写在前面的话

自从有了儿子，下班到家几乎没有自己的独立时间。眼看这篇教程就要完蛋，有点不舍得。现将残缺版放出，供大家阅读。欢迎大家反馈问题至 donlianli@126.com。
我的微博：http://weibo.com/donlianli
我的博客：http://donlianli.iteye.com/

第一部分：搜索的背景

浅谈搜索

让我们先简单了解一下搜索业务需求。不论你在哪个行业，这些搜索业务都是通用的。如果你正在被这些业务需求所困扰，你就算找对地方了。

1、用户对检索结果不满意，希望查找的内容，没有排在前面。
2、用户搜索“苹果”，希望看到的是苹果公司的产品，而不是真正的水果。
3、用户希望你的软件能够容错，写个错别字，软件能够智能提示出来。
4、用户希望搜索足够的快，就几百万的数据，别让我等得睡着了。
。。。。。。

估计大家都被百度 google 这类的软件给宠坏了，这分明就是要复制一个类似百度的产品吗？如果你还没有遇到以上这些业务需求，可能你还不需要了解 elasticsearch。有人把 elasticsearch 当成 mysql 一样来使，我觉得这样是违背了软件的设计初衷的，就好像你用拿一辆轿车去耕地一样，是一种误用。

根据以上需求我总结出，一个完善的搜索引擎，应该至少提供以下功能（或者说能够解决下面的问题）。当然，想 BAIDU,GOOGLE 这些公司，考虑的要比我这要多的更多了。
搜索业务总结
1）全角转半角（全角数字转半角数字）。
2）大写转小写（英文字母）
3）错别字纠正
4）自动补全或精简（比如输入北京市，自动缩减成北京）
5）繁体转简体
6）拼音转汉字
7）同义词转换（某些领域，同义词的情况和常见，比如医学这方面）
8）加权处理（比如地名加权）
9）完善的领域词库
10）相关度排序
但是，es 默认仅包含了前两项功能，剩下的需求，如果你需要，你可能需要在 es 的基础上做一些二次开发。甚至，作为一个非英语国家，你还需要选择一个分词插件或者自定义一个分词插件，来满足某些领域的词汇搜索。但是 ES 比较为你实现搜索功能提供了一个基础，这就已经很不容易了。

Lucene

在介绍 ES 之前，我们先介绍一下 Lucene。因为 ES 是建立在 Lucene 的基础上的。ES 对搜索功能的支持在很大程度上取决于 Lucene。
我们有必要澄清 Lucene 和 ES 的功能划分。Lucene 是一个全文检索库(lib)，但并不是像 es 一样可以独立运行。单独使用 Lucene 库也可以构建一个本地的全文检索库。也有很多公司直接使用 Lucene 来构建搜索引擎。也就是说，没有 es，我们光使用 lucene 库，也是可以做到全文检索的。Es 中的分词，搜索，排序等功能，就是直接调用了 Lucene 的接口而已。
另外，es 的开发者中有好几个人也是 Lucene 的 Committer。

Elasticsearch

Es 官方宣传的主要特点有：
1、 实时更新
2、 实时分析统计
3、 分布式
4、 高可用
5、 多租户（实际上就是可以包含多个索引）
6、 全文检索
7、 。。。。。
其实在我接触搜索引擎之前，我对以上的特点也没有觉得有多大的好处。比如，实时更新，是说被索引的数据能够马上被搜索到。按照我们的常规思维，存到数据库的数据当然应该被立即检索到，这很正常。后来才知道，实时更新这个特性是针对 Lucene 做的改进，在 lucene 中，被索引的数据不能马上被搜索出来。总之，所有的这些特性，可能都是针对 lucene，或者跟 solr 对比有这些优点。
个人觉得 es 有一个很大的好处就是部署简单，不需要 web 容器，不需要 war 包，不需要你去系统配置（但是 Java 的环境变量还是要配置的）。创建集群也非常简单，同一个命令运行多次就创建了一个集群。

第二部分：elasticsearch 的基本应用

官方的教程，都是基于 rest 和 json 格式的 api 说明，其对应的 Java api 却很少有例子，这就给我们这些 Java 程序员一个很大的障碍。Es 本身是用 Java 写成的，所以其 rest api 都有对应的 Java api，其实仔细观察一下 Java 的 api，了解了 es 的结构，我们就能猜出 rest 对应的 Java api 是哪个。我们先从入门级的增删改查开始练习（关于如何启动 es，我就不介绍了，本文所有代码都是基于 elasticsearch1.2.1）

增删改查（CRUD）

为了测试，我们假设存在一个叫 donlianli 的 es 集群。现在，我们需要将我们的数据存入到 es 中。Es 存储时，需要客户端指定需要存入的索引(index)和类型(type)，这两个分别对应传

统数据库的 schema 和 table，我们假设 index 叫 esindex,和 estype,现在我们需要将我们的数据存入这个 donlianli 集群中的 esindex 和 estype 中。

数据模型

我们现在假设需要索引的是一件商品，简单定义如下（假设是数据库表，java 中以骆驼命名法）。

名称	标识	类型
商品 id	prod_id	long
商品名称	prod_name	text
描述	prod_desc	text
分类	cat_id	long

工具类

这个就是一个连接 elasticsearch 的工具类，把要连接的集群的 ip，端口和集群名称都独立出来。代码如下：

```
import org.elasticsearch.client.Client;
import org.elasticsearch.client.transport.TransportClient;
import org.elasticsearch.common.settings.ImmutableSettings;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.common.transport.InetSocketTransportAddress;

public class ESUtils {
	public static final String INDEX_NAME="esindex";
	public static String getIndexName(){
		return INDEX_NAME;
	}
	public static final String TYPE_NAME="estype";
	public static String getTypeName(){
		return TYPE_NAME;
	}
	public static Client getClient(){
		Settings settings = ImmutableSettings.settingsBuilder()
				//指定集群名称
				.put("cluster.name", "donlianli")
				//探测集群中机器状态
				.put("client.transport.sniff", true).build();
		/*
		 * 创建客户端，所有的操作都由客户端开始，这个就好像是 JDBC 的 Connection 对象
		 * 用完记得要关闭
```

```
            */
            Client client = new TransportClient(settings)
            .addTransportAddress(new InetSocketTransportAddress("192.168.1.106", 9300));
            return client;
      }
      public static void closeClient(Client esClient){
            if(esClient !=null){
                  esClient.close();
            }
      }
}
```

增加

在 es 中，没有增加这个概念，所谓增加，就是将一条记录存储到 es 里面。Es 里面将这个过程叫做索引（index），代码大概如下：

```
Client client = ESUtils.getClient();
IndexResponse indexResponse =
client.prepareIndex().setIndex(ESUtils.getIndexName())
                .setType(ESUtils.getTypeName())
        .setSource("{\"prodId\":1,\"prodName\":\"ipad5\",\"prodDesc\":\"
比你想的更强大\",\"catId\":1}")
                .setId("1")
                .execute()
                .actionGet();
System.out.println("添加成功,isCreated="+indexResponse.isCreated());
ESUtils.closeClient(client);
```

可以执行代码，输出结果为“添加成功，isCreated=true”。

这索引的内容就是一个 json 串，目前 es 只支持 json，当你索引一个对象的时候，其实 es 帮你做了一个转换，转换成 json 以后，再存储到 es 里面。索引以后，怎么知道实际上已经存到 es 里面了呢？如果是在 linux 平台，可以使用下面的命令查看

```
curl ‘http://localhost:9200/esindex/estype/1’
```

请注意上面的端口，我们使用 Java 程序连接使用的 9300 端口，使用命令行使用的 9200 端口。

查询

查询很简单，代码如下：

```
GetResponse getResponse =
client.prepareGet().setIndex(ESUtils.getIndexName())
                .setType(ESUtils.getTypeName())
                .setId("1")
```

```
                .execute()
                .actionGet();
System.out.println("get="+getResponse.getSourceAsString());
```

首先将 prepareIndex 换成 prepareGet，然后将 IndexResponse 改成 GetResponse，再把结果输出即可。

删除

```
DeleteResponse delResponse =
client.prepareDelete().setIndex(ESUtils.getIndexName())
                .setType(ESUtils.getTypeName())
                .setId("1")
                .execute()
                .actionGet();
System.out.println("del is found="+delResponse.isFound());
```

修改

修改和增加的代码相类似，只是，修改的时候不能够使用 setSource 的方式，而是使用 setDoc 方法，每次被更新，文档的版本号,即 version 都会自动加 1。你可以自己验证。

```
GetResponse getResponse =
client.prepareGet().setIndex(ESUtils.getIndexName())
                .setType(ESUtils.getTypeName())
                .setId("1")
                .execute()
                .actionGet();
System.out.println("berfore update version="+getResponse.getVersion());

UpdateResponse updateResponse =
client.prepareUpdate().setIndex(ESUtils.getIndexName())
                .setType(ESUtils.getTypeName())
    .setDoc("{\"prodId\":1,\"prodName\":\"ipad5\",\"prodDesc\":\"比你想的
更强大噢耶\",\"catId\":1}")
                .setId("1")
                .execute()
                .actionGet();

System.out.println("更新成功，isCreated="+updateResponse.isCreated());
        getResponse =
client.prepareGet().setIndex(ESUtils.getIndexName())
                .setType(ESUtils.getTypeName())
                .setId("1")
```

```
                .execute()
                .actionGet();
        System.out.println("get version="+getResponse.getVersion());
```

Search

我们对 CRUD 的数据进行搜索，代码：

```
QueryBuilder query = QueryBuilders.queryString("ipad5");
        SearchResponse response = client.prepareSearch(ESUtils.INDEX_NAME)
                //设置查询条件,
                .setQuery(query)
                .setFrom(0).setSize(60)
                .execute()
                .actionGet();
        /**
         * SearchHits是SearchHit的复数形式，表示这个是一个列表
         */
        SearchHits shs = response.getHits();
        System.out.println("总共有："+shs.hits().length);
        for(SearchHit hit : shs){
            System.out.println(hit.getSourceAsString());
        }
        client.close();
```

输出结果大概如下：

```
总共有：2
{"prodId":1,"prodName":"ipad5","prodDesc":"比你想的更强大","catId":1}
{"prodId":1,"prodName":"ipad5","prodDesc":"比你想的更强大","catId":1}
```

这是因为我索引里面存储了两条相同的记录。但是如果你把 ipad5 修改为 ipad，还能搜索到结果么？答案是不能，为什么呢？上面这个搜索既没有指定索引的 type，也没有指定搜索的 field，那么这个搜索怎么也能搜索出数据呢？这个我们稍后再说。

基础概念

我们先暂停一下，了解一下 es 的一些基础概念。之前我们也简单描述过，假设拿 mysql 和 es 对比的话，可以用下面的关系对比。

MySQL	elasticsearch
database	index
table	type
row	document
field	field

shard 是创建索引时指定的一个分片。分片越多，索引的速度理论上越快。

node 就是一个 es 实例,启动一个 java 进程就启动了一个 node。

Index 是逻辑概念，可以有多个 shard。type 是比 index 更小的一个逻辑单元。

Bulk

Bulk，英文单词是散装，有多并且大的意思。在这里就是批量组装查询或者索引数据的意思。通常，如果一个程序有批量接口，则性能肯定比单个数据处理的要快。在 es 中，索引与搜索是一个互相矛盾和对立的。如果你希望查询的速度快，则应该让索引尽量少的改变，如果你希望索引的速度快些，则查询的性能必将降低。ES 索引文档，通常是比较耗费性能的，而且会影响到查询的速度。但如果一次索引多条数据，则 es 会将这一批数据一起刷入磁盘，提高性能。Bulk 就是这样一个批量接口，可以将很多待索引的数据一次发生给 ES，然后 ES 将这些数据批量的刷入内存和磁盘，从而提高性能。另外，ES 不仅有批量索引的接口，还有批量查询的接口。
我们现在假设有一个 User 对象，需要批量索引到 es 中。代码如下：

```
Client client = ESUtils.getClient();
        BulkRequestBuilder bulkRequest = client.prepareBulk();
        User user = new User();
        for(int i=1;i<40001;i++){
            user.setName(RandomData.randomName("user_"));
            SecureRandom random = new SecureRandom();
            long l = Math.abs(random.nextLong());
            user.setWeight(l);
            user.setMarried(l%1==0?true:false);
            user.setAge(l%2==0?28:82);
            IndexRequestBuilder ir =
client.prepareIndex(ESUtils.INDEX_NAME,
                    ESUtils.USER_TYPE,
String.valueOf(i)).setSource(Utils.toJson(user));
            bulkRequest.add(ir);
        }
        long beginTime = System.currentTimeMillis();
        BulkResponse bulkResponse = bulkRequest.execute().actionGet();
        long useTime = System.currentTimeMillis() - beginTime;
        //1406ms
        System.out.println("useTime:" + useTime);
        if (bulkResponse.hasFailures()) {
            // process failures by iterating through each bulk response item
        }
```

User 对象就是一个 POJO 对象，其中里面的 Utils.toJson 是将 java 的 POJO 转换为 JSON 字符串。代码如下：

```java
public class User {
    public int age;
    public String name;
    public long weight;
    public boolean married;
    User(){
        this.age=11;
        this.name = "default";
        this.weight = 1000l;
        this.married = false;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public long getWeight() {
        return weight;
    }
    public void setWeight(long weight) {
        this.weight = weight;
    }
    public boolean isMarried() {
        return married;
    }
    public void setMarried(boolean married) {
        this.married = married;
    }
}
```

Utils 代码（这个使用了 jackson 包）

```
package com.donlianli.es.util;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.ObjectWriter;
```

```
public class Utils {
	public static ObjectMapper mapper = new ObjectMapper();
	public static ObjectMapper getMapper(){
		return mapper;
	}
	public static String toJson(Object o){
		try {
			ObjectWriter writer = mapper.writerWithDefaultPrettyPrinter();
			return writer.writeValueAsString(o);
		} catch (JsonProcessingException e) {
			e.printStackTrace();
			return "转换 JSON 时发生异常";
		}
	}
}
```

因为这个数据，我们下一节需要使用。

聚合 facet

下面对上一节的 bulk 索引的数据进行统计，做一下类似 sql 的 group 统计，看一下 married 有多少，没有 married 有多少。

```
package com.donlianli.es.test1;

import java.util.Map;

import org.elasticsearch.action.search.SearchRequestBuilder;
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.client.Client;
import org.elasticsearch.index.query.BoolFilterBuilder;
import org.elasticsearch.index.query.FilterBuilders;
import org.elasticsearch.search.facet.Facet;
import org.elasticsearch.search.facet.FacetBuilder;
import org.elasticsearch.search.facet.FacetBuilders;
import org.elasticsearch.search.facet.Facets;
import org.elasticsearch.search.facet.terms.TermsFacet;

import com.donlianli.es.util.ESUtils;
/**
 * Facet 测试,即聚合测试
 * @author donlianli@126.com
 */
public class FacetTest {
	/**
```

```
     * @param args
     */
    public static void main(String[] args) {
        Client client = ESUtils.getClient();
        //过滤器
        BoolFilterBuilder filter = FilterBuilders.boolFilter();
        filter.must(FilterBuilders.matchAllFilter());
//      filter.must(FilterBuilders.rangeFilter("age").from(0).to(3));
        //请求
        SearchRequestBuilder searchBuilder =
client.prepareSearch(ESUtils.INDEX_NAME).setTypes(ESUtils.USER_TYPE);
        //分组条件
        FacetBuilder marriedFacet =
FacetBuilders.termsFacet("married").field("married").allTerms(true);;
        marriedFacet.facetFilter(filter);
        searchBuilder.addFacet(marriedFacet);
        long beginTime = System.currentTimeMillis();
        SearchResponse response = searchBuilder
        //设置过滤条件
        .execute()
        .actionGet();
        Facets facets = response.getFacets();
        Map<String, Facet> map = facets.getFacets();

        Facet facet = map.get("married");
        TermsFacet mFacet = (TermsFacet)facet;
        for(TermsFacet.Entry entry : mFacet.getEntries()){
            System.out.println("key:" +entry.getTerm().toString()
                    + " count:" + entry.getCount());
        }
        long total = System.currentTimeMillis() - beginTime;
        System.out.println("useTime:"+total);
    }
}
```

输出结果如下：

```
key:F count:20043

key:T count:19957

useTime:96
```

所以大家要注意，对 boolean 的值进行索引后，得到的并不是 true，或者 false 这样的分组，而是 T(TRUE)和 F(FALSE)。ES 的 facet 是一项很强大的功能，支持 bulk facet,可以将多个 facet 一次传递给 es，一次性返回结果。进行 facet 应用的最广的，可能就是一些生活服务电商。

比如下面分类后面的数量。

结构定义 mapping

相当于数据库的表结构的定义，elasticsearch 的 mapping 也很重要。直接关系到性能及搜索结果的准确性。为了说明 mapping 的定义，我这里定义了一个简单的模型，就 ID,type, 和 catIds 3 个属性，重在说明如何使用 java api 来定义 mapping,具体各 field 应该如何定义，这里不做讨论。

```
import java.io.Serializable;
import java.util.List;
/**
 * 这个是为分组定义的一个模型
 * catIds 通常为一对多的分类 ID
 * @author donlianli@126.com
 */
public class TestModel implements Serializable {
	private static final long serialVersionUID = 3174577828007649745L;
	//主 ID
	private long id;
	//类型，为 types 之一
	private String type;
	/**
	 * 这里是一个列表
	 */
	private List<Integer> catIds;


	public long getId() {
		return id;
	}
	public void setId(long id) {
		this.id = id;
	}
	public String getType() {
		return type;
	}
```

```
    public void setType(String type) {
        this.type = type;
    }
    public List<Integer> getCatIds() {
        return catIds;
    }
    public void setCatIds(List<Integer> catIds) {
        this.catIds = catIds;
    }
}
```

我们假设 id 就存储为 long 类型，type 存储为字符串类型，catIds 为一个列表，其实际类型为 integer 类型。定义的 mapping 如下：

```
/**
     * mapping 一旦定义，之后就不能修改。
     * @return
     * @throws Exception
     */
    private static XContentBuilder getMapping() throws Exception{
        XContentBuilder mapping = jsonBuilder()
                    .startObject()
                      .startObject("test")
                      .startObject("properties")
                        .startObject("id")
                            .field("type", "long")
                            .field("store", "yes")
                         .endObject()

                        .startObject("type")
                            .field("type", "string")
                            .field("index", "not_analyzed")
                         .endObject()

                        .startObject("catIds")
                            .field("type", "integer")
                        .endObject()
                      .endObject()
                    .endObject()
                .endObject();
        return mapping;
    }
```

注意：elasticsearch 的 field 一旦定义后就无法修改，你想增加一个 store 属性，都不行。

下面就是调用 JAVA API 了，注意，在定义 mapping 之前，还需要先创建一个 index 库。这里，我 index 库和 mapping 写到一个方法里面了。

```
      Client client = ESUtils.getClient();
       //首先创建索引库
       CreateIndexResponse  indexresponse = client.admin().indices()
       //索引名称不能包含大写字母
       .prepareCreate("esindex").execute().actionGet();
       System.out.println(indexresponse.acknowledged());;
       //如果是在两台机器上，下面直接 putMapping 可能会报异常
       PutMappingRequestBuilder builder =
client.admin().indices().preparePutMapping("esindex");
       // testMapping 就像当于数据的 table
       builder.setType("testMapping");
       XContentBuilder mapping = getMapping();
       builder.setSource(mapping);
       PutMappingResponse  response = builder.execute().actionGet();
       System.out.println(response.isAcknowledged());
```

其中，这个代码在我本机出现一点问题，当我创建完 index 后，直接创建 mapping 的时候，报 index missing。我把两个 es Node 停掉一个就没有问题了。可见，ES 将 create index 和 putMapping 放到了两个不同的 es Node 下面，导致了上面那个异常。

查询 mapping.

```
import org.elasticsearch.client.Client;
import org.elasticsearch.cluster.ClusterState;
import org.elasticsearch.cluster.metadata.IndexMetaData;
import org.elasticsearch.cluster.metadata.MappingMetaData;

import com.donlianli.es.ESUtils;
public class GetMappingTest {
    public static void    main(String[] argv) throws Exception{
        Client client = ESUtils.getClient();

        ClusterState cs = client.admin().cluster().prepareState()
                //索引库名称为 esIndex
                .setFilterIndices("esindex").execute().actionGet().getState();

        IndexMetaData imd = cs.getMetaData()
                //这个名称同上面的名称
                .index("esindex");
        //type 的名称
        MappingMetaData mdd = imd.mapping("testMapping");
        System.out.println(mdd.sourceAsMap());
    }
}
```

好了，有时为了测试，可能需要删除某个索引，代码如下：

```
Client client = ESUtils.getClient();
            client.admin().indices()
            .prepareDelete("testindex").execute().actionGet();
```

添加字段

如果日后，希望为之前的 type 增加一个字段，怎么办？其实只要把新的 mapping 重新执行一遍即可。Es 会自动发现新添加的字段，然后自动增加上去（增加字段的时候，请最好考虑默认值，历史数据如何处理等）。
我们为上面的 mapping 增加一个 firstKeyword 字段。

```
import static org.elasticsearch.common.xcontent.XContentFactory.jsonBuilder;

import org.elasticsearch.action.admin.indices.mapping.put.PutMappingRequestBuilder;
import org.elasticsearch.action.admin.indices.mapping.put.PutMappingResponse;
import org.elasticsearch.client.Client;
import org.elasticsearch.common.xcontent.XContentBuilder;

import com.donlianli.es.util.ESUtils;
/**
 * 在原有的 type 上面增加一个字段
 * @author donlianli@126.com
*/
public class AddFieldTest {
      public static void    main(String[] argv) throws Exception{
            Client client = ESUtils.getClient();
            //如果是在两台机器上，下面直接 putMapping 可能会报异常
            PutMappingRequestBuilder                    builder                    =
client.admin().indices().preparePutMapping("testindex");
            //testType 就像当于数据的 table
            builder.setType("testType");
            XContentBuilder mapping = getMapping();
            builder.setSource(mapping);
            PutMappingResponse    response = builder.execute().actionGet();
            System.out.println(response.isAcknowledged());
      }
      /**
       * mapping 一旦定义，之后就不能修改。
       * @return
       * @throws Exception
```

```
     */
    private static XContentBuilder getMapping() throws Exception{
        XContentBuilder mapping = jsonBuilder()
                        .startObject()
                          .startObject("test")
                          .startObject("properties")
                            .startObject("id")
                                .field("type", "long")
                                .field("store", "yes")
                             .endObject()

                            .startObject("type")
                                .field("type", "string")
                                .field("index", "not_analyzed")
                             .endObject()
                             //增加字段
                             .startObject("firstKeyword")
                                .field("type", "string")
                                .field("index", "analyzed")
                             .endObject()

                            .startObject("catIds")
                                .field("type", "integer")
                            .endObject()
                          .endObject()
                        .endObject()
                      .endObject();
        return mapping;
    }
}
```

不停服务更改 mapping

Elasticsearch 的 mapping 一旦创建，只能增加字段，而不能修改已经 mapping 的字段。但现实往往并非如此啊，有时增加一个字段，就好像打了一个补丁，一个可以，但是越补越多，最后自己都觉得惨不忍睹了。怎么办？？
这里有一个方法修改 mapping，那就是重新建立一个 index，然后创建一个新的 mapping。你可能会问，这要是在生产环境，可行吗？答案是，如果你一开始就采取了合适的设计，这个完全是可以做到平滑过渡的。
采取什么合理设计呢？就是我们的程序访问索引库时，始终使用同义词来访问，而不要使用真正的 indexName。在 reindex 完数据之后，修改之前的同义词即可。明白了吗？

参考上面的思路，我们来一步一步做。

step1、创建一个索引，这个索引的名称最好带上版本号，比如 my_index_v1,my_index_v2 等。

```
curl -XPUT 'http://localhost:9200/myindex_v1'
```

step2、创建一个指向本索引的同义词。

```
curl -XPOST localhost:9200/_aliases -d '
{
    "actions": [
        { "add": {
            "alias": "my_index",
            "index": "myindex_v1"
        }}
    ]
} '
```

此时，你可以通过同义词 my_index 访问。包括创建索引，删除索引等。

step3，需求来了，需要更改 mapping 了，此时，你需要创建一个新的索引，比如名称叫 my_index_v2（版本升级），在这个索引里面创建你新的 mapping 结构。然后，将新的数据刷入新的 index 里面。在刷数据的过程中，你可能想到直接从老的 index 中取出数据，然后更改一下格式即可。如何遍历所有的老的 index 数据，请参考游标那一节。
step4，修改同义词。将指向 v1 的同义词，修改为指向 v2。http 接口如下：

```
curl -XPOST localhost:9200/_aliases -d '
{
    "actions": [
        { "remove": {
            "alias": "my_index",
            "index": " myindex _v1"
        }},
        { "add": {
            "alias": "my_index",
            "index": " myindex_v2"
        }}
    ]
}
'
```

step5，删除老的索引。

```
curl -XDELETE localhost:9200/ myindex_v1
```

除此之外，还有几个其他的方法也可以更改 mapping。
1、修改程序，添加字段。
就是说，你可以在 mapping 中增加一个新的字段，然后你对新的字段进行访问统计搜索。

这个就要修改两个地方，一个是修改 mapping 增加字段，还有就是修改你的程序，把字段改成新的字段。

2、更改字段类型为 multi_field。

multi_field 允许为一个字段设置多个数据类型。应用 multi_field 的一个最典型的场景是：一个类型定义为 analyed，这个字段可以被搜索到，一个类型定义为不分词，这个字段用于排序。

任何字段都可以被更新为 multi_field（类型为 object 和 nested 的类型除外）。假设现在有一个字段，名字叫 created，类型现在为 string。

```
{     "created": { "type": "string"} }
```

我们可以将它增加一种类型，使他既能被当做字符串又能当做日期型。

```
curl -XPUT localhost:9200/my_index/my_type/_mapping -d '
{
    "my_type": {
        "properties": {
            "created": {
                "type":   "multi_field",
                "fields": {
                    "created": { "type": "string" },
                    "date":    { "type": "date"   }
                }
            }
        }
    }
}
'
```

采用标准的重建索引方式的时候，我们推荐大家为每一个 type 都建立一个索引同义词，即便在同一个索引库中的多个 type，也推荐使用建立一个同义词来访问。即一个 index 里面包含一个 type，因为在 elasticsearch 中，跨 index 查询数据是很方便的。这样，我们就可以在 reindex 一个 type 后，立即将 type 生效，而不是将 index 下面所有的 type 都重建完后，同义词才能生效。

索引设置

当我们没有对索引增加设置时，ES 默认对每个索引都创建了 5 个分片，1 个副本。也就是说，如果你没有显示的指明如何设置 shard 和 replica，那么 es 将把索引平均分片到这 5 个分片上。并且，如果你启动了多余 1 个 node，每个 node 都会启动一个副本。之所以 es 这么做是因为 shard 越多，理论上索引的速度就会越快。Replica 设置为 1 个是因为，防止 shard 不可用时，有一个备份可用做替换。如果我们希望更改这个默认配置，那么用 Java 代码应该怎么做呢？

我们默认的设置如下：

```
curl http://localhost:9200/esindex/_settings?pretty=1
```

显示：

```
{
  "esindex" : {
    "settings" : {
      "index" : {
        "uuid" : "0bN5pNnQQpirzOxRkZQu-Q",
        "number_of_replicas" : "1",
        "number_of_shards" : "5",
        "version" : {
          "created" : "1020299"
        }
      }
    }
  }
}
```

我们使用 Java 代码更改分配为 2，副本为 0.

```
import
org.elasticsearch.action.admin.indices.create.CreateIndexResponse;
import org.elasticsearch.client.Client;
import org.elasticsearch.common.settings.ImmutableSettings;
import org.elasticsearch.common.settings.Settings;

import com.donlianli.es.util.ESUtils;
/**
 * 创建索引时，指定分片及副本
 * 并且创建type一起执行
 * @author donlianli@126.com
 */
public class IndexSettingTest {
    public static void  main(String[] argv) throws Exception{
        Client client = ESUtils.getClient();
        Settings settings = ImmutableSettings.settingsBuilder()
                //2个主分片
                .put("number_of_shards", 2)
                //测试环境，减少副本提高速度
                .put("number_of_replicas", 0).build();
        //首先创建索引库
        CreateIndexResponse  indexresponse = client.admin().indices()
        //索引名称不能包含大写字母
        .prepareCreate("testindex").setSettings(settings)
```

```
        .execute().actionGet();
        System.out.println(indexresponse.isAcknowledged());
    }
}
```

为索引设置别名

```
import
org.elasticsearch.action.admin.indices.alias.IndicesAliasesResponse;
import org.elasticsearch.client.Client;

import com.donlianli.es.util.ESUtils;
/**
 * 为索引指定别名
 * @author donlianli@126.com
 */
public class AliasTest {
    public static void  main(String[] argv) throws Exception{
        Client client = ESUtils.getClient();
        IndicesAliasesResponse  aliasesresponse =
client.admin().indices()
        //索引起别名,一个索引可以起多个别名
        .prepareAliases().addAlias("testindex", "testalias")
        .execute().actionGet();
        System.out.println(aliasesresponse.isAcknowledged());
    }
}
```

这个时候使用以下命令时，也可以查看到索引的配置。

```
curl http://localhost:9200/testalias/_settings?pretty=1
```

注意，为了以后代码能够平稳升级及增加不兼容字段，最好为每个索引设置别名，并且在代码中都使用别名进行搜索和索引。另外，还有一点比较重要就是 shard 的数量一旦创建是不能更改的，replica 的数量是可以动态修改的。

测试连接

作为运维人员，如果 es 出现问题，希望今早发现问题，我们用程序应该如何测试 es 是否还活着？是否还能连接得上？

其实最简单的方法，就是使用 telnet 协议，看端口是否能 telnet 上。如果我们希望进一步更加准确的控制，比如说，看 es 的集群状态是否处于绿色状态。Es 的是否有节点已经出现分裂的节点。这些都可以通过 es 标准的 http 协议来查询。

查询 es 集群状态的命令。

```
curl -XGET 'http://localhost:9200/_cluster/health?pretty=true'
```

注意，es 为了将索引和命令进行区分，端口后面的命令都是以下划线为开始的，索引，你的索引名称就不能以下划线开头进行命名。
返回结果大概如下：

```
{
  "cluster_name" : "donlianli",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 2,
  "number_of_data_nodes" : 2,
  "active_primary_shards" : 5,
  "active_shards" : 10,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0
}
```

第二行 green 表示集群处于正常状态，如果是 red，表示集群处于不可用的状态。如果是 yellow，则表示在这不可用和正常状态之间。

游标

我们都知道，在 sql 中有一种查询就是 select * from table_x 的查询条件，在 es 中似乎只有一个 match_all 查询，但 es 并不推荐使用 match_all 进行遍历。Es 提供了一种遍历所有数据的方式，那就是游标的方式。
（注：什么时候需要我们遍历呢？我们可能为了复制一份集群，才会用到这种遍历的方式）

```
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.action.search.SearchType;
import org.elasticsearch.client.Client;
import org.elasticsearch.common.unit.TimeValue;
import org.elasticsearch.search.SearchHit;

import com.donlianli.es.util.ESUtils;
/**
 * 使用scroll方法实现复制索引
 * @author donlianli@126.com
 */
public class ScrollTest {
    public static void main(String[] args) {
        Client esClient = ESUtils.getClient();
        SearchResponse searchResponse =
esClient.prepareSearch(ESUtils.getIndexName())
        //加上这个据说可以提高性能，但第一次却不返回结果
        .setSearchType(SearchType.SCAN)
```

```
        //实际返回的数量为5*index的主分片格式，如果index是默认配置的话
        .setSize(5)
        //这个游标维持多长时间
        .setScroll(TimeValue.timeValueMinutes(8))
        .execute().actionGet();
        //第一次查询，只返回数量和一个scrollId
        System.out.println(searchResponse.getHits().getTotalHits());
        System.out.println(searchResponse.getHits().hits().length);
        //第一次运行没有结果
        for (SearchHit hit : searchResponse.getHits()) {
            System.out.println(hit.getSourceAsString());
        }
        System.out.println("------------------------------");
        //使用上次的scrollId继续访问
        searchResponse =
esClient.prepareSearchScroll(searchResponse.getScrollId())
        .setScroll(TimeValue.timeValueMinutes(8))
        .execute().actionGet();
        System.out.println(searchResponse.getHits().getTotalHits());
        System.out.println(searchResponse.getHits().hits().length);
        for (SearchHit hit : searchResponse.getHits()) {
            System.out.println(hit.getSourceAsString());
        }
    }
}
```

安装插件

Store 属性

在 Elasticsearch 创建 mapping 的时候，需要指定 store 属性和 index 属性，对于入门的学者，通常对 lucene 又不熟悉的人，通常不知道如何设置这两个值。

其中 index 属性还比较容易理解，一共 3 个值，no,analyzied，not_analyzied，分别对应'不对该字段进行索引（无法搜索）'，'分词后索引'，'以单个关键词进行索引'。就是说，如果这个字段不需要搜索，一般不需要设置为 analyzied。

在做搜索时，经常会需要对一个分类进行分组(facet)，比如搜索"ipad",需要在平板电脑分类下面统计有多少个商品，在电脑配件下面统计有多少个商品，这种统计数量的需求，我们通常会在 document 里面添加一个分类 ID，然后对所有的分类做 facet。那么问题出来了，如果分类 ID 是个整数的话，store 属性应该设置 true or false?这种整数是否还需要索引？

通过对 lucene 的学习得出，以上需求的设置 store 应该设置为 no，而 index 应该设置为 not_analyzied。

你可能会问，如果设置 store:"no"，会不会影响 facet 的性能。答案为不会，因为做 facet 的

时候并不会使用 store 的字段，而是使用的索引表的词。也就是后面设置的 index:'not_analyzied"。
实际上，我们在创建 mapping 的时候，如果设置了数据类型为 integer，那么通常采用默认的 index 设置即可。默认的 index 即为 no_analyzied。至于 store，采用默认值也是合适的（默认为 no)。因为一般我们是用不到 store='yes'的功能的，除非，我们需要对某个域（就是字段）进行高亮显示。

我自己做过一个测试，如果设置分类 ID 不索引（not_analyzied)，则无法进行 facet，可见任何需要 facet 的字段，必须进行索引。

还有一种情况，就是查询的时候，不想返回所有的字段，只想返回某些字段，是否需要将待返回的字段设置为 store 呢？
答案是取决于你到底返回几个字段。Es 在处理这种指定返回字段的时候，是这样处理的。如果指定返回的字段没有设置 store=true，则会从_source 字段加载，然后解析出来返回给客户端。如果客户端返回的字段设置了 store=true，则直接使用 store 的字段返回。那么当我指定的字段有的是有 store 的，有的是没有 store 的，es 怎么处理的呢？es 会把有 store 的都直接 load 出来，然后将没有 store 的字段从_source 解析，重新组装后返回给客户端。
下面是摘自 es 作者的解释。
By default in elasticsearch, the _source (the document one indexed) is stored. This means when you search, you can get the actual document source back. Moreover, elasticsearch will automatically extract fields / objects from the _source and return them if you explicitly ask for it (as well as possibly use it in other components, like highlighting).

You can specify that a specific field is also stored. This menas that the data for that field will be stored "on its own". Meaning that if you ask for "field1" (which is stored), elasticsearch will identify that its stored, and load it from the index instead of getting it from the _source (assuming _source is enabled).
这句话的意思是说，如果你指定要存储某个字段 A，并且指定要返回的字段包括这个字段 A,则 ES 会优先从 store 的字段加载 A，而不是从_source 字段加载后进行解析。
When do you want to enable storing specific fields? Most times, you don't. Fetching the _source is fast and extracting it is fast as well. If you have very large documents, where the cost of storing the _source, or the cost of parsing the _source is high, you can explicitly map some fields to be stored instead.
大部分情况，你不需要设置某个字段的 store 属性。

Query type 详解

es 在查询中，可以指定搜索类型为 QUERY_THEN_FETCH, QUERY_AND_FEATCH, DFS_QUERY_THEN_FEATCH 和 DFS_QUERY_AND_FEATCH。那么这 4 种搜索类型有什么区别？

分布式搜索背景介绍：

ES 天生就是为分布式而生，但分布式有分布式的缺点。比如要搜索某个单词，但是数据却分别在 5 个分片（Shard)上面，这 5 个分片可能在 5 台主机上面。因为全文搜索天生就要排序（按照匹配度进行排名）,但数据却在 5 个分片上，如何得到最后正确的排序呢？ES 是这样做的，大概分两步。
step1、ES 客户端会将这个搜索词同时向 5 个分片发起搜索请求，这叫 Scatter,
step2、这 5 个分片基于本 Shard 独立完成搜索，然后将符合条件的结果全部返回，这一步叫 Gather。
客户端将返回的结果进行重新排序和排名，最后返回给用户。也就是说，ES 的一次搜索，是一次 scatter/gather 过程（这个跟 mapreduce 也很类似）.

然而这其中有两个问题。
第一、数量问题。比如，用户需要搜索"双黄连"，要求返回最符合条件的前 10 条。但在 5 个分片中，可能都存储着双黄连相关的数据。所以 ES 会向这 5 个分片都发出查询请求，并且要求每个分片都返回符合条件的 10 条记录。当 ES 得到返回的结果后，进行整体排序，然后取最符合条件的前 10 条返给用户。这种情况，ES5 个 shard 最多会收到 10*5=50 条记录，这样返回给用户的结果数量会多于用户请求的数量。
第二、排名问题。上面搜索，每个分片计算分值都是基于自己的分片数据进行计算的。计算分值使用的词频率和其他信息都是基于自己的分片进行的，而 ES 进行整体排名是基于每个分片计算后的分值进行排序的，这就可能会导致排名不准确的问题。如果我们想更精确的控制排序，应该先将计算排序和排名相关的信息（词频率等）从 5 个分片收集上来，进行统一计算，然后使用整体的词频率去每个分片进行查询。

这两个问题，估计 ES 也没有什么较好的解决方法，最终把选择的权利交给用户，方法就是在搜索的时候指定 query type。
1、query and fetch
向索引的所有分片（shard）都发出查询请求，各分片返回的时候把元素文档（document）和计算后的排名信息一起返回。这种搜索方式是最快的。因为相比下面的几种搜索方式，这种查询方法只需要去 shard 查询一次。但是各个 shard 返回的结果的数量之和可能是用户要求的 size 的 n 倍。
2、query then fetch（默认的搜索方式）
如果你搜索时，没有指定搜索方式，就是使用的这种搜索方式。这种搜索方式，大概分两个步骤，第一步，先向所有的 shard 发出请求，各分片只返回排序和排名相关的信息（注意，不包括文档 document)，然后按照各分片返回的分数进行重新排序和排名，取前 size 个文档。然后进行第二步，去相关的 shard 取 document。这种方式返回的 document 与用户要求的 size 是相等的。
3、DFS query and fetch
这种方式比第一种方式多了一个初始化散发(initial scatter)步骤，有这一步，据说可以更精确控制搜索打分和排名。
4、DFS query then fetch
比第 2 种方式多了一个初始化散发(initial scatter)步骤。

DSF 是什么缩写？初始化散发是一个什么样的过程？

从 es 的官方网站我们可以指定，初始化散发其实就是在进行真正的查询之前，先把各个分片的词频率和文档频率收集一下，然后进行词搜索的时候，各分片依据全局的词频率和文档频率进行搜索和排名。显然如果使用 DFS_QUERY_THEN_FETCH 这种查询方式，效率是最低的，因为一个搜索，可能要请求 3 次分片。但，使用 DFS 方法，搜索精度应该是最高的。
至于 DFS 是什么缩写，没有找到相关资料，这个 D 可能是 Distributed，F 可能是 frequency 的缩写，至于 S 可能是 Scatter 的缩写，整个单词可能是分布式词频率和文档频率散发的缩写。

参考资料：

http://www.elasticsearch.org/blog/understanding-query-then-fetch-vs-dfs-query-then-fetch/

0.90.x 升级至 1.x 后问题

Es 升级到 1.x 后，之前的部分 api 不能再使用。主要的变化有下面这些。

一、系统级别及设置方面

1.1 es 启动时，默认是作为一个前台程序启动。如果你想让 es 作为一个后台守护进程，需要在启动命令后面加-d 参数。

1.2 命令行参数，默认不需要再加-Des.前缀。新的格式如下：

```
./bin/elasticsearch --node.name=search_1 --cluster.name=production
```

1.3 在 64 位的 linux 系统上面，默认采用内存映射文件(mmapfs)作为底层的存储结构。请确保 linux 参数 MAX_MAP_COUNT 设置的足够大。因为在 redhat 系列和 Debian 系列的 linux 系统中，这个默认值是 262144。

1.4 redhat 和 Debian 系列的 linux 系统，安装 elasticsearch，默认不是随系统自动启动。个人理解为采用 rpm 公用库安装的 elasticsearch 原来可能是随系统自动启动。

1.5 cluster.routing.allocation.disable_allocation,

cluster.routing.allocation.disable_new_allocation 和

cluster.routing.allocation.disable_replica_location 这三个参数被合并成了一个参数，新的参数名称及取值如下：

```
cluster.routing.allocation.enable:
all|primaries|new_primaries|none
```

个人只说一点，mongodb 也采用了 mmap 作为底层的存储方法，这种文件系统跟传统的文件系统有一个很大的区别就是，减少操作系统作为中间人将数据传来传去的麻烦，程序可以直接将数据刷入磁盘或者将数据从磁盘加载到内存，而不用操作系统先把磁盘数据先加载到内核区，再传递到用户程序的缓冲区步骤。

有关内存映射文件的更多介绍及可能遇到的问题，可以查看我之前的博客，

Mongodb FAQ 存储(storage)篇
mongodb Can't map file memory

二、统计信息相关命令的变化

2.1 有关集群状态 cluster_state，节点信息 nodes_info，节点统计信息 nodes_stats 和索引信息 indices_stats 命令格式进行了统一，比如查看集群信息使用命令：

curl -XGET http://localhost:9200/_cluster/state/nodes?pretty=1

查看节点统计信息：

curl -XGET http://localhost:9200/_nodes/stats?pretty=1

集群统计信息

curl -XGET http://localhost:9200/_cluster/stats?pretty=1

三、索引相关 api

mapping, alias, settings 和 warmer 相关命令和参数的顺序有所调整。新的顺序及格式如下：

Java 代码

```
curl -XPUT http://localhost:9200/{indices}/_mapping/{type}
curl -XPUT http://localhost:9200/{indices}/_alias/{name}
curl -XPUT http://localhost:9200/{indices}/_warmer/{name}

curl -XGET http://localhost:9200/{indices}/_mapping/{types}
curl -XGET http://localhost:9200/{indices}/_alias/{names}
curl -XGET http://localhost:9200/{indices}/_settings/{names}
curl -XGET http://localhost:9200/{indices}/_warmer/{names}

curl -XDELETE http://localhost:9200/{indices}/_mapping/{types}
curl -XDELETE http://localhost:9200/{indices}/_alias/{names}
curl -XDELETE http://localhost:9200/{indices}/_warmer/{names}
```

其中{indices},{type}和{name}可以是下面的任意一种：

- _all, * 或者为空，这 3 种取值意思都一样，代表所有可能的值
- 通配符，比如“test*”
- 逗号分隔的列表，比如： index_1,test_*

唯一的例外就是 DELETE 命令 ,这个命令不接收空的值。如果你想删除什么 ,必须明确指定。

同样，Get 命令返回的结果也进行了统一。

1. 只有查询有结果时，才返回具体的值，否则的话，只返回一个空对象{}。当查询的 mapping,warmer,alias,setting 不存在时，不再返回 404。
2. 如果查询到了结果，则结果中总是包含索引名称，然后是 section,然后是元素名称。例如：

```
{
    "my_index": {
        "mappings": {
            "my_type": {...}
        }
    }
}
```

上面是 `get_mapping` API 返回的结果。

In the future we will also provide plural versions to allow putting multiple mappings etc in a single request.

这句话的意思，好像是在说，可以在一个请求中设置两个版本的 mappings。

1、索引格式

1.x 之前的版本，被索引的文档 type 会同时出现在 url 和传输的数据格式中，如下：

```
PUT /my_index/my_type/1
{
  "my_type": {
     ... doc fields ...
  }
}
```

这种方式不太妥，如果一个 document，本身也有 my_type 域，那么就会有歧义。1.x 版本如果碰到上面的命令，会把 my_type 当成一个 document 的域进行处理。如果还想像以前一样使用，可以设置参数 index.mapping.allow_type_wrapper.

2、搜索

新的搜索接口，要求参数中必须有一个顶级参数 query。count，delete-by-query 和 validate-query 这三个命令，认为 query 这个 json 对象全是查询条件。新的格式如下：

```
GET /_count
{
    "query":{
        "match":{
            "title":"Interesting stuff"
        }
```

```
    }
}
```

1.x 之前，query 下面还可以包含 filter。下面是 0.90 版的查询：

```
GET 'http://localhost:9200/twitter/tweet/_search?routing=kimchy'-d
'{
    "query":{
        "filtered":{
            "query":{
                "query_string":{
                    "query":"some query string here"
                }
            },
            "filter":{
                "term":{"user":"kimchy"}
            }
        }
    }
}
'//这种格式不再支持
```

3、多字段搜索（Multi-fields)

新的格式如下：

```
"title":{
    "type":"string",
    "fields":{
        "raw":   {"type":"string","index":"not_analyzed"}
    }
}
```

4、停止词

新的版本没有使用英语默认的停止词(默认没有停止词）。

5、不带年份的日期值

新的版本，当没有指定是哪一年时，默认认为是 1970 年（在索引和搜索时都这样对待）。

6、参数

- meters 作为地理搜索的默认单位（1.x 之前是 miles)

- min_similarity,fuzziness 和 edit_distance 这些参数都别统一修改成了 fuzziness。
- ignore_missing 参数被 expand_wildcard，ignore_unavailable 和 allow_no_indices 替代。
- 在删除操作中，必须指定一个索引名称或者匹配符。

```
# v0.90 - delete all indices:
DELETE /

# v1.0 - delete all indices:
DELETE /_all
DELETE /*
```

7、返回值

- 返回值中 ok 字符已经被移除，作者认为这个附加信息毫无意义
- found,not_found 和 exists 参数被统一成为“found"。

Field values, in response to the `fields` parameter, are now always returned as arrays. A field could have single or multiple values, which meant that sometimes they were returned as scalars and sometimes as arrays. By always returning arrays, this simplifies user code. The only exception to this rule is when `fields` is used to retrieve metadata like the `routing` value, which are always singular. Metadata fields are always returned as scalars.

The `fields` parameter is intended to be used for retrieving stored fields, rather than for fields extracted from the `_source`. That means that it can no longer be used to return whole objects and it no longer accepts the `_source.fieldname` format. For these you should use

the `_source` `_source_include` and `_source_exclude` parameters instead.

- 参数查询中，返回的结果也是 JSON 格式。
- analyze 相关的 API，不再支持直接返回文本格式的结果，只支持 JSON 和 YAML 格式。

四、不支持的操作

- 文本搜索(text query)已经被移除，请使用匹配查询(match query)
- 面向域的搜索（field query）已经被移除，请使用 query_string 代替
- 面向文档的加权_boost 字段已经移除，请使用 function_score 代替
- path 参数被移除，请使用 copy_to 参数
- custom_score 和 custom_boost_score 不再被支持，请使用 fuction_score 代替

五、Elasticsearch 中的 FieldQuery 升级后的替代方法

在升级到 1.x 后，es 已经将 fieldQuery 搜索方法给去掉，其官方推荐的替代方法是 queryString。那么原来的 FiledQuery 新的搜索方法该如何使用？

下面是我写的一个对比方法：

0.90.x 版本

```
QueryBuilder query = QueryBuilders.fieldQuery("systemName", "*sys*");
```

1.x 版本

```
QueryBuilder query = QueryBuilders.queryString("systemName:*sys*");
```

可见，新的版本不仅 API 变了，而且写起来似乎比以前稍微复杂了点。其实比较复杂的是 queryString 的语法。但其实 queryString 方式的搜索，在 1.x 之前一直存在，属于一种比较灵活的搜索方法。下面我们对 queryString 的这种语法稍微做点研究。

- 查询 status 字段上包含 active 的写法

  status:active

- 查询 title 字段上包含 quick 或者 brown

  title:(quick brown)

- author 字段包含精确的"john simth"

  author:"John Simith"

- book.title,book.content 或者 book.date 字段任意一个包含"quick “或者包含 "brown"

  book.*(quick brown)

- 某个字段不包含值或者某个文档不存在某个字段

  missing:title

- 查询某个字段所有非 null 的值

  exists:title

queryString 大概有两点需要注意，第一，如果你不知道搜索的字段（域）,es 默认搜索_all 字段，如果你不指定默认的操作符，es 默认将分词的后 term 作 or 运算。也就是只要文档中包含分词后的任意一个词就算符合要求。

此外 queryString 还支持很多其他的搜索方法，更多搜索方法，请参考：

http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html#query-string-syntax

性能优化

垃圾回收使用 G1

Java 的垃圾回收真是让人又恨又爱。当今大内存已经成为服务器的趋势，使用 CMS 垃圾回收有点捉襟见肘。下面我就介绍一下在 elasticsearch 中，如何使用 G1 垃圾回收。
首先找到 es 的配置文件。elasticsearch.in.sh，这个文件在 es 的 bin 目录下面。
然后找到配置垃圾回收的配置。

Java 代码

```
# Force the JVM to use IPv4 stack
if [ "x$ES_USE_IPV4" != "x" ]; then
  JAVA_OPTS="$JAVA_OPTS -Djava.net.preferIPv4Stack=true"
fi

JAVA_OPTS="$JAVA_OPTS -XX:+UseParNewGC"
JAVA_OPTS="$JAVA_OPTS -XX:+UseConcMarkSweepGC"

JAVA_OPTS="$JAVA_OPTS -XX:CMSInitiatingOccupancyFraction=75"
JAVA_OPTS="$JAVA_OPTS -XX:+UseCMSInitiatingOccupancyOnly"
```

标红的这 4 行都删掉。
然后换成：

Java 代码

```
JAVA_OPTS="$JAVA_OPTS -XX:+UseG1GC"
JAVA_OPTS="$JAVA_OPTS -XX:MaxGCPauseMillis=200"
```

然后启动 es 就可以了。

好了，然后咱们回过头来说说为啥要使用 g1 垃圾回收。
首先：G1 是 Java7 以后推荐的垃圾回收方式。自然有他很多的好处，jdk 本身肯定对他做了很多优化。这个详细的我也不清楚。
但是，更重要的是，现在的 Java 程序，谁没有十来 G 的内存可以使用。特别像 es 这样的吃内存的存储系统。你甚至都想给他配个几百 G，省得你内存不够用。大内存导致 Java 的 GC 停顿时间长到让人难以容忍的程度。G1 就是专门为这种大内存的应用程序提出的新一代内存垃圾回收解决方案。

另外，oracle 官方建议，使用 g1 回收时，不要设置年轻代内存大小。
G1 的配置还有很多参数，详细参数见官网：
http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/G1GettingStarted/index.html

分片分布在不同机器

假如我们创建了一个集群，这个集群中有 2 台机器。集群中运行着无数的 index。我们希望其中任意一台机器 down 掉之后，仍然能够正常的对外提供服务。这能做到吗？
一般 es 默认的设置是为一个 index 创建 5 个 shard 和 1 个 replica（副本）。Shard 与 shard 之间存储的内容是不一样，但是 shard 和 replica 的内容却是一样的，如果我们能够保证 shard 和 replica 分布在不同的机器上，这样，即便其中一台机器 down，另外一台机器也能正常运作。那这样的需求在 es 如何设置呢？

有两种方法，都是修改 es 的配置文件（***%ES_HOME%/config/elasticsearch.yml***）：

方法一、设置 **cluster.routing.allocation.same_shard.host**: true。

这个设置是告诉 es,将同一 shard 的 primary shard 和 replica shard 分步在不同的主机上（官方文档说按照不同的 IP 或主机名称来判断是否是同一主机）。这个值，ES 默认是 false。

注意：如果是已经在生产环境使用的 ES（每个机器的节点大于 1 个时），修改完配置重启节点时，可能会导致最后一个节点没有数据（因为最后一个节点在重启的时候，其他节点已经将他这个节点上的数据分配到同一机器上的另外一个节点）。

方法二、设置 rack_id

设置如下：

```
node.rack_id: rack_1

cluster.routing.allocation.awareness.attributes: rack_id
```

Rack 原义是货架，在这里是指机柜。这个 rack_id 其实可以自己定义, 上面配置的第二行就是如何定义这个名称。这样定义后 ,分片会被尽量的分片在不同的 rack_id 上面。如果 rack_1 和 rack_2 在不同的机器上，则就能实现将 shard 分布在不同的机器上的效果。定义这个名称后，如何查看索引是否分布在不同的机器上(第一个命令只记录了 node_name,需要用第二个命令查询对应是那台机器）：

```
curl -XGET 'http://127.0.0.1:9250/index_name/_status?pretty=true'

curl -XGET 'http://127.0.0.1:9250/_cluster/nodes?pretty=true'
```

禁用自动 mapping

Filter cache

缓存是提高性能的很重要的手段，es 中的 filter cache 能够把搜索时的 filter 条件的结果进行缓存，当进行相同的 filter 搜索时（query 不同，filter 条件相同）,es 能够很快的返回结果。这是因为第一次计算完 filter 后，es 就把结果存储到了缓存中，下次搜索时，es 就不用再计算。

Es 的 filter cache 有两种，一种是 node 级别的 cache(filter cache 默认类型),一种是 index 级别的 filter cache。Node 级别的 cache 被整个 node 共享，并且可以使用百分比设置，

对应的属性为 **indices.cache.filter.size**，这个属性的值可以是百分比，也可以是具体的大小。Index 级别的 cache，顾名思义，就是针对单个索引的大小。Es 官方并不推荐使用这种设置，因为谁也无法预测索引级别的缓存到底有多大（可能非常大，超过了 node 的对内存），一个索引可能分布在多个 node 上面，而多个 node 的结果如果汇总到一个 node 上，其结果可想而知。

Field cache

当对字段排序或者对字段做聚合（如 facet）时，字段缓存（Field cache）非常重要。Es 会将这些待排序或者聚合字段都加载到内存，以提高对这些字段的快速访问。注意，将字段都加载到内存是非常耗费资源的，所以，你应该保证 field cache 足够大，以足以将所有的结果都缓存起来，下次排序或 facet 时不用再次从磁盘进行加载。

可以通过设置 **indices.fielddata.cache.size** 为具体的大小，比如 2GB，或者可用内存的百分比，比如 40%。请注意，这个属性是 node 级别(不是 index 级别的).当这个缓存不够用时，为了跟新的缓存对象腾出空间，原来缓存的字段会被挤出来，这会导致系统性能下降。所以，请保证这个值足够大，能够满足业务需求。另外，如果你没有设置这个值，es 默认缓存可以无限大。所以，在生产环境注意要设置这个值。

同时，我们也可以为 field cache 指定过期时间，系统默认缓存不过期。可以通过设置 indices.fielddata.cache.expire 为 10m，表示缓存 10 分钟过期。Es 建议，最好不要设置过期时间，因为将字段加载到内存是很浪费资源的。

设置 circuit breaker

Circuit breaker，断路器。这个和 field cache 有关系。断路器可以估算待加载的 field 的大小。通过断路器，可以防止将特别大的 field 加载到内存，导致内存溢出。断路器发现待加载的 filed 超过 java 的对内存时，会产生一个异常，防止 field 的继续加载，从而起到保护系统的作用。有两个属性可以设置断路器，一个是 **indices.fielddata.breaker.limit**,这

个值默认是 80%。这个值可以动态修改，通过集群设置的 api 就可以修改。80%就是说，当待加载的 field 超过 es 可用堆内存的 80%时，就会抛一个异常。

另一个属性是 **indices.fielddata.breaker.overhead**,默认值为 1.03，es 将使用这个值乘以 field 实际的大小作 Field 估算值。

Index Buffers

indices.memory.index_buffer _size ,这个值默认为 10% ,即堆内存的 10%会被用作 index 时的缓存，这个值可以设置百分比也可以是固定的大小。缓冲自然是越大越好。但记得千万不能超过可用的对内存，并且要跟 filter cache 和 filed cache 保证在一个合理的比例。

Index Refresh rate(索引刷新频率）

index.refresh_interval，默认为 1 秒。即被索引的数据，1 秒之后才能够被搜索到。这个时间越小，搜索和索引的性能就越低。这个时间越大，索引和搜索的性能就越高。es 建议，在 bulk index 非常大的索引数据时，将此值设置为-1，索引完毕之后再将此值修改回来。

综合考虑

记住，对 ES 来说，缓存（caches）与缓冲(buffers)是提高索引（index）和搜索（query）性能的关键因素。

在我们优化 es 之前，我们必须时刻牢记一点，es 需要足够多的内存，越多越好。但是，也不能把所有的内存都分配给 es。分配给 es 的内存最好是保持在物理内存的 50-60%左右，因为 os 也需要内存支持用户进程，比如分配线程，io 缓存等。但是，物理内存的 50-60%也不是唯一标准。假如你的内存有 256G，即便和 OS 预留 10%的内存，也有 25G，足够

操作系统使用。另外，最好设置 Xmx 和 Xms 一样大，避免 heap size 的 resizing。

做性能测试时，在相同的情形下，测试结果应该是可以重现的。你做的任何参数的修改，都应该使用进行性能测试，看性能是否有所提高。以性能测试为检验标准，是我们进行优化的必要前提。

建议

选择合适的存储方式

这里说的不是指内存。而是我们如何将数据存储到磁盘上。通常，如果 es 运行在 64 位系统上，建议采用 nmapfs 文件存储。如果不是 64 位系统，unix 系统使用 niofs 的方式，window 采用 simplefs 文件。如果你追求最大化的性能和吞吐量，最好让 es 运行在内存里面。文件则使用 memory。但这样，你得保证有足够大的内存可用。

降低刷新频率（index refresh rate)

如之前所述，这个频率越低，性能越高。但是可见性就越慢。

优化线程池

尤其要关注查询的线程池。当你进行压力测试时，你会发现瓶颈通常在搜索接口上。es 团队认为，当系统负载比较大时，最好立即决绝任何其他请求，而不是让请求排队等待更久的时间。es 团队也很想给一个比较合理的线程池设置数，但是，这个真的很难确定，这取决于具体的业务。很难有一个统一的标准。

优化段合并策略

如果想搜索快，保留尽量少的段。如果想索引快，段越零散越好。

设置 filter cache,filed cache 和断路器

提高 RAM buffer 的大小，以提高索引的速度

RAM buffer 默认为可用堆大小的 10%，提高这个值，可用极大的提高索引的速度。我们可以在集群的某些写节点上，提高这个比例，以达到提高索引速度的目的。

优化 transaction logging

es 在内部采用了 transaction logging，可以让索引的数据，很快就可以 get 到，并且保证数据持久化到磁盘上，从而提高索引的性能。这个是一个 shard 级别的数据结构。

默认情况下，当 transaction logging 记录的操作达到 5000，或者 transaction logging 的大小超过 200M 时 ,就会将数据立即写回磁盘。如果我们队数据的实时性要求不高的话，我们可以调整 index. translog.flush_threshold_ops 和 index.translog.flush_threshold_size 参数 ,我们见过有的生产环境 ,值是默认值的 10 倍。

另外，当 node 恢复时，如果 transaction logging 太大，恢复过程会比较漫长。

数据文件简介

第三部分：lucene 基本应用

Lucene4 基础概念

学习 Lucene 是为了更深入搜索，学习 Lucene4 是为了弄懂 ES 中没有解释的疑问，等看完 Lucene，才发现，搜索的核心原来都是 Lucene。

Lucene 是一个搜索引擎库（library），它并非一个像 tomcat 一样的产品。它衍生出了 solr 和 elasticsearch。当然，我学习的是后者。solr 是 apache 孵化的一个搜索引擎。下面这些概念是我做的一个粗滤的总结，如果需要详细了解，还是请参考 lucene 的官方文档。

索引(Index)
对应一个倒排表，一个检索的基本单位。在 lucene 中就对应一个目录。

段(Segment)

一个索引可以包含多个段，段与段之间是独立的，添加新文档可以生成新的段，不同的段可以合并。段是索引数据存储的单元。

文档(Document)

•文档是我们建索引的基本单位，不同的文档是保存在不同的段中的，一个段可以包含多篇文档。

•新添加的文档是单独保存在一个新生成的段中，随着段的合并，不同的文档合并到同一个段中。

域(Field)

•一篇文档包含不同类型的信息，可以分开索引，比如标题，时间，正文，作者等，都可以保存在不同的域里。

•不同域的索引方式可以不同。

词(Term)

词是索引的最小单位，是经过词法分析和语言处理后的字符串。

词相同,但域不同被认为是两个不同的词,也就是说词是词根和域名的一个组合。

词向量(Term Vector)

又称文档向量（document vector),由词文本和词频率组成。

语义树

语义树是构成搜索处理的一个中间结果，搜索时，会生成语义树，然后再进行搜索。

权重（Term Weight)

计算分值时使用的主要指标，指词(Term）在文档中的分值，脱离文档单独说某个词的权重是没有意义的。

Term Frequency (tf)：即此 Term 在此文档中出现了多少次。tf 越大说明越重要。

Document Frequency (df)：即有多少文档包含次 Term。df 越大说明越不重要 。

Posting

一般情况下,将一个词条所索引的文档(一般用文档编号表示)称之为 Posting，那么一个词条索引的多个文档就称之为 Posting-list。这个词我们在看 java api 的时候会经常看到

Payload

即词条（Term）的元数据或称载荷，Lucene 支持用户在索引的过程中将词条的元数据添加的索引库中，同时也提供了在检索结果时读取 Payload 信息的功能。Payload 的诞生为用户提供了一种可灵活配置的高级索引技术，为支持更加丰富的搜索体验创造了条件。

倒排表（Inverted Indexing）

倒排表是 Lucene 索引采用的一套数据结构，这种结构以词为中心，能够快速找到包含该词根的文档。因为跟正常的遍历文档检索采用的方法相反，采用先匹配词，在去找包含词的文档，因此叫倒排表。倒排表是一种数据结构，lucene 的数据文件一起构成了一张大的倒排表，而不是具体的某文件存储的倒排结构。

文档编号（Document Number）

Lucene 内部通过文档编号索引文档。这个编号在一个段内部唯一，一个段的第一个文档的编号为 0，依次递增。不过这个编号仅用于 lucene 内部使用，而且这个编号在段合并的时候会发生改变。如果需要在段外部使用，必须对这个编号进行唯一性重新编排，确保一个文档在更大的范围也是唯一的。重新编排的一个实现方法是，基数+段内序号的方法。比如有两个段，每个段里面都有 5 个文档，则第一个段的文档编号=0+段内编号，第二个段的文档编号=5+段内编号。