谷粒商城

版本：V 1.0

# 一、 项目整合redis

## 1 思路：

虽然咱们实现了页面需要的功能，但是考虑到该页面是被用户高频访问的，所以性能需要优化。

一般一个系统最大的性能瓶颈，就是数据库的io操作。从数据库入手也是调优性价比最高的切入点。

一般分为两个层面，一是提高数据库sql本身的性能，二是尽量避免直接查询数据库。

提高数据库本身的性能首先是优化sql，包括：使用索引，减少不必要的大表关联次数，控制查询字段的行数和列数。另外当数据量巨大是可以考虑分库分表，以减轻单点压力。

这部分知识在mysql高级已有讲解，这里大家可以以详情页中的sql作为练习，尝试进行优化，这里不做赘述。

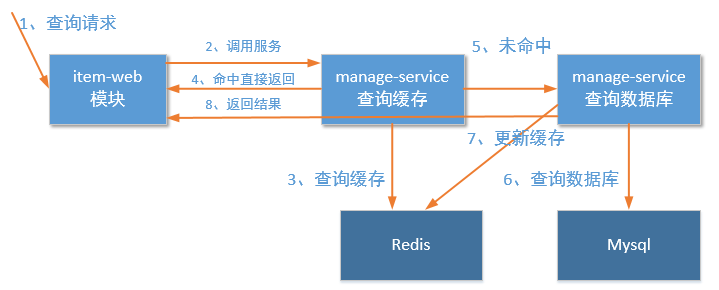
重点要讲的是另外一个层面：尽量避免直接查询数据库。

解决办法就是：缓存

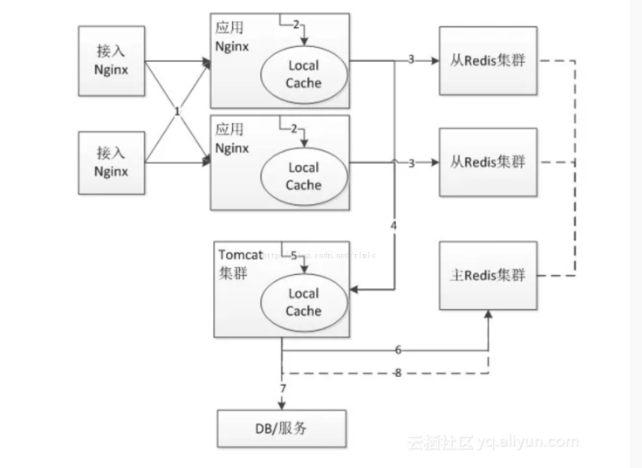
缓存可以理解是数据库的一道保护伞，任何请求只要能在缓存中命中，都不会直接访问数据库。而缓存的处理性能是数据库10-100倍。

咱们就用Redis作为缓存系统进行优化。

结构图：



多级缓存应用：



安装Redis ：

1. 安装依赖
2. 导入jar包
3. 创建安装目录
4. Make
5. Make install
6. 修改配置文件

## 2 整合redis到大工程中。

由于redis作为缓存数据库，要被多个项目使用，所以要制作一个通用的工具类，方便工程中的各个模块使用。

而主要使用redis的模块，都是后台服务的模块，xxx-service工程。所以咱们把redis的工具类放到service-util模块中，这样所有的后台服务模块都可以使用redis。

首先引入依赖包

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/redis.clients/jedis -->  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.9.0</version>  </dependency> |

分别按照之前的方式放到parent模块和service-util的pom文件中。

然后在service-util中创建两个类RedisConfig和RedisUtil

RedisConfig负责在spring容器启动时自动注入，而RedisUtil就是被注入的工具类以供其他模块调用。

Spring boot 项目 推荐使用注解方式来完成配置。

RedisUtil

|  |
| --- |
| **package** com.atguigu.gmall.config;  **public class** RedisUtil {   **private** JedisPool **jedisPool**;   **public void** initJedisPool(String host,**int** port,**int** database){  JedisPoolConfig jedisPoolConfig = **new** JedisPoolConfig();  *// 总数* jedisPoolConfig.setMaxTotal(200);  *// 获取连接时等待的最大毫秒* jedisPoolConfig.setMaxWaitMillis(10\*1000);  *// 最少剩余数* jedisPoolConfig.setMinIdle(10);  *// 如果到最大数，设置等待* jedisPoolConfig.setBlockWhenExhausted(**true**);  *// 在获取连接时，检查是否有效* jedisPoolConfig.setTestOnBorrow(**true**);  *// 创建连接池* **jedisPool** = **new** JedisPool(jedisPoolConfig,host,port,20\*1000);  }  **public** Jedis getJedis(){  Jedis jedis = **jedisPool**.getResource();  **return** jedis;  } } |

RedisConfig

@Configuration 相当于spring3.0版本的xml

|  |
| --- |
| @Configuration public class RedisConfig {   *//读取配置文件中的redis的ip地址* @Value("${spring.redis.host:disabled}")  private String host;   @Value("${spring.redis.port:0}")  private int port;   @Value("${spring.redis.database:0}")  private int database;   @Bean  public RedisUtil getRedisUtil(){  if(host.equals("disabled")){  return null;  }  RedisUtil redisUtil=new RedisUtil();  redisUtil.initPool(host,port,database);  return redisUtil;  }  } |

同时，任何模块想要调用redis都必须在application.properties配置，否则不会进行注入。

|  |
| --- |
| **spring.redis.host**=**192.168.67.204 spring.redis.port**=**6379 spring.redis.database**=**0** |

现在可以在manage-service中的getSkuInfo()方法测试一下

|  |
| --- |
| **try** {  Jedis jedis = **redisUtil**.getJedis();  jedis.set(**"test"**,**"text\_value"** ); }**catch** (JedisConnectionException e){  e.printStackTrace(); } |
| **在启动类上加上@ComonentScan注解** |

## 3 使用redis进行业务开发

开始开发先说明redis key的命名规范，由于Redis不像数据库表那样有结构，其所有的数据全靠key进行索引，所以redis数据的可读性，全依靠key。

企业中最常用的方式就是：object:id:field

比如：sku:1314:info

user:1092:info

:表示根据windows的 /一个意思

重构getSkuInfo方法

在gmall-manage-service项目中定义常量

|  |
| --- |
| **package** com.atguigu.gmall.manage.constant;  **public class** ManageConst {  **public static final** String ***SKUKEY\_PREFIX***=**"sku:"**;   **public static final** String ***SKUKEY\_SUFFIX***=**":info"**;   **public static final int *SKUKEY\_TIMEOUT***=24\*60\*60;  } |

|  |
| --- |
| @Override **public** SkuInfo getSkuInfo(String skuId) {  *// 缓存测试-* Jedis jedis = **redisUtil**.getJedis();  *// Ctrl+Alt+M 提取方法* SkuInfo skuInfo=**null**;  String skuInfoKey = ManageConst.***SKUKEY\_PREFIX***+skuId+ManageConst.***SKUKEY\_SUFFIX***;  **if** (jedis.exists(skuInfoKey)){  *// 取出数据* String skuInfoJson = jedis.get(skuInfoKey);  **if** (skuInfoJson!=**null** && skuInfoJson.length()!=0){  skuInfo = JSON.*parseObject*(skuInfoJson, SkuInfo.**class**);  }  *// 将数据转换成对象* }**else**{  *// 从数据库中取得数据* skuInfo = getSkuInfoDB(skuId);  *// 将最新的数据放入到缓存中* String jsonString = JSON.*toJSONString*(skuInfo);  jedis.setex(skuInfoKey,ManageConst.***SKUKEY\_TIMEOUT***,jsonString);  }  jedis.close();  **return** skuInfo; } **public** SkuInfo getSkuInfoDB(String skuId){  *// 单纯的信息* SkuInfo skuInfo = **skuInfoMapper**.selectByPrimaryKey(skuId);  *// 查询图片* SkuImage skuImage = **new** SkuImage();  skuImage.setSkuId(skuId);  List<SkuImage> imageList = **skuImageMapper**.select(skuImage);  *// 将查询出来所有图片赋予对象* skuInfo.setSkuImageList(imageList);  ***// 查询属性值* SkuSaleAttrValue skuSaleAttrValue = new SkuSaleAttrValue();  skuSaleAttrValue.setSkuId(skuId);  List<SkuSaleAttrValue> skuSaleAttrValueList = skuSaleAttrValueMapper.select(skuSaleAttrValue);    *// 将查询出来所有商品属性值赋给对象* skuInfo.setSkuSaleAttrValueList(skuSaleAttrValueList);**  **return** skuInfo; } |

以上基本实现使用缓存的方案。

如果说：redis服务没有启动，宕机！如何解决？

Try-catch 获取mysql数据返回

如果说：访问redis的时候，可能会有产生高并发，如何解决这种高并发访问？

使用分布式锁！

set test ok px 10000 nx

## 4 解决缓存击穿问题：

Redis:命令

# set sku:1:info “OK” NX PX 10000

EX second ：设置键的过期时间为 second 秒。 SET key value EX second 效果等同于 SETEX key second value 。

PX millisecond ：设置键的过期时间为 millisecond 毫秒。 SET key value PX millisecond 效果等同于 PSETEX key millisecond value 。

NX ：只在键不存在时，才对键进行设置操作。 SET key value NX 效果等同于 SETNX key value 。

XX ：只在键已经存在时，才对键进行设置操作。

|  |
| --- |
| **public static final int *SKULOCK\_EXPIRE\_PX***=10000; **public static final** String ***SKULOCK\_SUFFIX***=**":lock"**; |

|  |
| --- |
| @Override **public** SkuInfo getSkuInfo(String skuId) {  SkuInfo skuInfo = **null**; **try**{  Jedis jedis = **redisUtil**.getJedis();  *// 定义key* String skuInfoKey = ManageConst.***SKUKEY\_PREFIX***+skuId+ManageConst.***SKUKEY\_SUFFIX***; *//key= sku:skuId:info* String skuJson = jedis.get(skuInfoKey);   **if** (skuJson==**null** || skuJson.length()==0){  *// 没有数据 ,需要加锁！取出完数据，还要放入缓存中，下次直接从缓存中取得即可！* System.***out***.println(**"没有命中缓存"**);  *// 定义key user:userId:lock* String skuLockKey=ManageConst.***SKUKEY\_PREFIX***+skuId+ManageConst.***SKULOCK\_SUFFIX***;  *// 生成锁* String lockKey = jedis.set(skuLockKey, **"OK"**, **"NX"**, **"PX"**, ManageConst.***SKULOCK\_EXPIRE\_PX***);  **if** (**"OK"**.equals(lockKey)){  System.***out***.println(**"获取锁！"**);  *// 从数据库中取得数据* skuInfo = getSkuInfoDB(skuId);  *// 将是数据放入缓存  // 将对象转换成字符串* String skuRedisStr = JSON.*toJSONString*(skuInfo);  jedis.setex(skuInfoKey,ManageConst.***SKUKEY\_TIMEOUT***,skuRedisStr);  jedis.close();  **return** skuInfo;  }**else** {  System.***out***.println(**"等待！"**);  *// 等待* Thread.*sleep*(1000);  *// 自旋* **return** getSkuInfo(skuId);  }  }**else**{  *// 有数据* skuInfo = JSON.*parseObject*(skuJson, SkuInfo.**class**);  jedis.close();  **return** skuInfo;  } }**catch** (Exception e){  e.printStackTrace(); }  *// 从数据库返回数据* **return** getSkuInfoDB(skuId); } |

## 5 redisson 解决分布式锁

|  |
| --- |
| 1. 导入依赖 service-util   <dependency>  <groupId>org.redisson</groupId>  <artifactId>redisson</artifactId>  <version>3.11.1</version>  </dependency> |
| 1. 修改实现类   **private** SkuInfo getSkuInfoRedisson(String skuId) {  Config config = **new** Config();  config.useSingleServer().setAddress(**"redis://192.168.67.219:6379"**);   RedissonClient redissonClient = Redisson.*create*(config);   *// 使用redisson 调用getLock* RLock lock = redissonClient.getLock(**"yourLock"**);    *// 加锁* lock.lock(10, TimeUnit.***SECONDS***);  *// try { // boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS); // // } catch (InterruptedException e) { // e.printStackTrace(); // }  // 放入业务逻辑代码* SkuInfo skuInfo =**null**;  Jedis jedis = **null**;  *// ctrl+alt+t* **try** {  jedis = **redisUtil**.getJedis();  *// 定义key： 见名之意： sku：skuId:info* String skuKey = ManageConst.***SKUKEY\_PREFIX***+skuId+ManageConst.***SKUKEY\_SUFFIX***;  *// 判断缓存中是否有数据，如果有，从缓存中获取，没有从db获取并将数据放入缓存！  // 判断redis 中是否有key* **if** (jedis.exists(skuKey)){  *// 取得key 中的value* String skuJson = jedis.get(skuKey);  *// 将字符串转换为对象* skuInfo = JSON.*parseObject*(skuJson, SkuInfo.**class**); *// jedis.close();* **return** skuInfo;  }**else** {  skuInfo = getSkuInfoDB(skuId);  *// 放redis 并设置过期时间* jedis.setex(skuKey,ManageConst.***SKUKEY\_TIMEOUT***,JSON.*toJSONString*(skuInfo));  **return** skuInfo;  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **if** (jedis!=**null**){  jedis.close();  }  lock.unlock();  }  **return** getSkuInfoDB(skuId);  } |
|  |

# 搜索



**当前的位置**

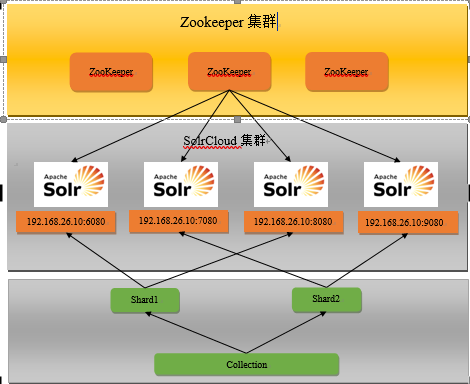
什么是搜索， 计算机根据用户输入的关键词进行匹配，从已有的数据库中摘录出相关的记录反馈给用户。

常见的全文搜索引擎，像百度、谷歌这样的。但是除此以外，搜索技术在垂直领域也有广泛的使用，比如淘宝、京东搜索商品，万芳、知网搜索期刊，csdn中搜索问题贴。也都是基于海量数据的搜索。

Elasticsearch：全文检索

Solr：本质web项目 xxx.war

SolrCloud简介 – solr集群。



## 1 如何处理搜索

### 1.1 用传统关系性数据库



弊端：

1、 对于传统的关系性数据库对于关键词的查询，只能逐字逐行的匹配，性能非常差。

2、匹配方式不合理，比如搜索“小密手机” ，如果用like进行匹配， 根本匹配不到。但是考虑使用者的用户体验的话，除了完全匹配的记录，还应该显示一部分近似匹配的记录，至少应该匹配到“手机”。

### **1.2 专业全文索引是怎么处理的**

全文搜索引擎目前主流的索引技术就是**倒排索引**的方式。

传统的保存数据的方式都是

记录→单词

而倒排索引的保存数据的方式是

单词→记录

例如

 搜索“红海行动”

但是数据库中保存的数据如图：

那么搜索引擎是如何能将两者匹配上的呢？

基于分词技术构建倒排索引：

首先每个记录保存数据时，都不会直接存入数据库。系统先会对数据进行分词，然后以倒排索引结构保存。如下：



然后等到用户搜索的时候，会把搜索的关键词也进行分词，会把“红海行动”分词分成：红海和行动两个词。

这样的话，先用红海进行匹配，得到id=1和id=2的记录编号，再用行动匹配可以迅速定位id为1,3的记录。

那么全文索引通常，还会根据匹配程度进行打分，显然1号记录能匹配的次数更多。所以显示的时候以评分进行排序的话，1号记录会排到最前面。而2、3号记录也可以匹配到。

### 1.3 首页搭建

在nginx 目录下新建文件夹front

然后将 首页资源统统放入进去，然后nginx.conf 配置添加配置

|  |
| --- |
| server {  listen 80;  server\_name www.gmall.com;  location / {  root front;  index index.htm;  }  }  =========================item.gmall.com=====================================  upstream item.gmall.com{  server 192.168.67.1:8084;  }  server {  listen 80;  server\_name item.gmall.com;  location / {  proxy\_pass http://item.gmall.com;  }  } |

# 三、 全文检索工具elasticsearch

如果es打不开：卸掉，把一些不相关tar.gz 删掉！

## 1 lucene与elasticsearch

咱们之前讲的处理分词，**构建倒排索引**，等等，都是这个叫lucene的做的。那么能不能说这个lucene就是搜索引擎呢？

还不能。lucene只是一个提供全文搜索功能类库的核心工具包，而真正使用它还需要一个完善的服务框架搭建起来的应用。

好比lucene是类似于jdk，而搜索引擎软件就是tomcat 的。

目前市面上流行的搜索引擎软件，主流的就两款，elasticsearch和solr,这两款都是基于lucene的搭建的，可以独立部署启动的搜索引擎服务软件。由于内核相同，所以两者除了服务器安装、部署、管理、集群以外，对于数据的操作，修改、添加、保存、查询等等都十分类似。就好像都是支持sql语言的两种数据库软件。只要学会其中一个另一个很容易上手。

从实际企业使用情况来看，elasticSearch的市场份额逐步在取代solr，国内百度、京东、新浪都是基于elasticSearch实现的搜索功能。国外就更多了 像维基百科、GitHub、Stack Overflow等等也都是基于ES的

## 2 elasticSearch的使用场景

1. 为用户提供按关键字查询的全文搜索功能。
2. 著名的ELK框架(ElasticSearch,Logstash,Kibana)，实现企业海量日志的处理分析的解决方案。大数据领域的重要一份子。

## 3 elasticSearch的安装

**详见《elasticSearch的安装手册》**

## 4 elasticsearch的基本概念

|  |  |
| --- | --- |
| cluster | 整个elasticsearch 默认就是集群状态，整个集群是一份完整、互备的数据。 |
| node | 集群中的一个节点，一般只一个进程就是一个node |
| Shard | 分片，即使是一个节点中的数据也会通过hash算法，分成多个片存放，默认是5片。 |
| **Index** | 相当于rdbms的database, 对于用户来说是一个逻辑数据库，虽然物理上会被分多个shard存放，也可能存放在多个node中。 |
| **Type** | 类似于rdbms的table，但是与其说像table，其实更像面向对象中的class , 同一Json的格式的数据集合。 |
| **Document** | 类似于rdbms的 row、面向对象里的object |
| **Field** | 相当于字段、属性 |

## 5 利用kibana学习 elasticsearch restful api (DSL)

### 5.1 es中保存的数据结构

|  |
| --- |
| public class Movie {  String id;  String name;  Double doubanScore;  List<Actor> actorList;  }  public class Actor{  String id;  String name;  } |

这两个对象如果放在关系型数据库保存，会被拆成2张表，但是elasticsearch是用一个json来表示一个document。

所以它保存到es中应该是：

|  |
| --- |
| {  “id”:”1”,  “name”:”operation red sea”,  “doubanScore”:”8.5”,  “actorList”:[  {“id”:”1”,”name”:”zhangyi”},  {“id”:”2”,”name”:”haiqing”},  {“id”:”3”,”name”:”zhanghanyu”}  ]  } |

### 5.2 对数据的操作

#### 5.2.1 查看es中有哪些索引

|  |
| --- |
| **GET /\_cat/indices?v** |

es 中会默认存在一个名为.kibana的索引

表头的含义

|  |  |
| --- | --- |
| health | green(集群完整) yellow(单点正常、集群不完整) red(单点不正常) |
| status | 是否能使用 |
| index | 索引名 |
| uuid | 索引统一编号 |
| pri | 主节点几个 |
| rep | 从节点几个 |
| docs.count | 文档数 |
| docs.deleted | 文档被删了多少 |
| store.size | 整体占空间大小 |
| pri.store.size | 主节点占 |

#### 5.2.2 增加一个索引

|  |
| --- |
| **PUT /movie\_index** |

#### 5.2.3 删除一个索引

ES **是不删除也不修改任何数据 ,伪删除更新当前index的版本**。

|  |
| --- |
| DELETE /movie\_index |

#### 5.2.4 新增文档

1. 格式 PUT /index/type/id

|  |
| --- |
| PUT /movie\_index/movie/1  { "id":1,  "name":"operation red sea",  "doubanScore":8.5,  "actorList":[  {"id":1,"name":"zhang yi"},  {"id":2,"name":"hai qing"},  {"id":3,"name":"zhang han yu"}  ]  }  PUT /movie\_index/movie/2  {  "id":2,  "name":"operation meigong river",  "doubanScore":8.0,  "actorList":[  {"id":3,"name":"zhang han yu"}  ]  }  PUT /movie\_index/movie/3  {  "id":3,  "name":"incident red sea",  "doubanScore":5.0,  "actorList":[  {"id":4,"name":"liu de hua"}  ]  } |

如果之前没建过index或者type，es 会自动创建。

#### 5.2.5 直接用id查找

|  |
| --- |
| GET movie\_index/movie/1 |

#### 5.2.6 修改—整体替换

和新增没有区别

|  |
| --- |
| PUT /movie\_index/movie/3  {  "id":"3",  "name":"incident red sea",  "doubanScore":"5.0",  "actorList":[  {"id":"1","name":"zhang guo li"}  ]  } |

#### 5.2.7 修改—某个字段 更新es商品中的排名

|  |
| --- |
| **POST movie\_index/movie/3/\_update**  **{**  **"doc": {**  **"doubanScore":"7.0"**  **}**  **}** |

#### 5.2.8 删除一个document

|  |
| --- |
| DELETE movie\_index/movie/3 |

#### 5.2.9 搜索type全部数据

|  |
| --- |
| GET movie\_index/movie/\_search |

结果

|  |
| --- |
| {  "took": 2, //耗费时间 毫秒  "timed\_out": false, //是否超时  "\_shards": {  "total": 5, //发送给全部5个分片  "successful": 5,  "skipped": 0,  "failed": 0  },  "hits": {  "total": 3, //命中3条数据  "max\_score": 1, //最大评分  "hits": [ // 结果  {  "\_index": "movie\_index",  "\_type": "movie",  "\_id": 2,  "\_score": 1,  "\_source": {  "id": "2",  "name": "operation meigong river",  "doubanScore": 8.0,  "actorList": [  {  "id": "1",  "name": "zhang han yu"  }  ]  }  。。。。。。。。  。。。。。。。。  } |

#### 5.2.10 按条件查询(全部)

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match\_all": {}  }  } |

#### 5.2.11 按分词查询

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match": {"name":"red"}  }  } |

注意结果的评分

#### 5.2.12 按分词子属性查询

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match": {"actorList.name":"zhang"}  }  } |

#### 5.2.13 match phrase 按词组查询

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match\_phrase": {"name":"operation red"}  }  } |

按短语查询，不再利用分词技术，直接用短语在原始数据中匹配

#### 5.2.14 fuzzy查询

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "fuzzy": {"name":"rad"}  }  } |

校正匹配分词，当一个单词都无法准确匹配，es通过一种算法对非常接近的单词也给与一定的评分，能够查询出来，但是消耗更多的性能。

#### 5.2.15 过滤--查询后过滤

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match": {"name":"red"}  },  "post\_filter":{  "term": {  "actorList.id": 3  }  }  } |

#### 5.2.16 过滤--查询前过滤（推荐）

|  |
| --- |
| 其实准确来说，ES中的查询操作分为2种：查询（query）和过滤（filter）。**查询**即是之前提到的query查询，它**（查询）默认会计算每个返回文档的得分，然后根据得分排序**。而**过滤（filter）只会筛选出符合的文档，并不计算得分，且它可以缓存文档。**所以，单从性能考虑，过滤比查询更快。  换句话说，过滤适合在大范围筛选数据，而查询则适合精确匹配数据。一般应用时，**应先使用过滤操作过滤数据，然后使用查询匹配数据。** |

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "bool":{  "filter":[ {"term": { "actorList.id": "1" }},  {"term": { "actorList.id": "3" }}  ],  "must":{"match":{"name":"red"}}  }  }  } |
| term、terms过滤  term、terms的含义与查询时一致。term用于精确匹配、terms用于多词条匹配。不过既然过滤器适用于大氛围过滤，term、terms在过滤中使用意义不大。在项目中建议使用term。  Term: where id = ?  Terms: where id in ()  # select \* from skuInfo where id=? Select \* from skuInfo where id in () |

#### 5.2.17 过滤--按范围过滤

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query": {  "bool": {  "filter": {  "range": {  "doubanScore": {"gte": 8}  }  }  }  }  } |

关于范围操作符：跟html标签中的转义字符一样！

|  |  |
| --- | --- |
| gt | 大于 |
| lt | 小于 |
| gte | 大于等于 |
| lte | 小于等于 |

#### 5.2.18 排序

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match": {"name":"red sea"}  }  , "sort": [  {  "doubanScore": {  "order": "desc"  }  }  ]  } |

面试题：

Mysql 默认升序

Oracle 默认是升序

Sqlserver 默认是升序

端口号不一样，分页语句不一样！

3306 1521 1433

Limit rownum top

#### 5.2.19 分页查询

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query": { "match\_all": {} },  // 第几条开始查询！  "from": 1,  "size": 1  } |

#### 5.2.20 指定查询的字段

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query": { "match\_all": {} },  "\_source": ["name", "doubanScore"]  } |

#### 5.2.21 高亮

|  |
| --- |
| GET movie\_index/movie/\_search  {  "query":{  "match": {"name":"red sea"}  },  "highlight": {  "fields": {"name":{} }  }    } |
| 修改自定义高亮标签  **GET movie\_index/movie/\_search**  **{**  **"query":{**  **"match": {"name":"red sea"}**  **},**  **"highlight": {**  **"post\_tags": ["</span>"],**  **"pre\_tags": ["<span>"],**  **"fields": {"name":{} }**  **}**  **}** |

#### 5.2.22 聚合

取出每个演员共参演了多少部电影 – sql ： group by ！

|  |
| --- |
| GET movie\_index/movie/\_search  {  "aggs": {  "groupby\_actor": {  "terms": {  "field": "actorList.name.keyword"  }  }  }  } |

每个演员参演电影的平均分是多少，并按评分排序

|  |
| --- |
| GET movie\_index/movie/\_search  {  "aggs": {  "groupby\_actor\_id": {  "terms": {  "field": "actorList.name.keyword" ,  "order": {  "avg\_score": "desc"  }  },  "aggs": {  "avg\_score":{  "avg": {  "field": "doubanScore"  }  }  }  }  }  } |

### 5.3 关于mapping

之前说type可以理解为table，那每个字段的数据类型是如何定义的呢

查看看mapping

|  |
| --- |
| **GET movie\_index/\_mapping/movie** |

实际上每个type中的字段是什么数据类型，由mapping定义。

但是如果没有设定mapping系统会自动，根据一条数据的格式来推断出应该的数据格式。

* true/false → boolean
* 1020 → long
* 20.1 → double，float
* “2018-02-01” → date
* “hello world” → text +keyword

**默认只有text会进行分词，keyword是不会分词的字符串。**

mapping除了自动定义，还可以手动定义，但是只能对新加的、没有数据的字段进行定义。一旦有了数据就无法再做修改了。

注意：虽然每个Field的数据放在不同的type下,但是同一个名字的Field在一个index下只能有一种mapping定义。

### 5.4 中文分词

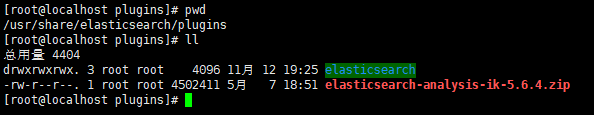
elasticsearch本身自带的中文分词，就是单纯把中文一个字一个字的分开，根本没有词汇的概念。但是实际应用中，用户都是以词汇为条件，进行查询匹配的，如果能够把文章以词汇为单位切分开，那么与用户的查询条件能够更贴切的匹配上，查询速度也更加快速。

分词器下载网址：<https://github.com/medcl/elasticsearch-analysis-ik>

#### 5.4.1 安装

下载好的zip包，请解压后放到 /usr/share/elasticsearch/plugins/

[root@localhost plugins]# unzip elasticsearch-analysis-ik-5.6.4.zip



将压缩包文件删除！否则启动失败！

然后重启es

[root@localhost plugins]# service elasticsearch restart

#### 5.4.2 测试使用

使用默认

|  |
| --- |
| GET movie\_index/\_analyze  {  "text": "我是中国人"  } |

请观察结果

使用分词器

|  |
| --- |
| GET movie\_index/\_analyze  { "analyzer": "ik\_smart",  "text": "我是中国人"  } |

请观察结果

另外一个分词器

ik\_max\_word

|  |
| --- |
| GET movie\_index/\_analyze  { "analyzer": "ik\_max\_word",  "text": "我是中国人"  } |

请观察结果

能够看出不同的分词器，分词有明显的区别，所以以后定义一个type不能再使用默认的mapping了，要手工建立mapping, 因为要选择分词器。

#### 5.4.3 基于中文分词搭建索引

1、建立mapping

|  |
| --- |
| PUT movie\_chn  {  "mappings": {  "movie":{  "properties": {  "id":{  "type": "long"  },  "name":{  "type": "text"  , "analyzer": "ik\_smart"  },  "doubanScore":{  "type": "double"  },  "actorList":{  "properties": {  "id":{  "type":"long"  },  "name":{  "type":"keyword"  }  }  }  }  }  }  } |

插入数据

|  |
| --- |
| PUT /movie\_chn/movie/1  { "id":1,  "name":"红海行动",  "doubanScore":8.5,  "actorList":[  {"id":1,"name":"张译"},  {"id":2,"name":"海清"},  {"id":3,"name":"张涵予"}  ]  }  PUT /movie\_chn/movie/2  {  "id":2,  "name":"湄公河行动",  "doubanScore":8.0,  "actorList":[  {"id":3,"name":"张涵予"}  ]  }  PUT /movie\_chn/movie/3  {  "id":3,  "name":"红海事件",  "doubanScore":5.0,  "actorList":[  {"id":4,"name":"张国立"}  ]  } |

查询测试

|  |
| --- |
| GET /movie\_chn/movie/\_search  {  "query": {  "match": {  "name": "红海战役"  }  }  }  GET /movie\_chn/movie/\_search  {  "query": {  "term": {  "actorList.name": "张译"  }  }  } |

#### 5.4.4 自定义词库

什么使用？

当词库满足不了你的需要，可以使用自定义词库！

修改/usr/share/elasticsearch/plugins/ik/config/中的IKAnalyzer.cfg.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">  <properties>  <comment>IK Analyzer 扩展配置</comment>  <!--用户可以在这里配置自己的扩展字典 -->  <entry key="ext\_dict"></entry>  <!--用户可以在这里配置自己的扩展停止词字典-->  <entry key="ext\_stopwords"></entry>  <!--用户可以在这里配置远程扩展字典 -->  <entry key="remote\_ext\_dict">http://192.168.67.163/fenci/myword.txt</entry>  <!--用户可以在这里配置远程扩展停止词字典-->  <!-- <entry key="remote\_ext\_stopwords">words\_location</entry> -->  </properties> |

按照标红的路径利用nginx发布静态资源

在nginx.conf中配置

|  |
| --- |
| server {  listen 80;  server\_name 192.168.67.163;  location /fenci/ {  root es;  }  } |

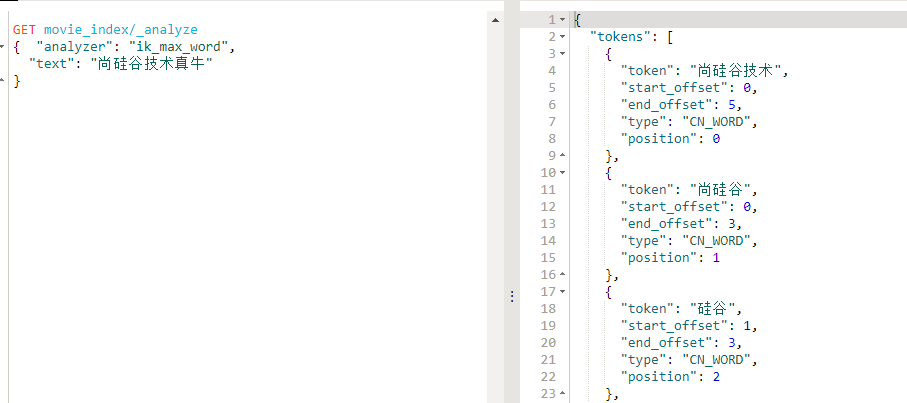
并且在/usr/local/nginx/下建/es/fenci/目录，目录下加myword.txt

myword.txt中编写关键词，每一行代表一个词。



然后重启es服务器，重启nginx。

在kibana中测试分词效果

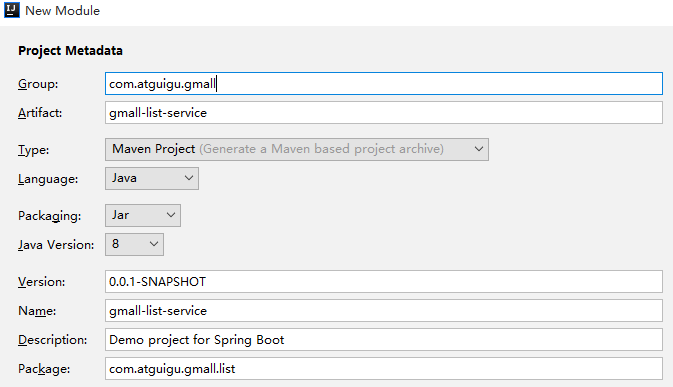


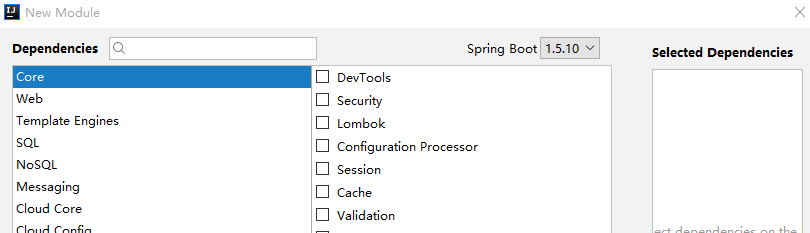
更新完成后，es只会对新增的数据用新词分词。历史数据是不会重新分词的。如果想要历史数据重新分词。需要执行：

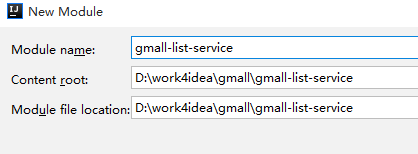
|  |
| --- |
| POST movies\_index\_chn/\_update\_by\_query?conflicts=proceed |

# 四、 Java程序中的应用

### 1 、搭建模块







pom.xml

|  |
| --- |
| *<?*xml version="1.0" encoding="UTF-8"*?>* <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>   <groupId>com.atguigu.gmall</groupId>  <artifactId>gmall-list-service</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>   <name>gmall-list-service</name>  <description>Demo project for Spring Boot</description>   <parent>  <groupId>com.atguigu.gmall</groupId>  <artifactId>gmall-parent</artifactId>  <version>1.0-SNAPSHOT</version>  </parent>  <dependencies>   <dependency>  <groupId>com.atguigu.gmall</groupId>  <artifactId>gmall-interface</artifactId>  <version>1.0-SNAPSHOT</version>  </dependency>   <dependency>  <groupId>com.atguigu.gmall</groupId>  <artifactId>gmall-service-util</artifactId>  <version>1.0-SNAPSHOT</version>  </dependency>   </dependencies>   <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>   </project> |

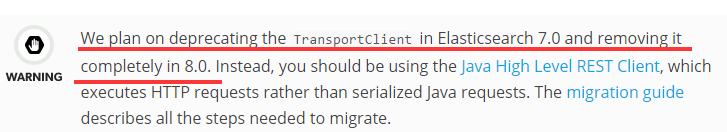
### 2、 关于es 的java 客户端的选择

目前市面上有两类客户端

一类是TransportClient 为代表的ES原生客户端，不能执行原生dsl语句必须使用它的Java api方法。

另外一种是以Rest Api为主的missing client，最典型的就是jest。 这种客户端可以直接使用dsl语句拼成的字符串，直接传给服务端，然后返回json字符串再解析。

两种方式各有优劣，但是最近elasticsearch官网，宣布计划在7.0以后的版本中废除TransportClient。以RestClient为主。



所以在官方的RestClient 基础上，进行了简单包装的Jest客户端，就成了首选，而且该客户端也与springboot完美集成。

### 3 、在gmall-list-service项目中导入Jest依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-elasticsearch</**artifactId**> </**dependency**>  *<!-- https://mvnrepository.com/artifact/io.searchbox/jest -->* <**dependency**>  <**groupId**>io.searchbox</**groupId**>  <**artifactId**>jest</**artifactId**> </**dependency**>  *<!-- https://mvnrepository.com/artifact/net.java.dev.jna/jna -->* <**dependency**>  <**groupId**>net.java.dev.jna</**groupId**>  <**artifactId**>jna</**artifactId**>  </**dependency**> |

其中jest和jna请将版本号，部分纳入gmall-parent中管理。spring-boot-starter-data-elasticsearch不用管理版本号，其版本跟随springboot的1.5.10大版本号。

### 4 、在测试类中测试ES

application.properties中加入

|  |
| --- |
| **server.port**=**8085 logging.level.root**=**error spring.dubbo.application.name**=**list-service spring.dubbo.registry.protocol**=**zookeeper spring.dubbo.registry.address**=**192.168.67.203:2181 spring.dubbo.base-package**=**com.atguigu.gmall0319 spring.dubbo.protocol.name**=**dubbo spring.datasource.url**=**jdbc:mysql://localhost:3306/gmall?characterEncoding=UTF-8 spring.datasource.username**=**root spring.datasource.password**=**root spring.datasource.driver-class-name**=**com.mysql.jdbc.Driver** *#mybatis* **mybatis.mapper-locations**=**classpath:mapper/\*Mapper.xml mybatis.configuration.mapUnderscoreToCamelCase**=**true** *# redis* **spring.redis.host**=**192.168.67.203 spring.redis.port**=**6379 spring.redis.database**=**0** |

|  |
| --- |
| **spring.elasticsearch.jest.uris**=**http://192.168.67.163:9200** |

在springBoot 测试类 中添加

|  |
| --- |
| @Autowired JestClient jestClient;  @Test public void testEs() throws IOException {  String query="{\n" +  " \"query\": {\n" +  " \"match\": {\n" +  " \"actorList.name\": \"张译\"\n" +  " }\n" +  " }\n" +  "}";  Search search = new Search.Builder(query).addIndex("movie\_chn").addType("movie").build();   SearchResult result = jestClient.execute(search);   List<SearchResult.Hit<HashMap, Void>> hits = result.getHits(HashMap.class);   for (SearchResult.Hit<HashMap, Void> hit : hits) {  HashMap source = hit.source;  System.*err*.println("source = " + source);  }  } |

打印结果：



以上技术方面的准备就做好了。下面回到咱们电商的业务