狂神聊ElasticSearch

聊聊Doug Cutting

1998年9月4日，Google公司在美国硅谷成立。正如大家所知，它是一家做搜索引擎起家的公司。

无独有偶，一位名叫**Doug Cutting**的美国工程师，也迷上了搜索引擎。他做了一个用于文本搜索的函数库（姑且理解为软件的功能组件），命名为**Lucene**。

左为Doug Cutting，右为Lucene的LOGO

Lucene是用JAVA写成的，目标是为各种中小型应用软件加入全文检索功能。因为好用而且开源（代码公开），非常受程序员们的欢迎。

早期的时候，这个项目被发布在Doug Cutting的个人网站和SourceForge（一个开源软件网站）。后来，2001年底，Lucene成为**Apache软件基金会**jakarta项目的一个子项目。

Apache软件基金会，搞IT的应该都认识

2004年，Doug Cutting再接再励，在Lucene的基础上，和Apache开源伙伴Mike Cafarella合作，开发了一款可以代替当时的主流搜索的开源搜索引擎，命名为**Nutch**。

Nutch是一个建立在Lucene核心之上的网页搜索应用程序，可以下载下来直接使用。它在Lucene的基础上加了网络爬虫和一些网页相关的功能，目的就是从一个简单的站内检索推广到全球网络的搜索上，就像Google一样。

Nutch在业界的影响力比Lucene更大。

大批网站采用了Nutch平台，大大降低了技术门槛，使低成本的普通计算机取代高价的Web服务器成为可能。甚至有一段时间，在硅谷有了一股用Nutch低成本创业的潮流。

随着时间的推移，无论是Google还是Nutch，都面临搜索对象“体积”不断增大的问题。

尤其是Google，作为互联网搜索引擎，需要存储大量的网页，并不断优化自己的搜索算法，提升搜索效率。

Google搜索栏

在这个过程中，Google确实找到了不少好办法，并且无私地分享了出来。

2003年，Google发表了一篇技术学术论文，公开介绍了自己的谷歌文件系统**GFS（Google File System）**。这是Google公司为了存储海量搜索数据而设计的专用文件系统。

第二年，也就是2004年，Doug Cutting基于Google的GFS论文，实现了**分布式文件存储系统**，并将它命名为**NDFS（Nutch Distributed File System）**。

还是2004年，Google又发表了一篇技术学术论文，介绍自己的**MapReduce编程模型**。这个编程模型，用于大规模数据集（大于1TB）的并行分析运算。

第二年（2005年），Doug Cutting又基于MapReduce，在Nutch搜索引擎实现了该功能。

2006年，当时依然很厉害的**Yahoo（雅虎）公司**，招安了Doug Cutting。

这里要补充说明一下雅虎招安Doug的背景：2004年之前，作为互联网开拓者的雅虎，是使用Google搜索引擎作为自家搜索服务的。在2004年开始，雅虎放弃了Google，开始自己研发搜索引擎。所以。。。

加盟Yahoo之后，Doug Cutting将NDFS和MapReduce进行了升级改造，并重新命名为**Hadoop**（NDFS也改名为HDFS，Hadoop Distributed File System）。

这个，就是后来大名鼎鼎的大数据框架系统——Hadoop的由来。而Doug Cutting，则被人们称为**Hadoop之父**。

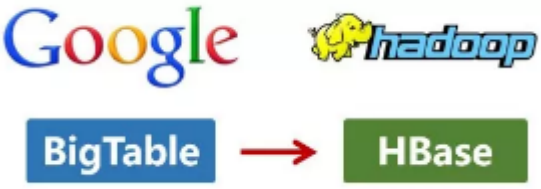
Hadoop这个名字，实际上是Doug Cutting他儿子的黄色玩具大象的名字。所以，Hadoop的Logo，就是一只奔跑的黄色大象。

我们继续往下说。

还是2006年，Google又发论文了。

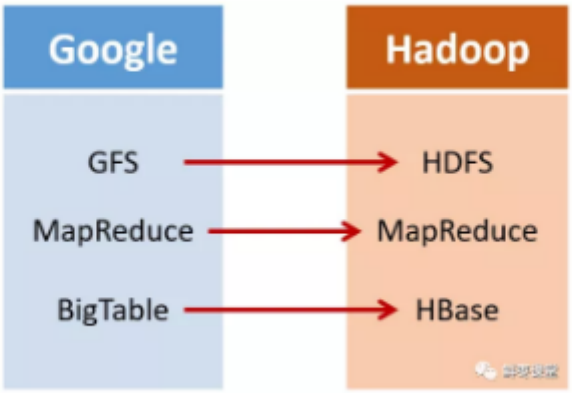
这次，它们介绍了自己的**BigTable**。这是一种分布式数据存储系统，一种用来处理海量数据的非关系型数据库。

Doug Cutting当然没有放过，在自己的hadoop系统里面，引入了BigTable，并命名为**HBase**。



好吧，反正就是紧跟Google时代步伐，你出什么，我学什么。

所以，Hadoop的核心部分，基本上都有Google的影子。



2008年1月，Hadoop成功上位，正式成为Apache基金会的顶级项目。

同年2月，Yahoo宣布建成了一个拥有1万个内核的Hadoop集群，并将自己的搜索引擎产品部署在上面。

7月，Hadoop打破世界纪录，成为最快排序1TB数据的系统，用时209秒。

> 回到主题

Lucene是一套信息检索工具包，并不包含搜索引擎系统，它包含了索引结构、读写索引工具、相关性工具、排序等功能，因此在使用Lucene时仍需要关注搜索引擎系统，例如数据获取、解析、分词等方面的东西。

为什么要给大家介绍下Lucene呢，因为我们学过的solr 和 即将要学习的elasticsearch都是基于该工具包做的一些封装和增强罢了~

按照我的风格，一般在开始都会给大家聊聊历史！学习更重要的是培养编程的兴趣，而不是只会使用一些基本的API！

狂神的教学风格，免费，授人以渔，还有我那句每个课程都会说的至理名言：只要学不死，就往死里学！

ElasticSearch概述

Elaticsearch，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。

据国际权威的数据库产品评测机构DB Engines的统计，在2016年1月，ElasticSearch已超过Solr等，成为排名第一的搜索引擎类应用。

> 历史

多年前，一个叫做Shay Banon的刚结婚不久的失业开发者，由于妻子要去伦敦学习厨师，他便跟着也去了。在他找工作的过程中，为了给妻子构建一个食谱的搜索引擎，他开始构建一个早期版本的Lucene。

直接基于Lucene工作会比较困难，所以Shay开始抽象Lucene代码以便Java程序员可以在应用中添加搜索功能。他发布了他的第一个开源项目，叫做“Compass”。

后来Shay找到一份工作，这份工作处在高性能和内存数据网格的分布式环境中，因此高性能的、实时的、分布式的搜索引擎也是理所当然需要的。然后他决定重写Compass库使其成为一个独立的服务叫做Elasticsearch。

第一个公开版本出现在2010年2月，在那之后Elasticsearch已经成为Github上最受欢迎的项目之一，代码贡献者超过300人。一家主营Elasticsearch的公司就此成立，他们一边提供商业支持一边开发新功能，不过Elasticsearch将永远开源且对所有人可用。

Shay的妻子依旧等待着她的食谱搜索……

谁在使用：

1、维基百科，类似百度百科，全文检索，高亮，搜索推荐/2

2、The Guardian（国外新闻网站），类似搜狐新闻，用户行为日志（点击，浏览，收藏，评论）+社交网络数据（对某某新闻的相关看法），数据分析，给到每篇新闻文章的作者，让他知道他的文章的公众反馈（好，坏，热门，垃圾，鄙视，崇拜）

3、Stack Overflow（国外的程序异常讨论论坛），IT问题，程序的报错，提交上去，有人会跟你讨论和回答，全文检索，搜索相关问题和答案，程序报错了，就会将报错信息粘贴到里面去，搜索有没有对应的答案

4、GitHub（开源代码管理），搜索上千亿行代码

5、电商网站，检索商品

6、日志数据分析，logstash采集日志，ES进行复杂的数据分析，ELK技术，elasticsearch+logstash+kibana

7、商品价格监控网站，用户设定某商品的价格阈值，当低于该阈值的时候，发送通知消息给用户，比如说订阅牙膏的监控，如果高露洁牙膏的家庭套装低于50块钱，就通知我，我就去买

8、BI系统，商业智能，Business Intelligence。比如说有个大型商场集团，BI，分析一下某某区域最近3年的用户消费金额的趋势以及用户群体的组成构成，产出相关的数张报表，**区，最近3年，每年消费金额呈现100%的增长，而且用户群体85%是高级白领，开一个新商场。ES执行数据分析和挖掘，Kibana进行数据可视化

9、国内：站内搜索（电商，招聘，门户，等等），IT系统搜索（OA，CRM，ERP，等等），数据分析（ES热门

的一个使用场景）

ES和solr的差别

Elasticsearch简介

Elasticsearch是一个实时分布式搜索和分析引擎。它让你以前所未有的速度处理大数据成为可能。
它用于全文搜索、结构化搜索、分析以及将这三者混合使用：
维基百科使用Elasticsearch提供全文搜索并高亮关键字，以及输入实时搜索(search-asyou-type)和搜索纠错(did-you-mean)等搜索建议功能。
英国卫报使用Elasticsearch结合用户日志和社交网络数据提供给他们的编辑以实时的反馈，以便及时了解公众对新发表的文章的回应。
StackOverflow结合全文搜索与地理位置查询，以及more-like-this功能来找到相关的问题和答案。
Github使用Elasticsearch检索1300亿行的代码。
但是Elasticsearch不仅用于大型企业，它还让像DataDog以及Klout这样的创业公司将最初的想法变成可

扩展的解决方案。Elasticsearch可以在你的笔记本上运行，也可以在数以百计的服务器上处理PB级别的数据 。

Elasticsearch是一个基于Apache Lucene(TM)的开源搜索引擎。无论在开源还是专有领域，Lucene可以被认为是迄今为止最先进、性能最好的、功能最全的搜索引擎库。
但是，Lucene只是一个库。想要使用它，你必须使用Java来作为开发语言并将其直接集成到你的应用中，更糟糕的是，Lucene非常复杂，你需要深入了解检索的相关知识来理解它是如何工作的。
Elasticsearch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。

Solr简介

Solr 是Apache下的一个顶级开源项目，采用Java开发，它是基于Lucene的全文搜索服务器。Solr提供了比Lucene更为丰富的查询语言，同时实现了可配置、可扩展，并对索引、搜索性能进行了优化

Solr可以独立运行，运行在Jetty、Tomcat等这些Servlet容器中，Solr 索引的实现方法很简单，用 POST 方法向 Solr 服务器发送一个描述 Field 及其内容的 XML 文档，Solr根据xml文档添加、删除、更新索引 。Solr 搜索只需要发送 HTTP GET 请求，然后对 Solr 返回Xml、json等格式的查询结果进行解析，组织页面布局。Solr不提供构建UI的功能，Solr提供了一个管理界面，通过管理界面可以查询Solr的配置和运行情况。

solr是基于lucene开发企业级搜索服务器，实际上就是封装了lucene。

Solr是一个独立的企业级搜索应用服务器，它对外提供类似于Web-service的API接口。用户可以通过http请求，向搜索引擎服务器提交一定格式的文件，生成索引；也可以通过提出查找请求，并得到返回结果。

Lucene简介

Lucene是apache软件基金会4 jakarta项目组的一个子项目，是一个开放源代码的全文检索引擎工具包，但它不是一个完整的全文检索引擎，而是一个全文检索引擎的架构，提供了完整的查询引擎和索引引擎，部分文本分析引擎（英文与德文两种西方语言）。Lucene的目的是为软件开发人员提供一个简单易用的工具包，以方便的在目标系统中实现全文检索的功能，或者是以此为基础建立起完整的全文检索引擎。Lucene是一套用于全文检索和搜寻的开源程式库，由Apache软件基金会支持和提供。Lucene提供了一个简单却强大的应用程式接口，能够做全文索引和搜寻。在Java开发环境里Lucene是一个成熟的免费开源工具。就其本身而言，Lucene是当前以及最近几年最受欢迎的免费Java信息检索程序库。人们经常提到信息检索程序库，虽然与搜索引擎有关，但不应该将信息检索程序库与搜索引擎相混淆。

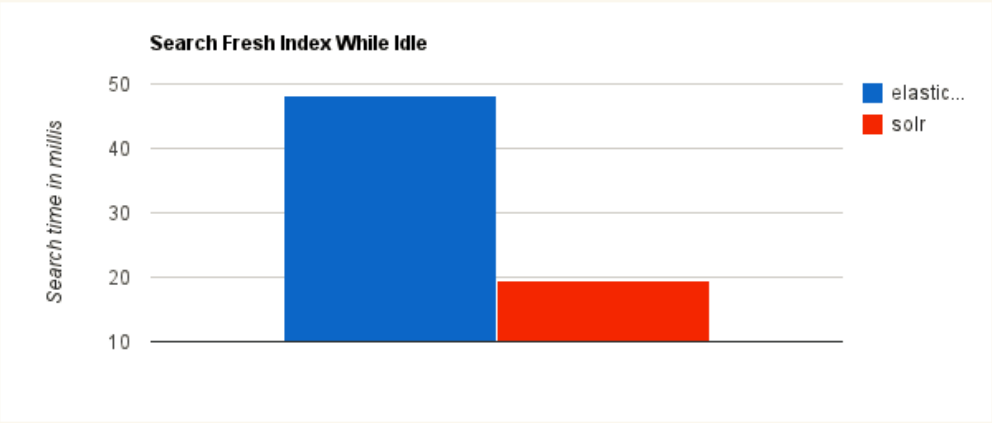
Lucene是一个全文检索引擎的架构。那什么是全文搜索引擎?

全文搜索引擎是名副其实的搜索引擎，国外具代表性的有Google、Fast/AllTheWeb、AltaVista、Inktomi、Teoma、WiseNut等，国内著名的有百度（Baidu）。它们都是通过从互联网上提取的各个网站的信息（以网页文字为主）而建立的数据库中，检索与用户查询条件匹配的相关记录，然后按一定的排列顺序将结果返回给用户，因此他们是真正的搜索引擎。

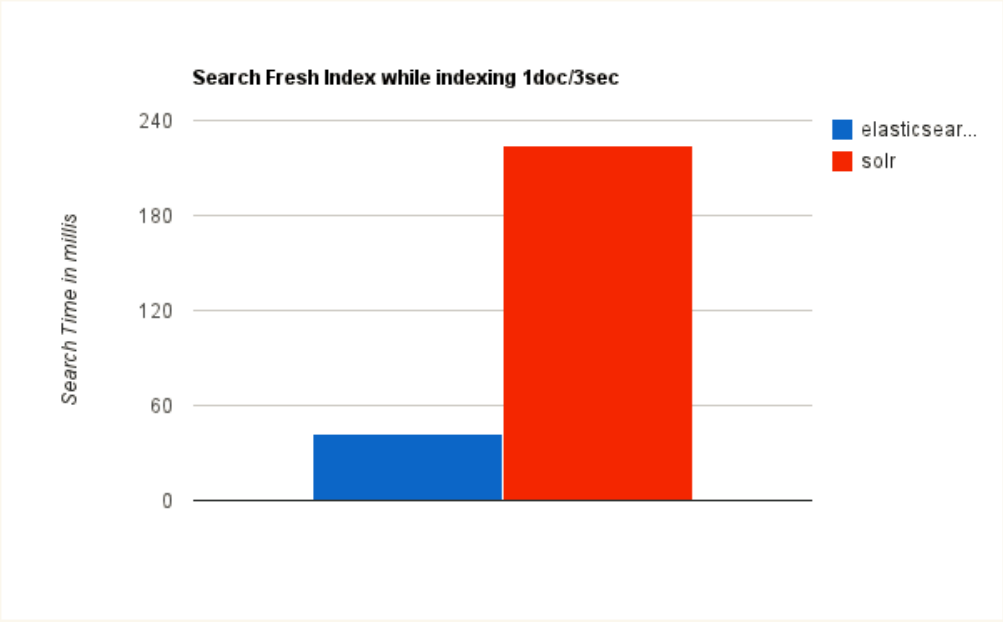
从搜索结果来源的角度，全文搜索引擎又可细分为两种，一种是拥有自己的检索程序（Indexer），俗称“蜘蛛”（Spider）程序或“机器人”（Robot）程序，并自建网页数据库，搜索结果直接从自身的数据库中调用，如上面提到的7家引擎；另一种则是租用其他引擎的数据库，并按自定的格式排列搜索结果，如Lycos引擎。

Elasticsearch和Solr比较

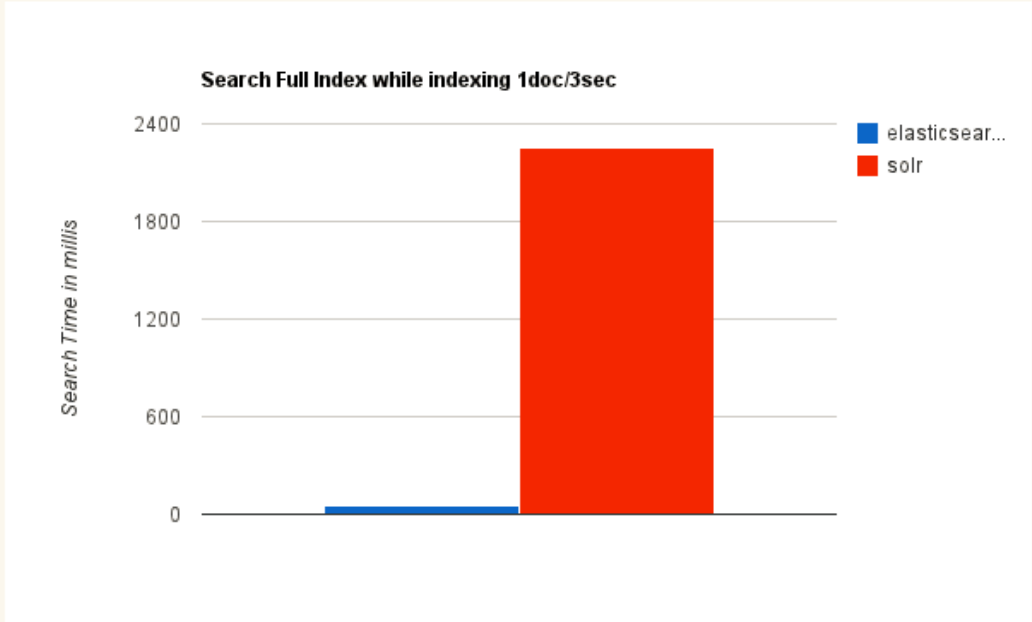
当单纯的对已有数据进行搜索时，Solr更快。

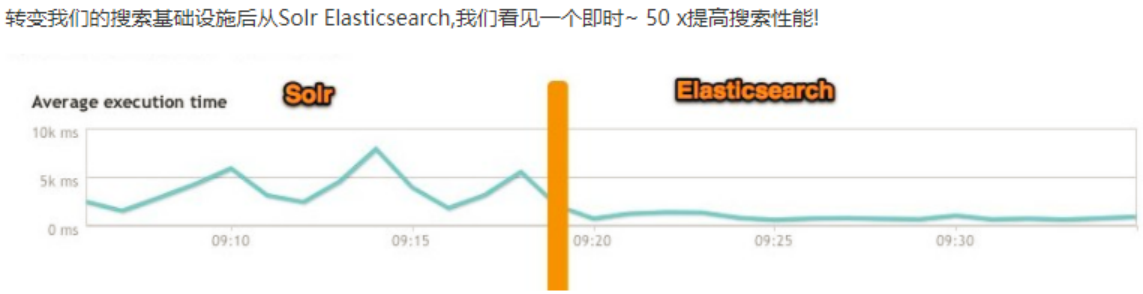


当实时建立索引时，Solr会产生io阻塞，查询性能较差，Elasticsearch具有明显的优势。



随着数据量的增加，Solr的搜索效率会变得更低，而Elasticsearch却没有明显的变化。





ElasticSearch vs Solr 总结

1、es基本是开箱即用，非常简单。Solr安装略微复杂一丢丢！

2、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。

（3）Solr 支持更多格式的数据，比如JSON、XML、CSV，而 Elasticsearch 仅支持json文件格式。

4、Solr 官方提供的功能更多，而 Elasticsearch 本身更注重于核心功能，高级功能多有第三方插件提供，例如图形化界面需要kibana友好支撑

5、Solr 查询快，但更新索引时慢（即插入删除慢），用于电商等查询多的应用；

- ES建立索引快（即查询慢），即实时性查询快，用于facebook新浪等搜索。
- Solr 是传统搜索应用的有力解决方案，但 Elasticsearch 更适用于新兴的实时搜索应用。

6、Solr比较成熟，有一个更大，更成熟的用户、开发和贡献者社区，而 Elasticsearch相对开发维护者较少，更新太快，学习使用成本较高。

ElasticSearch安装

> 说明

我们需要下载和安装ElasticSearch的服务端和客户端！

注意：

ElasticSearch是使用java开发的，且本版本的es需要的jdk版本要是1.8以上，所以安装ElasticSearch 之前保证JDK1.8+安装完毕，并正确的配置好JDK环境变量，否则启动ElasticSearch失败。

> 下载

ElasticSearch的官方地址： https://www.elastic.co/products/elasticsearch

官方下载地址：https://www.elastic.co/cn/downloads/elasticsearch （很慢，可以翻墙下载！）

win下载：https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.6.1.zip

我这里已经帮大家下载好了，Linux 和 window 版的！

我们学习的话使用 window 或者 linux 都是可以的，对于我们 java 开发来说没有区别，只是连接的问题！

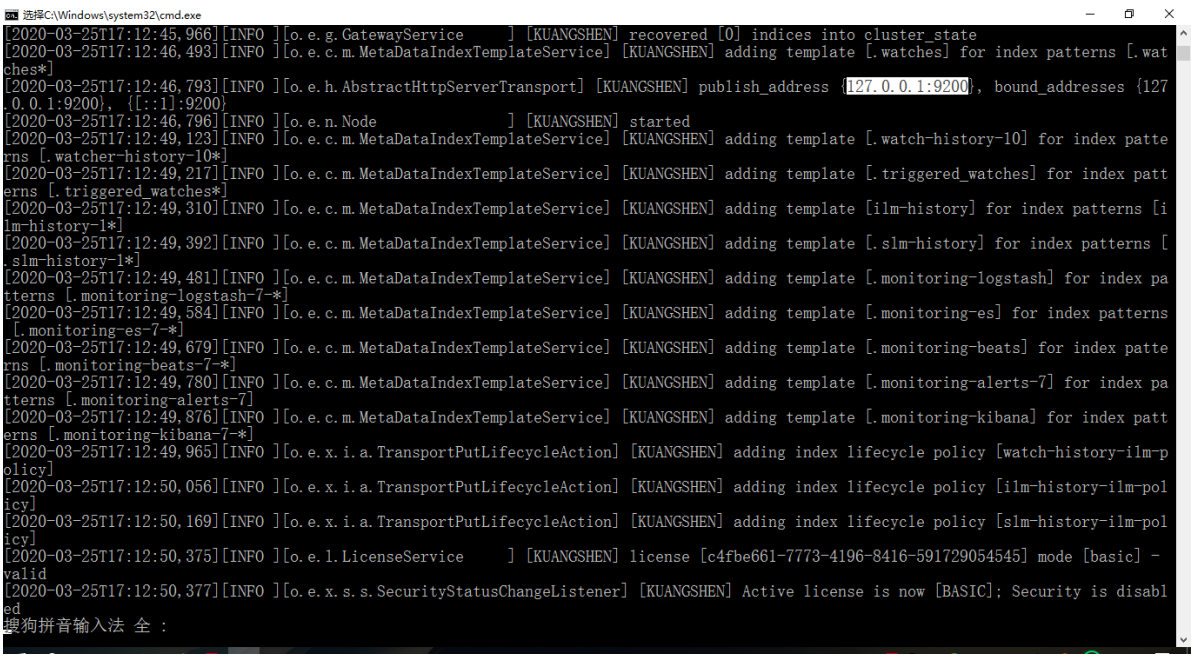
Windows更加方便一点！所以我们前期都是用Window安装使用！后面我们再真正的安装到Linux服务器上跑项目！

> window 下安装使用

1、解压window的压缩包！


```
bin：启动文件
config：配置文件
    log4j2.properties：日志配置文件
    jvm.options：java虚拟机的配置
    elasticsearch.yml：es的配置文件
data：索引数据目录
lib：相关类库Jar包
logs：日志目录
modules：功能模块
plugins：插件
```

2、双击ElasticSearch下的bin目录中的elasticsearch.bat启动，控制台显示的日志（等待启动完毕！）：



3、然后在浏览器访问：http://localhost:9200 得到如下信息，说明安装成功了：

> 安装ES的图形化界面插件客户端

注意：需要NodeJS的环境，我们讲解大前端进阶已经安装过了，没安装的需要安装！

Head是elasticsearch的集群管理工具，可以用于数据的浏览查询！被托管在github上面！

地址： https://github.com/mobz/elasticsearch-head/

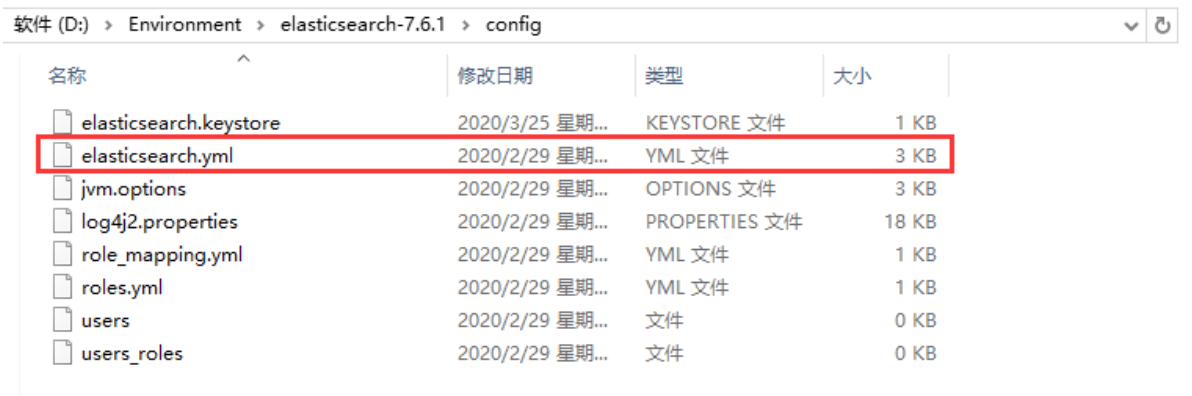
1、下载 elasticsearch-head-master.zip

2、解压之后安装依赖！

```
cnpm install
npm run start
```

这将启动在端口9100上运行的本地web服务器，为elasticsearch-head服务！访问测试：

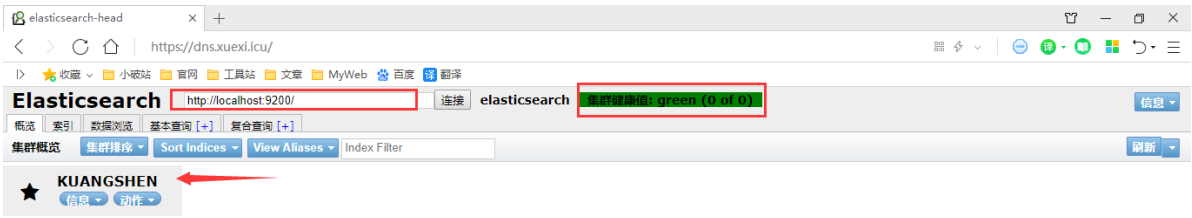
3、由于ES进程和客户端进程端口号不同，存在跨域问题，所以我们要在ES的配置文件中配置下跨域问题：




```
# 跨域配置：
http.cors.enabled: true
http.cors.allow-origin: "*"
```

```
84  # ---------------------------------- Various -----------------------------------
85  #
86  # Require explicit names when deleting indices:
87  #
88  #action.destructive_requires_name: true
89  # 跨域配置：
90  http.cors.enabled: true
91  http.cors.allow-origin: "*"
```

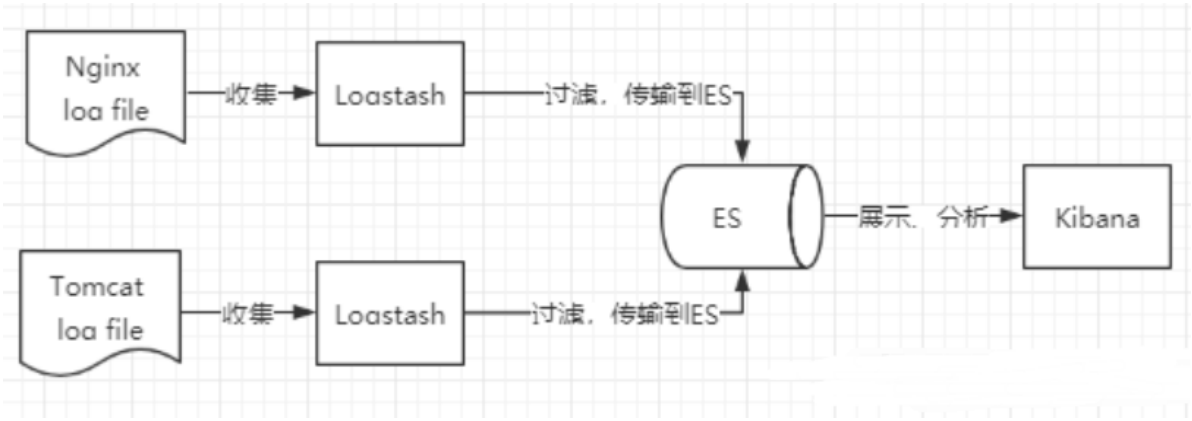
4、启动ElasticSearch，使用head工具进行连接测试！



了解 ELK

ELK是Elasticsearch、Logstash、Kibana三大开源框架首字母大写简称。市面上也被成为Elastic Stack。其中Elasticsearch是一个基于Lucene、分布式、通过Restful方式进行交互的近实时搜索平台框架。像类似百度、谷歌这种大数据全文搜索引擎的场景都可以使用Elasticsearch作为底层支持框架，可见Elasticsearch提供的搜索能力确实强大,市面上很多时候我们简称Elasticsearch为es。Logstash是ELK的中央数据流引擎，用于从不同目标（文件/数据存储/MQ）收集的不同格式数据，经过过滤后支持输出到不同目的地（文件/MQ/redis/elasticsearch/kafka等）。Kibana可以将elasticsearch的数据通过友好的页面展示出来，提供实时分析的功能。

市面上很多开发只要提到ELK能够一致说出它是一个日志分析架构技术栈总称，但实际上ELK不仅仅适用于日志分析，它还可以支持其它任何数据分析和收集的场景，日志分析和收集只是更具有代表性。并非唯一性。

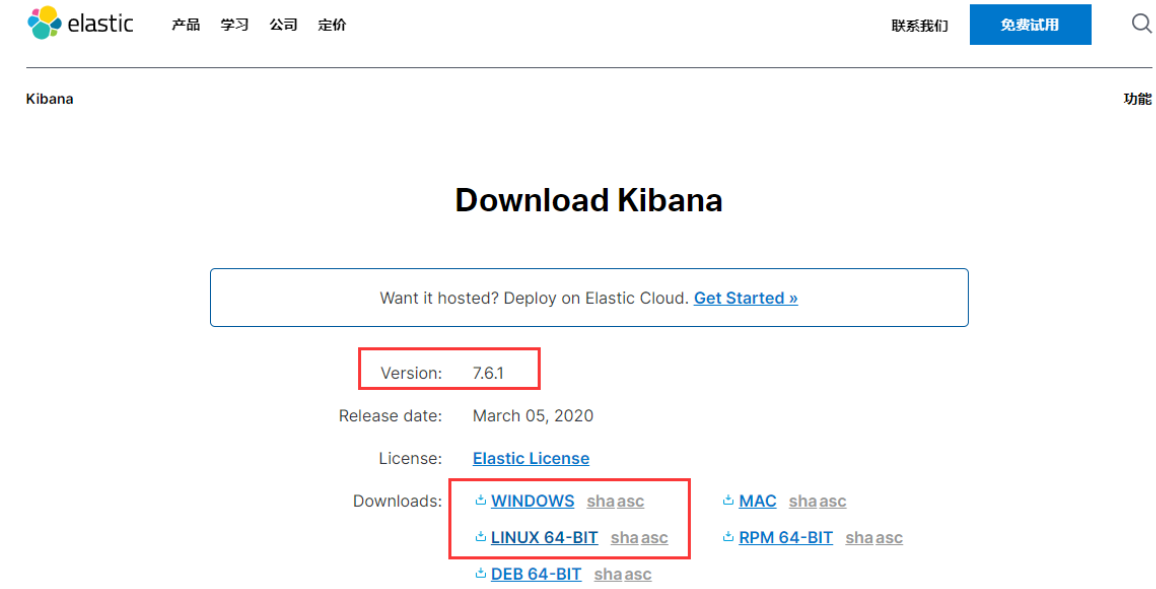


安装Kibana

Kibana是一个针对Elasticsearch的开源分析及可视化平台，用来搜索、查看交互存储在Elasticsearch索引中的数据。使用Kibana，可以通过各种图表进行高级数据分析及展示。Kibana让海量数据更容易理解。它操作简单，基于浏览器的用户界面可以快速创建仪表板（dashboard）实时显示Elasticsearch查询动态。设置Kibana非常简单。无需编码或者额外的基础架构，几分钟内就可以完成Kibana安装并启动Elasticsearch索引监测。

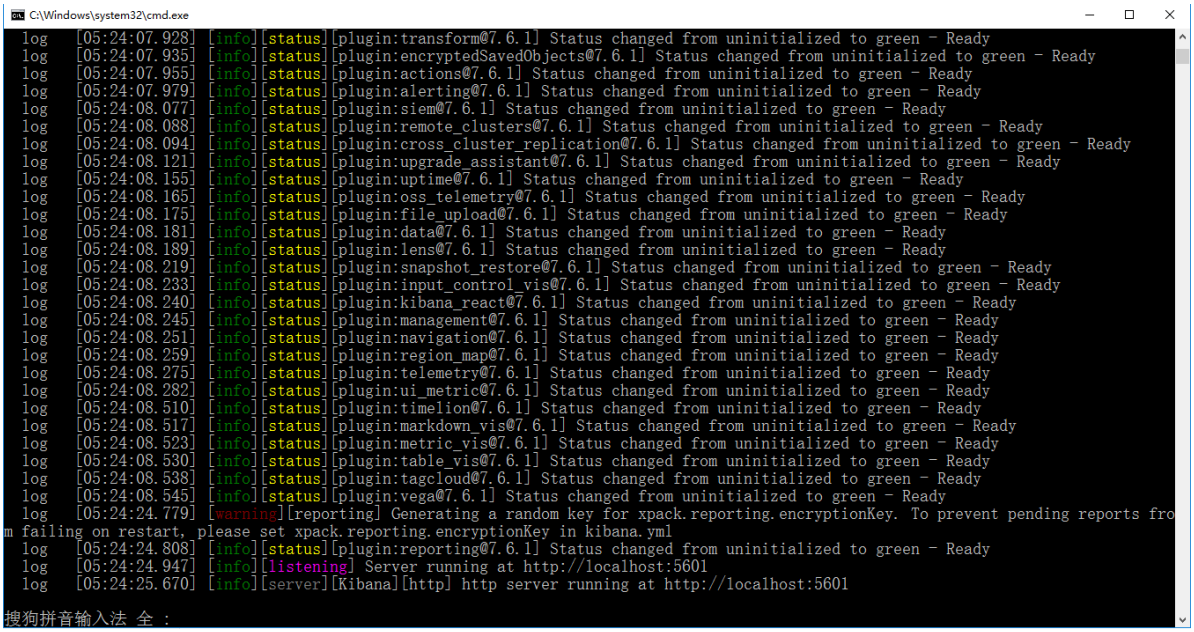
官网：https://www.elastic.co/cn/kibana

1、下载Kibana https://www.elastic.co/cn/downloads/kibana （注意版本对应关系）

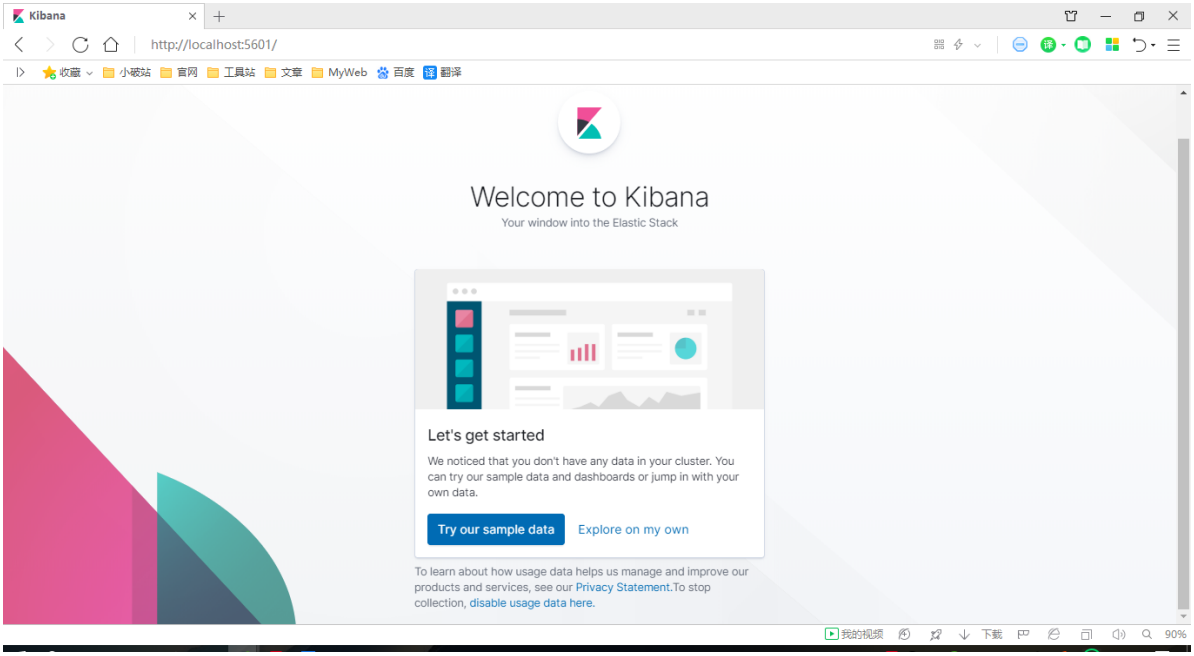


2、将压缩包解压即可（需要一些时间）！

3、然后进入到bin目录下，启动服务就可以了（需要等待启动完成），ELK基本上都是拆箱即用的



4、然后访问IP:5601，kibana会自动去访问9200，也就是elasticsearch的端口号（当然elasticsearch这个时候必须启动着），然后就可以使用kibana了！

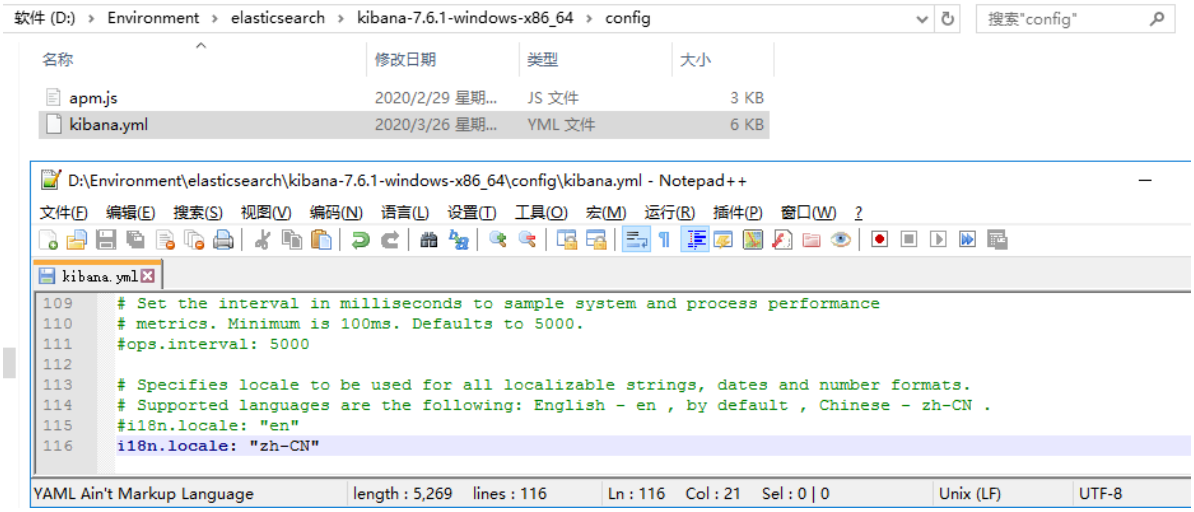


5、现在是英文的，看着有些吃力，我们配置为中文的！

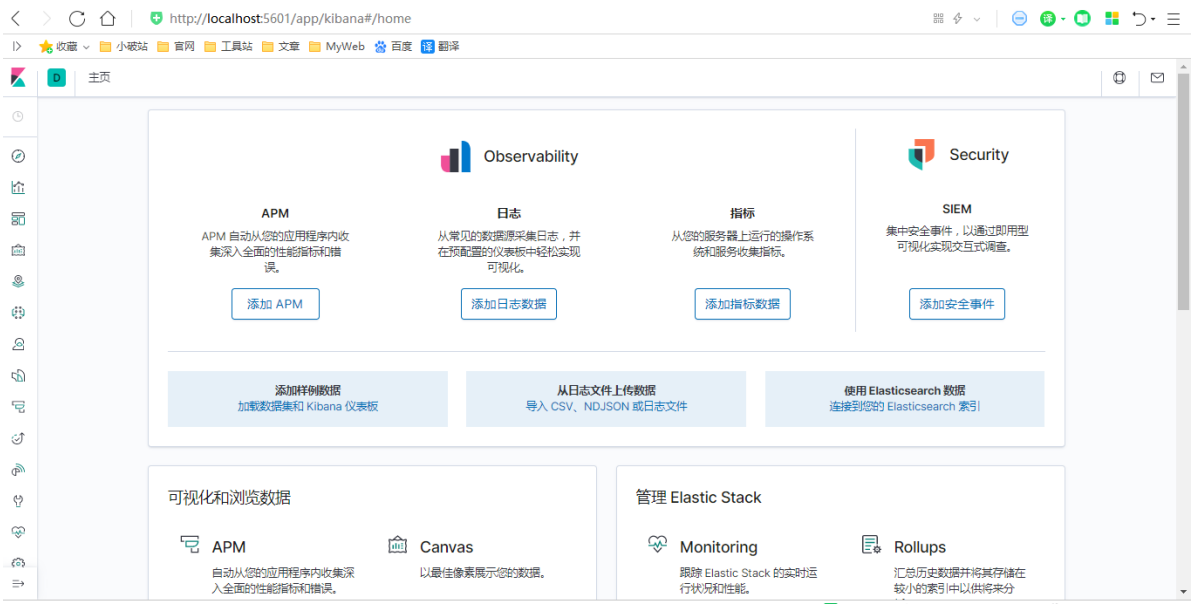
中文包在 `kibana\x-pack\plugins\translations\translations\zh-CN.json`

只需要在配置文件 kibana.yml 中加入

```
i18n.locale: "zh-CN"
```



6、重启查看效果！成功切换为中文的了！



ES核心概念

> 概述

在前面的学习中，我们已经掌握了es是什么，同时也把es的服务已经安装启动，那么es是如何去存储数据，数据结构是什么，又是如何实现搜索的呢？我们先来聊聊ElasticSearch的相关概念吧！

集群，节点，索引，类型，文档，分片，映射是什么？

> elasticsearch是面向文档，关系行数据库 和 elasticsearch 客观的对比！

Relational DB	Elasticsearch
数据库(database)	索引(indices)
表(tables)	types
行(rows)	documents
字段(columns)	fields

elasticsearch(集群)中可以包含多个索引(数据库)，每个索引中可以包含多个类型(表)，每个类型下又包含多 个文档(行)，每个文档中又包含多个字段(列)。

物理设计：

elasticsearch 在后台把每个**索引划分成多个分片**，每分分片可以在集群中的不同服务器间迁移

逻辑设计：

一个索引类型中，包含多个文档，比如说文档1，文档2。 当我们索引一篇文档时，可以通过这样的一各顺序找到 它: 索引 ▷ 类型 ▷ 文档ID ，通过这个组合我们就能索引到某个具体的文档。 注意:ID不必是整数，实际上它是个字 符串。

> 文档

之前说elasticsearch是面向文档的，那么就意味着索引和搜索数据的最小单位是文档，elasticsearch中，文档有几个 重要属性 :

- 自我包含，一篇文档同时包含字段和对应的值，也就是同时包含 key:value!
- 可以是层次型的，一个文档中包含自文档，复杂的逻辑实体就是这么来的!
- 灵活的结构，文档不依赖预先定义的模式，我们知道关系型数据库中，要提前定义字段才能使用，在elasticsearch中，对于字段是非常灵活的，有时候，我们可以忽略该字段，或者动态的添加一个新的字段。

尽管我们可以随意的新增或者忽略某个字段，但是，每个字段的类型非常重要，比如一个年龄字段类型，可以是字符 串也可以是整形。因为elasticsearch会保存字段和类型之间的映射及其他的设置。这种映射具体到每个映射的每种类型，这也是为什么在elasticsearch中，类型有时候也称为映射类型。

> 类型

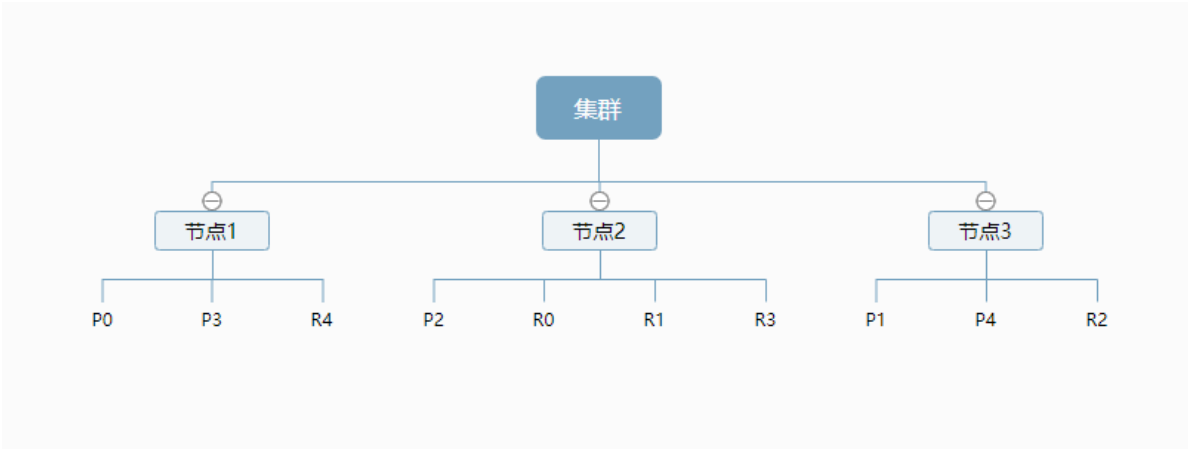
类型是文档的逻辑容器，就像关系型数据库一样，表格是行的容器。 类型中对于字段的定义称为映射，比如 name 映 射为字符串类型。 我们说文档是无模式的，它们不需要拥有映射中所定义的所有字段，比如新增一个字段，那么elasticsearch是怎么做的呢?elasticsearch会自动的将新字段加入映射，但是这个字段的不确定它是什么类型，elasticsearch就开始猜，如果这个值是18，那么elasticsearch会认为它是整形。 但是elasticsearch也可能猜不对， 所以最安全的方式就是提前定义好所需要的映射，这点跟关系型数据库殊途同归了，先定义好字段，然后再使用，别 整什么幺蛾子。

> 索引

索引是映射类型的容器，elasticsearch中的索引是一个非常大的文档集合。索引存储了映射类型的字段和其他设置。 然后它们被存储到了各个分片上了。 我们来研究下分片是如何工作的。

物理设计： 节点和分片 如何工作

一个集群至少有一个节点，而一个节点就是一个elasricsearch进程，节点可以有多个索引默认的，如果你创建索引，那么索引将会有个5个分片 (primary shard ,又称主分片) 构成的，每一个主分片会有一个副本 (replica shard ,又称复制分片)



上图是一个有3个节点的集群，可以看到主分片和对应的复制分片都不会在同一个节点内，这样有利于某个节点挂掉 了，数据也不至于丢失。 实际上，一个分片是一个Lucene索引，一个包含倒排索引的文件目录，倒排索引的结构使 得elasticsearch在不扫描全部文档的情况下，就能告诉你哪些文档包含特定的关键字。 不过，等等，倒排索引是什 么鬼?

> 倒排索引

elasticsearch使用的是一种称为倒排索引的结构，采用Lucene倒排索作为底层。这种结构适用于快速的全文搜索， 一个索引由文档中所有不重复的列表构成，对于每一个词，都有一个包含它的文档列表。 例如，现在有两个文档， 每个文档包含如下内容：

```
Study every day, good good up to forever  # 文档1包含的内容
To forever, study every day, good good up # 文档2包含的内容
```

为了创建倒排索引，我们首先要将每个文档拆分成独立的词(或称为词条或者tokens)，然后创建一个包含所有不重 复的词条的排序列表，然后列出每个词条出现在哪个文档：

term	**doc_1**	**doc_2**
Study	√	×
To	x	×
every	√	√
forever	√	√
day	√	√
study	×	√
good	√	√
every	√	√
to	√	×
up	√	√

现在，我们试图搜索 to forever，只需要查看包含每个词条的文档

term	**doc_1**	**doc_2**
to	√	×
forever	√	√
total	2	1

两个文档都匹配，但是第一个文档比第二个匹配程度更高。如果没有别的条件，现在，这两个包含关键字的文档都将返回。

再来看一个示例，比如我们通过博客标签来搜索博客文章。那么倒排索引列表就是这样的一个结构：

博客文章(原始数据)		索引列表(倒排索引)	
博客文章ID	标签	标签	博客文章ID
1	python	python	1, 2, 3
2	python	linux	3, 4
3	linux, python		
4	linux		

如果要搜索含有 python 标签的文章，那相对于查找所有原始数据而言，查找倒排索引后的数据将会快的多。只需要 查看标签这一栏，然后获取相关的文章ID即可。

elasticsearch的索引和Lucene的索引对比

在elasticsearch中， 索引 这个词被频繁使用，这就是术语的使用。 在elasticsearch中，索引被分为多个分片，每份 分片是一个Lucene的索引。所以一个elasticsearch索引是由多个Lucene索引组成的。别问为什么，谁让elasticsearch使用Lucene作为底层呢! 如无特指，说起索引都是指elasticsearch的索引。

接下来的一切操作都在kibana中Dev Tools下的Console里完成。基础操作!

ES基础操作

IK分词器插件

> 什么是IK分词器?

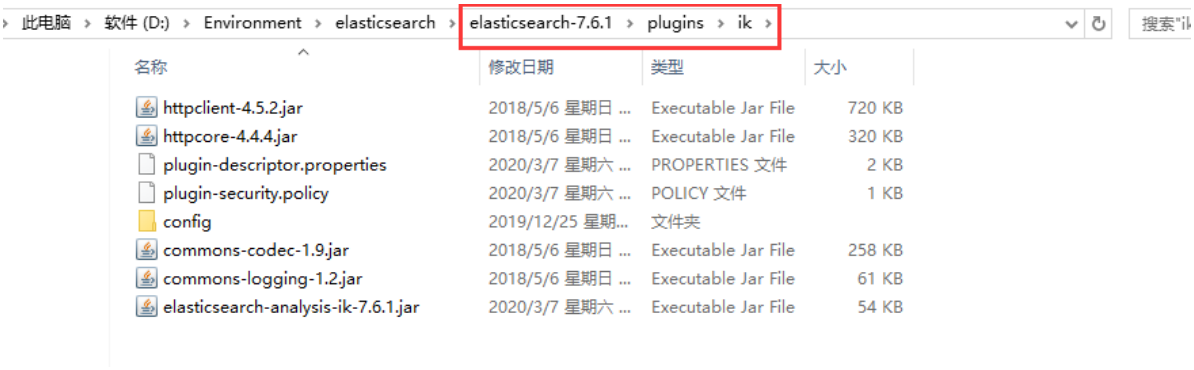
分词：即把一段中文或者别的划分成一个个的关键字，我们在搜索时候会把自己的信息进行分词，会把数据库中或者索引库中的数据进行分词，然后进行一个匹配操作，默认的中文分词是将每个字看成一个词，比如 “我爱狂神” 会被分为"我","爱","狂","神"，这显然是不符合要求的，所以我们需要安装中文分词器ik来解决这个问题。

IK提供了两个分词算法：ik_smart 和 ik_max_word，其中 ik_smart 为最少切分，ik_max_word为最细粒度划分！一会我们测试!

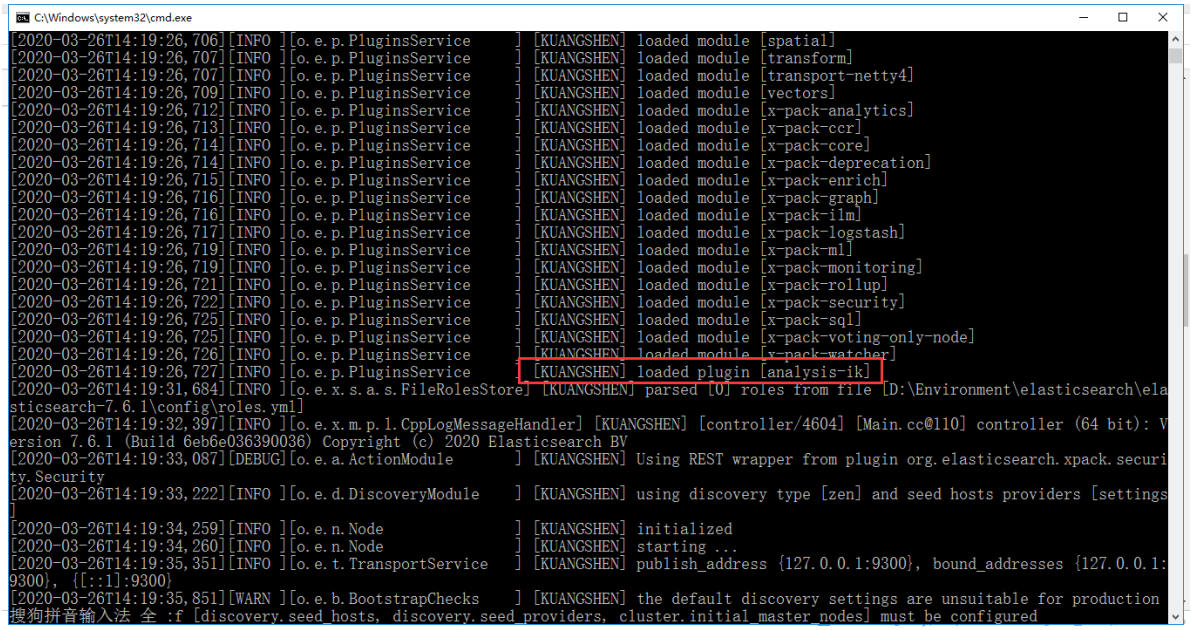
> 安装步骤

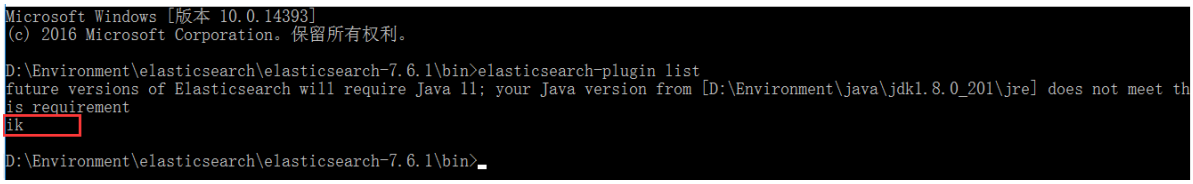
1、下载ik分词器的包，Github地址：https://github.com/medcl/elasticsearch-analysis-ik/ （版本要对应）

2、下载后解压，并将目录拷贝到ElasticSearch根目录下的 plugins 目录中。



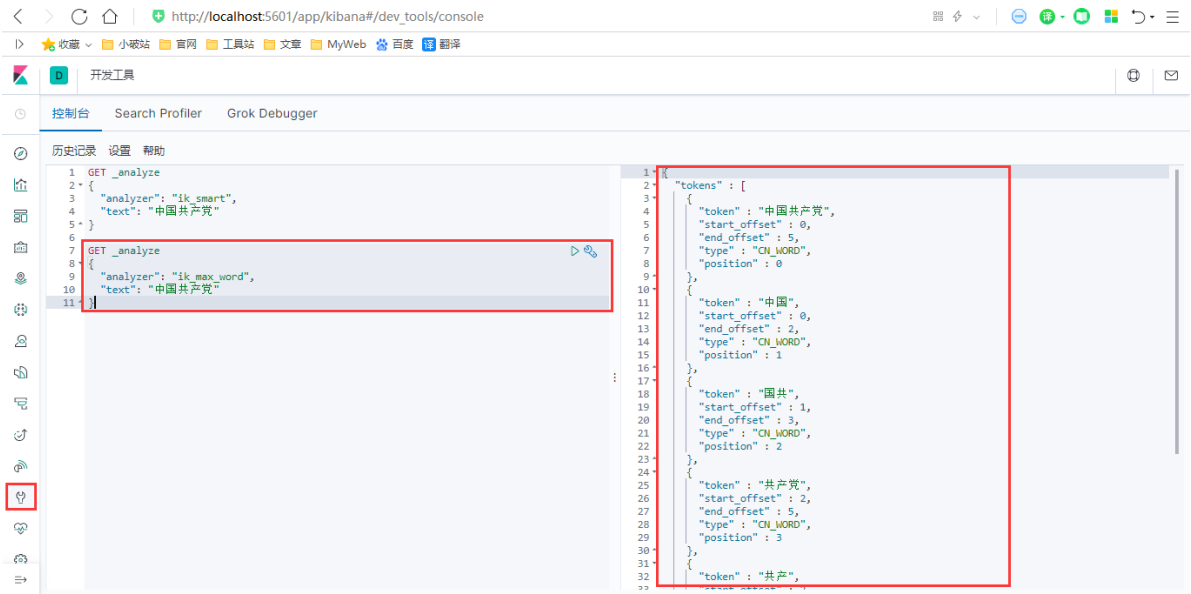
3、重新启动 ElasticSearch 服务，在启动过程中，你可以看到正在加载**"analysis-ik"**插件的提示信息，服务启动后，在命令行运行 **elasticsearch-plugin list** 命令，确认 ik 插件安装成功。



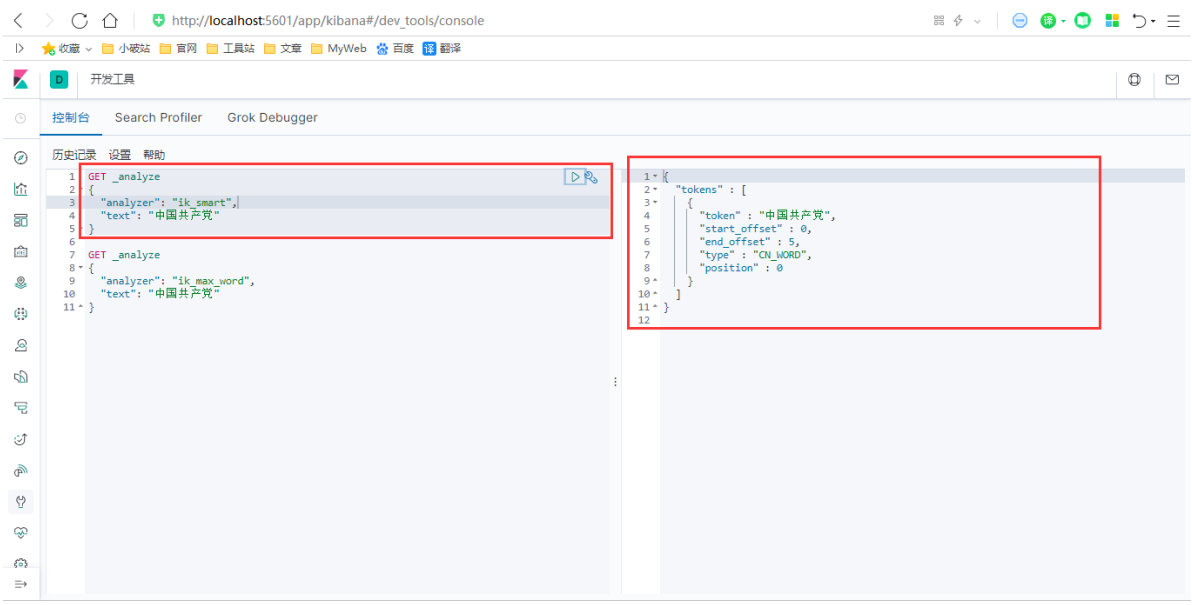


4、在 kibana 中测试 ik 分词器，并就相关分词结果和 icu 分词器进行对比。

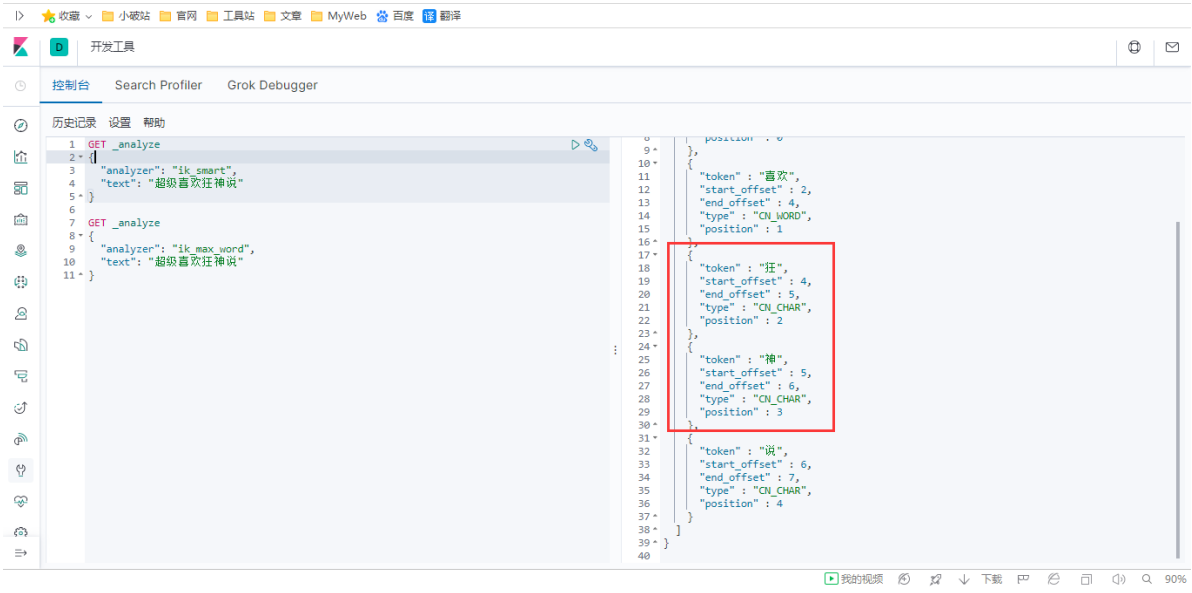
ik_max_word : 细粒度分词，会穷尽一个语句中所有分词可能，测试！



ik_smart : 粗粒度分词，优先匹配最长词，只有1个词！



5、我们输入超级喜欢狂神说！发现狂神说被切分了



如果我们想让系统识别"狂神说"是一个词，需要编辑自定义词库。

步骤：

（1）进入elasticsearch/plugins/ik/config目录

（2）新建一个my.dic文件，编辑内容：

```
狂神说
```

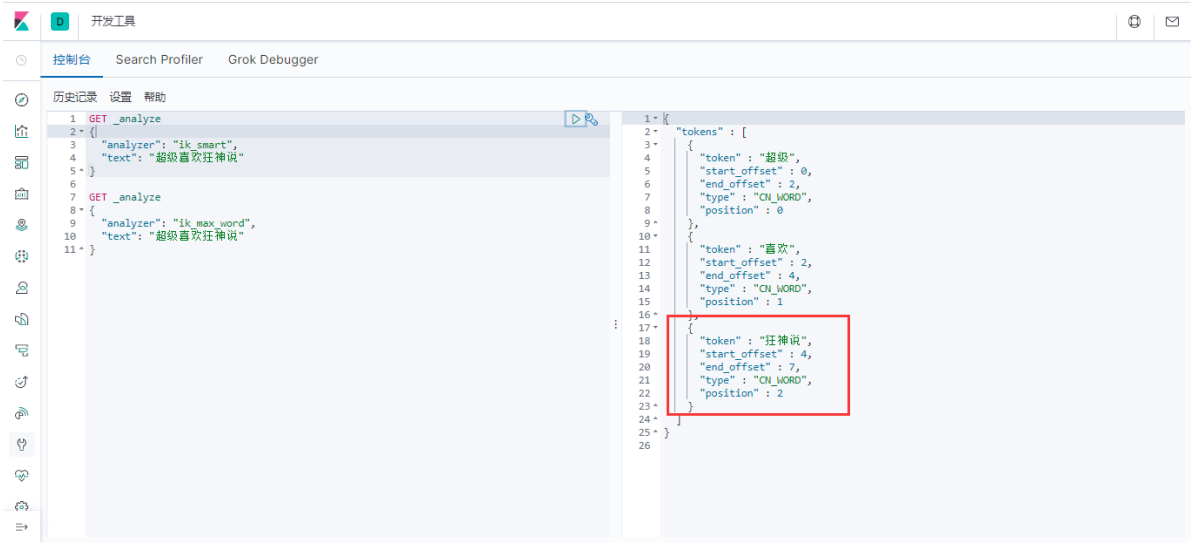
（3）修改IKAnalyzer.cfg.xml（在ik/config目录下）

```
<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">my.dic</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>
</properties>
```

修改完配置重新启动elasticsearch，再次测试！

发现监视了我们自己写的规则文件：

再次测试，发现狂神说变成了一个词：



到了这里，我们就明白了分词器的基本规则和使用了！

Rest风格说明

一种软件架构风格，而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

基本Rest命令说明：

method	url地址	描述
PUT	localhost:9200/索引名称/类型名称/文档id	创建文档（指定文档id）
POST	localhost:9200/索引名称/类型名称	创建文档（随机文档id）
POST	localhost:9200/索引名称/类型名称/文档id/_update	修改文档
DELETE	localhost:9200/索引名称/类型名称/文档id	删除文档
GET	localhost:9200/索引名称/类型名称/文档id	查询文档通过文档id
POST	localhost:9200/索引名称/类型名称/_search	查询所有数据

> 基础测试

1、首先我们浏览器 http://localhost:5601/ 进入 kibana里的Console

2、首先让我们在 Console 中输入：

```
// 命令解释
// PUT   创建命令   test1 索引  type1 类型  1     id
PUT /test1/type1/1
{
    "name":"狂神说",   // 属性
    "age":16           // 属性
}
```

返回结果 （是以REST ful 风格返回的 ）：

```
// 警告信息：不支持在文档索引请求中指定类型
// 而是使用无类型的端点(/{index}/_doc/{id}， /{index}/_doc，或
/{index}/_create/{id})。
{
  "_index" : "test1",     // 索引
  "_type" : "type1",      // 类型
  "_id" : "1",            // id
  "_version" : 1,         // 版本
  "result" : "created",   // 操作类型
  "_shards" : {           // 分片信息
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 0,
  "_primary_term" : 1
}
```

3、那么 name 这个字段用不用指定类型呢。毕竟我们关系型数据库 是需要指定类型的啊！

- 字符串类型
  text 、 keyword
- 数值类型
  long, integer, short, byte, double, float, half_float, scaled_float
- 日期类型
  date
- te布尔值类型
  boolean
- 二进制类型
  binary
- 等等.....

4、指定字段类型

```
PUT /test2
{
  "mappings": {
    "properties": {
      "name":{
        "type": "text"
      },
      "age":{
        "type": "long"
      },
      "birthday":{
        "type": "date"
      }
```

```
    }
  }
}
```

输出：

```
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "test2"
}
```

5、查看一下索引字段

```
GET test2
```

输出：

```
{
  "test2" : {
    "aliases" : { },
    "mappings" : {
      "properties" : {
        "age" : {
          "type" : "long"
        },
        "birthday" : {
          "type" : "date"
        },
        "name" : {
          "type" : "text"
        }
      }
    },
    "settings" : {
      "index" : {
        "creation_date" : "1585384302712",
        "number_of_shards" : "1",
        "number_of_replicas" : "1",
        "uuid" : "71TUZ84wRTW5P8lKeN4I4Q",
        "version" : {
          "created" : "7060199"
        },
        "provided_name" : "test2"
      }
    }
  }
}
```

6、我们看上列中 字段类型是我自己定义的 那么 我们不定义类型 会是什么情况呢？

```
PUT /test3/_doc/1
{
  "name":"狂神说",
  "age":13,
  "birth":"1997-01-05"
}
# 输出
```

```
{
  "_index" : "test3",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 0,
  "_primary_term" : 1
}

```

查看一下test3索引：

```
GET test3
```

返回结果：

```
{
  "test3" : {
    "aliases" : { },
    "mappings" : {
      "properties" : {
        "age" : {
          "type" : "long"
        },
        "birth" : {
          "type" : "date"
        },
        "name" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        }
      }
    },
    "settings" : {
      "index" : {
        "creation_date" : "1585384497051",
        "number_of_shards" : "1",
        "number_of_replicas" : "1",
        "uuid" : "xESBKF1XTpCAZOgMqBNUbA",
        "version" : {
          "created" : "7060199"
        },
        "provided_name" : "test3"
      }
    }
  }
}
```

我们看上列没有给字段指定类型那么es就会默认给我配置字段类型!

对比关系型数据库：

PUT test1/type1/1 ： 索引test1相当于关系型数据库的*库*，类型type1就相当于表， 1 代表数据中的主键 id

这里需要补充的是，在elastisearch5版本前，一个索引下可以创建多个类型，但是在elastisearch5后，一个索引只能对应一个类型，而id相当于关系型数据库的主键id若果不指定就会默认生成一个20位的uuid，属性相当关系型数据库的column(列)。

而结果中的 result 则是操作类型，现在是 created ，表示第一次创建。如果再次点击执行该命令那么 result 则会是 updated ，我们细心则会发现 _version 开始是1，现在你每点击一次就会增加一次。表示第几次更改。

7、我们在来学一条命令 (elasticsearch 中的索引的情况)：

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

返回结果：查看我们所有索引的状态健康情况 分片，数据储存大小等等。

```
health status index                    uuid                   pri rep docs.count docs.deleted store.size pri.store.size
yellow open   test2                    71TUZ84wRTW5P8lKeN4I4Q   1   1          0            0       283b           283b
yellow open   test3                    tpIP1z3wQz6Uqp2xtPrVvA   1   1          1            0      4.3kb          4.3kb
green  open   .kibana_task_manager_1   1mgzW7HzTJ-JXItsb2QdWA   1   0          2            0     49.2kb         49.2kb
green  open   .apm-agent-configuration UOPaBD02SEue8Eo4pqtphg   1   0          0            0       283b           283b
green  open   .kibana_1                AVFXLZTqSe-8v1SBmnUx5w   1   0         14            4     53.3kb         53.3kb
yellow open   test1                    yD_14X6uQyqe-6MBTXsM0w   1   1          1            0      3.7kb          3.7kb
```

8、那么怎么删除一条索引呢(库)呢?

```
DELETE /test1
```

返回：

```
{
  "acknowledged" : true   # 表示删除成功!
}
```

增删改查命令

> 创建数据PUT

第一条数据：

```
PUT /kuangshen/user/1
{
  "name":"狂神说",
  "age":18,
  "desc":"一顿操作猛如虎，一看工资2500",
  "tags":["直男","技术宅","温暖"]
}
```

第二条数据：

```
PUT /kuangshen/user/2
{
  "name":"张三",
  "age":3,
  "desc":"法外狂徒",
  "tags":["渣男","旅游","交友"]
}
```

第三条数据：

```
PUT /kuangshen/user/3
{
  "name":"李四",
  "age":30,
  "desc":"mmp，不知道怎么形容",
  "tags":["靓女","旅游","唱歌"]
}
```

查看下数据：

注意⚠：**当执行 命令时，如果数据不存在，则新增该条数据，如果数据存在则修改该条数据。**

咱们通过 GET 命令查询一下：

```
GET kuangshen/user/1
```

返回结果：

```
{
  "_index" : "kuangshen",
  "_type" : "user",
  "_id" : "1",
  "_version" : 1,
  "_seq_no" : 0,
  "_primary_term" : 1,
  "found" : true,
  "_source" : {
    "name" : "狂神说",
    "age" : 18,
    "desc" : "一顿操作猛如虎，一看工资2500",
    "tags" : [
      "直男",
      "技术宅",
      "温暖"
    ]
  }
}
```

如果你想更新数据 可以覆盖这条数据：

```
PUT /kuangshen/user/1
{
  "name":"狂神说Java",
  "age":18,
  "desc":"一顿操作猛如虎，一看工资2.5",
  "tags":["直男","技术宅","温暖"]
}
```

返回结果：

```
{
  "_index" : "kuangshen",
  "_type" : "user",
  "_id" : "1",
  "_version" : 2,
  "result" : "updated",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 3,
  "_primary_term" : 1
}
```

已经修改了 那么 **PUT** 可以更新数据但是。**麻烦的是 原数据你还要重写一遍要 这不符合我们规矩。**

> 更新数据 POST

我们使用 POST 命令，在 id 后面跟 **_update**，要修改的内容放到 **doc** 文档(属性)中即可。

```
POST /kuangshen/user/1/_update
{
  "doc":{
    "name":"狂神说Java",
    "desc":"关注狂神公众号每日更新文章哦"
  }
}
```

返回结果：

```
{
  "_index" : "kuangshen",
  "_type" : "user",
  "_id" : "1",
  "_version" : 3,
  "result" : "updated",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 4,
  "_primary_term" : 1
}
```

> 条件查询_search?q=

简单的查询，我们上面已经不知不觉的使用熟悉了：

```
GET kuangshen/user/1
```

我们来学习下条件查询 _search?q=

```
GET kuangshen/user/_search?q=name:狂神说
```

通过 _serarch?q=name:狂神说　查询条件是name属性有狂神说的那些数据。

别忘了 _search 和 from 属性中间的分隔符？。

返回结果：

```
{
  "took" : 16,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,              # 一共1条数据
      "relation" : "eq"
    },
    "max_score" : 1.4229509,
    "hits" : [
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "1",
        "_score" : 1.4229509,
        "_source" : {
          "name" : "狂神说Java",
          "age" : 18,
          "desc" : "关注狂神公众号每日更新文章哦",
          "tags" : [
            "直男",
            "技术宅",
            "温暖"
          ]
        }
      }
    ]
  }
}
```

我们看一下结果 返回并不是 数据本身，是给我们了一个 **hits** ，还有 _score得分，就是根据算法算出和查询条件匹配度高得分就搞。

> 构建查询

```
GET kuangshen/user/_search
{
  "query":{
    "match":{
      "name": "狂神"
    }
  }
}
```

上例，查询条件是一步步构建出来的，将查询条件添加到 match 中即可。返回结果还是一样的：

```
{
  "took" : 0,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 1.6285465,
    "hits" : [
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "1",
        "_score" : 1.6285465,
        "_source" : {
          "name" : "狂神说Java",
          "age" : 18,
          "desc" : "关注狂神公众号每日更新文章哦",
          "tags" : [
            "直男",
            "技术宅",
            "温暖"
          ]
        }
      }
    ]
  }
}
```

除此之外，我们还可以查询全部：

```
GET kuangshen/user/_search  #这是一个查询但是没有条件
```

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

match_all的值为空，表示没有查询条件，就像select * from table_name一样。

返回结果：全部查询出来了！

如果有个需求，**我们仅是需要查看 name 和 desc 两个属性**，其他的不要怎么办?

```
GET kuangshen/user/_search
{
  "query":{
    "match_all": {}
  },
  "_source": ["name","desc"]
}
```

如上例所示，在查询中，通过 _source 来控制仅返回 name 和 age 属性。

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 3,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "2",
        "_score" : 1.0,
        "_source" : {
          "name" : "张三",
          "desc" : "法外狂徒"
        }
      },
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "3",
        "_score" : 1.0,
        "_source" : {
          "name" : "李四",
          "desc" : "mmp，不知道怎么形容"
        }
```

```
      },
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "1",
        "_score" : 1.0,
        "_source" : {
          "name" : "狂神说Java",
          "desc" : "关注狂神公众号每日更新文章哦"
        }
      }
    ]
  }
}
```

一般的，我们推荐使用构建查询，以后在与程序交互时的查询等也是使用构建查询方式处理查询条件，因为该方 式可以构建更加复杂的查询条件，也更加一目了然

> 排序查询

我们说到排序 有人就会想到：正序 或 倒序 那么我们先来倒序：

```
GET kuangshen/user/_search
{
  "query":{
    "match_all": {}
  },
  "sort": [
    {
      "age": {
        "order": "desc"
      }
    }
  ]
}
```

上例，在条件查询的基础上，我们又通过 sort 来做排序，排序对象是 age ， order 是 desc 降序。

```
{
  "took" : 0,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 3,
      "relation" : "eq"
    },
    "max_score" : null,
    "hits" : [
      {
        "_index" : "kuangshen",
        "_type" : "user",
```

```
        "_id" : "3",
        "_score" : null,
        "_source" : {
          "name" : "李四",
          "age" : 30,
          "desc" : "mmp，不知道怎么形容",
          "tags" : [
            "靓女",
            "旅游",
            "唱歌"
          ]
        },
        "sort" : [
          30
        ]
      },
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "1",
        "_score" : null,
        "_source" : {
          "name" : "狂神说Java",
          "age" : 18,
          "desc" : "关注狂神公众号每日更新文章哦",
          "tags" : [
            "直男",
            "技术宅",
            "温暖"
          ]
        },
        "sort" : [
          18
        ]
      },
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "2",
        "_score" : null,
        "_source" : {
          "name" : "张三",
          "age" : 3,
          "desc" : "法外狂徒",
          "tags" : [
            "渣男",
            "旅游",
            "交友"
          ]
        },
        "sort" : [
          3
        ]
      }
    ]
  }
}
```

正序，就是 desc 换成了 asc

```
GET kuangshen/user/_search
{
  "query":{
    "match_all": {}
  },
  "sort": [
    {
      "age": {
        "order": "asc"
      }
    }
  ]
}
```

注意:在排序的过程中，只能使用可排序的属性进行排序。那么可以排序的属性有哪些呢?

- 数字
- 日期
- ID

其他都不行!

> 分页查询

```
GET kuangshen/user/_search
{
  "query":{
    "match_all": {}
  },
  "sort": [
    {
      "age": {
        "order": "asc"
      }
    }
  ],
  "from": 0, # 从第n条开始
  "size": 1  # 返回n条数据
}
```

返回结果:

```
{
  "took" : 0,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 3,
      "relation" : "eq"
```

```
    },
    "max_score" : null,
    "hits" : [
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "2",
        "_score" : null,
        "_source" : {
          "name" : "张三",
          "age" : 3,
          "desc" : "法外狂徒",
          "tags" : [
            "渣男",
            "旅游",
            "交友"
          ]
        },
        "sort" : [
          3
        ]
      }
    ]
  }
}
```

就返回了一条数据 是从第0条开始的返回一条数据 。可以再测试！

学到这里，我们也可以看到，我们的查询条件越来越多，开始仅是简单查询，慢慢增加条件查询，增加排序，对返回 结果进行限制。所以，我们可以说:**对elasticsearch于 来说，所有的查询条件都是可插拔的，彼此之间用 分 割。比如说，我们在查询中，仅对返回结果进行限制:**

```
GET kuangshen/user/_search
{
  "query":{
    "match_all": {}
  },

  "from": 0, # 从第n条开始
  "size": 1  # 返回n条数据
}
```

> 布尔查询

先增加一个数据:

```
PUT /kuangshen/user/4
{
  "name":"狂神说",
  "age":3,
  "desc":"一顿操作猛如虎，一看工资2500",
  "tags":["直男","技术宅","温暖"]
}
```

must (and)

我要查询所有 name 属性为“ 狂神 “的数据，并且年龄为18岁的！

```
GET kuangshen/user/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "name": "狂神说"
          }
        },
        {
          "match": {
            "age": 3
          }
        }
      ]
    }
  }
}
```

我们通过在 bool 属性内使用 must 来作为查询条件！看结果，是不是 有点像 **and** 的感觉，里面的条件需要都满足！

should (or)

那么我要查询name为狂神 或 age 为18 的呢?

```
GET kuangshen/user/_search
{
  "query": {
    "bool": {
      "should": [
        {
          "match": {
            "name": "狂神说"
          }
        },
        {
          "match": {
            "age": 18
          }
        }
      ]
    }
  }
}
```

返回结果：

```
{
  "took" : 0,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
```

```
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
      "relation" : "eq"
    },
    "max_score" : 3.1522982,
    "hits" : [
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "1",
        "_score" : 3.1522982,
        "_source" : {
          "name" : "狂神说Java",
          "age" : 18,
          "desc" : "关注狂神公众号每日更新文章哦",
          "tags" : [
            "直男",
            "技术宅",
            "温暖"
          ]
        }
      },
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "4",
        "_score" : 2.4708953,
        "_source" : {
          "name" : "狂神说",
          "age" : 3,
          "desc" : "一顿操作猛如虎，一看工资2500",
          "tags" : [
            "直男",
            "技术宅",
            "温暖"
          ]
        }
      }
    ]
  }
}

```


我们的返回结果 是不是 出现了一个 age : 3的。是不是有点像 **or** 呢

must_not (not)

我想要查询 年龄不是 18 的 数据

```
GET kuangshen/user/_search
{
  "query": {
    "bool": {
      "must_not": [
        {
```

```
          "match": {
            "age": 18
          }
        }
      ]
    }
  }
}
```

Fitter

我要查询 name 为狂神 的，age大于10的数据

```
GET kuangshen/user/_search
{
  "query":{
    "bool": {
      "must": [
        {
          "match": {
            "name": "狂神"
          }
        }
      ],
      "filter": {
        "range": {
          "age": {
            "gt": 10
          }
        }
      }
    }
  }
}
```

这里就用到了 filter 条件过滤查询，过滤条件的范围用 range 表示， gt 表示大于，大于多少呢?是10。
其余操作如下：

- gt 表示大于
- gte 表示大于等于
- lt 表示小于
- lte 表示小于等于

要查询 name 是 狂神， age 在 25~30 之间的怎么查?

```
GET kuangshen/user/_search
{
  "query":{
    "bool": {
      "must": [
        {
          "match": {
            "name": "狂神"
          }
        }
      ],
      "filter": {
        "range": {
```

```
          "age": {
            "gte": 25,
            "lte": 30
          }
        }
      }
    }
  }
}
```

> 短语检索

我要查询 tags为男的数据

```
GET kuangshen/user/_search
{
  "query":{
    "match": {
      "tags": "男"
    }
  }
}
```

返回了所有标签中带 男 的记录!

既然按照标签检索，那么，能不能写多个标签呢?又该怎么写呢?

```
GET kuangshen/user/_search
{
  "query":{
    "match": {
      "tags": "男 技术"
    }
  }
}
```

返回：只要含有这个标签满足一个就给我返回这个数据了。

> term查询精确查询

`term`查询是直接通过倒排索引指定的 词条，也就是精确查找。

term和match的区别:

- match是经过分析(analyer)的，也就是说，文档是先被分析器处理了，根据不同的分析器，分析出的结果也会不同，在会根据分词 结果进行匹配。
- term是不经过分词的，直接去倒排索引查找精确的值。

注意 ⚠：我们现在 用的es7版本 所以我们用 mappings properties 去给多个字段(fields)指定类型的时候,不能给我们的 索引制定类型:

```
PUT testdb
{
  "mappings": {
```

```
    "properties": {
      "name":{
        "type":"text"
      },
      "desc":{
        "type": "keyword"
      }
    }
  }
}
// 插入数据
PUT testdb/_doc/1
{
  "name": "狂神说Java name",
  "desc": "狂神说Java desc"
}
PUT testdb/_doc/2
{
  "name": "狂神说Java name",
  "desc": "狂神说Java desc2"
}
```

上述中testdb索引中,字段name在被查询时会被分析器进行分析后匹配查询。而**属于keyword类型不会被分析器处理。**

我们来验证一下:

```
GET _analyze
{
  "analyzer": "keyword",
  "text": "狂神说Java name"
}
```

结果:

```
{
  "tokens" : [
    {
      "token" : "狂神说Java name",
      "start_offset" : 0,
      "end_offset" : 12,
      "type" : "word",
      "position" : 0
    }
  ]
}
```

是不是没有被分析啊。**就是简单的一个字符串啊**。再测试

```
GET _analyze
{
  "analyzer": "standard",
  "text": "狂神说Java name"
}
```

结果:

```
{
```

```
  "tokens" : [
    {
      "token" : "狂",
      "start_offset" : 0,
      "end_offset" : 1,
      "type" : "<IDEOGRAPHIC>",
      "position" : 0
    },
    {
      "token" : "神",
      "start_offset" : 1,
      "end_offset" : 2,
      "type" : "<IDEOGRAPHIC>",
      "position" : 1
    },
    {
      "token" : "说",
      "start_offset" : 2,
      "end_offset" : 3,
      "type" : "<IDEOGRAPHIC>",
      "position" : 2
    },
    {
      "token" : "java",
      "start_offset" : 3,
      "end_offset" : 7,
      "type" : "<ALPHANUM>",
      "position" : 3
    },
    {
      "token" : "name",
      "start_offset" : 8,
      "end_offset" : 12,
      "type" : "<ALPHANUM>",
      "position" : 4
    }
  ]
}
```

那么我们看一下 们字符串是不是被分析了啊。

总结：keyword 字段类型不会被分析器分析！

现在我们来查询一下：

```
GET testdb/_search   // text 会被分析器分析 查询
{
  "query": {
    "term": {
      "name": "狂"
    }
  }
}

GET testdb/_search     // keyword 不会被分析所以直接查询
{
  "query": {
    "match": {
```

```
      "desc":"狂神说Java desc"
    }
  }
}
```

查找多个精确值(terms)

官网地址：https://www.elastic.co/guide/cn/elasticsearch/guide/current/_finding_multiple_exact_values.html

```
PUT testdb/_doc/3
{
  "t1": "22",
  "t2": "2020-4-16"
}
PUT testdb/_doc/4
{
  "t1": "33",
  "t2": "2020-4-17"
}

# 查询  精确查找多个值
GET testdb/_search
{
  "query": {
    "bool": {
      "should": [
        {
          "term": {
            "t1": "22"
          }
        },
        {
          "term": {
            "t1": "33"
          }
        }
      ]
    }
  }
}
```

除了bool查询之外：

```
GET testdb/_doc/_search
{
  "query": {
    "terms": {
      "t1": ["22", "33"]
    }
  }
}
```

> 高亮显示

```
GET kuangshen/user/_search
{
  "query":{
    "match": {
      "name": "狂神"
    }
  },
  "highlight" :{
    "fields": {
      "name":{}
    }
  }
}
```

返回结果：

```
#! Deprecation: [types removal] Specifying types in search requests is deprecated.
{
  "took" : 62,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
      "relation" : "eq"
    },
    "max_score" : 1.6472635,
    "hits" : [
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "4",
        "_score" : 1.6472635,
        "_source" : {
          "name" : "狂神说",
          "age" : 3,
          "desc" : "一顿操作猛如虎，一看工资2500",
          "tags" : [
            "直男",
            "技术宅",
            "温暖"
          ]
        },
        "highlight" : {
          "name" : [
            "<em>狂</em><em>神</em>说"
          ]
        }
      },
      {
        "_index" : "kuangshen",
```

```
        "_type" : "user",
        "_id" : "1",
        "_score" : 1.4348655,
        "_source" : {
          "name" : "狂神说Java",
          "age" : 18,
          "desc" : "关注狂神公众号每日更新文章哦",
          "tags" : [
            "直男",
            "技术宅",
            "温暖"
          ]
        },
        "highlight" : {
          "name" : [
            "<em>狂</em><em>神</em>说Java"
          ]
        }
      }
    ]
  }
}
```

我们可以看到 已 < em>狂神< /em>经帮我们加上了一个< em>标签

这是es帮我们加的标签。那我·**也可以自己自定义样式**

```
GET kuangshen/user/_search
{
  "query":{
    "match": {
      "name": "狂神"
    }
  },
  "highlight" :{
    "pre_tags": "<b class='key' style='color:red'>",
    "post_tags": "</b>",
    "fields": {
      "name":{}
    }
  }
}
```

结果:

```
#! Deprecation: [types removal] Specifying types in search requests is deprecated.
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
```

```
      "value" : 2,
      "relation" : "eq"
    },
    "max_score" : 1.6472635,
    "hits" : [
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "4",
        "_score" : 1.6472635,
        "_source" : {
          "name" : "狂神说",
          "age" : 3,
          "desc" : "一顿操作猛如虎，一看工资2500",
          "tags" : [
            "直男",
            "技术宅",
            "温暖"
          ]
        },
        "highlight" : {
          "name" : [
            "<b class='key' style='color:red'>狂</b><b class='key'
style='color:red'>神</b>说"
          ]
        }
      },
      {
        "_index" : "kuangshen",
        "_type" : "user",
        "_id" : "1",
        "_score" : 1.4348655,
        "_source" : {
          "name" : "狂神说Java",
          "age" : 18,
          "desc" : "关注狂神公众号每日更新文章哦",
          "tags" : [
            "直男",
            "技术宅",
            "温暖"
          ]
        },
        "highlight" : {
          "name" : [
            "<b class='key' style='color:red'>狂</b><b class='key'
style='color:red'>神</b>说Java"
          ]
        }
      }
    ]
  }
}
```

需要注意的是:自定义标签中属性或样式中的逗号一律用英文状态的单引号表示，应该与外部 es 语法 的双引号区分开。

> 说明：Deprecation

注意 elasticsearch 在第一个版本的开始 每个文档都储存在一个索引中，并分配一个 映射类型，映射类型用于表示被索引的文档或者实体的类型，这样带来了一些问题，导致后来在 elasticsearch6.0.0 版本中一个文档只能包含一个映射类型，而在 7.0.0 中，映 射类型则将被弃用，到了 8.0.0 中则将完全被删除。

只要记得，一个索引下面只能创建一个类型就行了，其中各字段都具有唯一性，如果在创建映射的时候，如果没有指定文档类型，那么该索引的默认索引类型是`_doc`，不指定文档id则会内部帮我们生成一个id字符串。

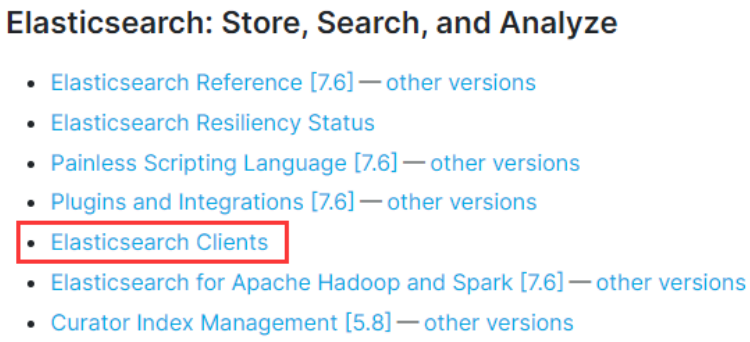
API创建索引及文档

> 找文档

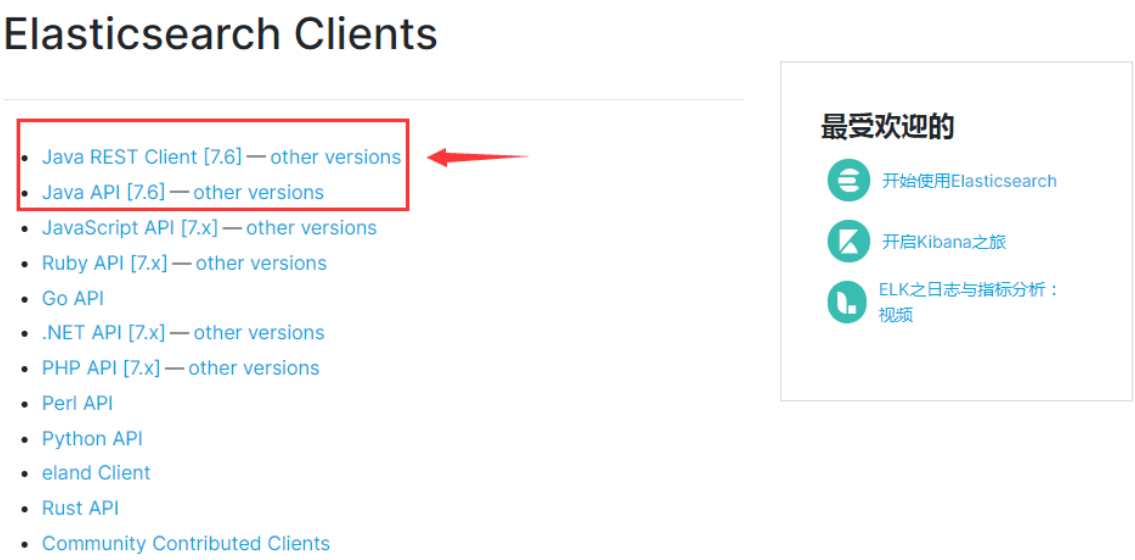
网上的es教程大都十分老旧，而且es的版本众多，个别版本的差异还较大，另外es本身提供多种api，导致许多文章各种乱七八糟实例！所以后面直接放弃，从官网寻找方案，这里我使用elasticsearch最新的7.6.1版本来讲解：

1、进入es的官网指导文档 https://www.elastic.co/guide/index.html

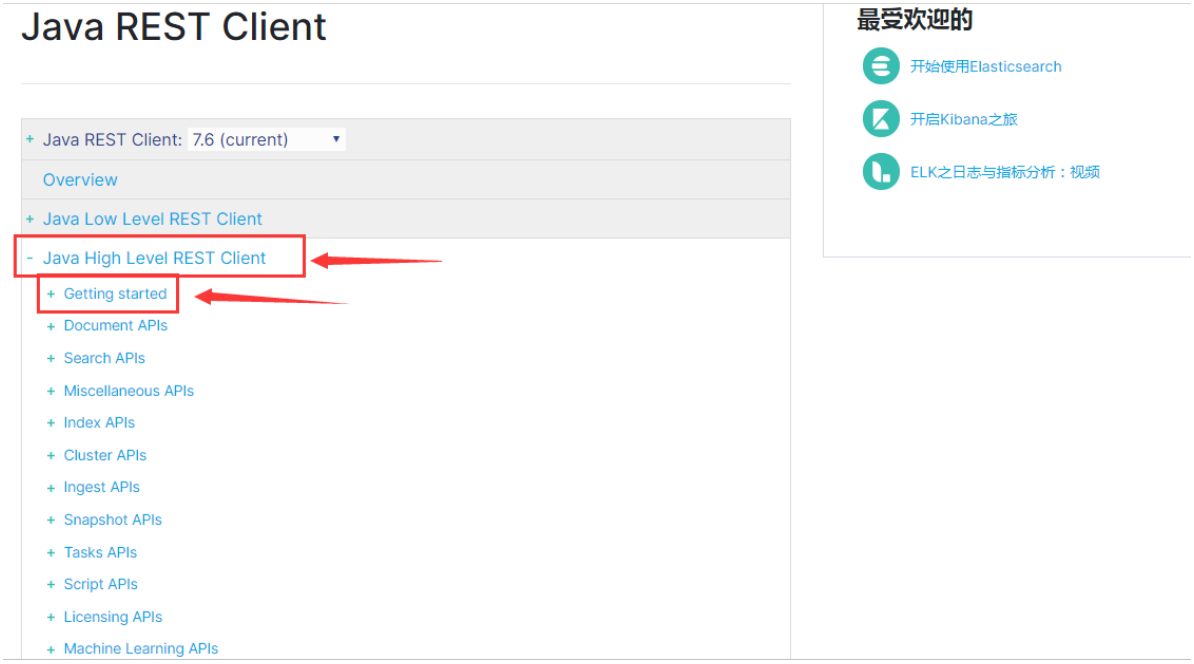
2、找到 Elasticsearch Clients（这个就是客户端api文档）



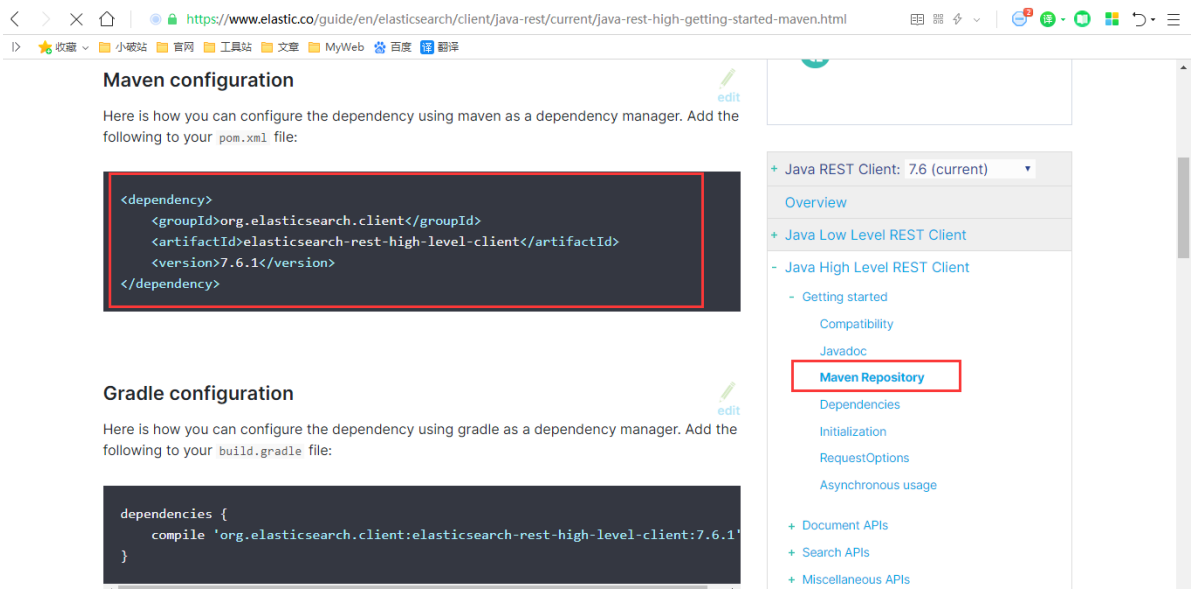
3、我们使用java rest风格api，大家可以更加自己的版本选择特定的other versions。



4、rest又分为high level和low level，我们直接选择high level下面的 Getting started



5、向下阅读找到Maven依赖和基本配置！



> Java REST Client 说明

Java REST Client 有两种风格：

Java Low Level REST Client：用于Elasticsearch的官方低级客户端。它允许通过http与Elasticsearch集群通信。将请求编排和响应反编排留给用户自己处理。它兼容所有的Elasticsearch版本。（PS：学过WebService的话，对编排与反编排这个概念应该不陌生。可以理解为对请求参数的封装，以及对响应结果的解析）

Java High Level REST Client：用于Elasticsearch的官方高级客户端。它是基于低级客户端的，它提供很多API，并负责请求的编排与响应的反编排。（PS：就好比是，一个是传自己拼接好的字符串，并且自己解析返回的结果；而另一个是传对象，返回的结果也已经封装好了，直接是对象，更加规范了参数的名称以及格式，更加面对对象一点）

（PS：所谓低级与高级，我觉得一个很形象的比喻是，面向过程编程与面向对象编程）

网上很多教程比较老旧，都是使用TransportClient操作的，在 Elasticsearch 7.0 中不建议使用TransportClient，并且在8.0中会完全删除TransportClient。因此，官方更建议我们用Java High Level REST Client，它执行HTTP请求，而不是序列号的Java请求。既然如此，这里我们就直接用高级了。

> 配置基本项目依赖

1、新建一个springboot（2.2.5版）项目`kuang-elasticsearch`，导入web依赖即可！

2、配置es的依赖！

```xml
<properties>
    <java.version>1.8</java.version>
    <!-- 这里SpringBoot默认配置的版本不匹配，我们需要自己配置版本！ -->
    <elasticsearch.version>7.6.1</elasticsearch.version>
</properties>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
</dependency>
```

3、继续阅读文档到Initialization ，我们看到需要构建RestHighLevelClient对象；

```
RestHighLevelClient client = new RestHighLevelClient(
        RestClient.builder(
                new HttpHost("localhost", 9200, "http"),
                new HttpHost("localhost", 9201, "http"))); // 构建客户端对象

// 操作....
// 高级客户端内部会创建低级客户端用于基于提供的builder执行请求。低级客户端维护一个连接池，并启动一些线程，因此当你用完以后应该关闭高级客户端，并且在内部它将会关闭低级客户端，以释放这些资源。关闭客户端可以使用close()方法：
client.close(); // 关闭
```

4、我们编写一个配置类，提供这个bean来进行操作

```
package com.kuang.config;

import org.apache.http.HttpHost;
import org.elasticsearch.client.RestClient;
import org.elasticsearch.client.RestHighLevelClient;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ElasticsearchClientConfig {

    @Bean
    public RestHighLevelClient restHighLevelClient() {
        RestHighLevelClient client = new RestHighLevelClient(
                RestClient.builder(
                        new HttpHost("127.0.0.1", 9200, "http")));
        return client;
    }

```

```
}
```

5、常用方法工具类封装

```
package com.kuang.utils;

import com.alibaba.fastjson.JSON;
import org.elasticsearch.action.admin.indices.delete.DeleteIndexRequest;
import org.elasticsearch.action.bulk.BulkRequest;
import org.elasticsearch.action.bulk.BulkResponse;
import org.elasticsearch.action.delete.DeleteRequest;
import org.elasticsearch.action.delete.DeleteResponse;
import org.elasticsearch.action.get.GetRequest;
import org.elasticsearch.action.get.GetResponse;
import org.elasticsearch.action.index.IndexRequest;
import org.elasticsearch.action.index.IndexResponse;
import org.elasticsearch.action.search.SearchRequest;
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.action.support.master.AcknowledgedResponse;
import org.elasticsearch.action.update.UpdateRequest;
import org.elasticsearch.action.update.UpdateResponse;
import org.elasticsearch.client.RequestOptions;
import org.elasticsearch.client.RestHighLevelClient;
import org.elasticsearch.client.indices.CreateIndexRequest;
import org.elasticsearch.client.indices.CreateIndexResponse;
import org.elasticsearch.client.indices.GetIndexRequest;
import org.elasticsearch.common.unit.TimeValue;
import org.elasticsearch.common.xcontent.XContentType;
import org.elasticsearch.index.query.QueryBuilders;
import org.elasticsearch.rest.RestStatus;
import org.elasticsearch.search.builder.SearchSourceBuilder;
import org.elasticsearch.search.fetch.subphase.FetchSourceContext;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.util.List;
import java.util.concurrent.TimeUnit;

@Component
public class EsUtils<T> {

    @Autowired
    @Qualifier("restHighLevelClient")
    private RestHighLevelClient client;
    /**
     * 判断索引是否存在
     * @param index
     * @return
     * @throws IOException
     */
    public boolean existsIndex(String index) throws IOException {
        GetIndexRequest request = new GetIndexRequest(index);
        boolean exists = client.indices().exists(request,
RequestOptions.DEFAULT);
        return exists;
    }
```

```
    /**
     * 创建索引
     * @param index
     * @throws IOException
     */
    public boolean createIndex(String index) throws IOException {
        CreateIndexRequest request = new CreateIndexRequest(index);
        CreateIndexResponse createIndexResponse
 =client.indices().create(request,RequestOptions.DEFAULT);
        return createIndexResponse.isAcknowledged();
    }

    /**
     * 删除索引
     * @param index
     * @return
     * @throws IOException
     */
    public boolean deleteIndex(String index) throws IOException {
        DeleteIndexRequest deleteIndexRequest = new
DeleteIndexRequest(index);
        AcknowledgedResponse response =
client.indices().delete(deleteIndexRequest, RequestOptions.DEFAULT);
        return response.isAcknowledged();

    }

    /**
     * 判断某索引下文档id是否存在
     * @param index
     * @param id
     * @return
     * @throws IOException
     */
    public boolean docExists(String index, String id) throws IOException {
        GetRequest getRequest = new GetRequest(index,id);
        //只判断索引是否存在不需要获取_source
        getRequest.fetchSourceContext(new FetchSourceContext(false));
        getRequest.storedFields("_none_");
        boolean exists = client.exists(getRequest, RequestOptions.DEFAULT);
        return  exists;
    }

    /**
     * 添加文档记录
     * @param index
     * @param id
     * @param t 要添加的数据实体类
     * @return
     * @throws IOException
     */
    public boolean addDoc(String index,String id,T t) throws IOException {

        IndexRequest request = new IndexRequest(index);
        request.id(id);
        //timeout
        request.timeout(TimeValue.timeValueSeconds(1));
        request.timeout("1s");
```

```
        request.source(JSON.toJSONString(t), XContentType.JSON);
        IndexResponse indexResponse = client.index(request,
RequestOptions.DEFAULT);
        RestStatus Status = indexResponse.status();
        return Status==RestStatus.OK||Status== RestStatus.CREATED;
    }

    /**
     * 根据id来获取记录
     * @param index
     * @param id
     * @return
     * @throws IOException
     */
    public GetResponse getDoc(String index, String id) throws IOException {
        GetRequest request = new GetRequest(index,id);
        GetResponse getResponse = client.get(request,
RequestOptions.DEFAULT);
        return getResponse;
    }

    /**
     * 批量添加文档记录
     * 没有设置id ES会自动生成一个，如果要设置 IndexRequest的对象.id()即可
     * @param index
     * @param list
     * @return
     * @throws IOException
     */
    public boolean bulkAdd(String index, List<T> list) throws IOException {
        BulkRequest bulkRequest = new BulkRequest();
        //timeout
        bulkRequest.timeout(TimeValue.timeValueMinutes(2));
        bulkRequest.timeout("2m");

        for (int i =0;i<list.size();i++){
            bulkRequest.add(new IndexRequest(index)
                    .source(JSON.toJSONString(list.get(i))));
        }
        BulkResponse bulkResponse = client.bulk(bulkRequest,
RequestOptions.DEFAULT);

        return !bulkResponse.hasFailures();

    }

    /**
     * 批量删除和更新就不写了可根据上面几个方法来写
     */

    /**
     * 更新文档记录
     * @param index
     * @param id
     * @param t
     * @return
     * @throws IOException
     */
```

```
    public boolean updateDoc(String index,String id,T t) throws IOException {
        UpdateRequest request = new UpdateRequest(index,id);
        request.doc(JSON.toJSONString(t));
        request.timeout(TimeValue.timeValueSeconds(1));
        request.timeout("1s");
        UpdateResponse updateResponse = client.update(
                request, RequestOptions.DEFAULT);
        return updateResponse.status()==RestStatus.OK;
    }


    /**
     * 删除文档记录
     * @param index
     * @param id
     * @return
     * @throws IOException
     */
    public boolean deleteDoc(String index,String id) throws IOException {
        DeleteRequest request = new DeleteRequest(index,id);
        //timeout
        request.timeout(TimeValue.timeValueSeconds(1));
        request.timeout("1s");
        DeleteResponse deleteResponse = client.delete(
                request, RequestOptions.DEFAULT);

        return deleteResponse.status()== RestStatus.OK;
    }

    /**
     * 根据某字段来搜索
     * @param index
     * @param field
     * @param key 要收搜的关键字
     * @throws IOException
     */
    public void search(String index,String field ,String key,Integer from,Integer size) throws IOException {
        SearchRequest searchRequest = new SearchRequest(index);
        SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
        sourceBuilder.query(QueryBuilders.termQuery(field, key));
        //控制搜索
        sourceBuilder.from(from);
        sourceBuilder.size(size);
        //最大搜索时间。
        sourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));
        searchRequest.source(sourceBuilder);
        SearchResponse searchResponse = client.search(searchRequest, RequestOptions.DEFAULT);
        System.out.println(JSON.toJSONString(searchResponse.getHits()));

    }

}
```

> APIs 测试

测试创建索引：

```
@Test
void testCreateIndex() throws IOException {
    CreateIndexRequest request = new CreateIndexRequest("kuang_index");
    CreateIndexResponse createIndexResponse

 =restHighLevelClient.indices().create(request,RequestOptions.DEFAULT);
    System.out.println(createIndexResponse);
}
```

测试获取索引：

```
@Test
void testExistsIndex() throws IOException {
    GetIndexRequest request = new GetIndexRequest("kuang_index");
    boolean exists = restHighLevelClient.indices().exists(request,
RequestOptions.DEFAULT);
    System.out.println(exists);
}
```

测试删除索引：

```
@Test
void testDeleteIndexRequest() throws IOException {
    DeleteIndexRequest deleteIndexRequest = new
DeleteIndexRequest("kuang_index");
    AcknowledgedResponse response
        = restHighLevelClient.indices().delete(deleteIndexRequest,
RequestOptions.DEFAULT);
    System.out.println(response.isAcknowledged());
}
```

测试添加文档记录：

创建一个实体类User

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Component
public class User {
    private String name;
    private int age;
}
```

测试添加文档记录

```
@Test
void testAddDocument() throws IOException {
    // 创建对象
    User user = new User("狂神说", 3);
    // 创建请求
    IndexRequest request = new IndexRequest("kuang_index");
    // 规则
    request.id("1");
```

```
    request.timeout(TimeValue.timeValueSeconds(1));
    request.timeout("1s");
    request.source(JSON.toJSONString(user), XContentType.JSON);
    // 发送请求
    IndexResponse indexResponse = restHighLevelClient.index(request,
RequestOptions.DEFAULT);
    System.out.println(indexResponse.toString());
    RestStatus Status = indexResponse.status();
    System.out.println(Status == RestStatus.OK || Status ==
RestStatus.CREATED);
}
```

测试：判断某索引下文档id是否存在

```
// 判断此id是否存在这个索引库中
@Test
void testIsExists() throws IOException {
    GetRequest getRequest = new GetRequest("kuang_index","1");
    // 不获取_source上下文  storedFields
    getRequest.fetchSourceContext(new FetchSourceContext(false));
    getRequest.storedFields("_none_");
    // 判断此id是否存在!
    boolean exists = restHighLevelClient.exists(getRequest,
RequestOptions.DEFAULT);
    System.out.println(exists);
}
```

测试：根据id获取记录

```
// 获得文档记录
@Test
void testGetDocument() throws IOException {
    GetRequest getRequest = new GetRequest("kuang_index","3");
    GetResponse getResponse = restHighLevelClient.get(getRequest,
RequestOptions.DEFAULT);
    System.out.println(getResponse.getSourceAsString()); // 打印文档内容
    System.out.println(getResponse);
}
```

测试：更新文档记录

```
// 更新文档记录
@Test
void testUpdateDocument() throws IOException {
    UpdateRequest request = new UpdateRequest("kuang_index","1");
    request.timeout(TimeValue.timeValueSeconds(1));
    request.timeout("1s");

    User user = new User("狂神说", 18);
    request.doc(JSON.toJSONString(user), XContentType.JSON);

    UpdateResponse updateResponse = restHighLevelClient.update(
        request, RequestOptions.DEFAULT);

    System.out.println(updateResponse.status() == RestStatus.OK);
}
```

测试：删除文档记录

```java
// 删除文档测试
@Test
void testDelete() throws IOException {
    DeleteRequest request = new DeleteRequest("kuang_index","3");
    //timeout
    request.timeout(TimeValue.timeValueSeconds(1));
    request.timeout("1s");

    DeleteResponse deleteResponse = restHighLevelClient.delete(
        request, RequestOptions.DEFAULT);
    System.out.println(deleteResponse.status() == RestStatus.OK);
}
```

测试：批量添加文档

```java
// 批量添加数据
@Test
void testBulkRequest() throws IOException {
    BulkRequest bulkRequest = new BulkRequest();
    //timeout
    bulkRequest.timeout(TimeValue.timeValueMinutes(2));
    bulkRequest.timeout("2m");

    ArrayList<User> userList = new ArrayList<>();
    userList.add(new User("kuangshen1",3));
    userList.add(new User("kuangshen2",3));
    userList.add(new User("kuangshen3",3));
    userList.add(new User("qinjiang1",3));
    userList.add(new User("qinjiang2",3));
    userList.add(new User("qinjiang3",3));

    for (int i =0;i<userList.size();i++){
        bulkRequest
            .add(new IndexRequest("kuang_index")
                 .id(""+(i+1))

.source(JSON.toJSONString(userList.get(i)),XContentType.JSON));
    }
    // bulk
    BulkResponse bulkResponse =
restHighLevelClient.bulk(bulkRequest,RequestOptions.DEFAULT);

    System.out.println(!bulkResponse.hasFailures());
}
```

查询测试：

```java
// 查询测试
/**
     * 使用QueryBuilder
     * termQuery("key", obj) 完全匹配
     * termsQuery("key", obj1, obj2..)   一次匹配多个值
     * matchQuery("key", Obj) 单个匹配, field不支持通配符, 前缀具高级特性
     * multiMatchQuery("text", "field1", "field2"..);  匹配多个字段, field有通
配符忒行
     * matchAllQuery();         匹配所有文件
     */
```

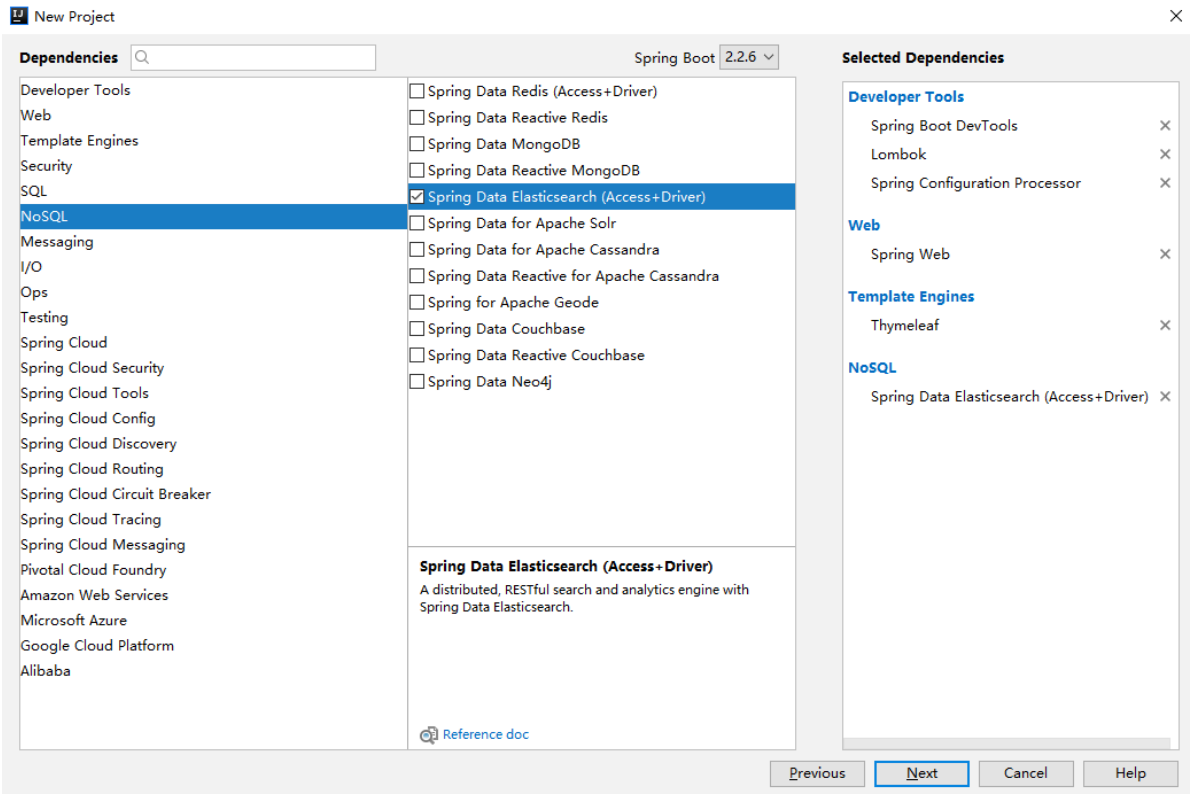
```java
@Test
void testSearch() throws IOException {
    SearchRequest searchRequest = new SearchRequest("kuang_index");
    SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();

    // TermQueryBuilder termQueryBuilder = QueryBuilders.termQuery("name",
"qinjiang1");
    MatchAllQueryBuilder matchAllQueryBuilder =
QueryBuilders.matchAllQuery();
    sourceBuilder.query(matchAllQueryBuilder);

    sourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));
    searchRequest.source(sourceBuilder);

    SearchResponse response = restHighLevelClient.search(searchRequest,
RequestOptions.DEFAULT);
    System.out.println(JSON.toJSONString(response.getHits()));
    System.out.println("================SearchHit=================");
    for (SearchHit documentFields : response.getHits().getHits()) {
        System.out.println(documentFields.getSourceAsMap());
    }
}
```

实战测试

初始化项目

1、启动es服务和客户端

2、使用springboot快速构建服务



3、修改版本依赖!

```xml
<properties>
    <java.version>1.8</java.version>
    <!-- 这里SpringBoot默认配置的版本不匹配，我们需要自己配置版本！ -->
    <elasticsearch.version>7.6.1</elasticsearch.version>
</properties>
```

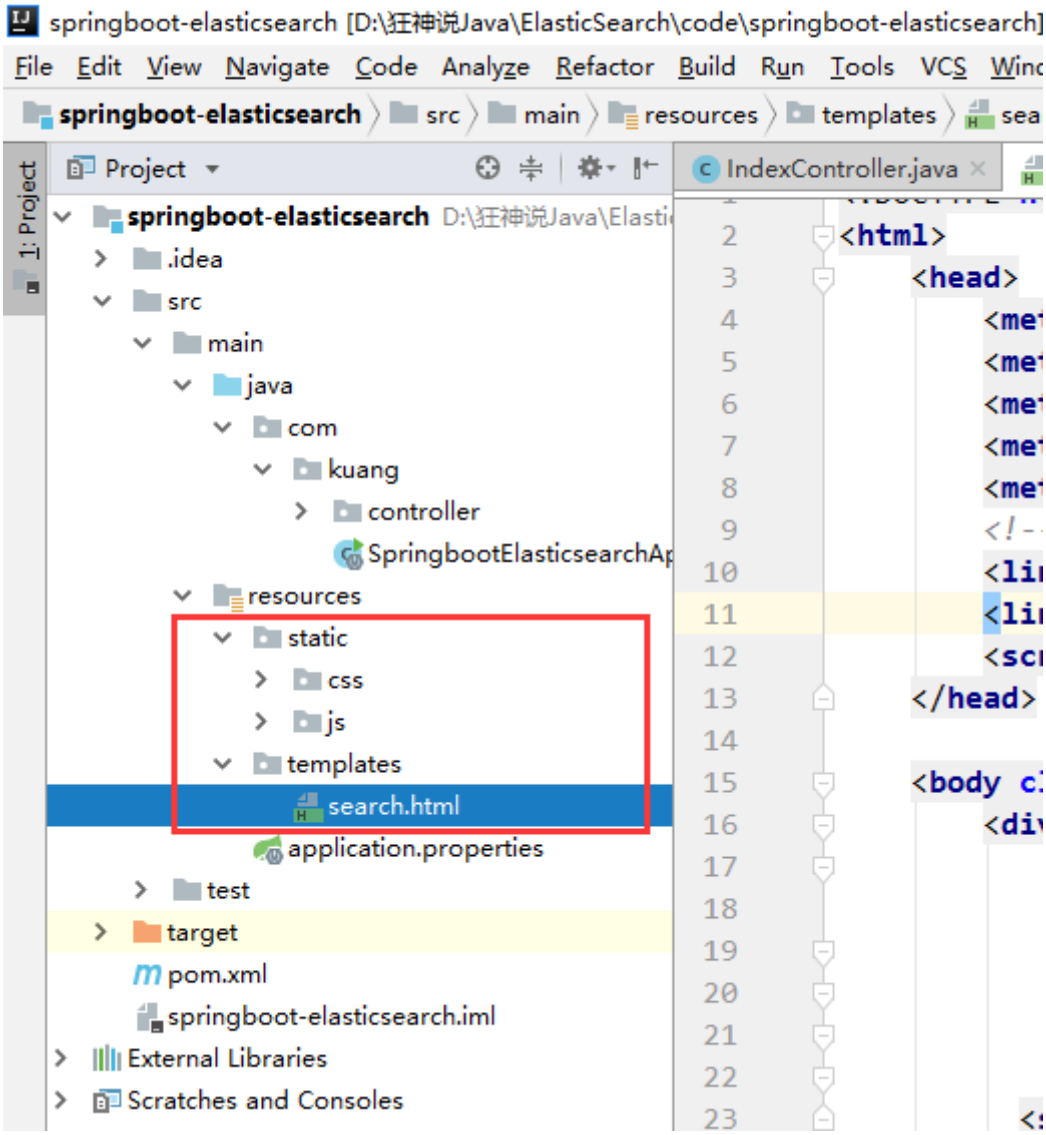
4、配置`application.properties`文件

```properties
server.port=9090
# 关闭thymeleaf缓存
spring.thymeleaf.cache=false
```

5、导入前端的素材！修改为Thymeleaf支持的格式！

```xml
<html xmlns:th="http://www.thymeleaf.org">
```

6、编写IndexController进行跳转测试！



jsoup讲解

1、导入jsoup的依赖

```xml
<!-- https://mvnrepository.com/artifact/org.jsoup/jsoup -->
<dependency>
    <groupId>org.jsoup</groupId>
    <artifactId>jsoup</artifactId>
    <version>1.13.1</version>
</dependency>
```

2、编写一个工具类 HtmlParseUtil

```java
public class HtmlParseUtil {
    public static void main(String[] args) throws IOException {
        // jsoup不能抓取ajax的请求，除非自己模拟浏览器进行请求！

        // 1、https://search.jd.com/Search?keyword=java
        String url = "https://search.jd.com/Search?keyword=java";
        // 2、解析网页（需要联网）
        Document document = Jsoup.parse(new URL(url), 30000);

        // 3、抓取搜索到的数据！
        // Document 就是我们JS的Document对象，你可以看到很多JS语法
        Element element = document.getElementById("J_goodsList");

        // 4、找到所有的li元素
        Elements elements = element.getElementsByTag("li");

        // 获取京东的商品信息
        for (Element el : elements) {
            //  这种网站，一般为了保证效率，一般会延时加载图片
            //  String img = el.getElementsByTag("img").eq(0).attr("src");
            String img = el.getElementsByTag("img").eq(0).attr("source-data-lazy-img");
            String price = el.getElementsByClass("p-price").eq(0).text();
            String title = el.getElementsByClass("p-name").eq(0).text();

            System.out.println(img);
            System.out.println(price);
            System.out.println(title);
            System.out.println("==================================");
        }
    }
}
```

3、封装一个实体类保存爬取下来的数据

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Content {
    private String title; // 商品名称
    private String price; // 商品价格
    private String img;   // 商品封面
    // 大家可以自行扩展使用
}
```

4、封装为工具使用！

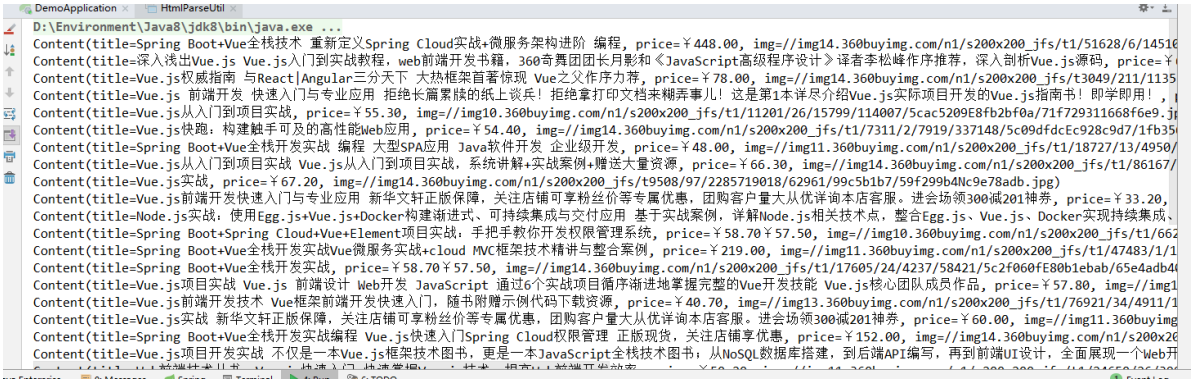
```java
/**
```

```
     * @author 狂神说Java  公众号：狂神说
     * @param keywords 要搜索的关键字!
     * @return 抓取的商品集合
     */
public List<Content> parseJD(String keywords) throws Exception {
    String url = "https://search.jd.com/Search?keyword="+keywords;
    Document document = Jsoup.parse(new URL(url), 30000);
    Element element = document.getElementById("J_goodsList");
    Elements elements = element.getElementsByTag("li");

    ArrayList<Content> goodsList = new ArrayList<>();

    // 获取京东的商品信息
    for (Element el : elements) {
        String img = el.getElementsByTag("img").eq(0).attr("source-data-lazy-img");
        String price = el.getElementsByClass("p-price").eq(0).text();
        String title = el.getElementsByClass("p-name").eq(0).text();
        // 封装获取的数据
        Content content = new Content();
        content.setImg(img);
        content.setPrice(price);
        content.setTitle(title);
        goodsList.add(content);
    }
    return goodsList;
}
```

5、测试工具类的使用!

```
public static void main(String[] args) throws Exception {
    new HtmlParseUtil().parseJD("vue").forEach(System.out::println);
}
```

搞定收工！简单爬虫编写完毕！我们这里的数据就使用爬取的即可，平时开发es的数据可能来自多个地方，你们可以从数据库查询获取也是一样的，后面我们来测试下效果!



业务编写

1、导入ElasticsearchClientConfig 配置类

```java
@Configuration
public class ElasticsearchClientConfig {

    @Bean
    public RestHighLevelClient restHighLevelClient() {
        RestHighLevelClient client = new RestHighLevelClient(
                RestClient.builder(
                        new HttpHost("127.0.0.1", 9200, "http")));
        return client;
    }

}
```

2、编写业务！

```java
@Service
public class ContentService {

    @Autowired
    private RestHighLevelClient restHighLevelClient;

    // 1、解析数据存入es
    public Boolean parseContent(String keywords) throws Exception {
        // 解析查询出来的数据
        List<Content> contents = new HtmlParseUtil().parseJD(keywords);
        // 封装数据到索引库中！
        BulkRequest bulkRequest = new BulkRequest();
        bulkRequest.timeout(TimeValue.timeValueMinutes(2));
        bulkRequest.timeout("2m");
        for(int i =0;i<contents.size();i++){
            bulkRequest
                .add(new IndexRequest("jd_goods")

.source(JSON.toJSONString(contents.get(i)),XContentType.JSON));
        }

        BulkResponse bulkResponse =
restHighLevelClient.bulk(bulkRequest,RequestOptions.DEFAULT);
        return !bulkResponse.hasFailures();
    }

    // 2、实现搜索功能，带分页处理
    public List<Map<String, Object>> searchContentPage(String keyword, int
pageNo, int pageSize) throws IOException {

        // 基本的参数判断！
        if(pageNo <= 1){
            pageNo = 1;
        }
        // 基本的条件搜索
        SearchRequest searchRequest = new SearchRequest("jd_goods");
        SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
        // 分页
        sourceBuilder.from(pageNo);
        sourceBuilder.size(pageSize);
        // 精准匹配 QueryBuilders 根据自己要求配置查询条件即可！
```

```
        TermQueryBuilder termQueryBuilder = QueryBuilders.termQuery("title", keyword);
        sourceBuilder.query(termQueryBuilder);
        sourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));

        // 搜索
        searchRequest.source(sourceBuilder);
        SearchResponse response = restHighLevelClient.search(searchRequest, RequestOptions.DEFAULT);
        // 解析结果!
        List<Map<String, Object>> list = new ArrayList<>();
        for (SearchHit documentFields : response.getHits().getHits()) {
            list.add(documentFields.getSourceAsMap());
        }
        return list;
    }

}
```

3、controller

```
@RestController
public class ContentController {

    @Autowired
    private ContentService contentService;

    @GetMapping("/parse/{keyword}")
    public Boolean parse(@PathVariable("keyword") String keyword) throws Exception {
        return contentService.parseContent(keyword);
    }

    //http://localhost:9090/search/java/1/10
    @GetMapping("/search/{keyword}/{pageNo}/{pageSize}")
    public List<Map<String, Object>> search(@PathVariable("keyword") String keyword,
                            @PathVariable("pageNo") int pageNo,
                            @PathVariable("pageSize") int pageSize) throws Exception {

        return contentService.searchContentPage(keyword,pageNo,pageSize);
    }

}
```

前端逻辑

1、定义导入vue和axios的依赖!

```
<script th:src="@{/js/axios.js}"></script>
<script th:src="@{/js/vue.min.js}"></script>
```

2、初始化Vue对象，给外层div绑定app对象!

```
<script>

    new Vue({
        el: '#app',
        data: {
            keyword: '',  // 搜索关键字
            results: []   // 搜索的结果
        }
    })

</script>
```

3、绑定搜索框及相关事件！

```
<!--搜索-->
<div id="mallSearch" class="mall-search">
    <form name="searchTop" class="mallSearch-form clearfix">
        <fieldset>
            <legend>天猫搜索</legend>
            <div class="mallSearch-input clearfix">
                <div class="s-combobox" id="s-combobox-685" >
                    <div class="s-combobox-input-wrap">
                        <input v-model="keyword" type="text" autocomplete="off" value="dd" id="mq"
                               class="s-combobox-input" aria-haspopup="true">
                    </div>
                </div>
                <button type="submit" id="searchbtn" @click.prevent="searchKey">搜索</button>
            </div>
        </fieldset>
    </form>
    <ul class="relKeyTop">
        <li><a>狂神说Java</a></li>
        <li><a>狂神说前端</a></li>
        <li><a>狂神说Linux</a></li>
        <li><a>狂神说大数据</a></li>
        <li><a>狂神聊理财</a></li>
    </ul>
</div>
```

双向绑定

绑定事件

4、编写方法，获取后端传递的数据！

```
<script>

    new Vue({
        el: '#app',
        data: {
            keyword: '',
            results: []
        },
        methods: {
            searchKey(){
                var keyword = this.keyword;
                console.log(keyword);
                axios.get('search/'+keyword+"/1/10").then(response=>{
                    console.log(response);
                    this.results = response.data;
                });
            }
        }
    })

</script>
```

5、渲染解析回来的数据！

```
<div class="product" v-for="result in results">   遍历响应结果
    <div class="product-iWrap">
        <!--商品封面-->
        <div class="productImg-wrap">
            <a class="productImg">
                <img :src="result.img">   图片赋值
            </a>
        </div>
        <!--价格-->
        <p class="productPrice">
            <em> {{result.price}} </em>   价格
        </p>
        <!--标题-->
        <p class="productTitle">
            <a> {{result.title}} </a>   标题
        </p>
        <!-- 店铺名 -->
        <div class="productShop">
            <span>店铺： 狂神说Java </span>
        </div>
        <!-- 成交信息 -->
        <p class="productStatus">
            <span>月成交<em>999笔</em></span>
            <span>评价 <a>3</a></span>
        </p>
    </div>
</div>
```

搜索高亮

1、编写业务类，处理高亮字段

```java
// 3、实现搜索功能，带高亮
public List<Map<String, Object>> searchContentHighlighter(String keyword, int pageNo, int pageSize) throws IOException {

    // 基本的参数判断！
    if(pageNo <= 1){
        pageNo = 1;
    }
    // 基本的条件搜索
    SearchRequest searchRequest = new SearchRequest("jd_goods");
    SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
    // 分页
    sourceBuilder.from(pageNo);
    sourceBuilder.size(pageSize);
    // 精准匹配 QueryBuilders 根据自己要求配置查询条件即可！
    TermQueryBuilder termQueryBuilder = QueryBuilders.termQuery("title", keyword);
    sourceBuilder.query(termQueryBuilder);

    // 高亮构建！
    HighlightBuilder highlightBuilder = new HighlightBuilder(); //生成高亮查询器
    highlightBuilder.field("title");      //高亮查询字段
    highlightBuilder.requireFieldMatch(false);     //如果要多个字段高亮,这项要为false
    highlightBuilder.preTags("<span style=\"color:red\">");   //高亮设置
    highlightBuilder.postTags("</span>");

    sourceBuilder.highlighter(highlightBuilder);
    sourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));

    // 搜索
    searchRequest.source(sourceBuilder);
    SearchResponse response = restHighLevelClient.search(searchRequest, RequestOptions.DEFAULT);
```

```
    // 解析结果!
    List<Map<String, Object>> list = new ArrayList<>();
    for (SearchHit hit : response.getHits()) {
        //获取高亮字段
        Map<String, HighlightField> highlightFields =
hit.getHighlightFields();
        HighlightField titleField = highlightFields.get("title");

        Map<String, Object> source = hit.getSourceAsMap();
        //千万记得要记得判断是不是为空,不然你匹配的第一个结果没有高亮内容,那么就会报空指
针异常,这个错误一开始真的搞了很久
        if(titleField!=null){
            Text[] fragments = titleField.fragments();
            String name = "";
            for (Text text : fragments) {
                name += text;
            }
            source.put("title", name);   //高亮字段替换掉原本的内容
        }
        list.add(source);
    }
    return list;
}
```

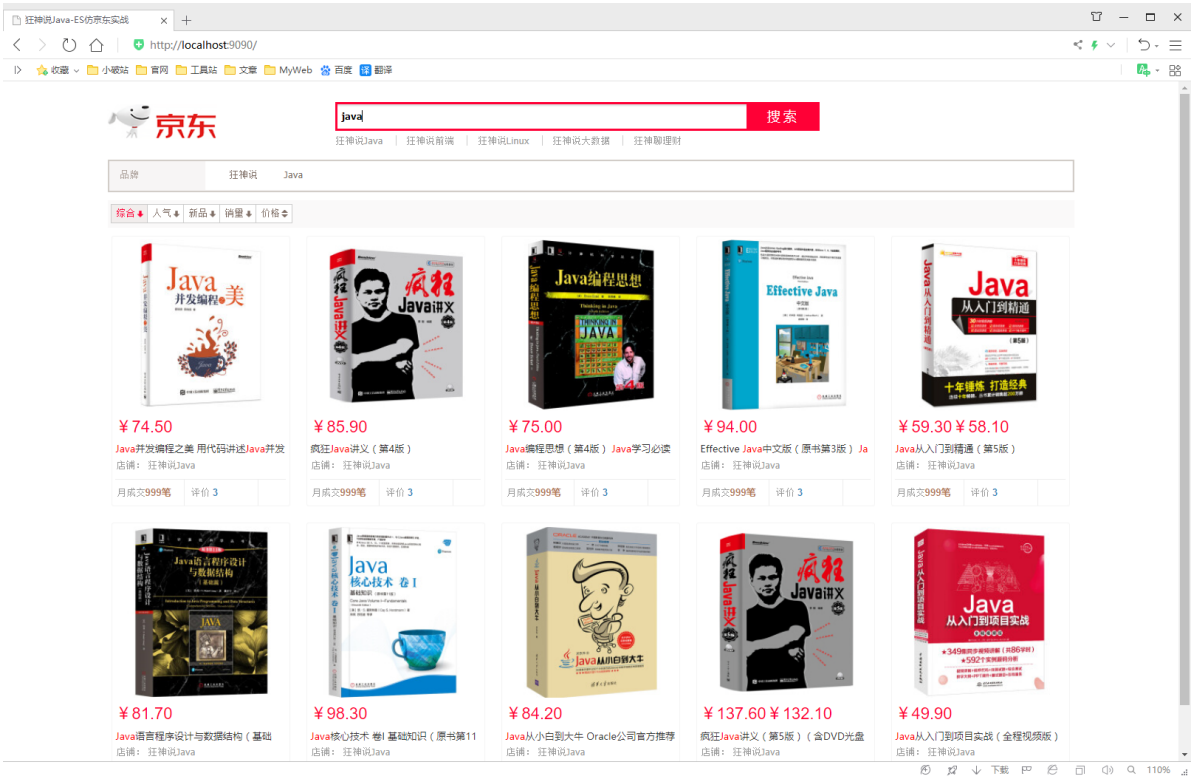
2、controller层调用新的高亮业务!

```
//http://localhost:9090/search/java/1/10
@GetMapping("/search/{keyword}/{pageNo}/{pageSize}")
public List<Map<String, Object>> search(@PathVariable("keyword") String
keyword,
                                        @PathVariable("pageNo") int pageNo,
                                        @PathVariable("pageSize") int
pageSize) throws Exception {

    return contentService.searchContentHighlighter(keyword,pageNo,pageSize);
}
```

3、前端vue指令解析html!

```
<!--标题-->
<p class="productTitle">
    <a v-html="result.title"> </a>
</p>
```

4、最终效果!



ES集群

为什么要实现集群

> ES基本概念名词

Cluster

代表一个集群，集群中有多个节点，其中有一个为主节点，这个主节点是可以通过选举产生的，主从节点是对于集群内部来说的。es的一个概念就是去中心化，字面上理解就是无中心节点，这是对于集群外部来说的，因为从外部来看es集群，在逻辑上是个整体，你与任何一个节点的通信和与整个es集群通信是等价的。

Shards

代表索引分片，es可以把一个完整的索引分成多个分片，这样的好处是可以把一个大的索引拆分成多个，分布到不同的节点上。构成分布式搜索。分片的数量只能在索引创建前指定，并且索引创建后不能更改。

Replicas

代表索引副本，es可以设置多个索引的副本，副本的作用一是提高系统的容错性，当某个节点某个分片损坏或丢失时可以从副本中恢复。二是提高es的查询效率，es会自动对搜索请求进行负载均衡。

Recovery

代表数据恢复或叫数据重新分布，es在有节点加入或退出时会根据机器的负载对索引分片进行重新分配，挂掉的节点重新启动时也会进行数据恢复。

> ES为什么要实现集群

在单台ES服务器节点上，随着业务量的发展索引文件慢慢增多，会影响到效率和内存存储问题等。

我们可以采用ES集群，将单个索引的分片到多个不同分布式物理机器上存储，从而可以实现高可用、容错性等。

ES集群中索引可能由多个分片构成，并且每个分片可以拥有多个副本。通过将一个单独的索引分为多个分片，我们可以处理不能在一个单一的服务器上面运行的大型索引，简单的说就是索引的大小过大，导致效率问题。不能运行的原因可能是内存也可能是存储。由于每个分片可以有多个副本，通过将副本分配到多个服务器，可以提高查询的负载能力。

> ES是如何解决高并发

ES是一个分布式全文检索框架，隐藏了复杂的处理机制，内部使用 分片机制、集群发现、分片负载均衡请求路由。

Shards 分片:代表索引分片，es可以把一个完整的索引分成多个分片，这样的好处是可以把一个大的索引拆分成多个，分布到不同的节点上。构成分布式搜索。分片的数量只能在索引创建前指定，并且索引创建后不能更改。

Replicas分片:代表索引副本，es可以设置多个索引的副本，副本的作用一是提高系统的容错性，当某个节点某个分片损坏或丢失时可以从副本中恢复。二是提高es的查询效率，es会自动对搜索请求进行负载均衡。

集群规划

搭建一个集群我们需要考虑如下几个问题：

1、我们需要多大规模的集群？

2、集群中的节点角色如何分配？

3、如何避免脑裂问题？

4、索引应该设置多少个分片？

5、分片应该设置几个副本？

下面我们就来分析和回答这几个问题：

> 我们需要多大规模的集群？

需要从以下两个方面考虑：

1、当前的数据量有多大？数据增长情况如何？
2、你的机器配置如何？cpu、多大内存、多大硬盘容量？

推算的依据：

ES JVM heap 最大可以设置32G 。

30G heap 大概能处理的数据量 10 T。如果内存很大如128G，可在一台机器上运行多个ES节点实例。

备注：集群规划满足当前数据规模+适量增长规模即可，后续可按需扩展。

两类应用场景：

A、用于构建业务搜索功能模块，且多是垂直领域的搜索。数据量级几千万到数十亿级别。一般2-4台机器的规模。

B、用于大规模数据的实时OLAP（联机处理分析），经典的如ELK Stack，数据规模可能达到千亿或更多。几十到上百节点的规模。

> 集群中的节点角色如何分配?

节点角色：

Master

node.master: true 节点可以作为主节点

DataNode

node.data: true 默认是数据节点。

Coordinate node 协调节点

如果仅担任协调节点，将上两个配置设为false。

说明：

一个节点可以充当一个或多个角色，默认三个角色都有

协调节点：一个节点只作为接收请求、转发请求到其他节点、汇总各个节点返回数据等功能的节点。就叫协调节点

如何分配：

A、小规模集群，不需严格区分。
B、中大规模集群（十个以上节点），应考虑单独的角色充当。特别并发查询量大，查询的合并量大，可以增加独立的协调节点。角色分开的好处是分工分开，不互影响。如不会因协调角色负载过高而影响数据节点的能力。

> 如何避免脑裂问题?

脑裂问题：

一个集群中只有一个A主节点，A主节点因为需要处理的东西太多或者网络过于繁忙，从而导致其他从节点ping不通A主节点，这样其他从节点就会认为A主节点不可用了，就会重新选出一个新的主节点B。过了一会A主节点恢复正常了，这样就出现了两个主节点，导致一部分数据来源于A主节点，另外一部分数据来源于B主节点，出现数据不一致问题，这就是**脑裂。**

尽量避免脑裂，需要添加最小数量的主节点配置：

discovery.zen.minimum_master_nodes: (有master资格节点数/2) + 1

这个参数控制的是，选举主节点时需要看到最少多少个具有master资格的活节点，才能进行选举。官方的推荐值是(N/2)+1，其中N是具有master资格的节点的数量。

常用做法（中大规模集群）：

1、Master 和 dataNode 角色分开，配置奇数个master，如3

2、单播发现机制，配置master资格节点：

discovery.zen.ping.multicast.enabled: false —— 关闭多播发现机制，默认是关闭的

discovery.zen.ping.unicast.hosts: ["master1", "master2", "master3"] —— 配置单播发现的主节点ip地址，其他从节点要加入进来，就得去询问单播发现机制里面配置的主节点我要加入到集群里面了，主节点同意以后才能加入，然后主节点再通知集群中的其他节点有新节点加入

3、配置选举发现数，及延长ping master的等待时长

discovery.zen.ping_timeout: 30（默认值是3秒）——其他节点ping主节点多久时间没有响应就认为主节点不可用了
discovery.zen.minimum_master_nodes: 2 —— 选举主节点时需要看到最少多少个具有master资格的活节点，才能进行选举

> 索引应该设置多少个分片?

说明：分片数指定后不可变，除非重索引。

思考：

分片对应的存储实体是什么？存储的实体是索引

分片是不是越多越好？不是

分片多有什么影响？分片多浪费存储空间、占用资源、影响性能

分片过多的影响：

每个分片本质上就是一个Lucene索引, 因此会消耗相应的文件句柄, 内存和CPU资源。

每个搜索请求会调度到索引的每个分片中. 如果分片分散在不同的节点倒是问题不太. 但当分片开始竞争相同的硬件资源时, 性能便会逐步下降。

ES使用词频统计来计算相关性. 当然这些统计也会分配到各个分片上. 如果在大量分片上只维护了很少的数据, 则将导致最终的文档相关性较差。

分片设置的可参考原则：

ElasticSearch推荐的最大JVM堆空间是30~32G, 所以把你的分片最大容量限制为30GB, 然后再对分片数量做合理估算. 例如, 你认为你的数据能达到200GB, 推荐你最多分配7到8个分片。

在开始阶段, 一个好的方案是根据你的节点数量按照1.5~3倍的原则来创建分片. 例如,如果你有3个节点, 则推荐你创建的分片数最多不超过9(3x3)个。当性能下降时，增加节点，ES会平衡分片的放置。

对于基于日期的索引需求, 并且对索引数据的搜索场景非常少. 也许这些索引量将达到成百上千, 但每个索引的数据量只有1GB甚至更小. 对于这种类似场景, 建议只需要为索引分配1个分片。如日志管理就是一个日期的索引需求，日期索引会很多，但每个索引存放的日志数据量就很少。

> 分片应该设置几个副本?

说明：副本数是可以随时调整的!

思考：

副本的用途是什么？备份数据保证高可用数据不丢失，高并发的时候参与数据查询

针对它的用途，我们该如何设置它的副本数？一般一个分片有1-2个副本即可保证高可用

集群规模没变的情况下副本过多会有什么影响？副本多浪费存储空间、占用资源、影响性能

副本设置基本原则：

为保证高可用，副本数设置为2即可。要求集群至少要有3个节点，来分开存放主分片、副本。
如发现并发量大时，查询性能会下降，可增加副本数，来提升并发查询能力。

注意：新增副本时主节点会自动协调，然后拷贝数据到新增的副本节点

集群核心原理分析

1、每个索引会被分成多个分片shards进行存储，默认创建索引是分配5个分片进行存储。每个分片都会分布式部署在多个不同的节点上进行部署，该分片成为primary shards。

注意：索引的主分片primary shards定义好后，后面不能做修改。

2、为了实现高可用数据的高可用，主分片可以有对应的备分片replics shards，replic shards分片承载了负责容错、以及请求的负载均衡。

注意: 每一个主分片为了实现高可用，都会有自己对应的备分片，主分片对应的备分片不能存放同一台服务器上。主分片primary shards可以和其他replics shards存放在同一个node节点上。

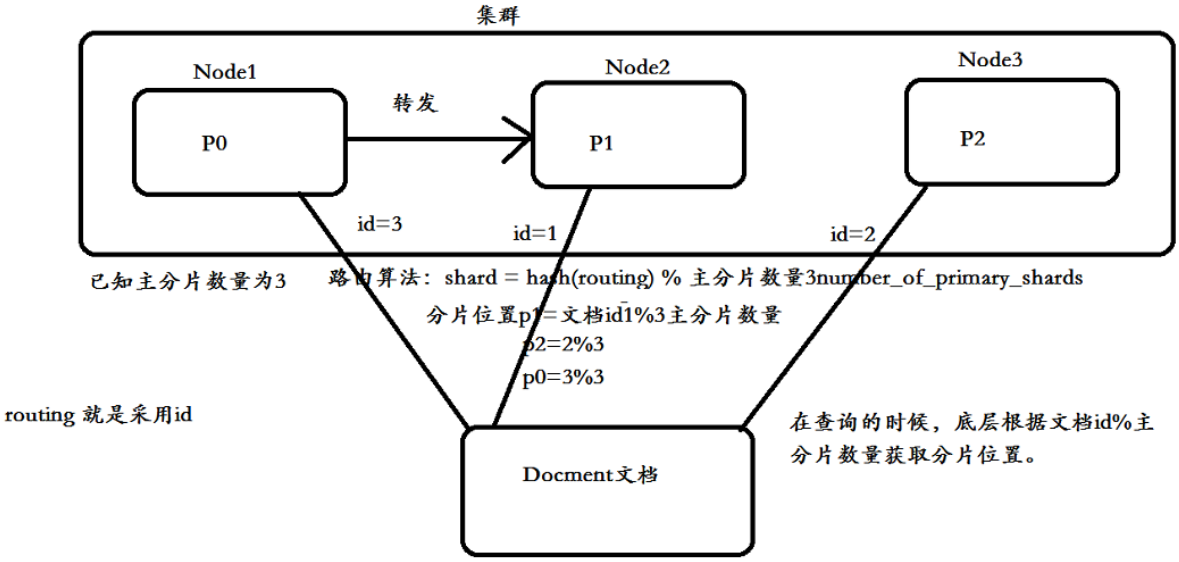
3、documnet routing（数据路由）

当客户端发起创建document的时候，es需要确定这个document放在该index哪个shard上。这个过程就是数据路由。

路由算法：shard = hash(routing) % number_of_primary_shards

如果number_of_primary_shards在查询的时候取余发生的变化，无法获取到该数据

注意：索引的主分片数量定义好后，不能被修改



1.ES核心存放的是那些？ 索引

2. 如果ES实现了集群的话，会将单台服务器节点的索引文件使用分片技术，分布式存放在多个不同的物理机器上。

3.什么是分片技术？ 将数据拆分成多台节点进行存放。

4. 在ES分片技术中，分为主（primary）分片、副本Replicas分片

1.ES为了高可用，每个主的分片都会对应有一个备分的分片。

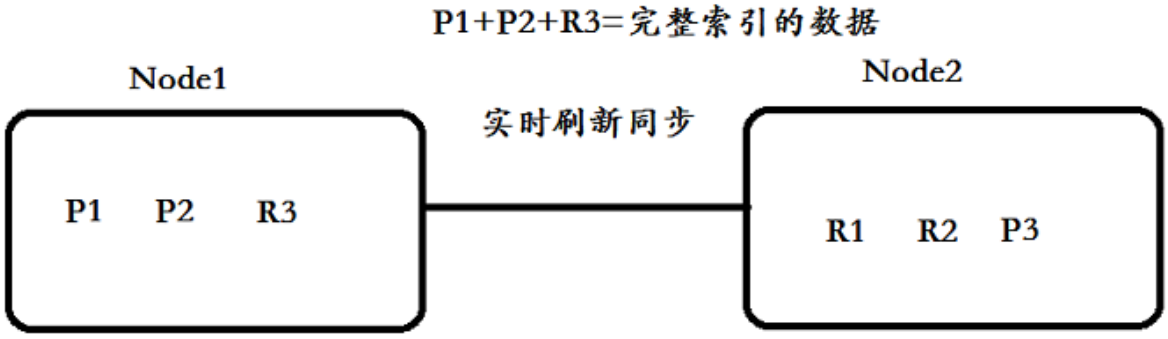
注意的是对应的主的分片和备的分片不能够在同一台节点上进行存放。

P 表示主分片

R 表示备分片

2.单台ES服务器中是没有备用分片的

number_of_shards=3 3+3=6

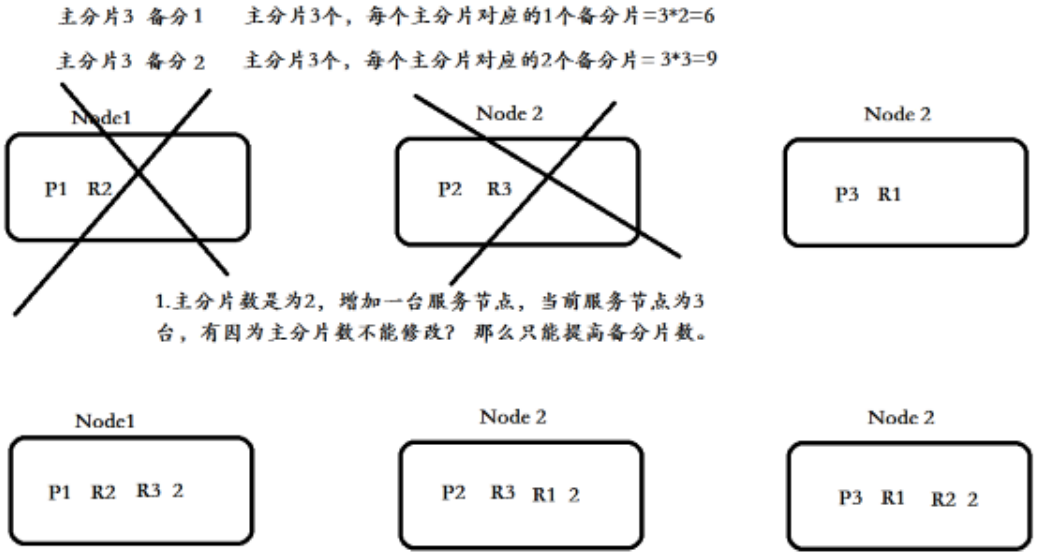


注意：在节点服务器上 既可存放主分片也可以存放备用分片

高可用视图分析

下图所示：上面的图，如果节点1与节点2宕机了，es集群数据就不完整了。

下图，如果节点1与节点2宕机了，es集群数据还是完整的



集群搭建

1、复制三个es的文件

2、进入elasticsearch的config目录，修改elasticsearch.yml的配置

```
# ================== Elasticsearch Configuration ===================
# 配置es的集群名称， es会自动发现在同一网段下的es,如果在同一网段下有多个集群,就可以用这个属性来区分不同的集群。
cluster.name: elasticsearch
# 节点名称（要修改）
node.name: node-001
# 指定该节点是否有资格被选举成为node
node.master: true
# 指定该节点是否存储索引数据,默认为true。
node.data: true
# 设置绑定的ip地址还有其它节点和该节点交互的ip地址,本机ip
network.host: 127.0.0.1
# 指定http端口,你使用head、kopf等相关插件使用的端口 （要修改）
http.port: 9200
# 设置节点间交互的tcp端口,默认是9300。（要修改）
transport.tcp.port: 9300
#设置集群中master节点的初始列表,可以通过这些节点来自动发现新加入集群的节点。
#因为下两台elasticsearch的port端口会设置成9301 和 9302 所以写入两台#elasticsearch地址的完整路径
discovery.zen.ping.unicast.hosts:
["127.0.0.1:9300","127.0.0.1:9301","127.0.0.1:9302"]
#如果要使用head,那么需要解决跨域问题,使head插件可以访问es
http.cors.enabled: true
http.cors.allow-origin: "*"
```

> 配置解析

IP访问限制、默认端口修改9200

这里有两个需要提醒下，第一个就是IP访问限制，第二个就是es实例的默认端口号9200。IP访问限制可以限定具体的IP访问服务器，这有一定的安全过滤作用。

```
# Set the bind address to a specific IP (IPv4 or IPv6):
#
network.host: 192.168.152.128
```

如果设置成0.0.0.0则是不限制任何IP访问。一般在生产的服务器可能会限定几台IP，通常用于管理使用。

默认的端口9200在一般情况下也有点风险，可以将默认的端口修改成另外一个，这还有一个原因就是怕开发人员误操作，连接上集群。当然，如果你的公司网络隔离做的很好也无所谓。

```
#
# Set a custom port for HTTP:
#
http.port: 9200
transport.tcp.port: 9300
```

这里的9300是集群内部通讯使用的端口，这个也可以修改掉。因为连接集群的方式有两种，通过扮演集群node也是可以进入集群的，所以还是安全起见，修改掉默认的端口。

说明：记得修改安装了ES的3台虚拟机（三个节点）的相同配置，要不然节点之间无法建立连接工作，也会报错。

集群发现IP列表、node、cluster名称

紧接着修改集群节点IP地址，这样可以让集群在规定的几个节点之间工作。elasticsearch，默认是使用自动发现IP机制。就是在当前网段内，只要能被自动感知到的IP就能自动加入到集群中。这有好处也有坏处。好处就是自动化了，当你的es集群需要云化的时候就会非常方便。但是也会带来一些不稳定的情况，如，master的选举问题、数据复制问题。

导致master选举的因素之一就是集群有节点进入。当数据复制发生的时候也会影响集群，因为要做数据平衡复制和冗余。这里面可以独立master集群，剔除master集群的数据节点能力。

固定列表的IP发现有两种配置方式，一种是互相依赖发现，一种是全量发现。各有优势吧，我是使用的依赖发现来做的。这有个很重要的参考标准，就是你的集群扩展速度有多快。因为这有个问题就是，当全量发现的时候，如果是初始化集群会有很大的问题，就是master全局会很长，然后节点之间的启动速度各不一样。所以我采用了靠谱点的依赖发现。

你需要在192.168.152.128的elasticsearch中配置成：

```
# ---------------------------------- Discovery -----------------------------------
#
# Pass an initial list of hosts to perform discovery when new node is started:
# The default list of hosts is ["127.0.0.1", "[::1]"]
#
discovery.zen.ping.unicast.hosts: [
"192.168.152.129:9300","192.168.152.130:9300" ]
```

让他去发现129，130的机器，以此内推，完成剩下的129和130机器的配置。

然后你需要配置下集群名称，就是你当前节点所在集群的名称，这有助于你规划你的集群。集群中的所有节点的集群名称必须一样，只有集群名称一样才能组成一个逻辑集群。

```
# ---------------------------------- Cluster -----------------------------------
#
# Use a descriptive name for your cluster:
#
cluster.name: mycluster
```

配置你当前节点的名称

```
# ------------------------------------ Node ------------------------------------
#
# Use a descriptive name for the node:
#
node.name: node-1
```

以此类推，完成另外两个节点的配置。cluster.name的名称必须保持一样。然后分别设置node.name。

说明：

这里搭建的是一个简单的集群，没有做集群节点角色的区分，所以3个节点默认的角色有主节点、数据节点、协调节点

选举ES主节点的逻辑：

选举的大概逻辑，它会根据分片的数据的前后新鲜程度来作为选举的一个重要逻辑。（日志、数据、时间都会作为集群master全局的重要指标）

因为考虑到数据一致性问题，当然是用最新的数据节点作为master，然后进行新数据的复制和刷新其他node。