ELK搜索高级课程

#1． 课程简介

#1.1 课程内容

ELK是包含但不限于Elasticsearch（简称es）、Logstash、Kibana 三个开源软件的组成的一个整体，分别取其首字母组成ELK。ELK是用于数据抽取（Logstash）、搜索分析（Elasticsearch）、数据展现（Kibana）的一整套解决方案，所以也称作ELK stack。

本课程分别对三个组件进行详细介绍，尤其是Elasticsearch，因为它是ELK的核心。Elasticsearch会对底层的文档、索引、搜索、聚合、集群进行介绍，从搜索和聚合分析实例来展现它的魅力。Logstash会从内部如何采集数据到指定地方来展现它数据采集的功能。Kibana则从数据绘图来展现它数据可视化的能力。

#1.2 面向人员

- java工程师：深入研究es,使得java工程师向搜索工程师迈进。
- 运维工程师：搭建整体elk集群。不需写代码，仅需配置，即可收集服务器指标、日志文件、数据库数据，并在前端华丽展现。
- 数据分析人员：不需写代码，仅需配置kibana图表，即可完成数据可视化工作，得到想要的数据图表。
- 大厂架构师：完成数据中台的搭建。对公司数据流的处理得心应手，对接本公司大数据业务。

#1.3 课程优势

- 基于最新的elk7.3版本讲解。最新api。包含sql功能。
- 理论和实际代码相辅相成。理论结合画图讲解。代码与spring boot结合。
- 包含实际运维部署理论与实践。
- Elk整体流程项目，包含数据采集。

#1.4 学习路径

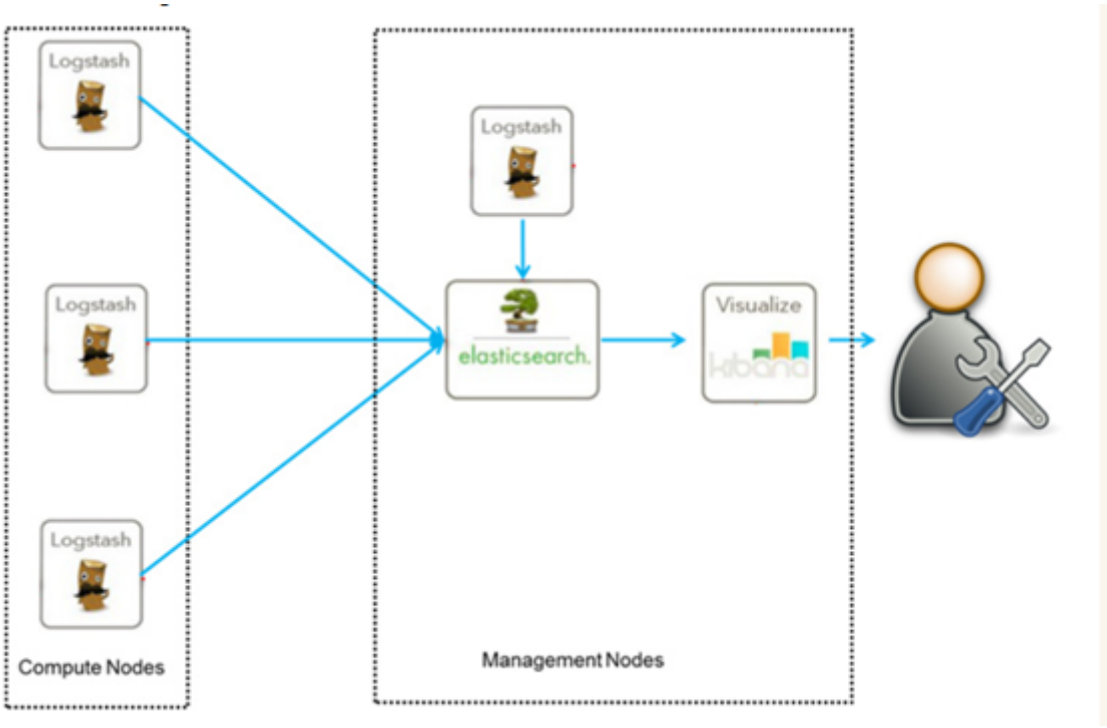
参照目录，按照介绍，es入门，文档、映射、索引、分词器、搜索、聚合。logstash、kibana。集群部署。项目实战。

每个知识点先学概念，在学rest api,最后java代码上手。

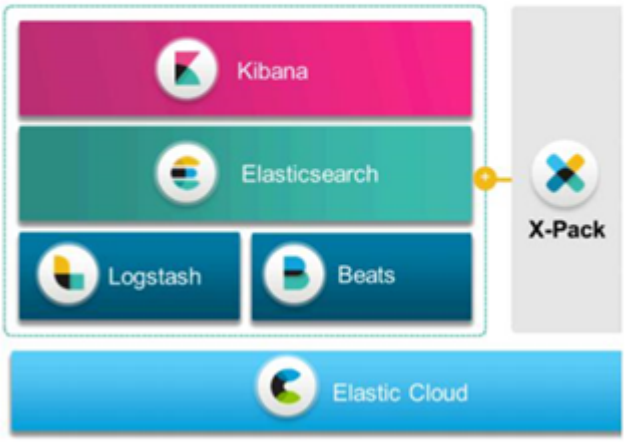
#2． Elastic Stack简介

#2.1简介

ELK是一个免费开源的日志分析架构技术栈总称，官网https://www.elastic.co/cn。包含三大基础组件，分别是Elasticsearch、Logstash、Kibana。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搜索、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。下面是ELK架构：



随着elk的发展，又有新成员Beats、elastic cloud的加入，所以就形成了Elastic Stack。所以说，ELK是旧的称呼，Elastic Stack是新的名字。



#2.2特色

- 处理方式灵活：elasticsearch是目前最流行的准实时全文检索引擎，具有高速检索大数据的能力。
- 配置简单：安装elk的每个组件，仅需配置每个组件的一个配置文件即可。修改处不多，因为大量参数已经默认配在系统中，修改想要修改的选项即可。
- 接口简单：采用json形式RESTFUL API接受数据并响应，无关语言。
- 性能高效：elasticsearch基于优秀的全文搜索技术Lucene，采用倒排索引，可以轻易地在百亿级别数据量下，搜索出想要的内容，并且是秒级响应。

- 灵活扩展：elasticsearch和logstash都可以根据集群规模线性拓展，elasticsearch内部自动实现集群协作。
- 数据展现华丽：kibana作为前端展现工具，图表华丽，配置简单。

#2.3组件介绍

Elasticsearch

Elasticsearch 是使用java开发，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。它的特点有：分布式，零配置，自动发现，索引自动分片，索引副本机制，restful风格接口，多数据源，自动搜索负载等。

Logstash

Logstash 基于java开发，是一个数据抽取转化工具。一般工作方式为c/s架构，client端安装在需要收集信息的主机上，server端负责将收到的各节点日志进行过滤、修改等操作在一并发往elasticsearch或其他组件上去。

Kibana

Kibana 基于nodejs，也是一个开源和免费的可视化工具。Kibana可以为 Logstash 和 ElasticSearch 提供的日志分析友好的 Web 界面，可以汇总、分析和搜索重要数据日志。

Beats

Beats 平台集合了多种单一用途数据采集器。它们从成百上千或成千上万台机器和系统向 Logstash 或 Elasticsearch 发送数据。

Beats由如下组成:

Packetbeat：轻量型网络数据采集器，用于深挖网线上传输的数据，了解应用程序动态。Packetbeat 是一款轻量型网络数据包分析器，能够将数据发送至 Logstash 或 Elasticsearch。其支 持ICMP (v4 and v6)、DNS、HTTP、Mysql、PostgreSQL、Redis、MongoDB、Memcache等协议。

Filebeat：轻量型日志采集器。当您要面对成百上千、甚至成千上万的服务器、虚拟机和容器生成的日志时，请告别 SSH 吧。Filebeat 将为您提供一种轻量型方法，用于转发和汇总日志与文件，让简单的事情不再繁杂。

Metricbeat ：轻量型指标采集器。Metricbeat 能够以一种轻量型的方式，输送各种系统和服务统计数据，从 CPU 到内存，从 Redis 到 Nginx，不一而足。可定期获取外部系统的监控指标信息，其可以监控、收集 Apache http、HAProxy、MongoDB、MySQL、Nginx、PostgreSQL、Redis、System、Zookeeper 等服务。

Winlogbeat：轻量型 Windows 事件日志采集器。用于密切监控基于 Windows 的基础设施上发生的事件。Winlogbeat 能够以一种轻量型的方式，将 Windows 事件日志实时地流式传输至 Elasticsearch 和 Logstash。

Auditbeat：轻量型审计日志采集器。收集您 Linux 审计框架的数据，监控文件完整性。Auditbeat 实时采集这些事件，然后发送到 Elastic Stack 其他部分做进一步分析。

Heartbeat：面向运行状态监测的轻量型采集器。通过主动探测来监测服务的可用性。通过给定 URL 列表，Heartbeat 仅仅询问：网站运行正常吗？Heartbeat 会将此信息和响应时间发送至 Elastic 的其他部分，以进行进一步分析。

Functionbeat：面向云端数据的无服务器采集器。在作为一项功能部署在云服务提供商的功能即服务 (FaaS) 平台上后，Functionbeat 即能收集、传送并监测来自您的云服务的相关数据。

Elastic cloud

基于 Elasticsearch 的软件即服务(SaaS)解决方案。通过 Elastic 的官方合作伙伴使用托管的 Elasticsearch 服务。

#3． Elasticsearch是什么

#3.1搜索是什么

概念：用户输入想要的关键词，返回含有该关键词的所有信息。

场景：

1互联网搜索：谷歌、百度、各种新闻首页

2 站内搜索（垂直搜索）：企业OA查询订单、人员、部门，电商网站内部搜索商品（淘宝、京东）场景。

#3.2 数据库做搜索弊端

#3.2.1站内搜索（垂直搜索）：数据量小，简单搜索，可以使用数据库。

问题出现：

l 存储问题。电商网站商品上亿条时，涉及到单表数据过大必须拆分表，数据库磁盘占用过大必须分库（mycat）。

l 性能问题：解决上面问题后，查询“笔记本电脑”等关键词时，上亿条数据的商品名字段逐行扫描，性能跟不上。

l 不能分词。如搜索“笔记本电脑”，只能搜索完全和关键词一样的数据，那么数据量小时，搜索“笔记电脑”，“电脑”数据要不要给用户。

#3.2.2互联网搜索，肯定不会使用数据库搜索。数据量太大。PB级。

#3.3全文检索、倒排索引和Lucene

#全文检索

倒排索引。数据存储时，经行分词建立term索引库。见画图。

倒排索引源于实际应用中需要根据属性的值来查找记录。这种索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。带有倒排索引的文件我们称为倒排**索引文件**open in new window，简称**倒排文件**open in new window (inverted file)。

#Lucene

就是一个jar包，里面封装了全文检索的引擎、搜索的算法代码。开发时，引入lucene的jar包，通过api开发搜索相关业务。底层会在磁盘建立索引库。

#3.4 什么是Elasticsearch

#简介

ElasticSearch是一个基于Lucene的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于RESTful web接口。Elasticsearch是用Java语言开发的，并作为Apache许可条款下的开放源码发布，是一种流行的企业级搜索引擎。ElasticSearch用于云计算中，能够达到实时搜索，稳定，可靠，快速，安装使用方便。官方客户端在Java、.NET（C#）、PHP、Python、Apache Groovy、Ruby和许多其他语言中都是可用的。根据DB-Engines的排名显示，Elasticsearch是最受欢迎的企业搜索引擎，其次是Apache Solr，也是基于Lucene。

官网： https://www.elastic.co/cn/products/elasticsearch

#Elasticsearch的功能

- 分布式的搜索引擎和数据分析引擎

搜索：互联网搜索、电商网站站内搜索、OA系统查询

数据分析：电商网站查询近一周哪些品类的图书销售前十；新闻网站，最近3天阅读量最高的十个关键词，舆情分析。

- 全文检索，结构化检索，数据分析

全文检索：搜索商品名称包含java的图书select * from books where book_name like "%java%"。

结构化检索：搜索商品分类为spring的图书都有哪些，select * from books where category_id='spring'

数据分析：分析每一个分类下有多少种图书，select category_id,count(*) from books group by category_id

- 对海量数据进行近实时的处理

分布式：ES自动可以将海量数据分散到多台服务器上去存储和检索,经行并行查询，提高搜索效率。相对的，Lucene是单机应用。

近实时：数据库上亿条数据查询，搜索一次耗时几个小时，是批处理（batch-processing）。而es只需秒级即可查询海量数据，所以叫近实时。秒级。

#Elasticsearch的使用场景

国外：

- 维基百科，类似百度百科，“网络七层协议”的维基百科，全文检索，高亮，搜索推荐
- Stack Overflow（国外的程序讨论论坛），相当于程序员的贴吧。遇到it问题去上面发帖，热心网友下面回帖解答。
- GitHub（开源代码管理），搜索上千亿行代码。
- 电商网站，检索商品
- 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）
- 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《java编程思想》的监控，如果价格低于27块钱，就通知我，我就去买。
- BI系统，商业智能（Business Intelligence）。大型连锁超市，分析全国网点传回的数据，分析各个商品在什么季节的销售量最好、利润最高。成本管理，店面租金、员工工资、负债等信息进行分析。从而部署下一个阶段的战略目标。

国内：

- 百度搜索，第一次查询，使用es。
- OA、ERP系统站内搜索。

#Elasticsearch的特点

- 可拓展性：大型分布式集群（数百台服务器）技术，处理PB级数据，大公司可以使用。小公司数据量小，也可以部署在单机。大数据领域使用广泛。
- 技术整合：将全文检索、数据分析、分布式相关技术整合在一起：lucene（全文检索），商用的数据分析软件（BI软件），分布式数据库（mycat）
- 部署简单：开箱即用，很多默认配置不需关心，解压完成直接运行即可。拓展时，只需多部署几个实例即可，负载均衡、分片迁移集群内部自己实施。
- 接口简单：使用restful api经行交互，跨语言。
- 功能强大：Elasticsearch作为传统数据库的一个补充，提供了数据库所不不能提供的很多功能，如全文检索，同义词处理，相关度排名。

相关度

搜索所有内容。找到所需的具体信息。

基于各项元素（从词频或近因到热门度等）对搜索结果进行排序。将这些内容与功能进行混合和匹配，以对向用户显示结果的方式进行微调。

而且，由于我们的大部分用户都是真实的人，Elasticsearch 具备齐全功能，可以处理包括各种复杂情况（例如拼写错误）在内的人为错误。

弹性

我们在您高飞的时候保驾护航。

硬件故障。网络分割。Elasticsearch 为您检测这些故障并确保您的集群（和数据）的安全性和可用性。通过跨集群复制功能，辅助集群可以作为热备份随时投入使用。Elasticsearch 运行在一个分布式的环境中，从设计之初就考虑到了这一点，目的只有一个，让您永远高枕无忧。

#3.5 elasticsearch核心概念

#3.5.1 lucene和elasticsearch的关系

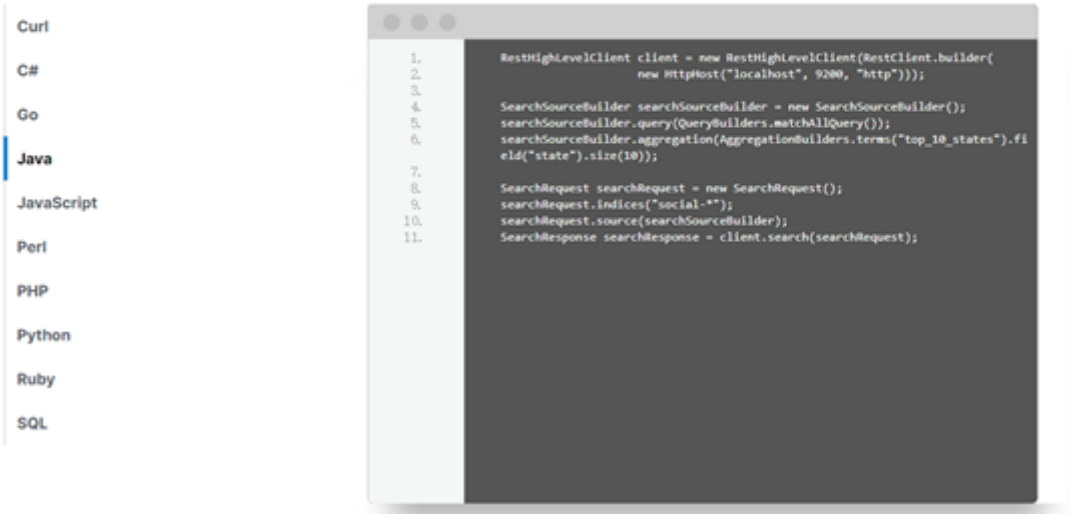
Lucene：最先进、功能最强大的搜索库，直接基于lucene开发，非常复杂，api复杂

Elasticsearch：基于lucene，封装了许多lucene底层功能，提供简单易用的restful api接口和许多语言的客户端，如java的高级客户端（Java High Level REST Clientopen in new window）和底层客户端（Java Low Level REST Clientopen in new window）

客户端库

使用您自己的编程语言与 Elasticsearch 进行交互

Elasticsearch 使用的是标准的 RESTful 风格的 API 和 JSON。此外，我们还构建和维护了很多其他语言的客户端，例如 Java、Python、.NET、SQL 和 PHP。与此同时，我们的社区也贡献了很多客户端。这些客户端使用起来简单自然，而且就像 Elasticsearch 一样，不会对您的使用方式进行限制。



起源：Shay Banon。2004年失业，陪老婆去伦敦学习厨师。失业在家帮老婆写一个菜谱搜索引擎。封装了lucene的开源项目，compass。找到工作后，做分布式高性能项目，再封装compass，写出了elasticsearch，使得lucene支持分布式。现在是Elasticsearch创始人兼Elastic首席执行官。

#3.5.2 elasticsearch的核心概念

#1 NRT（Near Realtime）：近实时

两方面：

- 写入数据时，过1秒才会被搜索到，因为内部在分词、录入索引。
- es搜索时：搜索和分析数据需要秒级出结果。

#2 Cluster：集群

包含一个或多个启动着es实例的机器群。通常一台机器起一个es实例。同一网络下，集群名一样的多个es实例自动组成集群，自动均衡分片等行为。默认集群名为“elasticsearch”。

#3 Node：节点

每个es实例称为一个节点。节点名自动分配，也可以手动配置。

#4 Index：索引

包含一堆有相似结构的文档数据。

索引创建规则：

- 仅限小写字母
- 不能包含\、/、 *、?、"、<、>、|、#以及空格符等特殊符号
- 从7.0版本开始不再包含冒号
- 不能以-、_或+开头
- 不能超过255个字节（注意它是字节，因此多字节字符将计入255个限制）

#5 Document：文档

es中的最小数据单元。一个document就像数据库中的一条记录。通常以json格式显示。多个document存储于一个索引（Index）中。

```
book document

{
  "book_id": "1",
  "book_name": "java编程思想",
  "book_desc": "从Java的基础语法到最高级特性（深入的[面向对象](https://baike.baidu.com/item/面向对象)概念、多线程、自动项目构建、单元测试和调试等），本书都能逐步指导你轻松掌握。",
  "category_id": "2",
  "category_name": "java"
}
```

#6 Field:字段

就像数据库中的列（Columns），定义每个document应该有的字段。

#7 Type：类型

每个索引里都可以有一个或多个type，type是index中的一个逻辑数据分类，一个type下的document，都有相同的field。

注意：6.0之前的版本有type（类型）概念，type相当于关系数据库的表，ES官方将在ES9.0版本中彻底删除type。本教程typy都为_doc。

#8 shard：分片

index数据过大时，将index里面的数据，分为多个shard，分布式的存储在各个服务器上面。可以支持海量数据和高并发，提升性能和吞吐量，充分利用多台机器的cpu。

#9 replica：副本

在分布式环境下，任何一台机器都会随时宕机，如果宕机，index的一个分片没有，导致此index不能搜索。所以，为了保证数据的安全，我们会将每个index的分片经行备份，存储在另外的机器上。保证少数机器宕机es集群仍可以搜索。

能正常提供查询和插入的分片我们叫做主分片（primary shard），其余的我们就管他们叫做备份的分片（replica shard）。

es6默认新建索引时，5分片，2副本，也就是一主一备，共10个分片。所以，es集群最小规模为两台。

#3.5.3 elasticsearch核心概念 vs. 数据库核心概念

关系型数据库（比如Mysql）	非关系型数据库（Elasticsearch）
数据库Database	索引Index
表Table	索引Index（原为Type）
数据行Row	文档Document
数据列Column	字段Field
约束 Schema	映射Mapping

#4． Elasticsearch相关软件安装

#4.1. Windows安装elasticsearch

#1、安装JDK，至少1.8.0_73以上版本，验证：java -version。

#2、下载和解压缩Elasticsearch安装包，查看目录结构。

https://www.elastic.co/cn/downloads/elasticsearch

bin：脚本目录，包括：启动、停止等可执行脚本

config：配置文件目录

data：索引目录，存放索引文件的地方

logs：日志目录

modules：模块目录，包括了es的功能模块

plugins :插件目录，es支持插件机制

#3、配置文件：

位置：

ES的配置文件的地址根据安装形式的不同而不同：

使用zip、tar安装，配置文件的地址在安装目录的config下。

使用RPM安装，配置文件在/etc/elasticsearch下。

使用MSI安装，配置文件的地址在安装目录的config下，并且会自动将config目录地址写入环境变量ES_PATH_CONF。

elasticsearch.yml

配置格式是YAML，可以采用如下两种方式：

方式1：层次方式

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

方式2：属性方式

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

常用的配置项如下

```
cluster.name:
    配置elasticsearch的集群名称，默认是elasticsearch。建议修改成一个有意义的名称。
node.name:
    节点名，通常一台物理服务器就是一个节点，es会默认随机指定一个名字，建议指定一个有意义的名称，方便管理
    一个或多个节点组成一个cluster集群，集群是一个逻辑的概念，节点是物理概念，后边章节会详细介绍。
path.conf:
    设置配置文件的存储路径，tar或zip包安装默认在es根目录下的config文件夹，rpm安装默认在/etc/elasticsearch
path.data:
    设置索引数据的存储路径，默认是es根目录下的data文件夹，可以设置多个存储路径，用逗号隔开。
path.logs:
    设置日志文件的存储路径，默认是es根目录下的logs文件夹
path.plugins:
    设置插件的存放路径，默认是es根目录下的plugins文件夹
bootstrap.memory_lock: true
    设置为true可以锁住ES使用的内存，避免内存与swap分区交换数据。
network.host:
    设置绑定主机的ip地址，设置为0.0.0.0表示绑定任何ip，允许外网访问，生产环境建议设置为具体的ip。
http.port: 9200
    设置对外服务的http端口，默认为9200。
transport.tcp.port: 9300  集群结点之间通信端口
node.master:
    指定该节点是否有资格被选举成为master结点，默认是true，如果原来的master宕机会重新选举新的master。
node.data:
    指定该节点是否存储索引数据，默认为true。
discovery.zen.ping.unicast.hosts: ["host1:port", "host2:port", "..."]
```

```
    设置集群中master节点的初始列表。
discovery.zen.ping.timeout: 3s
    设置ES自动发现节点连接超时的时间，默认为3秒，如果网络延迟高可设置大些。
discovery.zen.minimum_master_nodes:
    主结点数量的最少值 ,此值的公式为：(master_eligible_nodes / 2) + 1 ，比如：有3个符合要求的主结点，那么这里要设置为2。
node.max_local_storage_nodes:
    单机允许的最大存储结点数，通常单机启动一个结点建议设置为1，开发环境如果单机启动多个节点可设置大于1。
```

jvm.options

设置最小及最大的JVM堆内存大小：

在jvm.options中设置 -Xms和-Xmx：

1） 两个值设置为相等

2） 将Xmx 设置为不超过物理内存的一半。

log4j2.properties

日志文件设置，ES使用log4j，注意日志级别的配置。

#4、启动Elasticsearch：bin\elasticsearch.bat，es的特点就是开箱即，无需配置，启动即可。

注意：es7 windows版本不支持机器学习，所以elasticsearch.yml中添加如下几个参数：

```
node.name: node-1
cluster.initial_master_nodes: ["node-1"]
xpack.ml.enabled: false
http.cors.enabled: true
http.cors.allow-origin: /.*/
```

#5、检查ES是否启动成功：浏览器访问 http://localhost:9200/?Pretty

```
{
    "name": "node-1",
    "cluster_name": "elasticsearch",
    "cluster_uuid": "HqAKQ_0tQOOm8b6qU-2Qug",
    "version": {
        "number": "7.3.0",
        "build_flavor": "default",
        "build_type": "zip",
        "build_hash": "de777fa",
        "build_date": "2019-07-24T18:30:11.767338Z",
        "build_snapshot": false,
        "lucene_version": "8.1.0",
        "minimum_wire_compatibility_version": "6.8.0",
        "minimum_index_compatibility_version": "6.0.0-beta1"
    },
    "tagline": "You Know, for Search"
}
```

解释：

name: node名称，取自机器的hostname

cluster_name: 集群名称（默认的集群名称就是elasticsearch）

version.number: 7.3.0，es版本号

version.lucene_version:封装的lucene版本号

#6、浏览器访问 http://localhost:9200/_cluster/health 查询集群状态

```
{
    "cluster_name": "elasticsearch",
    "status": "green",
    "timed_out": false,
    "number_of_nodes": 1,
    "number_of_data_nodes": 1,
    "active_primary_shards": 0,
    "active_shards": 0,
    "relocating_shards": 0,
    "initializing_shards": 0,
    "unassigned_shards": 0,
    "delayed_unassigned_shards": 0,
    "number_of_pending_tasks": 0,
    "number_of_in_flight_fetch": 0,
    "task_max_waiting_in_queue_millis": 0,
    "active_shards_percent_as_number": 100
}
```

解释：

Status：集群状态。Green 所有分片可用。Yellow所有主分片可用。Red主分片不可用，集群不可用。

#4.2. Windows安装Kibana

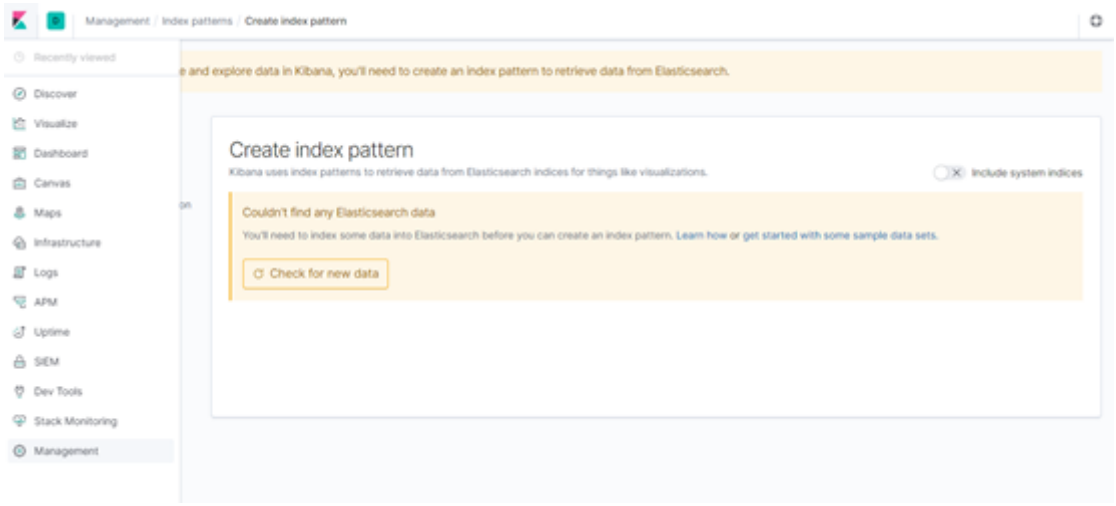
1、kibana是es数据的前端展现，数据分析时，可以方便地看到数据。作为开发人员，可以方便访问es。

2、下载，解压kibana。

3、启动Kibana：bin\kibana.bat

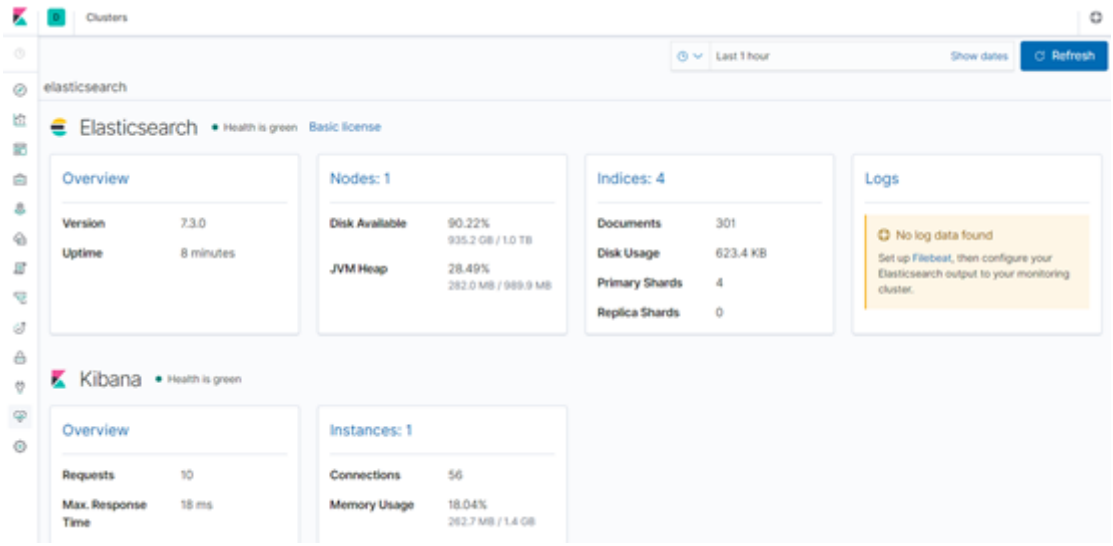
4、浏览器访问 http://localhost:5601 进入Dev Tools界面。像plsql一样支持代码提示。

5、发送get请求，查看集群状态GET _cluster/health。相当于浏览器访问。

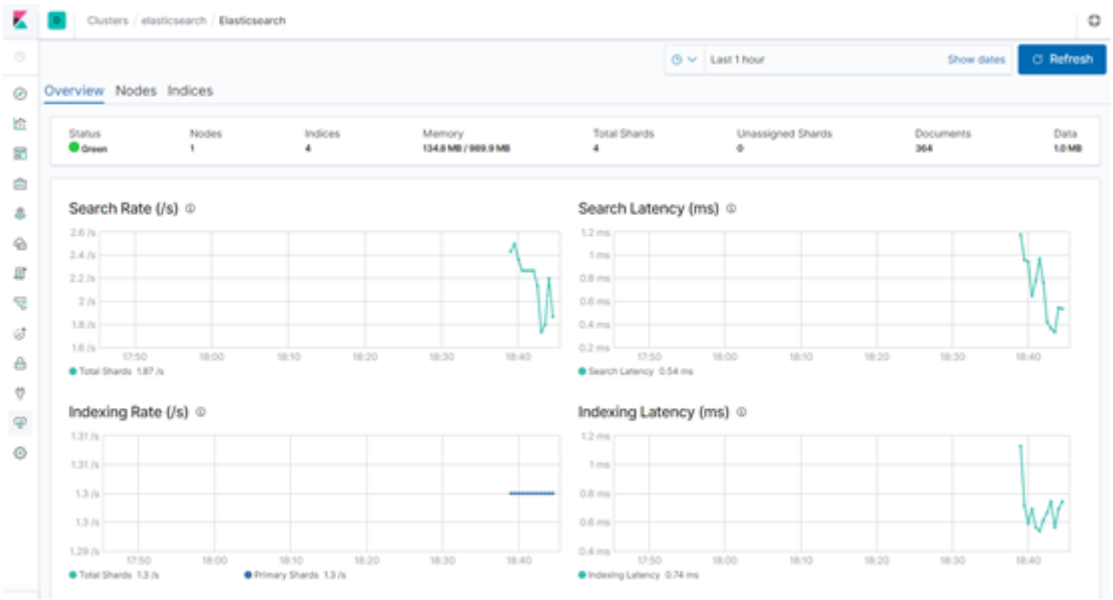


总览

Dev Tools界面



监控集群界面



集群状态（搜索速率、索引速率等）

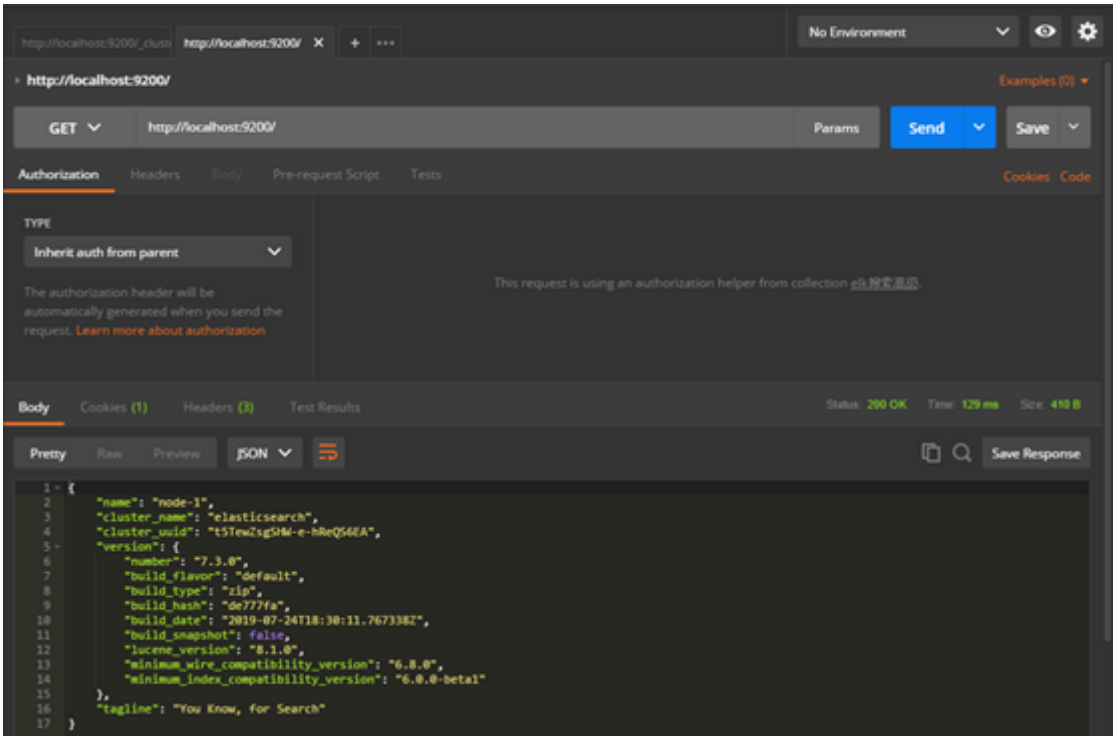
#4.3 Windows安装postman

是什么：postman是一个模拟http请求的工具。能够非常细致地定制化各种http请求。如get]\post\pu\delete,携带body参数等。

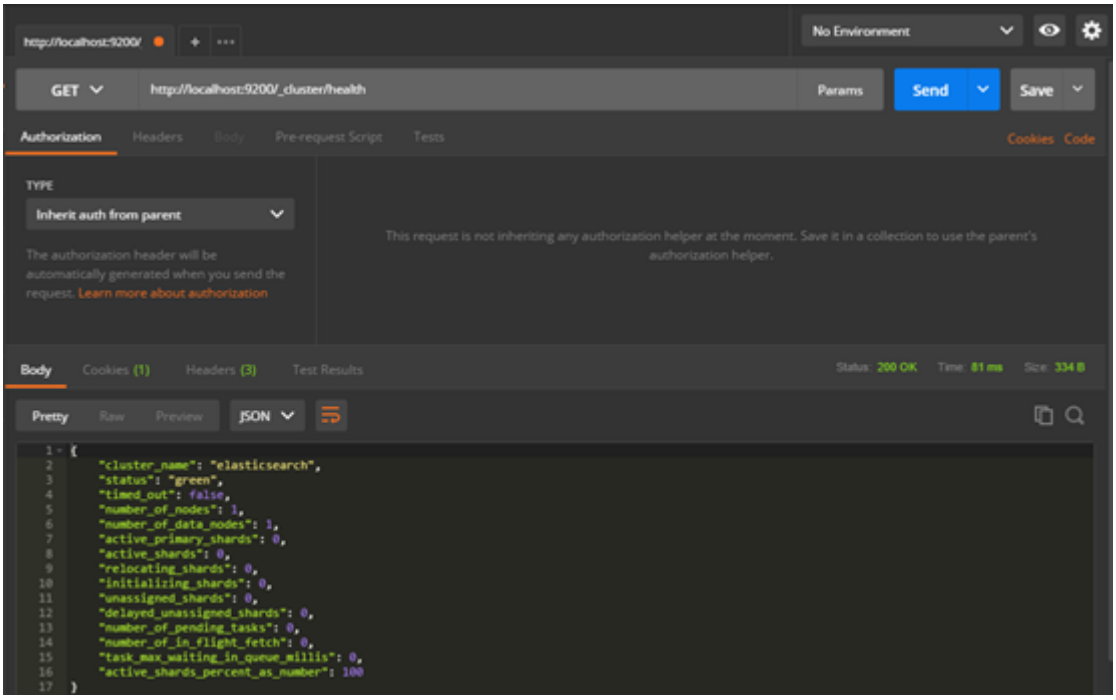
为什么：在没有kibana时，可以使用postman调试。

怎么用：

get http://localhost:9200/



测试一下get方式查询集群状态 http://localhost:9200/_cluster/health



#4.4 Windows安装head插件

head插件是ES的一个可视化管理插件，用来监视ES的状态，并通过head客户端和ES服务进行交互，比如创建映射、创建索引等，head的项目地址在 https://github.com/mobz/elasticsearch-head 。

从ES6.0开始，head插件支持使得node.js运行。

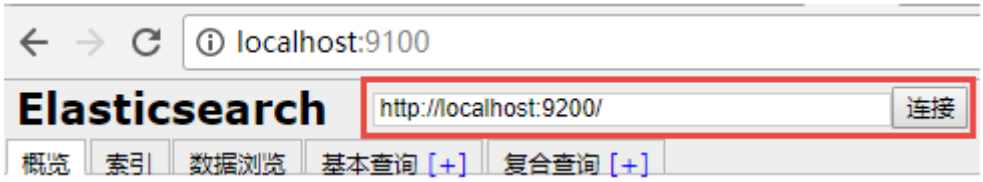
#1安装node.js

#2下载head并运行

```
git clone git://github.com/mobz/elasticsearch-head.git
cd elasticsearch-head
npm install
npm run start
```

浏览器打开 http://localhost:9100/

#3运行



打开浏览器调试工具发现报错：

Origin null is not allowed by Access-Control-Allow-Origin.

原因是：head插件作为客户端要连接ES服务（localhost:9200），此时存在跨域问题，elasticsearch默认不允许跨域访问。

解决方案：

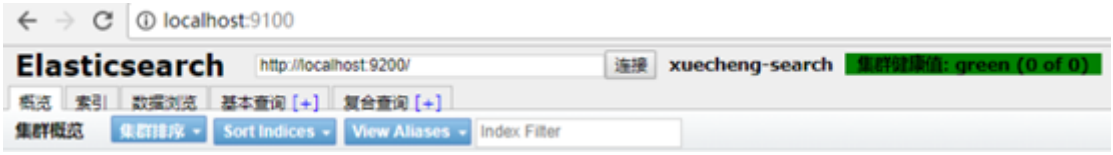
设置elasticsearch允许跨域访问。

在config/elasticsearch.yml 后面增加以下参数：

```
#开启cors跨域访问支持，默认为false
http.cors.enabled: true
#跨域访问允许的域名地址，(允许所有域名)以上使用正则
http.cors.allow-origin: /.*/
```

注意：将config/elasticsearch.yml另存为utf-8编码格式。

成功连接ES



注意：kibana\postman\head插件选择自己喜欢的一种使用即可。

本教程使用kibana的dev tool，因为地址栏省略了 http://localhost:9200 。

#5． es快速入门

#5.1． 文档（document）的数据格式

（1）应用系统的数据结构都是面向对象的，具有复杂的数据结构

（2）对象存储到数据库，需要将关联的复杂对象属性插到另一张表，查询时再拼接起来。

（3）es面向文档，文档中存储的数据结构，与对象一致。所以一个对象可以直接存成一个文档。

（4）es的document用json数据格式来表达。

例如：班级和学生关系

```
public class Student {
  private String id;
  private String name;

  private String classInfoId;
}

private class ClassInfo {
  private String id;
  private String className;
。。。。。

}
```

数据库中要设计所谓的一对多，多对一的两张表，外键等。查询出来时，还要关联，mybatis写映射文件，很繁琐。

而在es中，一个学生存成文档如下：

```
{
    "id":"1",
    "name": "张三",
    "last_name": "zhang",
    "classInfo": {
        "id": "1",
        "className": "三年二班",
    }
}
```

#5.2图书网站商品管理案例：背景介绍

有一个售卖图书的网站，需要为其基于ES构建一个后台系统，提供以下功能：

（1）对商品信息进行CRUD（增删改查）操作

（2）执行简单的结构化查询

（3）可以执行简单的全文检索，以及复杂的phrase（短语）检索

（4）对于全文检索的结果，可以进行高亮显示

（5）对数据进行简单的聚合分析

#5.3． 简单的集群管理

#5.3.1快速检查集群的健康状况

es提供了一套api，叫做cat api，可以查看es中各种各样的数据

GET /_cat/health?v

```
epoch      timestamp cluster       status node.total node.data shards pri relo
init unassign pending_tasks max_task_wait_time active_shards_percent
1568635460 12:04:20  elasticsearch green           1         1      4   4    0
0        0             0                  -                100.0%
```

如何快速了解集群的健康状况？green、yellow、red？

green：每个索引的primary shard和replica shard都是active状态的

yellow：每个索引的primary shard都是active状态的，但是部分replica shard不是active状态，处于不可用的状态

red：不是所有索引的primary shard都是active状态的，部分索引有数据丢失了

#5.3.2 快速查看集群中有哪些索引

GET /_cat/indices?v

```
health status index                           uuid                   pri rep
docs.count docs.deleted store.size pri.store.size
green  open   .kibana_task_manager            JBMgpucOSzenstLcjA_G4A   1   0
   2            0     45.5kb         45.5kb
green  open   .monitoring-kibana-7-2019.09.16 LIskf15DTcS70n4Q6t2bTA   1   0
433            0    218.2kb        218.2kb
green  open   .monitoring-es-7-2019.09.16     RMeUN3tQRjqM8xBgw7Zong   1   0
3470         1724      1.9mb          1.9mb
green  open   .kibana_1                       1cRiyIdATya5xS6qK5pGJw   1   0
   4            0     18.2kb         18.2kb
```

#5.3.3 简单的索引操作

创建索引：PUT /demo_index?pretty

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

删除索引：DELETE /demo_index?pretty

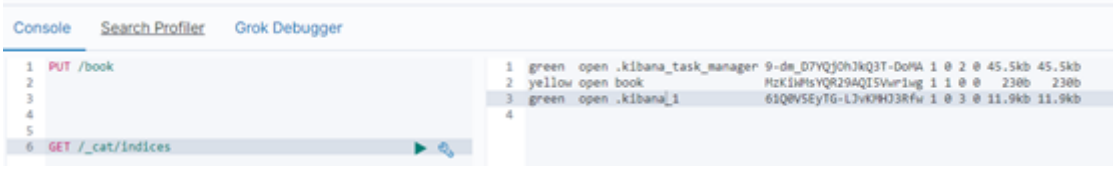
#5.4商品的CRUD操作（document CRUD操作）

#5.4.1 新建图书索引

首先建立图书索引 book

语法：put /index

PUT /book



#5.4.2 新增图书 :新增文档

语法：PUT /index/type/id

```
PUT /book/_doc/1

{
"name": "Bootstrap开发",
"description": "Bootstrap是由Twitter推出的一个前台页面开发css框架，是一个非常流行的开发框架，此框架集成了多种页面效果。此开发框架包含了大量的CSS、JS程序代码，可以帮助开发者（尤其是不擅长css页面开发的程序人员）轻松的实现一个css，不受浏览器限制的精美界面css效果。",
"studymodel": "201002",
"price":38.6,
"timestamp":"2019-08-25 19:11:35",
"pic":"group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
"tags": [ "bootstrap", "dev"]
}
```

```
PUT /book/_doc/2
{
"name": "java编程思想",
"description": "java语言是世界第一编程语言，在软件开发领域使用人数最多。",
"studymodel": "201001",
"price":68.6,
"timestamp":"2019-08-25 19:11:35",
"pic":"group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
"tags": [ "java", "dev"]
}
```

```
PUT /book/_doc/3
{
"name": "spring开发基础",
"description": "spring 在java领域非常流行，java程序员都在用。",
"studymodel": "201001",
"price":88.6,
"timestamp":"2019-08-24 19:11:35",
"pic":"group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
"tags": [ "spring", "java"]
}
```

结果

```
{
  "_index" : "book",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
```

```
  },
  "_seq_no" : 0,
  "_primary_term" : 1
}
```

#5.4.3 查询图书：检索文档

语法：GET /index/type/id

查看图书:GET /book/_doc/1 就可看到json形式的文档。方便程序解析。

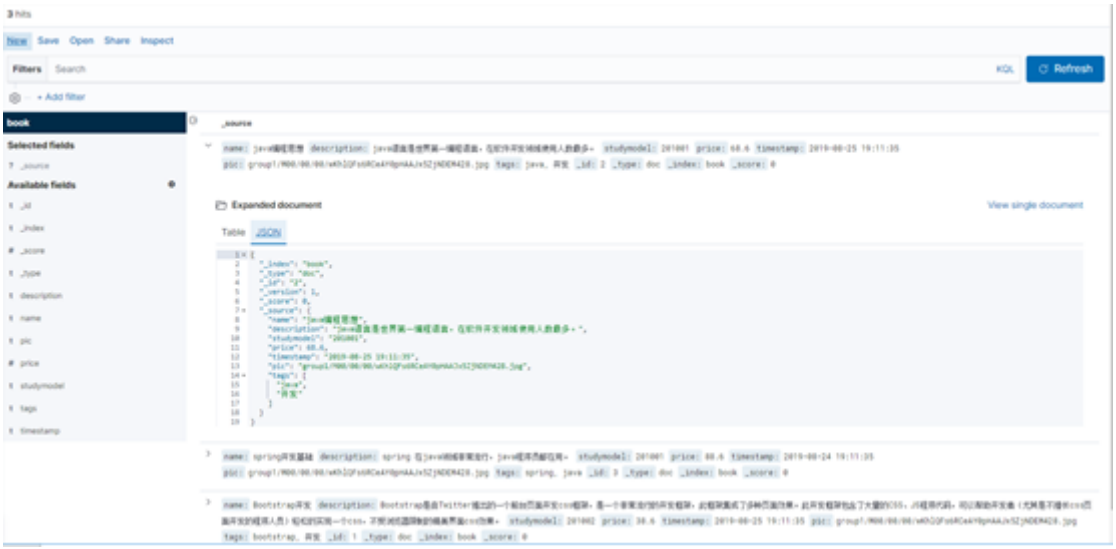
```
{
  "_index" : "book",
  "_type" : "_doc",

  "_id" : "1",

  "_version" : 4,

  "_seq_no" : 5,

  "_primary_term" : 1,

  "found" : true,

  "_source" : {

    "name" : "Bootstrap开发",

    "description" : "Bootstrap是由Twitter推出的一个前台页面开发css框架，是一个非常流行的开发框架，此框架集成了多种页面效果。此开发框架包含了大量的CSS、JS程序代码，可以帮助开发者（尤其是不擅长css页面开发的程序人员）轻松的实现一个css，不受浏览器限制的精美界面css效果。",

    "studymodel" : "201002",

    "price" : 38.6,

    "timestamp" : "2019-08-25 19:11:35",

    "pic" : "group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",

    "tags" : [

      "bootstrap",

      "开发"

    ]

  }

}
```

为方便查看索引中的数据，kibana可以如下操作

Kibana-discover- Create index pattern- Index pattern填book

下一步，再点击discover就可看到数据。

点击json还可以看到原始数据



为方便查看索引中的数据，head可以如下操作

点击数据浏览，点击book索引。

#5.4.4 修改图书：替换操作

```
PUT /book/_doc/1
{
    "name": "Bootstrap开发教程1",
    "description": "Bootstrap是由Twitter推出的一个前台页面开发css框架，是一个非常流行的开发框架，此框架集成了多种页面效果。此开发框架包含了大量的CSS、JS程序代码，可以帮助开发者（尤其是不擅长css页面开发的程序人员）轻松的实现一个css，不受浏览器限制的精美界面css效果。",
    "studymodel": "201002",
    "price":38.6,
    "timestamp":"2019-08-25 19:11:35",
    "pic":"group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
    "tags": [ "bootstrap", "开发"]
}
```

替换操作是整体覆盖，要带上所有信息。

#5.4.5 修改图书：更新文档

语法：POST /{index}/type /{id}/_update

或者POST /{index}/_update/{id}

```
POST /book/_update/1/
{
  "doc": {
   "name": " Bootstrap开发教程高级"
  }
}
```

返回：

```
{
  "_index" : "book",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 10,
  "result" : "updated",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 11,
  "_primary_term" : 1
}
```

#5.4.6 删除图书：删除文档

语法：

```
DELETE /book/_doc/1
```

返回：

```
{
  "_index" : "book",
  "_type" : "_doc",
```

```
    "_id" : "1",
    "_version" : 11,
    "result" : "deleted",
    "_shards" : {
      "total" : 2,
      "successful" : 1,
      "failed" : 0
    },
    "_seq_no" : 12,
    "_primary_term" : 1

}
```

#6. 文档document入门

#6.1. 默认自带字段解析

```
{
  "_index" : "book",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 1,
  "_seq_no" : 10,
  "_primary_term" : 1,
  "found" : true,
  "_source" : {
    "name" : "Bootstrap开发教程1",
    "description" : "Bootstrap是由Twitter推出的一个前台页面开发css框架，是一个非常流行的开发框架，此框架集成了多种页面效果。此开发框架包含了大量的CSS、JS程序代码，可以帮助开发者（尤其是不擅长css页面开发的程序人员）轻松的实现一个css，不受浏览器限制的精美界面css效果。",
    "studymodel" : "201002",
    "price" : 38.6,
    "timestamp" : "2019-08-25 19:11:35",
    "pic" : "group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
    "tags" : [
      "bootstrap",
      "开发"
    ]
  }
}
```

#6.1.1 _index

- 含义：此文档属于哪个索引
- 原则：类似数据放在一个索引中。数据库中表的定义规则。如图书信息放在book索引中，员工信息放在employee索引中。各个索引存储和搜索时互不影响。
- 定义规则：英文小写。尽量不要使用特殊字符。order user

#6.1.2 _type

- 含义：类别。book java node
- 注意：以后的es9将彻底删除此字段，所以当前版本在不断弱化type。不需要关注。见到_type都为doc。

#6.1.3 _id

含义：文档的唯一标识。就像表的id主键。结合索引可以标识和定义一个文档。

生成：手动（put /index/_doc/id）、自动

#6.1.4 创建索引时，不同数据放到不同索引中

#6.2. 生成文档id

#6.2.1 手动生成id

场景：数据从其他系统导入时，本身有唯一主键。如数据库中的图书、员工信息等。

用法：put /index/_doc/id

```
PUT /test_index/_doc/1
{
  "test_field": "test"
}
```

#6.2.2 自动生成id

用法：POST /index/_doc

```
POST /test_index/_doc
{
  "test_field": "test1"
}
```

返回：

```
{
  "_index" : "test_index",
  "_type" : "_doc",
  "_id" : "x29LOm0BPsY0gSJFYZAl",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 0,
  "_primary_term" : 1
}
```

自动id特点：

长度为20个字符，URL安全，base64编码，GUID，分布式生成不冲突

#6.3． _source 字段

#6.3.1 _source

含义：插入数据时的所有字段和值。在get获取数据时，在_source字段中原样返回。

GET /book/_doc/1

#6.3.2 定制返回字段

就像sql不要select *,而要select name,price from book …一样。

GET /book/_doc/1?__source_includes=name,price

```
{
  "_index" : "book",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 1,
  "_seq_no" : 10,
  "_primary_term" : 1,
  "found" : true,
  "_source" : {
    "price" : 38.6,
    "name" : "Bootstrap开发教程1"
  }
}
```

#6.4． 文档的替换与删除

#6.4.1全量替换

执行两次，返回结果中版本号（_version）在不断上升。此过程为全量替换。

```
PUT /test_index/_doc/1
{
  "test_field": "test"
}
```

实质：旧文档的内容不会立即删除，只是标记为deleted。适当的时机，集群会将这些文档删除。

#6.4.2 强制创建

为防止覆盖原有数据，我们在新增时，设置为强制创建，不会覆盖原有文档。

语法：PUT /index/ *doc/id/* create

```
PUT /test_index/_doc/1/_create
{
  "test_field": "test"
}
```

返回

```
{
  "error": {
    "root_cause": [
      {
        "type": "version_conflict_engine_exception",
        "reason": "[2]: version conflict, document already exists (current version [1])",
        "index_uuid": "lqzVqxZLQuCnd6LYtZsMkg",
        "shard": "0",
        "index": "test_index"
      }
    ],
    "type": "version_conflict_engine_exception",
    "reason": "[2]: version conflict, document already exists (current version [1])",
    "index_uuid": "lqzVqxZLQuCnd6LYtZsMkg",
    "shard": "0",
    "index": "test_index"
  },
  "status": 409
}
```

#6.4.3 删除

DELETE /index/_doc/id

```
DELETE  /test_index/_doc/1/
```

实质：旧文档的内容不会立即删除，只是标记为deleted。适当的时机，集群会将这些文档删除。

lazy delete

#6.5． 局部替换 partial update

使用 PUT /index/type/id 为文档全量替换，需要将文档所有数据提交。

partial update局部替换则只修改变动字段。

用法：

```
post /index/type/id/_update
{
   "doc": {
      "field": "value"
   }
}
```

#图解内部原理

内部与全量替换是一样的，旧文档标记为删除，新建一个文档。

优点：

- 大大减少网络传输次数和流量，提升性能
- 减少并发冲突发生的概率。

#演示

插入文档

```
PUT /test_index/_doc/5
{
  "test_field1": "ydl",
  "test_field2": "ydlclass"
}
```

修改字段1

```
POST /test_index/_doc/5/_update
{
  "doc": {
    "test_field2": " ydlclass 2"
  }
}
```

#6.6. 使用脚本更新

es可以内置脚本执行复杂操作。例如painless脚本。

注意：groovy脚本在es6以后就不支持了。原因是耗内存，不安全远程注入漏洞。

#6.6.1内置脚本

需求1：修改文档6的num字段，+1。

插入数据

```
PUT /test_index/_doc/6
{
  "num": 0,
  "tags": []
}
```

执行脚本操作

```
POST /test_index/_doc/6/_update
{
   "script" : "ctx._source.num+=1
}
```

查询数据

```
GET /test_index/_doc/6
```

返回

```
{
  "_index" : "test_index",
  "_type" : "_doc",
  "_id" : "6",
  "_version" : 2,
  "_seq_no" : 23,
  "_primary_term" : 1,
  "found" : true,
  "_source" : {
    "num" : 1,
    "tags" : [ ]
  }
}
```

需求2：搜索所有文档，将num字段乘以2输出

插入数据

```
PUT /test_index/_doc/7
{
  "num": 5
}
```

查询

```
GET /test_index/_search
{
  "script_fields": {
    "my_doubled_field": {
      "script": {
       "lang": "expression",
        "source": "doc['num'] * multiplier",
        "params": {
          "multiplier": 2
        }
      }
    }
  }
}
```

返回

```
{
        "_index" : "test_index",
        "_type" : "_doc",
        "_id" : "7",
        "_score" : 1.0,
        "fields" : {
          "my_doubled_field" : [
            10.0
          ]
        }
      }
```

#6.6.2 外部脚本

Painless是内置支持的。脚本内容可以通过多种途径传给 es，包括 rest 接口，或者放到 config/scripts目录等，默认开启。

注意：脚本性能低下，且容易发生注入，本教程忽略。

官方文档： https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-scripting-using.html

#6.7． 图解es的并发问题

如同秒杀，多线程情况下，es同样会出现并发冲突问题。

#6.8． 图解悲观锁与乐观锁机制

为控制并发问题，我们通常采用锁机制。分为悲观锁和乐观锁两种机制。

悲观锁：很悲观，所有情况都上锁。此时只有一个线程可以操作数据。具体例子为数据库中的行级锁、表级锁、读锁、写锁等。

特点：优点是方便，直接加锁，对程序透明。缺点是效率低。

乐观锁：很乐观，对数据本身不加锁。提交数据时，通过一种机制验证是否存在冲突，如es中通过版本号验证。

特点：优点是并发能力高。缺点是操作繁琐，在提交数据时，可能反复重试多次。

#6.9． 图解es内部基于_version乐观锁控制

#实验基于_version的版本控制

es对于文档的增删改都是基于版本号。

1新增多次文档：

```
PUT /test_index/_doc/3
{
  "test_field": "test"
}
```

返回版本号递增

2删除此文档

```
DELETE /test_index/_doc/3
```

返回

```
 DELETE /test_index/_doc/3
 {
   "_index" : "test_index",
   "_type" : "_doc",
   "_id" : "2",
```

```
    "_version" : 6,
    "result" : "deleted",
    "_shards" : {
      "total" : 2,
      "successful" : 1,
      "failed" : 0
    },
    "_seq_no" : 7,
    "_primary_term" : 1
  }
```

3再新增

```
PUT /test_index/_doc/3
{
  "test_field": "test"
}
```

可以看到版本号依然递增，验证延迟删除策略。

如果删除一条数据立马删除的话，所有分片和副本都要立马删除，对es集群压力太大。

#图解es内部并发控制

es内部主从同步时，是多线程异步。乐观锁机制。

#6.10． 演示客户端程序基于_version并发操作流程

java python客户端更新的机制。

#新建文档

```
PUT /test_index/_doc/5
{
  "test_field": "ydl"
}
```

返回：

```
 {
    "_index" : "test_index",
    "_type" : "_doc",
    "_id" : "3",
    "_version" : 1,
    "result" : "created",
    "_shards" : {
      "total" : 2,
      "successful" : 1,
      "failed" : 0
    },
    "_seq_no" : 8,
    "_primary_term" : 1
  }
```

#客户端1修改。带版本号1。

首先获取数据的当前版本号

```
GET /test_index/_doc/5
```

更新文档

```
PUT /test_index/_doc/5?version=1
{
  "test_field": "ydl1"
}
PUT /test_index/_doc/5?if_seq_no=21&if_primary_term=1
{
  "test_field": "ydl1"
}
```

#客户端2并发修改。带版本号1。

```
PUT /test_index/_doc/5?version=1
{
  "test_field": "ydl2"
}
PUT /test_index/_doc/5?if_seq_no=21&if_primary_term=1
{
  "test_field": "ydl1"
}
```

报错。

#客户端2重新查询。得到最新版本为2。seq_no=22

```
GET /test_index/_doc/4
```

#客户端2并发修改。带版本号2。

```
PUT /test_index/_doc/4?version=2
{
  "test_field": "ydl2"
}
es7
PUT /test_index/_doc/5?if_seq_no=22&if_primary_term=1
{
  "test_field": "ydl2"
}
```

修改成功。

#6.11． 演示自己手动控制版本号 external version

背景：已有数据是在数据库中，有自己手动维护的版本号的情况下，可以使用external version控制。hbase。

要求：修改时external version要大于当前文档的_version

对比：基于_version时，修改的文档version等于当前文档的版本号。

使用?version=1&version_type=external

#新建文档

```
PUT /test_index/_doc/4
{
  "test_field": "ydl"
}
```

更新文档:

#客户端1修改文档

```
PUT /test_index/_doc/4?version=2&version_type=external
{
  "test_field": "ydl1"
}
```

#客户端2同时修改

```
PUT /test_index/_doc/4?version=2&version_type=external
{
  "test_field": "ydl2"
}
```

返回:

```
{
  "error": {
    "root_cause": [
      {
        "type": "version_conflict_engine_exception",
        "reason": "[4]: version conflict, current version [2] is higher or equal to the one provided [2]",
        "index_uuid": "-rqYZ2EcSPqL6pu8Gi35jw",
        "shard": "1",
        "index": "test_index"
      }
    ],
    "type": "version_conflict_engine_exception",
    "reason": "[4]: version conflict, current version [2] is higher or equal to the one provided [2]",
    "index_uuid": "-rqYZ2EcSPqL6pu8Gi35jw",
    "shard": "1",
    "index": "test_index"
  },
  "status": 409
}
```

#客户端2重新查询数据

```
GET /test_index/_doc/4
```

#客户端2重新修改数据

```
PUT /test_index/_doc/4?version=3&version_type=external
{
  "test_field": "ydl2
}
```

#6.12. 更新时 retry_on_conflict 参数

#retry_on_conflict

指定重试次数

```
POST /test_index/_doc/5/_update?retry_on_conflict=3
{
  "doc": {
    "test_field": "ydl1"
  }
}
```

#与 _version结合使用

```
POST /test_index/_doc/5/_update?
retry_on_conflict=3&version=22&version_type=external
{
  "doc": {
    "test_field": "ydl1"
  }
}
```

#6.13. 批量查询 mget

单条查询 GET /test_index/_doc/1，如果查询多个id的文档一条一条查询，网络开销太大。

#mget 批量查询：

```
 GET /_mget
 {
    "docs" : [
       {
          "_index" : "test_index",
          "_type" :  "_doc",
          "_id" :    1
       },
       {
          "_index" : "test_index",
          "_type" :  "_doc",
          "_id" :    7
       }
    ]
 }
```

返回：

```
{
  "docs" : [
    {
      "_index" : "test_index",
      "_type" : "_doc",
      "_id" : "2",
      "_version" : 6,
      "_seq_no" : 12,
      "_primary_term" : 1,
      "found" : true,
      "_source" : {
        "test_field" : "test12333123321321"
      }
    },
    {
      "_index" : "test_index",
      "_type" : "_doc",
      "_id" : "3",
      "_version" : 6,
      "_seq_no" : 18,
      "_primary_term" : 1,
      "found" : true,
      "_source" : {
        "test_field" : "test3213"
      }
    }
  ]
}
```

提示去掉type

```
GET /_mget
{
   "docs" : [
      {
         "_index" : "test_index",
         "_id" :    2
      },
      {
         "_index" : "test_index",
         "_id" :    3
      }
   ]
}
```

#同一索引下批量查询：

```
GET /test_index/_mget
{
   "docs" : [
      {
         "_id" :    2
      },
      {
         "_id" :    3
      }
   ]
}
```

#第三种写法：搜索写法

```
post /test_index/_doc/_search
{
    "query": {
        "ids" : {
            "values" : ["1", "7"]
        }
    }
}
```

#6.14． 批量增删改 bulk

Bulk 操作解释将文档的增删改查一些列操作，通过一次请求全都做完。减少网络传输次数。

语法：

```
POST /_bulk
{"action": {"metadata"}}
{"data"}
```

如下操作，删除5，新增14，修改2。

```
POST /_bulk
{ "delete": { "_index": "test_index",  "_id": "5" }}
{ "create": { "_index": "test_index",  "_id": "14" }}
{ "test_field": "test14" }
{ "update": { "_index": "test_index",  "_id": "2"} }
{ "doc" : {"test_field" : "bulk test"} }
```

总结：

1功能：

- delete：删除一个文档，只要1个json串就可以了
- create：相当于强制创建 PUT /index/type/id/_create
- index：普通的put操作，可以是创建文档，也可以是全量替换文档
- update：执行的是局部更新partial update操作

2格式：每个json不能换行。相邻json必须换行。

3隔离：每个操作互不影响。操作失败的行会返回其失败信息。

4实际用法：bulk请求一次不要太大，否则一下积压到内存中，性能会下降。所以，一次请求几千个操作、大小在几M正好。

#6.15． 文档概念学习总结

章节回顾

1文档的增删改查

2文档字段解析

3内部锁机制

4批量查询修改

es是什么

一个分布式的文档数据存储系统distributed document store。es看做一个分布式nosql数据库。如redis\mongoDB\hbase。

文档数据：es可以存储和操作json文档类型的数据，而且这也是es的核心数据结构。 存储系统：es可以对json文档类型的数据进行存储，查询，创建，更新，删除，等等操作。

应用场景

- 大数据。es的分布式特点，水平扩容承载大数据。
- 数据结构灵活。列随时变化。使用关系型数据库将会建立大量的关联表，增加系统复杂度。
- 数据操作简单。就是查询，不涉及事务。

举例

电商页面、传统论坛页面等。面向的对象比较复杂，但是作为终端，没有太复杂的功能（事务），只涉及简单的增删改查crud。

这个时候选用ES这种NoSQL型的数据存储，比传统的复杂的事务强大的关系型数据库，更加合适一些。无论是性能，还是吞吐量，可能都会更好。

#7． Java api 实现文档管理

#7.1 es技术特点

1es技术比较特殊，不像其他分布式、大数据课程，haddop、spark、hbase。es代码层面很好写，难的是概念的理解。

2es最重要的是他的rest api。跨语言的。在真实生产中，探查数据、分析数据，使用rest更方便。

3本课程将会大量讲解内部原理及rest api。java代码会在重要的api后学习。

#7.2 java 客户端简单获取数据

java api 文档 https://www.elastic.co/guide/en/elasticsearch/client/java-rest/7.3/java-rest-overview.html

low : 偏向底层。

high：高级封装。足够。

1导包

```
1        <dependency>
```

```
        <groupId>org.elasticsearch.client</groupId>
        <artifactId>elasticsearch-rest-high-level-client</artifactId>
        <version>7.3.0</version>
        <exclusions>
            <exclusion>
                <groupId>org.elasticsearch</groupId>
                <artifactId>elasticsearch</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.elasticsearch</groupId>
        <artifactId>elasticsearch</artifactId>
        <version>7.3.0</version>
    </dependency>
```

2代码

步骤

1 获取连接客户端

2构建请求

3执行

4获取结果

```
    //获取连接客户端
    RestHighLevelClient client = new RestHighLevelClient(
            RestClient.builder(
                    new HttpHost("localhost", 9200, "http")));
    //构建请求
    GetRequest getRequest = new GetRequest("book", "1");
    // 执行
    GetResponse getResponse = client.get(getRequest, RequestOptions.DEFAULT);
    // 获取结果
    if (getResponse.isExists()) {
        long version = getResponse.getVersion();
        String sourceAsString = getResponse.getSourceAsString();//检索文档(String形式)
        System.out.println(sourceAsString);
    }
```

#7.3 结合spring-boot-test测试文档查询

0为什么使用spring boot test

- 当今趋势
- 方便开发
- 创建连接交由spring容器，避免每次请求的网络开销。

1导包

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
            <version>2.0.6.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <version>2.0.6.RELEASE</version>
        </dependency>
```

2配置 application.yml

```
spring:
  application:
    name: service-search
heima:
  elasticsearch:
    hostlist: 127.0.0.1:9200 #多个结点中间用逗号分隔
```

3代码

主类

配置类

测试类

```
@SpringBootTest
@RunWith(SpringRunner.class)
//查询文档
       @Test
    public void testGet() throws IOException {
        //构建请求
        GetRequest getRequest = new GetRequest("test_post", "1");

        //=========================可选参数 start======================
        //为特定字段配置_source_include
//        String[] includes = new String[]{"user", "message"};
//        String[] excludes = Strings.EMPTY_ARRAY;
//        FetchSourceContext fetchSourceContext = new FetchSourceContext(true, includes, excludes);
//        getRequest.fetchSourceContext(fetchSourceContext);

        //为特定字段配置_source_excludes
//        String[] includes1 = new String[]{"user", "message"};
//        String[] excludes1 = Strings.EMPTY_ARRAY;
//        FetchSourceContext fetchSourceContext1 = new FetchSourceContext(true, includes1, excludes1);
//        getRequest.fetchSourceContext(fetchSourceContext1);

        //设置路由
//        getRequest.routing("routing");

        // =========================可选参数 end=====================


        //查询 同步查询
```

```
        GetResponse getResponse = client.get(getRequest, RequestOptions.DEFAULT);

          //异步查询
  //        ActionListener<GetResponse> listener = new ActionListener<GetResponse>() {
  //            //查询成功时的立马执行的方法
  //            @Override
  //            public void onResponse(GetResponse getResponse) {
  //                long version = getResponse.getVersion();
  //                String sourceAsString = getResponse.getSourceAsString();//检索文档(String形式)
  //                System.out.println(sourceAsString);
  //            }
  //
  //            //查询失败时的立马执行的方法
  //            @Override
  //            public void onFailure(Exception e) {
  //                e.printStackTrace();
  //            }
  //        };
  //        //执行异步请求
  //        client.getAsync(getRequest, RequestOptions.DEFAULT, listener);
  //        try {
  //            Thread.sleep(5000);
  //        } catch (InterruptedException e) {
  //            e.printStackTrace();
  //        }

          // 获取结果
          if (getResponse.isExists()) {
              long version = getResponse.getVersion();

              String sourceAsString = getResponse.getSourceAsString();//检索文档(String形式)
              System.out.println(sourceAsString);
              byte[] sourceAsBytes = getResponse.getSourceAsBytes();//以字节接受
              Map<String, Object> sourceAsMap = getResponse.getSourceAsMap();
              System.out.println(sourceAsMap);

          }else {

          }

      }
```

#7.4 结合spring-boot-test测试文档新增

rest api

```
PUT test_post/_doc/2
{
  "user":"tomas",
  "postDate":"2019-07-18",
  "message":"trying out es1"
}
```

代码：

```
@Test
    public void testAdd() throws IOException {
//        1构建请求
        IndexRequest request=new IndexRequest("test_posts");
        request.id("3");
//        ======================构建文档============================
//        构建方法1
        String jsonString="{\n" +
                "  \"user\":\"tomas J\",\n" +
                "  \"postDate\":\"2019-07-18\",\n" +
                "  \"message\":\"trying out es3\"\n" +
                "}";
        request.source(jsonString, XContentType.JSON);

//        构建方法2
//        Map<String,Object> jsonMap=new HashMap<>();
//        jsonMap.put("user", "tomas");
//        jsonMap.put("postDate", "2019-07-18");
//        jsonMap.put("message", "trying out es2");
//        request.source(jsonMap);

//        构建方法3
//        XContentBuilder builder= XContentFactory.jsonBuilder();
//        builder.startObject();
//        {
//            builder.field("user", "tomas");
//            builder.timeField("postDate", new Date());
//            builder.field("message", "trying out es2");
//        }
//        builder.endObject();
//        request.source(builder);
//        构建方法4
//        request.source("user","tomas",
//                    "postDate",new Date(),
//                "message","trying out es2");
//
//        ========================可选参数===================================
        //设置超时时间
        request.timeout(TimeValue.timeValueSeconds(1));
        request.timeout("1s");

        //自己维护版本号
//        request.version(2);
//        request.versionType(VersionType.EXTERNAL);



//        2执行
        //同步
        IndexResponse indexResponse = client.index(request,
RequestOptions.DEFAULT);
        //异步
//        ActionListener<IndexResponse> listener=new
ActionListener<IndexResponse>() {
//            @Override
//            public void onResponse(IndexResponse indexResponse) {
```

```
//
//            }
//
//            @Override
//            public void onFailure(Exception e) {
//
//            }
//        };
//        client.indexAsync(request,RequestOptions.DEFAULT, listener );
//        try {
//            Thread.sleep(5000);
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        }


//        3获取结果
        String index = indexResponse.getIndex();
        String id = indexResponse.getId();
        //获取插入的类型
        if(indexResponse.getResult()== DocWriteResponse.Result.CREATED){
            DocWriteResponse.Result result=indexResponse.getResult();
            System.out.println("CREATED:"+result);
        }else if(indexResponse.getResult()== DocWriteResponse.Result.UPDATED){
            DocWriteResponse.Result result=indexResponse.getResult();
            System.out.println("UPDATED:"+result);
        }

        ReplicationResponse.ShardInfo shardInfo = indexResponse.getShardInfo();
        if(shardInfo.getTotal()!=shardInfo.getSuccessful()){
            System.out.println("处理成功的分片数少于总分片！");
        }
        if(shardInfo.getFailed()>0){
           for (ReplicationResponse.ShardInfo.Failure
failure:shardInfo.getFailures()) {
                String reason = failure.reason();//处理潜在的失败原因
                System.out.println(reason);
            }
        }
    }
```

#7.5结合spring-boot-test测试文档修改

rest api

```
post /test_posts/_doc/3/_update
{
   "doc": {
      "user": "tomas J"
   }
}
```

代码：

```
    @Test
      public void testUpdate() throws IOException {
```

```
//        1构建请求
        UpdateRequest request = new UpdateRequest("test_posts", "3");
        Map<String, Object> jsonMap = new HashMap<>();
        jsonMap.put("user", "tomas JJ");
        request.doc(jsonMap);
//==============================可选参数=========================================
        request.timeout("1s");//超时时间

        //重试次数
        request.retryOnConflict(3);

        //设置在继续更新之前，必须激活的分片数
//        request.waitForActiveShards(2);
        //所有分片都是active状态，才更新
//        request.waitForActiveShards(ActiveShardCount.ALL);

//        2执行
//        同步
        UpdateResponse updateResponse = client.update(request,
RequestOptions.DEFAULT);
//        异步

//        3获取数据
        updateResponse.getId();
        updateResponse.getIndex();

        //判断结果
        if (updateResponse.getResult() == DocWriteResponse.Result.CREATED) {
            DocWriteResponse.Result result = updateResponse.getResult();
            System.out.println("CREATED:" + result);
        } else if (updateResponse.getResult() == DocWriteResponse.Result.UPDATED)
{
            DocWriteResponse.Result result = updateResponse.getResult();
            System.out.println("UPDATED:" + result);
        }else if(updateResponse.getResult() == DocWriteResponse.Result.DELETED){
            DocWriteResponse.Result result = updateResponse.getResult();
            System.out.println("DELETED:" + result);
        }else if (updateResponse.getResult() == DocWriteResponse.Result.NOOP){
            //没有操作
            DocWriteResponse.Result result = updateResponse.getResult();
            System.out.println("NOOP:" + result);
        }
    }
```

#7.6结合spring-boot-test测试文档删除

rest api

```
DELETE /test_posts/_doc/3
```

代码

```
 @Test
    public void testDelete() throws IOException {
//        1构建请求
        DeleteRequest request =new DeleteRequest("test_posts","3");
```

```
        //可选参数


//        2执行
        DeleteResponse deleteResponse = client.delete(request,
RequestOptions.DEFAULT);


//        3获取数据
        deleteResponse.getId();
        deleteResponse.getIndex();

        DocWriteResponse.Result result = deleteResponse.getResult();
        System.out.println(result);
}
```

#7.7结合spring-boot-test测试文档bulk

rest api

```
POST /_bulk
{"action": {"metadata"}}
{"data"}
```

代码

```
@Test
    public void testBulk() throws IOException {
//        1创建请求
        BulkRequest request = new BulkRequest();
//        request.add(new IndexRequest("post").id("1").source(XContentType.JSON,
"field", "1"));
//        request.add(new IndexRequest("post").id("2").source(XContentType.JSON,
"field", "2"));

        request.add(new UpdateRequest("post","2").doc(XContentType.JSON, "field",
"3"));
        request.add(new DeleteRequest("post").id("1"));

//        2执行
        BulkResponse bulkResponse = client.bulk(request, RequestOptions.DEFAULT);

        for (BulkItemResponse itemResponse : bulkResponse) {
            DocWriteResponse itemResponseResponse = itemResponse.getResponse();

            switch (itemResponse.getOpType()) {
                case INDEX:
                case CREATE:
                    IndexResponse indexResponse = (IndexResponse)
itemResponseResponse;
                    indexResponse.getId();
                    System.out.println(indexResponse.getResult());
                    break;
                case UPDATE:
                    UpdateResponse updateResponse = (UpdateResponse)
itemResponseResponse;
```

```
                    updateResponse.getIndex();
                    System.out.println(updateResponse.getResult());
                    break;
                case DELETE:
                    DeleteResponse deleteResponse = (DeleteResponse)
    itemResponseResponse;
                    System.out.println(deleteResponse.getResult());
                    break;
            }
        }
    }

```

#8. 图解es内部机制

#8.1. 图解es分布式基础

#8.1.1es对复杂分布式机制的透明隐藏特性

- 分布式机制：分布式数据存储及共享。
- 分片机制：数据存储到哪个分片，副本数据写入。
- 集群发现机制：cluster discovery。新启动es实例，自动加入集群。
- shard负载均衡：大量数据写入及查询，es会将数据平均分配。
- shard副本：新增副本数，分片重分配。

#8.1.2Elasticsearch的垂直扩容与水平扩容

垂直扩容：使用更加强大的服务器替代老服务器。但单机存储及运算能力有上线。且成本直线上升。如10t服务器1万。单个10T服务器可能20万。

水平扩容：采购更多服务器，加入集群。大数据。

#8.1.3增减或减少节点时的数据rebalance

新增或减少es实例时，es集群会将数据重新分配。

#8.1.4master节点

功能：

- 创建删除节点
- 创建删除索引

#8.1.5节点对等的分布式架构

- 节点对等，每个节点都能接收所有的请求
- 自动请求路由
- 响应收集

#8.2. 图解分片shard、副本replica机制

#8.2.1shard&replica机制

（1）每个index包含一个或多个shard

（2）每个shard都是一个最小工作单元，承载部分数据，lucene实例，完整的建立索引和处理请求的能力

（3）增减节点时，shard会自动在nodes中负载均衡

（4）primary shard和replica shard，每个document肯定只存在于某一个primary shard以及其对应的replica shard中，不可能存在于多个primary shard

（5）replica shard是primary shard的副本，负责容错，以及承担读请求负载

（6）primary shard的数量在创建索引的时候就固定了，replica shard的数量可以随时修改

（7）primary shard的默认数量是1，replica默认是1，默认共有2个shard，1个primary shard，1个replica shard

注意：es7以前primary shard的默认数量是5，replica默认是1，默认有10个shard，5个primary shard，5个replica shard

（8）primary shard不能和自己的replica shard放在同一个节点上（否则节点宕机，primary shard和副本都丢失，起不到容错的作用），但是可以和其他primary shard的replica shard放在同一个节点上

#8.3图解单node环境下创建index是什么样子的

（1）单node环境下，创建一个index，有3个primary shard，3个replica shard （2）集群status是yellow （3）这个时候，只会将3个primary shard分配到仅有的一个node上去，另外3个replica shard是无法分配的 （4）集群可以正常工作，但是一旦出现节点宕机，数据全部丢失，而且集群不可用，无法承接任何请求

```
PUT /test_index1
{
   "settings" : {
      "number_of_shards" : 3,
      "number_of_replicas" : 1
   }
}
```

#8.4图解2个node环境下replica shard是如何分配的

（1）replica shard分配：3个primary shard，3个replica shard，1 node （2）primary ---> replica同步 （3）读请求：primary/replica

#8.5图解横向扩容

- 分片自动负载均衡，分片向空闲机器转移。
- 每个节点存储更少分片，系统资源给与每个分片的资源更多，整体集群性能提高。
- 扩容极限：节点数大于整体分片数，则必有空闲机器。
- 超出扩容极限时，可以增加副本数，如设置副本数为2，总共3*3=9个分片。9台机器同时运行，存储和搜索性能更强。容错性更好。

- 容错性：只要一个索引的所有主分片在，集群就就可以运行。

#8.6 图解es容错机制 master选举，replica容错，数据恢复

以3分片，2副本数，3节点为例介绍。

- master node宕机，自动master选举，集群为red
- replica容错：新master将replica提升为primary shard，yellow
- 重启宕机node，master copy replica到该node，使用原有的shard并同步宕机后的修改，green

#9． 图解文档存储机制

#9.1． 数据路由

#9.1.1文档存储如何路由到相应分片

一个文档，最终会落在主分片的一个分片上，到底应该在哪一个分片？这就是数据路由。

#9.1.2路由算法

```
shard = hash(routing) % number_of_primary_shards
```

哈希值对主分片数取模。

举例：

对一个文档经行crud时，都会带一个路由值 routing number。默认为文档_id（可能是手动指定，也可能是自动生成）。

存储1号文档，经过哈希计算，哈希值为2,此索引有3个主分片，那么计算2%3=2，就算出此文档在P2分片上。

决定一个document在哪个shard上，最重要的一个值就是routing值，默认是_id，也可以手动指定，相同的routing值，每次过来，从hash函数中，产出的hash值一定是相同的

无论hash值是几，无论是什么数字，对number_of_primary_shards求余数，结果一定是在0~number_of_primary_shards-1之间这个范围内的。0,1,2。

#9.1.3手动指定 routing number

```
PUT /test_index/_doc/15?routing=num
{
  "num": 0,
  "tags": []
}
```

场景：在程序中，架构师可以手动指定已有数据的一个属性为路由值，好处是可以定制一类文档数据存储到一个分片中。缺点是设计不好，会造成数据倾斜。

所以，不同文档尽量放到不同的索引中。剩下的事情交给es集群自己处理。

#9.1.4主分片数量不可变

涉及到以往数据的查询搜索，所以一旦建立索引，主分片数不可变。

#9.2． 图解文档的增删改内部机制

增删改可以看做update,都是对数据的改动。一个改动请求发送到es集群，经历以下四个步骤：

（1）客户端选择一个node发送请求过去，这个node就是coordinating node（协调节点）

（2）coordinating node，对document进行路由，将请求转发给对应的node（有primary shard）

（3）实际的node上的primary shard处理请求，然后将数据同步到replica node。

（4）coordinating node，如果发现primary node和所有replica node都搞定之后，就返回响应结果给客户端。

#9.3． 图解文档的查询内部机制

1、客户端发送请求到任意一个node，成为coordinate node

2、coordinate node对document进行路由，将请求转发到对应的node，此时会使用round-robin随机轮询算法，在primary shard以及其所有replica中随机选择一个，让读请求负载均衡

3、接收请求的node返回document给coordinate node

4、coordinate node返回document给客户端

5、特殊情况：document如果还在建立索引过程中，可能只有primary shard有，任何一个replica shard都没有，此时可能会导致无法读取到document，但是document完成索引建立之后，primary shard和replica shard就都有了。

#9.4． bulk api奇特的json格式

```
POST /_bulk
{"action": {"meta"}}\n
{"data"}\n
{"action": {"meta"}}\n
{"data"}\n

[
    {
        "action":{
            "method":"create"
        },
        "data":{
            "id":1,
            "field1":"java",
            "field1":"spring",
        }
    },
      {
        "action":{
```

```
            "method":"create"
        },
        "data":{
            "id":2,
            "field1":"java",
            "field1":"spring",
        }
    }
]
```

1、bulk中的每个操作都可能要转发到不同的node的shard去执行

2、如果采用比较良好的json数组格式

允许任意的换行，整个可读性非常棒，读起来很爽，es拿到那种标准格式的json串以后，要按照下述流程去进行处理

（1）将json数组解析为JSONArray对象，这个时候，整个数据，就会在内存中出现一份一模一样的拷贝，一份数据是json文本，一份数据是JSONArray对象

（2）解析json数组里的每个json，对每个请求中的document进行路由

（3）为路由到同一个shard上的多个请求，创建一个请求数组。100请求中有10个是到P1.

（4）将这个请求数组序列化

（5）将序列化后的请求数组发送到对应的节点上去

3、耗费更多内存，更多的jvm gc开销

我们之前提到过bulk size最佳大小的那个问题，一般建议说在几千条那样，然后大小在10MB左右，所以说，可怕的事情来了。假设说现在100个bulk请求发送到了一个节点上去，然后每个请求是10MB，100个请求，就是1000MB = 1GB，然后每个请求的json都copy一份为jsonarray对象，此时内存中的占用就会翻倍，就会占用2GB的内存，甚至还不止。因为弄成jsonarray之后，还可能会多搞一些其他的数据结构，2GB+的内存占用。

占用更多的内存可能就会积压其他请求的内存使用量，比如说最重要的搜索请求，分析请求，等等，此时就可能会导致其他请求的性能急速下降。

另外的话，占用内存更多，就会导致java虚拟机的垃圾回收次数更多，跟频繁，每次要回收的垃圾对象更多，耗费的时间更多，导致es的java虚拟机停止工作线程的时间更多。

4、现在的奇特格式

```
POST /_bulk
{ "delete": { "_index": "test_index",  "_id": "5" }} \n
{ "create": { "_index": "test_index",  "_id": "14" }}\n
{ "test_field": "test14" }\n
{ "update": { "_index": "test_index",  "_id": "2"} }\n
{ "doc" : {"test_field" : "bulk test"} }\n
```

（1）不用将其转换为json对象，不会出现内存中的相同数据的拷贝，直接按照换行符切割json

（2）对每两个一组的json，读取meta，进行document路由

（3）直接将对应的json发送到node上去

5、最大的优势在于，不需要将json数组解析为一个JSONArray对象，形成一份大数据的拷贝，浪费内存空间，尽可能地保证性能。

#10. Mapping映射入门

#10.1. 什么是mapping映射

概念：自动或手动为index中的_doc建立的一种数据结构和相关配置，简称为mapping映射。

插入几条数据，让es自动为我们建立一个索引

```
PUT /website/_doc/1
{
  "post_date": "2019-01-01",
  "title": "my first article",
  "content": "this is my first article in this website",
  "author_id": 11400
}

PUT /website/_doc/2
{
  "post_date": "2019-01-02",
  "title": "my second article",
  "content": "this is my second article in this website",
  "author_id": 11400
}

PUT /website/_doc/3
{
  "post_date": "2019-01-03",
  "title": "my third article",
  "content": "this is my third article in this website",
  "author_id": 11400
}
```

对比数据库建表语句

```
create table website(
     post_date date,
     title varchar(50),
     content varchar(100),
     author_id int(11)
);
```

动态映射：dynamic mapping，自动为我们建立index，以及对应的mapping，mapping中包含了每个field对应的数据类型，以及如何分词等设置。

重点：我们当然，后面会讲解，也可以手动在创建数据之前，先创建index，以及对应的mapping

```
GET  /website/_mapping/
{
  "website" : {
    "mappings" : {
      "properties" : {
        "author_id" : {
          "type" : "long"
        },
        "content" : {
```

```
            "type" : "text",
            "fields" : {
              "keyword" : {
                "type" : "keyword",
                "ignore_above" : 256
              }
            }
          },
          "post_date" : {
            "type" : "date"
          },
          "title" : {
            "type" : "text",
            "fields" : {
              "keyword" : {
                "type" : "keyword",
                "ignore_above" : 256
              }
            }
          }
        }
      }
    }
  }
```

尝试各种搜索

```
GET /website/_search?q=2019        0条结果
GET /website/_search?q=2019-01-01           1条结果
GET /website/_search?q=post_date:2019-01-01     1条结果
GET /website/_search?q=post_date:2019          0 条结果
```

搜索结果为什么不一致，因为es自动建立mapping的时候，设置了不同的field不同的data type。不同的data type的分词、搜索等行为是不一样的。所以出现了_all field和post_date field的搜索表现完全不一样。

#10.2． 精确匹配与全文搜索的对比分析

#10.2.1 exact value 精确匹配

2019-01-01，exact value，搜索的时候，必须输入2019-01-01，才能搜索出来

如果你输入一个01，是搜索不出来的

select * from book where name= 'java'

#10.2.2 full text 全文检索

搜“笔记电脑”，笔记本电脑词条会不会出现。

select * from book where name like '%java%'

（1）缩写 vs. 全称：cn vs. china

（2）格式转化：like liked likes

（3）大小写：Tom vs tom

（4）同义词：like vs love

2019-01-01，2019 01 01，搜索2019，或者01，都可以搜索出来

china，搜索cn，也可以将china搜索出来

likes，搜索like，也可以将likes搜索出来

Tom，搜索tom，也可以将Tom搜索出来

like，搜索love，同义词，也可以将like搜索出来

就不是说单纯的只是匹配完整的一个值，而是可以对值进行拆分词语后（分词）进行匹配，也可以通过缩写、时态、大小写、同义词等进行匹配。深入 NPL,自然语义处理。

#10.3． 全文检索下倒排索引核心原理快速揭秘

doc1：I really liked my small dogs, and I think my mom also liked them.

doc2：He never liked any dogs, so I hope that my mom will not expect me to liked him.

#分词，初步的倒排索引的建立

term	doc1	doc2
I	*	*
really	*	
liked	*	*
my	*	*
small	*	
dogs	*	
and	*	
think	*	
mom	*	*
also	*	
them	*	
He		*
never		*
any		*
so		*
hope		*
that		*
will		*
not		*
expect		*
me		*
to		*
him		*

演示了一下倒排索引最简单的建立的一个过程

#搜索

mother like little dog，不可能有任何结果

mother

like

little

dog

这不是我们想要的结果。同义词mom\mother在我们人类看来是一样。想进行标准化操作。

#重建倒排索引

normalization正规化，建立倒排索引的时候，会执行一个操作，也就是说对拆分出的各个单词进行相应的处理，以提升后面搜索的时候能够搜索到相关联的文档的概率

时态的转换，单复数的转换，同义词的转换，大小写的转换

mom -> mother

liked -> like

small -> little

dogs -> dog

重新建立倒排索引，加入normalization，再次用mother liked little dog搜索，就可以搜索到了

word	doc1	doc2	normalization
I	*	*	
really	*		
like	*	*	liked -> like
my	*	*	
little	*		small -> little
dog	*		dogs -> dog
and	*		
think	*		
mother	*	*	mom -> mother
also	*		
them	*		
He		*	
never		*	
any		*	
so		*	
hope		*	
that		*	
will		*	
not		*	
expect		*	
me		*	
to		*	
him		*	

#重新搜索

搜索：mother liked little dog，

对搜索条件经行分词 normalization

mother

liked -》like

little

dog

doc1和doc2都会搜索出来

#10.4. 分词器 analyzer

#10.4.1什么是分词器 analyzer

作用：切分词语，normalization（提升recall召回率）

给你一段句子，然后将这段句子拆分成一个一个的单个的单词，同时对每个单词进行normalization（时态转换，单复数转换）

recall，召回率：搜索的时候，增加能够搜索到的结果的数量

analyzer 组成部分：

1、character filter：在一段文本进行分词之前，先进行预处理，比如说最常见的就是，过滤html标签（hello --> hello），& --> and（I&you --> I and you）

2、tokenizer：分词，hello you and me --> hello, you, and, me

3、token filter：lowercase，stop word，synonymom，dogs --> dog，liked --> like，Tom --> tom，a/the/an --> 干掉，mother --> mom，small --> little

stop word 停用词： 了 的 呢。

一个分词器，很重要，将一段文本进行各种处理，最后处理好的结果才会拿去建立倒排索引。

#10.4.2内置分词器的介绍

例句：Set the shape to semi-transparent by calling set_trans(5)

standard analyzer标准分词器：set, the, shape, to, semi, transparent, by, calling, set_trans, 5（默认的是standard）

simple analyzer简单分词器：set, the, shape, to, semi, transparent, by, calling, set, trans

whitespace analyzer：Set, the, shape, to, semi-transparent, by, calling, set_trans(5)

language analyzer（特定的语言的分词器，比如说，english，英语分词器）：set, shape, semi, transpar, call, set_tran, 5

官方文档：

https://www.elastic.co/guide/en/elasticsearch/reference/7.4/analysis-analyzers.html

Analyzers

edit

Elasticsearch ships with a wide range of built-in analyzers, which can be used in any index without further configuration:

Standard Analyzer
: The `standard` analyzer divides text into terms on word boundaries, as defined by the Unicode Text Segmentation algorithm. It removes most punctuation, lowercases terms, and supports removing stop words.

Simple Analyzer
: The `simple` analyzer divides text into terms whenever it encounters a character which is not a letter. It lowercases all terms.

Whitespace Analyzer
: The `whitespace` analyzer divides text into terms whenever it encounters any whitespace character. It does not lowercase terms.

Stop Analyzer
: The `stop` analyzer is like the `simple` analyzer, but also supports removal of stop words.

Keyword Analyzer
: The `keyword` analyzer is a "noop" analyzer that accepts whatever text it is given and outputs the exact same text as a single term.

Pattern Analyzer
: The `pattern` analyzer uses a regular expression to split the text into terms. It supports lower-casing and stop words.

Language Analyzers
: Elasticsearch provides many language-specific analyzers like `english` or `french`.

Fingerprint Analyzer
: The `fingerprint` analyzer is a specialist analyzer which creates a fingerprint which can be used for duplicate detection.

#10.5. query string根据字段分词策略

#10.5.1query string分词

query string必须以和index建立时相同的analyzer进行分词

query string对exact value和full text的区别对待

如： date：exact value 精确匹配

text: full text 全文检索

#10.5.2测试分词器

```
GET /_analyze
{
  "analyzer": "standard",
  "text": "Text to analyze 80"
}
```

返回值：

```
{
  "tokens" : [
    {
      "token" : "text",
      "start_offset" : 0,
      "end_offset" : 4,
      "type" : "<ALPHANUM>",
      "position" : 0
    },
    {
      "token" : "to",
      "start_offset" : 5,
      "end_offset" : 7,
      "type" : "<ALPHANUM>",
      "position" : 1
    },
    {
      "token" : "analyze",
      "start_offset" : 8,
      "end_offset" : 15,
      "type" : "<ALPHANUM>",
      "position" : 2
    },
    {
      "token" : "80",
      "start_offset" : 16,
      "end_offset" : 18,
      "type" : "<NUM>",
      "position" : 3
    }
  ]
}
```

token 实际存储的term 关键字

position 在此词条在原文本中的位置

start_offset/end_offset字符在原始字符串中的位置

#10.6． mapping回顾总结

（1）往es里面直接插入数据，es会自动建立索引，同时建立对应的mapping。(dynamic mapping)

（2）mapping中就自动定义了每个field的数据类型

（3）不同的数据类型（比如说text和date），可能有的是exact value，有的是full text

（4）exact value，在建立倒排索引的时候，分词的时候，是将整个值一起作为一个关键词建立到倒排索引中的；full text，会经历各种各样的处理，分词，normaliztion（时态转换，同义词转换，大小写转换），才会建立到倒排索引中。

（5）同时呢，exact value和full text类型的field就决定了，在一个搜索过来的时候，对exact value field或者是full text field进行搜索的行为也是不一样的，会跟建立倒排索引的行为保持一致；比如说exact value搜索的时候，就是直接按照整个值进行匹配，full text query string，也会进行分词和normalization再去倒排索引中去搜索

（6）可以用es的dynamic mapping，让其自动建立mapping，包括自动设置数据类型；也可以提前手动创建index和tmapping，自己对各个field进行设置，包括数据类型，包括索引行为，包括分词器，等。

#10.7． mapping的核心数据类型以及dynamic mapping

#10.7.1 核心的数据类型

string :text and keyword

byte，short，integer，long,float，double

boolean

date

详见：https://www.elastic.co/guide/en/elasticsearch/reference/7.3/mapping-types.html

下图是ES7.3核心的字段类型如下：

Field datatypes

Elasticsearch supports a number of different datatypes for the fields in a document:

Core datatypes

string

`text` and `keyword`

Numeric

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

Date

`date`

Date nanoseconds

`date_nanos`

Boolean

`boolean`

Binary

`binary`

Range

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

Complex datatypes

Object

`object` for single JSON objects

Nested

`nested` for arrays of JSON objects

Geo datatypes

Geo-point

`geo_point` for lat/lon points

#10.7.2 dynamic mapping 推测规则

true or false --> boolean

123 --> long

123.45 --> double

2019-01-01 --> date

"hello world" --> text/keywod

#10.7.3 查看mapping

GET /index/_mapping/

#10.8 手动管理mapping

#10.8.1查询所有索引的映射

GET /_mapping

#10.8.2 创建映射！！

创建索引后，应该立即手动创建映射

```
PUT book/_mapping
{
    "properties": {
           "name": {
                  "type": "text"
            },
           "description": {
              "type": "text",
              "analyzer":"english",
              "search_analyzer":"english"
           },
           "pic":{
             "type":"text",
             "index":false
           },
           "studymodel":{
             "type":"text"
           }
    }
}
```

Text 文本类型

1）analyzer

通过analyzer属性指定分词器。

上边指定了analyzer是指在索引和搜索都使用english，如果单独想定义搜索时使用的分词器则可以通过search_analyzer属性。

2）index

index属性指定是否索引。

默认为index=true，即要进行索引，只有进行索引才可以从索引库搜索到。

但是也有一些内容不需要索引，比如：商品图片地址只被用来展示图片，不进行搜索图片，此时可以将index设置为false。

删除索引，重新创建映射，将pic的index设置为false，尝试根据pic去搜索，结果搜索不到数据。

3）store

是否在source之外存储，每个文档索引后会在 ES中保存一份原始文档，存放在"*source*"中，一般情况下不需要设置*store*为*true*，因为在 source中已经有一份原始文档了。

测试

```
PUT book/_mapping
{
        "properties": {
           "name": {
                  "type": "text"
            },
           "description": {
              "type": "text",
              "analyzer":"english",
              "search_analyzer":"english"
           },
           "pic":{
             "type":"text",
             "index":false
           },
           "studymodel":{
             "type":"text"
           }
    }
}
```

插入文档：

```
PUT /book/_doc/1
{
  "name":"Bootstrap开发框架",
  "description":"Bootstrap是由Twitter推出的一个前台页面开发框架，在行业之中使用较为广泛。此开发框架包含了大量的CSS、JS程序代码，可以帮助开发者（尤其是不擅长页面开发的程序人员）轻松的实现一个不受浏览器限制的精美界面效果。",
  "pic":"group1/M00/00/01/wKhlQFqO4MmAOP53AAAcwDwm6SU490.jpg",
  "studymodel":"201002"
}
```

Get /book/_search?q=name:开发

Get /book/_search?q=description:开发

Get /book/_search?q=pic:group1/M00/00/01/wKhlQFqO4MmAOP53AAAcwDwm6SU490.jpg

Get /book/_search?q=studymodel:201002

通过测试发现：name和description都支持全文检索，pic不可作为查询条件。

#keyword关键字字段

目前已经取代了"index": false。上边介绍的text文本字段在映射时要设置分词器，keyword字段为关键字字段，通常搜索keyword是按照整体搜索，所以创建keyword字段的索引时是不进行分词的，比如：邮政编码、手机号码、身份证等。keyword字段通常用于过虑、排序、聚合等。

#date日期类型

日期类型不用设置分词器。

通常日期类型的字段用于排序。

format

通过format设置日期格式

例子：

下边的设置允许date字段存储年月日时分秒、年月日及毫秒三种格式。

```
{
 "properties": {
 "timestamp": {
 "type": "date",
 "format": "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd"
 }
 }
}
```

插入文档：

```
Post book/doc/3
{
"name": "spring开发基础",
"description": "spring 在java领域非常流行，java程序员都在用。",
"studymodel": "201001",
"pic":"group1/M00/00/01/wKhlQFqO4MmAOP53AAAcwDwm6SU490.jpg",
"timestamp":"2018-07-04 18:28:58"
}
```

#数值类型

下边是ES支持的数值类型

Numeric datatypes

edit

The following numeric types are supported:

`long`	A signed 64-bit integer with a minimum value of -2^{63} and a maximum value of $2^{63}-1$.
`integer`	A signed 32-bit integer with a minimum value of -2^{31} and a maximum value of $2^{31}-1$.
`short`	A signed 16-bit integer with a minimum value of `-32,768` and a maximum value of `32,767`.
`byte`	A signed 8-bit integer with a minimum value of `-128` and a maximum value of `127`.
`double`	A double-precision 64-bit IEEE 754 floating point number, restricted to finite values.
`float`	A single-precision 32-bit IEEE 754 floating point number, restricted to finite values.
`half_float`	A half-precision 16-bit IEEE 754 floating point number, restricted to finite values.
`scaled_float`	A floating point number that is backed by a `long`, scaled by a fixed `double` scaling factor.

1、尽量选择范围小的类型，提高搜索效率

2、对于浮点数尽量用比例因子，比如一个价格字段，单位为元，我们将比例因子设置为100这在ES中会按 分 存储，映射如下：

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

由于比例因子为100，如果我们输入的价格是23.45则ES中会将23.45乘以100存储在ES中。

如果输入的价格是23.456，ES会将23.456乘以100再取一个接近原始值的数，得出2346。

使用比例因子的好处是整型比浮点型更易压缩，节省磁盘空间。

如果比例因子不适合，则从下表选择范围小的去用：

更新已有映射，并插入文档：

```
PUT book/doc/3
{
"name": "spring开发基础",
"description": "spring 在java领域非常流行，java程序员都在用。",
"studymodel": "201001",
 "pic":"group1/M00/00/01/wKhlQFqO4MmAOP53AAAcwDwm6SU490.jpg",
 "timestamp":"2018-07-04 18:28:58",
 "price":38.6
}
```

#10.8.3修改映射

只能创建index时手动建立mapping，或者新增field mapping，但是不能update field mapping。

因为已有数据按照映射早已分词存储好。如果修改，那这些存量数据怎么办。

新增一个字段mapping

```
PUT /book/_mapping/
{
  "properties" : {
    "new_field" : {
      "type" :    "text",
     "index":    "false"
    }
  }
}
```

如果修改mapping,会报错

```
PUT /book/_mapping/
{
  "properties" : {
    "studymodel" : {
     "type" :    "keyword"
    }
  }
}
```

返回:

```
{
  "error": {
    "root_cause": [
      {
        "type": "illegal_argument_exception",
        "reason": "mapper [studymodel] of different type, current_type [text], merged_type [keyword]"
      }
    ],
    "type": "illegal_argument_exception",
    "reason": "mapper [studymodel] of different type, current_type [text], merged_type [keyword]"
  },
  "status": 400
}
```

#10.8.4删除映射

通过删除索引来删除映射。

#10.9 复杂数据类型

#10.9 .1 multivalue field

{ "tags": ["tag1", "tag2"]}

建立索引时与string是一样的，数据类型不能混

#10.9 .2. empty field

null，[]，[null]

#10.9 .3. object field

```
PUT /company/_doc/1
{
  "address": {
    "country": "china",
    "province": "guangdong",
    "city": "guangzhou"
  },
  "name": "jack",
  "age": 27,
  "join_date": "2019-01-01"
}
```

address：object类型

查询映射

```
GET /company/_mapping
{
  "company" : {
    "mappings" : {
      "properties" : {
        "address" : {
          "properties" : {
            "city" : {
              "type" : "text",
              "fields" : {
                "keyword" : {
                  "type" : "keyword",
                  "ignore_above" : 256
                }
              }
            },
            "country" : {
              "type" : "text",
              "fields" : {
                "keyword" : {
                  "type" : "keyword",
                  "ignore_above" : 256
                }
              }
            },
            "province" : {
              "type" : "text",
              "fields" : {
                "keyword" : {
                  "type" : "keyword",
```

```
                    "ignore_above" : 256
                  }
                }
              }
            }
          },
          "age" : {
            "type" : "long"
          },
          "join_date" : {
            "type" : "date"
          },
          "name" : {
            "type" : "text",
            "fields" : {
              "keyword" : {
                "type" : "keyword",
                "ignore_above" : 256
              }
            }
          }
        }
      }
    }
  }
```

object

```
{
  "address": {
    "country": "china",
    "province": "guangdong",
    "city": "guangzhou"
  },
  "name": "jack",
  "age": 27,
  "join_date": "2017-01-01"
}
```

底层存储格式

```
{
    "name":            [jack],
    "age":          [27],
    "join_date":      [2017-01-01],
    "address.country":         [china],
    "address.province":   [guangdong],
    "address.city":  [guangzhou]
}
```

对象数组：

```
{
    "authors": [
        { "age": 26, "name": "Jack White"},
        { "age": 55, "name": "Tom Jones"},
        { "age": 39, "name": "Kitty Smith"}
    ]
}
```

存储格式：

```
{
    "authors.age":    [26, 55, 39],
    "authors.name":   [jack, white, tom, jones, kitty, smith]
}
```

#11. 索引Index入门

#11.1. 索引管理

#11.1.1. 为什么我们要手动创建索引

直接put数据 PUT index/_doc/1,es会自动生成索引，并建立动态映射dynamic mapping。

在生产上，我们需要自己手动建立索引和映射，为了更好地管理索引。就像数据库的建表语句一样。

#11.1.2. 索引管理

#11.1.2.1 创建索引

创建索引的语法

```
PUT /index
{
    "settings": { ... any settings ... },
    "mappings": {
       "properties" : {
            "field1" : { "type" : "text" }
        }
    },
    "aliases": {
        "default_index": {}
  }
}
```

举例：

```
PUT /my_index
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
```

```
        "field1":{
          "type": "text"
        },
        "field2":{
          "type": "text"
        }
      }
    },
    "aliases": {
      "default_index": {}
    }
  }
```

索引别名

插入数据

```
POST /my_index/_doc/1
{
    "field1":"java",
    "field2":"js"
}
```

查询数据 都可以查到

GET /my_index/_doc/1

GET /default_index/_doc/1

#11.1.2.2查询索引

GET /my_index/_mapping

GET /my_index/_setting

#11.1.2.3修改索引

修改副本数

```
PUT /my_index/_settings
{
    "index" : {
        "number_of_replicas" : 2
    }
}
```

#11.1.2.4删除索引

DELETE /my_index

DELETE /index_one,index_two

DELETE /index_*

DELETE /_all

为了安全起见，防止恶意删除索引，删除时必须指定索引名：

elasticsearch.yml

action.destructive_requires_name: true

#11.2． 定制分词器

#11.2.1 默认的分词器

standard

分词三个组件，character filter，tokenizer，token filter

standard tokenizer：以单词边界进行切分

standard token filter：什么都不做

lowercase token filter：将所有字母转换为小写

stop token filer（默认被禁用）：移除停用词，比如a the it等等

#11.2.2 修改分词器的设置

启用english停用词token filter

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "es_std": {
          "type": "standard",
          "stopwords": "_english_"
        }
      }
    }
  }
}
```

测试分词

```
GET /my_index/_analyze
{
  "analyzer": "standard",
  "text": "a dog is in the house"
}

GET /my_index/_analyze
{
  "analyzer": "es_std",
  "text":"a dog is in the house"
}
```

#11.2.3 定制化自己的分词器

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "char_filter": {
        "&_to_and": {
          "type": "mapping",
```

```
            "mappings": ["&=> and"]
          }
        },
        "filter": {
          "my_stopwords": {
            "type": "stop",
            "stopwords": ["the", "a"]
          }
        },
        "analyzer": {
          "my_analyzer": {
            "type": "custom",
            "char_filter": ["html_strip", "&_to_and"],
            "tokenizer": "standard",
            "filter": ["lowercase", "my_stopwords"]
          }
        }
      }
    }
  }
```

测试

```
GET /my_index/_analyze
{
  "analyzer": "my_analyzer",
  "text": "tom&jerry are a friend in the house, <a>, HAHA!!"
}
```

设置字段使用自定义分词器

```
PUT /my_index/_mapping/
{
  "properties": {
    "content": {
      "type": "text",
      "analyzer": "my_analyzer"
    }
  }
}
```

#11.3 type底层结构及弃用原因

#11.3.1type是什么

type，是一个index中用来区分类似的数据的，类似的数据，但是可能有不同的fields，而且有不同的属性来控制索引建立、分词器. field的value，在底层的lucene中建立索引的时候，全部是opaque bytes类型，不区分类型的。 lucene是没有type的概念的，在document中，实际上将type作为一个document的field来存储，即 *type*，*es*通过 type来进行type的过滤和筛选。

#11.3.2es中不同type存储机制

一个index中的多个type，实际上是放在一起存储的，因此一个index下，不能有多个type重名，而类型或者其他设置不同的，因为那样是无法处理的

```
{
   "goods": {
      "mappings": {
         "electronic_goods": {
            "properties": {
               "name": {
                  "type": "string",
               },
               "price": {
                  "type": "double"
               },
               "service_period": {
                  "type": "string"
                   }
                }
         },
         "fresh_goods": {
            "properties": {
               "name": {
                  "type": "string",
               },
               "price": {
                  "type": "double"
               },
               "eat_period": {
                    "type": "string"
               }
                }
         }
      }
   }
}
```

```
PUT /goods/electronic_goods/1
{
  "name": "小米空调",
  "price": 1999.0,
  "service_period": "one year"
}
```

```
PUT /goods/fresh_goods/1
{
  "name": "澳洲龙虾",
  "price": 199.0,
  "eat_period": "one week"
}
```

es文档在底层的存储是这样子的

```
{
   "goods": {
      "mappings": {
```

```
          "_type": {
            "type": "string",
            "index": "false"
          },
          "name": {
            "type": "string"
          }
          "price": {
            "type": "double"
          }
          "service_period": {
            "type": "string"
          },
          "eat_period": {
            "type": "string"
          }
        }
    }
 }
```

底层数据存储格式

```
{
  "_type": "electronic_goods",
  "name": "小米空调",
  "price": 1999.0,
  "service_period": "one year",
  "eat_period": ""
}
```

```
{
  "_type": "fresh_goods",
  "name": "澳洲龙虾",
  "price": 199.0,
  "service_period": "",
  "eat_period": "one week"
}

```

#11.3.3 type弃用

同一索引下，不同type的数据存储其他type的field 大量空值，造成资源浪费。

所以，不同类型数据，要放到不同的索引中。

es9中，将会彻底删除type。

#11.4.定制dynamic mapping

#11.4.1定制dynamic策略

true：遇到陌生字段，就进行dynamic mapping

false：新检测到的字段将被忽略。这些字段将不会被索引，因此将无法搜索，但仍将出现在返回点击的源字段中。这些字段不会添加到映射中，必须显式添加新字段。

strict：遇到陌生字段，就报错

创建mapping

```
PUT /my_index
{
    "mappings": {
      "dynamic": "strict",
       "properties": {
        "title": {
          "type": "text"
        },
        "address": {
          "type": "object",
          "dynamic": "true"
        }
        }
    }
}
```

插入数据

```
PUT /my_index/_doc/1
{
  "title": "my article",
  "content": "this is my article",
  "address": {
    "province": "guangdong",
    "city": "guangzhou"
  }
}
```

报错

```
{
  "error": {
    "root_cause": [
      {
        "type": "strict_dynamic_mapping_exception",
        "reason": "mapping set to strict, dynamic introduction of [content] within [_doc] is not allowed"
      }
    ],
    "type": "strict_dynamic_mapping_exception",
    "reason": "mapping set to strict, dynamic introduction of [content] within [_doc] is not allowed"
  },
  "status": 400
}
```

#11.4.2自定义 dynamic mapping策略

es会根据传入的值，推断类型。

JSON datatype	Elasticsearch datatype
`null`	No field is added.
`true` or `false`	`boolean` field
floating point number	`float` field
integer	`long` field
object	`object` field
array	Depends on the first non-`null` value in the array.
string	Either a `date` field (if the value passes date detection), a `double` or `long` field (if the value passes numeric detection) or a `text` field, with a `keyword` sub-field.

#date_detection 日期探测

默认会按照一定格式识别date，比如yyyy-MM-dd。但是如果某个field先过来一个2017-01-01的值，就会被自动dynamic mapping成date，后面如果再来一个"hello world"之类的值，就会报错。可以手动关闭某个type的date_detection，如果有需要，自己手动指定某个field为date类型。

```
PUT /my_index
{
    "mappings": {
      "date_detection": false,
       "properties": {
        "title": {
          "type": "text"
        },
        "address": {
          "type": "object",
          "dynamic": "true"
        }
        }
    }
}
```

测试

```
PUT /my_index/_doc/1
{
  "title": "my article",
  "content": "this is my article",
  "address": {
    "province": "guangdong",
    "city": "guangzhou"
  },
  "post_date":"2019-09-10"
}
```

查看映射

```
GET /my_index/_mapping
```

#自定义日期格式

```
PUT my_index
{
  "mappings": {
    "dynamic_date_formats": ["MM/dd/yyyy"]
  }
}
```

插入数据

```
PUT my_index/_doc/1
{
  "create_date": "09/25/2019"
}
```

#numeric_detection 数字探测

虽然json支持本机浮点和整数数据类型，但某些应用程序或语言有时可能将数字呈现为字符串。通常正确的解决方案是显式地映射这些字段，但是可以启用数字检测（默认情况下禁用）来自动完成这些操作。

```
PUT my_index
{
  "mappings": {
    "numeric_detection": true
  }
}
```

```
PUT my_index/_doc/1
{
  "my_float":   "1.0",
  "my_integer": "1"
}
```

#11.4.3定制自己的dynamic mapping template

```
 PUT /my_index
 {
     "mappings": {
             "dynamic_templates": [
                 {
                   "en": {
                       "match":              "*_en",
                       "match_mapping_type": "string",
                       "mapping": {
                           "type":           "text",
                           "analyzer":       "english"
                       }
                 }
             }
         ]
     }
```

```
}
```

插入数据

```
PUT /my_index/_doc/1
{
  "title": "this is my first article"
}

PUT /my_index/_doc/2
{
  "title_en": "this is my first article"
}
```

搜索

```
GET my_index/_search?q=first
GET my_index/_search?q=is
```

title没有匹配到任何的dynamic模板，默认就是standard分词器，不会过滤停用词，is会进入倒排索引，用is来搜索是可以搜索到的

title_en匹配到了dynamic模板，就是english分词器，会过滤停用词，is这种停用词就会被过滤掉，用is来搜索就搜索不到了

#模板写法

```
PUT my_index
{
  "mappings": {
    "dynamic_templates": [
      {
        "integers": {
          "match_mapping_type": "long",
          "mapping": {
            "type": "integer"
          }
        }
      },
      {
        "strings": {
          "match_mapping_type": "string",
          "mapping": {
            "type": "text",
            "fields": {
              "raw": {
                "type":  "keyword",
                "ignore_above": 256
              }
            }
          }
        }
      }
    ]
  }
}
```

模板参数

```
"match":   "long_*",
"unmatch": "*_text",
"match_mapping_type": "string",
"path_match":   "name.*",
"path_unmatch": "*.middle",
```

```
"match_pattern": "regex",
"match": "^profit_\d+$"
```

#场景

1结构化搜索

默认情况下，elasticsearch将字符串字段映射为带有子关键字字段的文本字段。但是，如果只对结构化内容进行索引，而对全文搜索不感兴趣，则可以仅将“字段”映射为“关键字”。请注意，这意味着为了搜索这些字段，必须搜索索引所用的完全相同的值。

```
      {
          "strings_as_keywords": {
            "match_mapping_type": "string",
            "mapping": {
              "type": "keyword"
            }
          }
        }
```

2仅搜索

与前面的示例相反，如果您只关心字符串字段的全文搜索，并且不打算对字符串字段运行聚合、排序或精确搜索，您可以告诉弹性搜索将其仅映射为文本字段（这是5之前的默认行为）

```
      {
          "strings_as_text": {
            "match_mapping_type": "string",
            "mapping": {
              "type": "text"
            }
          }
        }
```

3norms 不关心评分

norms是指标时间的评分因素。如果您不关心评分，例如，如果您从不按评分对文档进行排序，则可以在索引中禁用这些评分因子的存储并节省一些空间。

```
{
         "strings_as_keywords": {
           "match_mapping_type": "string",
           "mapping": {
             "type": "text",
             "norms": false,
             "fields": {
               "keyword": {
                 "type": "keyword",
                 "ignore_above": 256
               }
             }
           }
```

```
14          }
15        }
```

#11.5． 零停机重建索引

#11.5.1零停机重建索引

场景：

一个field的设置是不能被修改的，如果要修改一个Field，那么应该重新按照新的mapping，建立一个index，然后将数据批量查询出来，重新用bulk api写入index中。

批量查询的时候，建议采用scroll api，并且采用多线程并发的方式来reindex数据，每次scoll就查询指定日期的一段数据，交给一个线程即可。

(1)一开始，依靠dynamic mapping，插入数据，但是不小心有些数据是2019-09-10这种日期格式的，所以title这种field被自动映射为了date类型，实际上它应该是string类型的

```
PUT /my_index/_doc/1
{
  "title": "2019-09-10"
}

PUT /my_index/_doc/2
{
  "title": "2019-09-11"
}
```

（2）当后期向索引中加入string类型的title值的时候，就会报错

```
PUT /my_index/_doc/3
{
  "title": "my first article"
}
```

报错

```
{
  "error": {
    "root_cause": [
      {
        "type": "mapper_parsing_exception",
        "reason": "failed to parse [title]"
      }
    ],
    "type": "mapper_parsing_exception",
    "reason": "failed to parse [title]",
    "caused_by": {
      "type": "illegal_argument_exception",
      "reason": "Invalid format: \"my first article\""
    }
  },
  "status": 400
}
```

（3）如果此时想修改title的类型，是不可能的

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

报错

```
{
  "error": {
    "root_cause": [
      {
        "type": "illegal_argument_exception",
        "reason": "mapper [title] of different type, current_type [date], merged_type [text]"
      }
    ],
    "type": "illegal_argument_exception",
    "reason": "mapper [title] of different type, current_type [date], merged_type [text]"
  },
  "status": 400
}
```

（4）此时，唯一的办法，就是进行reindex，也就是说，重新建立一个索引，将旧索引的数据查询出来，再导入新索引。

（5）如果说旧索引的名字，是old_index，新索引的名字是new_index，终端java应用，已经在使用old_index在操作了，难道还要去停止java应用，修改使用的index为new_index，才重新启动java应用吗？这个过程中，就会导致java应用停机，可用性降低。

（6）所以说，给java应用一个别名，这个别名是指向旧索引的，java应用先用着，java应用先用prod_index alias来操作，此时实际指向的是旧的my_index

```
PUT /my_index/_alias/prod_index
```

1

（7）新建一个index，调整其title的类型为string

```
PUT /my_index_new
{
  "mappings": {
    "properties": {
        "title": {
         "type": "text"
        }
    }
  }
}
```

（8）使用scroll api将数据批量查询出来

```
GET /my_index/_search?scroll=1m
{
    "query": {
        "match_all": {}
    },
    "size":  1
}
```

返回

```
 {
   "_scroll_id":
 "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAADpAFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAA6QRY0b25z
 VFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAAOkIWNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAADpDFjRvb
 nNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAA6RBY0b25zVFlWWlRqR3ZJajlfc3BXejJ3",
   "took": 1,
   "timed_out": false,
   "_shards": {
     "total": 5,
     "successful": 5,
     "failed": 0
   },
   "hits": {
     "total": 3,
     "max_score": null,
     "hits": [
       {
         "_index": "my_index",
         "_type": "my_type",
         "_id": "1",
         "_score": null,
         "_source": {
           "title": "2019-01-02"
         },
         "sort": [
           0
         ]
       }
     ]
   }
 }
```

（9）采用bulk api将scoll查出来的一批数据，批量写入新索引

```
POST /_bulk
{ "index":  { "_index": "my_index_new", "_id": "1" }}
{ "title":    "2019-09-10" }
```

（10）反复循环8~9，查询一批又一批的数据出来，采取bulk api将每一批数据批量写入新索引

（11）将prod_index alias切换到my_index_new上去，java应用会直接通过index别名使用新的索引中的数据，java应用程序不需要停机，零提交，高可用

```
POST /_aliases
{
    "actions": [
        { "remove": { "index": "my_index", "alias": "prod_index" }},
        { "add":    { "index": "my_index_new", "alias": "prod_index" }}
    ]
}
```

（12）直接通过prod_index别名来查询，是否ok

```
GET /prod_index/_search
```

#11.5.2 生产实践：基于alias对client透明切换index

```
PUT /my_index_v1/_alias/my_index
```

client对my_index进行操作

reindex操作，完成之后，切换v1到v2

```
POST /_aliases
{
    "actions": [
        { "remove": { "index": "my_index_v1", "alias": "my_index" }},
        { "add":    { "index": "my_index_v2", "alias": "my_index" }}
    ]
}
```

#12． 中文分词器 IK分词器

#12.1． Ik分词器安装使用

#12.1.1 中文分词器

standard 分词器，仅适用于英文。

```
GET /_analyze
{
  "analyzer": "standard",
  "text": "中华人民共和国人民大会堂"
}
```

我们想要的效果是什么：中华人民共和国，人民大会堂

IK分词器就是目前最流行的es中文分词器

#12.1.2 安装

官网： https://github.com/medcl/elasticsearch-analysis-ik

下载地址： https://github.com/medcl/elasticsearch-analysis-ik/releases

根据es版本下载相应版本包。

解压到 es/plugins/ik中。

重启es

#12.1.3 ik分词器基础知识

ik_max_word: 会将文本做最细粒度的拆分，比如会将“中华人民共和国人民大会堂”拆分为“中华人民共和国，中华人民，中华，华人，人民共和国，人民大会堂，人民大会，大会堂”，会穷尽各种可能的组合；

ik_smart: 会做最粗粒度的拆分，比如会将“中华人民共和国人民大会堂”拆分为“中华人民共和国，人民大会堂”。

#12.1.4 ik分词器的使用

存储时，使用ik_max_word，搜索时，使用ik_smart

```
PUT /my_index
{
  "mappings": {
      "properties": {
        "text": {
          "type": "text",
          "analyzer": "ik_max_word",
          "search_analyzer": "ik_smart"
        }
      }
  }
}
```

搜索

```
GET /my_index/_search?q=中华人民共和国人民大会堂
```

#12.2. ik配置文件

#12.2.1 ik配置文件

ik配置文件地址：es/plugins/ik/config目录

IKAnalyzer.cfg.xml：用来配置自定义词库

main.dic：ik原生内置的中文词库，总共有27万多条，只要是这些单词，都会被分在一起

preposition.dic: 介词

quantifier.dic：放了一些单位相关的词，量词

suffix.dic：放了一些后缀

surname.dic：中国的姓氏

stopword.dic：英文停用词

ik原生最重要的两个配置文件

main.dic：包含了原生的中文词语，会按照这个里面的词语去分词

stopword.dic：包含了英文的停用词

停用词，stopword

a the and at but

一般，像停用词，会在分词的时候，直接被干掉，不会建立在倒排索引中

#12.2.1 自定义词库

（1）自己建立词库：每年都会涌现一些特殊的流行词，网红，蓝瘦香菇，喊麦，鬼畜，一般不会在ik的原生词典里

自己补充自己的最新的词语，到ik的词库里面

IKAnalyzer.cfg.xml：ext_dict，创建mydict.dic。

补充自己的词语，然后需要重启es，才能生效

（2）自己建立停用词库：比如了，的，啥，么，我们可能并不想去建立索引，让人家搜索

custom/ext_stopword.dic，已经有了常用的中文停用词，可以补充自己的停用词，然后重启es

#12.3． 使用mysql热更新 词库

#12.3.1热更新

每次都是在es的扩展词典中，手动添加新词语，很坑

（1）每次添加完，都要重启es才能生效，非常麻烦

（2）es是分布式的，可能有数百个节点，你不能每次都一个一个节点上面去修改

es不停机，直接我们在外部某个地方添加新的词语，es中立即热加载到这些新词语

热更新的方案

（1）基于ik分词器原生支持的热更新方案，部署一个web服务器，提供一个http接口，通过modified和tag两个http响应头，来提供词语的热更新

（2）修改ik分词器源码，然后手动支持从mysql中每隔一定时间，自动加载新的词库

用第二种方案，第一种，ik git社区官方都不建议采用，觉得不太稳定

#12.3.2步骤

1、下载源码

https://github.com/medcl/elasticsearch-analysis-ik/releases

ik分词器，是个标准的java maven工程，直接导入eclipse就可以看到源码

2、修改源

org.wltea.analyzer.dic.Dictionary类，160行Dictionary单例类的初始化方法，在这里需要创建一个我们自定义的线程，并且启动它

org.wltea.analyzer.dic.HotDictReloadThread类：就是死循环，不断调用Dictionary.getSingleton().reLoadMainDict()，去重新加载词典

Dictionary类，399行：this.loadMySQLExtDict(); 加载mymsql字典。

Dictionary类，609行：this.loadMySQLStopwordDict();加载mysql停用词

config下jdbc-reload.properties。mysql配置文件

3、mvn package打包代码

target\releases\elasticsearch-analysis-ik-7.3.0.zip

4、解压缩ik压缩包

将mysql驱动jar，放入ik的目录下

5、修改jdbc相关配置

6、重启es

观察日志，日志中就会显示我们打印的那些东西，比如加载了什么配置，加载了什么词语，什么停用词

7、在mysql中添加词库与停用词

8、分词实验，验证热更新生效

```
GET /_analyze
{
  "analyzer": "ik_smart",
  "text": "传智播客"
}
```

#13． java api 实现索引管理

代码

```
package com.ydlclass.es;

import org.elasticsearch.action.ActionListener;
import org.elasticsearch.action.admin.indices.alias.Alias;
import org.elasticsearch.action.admin.indices.close.CloseIndexRequest;
import org.elasticsearch.action.admin.indices.delete.DeleteIndexRequest;
import org.elasticsearch.action.admin.indices.open.OpenIndexRequest;
import org.elasticsearch.action.admin.indices.open.OpenIndexResponse;
import org.elasticsearch.action.support.ActiveShardCount;
import org.elasticsearch.action.support.master.AcknowledgedResponse;
import org.elasticsearch.client.IndicesClient;
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.settings.Settings;
import org.elasticsearch.common.unit.TimeValue;
import org.elasticsearch.common.xcontent.XContentType;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import java.io.IOException;

/**

- @author Administrator

```

```
- @version 1.0
  **/
  @SpringBootTest
  @RunWith(SpringRunner.class)
  public class TestIndex {

  @Autowired
  RestHighLevelClient client;

//    @Autowired
//    RestClient restClient;

```
//创建索引
@Test
public void testCreateIndex() throws IOException {
    //创建索引对象
    CreateIndexRequest createIndexRequest = new CreateIndexRequest("ydlclass_book");
    //设置参数
    createIndexRequest.settings(Settings.builder().put("number_of_shards", "1").put("number_of_replicas", "0"));
    //指定映射1
    createIndexRequest.mapping(" {\n" +
            " \t\"properties\": {\n" +
            "            \"name\":{\n" +
            "              \"type\":\"keyword\"\n" +
            "           },\n" +
            "           \"description\": {\n" +
            "              \"type\": \"text\"\n" +
            "           },\n" +
            "            \"price\":{\n" +
            "              \"type\":\"long\"\n" +
            "           },\n" +
            "           \"pic\":{\n" +
            "              \"type\":\"text\",\n" +
            "              \"index\":false\n" +
            "           }\n" +
            " \t}\n" +
            "}", XContentType.JSON);

    //指定映射2
```

```
//        Map<String, Object> message = new HashMap<>();
//        message.put("type", "text");
//        Map<String, Object> properties = new HashMap<>();
//        properties.put("message", message);
//        Map<String, Object> mapping = new HashMap<>();
//        mapping.put("properties", properties);
//        createIndexRequest.mapping(mapping);

```
    //指定映射3
```

```
//        XContentBuilder builder = XContentFactory.jsonBuilder();
//        builder.startObject();
```

```
//        {
//            builder.startObject("properties");
//            {
//                builder.startObject("message");
//                {
//                    builder.field("type", "text");
//                }
//                builder.endObject();
//            }
//            builder.endObject();
//        }
//        builder.endObject();
//        createIndexRequest.mapping(builder);

```
    //设置别名
    createIndexRequest.alias(new Alias("ydlclass_index_new"));

    // 额外参数
    //设置超时时间
    createIndexRequest.setTimeout(TimeValue.timeValueMinutes(2));
    //设置主节点超时时间
    createIndexRequest.setMasterTimeout(TimeValue.timeValueMinutes(1));
    //在创建索引API返回响应之前等待的活动分片副本的数量，以int形式表示
    createIndexRequest.waitForActiveShards(ActiveShardCount.from(2));
    createIndexRequest.waitForActiveShards(ActiveShardCount.DEFAULT);

    //操作索引的客户端
    IndicesClient indices = client.indices();
    //执行创建索引库
    CreateIndexResponse createIndexResponse = indices.create(createIndexRequest,
RequestOptions.DEFAULT);

    //得到响应（全部）
    boolean acknowledged = createIndexResponse.isAcknowledged();
    //得到响应 指示是否在超时前为索引中的每个分片启动了所需数量的碎片副本
    boolean shardsAcknowledged = createIndexResponse.isShardsAcknowledged();

    System.out.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!" + acknowledged);
    System.out.println(shardsAcknowledged);

}

//异步新增索引
@Test
public void testCreateIndexAsync() throws IOException {
    //创建索引对象
    CreateIndexRequest createIndexRequest = new
CreateIndexRequest("ydlclass_book2");
    //设置参数
    createIndexRequest.settings(Settings.builder().put("number_of_shards",
"1").put("number_of_replicas", "0"));
    //指定映射1
    createIndexRequest.mapping(" {\n" +
            " \t\"properties\": {\n" +
            "   \"name\":{\n" +
            "   \"type\":\"keyword\"\n" +
            "   },\n" +
```
```

```
                "            \"description\": {\n" +
                "               \"type\": \"text\"\n" +
                "            },\n" +
                "             \"price\":{\n" +
                "               \"type\":\"long\"\n" +
                "            },\n" +
                "            \"pic\":{\n" +
                "               \"type\":\"text\",\n" +
                "               \"index\":false\n" +
                "            }\n" +
                " \t}\n" +
                "}", XContentType.JSON);

        //监听方法
        ActionListener<CreateIndexResponse> listener =
                new ActionListener<CreateIndexResponse>() {

                    @Override
                    public void onResponse(CreateIndexResponse createIndexResponse)
    {
                        System.out.println("!!!!!!!!创建索引成功");
                        System.out.println(createIndexResponse.toString());
                    }

                    @Override
                    public void onFailure(Exception e) {
                        System.out.println("!!!!!!!!创建索引失败");
                        e.printStackTrace();
                    }
                };

        //操作索引的客户端
        IndicesClient indices = client.indices();
        //执行创建索引库
        indices.createAsync(createIndexRequest, RequestOptions.DEFAULT, listener);

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    ```

    ```
    }
    ```

    ```
    //删除索引库
    @Test
    public void testDeleteIndex() throws IOException {
        //删除索引对象
        DeleteIndexRequest deleteIndexRequest = new
    DeleteIndexRequest("ydlclass_book2");
        //操作索引的客户端
        IndicesClient indices = client.indices();
        //执行删除索引
```

```
    AcknowledgedResponse delete = indices.delete(deleteIndexRequest, RequestOptions.DEFAULT);
    //得到响应
    boolean acknowledged = delete.isAcknowledged();
    System.out.println(acknowledged);

}

//异步删除索引库
@Test
public void testDeleteIndexAsync() throws IOException {
    //删除索引对象
    DeleteIndexRequest deleteIndexRequest = new DeleteIndexRequest("ydlclass_book2");
    //操作索引的客户端
    IndicesClient indices = client.indices();

    //监听方法
    ActionListener<AcknowledgedResponse> listener =
            new ActionListener<AcknowledgedResponse>() {
                @Override
                public void onResponse(AcknowledgedResponse deleteIndexResponse) {
                    System.out.println("!!!!!!!!删除索引成功");
                    System.out.println(deleteIndexResponse.toString());
                }

                @Override
                public void onFailure(Exception e) {
                    System.out.println("!!!!!!!!删除索引失败");
                    e.printStackTrace();
                }
            };
    //执行删除索引
    indices.deleteAsync(deleteIndexRequest, RequestOptions.DEFAULT, listener);

    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

}

// Indices Exists API
@Test
public void testExistIndex() throws IOException {
    GetIndexRequest request = new GetIndexRequest("ydlclass_book");
    request.local(false);//从主节点返回本地信息或检索状态
    request.humanReadable(true);//以适合人类的格式返回结果
    request.includeDefaults(false);//是否返回每个索引的所有默认设置

    boolean exists = client.indices().exists(request, RequestOptions.DEFAULT);
    System.out.println(exists);
}
```

```
```

```
253    // Indices Open API
254    @Test
255    public void testOpenIndex() throws IOException {
256        OpenIndexRequest request = new OpenIndexRequest("ydlclass_book");
257
258        OpenIndexResponse openIndexResponse = client.indices().open(request,
    RequestOptions.DEFAULT);
259        boolean acknowledged = openIndexResponse.isAcknowledged();
260        System.out.println("!!!!!!!!!!"+acknowledged);
261    }
262
263    // Indices Close API
264    @Test
265    public void testCloseIndex() throws IOException {
266        CloseIndexRequest request = new CloseIndexRequest("index");
267        AcknowledgedResponse closeIndexResponse = client.indices().close(request,
    RequestOptions.DEFAULT);
268        boolean acknowledged = closeIndexResponse.isAcknowledged();
269        System.out.println("!!!!!!!!!!"+acknowledged);
270
271    }
272    }
```

#14. search搜索入门

#14.1. 搜索语法入门

#14.1.1query string search

无条件搜索所有

```
1    GET /book/_search
```

```
1    {
2      "took" : 969,
3      "timed_out" : false,
4      "_shards" : {
5        "total" : 1,
6        "successful" : 1,
7        "skipped" : 0,
8        "failed" : 0
9      },
10     "hits" : {
11       "total" : {
12         "value" : 3,
13         "relation" : "eq"
14       },
15       "max_score" : 1.0,
16       "hits" : [
17         {
18           "_index" : "book",
19           "_type" : "_doc",
20           "_id" : "1",
21           "_score" : 1.0,
```

```
        "_source" : {
          "name" : "Bootstrap开发",
          "description" : "Bootstrap是由Twitter推出的一个前台页面开发css框架，是一个非常流行的开发框架，此框架集成了多种页面效果。此开发框架包含了大量的CSS、JS程序代码，可以帮助开发者（尤其是不擅长css页面开发的程序人员）轻松的实现一个css，不受浏览器限制的精美界面css效果。",
          "studymodel" : "201002",
          "price" : 38.6,
          "timestamp" : "2019-08-25 19:11:35",
          "pic" : "group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
          "tags" : [
            "bootstrap",
            "dev"
          ]
        }
      },
      {
        "_index" : "book",
        "_type" : "_doc",
        "_id" : "2",
        "_score" : 1.0,
        "_source" : {
          "name" : "java编程思想",
          "description" : "java语言是世界第一编程语言，在软件开发领域使用人数最多。",
          "studymodel" : "201001",
          "price" : 68.6,
          "timestamp" : "2019-08-25 19:11:35",
          "pic" : "group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
          "tags" : [
            "java",
            "dev"
          ]
        }
      },
      {
        "_index" : "book",
        "_type" : "_doc",
        "_id" : "3",
        "_score" : 1.0,
        "_source" : {
          "name" : "spring开发基础",
          "description" : "spring 在java领域非常流行，java程序员都在用。",
          "studymodel" : "201001",
          "price" : 88.6,
          "timestamp" : "2019-08-24 19:11:35",
          "pic" : "group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
          "tags" : [
            "spring",
            "java"
          ]
        }
      }
    ]
  }
}
```


解释

took：耗费了几毫秒

timed_out：是否超时，这里是没有

_shards：到几个分片搜索，成功几个，跳过几个，失败几个。

hits.total：查询结果的数量，3个document

hits.max_score：score的含义，就是document对于一个search的相关度的匹配分数，越相关，就越匹配，分数也高

hits.hits：包含了匹配搜索的document的所有详细数据

#14.1.2传参

与http请求传参类似

```
GET /book/_search?q=name:java&sort=price:desc
```

类比sql: select * from book where name like ’ %java%’ order by price desc

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : null,
    "hits" : [
      {
        "_index" : "book",
        "_type" : "_doc",
        "_id" : "2",
        "_score" : null,
        "_source" : {
          "name" : "java编程思想",
          "description" : "java语言是世界第一编程语言，在软件开发领域使用人数最多。",
          "studymodel" : "201001",
          "price" : 68.6,
          "timestamp" : "2019-08-25 19:11:35",
          "pic" : "group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
          "tags" : [
            "java",
            "dev"
          ]
        },
        "sort" : [
          68.6
        ]
      }
    ]
  }
}
```

#14.1.3图解timeout

GET /book/_search?timeout=10ms

全局设置：配置文件中设置 search.default_search_timeout：100ms。默认不超时。

#14.2． multi-index 多索引搜索

#14.2.1multi-index搜索模式

告诉你如何一次性搜索多个index和多个type下的数据

```
/_search：所有索引下的所有数据都搜索出来
/index1/_search：指定一个index，搜索其下所有的数据
/index1,index2/_search：同时搜索两个index下的数据
/index*/_search：按照通配符去匹配多个索引
```

应用场景：生产环境log索引可以按照日期分开。

log_to_es_20190910

log_to_es_20190911

log_to_es_20180910

#14.2.2初步图解一下简单的搜索原理

搜索原理初步图解

#14.3． 分页搜索

#14.3.1分页搜索的语法

sql: select * from book limit 1,5

size，from

GET /book/_search?size=10

GET /book/_search?size=10&from=0

GET /book/_search?size=10&from=20

GET /book_search?from=0&size=3

#14.3.2deep paging

#什么是deep paging

根据相关度评分倒排序，所以分页过深，协调节点会将大量数据聚合分析。

#deep paging 性能问题

1消耗网络带宽，因为所搜过深的话，各 shard 要把数据传递给 coordinate node，这个过程是有大量数据传递的，消耗网络。

2消耗内存，各 shard 要把数据传送给 coordinate node，这个传递回来的数据，是被 coordinate node 保存在内存中的，这样会大量消耗内存。

3消耗cup，coordinate node 要把传回来的数据进行排序，这个排序过程很消耗cpu。 所以：鉴于deep paging的性能问题，所有应尽量减少使用。

#14.4． query string基础语法

#14.4.1query string基础语法

GET /book/_search?q=name:java

GET /book/_search?q=+name:java

GET /book/_search?q=-name:java

一个是掌握q=field:search content的语法，还有一个是掌握+和-的含义

#14.4.2、 _all metadata的原理和作用

```
GET /book/_search?q=java
```

直接可以搜索所有的field，任意一个field包含指定的关键字就可以搜索出来。我们在进行中搜索的时候，难道是对document中的每一个field都进行一次搜索吗？不是的。

es中 *all*元数据。建立索引的时候，插入一条*docunment*，*es*会将所有的*field*值经行全量分词，把这些分词，放到 all field中。在搜索的时候，没有指定field，就在_all搜索。

举例

```
{
    name:jack
    email:123@qq.com
    address:beijing
}
```

_all : jack, 123@qq.com ,beijing

#14.5． query DSL入门

#14.5.1 DSL

query string 后边的参数原来越多，搜索条件越来越复杂，不能满足需求。

GET /book/_search?q=name:java&size=10&from=0&sort=price:desc

DSL:Domain Specified Language，特定领域的语言

es特有的搜索语言，可在请求体中携带搜索条件，功能强大。

查询全部 GET /book/_search

```
GET /book/_search
{
  "query": { "match_all": {} }
}
```

排序 GET /book/_search?sort=price:desc

```
 GET /book/_search
 {
     "query" : {
         "match" : {
             "name" : " java"
         }
     },
     "sort": [
         { "price": "desc" }
     ]
 }
```

分页查询 GET /book/_search?size=10&from=0

```
GET  /book/_search
{
  "query": { "match_all": {} },
  "from": 0,
  "size": 1
}
```

指定返回字段 GET /book/ _search? _source=name,studymodel

```
GET /book/_search
{
  "query": { "match_all": {} },
  "_source": ["name", "studymodel"]
}
```

通过组合以上各种类型查询，实现复杂查询。

#14.5.2． Query DSL语法

```
{
    QUERY_NAME: {
        ARGUMENT: VALUE,
        ARGUMENT: VALUE,...
    }
}
```

```
{
    QUERY_NAME: {
        FIELD_NAME: {
            ARGUMENT: VALUE,
            ARGUMENT: VALUE,...
        }
    }
}
```

```
GET /test_index/_search
{
  "query": {
    "match": {
      "test_field": "test"
    }
  }
}
```

#14.5.3 组合多个搜索条件

搜索需求：title必须包含elasticsearch，content可以包含elasticsearch也可以不包含，author_id必须不为111

sql where and or !=

初始数据：

```
POST /website/_doc/1
{
          "title": "my hadoop article",
          "content": "hadoop is very bad",
          "author_id": 111
}

POST /website/_doc/2
{
          "title": "my elasticsearch  article",
          "content": "es is very bad",
          "author_id": 112
}
POST /website/_doc/3
{
          "title": "my elasticsearch article",
          "content": "es is very goods",
          "author_id": 111
}
```

搜索：

```
GET /website/_doc/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "title": "elasticsearch"
          }
        }
      ],
      "should": [
        {
          "match": {
            "content": "elasticsearch"
          }
        }
      ],
```

```
      "must_not": [
        {
          "match": {
            "author_id": 111
          }
        }
      ]
    }
  }
}
```

返回：

```
{
  "took" : 488,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 0.47000363,
    "hits" : [
      {
        "_index" : "website",
        "_type" : "_doc",
        "_id" : "2",
        "_score" : 0.47000363,
        "_source" : {
          "title" : "my elasticsearch  article",
          "content" : "es is very bad",
          "author_id" : 112
        }
      }
    ]
  }
}
```

更复杂的搜索需求：

select * from test_index where name='tom' or (hired =true and (personality ='good' and rude != true))

```
GET /test_index/_search
{
    "query": {
            "bool": {
                "must": { "match":{ "name": "tom" }},
                "should": [
                    { "match":{ "hired": true }},
                    { "bool": {
                        "must":{ "match": { "personality": "good" }},
                        "must_not": { "match": { "rude": true }}
```

```
                    }}
                ],
                "minimum_should_match": 1
            }
    }
}
```

#14.6． full-text search 全文检索

#14.6.1 全文检索

重新创建book索引

```
PUT /book/
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 0
  },
  "mappings": {
    "properties": {
      "name":{
        "type": "text",
        "analyzer": "ik_max_word",
        "search_analyzer": "ik_smart"
      },
      "description":{
        "type": "text",
        "analyzer": "ik_max_word",
        "search_analyzer": "ik_smart"
      },
      "studymodel":{
        "type": "keyword"
      },
      "price":{
        "type": "double"
      },
      "timestamp": {
         "type": "date",
         "format": "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd||epoch_millis"
      },
      "pic":{
        "type":"text",
        "index":false
      }
    }
  }
}
```

插入数据

```
PUT /book/_doc/1
{
"name": "Bootstrap开发",
```

```
"description": "Bootstrap是由Twitter推出的一个前台页面开发css框架，是一个非常流行的开发框架，此框架集成了多种页面效果。此开发框架包含了大量的CSS、JS程序代码，可以帮助开发者（尤其是不擅长css页面开发的程序人员）轻松的实现一个css，不受浏览器限制的精美界面css效果。",
"studymodel": "201002",
"price":38.6,
"timestamp":"2019-08-25 19:11:35",
"pic":"group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
"tags": [ "bootstrap", "dev"]
}

PUT /book/_doc/2
{
"name": "java编程思想",
"description": "java语言是世界第一编程语言，在软件开发领域使用人数最多。",
"studymodel": "201001",
"price":68.6,
"timestamp":"2019-08-25 19:11:35",
"pic":"group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
"tags": [ "java", "dev"]
}

PUT /book/_doc/3
{
"name": "spring开发基础",
"description": "spring 在java领域非常流行，java程序员都在用。",
"studymodel": "201001",
"price":88.6,
"timestamp":"2019-08-24 19:11:35",
"pic":"group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
"tags": [ "spring", "java"]
}
```

搜索

```
GET  /book/_search
{
    "query" : {
        "match" : {
            "description" : "java程序员"
        }
    }
}
```

#14.6.2 _score初探

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
```

```
        "relation" : "eq"
      },
      "max_score" : 2.137549,
      "hits" : [
        {
          "_index" : "book",
          "_type" : "_doc",
          "_id" : "3",
          "_score" : 2.137549,
          "_source" : {
            "name" : "spring开发基础",
            "description" : "spring 在java领域非常流行，java程序员都在用。",
            "studymodel" : "201001",
            "price" : 88.6,
            "timestamp" : "2019-08-24 19:11:35",
            "pic" : "group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
            "tags" : [
              "spring",
              "java"
            ]
          }
        },
        {
          "_index" : "book",
          "_type" : "_doc",
          "_id" : "2",
          "_score" : 0.57961315,
          "_source" : {
            "name" : "java编程思想",
            "description" : "java语言是世界第一编程语言，在软件开发领域使用人数最多。",
            "studymodel" : "201001",
            "price" : 68.6,
            "timestamp" : "2019-08-25 19:11:35",
            "pic" : "group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
            "tags" : [
              "java",
              "dev"
            ]
          }
        }
      ]
    }
  }
```


结果分析

1、建立索引时, description字段 term倒排索引

java 2,3

程序员 3

2、搜索时，直接找description中含有java的文档 2,3，并且3号文档含有两个java字段，一个程序员，所以得分高，排在前面。2号文档含有一个java，排在后面。

#14.7． DSL 语法练习

#14.7.1 match_all

```
GET /book/_search
{
    "query": {
        "match_all": {}
    }
}
```

#14.7.2 match

```
GET /book/_search
{
    "query": {
        "match": {
            "description": "java程序员"
        }
    }
}
```

#14.7.3 multi_match

```
GET /book/_search
{
  "query": {
    "multi_match": {
      "query": "java程序员",
      "fields": ["name", "description"]
    }
  }
}
```

#14.7.4、range query 范围查询

```
 GET /book/_search
 {
   "query": {
     "range": {
       "price": {
         "gte": 80,
         "lte": 90
       }
     }
   }
 }
```

#14.7.5、term query

字段为keyword时，存储和搜索都不分词

```
GET /book/_search
{
  "query": {
    "term": {
      "description": "java程序员"
    }
  }
}
```

#14.7.6、terms query

```
GET /book/_search
{
    "query": { "terms": { "tag": [ "search", "full_text", "nosql" ] }}
}
```

#14.7.7、exist query 查询有某些字段值的文档

```
GET /_search
{
    "query": {
        "exists": {
            "field": "join_date"
        }
    }
}
```

#14.7. 8、Fuzzy query

返回包含与搜索词类似的词的文档，该词由Levenshtein编辑距离度量。

包括以下几种情况：

- 更改角色（box→fox）
- 删除字符（aple→apple）
- 插入字符（sick→sic）
- 调换两个相邻字符（ACT→CAT）

```
GET /book/_search
{
    "query": {
        "fuzzy": {
            "description": {
                "value": "jave"
            }
        }
    }
}
```

#14.7.9、IDs

```
GET /book/_search
{
    "query": {
        "ids" : {
            "values" : ["1", "4", "100"]
        }
    }
}
```

#14.7.10、prefix 前缀查询

```
GET /book/_search
{
    "query": {
        "prefix": {
            "description": {
                "value": "spring"
            }
        }
    }
}
```

#14.7.11、regexp query 正则查询

```
GET /book/_search
{
    "query": {
        "regexp": {
            "description": {
                "value": "j.*a",
                "flags" : "ALL",
                "max_determinized_states": 10000,
                "rewrite": "constant_score"
            }
        }
    }
}
```

#14.8． Filter

#14.8.1 filter与query示例

需求：用户查询description中有"java程序员"，并且价格大于80小于90的数据。

```
GET /book/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "description": "java程序员"
          }
```

```
          },
          {
            "range": {
              "price": {
                "gte": 80,
                "lte": 90
              }
            }
          }
        ]
      }
    }
}
```

使用filter:

```
GET /book/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "description": "java程序员"
          }
        }
      ],
      "filter": {
        "range": {
          "price": {
            "gte": 80,
             "lte": 90
          }
        }
      }
    }
  }
}
```

#14.8.2 filter与query对比

filter，仅仅只是按照搜索条件过滤出需要的数据而已，不计算任何相关度分数，对相关度没有任何影响。

query，会去计算每个document相对于搜索条件的相关度，并按照相关度进行排序。

应用场景：

一般来说，如果你是在进行搜索，需要将最匹配搜索条件的数据先返回，那么用query 如果你只是要根据一些条件筛选出一部分数据，不关注其排序，那么用filter

#14.8.3 filter与query性能

filter，不需要计算相关度分数，不需要按照相关度分数进行排序，同时还有内置的自动cache最常使用filter的数据

query，相反，要计算相关度分数，按照分数进行排序，而且无法cache结果

#14.9. 定位错误语法

验证错误语句:

```
GET /book/_validate/query?explain
{
  "query": {
    "mach": {
      "description": "java程序员"
    }
  }
}
```

返回:

```
{
  "valid" : false,
  "error" : "org.elasticsearch.common.ParsingException: no [query] registered for [mach]"
}
```

正确

```
GET /book/_validate/query?explain
{
  "query": {
    "match": {
      "description": "java程序员"
    }
  }
}
```

返回

```
 {
   "_shards" : {
     "total" : 1,
     "successful" : 1,
     "failed" : 0
   },
   "valid" : true,
   "explanations" : [
     {
       "index" : "book",
       "valid" : true,
       "explanation" : "description:java description:程序员"
     }
   ]
 }
```

一般用在那种特别复杂庞大的搜索下，比如你一下子写了上百行的搜索，这个时候可以先用validate api去验证一下，搜索是否合法。

合法以后，explain就像mysql的执行计划，可以看到搜索的目标等信息。

#14.10． 定制排序规则

#14.10.1 默认排序规则

默认情况下，是按照_score降序排序的

然而，某些情况下，可能没有有用的_score，比如说filter

```
GET book/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "description": "java程序员"
          }
        }
      ]
    }
  }
}
```

当然，也可以是constant_score

#14.10.2 定制排序规则

相当于sql中order by ?sort=sprice:desc

```
GET /book/_search
{
  "query": {
    "constant_score": {
      "filter" : {
            "term" : {
                "studymodel" : "201001"
            }
        }
    }
  },
  "sort": [
    {
      "price": {
        "order": "asc"
      }
    }
  ]
}
```

#14.11． Text字段排序问题

如果对一个text field进行排序，结果往往不准确，因为分词后是多个单词，再排序就不是我们想要的结果了。

通常解决方案是，将一个text field建立两次索引，一个分词，用来进行搜索；一个不分词，用来进行排序。

fielddate:true

```
PUT /website
{
  "mappings": {
  "properties": {
    "title": {
      "type": "text",
      "fields": {
        "keyword": {
          "type": "keyword"
        }
      }
    },
    "content": {
      "type": "text"
    },
    "post_date": {
      "type": "date"
    },
    "author_id": {
      "type": "long"
    }
  }
 }
}
```

插入数据

```
PUT /website/_doc/1
{
  "title": "first article",
  "content": "this is my second article",
  "post_date": "2019-01-01",
  "author_id": 110
}

PUT /website/_doc/2
{
    "title": "second article",
    "content": "this is my second article",
     "post_date": "2019-01-01",
    "author_id": 110
}

PUT /website/_doc/3
{
     "title": "third article",
     "content": "this is my third article",
     "post_date": "2019-01-02",
     "author_id": 110
}
```

搜索

```
GET /website/_search
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "title.keyword": {
        "order": "desc"
      }
    }
  ]
}
```

#14.12. Scroll分批查询

场景：下载某一个索引中1亿条数据，到文件或是数据库。

不能一下全查出来，系统内存溢出。所以使用scoll滚动搜索技术，一批一批查询。

scoll搜索会在第一次搜索的时候，保存一个当时的视图快照，之后只会基于该旧的视图快照提供数据搜索，如果这个期间数据变更，是不会让用户看到的

每次发送scroll请求，我们还需要指定一个scoll参数，指定一个时间窗口，每次搜索请求只要在这个时间窗口内能完成就可以了。

搜索

```
GET /book/_search?scroll=1m
{
  "query": {
    "match_all": {}
  },
  "size": 3
}
```

返回

```
{
  "_scroll_id" :
"DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAMOkWTURBNDUtcjZTVUdKMFp5cXloVElOQQ==",
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 3,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [

    ]
```

```
    }
  }
```

获得的结果会有一个scoll_id，下一次再发送scoll请求的时候，必须带上这个scoll_id

```
GET /_search/scroll
{
    "scroll": "1m",
    "scroll_id" :
"DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAMOkWTURBNDUtcjZTVUdKMFp5cXloVElOQQ=="
}
```

与分页区别：

分页给用户看的 deep paging

scroll是用户系统内部操作，如下载批量数据，数据转移。零停机改变索引映射。

#15． java api实现搜索

#16． 评分机制详解

#16.1． 评分机制 TF\IDF

#16.1.1 算法介绍

relevance score算法，简单来说，就是计算出，一个索引中的文本，与搜索文本，他们之间的关联匹配程度。

Elasticsearch使用的是 term frequency/inverse document frequency算法，简称为TF/IDF算法。TF词频(Term Frequency)，IDF逆向文件频率(Inverse Document Frequency)

Term frequency：搜索文本中的各个词条在field文本中出现了多少次，出现次数越多，就越相关。

公式：$$tf_{ij} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

即：$$TF_w = \frac{在某一类中词条w出现的次数}{该类中所有的词条数目}$$

举例：搜索请求：hello world

doc1 : hello you and me,and world is very good.

doc2 : hello,how are you

Inverse document frequency：搜索文本中的各个词条在整个索引的所有文档中出现了多少次，出现的次数越多，就越不相关.

公式：$$idf_i = \log \frac{|D|}{|\{j : t_i \in d_j\}|}$$

$$IDF = log(\frac{\text{语料库的文档总数}}{\text{包含词条}w\text{的文档数}+1}),\text{分母之所以要加}1\text{，是为了避免分母为}0$$

即：

举例：搜索请求：hello world

doc1 : hello ,today is very good

doc2 : hi world ,how are you

整个index中1亿条数据。hello的document 1000个，有world的document 有100个。

doc2 更相关

Field-length norm：field长度，field越长，相关度越弱

举例：搜索请求：hello world

doc1 : {"title":"hello article","content ":"balabalabal 1万个"}

doc2 : {"title":"my article","content ":"balabalabal 1万个,world"}

#16.1.2 _score是如何被计算出来的

```
GET /book/_search?explain=true
{
  "query": {
    "match": {
      "description": "java程序员"
    }
  }
}
```

返回

```
 {
   "took" : 5,
   "timed_out" : false,
   "_shards" : {
     "total" : 1,
     "successful" : 1,
     "skipped" : 0,
     "failed" : 0
   },
   "hits" : {
     "total" : {
       "value" : 2,
       "relation" : "eq"
     },
     "max_score" : 2.137549,
     "hits" : [
       {
         "_shard" : "[book][0]",
         "_node" : "MDA45-r6SUGJ0ZyqyhTINA",
         "_index" : "book",
         "_type" : "_doc",
         "_id" : "3",
         "_score" : 2.137549,
```

```
        "_source" : {
          "name" : "spring开发基础",
          "description" : "spring 在java领域非常流行，java程序员都在用。",
          "studymodel" : "201001",
          "price" : 88.6,
          "timestamp" : "2019-08-24 19:11:35",
          "pic" : "group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
          "tags" : [
            "spring",
            "java"
          ]
        },
        "_explanation" : {
          "value" : 2.137549,
          "description" : "sum of:",
          "details" : [
            {
              "value" : 0.7936629,
              "description" : "weight(description:java in 0)
[PerFieldSimilarity], result of:",
              "details" : [
                {
                  "value" : 0.7936629,
                  "description" : "score(freq=2.0), product of:",
                  "details" : [
                    {
                      "value" : 2.2,
                      "description" : "boost",
                      "details" : [ ]
                    },
                    {
                      "value" : 0.47000363,
                      "description" : "idf, computed as log(1 + (N - n + 0.5) /
(n + 0.5)) from:",
                      "details" : [
                        {
                          "value" : 2,
                          "description" : "n, number of documents containing
term",
                          "details" : [ ]
                        },
                        {
                          "value" : 3,
                          "description" : "N, total number of documents with
field",
                          "details" : [ ]
                        }
                      ]
                    },
                    {
                      "value" : 0.7675597,
                      "description" : "tf, computed as freq / (freq + k1 * (1 -
b + b * dl / avgdl)) from:",
                      "details" : [
                        {
                          "value" : 2.0,
                          "description" : "freq, occurrences of term within
document",
```

```
                          "details" : [ ]
                        },
                        {
                          "value" : 1.2,
                          "description" : "k1, term saturation parameter",
                          "details" : [ ]
                        },
                        {
                          "value" : 0.75,
                          "description" : "b, length normalization parameter",
                          "details" : [ ]
                        },
                        {
                          "value" : 12.0,
                          "description" : "dl, length of field",
                          "details" : [ ]
                        },
                        {
                          "value" : 35.333332,
                          "description" : "avgdl, average length of field",
                          "details" : [ ]
                        }
                      ]
                    }
                  ]
                }
              ]
            },
            {
              "value" : 1.3438859,
              "description" : "weight(description:程序员 in 0)
[PerFieldSimilarity], result of:",
              "details" : [
                {
                  "value" : 1.3438859,
                  "description" : "score(freq=1.0), product of:",
                  "details" : [
                    {
                      "value" : 2.2,
                      "description" : "boost",
                      "details" : [ ]
                    },
                    {
                      "value" : 0.98082924,
                      "description" : "idf, computed as log(1 + (N - n + 0.5) /
(n + 0.5)) from:",
                      "details" : [
                        {
                          "value" : 1,
                          "description" : "n, number of documents containing
term",
                          "details" : [ ]
                        },
                        {
                          "value" : 3,
                          "description" : "N, total number of documents with
field",
                          "details" : [ ]
```

```
                        }
                      ]
                    },
                    {
                      "value" : 0.6227967,
                      "description" : "tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:",
                      "details" : [
                        {
                          "value" : 1.0,
                          "description" : "freq, occurrences of term within document",
                          "details" : [ ]
                        },
                        {
                          "value" : 1.2,
                          "description" : "k1, term saturation parameter",
                          "details" : [ ]
                        },
                        {
                          "value" : 0.75,
                          "description" : "b, length normalization parameter",
                          "details" : [ ]
                        },
                        {
                          "value" : 12.0,
                          "description" : "dl, length of field",
                          "details" : [ ]
                        },
                        {
                          "value" : 35.333332,
                          "description" : "avgdl, average length of field",
                          "details" : [ ]
                        }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      },
      {
        "_shard" : "[book][0]",
        "_node" : "MDA45-r6SUGJ0ZyqyhTINA",
        "_index" : "book",
        "_type" : "_doc",
        "_id" : "2",
        "_score" : 0.57961315,
        "_source" : {
          "name" : "java编程思想",
          "description" : "java语言是世界第一编程语言，在软件开发领域使用人数最多。",
          "studymodel" : "201001",
          "price" : 68.6,
          "timestamp" : "2019-08-25 19:11:35",
          "pic" : "group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg",
          "tags" : [
```

```
              "java",
              "dev"
            ]
          },
          "_explanation" : {
            "value" : 0.57961315,
            "description" : "sum of:",
            "details" : [
              {
                "value" : 0.57961315,
                "description" : "weight(description:java in 0)
[PerFieldSimilarity], result of:",
                "details" : [
                  {
                    "value" : 0.57961315,
                    "description" : "score(freq=1.0), product of:",
                    "details" : [
                      {
                        "value" : 2.2,
                        "description" : "boost",
                        "details" : [ ]
                      },
                      {
                        "value" : 0.47000363,
                        "description" : "idf, computed as log(1 + (N - n + 0.5) /
(n + 0.5)) from:",
                        "details" : [
                          {
                            "value" : 2,
                            "description" : "n, number of documents containing
term",
                            "details" : [ ]
                          },
                          {
                            "value" : 3,
                            "description" : "N, total number of documents with
field",
                            "details" : [ ]
                          }
                        ]
                      },
                      {
                        "value" : 0.56055,
                        "description" : "tf, computed as freq / (freq + k1 * (1 -
b + b * dl / avgdl)) from:",
                        "details" : [
                          {
                            "value" : 1.0,
                            "description" : "freq, occurrences of term within
document",
                            "details" : [ ]
                          },
                          {
                            "value" : 1.2,
                            "description" : "k1, term saturation parameter",
                            "details" : [ ]
                          },
                          {
```

```
                            "value" : 0.75,
                            "description" : "b, length normalization parameter",
                            "details" : [ ]
                          },
                          {
                            "value" : 19.0,
                            "description" : "dl, length of field",
                            "details" : [ ]
                          },
                          {
                            "value" : 35.333332,
                            "description" : "avgdl, average length of field",
                            "details" : [ ]
                          }
                        ]
                      }
                    ]
                  }
                ]
              }
            ]
          }
        ]
      }
    }
```

#16.1.3 分析一个document是如何被匹配上的

```
GET /book/_explain/3
{
  "query": {
    "match": {
      "description": "java程序员"
    }
  }
}
```

#16.2． Doc value

搜索的时候，要依靠倒排索引；排序的时候，需要依靠正排索引，看到每个document的每个field，然后进行排序，所谓的正排索引，其实就是doc values

在建立索引的时候，一方面会建立倒排索引，以供搜索用；一方面会建立正排索引，也就是doc values，以供排序，聚合，过滤等操作使用

doc values是被保存在磁盘上的，此时如果内存足够，os会自动将其缓存在内存中，性能还是会很高；如果内存不足够，os会将其写入磁盘上

倒排索引

doc1: hello world you and me

doc2: hi, world, how are you

term	doc1	doc2
hello	*	
world	*	*
you	*	*
and	*	
me	*	
hi		*
how		*
are		*

搜索时：

hello you --> hello, you

hello --> doc1

you --> doc1,doc2

doc1: hello world you and me

doc2: hi, world, how are you

sort by 出现问题

正排索引

doc1: { "name": "jack", "age": 27 }

doc2: { "name": "tom", "age": 30 }

document	name	age
doc1	jack	27
doc2	tom	30

#16.3． query phase

#1、query phase

（1）搜索请求发送到某一个coordinate node，构构建一个priority queue，长度以paging操作from和size为准，默认为10

（2）coordinate node将请求转发到所有shard，每个shard本地搜索，并构建一个本地的priority queue

（3）各个shard将自己的priority queue返回给coordinate node，并构建一个全局的priority queue

#2、replica shard如何提升搜索吞吐量

一次请求要打到所有shard的一个replica/primary上去，如果每个shard都有多个replica，那么同时并发过来的搜索请求可以同时打到其他的replica上去

#16.4. fetch phase

#1、fetch phbase工作流程

（1）coordinate node构建完priority queue之后，就发送mget请求去所有shard上获取对应的document

（2）各个shard将document返回给coordinate node

（3）coordinate node将合并后的document结果返回给client客户端

#2、一般搜索，如果不加from和size，就默认搜索前10条，按照_score排序

#16.5. 搜索参数小总结

#1、preference

决定了哪些shard会被用来执行搜索操作

_primary, _primary_first, _local, _only_node:xyz, _prefer_node:xyz, _shards:2,3

bouncing results问题，两个document排序，field值相同；不同的shard上，可能排序不同；每次请求轮询打到不同的replica shard上；每次页面上看到的搜索结果的排序都不一样。这就是bouncing result，也就是跳跃的结果。

搜索的时候，是轮询将搜索请求发送到每一个replica shard（primary shard），但是在不同的shard上，可能document的排序不同

解决方案就是将preference设置为一个字符串，比如说user_id，让每个user每次搜索的时候，都使用同一个replica shard去执行，就不会看到bouncing results了

#2、timeout

已经讲解过原理了，主要就是限定在一定时间内，将部分获取到的数据直接返回，避免查询耗时过长

#3、routing

document文档路由，_id路由，routing=user_id，这样的话可以让同一个user对应的数据到一个shard上去

#4、search_type

default：query_then_fetch

dfs_query_then_fetch，可以提升revelance sort精准度

#17. 聚合入门

#17.1聚合示例

#17.1.1需求：计算每个studymodel下的商品数量

sql语句： select studymodel，count(*) from book group by studymodel

```
GET /book/_search
{
  "size": 0,
  "query": {
    "match_all": {}
  },
  "aggs": {
    "group_by_model": {
      "terms": { "field": "studymodel" }
    }
  }
}
```

#17.1.2 需求：计算每个tags下的商品数量

设置字段"fielddata": true

```
PUT /book/_mapping/
{
  "properties": {
    "tags": {
      "type": "text",
      "fielddata": true
    }
  }
}
```

查询

```
GET /book/_search
{
  "size": 0,
  "query": {
    "match_all": {}
  },
  "aggs": {
    "group_by_tags": {
      "terms": { "field": "tags" }
    }
  }
}
```

#17.1.3 需求：加上搜索条件，计算每个tags下的商品数量

```
GET /book/_search
{
  "size": 0,
  "query": {
    "match": {
```

```
        "description": "java程序员"
      }
    },
    "aggs": {
      "group_by_tags": {
        "terms": { "field": "tags" }
      }
    }
}
```

#17.1.4 需求：先分组，再算每组的平均值，计算每个tag下的商品的平均价格

```
GET /book/_search
{
    "size": 0,
    "aggs" : {
        "group_by_tags" : {
            "terms" : {
              "field" : "tags"
            },
            "aggs" : {
                "avg_price" : {
                    "avg" : { "field" : "price" }
                }
            }
        }
    }
}
```

#17.1.5 需求：计算每个tag下的商品的平均价格，并且按照平均价格降序排序

```
GET /book/_search
{
    "size": 0,
    "aggs" : {
        "group_by_tags" : {
            "terms" : {
              "field" : "tags",
              "order": {
                "avg_price": "desc"
              }
            },
            "aggs" : {
                "avg_price" : {
                    "avg" : { "field" : "price" }
                }
            }
        }
    }
}
```

#17.1.6 需求：按照指定的价格范围区间进行分组，然后在每组内再按照tag进行分组，最后再计算每组的平均价格

```
GET /book/_search
{
  "size": 0,
  "aggs": {
    "group_by_price": {
      "range": {
        "field": "price",
        "ranges": [
          {
            "from": 0,
            "to": 40
          },
          {
            "from": 40,
            "to": 60
          },
          {
            "from": 60,
            "to": 80
          }
        ]
      },
      "aggs": {
        "group_by_tags": {
          "terms": {
            "field": "tags"
          },
          "aggs": {
            "average_price": {
              "avg": {
                "field": "price"
              }
            }
          }
        }
      }
    }
  }
}
```

#17.2两个核心概念：bucket和metric

#17.2.1 bucket：一个数据分组

city name 北京 张三 北京 李四 天津 王五 天津 赵六

天津 王麻子

划分出来两个bucket，一个是北京bucket，一个是天津bucket 北京bucket：包含了2个人，张三，李四 上海bucket：包含了3个人，王五，赵六，王麻子

#17.2.2metric：对一个数据分组执行的统计

metric，就是对一个bucket执行的某种聚合分析的操作，比如说求平均值，求最大值，求最小值

select count(*) from book group studymodel

bucket：group by studymodel --> 那些studymodel相同的数据，就会被划分到一个bucket中 metric：count(*)，对每个user_id bucket中所有的数据，计算一个数量。还有avg()，sum()，max()，min()

#17.3 电视案例

创建索引及映射

```
PUT /tvs
PUT /tvs/_search
{
            "properties": {
                "price": {
                    "type": "long"
                },
                "color": {
                    "type": "keyword"
                },
                "brand": {
                    "type": "keyword"
                },
                "sold_date": {
                    "type": "date"
                }
            }
}
```

插入数据

```
POST /tvs/_bulk
{ "index": {}}
{ "price" : 1000, "color" : "红色", "brand" : "长虹", "sold_date" : "2019-10-28" }
{ "index": {}}
{ "price" : 2000, "color" : "红色", "brand" : "长虹", "sold_date" : "2019-11-05" }
{ "index": {}}
{ "price" : 3000, "color" : "绿色", "brand" : "小米", "sold_date" : "2019-05-18" }
{ "index": {}}
{ "price" : 1500, "color" : "蓝色", "brand" : "TCL", "sold_date" : "2019-07-02" }
{ "index": {}}
{ "price" : 1200, "color" : "绿色", "brand" : "TCL", "sold_date" : "2019-08-19" }
{ "index": {}}
{ "price" : 2000, "color" : "红色", "brand" : "长虹", "sold_date" : "2019-11-05" }
{ "index": {}}
{ "price" : 8000, "color" : "红色", "brand" : "三星", "sold_date" : "2020-01-01" }
{ "index": {}}
{ "price" : 2500, "color" : "蓝色", "brand" : "小米", "sold_date" : "2020-02-12" }
```

#需求1 统计哪种颜色的电视销量最高

```
GET /tvs/_search
{
    "size" : 0,
    "aggs" : {
        "popular_colors" : {
            "terms" : {
              "field" : "color"
            }
        }
    }
}
```

查询条件解析

size：只获取聚合结果，而不要执行聚合的原始数据 aggs：固定语法，要对一份数据执行分组聚合操作 popular_colors：就是对每个aggs，都要起一个名字， terms：根据字段的值进行分组 field：根据指定的字段的值进行分组

返回

```
{
  "took" : 18,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 8,
      "relation" : "eq"
    },
    "max_score" : null,
    "hits" : [ ]
  },
  "aggregations" : {
    "popular_colors" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 0,
      "buckets" : [
        {
          "key" : "红色",
          "doc_count" : 4
        },
        {
          "key" : "绿色",
          "doc_count" : 2
        },
        {
          "key" : "蓝色",
          "doc_count" : 2
        }
      ]
    }
```

```
    }
}
```

返回结果解析

hits.hits：我们指定了size是0，所以hits.hits就是空的 aggregations：聚合结果 popular_color：我们指定的某个聚合的名称 buckets：根据我们指定的field划分出的buckets key：每个bucket对应的那个值 doc_count：这个bucket分组内，有多少个数据 数量，其实就是这种颜色的销量

每种颜色对应的bucket中的数据的默认的排序规则：按照doc_count降序排序

#需求2 统计每种颜色电视平均价格

```
GET /tvs/_search
{
   "size" : 0,
   "aggs": {
      "colors": {
         "terms": {
            "field": "color"
         },
         "aggs": {
            "avg_price": {
               "avg": {
                  "field": "price"
               }
            }
         }
      }
   }
}
```

在一个aggs执行的bucket操作（terms），平级的json结构下，再加一个aggs，这个第二个aggs内部，同样取个名字，执行一个metric操作，avg，对之前的每个bucket中的数据的指定的field，price field，求一个平均值

返回：

```
{
  "took" : 4,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 8,
      "relation" : "eq"
    },
    "max_score" : null,
    "hits" : [ ]
  },
  "aggregations" : {
    "colors" : {
      "doc_count_error_upper_bound" : 0,
```

```
        "sum_other_doc_count" : 0,
        "buckets" : [
          {
            "key" : "红色",
            "doc_count" : 4,
            "avg_price" : {
              "value" : 3250.0
            }
          },
          {
            "key" : "绿色",
            "doc_count" : 2,
            "avg_price" : {
              "value" : 2100.0
            }
          },
          {
            "key" : "蓝色",
            "doc_count" : 2,
            "avg_price" : {
              "value" : 2000.0
            }
          }
        ]
      }
    }
}
```

buckets，除了key和doc_count avg_price：我们自己取的metric aggs的名字 value：我们的metric计算的结果，每个bucket中的数据的price字段求平均值后的结果

相当于sql: select avg(price) from tvs group by color

#需求3 继续下钻分析

每个颜色下，平均价格及每个颜色下，每个品牌的平均价格

```
GET /tvs/_search
{
  "size": 0,
  "aggs": {
    "group_by_color": {
      "terms": {
        "field": "color"
      },
      "aggs": {
        "color_avg_price": {
          "avg": {
            "field": "price"
          }
        },
        "group_by_brand": {
          "terms": {
            "field": "brand"
          },
          "aggs": {
            "brand_avg_price": {
              "avg": {
```

```
                "field": "price"
              }
            }
          }
        }
      }
    }
  }
```

#需求4：更多的metric

count：bucket，terms，自动就会有一个doc_count，就相当于是count avg：avg aggs，求平均值 max：求一个bucket内，指定field值最大的那个数据 min：求一个bucket内，指定field值最小的那个数据 sum：求一个bucket内，指定field值的总和

```
GET /tvs/_search
{
   "size" : 0,
   "aggs": {
      "colors": {
         "terms": {
            "field": "color"
         },
         "aggs": {
            "avg_price": { "avg": { "field": "price" } },
            "min_price" : { "min": { "field": "price"} },
            "max_price" : { "max": { "field": "price"} },
            "sum_price" : { "sum": { "field": "price" } }
         }
      }
   }
}
```

#需求5：划分范围 histogram

```
GET /tvs/_search
{
   "size" : 0,
   "aggs":{
      "price":{
         "histogram":{
            "field": "price",
            "interval": 2000
         },
         "aggs":{
            "income": {
               "sum": {
                 "field" : "price"
               }
             }
         }
      }
   }
}
```

histogram：类似于terms，也是进行bucket分组操作，接收一个field，按照这个field的值的各个范围区间，进行bucket分组操作

```
"histogram":{
  "field": "price",
  "interval": 2000
}
```

interval：2000，划分范围，0~2000，2000~4000，4000~6000，6000~8000，8000~10000，buckets

bucket有了之后，一样的，去对每个bucket执行avg，count，sum，max，min，等各种metric操作，聚合分析

#需求6：按照日期分组聚合

date_histogram，按照我们指定的某个date类型的日期field，以及日期interval，按照一定的日期间隔，去划分bucket

min_doc_count：即使某个日期interval，2017-01-01~2017-01-31中，一条数据都没有，那么这个区间也是要返回的，不然默认是会过滤掉这个区间的 extended_bounds，min，max：划分bucket的时候，会限定在这个起始日期，和截止日期内

```
GET /tvs/_search
{
   "size" : 0,
   "aggs": {
      "sales": {
         "date_histogram": {
            "field": "sold_date",
            "interval": "month",
            "format": "yyyy-MM-dd",
            "min_doc_count" : 0,
            "extended_bounds" : {
                "min" : "2019-01-01",
                "max" : "2020-12-31"
            }
         }
      }
   }
}
```

#需求7 统计每季度每个品牌的销售额

```
GET /tvs/_search
{
  "size": 0,
  "aggs": {
    "group_by_sold_date": {
      "date_histogram": {
        "field": "sold_date",
        "interval": "quarter",
        "format": "yyyy-MM-dd",
        "min_doc_count": 0,
        "extended_bounds": {
          "min": "2019-01-01",
          "max": "2020-12-31"
        }
```

```
      },
      "aggs": {
        "group_by_brand": {
          "terms": {
            "field": "brand"
          },
          "aggs": {
            "sum_price": {
              "sum": {
                "field": "price"
              }
            }
          }
        },
        "total_sum_price": {
          "sum": {
            "field": "price"
          }
        }
      }
    }
  }
```

求8： 搜索与聚合结合，查询某个品牌按颜色销量

搜索与聚合可以结合起来。

sql select count(*)

from tvs

where brand like "%小米%"

group by color

es aggregation，scope，任何的聚合，都必须在搜索出来的结果数据中之行，搜索结果，就是聚合分析操作的scope

```
GET /tvs/_search
{
  "size": 0,
  "query": {
    "term": {
      "brand": {
        "value": "小米"
      }
    }
  },
  "aggs": {
    "group_by_color": {
      "terms": {
        "field": "color"
      }
    }
  }
}
```

#需求9 global bucket：单个品牌与所有品牌销量对比

aggregation，scope，一个聚合操作，必须在query的搜索结果范围内执行

出来两个结果，一个结果，是基于query搜索结果来聚合的; 一个结果，是对所有数据执行聚合的

```
GET /tvs/_search
{
  "size": 0,
  "query": {
    "term": {
      "brand": {
        "value": "小米"
      }
    }
  },
  "aggs": {
    "single_brand_avg_price": {
      "avg": {
        "field": "price"
      }
    },
    "all": {
      "global": {},
      "aggs": {
        "all_brand_avg_price": {
          "avg": {
            "field": "price"
          }
        }
      }
    }
  }
}
```

#需求10：过滤+聚合：统计价格大于1200的电视平均价格

搜索+聚合

过滤+聚合

```
GET /tvs/_search
{
  "size": 0,
  "query": {
    "constant_score": {
      "filter": {
        "range": {
          "price": {
            "gte": 1200
          }
        }
      }
    }
  },
  "aggs": {
    "avg_price": {
      "avg": {
```

```
          "field": "price"
        }
      }
    }
  }
```

#需求11 bucket filter：统计品牌最近一个月的平均价格

```
GET /tvs/_search
{
  "size": 0,
  "query": {
    "term": {
      "brand": {
        "value": "小米"
      }
    }
  },
  "aggs": {
    "recent_150d": {
      "filter": {
        "range": {
          "sold_date": {
            "gte": "now-150d"
          }
        }
      },
      "aggs": {
        "recent_150d_avg_price": {
          "avg": {
            "field": "price"
          }
        }
      }
    },
    "recent_140d": {
      "filter": {
        "range": {
          "sold_date": {
            "gte": "now-140d"
          }
        }
      },
      "aggs": {
        "recent_140d_avg_price": {
          "avg": {
            "field": "price"
          }
        }
      }
    },
    "recent_130d": {
      "filter": {
        "range": {
          "sold_date": {
            "gte": "now-130d"
          }
```

```
          }
        },
        "aggs": {
          "recent_130d_avg_price": {
            "avg": {
              "field": "price"
            }
          }
        }
      }
    }
  }
```

aggs.filter，针对的是聚合去做的

如果放query里面的filter，是全局的，会对所有的数据都有影响

但是，如果，比如说，你要统计，长虹电视，最近1个月的平均值; 最近3个月的平均值; 最近6个月的平均值

bucket filter：对不同的bucket下的aggs，进行filter

#需求12 排序：按每种颜色的平均销售额降序排序

```
GET /tvs/_search
{
  "size": 0,
  "aggs": {
    "group_by_color": {
      "terms": {
        "field": "color",
        "order": {
          "avg_price": "asc"
        }
      },
      "aggs": {
        "avg_price": {
          "avg": {
            "field": "price"
          }
        }
      }
    }
  }
}
```

相当于sql子表数据字段可以立刻使用。

#需求13 排序：按每种颜色的每种品牌平均销售额降序排序

```
GET /tvs/_search
{
  "size": 0,
  "aggs": {
    "group_by_color": {
      "terms": {
        "field": "color"
      },
```

```
      "aggs": {
        "group_by_brand": {
          "terms": {
            "field": "brand",
            "order": {
              "avg_price": "desc"
            }
          },
          "aggs": {
            "avg_price": {
              "avg": {
                "field": "price"
              }
            }
          }
        }
      }
    }
  }
```

#18. java api实现聚合

简单聚合，多种聚合，详见代码。

```
package com.ydlclass.es;

import org.elasticsearch.action.search.SearchRequest;
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.client.RequestOptions;
import org.elasticsearch.client.RestHighLevelClient;
import org.elasticsearch.index.query.QueryBuilders;
import org.elasticsearch.search.aggregations.Aggregation;
import org.elasticsearch.search.aggregations.AggregationBuilders;
import org.elasticsearch.search.aggregations.Aggregations;
import org.elasticsearch.search.aggregations.bucket.histogram.*;
import org.elasticsearch.search.aggregations.bucket.terms.Terms;
import
org.elasticsearch.search.aggregations.bucket.terms.TermsAggregationBuilder;
import org.elasticsearch.search.aggregations.metrics.*;
import org.elasticsearch.search.builder.SearchSourceBuilder;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

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

/**
 * creste by ydlclass.ydl
 */
@SpringBootTest
@RunWith(SpringRunner.class)
public class TestAggs {
    @Autowired
```

```
32      RestHighLevelClient client;
33
34      //需求一：按照颜色分组，计算每个颜色卖出的个数
35      @Test
36      public void testAggs() throws IOException {
37          // GET /tvs/_search
38          // {
39          //     "size": 0,
40          //     "query": {"match_all": {}},
41          //     "aggs": {
42          //       "group_by_color": {
43          //         "terms": {
44          //             "field": "color"
45          //         }
46          //     }
47          // }
48          // }
49
50          //1 构建请求
51          SearchRequest searchRequest=new SearchRequest("tvs");
52
53          //请求体
54          SearchSourceBuilder searchSourceBuilder=new SearchSourceBuilder();
55          searchSourceBuilder.size(0);
56          searchSourceBuilder.query(QueryBuilders.matchAllQuery());
57
58          TermsAggregationBuilder termsAggregationBuilder =
    AggregationBuilders.terms("group_by_color").field("color");
59          searchSourceBuilder.aggregation(termsAggregationBuilder);
60
61          //请求体放入请求头
62          searchRequest.source(searchSourceBuilder);
63
64          //2 执行
65          SearchResponse searchResponse = client.search(searchRequest,
    RequestOptions.DEFAULT);
66
67          //3 获取结果
68        //   "aggregations" : {
69        //       "group_by_color" : {
70        //           "doc_count_error_upper_bound" : 0,
71        //           "sum_other_doc_count" : 0,
72        //            "buckets" : [
73        //           {
74        //               "key" : "红色",
75        //               "doc_count" : 4
76        //           },
77        //           {
78        //               "key" : "绿色",
79        //                   "doc_count" : 2
80        //           },
81        //           {
82        //               "key" : "蓝色",
83        //                   "doc_count" : 2
84        //           }
85        // ]
86        //       }
87          Aggregations aggregations = searchResponse.getAggregations();
```

```
        Terms group_by_color = aggregations.get("group_by_color");
        List<? extends Terms.Bucket> buckets = group_by_color.getBuckets();
        for (Terms.Bucket bucket : buckets) {
            String key = bucket.getKeyAsString();
            System.out.println("key:"+key);

            long docCount = bucket.getDocCount();
            System.out.println("docCount:"+docCount);

            System.out.println("================================");
        }
    }

    // #需求二：按照颜色分组，计算每个颜色卖出的个数，每个颜色卖出的平均价格
    @Test
    public void testAggsAndAvg() throws IOException {
        // GET /tvs/_search
        // {
        //     "size": 0,
        //      "query": {"match_all": {}},
        //     "aggs": {
        //     "group_by_color": {
        //         "terms": {
        //             "field": "color"
        //         },
        //         "aggs": {
        //             "avg_price": {
        //                 "avg": {
        //                     "field": "price"
        //                 }
        //             }
        //         }
        //     }
        // }
        // }

        //1 构建请求
        SearchRequest searchRequest=new SearchRequest("tvs");

        //请求体
        SearchSourceBuilder searchSourceBuilder=new SearchSourceBuilder();
        searchSourceBuilder.size(0);
        searchSourceBuilder.query(QueryBuilders.matchAllQuery());

        TermsAggregationBuilder termsAggregationBuilder =
AggregationBuilders.terms("group_by_color").field("color");

        //terms聚合下填充一个子聚合
        AvgAggregationBuilder avgAggregationBuilder =
AggregationBuilders.avg("avg_price").field("price");
        termsAggregationBuilder.subAggregation(avgAggregationBuilder);

        searchSourceBuilder.aggregation(termsAggregationBuilder);

        //请求体放入请求头
        searchRequest.source(searchSourceBuilder);

        //2 执行
```

```
144             SearchResponse searchResponse = client.search(searchRequest,
    RequestOptions.DEFAULT);
145
146             //3 获取结果
147             // {
148             //     "key" : "红色",
149             //      "doc_count" : 4,
150             //      "avg_price" : {
151             //        "value" : 3250.0
152             //       }
153             // }
154             Aggregations aggregations = searchResponse.getAggregations();
155             Terms group_by_color = aggregations.get("group_by_color");
156             List<? extends Terms.Bucket> buckets = group_by_color.getBuckets();
157             for (Terms.Bucket bucket : buckets) {
158                 String key = bucket.getKeyAsString();
159                 System.out.println("key:"+key);
160
161                 long docCount = bucket.getDocCount();
162                 System.out.println("docCount:"+docCount);
163
164                 Aggregations aggregations1 = bucket.getAggregations();
165                 Avg avg_price = aggregations1.get("avg_price");
166                 double value = avg_price.getValue();
167                 System.out.println("value:"+value);
168
169                 System.out.println("================================");
170             }
171         }
172
173         // #需求三：按照颜色分组，计算每个颜色卖出的个数，以及每个颜色卖出的平均值、最大值、最小
    值、总和。
174         @Test
175         public void testAggsAndMore() throws IOException {
176             // GET /tvs/_search
177             // {
178             //     "size" : 0,
179             //     "aggs": {
180             //      "group_by_color": {
181             //         "terms": {
182             //             "field": "color"
183             //         },
184             //         "aggs": {
185             //             "avg_price": { "avg": { "field": "price" } },
186             //             "min_price" : { "min": { "field": "price"} },
187             //             "max_price" : { "max": { "field": "price"} },
188             //             "sum_price" : { "sum": { "field": "price" } }
189             //         }
190             //     }
191             // }
192             // }
193
194             //1 构建请求
195             SearchRequest searchRequest=new SearchRequest("tvs");
196
197             //请求体
198             SearchSourceBuilder searchSourceBuilder=new SearchSourceBuilder();
199             searchSourceBuilder.size(0);
```

```
200         searchSourceBuilder.query(QueryBuilders.matchAllQuery());
201
202         TermsAggregationBuilder termsAggregationBuilder =
    AggregationBuilders.terms("group_by_color").field("color");
203
204
205         //termsAggregationBuilder里放入多个子聚合
206         AvgAggregationBuilder avgAggregationBuilder =
    AggregationBuilders.avg("avg_price").field("price");
207         MinAggregationBuilder minAggregationBuilder =
    AggregationBuilders.min("min_price").field("price");
208         MaxAggregationBuilder maxAggregationBuilder =
    AggregationBuilders.max("max_price").field("price");
209         SumAggregationBuilder sumAggregationBuilder =
    AggregationBuilders.sum("sum_price").field("price");
210
211         termsAggregationBuilder.subAggregation(avgAggregationBuilder);
212         termsAggregationBuilder.subAggregation(minAggregationBuilder);
213         termsAggregationBuilder.subAggregation(maxAggregationBuilder);
214         termsAggregationBuilder.subAggregation(sumAggregationBuilder);
215
216
217         searchSourceBuilder.aggregation(termsAggregationBuilder);
218
219         //请求体放入请求头
220         searchRequest.source(searchSourceBuilder);
221
222         //2 执行
223         SearchResponse searchResponse = client.search(searchRequest,
    RequestOptions.DEFAULT);
224
225         //3 获取结果
226         // {
227         //     "key" : "红色",
228         //     "doc_count" : 4,
229         //     "max_price" : {
230         //          "value" : 8000.0
231         //     },
232         //     "min_price" : {
233         //          "value" : 1000.0
234         // },
235         //     "avg_price" : {
236         //         "value" : 3250.0
237         // },
238         //     "sum_price" : {
239         //         "value" : 13000.0
240         // }
241         // }
242         Aggregations aggregations = searchResponse.getAggregations();
243         Terms group_by_color = aggregations.get("group_by_color");
244         List<? extends Terms.Bucket> buckets = group_by_color.getBuckets();
245         for (Terms.Bucket bucket : buckets) {
246             String key = bucket.getKeyAsString();
247             System.out.println("key:"+key);
248
249             long docCount = bucket.getDocCount();
250             System.out.println("docCount:"+docCount);
251
```

```
            Aggregations aggregations1 = bucket.getAggregations();

            Max max_price = aggregations1.get("max_price");
            double maxPriceValue = max_price.getValue();
            System.out.println("maxPriceValue:"+maxPriceValue);

            Min min_price = aggregations1.get("min_price");
            double minPriceValue = min_price.getValue();
            System.out.println("minPriceValue:"+minPriceValue);

            Avg avg_price = aggregations1.get("avg_price");
            double avgPriceValue = avg_price.getValue();
            System.out.println("avgPriceValue:"+avgPriceValue);

            Sum sum_price = aggregations1.get("sum_price");
            double sumPriceValue = sum_price.getValue();
            System.out.println("sumPriceValue:"+sumPriceValue);

            System.out.println("==================================");
        }
    }

    // #需求四：按照售价每2000价格划分范围，算出每个区间的销售总额 histogram
    @Test
    public void testAggsAndHistogram() throws IOException {
        // GET /tvs/_search
        // {
        //     "size" : 0,
        //     "aggs":{
        //      "by_histogram":{
        //         "histogram":{
        //             "field": "price",
        //             "interval": 2000
        //         },
        //         "aggs":{
        //             "income": {
        //                 "sum": {
        //                     "field" : "price"
        //                 }
        //             }
        //         }
        //     }
        // }
        // }

        //1 构建请求
        SearchRequest searchRequest=new SearchRequest("tvs");

        //请求体
        SearchSourceBuilder searchSourceBuilder=new SearchSourceBuilder();
        searchSourceBuilder.size(0);
        searchSourceBuilder.query(QueryBuilders.matchAllQuery());

        HistogramAggregationBuilder histogramAggregationBuilder =
AggregationBuilders.histogram("by_histogram").field("price").interval(2000);

        SumAggregationBuilder sumAggregationBuilder =
AggregationBuilders.sum("income").field("price");
```

```
308             histogramAggregationBuilder.subAggregation(sumAggregationBuilder);
309             searchSourceBuilder.aggregation(histogramAggregationBuilder);
310 
311             //请求体放入请求头
312             searchRequest.source(searchSourceBuilder);
313 
314             //2 执行
315             SearchResponse searchResponse = client.search(searchRequest,
    RequestOptions.DEFAULT);
316 
317             //3 获取结果
318             // {
319             //     "key" : 0.0,
320             //     "doc_count" : 3,
321             //      income" : {
322             //          "value" : 3700.0
323             //       }
324             // }
325             Aggregations aggregations = searchResponse.getAggregations();
326             Histogram group_by_color = aggregations.get("by_histogram");
327             List<? extends Histogram.Bucket> buckets = group_by_color.getBuckets();
328             for (Histogram.Bucket bucket : buckets) {
329                 String keyAsString = bucket.getKeyAsString();
330                 System.out.println("keyAsString:"+keyAsString);
331                 long docCount = bucket.getDocCount();
332                 System.out.println("docCount:"+docCount);
333 
334                 Aggregations aggregations1 = bucket.getAggregations();
335                 Sum income = aggregations1.get("income");
336                 double value = income.getValue();
337                 System.out.println("value:"+value);
338 
339                 System.out.println("==================================");
340 
341             }
342         }
343 
344         // #需求五：计算每个季度的销售总额
345         @Test
346         public void testAggsAndDateHistogram() throws IOException {
347             // GET /tvs/_search
348             // {
349             //     "size" : 0,
350             //     "aggs": {
351             //     "sales": {
352             //         "date_histogram": {
353             //                      "field": "sold_date",
354             //                     "interval": "quarter",
355             //                     "format": "yyyy-MM-dd",
356             //                     "min_doc_count" : 0,
357             //                     "extended_bounds" : {
358             //                         "min" : "2019-01-01",
359             //                         "max" : "2020-12-31"
360             //             }
361             //         },
362             //         "aggs": {
363             //             "income": {
364             //                 "sum": {
```

```
365             //                     "field": "price"
366             //                 }
367             //             }
368             //         }
369             //     }
370             // }
371             // }
372
373             //1 构建请求
374             SearchRequest searchRequest=new SearchRequest("tvs");
375
376             //请求体
377             SearchSourceBuilder searchSourceBuilder=new SearchSourceBuilder();
378             searchSourceBuilder.size(0);
379             searchSourceBuilder.query(QueryBuilders.matchAllQuery());
380
381             DateHistogramAggregationBuilder dateHistogramAggregationBuilder =
    AggregationBuilders.dateHistogram("date_histogram").field("sold_date").calendarI
    nterval(DateHistogramInterval.QUARTER)
382                     .format("yyyy-MM-dd").minDocCount(0).extendedBounds(new
    ExtendedBounds("2019-01-01", "2020-12-31"));
383             SumAggregationBuilder sumAggregationBuilder =
    AggregationBuilders.sum("income").field("price");
384             dateHistogramAggregationBuilder.subAggregation(sumAggregationBuilder);
385
386             searchSourceBuilder.aggregation(dateHistogramAggregationBuilder);
387             //请求体放入请求头
388             searchRequest.source(searchSourceBuilder);
389
390             //2 执行
391             SearchResponse searchResponse = client.search(searchRequest,
    RequestOptions.DEFAULT);
392
393             //3 获取结果
394             // {
395             //     "key_as_string" : "2019-01-01",
396             //      "key" : 1546300800000,
397             //      "doc_count" : 0,
398             //      "income" : {
399             //         "value" : 0.0
400             //      }
401             // }
402             Aggregations aggregations = searchResponse.getAggregations();
403             ParsedDateHistogram date_histogram = aggregations.get("date_histogram");
404             List<? extends Histogram.Bucket> buckets = date_histogram.getBuckets();
405             for (Histogram.Bucket bucket : buckets) {
406                 String keyAsString = bucket.getKeyAsString();
407                 System.out.println("keyAsString:"+keyAsString);
408                 long docCount = bucket.getDocCount();
409                 System.out.println("docCount:"+docCount);
410
411                 Aggregations aggregations1 = bucket.getAggregations();
412                 Sum income = aggregations1.get("income");
413                 double value = income.getValue();
414                 System.out.println("value:"+value);
415
416                 System.out.println("====================");
417             }
```

```
418
419         }
420
421     }
```

#19． es7 sql新特性

#19.1 快速入门

```
1   POST /_sql?format=txt
2   {
3       "query": "SELECT * FROM tvs "
4   }
```

#19.2启动方式

1http 请求

2客户端：elasticsearch-sql-cli.bat

3代码

#19.3显示方式

format	`Accept` HTTP header	Description
	Human Readable	
`csv`	`text/csv`	Comma-separated values
`json`	`application/json`	JSON (JavaScript Object Notation) human-readable format
`tsv`	`text/tab-separated-values`	Tab-separated values
`txt`	`text/plain`	CLI-like representation
`yaml`	`application/yaml`	YAML (YAML Ain't Markup Language) human-readable format
	Binary Formats	
`cbor`	`application/cbor`	Concise Binary Object Representation
`smile`	`application/smile`	Smile binary data format similar to CBOR

#19.4 sql 翻译

```
POST /_sql/translate
{
    "query": "SELECT * FROM tvs "
}
```

返回:

```
 {
   "size" : 1000,
   "_source" : false,
   "stored_fields" : "_none_",
   "docvalue_fields" : [
     {
       "field" : "brand"
     },
     {
       "field" : "color"
     },
     {
       "field" : "price"
     },
     {
       "field" : "sold_date",
       "format" : "epoch_millis"
```

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

#19.5 与其他DSL结合

```
POST /_sql?format=txt
{
    "query": "SELECT * FROM tvs",
    "filter": {
        "range": {
            "price": {
                "gte" : 1200,
                "lte" : 2000
            }
        }
    }
}
```

#19.6 java 代码实现sql功能

1前提 es拥有白金版功能

kibana中管理-》许可管理 开启白金版试用

2导入依赖

```
    <dependency>
        <groupId>org.elasticsearch.plugin</groupId>
        <artifactId>x-pack-sql-jdbc</artifactId>
        <version>7.3.0</version>
    </dependency>

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

3代码

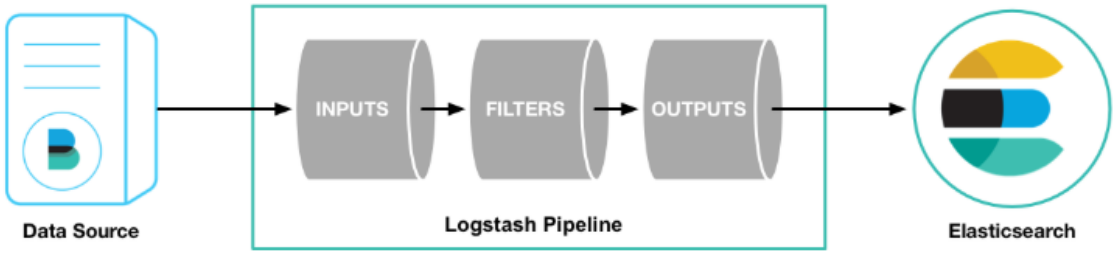
```
public static void main(String[] args) {
        try  {
            Connection connection =
DriverManager.getConnection("jdbc:es://http://localhost:9200");
            Statement statement = connection.createStatement();
```

```
            ResultSet results = statement.executeQuery(
                    "select * from tvs");
            while(results.next()){
                System.out.println(results.getString(1));
                System.out.println(results.getString(2));
                System.out.println(results.getString(3));
                System.out.println(results.getString(4));
                System.out.println("============================");
            }
        }catch (Exception e){
            e.printStackTrace();
        }
```

大型企业可以购买白金版，增加Machine Learning、高级安全性x-pack。

#20． Logstash学习

#20.1Logstash基本语法组成



#1什么是Logstash

logstash是一个数据抽取工具，将数据从一个地方转移到另一个地方。如hadoop生态圈的sqoop等。下载地址： https://www.elastic.co/cn/downloads/logstash

logstash之所以功能强大和流行，还与其丰富的过滤器插件是分不开的，过滤器提供的并不单单是过滤的功能，还可以对进入过滤器的原始数据进行复杂的逻辑处理，甚至添加独特的事件到后续流程中。Logstash配置文件有如下三部分组成，其中input、output部分是必须配置，filter部分是可选配置，而filter就是过滤器插件，可以在这部分实现各种日志过滤功能。

#2配置文件：

```
input {
    #输入插件
}
filter {
    #过滤匹配插件
}
output {
    #输出插件
}
```

#3启动操作：

```
logstash.bat -e 'input{stdin{}} output{stdout{}}'
```

为了好维护，将配置写入文件，启动

```
logstash.bat -f ../config/test1.conf
```

#20.2 Logstash输入插件（input）

https://www.elastic.co/guide/en/logstash/current/input-plugins.html

#1、标准输入(Stdin)

```
input{
    stdin{

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

#2、读取文件(File)

logstash使用一个名为filewatch的ruby gem库来监听文件变化,并通过一个叫.sincedb的数据库文件来记录被监听的日志文件的读取进度（时间戳），这个sincedb数据文件的默认路径在<path.data>/plugins/inputs/file下面，文件名类似于.sincedb_123456，而<path.data>表示logstash插件存储目录，默认是LOGSTASH_HOME/data。

```
input {
    file {
        path => ["/var/*/*"]
        start_position => "beginning"
    }
}
output {
    stdout{
        codec=>rubydebug
    }
}
```

默认情况下，logstash会从文件的结束位置开始读取数据，也就是说logstash进程会以类似tail -f命令的形式逐行获取数据。

#3、读取TCP网络数据

```
input {
  tcp {
    port => "1234"
  }
}
```

```
6
7   filter {
8     grok {
9       match => { "message" => "%{SYSLOGLINE}" }
10    }
11  }
12
13  output {
14      stdout{
15          codec=>rubydebug
16      }
17  }
```

#20.3 Logstash过滤器插件(Filter)

https://www.elastic.co/guide/en/logstash/current/filter-plugins.html

#20.13.1Grok 正则捕获

grok是一个十分强大的logstash filter插件，他可以通过正则解析任意文本，将非结构化日志数据弄成结构化和方便查询的结构。他是目前logstash 中解析非结构化日志数据最好的方式。

Grok 的语法规则是：

```
1   %{语法：语义}
```

例如输入的内容为：

```
1   172.16.213.132 [07/Feb/2019:16:24:19 +0800] "GET / HTTP/1.1" 403 5039
```

%{IP:clientip}匹配模式将获得的结果为：clientip: 172.16.213.132 %{HTTPDATE:timestamp}匹配模式将获得的结果为：timestamp: 07/Feb/2018:16:24:19 +0800 而%{QS:referrer}匹配模式将获得的结果为：referrer: "GET / HTTP/1.1"

下面是一个组合匹配模式，它可以获取上面输入的所有内容：

```
1   %{IP:clientip}\ \[%{HTTPDATE:timestamp}\]\ %{QS:referrer}\ %{NUMBER:response}\ %
    {NUMBER:bytes}
```

通过上面这个组合匹配模式，我们将输入的内容分成了五个部分，即五个字段，将输入内容分割为不同的数据字段，这对于日后解析和查询日志数据非常有用，这正是使用grok的目的。

例子：

```
input{
    stdin{}
}
filter{
    grok{
        match => ["message","%{IP:clientip}\ \[%{HTTPDATE:timestamp}\]\ %{QS:referrer}\ %{NUMBER:response}\ %{NUMBER:bytes}"]
    }
}
output{
    stdout{
        codec => "rubydebug"
    }
}
```

输入内容：

```
172.16.213.132 [07/Feb/2019:16:24:19 +0800] "GET / HTTP/1.1" 403 5039
```

#20.13.2时间处理(Date)

date插件是对于排序事件和回填旧数据尤其重要，它可以用来转换日志记录中的时间字段，变成LogStash::Timestamp对象，然后转存到@timestamp字段里，这在之前已经做过简单的介绍。 下面是date插件的一个配置示例（这里仅仅列出filter部分）：

```
filter {
    grok {
        match => ["message", "%{HTTPDATE:timestamp}"]
    }
    date {
        match => ["timestamp", "dd/MMM/yyyy:HH:mm:ss Z"]
    }
}
```

#20.13.3、数据修改(Mutate)

#（1）正则表达式替换匹配字段

gsub可以通过正则表达式替换字段中匹配到的值，只对字符串字段有效，下面是一个关于mutate插件中gsub的示例（仅列出filter部分）：

```
filter {
    mutate {
        gsub => ["filed_name_1", "/" , "_"]
    }
}
```

这个示例表示将filed_name_1字段中所有"/"字符替换为"_"。

#（2）分隔符分割字符串为数组

split可以通过指定的分隔符分割字段中的字符串为数组，下面是一个关于mutate插件中split的示例（仅列出filter部分）：

```
filter {
    mutate {
        split => ["filed_name_2", "|"]
    }
}
```

这个示例表示将filed_name_2字段以"|"为区间分隔为数组。

（3）重命名字段

rename可以实现重命名某个字段的功能，下面是一个关于mutate插件中rename的示例（仅列出filter部分）：

```
filter {
    mutate {
        rename => { "old_field" => "new_field" }
    }
}
```

这个示例表示将字段old_field重命名为new_field。

（4）删除字段

remove_field可以实现删除某个字段的功能，下面是一个关于mutate插件中remove_field的示例（仅列出filter部分）：

```
filter {
    mutate {
        remove_field  =>  ["timestamp"]
    }
}
```

这个示例表示将字段timestamp删除。

（5）GeoIP 地址查询归类

```
filter {
    geoip {
        source => "ip_field"
    }
}
```

#综合例子：

```
 input {
     stdin {}
 }
 filter {
     grok {
         match => { "message" => "%{IP:clientip}\ \[%{HTTPDATE:timestamp}\]\ %{QS:referrer}\ %{NUMBER:response}\ %{NUMBER:bytes}" }
         remove_field => [ "message" ]
     }
 date {
         match => ["timestamp", "dd/MMM/yyyy:HH:mm:ss Z"]
     }
 mutate {
           convert => [ "response","float" ]
```

```
            rename => { "response" => "response_new" }
            gsub => ["referrer","\"",""]
            split => ["clientip", "."]
        }
    }
    output {
        stdout {
            codec => "rubydebug"
        }
```

#20.4 Logstash输出插件（output）

https://www.elastic.co/guide/en/logstash/current/output-plugins.html

output是Logstash的最后阶段，一个事件可以经过多个输出，而一旦所有输出处理完成，整个事件就执行完成。 一些常用的输出包括：

- file： 表示将日志数据写入磁盘上的文件。
- elasticsearch：表示将日志数据发送给Elasticsearch。Elasticsearch可以高效方便和易于查询的保存数据。

1、输出到标准输出(stdout)

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

2、保存为文件（file）

```
output {
    file {
        path => "/data/log/%{+yyyy-MM-dd}/%{host}_%{+HH}.log"
    }
}
```

3、输出到elasticsearch

```
output {
    elasticsearch {
        host => ["192.168.1.1:9200","172.16.213.77:9200"]
        index => "logstash-%{+YYYY.MM.dd}"
    }
}
```

- host：是一个数组类型的值，后面跟的值是elasticsearch节点的地址与端口，默认端口是9200。可添加多个地址。
- index：写入elasticsearch的索引的名称，这里可以使用变量。Logstash提供了%{+YYYY.MM.dd}这种写法。在语法解析的时候，看到以+ 号开头的，就会自动认为后面是时间格式，尝试用时间格式来解析后续字符串。这种以天为单位分割的写法，可以很容易的删除老的数据或者搜索指定时间范围内的数据。此外，注意索引名中不能有大写字母。
- manage_template:用来设置是否开启logstash自动管理模板功能，如果设置为false将关闭自动管理模板功能。如果我们自定义了模板，那么应该设置为false。
- template_name:这个配置项用来设置在Elasticsearch中模板的名称。

#20.5 综合案例

```
input {
    file {
        path => ["D:/ES/logstash-7.3.0/nginx.log"]
        start_position => "beginning"
    }
}

filter {
    grok {
        match => { "message" => "%{IP:clientip}\ \[%{HTTPDATE:timestamp}\]\ %{QS:referrer}\ %{NUMBER:response}\ %{NUMBER:bytes}" }
        remove_field => [ "message" ]
   }
    date {
        match => ["timestamp", "dd/MMM/yyyy:HH:mm:ss Z"]
    }
    mutate {
           rename => { "response" => "response_new" }
           convert => [ "response","float" ]
           gsub => ["referrer","\"",""]
           remove_field => ["timestamp"]
           split => ["clientip", "."]
        }
}

output {
    stdout {
        codec => "rubydebug"
    }

elasticsearch {
    host => ["localhost:9200"]
    index => "logstash-%{+YYYY.MM.dd}"
}

}
```

#21． kibana学习

#21.1基本查询

1是什么：elk中数据展现工具。

2下载： https://www.elastic.co/cn/downloads/kibana

3使用：建立索引模式，index partten

discover 中使用DSL搜索。

#21.2 可视化

绘制图形

#21.3 仪表盘

将各种可视化图形放入，形成大屏幕。

#21.4 使用模板数据指导绘图

点击主页的添加模板数据，可以看到很多模板数据以及绘图。

#21.5 其他功能

监控，日志，APM等功能非常丰富。

#22． 集群部署

见部署图

#结点的三个角色

主结点：master节点主要用于集群的管理及索引 比如新增结点、分片分配、索引的新增和删除等。 数据结点：data 节点上保存了数据分片，它负责索引和搜索操作。 客户端结点：client 节点仅作为请求客户端存在，client的作用也作为负载均衡器，client 节点不存数据，只是将请求均衡转发到其它结点。

通过下边两项参数来配置结点的功能：

```
node.master: #是否允许为主结点
node.data: #允许存储数据作为数据结点
node.ingest: #是否允许成为协调节点
```

四种组合方式：

```
master=true，data=true：即是主结点又是数据结点
master=false，data=true：仅是数据结点
master=true，data=false：仅是主结点，不存储数据
master=false，data=false：即不是主结点也不是数据结点，此时可设置ingest为true表示它是一个客户端。
```

#23． 项目实战

#23.1项目一： ELK用于日志分析

需求：集中收集分布式服务的日志

1逻辑模块程序随时输出日志

```
package com.ydlclass.es;

```

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.Random;

/**
 * creste by ydlclass.ydl
 */
@SpringBootTest
@RunWith(SpringRunner.class)
public class TestLog {
    private static final Logger LOGGER= LoggerFactory.getLogger(TestLog.class);

    @Test
    public void testLog(){
        Random random =new Random();

        while (true){
            int userid=random.nextInt(10);
            LOGGER.info("userId:{},send:{}",userid,"hello world.I am "+userid);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

2logstash收集日志到es

grok 内置类型

```
USERNAME [a-zA-Z0-9._-]+
USER %{USERNAME}
INT (?:[+-]?(?:[0-9]+))
BASE10NUM (?<![0-9.+-])(?>[+-]?(?:(?:[0-9]+(?:\.[0-9]+)?)|(?:\.[0-9]+)))
NUMBER (?:%{BASE10NUM})
BASE16NUM (?<![0-9A-Fa-f])(?:[+-]?(?:0x)?(?:[0-9A-Fa-f]+))
BASE16FLOAT \b(?<![0-9A-Fa-f.])(?:[+-]?(?:0x)?(?:(?:[0-9A-Fa-f]+(?:\.[0-9A-Fa-f]*)?)|(?:\.[0-9A-Fa-f]+)))\b

POSINT \b(?:[1-9][0-9]*)\b
NONNEGINT \b(?:[0-9]+)\b
WORD \b\w+\b
NOTSPACE \S+
SPACE \s*
DATA .*?
GREEDYDATA .*
QUOTEDSTRING (?>(?<!\\)(?>"(?>\\.|[^\\"]+)+"|""|(?>'(?>\\.|[^\\']+)+')|''|(?>`(?>\\.|[^\\`]+)+`)|``))
UUID [A-Fa-f0-9]{8}-(?:[A-Fa-f0-9]{4}-){3}[A-Fa-f0-9]{12}

# Networking
```

```
MAC (?:%{CISCOMAC}|%{WINDOWSMAC}|%{COMMONMAC})
CISCOMAC (?:(?:[A-Fa-f0-9]{4}\.){2}[A-Fa-f0-9]{4})
WINDOWSMAC (?:(?:[A-Fa-f0-9]{2}-){5}[A-Fa-f0-9]{2})
COMMONMAC (?:(?:[A-Fa-f0-9]{2}:){5}[A-Fa-f0-9]{2})
IPV6 ((([0-9A-Fa-f]{1,4}:){7}([0-9A-Fa-f]{1,4}|:))|(([0-9A-Fa-f]{1,4}:){6}(:[0-9A-Fa-f]{1,4}|((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3})|:))|(([0-9A-Fa-f]{1,4}:){5}(((:[0-9A-Fa-f]{1,4}){1,2})|:((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3})|:))|(([0-9A-Fa-f]{1,4}:){4}(((:[0-9A-Fa-f]{1,4}){1,3})|((:[0-9A-Fa-f]{1,4})?:((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:))|(([0-9A-Fa-f]{1,4}:){3}(((:[0-9A-Fa-f]{1,4}){1,4})|((:[0-9A-Fa-f]{1,4}){0,2}:((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:))|(([0-9A-Fa-f]{1,4}:){2}(((:[0-9A-Fa-f]{1,4}){1,5})|((:[0-9A-Fa-f]{1,4}){0,3}:((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:))|(([0-9A-Fa-f]{1,4}:){1}(((:[0-9A-Fa-f]{1,4}){1,6})|((:[0-9A-Fa-f]{1,4}){0,4}:((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:))|(:(((:[0-9A-Fa-f]{1,4}){1,7})|((:[0-9A-Fa-f]{1,4}){0,5}:((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:)))(%.+)?
IPV4 (?<![0-9])(?:(?:25[0-5]|2[0-4][0-9]|[0-1]?[0-9]{1,2})[.](?:25[0-5]|2[0-4][0-9]|[0-1]?[0-9]{1,2})[.](?:25[0-5]|2[0-4][0-9]|[0-1]?[0-9]{1,2})[.](?:25[0-5]|2[0-4][0-9]|[0-1]?[0-9]{1,2}))(?![0-9])
IP (?:%{IPV6}|%{IPV4})
HOSTNAME \b(?:[0-9A-Za-z][0-9A-Za-z-]{0,62})(?:\.(?:[0-9A-Za-z][0-9A-Za-z-]{0,62}))*(\.?|\b)
HOST %{HOSTNAME}
IPORHOST (?:%{HOSTNAME}|%{IP})
HOSTPORT %{IPORHOST}:%{POSINT}

# paths
PATH (?:%{UNIXPATH}|%{WINPATH})
UNIXPATH (?>/(?>[\w_%!$@:.,-]+|\\.)*)+
TTY (?:/dev/(pts|tty([pq])?)(\w+)?/?(?:[0-9]+))
WINPATH (?>[A-Za-z]+:|\\)(?:\\[^\\?*]*)+
URIPROTO [A-Za-z]+(\+[A-Za-z+]+)?
URIHOST %{IPORHOST}(?::%{POSINT:port})?
# uripath comes loosely from RFC1738, but mostly from what Firefox
# doesn't turn into %XX
URIPATH (?:/[A-Za-z0-9$.+!*'(){},~:;=@#%_\-]*)+
#URIPARAM \?(?:[A-Za-z0-9]+(?:=(?:[^&]*))?(?:&(?:[A-Za-z0-9]+(?:=(?:[^&]*))?)?)*)?
URIPARAM \?[A-Za-z0-9$.+!*'|(){},~@#%&/=:;_?\-\[\]]*
URIPATHPARAM %{URIPATH}(?:%{URIPARAM})?
URI %{URIPROTO}://(?:%{USER}(?::[^@]*)?@)?(?:%{URIHOST})?(?:%{URIPATHPARAM})?

# Months: January, Feb, 3, 03, 12, December
MONTH \b(?:Jan(?:uary)?|Feb(?:ruary)?|Mar(?:ch)?|Apr(?:il)?|May|Jun(?:e)?|Jul(?:y)?|Aug(?:ust)?|Sep(?:tember)?|Oct(?:ober)?|Nov(?:ember)?|Dec(?:ember)?)\b
MONTHNUM (?:0?[1-9]|1[0-2])
MONTHNUM2 (?:0[1-9]|1[0-2])
MONTHDAY (?:(?:0[1-9])|(?:[12][0-9])|(?:3[01])|[1-9])

# Days: Monday, Tue, Thu, etc...
DAY (?:Mon(?:day)?|Tue(?:sday)?|Wed(?:nesday)?|Thu(?:rsday)?|Fri(?:day)?|Sat(?:urday)?|Sun(?:day)?)

# Years?
YEAR (?>\d\d){1,2}
HOUR (?:2[0123]|[01]?[0-9])
```

```
MINUTE (?:[0-5][0-9])
# '60' is a leap second in most time standards and thus is valid.
SECOND (?:(?:[0-5]?[0-9]|60)(?:[:.,][0-9]+)?)
TIME (?!<[0-9])%{HOUR}:%{MINUTE}(?::%{SECOND})(?![0-9])
# datestamp is YYYY/MM/DD-HH:MM:SS.UUUU (or something like it)
DATE_US %{MONTHNUM}[/-]%{MONTHDAY}[/-]%{YEAR}
DATE_EU %{MONTHDAY}[./-]%{MONTHNUM}[./-]%{YEAR}
ISO8601_TIMEZONE (?:Z|[+-]%{HOUR}(?::?%{MINUTE}))
ISO8601_SECOND (?:%{SECOND}|60)
TIMESTAMP_ISO8601 %{YEAR}-%{MONTHNUM}-%{MONTHDAY}[T ]%{HOUR}:?%{MINUTE}(?::?%{SECOND})?%{ISO8601_TIMEZONE}?
DATE %{DATE_US}|%{DATE_EU}
DATESTAMP %{DATE}[- ]%{TIME}
TZ (?:[PMCE][SD]T|UTC)
DATESTAMP_RFC822 %{DAY} %{MONTH} %{MONTHDAY} %{YEAR} %{TIME} %{TZ}
DATESTAMP_RFC2822 %{DAY}, %{MONTHDAY} %{MONTH} %{YEAR} %{TIME} %{ISO8601_TIMEZONE}
DATESTAMP_OTHER %{DAY} %{MONTH} %{MONTHDAY} %{TIME} %{TZ} %{YEAR}
DATESTAMP_EVENTLOG %{YEAR}%{MONTHNUM2}%{MONTHDAY}%{HOUR}%{MINUTE}%{SECOND}

# Syslog Dates: Month Day HH:MM:SS
SYSLOGTIMESTAMP %{MONTH} +%{MONTHDAY} %{TIME}
PROG (?:[\w._/%-]+)
SYSLOGPROG %{PROG:program}(?:\[%{POSINT:pid}\])?
SYSLOGHOST %{IPORHOST}
SYSLOGFACILITY <%{NONNEGINT:facility}.%{NONNEGINT:priority}>
HTTPDATE %{MONTHDAY}/%{MONTH}/%{YEAR}:%{TIME} %{INT}

# Shortcuts
QS %{QUOTEDSTRING}

# Log formats
SYSLOGBASE %{SYSLOGTIMESTAMP:timestamp} (?:%{SYSLOGFACILITY} )?%{SYSLOGHOST:logsource} %{SYSLOGPROG}:
COMMONAPACHELOG %{IPORHOST:clientip} %{USER:ident} %{USER:auth} \[%{HTTPDATE:timestamp}\] "(?:%{WORD:verb} %{NOTSPACE:request}(?: HTTP/%{NUMBER:httpversion})?|%{DATA:rawrequest})" %{NUMBER:response} (?:%{NUMBER:bytes}|-)
COMBINEDAPACHELOG %{COMMONAPACHELOG} %{QS:referrer} %{QS:agent}

# Log Levels
LOGLEVEL ([Aa]lert|ALERT|[Tt]race|TRACE|[Dd]ebug|DEBUG|[Nn]otice|NOTICE|[Ii]nfo|INFO|[Ww]arn?(?:ing)?|WARN?(?:ING)?|[Ee]rr?(?:or)?|ERR?(?:OR)?|[Cc]rit?(?:ical)?|CRIT?(?:ICAL)?|[Ff]atal|FATAL|[Ss]evere|SEVERE|EMERG(?:ENCY)?|[Ee]merg(?:ency)?)
```

写logstash配置文件。

3kibana展现数据

#23.2项目二：学成在线站内搜索模块

#1mysql导入course_pub表

#2创建索引xc_course

#3创建映射

```
PUT /xc_course
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 0
  },
  "mappings": {
    "properties": {
      "description" : {
                 "analyzer" : "ik_max_word",
                 "search_analyzer": "ik_smart",
                "type" : "text"
            },
            "grade" : {
               "type" : "keyword"
            },
            "id" : {
               "type" : "keyword"
            },
            "mt" : {
               "type" : "keyword"
            },
            "name" : {
                "analyzer" : "ik_max_word",
           "search_analyzer": "ik_smart",
               "type" : "text"
            },
            "users" : {
               "index" : false,
               "type" : "text"
            },
            "charge" : {
               "type" : "keyword"
            },
            "valid" : {
               "type" : "keyword"
            },
            "pic" : {
               "index" : false,
               "type" : "keyword"
            },
            "qq" : {
               "index" : false,
               "type" : "keyword"
            },
            "price" : {
               "type" : "float"
            },
            "price_old" : {
               "type" : "float"
```

```
            },
            "st" : {
               "type" : "keyword"
            },
            "status" : {
               "type" : "keyword"
            },
            "studymodel" : {
               "type" : "keyword"
            },
            "teachmode" : {
               "type" : "keyword"
            },
            "teachplan" : {
                "analyzer" : "ik_max_word",
           "search_analyzer": "ik_smart",
               "type" : "text"
            },
           "expires" : {
               "type" : "date",
            "format": "yyyy-MM-dd HH:mm:ss"
            },
            "pub_time" : {
               "type" : "date",
             "format": "yyyy-MM-dd HH:mm:ss"
            },
            "start_time" : {
               "type" : "date",
           "format": "yyyy-MM-dd HH:mm:ss"
            },
          "end_time" : {
                 "type" : "date",
            "format": "yyyy-MM-dd HH:mm:ss"
            }
     }
   }
}
```

#4logstash创建模板文件

Logstash的工作是从MySQL中读取数据，向ES中创建索引，这里需要提前创建mapping的模板文件以便logstash使用。

在logstach的config目录创建xc_course_template.json，内容如下：

```
{
   "mappings" : {
      "doc" : {
         "properties" : {
            "charge" : {
               "type" : "keyword"
            },
            "description" : {
               "analyzer" : "ik_max_word",
               "search_analyzer" : "ik_smart",
               "type" : "text"
            },
```

```
            "end_time" : {
               "format" : "yyyy-MM-dd HH:mm:ss",
               "type" : "date"
            },
            "expires" : {
               "format" : "yyyy-MM-dd HH:mm:ss",
               "type" : "date"
            },
            "grade" : {
               "type" : "keyword"
            },
            "id" : {
               "type" : "keyword"
            },
            "mt" : {
               "type" : "keyword"
            },
            "name" : {
               "analyzer" : "ik_max_word",
               "search_analyzer" : "ik_smart",
               "type" : "text"
            },
            "pic" : {
               "index" : false,
               "type" : "keyword"
            },
            "price" : {
               "type" : "float"
            },
            "price_old" : {
               "type" : "float"
            },
            "pub_time" : {
               "format" : "yyyy-MM-dd HH:mm:ss",
               "type" : "date"
            },
            "qq" : {
               "index" : false,
               "type" : "keyword"
            },
            "st" : {
               "type" : "keyword"
            },
            "start_time" : {
               "format" : "yyyy-MM-dd HH:mm:ss",
               "type" : "date"
            },
            "status" : {
               "type" : "keyword"
            },
            "studymodel" : {
               "type" : "keyword"
            },
            "teachmode" : {
               "type" : "keyword"
            },
            "teachplan" : {
               "analyzer" : "ik_max_word",
```

```
                "search_analyzer" : "ik_smart",
                "type" : "text"
            },
            "users" : {
                "index" : false,
                "type" : "text"
            },
            "valid" : {
                "type" : "keyword"
            }
        }
      }
    },
    "template" : "xc_course"
}
```

#5logstash配置mysql.conf

1、ES采用UTC时区问题

ES采用UTC 时区，比北京时间早8小时，所以ES读取数据时让最后更新时间加8小时

where timestamp > date_add(:sql_last_value,INTERVAL 8 HOUR)

2、logstash每个执行完成会在/config/logstash_metadata记录执行时间下次以此时间为基准进行增量同步数据到索引库。

#6启动

```
.\logstash.bat -f ..\config\mysql.conf
```

7后端代码

7.1Controller

```
@RestController
@RequestMapping("/search/course")
public class EsCourseController  {
    @Autowired
    EsCourseService esCourseService;

    @GetMapping(value="/list/{page}/{size}")
    public QueryResponseResult<CoursePub> list(@PathVariable("page") int page, @PathVariable("size") int size, CourseSearchParam courseSearchParam) {
        return esCourseService.list(page,size,courseSearchParam);
    }

}
```

7.2

```
@Service
public class EsCourseService {
    @Value("${heima.course.source_field}")
    private String source_field;

    @Autowired
    RestHighLevelClient restHighLevelClient;
```

```
8
9       //课程搜索
10      public QueryResponseResult<CoursePub> list(int page, int size,
    CourseSearchParam courseSearchParam) {
11          if (courseSearchParam == null) {
12              courseSearchParam = new CourseSearchParam();
13          }
14          //1创建搜索请求对象
15          SearchRequest searchRequest = new SearchRequest("xc_course");
16
17          SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();
18          //过虑源字段
19          String[] source_field_array = source_field.split(",");
20          searchSourceBuilder.fetchSource(source_field_array, new String[]{});
21          //创建布尔查询对象
22          BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();
23          //搜索条件
24          //根据关键字搜索
25          if (StringUtils.isNotEmpty(courseSearchParam.getKeyword())) {
26              MultiMatchQueryBuilder multiMatchQueryBuilder =
    QueryBuilders.multiMatchQuery(courseSearchParam.getKeyword(), "name",
    "description", "teachplan")
27                      .minimumShouldMatch("70%")
28                      .field("name", 10);
29              boolQueryBuilder.must(multiMatchQueryBuilder);
30          }
31          if (StringUtils.isNotEmpty(courseSearchParam.getMt())) {
32              //根据一级分类
33              boolQueryBuilder.filter(QueryBuilders.termQuery("mt",
    courseSearchParam.getMt()));
34          }
35          if (StringUtils.isNotEmpty(courseSearchParam.getSt())) {
36              //根据二级分类
37              boolQueryBuilder.filter(QueryBuilders.termQuery("st",
    courseSearchParam.getSt()));
38          }
39          if (StringUtils.isNotEmpty(courseSearchParam.getGrade())) {
40              //根据难度等级
41              boolQueryBuilder.filter(QueryBuilders.termQuery("grade",
    courseSearchParam.getGrade()));
42          }
43
44          //设置boolQueryBuilder到searchSourceBuilder
45          searchSourceBuilder.query(boolQueryBuilder);
46          //设置分页参数
47          if (page <= 0) {
48              page = 1;
49          }
50          if (size <= 0) {
51              size = 12;
52          }
53          //起始记录下标
54          int from = (page - 1) * size;
55          searchSourceBuilder.from(from);
56          searchSourceBuilder.size(size);
57
58          //设置高亮
59          HighlightBuilder highlightBuilder = new HighlightBuilder();
```

```
60          highlightBuilder.preTags("<font class='eslight'>");
61          highlightBuilder.postTags("</font>");
62          //设置高亮字段
63  //        <font class='eslight'>node</font>学习
64          highlightBuilder.fields().add(new HighlightBuilder.Field("name"));
65          searchSourceBuilder.highlighter(highlightBuilder);
66
67          searchRequest.source(searchSourceBuilder);
68
69          QueryResult<CoursePub> queryResult = new QueryResult();
70          List<CoursePub> list = new ArrayList<CoursePub>();
71          try {
72              //2执行搜索
73              SearchResponse searchResponse =
    restHighLevelClient.search(searchRequest, RequestOptions.DEFAULT);
74              //3获取响应结果
75              SearchHits hits = searchResponse.getHits();
76              long totalHits=hits.getTotalHits().value;
77              //匹配的总记录数
78  //            long totalHits = hits.totalHits;
79              queryResult.setTotal(totalHits);
80              SearchHit[] searchHits = hits.getHits();
81              for (SearchHit hit : searchHits) {
82                  CoursePub coursePub = new CoursePub();
83                  //源文档
84                  Map<String, Object> sourceAsMap = hit.getSourceAsMap();
85                  //取出id
86                  String id = (String) sourceAsMap.get("id");
87                  coursePub.setId(id);
88                  //取出name
89                  String name = (String) sourceAsMap.get("name");
90                  //取出高亮字段name
91                  Map<String, HighlightField> highlightFields =
    hit.getHighlightFields();
92                  if (highlightFields != null) {
93                      HighlightField highlightFieldName =
    highlightFields.get("name");
94                      if (highlightFieldName != null) {
95                          Text[] fragments = highlightFieldName.fragments();
96                          StringBuffer stringBuffer = new StringBuffer();
97                          for (Text text : fragments) {
98                              stringBuffer.append(text);
99                          }
100                         name = stringBuffer.toString();
101                     }
102                 }
103                 coursePub.setName(name);
104                 //图片
105                 String pic = (String) sourceAsMap.get("pic");
106                 coursePub.setPic(pic);
107                 //价格
108                 Double price = null;
109                 try {
110                     if (sourceAsMap.get("price") != null) {
111                         price = (Double) sourceAsMap.get("price");
112                     }
113
114                 } catch (Exception e) {
```

```
                    e.printStackTrace();
                }
                coursePub.setPrice(price);
                //旧价格
                Double price_old = null;
                try {
                    if (sourceAsMap.get("price_old") != null) {
                        price_old = (Double) sourceAsMap.get("price_old");
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
                coursePub.setPrice_old(price_old);
                //将coursePub对象放入list
                list.add(coursePub);
            }


        } catch (IOException e) {
            e.printStackTrace();
        }

        queryResult.setList(list);
        QueryResponseResult<CoursePub> queryResponseResult = new
QueryResponseResult<CoursePub>(CommonCode.SUCCESS, queryResult);

        return queryResponseResult;
    }


}
```