# ELK学习笔记（来自龙果学院，内容非原创。）

## 什么是elasticsearch？

elasticsearch是分布式，高性能，高可用，可伸缩的搜索和分析系统

### 什么是搜索？

比如最常用的百度搜索，我们想找寻任何的信息的时候，就会去百度搜索一下，比如找电影，找书，找新闻。搜索简单来说就是根据关键字，快速查找我们想要的信息。

### 如果用数据库做搜索会怎么样？

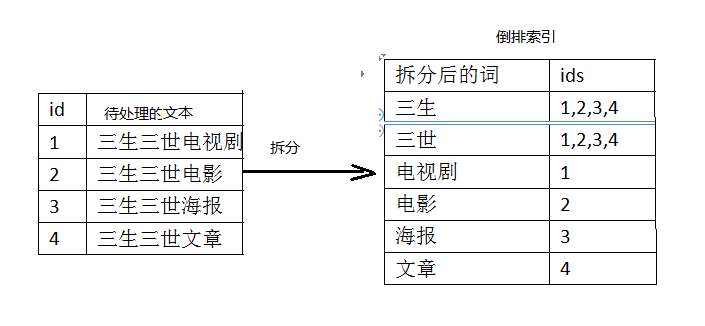
1、用数据库的话，搜索得用like查询，会全表扫描，拿到每一条数据都会拿到文本再进行比较。

比如某种表有商品名称和商品描述，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，有成千上万个字符，这个时候，每次都要对每条记录的所有文本进行扫描判断，是否包含指定的搜索关键字。

1. 无法将词拆分，搜索出尽可能多的内容，比如搜索生化危机，但是使用生化机去搜索，是搜索不到的。

### 什么是全文检索和Lucene？

倒排索引：会对每一个词进行拆分，把拆出来的词放到索引中，比如：



现在用“三生”去搜索，就可以到索引中找，可以找到id为1,2,3,4，然后根据id去查询即可。扫描索引比扫描记录要快得多。而且找到之后，就不用继续扫描。如果是存储在关系型数据库中，一共有100万条，按照之前的思路，其实就要扫描100万次，而且每次扫描，都需要匹配那个文本所有字符，确认是否包含搜索的关键词，而且还不能将搜索词拆解进行检索。

上述过程就是全文检索。

lucene：

就是一个jar包，封装好了建立倒排索引，搜索的各种算法。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。我们就可以用lucene提供的一些api来针对磁盘上的索引数据，进行搜索。

### 什么是elasticsearch？

使用lucene只能部署在单台机器上，如果数据量过大，无法满足要求，如果多台机器，每天机器上都装lucene，如果搜索，需要考虑如下问题：

1. 多台机器如何通信？
2. 搜索的时候去哪台机器上找？
3. 怎么保持高可用性，比如宕机，怎么保证数据仍可被搜索？
4. 索引怎么放？

elasticsearch就是用来解决上面的问题的。我们只需发送请求，es会把数据返回给我们。

1. 自动维护数据分布到多个节点的索引的建立，搜索请求分布到多个节点的执行。
2. 自动维护数据的副本，保证一旦某个机器宕机，不会丢失任何的数据。
3. 封装了更多高级功能，以给我们提供更多高级的支持，让我们快速的开发应用，开发更加复杂的应用，比如复杂的搜索功能，聚合分析功能，基于地理位置的搜索。

## es功能、场景、特点介绍

### 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：在秒级别对数据进行搜索和分析。如果检索数据要花费1小时,这就不叫近实时，这叫离线批处理（batch-processing）。

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）

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

### **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

## 核心概念

### 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。

### 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（相当于mysql表中的一条记录（行）），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 {

"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"

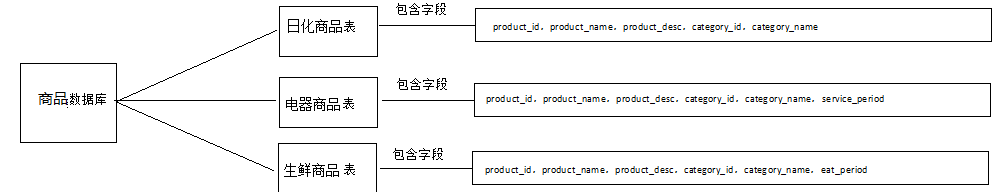
}

（5）Index：索引（相当于mysql中的数据库），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。

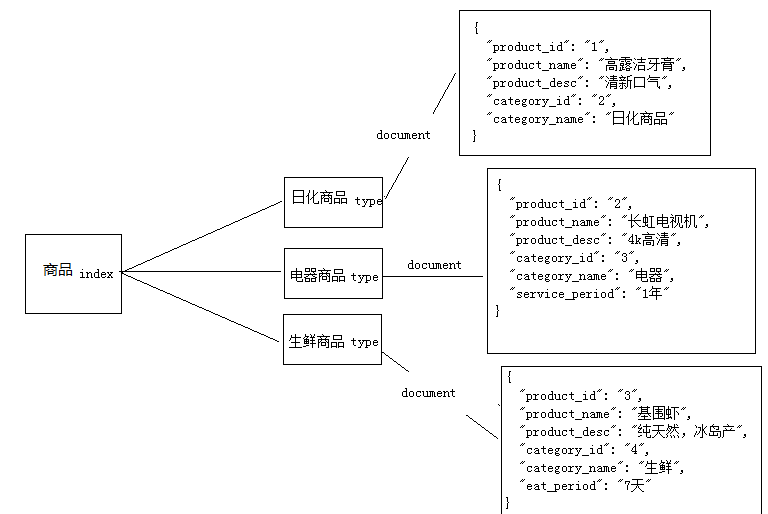
（6）Type：类型（相当于mysql中的表），每个索引里都可以有一个或多个type，type是index中的一个逻辑数据分类，一个type下的document，都有相同的field，比如博客系统，有一个索引，可以定义用户数据type，博客数据type，评论数据type。

举例：

在mysql中,数据存储如下：

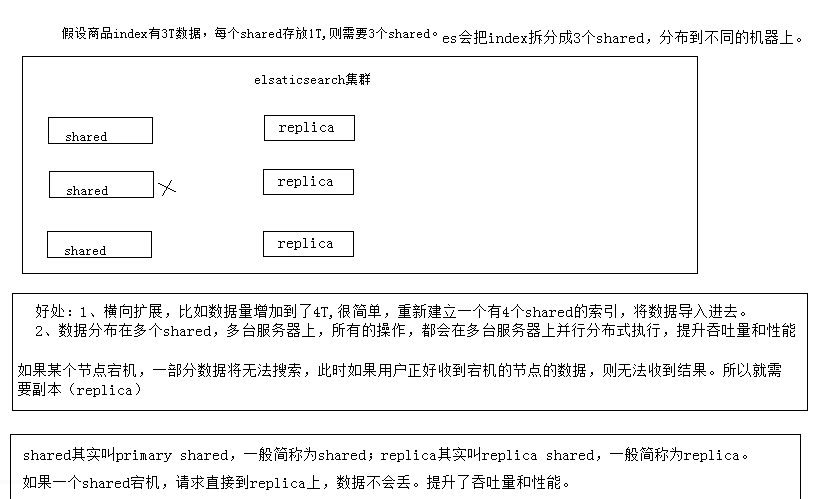


在es中，存储如下：



（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。

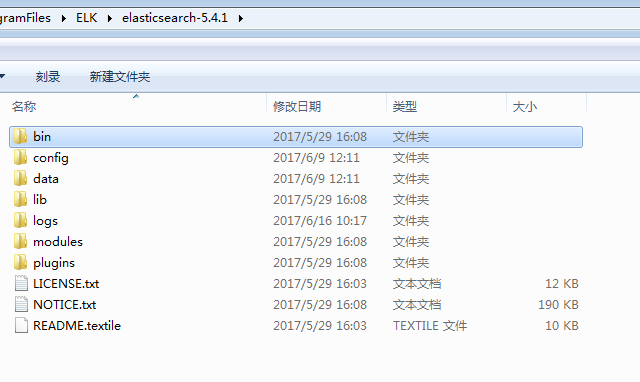
1. replica：任何一个服务器随时可能故障或宕机，此时shard可能就会丢失，因此可以为每个shard创建多个replica副本。replica可以在shard故障时提供备用服务，保证数据不丢失，多个replica还可以提升搜索操作的吞吐量和性能。primary shard（建立索引时一次设置，不能修改，默认5个），replica shard（随时修改数量，默认1个），默认每个索引10个shard，5个primary shard，5个replica shard，最小的高可用配置，是2台服务器。



## windows上安装

1、安装JDK，至少1.8.0\_73以上版本，java -version

2、下载和解压缩Elasticsearch安装包，目录结构



3、启动Elasticsearch：bin\elasticsearch.bat，es本身特点之一就是开箱即用，如果是中小型应用，数据量少，操作不是很复杂，直接启动就可以用了

4、检查ES是否启动成功：http://localhost:9200/?pretty

{

"name" : "9fzvF2H",

"cluster\_name" : "elasticsearch",

"cluster\_uuid" : "xBoSQEYqQSaJjzzpmAj7aw",

"version" : {

"number" : "5.4.1",

"build\_hash" : "2cfe0df",

"build\_date" : "2017-05-29T16:05:51.443Z",

"build\_snapshot" : false,

"lucene\_version" : "6.5.1"

},

"tagline" : "You Know, for Search"

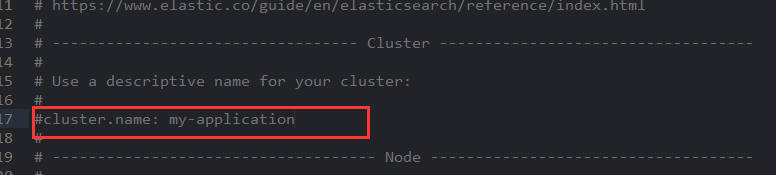
}

name: node名称

cluster\_name: 集群名称（默认的集群名称就是elasticsearch）

version.number: 5.4.1，es版本号

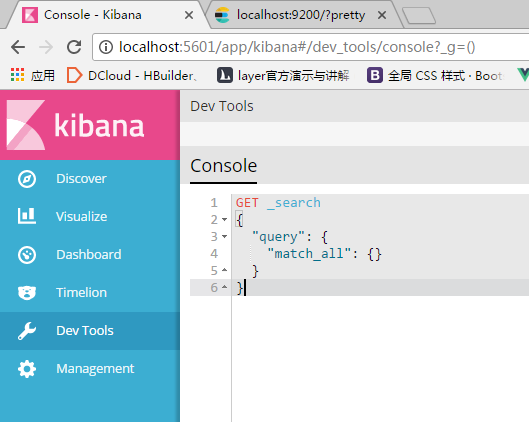
1. 修改集群名称：elasticsearch.yml



6、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

7、启动Kibana：bin\kibana.bat

8、进入Dev Tools界面



1. GET \_cluster/health

{

"cluster\_name": "elasticsearch",

"status": "yellow",

"timed\_out": false,

"number\_of\_nodes": 1,

"number\_of\_data\_nodes": 1,

"active\_primary\_shards": 11,

"active\_shards": 11,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 11,

"delayed\_unassigned\_shards": 0,

"number\_of\_pending\_tasks": 0,

"number\_of\_in\_flight\_fetch": 0,

"task\_max\_waiting\_in\_queue\_millis": 0,

"active\_shards\_percent\_as\_number": 50

}

## 集群健康、文档crud操作

1、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

{

"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"

}

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 --v显示列头

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。

（2）快速查看集群中有哪些索引

GET /\_cat/indices?v

（3）简单的索引操作

创建索引：PUT /test\_index?pretty

删除索引：DELETE /test\_index?pretty

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"

}

比如：

PUT /ecommerce/product/1

{

"name" : "gaolujie yagao",

"desc" : "gaoxiao meibai",

"price" : 30,

"producer" : "gaolujie producer",

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

}

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": "2",

"\_version": 1,

"found": true,

"\_source": {

"name": "jiajieshi yagao",

"desc": "youxiao fangzhu",

"price": 25,

"producer": "jiajieshi producer",

"tags": [

"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" ]

}

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"

}

}

（5）删除商品：删除文档

DELETE /ecommerce/product/1

## 搜索方式

1、query string search

2、query DSL

3、query filter

4、full-text search

5、phrase search

6、highlight search

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

### query string search

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

搜索全部商品：GET /ecommerce/product/\_search

{

"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"

]

}

}

]

}

}

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

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强大.更加适合生产环境的使用，可以构建复杂的查询.

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

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"]

}

### 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"

}

}

}

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": "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）

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"

]

}

}

]

}

}

### highlight search（高亮搜索结果）

返回的结果会包含<em></em>标签

GET /ecommerce/product/\_search

{

"query" : {

"match" : {

"producer" : "producer"

}

},

"highlight": {

"fields" : {

"producer" : {}

}

}

}

## 聚合

1、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,不然会报错。

PUT /ecommerce/\_mapping/product

{

"properties": {

"tags": {

"type": "text",

"fielddata": true

}

}

}

GET /ecommerce/product/\_search

{

"size": 0,

"aggs": {

"all\_tags": {

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

}

}

}

结果如下：

{

"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

}

]

}

}

}

2、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"

}

}

}

}

3、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" }

}

}

}

}

}

结果如下：

{

"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

}

}

]

}

}

}

4、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" }

}

}

}

}

}

5、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"

}

}

}

}

}

}

}

}

## 画图剖析es的分布式架构

### 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

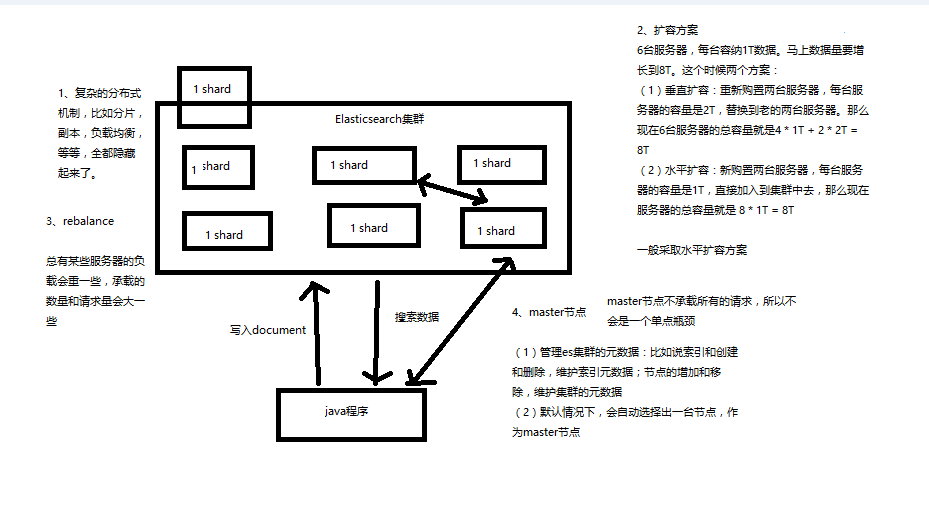
**ES隐藏了复杂的分布式机制**

1、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，我们有没有关心过数据怎么进行分片的，数据到哪个shard中去）

2、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）

3、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）

1. shard副本，请求路由，集群扩容，shard重分配



### 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

扩容对应用程序的透明性

### 增减或减少节点时的数据rebalance

保持负载均衡

### master节点

（1）创建或删除索引

（2）增加或删除节点

### 5、节点对等的分布式架构

（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

## 分片和副本再次梳理

### 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

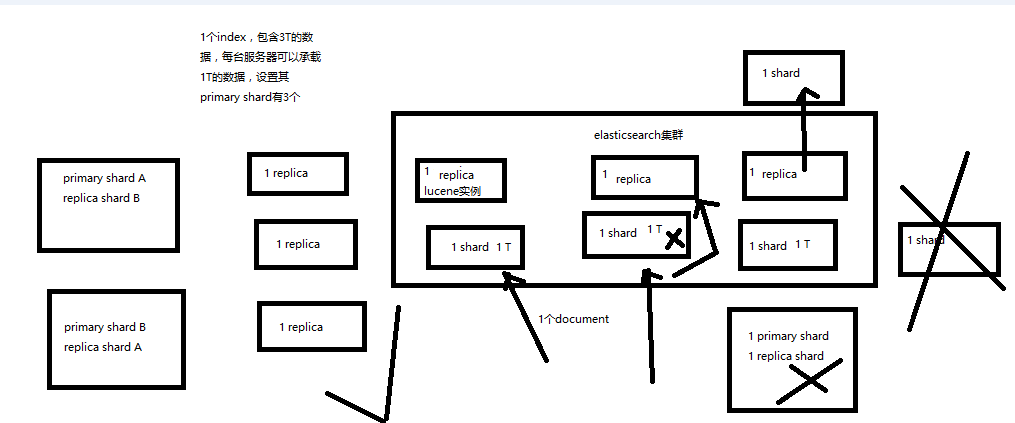
（5）replica shard是primary shard的副本，负责容错，以及承担读请求负载

（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

### 2、图解单node环境下创建index是什么样子的



1. 单node环境下，创建一个index，有3个primary shard，3个replica shard
2. 集群status是yellow

（3）这个时候，只会将3个primary shard分配到仅有的一个node上去，另外3个replica shard是无法分配的

（4）集群可以正常工作，但是一旦出现节点宕机，数据全部丢失，而且集群不可用，无法承接任何请求

（5）指定分片和副本个数

PUT /test\_index

{

"settings" : {

"number\_of\_shards" : 3,

"number\_of\_replicas" : 1

}

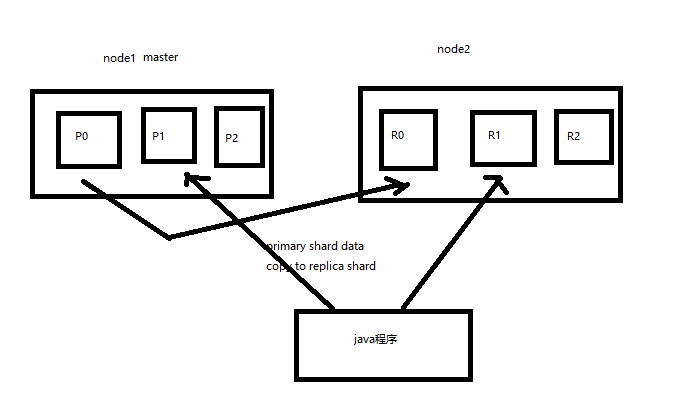
}

### 3、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

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

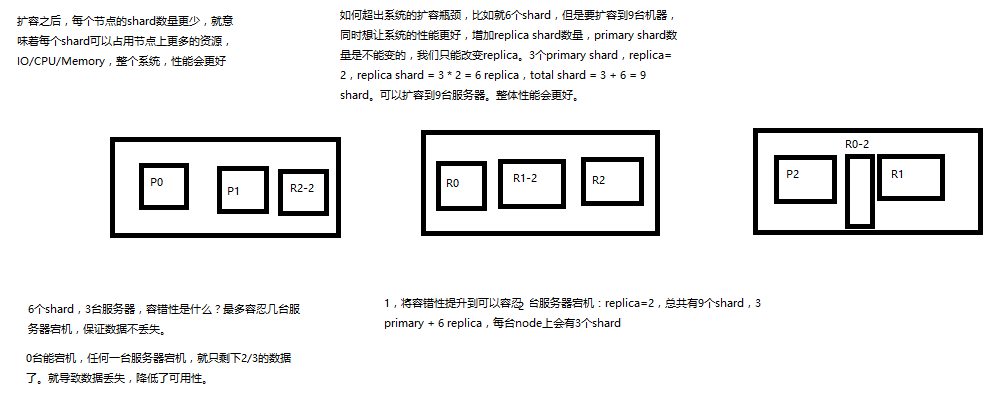
（2）每个node有更少的shard，IO/CPU/Memory资源给每个shard分配更多，每个shard性能更好

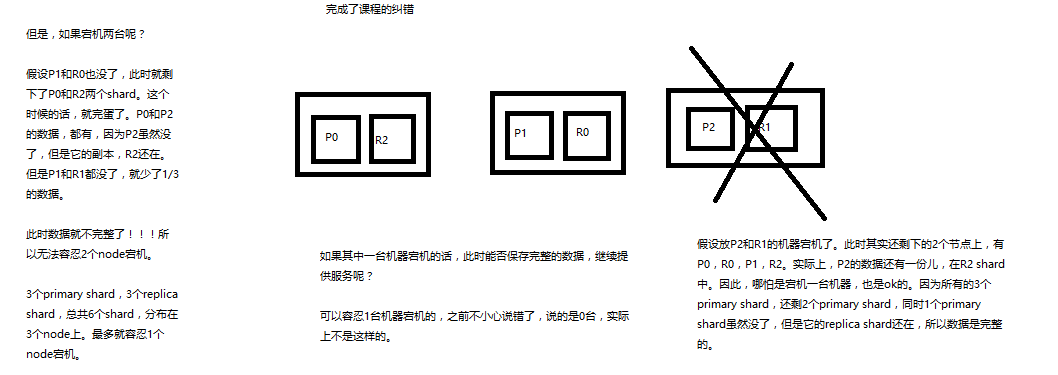
（3）扩容的极限，6个shard（3 primary，3 replica），最多扩容到6台机器，每个shard可以占用单台服务器的所有资源，性能最好

（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0台机器宕机

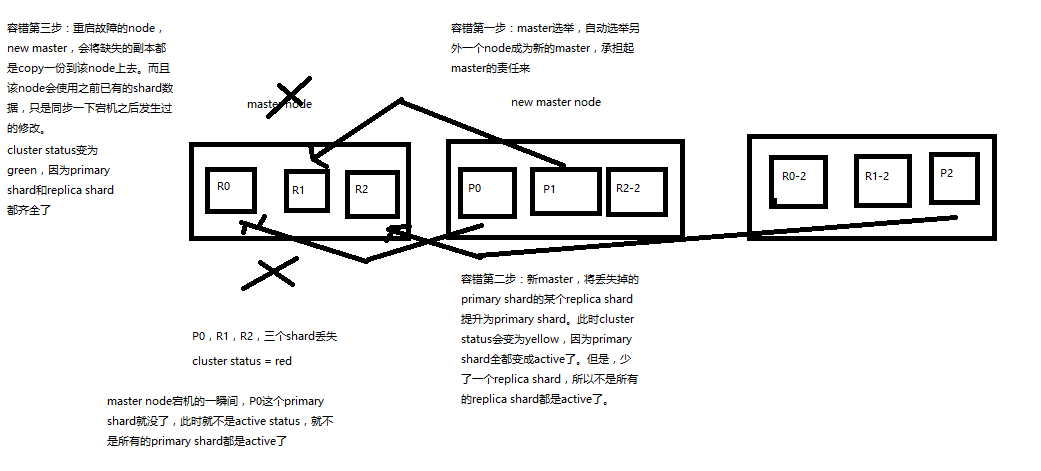
1. 综合起来看，就是说，一方面告诉你扩容的原理，怎么扩容，怎么提升系统整体吞吐量；另一方面要考虑到系统的容错性，怎么保证提高容错性，让尽可能多的服务器宕机，保证数据不丢失





## es容错机制

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

## 文档

### 初步解析document的核心元数据以及图解剖析index创建反例

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "1",

"\_version": 1,

"found": true,

"\_source": {

"test\_content": "test test"

}

}

**1.1、\_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

**1.2、\_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

**1.3、\_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

分布式文档系统-document id的手动指定与自动生成两种方式解析:

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

}

1. 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。

**1.4、\_source元数据**

\_source元数据：默认情况下，查询文档的时候会返回所有的字段，\_source可以定制返回的结果，指定\_source中，返回哪些field。

我们先插入一条数据：

put /test\_index/test\_type/1

{

"test\_field1": "test field1",

"test\_field2": "test field2"

}

再获取刚添加的数据：

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"

}

}

2、定制返回结果

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lazy delete机制

### 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

（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

### document的强制创建

（1）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

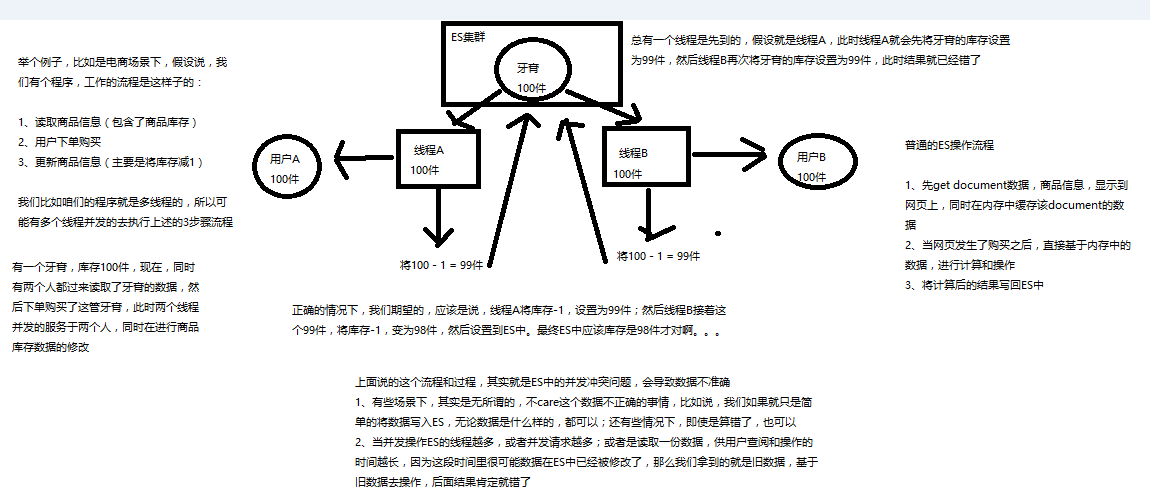
### 3、document的删除

DELETE /index/type/id

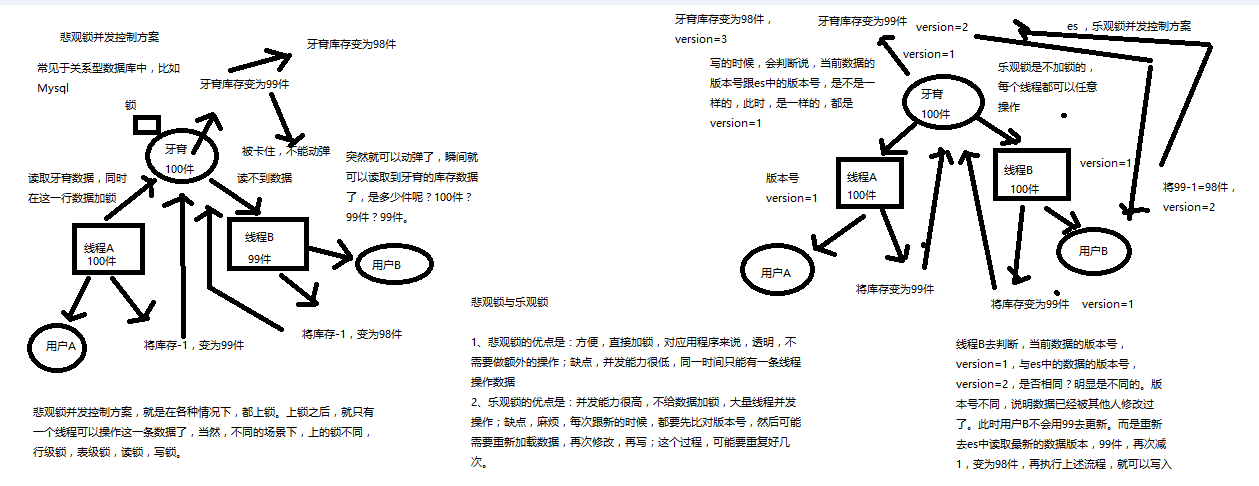
不会物理删除，只会将其标记为deleted，当数据越来越多的时候，在后台自动删除。

## 并发问题（\_version元数据）

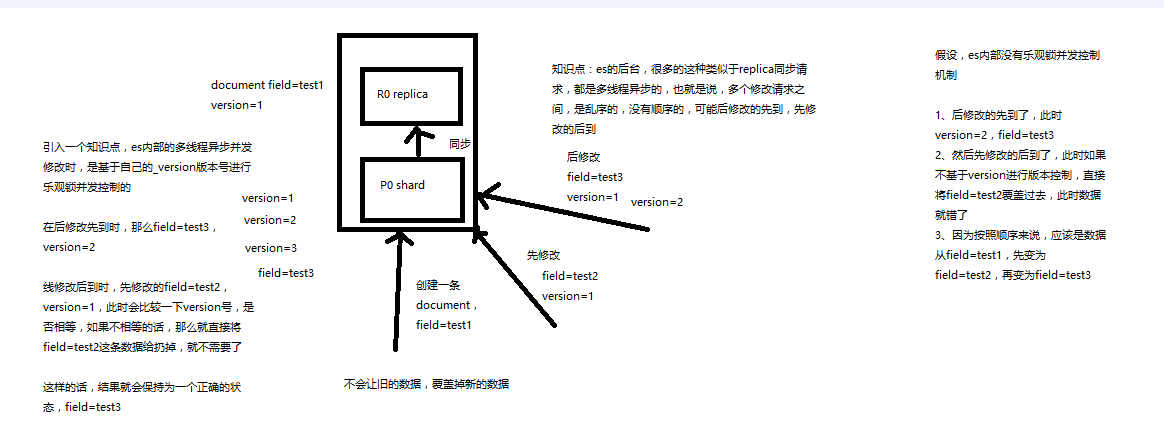
### 1、深度图解剖析Elasticsearch并发冲突问题



### 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



具体用代码来演示一下：

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

## 分布式文档系统-图解partial update实现原理以及动手实战演练

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

那么partial update，也就是每次只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。

post /index/type/id/\_update

{

"doc": {

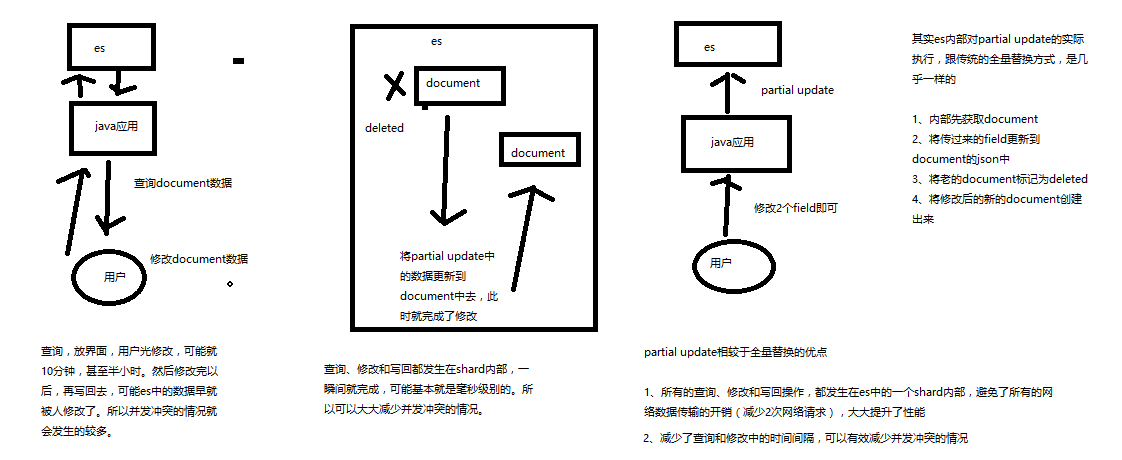
"要修改的少数几个field即可，不需要全量的数据"

}

}

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"

}

}

1. 基于groovy脚本进行partial update

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**

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"

}

返回结果：

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "11",

"\_version": 2,

"found": true,

"\_source": {

"num": 1,

"tags": []

}

}

（2）外部脚本，演示如何向数组中添加一项。

在es的安装目录的config下的script新建一个test-add-tags.groovy文件，写入如下脚本：

ctx.\_source.tags+=new\_tag

然后执行以下操作：

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

{

"script": {

"lang": "groovy",

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

"params": {

"new\_tag": "tag1"

}

}

}

1. 用脚本删除文档

新建一个test-delete-document.groovy文件，写入如下脚本：

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

然后执行以下操作：

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

{

"script": {

"lang": "groovy",

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

"params": {

"count": 1

}

}

}

（4）upsert操作

当文档被删除时，执行更新操作会返回错误

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

{

"doc": {

"num": 1

}

}

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": []

}

}

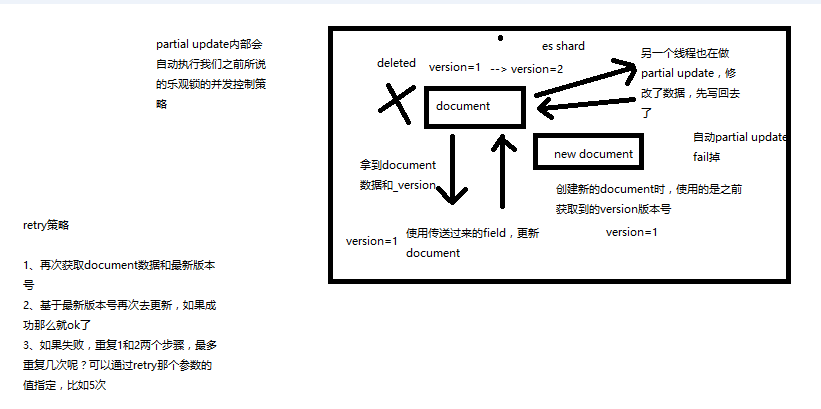
**图解partial update乐观锁并发控制原理以及相关操作讲解**

（1）partial update内置乐观锁并发控制

（2）retry\_on\_conflict

（3）\_version

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



## 批量操作

如果一条一条的查询，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

### mget的语法

一条一条的查询

GET /test\_index/test\_type/1

GET /test\_index/test\_type/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"

}

}

]

}

如果查询的document是一个index下的不同type的话

GET /test\_index/\_mget

{

"docs" : [

{

"\_type" : "test\_type",

"\_id" : 1

},

{

"\_type" : "test\_type",

"\_id" : 2

}

]

}

如果查询的数据都在同一个index下的同一个type下

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

{

"ids": [1, 2]

}

### 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，可能可以将性能提升数倍，甚至数十倍，非常非常之重要

### 批量增删改

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。

换行会报如下错误：

{

"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

}

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。

索引相同时，可以简化如下：

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"} }

索引、类型都相同时，可以简化如下：

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。

## 总结

Elasticsearch在跑起来以后，其实起到的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。