
POST /_bulk
{ "create": { "_index": "us", "_type": "tweet", "_id": "1" }}
{ "email": "john@smith.com", "name": "John Smith", "username": "@john" }
{ "create": { "_index": "us", "_type": "tweet", "_id": "2" }}
{ "email": "mary@jones.com", "name": "Mary Jones", "username": "@mary" }
{ "create": { "_index": "us", "_type": "tweet", "_id": "3" }}
{ "date": "2014-09-13", "name": "Mary Jones", "tweet": "Elasticsearch means full text search has never been so easy", "user_id": 2 }
{ "create": { "_index": "us", "_type": "tweet", "_id": "4" }}
{ "date": "2014-09-14", "name": "John Smith", "tweet": "@mary it is not just text, it does everything", "user_id": 1 }
{ "create": { "_index": "us", "_type": "tweet", "_id": "5" }}
{ "date": "2014-09-15", "name": "Mary Jones", "tweet": "However did I manage before Elasticsearch?", "user_id": 2 }
{ "create": { "_index": "us", "_type": "tweet", "_id": "6" }}
{ "date": "2014-09-16", "name": "John Smith", "tweet": "The Elasticsearch API is really easy to use", "user_id": 1 }
{ "create": { "_index": "us", "_type": "tweet", "_id": "7" }}
{ "date": "2014-09-17", "name": "Mary Jones", "tweet": "The Query DSL is really powerful and flexible", "user_id": 2 }
{ "create": { "_index": "us", "_type": "tweet", "_id": "8" }}
{ "date": "2014-09-18", "name": "John Smith", "user_id": 1 }
{ "create": { "_index": "us", "_type": "tweet", "_id": "9" }}
{ "date": "2014-09-19", "name": "Mary Jones", "tweet": "Geo-location aggregations are really cool", "user_id": 2 }
{ "create": { "_index": "us", "_type": "tweet", "_id": "10" }}
{ "date": "2014-09-20", "name": "John Smith", "tweet": "Elasticsearch surely is one of the hottest new NoSQL products", "user_id": 1 }
{ "create": { "_index": "us", "_type": "tweet", "_id": "11" }}
{ "date": "2014-09-21", "name": "Mary Jones", "tweet": "Elasticsearch is built for the cloud, easy to scale", "user_id": 2 }
{ "create": { "_index": "us", "_type": "tweet", "_id": "12" }}
{ "date": "2014-09-22", "name": "John Smith", "tweet": "Elasticsearch and I have left the honeymoon stage, and I still love her.", "user_id": 1 }
{ "create": { "_index": "us", "_type": "tweet", "_id": "13" }}
{ "date": "2014-09-23", "name": "Mary Jones", "tweet": "So yes, I am an Elasticsearch fanboy", "user_id": 2 }
{ "create": { "_index": "us", "_type": "tweet", "_id": "14" }}
```

```
{ "date" : "2014-09-24", "name" : "John Smith", "tweet" : "How many more cheesy tweets do I have to write?", "user_id" : 1 }
```

2、size+from 浅分页

按照一般的查询流程来说，如果我想查询前 10 条数据：

- 1 客户端请求发给某个节点
- 2 节点转发给各个分片，查询每个分片上的前 10 条
- 3 结果返回给节点，整合数据，提取前 10 条
- 4 返回给请求客户端

from 定义了目标数据的偏移值，**size** 定义当前返回的事件数目

```
GET /us/_search?pretty
{
  "from" : 0 , "size" : 5
}
```

```
GET /us/_search?pretty
{
  "from" : 5 , "size" : 5
}
```

这种浅分页只适合少量数据，因为随 from 增大，查询的时间就会越大，而且数据量越大，查询的效率指数下降

优点：from+size 在数据量不大的情况下，效率比较高

缺点：在数据量非常大的情况下，from+size 分页会把全部记录加载到内存中，这样做不但运行速度特别慢，而且容易让 es 出现内存不足而挂掉

3、scroll 深分页

对于上面介绍的浅分页，当 Elasticsearch 响应请求时，它必须确定 docs 的顺序，排列响应结果。

如果请求的页数较少（假设每页 20 个 docs），Elasticsearch 不会有什么问题，但是如果页数较大时，比如请求第 20 页，Elasticsearch 不得不取出第 1 页到第 20 页的所有 docs，再去除第 1 页到第 19 页的 docs，得到第 20 页的 docs。

解决的方式就是使用 scroll，**scroll 就是维护了当前索引段的一份快照信息--缓存**（这个快照信息是你执行这个 scroll 查询时的快照）。

可以把 scroll 分为初始化和遍历两步： 1、初始化时将所有符合搜索条件的搜索结果缓存起来，可以想象成快照； 2、遍历时，从这个快照里取数据；

初始化

```
GET us/_search?scroll=3m
{
  "query": {"match_all": {}},
  "size": 3
}
```

初始化的时候就像是普通的 *search* 一样

其中的 scroll=3m 代表当前查询的数据缓存 3 分钟

Size: 3 代表当前查询 3 条数据

遍历

在遍历时候，拿到上一次遍历中的 scrollid，然后带 scroll 参数，重复上一次的遍历步骤，知道返回的数据为空，就表示遍历完成

```
GET /_search/scroll
{
  "scroll" : "1m",
  "scroll_id" :
  "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAAAAPXfk0xN1BmSnLVULdhYThEdWVzZ19xbkEAAAAAAAAAAIxZuQWVJU0VSZ1JzcVZtMGVYZ3RDaFlBAAAAAAAAAA9oWTVZOdHJ2cXBSoU9wN3c1dk5vcWd4Q0AAAAAAAAAPYFk0xN1BmSnLVULdhYThEdWVzZ19xbkEAAAAAAAAAAIhZuQWVJU0VSZ1JzcVZtMGVYZ3RDaFlB"
}
```

【注意】：每次都要传参数 scroll，刷新搜索结果的缓存时间，另外不需要指定 index 和 type（不要把缓存的时间设置太长，占用内存）

对比

浅分页，每次查询都会去索引库（本地文件夹）中查询 `pageNum*page` 条数据，然后截取掉前面的数据，留下最后的数据。这样的操作在每个分片上都会执行，最后会将多个分片的数据合并到一起，再次排序，截取需要的。

深分页，可以一次性将所有满足查询条件的数据，都放到内存中。分页的时候，在内存中查询。相对浅分页，就可以避免多次读取磁盘。

10、三台机器安装 IK 分词器

我们在搜索的时候，都会对数据进行分词，英文的分词很简单，我们可以直接按照空格进行切分即可，但是中文的分词太过复杂，例如：夏天太热，能穿多少穿多少，冬天太冷，能穿多少穿多少。下雨地滑，还好我一把把车把把住了，才没有摔倒。人要是行，干一行行一行，一行行行行行等等的分词都是非常麻烦的，所以针对中文的分词，专门出了一个叫做 IK 的分词器来解决对中文的分词问题。

1、安装

每台机器都要配置。配置完成之后，需要重启 ES 服务

将安装包上传到 node01 机器的 `/home/es` 路径下

```
cd /home/es

wget
https://github.com/medcl/elasticsearch-analysis-ik/releases/download/v6.7.0/elasticsearch-analysis-ik-6.7.0.zip

# 将 ik 分词器的插件，上传到/home/es 目录下

cd /home/es

mkdir /export/servers/es/elasticsearch-6.7.0/plugins/analysis-ik/

unzip elasticsearch-analysis-ik-6.7.0.zip -d
/export/servers/es/elasticsearch-6.7.0/plugins/analysis-ik/
```

将安装包分发到其他机器上

node01 机器执行以下命令进行安装包的分发

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090

```
cd /export/servers/es/elasticsearch-6.7.0/plugins
```

```
scp -r analysis-ik/ node02:$PWD
```

```
scp -r analysis-ik/ node03:$PWD
```

三台机器都配置完成

配置完成之后，需要重启 ES 服务。

三台机器重启 es 服务

三台机器执行以下命令停止 es 服务

```
jps
```

```
kill -9
```

```
nohup /export/servers/es/elasticsearch-6.7.0/bin/elasticsearch >/dev/null 2>&1 &
```

2、配置

```
delete iktest
PUT /iktest?pretty
{
  "settings" : {
    "analysis" : {
      "analyzer" : {
        "ik" : {
          "tokenizer" : "ik_max_word"
        }
      }
    }
  },
  "mappings" : {
    "article" : {
      "dynamic" : true,
      "properties" : {
        "subject" : {
          "type" : "text",
          "analyzer" : "ik_max_word"
        }
      }
    }
  }
}
```

```
}  
}  
}
```

说明：ik 带有两个分词器：

- **ikmaxword**：会将文本做最细粒度的拆分；尽可能多的拆分出词语

句子：我爱我的祖国

结果： 我|爱|我的|祖|国|祖国

- **ik_smart**：会做最粗粒度的拆分；已被分出的词语将不会再次被其它词语占有

句子：我爱我的祖国

结果： 我|爱|我|的|祖国

3、查看分词效果

```
GET _analyze?pretty  
{  
  "analyzer": "ik_max_word",  
  "text": "希拉里是个妹子"  
}
```

4、插入测试数据

```
POST /iktest/article/_bulk?pretty  
{ "index" : { "_id" : "1" } }  
{"subject" : " "闺蜜" 崔顺实被韩检方传唤 韩总统府促彻查真相" }  
{ "index" : { "_id" : "2" } }  
{"subject" : "韩举行 "护国训练" 青瓦台:决不许国家安全出问题" }  
{ "index" : { "_id" : "3" } }  
{"subject" : "媒体称 FBI 已经取得搜查令 检视希拉里电邮" }  
{ "index" : { "_id" : "4" } }  
{"subject" : "村上春树获安徒生奖 演讲中谈及欧洲排外问题" }  
{ "index" : { "_id" : "5" } }  
{"subject" : "希拉里团队炮轰 FBI 参院民主党领袖批其"违法"" }
```

查看分词器

对"希拉里和韩国"进行分词查询

ikmaxword 分词后的效果：希|拉|里|希拉里|和|韩国

```
POST /iktest/article/_search?pretty  
{  
  "query" : { "match" : { "subject" : "希拉里和韩国" } },
```

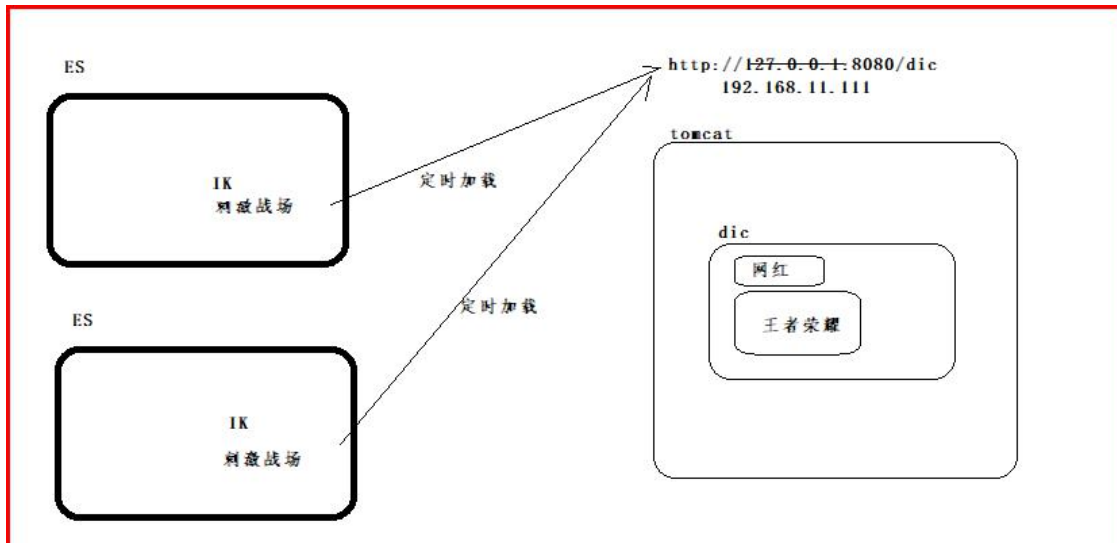
```
"highlight" : {  
  "pre_tags" : ["<font color=red>"],  
  "post_tags" : ["</font>"],  
  "fields" : {  
    "subject" : {}  
  }  
}  
}
```

5、热词更新

查看分词效果

```
GET _analyze?pretty  
{  
  "analyzer": "ik_max_word",  
  "text": "传智播客在哪里"  
}
```

1、node03 配置 Tomcat



使用 es 用户来进行配置 tomcat，此处我们将 tomcat 装在 **node03** 机器上面即可

```
cd /home/es
```

```
tar -zxvf apache-tomcat-8.5.34.tar.gz -C /export/servers/es/
```

tomcat 当中添加配置 hot.dic

```
cd /export/servers/es/apache-tomcat-8.5.34/webapps/ROOT/
```

```
vi hot.dic
```

传智播客

启动 tomcat

```
/export/servers/es/apache-tomcat-8.5.34/bin/startup.sh
```

浏览器访问

```
http://node03:8080/hot.dic
```

2、三台机器修改配置文件

三台机器都要修改 es 的配置文件（使用 es 用户来进行修改即可）

第一台机器 node01 修改 es 的配置

```
cd /export/servers/es/elasticsearch-6.0.0/plugins/analysis-ik/config
```

```
vim IKAnalyzer.cfg.xml
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM
"http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>IK Analyzer 扩展配置</comment>
    <!-- 用户可以在这里配置自己的扩展字典 -->
    <entry key="ext_dict"></entry>
    <!-- 用户可以在这里配置自己的扩展停止词字典 -->
    <entry key="ext_stopwords"></entry>
    <!-- 用户可以在这里配置远程扩展字典 -->
```

```
<entry
key="remote_ext_dict">http://node03:8080/hot.dic</entry>

<!-- 用户可以在这里配置远程扩展停止词字典-->

<!-- <entry
key="remote_ext_stopwords">words_location</entry> -->

</properties>
```

修改完成之后拷贝到 node02 与 node03 机器上面去

node01 执行以下命令进行拷贝

```
cd /export/servers/es/elasticsearch-6.0.0/plugins/analysis-ik/config
sudo scp IKAnalyzer.cfg.xml node02:$PWD
sudo scp IKAnalyzer.cfg.xml node03:$PWD
```

3、三台机器修改 JDK 权限

三台机器修改 jdk 的权限问题

第一台机器执行以下命令修改 jdk 的权限问题

```
#修改 JDK 安全设置
cd /export/servers/jdk1.8.0_141/jre/lib/security
sudo vim java.policy
```

添加以下四行配置

```
permission java.net.SocketPermission "192.168.227.30:8080","accept";
permission java.net.SocketPermission "192.168.227.30:8080","listen";
permission java.net.SocketPermission "192.168.227.30:8080","resolve";
permission java.net.SocketPermission "192.168.227.30:8080","connect";
```

修改完成之后拷贝到第二台和第三台机器

node01 执行以下命令拷贝到第二台和第三台

```
cd /export/servers/jdk1.8.0_141/jre/lib/security
sudo scp java.policy node02:$PWD
```

```
sudo scp java.policy node03:$PWD
```

<http://mahilion.blog.163.com/blog/static/1830872952012101225243655/>

4、三台机器重新启动 es

三台机器重新启动 es 服务，三台机器先使用 kill -9 杀死 es 的服务，然后再执行以下命令进行重启

```
cd /export/servers/es/elasticsearch-6.7.0/  
nohup bin/elasticsearch >/dev/null 2>&1 &
```

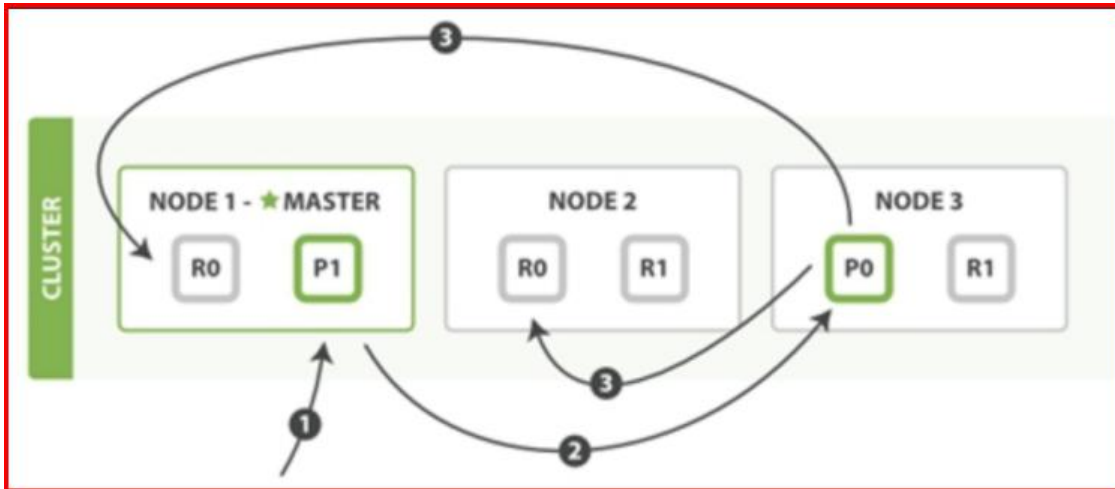
```
[2018-10-14T01:23:32,850][INFO ][o.w.a.d.Monitor] [Dict Loading] http://192.168.140.133:8080/hot.dic  
[2018-10-14T01:23:33,351][INFO ][o.w.a.d.Monitor] 传智播客  
[2018-10-14T01:23:43,392][INFO ][o.w.a.d.Monitor] 重新加载词典...  
[2018-10-14T01:23:43,395][INFO ][o.w.a.d.Monitor] try load config from /export/servers/es/elasticsearch-6.0.0/config/a  
[2018-10-14T01:23:43,397][INFO ][o.w.a.d.Monitor] try load config from /export/servers/es/elasticsearch-6.0.0/plugins/  
[2018-10-14T01:23:43,767][INFO ][o.w.a.d.Monitor] [Dict Loading] http://192.168.140.133:8080/hot.dic  
[2018-10-14T01:23:43,790][INFO ][o.w.a.d.Monitor] 传智播客  
[2018-10-14T01:23:43,791][INFO ][o.w.a.d.Monitor] 重新加载词典完毕...  
[2018-10-14T01:24:43,383][INFO ][o.w.a.d.Monitor] 重新加载词典...  
[2018-10-14T01:24:43,383][INFO ][o.w.a.d.Monitor] try load config from /export/servers/es/elasticsearch-6.0.0/config/a  
[2018-10-14T01:24:43,384][INFO ][o.w.a.d.Monitor] try load config from /export/servers/es/elasticsearch-6.0.0/plugins/  
[2018-10-14T01:24:43,512][INFO ][o.w.a.d.Monitor] [Dict Loading] http://192.168.140.133:8080/hot.dic  
[2018-10-14T01:24:43,528][INFO ][o.w.a.d.Monitor] 传智播客  
[2018-10-14T01:24:43,529][INFO ][o.w.a.d.Monitor] 王者荣耀  
[2018-10-14T01:24:43,529][INFO ][o.w.a.d.Monitor] 重新加载词典完毕...
```

查看我们的分词过程

```
GET _analyze?pretty  
  
{  
  "analyzer": "ik_max_word",  
  "text": "传智播客在哪里"  
}
```


11、分片交互过程

1、创建索引



首先：发送一个索引或者删除的请求给 node1

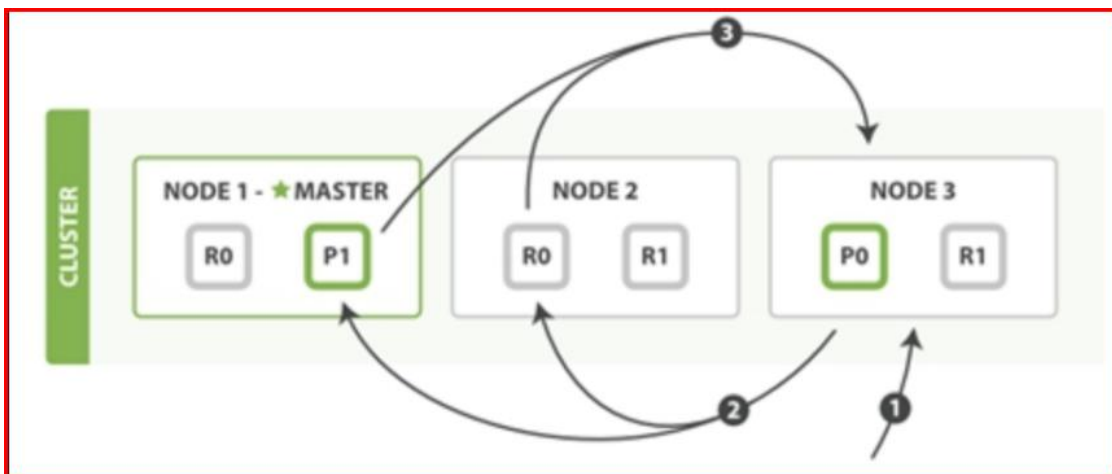
其次：node1 介绍到请求之后，会根据请求中携带的参数“文档 id”判断出该文档应该存储在具体哪一个 shard 中

$\text{shard} = \text{hash}(\text{routing}) \% \text{numberofprimary_shards}$

，比如 shard0；其次就是 node1 通过元数据信息可以知道 shard0 在具体哪一个节点，于是 node1 会把请求转发给 node3

最后：node3 接收到请求之后会将请求并行的分发给 shard0 的所有 replica shard 之上，也就是存在于 node 1 和 node 2 中的 replica shard；如果所有的 replica shard 都成功地执行了请求，那么将会向 node 3 回复一个成功确认，当 node 3 收到了所有 replica shard 的确认信息后，则最后向用户返回一个 Success 的消息。

3、查询索引



检索文档的时候，我们并不知道文档在集群中的哪个位置，所以一般情况下不得不去询问 index 中的每一个 shard，然后将结果拼接成一个大的已排好序的汇总结果列表；

(1)：客户端发送一个检索请求给 node3，此时 node3 会创建一个空的优先级队列并且配置好分页参数 from 与 size。

(2)：node3 将检索请求发送给 index 中的每一个 shard (primary 和 replica)，每一个在本地执行检索，并将结果添加到本地的优先级队列中；

(3)：每个 shard 返回本地优先级序列中所记录的_id 与**score 值**，并发送 node3。Node3 将这些值合并到自己的本地的优先级队列中，并做全局的排序（node 3 将它们合并成一条汇总的结果），返回给客户端。

12、使用 Java API 访问集群

1、导入 pom

```

<dependencies>
  <dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>transport</artifactId>
    <version>6.7.0</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
  
```

```
<artifactId>log4j-core</artifactId>
<version>2.9.1</version>
</dependency>

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>6.14.3</version>
  <scope>test</scope>
</dependency>

<!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.47</version>
</dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

2、创建索引 prepareIndex

1、创建 Client

```
private TransportClient client;
@BeforeEach
public void test1() throws UnknownHostException {
  Settings settings = Settings.builder().put("cluster.name",
    "myes").build();
  client = new
```

```
PreBuiltTransportClient(settings).addTransportAddress(new  
TransportAddress(InetAddress.getByName("node01"),9300)).addTransport  
Address(new TransportAddress(InetAddress.getByName("node02"),9300));  
}
```

2、自己拼装json 创建索引保存到 myindex1 索引库下面的 article 当中去

```
/**  
 * 插入 json 格式的索引数据  
 */  
@Test  
public void createIndex(){  
    String json = "{" +  
        "\"user\":\"kimchy\"," +  
        "\"postDate\":\"2013-01-30\"," +  
        "\"message\":\"travelling out Elasticsearch\"" +  
        "}" ;  
    IndexResponse indexResponse = client.prepareIndex("myindex1",  
"article", "1").setSource(json, XContentType.JSON).get();  
    client.close();  
}
```

3、使用 map 创建索引

```
@Test  
public void index2() throws Exception {  
    HashMap<String, String> jsonMap = new HashMap<String, String>();  
    jsonMap.put("name", "zhangsan");  
    jsonMap.put("sex", "1");  
    jsonMap.put("age", "18");  
    jsonMap.put("address", "bj");  
    IndexResponse indexResponse = client.prepareIndex("myindex1",  
"article", "2")  
        .setSource(jsonMap)  
        .get();  
    client.close();  
}
```

4、XcontentBuilder 实现创建索引

```
/**  
 * 通过 XContentBuilder 来实现索引的创建  
 * @throws IOException  
 */  
@Test  
public void index3() throws IOException {  
    IndexResponse indexResponse = client.prepareIndex("myindex1",
```

```
"article", "3")
        .setSource(new XContentFactory().jsonBuilder()
            .startObject()
            .field("name", "lisi")
            .field("age", "18")
            .field("sex", "0")
            .field("address", "bj")
            .endObject())
        .get();
client.close();
}
```

5、将对象转换为 json 格式字符串进行创建索引

定义 person 对象

```
public class Person implements Serializable{
    private Integer id;
    private String name ;
    private Integer age;
    private Integer sex;
    private String address;
    private String phone;
    private String email;
    private String say;

    public Person() {
    }

    public Person(Integer id, String name, Integer age, Integer sex, String
address, String phone, String email,String say) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.sex = sex;
        this.address = address;
        this.phone = phone;
        this.email = email;
        this.say = say;
    }

    public String getSay() {
        return say;
    }

    public void setSay(String say) {
        this.say = say;
    }
}
```

```
public Integer getId() {  
    return id;  
}  
  
public void setId(Integer id) {  
    this.id = id;  
}  
  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
public Integer getAge() {  
    return age;  
}  
  
public void setAge(Integer age) {  
    this.age = age;  
}  
  
public Integer getSex() {  
    return sex;  
}  
  
public void setSex(Integer sex) {  
    this.sex = sex;  
}  
  
public String getAddress() {  
    return address;  
}  
  
public void setAddress(String address) {  
    this.address = address;  
}  
  
public String getPhone() {  
    return phone;  
}  
  
public void setPhone(String phone) {  
    this.phone = phone;  
}  
  
public String getEmail() {
```

```
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

插入索引数据

```
/**
 * 将 java 对象转换为 json 格式字符串进行创建索引
 */
@Test
public void objToIndex(){
    Person person = new Person();
    person.setAge(18);
    person.setId(20);
    person.setName("张三丰");
    person.setAddress("武当山");
    person.setEmail("zhangsanfeng@163.com");
    person.setPhone("18588888888");
    person.setSex(1);
    String json = JSONObject.toJSONString(person);
    System.out.println(json);

    client.prepareIndex("myindex1","article","32").setSource(json,XContentType.JSON).get();
    client.close();
}
```

6、批量创建索引

```
/**
 * 批量创建索引
 * @throws IOException
 */
@Test
public void index4() throws IOException {
    BulkRequestBuilder bulk = client.prepareBulk();
    bulk.add(client.prepareIndex("myindex1", "article", "4")
        .setSource(new XContentFactory().jsonBuilder()
            .startObject()
            .field("name", "wangwu")
            .field("age", "18")
            .field("sex", "0")
            .field("address", "bj")
            .endObject()));
    bulk.add(client.prepareIndex("news", "article", "5")
        .setSource(new XContentFactory().jsonBuilder()
            .startObject()
            .field("name", "wangwu")
            .field("age", "18")
            .field("sex", "0")
            .field("address", "bj")
            .endObject()));
}
```



```
        .startObject()
        .field("name", "zhaoliu")
        .field("age", "18")
        .field("sex", "0")
        .field("address", "bj")
        .endObject());
BulkResponse bulkResponse = bulk.get();
System.out.println(bulkResponse);
client.close();
}
```

3、更新索引

根据系统给数据生成的 id 来进行更新索引

```
/**
 * 更新索引
 * 根据数据 id 来进行更新索引
 */
@Test
public void updateIndex(){
    Person guansheng = new Person(5, "宋江", 88, 0, "水泊梁山",
    "17666666666", "wusong@itcast.com", "及时雨宋江");

    client.prepareUpdate().setIndex("myindex1").setType("article").setId("8")
        .setDoc(JSONObject.toJSONString(guansheng), XContentType
        .JSON)
        .get();
    client.close();
}
```

4、删除索引

1、按照 id 进行删除

```
/**
 * 按照 id 进行删除数据
 */
@Test
public void deleteById(){
    DeleteResponse deleteResponse = client.prepareDelete("myindex1",
    "article", "8").get();
}
```

```
}  
    client.close();  
}
```

2、删除整个索引库

```
/**  
 * 删除索引  
 * 删除整个索引库  
 */  
@Test  
public void deleteIndex(){  
    DeleteIndexResponse indexsearch =  
    client.admin().indices().prepareDelete("myindex1").execute().actionGet();  
    client.close();  
}
```

5、查询索引

1、初始化一批数据到索引库中准备查询

```
/**  
 * 初始化一批数据到索引库当中去准备做查询使用  
 * 注意这里初始化的时候，需要给我们的数据设置分词属性  
 * @throws Exception  
 */  
@Test  
public void createIndexBatch() throws Exception {  
    Settings settings = Settings  
        .builder()  
        .put("cluster.name", "myes") //节点名称， 在 es 配置的时候  
        //自动发现我们其他的 es 的服务器  
        .put("client.transport.sniff", "true")  
        .build();  
    //创建客户端  
    TransportClient client = new PreBuiltTransportClient(settings)  
        .addTransportAddress(new  
    TransportAddress(InetAddress.getByName("node01"), 9300)); //以本机作为  
    节点  
    //创建映射  
    XContentBuilder mapping = XContentFactory.jsonBuilder()  
        .startObject()  
        .startObject("properties")
```

```
//      .startObject("m_id").field("type", "keyword").endObject()
      .startObject("id").field("type", "integer").endObject()
      .startObject("name").field("type", "text").field("analyzer",
"ik_max_word").endObject()
      .startObject("age").field("type", "integer").endObject()
      .startObject("sex").field("type", "text").field("analyzer",
"ik_max_word").endObject()
      .startObject("address").field("type", "text").field("analyzer",
"ik_max_word").endObject()
      .startObject("phone").field("type", "text").endObject()
      .startObject("email").field("type", "text").endObject()
      .startObject("say").field("type", "text").field("analyzer",
"ik_max_word").endObject()
      .endObject()
      .endObject();
      //pois: 索引名      cxyword: 类型名 (可以自己定义)
      PutMappingRequest putmap =
Requests.putMappingRequest("indexsearch").type("mysearch").source(map
ping);
      //创建索引
client.admin().indices().prepareCreate("indexsearch").execute().actionGet();
      //为索引添加映射
      client.admin().indices().putMapping(putmap).actionGet();

      BulkRequestBuilder bulkRequestBuilder = client.prepareBulk();
      Person lujunyi = new Person(2, "玉麒麟卢俊义", 28, 1, "水泊梁山",
"1766666666", "lujunyi@itcast.com", "hello world 今天天气还不错");
      Person wuyong = new Person(3, "智多星吴用", 45, 1, "水泊梁山",
"1766666666", "wuyong@itcast.com", "行走四方，抱打不平");
      Person gongsunsheng = new Person(4, "入云龙公孙胜", 30, 1, "水泊梁
山", "1766666666", "gongsunsheng@itcast.com", "走一个");
      Person guansheng = new Person(5, "大刀关胜", 42, 1, "水泊梁山",
"1766666666", "wusong@itcast.com", "我的大刀已经饥渴难耐");
      Person linchong = new Person(6, "豹子头林冲", 18, 1, "水泊梁山",
"1766666666", "linchong@itcast.com", "梁山好汉");
      Person qinming = new Person(7, "霹雳火秦明", 28, 1, "水泊梁山",
"1766666666", "qinming@itcast.com", "不太了解");
      Person huyanzhuo = new Person(8, "双鞭呼延灼", 25, 1, "水泊梁山",
"1766666666", "huyanzhuo@itcast.com", "不是很熟悉");
      Person huarong = new Person(9, "小李广花荣", 50, 1, "水泊梁山",
"1766666666", "huarong@itcast.com", "打酱油的");
      Person chaijin = new Person(10, "小旋风柴进", 32, 1, "水泊梁山",
"1766666666", "chaijin@itcast.com", "吓唬人的");
      Person zhisheng = new Person(13, "花和尚鲁智深", 15, 1, "水泊梁山",
"1766666666", "luzhisheng@itcast.com", "倒拔杨垂柳");
      Person wusong = new Person(14, "行者武松", 28, 1, "水泊梁山",
"1766666666", "wusong@itcast.com", "二营长。。。。。。");
```

```
bulkRequestBuilder.add(client.prepareIndex("indexsearch",
"mysearch", "1")
    .setSource(JSONObject.toJSONString(lujunyi),
XContentType.JSON)
);
bulkRequestBuilder.add(client.prepareIndex("indexsearch",
"mysearch", "2")
    .setSource(JSONObject.toJSONString(wuyong),
XContentType.JSON)
);
bulkRequestBuilder.add(client.prepareIndex("indexsearch",
"mysearch", "3")
    .setSource(JSONObject.toJSONString(gongsunsheng),
XContentType.JSON)
);
bulkRequestBuilder.add(client.prepareIndex("indexsearch",
"mysearch", "4")
    .setSource(JSONObject.toJSONString(guansheng),
XContentType.JSON)
);
bulkRequestBuilder.add(client.prepareIndex("indexsearch",
"mysearch", "5")
    .setSource(JSONObject.toJSONString(linchong),
XContentType.JSON)
);
bulkRequestBuilder.add(client.prepareIndex("indexsearch",
"mysearch", "6")
    .setSource(JSONObject.toJSONString(qinming),
XContentType.JSON)
);
bulkRequestBuilder.add(client.prepareIndex("indexsearch",
"mysearch", "7")
    .setSource(JSONObject.toJSONString(huyanzhuo),
XContentType.JSON)
);
bulkRequestBuilder.add(client.prepareIndex("indexsearch",
"mysearch", "8")
    .setSource(JSONObject.toJSONString(huarong),
XContentType.JSON)
);
bulkRequestBuilder.add(client.prepareIndex("indexsearch",
"mysearch", "9")
    .setSource(JSONObject.toJSONString(chaijin),
XContentType.JSON)
);
bulkRequestBuilder.add(client.prepareIndex("indexsearch",
"mysearch", "10")
    .setSource(JSONObject.toJSONString(zhisheng),
XContentType.JSON)
);
```

```
bulkRequestBuilder.add(client.prepareIndex("indexsearch",
"mysearch", "11")
    .setSource(JSONObject.toJSONString(wusong),
XContentType.JSON)
);

bulkRequestBuilder.get();
client.close();
}
```

2、通过数据 id 使用 prepareGet 来查询索引

通过 id 来进行查询索引

```
/**
 * 通过 id 来进行精确查询
 */
@Test
public void query1() {
    GetResponse documentFields = client.prepareGet("indexsearch",
"mysearch", "11").get();
    String index = documentFields.getIndex();
    String type = documentFields.getType();
    String id = documentFields.getId();
    System.out.println(index);
    System.out.println(type);
    System.out.println(id);
    Map<String, Object> source = documentFields.getSource();
    for (String s : source.keySet()) {
        System.out.println(source.get(s));
    }
}
```

3、查询索引库当中的所有数据

```
/**
 * 查询所有数据
 */
@Test
public void queryAll() {
    SearchResponse searchResponse = client
        .prepareSearch("indexsearch")
        .setTypes("mysearch")
```

```
        .setQuery(new MatchAllQueryBuilder())
        .get();
SearchHits searchHits = searchResponse.getHits();
SearchHit[] hits = searchHits.getHits();
for (SearchHit hit : hits) {
    String sourceAsString = hit.getSourceAsString();
    System.out.println(sourceAsString);
}
client.close();
}
```

4、RangeQuery 范围值查询

查找索引库当中年龄为 18 到 28 的所有人

```
/**
 * 查找年龄 18 到 28 的人,包含 18 和 28
 */
@Test
public void rangeQuery(){
    SearchResponse searchResponse = client.prepareSearch("indexsearch")
        .setTypes("mysearch")
        .setQuery(new RangeQueryBuilder("age").gt(17).lt(29))
        .get();
    SearchHits hits = searchResponse.getHits();
    SearchHit[] hits1 = hits.getHits();
    for (SearchHit documentFields : hits1) {
        System.out.println(documentFields.getSourceAsString());
    }
    client.close();
}
```

5、termQuery 词条查询

```
/**
 * 词条查询
 */
@Test
public void termQuery(){
    SearchResponse searchResponse =
    client.prepareSearch("indexsearch").setTypes("mysearch")
        .setQuery(new TermQueryBuilder("say", "熟悉"))
        .get();
    SearchHits hits = searchResponse.getHits();
    SearchHit[] hits1 = hits.getHits();
    for (SearchHit documentFields : hits1) {
```

```
        System.out.println(documentFields.getSourceAsString());  
    }  
}
```

6、fuzzyQuery 模糊查询

模糊查询可以自动帮我们纠正写错误的英文单词，最大纠正次数两次

```
/**  
 * fuzzyQuery 表示英文单词的最大可纠正次数，最大可以自动纠正两次  
 */  
@Test  
public void fuzzyQuery(){  
    SearchResponse searchResponse =  
client.prepareSearch("indexsearch").setTypes("mysearch")  
        .setQuery(QueryBuilders.fuzzyQuery("say",  
"helOL").fuzziness(Fuzziness.TWO))  
        .get();  
    SearchHits hits = searchResponse.getHits();  
    SearchHit[] hits1 = hits.getHits();  
    for (SearchHit documentFields : hits1) {  
        System.out.println(documentFields.getSourceAsString());  
    }  
    client.close();  
}
```

7、wildCardQuery 通配符查询

*：匹配任意多个字符

?：仅匹配一个字符

```
/**  
 * 模糊匹配查询有两种匹配符，分别是"*" 以及 "?", 用"*"来匹配任何  
 * 字符，包括空字符串。用"?"来匹配任意的单个字符  
 */  
@Test  
public void wildCardQueryTest(){  
    SearchResponse searchResponse =  
client.prepareSearch("indexsearch").setTypes("mysearch")  
        .setQuery(QueryBuilders.wildcardQuery("say", "hel*"))  
        .get();  
    SearchHits hits = searchResponse.getHits();  
    SearchHit[] hits1 = hits.getHits();  
    for (SearchHit documentFields : hits1) {  
        System.out.println(documentFields.getSourceAsString());  
    }  
}
```

```
}  
    client.close();  
}
```

8、boolQuery 多条件组合查询

使用 boolQuery 实现多条件组合查询

查询年龄是 18 到 28 范围内且性别是男性的，或者 id 范围在 10 到 13 范围内的

```
/**  
 * 多条件组合查询 boolQuery  
 * 查询年龄是 18 到 28 范围内且性别是男性的，或者 id 范围在 10 到 13 范围内的  
 */  
@Test  
public void boolQueryTest(){  
    RangeQueryBuilder age =  
        QueryBuilders.rangeQuery("age").gt(17).lt(29);  
    TermQueryBuilder sex = QueryBuilders.termQuery("sex", "1");  
    RangeQueryBuilder id =  
        QueryBuilders.rangeQuery("id").gt("9").lt("15");  
  
    SearchResponse searchResponse =  
        client.prepareSearch("indexsearch").setTypes("mysearch")  
            .setQuery(  
                QueryBuilders.boolQuery().should(id)  
                    .should(QueryBuilders.boolQuery().must(sex).must(age))  
            ).get();  
    SearchHits hits = searchResponse.getHits();  
    SearchHit[] hits1 = hits.getHits();  
    for (SearchHit documentFields : hits1) {  
        System.out.println(documentFields.getSourceAsString());  
    }  
    client.close();  
}
```

9、分页与高亮查询

1、分页查询

```
/**分页查询  
 */
```



```
@Test
public void getPageIndex(){
    int    pageSize = 5;
    int    pageNum = 2;
    int    startNum = (pageNum-1)*5;
    SearchResponse searchResponse = client.prepareSearch("indexsearch")
        .setTypes("mysearch")
        .setQuery(QueryBuilders.matchAllQuery())
        .addSort("id",SortOrder.ASC)
        .setFrom(startNum)
        .setSize(pageSize)
        .get();
    SearchHits hits = searchResponse.getHits();
    SearchHit[] hits1 = hits.getHits();
    for (SearchHit documentFields : hits1) {
        System.out.println(documentFields.getSourceAsString());
    }
    client.close();
}
```

2、高亮查询

在进行关键字搜索时，搜索出的内容中的关键字会显示不同的颜色，称之为高亮
百度搜索关键字"传智播客"

Baidu 传智播客

网页 新闻 贴吧 知道 音乐 图片 视频 地图 文库 更多»

百度为您找到相关结果约2,150,000个 搜索工具

传智播客官网-一样的教育,不一样的品质,改变中国IT教育,我们正...

传智播客 www.itcast.cn

传智播客 专注IT培训,Java培训,Android培训,安卓培训,PHP培训,C++培训,网页设计培训,平面设计培训,UI设计培训,移动开发培训,网络营销培训,web前端培训,云计算大数据...

www.itcast.cn/ 百度快照 - 325条评价

传智师资

传智播客师资库聚集众多传智播客 JavaEE,Android,PHP,iOS,UI设计, 前端与

java培训

传智播客Java培训是Java培训佼佼者,口碑良好的java培训学校,并提供Java培

传智播客和黑马程序员视频库 传智播客和黑马程序员全套视频教程下载

传智播客和黑马程序员视频库-免费提供传智播客和黑马程序员全套视频教程下载和免费公开课,以及各学科学习路线图。

yun.itheima.com/ 百度快照

京东商城搜索"笔记本"

联想 GTX 1050 Ti

¥6099.00

已浏览品牌 联想(Lenovo)拯救者R720 15.6英寸大屏游戏笔记本电脑(i5-7300HQ 13万+条评价 二手有售 联想电脑京东自营旗舰店 自营

戴尔 GTX 1050

¥6699.00

戴尔DELL灵越游匣15.6英寸"吃鸡"游戏笔记本电脑(i7-7700HQ 8G 128GSSD+1T 11万+条评价 二手有售 戴尔京东自营官方旗舰店 自营

Apple

¥7899.00 ¥7488.00 PLUS

京东精选 Apple MacBook Air 13.3英寸笔记本电脑 银色(2017新款Core i5 处理 19万+条评价 二手有售 京东Apple产品专营店 自营

3、高亮显示的html 分析

通过开发者工具查看高亮数据的 html 代码实现：



ElasticSearch 可以对查询出的内容中关键字部分进行标签和样式的设置，但是你需要告诉 ElasticSearch 使用什么标签对高亮关键字进行包裹

4、高亮显示代码实现

```
/**
 * 高亮查询
 */
@Test
public void highLight(){
    //设置我们的查询高亮字段
    SearchRequestBuilder searchRequestBuilder =
    client.prepareSearch("indexsearch")
        .setTypes("mysearch")
        .setQuery(QueryBuilders.termQuery("say", "hello"));
```

```
//设置我们字段高亮的前缀与后缀
HighlightBuilder highlightBuilder = new HighlightBuilder();
highlightBuilder.field("say").preTags("<font
style='color:red'>").postTags("</font>");

//通过高亮来进行我们的数据查询
SearchResponse searchResponse =
searchRequestBuilder.highlighter(highlightBuilder).get();
SearchHits hits = searchResponse.getHits();
System.out.println("查询出来一共" + hits.getTotalHits() + "条数据");
for (SearchHit hit : hits) {
    //打印没有高亮显示的数据
    System.out.println(hit.getSourceAsString());
    System.out.println("=====");
    //打印我们经过高亮显示之后的数据
    Text[] says = hit.getHighlightFields().get("say").getFragments();
    for (Text say : says) {
        System.out.println(say);
    }
}

/*    Map<String, HighlightField> highlightFields =
hit.getHighlightFields();
    System.out.println(highlightFields);*/
}
client.close();
}
```

13、ES 的 java 操作高级 API

现有结构化数据内容如下：

name	age	salary	team	position
张云雷	26	2000	war	pf

特斯拉	20	500	tim	sf
于谦	25	2000	cav	pg
爱迪生	40	1000	tim	pf
爱因斯坦	21	300	tim	sg
郭德纲	33	3000	cav	sf
牛顿	21	500	tim	c
岳云鹏	29	1000	war	pg

初始化一批数据到 es 索引库当中去

```
/**
 * 批量添加数据
 * @throws IOException
 * @throws ExecutionException
 * @throws InterruptedException
 */
@Test
public void addIndexDatas() throws IOException, ExecutionException,
InterruptedException {
    //获取 settings
    //配置 es 集群的名字
    Settings settings = Settings.builder().put("cluster.name",
"myes").build();
    //获取客户端
    TransportAddress transportAddress = new
TransportAddress(InetAddress.getByName("node01"), 9300);

    TransportAddress transportAddress2 = new
TransportAddress(InetAddress.getByName("node02"), 9300);

    TransportAddress transportAddress3 = new
TransportAddress(InetAddress.getByName("node03"), 9300);
    //获取 client 客户端
    TransportClient client = new
PreBuiltTransportClient(settings).addTransportAddress(transportAddress).a
```

```
ddTransportAddress(transportAddress2).addTransportAddress(transportAd
dress3);

/**
 * 创建索引
 */
client.admin().indices().prepareCreate("player").get();
//构建 json 的数据格式，创建映射
XContentBuilder mappingBuilder = jsonBuilder()
    .startObject()
    .startObject("player")
    .startObject("properties")
    .startObject("name").field("type", "text").field("index",
"true").field("fielddata", "true").endObject()
    .startObject("age").field("type", "integer").endObject()
    .startObject("salary").field("type", "integer").endObject()
    .startObject("team").field("type", "text").field("index",
"true").field("fielddata", "true").endObject()
    .startObject("position").field("type", "text").field("index",
"true").field("fielddata", "true").endObject()
    .endObject()
    .endObject()
    .endObject();
PutMappingRequest request =
Requests.putMappingRequest("player")
    .type("player")
    .source(mappingBuilder);
client.admin().indices().putMapping(request).get();

//批量添加数据开始

BulkRequestBuilder bulkRequest = client.prepareBulk();

// either use client#prepare, or use Requests# to directly build index/delete
requests
    bulkRequest.add(client.prepareIndex("player", "player", "1")
        .setSource(jsonBuilder()
            .startObject()
            .field("name", "郭德纲")
            .field("age", 33)
            .field("salary", 3000)
            .field("team", "cav")
            .field("position", "sf")
            .endObject()
        )
    );
    bulkRequest.add(client.prepareIndex("player", "player", "2")
        .setSource(jsonBuilder()
            .startObject()
```

```
.field("name", "于谦")
.field("age", 25)
.field("salary", 2000)
.field("team", "cav")
.field("position", "pg")
.endObject()
)
);
bulkRequest.add(client.prepareIndex("player", "player", "3")
.setSource(jsonBuilder()
.startObject()
.field("name", "岳云鹏")
.field("age", 29)
.field("salary", 1000)
.field("team", "war")
.field("position", "pg")
.endObject()
)
);
bulkRequest.add(client.prepareIndex("player", "player", "4")
.setSource(jsonBuilder()
.startObject()
.field("name", "爱因斯坦")
.field("age", 21)
.field("salary", 300)
.field("team", "tim")
.field("position", "sg")
.endObject()
)
);
bulkRequest.add(client.prepareIndex("player", "player", "5")
.setSource(jsonBuilder()
.startObject()
.field("name", "张云雷")
.field("age", 26)
.field("salary", 2000)
.field("team", "war")
.field("position", "pf")
.endObject()
)
);
bulkRequest.add(client.prepareIndex("player", "player", "6")
.setSource(jsonBuilder()
.startObject()
.field("name", "爱迪生")
.field("age", 40)
.field("salary", 1000)
.field("team", "tim")
```

```
                .field("position", "pf")
                .endObject()
            )
        );
        bulkRequest.add(client.prepareIndex("player", "player", "7")
            .setSource(jsonBuilder()
                .startObject()
                .field("name", "牛顿")
                .field("age", 21)
                .field("salary", 500)
                .field("team", "tim")
                .field("position", "c")
                .endObject()
            )
        );
        bulkRequest.add(client.prepareIndex("player", "player", "8")
            .setSource(jsonBuilder()
                .startObject()
                .field("name", "特斯拉")
                .field("age", 20)
                .field("salary", 500)
                .field("team", "tim")
                .field("position", "sf")
                .endObject()
            )
        );
        BulkResponse bulkResponse = bulkRequest.get();
        client.close();
    }
```

需求一：统计每个球队当中球员的数量

sql 语句实现

```
select team, count(1) as player_count from player group by team;
```

使用 javaAPI 实现


```
@Test
public void groupAndCount() {
    //1: 构建查询提交
    SearchRequestBuilder builder =
    client.prepareSearch("player").setTypes("player");
    //2: 指定聚合条件
    TermsAggregationBuilder team =
    AggregationBuilders.terms("player_count").field("team");
    //3:将聚合条件放入查询条件中
    builder.addAggregation(team);
    //4:执行 action, 返回 searchResponse
    SearchResponse searchResponse = builder.get();
    Aggregations aggregations = searchResponse.getAggregations();
    for (Aggregation aggregation : aggregations) {
        StringTerms stringTerms = (StringTerms) aggregation;
        List<StringTerms.Bucket> buckets = stringTerms.getBuckets();
        for (StringTerms.Bucket bucket : buckets) {
            System.out.println(bucket.getKey());
            System.out.println(bucket.getDocCount());
        }
    }
}
```

需求二：统计每个球队中每个位置的球员数量

sql 语句实现

```
select team, position, count(1) as pos_count from player group by team, position;
```

java 代码实现

```
/**
 * 统计每个球队中每个位置的球员数量
 */
@Test
public void teamAndPosition(){
    SearchRequestBuilder builder =
    client.prepareSearch("player").setTypes("player");
    TermsAggregationBuilder team =
    AggregationBuilders.terms("player_count").field("team");
    TermsAggregationBuilder position =
    AggregationBuilders.terms("posititon_count").field("position");
    team.subAggregation(position);
    SearchResponse searchResponse =
```

```
builder.addAggregation(team).addAggregation(position).get();
Aggregations aggregations = searchResponse.getAggregations();
for (Aggregation aggregation : aggregations) {
    // System.out.println(aggregation.toString());
    StringTerms stringTerms = (StringTerms) aggregation;
    List<StringTerms.Bucket> buckets = stringTerms.getBuckets();
    for (StringTerms.Bucket bucket : buckets) {
        long docCount = bucket.getDocCount();
        Object key = bucket.getKey();
        System.out.println("当前队伍名称为" + key + "该队伍下有"
            "+docCount + "个球员");

        Aggregation posititon_count =
            bucket.getAggregations().get("posititon_count");
        if(null != posititon_count){
            StringTerms positionTrem = (StringTerms)
                posititon_count;
            List<StringTerms.Bucket> buckets1 =
                positionTrem.getBuckets();
            for (StringTerms.Bucket bucket1 : buckets1) {
                Object key1 = bucket1.getKey();
                long docCount1 = bucket1.getDocCount();
                System.out.println("该队伍下面的位置为" +
                    key1+"该位置下有" + docCount1 +"人");
            }
        }
    }
}
```

需求三：分组求各种值

计算每个球队年龄最大值

select team, max(age) as max_age from player group by team;

```
/**
 * 计算每个球队年龄最大值
 */
@Test
public void groupAndMax(){
    SearchRequestBuilder builder =
        client.prepareSearch("player").setTypes("player");
    TermsAggregationBuilder team =
        AggregationBuilders.terms("team_group").field("team");
```

```
MaxAggregationBuilder age =
AggregationBuilders.max("max_age").field("age");

team.subAggregation(age);
SearchResponse searchResponse =
builder.addAggregation(team).get();
Aggregations aggregations = searchResponse.getAggregations();
for (Aggregation aggregation : aggregations) {
    StringTerms stringTerms = (StringTerms) aggregation;
    List<StringTerms.Bucket> buckets = stringTerms.getBuckets();
    for (StringTerms.Bucket bucket : buckets) {
        Aggregation max_age =
bucket.getAggregations().get("max_age");
        System.out.println(max_age.toString());
    }
}
```

需求四：统计每个球队年龄最小值

计算每个球队年龄最大/最小/总/平均的球员年龄

select team, min(age) as min_age from player group by team;

```
/**
 * 统计每个球队中年龄最小值
 */
@Test
public void teamMinAge(){

    SearchRequestBuilder builder =
client.prepareSearch("player").setTypes("player");

    TermsAggregationBuilder team =
AggregationBuilders.terms("team_count").field("team");

    MinAggregationBuilder age =
AggregationBuilders.min("min_age").field("age");

    TermsAggregationBuilder termAggregation =
team.subAggregation(age);

    SearchResponse searchResponse =
builder.addAggregation(termAggregation).get();
    Aggregations aggregations = searchResponse.getAggregations();
    for (Aggregation aggregation : aggregations) {
```

```
System.out.println(aggregation.toString());

StringTerms stringTerms = (StringTerms) aggregation;
List<StringTerms.Bucket> buckets = stringTerms.getBuckets();
for (StringTerms.Bucket bucket : buckets) {
    Aggregations aggregations1 = bucket.getAggregations();
    for (Aggregation aggregation1 : aggregations1) {
        System.out.println(aggregation1.toString());
    }
}
}
```

需求五：分组求平均值

计算每个球队年龄最大/最小/总/平均的球员年龄

select team, avg(age) as max_age from player group by team;

```
/**
 * 计算每个球队的年龄平均值
 */
@Test
public void avgTeamAge(){
    SearchRequestBuilder builder =
    client.prepareSearch("player").setTypes("player");

    TermsAggregationBuilder team_field =
    AggregationBuilders.terms("player_count").field("team");

    AvgAggregationBuilder age_avg =
    AggregationBuilders.avg("age_avg").field("age");

    team_field.subAggregation(age_avg);

    SearchResponse searchResponse =
    builder.addAggregation(team_field).get();

    Aggregations aggregations = searchResponse.getAggregations();
    for (Aggregation aggregation : aggregations) {
        System.out.println(aggregation.toString());
        StringTerms stringTerms = (StringTerms) aggregation;
    }
}
```

需求六：分组求和

计算每个球队球员的平均年龄，同时又要计算总年薪

```
select team, avg(age)as avg_age, sum(salary) as total_salary from player group by team;
```

```
/**
 * 统计每个球队当中的球员平均年龄，以及队员总年薪
 */
@Test
public void avgAndSum(){
    SearchRequestBuilder builder =
    client.prepareSearch("player").setTypes("player");

    TermsAggregationBuilder team_group =
    AggregationBuilders.terms("team_group").field("team");

    AvgAggregationBuilder avg_age =
    AggregationBuilders.avg("avg_age").field("age");

    SumAggregationBuilder sumMoney =
    AggregationBuilders.sum("sum_money").field("salary");

    TermsAggregationBuilder termsAggregationBuilder =
    team_group.subAggregation(avg_age).subAggregation(sumMoney);

    SearchResponse searchResponse =
    builder.addAggregation(termsAggregationBuilder).get();
    Aggregations aggregations = searchResponse.getAggregations();
    for (Aggregation aggregation : aggregations) {
        System.out.println(aggregation.toString());
    }
}
```

需求七：聚合排序

计算每个球队总年薪，并按照总年薪倒序排列

```
/**
 * 计算每个球队总年薪，并按照年薪进行排序
 * select team, sum(salary) as total_salary from player group by team
 order by total_salary desc;
 */
@Test
```

```
public void orderBySum(){
    SearchRequestBuilder builder =
client.prepareSearch("player").setTypes("player");
    TermsAggregationBuilder teamGroup =
AggregationBuilders.terms("team_group").field("team").order(BucketOrder.
count(true));
    SumAggregationBuilder sumSalary =
AggregationBuilders.sum("sum_salary").field("salary");
    TermsAggregationBuilder termsAggregationBuilder =
teamGroup.subAggregation(sumSalary);
    SearchResponse searchResponse =
builder.addAggregation(termsAggregationBuilder).get();

    Map<String, Aggregation> stringAggregationMap =
searchResponse.getAggregations().asMap();
    System.out.println(stringAggregationMap.toString());
    Aggregations aggregations = searchResponse.getAggregations();
    for (Aggregation aggregation : aggregations) {
        System.out.println(aggregation.toString());
    }
}
```

13、elasticsearch 的 sql 插件使用

对于这些复杂的查询，es 使用 javaAPI 都可以实现，但是相较于 sql 语句来说，我们更加熟悉 sql 语句，所以 es 也提供了 sql 语句的开发，让我们通过 sql 语句即可实现 ES 的查询，接下来我们就来安装并学习 sql 的插件的使用方法吧！

在 es 版本 6.3 之前都不支持 sql 语句的开发，如果需要使用 sql 语句来开发 es 的数据查询，那么我们需要手动自己安装插件，插件下载地址如下，

<https://github.com/NLPchina/elasticsearch-sql/>

但是在 6.3 版本之后，es 自带就安装了 sql 的插件，我们可以直接通过 sql 语句的方式实现 es 当中的数据查询

对于 sql 语句的使用，es 给我们提供了三种方式，接下来我们分别看看三种方式如何实现 es 数据的查询

第一种方式：通过 rest 风格实现数据的查询

第一步：使用 rest 方式向索引库当中添加数据

使用 kibana 向索引库当中添加数据

```
PUT /library/book/_bulk?refresh
{"index":{"_id": "Leviathan Wakes"}}
{"name": "Leviathan Wakes", "author": "James S.A. Corey", "release_date":
"2011-06-02", "page_count": 561}
{"index":{"_id": "Hyperion"}}
{"name": "Hyperion", "author": "Dan Simmons", "release_date":
"1989-05-26", "page_count": 482}
{"index":{"_id": "Dune"}}
{"name": "Dune", "author": "Frank Herbert", "release_date": "1965-06-01",
"page_count": 604}
```

第二步：使用 rest 风格方式查询数据

```
curl -X POST "node01:9200/_xpack/sql?format=txt" -H 'Content-Type:
application/json' -d'
{
  "query": "SELECT * FROM library WHERE release_date <
\u00272000-01-01\u0027"
}'
```

第二种方式：使用 sql 脚本的方式进入 sql 客户端进行查询

我们也可以使用 sql 脚本的方式，进入 sql 客户端，通过 sql 语句的方式实现数据的查询

第一步：node01 进入 sql 脚本客户端

node01 执行以下命令进入 sql 脚本客户端

```
cd /export/servers/es/elasticsearch-6.7.0
```

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090

```
bin/elasticsearch-sql-cli node01:9200
```

第二步：执行 sql 语句

```
sql> select * from library;
```

author release_date	name	page_count
-----+-----+-----+-----		
Dan Simmons 1989-05-26T00:00:00.000Z	Hyperion	482
James S.A. Corey 2011-06-02T00:00:00.000Z	Leviathan Wakes	561
Frank Herbert 1965-06-01T00:00:00.000Z	Dune	604

第三种方式：通过 jdbc 连接的方式

当然了，我们也可以通过 jdbc 连接的方式，通过 java 代码来实现 ES 当中数据的查询操作

官网介绍操作连接

<https://www.elastic.co/guide/en/elasticsearch/reference/6.7/sql-jdbc.html>

使用 javaAPI 访问数据库当中的数据，会产生一个错误，参见这篇文章

<https://www.cnblogs.com/hts-technology/p/9282421.html>

第一步：导入 jar 包

在我们的 maven 依赖中添加以下坐标，导入 es-sql 的 jar 包

```
<repositories>
  <repository>
    <id>elastic.co</id>
```



```
<url>https://artifacts.elastic.co/maven</url>
</repository>
</repositories>

<dependency>
    <groupId>org.elasticsearch.plugin</groupId>
    <artifactId>x-pack-sql-jdbc</artifactId>
    <version>6.7.0</version>
</dependency>
```

第二步：开发 java 代码，实现查询

通过 java 代码，使用 jdbc 连接 es 服务器，然后查询数据

```
@Test
public void esJdbc() throws SQLException {
    EsDataSource dataSource = new EsDataSource();
    String address = "jdbc:es://http://node01:9200" ;
    dataSource.setUrl(address);
    Properties connectionProperties = new Properties();
    dataSource.setProperties(connectionProperties);
    Connection connection = dataSource.getConnection();
    Statement statement = connection.createStatement();
    ResultSet resultSet = statement.executeQuery("select * from
library");
    while(resultSet.next()){
        String string = resultSet.getString(0);
        String string1 = resultSet.getString(1);
    }
}
```

```
        int anInt = resultSet.getInt(2);
        String string2 = resultSet.getString(4);
        System.out.println(string + "\t" + string1 + "\t" + anInt +
"\t" + string2);
    }
    connection.close();
}
```

14、LogStash 介绍及安装

官网：

<https://www.elastic.co/guide/en/logstash/current/index.html>

1、介绍

logstash 就是一个具备实时数据传输能力的管道，负责将数据信息从管道的输入端传输到管道的输出端；与此同时这根管道还可以让你根据自己的需求在中间加上滤网，Logstash 提供很多功能强大的滤网以满足你的各种应用场景。是一个 input | filter | output 的数据流。

2 、node01 机器安装 LogStash

下载 logstash 并上传到第一台服务器的/home/es 路径下，然后进行解压

下载安装包---可以直接将已经下载好的安装包上传到/home/es 路径下即可

```
cd /home/es
```

```
wget https://artifacts.elastic.co/downloads/logstash/logstash-6.7.0.tar.gz
```

```
# 解压
```

```
tar -zxvf logstash-6.7.0.tar.gz -C /export/servers/es/
```

3、Input 插件

1、stdin 标准输入和 stdout 标准输出

使用标准的输入与输出组件，实现将我们的数据从控制台输入，从控制台输出

```
cd /export/servers/es/logstash-6.7.0/
```

```
bin/logstash -e 'input {stdin {}} output {stdout {codec=>rubydebug}}'
```

```
[es@node01 logstash]$ bin/logstash -e 'input { stdin { } } output { stdout { codec => rubydebug } }'
hello
Sending Logstash's logs to /export/servers/logstash/logs which is now configured via log4j2.properties
(2018-10-13T16:33:10.663)[INFO] [logstash.modules.scaffold] Initializing module {:module_name=>"netflow", :directory=>"/export/servers/logstash/modules/netflow/configuration"}
(2018-10-13T16:33:10.666)[INFO] [logstash.modules.scaffold] Initializing module {:module_name=>"fb_apache", :directory=>"/export/servers/logstash/modules/fb_apache/configuration"}
(2018-10-13T16:33:10.905)[WARN] [logstash.config.source.multilocal] Ignoring the 'pipelines.yml' file because modules or command line options are specified
(2018-10-13T16:33:11.236)[INFO] [logstash.agent] Successfully started Logstash API endpoint {:port=>9600}
(2018-10-13T16:33:13.045)[INFO] [logstash.pipeline] Starting pipeline {:pipeline_id=>"main", :pipeline.workers=>2, :pipeline.batch.size=>125, :pipeline.batch.delay=>5, :pipeline.max_inflight=>250, :thread=>#<Thread:0x7e46f496@/export/servers/logstash/logstash-core/lib/logstash/pipeline.rb:290 run>}
(2018-10-13T16:33:13.072)[INFO] [logstash.pipeline] Pipeline started {:pipeline.id=>"main"}
The stdin plugin is now waiting for input:
(2018-10-13T16:33:13.110)[INFO] [logstash.agent] ] Pipelines running (:count=>1, :pipelines=>["main"])

{"@version" => "1",
 "host" => "node01",
 "@timestamp" => 2018-10-13T08:33:13.126Z,
 "message" => "hello"}
```

```
{
  "@version" => "1",
  "host" => "node01",
  "@timestamp" => 2018-10-13T08:33:13.126Z,
  "message" => "hello"
}
```

2、监控日志文件变化

Logstash 使用一个名叫 *FileWatch* 的 Ruby Gem 库来监听文件变化。这个库支持 glob 展开文件路径，而且会记录一个叫 *.sincedb* 的数据库文件来跟踪被监听的日志文件的当前读取位置。所以，不要担心 logstash 会漏过你的数据。

编写脚本

```
cd /export/servers/es/logstash-6.7.0/config
```

```
vim monitor_file.conf
```

#输入一下信息

```
input{
  file{
    path => "/export/servers/es/datas/tomcat.log"
    type => "log"
    start_position => "beginning"
  }
}
```

```
}  
output{  
    stdout{  
        codec=>rubydebug  
    }  
}
```

检查配置文件是否可用

```
cd /export/servers/es/logstash-6.7.0/  
bin/logstash -f  
/export/servers/es/logstash-6.7.0/config/monitor_file.conf -t
```

成功会出现一下信息：

```
Config Validation Result: OK. Exiting Logstash
```

启动服务

```
cd /export/servers/es/logstash-6.7.0  
bin/logstash -f  
/export/servers/es/logstash-6.7.0/config/monitor_file.conf
```

发送数据

新开 xshell 窗口通过以下命令发送数据

```
mkdir -p /export/servers/es/datas  
echo "hello logstash" >> /export/servers/es/datas/tomcat.log
```

其它参数说明

Path=>表示监控的文件路径

Type=>给类型打标记，用来区分不同的文件类型。

Start_postion=>从哪里开始记录文件，默认是从结尾开始标记，要是你从头导入一个文件就把改成"beginning".

discover_interval=>多久去监听 path 下是否有文件，默认是 15s

exclude=>排除什么文件

close_older=>一个已经监听中的文件，如果超过这个值的时间内没有更新内容，就关闭监听它的文件句柄。默认是 3600 秒，即一个小时。

sincedb_path=>监控库存放位置(默认的读取文件信息记录在哪个文件中)。默认在: /data/plugins/inputs/file。

sincedb_write_interval=> logstash 每隔多久写一次 sincedb 文件，默认是 15 秒。

stat_interval=>logstash 每隔多久检查一次被监听文件状态(是否有更新)，默认是 1 秒。

3、jdbc 插件

jdbc 插件允许我们采集某张数据库表当中的数据到我们的 logstash 当中来

第一步：编写脚本

开发脚本配置文件

```
cd /export/servers/es/logstash-6.7.0/config
```

```
vim jdbc.conf
```

```
input {
  jdbc {
    jdbc_driver_library =>
"/export/servers/es/datas/mysql-connector-java-5.1.38.jar"
    jdbc_driver_class => "com.mysql.jdbc.Driver"
    jdbc_connection_string => "jdbc:mysql://192.168.227.30:3306/userdb"
    jdbc_user => "root"
```

```
jdbc_password => "123456"
use_column_value => true
tracking_column => "create_time"
# parameters => { "favorite_artist" => "Beethoven" }
schedule => "* * * * *"
statement => "select * from emp where create_time > :sql_last_value ;"
}
}
output{
  stdout{
    codec=>rubydebug
  }
}
```

第二步：上传 mysql 连接驱动包到指定路劲

将我们 mysql 的连接驱动包上传到我们指定的/home/es/路径下

```
[es@node01 ~]$ pwd
/home/es
[es@node01 ~]$ ll
total 203632
-rw-r--r-- 1 es es 28017602 May 15 12:56 elasticsearch-6.0.0.tar.gz
-rw-r--r-- 1 es es 4502425 Sep 29 16:17 elasticsearch-analysis-ik-6.0.0.zip
-rw-r--r-- 1 es es 62681537 May 15 12:58 kibana-6.0.0-linux-x86_64.tar.gz
-rw-r--r-- 1 es es 112316625 May 15 12:59 logstash-6.0.0.tar.gz
-rw-rw-r-- 1 es es 983911 Nov 7 18:58 mysql-connector-java-5.1.38.jar
[es@node01 ~]$
```

第三步：检查配置文件是否可用

```
cd /export/servers/es/logstash-6.7.0/
bin/logstash -f config/jdbc.conf -t
```

通过之后

```
Config Validation Result: OK. Exiting Logstash
```

第四步：启动服务

通过以下命令启动 logstash

```
cd /export/servers/es/logstash-6.7.0  
bin/logstash -f config/jdbc.conf
```

第五步：数据库当中添加数据

在我们的数据库当中手动随便插入数据，发现我们的 logstash 可以进行收集

4 syslog 插件

syslog 机制负责记录内核和应用程序产生的日志信息，管理员可以通过查看日志记录，来掌握系统状况

默认系统已经安装了 rsyslog.直接启动即可

编写脚本

```
cd /export/servers/es/logstash-6.7.0/config  
vim syslog.conf
```

```
input{  
  tcp{  
    port=> 6789  
    type=> syslog  
  }  
  udp{  
    port=> 6789  
    type=> syslog  
  }  
}
```

```
filter{
  if [type] == "syslog" {
    grok {
      match => { "message" =>
"%{SYSLOGTIMESTAMP:syslog_timestamp} %{SYSLOGHOST:syslog_hostname}
%{DATA:syslog_program}(?:\[ %{POSINT:syslog_pid}\])?:
%{GREEDYDATA:syslog_message}" }
      add_field => [ "received_at", "%{@timestamp}" ]
      add_field => [ "received_from", "%{host}" ]
    }
    date {
      match => [ "syslog_timestamp", "MMM d HH:mm:ss", "MMM
dd HH:mm:ss" ]
    }
  }
}

output{
  stdout{
    codec=> rubydebug
  }
}
```

检查配置文件是否可用

```
cd /export/servers/es/logstash-6.7.0
bin/logstash -f /export/servers/es/logstash-6.7.0/config/syslog.conf -t
```

启动服务

执行以下命令启动 logstash 服务

```
cd /export/servers/es/logstash-6.7.0
bin/logstash -f /export/servers/es/logstash-6.7.0/config/syslog.conf
```

发送数据

修改系统日志配置文件

```
sudo vim /etc/rsyslog.conf
```


添加一行以下配置

```
*.* @@node01:6789
```

```
$ModLoad imuxsock # provides support for local system logging (e.g.
$ModLoad imklog   # provides kernel logging support (previously do
#$ModLoad immark  # provides --MARK-- message capability

# Provides UDP syslog reception
#$ModLoad imudp
#$UDPServerRun 514

# Provides TCP syslog reception
#$ModLoad imtcp
#$InputTCPServerRun 514
*.* @@node01:6789
#### GLOBAL DIRECTIVES ####

# Use default timestamp format
```

重启系统日志服务

```
sudo /etc/init.d/rsyslog restart
```

其它参数说明

在 logstash 中的 grok 是正则表达式，用来解析当前数据

原始数据其实是：

```
Dec 23 12:11:43 louis postfix/smtpd[31499]: connect from unknown[95.75.9
3.154]
Jun 05 08:00:00 louis named[16000]: client 199.48.164.7#64817: query (ca
che) 'amsterdamboothuren.com/MX/IN' denied
Jun 05 08:10:00 louis CRON[620]: (www-data) CMD (php /usr/share/cacti/si
te/poller.php >/dev/null 2>/var/log/cacti/poller-error.log)
Jun 05 08:05:06 louis rsyslogd: [origin software="rsyslogd" swVersion="4.
2.0" x-pid="2253" x-info="http://www.rsyslog.com"] rsyslogd was HUPed,
type 'lightweight'.
```

4、filter 插件

Logstash 之所以强悍的主要原因是 filter 插件；通过过滤器的各种组合可以得到我们想要的结构化数据。

1、grok 正则表达式

grok 正则表达式是 logstash 非常重要的一个环节；可以通过 grok 非常方便的将数据拆分和索引

语法格式：

(?<name>pattern)

? <name>表示要取出里面的值，pattern 就是正则表达式

1、收集控制台输入数据，采集日期时间出来

第一步：开发配置文件

```
cd /export/servers/es/logstash-6.7.0/config/  
vim filter.conf
```

```
input {stdin{}} filter {  
    grok {  
        match => {  
"message" => "(?<date>\d+\.\d+)\s+"  
        }  
    }  
}  
output {stdout{codec => rubydebug}}
```

第二步：启动 logstash 服务

```
cd /export/servers/es/logstash-6.7.0/  
bin/logstash -f config/filter.conf
```

第三步：控制台输入文字

5.20 今天天气还不错

```
5.20 今天天气还不错
{
  "@version" => "1",
  "host" => "node01.hadoop.com",
  "date" => "5.20",
  "@timestamp" => 2018-11-08T01:45:16.950Z,
  "message" => "5.20 今天天气还不错"
}
```

将我们的日期时间按照正则匹配取出来放到了date这个字段

2、使用 grok 收集 nginx 日志数据

nginx 一般打印出来的日志格式如下

```
36.157.150.1 - - [05/Nov/2018:12:59:28 +0800] "GET
/phpmyadmin_8c1019c9c0de7a0f/js/get_scripts.js.php?scripts%5B%5D=jqu
ery/jquery-1.11.1.min.js&scripts%5B%5D=sprintf.js&scripts%5B%5D=ajax.j
s&scripts%5B%5D=keyhandler.js&scripts%5B%5D=query/jquery-ui-1.11.2.
min.js&scripts%5B%5D=query/jquery.cookie.js&scripts%5B%5D=query/jq
query.mousewheel.js&scripts%5B%5D=query/jquery.event.drag-2.2.js&scri
pts%5B%5D=query/jquery-ui-timepicker-addon.js&scripts%5B%5D=query
/jquery.ba-hashchange-1.3.js HTTP/1.1" 200 139613 "-" "Mozilla/5.0
(Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/45.0.2454.101 Safari/537.36"
```

这种日志是非格式化的，通常，我们获取到日志后，还要使用mapreduce 或者spark 做一下清洗操作，

就是将非格式化日志编程格式化日志；

在清洗的时候，如果日志的数据量比较大，那么也是需要花费一定的时间的；

所以可以使用logstash 的grok 功能，将nginx 的非格式化数据采集成格式化数据：

第一步：安装 grok 插件

第一种方式安装：在线安装（强烈不推荐）

在线安装 grok 插件

```
cd /export/servers/es/logstash-6.7.0/  
vim Gemfile
```

```
#source 'https://rubygems.org'      # 将这个镜像源注释掉  
source https://gems.ruby-china.com/ # 配置成中国的这个镜像源
```

准备在线安装

```
cd /export/servers/es/logstash-6.7.0/  
bin/logstash-plugin install logstash-filter-grok
```

第二种安装方式，直接使用我已经下载好的安装包进行本地安装（强烈推荐使用）

上传我们的压缩包 logstash-filter-grok-4.0.4.zip 上传到
/export/servers/es/logstash-6.7.0 这个路径下面

然后准备执行本地安装

```
cd /export/servers/es/logstash-6.7.0/  
bin/logstash-plugin install  
file:///export/servers/es/logstash-6.7.0/logstash-filter-grok-4.0.4.zip
```

安装成功之后查看 logstash 的插件列表

```
cd /export/servers/es/logstash-6.7.0/  
bin/logstash-plugin list
```

第二步：开发 logstash 的配置文件

定义 logstash 的配置文件如下，我们从控制台输入 nginx 的日志数据，然后经过 filter 的过滤，将我们的日志文件转换成为标准的数据格式

```
cd /export/servers/es/logstash-6.7.0/config  
vim monitor_nginx.conf
```

```
input {stdin{}}  
filter {  
  grok {  
    match => {  
      "message" => "%{IPORHOST:clientip} \- \- \[%{HTTPDATE:time_local}\]  
\"(?:%{WORD:method} %{NOTSPACE:request}(?:HTTP/%{NUMBER:httpversion})?|%{DATA:rawrequest})\" %{NUMBER:status} %{NUMBER:body_bytes_sent} %{QS:http_referer} %{QS:agent}"  
    }  
  }  
}  
output {stdout{codec => rubydebug}}
```

第三步：启动 logstash

执行以下命令启动 logstash

```
cd /export/servers/es/logstash-6.7.0  
bin/logstash -f  
/export/servers/es/logstash-6.7.0/config/monitor_nginx.conf
```

第四步：从控制台输入 nginx 日志文件数据

输入第一条数据

```
36.157.150.1 - - [05/Nov/2018:12:59:27 +0800] "GET /phpmyadmin_8c1019c9c0de7a0f/js/messages.php?lang=zh_CN&db=&collation_connection=utf8_unicode_ci&token=6a44d72481633c90bffcfd42f11e25a1 HTTP/1.1" 200 8131 "-" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.101 Safari/537.36"
```

输入第二条数据

```
36.157.150.1 - - [05/Nov/2018:12:59:28 +0800] "GET /phpmyadmin_8c1019c9c0de7a0f/js/get_scripts.js.php?scripts%5B%5D=jquery/jquery-1.11.1.min.js&scripts%5B%5D=sprintf.js&scripts%5B%5D=ajax.js&scripts%5B%5D=keyhandler.js&scripts%5B%5D=jquery/jquery-ui-1.11.2.min.js&scripts%5B%5D=jquery/jquery.cookie.js&scripts%5B%5D=jquery/jquery.mousewheel.js&scripts%5B%5D=jquery/jquery.event.drag-2.2.js&scripts%5B%5D=jquery/jquery-ui-timepicker-addon.js&scripts%5B%5D=jquery/jquery.ba-hashchange-1.3.js HTTP/1.1" 200 139613 "-" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.101 Safari/537.36"
```

5、 Output 插件

1 标准输出到控制台

将我们收集的数据直接打印到控制台

```
output {
  stdout {
    codec => rubydebug
  }
}
```

```
bin/logstash -e 'input{stdin{}}output{stdout{codec=>rubydebug}}'
```

```
[es@node01 logstash]$ bin/logstash -e 'input{stdin{}}output{stdout{codec=>rubydebug}}'
hello
```

2 将采集数据保存到 file 文件中

logstash 也可以将收集到的数据写入到文件当中去永久保存，接下来我们来看看 logstash 如何配置以实现将数据写入到文件当中

第一步：开发 logstash 的配置文件

```
cd /export/servers/es/logstash-6.7.0/config  
vim output_file.conf
```

```
input {stdin{}}  
output {  
  file {  
    path =>  
    "/export/servers/es/datas/%{+YYYY-MM-dd}-%{host}.txt"  
    codec => Line {  
      format => "%{message}"  
    }  
    flush_interval => 0  
  }  
}
```

第二步：检测配置文件并启动 logstash 服务

```
cd /export/servers/es/logstash-6.7.0
```

检测配置文件是否正确

```
bin/logstash -f config/output_file.conf -t
```

启动服务，然后从控制台输入一些数据

```
bin/logstash -f config/output_file.conf
```

查看文件写入的内容

```
cd /export/servers/es/datas
```

```
more 2018-11-08-node01.hadoop.com.txt
```

```
[es@node01 datas]$ cd /export/servers/es/datas
[es@node01 datas]$ ll
total 8
-rw-rw-r-- 1 es es 179 Nov  8 19:23 2018-11-08-node01.hadoop.com.txt
-rw-rw-r-- 1 es es  15 Nov  7 17:33 tomcat.log
[es@node01 datas]$ more 2018-11-08-node01.hadoop.com.txt
helloabc
xxxabc
bbbxxx
abcddd
gogogog
gogogoohleohleohle
[es@node01 datas]$
```

3 将采集数据保存到 elasticsearch

第一步：开发 logstash 的配置文件

```
cd /export/servers/es/logstash-6.7.0/config
```

```
vim output_es.conf
```

```
input {stdin{}}
output {
  elasticsearch {
    hosts => ["node01:9200"]
    index => "logstash-%{+YYYY.MM.dd}"
  }
}
```

这个 **index** 是保存到 **elasticsearch** 上的索引名称，如何命名特别重要，因为我们很可能后续根据某些需求做查询，所以最好带时间，因为我们在中间加上 **type**，就代表不同的业务，这样我们在查询当天数据的时候，就可以根据类型+时间做范围查询

第二步：检测配置文件并启动 logstash

检测配置文件是否正确

```
cd /export/servers/es/logstash-6.7.0
```

```
bin/logstash -f config/output_es.conf -t
```

启动 logstash

```
bin/logstash -f config/output_es.conf
```

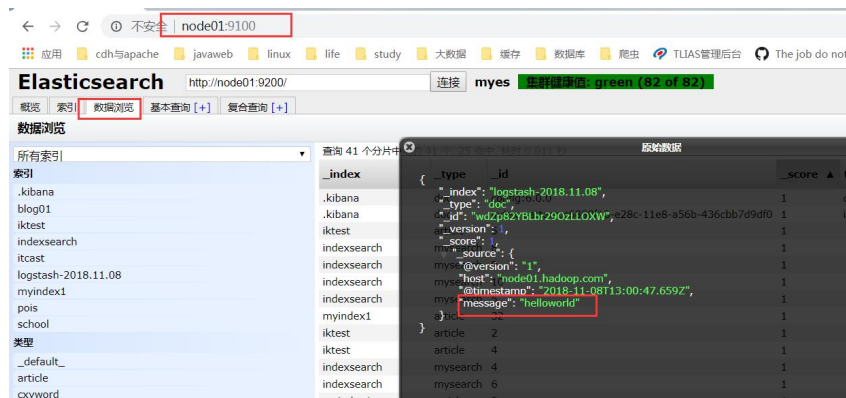
```
The stdin plugin is now waiting for input:
[2018-10-13T18:36:44,638][INFO ][logstash.agent          ] Pipelines running {:count=>1, :
ipipelines=>["main"]}
hello hadoop
hello storm
```

第三步：es 当中查看数据

访问

<http://node01:9100/>

查看 es 当中的数据



The screenshot shows the Kibana interface for Elasticsearch. The 'Discover' tab is active, displaying a list of documents. The first document is highlighted, and its 'message' field is expanded, showing the value 'helloworld'. The interface includes a sidebar with various tools and a main area for document listing and details.

注意：

更多的 input, filter, output 组件参见以下链接

<https://www.elastic.co/guide/en/logstash/current/index.html>

15、kibana 报表展示

官网对于 kibana 的基本简介

<https://www.elastic.co/guide/cn/kibana/current/index.html>

kibana 是一个强大的报表展示工具，可以通过 kibana 自定义我们的数据报表展示，实现我们的数据的各种图表查看

我们可以通过官网提供的数据集来实现我们的数据的报表展示

第一步：下载数据集

下载账户数据集

<https://download.elastic.co/demos/kibana/gettingstarted/accounts.zip>

下载日志数据集

<https://download.elastic.co/demos/kibana/gettingstarted/logs.jsonl.gz>

第二步：上传我们的数据集并解压

将我们以上下载的数据集全部上传到 node01 服务器的/home/es 路径下

```
[es@node01 ~]$ ll
total 198820
-rw-r--r-- 1 es es 244848 Mar 6 2014 accounts.json
-rw-r--r-- 1 es es 28017602 May 15 2018 elasticsearch-6.0.0.tar.gz
-rw-r--r-- 1 es es 4502425 Sep 29 16:17 elasticsearch-analysis-ik-6.0.0.zip
-rw-r--r-- 1 es es 37732586 Dec 4 21:47 elasticsearch-head-compile-after.tar.gz
-rw-r--r-- 1 es es 62681537 May 15 2018 kibana-6.0.0-linux-x86_64.tar.gz
-rw-r--r-- 1 es es 53347384 Jun 25 2015 logs.json
-rw-r--r-- 1 es es 17044698 Nov 5 17:20 node-v8.1.0-linux-x64.tar.gz
[es@node01 ~]$
```

第三步：创建对应的索引库

创建我们的索引库，然后加载数据

第一个索引库：日志数据集数据索引库

创建第一个索引库，将我们第一个索引库日志数据也创建好

我们这里实际上按照日期，创建了三个索引库，都是用于加载我们的日志数据

```
PUT /logstash-2015.05.18
{
  "mappings": {
    "log": {
      "properties": {
        "geo": {
          "properties": {
            "coordinates": {
              "type": "geo_point"
            }
          }
        }
      }
    }
  }
}

PUT /logstash-2015.05.19
{
  "mappings": {
    "log": {
      "properties": {
        "geo": {
```

PUT /logstash-2015.05.20

第四步：加载示例数据到我们的索引库当中来

直接在 node01 的 /home/es 路径下执行以下命令来加载我们的示例数据到索引库当中来

node01 机器上面执行以下命令

```
cd /home/es  
  
curl -H 'Content-Type: application/x-ndjson' -XPOST  
'node01:9200/bank/account/_bulk?pretty' --data-binary  
@accounts.json  
  
curl -H 'Content-Type: application/x-ndjson' -XPOST  
'node01:9200/_bulk?pretty' --data-binary @logs.json
```

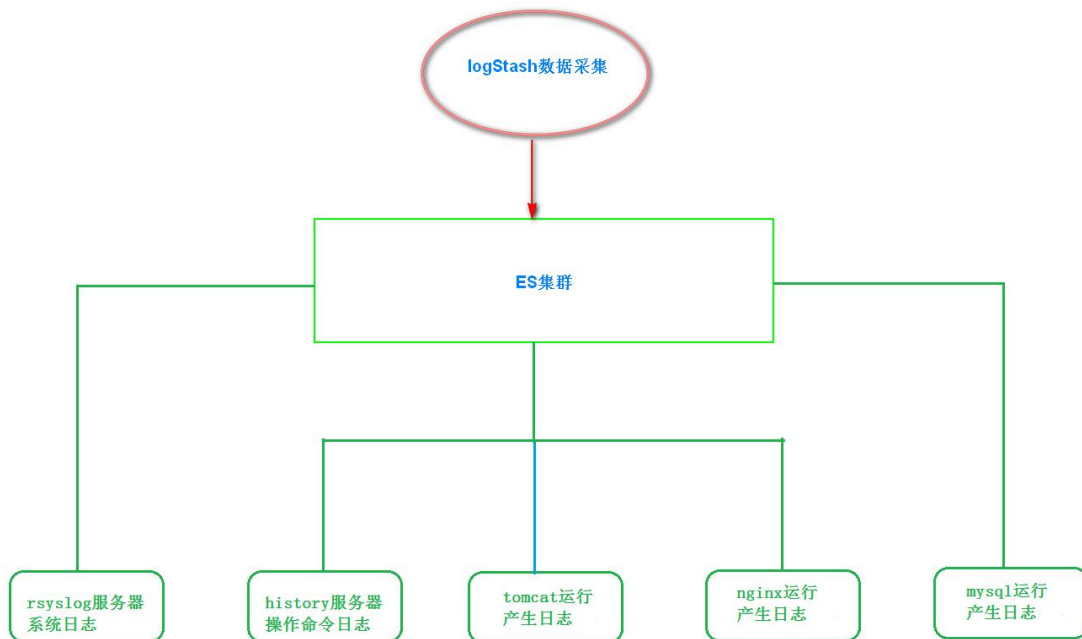
第五步：实现数据的报表展示

使用饼图来展示银行数据，使用地图来实现日志数据展示

16、logStash+ES 综合案例

ELK Stack 全量日志查询

在实际工作当中，我们经常会对服务器当中产生的各种日志比较感兴趣，因为产生的日志可以很好的说明我们的服务器的工作状态是否正常，日志的也可以供我们排查各种错误等。所以很多时候我们都会收集各种日志进行汇总，方便我们统一的查看，有了 ELK 技术栈之后，我们就可以很轻松方便的使用 logstash 来进行收集我们的日志数据，然后将数据存储到 ES 当中，然后通过 kibana 的可视化工具来查看我们的日志数据了



1、 采集服务器运行产生的日志

1.1、rsyslog 的基本介绍

每台 linux 服务器运行的时候，系统都会产生一些日志出来，我们可以收集这些日志以便于我们查看 linux 服务器运行的状况等

Rsyslog 是 CentOS6.X 自带的一款系统日志工具：

具有以下多种特性

- 1.支持多线程
- 2.支持 TCP,SSL,TLS,RELP 等协议
- 3.支持将日志写入 MySQL, PGSQL, Oracle 等多种关系型数据中
- 4.拥有强大的过滤器，可实现过滤系统信息中的任意部分
- 5.可以自定义日志输出格式

接下来我们来看如何通过 rsyslog 来收集我们的系统日志

Rsyslog 配置文件介绍：

/etc/rsyslog.conf 文件：

*.info;mail.none;authpriv.none;cron.none	/var/log/messages.	各类型日志存放位置
cron.*	/var/log/cron	具体日志存放的位置
authpriv.*	/var/log/secure	认证授权认证
mail.*	-/var/log/maillog	邮件日志
cron.*	/var/log/cron	任务计划相关日志
kern		内核相关日志
lpr		打印
mark(syslog)		rsyslog 服务内部的信息,时间标识
news		新闻组
user		用户程序产生的相关信息
uucp		协议
local 0~7		用户自定义日志级别

日志级别：

rsyslog 共有7 种日志级别，数字代号从 0~7。具体的意义如下所示：

0 debug - 有调式信息的，日志信息最多

1 info 一般信息的日志，最常用

2 notice - 最具有重要性的普通条件的信息

3 warning - 警告级别

4 err - 错误级别，阻止某个功能或者模块不能正常工作的信息

5 crit - 严重级别，阻止整个系统或者整个软件不能正常工作的信息

6 alert - 需要立刻修改的信息

7 emerg - 内核崩溃等严重信息

本项目中，将日志级别调整成 3，并将日志信息发送至 node01 服务器的 6789 这个端口

1.2、修改 rsyslog 的配置文件

我们修改 node01 服务器的 rsyslog 的配置,

```
sudo vim /etc/rsyslog.conf
```

添加以下三行配置

```
local3.* /var/log/boot.log
*.warning /var/log/warning_Log
*. * @@node01:6789
```

```
$ModLoad imuxsock # provides support for local system logging (e.g. via logger command)
$ModLoad imklog    # provides kernel logging support (previously done by rklogd)
#$ModLoad immark   # provides --MARK-- message capability
```

```
local3.* /var/log/boot.log
*.warning /var/log/warning_Log
*. * @@node01:6789
```

添加以下这三行配置

1.3、重启 linux 服务器的 rsyslog 服务

执行以下命令重启 rsyslog 服务

```
sudo /etc/init.d/rsyslog restart
```

1.4、开发 logstash 的配置文件，收集 rsyslog 日志

切换到 logstash 的配置文件目录下，开发 logstash 的配置文件

```
cd /export/servers/es/logstash-6.7.0/config
```

```
vim rsyslog.conf
```

```
input {
  tcp {
    port => "6789"    #监控 6789 端口
    type => "rsyslog" #日志类型是 rsyslog
  }
}
```



```
}  
}  
filter {  
  if [type] == "rsyslog" {    # 做一次判断，只要从 6789 端口过来的 rsyslog  
    日志  
    grok { # 通过正则表达式，取出想要的字段  
      match => { "message" =>  
"%{SYSLOGTIMESTAMP:syslog_timestamp} %{SYSLOGHOST:syslog_hostna  
me} %{DATA:syslog_program}(?:\[ %{POSINT:syslog_pid}\])?: %{GREEDYDAT  
A:syslog_message}" }  
      add_field => [ "received_at", "%{@timestamp}" ]  
      add_field => [ "received_from", "%{host}" ]  
    }  
    date {  
      match => [ "syslog_timestamp", "MMM  d HH:mm:ss", "MMM dd  
HH:mm:ss" ] #将系统的日志格式化标准国际化时间  
    }  
  }  
}  
output{    #将日志打入 elasticsearch  
  if [type] == "rsyslog"{  
    stdout{codec=>rubydebug}  
    elasticsearch {  
      action => "index"  
      hosts    => "node01:9200"  
      index    => "logstash-%{type}-%{+yyyy.MM.dd}"  
    }  
  }  
}
```

1.5、启动 logstash 服务

执行以下命令启动 logstash 服务

```
cd /export/servers/es/logstash-6.7.0  
bin/logstash -f config/rsyslog.conf
```

2、采集服务器用户操作所有历史日志

用户在命令行环境下的操作日志都会被系统记录下来；比如我们输入 history 命令，都会展示出每一个用户输入过的命令；

.bash_history 文件，这个日志格式我们可以定义成我们需要显示的内容，方便我们排查或者做入侵检查的时候；

自定义日志格式：

```
HISTFILESIZE=4000      #保存命令的记录总数  
HISTSIZE=4000         # history 命令输出的记录数  
HISTTIMEFORMAT='%F %T' #输出时间格式  
export HISTTIMEFORMAT. #自定义日志输出格式，也就是取出我们想要的  
                        字段，以 json 的形式  
HISTTIMEFORMAT 修改线上的相关格式  
PROMPT_COMMAND 实时记录历史命令（一般用在存储 history 命令道文件中）
```

第一步：定义日志格式

修改/etc/bashrc 配置文件，然后添加以下配置

```
sudo vim /etc/bashrc
```

```
HISTDIR='/var/log/command.log'  
if [ ! -f $HISTDIR ];then  
touch $HISTDIR  
chmod 666 $HISTDIR  
fi
```

```
export
HISTTIMEFORMAT="{\"TIME\": \"%F%T\", \"HOSTNAME\": \"$HOSTNAME\", \"
LI\": \"$(who am i 2>/dev/null| awk '{print $NF}'|sed
-e's/[]//g')\", \"LOGIN_USER\": \"$(who am i|awk
'{print$1}')\", \"CHECK_USER\": \"${USER}\", \"CMD\": \"\"

export PROMPT_COMMAND='history 1|tail -1|sed "s/^[ ]+[0-9]\+ //"|sed
"s/$\"/">>/var/log/command.log'
```

使配置修改立即生效

这个命令必须使用 **root** 用户来执行

```
export PROMPT_COMMAND='history >> /var/log/command.log'

source /etc/bashrc
```

第二步：开发 logstash 的配置文件

继续开发我们 logstash 的配置文件

```
cd /export/servers/es/logstash-6.7.0/config

vim history.conf
```

```
input {
  file {
    path => ["/var/log/command.log"]
    type => "command"
    codec => "json"
  }
}

output{
  if [type] == "command"{
    stdout{codec=>rubydebug}
    elasticsearch {
      hosts => "node01:9200"
```

```
index => "history-%{+yyyy.MM.dd}"  
  
}  
  
}  
  
}
```

第三步：启动 logstash

启动 logstash:

```
cd /export/servers/es/logstash-6.7.0  
bin/logstash -f config/history.conf
```

执行 source，让环境变量立即生效

```
sudo source /etc/bashrc
```

node01 服务器任意目录执行任意命令，然后去 9100 页面，查看是否已经把 history 日志灌入 elasticsearch

3、采集 nginx 日志到 es 当中

第一步：上传日志文件

node01 机器创建文件夹，将我们的 nginx 的日志都上传到

```
/export/servers/es/esdatas
```

```
mkdir -p /export/servers/es/esdatas
```

第二步：开发 logstash 的配置文件

```
cd /export/servers/es/logstash-6.7.0/config  
vim nginxlog.conf
```

```
input{  
  file {  
    path => "/export/servers/es/esdatas/access.log"  
    type => "access.log"  
    start_position => "beginning"
```

```
}  
}  
filter {  
  grok {  
    match => {  
      "message" => "%{IPORHOST:clientip} \- \-  
\\[%{HTTPDATE:time_local}]\n  
\\(?:%{WORD:method} %{NOTSPACE:request}(?:HTTP/%{NUMBER:httpversi  
on})?|%{DATA:rawrequest})\" %{NUMBER:status} %{NUMBER:body_bytes_se  
nt} %{QS:http_referer}"  
    }  
  }  
}  
output {  
  stdout{codec=>rubydebug}  
  elasticsearch {  
    action => "index"  
    hosts => "node01:9200"  
    index => "nginxes"  
  }  
}
```

第三步：启动 logstash 并查看 es 当中的数据

执行以下命令启动 logstash

```
cd /export/servers/es/logstash-6.7.0  
bin/logstash -f /export/servers/es/logstash-6.7.0/config/nginxlog.conf
```

17、es 整合 Hbase 实现二级索引

需求：解决海量数据的存储，并且能够实现海量数据的秒级查询。

实际生产中，一篇文章要分成标题和正文；但是正文的量是比较大的，那么我们一般会在 **es** 中存储标题，在 **hbase** 中存储正文（**hbase** 本身就是做海量数据的存储）；这样通过 **es** 的倒排索引列表检索到关键词的文档 **id**，然后根据文档 **id** 在 **hbase** 中查询出具体的正文。

1、存储设计

分析，数据哪些字段需要构建索引： 文章数据（**id**、**title**、**author**、**describe**、**content**）

字段名称	是否需要索引	是否需要存储
Id	默认索引	默认存储
Title	需要	需要
Author	看需求	看需求
Dscribe	需要	存储
Content	看需求（高精度查询，是需要的）	看需求
Time	需要	需要

2、索引库设计

```
PUT /articles
{
  "settings":{
    "number_of_shards":3,
    "number_of_replicas":1,
    "analysis" : {
      "analyzer" : {
        "ik" : {
          "tokenizer" : "ik_max_word"
```

```
    }  
  }  
}  
,  
"mappings":{  
  "article":{  
    "dynamic":"strict",  
    "_source": {  
      "includes": [  
        "id","title","from","readCounts","times"  
      ],  
      "excludes": [  
        "content"  
      ]  
    },  
    "properties":{  
      "id":{"type": "keyword", "store": true},  
      "title":{"type": "text","store": true,"index" :  
true,"analyzer": "ik_max_word"},  
      "from":{"type": "keyword","store": true},  
      "readCounts":{"type": "integer","store": true},  
      "content":{"type": "text","store": false,"index": false},  
      "times": {"type": "keyword", "index": false}  
    }  
  }  
}  
}
```

3、导入 jar 包

创建 maven 工程并导入 jar 包

```
<dependencies>
<!--解析 excel 文件-->
    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi-ooxml-schemas</artifactId>
        <version>3.8</version>
    </dependency>
    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi-ooxml</artifactId>
        <version>3.8</version>
    </dependency>
    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi</artifactId>
        <version>3.8</version>
    </dependency>
    <dependency>
        <groupId>org.elasticsearch.client</groupId>
        <artifactId>transport</artifactId>
        <version>6.7.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
```



```
        <version>2.9.1</version>
    </dependency>
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.8.2</version>
    </dependency>
</dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>2.7.5</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.hbase/hbase-client -->
<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-client</artifactId>
    <version>2.0.0</version>
</dependency>
<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-server</artifactId>
    <version>2.0.0</version>
</dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.0</version>
```

```
<configuration>
    <source>1.8</source>
    <target>1.8</target>
    <encoding>UTF-8</encoding>
    <!--      <verbal>true</verbal>-->
</configuration>
</plugin>
</plugins>
</build>
```

4、代码开发

定义 Article 实体类(javabean)

```
public class Article {
    private String id;
    private String title;
    private String from;
    private String times;
    private String readCounts;
    private String content;

    public Article() {
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Article(String id, String title, String from, String times, String
readCounts, String content) {
        this.id = id;
        this.title = title;
        this.from = from;
        this.times = times;

        this.readCounts = readCounts;
        this.content = content;
    }

    public String getTitle() {
        return title;
    }
}
```

```
}

public void setTitle(String title) {
    this.title = title;
}

public String getFrom() {
    return from;
}

public void setFrom(String from) {
    this.from = from;
}

public String getTimes() {
    return times;
}

public void setTimes(String times) {
    this.times = times;
}

public String getReadCounts() {
    return readCounts;
}

public void setReadCounts(String readCounts) {
    this.readCounts = readCounts;
}

public String getContent() {
    return content;
}

public void setContent(String content) {
    this.content = content;
}
}
```

定义 excel 解析工具类

```
public class ExcelUtil {

    //读取 excel，将文件内容打印出来
    public static void main(String[] args) throws IOException {
        List<Article> excelInfo = getExcelInfo();
    }
}
```

北京市昌平区建材城西路金燕龙办公楼一层 电话：400-618-9090

```
}

    public static List<Article> getExcelInfo() throws IOException {
        FileInputStream fileInputStream = new FileInputStream("F:\\传智播客大数据离线阶段课程资料\\ELK 文档资料教案\\excel 数据集\\baijia.xlsx");
        //获取我们解析 excel 表格的对象
        XSSFWorkbook xssfSheets = new XSSFWorkbook(fileInputStream);
        //获取 excel 的第一个 sheet 页
        XSSFSheet sheetAt = xssfSheets.getSheetAt(0);
        //获取我们 sheet 页的最后一行的数字之，说白了就是看这个 excel 一共有多少行
        int lastRowNum = sheetAt.getLastRowNum();
        List<Article> articleList = new ArrayList<Article>();
        for(int i = 1 ; i <= lastRowNum ; i++){
            Article article = new Article();
            //获取我们一行 行的数据
            XSSFRow row = sheetAt.getRow(i);
            //通过我们的 row 对象，解析里面一个个的字段
            XSSFCell title = row.getCell(0);
            XSSFCell from = row.getCell(1);
            XSSFCell time = row.getCell(2);
            XSSFCell readCount = row.getCell(3);
            XSSFCell content = row.getCell(4);
            // System.out.println(title.toString());
            article.setIdx(i + "");
            article.setTitle(title.toString());
            article.setContent(content.toString());
            article.setFrom(from.toString());
            article.setReadCounts(readCount.toString());
            article.setTimes(time.toString());
            articleList.add(article);
        }
        fileInputStream.close();
        return articleList;
    }
}
```

定义 main 方法(ESToHBase)

```
public class AppMain {
    private static final String tableName = "hbase_es_article";
    private static final String familyName = "f1";
    private static final String title = "title";
    private static final String from = "from";
}
```

```
private static final String times = "times";
private static final String readCounts = "readCounts";
private static final String content = "content";

public static void main(String[] args) throws IOException {
    //使用 java 代码解析 excel 表格
    List<Article> excelInfo = ExcelUtil.getExcelInfo();

    /* //将集合当中的数据，保存到 es 当中去
    TransportClient client = getEsClient();
    save2Es(excelInfo, client);
    Table table = getTable();
    //循环遍历我们的数据，将我们的数据装到 List<Put>
    saveToHbase(excelInfo, table);*/

    /* //通过一个关键字进行搜索，将我们的数据从 es 当中查询出来
    TransportClient esClient = getEsClient();
    List<String> getAllKeyWord = getByKeyWord(esClient, "机器人");*/

    //拿到数据的 id，看数据详情 1216
    Table table = getTable();
    Get get = new Get("1216".getBytes());
    Result result = table.get(get);
    Cell[] cells = result.rawCells();
    for (Cell cell : cells) {
        byte[] value = cell.getValue();
        System.out.println(Bytes.toString(value));
        //将文章内容封装到 article 给前端返回即可
    }
}

private static List<String> getByKeyWord(TransportClient
esClient, String keyWord) {
    ArrayList<String> strings = new ArrayList<String>();
    SearchResponse searchResponse =
esClient.prepareSearch("articles").setTypes("article").setQuery(QueryBuilde
rs.termQuery("title", keyWord)).get();
    SearchHits hits = searchResponse.getHits();
    for (SearchHit hit : hits) {
        //获取我们数据的系统的 id
        String id = hit.getId();
        // System.out.println(id);
        strings.add(id);
    }
    return strings;
}

private static void saveToHbase(List<Article> excelInfo, Table table)
throws IOException {
```

```
System.out.println(excelInfo.size());
long startTime = System.currentTimeMillis();

List<Put> putList = new ArrayList<Put>();
for (Article article : excelInfo) {
    System.out.println(article.getTitle());
    Put put = new Put(Bytes.toBytes(article.getId()));
    if(article.getTitle() != null && article.getTitle() != ""){
        put.addColumn(familyName.getBytes(), title.getBytes(), article.getTitle().getBytes());

        put.addColumn(familyName.getBytes(), from.getBytes(), article.getFrom().getBytes());

        put.addColumn(familyName.getBytes(), times.getBytes(), article.getTimes().getBytes());

        put.addColumn(familyName.getBytes(), readCounts.getBytes(), article.getReadCounts().getBytes());

        put.addColumn(familyName.getBytes(), content.getBytes(), article.getContent().getBytes());
        putList.add(put);
    }
}
table.put(putList);

long endTime = System.currentTimeMillis();
System.out.println((endTime-startTime)/1000);
table.close();
}

private static Table getTable() throws IOException {
    //将集合当中的数据，保存到 hbase 当中去
    //第一步：获取 hbase 的客户端连接
    Configuration configuration = HBaseConfiguration.create();

    configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181,node03:2181");
    Connection connection =
    ConnectionFactory.createConnection(configuration);
    Admin admin = connection.getAdmin();
    //设置我们表名
    HTableDescriptor hTableDescriptor = new
    HTableDescriptor(TableName.valueOf(tableName));
    HColumnDescriptor f1 = new HColumnDescriptor(familyName);
    hTableDescriptor.addFamily(f1);
    if(!admin.tableExists(TableName.valueOf(tableName))){
```

```
        admin.createTable(hTableDescriptor);
    }
    return connection.getTable(TableName.valueOf(tableName));
}

private static void save2Es(List<Article> excelInfo, TransportClient
client) {
    //通过批量添加，将我们的数据保存到 es 当中去
    BulkRequestBuilder bulk = client.prepareBulk();
    /**
     * 循环遍历我们的集合，组织我们 IndexRequestBuilder
     */
    for (Article article : excelInfo) {
        IndexRequestBuilder indexRequestBuilder =
client.prepareIndex("articles", "article", article.getId());
        Gson gson = new Gson();
        String jsonStr = gson.toJson(article);
        indexRequestBuilder.setSource(jsonStr, XContentType.JSON);
        bulk.add(indexRequestBuilder);
    }
    //触发我们的数据真正的保存到 es 当中去
    BulkResponse bulkItemResponses = bulk.get();
    client.close();
}

private static TransportClient getEsClient() throws
UnknownHostException {
    Settings settings = Settings.builder().put("cluster.name",
"myes").build();
    TransportClient client = new PreBuiltTransportClient(settings)
        .addTransportAddress(new
TransportAddress(InetAddress.getByAddress("node01"),9300))
        .addTransportAddress(new
TransportAddress(InetAddress.getByAddress("node02"),9300));
    return client;
}
}
```