## elasticsearch顶尖高手

# 什么是elasticsearch(5.2.0)

课程大纲

大白话、什么是Elasticsearch

Elasticsearch，分布式，高性能，高可用，可伸缩的搜索和分析系统

1、什么是搜索？

2、如果用数据库做搜索会怎么样？

3、什么是全文检索、倒排索引和Lucene？

4、什么是Elasticsearch？

------------------------------------------------------------------------------------------------------------------------

## 1、什么是搜索？

百度：我们比如说想找寻任何的信息的时候，就会上百度去搜索一下，比如说找一部自己喜欢的电影，或者说找一本喜欢的书，或者找一条感兴趣的新闻（提到搜索的第一印象）

百度 != 搜索，这是不对的

垂直搜索（站内搜索）

互联网的搜索：电商网站，招聘网站，新闻网站，各种app

IT系统的搜索：OA软件，办公自动化软件，会议管理，日程管理，项目管理，员工管理，搜索“张三”，“张三儿”，“张小三”；有个电商网站，卖家，后台管理系统，搜索“牙膏”，订单，“牙膏相关的订单”

搜索，就是在任何场景下，找寻你想要的信息，这个时候，会输入一段你要搜索的关键字，然后就期望找到这个关键字相关的有些信息

------------------------------------------------------------------------------------------------------------------------

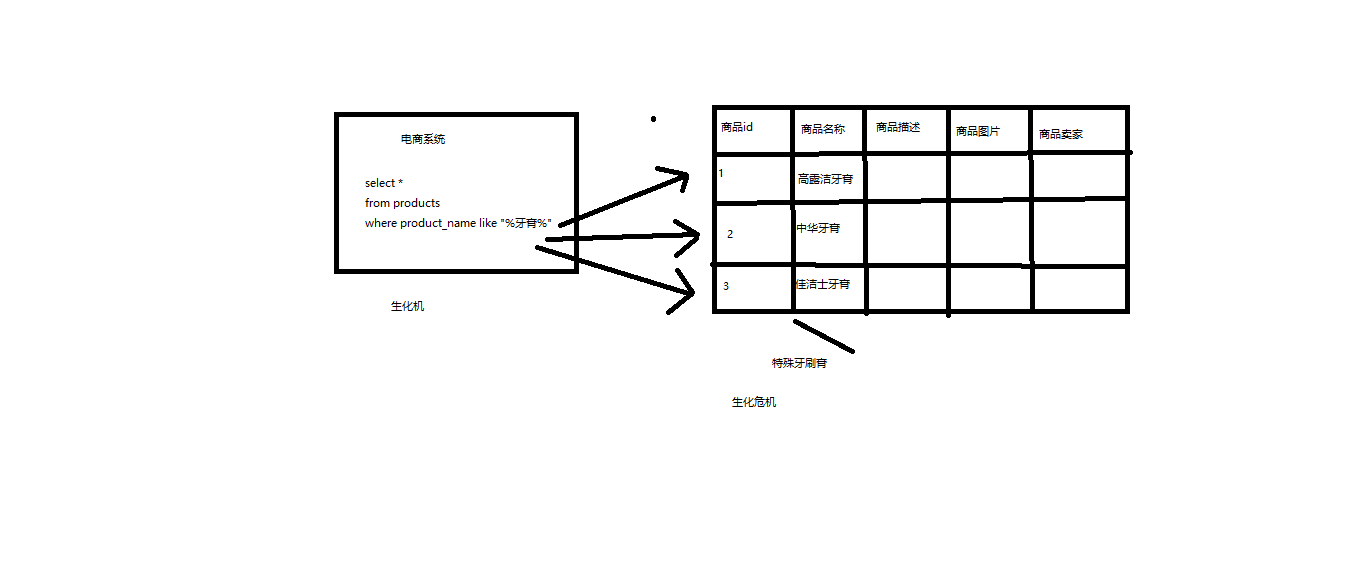
## 2、如果用数据库做搜索会怎么样？

做软件开发的话，或者对IT、计算机有一定的了解的话，都知道，数据都是存储在数据库里面的，比如说电商网站的商品信息，招聘网站的职位信息，新闻网站的新闻信息，等等吧。所以说，很自然的一点，如果说从技术的角度去考虑，如何实现如说，电商网站内部的搜索功能的话，就可以考虑，去使用数据库去进行搜索。

1、每条记录的指定字段的文本，可能会很长，比如“商品描述”字段的长度，有长达数千个，甚至数万个字符，这个时候，每次都要对每条记录的所有文本进行扫描，懒判断说，你包不包含我指定的这个关键词（比如说“牙膏”）

2、还不能将搜索词拆分开来，尽可能去搜索更多的符合你的期望的结果，比如输入“生化机”，就搜索不出来“生化危机”

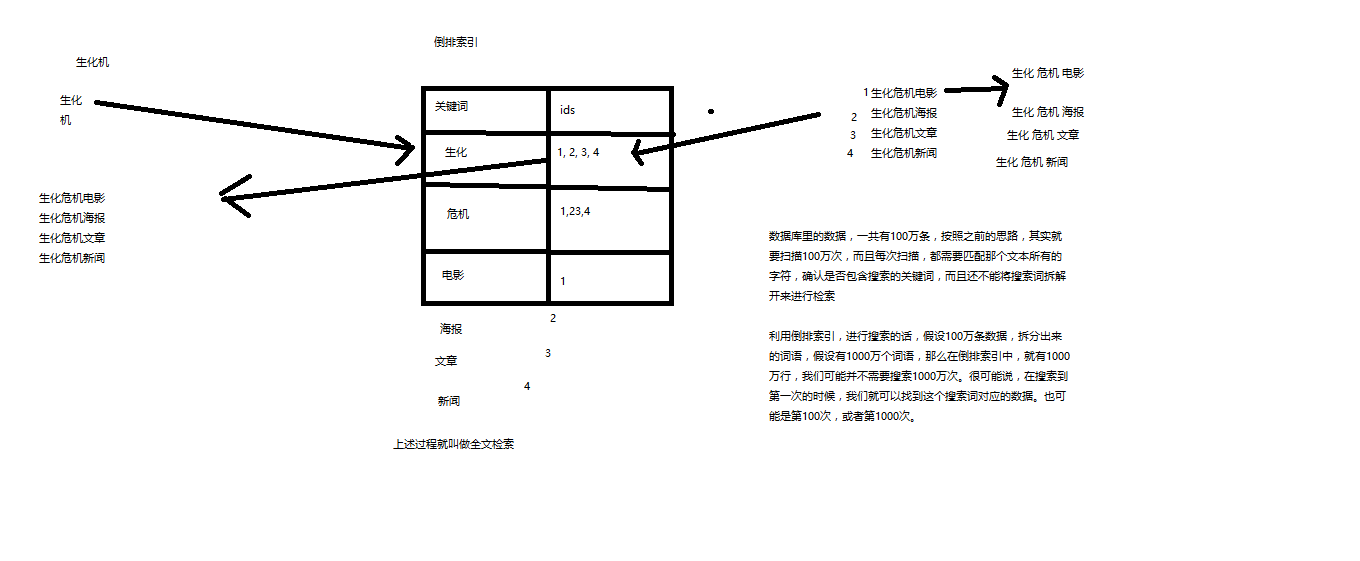
用数据库来实现搜索，是不太靠谱的。通常来说，性能会很差的。



------------------------------------------------------------------------------------------------------------------------

## 3、什么是全文检索和Lucene？

1. 全文检索，倒排索引

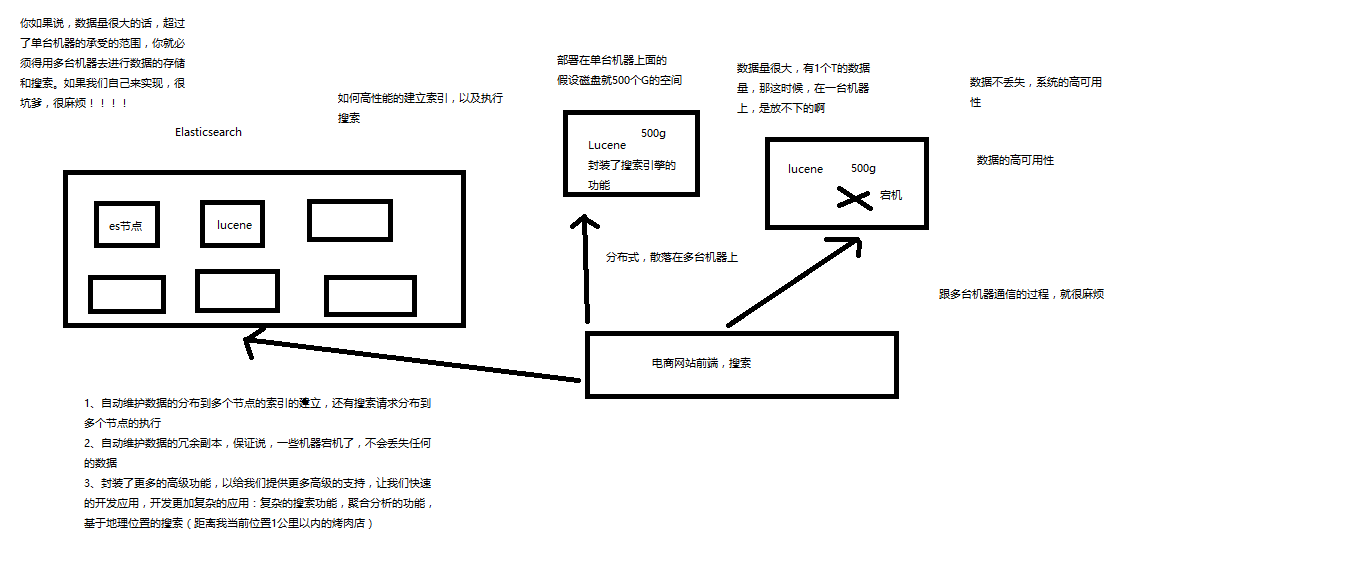


（2）lucene，就是一个jar包，里面包含了封装好的各种建立倒排索引，以及进行搜索的代码，包括各种算法。我们就用java开发的时候，引入lucene jar，然后基于lucene的api进行去进行开发就可以了。用lucene，我们就可以去将已有的数据建立索引，lucene会在本地磁盘上面，给我们组织索引的数据结构。另外的话，我们也可以用lucene提供的一些功能和api来针对磁盘上额

------------------------------------------------------------------------------------------------------------------------

## 4、什么是Elasticsearch？

（1）图解分析



# Elasticsearch的功能、适用场景以及特点介绍

课程大纲

1、Elasticsearch的功能，干什么的

2、Elasticsearch的适用场景，能在什么地方发挥作用

3、Elasticsearch的特点，跟其他类似的东西不同的地方在哪里

----------------------------------------------------------------------------------------------------------------------

## 1、Elasticsearch的功能

（1）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

（2）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

（3）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

海量数据的处理：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

----------------------------------------------------------------------------------------------------------------------

## 2、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

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

----------------------------------------------------------------------------------------------------------------------

## 3、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

# 三、es基本概念

课程大纲

1、lucene和elasticsearch的前世今生

2、elasticsearch的核心概念

3、elasticsearch核心概念 vs. 数据库核心概念

----------------------------------------------------------------------------------------------------------------------------------------

## 1、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

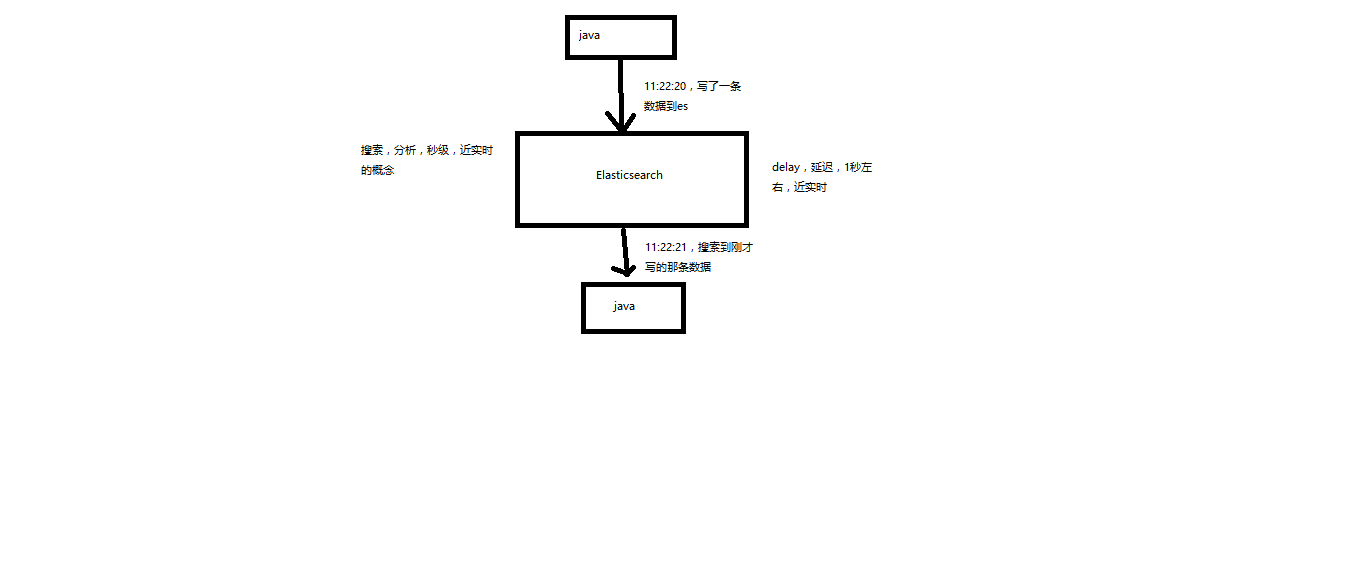
开箱即用，优秀的默认参数，不需要任何额外设置，完全开源

关于elasticsearch的一个传说，有一个程序员失业了，陪着自己老婆去英国伦敦学习厨师课程。程序员在失业期间想给老婆写一个菜谱搜索引擎，觉得lucene实在太复杂了，就开发了一个封装了lucene的开源项目，compass。后来程序员找到了工作，是做分布式的高性能项目的，觉得compass不够，就写了elasticsearch，让lucene变成分布式的系统。

----------------------------------------------------------------------------------------------------------------------------------------

## 2、elasticsearch的核心概念

（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

{

"product\_id": "2",

"product\_name": "长虹电视机",

"product\_desc": "4k高清",

"category\_id": "3",

"category\_name": "电器",

"service\_period": "1年"

}

{

"product\_id": "3",

"product\_name": "基围虾",

"product\_desc": "纯天然，冰岛产",

"category\_id": "4",

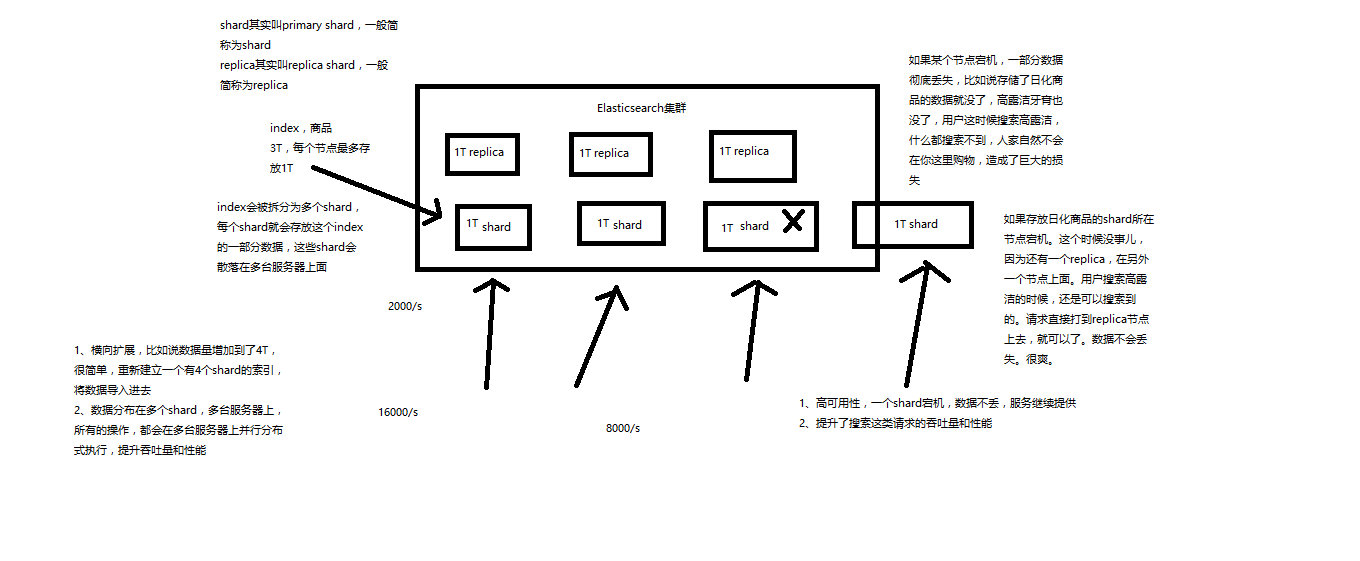
"category\_name": "生鲜",

"eat\_period": "7天"

}

（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（mapreduce），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。



----------------------------------------------------------------------------------------------------------------------------------------

## 3、elasticsearch核心概念 vs. 数据库核心概念

Elasticsearch 数据库

-----------------------------------------

Document 行

Type 表

Index 库

# 四、集群健康检查 document的CRUD

课程大纲

1、document数据格式

2、电商网站商品管理案例：背景介绍

3、简单的集群管理

4、商品的CRUD操作（document CRUD操作）

----------------------------------------------------------------------------------------------------------------------------

## 1、document数据格式

面向文档的搜索分析引擎

（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

public class Employee {

private String email;

private String firstName;

private String lastName;

private EmployeeInfo info;

private Date joinDate;

}

private class EmployeeInfo {

private String bio; // 性格

private Integer age;

private String[] interests; // 兴趣爱好

}

EmployeeInfo info = new EmployeeInfo();

info.setBio("curious and modest");

info.setAge(30);

info.setInterests(new String[]{"bike", "climb"});

Employee employee = new Employee();

employee.setEmail("zhangsan@sina.com");

employee.setFirstName("san");

employee.setLastName("zhang");

employee.setInfo(info);

employee.setJoinDate(new Date());

employee对象：里面包含了Employee类自己的属性，还有一个EmployeeInfo对象

两张表：employee表，employee\_info表，将employee对象的数据重新拆开来，变成Employee数据和EmployeeInfo数据

employee表：email，first\_name，last\_name，join\_date，4个字段

employee\_info表：bio，age，interests，3个字段；此外还有一个外键字段，比如employee\_id，关联着employee表

{

"email": "zhangsan@sina.com",

"first\_name": "san",

"last\_name": "zhang",

"info": {

"bio": "curious and modest",

"age": 30,

"interests": [ "bike", "climb" ]

},

"join\_date": "2017/01/01"

}

我们就明白了es的document数据格式和数据库的关系型数据格式的区别

----------------------------------------------------------------------------------------------------------------------------

## 2、电商网站商品管理案例背景介绍

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

----------------------------------------------------------------------------------------------------------------------------

## 3、简单的集群管理

### （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

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。

做一个小实验：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。

### （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

yellow open .kibana rUm9n9wMRQCCrRDEhqneBg 1 1 1 0 3.1kb 3.1kb

### （3）简单的索引操作

创建索引：PUT /test\_index?pretty

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

yellow open test\_index XmS9DTAtSkSZSwWhhGEKkQ 5 1 0 0 650b 650b

yellow open .kibana rUm9n9wMRQCCrRDEhqneBg 1 1 1 0 3.1kb 3.1kb

删除索引：DELETE /test\_index?pretty

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

yellow open .kibana rUm9n9wMRQCCrRDEhqneBg 1 1 1 0 3.1kb 3.1kb

----------------------------------------------------------------------------------------------------------------------------

## 4、商品的CRUD操作

### （1）新增商品：新增文档，建立索引

PUT /index/type/id

{

"json数据"

}

PUT /ecommerce/product/1

{

"name" : "gaolujie yagao",

"desc" : "gaoxiao meibai",

"price" : 30,

"producer" : "gaolujie producer",

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

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

PUT /ecommerce/product/2

{

"name" : "jiajieshi yagao",

"desc" : "youxiao fangzhu",

"price" : 25,

"producer" : "jiajieshi producer",

"tags": [ "fangzhu" ]

}

PUT /ecommerce/product/3

{

"name" : "zhonghua yagao",

"desc" : "caoben zhiwu",

"price" : 40,

"producer" : "zhonghua producer",

"tags": [ "qingxin" ]

}

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

### （2）查询商品：检索文档

GET /index/type/id

GET /ecommerce/product/1

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 1,

"found": true,

"\_source": {

"name": "gaolujie yagao",

"desc": "gaoxiao meibai",

"price": 30,

"producer": "gaolujie producer",

"tags": [

"meibai",

"fangzhu"

]

}

}

### （3）修改商品：替换文档

PUT /ecommerce/product/1

{

"name" : "jiaqiangban gaolujie yagao",

"desc" : "gaoxiao meibai",

"price" : 30,

"producer" : "gaolujie producer",

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

}

//添加文档时的返回值

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

//替换文档时的返回值

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

PUT /ecommerce/product/1

{

"name" : "jiaqiangban gaolujie yagao"

}

替换方式有一个不好，即使必须带上所有的field，才能去进行信息的修改

### （4）修改商品：更新文档

POST /ecommerce/product/1/\_update

{

"doc": {

"name": "jiaqiangban gaolujie yagao"

}

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 8,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

我的风格，其实有选择的情况下，不太喜欢念ppt，或者照着文档做，或者直接粘贴写好的代码，尽量是纯手敲代码

### （5）删除商品：删除文档

DELETE /ecommerce/product/1

{

"found": true,

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 9,

"result": "deleted",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"found": false

}

# 多种搜索方式

课程大纲

1、query string search

2、query DSL

3、query filter

4、full-text search

5、phrase search

6、highlight search

---------------------------------------------------------------------------------------------------------------------------------

把英文翻译成中文，让我觉得很别扭，term，词项

## 1、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": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 1,

"hits": [

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "2",

"\_score": 1,

"\_source": {

"name": "jiajieshi yagao",

"desc": "youxiao fangzhu",

"price": 25,

"producer": "jiajieshi producer",

"tags": [

"fangzhu"

]

}

},

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_score": 1,

"\_source": {

"name": "gaolujie yagao",

"desc": "gaoxiao meibai",

"price": 30,

"producer": "gaolujie producer",

"tags": [

"meibai",

"fangzhu"

]

}

},

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "3",

"\_score": 1,

"\_source": {

"name": "zhonghua yagao",

"desc": "caoben zhiwu",

"price": 40,

"producer": "zhonghua producer",

"tags": [

"qingxin"

]

}

}

]

}

}

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yagao的商品，而且按照售价降序排序：GET /ecommerce/product/\_search?q=name:yagao&sort=price:desc

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

---------------------------------------------------------------------------------------------------------------------------------

## 2、query DSL

DSL：Domain 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

查询所有的商品

GET /ecommerce/product/\_search

{

"query": { "match\_all": {} }

}

查询名称包含yagao的商品，同时按照价格降序排序

GET /ecommerce/product/\_search

{

"query" : {

"match" : {

"name" : "yagao"

}

},

"sort": [

{ "price": "desc" }

]

}

分页查询商品，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/product/\_search

{

"query": { "match\_all": {} },

"from": 1,

"size": 1

}

指定要查询出来商品的名称和价格就可以

GET /ecommerce/product/\_search

{

"query": { "match\_all": {} },

"\_source": ["name", "price"]

}

更加适合生产环境的使用，可以构建复杂的查询

---------------------------------------------------------------------------------------------------------------------------------

## 3、query filter

搜索商品名称包含yagao，而且售价大于25元的商品

GET /ecommerce/product/\_search

{

"query" : {

"bool" : {

"must" : {

"match" : {

"name" : "yagao"

}

},

"filter" : {

"range" : {

"price" : { "gt" : 25 }

}

}

}

}

}

---------------------------------------------------------------------------------------------------------------------------------

## 4、full-text search（全文检索）

GET /ecommerce/product/\_search

{

"query" : {

"match" : {

"producer" : "yagao producer"

}

}

}

尽量，无论是学什么技术，比如说你当初学java，学linux，学shell，学javascript，学hadoop。。。。一定自己动手，特别是手工敲各种命令和代码，切记切记，减少复制粘贴的操作。只有自己动手手工敲，学习效果才最好。

producer这个字段，会先被拆解，建立倒排索引

special 4

yagao 4

producer 1,2,3,4

gaolujie 1

zhognhua 3

jiajieshi 2

yagao producer ---> yagao和producer

{

"took": 4,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 4,

"max\_score": 0.70293105,

"hits": [

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "4",

"\_score": 0.70293105,

"\_source": {

"name": "special yagao",

"desc": "special meibai",

"price": 50,

"producer": "special yagao producer",

"tags": [

"meibai"

]

}

},

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_score": 0.25811607,

"\_source": {

"name": "gaolujie yagao",

"desc": "gaoxiao meibai",

"price": 30,

"producer": "gaolujie producer",

"tags": [

"meibai",

"fangzhu"

]

}

},

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "3",

"\_score": 0.25811607,

"\_source": {

"name": "zhonghua yagao",

"desc": "caoben zhiwu",

"price": 40,

"producer": "zhonghua producer",

"tags": [

"qingxin"

]

}

},

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "2",

"\_score": 0.1805489,

"\_source": {

"name": "jiajieshi yagao",

"desc": "youxiao fangzhu",

"price": 25,

"producer": "jiajieshi producer",

"tags": [

"fangzhu"

]

}

}

]

}

}

---------------------------------------------------------------------------------------------------------------------------------

## 5、phrase search（短语搜索）

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/product/\_search

{

"query" : {

"match\_phrase" : {

"producer" : "yagao producer"

}

}

}

{

"took": 11,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 0.70293105,

"hits": [

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "4",

"\_score": 0.70293105,

"\_source": {

"name": "special yagao",

"desc": "special meibai",

"price": 50,

"producer": "special yagao producer",

"tags": [

"meibai"

]

}

}

]

}

}

---------------------------------------------------------------------------------------------------------------------------------

## 6、highlight search（高亮搜索结果）

GET /ecommerce/product/\_search

{

"query" : {

"match" : {

"producer" : "producer"

}

},

"highlight": {

"fields" : {

"producer" : {}

}

}

}

# 嵌套聚合、下钻分析、聚合分析

第一个分析需求：计算每个tag下的商品数量

GET /ecommerce/product/\_search

{

"aggs": {

"group\_by\_tags": {

"terms": { "field": "tags" }

}

}

}

将文本field的fielddata属性设置为true

PUT /ecommerce/\_mapping/product

{

"properties": {

"tags": {

"type": "text",

"fielddata": true

}

}

}

GET /ecommerce/product/\_search

{

"size": 0,

"aggs": {

"all\_tags": {

"terms": { "field": "tags" }

}

}

}

{

"took": 20,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 4,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_tags": {

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

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

"buckets": [

{

"key": "fangzhu",

"doc\_count": 2

},

{

"key": "meibai",

"doc\_count": 2

},

{

"key": "qingxin",

"doc\_count": 1

}

]

}

}

}

----------------------------------------------------------------------------------------------------------------

第二个聚合分析的需求：对名称中包含yagao的商品，计算每个tag下的商品数量

GET /ecommerce/product/\_search

{

"size": 0,

"query": {

"match": {

"name": "yagao"

}

},

"aggs": {

"all\_tags": {

"terms": {

"field": "tags"

}

}

}

}

----------------------------------------------------------------------------------------------------------------

第三个聚合分析的需求：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 /ecommerce/product/\_search

{

"size": 0,

"aggs" : {

"group\_by\_tags" : {

"terms" : { "field" : "tags" },

"aggs" : {

"avg\_price" : {

"avg" : { "field" : "price" }

}

}

}

}

}

{

"took": 8,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 4,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_tags": {

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

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

"buckets": [

{

"key": "fangzhu",

"doc\_count": 2,

"avg\_price": {

"value": 27.5

}

},

{

"key": "meibai",

"doc\_count": 2,

"avg\_price": {

"value": 40

}

},

{

"key": "qingxin",

"doc\_count": 1,

"avg\_price": {

"value": 40

}

}

]

}

}

}

----------------------------------------------------------------------------------------------------------------

第四个数据分析需求：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 /ecommerce/product/\_search

{

"size": 0,

"aggs" : {

"all\_tags" : {

"terms" : { "field" : "tags", "order": { "avg\_price": "desc" } },

"aggs" : {

"avg\_price" : {

"avg" : { "field" : "price" }

}

}

}

}

}

我们现在全部都是用es的restful api在学习和讲解es的所欲知识点和功能点，但是没有使用一些编程语言去讲解（比如java），原因有以下：

1、es最重要的api，让我们进行各种尝试、学习甚至在某些环境下进行使用的api，就是restful api。如果你学习不用es restful api，比如我上来就用java api来讲es，也是可以的，但是你根本就漏掉了es知识的一大块，你都不知道它最重要的restful api是怎么用的

2、讲知识点，用es restful api，更加方便，快捷，不用每次都写大量的java代码，能加快讲课的效率和速度，更加易于同学们关注es本身的知识和功能的学习

3、我们通常会讲完es知识点后，开始详细讲解java api，如何用java api执行各种操作

4、我们每个篇章都会搭配一个项目实战，项目实战是完全基于java去开发的真实项目和系统

----------------------------------------------------------------------------------------------------------------

第五个数据分析需求：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 /ecommerce/product/\_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"

}

}

}

}

}

}

}

}

# es的分布式架构

课程大纲

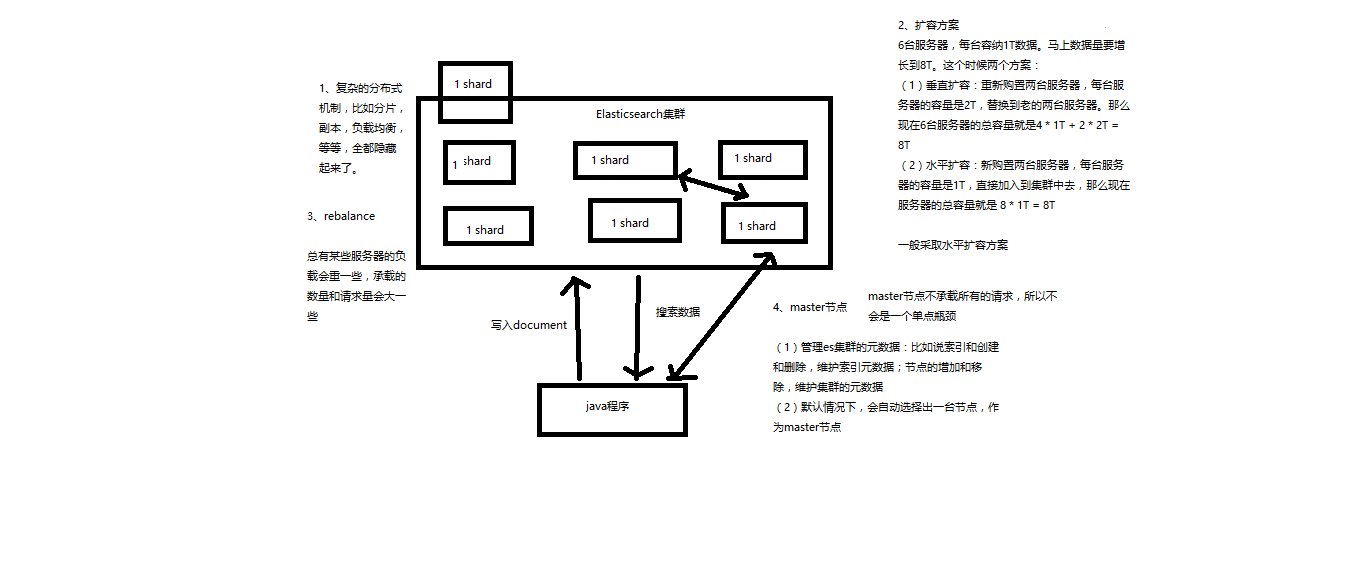
1、Elasticsearch对复杂分布式机制的透明隐藏特性

2、Elasticsearch的垂直扩容与水平扩容

3、增减或减少节点时的数据rebalance

4、master节点

5、节点对等的分布式架构



--------------------------------------------------------------------------------------------------------------------

## 1、Elasticsearch对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（我们之前随随便便就将一些document插入到es集群中去了，我们有没有care过数据怎么进行分片的，数据到哪个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

--------------------------------------------------------------------------------------------------------------------

## 2、Elasticsearch的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

普通服务器：1T，1万，100万

强大服务器：10T，50万，500万

扩容对应用程序的透明性

--------------------------------------------------------------------------------------------------------------------

## 3、增减或减少节点时的shard数据rebalance

保持负载均衡

--------------------------------------------------------------------------------------------------------------------

## 4、master节点

（1）创建或删除索引

（2）增加或删除节点

--------------------------------------------------------------------------------------------------------------------

## 5、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

# 八、shard&replica机制再次梳理以及单node环境中创建index图解

课程大纲

1、shard&replica机制再次梳理

2、图解单node环境下创建index是什么样子的

------------------------------------------------------------------------------------------------

## 1、shard&replica机制再次梳理

（1）index包含多个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(document不可再分)

（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

（9）同一个primary shard的replica shard不能放在同一个节点上

------------------------------------------------------------------------------------------------

## 2、单node环境下创建index

（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\_index

{

"settings" : {

"number\_of\_shards" : 3,

"number\_of\_replicas" : 1

}

}

# 九、2个node环境下replica shard的分配

课程大纲

1、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

# 横向扩容过程，如何超出扩容极限，以及如何提升容错性

课程大纲

1、图解横向扩容过程，如何超出扩容极限，以及如何提升容错性

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

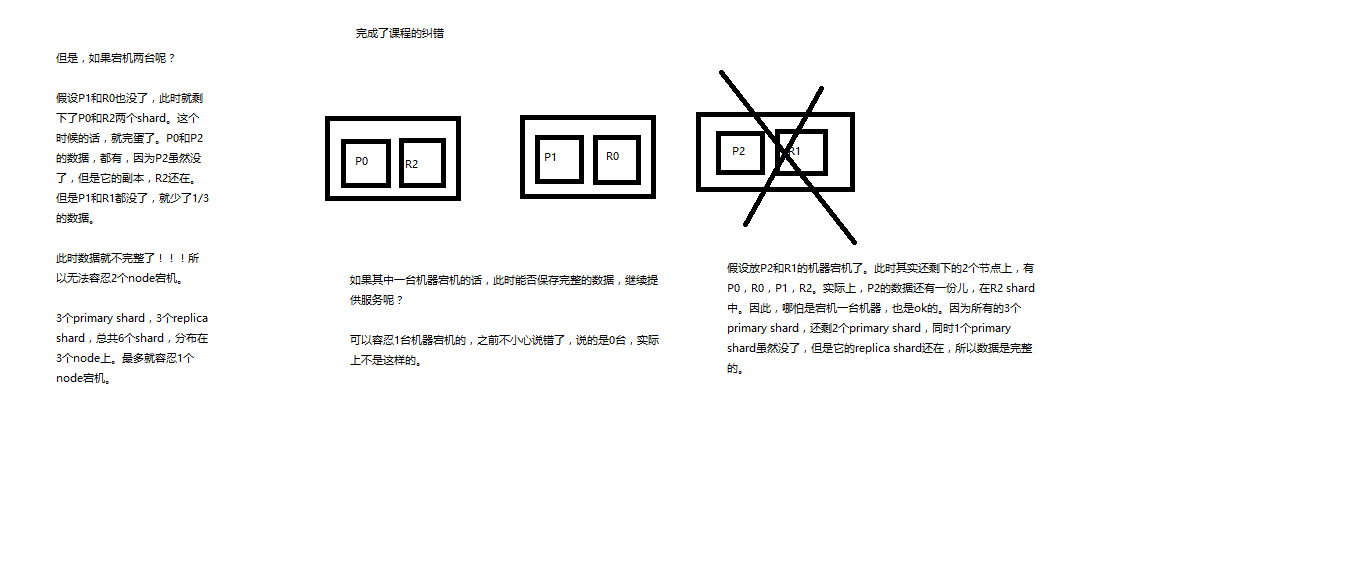
（2）每个node有更少的shard，IO/CPU/Memory资源给每个shard分配更多，每个shard性能更好

（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只能容纳1台机器宕机

1. 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



# Elasticsearch容错机制：master选举，replica容错，数据恢复

课程大纲

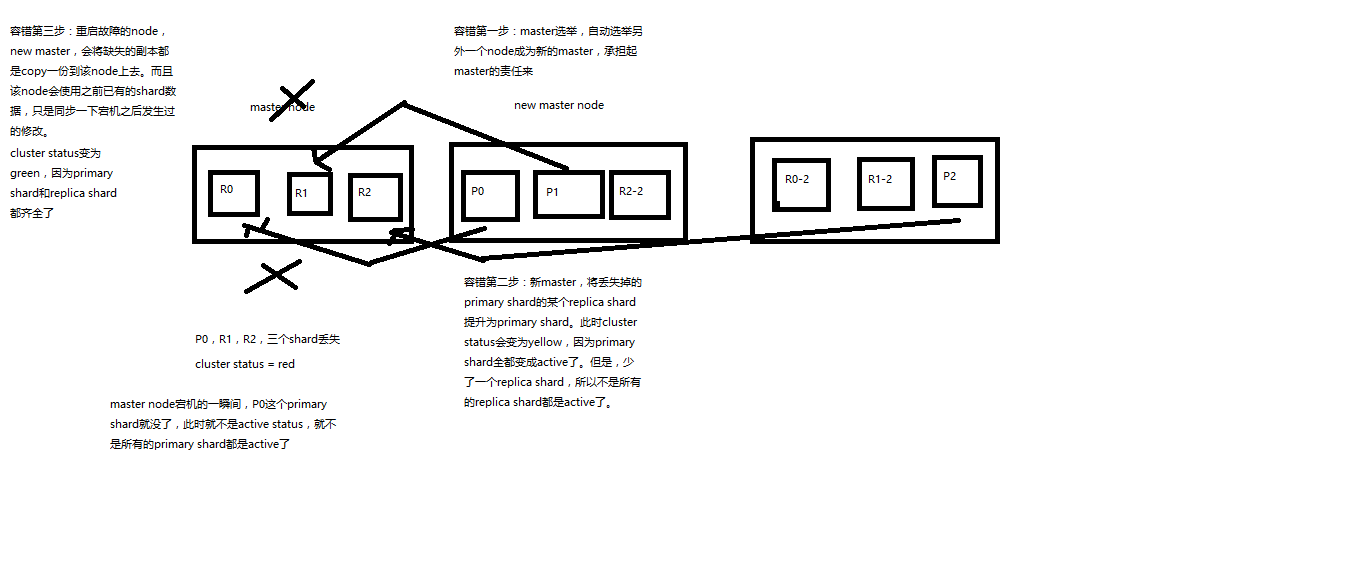
1、图解Elasticsearch容错机制：master选举，replica容错，数据恢复

（1）9 shard，3 node

（2）master node宕机，自动master选举，red

（3）replica容错：新master将replica提升为primary shard，yellow

（4）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



# 初步解析document的核心元数据以及图解剖析index创建反例

课程大纲

1、\_index元数据

2、\_type元数据

3、\_id元数据

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "1",

"\_version": 1,

"found": true,

"\_source": {

"test\_content": "test test"

}

}

------------------------------------------------------------------------------------------------------------------------------------------

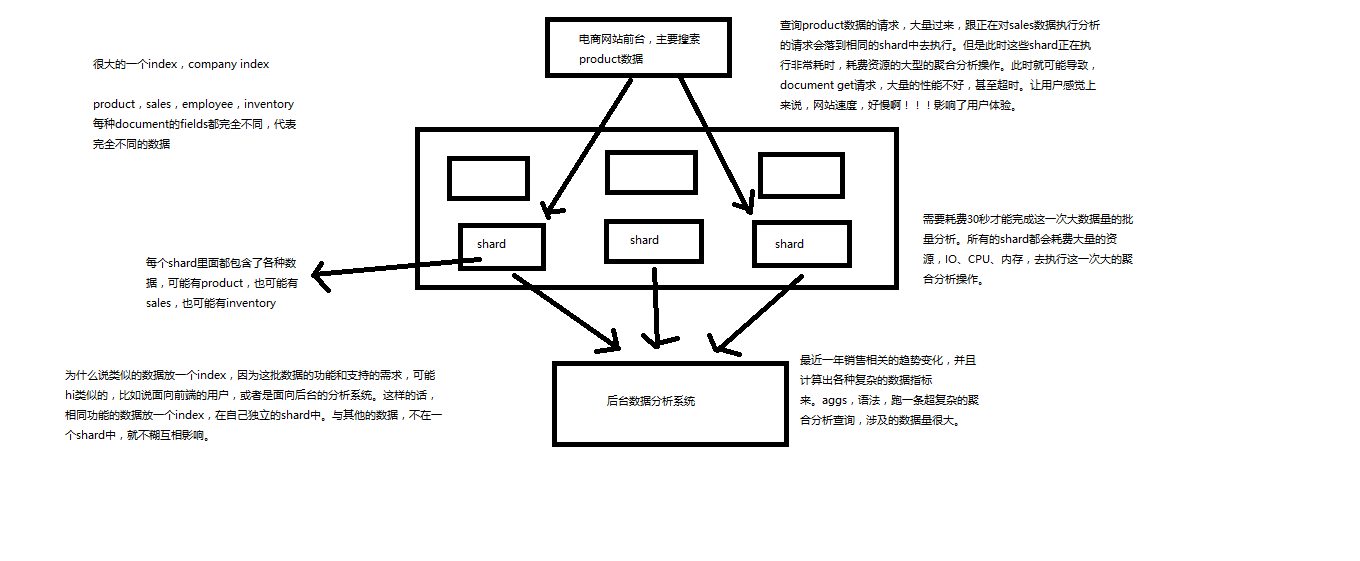
## 1、\_index元数据

（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：类似指的是，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



## 2、\_type元数据

（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

## 3、\_id元数据

（1）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

（2）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

# document id的手动指定与自动生成两种方式解析

课程大纲

1、手动指定document id

2、自动生成document id

------------------------------------------------------------------------------------------------------------

## 1、手动指定document id

（1）根据应用情况来说，是否满足手动指定document id的前提：

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。举个例子，比如说，我们现在在开发一个电商网站，做搜索功能，或者是OA系统，做员工检索功能。这个时候，数据首先会在网站系统或者IT系统内部的数据库中，会先有一份，此时就肯定会有一个数据库的primary key（自增长，UUID，或者是业务编号）。如果将数据导入到es中，此时就比较适合采用数据在数据库中已有的primary key。

如果说，我们是在做一个系统，这个系统主要的数据存储就是es一种，也就是说，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。

（2）put /index/type/id

PUT /test\_index/test\_type/2

{

"test\_content": "my test"

}

## 2、自动生成document id

（1）post /index/type

POST /test\_index/test\_type

{

"test\_content": "my test"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "AVp4RN0bhjxldOOnBxaE",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

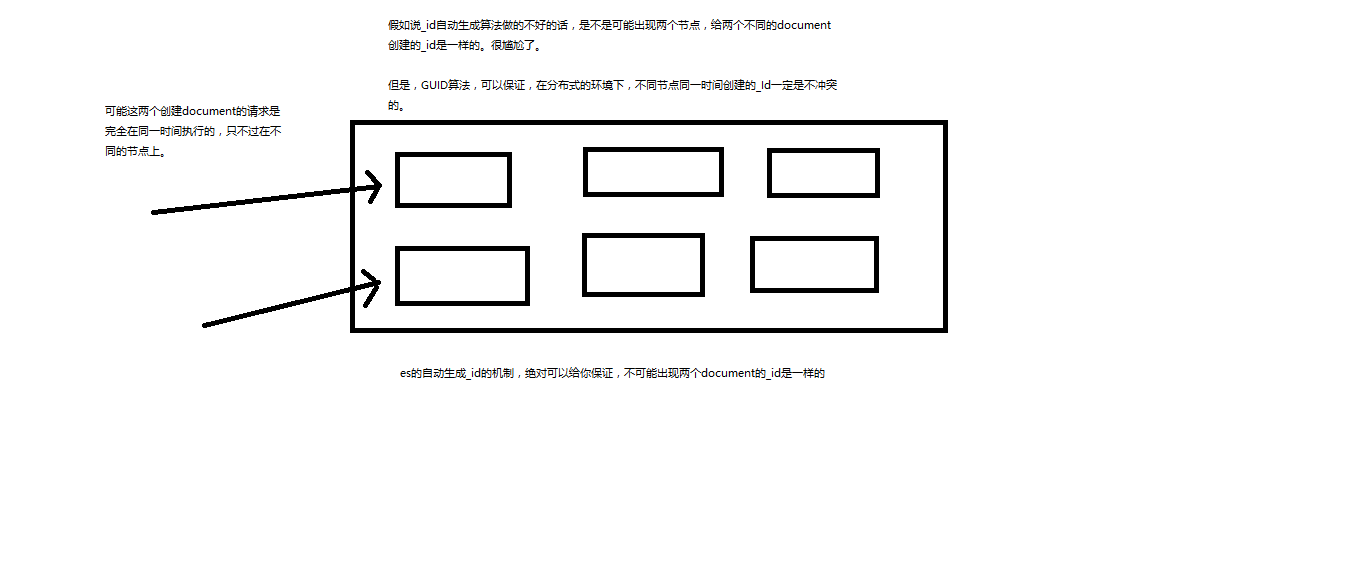
"failed": 0

},

"created": true

}

（2）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



# document的\_source元数据以及定制返回结果解析

课程大纲

## 1、\_source元数据

put /test\_index/test\_type/1

{

"test\_field1": "test field1",

"test\_field2": "test field2"

}

get /test\_index/test\_type/1

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "1",

"\_version": 2,

"found": true,

"\_source": {

"test\_field1": "test field1",

"test\_field2": "test field2"

}

}

\_source元数据：就是说，我们在创建一个document的时候，使用的那个放在request body中的json串，默认情况下，在get的时候，会原封不动的给我们返回回来。

------------------------------------------------------------------------------------------------------------------

## 2、定制返回结果

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

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "1",

"\_version": 2,

"found": true,

"\_source": {

"test\_field2": "test field2"

}

}

# document的全量替换、强制创建以及图解lazy delete机制

课程大纲

1、document的全量替换

2、document的强制创建

3、document的删除

------------------------------------------------------------------------------------------------------------------------

## 1、document的全量替换

（1）语法与创建文档是一样的，如果document id不存在，那么就是创建；如果document id已经存在，那么就是全量替换操作，替换document的json串内容，先delete原来的document（懒delete，当数据越来越多的时候，es在后台自动进行物理删除）,再创建新的document

（2）document是不可变的，如果要修改document的内容，第一种方式就是全量替换，直接对document重新建立索引，替换里面所有的内容

（3）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

------------------------------------------------------------------------------------------------------------------------

## 2、document的强制创建

（1）强制创建在document id不存在的情况下才会真正创建，如果document id已存在，那么它将不做任何创建或替换，强制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，如果进行强制创建呢？

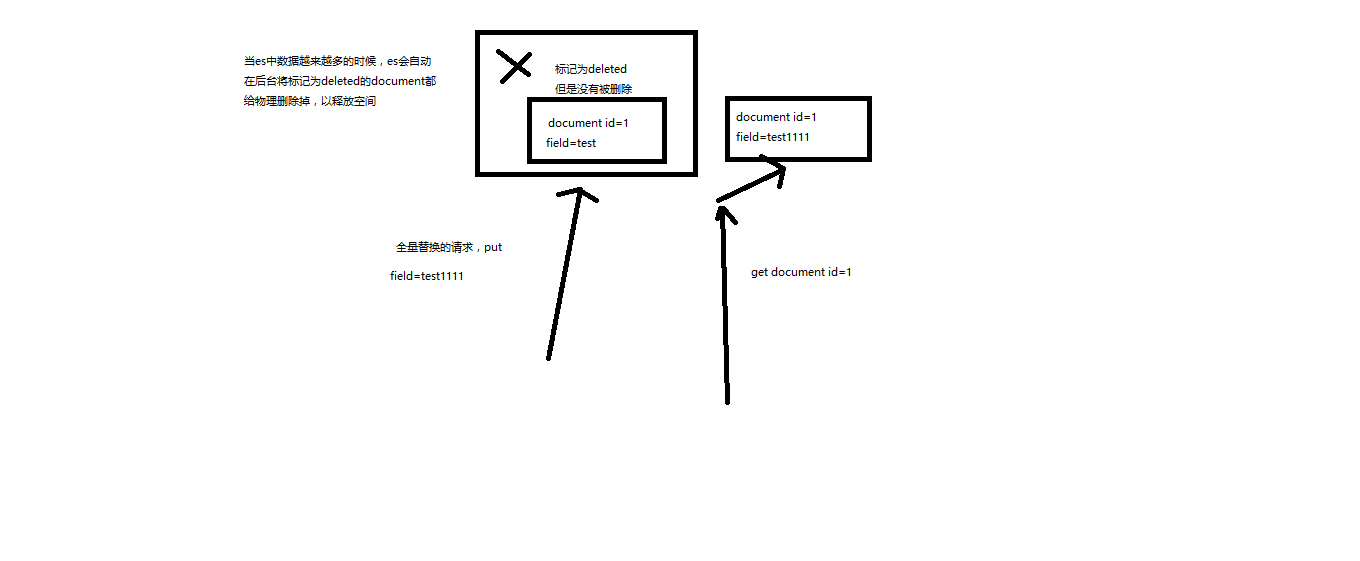
（2）PUT /index/type/id?op\_type=create，PUT /index/type/id/\_create

------------------------------------------------------------------------------------------------------------------------

## 3、document的删除

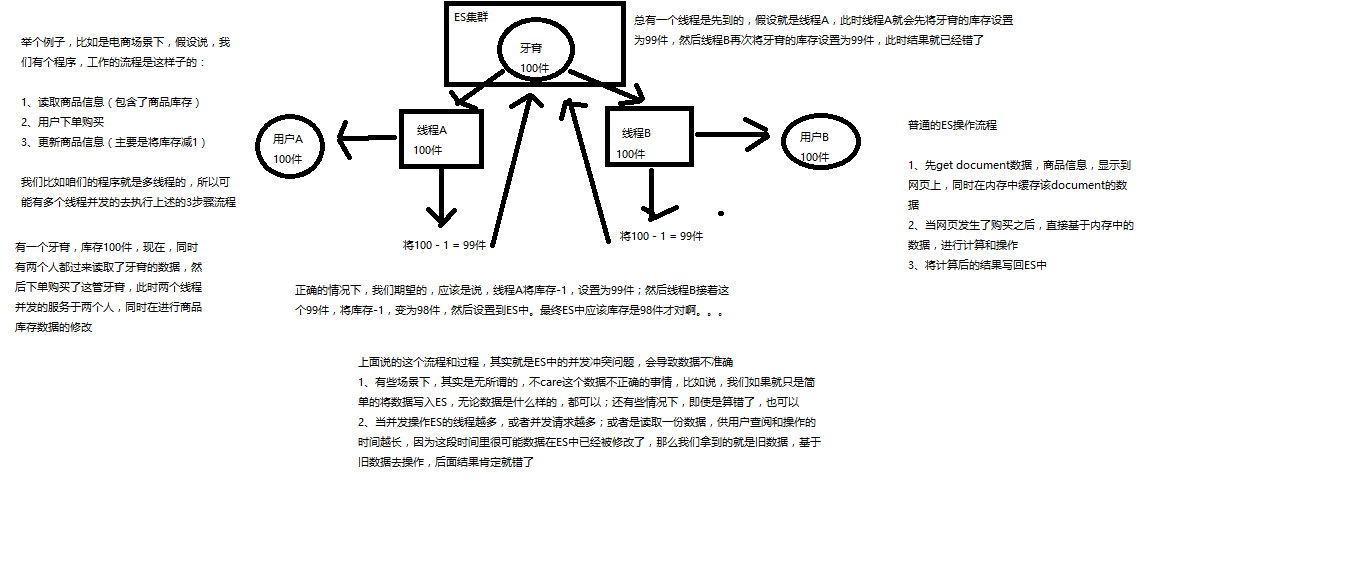
（1）DELETE /index/type/id

（2）不会物理删除，只会将其标记为deleted，当数据越来越多的时候，es在后台自动删进行物理除

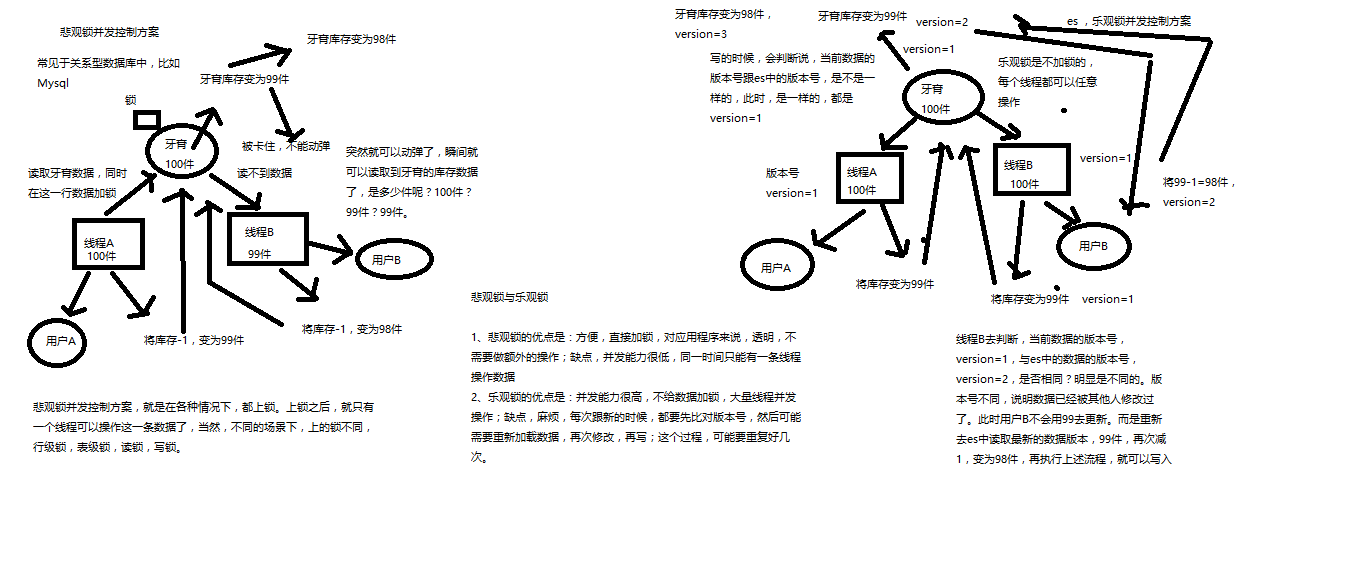


# 深度图解剖析Elasticsearch并发冲突问题

多线程环境下多个线程对共享数据的修改



# 深度图解剖析悲观锁与乐观锁两种并发控制方案



# 图解Elasticsearch内部如何基于\_version进行乐观锁并发控制

课程大纲

1、图解Elasticsearch内部如何基于\_version进行乐观锁并发控制

（1）\_version元数据

PUT /test\_index/test\_type/6

{

"test\_field": "test test"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "6",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

第一次创建一个document的时候，它的\_version内部版本号就是1；以后，每次对这个document执行修改或者删除操作，都会对这个\_version版本号自动加1；哪怕是删除，也会对这条数据的版本号加1

{

"found": true,

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "6",

"\_version": 4,

"result": "deleted",

"\_shards": {

"total": 2,

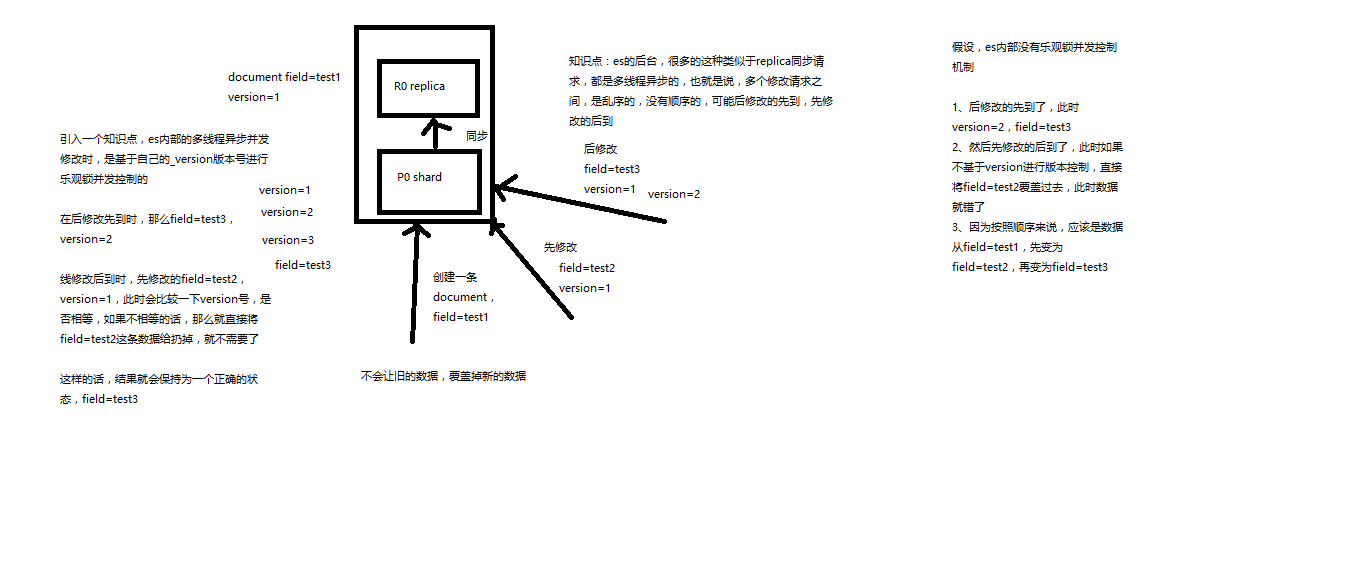
"successful": 1,

"failed": 0

}

}

我们会发现，在删除一个document之后，可以从一个侧面证明，它不是立即物理删除掉的，因为它的一些版本号等信息还是保留着的。先删除一条document，再重新创建这条document，其实会在delete version基础之上，再把version号加1



# 十九、上机动手实战演练基于\_version进行乐观锁并发控制

课程大纲

1、上机动手实战演练基于\_version进行乐观锁并发控制

（1）先构造一条数据出来

PUT /test\_index/test\_type/7

{

"test\_field": "test test"

}

（2）模拟两个客户端，都获取到了同一条数据

GET test\_index/test\_type/7

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_version": 1,

"found": true,

"\_source": {

"test\_field": "test test"

}

}

（3）其中一个客户端，先更新了一下这个数据

同时带上数据的版本号，确保说，es中的数据的版本号，跟客户端中的数据的版本号是相同的，才能修改

PUT /test\_index/test\_type/7?version=1

{

"test\_field": "test client 1"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

（4）另外一个客户端，尝试基于version=1的数据去进行修改，同样带上version版本号，进行乐观锁的并发控制

PUT /test\_index/test\_type/7?version=1

{

"test\_field": "test client 2"

}

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][7]: version conflict, current version [2] is different than the one provided [1]",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "3",

"index": "test\_index"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][7]: version conflict, current version [2] is different than the one provided [1]",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "3",

"index": "test\_index"

},

"status": 409

}

（5）在乐观锁成功阻止并发问题之后，尝试正确的完成更新

先查询出版本号

GET /test\_index/test\_type/7

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_version": 2,

"found": true,

"\_source": {

"test\_field": "test client 1"

}

}

基于最新的数据和版本号，去进行修改，修改后，带上最新的版本号，可能这个步骤会需要反复执行好几次，才能成功，特别是在多线程并发更新同一条数据很频繁的情况下

PUT /test\_index/test\_type/7?version=2

{

"test\_field": "test client 2"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_version": 3,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

# 二十、上机动手实战演练基于external version进行乐观锁并发控制

课程大纲

1、上机动手实战演练基于external version进行乐观锁并发控制

external version

es提供了一个feature，就是说，你可以不用它提供的内部\_version版本号来进行并发控制，可以基于你自己维护的一个版本号来进行并发控制。举个列子，加入你的数据在mysql里也有一份，然后你的应用系统本身就维护了一个版本号，无论是什么自己生成的，程序控制的。这个时候，你进行乐观锁并发控制的时候，可能并不是想要用es内部的\_version来进行控制，而是用你自己维护的那个version来进行控制。

?version=1

?version=1&version\_type=external

version\_type=external，唯一的区别在于，\_version，只有当你提供的version与es中的\_version一模一样的时候，才可以进行修改，只要不一样，就报错；当version\_type=external的时候，只有当你提供的version比es中的\_version大的时候，才能完成修改

es，\_version=1，?version=1，才能更新成功

es，\_version=1，?version>1&version\_type=external，才能成功，比如说?version=2&version\_type=external

（1）先构造一条数据

PUT /test\_index/test\_type/8

{

"test\_field": "test"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

（2）模拟两个客户端同时查询到这条数据

GET /test\_index/test\_type/8

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 1,

"found": true,

"\_source": {

"test\_field": "test"

}

}

（3）第一个客户端先进行修改，此时客户端程序是在自己的数据库中获取到了这条数据的最新版本号，比如说是2

PUT /test\_index/test\_type/8?version=2&version\_type=external

{

"test\_field": "test client 1"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

（4）模拟第二个客户端，同时拿到了自己数据库中维护的那个版本号，也是2，同时基于version=2发起了修改

PUT /test\_index/test\_type/8?version=2&version\_type=external

{

"test\_field": "test client 2"

}

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][8]: version conflict, current version [2] is higher or equal to the one provided [2]",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "1",

"index": "test\_index"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][8]: version conflict, current version [2] is higher or equal to the one provided [2]",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "1",

"index": "test\_index"

},

"status": 409

}

（5）在并发控制成功后，重新基于最新的版本号发起更新

GET /test\_index/test\_type/8

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 2,

"found": true,

"\_source": {

"test\_field": "test client 1"

}

}

PUT /test\_index/test\_type/8?version=3&version\_type=external

{

"test\_field": "test client 2"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 3,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

# 二十一：图解partial update实现原理以及动手实战演练

课程大纲

## 1、什么是partial update？

PUT /index/type/id，创建文档&替换文档，就是一样的语法

一般对应到应用程序中，每次的执行流程基本是这样的：

（1）应用程序先发起一个get请求，获取到document，展示到前台界面，供用户查看和修改

（2）用户在前台界面修改数据，发送到后台

（3）后台代码，会将用户修改的数据在内存中进行执行，然后封装好修改后的全量数据

（4）然后发送PUT请求，到es中，进行全量替换

（5）es将老的document标记为deleted，然后重新创建一个新的document

partial update

post /index/type/id/\_update

{

"doc": {

"要修改的少数几个field即可，不需要全量的数据"

}

}

或

post /index/type/id/

{

{

"要修改的少数几个field即可，不需要全量的数据"

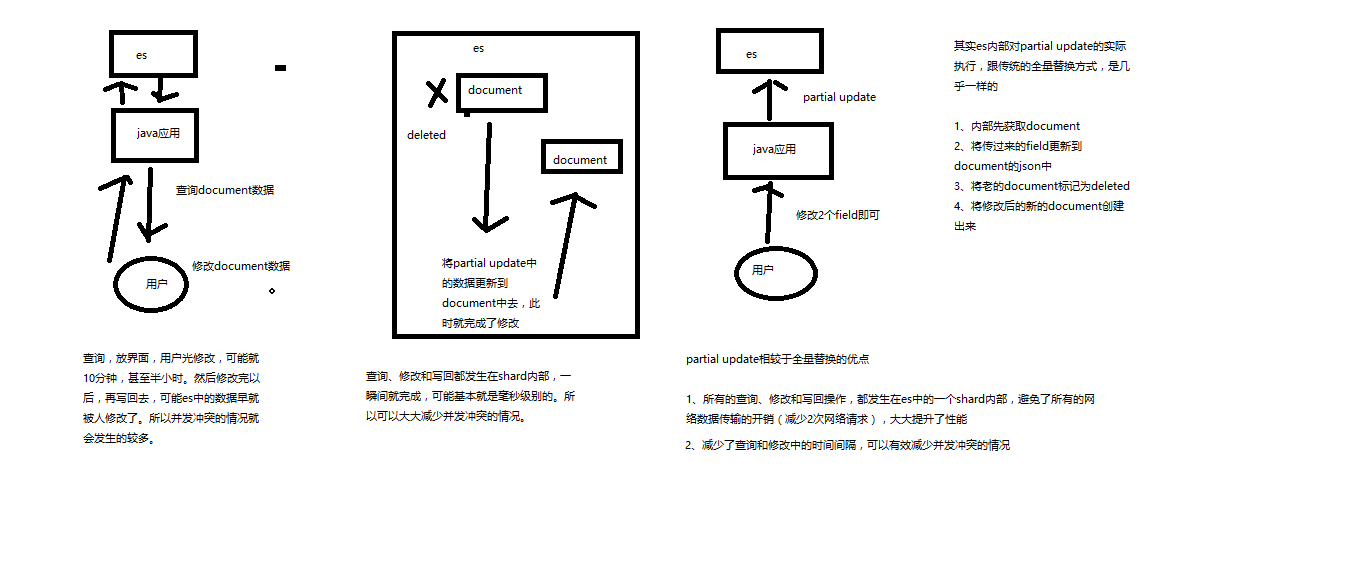
}

}

看起来，好像就比较方便了，每次就传递少数几个发生修改的field即可，不需要将全量的document数据发送过去

## 2、图解partial update实现原理以及其优点

partial update，看起来很方便的操作，实际内部的原理是什么样子的，然后它的优点是什么



## 3、上机动手实战演练partial update

PUT /test\_index/test\_type/10

{

"test\_field1": "test1",

"test\_field2": "test2"

}

POST /test\_index/test\_type/10/\_update

{

"doc": {

"test\_field2": "updated test2"

}

}

# 二十二、上机动手实战演练基于groovy脚本进行partial update

课程大纲

es，其实是有个内置的脚本支持的，可以基于groovy脚本实现各种各样的复杂操作

基于groovy脚本，如何执行partial update

es scripting module，我们会在高手进阶篇去讲解，这里就只是初步讲解一下

PUT /test\_index/test\_type/11

{

"num": 0,

"tags": []

}

（1）内置脚本

POST /test\_index/test\_type/11/\_update

{

"script" : "ctx.\_source.num+=1"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "11",

"\_version": 2,

"found": true,

"\_source": {

"num": 1,

"tags": []

}

}

（2）外部脚本

ctx.\_source.tags+=new\_tag

POST /test\_index/test\_type/11/\_update

{

"script": {

"lang": "groovy",

"file": "test-add-tags",

"params": {

"new\_tag": "tag1"

}

}

}

（3）用脚本删除文档

ctx.op = ctx.\_source.num == count ? 'delete' : 'none'

POST /test\_index/test\_type/11/\_update

{

"script": {

"lang": "groovy",

"file": "test-delete-document",

"params": {

"count": 1

}

}

}

（4）upsert操作

POST /test\_index/test\_type/11/\_update

{

"doc": {

"num": 1

}

}

{

"error": {

"root\_cause": [

{

"type": "document\_missing\_exception",

"reason": "[test\_type][11]: document missing",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "4",

"index": "test\_index"

}

],

"type": "document\_missing\_exception",

"reason": "[test\_type][11]: document missing",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "4",

"index": "test\_index"

},

"status": 404

}

如果指定的document不存在，就执行upsert中的初始化操作；如果指定的document存在，就执行doc或者script指定的partial update操作

POST /test\_index/test\_type/11/\_update

{

"script" : "ctx.\_source.num+=1",

"upsert": {

"num": 0,

"tags": []

}

}

# 二十三、图解partial update乐观锁并发控制原理以及相关操作讲解

课程大纲

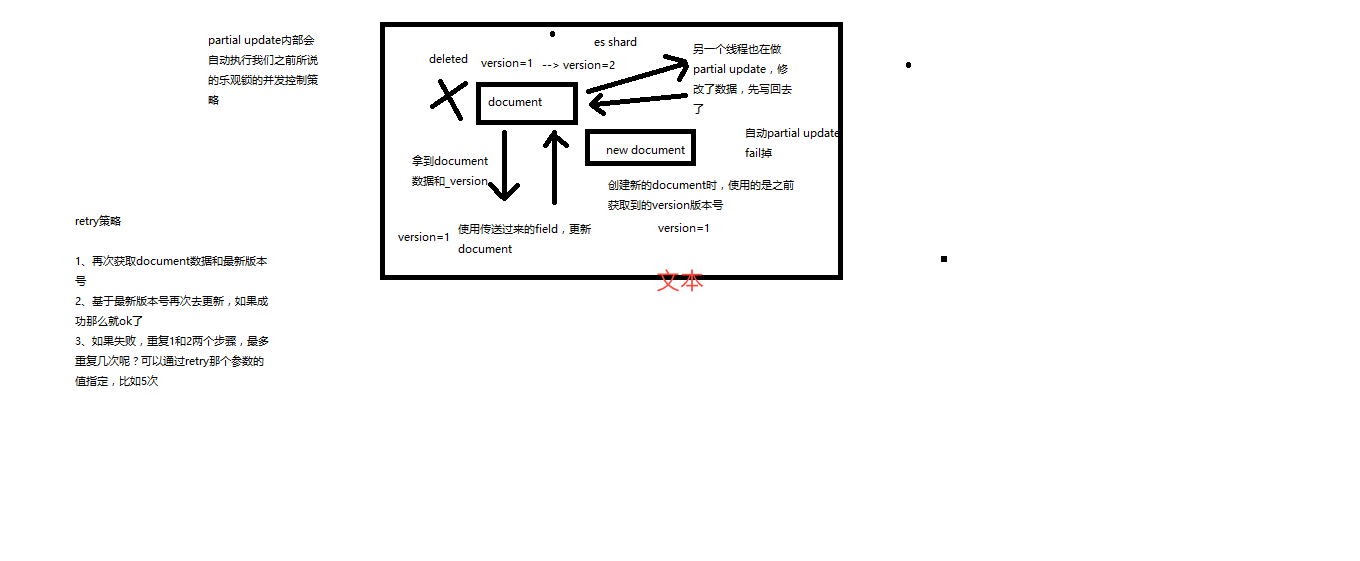
（1）partial update内置乐观锁并发控制

（2）retry\_on\_conflict

post /index/type/id/\_update?retry\_on\_conflict=5

（3）\_version

post /index/type/id/\_update?retry\_on\_conflict=5&version=6



# 二十四、上机动手实战演练mget批量查询api

课程大纲

## 1、批量查询的好处

就是一条一条的查询，比如说要查询100条数据，那么就要发送100次网络请求，这个开销还是很大的

如果进行批量查询的话，查询100条数据，就只要发送1次网络请求，网络请求的性能开销缩减100倍

## 2、mget的语法

（1）一条一条的查询

GET /test\_index/test\_type/1

GET /test\_index/test\_type/2

（2）mget批量查询

GET /\_mget

{

"docs" : [

{

"\_index" : "test\_index",

"\_type" : "test\_type",

"\_id" : 1

},

{

"\_index" : "test\_index",

"\_type" : "test\_type",

"\_id" : 2

}

]

}

{

"docs": [

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "1",

"\_version": 2,

"found": true,

"\_source": {

"test\_field1": "test field1",

"test\_field2": "test field2"

}

},

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "2",

"\_version": 1,

"found": true,

"\_source": {

"test\_content": "my test"

}

}

]

}

（3）如果查询的document是一个index下的不同type种的话

GET /test\_index/\_mget

{

"docs" : [

{

"\_type" : "test\_type",

"\_id" : 1

},

{

"\_type" : "test\_type",

"\_id" : 2

}

]

}

（4）如果查询的数据都在同一个index下的同一个type下，最简单了

GET /test\_index/test\_type/\_mget

{

"ids": [1, 2]

}

## 3、mget的重要性

可以说mget是很重要的，一般来说，在进行查询的时候，如果一次性要查询多条数据的话，那么一定要用batch批量操作的api，尽可能减少网络开销次数，可以极大地提升性能

# 二十五、上机动手实战演练bulk批量增删改

课程大纲

## 1、bulk语法

POST /\_bulk

{ "delete": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "3" }}

{ "create": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "12" }}

{ "test\_field": "test12" }

{ "index": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "2" }}

{ "test\_field": "replaced test2" }

{ "update": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "1", "\_retry\_on\_conflict" : 3} }

{ "doc" : {"test\_field2" : "bulk test1"} }

每一个操作（delete除外）要两个json串，语法如下：

{"action": {"metadata"}}

{"data"}

举例，比如你现在要创建一个文档，放bulk里面，看起来会是这样子的：

{"index": {"\_index": "test\_index", "\_type", "test\_type", "\_id": "1"}}

{"test\_field1": "test1", "test\_field2": "test2"}

有哪些类型的操作可以执行呢？

（1）delete：删除一个文档，只要1个json串就可以了

（2）create：PUT /index/type/id/\_create，强制创建

（3）index：普通的put操作，可以是创建文档，也可以是全量替换文档

（4）update：执行的partial update操作

bulk api对json的语法，有严格的要求，每个json串不能换行，只能放一行，同时一个json串和一个json串之间，必须有一个换行

{

"error": {

"root\_cause": [

{

"type": "json\_e\_o\_f\_exception",

"reason": "Unexpected end-of-input: expected close marker for Object (start marker at [Source: org.elasticsearch.transport.netty4.ByteBufStreamInput@5a5932cd; line: 1, column: 1])\n at [Source: org.elasticsearch.transport.netty4.ByteBufStreamInput@5a5932cd; line: 1, column: 3]"

}

],

"type": "json\_e\_o\_f\_exception",

"reason": "Unexpected end-of-input: expected close marker for Object (start marker at [Source: org.elasticsearch.transport.netty4.ByteBufStreamInput@5a5932cd; line: 1, column: 1])\n at [Source: org.elasticsearch.transport.netty4.ByteBufStreamInput@5a5932cd; line: 1, column: 3]"

},

"status": 500

}

{

"took": 41,

"errors": true,

"items": [

{

"delete": {

"found": true,

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "10",

"\_version": 3,

"result": "deleted",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"status": 200

}

},

{

"create": {

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "3",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true,

"status": 201

}

},

{

"create": {

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "2",

"status": 409,

"error": {

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][2]: version conflict, document already exists (current version [1])",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "2",

"index": "test\_index"

}

}

},

{

"index": {

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "4",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true,

"status": 201

}

},

{

"index": {

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "2",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false,

"status": 200

}

},

{

"update": {

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "1",

"\_version": 3,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"status": 200

}

}

]

}

bulk操作中，任意一个操作失败，是不会影响其他的操作的，但是在返回结果里，会告诉你异常日志

POST /test\_index/\_bulk

{ "delete": { "\_type": "test\_type", "\_id": "3" }}

{ "create": { "\_type": "test\_type", "\_id": "12" }}

{ "test\_field": "test12" }

{ "index": { "\_type": "test\_type" }}

{ "test\_field": "auto-generate id test" }

{ "index": { "\_type": "test\_type", "\_id": "2" }}

{ "test\_field": "replaced test2" }

{ "update": { "\_type": "test\_type", "\_id": "1", "\_retry\_on\_conflict" : 3} }

{ "doc" : {"test\_field2" : "bulk test1"} }

POST /test\_index/test\_type/\_bulk

{ "delete": { "\_id": "3" }}

{ "create": { "\_id": "12" }}

{ "test\_field": "test12" }

{ "index": { }}

{ "test\_field": "auto-generate id test" }

{ "index": { "\_id": "2" }}

{ "test\_field": "replaced test2" }

{ "update": { "\_id": "1", "\_retry\_on\_conflict" : 3} }

{ "doc" : {"test\_field2" : "bulk test1"} }

## 2、bulk size最佳大小

bulk request会加载到内存里，如果太大的话，性能反而会下降，因此需要反复尝试一个最佳的bulk size。一般从1000~5000条数据开始，尝试逐渐增加。另外，如果看大小的话，最好是在5~15MB之间。

# 二十六、阶段性总结以及什么是distributed document store

课程大纲

## 1、阶段性总结

1~8讲：快速入门了一下，es最基本的原理和操作

9~13讲：在入门之后，深入剖析ES的分布式的基本原理，

14~27讲：围绕着document这个东西，进行操作和分析

## 2、什么是distributed document store

到目前为止，你觉得你在学什么东西，给大家一个直观的感觉，好像已经知道了es是分布式的，包括一些基本的原理，然后花了不少时间在学习document本身相关的操作，增删改查。

Elasticsearch运行起来以后，第一个最核心的功能，就是一个分布式的文档数据存储系统。ES是分布式的。文档数据存储系统。文档数据，存储系统。

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。其实已经起到了一个什么样的效果呢？其实ES满足了这些功能，就可以说已经是一个NoSQL的存储系统了。

围绕着document在操作，其实就是把es当成了一个NoSQL存储引擎，一个可以存储文档类型数据的存储系统，在操作里面的document。

es可以作为一个分布式的文档存储系统，所以说，我们的应用系统，是不是就可以基于这个概念，去进行相关的应用程序的开发了。

什么类型的应用程序呢？

（1）数据量较大，es的分布式本质，可以帮助你快速进行扩容，承载大量数据

（2）数据结构灵活多变，随时可能会变化，而且数据结构之间的关系，非常复杂，如果我们用传统数据库，那是不是很坑，因为要面临大量的表

（3）对数据的相关操作，较为简单，比如就是一些简单的增删改查，用我们之前讲解的那些document操作就可以搞定

（4）NoSQL数据库，适用的也是类似于上面的这种场景

举个例子，比如说像一些网站系统，或者是普通的电商系统，博客系统，面向对象概念比较复杂，但是作为终端网站来说，没什么太复杂的功能，就是一些简单的CRUD操作，而且数据量可能还比较大。这个时候选用ES这种NoSQL型的数据存储，比传统的复杂的功能务必强大的支持SQL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。

# 二十七、深度图解剖析document数据路由原理

课程大纲

（1）document路由到shard上是什么意思？

（2）路由算法：shard = hash(routing) % number\_of\_primary\_shards

举个例子，一个index有3个primary shard，P0，P1，P2

每次增删改查一个document的时候，都会带过来一个routing number，默认就是这个document的\_id（可能是手动指定，也可能是自动生成）

routing = \_id，假设\_id=1

会将这个routing值，传入一个hash函数中，产出一个routing值的hash值，hash(routing) = 21

然后将hash函数产出的值对这个index的primary shard的数量求余数，21 % 3 = 0

就决定了，这个document就放在P0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。

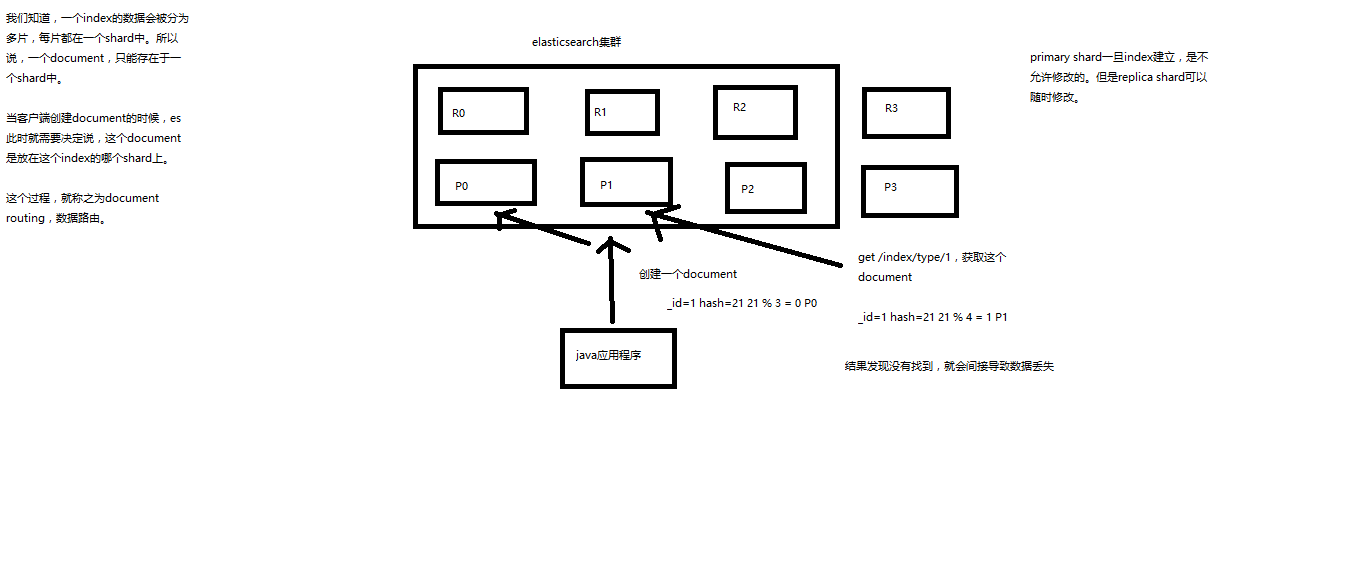
（3）\_id or custom routing value

默认的routing就是\_id

也可以在发送请求的时候，手动指定一个routing value，比如说put /index/type/id?routing=user\_id

手动指定routing value是很有用的，可以保证说，某一类document一定被路由到一个shard上去，那么在后续进行应用级别的负载均衡，以及提升批量读取的性能的时候，是很有帮助的

（4）primary shard数量不可变的谜底



# 二十八、document增删改内部原理图解揭秘

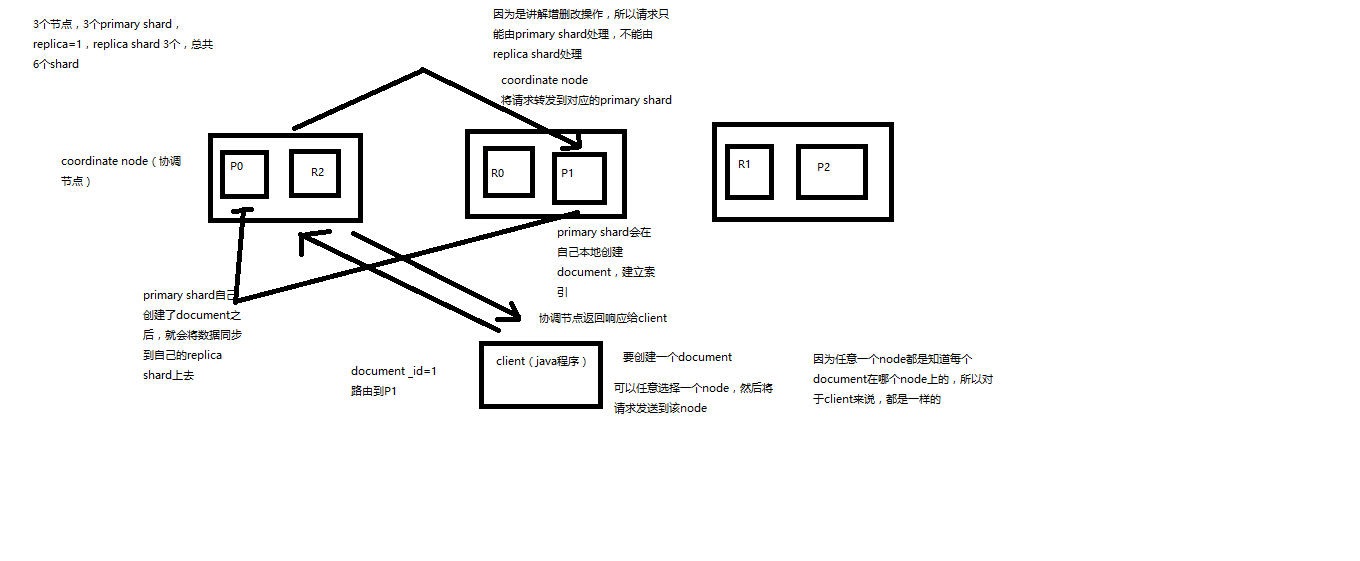
课程大纲

（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



# 二十九、图解写一致性原理以及quorum机制深入剖析

课程大纲

（1）consistency，one（primary shard），all（all shard），quorum（default）

我们在发送任何一个增删改操作的时候，比如说put /index/type/id，都可以带上一个consistency参数，指明我们想要的写一致性是什么？

put /index/type/id?consistency=quorum

one：要求我们这个写操作，只要有一个primary shard是active活跃可用的，就可以执行

all：要求我们这个写操作，必须所有的primary shard和replica shard都是活跃的，才可以执行这个写操作

quorum：默认的值，要求所有的shard中，必须是大部分的shard都是活跃的，可用的，才可以执行这个写操作

（2）quorum机制，写之前必须确保大多数shard都可用，int( (primary + number\_of\_replicas) / 2 ) + 1，当number\_of\_replicas>1时才生效

quroum = int( (primary + number\_of\_replicas) / 2 ) + 1

举个例子，3个primary shard，number\_of\_replicas=1，总共有3 + 3 \* 1 = 6个shard

quorum = int( (3 + 1) / 2 ) + 1 = 3

所以，要求6个shard中至少有3个shard是active状态的，才可以执行这个写操作

（3）如果节点数少于quorum数量，可能导致quorum不齐全，进而导致无法执行任何写操作

3个primary shard，replica=1，要求至少3个shard是active，3个shard按照之前学习的shard&replica机制，必须在不同的节点上，如果说只有2台机器的话，是不是有可能出现说，3个shard都没法分配齐全，此时就可能会出现写操作无法执行的情况

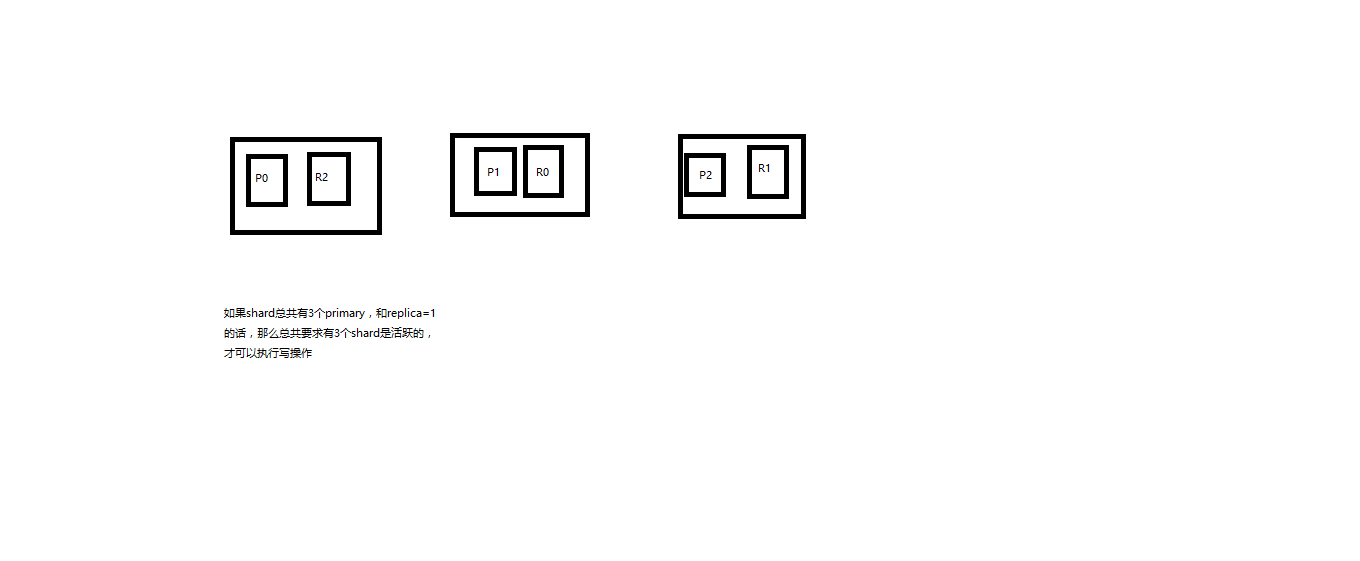
es提供了一种特殊的处理场景，就是说当number\_of\_replicas>1时才生效，因为假如说，你就一个primary shard，replica=1，此时就2个shard

(1 + 1 / 2) + 1 = 2，要求必须有2个shard是活跃的，但是可能就1个node，此时就1个shard是活跃的，如果你不特殊处理的话，导致我们的单节点集群就无法工作

（4）quorum不齐全时，wait，默认1分钟，timeout，100，30s

等待期间，期望活跃的shard数量可以增加，最后实在不行，就会timeout

我们其实可以在写操作的时候，加一个timeout参数，比如说put /index/type/id?timeout=30，这个就是说自己去设定quorum不齐全的时候，es的timeout时长，可以缩短，也可以增长



# 三十、document查询内部原理图解揭秘

课程大纲

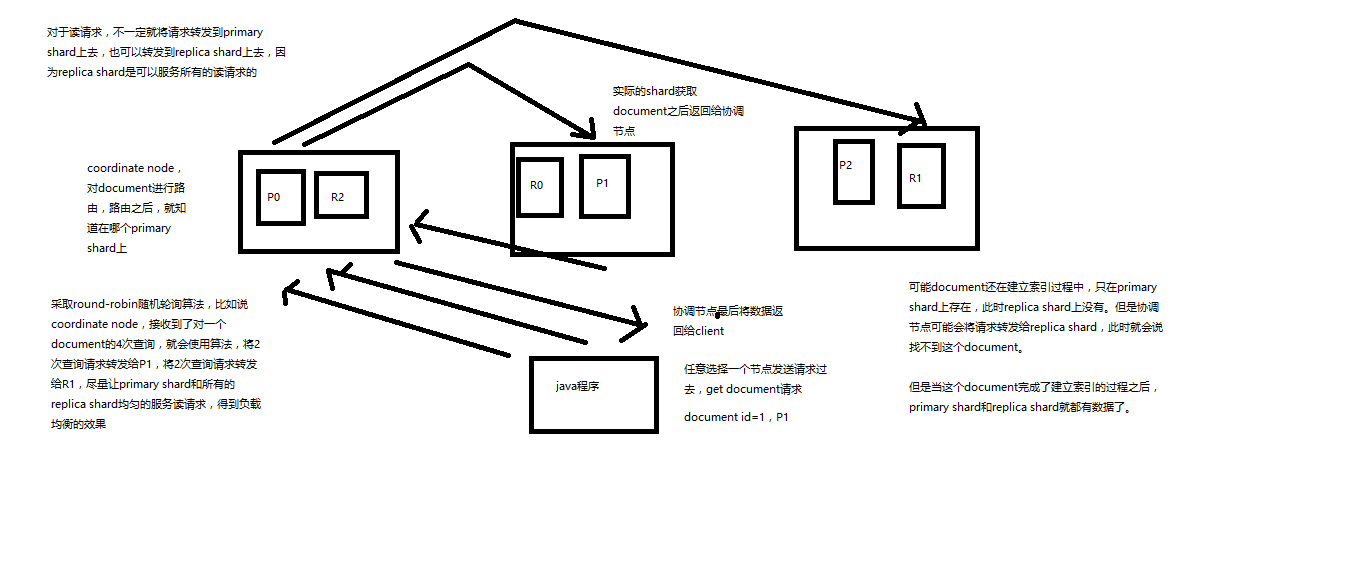
1、客户端发送请求到任意一个node，成为coordinate node

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



# 三十一、bulk api的奇特json格式与底层性能优化关系大揭秘

课程大纲

bulk api奇特的json格式

{"action": {"meta"}}\n

{"data"}\n

{"action": {"meta"}}\n

{"data"}\n

[{

"action": {

},

"data": {

}

}]

## 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

（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

{"action": {"meta"}}\n

{"data"}\n

{"action": {"meta"}}\n

{"data"}\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

1. 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

# 三十二、search结果深入解析（search timeout机制揭秘）

课程大纲

## 1、搜索请求结果解析

GET /\_search

{

"took": 6,

"timed\_out": false,

"\_shards": {

"total": 6,

"successful": 6,

"failed": 0

},

"hits": {

"total": 10,

"max\_score": 1,

"hits": [

{

"\_index": ".kibana",

"\_type": "config",

"\_id": "5.2.0",

"\_score": 1,

"\_source": {

"buildNum": 14695

}

}

]

}

}

took：整个搜索请求花费了多少毫秒

hits.total：本次搜索，返回了几条结果

hits.max\_score：本次搜索的所有结果中，最大的相关度分数是多少，每一条document对于search的相关度，越相关，\_score分数越大，排位越靠前

hits.hits：默认查询前10条数据，完整数据，\_score降序排序

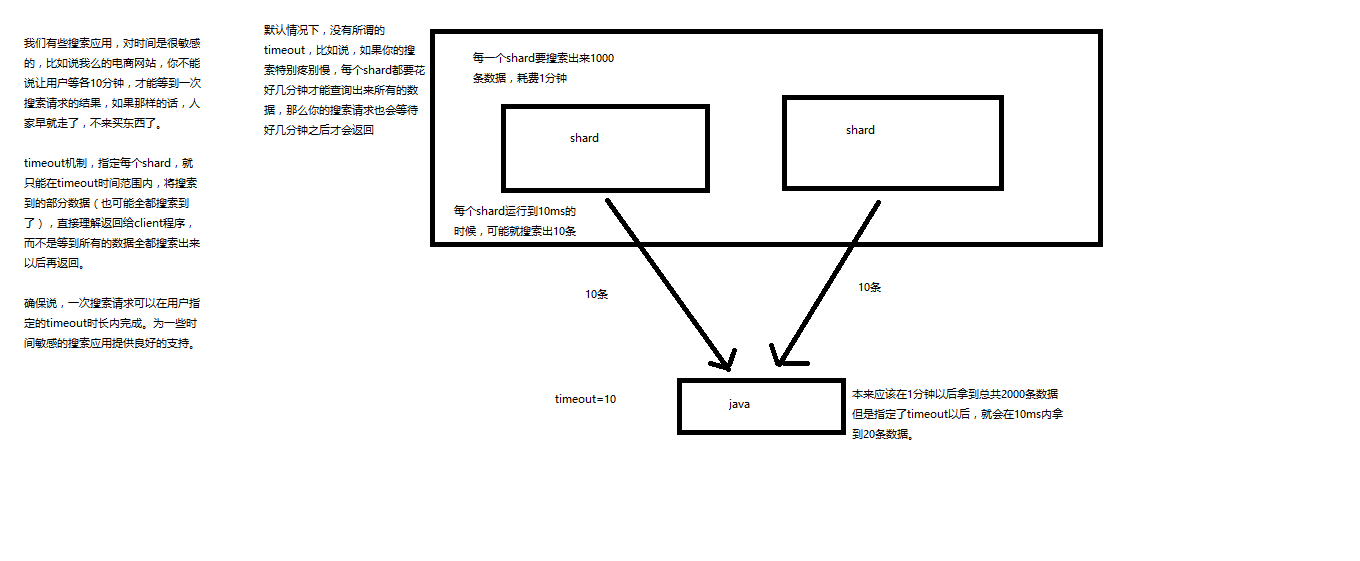
shards：shards fail的条件（primary和replica全部挂掉），不影响其他shard。默认情况下来说，一个搜索请求，会打到一个index的所有primary shard上去，当然了，每个primary shard都可能会有一个或多个replic shard，所以请求也可以到primary shard的其中一个replica shard上去。

## 2、timeout机制

timeout：默认无timeout，latency平衡completeness，手动指定timeout，timeout查询执行机制

timeout=10ms，timeout=1s，timeout=1m

GET /\_search?timeout=10m



# 三十三、multi-index&multi-type搜索模式解析

课程大纲

1、multi-index和multi-type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，所有type下的所有数据都搜索出来

/index1/\_search：指定一个index，搜索其下所有type的数据

/index1,index2/\_search：同时搜索两个index下的数据

/\*1,\*2/\_search：按照通配符去匹配多个索引

/index1/type1/\_search：搜索一个index下指定的type的数据

/index1/type1,type2/\_search：可以搜索一个index下多个type的数据

/index1,index2/type1,type2/\_search：搜索多个index下的多个type的数据

/\_all/type1,type2/\_search：\_all，可以代表搜索所有index下的指定type的数据

# 三十四、分页搜索以及deep paging性能问题深度图解揭秘

课程大纲

1、讲解如何使用es进行分页搜索的语法

size，from

GET /\_search?size=10

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

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

分页的上机实验

GET /test\_index/test\_type/\_search

"hits": {

"total": 9,

"max\_score": 1,

我们假设将这9条数据分成3页，每一页是3条数据，来实验一下这个分页搜索的效果

GET /test\_index/test\_type/\_search?from=0&size=3

{

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 9,

"max\_score": 1,

"hits": [

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_score": 1,

"\_source": {

"test\_field": "test client 2"

}

},

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "6",

"\_score": 1,

"\_source": {

"test\_field": "tes test"

}

},

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "4",

"\_score": 1,

"\_source": {

"test\_field": "test4"

}

}

]

}

}

第一页：id=8,6,4

GET /test\_index/test\_type/\_search?from=3&size=3

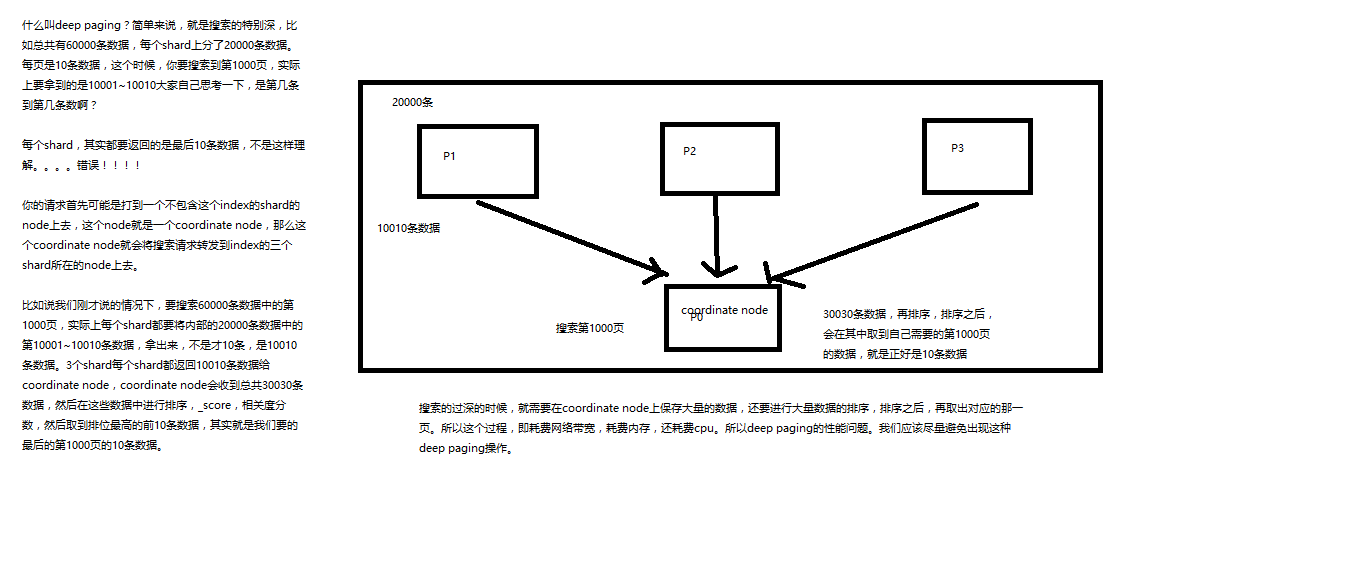
第二页：id=2,自动生成,7

GET /test\_index/test\_type/\_search?from=6&size=3

第三页：id=1,11,3

2、什么是deep paging问题？为什么会产生这个问题，它的底层原理是什么？

deep paging性能问题，以及原理深度图解揭秘，很高级的知识点



# 三十五、快速掌握query string search语法以及\_all metadata原理揭秘

课程大纲

## 1、query string基础语法

GET /test\_index/test\_type/\_search?q=test\_field:test

GET /test\_index/test\_type/\_search?q=+test\_field:test

GET /test\_index/test\_type/\_search?q=-test\_field:test：不包含test\_filed字段或包含test\_field字段但test\_field字段不包含test

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

## 2、\_all metadata的原理和作用

GET /test\_index/test\_type/\_search?q=test

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。我们在进行这种搜索的时候，难道是对document中的每一个field都进行一次搜索吗？不是的

es中的\_all元数据，在建立索引的时候，我们插入一条document，它里面包含了多个field，此时，es会自动将多个field的值，全部用字符串的方式串联起来，变成一个长的字符串，作为\_all field的值，同时建立索引

后面如果在搜索的时候，没有对某个field指定搜索，就默认搜索\_all field，其中是包含了所有field的值的

举个例子

{

"name": "jack",

"age": 26,

"email": "jack@sina.com",

"address": "guamgzhou"

}

"jack 26 jack@sina.com guangzhou"，作为这一条document的\_all field的值，同时进行分词后建立对应的倒排索引

生产环境不使用

# 三十六、用一个例子告诉你mapping到底是什么

课程大纲

插入几条数据，让es自动为我们建立一个索引

PUT /website/article/1

{

"post\_date": "2017-01-01",

"title": "my first article",

"content": "this is my first article in this website",

"author\_id": 11400

}

PUT /website/article/2

{

"post\_date": "2017-01-02",

"title": "my second article",

"content": "this is my second article in this website",

"author\_id": 11400

}

PUT /website/article/3

{

"post\_date": "2017-01-03",

"title": "my third article",

"content": "this is my third article in this website",

"author\_id": 11400

}

尝试各种搜索

GET /website/article/\_search?q=2017 3条结果

GET /website/article/\_search?q=2017-01-01 3条结果

GET /website/article/\_search?q=post\_date:2017-01-01 1条结果

GET /website/article/\_search?q=post\_date:2017 1条结果

查看es自动建立的mapping，带出什么是mapping的知识点

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

dynamic mapping，自动为我们建立index，创建type，以及type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

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和type，以及type对应的mapping

GET /website/\_mapping/article

{

"website": {

"mappings": {

"article": {

"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

}

}

}

}

}

}

}

}

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。

# 三十七、精确匹配与全文搜索的对比分析

课程大纲

## 1、exact value

2017-01-01，exact value，搜索的时候，必须输入2017-01-01，才能搜索出来

如果你输入一个01，是搜索不出来的

## 2、full text

（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

2017-01-01，2017 01 01，搜索2017，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

# 三十八、倒排索引核心原理快速揭秘

课程大纲

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

word 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

这个是不是我们想要的搜索结果？？？绝对不是，因为在我们看来，mother和mom有区别吗？同义词，都是妈妈的意思。like和liked有区别吗？没有，都是喜欢的意思，只不过一个是现在时，一个是过去时。little和small有区别吗？同义词，都是小小的。dog和dogs有区别吗？狗，只不过一个是单数，一个是复数。

normalization，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

I \* \*

really \*

like \* \* liked --> like

my \* \*

little \* small --> little

dog \* \* dogs --> dog

and \*

think \*

mom \* \*

also \*

them \*

He \*

never \*

any \*

so \*

hope \*

that \*

will \*

not \*

expect \*

me \*

to \*

him \*

mother like little dog，分词，normalization

mother --> mom

like --> like

little --> little

dog --> dog

doc1和doc2都会搜索出来

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.

# 三十九、分词器的内部组成到底是什么，以及内置分词器的介绍

课程大纲

1、什么是分词器

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），分瓷器

recall，召回率：搜索的时候，增加能够搜索到的结果的数量

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（<span>hello<span> --> hello），& --> and（I&you --> I and you）

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

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

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

2、内置分词器的介绍

Set the shape to semi-transparent by calling set\_trans(5)

standard analyzer：set, the, shape, to, semi, transparent, by, calling, set\_trans, 5（默认的是standard）

simple analyzer：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

# 四十、query string的分词以及mapping引入案例遗留问题的大揭秘

课程大纲

1、query 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

date：exact value

\_all：full text

比如我们有一个document，其中有一个field，包含的value是：hello you and me，建立倒排索引

我们要搜索这个document对应的index，搜索文本是hello me，这个搜索文本就是query string

query string，默认情况下，es会使用它对应的field建立倒排索引时相同的分词器去进行分词，分词和normalization，只有这样，才能实现正确的搜索

我们建立倒排索引的时候，将dogs --> dog，结果你搜索的时候，还是一个dogs，那不就搜索不到了吗？所以搜索的时候，那个dogs也必须变成dog才行。才能搜索到。

知识点：不同类型的field，可能有的就是full text，有的就是exact value

post\_date，date：exact value

\_all：full text，分词，normalization

2、mapping引入案例遗留问题大揭秘

GET /\_search?q=2017

搜索的是\_all field，document所有的field都会拼接成一个大串，进行分词

2017-01-02 my second article this is my second article in this website 11400

doc1 doc2 doc3

2017 \* \* \*

01 \*

02 \*

03 \*

\_all，2017，自然会搜索到3个docuemnt

GET /\_search?q=2017-01-01

\_all，2017-01-01，query string会用跟建立倒排索引一样的分词器去进行分词

2017

01

01

GET /\_search?q=post\_date:2017-01-01

date，会作为exact value去建立索引

doc1 doc2 doc3

2017-01-01 \*

2017-01-02 \*

2017-01-03 \*

post\_date:2017-01-01，2017-01-01，doc1一条document

GET /\_search?q=post\_date:2017，这个在这里不讲解，因为是es 5.2以后做的一个优化

3、测试分词器

GET /\_analyze

{

"analyzer": "standard",

"text": "Text to analyze"

}

# 四十一、什么是mapping再次回炉透彻理解

课程大纲

（1）往es里面直接插入数据，es会自动建立索引，同时建立type以及对应的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type的mapping，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，等等

mapping，就是index的type的元数据，每个type都有一个自己的mapping，决定了数据类型，建立倒排索引的行为，还有进行搜索的行为

# 四十二、mapping的核心数据类型以及dynamic mapping

课程大纲

## 1、核心的数据类型

string

byte，short，integer，long

float，double

boolean

date

## 2、dynamic mapping

true or false --> boolean

123 --> long

123.45 --> double

2017-01-01 --> date

"hello world" --> string/text

## 3、查看mapping

GET /index/\_mapping/type

# 四十三、手动建立和修改mapping以及定制string类型数据是否分词

课程大纲

## 1、如何建立索引

analyzed：分词

not\_analyzed：不分词（exact value）

no：禁止索引和搜索

## 2、修改mapping

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

PUT /website

{

"mappings": {

"article": {

"properties": {

"author\_id": {

"type": "long"

},

"title": {

"type": "text",

"analyzer": "english"

},

"content": {

"type": "text"

},

"post\_date": {

"type": "date"

},

"publisher\_id": {

"type": "text",

"index": "not\_analyzed"

}

}

}

}

}

//尝试修改 field mapping

PUT /website

{

"mappings": {

"article": {

"properties": {

"author\_id": {

"type": "text"

}

}

}

}

}

{

"error": {

"root\_cause": [

{

"type": "index\_already\_exists\_exception",

"reason": "index [website/co1dgJ-uTYGBEEOOL8GsQQ] already exists",

"index\_uuid": "co1dgJ-uTYGBEEOOL8GsQQ",

"index": "website"

}

],

"type": "index\_already\_exists\_exception",

"reason": "index [website/co1dgJ-uTYGBEEOOL8GsQQ] already exists",

"index\_uuid": "co1dgJ-uTYGBEEOOL8GsQQ",

"index": "website"

},

"status": 400

}

PUT /website/\_mapping/article

{

"properties" : {

"new\_field" : {

"type" : "string",

"index": "not\_analyzed"

}

}

}

## 3、测试mapping

GET /website/\_analyze

{

"field": "content",

"text": "my-dogs"

}

GET website/\_analyze

{

"field": "new\_field",

"text": "my dogs"

}

{

"error": {

"root\_cause": [

{

"type": "remote\_transport\_exception",

"reason": "[4onsTYV][127.0.0.1:9300][indices:admin/analyze[s]]"

}

],

"type": "illegal\_argument\_exception",

"reason": "Can't process field [new\_field], Analysis requests are only supported on tokenized fields"

},

"status": 400

}

# 四十四、mapping复杂数据类型以及object类型数据底层结构大揭秘

课程大纲

## 1、multivalue field

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

建立索引时与string是一样的，数据类型不能混

## 2、empty field

null，[]，[null]

## 3、object field

PUT /company/employee/1

{

"address": {

"country": "china",

"province": "guangdong",

"city": "guangzhou"

},

"name": "jack",

"age": 27,

"join\_date": "2017-01-01"

}

address：object类型

{

"company": {

"mappings": {

"employee": {

"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

}

}

}

}

}

}

}

}

{

"address": {

"country": "china",

"province": "guangdong",

"city": "guangzhou"

},

"name": "jack",

"age": 27,

"join\_date": "2017-01-01"

}

{

"name": [jack],

"age": [27],

"join\_date": [2017-01-01],

"address.country": [china],

"address.province": [guangdong],

"address.city": [guangzhou]

}

{

"authors": [

{ "age": 26, "name": "Jack White"},

{ "age": 55, "name": "Tom Jones"},

{ "age": 39, "name": "Kitty Smith"}

]

}

{

"authors.age": [26, 55, 39],

"authors.name": [jack, white, tom, jones, kitty, smith]

}

# 四十五、search api的基础语法介绍

课程大纲

## 1、search api的基本语法（总结）

GET /search

{}

GET /index1,index2/type1,type2/search

{}

GET /\_search

{

"from": 0,

"size": 10

}

## 2、http协议中get是否可以带上request body

HTTP协议，一般不允许get请求带上request body，但是因为get更加适合描述查询数据的操作，因此还是这么用了

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

POST /\_search

{

"from":0,

"size":10

}

碰巧，很多浏览器，或者是服务器，也都支持GET+request body模式

如果遇到不支持的场景，也可以用POST /\_search

# 四十六、快速上机动手实战Query DSL搜索语法

课程大纲

1、一个例子让你明白什么是Query DSL

GET /\_search

{

"query": {

"match\_all": {}

}

}

2、Query DSL的基本语法

{

QUERY\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

{

QUERY\_NAME: {

FIELD\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

}

示例：

GET /test\_index/test\_type/\_search

{

"query": {

"match": {

"test\_field": "test"

}

}

}

3、如何组合多个搜索条件

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

{

"took": 1,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 1,

"hits": [

{

"\_index": "website",

"\_type": "article",

"\_id": "2",

"\_score": 1,

"\_source": {

"title": "my hadoop article",

"content": "hadoop is very bad",

"author\_id": 111

}

},

{

"\_index": "website",

"\_type": "article",

"\_id": "1",

"\_score": 1,

"\_source": {

"title": "my elasticsearch article",

"content": "es is very bad",

"author\_id": 110

}

},

{

"\_index": "website",

"\_type": "article",

"\_id": "3",

"\_score": 1,

"\_source": {

"title": "my elasticsearch article",

"content": "es is very goods",

"author\_id": 111

}

}

]

}

}

GET /website/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"title": "elasticsearch"

}

}

],

"should": [

{

"match": {

"content": "elasticsearch"

}

}

],

"must\_not": [

{

"match": {

"author\_id": 111

}

}

]

}

}

}

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

}

}

}

# 四十七、filter与query深入对比解密：相关度，性能

课程大纲

## 1、filter与query示例

PUT /company/employee/2

{

"address": {

"country": "china",

"province": "jiangsu",

"city": "nanjing"

},

"name": "tom",

"age": 30,

"join\_date": "2016-01-01"

}

PUT /company/employee/3

{

"address": {

"country": "china",

"province": "shanxi",

"city": "xian"

},

"name": "marry",

"age": 35,

"join\_date": "2015-01-01"

}

搜索请求：年龄必须大于等于30，同时join\_date必须是2016-01-01

GET /company/employee/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"join\_date": "2016-01-01"

}

}

],

"filter": {

"range": {

"age": {

"gte": 30

}

}

}

}

}

}

## 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

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

除非是你的这些搜索条件，你希望越符合这些搜索条件的document越排在前面返回，那么这些搜索条件要放在query中；如果你不希望一些搜索条件来影响你的document排序，那么就放在filter中即可

## 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

# 四十八、上机动手实战常用的各种query搜索语法

课程大纲

1、match all

GET /\_search

{

"query": {

"match\_all": {}

}

}

2、match

GET /\_search

{

"query": { "match": { "title": "my elasticsearch article" }}

}

3、multi match

GET /test\_index/test\_type/\_search

{

"query": {

"multi\_match": {

"query": "test",

"fields": ["test\_field", "test\_field1"]

}

}

}

4、range query

查询指定区间内的数字或时间，query和filter都支持，一般是filter用得多，允许的操作符如下：

GET /company/employee/\_search

{

"query": {

"range": {

"age": {

"gte": 30

}

}

}

}

5、term query

用于精确值匹配，精确值可以是数字，日期，boolean或keyword类型的字符串

建立索引时mapping设置为not\_analyzed时，match等同于term，用得多的是match和range。

GET /test\_index/test\_type/\_search

{

"query": {

"term": {

"test\_field": "test hello"

}

}

}

6、terms query

跟term类似，只是允许一次指定多个值进行匹配，只要有任何一个匹配上，都满足条件

GET /\_search

{

"query": { "terms": { "tag": [ "search", "full\_text", "nosql" ] }}

}

7、exist query（2.x中的查询，现在已经不提供了）

# 四十九、上机动手实战多搜索条件组合查询

课程大纲

GET /website/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"title": "elasticsearch"

}

}

],

"should": [

{

"match": {

"content": "elasticsearch"

}

}

],

"must\_not": [

{

"match": {

"author\_id": 111

}

}

]

}

}

}

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }}

],

"filter": {

"range": { "date": { "gte": "2014-01-01" }}

}

}

}

bool

must，must\_not，should，filter

每个子查询都会计算一个document针对它的相关度分数，然后bool综合所有分数，合并为一个分数，当然filter是不会计算分数的

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }}

],

"filter": {

"bool": {

"must": [

{ "range": { "date": { "gte": "2014-01-01" }}},

{ "range": { "price": { "lte": 29.99 }}}

],

"must\_not": [

{ "term": { "category": "ebooks" }}

]

}

}

}

}

GET /company/employee/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"age": {

"gte": 30

}

}

}

}

}

}

# 五十、上机动手实战如何定位不合法的搜索以及其原因

课程大纲

GET /test\_index/test\_type/\_validate/query?explain

{

"query": {

"math": {

"test\_field": "test"

}

}

}

{

"valid": false,

"error": "org.elasticsearch.common.ParsingException: no [query] registered for [math]"

}

GET /test\_index/test\_type/\_validate/query?explain

{

"query": {

"match": {

"test\_field": "test"

}

}

}

{

"valid": true,

"\_shards": {

"total": 1,

"successful": 1,

"failed": 0

},

"explanations": [

{

"index": "test\_index",

"valid": true,

"explanation": "+test\_field:test #(#\_type:test\_type)"

}

]

}

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

# 五十一、初识搜素引擎\_上机动手实战如何定制搜索结果的排序规则

课程大纲

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 /\_search

{

"query" : {

"bool" : {

"filter" : {

"term" : {

"author\_id" : 1

}

}

}

}

}

当然，也可以是constant\_score

GET /\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"author\_id" : 1

}

}

}

}

}

2、定制排序规则

GET /company/employee/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"age": {

"gte": 30

}

}

}

}

},

"sort": [

{

"join\_date": {

"order": "asc"

}

}

]

}

# 五十二、解密如何将一个field索引两次来解决字符串排序问题

课程大纲

如果对一个string 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string 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

PUT /website

{

"mappings": {

"article": {

"properties": {

"title": {

"type": "text",

"fields": {

"raw": {

"type": "string",

"index": "not\_analyzed"

}

},

"fielddata": true

},

"content": {

"type": "text"

},

"post\_date": {

"type": "date"

},

"author\_id": {

"type": "long"

}

}

}

}

}

PUT /website/article/1

{

"title": "first article",

"content": "this is my second article",

"post\_date": "2017-01-01",

"author\_id": 110

}

{

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 1,

"hits": [

{

"\_index": "website",

"\_type": "article",

"\_id": "2",

"\_score": 1,

"\_source": {

"title": "first article",

"content": "this is my first article",

"post\_date": "2017-02-01",

"author\_id": 110

}

},

{

"\_index": "website",

"\_type": "article",

"\_id": "1",

"\_score": 1,

"\_source": {

"title": "second article",

"content": "this is my second article",

"post\_date": "2017-01-01",

"author\_id": 110

}

},

{

"\_index": "website",

"\_type": "article",

"\_id": "3",

"\_score": 1,

"\_source": {

"title": "third article",

"content": "this is my third article",

"post\_date": "2017-03-01",

"author\_id": 110

}

}

]

}

}

GET /website/article/\_search

{

"query": {

"match\_all": {}

},

"sort": [

{

"title.raw": {

"order": "desc"

}

}

]

}

# 五十五、相关度评分TF&IDF算法独家解密

课程大纲

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

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

搜索请求：hello world

doc1：hello you, 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

搜索请求：hello world

doc1：hello, today is very good

doc2：hi world, how are you

比如说，在index中有1万条document，hello这个单词在所有的document中，一共出现了1000次；world这个单词在所有的document中，一共出现了100次

doc2更相关

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

搜索请求：hello world

doc1：{ "title": "hello article", "content": "babaaba 1万个单词" }

doc2：{ "title": "my article", "content": "blablabala 1万个单词，hi world" }

hello world在整个index中出现的次数是一样多的

doc1更相关，title field更短

2、\_score是如何被计算出来的

GET /test\_index/test\_type/\_search?explain

{

"query": {

"match": {

"test\_field": "test hello"

}

}

}

3、分析一个document是如何被匹配上的

GET /test\_index/test\_type/6/\_explain

{

"query": {

"match": {

"test\_field": "test hello"

}

}

}