中文社区：[https://elasticsearch.cn](https://elasticsearch.cn/)

案例：<https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_025_se_elasticsearch>

教程：

SpringBoot集成Elasticsearch7.4 实战1：<https://www.jianshu.com/p/1fbfde2aefa5>

SpringBoot集成Elasticsearch7.4 实战2：<https://www.jianshu.com/p/acc8e86cc772>

SpringBoot集成Elasticsearch7.4 实战3：<https://www.jianshu.com/p/c02e5b412675>

# ElasticStack

ElasticStack 是一系列开源产品的合集，包括 Elasticsearch、Kibana、Logstash 以及 Beats 等，能够安全可靠地获取任何来源、任何格式的数据，并且能够实时地对数据进行搜索、分析和可视化。其中，Logstash和Beats负责数据的收集，Kibana负责结果数据的可视化展现，Elasticsearch作为核心部分用于数据的分布式存储以及索引。

我们可以使用ElasticStack进行收集并处理任何数据，最为经典的是用于实时日志数据的收集、存储、展现。在此之前，如果实现实时日志的分析，需要经过繁琐的架构设计并且使用多个系统实现，现在只需要使用ElasticStack即可实现功能强大的实时日志分析系统，ElasticStack不仅仅用来做日志分析，它可以处理任何的数据并且进行存储、展现。

在本套课程中，我们将全面的讲解ElasticStack技术栈，从环境的部署到技术的应用，再到项目实战，让我们不仅是学习框架技术的使用，而且可以学习到使用ElasticStack如何解决实际的问题，我们通过收集系统日志数据、服务器指标数据、nginx的运行数据等，多维度的展示了ElasticStack的应用范围，最终将分析的结果数据非常友好的展现出来，没错！不需要前端的参与也可以做出“高大上”的结果显示页。

# Elasticsearch

## 基础介绍

Elasticsearch 是一个建立在全文搜索引擎 Apache Lucene(TM) 基础上的搜索引擎，可以说 Lucene 是当今最先进，最高效的全功能开源搜索引擎框架。

Elasticsearch是基于Apache Lucene的搜索服务器。它由Shay Banon开发并于2010年发布。现在是由Elasticsearch BV负责维护。其最新版本是：7.3.2。

Elasticsearch是一个实时分布式和开源的全文搜索和分析引擎。 它可以从RESTful Web服务接口访问，并使用模式少JSON(JavaScript对象符号)文档来存储数据。它是基于Java编程语言，这使Elasticsearch能够在不同的平台上运行。使用户能够以非常快的速度来搜索非常大的数据量。

## Elasticsearch的功能

* 分布式的搜索引擎和数据分析引擎

搜索：百度，网站的站内搜索，IT系统的检索

数据分析：电商网站，最近7天牙膏这种商品销量排名前10的商家有哪些；新闻网站，最近1个月访问量排名前3的新闻版块是哪些

分布式，搜索，数据分析

* 全文检索，结构化检索，数据分析

全文检索：我想搜索商品名称包含牙膏的商品，select \* from products where product\_name like "%牙膏%"

结构化检索：我想搜索商品分类为日化用品的商品都有哪些，select \* from products where category\_id='日化用品'

部分匹配、自动完成、搜索纠错、搜索推荐

数据分析：我们分析每一个商品分类下有多少个商品，select category\_id,count(\*) from products 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

海联数据的处理：分布式以后，就可以采用大量的服务器去存储和检索数据，自然而然就可以实现海量数据的处理了

近实时：检索个数据要花费1小时（这就不要近实时，离线批处理，batch-processing）；在秒级别对数据进行搜索和分析

跟分布式/海量数据相反的：lucene，单机应用，只能在单台服务器上使用，最多只能处理单台服务器可以处理的数据量

## Elasticsearch的主要概念

（1）Near Realtime（NRT）：近实时，两个意思，从写入数据到数据可以被搜索到有一个小延迟（大概1秒）；基于es执行搜索和分析可以达到秒级

（2）Cluster：集群，包含多个节点，每个节点属于哪个集群是通过一个配置（集群名称，默认是elasticsearch）来决定的，对于中小型应用来说，刚开始一个集群就一个节点很正常

（3）Node：节点，集群中的一个节点，节点也有一个名称（默认是随机分配的），节点名称很重要（在执行运维管理操作的时候），默认节点会去加入一个名称为“elasticsearch”的集群，如果直接启动一堆节点，那么它们会自动组成一个elasticsearch集群，当然一个节点也可以组成一个elasticsearch集群

（4）Document&field：文档，es中的最小数据单元，一个document可以是一条客户数据，一条商品分类数据，一条订单数据，通常用JSON数据结构表示，每个index下的type中，都可以去存储多个document。一个document里面有多个field，每个field就是一个数据字段。

product document{

"product\_id": "1",

"product\_name": "高露洁牙膏",

"product\_desc": "高效美白",

"category\_id": "2",

"category\_name": "日化用品"

}

（5）Index：索引，包含一堆有相似结构的文档数据，比如可以有一个客户索引，商品分类索引，订单索引，索引有一个名称。一个index包含很多document，一个index就代表了一类类似的或者相同的document。比如说建立一个product index，商品索引，里面可能就存放了所有的商品数据，所有的商品document。

（6）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，比如博客系统，有一个索引，可以定义用户数据type，博客数据type，评论数据type。

商品index，里面存放了所有的商品数据，商品document

但是商品分很多种类，每个种类的document的field可能不太一样，比如说电器商品，可能还包含一些诸如售后时间范围这样的特殊field；生鲜商品，还包含一些诸如生鲜保质期之类的特殊field

type，日化商品type，电器商品type，生鲜商品type

日化商品type：product\_id，product\_name，product\_desc，category\_id，category\_name

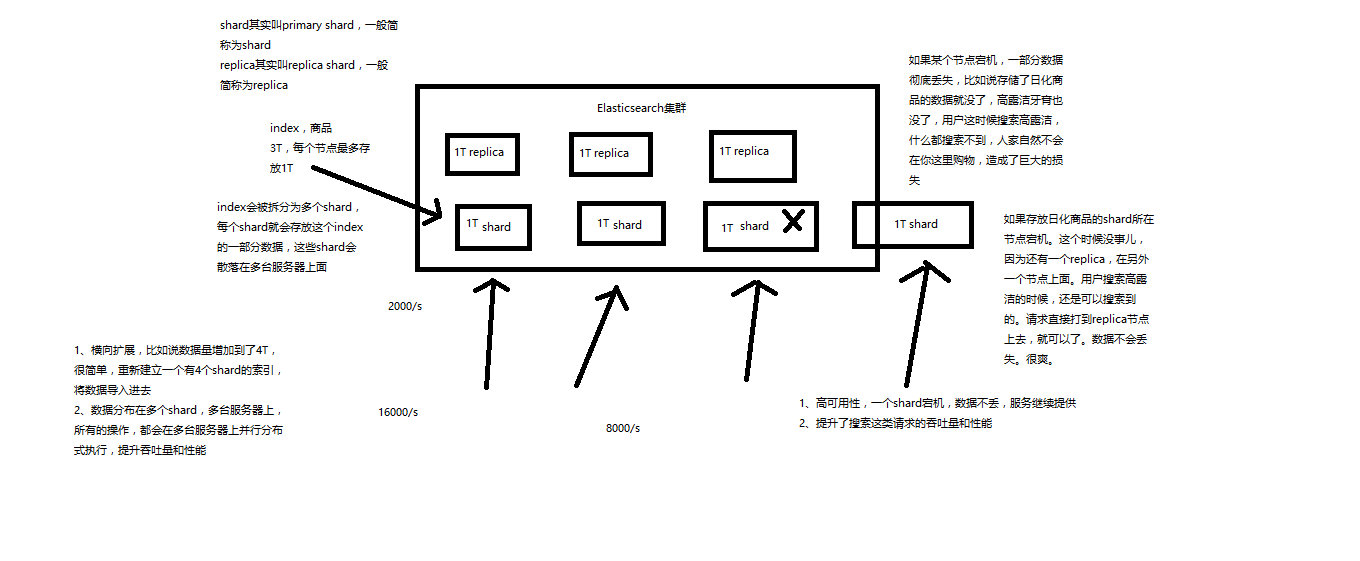
电器商品type：product\_id，product\_name，product\_desc，category\_id，category\_name，service\_period

生鲜商品type：product\_id，product\_name，product\_desc，category\_id，category\_name，eat\_period

每一个type里面，都会包含一堆document

（7）shard：单台机器无法存储大量数据，es可以将一个索引中的数据切分为多个shard，分布在多台服务器上存储。有了shard就可以横向扩展，存储更多数据，让搜索和分析等操作分布到多台服务器上去执行，提升吞吐量和性能。每个shard都是一个lucene index。

（8）replica：任何一个服务器随时可能故障或宕机，此时shard可能就会丢失，因此可以为每个shard创建多个replica副本。replica可以在shard故障时提供备用服务，保证数据不丢失，多个replica还可以提升搜索操作的吞吐量和性能。primary shard（建立索引时一次设置，不能修改，默认5个），replica shard（随时修改数量，默认1个），默认每个索引10个shard，5个primary shard，5个replica shard，最小的高可用配置，是2台服务器。



在Elasticsearch中，索引是类型的集合，因为数据库是RDBMS(关系数据库管理系统)中表的集合。每个表都是行的集合。

|  |  |
| --- | --- |
| Elasticsearch | 关系数据库 |
| 索引Index | 数据库 |
| 类型type | 表 |
| 文档Document | 一行数据 |
| 字段Field | Field |

## Elasticsearch的适用场景

* 国外

（1）维基百科，类似百度百科，牙膏，牙膏的维基百科，全文检索，高亮，搜索推荐

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

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

（4）GitHub（开源代码管理），搜索上千亿行代码

（5）电商网站，检索商品

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

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

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

* 国内

（1）国内：站内搜索（电商，招聘，门户，等等），IT系统搜索（OA，CRM，ERP，等等），数据分析（ES热门的一个使用场景）

## Elasticsearch的特点

（1）可以作为一个大型分布式集群（数百台服务器）技术，处理PB级数据，服务大公司；也可以运行在单机上，服务小公司

（2）Elasticsearch不是什么新技术，主要是将全文检索、数据分析以及分布式技术，合并在了一起，才形成了独一无二的ES；lucene（全文检索），商用的数据分析软件（也是有的），分布式数据库（mycat）

（3）对用户而言，是开箱即用的，非常简单，作为中小型的应用，直接3分钟部署一下ES，就可以作为生产环境的系统来使用了，数据量不大，操作不是太复杂

（4）数据库的功能面对很多领域是不够用的（事务，还有各种联机事务型的操作）；特殊的功能，比如全文检索，同义词处理，相关度排名，复杂数据分析，海量数据的近实时处理；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

## lucene和elasticsearch的前世今生

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（实现一些简单的功能，写大量的java代码），需要深入理解原理（各种索引结构）

elasticsearch，基于lucene，隐藏复杂性，提供简单易用的restful api接口、java api接口（还有其他语言的api接口）

（1）分布式的文档存储引擎

（2）分布式的搜索引擎和分析引擎

（3）分布式，支持PB级数据

开箱即用，优秀的默认参数，不需要任何额外设置，完全开源

## 面向文档型的数据库

（1）应用系统的数据结构都是面向对象的，复杂的

（2）对象数据存储到数据库中，只能拆解开来，变为扁平的多张表，每次查询的时候还得还原回对象格式，相当麻烦

（3）ES是面向文档的，文档中存储的数据结构，与面向对象的数据结构是一样的，基于这种文档数据结构，es可以提供复杂的索引，全文检索，分析聚合等功能

（4）es的document用json数据格式来表达

# 安装

## 下载

下文以最新的版本：7.3.2为例子

高版本的es自带jdk，但是如果发现本地存在Java环境变量的话，会自动选择服务器的环境

下载链接：<https://www.elastic.co/cn/downloads/elasticsearch>

或者再GitHub上：<https://github.com/elastic/elasticsearch/releases>

## 修改配置

config/elasticsearch.yml (注意要顶格写，冒号后面要加一个空格)

* cluster.name: chenkl (这个就是集群的名称，同一集群要一样)
* node.name： node-1 (这个是集群中当前节点的名称，每个节点的名字都应该不一样)
* path.data: D:/java/elasticsearch-7.3.2/data (【可选】设置数据保存的位置，默认为安装目录下)
* path.logs: D:/java/elasticsearch-7.3.2/logs (【可选】设置日志保存的位置，默认为安装目录下)
* network.Host: 127.0.0.1 如果是127.0.0.1的话，就不能被其他机器访问到了
* http.port: 9200 客户端默认的端口是9200
* cluster.initial\_master\_nodes: ["node-1"] (集群默认的节点有哪些1)
* 防止脑裂的配置(5以下的版本需要设置)

discovery.zen.ping.multicast.enabled: false

discovery.zen.ping\_timeout: 120s

client.transport.ping\_timeout: 60s

discovery.zen.ping.unicast.hosts: ["192.168.57.4","192.168.57.5", "192.168.57.6"]

## 启动

# 解压下载后安装包：

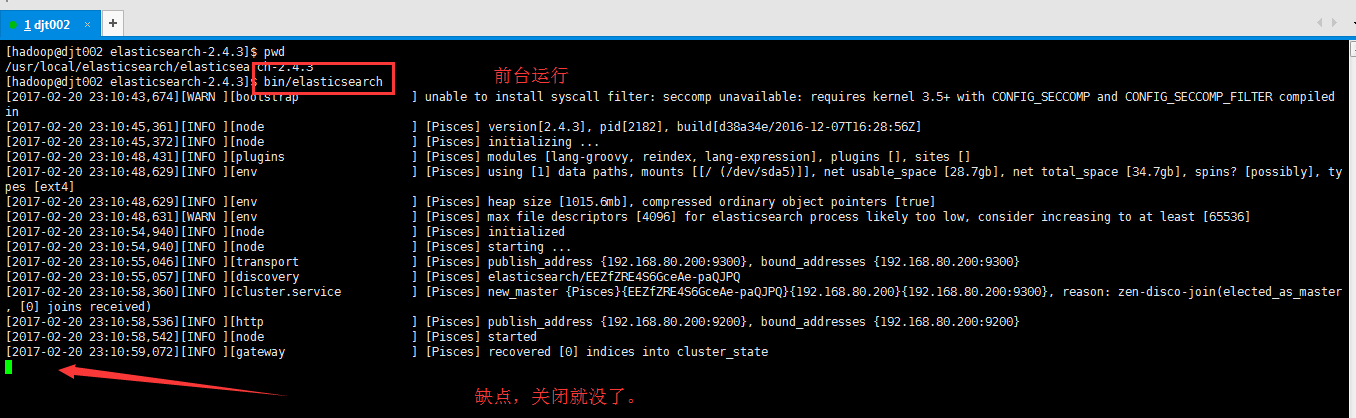
$ tar –xvf elasticsearch-5.2.0.tar.gz

# 进入目录

$ cd elasticsearch-5.1.0/bin

# 执行脚本即可

$ ./elasticsearch



注意 - 在Windows系统中，可能会收到错误，指出未设置JAVA\_HOME，请将环境变量设置为“C:\Program Files\Java\jre1.8.0\_65”或设置实际安装java的位置。

## 配置web界面

Elasticsearch Web界面的默认端口是9200，或者可以通过更改bin目录中的elasticsearch.yml文件中的http.port字段值来更改。可以通过浏览http:// localhost:9200来检查服务器是否已启动并正在运行。

如果没有问题，它将返回一个JSON对象，其中包含有关安装的Elasticsearch信息有以下方式:

{

"name" : "node-1",

"cluster\_name" : "my-application",

"cluster\_uuid" : "ljPbRAQ2RGW9lRMRi8jICg",

"version" : {

"number" : "7.3.2",

"build\_flavor" : "default",

"build\_type" : "zip",

"build\_hash" : "1c1faf1",

"build\_date" : "2019-09-06T14:40:30.409026Z",

"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"

}

## 注意

* 启动报错：can not run elasticsearch as root

ES5之后不允许使用root用户启动了,我们需要添加用户。

adduser \*\*\* //添加用户

passwd \*\*\* //给用户赋值

添加完用户之后：

用root用户执行 ： chown -R 用户名 文件夹名

将这几个压缩包所在的文件夹及解压完的文件夹权限给你新建的用户。之后再使用新用户启动就OK了

* max number of threads is too low

新建（为方便删除）

            /etc/security/limits.d/test-limits.conf

添加

            \* soft nofile 65536

            \* hard nofile 131072

            \* soft nproc 4096

            \* hard nproc 4096

重启服务器（不会重新编译文件，你如果会可以不重启）

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

每个进程最大同时打开文件数太小，可通过下面2个命令查看当前数量

ulimit -Hn

ulimit -Sn

　　修改/etc/security/limits.conf文件，增加配置，用户退出后重新登录生效

\* soft nofile 65536

\* hard nofile 65536

* max number of threads [3818] for user [es] is too low, increase to at least [4096]

问题同上，最大线程个数太低。修改配置文件/etc/security/limits.conf（和问题1是一个文件），增加配置

\* soft nproc 4096

\* hard nproc 4096

可通过命令查看

ulimit -Hu

ulimit -Su

* max virtual memory areas vm.max\_map\_count [65530] is too low, increase to at least [262144]

修改/etc/sysctl.conf文件，增加配置vm.max\_map\_count=262144

vi /etc/sysctl.conf

sysctl -p

* Exception in thread "main" java.nio.file.AccessDeniedException: /usr/local/elasticsearch/elasticsearch-6.2.2-1/

config/jvm.options

elasticsearch用户没有该文件夹的权限，执行命令

chown -R es:es /usr/local/elasticsearch/

## 基本命令

启动 ./elasticsearch 退出： ctrl+c

后台启动 ./elasticsearch -d 退出： kill -9 pid

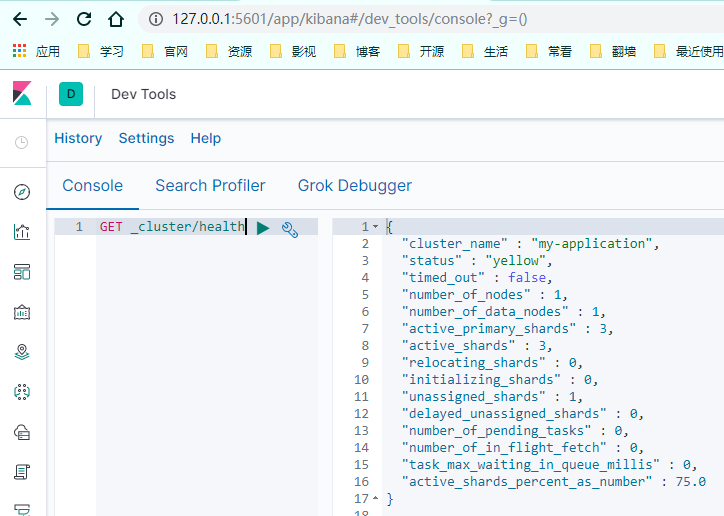
## 使用kibana

下载和解压缩Kibana安装包，使用里面的开发界面，去操作elasticsearch，作为我们学习es知识点的一个主要的界面入口。这里我们下载的是与es相同的版本7.3.2

启动Kibana：bin\kibana.bat

进入Dev Tools界面，浏览器输入 127.0.0.1:5601

GET \_cluster/health



# API操作

下面使用web界面进行基本crud操作。web的客户端工具为kibana。

下面的基本操作是以一个简单的案例展开的。

案例：

有一个电商网站，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

## 简单的集群管理

### 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

1488006741 15:12:21 elasticsearch yellow 1 1 1 1 0 0 1 0 - 50.0%

epoch timestamp cluster status node.total node.data shards pri relo init unassign pending\_tasks max\_task\_wait\_time active\_shards\_percent

1488007113 15:18:33 elasticsearch green 2 2 2 1 0 0 0 0 - 100.0%

epoch timestamp cluster status node.total node.data shards pri relo init unassign pending\_tasks max\_task\_wait\_time active\_shards\_percent

1488007216 15:20:16 elasticsearch yellow 1 1 1 1 0 0 1 0 - 50.0%

* 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

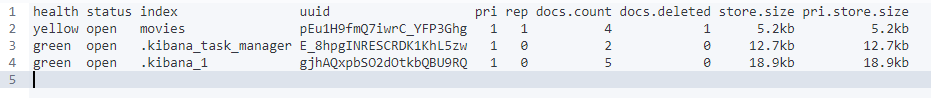
* 为什么现在会处于一个yellow状态？

我们现在就一个笔记本电脑，就启动了一个es进程，相当于就只有一个node。现在es中有一个index，就是kibana自己内置建立的index。由于默认的配置是给每个index分配5个primary shard和5个replica shard，而且primary shard和replica shard不能在同一台机器上（为了容错）。现在kibana自己建立的index是1个primary shard和1个replica shard。当前就一个node，所以只有1个primary shard被分配了和启动了，但是一个replica shard没有第二台机器去启动。

此时只要启动第二个es进程，就会在es集群中有2个node，然后那1个replica shard就会自动分配过去，然后cluster status就会变成green状态。

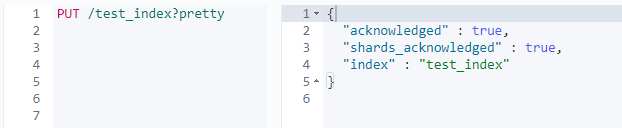
### 快速查看集群中有哪些索引

GET /\_cat/indices?v



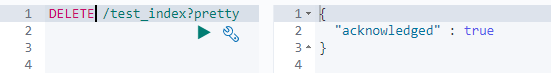
### 创建索引

PUT /test\_index?pretty



### 删除索引

DELETE /test\_index?pretty



## 基本CRUD

ES为用户提供了基本的restful风格的http api。基本的格式如下：http://localhost:9200/<位置1>/<位置2>/[<位置3>]。

* 位置1，是索引index，URL中必选的一个。
* 位置2，
* \_doc, \_create, 主要为了不指定文档的type。在较高的版本中，es将不建议使用type
* 类型type。主要用于指定类型
* 位置3，就是ID，可选，如果不传id，es会帮我们创建一个id

下面的部分案例将不再使用type

### Es为什么要取消type的概念

我们一直认为ES中的“index”类似于关系型数据库的“database”，而“type”相当于一个数据表。ES的开发者们认为这是一个糟糕的认识。例如：关系型数据库中两个数据表示是独立的，即使他们里面有相同名称的列也不影响使用，但ES中不是这样的。

我们都知道elasticsearch是基于Lucene开发的搜索引擎，而ES中不同type下名称相同的filed最终在Lucene中的处理方式是一样的。举个例子，两个不同type下的两个user\_name，在ES同一个索引下其实被认为是同一个filed，你必须在两个不同的type中定义相同的filed映射。否则，不同type中的相同字段名称就会在处理中出现冲突的情况，导致Lucene处理效率下降。

去掉type能够使数据存储在独立的index中，这样即使有相同的字段名称也不会出现冲突，就像ElasticSearch出现的第一句话一样“你知道的，为了搜索····”，去掉type就是为了提高ES处理数据的效率。

除此之外，在同一个索引的不同type下存储字段数不一样的实体会导致存储中出现稀疏数据，影响Lucene压缩文档的能力，导致ES查询效率的降低。

### 新增商品：新增文档，建立索引

新增文档的时候提交文档的类型应该是PUT。使用post也可以，但是含义就不同了。

es会自动建立index和type，不需要提前创建，而且es默认会对document每个field都建立倒排索引，让其可以被搜索

PUT /ecommerce/product/1

{

"name" : "高露洁牙垢",

"desc" : "高效 美白",

"price" : 30,

"producer" : "高露洁有限公司",

"tags": [ "牙膏", "美白", "防蛀" ]

}

PUT /ecommerce/product/2

{

"name" : "黑人牙膏",

"desc" : "高效 美白",

"price" : 11,

"producer" : "黑人有限公司",

"tags": [ "牙膏"]

}

PUT /ecommerce/product/3

{

"name" : "中华牙膏",

"desc" : "防蛀 美白",

"price" : 45,

"producer" : "中华有限公司",

"tags": [ "牙膏", "防蛀" ]

}



* 强制创建

创建文档与全量替换的语法是一样的，有时我们只是想新建文档，不想替换文档，如果强制进行创建呢

PUT /index/type/id?op\_type=create，PUT /index/type/id/\_create

### 查询商品：检索文档

基本格式GET /index/type/id

GET /ecommerce/product/1



### 修改商品：替换文档

同新增一个，也是使用PUT，这样es去进行的是替换（删除旧的，生成新的document）。

es会将老的document标记为deleted，然后新增我们给定的一个document，当我们创建越来越多的document的时候，es会在适当的时机在后台自动删除标记为deleted的document

PUT /ecommerce/product/1

{

"name" : "jiaqiangban gaolujie yagao",

"desc" : "gaoxiao meibai",

"price" : 30,

"producer" : "gaolujie producer",

"tags": [ "meibai", "fangzhu" ]

}

### 修改商品：更新文档

使用post提交修改，可以只传入需要修改的字段即可

POST /ecommerce/product/1/\_update

{

"doc": {

"name": "jiaqiangban gaolujie yagao"

}

}

### 删除商品：删除文档

DELETE /ecommerce/product/1

不会理解物理删除，只会将其标记为deleted，当数据越来越多的时候，在后台自动删除

## 多种搜索方式

### query string search

* 搜索全部商品

GET /ecommerce/product/\_search

took：耗费了几毫秒

timed\_out：是否超时，这里是没有

\_shards：数据拆成了5个分片，所以对于搜索请求，会打到所有的primary shard（或者是它的某个replica shard也可以）

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的详细数据

{

"took" : 6,

"timed\_out" : false,

"\_shards" : {

"total" : 1,

"successful" : 1,

"skipped" : 0,

"failed" : 0

},

"hits" : {

"total" : {

"value" : 3,

"relation" : "eq"

},

"max\_score" : 1.0,

"hits" : [

{

"\_index" : "ecommerce",

"\_type" : "product",

"\_id" : "3",

"\_score" : 1.0,

"\_source" : {

"name" : "中华牙膏",

"desc" : "防蛀 美白",

"price" : 30,

"producer" : "中华有限公司",

"tags" : [

"牙膏",

"防蛀"

]

}

},

{

"\_index" : "ecommerce",

"\_type" : "product",

"\_id" : "2",

"\_score" : 1.0,

"\_source" : {

"name" : "黑人牙膏",

"desc" : "高效 美白",

"price" : 30,

"producer" : "黑人有限公司",

"tags" : [

"牙膏"

]

}

},

{

"\_index" : "ecommerce",

"\_type" : "product",

"\_id" : "1",

"\_score" : 1.0,

"\_source" : {

"name" : "高露洁牙垢",

"desc" : "高效 美白",

"price" : 30,

"producer" : "高露洁有限公司",

"tags" : [

"牙膏",

"美白",

"防蛀"

]

}

}

]

}

}

query string search的由来，因为search参数都是以http请求的query string来附带的

搜索商品名称中包含药膏的商品，而且按照售价降序排序

GET /ecommerce/product/\_search?q=name:药膏&sort=price:asc

适用于临时的在命令行使用一些工具，比如curl，快速的发出请求，来检索想要的信息；但是如果查询请求很复杂，是很难去构建的

在生产环境中，几乎很少使用query string search

### query DSL

DSL : domian specified language特定领域的语言

http request body：请求体，可以用json的格式来构建查询语法，比较方便，可以构建各种复杂的语法，比query string search肯定强大多了

* 查询全部商品

GET /ecommerce/\_search

{

"query" : {

"match\_all": {}

}

}

* 查询所有的牙膏，并按照加个降序排序

GET /ecommerce/\_search

{

"query" : {

"match": {

"name" : "牙膏"

}

},

"sort": [

{

"price": {

"order": "desc"

}

}

]

}

* 分页查询

使用from和size表示分页信息

总共3条商品，假设每页就显示1条商品，现在显示第2页，所以就查出来第2个商品

GET /ecommerce/\_search

{

"query" : {

"match\_all": {}

},

"from": 2,

"size": 1

}

* 指定要查询出来商品的名称和价格就可以

GET /ecommerce/\_search

{

"query" : {

"match\_all": {}

},

"\_source": [

"name", "price"

]

}

### query filter

bool为使用filter的条件。

* must是需要匹配的条件，可以由多个match组成
* Filter是对上面的match的结果进行过滤的条件。

range是filter的查询范围： gt=大于 gte=大于等于 lt=小于 lte=小于等于

GET /ecommerce/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"name": "牙膏"

}

}

],

"filter": {

"range": {

"price": {

"gt": 30

}

}

}

}

}

}

### full-text search

全文检索

GET /ecommerce/\_search

{

"query": {

"match": {

"desc": "防蛀 美白"

}

}

}

desc这个字段，会先被拆解，建立倒排索引

查询条件其实被拆解为两个，一个是“防蛀”，另一个是“美白”，下面我们新增一个产品。

PUT /ecommerce/\_doc/4

{

"name" : "日本牙膏",

"desc" : "防蛀",

"price" : 45,

"producer" : "XXXXX",

"tags" : [

"牙膏",

"防蛀"

]

}

看下结果查询结果。我们发现与拆解的条件符合的越多的，结果越在前面

{

"took" : 2,

"timed\_out" : false,

"\_shards" : {

"total" : 1,

"successful" : 1,

"skipped" : 0,

"failed" : 0

},

"hits" : {

"total" : {

"value" : 4,

"relation" : "eq"

},

"max\_score" : 1.9837129,

"hits" : [

{

"\_index" : "ecommerce",

"\_type" : "product",

"\_id" : "3",

"\_score" : 1.9837129,

"\_source" : {

"name" : "中华牙膏",

"desc" : "防蛀 美白",

"price" : 45,

"producer" : "中华有限公司",

"tags" : [

"牙膏",

"防蛀"

]

}

},

{

"\_index" : "ecommerce",

"\_type" : "product",

"\_id" : "4",

"\_score" : 1.6810182,

"\_source" : {

"name" : "日本牙膏",

"desc" : "防蛀",

"price" : 45,

"producer" : "XXXXX",

"tags" : [

"牙膏",

"防蛀"

]

}

},

{

"\_index" : "ecommerce",

"\_type" : "product",

"\_id" : "2",

"\_score" : 0.6739625,

"\_source" : {

"name" : "黑人牙膏",

"desc" : "高效 美白",

"price" : 11,

"producer" : "黑人有限公司",

"tags" : [

"牙膏"

]

}

},

{

"\_index" : "ecommerce",

"\_type" : "product",

"\_id" : "1",

"\_score" : 0.6739625,

"\_source" : {

"name" : "高露洁牙垢",

"desc" : "高效 美白",

"price" : 30,

"producer" : "高露洁有限公司",

"tags" : [

"牙膏",

"美白",

"防蛀"

]

}

}

]

}

}

### phrase search

短语搜索，与全文检索是相对的。

跟全文检索相对应，相反，全文检索会将输入的搜索串拆解开来，去倒排索引里面去一一匹配，只要能匹配上任意一个拆解后的单词，就可以作为结果返回

phrase search，要求输入的搜索串，必须在指定的字段文本中，完全包含一模一样的，才可以算匹配，才能作为结果返回

GET /ecommerce/\_search

{

"query": {

"match\_phrase": {

"desc": "防蛀 美白"

}

}

}

查询结果如下，其实只能查询一条数据

{

"took" : 49,

"timed\_out" : false,

"\_shards" : {

"total" : 1,

"successful" : 1,

"skipped" : 0,

"failed" : 0

},

"hits" : {

"total" : {

"value" : 1,

"relation" : "eq"

},

"max\_score" : 1.9837129,

"hits" : [

{

"\_index" : "ecommerce",

"\_type" : "product",

"\_id" : "3",

"\_score" : 1.9837129,

"\_source" : {

"name" : "中华牙膏",

"desc" : "防蛀 美白",

"price" : 45,

"producer" : "中华有限公司",

"tags" : [

"牙膏",

"防蛀"

]

}

}

]

}

}

### highlight search

高亮搜索结果

GET /ecommerce/\_search

{

"query": {

"match": {

"producer": "高露洁"

}

},

"highlight": {

"fields": {

"producer": {}

}

}

}

查询结果如下:

{

"took" : 5,

"timed\_out" : false,

"\_shards" : {

"total" : 1,

"successful" : 1,

"skipped" : 0,

"failed" : 0

},

"hits" : {

"total" : {

"value" : 1,

"relation" : "eq"

},

"max\_score" : 3.1039925,

"hits" : [

{

"\_index" : "ecommerce",

"\_type" : "product",

"\_id" : "1",

"\_score" : 3.1039925,

"\_source" : {

"name" : "高露洁牙垢",

"desc" : "高效 美白",

"price" : 30,

"producer" : "高露洁有限公司",

"tags" : [

"牙膏",

"美白",

"防蛀"

]

},

"highlight" : {

"producer" : [

"<em>高</em><em>露</em><em>洁</em>有限公司"

]

}

}

]

}

}

## 聚合分析

### 分组聚合求和

统计每个tag下商品的数量，size是返回被统计的数据有哪些数据。正常是不需要看的，所有设置size=0即可

GET /ecommerce/\_search

{

"aggs": {

"group\_by\_tags": {

"terms": {

"field": "tags"

}

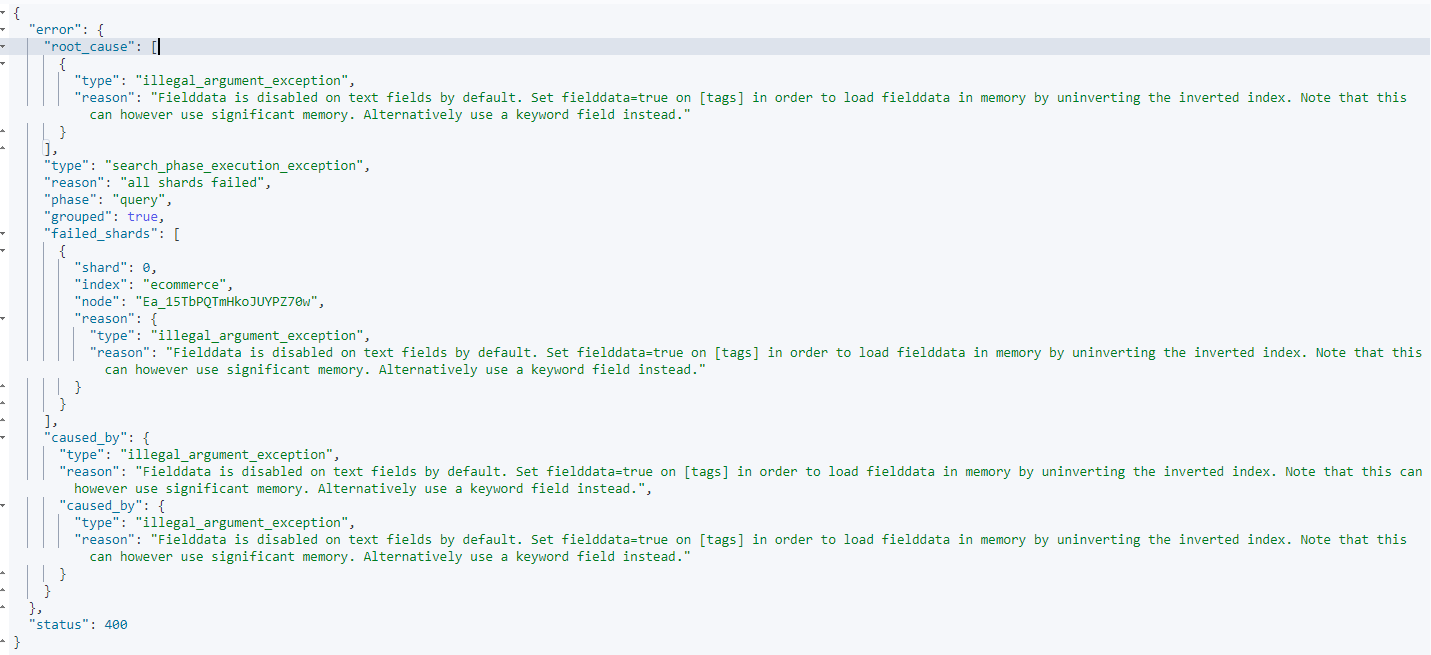
}

},

"size": 0

}

第一次查询的时候，应该是会出现异常。



将文本field的fielddata属性设置为true

PUT /ecommerce/\_mapping/product

{

"properties": {

"tags": {

"type": "text",

"fielddata": true

}

}

}

下面是因为分词器的问题，所以将汉字一个一个的拆分开了去统计的，但是也可以看出

{

"took" : 0,

"timed\_out" : false,

"\_shards" : {

"total" : 1,

"successful" : 1,

"skipped" : 0,

"failed" : 0

},

"hits" : {

"total" : {

"value" : 4,

"relation" : "eq"

},

"max\_score" : null,

"hits" : [ ]

},

"aggregations" : {

"group\_by\_tags" : {

"doc\_count\_error\_upper\_bound" : 0,

"sum\_other\_doc\_count" : 0,

"buckets" : [

{

"key" : "牙",

"doc\_count" : 4

},

{

"key" : "膏",

"doc\_count" : 4

},

{

"key" : "蛀",

"doc\_count" : 3

},

{

"key" : "防",

"doc\_count" : 3

},

{

"key" : "白",

"doc\_count" : 1

},

{

"key" : "美",

"doc\_count" : 1

}

]

}

}

}

### 查询聚合求和

对名称中包含高露洁的商品，计算每个tag下的商品数量，

就是在上面分组聚合的查询条件，加一个query就行了

GET /ecommerce/\_search

{

"query": {

"match": {

"name": "高露洁"

}

},

"aggs": {

"group\_by\_tags": {

"terms": {

"field": "tags"

}

}

},

"size": 20

}

### 分组求均值

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

这里其实是一个嵌套分组聚合的方式，第一个使用了terms函数，之后使用avg函数对第一个的结果进行均值

GET /ecommerce/\_search

{

"aggs": {

"group\_by\_tags": {

"terms": {

"field": "tags"

},

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

}

},

"size": 0

}

### 分组求均值排序

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 /ecommerce/\_search

{

"aggs": {

"group\_by\_tags": {

"terms": {

"field": "tags",

"order": {

"avg\_price": "desc"

}

},

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

}

},

"size": 0

}

### 按条件分组

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

其实就是嵌套三层去分组聚合

GET /ecommerce/\_search

{

"size": 0,

"aggs": {

"group\_by\_price": {

"range": {

"field": "price",

"ranges": [

{

"from": 0,

"to": 20

},

{

"from": 20,

"to": 40

},

{

"from": 40,

"to": 50

}

]

},

"aggs": {

"group\_by\_tags": {

"terms": {

"field": "tags"

},

"aggs": {

"average\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

}

}

# 基础分布式架构

## 核心元数据

我们已经知道es数据查询的数据结构如下：

"\_index" : "ecommerce",

"\_type" : "product",

"\_id" : "1",

"\_score" : 3.1039925,

"\_source" : {

"name" : "高露洁牙垢",

"desc" : "高效 美白",

"price" : 30,

"producer" : "高露洁有限公司",

"tags" : [

"牙膏", "美白", "防蛀"

]

}

### \_index元数据

（1）代表一个document存放在哪个index中

（2）类似的数据放在一个索引，非类似的数据放不同索引：product index（包含了所有的商品），sales index（包含了所有的商品销售数据），inventory index（包含了所有库存相关的数据）。如果你把比如product，sales，human resource（employee），全都放在一个大的index里面，比如说company index，不合适的。

（3）index中包含了很多类似的document：类似是什么意思，其实指的就是说，这些document的fields很大一部分是相同的，你说你放了3个document，每个document的fields都完全不一样，这就不是类似了，就不太适合放到一个index里面去了。

（4）索引名称必须是小写的，不能用下划线开头，不能包含逗号：product，website，blog

### \_type元数据

注意：在最新的es版本中已经不推荐使用type了，因为type会影响index的shard分布。

（1）代表document属于index中的哪个类别（type）

（2）一个索引通常会划分为多个type，逻辑上对index中有些许不同的几类数据进行分类：因为一批相同的数据，可能有很多相同的fields，但是还是可能会有一些轻微的不同，可能会有少数fields是不一样的，举个例子，就比如说，商品，可能划分为电子商品，生鲜商品，日化商品，等等。

（3）type名称可以是大写或者小写，但是同时不能用下划线开头，不能包含逗号

### \_id元数据

代表document的唯一标识，与index和type一起，可以唯一标识和定位一个document

我们可以手动指定document的id（put /index/type/id），也可以不指定，由es自动为我们创建一个id

* 手动指定document id

一般来说，是从某些其他的系统中，导入一些数据到es时，会采取这种方式，就是使用系统中已有数据的唯一标识，作为es中document的id。

* 自动生成document id

数据产生出来以后，可能就没有id，直接就放es一个存储，那么这个时候，可能就不太适合说手动指定document id的形式了，因为你也不知道id应该是什么，此时可以采取下面要讲解的让es自动生成id的方式。

自动生成的id，长度为20个字符，URL安全，base64编码，GUID，分布式系统并行生成时不可能会发生冲突

### \_source元数据

\_source元数据：在创建一个document的时候保存的数据，默认情况下，在get的时候，会原封不动的给我们返回回来。

定制返回的结果，指定\_source中，返回哪些field： GET /test\_index/test\_type/1?\_source=test\_field1,test\_field2

## 复杂分布式机制的透明隐藏特性

Elasticsearch是一套分布式的系统，分布式是为了应对大数据量隐藏了复杂的分布式机制

* 分片机制

我们可以简单的将一个document插入到es集群中去了，我们不用考虑过数据怎么进行分片的，数据被分配到那个shard中去

* cluster discovery

集群发现机制，我们之前在做那个集群status从yellow转green的实验里，直接启动了第二个es进程，那个进程作为一个node自动就发现了集群，并且加入了进去，还接受了部分数据，replica shard

* shard负载均衡

举例，假设现在有3个节点，总共有25个shard要分配到3个节点上去，es会自动进行均匀分配，以保持每个节点的均衡的读写负载请求

* shard副本

请求路由，集群扩容，shard重分配

### 垂直扩容与水平扩容

垂直扩容：采购更强大的服务器，成本非常高昂，而且会有瓶颈，假设世界上最强大的服务器容量就是10T，但是当你的总数据量达到5000T的时候，你要采购多少台最强大的服务器啊

水平扩容：业界经常采用的方案，采购越来越多的普通服务器，性能比较一般，但是很多普通服务器组织在一起，就能构成强大的计算和存储能力。一般在生产中会使用水平扩容。扩容对应用程序的透明性。

（1）primary&replica自动负载均衡，6个shard，3 primary，3 replica

（2）每个node有更少的shard，IO/CPU/Memory资源给每个shard分配更多，每个shard性能更好

（3）扩容的极限，6个shard（3 primary，3 replica），最多扩容到6台机器，每个shard可以占用单台服务器的所有资源，性能最好

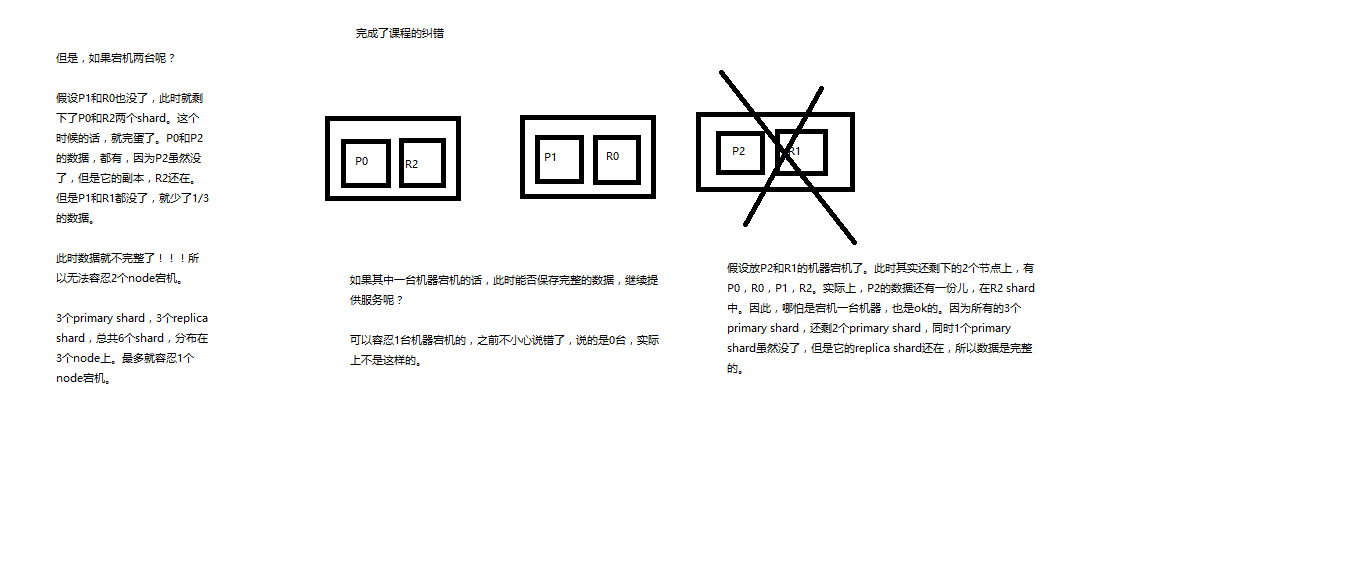
（4）超出扩容极限，动态修改replica数量，9个shard（3primary，6 replica），扩容到9台机器，比3台机器时，拥有3倍的读吞吐量

（5）3台机器下，

9个shard（3 primary，6 replica），资源更少，但是容错性更好，最多容纳2台机器宕机，

6个shard（3 primary，3 replica）只能容纳1台机器宕机。

（6）这里的这些知识点，你综合起来看，就是说，一方面告诉你扩容的原理，怎么扩容，怎么提升系统整体吞吐量；另一方面要考虑到系统的容错性，怎么保证提高容错性，让尽可能多的服务器宕机，保证数据不丢失



### rebalance

增减或减少节点时的数据rebalance。保持负载均衡，以确保，每个节点上数据量与请求量是均衡的。

### master节点

Maser节点不承担所有的请求。Master节点的主要作用

* 管理集群的元数据：比如，索引的创建与删除，维护索引元数据，节点的增加和迁移，维护集群的元数据
* 默认情况下，会自动选择出一台节点，作为master节点

### 节点平等的分布式架构

（1）节点对等，每个节点都能接收所有的请求

（2）自动请求路由

（3）响应收集

## shard&replica机制

### shard&replica基本概念

* 一个index包含多个shard

比如一个index数据量是3T，es可能会创建出3个shard，每个shard保存1T的数据。

* Shard是es的最小工作单元，能够承载部分index的数据，一个shard其实就是一个Lucene实例，具备完整的创建索引和处理请求的能力
* 增减节点的时候，shard会自动在node节点中进行负载均衡

比如：原有3个node上有7个shard，node1有3个shard， node2 node3各有2个shard，这个时候添加了一个节点，es集群就会把node1上的一个shard转移到新增的node4上

* 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。

也就是说一个document有一个副本。

* replica shard是primary shard的副本

负责容错，以及承担读请求负载。

也就是说，一个primary shard只能对应一个replica shard。但是多个primary shard可能都对应到一个replica shard上。

* 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

primary shard的默认数量是5，replica默认是1，默认有10个shard（10个是默认最大的shard的数量）

刚开始创建索引的时候，会直接创建出5个primary shard与1个replica shard。但磁盘不够的时候，可以对replica shard扩容到5个。

* 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。

### 单个node环境shard和replica分配

（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

设置一个索引的 shard 与 replica的数量

PUT /test\_index

{

"settings" : {

"number\_of\_shards" : 3,

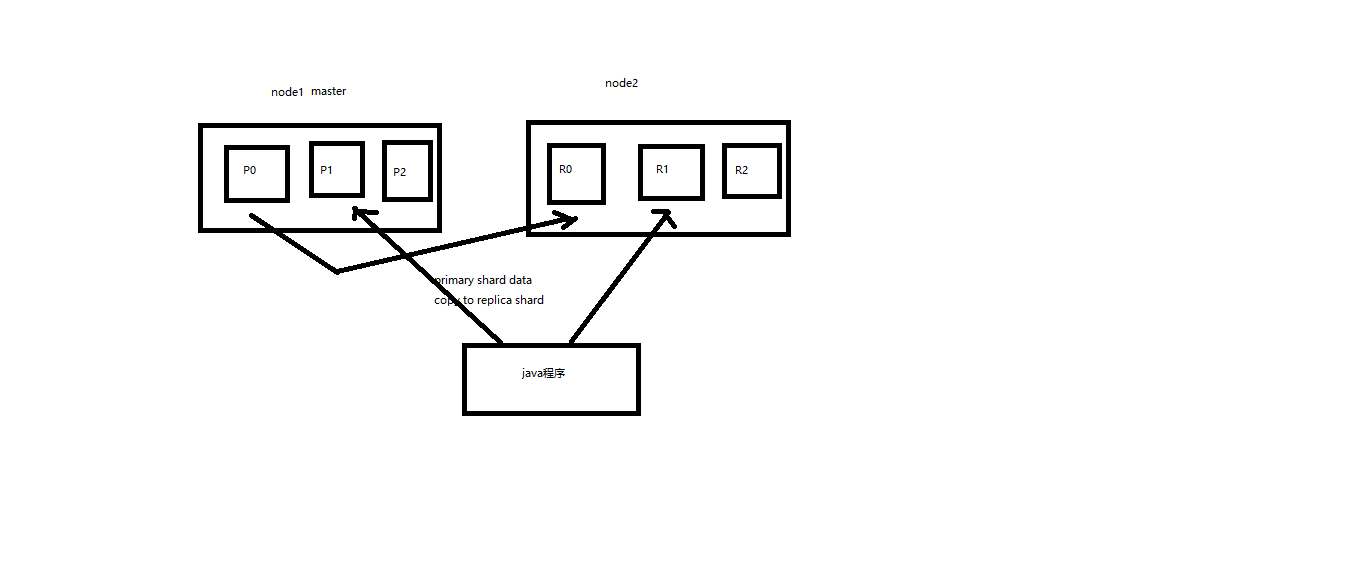
"number\_of\_replicas" : 1

}

}

### 两个node环境shard和replica分配

会创建出三个replica，数据会被同步到第二个节点。



## 容错机制

假设现有3个node，有9个shard, 3个primary shard, 6个 replica shard。1个shard有2个replica。主要分为三步：master选举，replica容错，数据恢复。

三个node上的shard分布分别是 master(p0, r1\_1, r2\_2), cluster1( r0\_2, p1, r2\_1), cluster2(r0\_1, r1\_2, p2)

* 这个时候master节点宕机了，集群的状态由green变成了red集群开始自动选举某个cluster作为master，变成下面的样子是 master( r0\_2, p1, r2\_1), cluster2(r0\_1, r1\_2, p2)。因为replica并不全是active状态的，所有状态还是red
* replica容错：新master将replica提升为primary shard，这个时候还有一个replica没有active，所有集群是yellow状态
* 重启宕机node，新的master copy replica到该node，使用原有的shard并同步宕机后的修改，集群变成green状态

# 最佳实践

## Index与shard的设计

Index与Shard，分别对应了Elasticsearch的两种数据组织方式：逻辑组织和物理组织。逻辑层面上，Index与业务数据的结构、类型、使用方式等息息相关；而物理层面上，Shard关系到数据在不同机器上的分布情况。

### 基于时间的Index设计

Index设计时要考虑的第一件事，就是基于时间对Index进行分割，即每隔一段时间产生一个新的Index。为什么要这样做呢？因为现实世界的数据是随着时间的变化而不断产生的，切分管理可以获得足够的灵活性和更好的性能。

* 如果数据都存储在一个Index中，很难进行扩展和调整，因为Elasticsearch中Index的某些设置在创建时就设定好了，是不能更改的，比如Primary Shard的个数。而根据时间来切分Index，则可以实现一定的灵活性，既可以在数据量过大时及时调整Shard个数，也可以及时响应新的业务需求。
* 大多数业务场景下，客户对数据的请求都会命中在最近一段时间上，通过切分Index，可以尽可能的避免扫描不必要的数据，提高性能。

当然，并不排除某些特定的业务场景下，不用对Index进行切分管理，比如一个固定的数据集或者一个增长非常缓慢的数据集。大多数情况下，笔者都建议按照时间进行分割。那么，要考虑的第二件事就是，按照什么规则来设定切分的间隔呢？根据上面的分析，自然是时间越短越能保持灵活性，但是这样做就会导致产生大量的Index，而每个Index都会消耗资源来维护其元信息的，因此需要在灵活性、资源和性能上做权衡。建议按照如下几点来思考：

* 常见的间隔有小时、天、周和月。先考虑总共要存储多久的数据，然后选一个既不会产生大量Index又能够满足一定灵活性的间隔。比如，你需要存储6个月的数据，那么一开始选择“周”这个间隔就会比较合适。
* 考虑业务增长速度。假如业务增长的特别快，比如上周产生了1亿数据，这周就增长到了10亿，那么就需要调低这个间隔来保证有足够的弹性能应对变化。
* 结合业务特性和性能测试来决定要不要调整间隔。这点更多的是从业务角度来考虑的，举个例子，在笔者的一个项目中，一开始选择了“周”作为间隔，一周产生一个新的Index来存储实时的分钟级数据，但是每周会将历史数据合并成小时级数据来降低数据量、提高查询速度。在性能测试中发现，合并后的查询性能相比合并前提升特别大，因此我们将整体间隔调整到“天”来缩短合并的周期。（后面会另撰文章更详细的分享这个案例）

接下来就是第三件事，如何实现Index的分割？切分行为是由客户端（数据的写入端）发起的，根据时间间隔与数据产生时间将数据写入不同的Index中，为了易于区分，会在Index的名字中加上对应的时间标识。创建新Index这件事，可以是客户端主动发起一个创建的请求，带上具体的Settings、Mappings等信息，但是可能会有一个时间错位，即有新数据写入时新的Index还没有建好。Elasticsearch提供了更优雅的方式来实现这个动作，即Index Template。

Index Template提供的功能是：先设置一个Template，定义好具体的Settings、Mappings等信息，当有数据需要写入新的Index时，就会根据Template内容自动创建Index。是否根据Template建立新的Index受三点因素制约：

* 是否是新的Index；
* Index的名字是否与template参数中定义的格式相匹配；
* 如果有多个Template匹配上了，则根据order参数的大小来依次生效，即从小到大逐步更新相应的配置信息（相同的配置信息会被覆盖）。

下面给出了一个具体的Index Template定义，来自于笔者项目中的真实定义（去掉了部分字段信息），该Template会匹配所有前缀为“ce-index-access-v1-”的Index，这个示例会被上下多个小节引用。

{

"facet\_internet\_access\_minute": {

"template": "ce-index-access-v1-\*",

"order": 0,

"settings": {

"number\_of\_shards": 5

},

"aliases": {

"{index}-query": {}

},

"mappings": {

"es\_doc": {

"dynamic": "strict",

"\_all": {

"enabled": false

},

"\_source": {

"enabled": false

},

"properties": {

"CLF\_Timestamp": {

"type": "long"

},

"CLF\_CustomerID": {

"type": "keyword"

},

"CLF\_ClientIP": {

"type": "ip",

"ignore\_malformed": true

}

}

}

}

}

}

### Mapping设计技巧

Index设计的第四件事，是结合业务需求来设定Mapping信息。如在专栏的第二篇中所述，Mapping定义了具体的数据结构与相关的元信息，合适的设置可以有效提高性能、节省磁盘空间。Mapping的设计，主要考虑两方面内容：

* Schema设计。第一，尽管Elasticsearch支持半结构化数据，但是在实际使用中还是应该尽最大可能对数据结构加以控制。第二，因为Elasticsearch不支持JOIN操作，所以Schema应该尽量扁平化。第三，对于只需要做精确匹配的字段，应该设置为不做分词，5.5中通过type=keyword来设定。
* 参数调整，即修改部分参数的默认值来适应自身业务。Elasticsearch中有很多参数可用，几乎可以满足大多数的业务需求，你很难记住所有的，不妨在有相关需求时先去文档查查看。

这里将结合上述Index Template示例，阐述几个笔者用过的Mapping参数。

* **\_all参数**

  假设在你的业务中，需要根据关键词来查找某条数据，但是并不明确知道要到哪个字段中去查，这时用\_all参数就可以帮助你解决。\_all=true时，Elasticsearch会在写入数据时将其所有字段值合并起来组合为一个新的值，对其建立索引，用来满足前面所述的搜索需求。也就是说，如果你的业务没有这种需求，那么将其设置为false，可以节省磁盘、提高性能。

* **\_source参数**

  我们知道Elasticsearch中存储的每一条数据都是一个JSON结构，在写入数据时会对每个字段建立Inverted Index、doc values等，如果\_source=true，Elasticsearch会将整个JSON数据也存储下来。如果你的业务中，不需要查询原始数据，只需要根据索引来过滤然后做聚合查询，那么可以将其设置为false，同样可以节省磁盘空间、提高性能。

* **dynamic参数**

  假如突然有一条数据里面包含了一个没有定义的字段，这时你期望Elasticsearch怎么做？dynanic参数便是用来处理此种情况的，有三种选择：strict表示不接受数据中包含没有定义的字段，发现了就报错；true表示允许没有定义的字段被插入进来；false表示忽略没有定义过的字段，继续写入数据的其他信息。具体怎么选，更多的取决于业务需求，但是要考虑清楚可能带来的后果。在笔者的项目中，就选择了strict，用来拒绝一切脏数据。

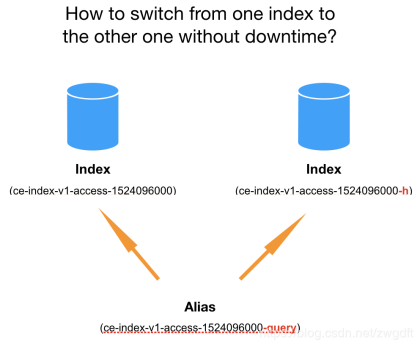
* **ignore\_malformed参数**

假设有一个字段CLF\_ClientIP传过来的是字符串，我们期望将其转换为ip数据类型，方便做查询。理论上，将type设置为ip就可以解决了，但是有这么一种情况，客户端在某些情况下没有拿到对应的IP信息，用了一个“–”来表示，这样的数据到了Elasticsearch端是没法转换为ip数据类型的，但是这又是一条正常数据，该怎么处理呢？通过ignore\_malformed=true，可以做到忽略CLF\_ClientIP字段而保留数据的其他信息。

### 巧妙的Alias

Alias，顾名思义，就是Index的别名。一个Alias可以指向多个Index，一个Index也可以关联多个Alias，具体可以通过Alias的REST API来设定。在查询数据时，Elasticsearch会自动检测请求的Path是否是Alias，是的话就会从其关联的Index中查询数据。Alias的巧妙之处在于，能帮助我们将查询请求无缝的从一个Index切换到另一个Index上面，这种切换对于客户端是透明的。

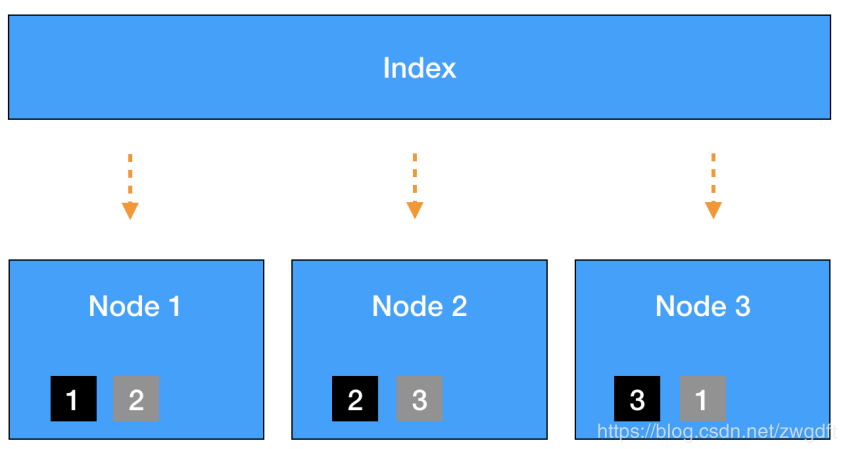
举个例子来阐述，上面有提到，在笔者业务中，存在一种数据合并：将分钟粒度数据合并成小时粒度的数据，从而可以降低数据量、提高查询性能。下图中，左边表示分钟粒度数据的Index，右边表示小时粒度数据的Index。用户的查询会先命中到分钟粒度Index上，一天之后，当历史数据完成合并后，会将查询请求转到小时粒度Index上（分钟粒度的Index会被删除）。通过Alias，我们就可以实现这种没有downtime的切换，并且对客户端透明。客户端只要按照固定的Alias来查询即可，至于Elasticsearch这边从哪个Index来查，则是取决于不同时间这个Alias关联的Index。（在数据合并完成后，切换Alias关联的Index，在一个请求中完成卸载老的Index和关联新的Index。）



### Shard分配原则

所谓Shard分配原则，就是如何设定Primary Shard的个数。看上去只是一个数字而已，也许在很多场景下，即使不设定也不会有问题（默认是5个Primary Shard）。但是如果不提前考虑，一旦出问题就可能导致系统性能下降、不可访问、甚至无法恢复。换句话说，即使使用默认值，也应该是通过足够的评估后作出的决定，而非拍脑袋定的。具体的分配方法，很难给出一个完美的套路适用于所有的业务需求，这里主要结合笔者的实践谈几点原则和调整方法，给大家提供一个思路。

* 单个Shard的存储大小不超过30GB。首先，为什么是30GB？关于这个值，来自于笔者跟AWS和Elastic专家的沟通结果，大家普遍认为30GB是个合适的上限值。目前我们尚未对此做过测试验证，不过实践中确实遇到过因为单个Shard过大（超过30GB）导致系统不稳定的情况。其次，为什么不能超过30GB？主要是考虑Shard Relocate过程的负载。我们知道，如果Shard不均衡或者部分节点故障，Elasticsearch会做Shard Relocate，在这个过程中会搬移Shard，如果单个Shard过大，会导致CPU、IO负载过高进而影响系统性能与稳定性。
* 单个Index的Primary Shard个数 = k \* 数据节点个数。在保证第一点的前提下，单个Index的Primary Shard个数不宜过多，否则相关的元信息与缓存会消耗过多的系统资源。这里的k，为一个较小的整数值，建议取值为1,2等，整数倍的关系可以让Shard更好地均匀分布，可以充分的将请求分散到不同节点上。
* 对于很小的Index，可以只分配1~2个Primary Shard的。有些情况下，Index很小，也许只有几十、几百MB左右，那么就不用按照第二点来分配了，只分配1~2个Primary Shard是可以，不用纠结。
* 聊一个特殊的例子。曾经遇到过一个Index，里面存放的是关键事件的日志信息，数据量不大（400MB左右），所以只分配了2个Primary Shard。但是因为“最近的事件”是用户非常关注的，所以这个Index承载了整个集群50%以上的查询请求，为了提高查询性能，将请求分散到不同机器上，最后还是将Primary Shard设置为数据节点的个数。



### 整体思路

Index与Shard的设计，并不是一下子就可以完成的，需要不断预估、测试、调整、再测试来达到最终模型。整个过程大致如下图时间轴所示，分为以下几个阶段，当然期间有很多阶段是需要迭代的。

第一阶段，Index的设计，即完成本文第1~3小节提到的内容。这一阶段是完成业务需求的分解与设计，之后剩下的就是如何构建Elasticsearch集群来承载这样的需求了。

第二阶段，预估数据量。预估单个Index的数据量，结合需要保留的数据周期可以得到整体的数据量。

第三阶段，预估机型。根据业务的场景，来决定选择CPU增强型、内存增强型还是通用型机器（针对云服务）。

第四阶段，预估机器个数。因为单机能挂载的磁盘大小是有限的，因此知道了数据量，也就知道了最少需要多少台机器。在预估磁盘空间时，还需要考虑另外两个因素：一个是Replica，即副本的个数，大多数情况下1个就够了（当然还是取决于业务）；另一个是预留30%的磁盘空间，这部分空间既是预留给系统使用的，也是为磁盘告警的处理预留足够的时间。

第五阶段，预估Primary Shard个数。即参考第4小节中的思路，预估Primary Shard的个数。

第六阶段，部署、测试。根据性能测试和业务的特殊情况，做适当调整，调整可能包括Primary Shard个数、机型、机器个数等。

# 代码样例

整合spring boot 2.1.9 + spring-data-elasticsearch

## Pom依赖

es 7.3.2 最低使用的客户端版本是6.8.0，目前还没有兼容 es 7以上的spring-boot-starter-data-elasticsearch 与spring-data-elasticsearch。

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

<elasticsearch.version>6.8.0</elasticsearch.version>

<!-- 目前最高的版本 -->

<spring.data.elasticsearch.version>3.2.0.RELEASE</spring.data.elasticsearch.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.elasticsearch</groupId>

<artifactId>elasticsearch</artifactId>

<version>${elasticsearch.version}</version>

</dependency>

<dependency>

<groupId>org.elasticsearch.client</groupId>

<artifactId>transport</artifactId>

<version>${elasticsearch.version}</version>

</dependency>

<dependency>

<groupId>org.elasticsearch.client</groupId>

<artifactId>elasticsearch-rest-client</artifactId>

<version>${elasticsearch.version}</version>

</dependency>

<dependency>

<groupId>org.elasticsearch.plugin</groupId>

<artifactId>transport-netty4-client</artifactId>

<version>${elasticsearch.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-elasticsearch</artifactId>

<version>${spring.data.elasticsearch.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-elasticsearch</artifactId>

</dependency>

</dependencies>

## 配置application.yml

spring:

data:

elasticsearch:

cluster-name: my-application

cluster-nodes: 127.0.0.1:9300

配置文件中明明写的端口是9200，为何这里配置文件中连接的时候写的端口是9300呢？

因为，配置9200是通过HTTP连接的端口，9300是TCP连接的端口

## 使用Spring Data Elasticsearch Repositories操作

首先，定义一个实体类

package com.zx.entity;

import java.io.Serializable;

import org.springframework.data.annotation.Id;

import org.springframework.data.elasticsearch.annotations.Document;

import lombok.Data;

@Data

@Document(indexName = "commodity")

public class Commodity implements Serializable {

private static final long serialVersionUID = 1L;

@Id

private String skuId;

private String name;

private String category;

private Integer price;

private String brand;

private Integer stock;

}

这里定义了Commodity实例，表示商品。在Elasticsearch 6.X 版本中，不建议使用type，而且在7.X版本中将会彻底废弃type，所以此处我只指定了indexName，没有指定type。

这里，一个Commodity代表一个商品，同时代表一条索引记录。

类比关系型数据库的话，Index相当于表，Document相当于记录，然后，需要自己定义一个接口，并继承ElasticsearchRepository。

package com.zx.dao;

import org.springframework.data.elasticsearch.repository.ElasticsearchRepository;

import org.springframework.stereotype.Repository;

import com.zx.entity.Commodity;

@Repository

public interface CommodityRepository extends ElasticsearchRepository<Commodity, String> {

}

这里的Repository相当于DAO，操作mysql还是elasticsearch都是一样的

接下来，定义service接口

package com.zx.service;

import java.util.List;

import org.springframework.data.domain.Page;

import com.zx.entity.Commodity;

public interface CommodityService {

long count();

Commodity save(Commodity commodity);

void delete(Commodity commodity);

Iterable<Commodity> getAll();

List<Commodity> getByName(String name);

Page<Commodity> pageQuery(Integer pageNo, Integer pageSize, String kw);

}

实现类，主要是操作es的地方

package com.zx.service.impl;

import java.util.ArrayList;

import java.util.List;

import org.elasticsearch.index.query.MatchQueryBuilder;

import org.elasticsearch.index.query.QueryBuilders;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.PageRequest;

import org.springframework.data.elasticsearch.core.query.NativeSearchQueryBuilder;

import org.springframework.data.elasticsearch.core.query.SearchQuery;

import org.springframework.stereotype.Service;

import com.zx.dao.CommodityRepository;

import com.zx.entity.Commodity;

import com.zx.service.CommodityService;

@Service

public class CommodityServiceImpl implements CommodityService {

@Autowired

private CommodityRepository commodityRepository;

@Override

public long count() {

return commodityRepository.count();

}

@Override

public Commodity save(Commodity commodity) {

return commodityRepository.save(commodity);

}

@Override

public void delete(Commodity commodity) {

commodityRepository.delete(commodity);

}

@Override

public Iterable<Commodity> getAll() {

return commodityRepository.findAll();

}

@Override

public List<Commodity> getByName(String name) {

List<Commodity> list = new ArrayList<>();

MatchQueryBuilder matchQueryBuilder = new MatchQueryBuilder("name", name);

Iterable<Commodity> iterable = commodityRepository.search(matchQueryBuilder);

iterable.forEach(e->list.add(e));

return list;

}

@Override

public Page<Commodity> pageQuery(Integer pageNo, Integer pageSize, String kw) {

SearchQuery searchQuery = new NativeSearchQueryBuilder()

.withQuery(QueryBuilders.matchPhraseQuery("name", kw))

.withPageable(PageRequest.of(pageNo, pageSize))

.build();

return commodityRepository.search(searchQuery);

}

}

单元测试

package com.zx;

import java.util.List;

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.data.domain.Page;

import org.springframework.test.context.junit4.SpringRunner;

import com.zx.entity.Commodity;

import com.zx.service.CommodityService;

@RunWith(SpringRunner.class)

@SpringBootTest

public class ApplicationTests {

@Autowired

private CommodityService commodityService;

@Test

public void contextLoads() {

System.out.println(commodityService.count());

}

@Test

public void testInsert() {

Commodity commodity = new Commodity();

commodity.setSkuId("1501009001");

commodity.setName("原味切片面包（10片装）");

commodity.setCategory("101");

commodity.setPrice(880);

commodity.setBrand("良品铺子");

commodityService.save(commodity);

commodity = new Commodity();

commodity.setSkuId("1501009002");

commodity.setName("原味切片面包（6片装）");

commodity.setCategory("101");

commodity.setPrice(680);

commodity.setBrand("良品铺子");

commodityService.save(commodity);

commodity = new Commodity();

commodity.setSkuId("1501009004");

commodity.setName("元气吐司850g");

commodity.setCategory("101");

commodity.setPrice(120);

commodity.setBrand("百草味");

commodityService.save(commodity);

}

@Test

public void testDelete() {

Commodity commodity = new Commodity();

commodity.setSkuId("1501009002");

commodityService.delete(commodity);

}

@Test

public void testGetAll() {

Iterable<Commodity> iterable = commodityService.getAll();

iterable.forEach(e -> System.out.println(e.toString()));

}

@Test

public void testGetByName() {

List<Commodity> list = commodityService.getByName("面包");

System.out.println(list);

}

@Test

public void testPage() {

Page<Commodity> page = commodityService.pageQuery(0, 10, "切片");

System.out.println(page.getTotalPages());

System.out.println(page.getNumber());

System.out.println(page.getContent());

}

}

## 使用ElasticsearchTemplate方式操作

Spring的模板模式，可以参考RedisTemplate

@RunWith(SpringRunner.class)

@SpringBootTest

public class ElasticsearchTemplateTest {

@Autowired

public ElasticsearchTemplate elasticsearchTemplate;

@Test

public void testInsert() {

Commodity commodity = new Commodity();

commodity.setSkuId("1501009005");

commodity.setName("葡萄吐司面包（10片装）");

commodity.setCategory("101");

commodity.setPrice(160);

commodity.setBrand("良品铺子");

IndexQuery indexQuery = new IndexQueryBuilder().withObject(commodity).build();

elasticsearchTemplate.index(indexQuery);

}

@Test

public void testQuery() {

SearchQuery searchQuery = new NativeSearchQueryBuilder()

.withQuery(QueryBuilders.matchQuery("name", "吐司"))

.build();

List<Commodity> list = elasticsearchTemplate.queryForList(searchQuery, Commodity.class);

System.out.println(list);

}

}

ElasticsearchTemplate是自动加载的ElasticsearchAutoConfiguration

满足这两个条件就会加载ElasticsearchTemplate

