# **Redis**

## **什么是redis**

Redis是用C语言开发的一个开源的高性能键值对（key-value）数据库。redis是将数据存放到内存中,由于内容存取速度快所以redis被广泛应用在互联网项目中。

## redis下载

<http://download.redis.io/releases/redis-3.0.0.tar.gz>

## 安装redis

### 准备

Redis是c语言开发的。安装redis需要c语言的环境。

yum install gcc-c++

### 安装redis

将redis-3.0.0.tar.gz拷贝到/usr/upload下解压源码

tar -zxvf redis-3.0.0.tar.gz

进入解压后的目录进行编译

cd /usr/local/redis-3.0.0  
​make

安装到指定目录

make install PREFIX=/usr/java/redis

## 启动和关闭redis

### 前端启动

**1、启动方式：**

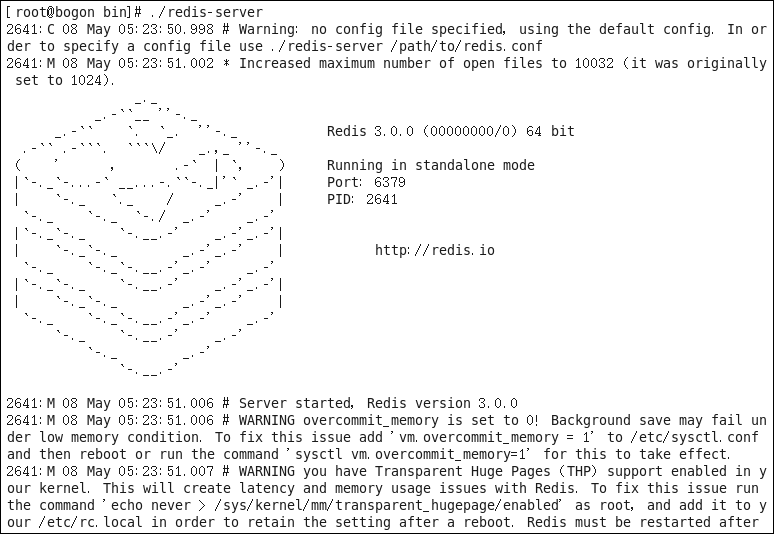
直接运行bin/redis-server将以前端模式启动。

cd /usr/java/redis/bin  
./redis-server

**2、启动缺点：**

ssh命令窗口关闭则redis-server程序结束，不推荐使用此方法

**3、启动图例：**



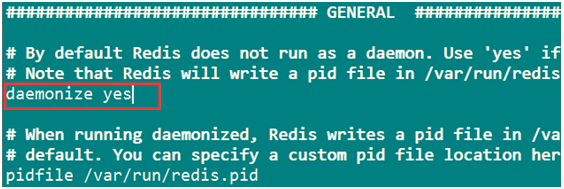
**4、前端启动的关闭：**ctrl+c

### 后端启动

第一步：进入源码目录，里面有一份配置文件 redis.conf，然后将其拷贝到安装路径下

cp /usr/upload/redis-3.0.0/redis.conf /usr/java/redis/bin

第二步：修改/usr/java/redis/bin下的redis.conf，将daemonize由no改为**yes**



第三步：启动

./redis-server redis.conf

### 关闭

到redis/bin目录下执行如下命令

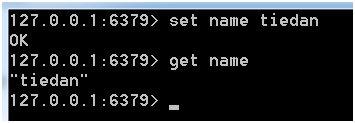
./redis-cli -p 端口号 shutdown

## redis的基本命令

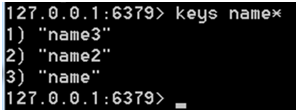
redis-cli连上redis服务后，可以在命令行发送命令：

./redis-cli -h 127.0.0.1 -p 6379

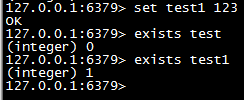
1、**set**和**get：**使用set和get可以向redis设置数据、获取数据



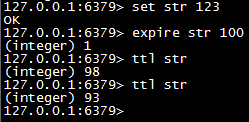
2、**keys \*：**查看当前库中所有的key值



3、**exists**：判断key值是否存在



4、**expire** 和 **ttl**：expire设置key的过期时间，ttl查看key的有效期

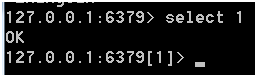


5、**clear：**可以使用clear命令对redis-cli终端屏幕清屏。

## 多数据库测试

​ 一个redis实例可以包括多个数据库，客户端可以指定连接某个redis实例的哪个数据库，就好比一个mysql中创建多个数据库，客户端连接时指定连接哪个数据库。

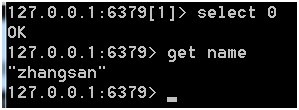
​ 一个redis实例最多可提供16个数据库，下标从**0**到**15**，客户端默认连接第0号数据库，也可以通过select选择连接哪个数据库，如下连接1号库：



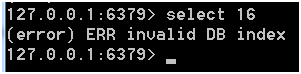
在1号库中查询上节设置的数据，结果查询不到：

IMG_264

重新选择第0号数据库，查询数据：



如果选择一个不存在数据库则会报错：



**注意：**

redis不支持修改数据库的名称，只能通过select 0、select 1...选择数据库。

FLUSHALL --清空所有数据库的所有数据

FLUSHDB --清空当前所在数据库的数据

## redis五种数据类型

### string

**String：key-value**

赋值：set key value

127.0.0.1:6379> set test 123​  
OK

取值：get key

127.0.0.1:6379> get test​  
"123“

删除：del key

127.0.0.1:6379>del test  
​(integer) 1

自增自减：incr key decr key

127.0.0.1:6379> incr num

(integer) 1

127.0.0.1:6379> incr num

(integer) 2

127.0.0.1:6379> incr num

(integer) 3

### hash

**Hash：key-field-value**

​ **key：map**

赋值：HSET key field value

127.0.0.1:6379> hset user name zhangsan

(integer) 1

取值：HGET key field

127.0.0.1:6379> hget user name

"zhangsan“

删除：HDEL key field

127.0.0.1:6379> hdel user name

(integer) 1

### list

List：有顺序可重复，内部是使用双向链表实现的

192.168.25.153:6379> lpush list a b c d

(integer) 4

192.168.25.153:6379> lrange list 0 -1

1) "d"

2) "c"

3) "b"

4) "a"

192.168.25.153:6379> rpush list 1 2 3 4

(integer) 8

192.168.25.153:6379> lrange list 0 -1

1) "d"

2) "c"

3) "b"

4) "a"

5) "1"

6) "2"

7) "3"

8) "4"

192.168.25.153:6379> lrem list 0 d

(integer) 1

192.168.25.153:6379> lrange list1 0 -1

1) "c"

2) "b"

3) "a"

4) "1"

5) "2"

6) "3"

7) "4"

### set

Set：元素无顺序，不能重复

192.168.25.153:6379> sadd set1 a b c c c d

(integer) 4

192.168.25.153:6379> smembers set1

1) "b"

2) "c"

3) "d"

4) "a"

192.168.25.153:6379> srem set1 a

(integer) 1

192.168.25.153:6379> smembers set

1) "b"

2) "c"

3) "d"

### zset

SortedSet（zset）：有顺序，不能重复

192.168.25.153:6379> zadd zset1 2 a 5 b 1 c 6 d

(integer) 4

192.168.25.153:6379> zrange zset1 0 -1

1) "c"

2) "a"

3) "b"

4) "d"

192.168.25.153:6379> zrem zset1 a

(integer) 1

192.168.25.153:6379> zrange zset1 0 -1

1) "c"

2) "b"

3) "d"

192.168.25.153:6379> zrevrange zset1 0 -1

1) "d"

2) "b"

3) "c"

192.168.25.153:6379> zrange zset1 0 -1 withscores

1) "c"

2) "1"

3) "b"

4) "5"

5) "d"

6) "6"

192.168.25.153:6379> zrevrange zset1 0 -1 withscores

1) "d"

2) "6"

3) "b”

4) "5"

5) "c"

6) "1"

### 总结



## 持久化

​ Redis的高性能是由于其将所有数据都存储在了内存中，为了使Redis在重启之后仍能保证数据不丢失，需要将数据从内存中同步到硬盘中，这一过程就是持久化。

​ Redis支持两种方式的持久化，一种是RDB方式，一种是AOF方式。可以单独使用其中一种或将二者结合用。

### RDB持久化

​ RDB方式的持久化是通过快照（snapshotting）完成的，当符合一定条件时Redis会自动将内存中的数据进行快照并持久化到硬盘。

​ RDB是Redis默认采用的持久化方式，在redis.conf配置文件中默认有此下配置：

save 900 1  #900秒内容如果超过1个key被修改，则发起快照保存  
save 300 10 #300秒内容如超过10个key被修改，则发起快照保存  
save 60 10000 #表示60秒内如果超过10000个key被修改，则发起快照保存

在redis.conf中：

​ 配置dir指定rdb快照文件的位置

​ 配置dbfilenam指定rdb快照文件的名称

Redis启动后会读取RDB快照文件，将数据从硬盘载入到内存。

### 实验

1、删除dump.rdb

2、set test abc

3、关闭服务端并观察bin目录的变化

总结：

​ 优点:让redis的数据存取速度变快

​ 缺点:服务器断电时会丢失部分数据(数据的完整性得不到保证)

## AOF持久化

默认情况下Redis没有开启AOF（append onlyfile）方式的持久化，可以通过appendonly参数开启：

appendonly yes

​ AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的，默认的文件名是appendonly.aof，可以通过appendfilename参数修改：

appendfilename appendonly.aof

​ AOF持久化策略如下：

#appendfsync always   #每次有数据修改发生时都会写入AOF文件。  
appendfsync everysec  #每秒钟同步一次，该策略为AOF的缺省策略。  
#appendfsyncno       #从不同步。高效但是数据不会被持久化。

### 实验

1、删除appendonly.aof

2、设置appendonly为yes

2、设置一个key

3、观察bin目录的变化

总结：

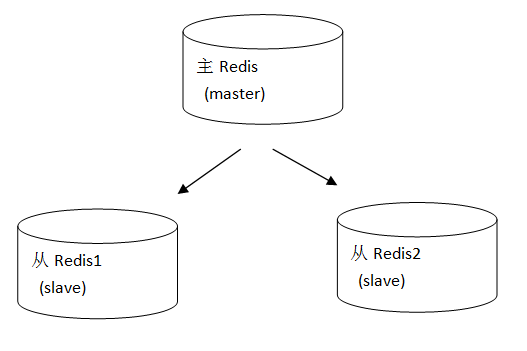
​ 优点:持久化良好，能包证数据的完整性

​ 缺点:大大降低了redis系统的存取速度

## **主从复制（了解）**

### **什么是主从复制**

​ 持久化保证了即使redis服务重启也会丢失数据，因为redis服务重启后会将硬盘上持久化的数据恢复到内存中，但是当redis服务器的硬盘损坏了可能会导致数据丢失，如果通过redis的主从复制机制就可以避免这种单点故障，如下图：



说明：

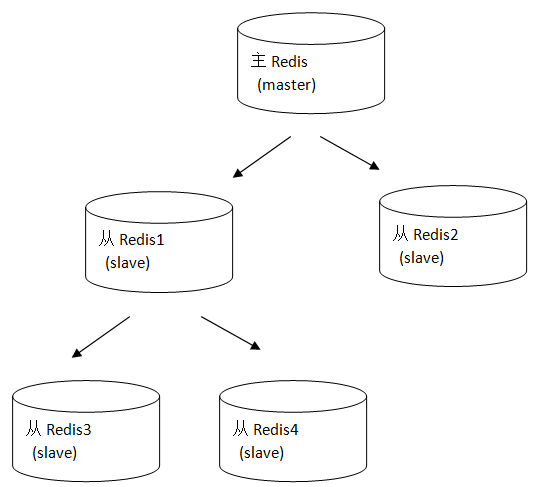
主redis中的数据有两个副本（replication）即从redis1和从redis2，即使一台redis服务器宕机其它两台redis服务也可以继续提供服务。

主redis中的数据和从redis上的数据保持实时同步，当主redis写入数据时通过主从复制机制会复制到两个从redis服务上。

只有一个主redis，可以有多个从redis。

主从复制不会阻塞master，在同步数据时，master可以继续处理client 请求

一个redis可以即是主又是从，如下图：



### **主从配置**

#### 1.10.2.1主redis配置

无需特殊配置。

#### 1.10.2.2**从redis配置**

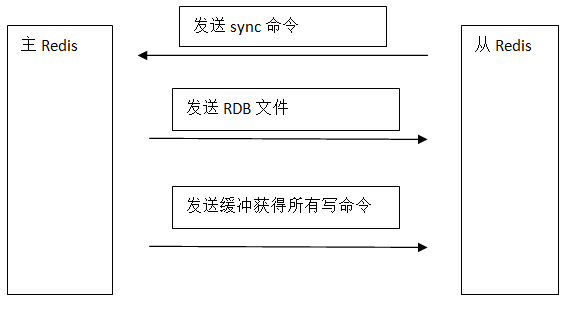
修改从redis服务器上的redis.conf文件，添加**slaveof** **主redis ip** **主redis**端口

IMG_258

上边的配置说明当前该从redis服务器所对应的主redis是192.168.101.3，端口是6379

### **主从复制过程**

主从复制过程如下图：



复制过程说明：

1、 slave 服务启动，slave 会建立和master 的连接，发送sync 命令。

2、master启动一个后台进程将数据库快照保存到RDB文件中

3、master 就发送RDB文件给slave

4、slave 将文件保存到磁盘上，然后加载到内存恢复

5、master把缓存的命令转发给slave

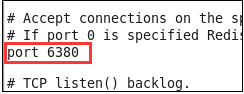
注意：主死了，从只能读

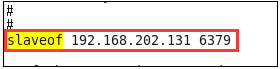
### **配置主从赋值**

1、拷贝redis

cd/usr/local/java  
cp -r redis redis-6380  
mkdir -p /a/b/c

2、修改redis-6380的redis.conf





3、分别开启主从redis，并在主redis存入数据，测试效果

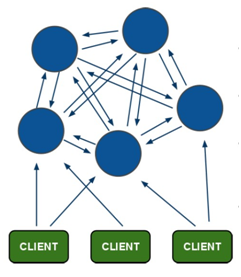
​ 效果：主从数据库数据一致

4、关闭主redis，使用从redis存入数据

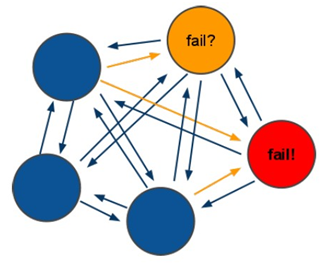
IMG_262

## redis集群的搭建

### redis-cluster架构图



redis-cluster投票:容错



架构细节:

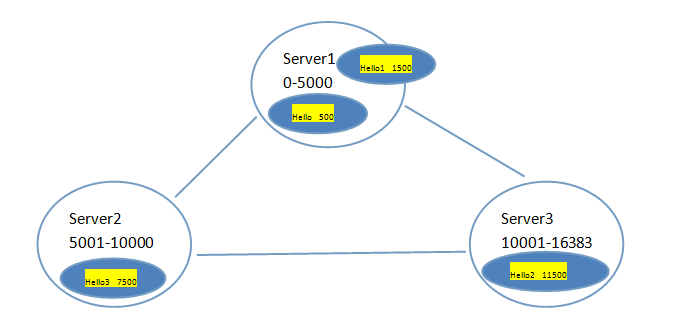
(1)所有的redis节点彼此互联(PING-PONG机制)

(2)节点的fail是通过集群中超过半数的节点检测失效时才生效.

(3)集群中<font color=red>有一个节点fail，则整个集群都会fail</font>

(4)redis-cluster把所有的物理节点映射到[0-16383]slot上，cluster 负责维护node<->slot<->value

Redis 集群中内置了 16384 个哈希槽，当需要在Redis 集群中放置一个 key-value 时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点



### redis集群的搭建

Redis集群中至少应该有三个节点。要保证集群的高可用，需要每个节点有一个备份机。

Redis集群至少需要6台服务器。

搭建伪分布式。可以使用一台虚拟机运行6个redis实例。需要修改redis的端口号7001-7006

### 集群搭建环境

使用ruby脚本搭建集群,需要安装ruby。

[root@upload ~]# yum install ruby  
[root@upload ~]# yum install rubygems  
[root@upload ~]# gem install redis-3.0.0.gem   
Successfully installed redis-3.0.0  
[root@localhost ~]# cd redis-3.0.0/src  
[root@localhost src]# ll \*.rb  
-rwxrwxr-x. 1 root root 48141 Apr  1 2015redis-trib.rb

### **搭建步骤**

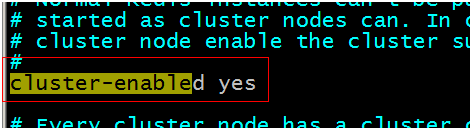
注意：必须删除dump.rdb和appendonly.aof文件

搭建伪分布式，需要6个redis实例放到/usr/java/redis-cluster目录下，并且运行在不同的端口7001-7006

cp -r /usr/java/redis /usr/java/redis-cluster/redis-7001

#### **1.11.4.1创建6个redis实例**

每个实例运行在不同的端口。需要修改redis.conf配置文件。配置文件中还需要把cluster-enabled yes前的注释去掉。



#### **1.11.4.2启动每个redis实例**

创建启动集群的脚本：start-all.sh 放在/usr/local/redis-cluster目录下。

cd redis-7001  
./bin/redis-server bin/redis.conf  
cd ..  
cd redis-7002  
./bin/redis-server bin/redis.conf  
cd ..  
cd redis-7003  
./bin/redis-server bin/redis.conf  
cd ..  
cd redis-7004  
./bin/redis-server bin/redis.conf  
cd ..  
cd redis-7005  
./bin/redis-server bin/redis.conf  
cd ..  
cd redis-7006  
./bin/redis-server bin/redis.conf  
cd ..   
[root@localhost redis-cluster]# chmod 777 start-all.sh

创建关闭集群的脚本：shutdown-all.sh，放在/usr/local/redis-cluster目录下。

cd redis-7001

./redis7001/ bin/redis-cli -p 7001 shutdown

./redis7001/ bin/redis-cli -p 7002 shutdown

./redis7001/ bin/redis-cli -p 7003 shutdown

./redis7001/ bin/redis-cli -p 7004 shutdown

./redis7001/ bin/redis-cli -p 7005 shutdown

./redis7001/ bin/redis-cli -p 7006 shutdown

[root@localhost redis-cluster]# chmod 777shutdown-all.sh

#### **1.11.4.3使用ruby搭建集群**

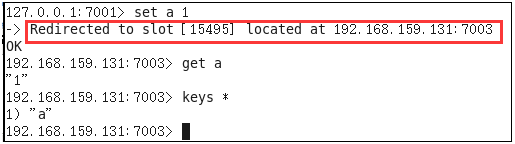
**切换到\*.rb目录**

[root@localhost src]# ./redis-trib.rb create --replicas 1 192.168.25.153:7001 192.168.25.153:7002 192.168.25.153:7003 192.168.25.153:7004 192.168.25.153:7005 192.168.25.153:7006  
>>> Creating cluster  
Connecting to node 192.168.25.153:7001: OK  
Connecting to node 192.168.25.153:7002: OK  
Connecting to node 192.168.25.153:7003: OK  
Connecting to node 192.168.25.153:7004: OK  
Connecting to node 192.168.25.153:7005: OK  
Connecting to node 192.168.25.153:7006: OK  
>>> Performing hash slots allocation on 6 nodes...  
Using 3 masters:  
192.168.25.153:7001  
192.168.25.153:7002  
192.168.25.153:7003  
Adding replica 192.168.25.153:7004 to 192.168.25.153:7001  
Adding replica 192.168.25.153:7005 to 192.168.25.153:7002  
Adding replica 192.168.25.153:7006 to 192.168.25.153:7003  
M: 2e48ae301e9c32b04a7d4d92e15e98e78de8c1f3 192.168.25.153:7001  
   slots:0-5460 (5461 slots) master  
M: 8cd93a9a943b4ef851af6a03edd699a6061ace01 192.168.25.153:7002  
   slots:5461-10922 (5462 slots) master  
M: 2935007902d83f20b1253d7f43dae32aab9744e6 192.168.25.153:7003  
   slots:10923-16383 (5461 slots) master  
S: 74f9d9706f848471583929fc8bbde3c8e99e211b 192.168.25.153:7004  
   replicates 2e48ae301e9c32b04a7d4d92e15e98e78de8c1f3  
S: 42cc9e25ebb19dda92591364c1df4b3a518b795b 192.168.25.153:7005  
   replicates 8cd93a9a943b4ef851af6a03edd699a6061ace01  
S: 8b1b11d509d29659c2831e7a9f6469c060dfcd39 192.168.25.153:7006  
   replicates 2935007902d83f20b1253d7f43dae32aab9744e6  
Can I set the above configuration? (type 'yes' to accept): yes  
>>> Nodes configuration updated  
>>> Assign a different config epoch to each node  
>>> Sending CLUSTER MEET messages to join the cluster  
Waiting for the cluster to join.....  
>>> Performing Cluster Check (using node 192.168.25.153:7001)  
M: 2e48ae301e9c32b04a7d4d92e15e98e78de8c1f3 192.168.25.153:7001  
   slots:0-5460 (5461 slots) master  
M: 8cd93a9a943b4ef851af6a03edd699a6061ace01 192.168.25.153:7002  
   slots:5461-10922 (5462 slots) master  
M: 2935007902d83f20b1253d7f43dae32aab9744e6 192.168.25.153:7003  
   slots:10923-16383 (5461 slots) master  
M: 74f9d9706f848471583929fc8bbde3c8e99e211b 192.168.25.153:7004  
   slots: (0 slots) master  
   replicates 2e48ae301e9c32b04a7d4d92e15e98e78de8c1f3  
M: 42cc9e25ebb19dda92591364c1df4b3a518b795b 192.168.25.153:7005  
   slots: (0 slots) master  
   replicates 8cd93a9a943b4ef851af6a03edd699a6061ace01  
M: 8b1b11d509d29659c2831e7a9f6469c060dfcd39 192.168.25.153:7006  
   slots: (0 slots) master  
   replicates 2935007902d83f20b1253d7f43dae32aab9744e6  
[OK] All nodes agree about slots configuration.  
>>> Check for open slots...  
>>> Check slots coverage...  
[OK] All 16384 slots covered.  
[root@localhost redis-cluster]#

#### **1.11.4.4测试**

启动时使用-c参数来启动集群模式，命令如下：

./redis-cli -c -p 7001



#### **1.11.4.5. redis cluster命令**

cluster info   #打印集群的信息  
cluster nodes  #列出集群当前已知的所有节点(node)，以及这些节点的相关信息

## 课堂总结

|  |
| --- |
| ----------------------------redis------------------------------  一、redis的介绍  redis是c语言编写的高性能的k-v形式的数据库，数据存储在内存中  二、redis的安装  yum install gcc-c++  tar -zxvf redis-3.0.0.tar.gz  cd /usr/upload/redis-3.0.0  make  make install -PREFIX=/usr/java/redis  cp /usr/upload/redis-3.0.0/redis.conf /usr/java/redis/bin  vim /usr/java/redis/bin/redis.conf  daemonize yes  三、redis的启动和关闭  1、启动  前端启动：  ./bin/redis-server  后端启动：  ./bin/redis-server redis.conf  2、关闭  ./bin/redis-cli shutdown  四、redis的基本命令  keys \* ：查看所有key值  exists[ɪɡˈzɪsts]：判断key值是否存在  expire[ɪkˈspaɪər]和ttl：设置和查看key的过期时间  五、redis的多数库  1、redis实例中提供了下标0-15的16个数据库，不能修改数据库的名字通过select切换数据库  2、flushall：清空所有数据库的数据  3、flushdb：清空当前数据库的数据  六、redis五种数据类型  赋值 取值 删除 特点  string set get del k-v  hash hset k k v hget k k hdel k k k-map  list lpush/rpush lrange list 0 -1 lrem list 0 v 双向链表  set sadd smambers srem 无序，不可重复  zset zadd zrange zset 0 -1 zrem zset v 有序，不可重复  其他命令：  incr num  decr nun  七、持久化  1、rdb：默认  缺点：不能保证数据的完整性  优点：不影响性能    实验：  1、删除dump.rdb  2、set test abc  3、关闭服务端观察bin目录变化  2、aof：配置  优点：保证数据的完整性  缺点：影响性能  实验：  1、设置appendonly yes  2、set test1 abc  3、观察appendonly.aof的变化  八、主从复制（了解）  1、配置步骤  主：无须配置  从：slaveof ip port  2、原理：  a、从会往主发送sync命令  b、主会往从发送rdb文件  c、主往从发送命令  注意：主死了，从只能读  九、redis集群搭建  1、redis集群是多少台？  投票容错：3台 高可用：3 共6台    2、为什么集群中有一个节点死了，整个集群都挂了？  redis集群中有16384个槽平均分给了3个节点，存数据：crc16(key)%16384=0~16383  3、搭建步骤  a、安装ruby环境  yum install ruby  yum install rubygems  gem install redis-3.0.0.gem  b、创建6个redis节点，并启动  rm dump.rdb  rm appendonly.aof  cp -r /usr/java/redis /usr/java/redis-cluster/redis-7001  vim /usr/java/redis-cluster/redis-7001/bin/redis.conf：  port 7001  cluster-enable yes  c、使用ruby脚本创建redis集群  ./redis-trib.rb create --replicas 1 192.168.25.153:7001 ... ...  d、测试  ./redis-cli -c -p 7001  set hello abc  set hello1 abc---->7003  get hello----->7001 |

# SpringDataRedis

## **Spring Data 介绍**

Spring Data是一个用于简化数据库访问的开源框架。其主要目标是使得对数据的访问变得方便快捷，包含多个子项目：

**Spring Data JDBC**- 对JDBC的Spring Data存储库支持。

**Spring Data JPA** - 对JPA的Spring Data存储库支持。

**Spring Data MongoDB** - 对MongoDB的基于Spring对象文档的存储库支持。

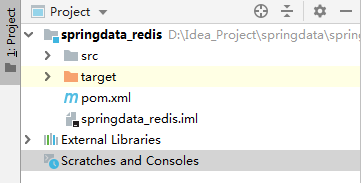
**Spring Data Redis** - 从Spring应用程序轻松配置和访问Redis。

## **Spring Data Redis 介绍**

Spring Data Redis 是属于 Spring Data 下的一个模块，作用就是简化对于 redis 的操作。

## **Spring Data Redis 使用**

### **创建工程**



### **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>  
    <parent>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-parent</artifactId>  
        <version>2.1.6.RELEASE</version>  
    </parent>  
​  
    <groupId>com.usian</groupId>  
    <artifactId>springdata\_redis</artifactId>  
    <version>1.0-SNAPSHOT</version>  
​  
    <properties>  
        <java.version>1.8</java.version>  
    </properties>  
​  
    <dependencies>  
        <!-- springBoot的启动器 -->  
        <dependency>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-starter-web</artifactId>  
        </dependency>  
        <!-- Spring Data Redis的启动器 -->  
        <dependency>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-starter-data-redis</artifactId>  
        </dependency>  
        <!--junit 的启动器-->  
        <dependency>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-starter-test</artifactId>  
        </dependency>  
    </dependencies>  
</project>

### **application.yml**

spring:  
 redis:  
   cluster:  
     nodes:  
       - 192.168.233.131:7001  
       - 192.168.233.131:7002  
       - 192.168.233.131:7003  
       - 192.168.233.131:7004  
       - 192.168.233.131:7005  
       - 192.168.233.131:7006  
   jedis:  
     pool:  
       max-active: 20 #连接池最大连接数  
       max-idle: 10 #连接池中的最大空闲连接  
       min-idle: 5 # 连接池中的最小空闲连接

### **config**

package com.usian.config;  
/\*\*  
 \* 完成对Redis的整合的一些配置  
 \*/  
@Configuration  
public class RedisConfig {​  
    /\*\*  
     \* 创建RedisTemplate:用于执行Redis操作的方法  
     \*/  
    @Bean  
    public RedisTemplate<String, Object> getRedisTemplate(RedisConnectionFactory factory) {  
        RedisTemplate<String, Object> template = new RedisTemplate<String, Object>();  
        template.setConnectionFactory(factory);  
        //为key设置序列化器  
        template.setKeySerializer(new StringRedisSerializer());  
        //为value设置序列化器  
        template.setValueSerializer(new StringRedisSerializer());  
        return template;  
   }  
}

### **App**

package com.usian;  
​@SpringBootApplication  
public class App {  
 public static void main(String[] args) {  
 SpringApplication.run(App.class, args);  
 }  
}

### **pojo**

package com.usian.pojo;  
public class User implements Serializable {  
 private Integer id;  
 private String name;  
 private Integer age;  
}

### **测试**

package com.usian.test;  
/\*\*  
 \* Spring Data Redis测试  
 \*/  
@RunWith(SpringJUnit4ClassRunner.class)  
@SpringBootTest(classes= App.class)  
public class RedisTest {  
​  
 @Autowired  
 private RedisTemplate<String, Object> redisTemplate;  
​  
 /\*\*  
 \* 添加一个字符串  
 \*/  
 @Test  
 public void testSet(){  
 this.redisTemplate.opsForValue().set("key", "师姐你好");  
 }  
   
 /\*\*  
 \* 获取一个字符串  
 \*/  
 @Test  
 public void testGet(){  
 String value = (String)this.redisTemplate.opsForValue().get("key");  
 System.out.println(value);  
 }  
 /\*\*  
 \* 基于JSON格式存Users对象  
 \*/  
 @Test  
 public void testSetUsersUseJSON() throws JsonProcessingException {  
 User user = new User();  
 user.setAge(20);  
 user.setName("张三丰");  
 user.setId(1);  
​  
 //Jackson库提供的实现json与bean之间转换工具类  
 ObjectMapper om = new ObjectMapper();  
 this.redisTemplate.opsForValue().set("user\_json", om.writeValueAsString(user));  
 }  
​  
 // 基于JSON格式取Users对象  
 @Test  
 public void testGetUseJSON() throws IOException {  
 //Jackson库提供的实现json与bean之间转换工具类  
 ObjectMapper om = new ObjectMapper();  
 User user = om.readValue(this.redisTemplate.opsForValue().get("user\_json").toString(),   
         User.class);  
 System.out.println(user);  
 }  
 // 基于JSON格式存Users对象  
 @Test  
 public void testSetUsersUseJSON2(){  
 User user = new User();  
 user.setAge(20);  
 user.setName("李四丰");  
 user.setId(1);  
 this.redisTemplate.setValueSerializer(new   
                                              Jackson2JsonRedisSerializer(User.class));  
 this.redisTemplate.opsForValue().set("user\_json2", user);  
 }  
 // 基于JSON格式取Users对象  
 @Test  
 public void testGetUseJSON2(){  
 this.redisTemplate.setValueSerializer(new   
                                         Jackson2JsonRedisSerializer(User.class));  
 User user = (User)this.redisTemplate.opsForValue().get("user\_json2");  
 System.out.println(user);  
 }  
}

**问题：**

​ 每次存取pojo数据都要重新设置value的序列化器

### **设置通用序列化器**

package com.usian.config;  
​/\*\*  
 \* 配置RedisTemplate  
 \*/  
@Configuration  
public class RedisConfig {  
​  
    // 创建RedisTemplate:用于执行Redis操作的方法  
    @Bean  
    public RedisTemplate<String, Object> setRedisTemplate(RedisConnectionFactory factory)  
   {  
        //创建 RedisTemplate  
        RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();  
        redisTemplate.setConnectionFactory(factory);  
​  
        //创建 Redis 中的 key 的序列化器  
        StringRedisSerializer stringRedisSerializer = new StringRedisSerializer();  
        //创建 Redis 中的 value 的序列化器  
        Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new  
                Jackson2JsonRedisSerializer(Object.class);  
          
        ObjectMapper om = new ObjectMapper();  
        // 指定要序列化的field,get和set,以及修饰符范围，ANY是都有包括private和public  
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);  
        // 指定序列化输入的类型，类必须是非final修饰的，final修饰的类比如String,Integer等会抛出异常  
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON\_FINAL);  
        jackson2JsonRedisSerializer.setObjectMapper(om);  
​  
​  
    //设置key:value和hashKey:hashValue的序列化的方式  
        redisTemplate.setKeySerializer(stringRedisSerializer);  
        redisTemplate.setValueSerializer(jackson2JsonRedisSerializer);  
        redisTemplate.setHashKeySerializer(stringRedisSerializer);  
        redisTemplate.setHashValueSerializer(jackson2JsonRedisSerializer);  
        //初始实例化redisTemplate  
        redisTemplate.afterPropertiesSet();  
        return redisTemplate;  
   }  
}

# ElasticSearch

## **为什么要用ElasticSearch？**

​ 当我们访问购物网站的时候，我们可以根据我们随意所想的内容输入关键字就可以查询出相关的内容，这是怎么做到呢？这些随意的数据不可能是根据数据库的字段查询的，那是怎么查询出来的呢，为什么千奇百怪的关键字都可以查询出来呢？​ 答案就是全文检索服务，ElasticSearch是一个基于Lucene的全文检索服务器，而lucene采用了词元匹配方案。举个例子：北京天安门----Lucene切分词：北京  京天  天安  安门  等等这些词元，当我们搜索的时候这些词元都可以检索到北京天安门。

## **ElasticSearch介绍**

​ ElasticSearch是一个基于[Lucene](https://baike.baidu.com/item/Lucene/6753302)的搜索服务器。它提供了一个基于[RESTful](https://www.jianshu.com/p/91600da4df95) web接口的分布式全文搜索引擎。ElasticSearch是用Java语言开发的，并作为Apache许可条款下的开放源码发布，是一种流行的企业级搜索引擎。ElasticSearch用于云计算中，能够达到实时搜索，稳定，可靠，快速，安装使用方便。根据DB-Engines的排名显示，ElasticSearch是最受欢迎的企业搜索引擎，其次是Apache Solr（也是基Lucene）。

总结：

​ 1、elasticsearch是一个基于Lucene的分布式全文检索服务器。

​ 2、elasticsearch隐藏了Lucene的复杂性，对外提供Restful 接口来操作索引、搜索。

es和solr选择哪个？

1.如果你公司现在用的solr可以满足需求就不要换了。

2.如果你公司准备进行全文检索项目的开发，建议优先考虑elasticsearch，因为像Github这样大规模的搜索都在用它。

## **Lucene->ES的发展**

​ 多年前，一个叫做Shay Banon的刚结婚不久的失业开发者，由于妻子要去伦敦学习厨师，他便跟着也去了。在他找工作的过程中，为了给妻子构建一个食谱的搜索引擎，他开始构建一个早期版本的Lucene。

​ 直接基于Lucene工作会比较困难，所以Shay开始抽象Lucene代码以便Java程序员可以在应用中添加搜索功能。他发布了他的第一个开源项目，叫做“Compass”。

​ 后来Shay找到一份工作，这份工作处在高性能和内存数据网格的分布式环境中，因此高性能的、实时的、分布式的搜索引擎也是理所当然需要的。然后他决定重写Compass库使其成为一个独立的服务叫做Elasticsearch。

​ 第一个公开版本出现在2010年2月，在那之后Elasticsearch已经成为Github上最受欢迎的项目之一，代码贡献者超过300人。一家主营Elasticsearch的公司就此成立，他们一边提供商业支持一边开发新功能，不过Elasticsearch将永远开源且对所有人可用。

## **原理与应用**

### **索引结构**

​ 下图是ElasticSearch的索引结构，右边黑蓝色色部分是原始文档，左边黄色部分是逻辑结构，逻辑结构也是为了更好的去描述ElasticSearch的工作原理及去使用物理结构中的索引文件。



### **倒排索引**

倒排索引（Inverted index）:也常被称为反向索引，倒排索引是**从关键字到文档**的映射（已知关键字求文档）。

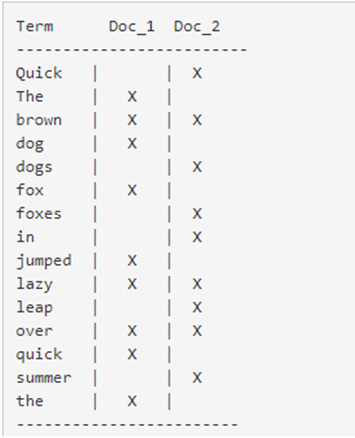
逻辑结构部分是一个倒排索引表，由三部分组成：

1、将搜索的文档最终以<font color=red>Document</font>方式存储起来。

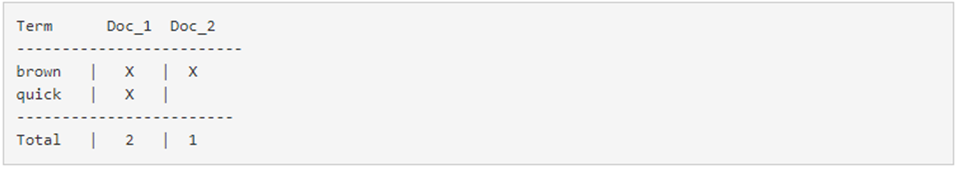
2、将要搜索的文档内容分词，所有不重复的词组成<font color=red>分词</font>列表。

3、每个分词和docment都有<font color=red>关联</font>。

如下：



现在，如果我们想搜索 包含quick brown词条的文档：



两个文档都匹配，但是第一个文档比第二个匹配度更高。如果我们使用仅计算匹配词条数量的简单 相似性算法 ，那么，我们可以说，对于我们查询的相关性来讲，第一个文档比第二个文档更佳。

### **RESTful应用方法**

如何使用es？

Elasticsearch提供 RESTful Api接口进行索引、搜索，并且支持多种客户端。



## 安装 ElasticSearch

### 环境需求

1、jdk必须是jdk1.8.0\_131以上版本。

2、ElasticSearch 需要至少4096 的线程池才能正常启动，ES 至少需要 65536 的文件创建权限，所以需要为虚拟机分配至少1.5G以上的内存

3、从5.0开始，ElasticSearch 安全级别提高了，不允许采用root帐号启动

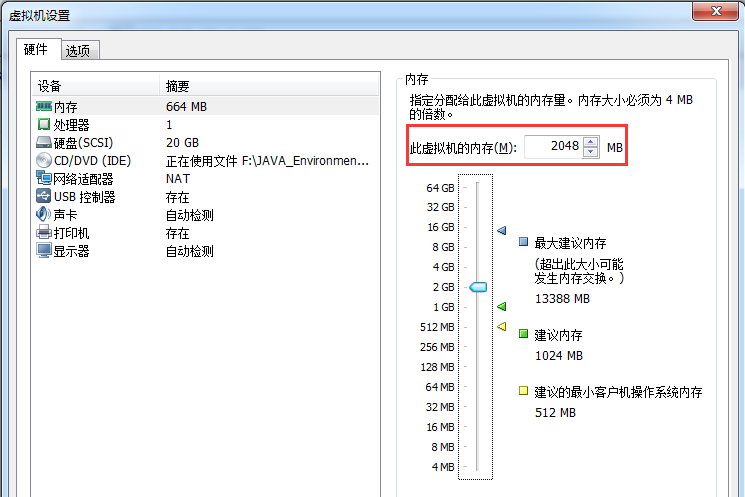
4、Elasticsearch的插件要求至少3.5以上版本

### 安装ES

### **下载**

ElasticSearch官网：<https://www.elastic.co/cn/>

### **设置虚拟机内存**



### **创建用户**

从5.0开始，ElasticSearch 安全级别提高了，不允许采用root帐号启动，所以我们要添加一个用户。

1.创建elk 用户组

groupadd elk

2.创建用户admin

useradd admin  
passwd admin

3.将admin用户添加到elk组

usermod -G elk admin

5.为用户分配权限

#chown将指定文件的拥有者改为指定的用户或组 -R处理指定目录以及其子目录下的所有文件  
chown -R admin:elk /usr/upload  
chown -R admin:elk /usr/java

切换用户：

su admin

### **.安装**

ES是Java开发的应用，解压即安装：

tar -zxvf elasticsearch-6.2.3.tar.gz -C /usr/java

### **ES目录结构**

bin 目录：可执行文件包  
config 目录：配置相关目录  
lib 目录：ES 需要依赖的 jar 包，ES 自开发的 jar 包  
logs 目录：日志文件相关目录  
modules 目录：功能模块的存放目录，如aggs、reindex、geoip、xpack、eval  
plugins 目录：插件目录包，三方插件或自主开发插件  
data 目录：在 ES 启动后，会自动创建的目录，内部保存 ES 运行过程中需要保存的数据。

## **配置文件**

ES安装目录config中配置文件如下：

​ elasticsearch.yml：用于配置Elasticsearch运行参数

​ jvm.options：用于配置Elasticsearch JVM设置

​ log4j2.properties：用于配置Elasticsearch日志

### **elasticsearch.yml**

本项目配置如下：

cluster.name: usian  
node.name: usian\_node\_1  
network.host: 0.0.0.0  
http.port: 9200  
transport.tcp.port: 9300  
node.master: true  
node.data: true  
discovery.zen.ping.unicast.hosts: ["0.0.0.0:9300", "0.0.0.0:9301"]  
bootstrap.memory\_lock: false  
path.data: /usr/java/elasticsearch-6.2.3/data  
path.logs: /usr/java/elasticsearch-6.2.3/logs  
http.cors.enabled: true  
http.cors.allow-origin: /.\*/

注意意path.data和path.logs路径配置正确。

常用的配置项如下：

cluster.name:  
       配置elasticsearch的集群名称，默认是elasticsearch。建议修改成一个有意义的名称。     
node.name:  
     节点名，通常一台物理服务器就是一个节点，es会默认随机指定一个名字，建议指定一个有意义的名称，方便管理一个或多个节点组成一个cluster集群，集群是一个逻辑的概念，节点是物理概念，后边章节会详细介绍。  
path.data:  
       设置索引数据的存储路径，默认是es\_home下的data文件夹，可以设置多个存储路径，用逗号隔开。        
path.logs:  
       设置日志文件的存储路径，默认是es\_home下的logs文件夹        
bootstrap.memory\_lock: true  
       设置为true可以锁住ES使用的内存，避免内存与swap分区交换数据。          
network.host:    
       设置绑定主机的ip地址，设置为0.0.0.0表示绑定任何ip，允许外网访问，生产环境建议设置为具体的ip。     
http.port: 9200  
       设置对外服务的http端口，默认为9200。        
transport.tcp.port: 9300   
       集群结点之间通信端口        
node.master:   
       指定该节点是否有资格被选举成为master结点，默认是true，如果原来的master宕机会重新选举新master。   
node.data:   
       指定该节点是否存储索引数据，默认为true。  
discovery.zen.ping.unicast.hosts:[“host1:port”, “host2:port”, “…”]    
       设置集群中master节点的初始列表。  
discovery.zen.ping.timeout: 3s    
       设置ES自动发现节点连接超时的时间，默认为3秒，如果网络延迟高可设置大些。  
http.cors.enabled：  
   是否支持跨域，默认为false  
http.cors.allow-origin：  
   当设置允许跨域，默认为\*,表示支持所有域名

### **jvm.options**

设置最小及最大的JVM堆内存大小：

在jvm.options中设置 -Xms和-Xmx：

1） 两个值设置为相等

2） 将Xmx 设置为不超过物理内存的一半。

默认内存占用太多了，我们调小一些：

-Xms512m  
-Xmx512m

### **log4j2.properties**

日志文件设置，ES使用log4j，注意日志级别的配置。

## **启动ES**

### **启动和关闭**

1、启动

./elasticsearch  
#或  
./elasticsearch -d

2、关闭

ps-ef|grep elasticsearch  
​  
kill -9 pid

### **解决内核问题**

我们使用的是centos6，其linux内核版本为2.6。而Elasticsearch的插件要求至少3.5以上版本。不过没关系，我们禁用这个插件即可。

修改elasticsearch.yml文件，在最下面添加如下配置：

bootstrap.system\_call\_filter: false

### **解决文件创建权限问题**

[1]: max file descriptors [4096] for elasticsearch process likely too low, increase to at least [65536]

Linux 默认来说，一般限制应用最多创建的文件是 4096个。但是 ES 至少需要 65536 的文件创建权限。我们用的是admin用户，而不是root，所以文件权限不足。

使用root用户修改配置文件:

vim /etc/security/limits.conf

添加下面的内容：

\* soft nofile 65536  
\* hard nofile 65536

### **解决线程开启限制问题**

[2]: max number of threads [1024] for user [admin] is too low, increase to at least [4096]

​ 默认的 Linux 限制 root 用户开启的进程可以开启任意数量的线程，其他用户开启的进程可以开启1024 个线程。必须修改限制数为4096+。因为 ES 至少需要 4096 的线程池预备。

​ 如果虚拟机的内存是 1G，最多只能开启 3000+个线程数。至少为虚拟机分配 1.5G 以上的内存。

使用root用户修改配置：

vim /etc/security/limits.d/90-nproc.conf

修改下面的内容：

\* soft nproc 1024

改为：

\* soft nproc 4096

### **解决虚拟内存问题**

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

ES 需要开辟一个 262144字节以上空间的虚拟内存。Linux 默认不允许任何用户和应用直接开辟虚拟内存。

vim /etc/sysctl.conf

追加下面内容：

vm.max\_map\_count=655360 #限制一个进程可以拥有的VMA(虚拟内存区域)的数量

然后执行命令，让sysctl.conf配置生效：

sysctl -p

## **测试**

​ ES 中只要启动了任意一个 ES 应用就是启动了一个 ES的 cluster 集群。默认的 ES集群命名为 elasticsearch。如果启动了多个应用（可以在多个节点或单一节点上启动多个应用），默认的ES 会自动找集群做加入集群的过程。

浏览器访问：<http://192.168.233.134:9200>

返回结果如下：  
{  
  "name" : " usian\_node\_1", # node name 结点名称。随机分配的结点名称  
  "cluster\_name" : "usian", # cluster name 集群名称。 默认的集群名称  
  "cluster\_uuid" : "RqHaIiYjSoOyrTGq3ggCOA", # 集群唯一 ID  
  "version" : {  
    "number" : "6.2.3", #版本号  
    "build\_hash" : "c59ff00",   
    "build\_date" : "2018-03-13T10:06:29.741383Z",#发布日期  
    "build\_snapshot" : false,#是否快照版本  
    "lucene\_version" : "7.2.1",#lucene版本号  
    "minimum\_wire\_compatibility\_version" : "5.6.0",  
    "minimum\_index\_compatibility\_version" : "5.0.0"  
 },  
  "tagline" : "You Know, for Search"  
}

## 安装Kibana

## **什么是Kibana**

​ Kibana是ES提供的一个基于Node.js的基于Node.js的管理控制台, 可以很容易实现高级的数据分析和可视化，以图标的形式展现出来。

​ kibana可以用来编辑请求语句的，方便学习操作es的语法。有时在进行编写程序，写到查询语句时，往往我会使用kibana进行书写，然后再粘贴到程序中。（不容易出错）

### **下载**

ElasticSearch官网：<https://www.elastic.co/cn/>

### **安装**

在window中安装Kibana很方便，解压即安装

### **修改配置**

修改config/kibana.yml配置：

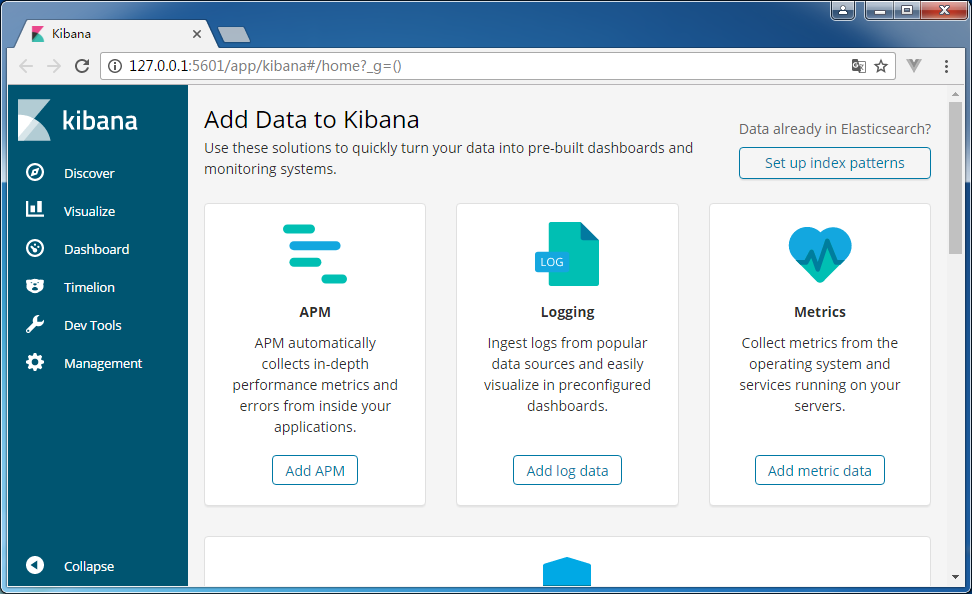
server.port:5601  
server.host:"0.0.0.0" #允许来自远程用户的连接  
elasticsearch.url:http://192.168.233.134:9200 #Elasticsearch实例的URL

### **启动**

./bin/kibana

### **测试**

浏览器访问：<http://127.0.0.1:5601>



## 安装head

### 什么是head

head插件是ES的一个可视化管理插件，用来监视ES的状态，并通过head客户端和ES服务进行交互，比如创建映射、创建索引等。从ES6.0开始，head插件支持使得node.js运行。

### 安装

1、下载head

​ 下载地址：<https://github.com/mobz/elasticsearch-head>

2、安装依赖

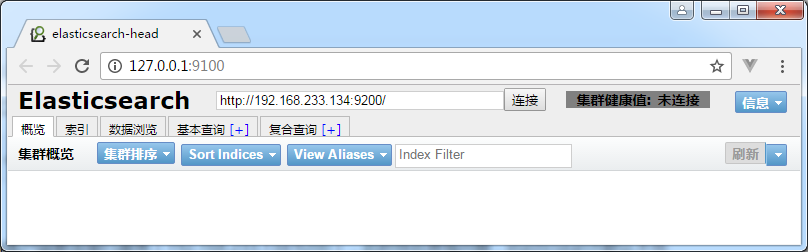
npm install

3、运行

npm run start

### 测试

浏览器访问：<http://127.0.0.1:9100/>



## ES快速入门

​ ES作为一个索引及搜索服务，对外提供丰富的REST接口，快速入门部分的实例使用kibana来测试，目的是对ES的使用方法及流程有个初步的认识。

## **index管理**

### **创建index**

索引。包含若干相似结构的 Document 数据，相当于数据库的database。

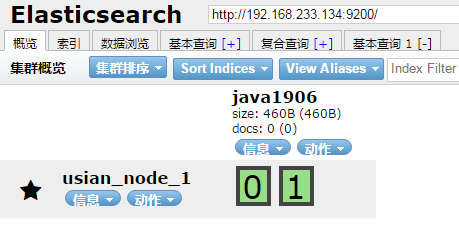
语法：

PUT /java1906  
{  
  "settings": {  
    "number\_of\_shards": 2,  
    "number\_of\_replicas": 0  
 }  
}

**number\_of\_shards** - 是表示一个索引库将拆分成多片分别存储不同的结点，提高了ES的处理能力和高可用性

**number\_of\_replicas**- 是为每个 primary shard分配的replica shard数，**如果只有一台机器，设置为0**

效果：



### **修改index**

注意：索引一旦创建，primary shard 数量不可变化，可以改变replica shard 数量。

PUT /java1906/\_settings  
{  
  "number\_of\_replicas" : 1  
}

ES 中对 shard 的分布是有要求的，有其内置的特殊算法：

​ Replica shard 会保证不和他的那个 primary shard 分配在同一个节点上；如过只有一个节点，则此案例执行后索引的状态一定是yellow。

### **删除index**

DELETE /java1906 [, other\_index]

## **mapping管理**

映射，创建映射就是向索引库中创建field（类型、是否索引、是否存储等特性）的过程，下边是document和field与关系数据库的概念的类比：

​ 索引库（index）--------------------------------Database数据库

​ 类型（type）-----------------------------Table数据表

​ 文档（Document）----------------Row 行

​ 字段（Field）-------------------Columns 列

注意：6.0之前的版本有type（类型）概念，type相当于关系数据库的表，ES6.x 版本之后，type概念被弱化ES官方将在ES7.0版本中彻底删除type。

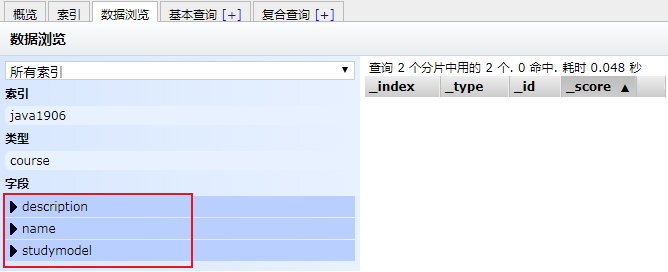
### **创建mapping**

语法：POST index\_name/type\_name/\_mapping

如：

POST /java1906/course/\_mapping  
{  
  "properties": {  
     "name": {  
        "type": "text"  
     },  
     "description": {  
        "type": "text"  
     },  
     "studymodel": {  
        "type": "keyword"  
     }  
 }  
}

效果：



### **查询mapping**

查询所有索引的映射：

GET /java1906/course/\_mapping

### **更新mapping**

映射创建成功可以添加新字段，已有字段不允许更新。

### **删除mapping**

通过删除索引来删除映射。

## **document管理**

### **创建document**

ES中的文档相当于MySQL数据库表中的记录。

#### 3.15.1.1POST语法

此操作为 ES 自动生成 id 的新增 Document 方式。

语法：POST/index\_name/type\_name{fieldname:fieldvalue}

如：

POST /java1906/course/1  
{  
  "name":"python从入门到放弃",  
  "description":"人生苦短，我用Python",  
  "studymodel":"201002"  
}  
​  
POST /java1906/course  
{  
  "name":".net从入门到放弃",  
  "description":".net程序员谁都不服",  
  "studymodel":"201003"  
}

#### **3.15.1.2PUT语法**

此操作为手工指定 id 的 Document 新增方式。

语法：PUT/index\_name/type\_name/id{field\_name:field\_value}

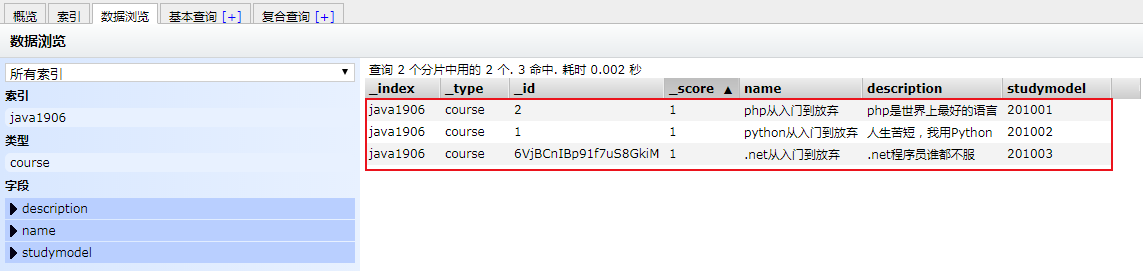
如：

PUT /java1906/course/2  
{  
  "name":"php从入门到放弃",  
  "description":"php是世界上最好的语言",  
  "studymodel":"201001"  
}

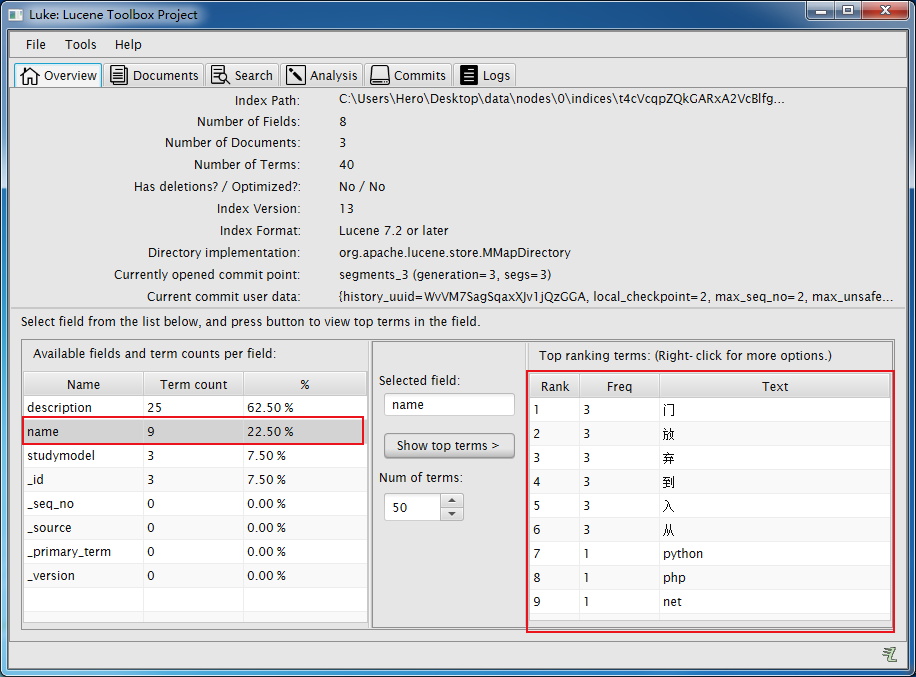
结果：

{  
  "\_index": "test\_index", 新增的 document 在什么 index 中，  
  "\_type": "my\_type", 新增的 document 在 index 中的哪一个 type 中。  
  "\_id": "1", 指定的 id 是多少  
  "\_version": 1, document 的版本是多少，版本从 1 开始递增，每次写操作都会+1  
  "result": "created", 本次操作的结果，created 创建，updated 修改，deleted 删除  
  "\_shards": { 分片信息  
      "total": 2, 分片数量只提示 primary shard  
      "successful": 1, 数据 document 一定只存放在 index 中的某一个 primary shard 中  
      "failed": 0  
 },  
  "\_seq\_no": 0,   
  "\_primary\_term": 1  
}

通过head查询数据：



通过luke查看分词列表：



### **查询document**

语法：

​ GET /index\_name/type\_name/id

或

​ GET /index\_name/type\_name/\_search?q=field\_name:field\_value

如：根据课程id查询文档

GET /java1906/course/1

如：查询所有记录

GET /java1906/course/\_search

如：查询名称中包括php 关键字的的记录

GET /java1906/course/\_search?q=name:门

结果：

{  
  "took": 1, # 执行的时长。单位毫秒  
  "timed\_out": false, # 是否超时  
  "\_shards": { # shard 相关数据  
    "total": 1, # 总计多少个 shard  
    "successful": 1, # 成功返回结果的 shard 数量  
    "skipped": 0,  
    "failed": 0  
 },  
  "hits": { # 搜索结果相关数据  
    "total": 3, # 总计多少数据，符合搜索条件的数据数量  
    "max\_score": 1, # 最大相关度分数，和搜索条件的匹配度  
    "hits": [# 具体的搜索结果  
     {  
        "\_index": "java1906",# 索引名称  
        "\_type": "course", # 类型名称  
        "\_id": "1",# id 值  
        "\_score": 1, # 匹配度分数，本条数据匹配度分数  
        "\_source": { # 具体的数据内容  
          "name": "php从入门到放弃",  
          "description": "php是世界上最好的语言",  
          "studymodel": "201001"  
       }, {  
 "\_index": "java1906",  
 "\_type": "course",  
 "\_id": "2",  
 "\_score": 0.13353139,  
 "\_source": {  
 "name": "php从入门到放弃",  
 "description": "php是世界上最好的语言",  
 "studymodel": "201001"  
 }  
 }, {  
 "\_index": "java1906",  
 "\_type": "course",  
 "\_id": "6ljFCnIBp91f7uS8FkjS",  
 "\_score": 0.13353139,  
 "\_source": {  
 "name": ".net从入门到放弃",  
 "description": ".net程序员谁都不服",  
 "studymodel": "201003"  
 }  
 }  
 ]  
 }  
}

### **删除Document**

​ ES 中执行删除操作时，ES先标记Document为deleted状态，而不是直接物理删除。当ES 存储空间不足或工作空闲时，才会执行物理删除操作，标记为deleted状态的数据不会被查询搜索到（ES 中删除 index ，也是标记。后续才会执行物理删除。所有的标记动作都是为了NRT（近实时）实现）

语法：DELETE/index\_name/type\_name/id

如：

DELETE /java1906/course/3

结果：

{  
  "\_index": "java1906",  
  "\_type": "course",  
  "\_id": "2",  
  "\_version": 2,  
  "result": "deleted",  
  "\_shards": {  
    "total": 1,  
    "successful": 1,  
    "failed": 0  
 },  
  "\_seq\_no": 3,  
  "\_primary\_term": 1  
}

## **ES读写原理**

### **documnet routing（数据路由）**

当客户端创建document的时候，es需要确定这个document放在该index哪个shard上，这个过程就是document routing。

路由过程：

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

　　　　routing：document的\_id，可能是手动指定，也可能是自动生成，决定一个document在哪个shard上

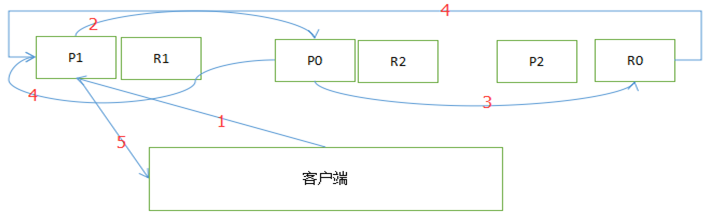
　　　　number\_of\_primary\_shards：主分片。

### **ES document写操作原理**

ES增删改的处理流程：增删改的请求一定作用在主分片上。

假如我们es集群有3个node，每个node上一个主分片一个复制分片,

如下图：



1、第一步 客户端发起一个PUT请求，假如该请求被发送到第一个node节点，那么该节点将成为协调节点

(coordinatingnode)，如图P1所在的节点就是协调节点。他将根据该请求的路由信息计算，该document将被存储 到哪个分片。

2、第二步 通过计算发现该document被存储到p0分片，那么就将请求转发到node2节点。

3、第三步P0根据请求信息创建document，和相应的索引信息，创建完毕后将信息同步到自己的副本节点R0上。

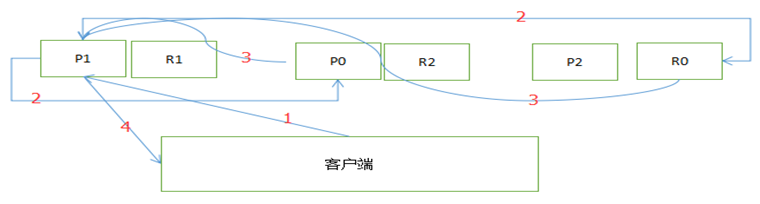
4、第四步P0和R0将通知我们的协调节点，任务完成情况。

5、第五部 协调节点响应客户端最终的处理结果。

### **ES document读操作原理**

假如我们es集群有3个node，每个node上一个主分片一个复制分片,

如下图：



1、第一步 客户端发送读器请求到协调节点(coordinate node)。

2、第二步 协调节点(coordinate node)根据请求信息对document进行路由计算，将请求转发到对应的node，node2

或者node3，此时会使用round-robin随机轮询算法，在primary shard以及其所有replica(副本)中随机选择一

个让读请求负载均衡。

3、第三步 相应接收到请求的节点(node2或者node3)将处理结果返回给协调节点(coordinate node)。

4、第四步 协调节点将最终的结果反馈给客户端。

### **为什么primary shard数量不可变？**

​ 原因：假如我们的集群在初始化的时候有5个primary shard，我们往里边加入一个document id=5，假如hash(5)=23,这时该document 将被加入 (shard=23%5=3)P3这个分片上。如果随后我们给es集群添加一个primary shard ，此时就有6个primary shard，当我们GET id=5 ，这条数据的时候，es会计算该请求的路由信息找到存储他的 primary shard（shard=23%6=5） ，根据计算结果定位到P5分片上。而我们的数据在P3上。所以es集群无法添加primary shard，但是可以扩展replicas shard。

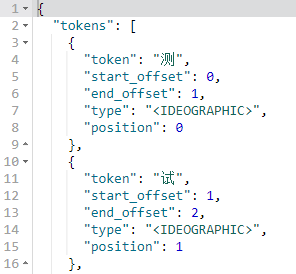
## IK分词器

### 测试分词器

在添加文档时会进行分词，索引中存放的就是一个一个的词（term），当你去搜索时就是拿关键字去匹配词，最终找到词关联的文档。

测试当前索引库使用的分词器：

POST /\_analyze  
{  
  "text":"测试分词器，后边是测试内容：spring cloud实战"  
}

结果如下：

会发现分词的效果将“测试”这个词拆分成两个单字“测”和“试”，这是因为当前索引库使用的分词器对中文就是单字分词。

### 中文分词器

#### 3.17.2.1Lucene自带中文分词器

l StandardAnalyzer：

单字分词：就是按照中文一个字一个字地进行分词。如：“我爱中国”，效果：“我”、“爱”、“中”、“国”。

l CJKAnalyzer

二分法分词：按两个字进行切分。如：“我是中国人”，效果：“我是”、“是中”、“中国”“国人”。

上边两个分词器无法满足需求。

l SmartChineseAnalyzer

对中文支持较好，但扩展性差，扩展词库，禁用词库和同义词库等不好处理

#### 3.17.2.2第三方中文分析器

**paoding**： 庖丁解牛最新版在 <https://code.google.com/p/paoding/> 中最多支持Lucene 3.0，且最新提交的代码在 2008-06-03，在svn中最新也是2010年提交，已经过时，不予考虑。

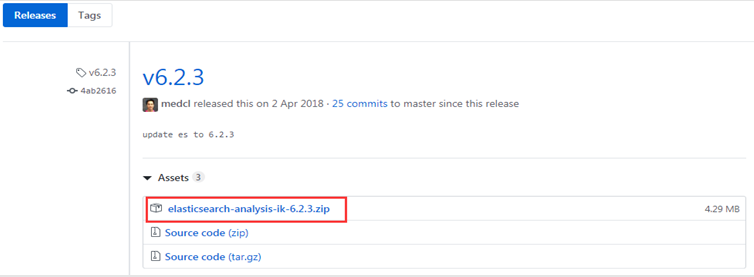
**IK-analyzer**：最新版在<https://code.google.com/p/ik-analyzer/>上，支持Lucene 4.10从2006年12月推出1.0版开始， IKAnalyzer已经推出了4个大版本。最初，它是以开源项目Luence为应用主体的，结合词典分词和文法分析算法的中文分词组件。从3.0版本开 始，IK发展为面向Java的公用分词组件，独立于Lucene项目，同时提供了对Lucene的默认优化实现。在2012版本中，IK实现了简单的分词 歧义排除算法，标志着IK分词器从单纯的词典分词向模拟语义分词衍化。 但是也就是2012年12月后没有在更新。

### 安装IK分词器

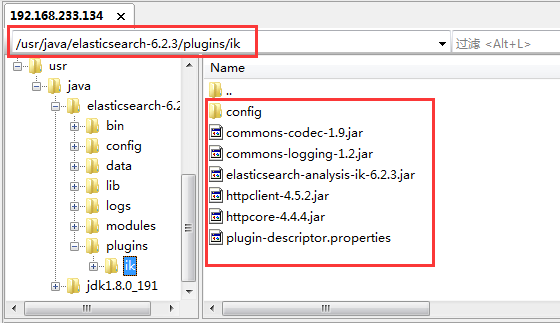
使用IK分词器可以实现对中文分词的效果。

下载IK分词器：（Github地址：<https://github.com/medcl/elasticsearch-analysis-ik>）

1. 下载zip：



2、解压，并将解压的文件拷贝到ES安装目录的plugins下的ik(重命名)目录下，重启es



3、测试分词效果：

POST /\_analyze  
{  
  "text":"中华人民共和国人民大会堂",  
  "analyzer":"ik\_smart"  
}

### 两种分词模式

ik分词器有两种分词模式：ik\_max\_word和ik\_smart模式。

1、ik\_max\_word

​ 会将文本做最细粒度的拆分，比如会将“中华人民共和国人民大会堂”拆分为“中华人民共和国、中华人民、中华、华人、人民共和国、人民、共和国、大会堂、大会、会堂等词语。

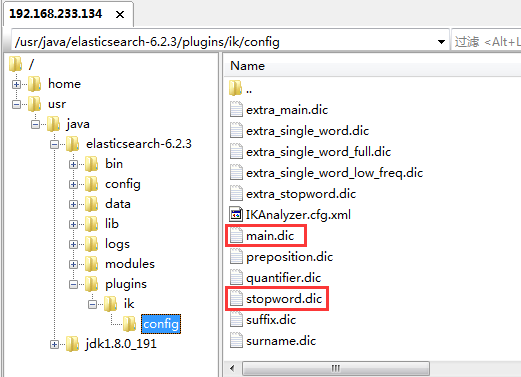
2、ik\_smart

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

### 自定义词库

如果要让分词器支持一些专有词语，可以自定义词库。

iK分词器自带的main.dic的文件为扩展词典，stopword.dic为停用词典。

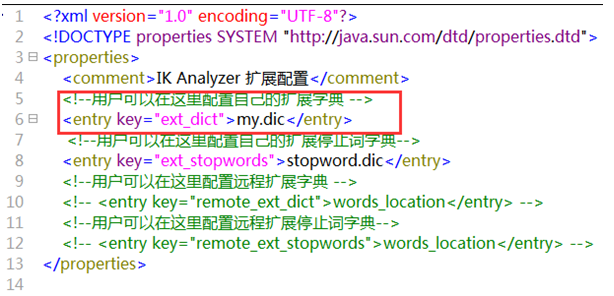


也可以上边的目录中新建一个my.dic文件（注意文件格式为utf-8（不要选择utf-8 BOM））

可以在其中自定义词汇：

比如定义：

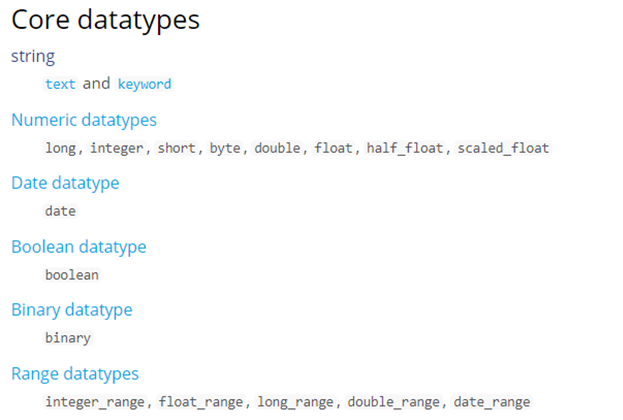
配置文件中配置my.dic，



## field详细介绍

上边章节安装了ik分词器，如何在索引和搜索时去使用ik分词器呢？如何指定field的类型？比如日期类型、数值类型等。

ES6.2核心的字段类型如下：



### field的属性介绍

#### 3.18.1.1type：

通过type属性指定field的类型。

"name":{   
       "type":"text"  
}

#### 3.18.1.2analyzer：

通过analyzer属性指定分词模式。

"name": {  
        "type": "text",  
         "analyzer":"ik\_max\_word"  
   }

上边指定了analyzer是指在索引和搜索都使用ik\_max\_word，如果单独想定义搜索时使用的分词器则可以通过search\_analyzer属性。对于ik分词器建议是索引时使用ik\_max\_word将搜索内容进行细粒度分词，搜索时使用ik\_smart提高搜索精确性。

"name": {  
                  "type": "text",  
                  "analyzer":"ik\_max\_word",  
                  "search\_analyzer":"ik\_smart"  
 }

#### 3.18.1.3index：

通过index属性指定是否<font color=red>索引</font>。默认为index=true，即要进行索引，只有进行索引才可以从索引库搜索到。但是也有一些内容不需要索引，比如：商品图片地址只被用来展示图片，不进行搜索图片，此时可以将index设置为false。 删除索引，重新创建映射，将pic的index设置为false，尝试根据pic去搜索，结果搜索不到数据

"pic": {  
   "type":"text",             
       "index":false  
}

#### 3.18.1.4field索引不存储

​ 如果某个字段内容非常多，业务里面只需要能对该字段进行搜索，比如：商品描述。查看文档内容会再次到mysql或者hbase中取数据，把大字段的内容存在Elasticsearch中只会增大索引，这一点文档数量越大结果越明显，如果一条文档节省几KB，放大到亿万级的量结果也是非常可观的。

如果只想存储某几个字段的原始值到Elasticsearch，可以通过incudes参数来设置，在mapping中的设置如下:

POST /java1906/course/\_mapping  
{  
  "\_source": {  
    "includes":["description"]  
 }  
}

同样，可以通过excludes参数排除某些字段：

POST /java1906/course/\_mapping  
{  
  "\_source": {  
    "excludes":["description"]  
 }  
}

## 常用field类型

### text文本字段

例如：1、创建新映射：

POST /java1906/course/\_mapping  
{  
  "properties": {     
       "name": {  
           "type": "text",  
           "analyzer":"ik\_max\_word",  
           "search\_analyzer":"ik\_smart"  
       },           
      "description": {  
          "type": "text",  
          "analyzer":"ik\_max\_word",  
          "search\_analyzer":"ik\_smart"  
     },  
      "pic":{  
          "type":"text",  
          "index":false  
     },  
      "studymodel":{  
          "type":"text"  
     }  
 }     
}

2、插入文档：

POST /java1906/course/1  
{  
  "name":"python从入门到放弃",  
  "description":"人生苦短，我用Python",  
  "pic":"250.jpg",  
  "studymodel":"201002"  
}

3、查询测试：

GET /java1906/course/\_search?q=name:放弃  
GET /java1906/course/\_search?q=description:人生  
GET /java1906/course /\_search?q=pic:250.jpg

结果：name和description都支持全文检索，pic不可作为查询条件

### keyword关键字字段

上边介绍的text文本字段在映射时要设置分词器，keyword字段为关键字字段，通常搜索keyword是按照整体搜索，所以创建keyword字段<font color=red>索引时是不进行分词</font>的，比如：邮政编码、手机号码、身份证等。keyword字段通常用于过虑、排序、聚合等。

例如：1、更改映射：

POST /java1906/course/\_mapping  
{  
 "properties": {  
       "studymodel":{  
          "type":"keyword"  
       },  
       "description": {  
          "type": "text",  
          "analyzer":"ik\_max\_word",  
          "search\_analyzer":"ik\_smart"  
       },  
       "pic":{  
         "type":"text",  
         "index":false  
       },  
       "name":{  
          "type":"keyword"  
       }  
 }  
}

2、插入文档：

PUT /java1906/course/2

{

"name": "java编程基础",

"description": "java语言是世界第一编程语言",

"pic":"250.jpg",

"studymodel": "201001"

}

3、根据name查询文档：

GET /java1906/course/\_search?q=name:java编程基础

name是keyword类型，所以查询方式是精确查询。

### date日期类型

日期类型不用设置分词器，通常日期类型的字段用于排序。1)format通过format设置日期格式，多个格式使用双竖线||分隔, 每个格式都会被依次尝试, 直到找到匹配的

例如：1、设置允许date字段存储年月日时分秒、年月日及毫秒三种格式。

POST /java1906/course/\_mapping  
{  
 "properties": {  
       "timestamp": {  
         "type":   "date",  
         "format": "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd"  
       }  
     }  
}

2、插入文档：

PUT /java1906/course/3  
{  
"name": "spring开发基础",  
"description": "spring 在java领域非常流行，java程序员都在用。",  
"studymodel": "201001",  
 "pic":"250.jpg",  
 "timestamp":"2018-07-04 18:28:58"  
}

### Numeric类型

下图是ES支持的数值类型：尽可能选择范围小的数据类型,字段的长度越短, 索引和搜索的效率越高;



例如：1、更新已有映射：

POST /java1906/course/\_mapping  
{  
 "properties": {  
 "price": {  
        "type": "double"  
     }  
 }  
}

2、插入文档

PUT /java1906/course/3  
{  
 "name": "spring开发基础",  
 "description": "spring 在java领域非常流行，java程序员都在用。",  
 "studymodel": "201001",  
 "pic":"250.jpg",  
 "timestamp":"2018-07-04 18:28:58",  
 "price":38.6  
}

### 总结

field属性的设置标准：

|  | **标准** |
| --- | --- |
| 分词 | 是否有意义 |
| 索引 | 是否搜索 |
| 存储 | 是否展示 |

## Spring Boot整合ElasticSearch

### ES客户端

ES提供多种不同的客户端：

1、TransportClient

​ ES提供的传统客户端，官方计划8.0版本删除此客户端。

2、RestClient

​ RestClient是官方推荐使用的，它包括两种：Java Low Level RESTClient和 Java HighLevel REST Client。ES在6.0之后提供 JavaHigh Level REST Client， 两种客户端官方更推荐使用 Java High Level RESTClient，不过当前它还处于完善中，有些功能还没有。

### 搭建工程

#### 3.20.2.1pom.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>  
    <parent>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-parent</artifactId>  
        <version>2.1.6.RELEASE</version>  
    </parent>  
​  
    <groupId>com.usian</groupId>  
    <artifactId>springboot\_elasticsearch</artifactId>  
    <version>1.0-SNAPSHOT</version>  
    <properties>  
        <java.version>1.8</java.version>  
    </properties>  
    <dependencies>  
        <dependency>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-starter-web</artifactId>  
        </dependency>  
        <dependency>  
            <groupId>org.elasticsearch.client</groupId>  
            <artifactId>elasticsearch-rest-high-level-client</artifactId>  
        </dependency>  
        <dependency>  
            <groupId>org.elasticsearch</groupId>  
            <artifactId>elasticsearch</artifactId>  
        </dependency>  
        <dependency>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-starter-test</artifactId>  
        </dependency>  
    </dependencies>  
</project>  
​

#### 3.20.2.2application.yml

spring:  
 data:  
   elasticsearch:  
     cluster-nodes: 192.168.233.134:9200

#### 3.20.2.3config

package com.usian.config;  
@Configuration  
public class ElasticsearchConfig extends ElasticsearchProperties{  
​  
 @Bean  
 public RestHighLevelClient getRestHighLevelClient() {  
        String[] hosts = getClusterNodes().split(",");  
        HttpHost[] httpHosts = new HttpHost[hosts.length];  
        for (int i = 0; i < httpHosts.length; i++) {  
            String h = hosts[i];  
            httpHosts[i] = new HttpHost(h.split(":")[0],  
                                        Integer.parseInt(h.split(":")[1]));  
       }  
 return new RestHighLevelClient(RestClient.builder(httpHosts));  
 }  
}

#### 3.20.2.4app

package com.usian;  
​  
@SpringBootApplication  
public class ElasticsearchApp {  
​  
 public static void main(String[] args) {  
 SpringApplication.run(ElasticsearchApp.class, args);  
 }  
}

### 索引管理

### 创建索引库

#### 3.20.4.1api

创建索引库：

PUT /java1906  
{  
  "settings":{  
       "number\_of\_shards" : 2,  
       "number\_of\_replicas" : 0  
 }  
}

创建映射：

POST /java1906/course/\_mapping  
{  
  "\_source": {  
    "excludes":["description"]  
 },   
 "properties": {  
      "name": {  
          "type": "text",  
          "analyzer":"ik\_max\_word",  
          "search\_analyzer":"ik\_smart"  
     },  
      "description": {  
          "type": "text",  
          "analyzer":"ik\_max\_word",  
          "search\_analyzer":"ik\_smart"  
       },  
       "studymodel": {  
          "type": "keyword"  
       },  
       "price": {  
          "type": "float"  
       },  
       "pic":{  
   "type":"text",  
   "index":false  
   },  
       "timestamp": {  
     "type":   "date",  
     "format": "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd"  
   }  
 }  
}

#### 3.20.4.2**Java Client**

package com.usian.test;  
​  
  
@RunWith(SpringJUnit4ClassRunner.class)  
@SpringBootTest(classes = {ElasticsearchApp.class})  
public class IndexWriterTest {  
 @Autowired  
    private RestHighLevelClient restHighLevelClient;  
​  
   //创建索引库  
    @Test  
    public void testCreateIndex() throws IOException {  
        //创建“创建索引请求”对象，并设置索引名称  
        CreateIndexRequest createIndexRequest = new CreateIndexRequest("java1906");  
        //设置索引参数  
        createIndexRequest.settings(Settings.builder().put("number\_of\_shards",1).  
                                    put("number\_of\_replicas",0));  
        createIndexRequest.mapping("course", "{\r\n" +   
       " \"\_source\": {\r\n" +   
       "   \"excludes\":[\"description\"]\r\n" +   
       " }, \r\n" +   
       " \"properties\": {\r\n" +   
       "           \"name\": {\r\n" +   
       "             \"type\": \"text\",\r\n" +   
       "             \"analyzer\":\"ik\_max\_word\",\r\n" +   
       "             \"search\_analyzer\":\"ik\_smart\"\r\n" +   
       "           },\r\n" +   
       "           \"description\": {\r\n" +   
       "             \"type\": \"text\",\r\n" +   
       "             \"analyzer\":\"ik\_max\_word\",\r\n" +   
       "             \"search\_analyzer\":\"ik\_smart\"\r\n" +   
       "           },\r\n" +   
       "           \"studymodel\": {\r\n" +   
       "             \"type\": \"keyword\"\r\n" +   
       "           },\r\n" +   
       "           \"price\": {\r\n" +   
       "             \"type\": \"float\"\r\n" +   
       "           },\r\n" +   
       "           \"timestamp\": {\r\n" +   
       "         \"type\":   \"date\",\r\n" +   
       "         \"format\": \"yyyy-MM-dd HH:mm:ss||yyyy-MM-dd\"\r\n" +   
       "       }\r\n" +   
       " }\r\n" +   
       "}", XContentType.JSON);  
        //创建索引操作客户端  
        IndicesClient indices = restHighLevelClient.indices();  
​  
        //创建响应对象  
        CreateIndexResponse createIndexResponse = indices.create(createIndexRequest);  
        //得到响应结果  
        boolean acknowledged = createIndexResponse.isAcknowledged();  
        System.out.println(acknowledged);  
   }   
 }

### 删除索引库

#### 3.20.4.1api

DELETE /index\_name

#### 3.20.4.2java client

//删除索引库  
 @Test  
 public void testDeleteIndex() throws IOException {  
 //创建“删除索引请求”对象  
 DeleteIndexRequest deleteIndexRequest = new DeleteIndexRequest("java1906");  
 //创建索引操作客户端  
 IndicesClient indices = restHighLevelClient.indices();  
 //创建响应对象  
 DeleteIndexResponse deleteIndexResponse =   
            indices.delete(deleteIndexRequest,RequestOptions.DEFAULT);  
 //得到响应结果  
 boolean acknowledged = deleteIndexResponse.isAcknowledged();  
 System.out.println(acknowledged);  
 }

### **添加文档**

#### **3.20.6.1api**

POST /java1906/course/1  
{  
 "name":"spring cloud实战",  
 "description":"本课程主要从四个章节进行讲解： 1.微服务架构入门 2.spring cloud 基础入门 3.实战Spring Boot 4.注册中心eureka。",  
 "studymodel":"201001",  
 "price":5.6  
}

#### **3.20.6.2java client**

//添加文档  
 @Test  
 public void testAddDocument() throws IOException {  
 //创建“索引请求”对象：索引当动词  
 IndexRequest indexRequest = new IndexRequest("java1906", "course", "1");  
 indexRequest.source("{\n" +  
 " \"name\":\"spring cloud实战\",\n" +  
 " \"description\":\"本课程主要从四个章节进行讲解： 1.微服务架构入门 " +  
 "2.spring cloud 基础入门 3.实战Spring Boot 4.注册中心eureka。\",\n" +  
 " \"studymodel\":\"201001\",\n" +  
 " \"price\":5.6\n" +  
 "}", XContentType.JSON);  
 IndexResponse indexResponse =   
            restHighLevelClient.index(indexRequest,RequestOptions.DEFAULT);  
 System.out.println(indexResponse.toString());  
 }

### **批量添加文档**

#### **3.20.7.1api**

POST /\_bulk  
{"index":{"\_index":"java1906","\_type":"course"}}  
{"name":"php实战","description":"php谁都不服","studymodel":"201001","price":"5.6"}  
{"index":{"\_index":"java1906","\_type":"course"}}  
{"name":"net实战","description":"net从入门到放弃","studymodel":"201001","price":"7.6"}

#### **3.20.7.2java client**

@Test  
public void testBulkAddDocument() throws IOException {  
    BulkRequest bulkRequest = new BulkRequest();  
    bulkRequest.add(new IndexRequest("java1906", "course").source("{...}", XContentType.JSON));  
    bulkRequest.add(new IndexRequest("java1906", "course").source("{...}", XContentType.JSON));  
    BulkResponse bulkResponse = restHighLevelClient.bulk(bulkRequest,RequestOptions.DEFAULT);  
    System.out.println(bulkResponse.hasFailures());  
}

### **修改文档**

#### **3.20.8.1api**

1、完全替换

PUT /java1906/course/1  
{  
 "name":"spring cloud实战",  
 "description":"本课程主要从四个章节进行讲解： 1.微服务架构入门 2.spring cloud 基础入门 3.实战Spring Boot 4.注册中心eureka。",  
 "studymodel":"201001",  
 "price":5.5  
}

2、局部更新

PUT /java1906/1  
{  
 "price":6.66  
}

#### **3.20.8.2java client**

//更新文档  
@Test  
public void testUpdateDocument() throws IOException {  
    UpdateRequest updateRequest = new UpdateRequest("java1906", "course", "1");  
    updateRequest.doc("{\n" +  
            " \"price\":7.6\n" +  
            "}", XContentType.JSON);  
    UpdateResponse updateResponse =   
        restHighLevelClient.update(updateRequest,RequestOptions.DEFAULT);  
    DocWriteResponse.Result result = updateResponse.getResult();  
    System.out.println(result);  
}

### **删除文档**

#### **3.20.9.1api**

DELETE /java1906/coures/1

#### **3.20.9.2java client**

   //根据id删除文档  
    @Test  
    public void testDelDocument() throws IOException {  
        //删除索引请求对象  
        DeleteRequest deleteRequest = new DeleteRequest("java1906","course","1");  
        //响应对象  
        DeleteResponse deleteResponse =   
            restHighLevelClient.delete(deleteRequest,RequestOptions.DEFAULT);  
        System.out.println(deleteResponse.getResult());  
   }

## **文档搜索**

### **准备环境**

向索引库中插入以下数据：

PUT /java1906/course/1  
{  
  "name": "Bootstrap开发",  
  "description": "Bootstrap是由Twitter推出的一个前台页面开发css框架，是一个非常流行的开发框架，此框架集成了多种页面效果。此开发框架包含了大量的CSS、JS程序代码，可以帮助开发者（尤其是不擅长css页面开发的程序人员）轻松的实现一个css，不受浏览器限制的精美界面css效果。",  
  "studymodel": "201002",  
  "price":38.6,  
  "timestamp":"2018-04-25 19:11:35",  
  "pic":"group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg"  
}  
​  
PUT /java1906/course/2  
{  
  "name": "java编程基础",  
  "description": "java语言是世界第一编程语言，在软件开发领域使用人数最多。",  
  "studymodel": "201001",  
  "price":68.6,  
  "timestamp":"2018-03-25 19:11:35",  
  "pic":"group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg"  
}  
​  
PUT /java1906/course/3  
{  
  "name": "spring开发基础",  
  "description": "spring 在java领域非常流行，java程序员都在用。",  
  "studymodel": "201001",  
  "price":88.6,  
  "timestamp":"2018-02-24 19:11:35",  
  "pic":"group1/M00/00/00/wKhlQFs6RCeAY0pHAAJx5ZjNDEM428.jpg"  
}

### **简单搜索**

简单搜索就是通过url进行查询，以get方式请求ES。语法：

GET /index\_name/type\_name/doc\_idGET [/index\_name/type\_name/]\_search[?parameter\_name=parameter\_value&...]

例如：

GET /java1906/course/\_search?q=name:spring&sort=price:desc

注意： 如果查询条件复杂，很难构建搜索条件 ，生产环境中很少使用。 例如：要求搜索条件为商品名称包含手机，价格在 1000~5000之间，销量在每月 500 以上，根据价格升序排列，分页查询第二页，每页 40 条数据：?q=xxxx:xxx&range=xxx:xxx:xxx&aggs&sort&from&size

#### **3.21.2.1api**

GET /java1906/course/1

#### **3.21.2.2java client**

   //查询文档  
    @Test  
    public void getDoc() throws IOException {  
        GetRequest getRequest = new GetRequest("java1906","course","1");  
        GetResponse getResponse = restHighLevelClient.get(getRequest);  
        boolean exists = getResponse.isExists();  
        System.out.println(exists);  
 String source = getResponse.getSourceAsString();  
 System.out.println(source);  
   }

### **DSL搜索**

DSL(Domain Specific Language)是ES提出的基于json的搜索方式，在搜索时传入特定的json格式的数据来完成不同的搜索需求，DSL比URI搜索方式功能强大，在项目中建议使用DSL方式来完成搜索。语法：

​ GET /index\_name/type\_name/\_search​ {​ "commond":{ ​ "parameter\_name" : "parameter\_value"​ }​ }

#### **3.21.3.1match\_all查询**

##### **3.21.3.1.1api**

GET /java1906/course/\_search  
{  
  "query" : {   
    "match\_all" : {}  
 }  
}

##### **3.21.3.2java client**

package com.usian.test;  
@SpringBootTest  
@RunWith(SpringRunner.class)  
public class IndexSearchTest {  
​  
    @Autowired  
    private RestHighLevelClient restHighLevelClient;  
​  
    private SearchRequest searchRequest;  
    private SearchResponse searchResponse;  
​  
    @Before  
    public void initSearchRequest() {  
        // 搜索请求对象  
        searchRequest = new SearchRequest("java1906");  
        searchRequest.types("course");  
   }  
​  
    // 搜索type下的全部记录  
    @Test  
    public void testSearchAll() throws Exception {  
​  
        // 搜索源构建对象  
        SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();  
        searchSourceBuilder.query(QueryBuilders.matchAllQuery());  
​  
        // 设置搜索源  
        searchRequest.source(searchSourceBuilder);  
        // 执行搜索  
        searchResponse = restHighLevelClient.search(searchRequest,RequestOptions.DEFAULT);  
​  
   }  
​  
    @After  
    public void displayDoc() throws ParseException {  
        // 搜索匹配结果  
        SearchHits hits = searchResponse.getHits();  
        // 搜索总记录数  
        long totalHits = hits.totalHits;  
        System.out.println("共搜索到" + totalHits + "条文档");  
        // 匹配的文档  
        SearchHit[] searchHits = hits.getHits();  
        // 日期格式化对象  
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
        for (SearchHit hit : searchHits) {  
            // 文档id  
            String id = hit.getId();  
            System.out.println("id：" + id);  
            // 源文档内容  
            String source = hit.getSourceAsString();  
            System.out.println(source);  
       }  
   }  
​  
}

#### **3.21.3.2分页查询**

##### **3.21.3.2.1api**

GET /test\_index/my\_type/\_search  
{  
  "query" : { "match\_all" : {} },  
  "from" : 1, # 从第几条数据开始查询，从0开始计数  
  "size" : 2, # 查询多少数据  
  "sort" : [  
   { "order\_no" : "asc" }  
 ]  
}

##### **3.21.3.2.2java client**

//分页查询  
@Test  
public void testSearchPage() throws Exception {  
    SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();  
    searchSourceBuilder.query(QueryBuilders.matchAllQuery());  
    searchSourceBuilder.from(1);  
    searchSourceBuilder.size(5);  
    searchSourceBuilder.sort("price", SortOrder.ASC);  
​  
    // 设置搜索源  
    searchRequest.source(searchSourceBuilder);  
    // 执行搜索  
    searchResponse = restHighLevelClient.search(searchRequest,RequestOptions.DEFAULT);  
}

#### **3.21.3.3match查询**

match Query即全文检索，它的搜索方式是先将搜索字符串分词，再使用各各词条从索引中搜索。

##### **3.21.3.3.1api**

query：搜索的关键字operator：or 表示 只要有一个词在文档中出现则就符合条件，and表示每个词都在文档中出现则才符合条件。

1、基本使用：

GET /java1906/course/\_search  
{  
  "query" : {  
    "match" : {  
      "name": {  
        "query": "spring开发"  
     }  
   }  
 }  
}

2、operator：

GET /java1906/course/\_search  
{  
  "query" : {  
    "match" : {  
      "name": {  
        "query": "spring开发",  
        "operator": "and"  
     }  
   }  
 }  
}

上边的搜索的执行过程是： 1、将“spring开发”分词，分为spring、开发两个词 2、再使用spring和开发两个词去匹配索引中搜索。 3、由于设置了operator为and，必须匹配两个词成功时才返回该文档。

##### **3.21.3.3.2 java client**

@Test  
public void testMatchQuery() throws Exception {  
    SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();  
    searchSourceBuilder.query(QueryBuilders.matchQuery("name", "spring开  
                                                       发").operator(Operator.AND));  
   
    // 设置搜索源  
    searchRequest.source(searchSourceBuilder);  
    // 执行搜索  
    searchResponse = restHighLevelClient.search(searchRequest,RequestOptions.DEFAULT);  
 }

#### **3.21.3.4multi\_match查询**

matchQuery是在一个field中去匹配，multiQuery是拿关键字去多个Field中匹配。

##### **3.21.3.4.1api**

1、基本使用例子：关键字 “开发”去匹配name 和description字段

GET /java1906/course/\_search  
{  
  "query": {  
    "multi\_match": {  
      "query": "开发",  
      "fields": ["name","description"]  
   }  
 }  
}

注意：此搜索操作适合构建复杂查询条件，生产环境常用。

##### **3.21.3.4.2java client**

@Test  
public void testMultiMatchQuery() throws Exception {  
    SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();  
    searchSourceBuilder.query(QueryBuilders.multiMatchQuery("开发","name","description"));  
   
    // 设置搜索源  
    searchRequest.source(searchSourceBuilder);  
    // 执行搜索  
    searchResponse = restHighLevelClient.search(searchRequest,RequestOptions.DEFAULT);  
}

#### **3.21.3.5bool查询**

布尔查询对应于Lucene的BooleanQuery查询，实现将多个查询组合起来。参数： must：表示必须，多个查询条件必须都满足。（通常使用must） should：表示或者，多个查询条件只要有一个满足即可。 must\_not：表示非。

##### **3.21.3.5.1api**

例如：查询name和description都包括“开发”的文档

GET /java1906/course/\_search  
{  
  "query": {  
    "bool": {  
      "must": [  
       {  
          "match": {  
            "name": "开发"  
         }  
       },{  
          "match": {  
            "description": "开发"  
         }  
       }  
     ]  
   }  
 }  
}

##### **3.21.3.5.2java client**

@Test  
public void testBooleanQuery() throws IOException {  
    SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();  
    //bool  
    BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();  
    //must  
    boolQueryBuilder.must(QueryBuilders.matchQuery("name", "开发"));  
    boolQueryBuilder.must(QueryBuilders.matchQuery("description","开发"));  
    searchSourceBuilder.query(boolQueryBuilder);  
    searchRequest.source(searchSourceBuilder);  
    searchResponse = restHighLevelClient.search(searchRequest,RequestOptions.DEFAULT);  
}

#### **3.21.3.6filter查询**

过滤查询。此操作实际上就是 query DSL 的补充语法。过滤的时候，不进行任何的匹配分数计算，相对于 query 来说，filter 相对效率较高。Query 要计算搜索匹配相关度分数。Query更加适合复杂的条件搜索。

##### **3.21.3.6.1api**

如：使用bool查询，搜索 name中包含 "开发"的数据，且price在 10~100 之间1、不使用 filter， name和price需要计算相关度分数：

GET /java1906/course/\_search  
{  
  "query": {  
     "bool" : {  
        "must":[  
           {  
               "match": {  
                 "name": "开发"  
               }  
           },  
           {  
              "range": {# 范围， 字段的数据必须满足某范围才有结果。  
                "price": {  
                  "gte": 1, # 比较符号 lt gt lte gte  
                  "lte": 100  
               }  
             }  
           }  
       ]  
     }  
 }  
}

2、使用 filter， price不需要计算相关度分数：

GET /java1906/course/\_search  
{  
  "query": {  
    "bool": {  
      "must": [  
       {  
          "match": {  
            "name": "开发"  
         }  
       }  
     ],  
      "filter": {# 过滤，在已有的搜索结果中进行过滤，满足条件的返回。  
        "range": {  
          "price": {  
            "gte": 1,  
            "lte": 100  
         }  
       }  
     }  
   }  
 }  
}

##### **3.21.3.6.2java client**

@Test  
public void testFilterQuery() throws IOException {  
    SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();  
    BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery();  
    boolQueryBuilder.must(QueryBuilders.matchQuery("name","开发"));  
    boolQueryBuilder.filter(QueryBuilders.rangeQuery("price").gte(10).lte(100))  
    searchSourceBuilder.query(boolQueryBuilder);  
    searchRequest.source(searchSourceBuilder);  
    searchResponse = restHighLevelClient.search(searchRequest,RequestOptions.DEFAULT);  
}

#### **3.21.3.7highlight查询**

高亮显示：高亮不是搜索条件，是显示逻辑，在搜索的时候，经常需要对条件实现高亮显示。

##### **3.21.3.7.1api**

例如：

GET /java1906/course/\_search  
{  
  "query": {  
    "multi\_match": {  
      "query": "开发",  
      "fields": ["name","description"]  
   }  
 },  
  "highlight": {  
    "pre\_tags": ["<red>"],   
    "post\_tags": ["</red>"],   
    "fields": {  
      "name": {}  
   }  
 }  
}

##### **3.21.3.7.2java clent**

1、查询：

 @Test  
  public void testHighLightQuery() throws Exception {  
      SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();  
      searchSourceBuilder.query(QueryBuilders.matchQuery("name", "spring"));  
      //设置高亮  
      HighlightBuilder highlightBuilder = new HighlightBuilder();  
      highlightBuilder.preTags("<red>");  
      highlightBuilder.postTags("</red>");  
      highlightBuilder.fields().add(new HighlightBuilder.Field("name"));  
      searchSourceBuilder.highlighter(highlightBuilder);  
​  
      searchRequest.source(searchSourceBuilder);  
      searchResponse = restHighLevelClient.search(searchRequest,RequestOptions.DEFAULT);  
}

2、遍历：

@After  
public void displayDoc() {  
    SearchHits searchHits = searchResponse.getHits();  
    long totalHits = searchHits.getTotalHits();  
    System.out.println("共搜索到" + totalHits + "条文档");  
​  
    SearchHit[] hits = searchHits.getHits();  
    for (int i = 0; i < hits.length; i++) {  
        SearchHit hit = hits[i];  
        String id = hit.getId();  
        System.out.println("id：" + id);  
        String source = hit.getSourceAsString();  
        System.out.println(source);  
​  
        Map<String, HighlightField> highlightFields = hit.getHighlightFields();  
        if (highlightFields != null) {  
            HighlightField highlightField = highlightFields.get("name");  
            Text[] fragments = highlightField.getFragments();  
            System.out.println("高亮字段：" + fragments[0].toString());  
       }  
   }  
​  
}

## 课堂笔记

|  |
| --- |
| -------------------------------ElasticSearch--------------------------  一、ElasticSearch介绍  1、什么是elasticsearch?  elasticsearch是基于lucene的全文检索服务器，对外提供restful接口  2、elasticsearch原理  正排索引：查字典时从第一页开始找，直到找到关键字为止（CTRL+F）  倒排索引：查字典时通过目录查找  逻辑结构：一个倒排索引表，由三部分组成  document  term  term----关联----document  二、ES安装  a、安装  1、设置虚拟机内存>1.5G  2、创建用户  3、安装  解压即安装  配置elasticsearch.yml  4、解决内核问题  5、解决文件创建权限问题  6、决绝线程开启限制问题  7、解决虚拟机内存问题  b、启动和关闭  启动：  ./bin/elasticsearch  ./elasticsearch -d  关闭：  kill -9 pid    三、ES快速入门  1、index管理  a、创建index  PUT /java1906  {  "settings"{  "number\_of\_shards":2,  "number\_of\_replicas":0 #备份分配不允许和主分片在同一个节点上  }  }  b、修改index  PUT /java1906/\_settings  {  "number\_of\_replicas":1 #主分片不能修改【hash(doc\_id)%number\_of\_shards=shard】  }  c、删除index  DELETE /java1906  2、mapping管理  a、关键mapping  POST /java1906/course/\_mapping  {  "properties":{  "name":{  "type":"text"  }  }  }  b、查询mapping  GET /java1906/couse/\_mapping  3、doucument管理  a、创建doucment  POST /java1907/couse/1  {  "name":"php从入门到放弃"  }  POST /java1907/couse  {  "name":"php从入门到放弃"  }  PUT /java1907/couse/1  {  "name":"php从入门到放弃"  }  b、查询doucument  GET /java1906/couse/1  c、删除document  DELETE /java1906/couse/1  四、IK分词器  1、安装  解压到plugs目录下，并重命名为ik  2、自定义词库  IkAnalyzer.cfg.xml：配置扩展词典和停用词典  main.dic：扩展词典  stopwords.dic：停用词典  3、两种分词模式  ik\_smart：粗粒度拆分  ik\_max\_word：细粒度拆分 |

# 商品搜索

## 4.1对应的业务域：

​ 1、商品Id

​ 2、商品标题

​ 3、商品卖点

​ 4、商品价格

​ 5、商品图片

​ 6、分类名称

​ 7、商品描述

需要从tb\_item, tb\_item\_cat, tb\_item\_desc表中查询商品数据

Sql：

SELECT

a.id,

a.title item\_title,

a.sell\_point item\_sell\_point,

a.price item\_price,

a.image item\_image,

b.name item\_category\_name,

c.item\_desc item\_desc

FROM

tb\_item a

JOIN tb\_item\_cat b ON a.cid = b.id

JOIN tb\_item\_desc c ON a.id = c.item\_id

WHERE

1. status = 1

业务逻辑：usian\_search\_service中实现

### **4.2对应业务域**

POST /usian/item/\_mapping

{

"\_source": {

"excludes": [

"item\_desc"

]

},

"properties": {

"item\_title": {

"type": "text",

"analyzer": "ik\_max\_word",

"search\_analyzer": "ik\_smart"

},

"item\_sell\_point": {

"type": "text",

"analyzer": "ik\_max\_word",

"search\_analyzer": "ik\_smart"

},

"item\_price": {

"type": "float"

},

"item\_image": {

"type": "text",

"index": false

},

"item\_category\_name": {

"type": "text",

"analyzer": "ik\_max\_word",

"search\_analyzer": "ik\_smart"

},

"item\_desc": {

"type": "text",

"analyzer": "ik\_max\_word",

"search\_analyzer": "ik\_smart"

}

}

}

## 4.3**创建工程**

### **4.3.1创建usian\_search\_service**

#### **pom.xml**

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>com.usian</groupId>

<artifactId>common\_mapper</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>com.usian</groupId>

<artifactId>common\_utils</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.elasticsearch</groupId>

<artifactId>elasticsearch</artifactId>

</dependency>

#### **application.yml**

spring:

application:

name: usian-search-service

datasource:

driver-class-name: com.mysql.jdbc.Driver

url: jdbc:mysql://127.0.0.1:3306/usian?characterEncoding=UTF-8

username: root

password: 1111

type: com.alibaba.druid.pool.DruidDataSource

server:

port: 8095

eureka:

client:

service-url:

defaultZone: <http://127.0.0.1:8761/eureka/>

#### **logback.xml**

#### **创建启动类**

@SpringBootApplication

@EnableDiscoveryClient

@MapperScan("com.usian.mapper")

public class SearchServiceApp {

public static void main(String[] args) {

SpringApplication.run(SearchServiceApp.class, args);

}

}

### **4.3.2创建usian\_search\_feign**

**pom.xml**

<dependencies>

<!--Spring Cloud OpenFeign Starter -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

<!--utils -->

<dependency>

<groupId>com.usian</groupId>

<artifactId>common\_utils</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>com.usian</groupId>

<artifactId>common\_pojo</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

</dependencies>

### **4.3.3创建usian\_search\_web**

#### **pom.xml**

<dependencies>

<dependency>

<groupId>com.usian</groupId>

<artifactId>usian\_search\_feign</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

</dependencies>

#### **application.yml**

spring:

application:

name: usian-search-web

server:

port: 8096

eureka:

client:

service-url:

defaultZone: http://127.0.0.1:8761/eureka/

#全局配置

#请求连接的超时时间

ribbon:

ConnectTimeout: 60000

#请求处理的超时时间

ReadTimeout: 60000

#### **logback.xml**

#### **启动类**

@SpringBootApplication

@EnableFeignClients

@EnableDiscoveryClient

public class SearchWebApp {

public static void main(String[] args) {

SpringApplication.run(SearchWebApp.class, args);

}

}

## 4.4导入商品数据到索引库

### 4.4.1usian-manage-web

#### **修改src\api\base.js**

baseSearchUrl:"/search\_api"

importAll:"/frontend/searchItem/importAll" // 一键导入商品数据到索引库

**修改src\api\index.js**

/\*\*

\* 一键导入商品数据到索引库

\*/

getImportAll(){

return axios.post(base.baseSearchUrl + base.importAll,null,{timeout:50000});

}

**修改src\pages\Product\ProductList\index.vue**

<div>

<el-button type="primary" @click="addProduct">添加商品</el-button>

<el-button type="primary" @click="importAll">一键导入商品</el-button>

</div>

/\*\*

\* 一键导入商品数据到索引库

\*/

importAll() {

this.$api.getImportAll().then(res => {

if (res.data.status == 200) {

this.$message({

type: "success",

message: "导入成功!"

});

} else {

this.$message({

type: "error",

message: "导入失败!"

});

}

});

}

**修改vue.config.js**

'/search\_api': {

target: 'http://127.0.0.1:8096',

// target: 'http://121.42.14.194:9021',

pathRewrite: {

'^/search\_api': ''

},

changeOrigin: true

}

### **4.4.2 common\_pojo**

package com.usian.pojo;

import java.io.Serializable;

public class SearchItem implements Serializable {

private String id;

private String item\_title;

private String item\_sell\_point;

private String item\_price;

private String item\_image;

private String item\_category\_name;

private String item\_desc;｝

### **4.4.3 common\_mapper**

#### **SearchItemMapper**

package com.usian.mapper;

import com.usian.pojo.SearchItem;

import java.util.List;

public interface SearchItemMapper {

List<SearchItem> getItemList();

}

#### **SearchItemMapper.xml**

<mapper namespace="com.usian.mapper.SearchItemMapper" >

<select id="getItemList" resultType="com.usian.pojo.SearchItem">

SELECT

a.id,

a.title item\_title,

a.sell\_point item\_sell\_point,

a.price item\_price,

a.image item\_image,

b.name item\_category\_name,

c.item\_desc item\_desc

FROM

tb\_item a

JOIN tb\_item\_cat b ON a.cid = b.id

JOIN tb\_item\_desc c ON a.id = c.item\_id

WHERE

a.status = 1

</select>

### **4.4.3 usian\_search\_service**

#### **application.yml**

spring:

data:

elasticsearch:

cluster-nodes: 192.168.233.134:9200,192.168.233.135:9200

#索引名称

ES\_INDEX\_NAME: usian

#类型名称

ES\_TYPE\_NAME: item

#### **config**

@Configuration

public class ElasticsearchConfig extends ElasticsearchProperties{

@Bean

public RestHighLevelClient getRestHighLevelClient() {

String[] hosts = getClusterNodes().split(",");

HttpHost[] httpHosts = new HttpHost[hosts.length];

for (int i = 0; i < httpHosts.length; i++) {

String h = hosts[i];

httpHosts[i] = new HttpHost(h.split(":")[0],

Integer.parseInt(h.split(":")[1]));

}

return new RestHighLevelClient(RestClient.builder(httpHosts));

}

}

#### **Service**

@Service

@Transactional

public class SearchItemServiceImpl implements SearchItemService{

@Autowired

private SearchItemMapper searchItemMapper;

@Autowired

private RestHighLevelClient restHighLevelClient;

@Value("${ES\_INDEX\_NAME}")

private String ES\_INDEX\_NAME;

@Value("${ES\_TYPE\_NAME}")

private String ES\_TYPE\_NAME;

@Override

public Boolean importAll() {

try {

if(!isExistsIndex()){

createIndex();

}

int page=1;

while (true){

/\*\*分页每次导入一千条\*/

PageHelper.startPage(page,1000);

//1、查询mysql中的商品信息

List<SearchItem> esDocumentList = searchItemMapper.getItemList();

if(esDocumentList==null || esDocumentList.size()==0){

break;

}

BulkRequest bulkRequest = new BulkRequest();

for (int i = 0; i < esDocumentList.size(); i++) {

SearchItem searchItem = esDocumentList.get(i);

//2、把商品信息添加到es中

bulkRequest.add(new IndexRequest(ES\_INDEX\_NAME, ES\_TYPE\_NAME).

source(JsonUtils.objectToJson(searchItem), XContentType.JSON));

}

restHighLevelClient.bulk(bulkRequest, RequestOptions.DEFAULT);

page++;

}

return true;

}catch (Exception e){

e.printStackTrace();

return false;

}

}

\* 索引库是否存在

\* @return

\* @throws IOException

private boolean isExistsIndex() throws IOException {

GetIndexRequest request = new GetIndexRequest();

request.indices(ES\_INDEX\_NAME);

return restHighLevelClient.indices().exists(request, RequestOptions.DEFAULT);

}

/\*\*

\* 创建索引库

\* @return

\* @throws IOException

private boolean createIndex() throws IOException {

//创建索引请求对象，并设置索引名称

CreateIndexRequest createIndexRequest = new CreateIndexRequest(ES\_INDEX\_NAME);

//设置索引参数

createIndexRequest.settings(Settings.builder().put("number\_of\_shards",2)

.put("number\_of\_replicas",1));

createIndexRequest.mapping(ES\_TYPE\_NAME, " api中对应域的内容复制到此处", XContentType.JSON);

//创建索引操作客户端

IndicesClient indices = restHighLevelClient.indices();

//创建响应对象

CreateIndexResponse createIndexResponse =

indices.create(createIndexRequest,RequestOptions.DEFAULT);

//得到响应结果

return createIndexResponse.isAcknowledged();

}

}

#### **controller**

@RestController

@RequestMapping("/service/searchItem")

public class SearchItemController {

@Autowired

private SearchItemService searchItemService;

@RequestMapping("/importAll")

public boolean importAll(){

return searchItemService.importAll();

}

}

### 4.4.4 usian\_search\_feign

#### **feign**

@FeignClient("usian-search-service")

public interface SearchItemFeign {

@RequestMapping("/importAll")

public Boolean importAll();

}

### 4.4.5 **usian\_search\_web**

#### **controller**

@RestController

@RequestMapping("/frontend/searchItem")

public class SearchItemController {

@Autowired

private SearchItemFeign searchItemFeign;

@RequestMapping("/importAll")

public boolean importAll(){

return searchItemFeign.importAll();

}

}

## **4.5商品搜索**

### **4.5.1 usian\_search\_service**

#### **service**

\* 分页查询名字、类别、描述、卖点包含q的商品

\* @param q

\* @param page

\* @param pageSize

\* @return

@Override

public List<SearchItem> selectByQ(String q, Long page, Integer pageSize) {

try{

SearchRequest searchRequest = new SearchRequest(ES\_INDEX\_NAME);

searchRequest.types(ES\_TYPE\_NAME);

SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();

//1、查询名字、描述、卖点、类别包括“q”的商品

searchSourceBuilder.query(QueryBuilders.multiMatchQuery(q,new String[]{

"item\_title","item\_desc","item\_sell\_point","item\_category\_name"}));

//2、分页

\* 1 0 20--->(p-1)\*pageSize

\* 2 20 20--->(2-1)\*20

\* 3 40 20--->(3-1)\*20

\*/

Long from = (page - 1) \* pageSize;

searchSourceBuilder.from(from.intValue());

searchSourceBuilder.size(pageSize);

//3、高亮

HighlightBuilder highlightBuilder = new HighlightBuilder();

highlightBuilder.preTags("<font color='red'>");

highlightBuilder.postTags("</font>");

highlightBuilder.field("item\_title");

searchSourceBuilder.highlighter(highlightBuilder);

searchRequest.source(searchSourceBuilder);

SearchResponse response = restHighLevelClient.search(

searchRequest, RequestOptions.DEFAULT);

SearchHit[] hits = response.getHits().getHits();

//4、返回查询结果

List<SearchItem> searchItemList = new ArrayList<>();

for (int i=0; i<hits.length; i++){

SearchHit hit = hits[i];

SearchItem searchItem = JsonUtils.jsonToPojo(hit.getSourceAsString(),

SearchItem.class);

Map<String, HighlightField> highlightFields = hit.getHighlightFields();

if(highlightFields!=null && highlightFields.size()>0) {

searchItem.setItem\_title(highlightFields.get("item\_title").

getFragments()[0].toString());

}

searchItemList.add(searchItem);

}

return searchItemList;

}catch (Exception e){

e.printStackTrace();

}

return null;

}

#### **controller**

@RequestMapping("/list")

public List<SearchItem> selectByq(String q, Long page, Integer pageSize)

throws IOException {

return searchItemService.selectByq(q,page,pageSize);

}

### 4.5.2 **usian\_search\_feign**

#### **feign**

@RequestMapping("/service/searchItem/list")

List<SearchItem> selectByq(@RequestParam String q, @RequestParam Long page,

@RequestParam Integer pageSize);

### 4.5.3 **usian\_search\_web**

#### **controller**

@RequestMapping("/list")

public List<SearchItem> selectByQ(String q, @RequestParam(defaultValue = "1")

Long page, @RequestParam(defaultValue = "20") Integer pageSize){

return searchItemFeign.selectByq(q,page,pageSize);

}

# 索引库同步

## 后台新增商品后需要把商品同步到索引库

**方案一**：业务逻辑在usian\_item\_sevice中实现，添加商品的业务逻辑中，添加一个同步索引库的业务逻辑。

缺点：

​ 1、业务逻辑耦合度高（既维护商品又维护索引库）

**方案二**：业务逻辑在usian\_search\_service中实现，usian\_item\_sevice先添加商品，再调用usian\_search\_service服务同步索引库。

缺点：

​ 1、系统间耦合性太强（如果将来其他服务接入，usian\_item\_sevice还需要修改代码）

**方案三**：使用消息中间件

| **类型** | **特点** |
| --- | --- |
| 基本消息模型 | 发送者------>queue----->消费者 |
| work消息模型 | 发送者------>queue----->多个消费者 |
| 广播-fanout消息模型 | 发送者----->exchange---->多个queue--->多个消费者 |
| 定向-direct消息模式 | 发送者----->exchange---routing key-->多个queue--->多个消费者 |
| 通配符-topic消息模型 | 发送者----->exchange---星.routing key.星-->多个queue---->多个消费者 |

## **索引库同步**

### **common\_mapper**

#### **SearchItemMapper**

### SearchItem getItemById(Long itemId);

#### **SearchItemMapper.xml**

<select id="getItemById" parameterType="long" resultType="com.usian.pojo.SearchItem">

SELECT

a.id,

a.title item\_title,

a.sell\_point item\_sell\_point,

a.price item\_price,

a.image item\_image,

b.name item\_category\_name,

c.item\_desc item\_desc

FROM

tb\_item a

JOIN tb\_item\_cat b ON a.cid = b.id

JOIN tb\_item\_desc c ON a.id = c.item\_id

WHERE a.status = 1

AND a.id=#{itemId}

</select>

### **usian\_search\_service**

#### **pom.xml**

### <dependency>

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

### <artifactId>spring-boot-starter-amqp</artifactId>

### </dependency>

#### **application.yml**

spring:

rabbitmq:

host: 192.168.157.132

username: admin

password: 1111

virtual-host: /

#### **service**

@Override

public int insertDocument(String itemId) throws IOException {

// 1、根据商品id查询商品信息。

SearchItem searchItem = searchItemMapper.getItemById(Long.valueOf(itemId));

//2、添加商品到索引库

IndexRequest indexRequest = new IndexRequest(ES\_INDEX\_NAME, ES\_TYPE\_NAME);

indexRequest.source(JsonUtils.objectToJson(searchItem), XContentType.JSON);

IndexResponse indexResponse =

restHighLevelClient.index(indexRequest,RequestOptions.DEFAULT);

return indexResponse.getShardInfo().getFailed();

}

#### **listerner**

@Component

public class SearchMQListener {

@Autowired

private RestHighLevelClient restHighLevelClient;

@Autowired

private SearchItemService searchItemService;

\* 监听者接收消息三要素：

\* 1、queue

\* 2、exchange

\* 3、routing key

@RabbitListener(bindings = @QueueBinding(

value = @Queue(value="search\_queue",durable = "true"),

exchange = @Exchange(value="item\_exchage",type= ExchangeTypes.TOPIC),

key= {"item.\*"}

))

public void listen(String msg) throws Exception {

System.out.println("接收到消息：" + msg);

int result = searchItemService.insertDocument(msg);

if(result>0){

throw new RuntimeException("同步失败");

}

### **usian\_item\_service**

#### **pom.xml**

<dependency>

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

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

</dependency>

#### **application.yml**

spring:

rabbitmq:

host: 192.168.157.132

username: admin

password: 1111

virtual-host: /

#### **service**

@Override

public Integer insertTbItem(TbItem tbItem, String desc, String itemParams) {

。。。。。。。。。。。

//添加商品发布消息到mq

amqpTemplate.convertAndSend("item\_exchage","item.add", itemId);

return tbItemNum + tbitemDescNum + itemParamItmeNum;

}

### **测试**

#### **后台添加新商品**

#### **前台搜索新商品**