Elasticsearch知识体系

**文档信息**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **文档版本** | **XX版本** | **作者** | **时间** | **备注** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[I. 生态圈介绍 4](#_Toc477536829)

[II. 架构介绍 4](#_Toc477536830)

[III. 原理描述 6](#_Toc477536831)

[启动过程 6](#_Toc477536832)

[探测失效节点 7](#_Toc477536833)

[与Elasticsearch进行通信 8](#_Toc477536834)

[索引数据 8](#_Toc477536835)

[数据查询 9](#_Toc477536836)

[IV. 通用应用方案 10](#_Toc477536837)

[Elasticsearch + Logstash + Kibana方案 10](#_Toc477536838)

[Transwarp Hyperbase + Elasticsearch方案 10](#_Toc477536839)

[V. 操作示例 11](#_Toc477536840)

[基础API 12](#_Toc477536841)

[编入Document 12](#_Toc477536842)

[获取整个Document 14](#_Toc477536843)

[获取部分Document 14](#_Toc477536844)

[查看Document是否存在 15](#_Toc477536845)

[删除Document 15](#_Toc477536846)

[更新Document 16](#_Toc477536847)

[新建一个Document 16](#_Toc477536848)

[轻量检索 17](#_Toc477536849)

[URI检索 18](#_Toc477536850)

[输出结果分页 19](#_Toc477536851)

[Query String语法 20](#_Toc477536852)

[Request Body检索 22](#_Toc477536853)

[Query DSL 22](#_Toc477536854)

[查询语句 23](#_Toc477536855)

[查询语句的组合和嵌套 23](#_Toc477536856)

[Query和Filter 24](#_Toc477536857)

[映射与分词基础 25](#_Toc477536858)

[数据类型差异 25](#_Toc477536859)

[确切值(Exact values) vs.全文文本(Full text) 26](#_Toc477536860)

[倒排索引(inverted index) 26](#_Toc477536861)

[分词与分词器 28](#_Toc477536862)

[映射(mapping) 31](#_Toc477536863)

[复合核心字段类型 34](#_Toc477536864)

[Mapping操作 37](#_Toc477536865)

[显式定义Mapping 38](#_Toc477536866)

[定义元数据字段Mapping 39](#_Toc477536867)

[定义字段Mapping 40](#_Toc477536868)

[常见查询语句 41](#_Toc477536869)

[Full Text查询语句 41](#_Toc477536870)

[Term-level查询语句 42](#_Toc477536871)

[复合查询语句 44](#_Toc477536872)

[VI. 场景方案实现 46](#_Toc477536873)

[ELK搭建 46](#_Toc477536874)

[VII. 源码编译 48](#_Toc477536875)

[VIII. 对应UML图 49](#_Toc477536876)

[IX. 二次开发介绍 50](#_Toc477536877)

[Elasticsearch的Java API的统计查询示例 50](#_Toc477536878)

[Elasticsearch的Java API的模糊查询示例 52](#_Toc477536879)

[Elasticsearch的Java API的多条件查询示例 53](#_Toc477536880)

[Elasticsearch的Java API的JSON查询 53](#_Toc477536881)

[Elasticsearch的Java API的创建索引 54](#_Toc477536882)

# 生态圈介绍

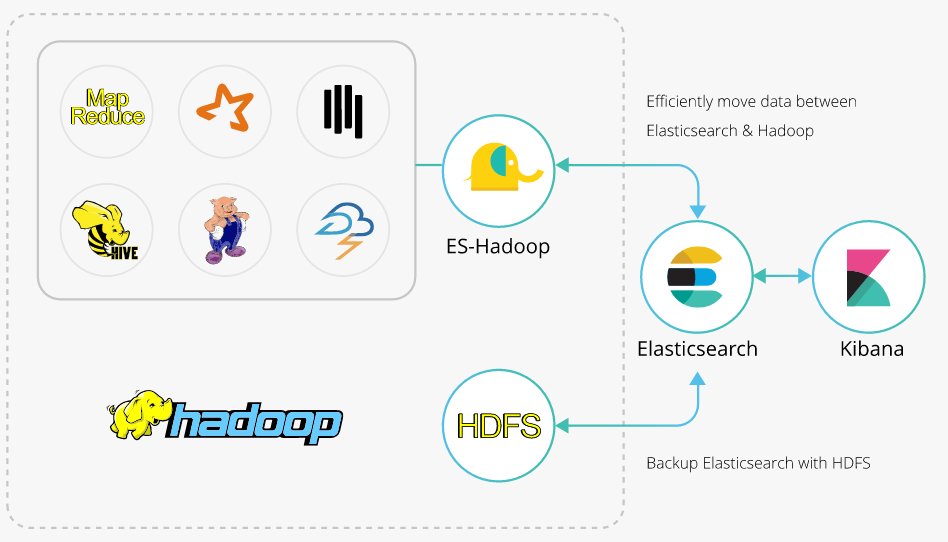
Elasticsearch是一个开源的搜索引擎，建立在一个全文搜索引擎库Apache Lucene基础之上。Lucene可能是目前存在的，不论开源还是私有的，拥有最先进，高性能和全功能搜索引擎功能的库。

但是 Lucene 仅仅只是一个库。为了利用它，需要编写Java程序，并在Java程序里面直接集成Lucene包。更坏的情况是，对信息检索有一定程度的理解才能明白Lucene是怎么工作的。Lucene是很复杂的。

Elasticsearch也是使用Java编写的，它的内部使用Lucene做索引与搜索，但是它的目标是使全文检索变得简单，通过隐藏Lucene的复杂性，取而代之的提供一套简单一致的RESTful API。然而，Elasticsearch不仅仅是Lucene，并且也不仅仅只是一个全文搜索引擎。它可以被下面这样准确的形容：

1. 一个分布式的实时文档存储，每个字段可以被索引与搜索
2. 一个分布式实时分析搜索引擎
3. 能胜任上百个服务节点的扩展，并支持PB级别的结构化或者非结构化数据

Elasticsearch将所有的功能打包成一个单独的服务，这样可以通过程序去访问它提供的简单的RESTful API服务，不论是编程语言还是直接使用命令行（去充当这个客户端）。就Elasticsearch而言，起步很简单。对于初学者来说，它预设了一些适当的默认值，并隐藏了复杂的搜索理论知识。它开箱即用。下图是展示Elasticsearch目前的生态圈：



# 架构介绍

Elasticsearch中的几个概念：

1、集群(Cluster)一组拥有共同的cluster name的节点

2、节点(Node)集群中的一个Elasticsearch实例

3、索引(Index)相当于关系数据库中的database概念，一个集群中可以包含多个索引。这个是个逻辑概念

4、主分片(Primary shard)索引的子集，索引可以切分成多个分片，分布到不同的集群节点上。分片对应的是Lucene中的索引

5、副本分片(Replica shard)每个主分片可以有一个或者多个副本

6、类型(Type)相当于数据库中的table概念，mapping是针对Type的。同一个索引里可以包含多个Type

7、Mapping 相当于数据库中的schema，用来约束字段的类型，不过 Elasticsearch的mapping可以自动根据数据创建

8、文档(Document)相当于数据库中的row

9、字段(Field)相当于数据库中的column

10、分配(Allocation)将分片分配给某个节点的过程，包括分配主分片或者副本。如果是副本，还包含从主分片复制数据的过程

Elasticsearch中的几个模块：

1、River: 代表Elasticsearch的一个数据源，也是其它存储方式同步数据到Elasticsearch的一个方法。它是以插件方式存在的一个Elasticsearch服务，通过读取River中的数据并把它索引到Elasticsearch中

2、Gateway: Gateway代表Elasticsearch索引的持久化存储方式，默认是先把索引存放到内存中去，当内存满了的时候再持久化到硬盘里。当这个 Elasticsearch 集群关闭或者再次重新启动时就会从Gateway中读取索引数据。Gateway代表的可以是本地文件系统、分片文件系统、Hadoop HDFS和Amazon S3存储系统

3、Discovery: ZenDiscovery代表Elasticsearch的自动节点发现机制，而且 Elasticsearch还是一个基于P2P的系统。首先它它会通过以广播的方式去寻找存在的节点，然后再通过多播协议来进行节点之间的通信，于此同时也支持点对点的交互操作。EC2Discovery代表了部署在AWS上的集群的多表协议的自动节点发现机制

4、Transport: Transport代表Elasticsearch内部的节点或者集群与客户端之间的交互方式。默认的内部是使用TCP协议来进行交互的，同时它支持HTTP、Thrift、Memcached等多种的传输协议（通过插件方式集成）

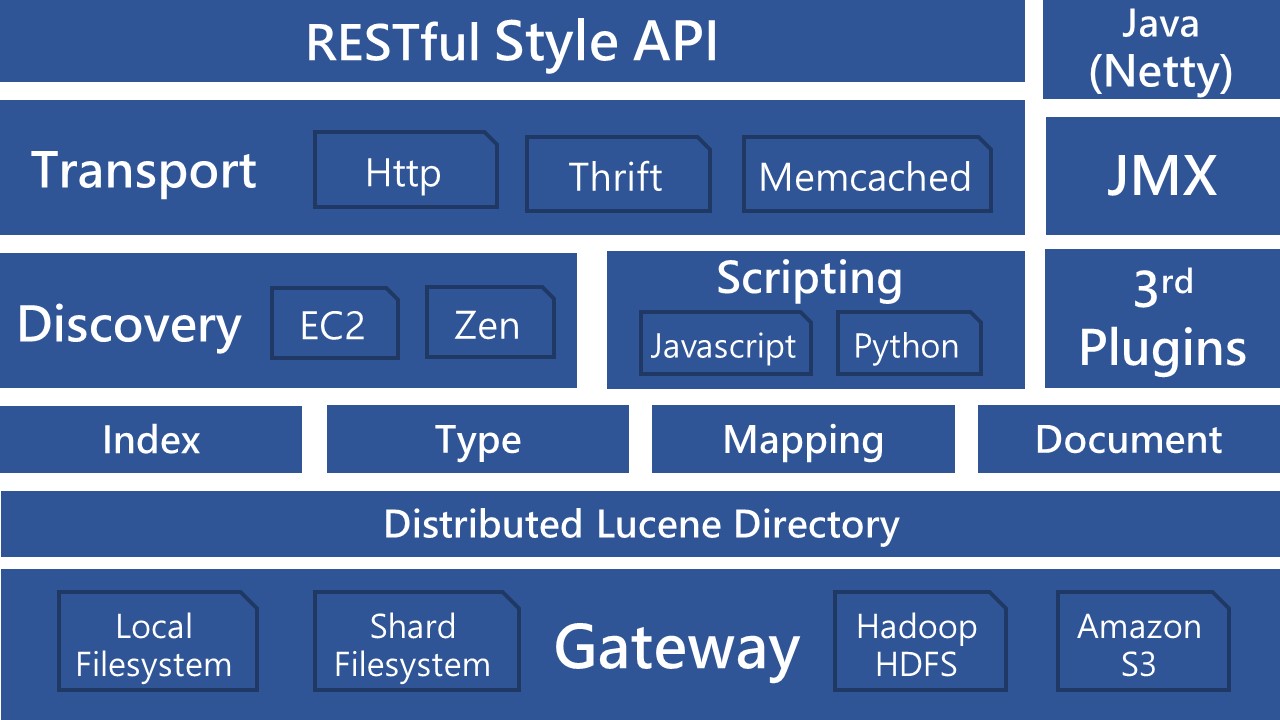
5、Search: Search代表了Elasticsearch中强大的搜索功能。Elasticsearch目前支持前后模糊、范围、精确查询等多种查询方式，并可在查询中快速得出相应的统计信息

Elasticsearch的常用插件包括head（提供图形界面）、analysis-ik（提供中文分词）、analysis-mmseg（提供中文分词）、bigdesk（提供Elasticsearch集群监控）、kopf（提供图形界面）等。

Elasticsearch还支持通过Javascript和Python等脚本接口的操作。但是更常用的操作还是通过REST API和Java API来进行。

此外，Elasticsearch的网络使用Netty，提供HTTP REST和RPC两种协议。

Elasticsearch分布式搜索引擎的总体框架图：

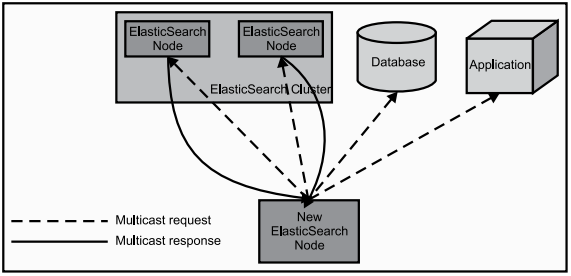


# 原理描述

接下来简单了解一下Elasticsearch的工作原理。

## 启动过程

当Elasticsearch的节点启动后，它会利用多播(multicast)（或者单播，如果用户更改了配置）寻找集群中的其它节点，并与之建立连接。这个过程如下图所示



在集群中，一个节点被选举成主节点(master node)。这个节点负责管理集群的状态，当群集的拓扑结构改变时把索引分片分派到相应的节点上。需要注意的是，从用户的角度来看，主节点在Elasticsearch中并没有占据着重要的地位，这与其它的系统（比如数据库系统）是不同的。实际上用户并不需要知道哪个节点是主节点；所有的操作需求可以分发到任意的节点，Elasticsearch内部会完成这些让用户感到不明觉历的工作。在必要的情况下，任何节点都可以并发地把查询子句分发到其它的节点，然后合并各个节点返回的查询结果。最后返回给用户一个完整的数据集。所有的这些工作都不需要经过主节点转发（节点之间通过P2P的方式通信）。

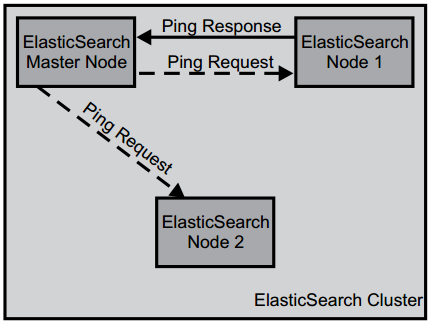
主节点会去读取集群状态信息；在必要的时候，会进行恢复工作。在这个阶段，主节点会去检查哪些分片可用，决定哪些分片作为主分片。处理完成后，集群就会转入到黄色状态。

这意味着集群已经可以处理搜索请求了，但是还没有火力全开（这主要是由于所有的主索引分片(primary shard)都已经分配好了，但是索引副本还没有）。接下来需要做的事情就是找到复制好的分片，并设置成索引副本。当一个分片的副本数量太少时，主节点会决定将缺少的分片放置到哪个节点中，并且依照主分片创建副本。所有工作完成后，集群就会变成绿色的状态（表示所有的主分片的索引副本都已经分配完成）。

## 探测失效节点

在正常工作时，主节点会监控所有的节点，查看各个节点是否工作正常。如果在指定的时间里面，节点无法访问，该节点就被视为出故障了，接下来错误处理程序就会启动。集群需要重新均衡——由于该节点出现故障，分配到该节点的索引分片丢失。其它节点上相应的分片就会把工作接管过来。换句话说，对于每个丢失的主分片，新的主分片将从剩余的分片副本(Replica)中选举出来。重新安置新的分片和副本的这个过程可以通过配置来满足用户需求。

由于只是展示Elasticsearch的工作原理，我们就以下图三个节点的集群为例。集群中有一个主节点和两个数据节点。主节点向其它的节点发送Ping命令然后等待回应。如果没有得到回应（实际上可能得不到回复的Ping命令个数取决于用户配置），该节点就会被移出集群。



## 与Elasticsearch进行通信

我们已经探讨了Elasticsearch是如何构建起来的，但是归根到底，最重要的是如何往Elasticsearch中添加数据以及如何查询数据。为了实现上述的需求，Elasticsearch提供了精心设计的API。这些主要的API都是基于REST风格。而且这些API非常容易与其它能够处理HTTP请求的系统进行集成。

Elasticsearch认为数据应该伴随在URL中，或者作为请求的主体(request body)，以一种JSON格式的文档发送给服务器。如果读者用Java或者其它运行在JVM虚拟机上的语言，应该关注一下Java API，它除了是群集中内置的REST风格API外，功能与URL请求是一样的。

## 索引数据

Elasticsearch提供了多种索引数据的办法。最简单的是使用索引API。通过它可以将文档添加到指定的索引中去。比如，通过curl工具，我们可以通过如下的命令创建一个新的文档：

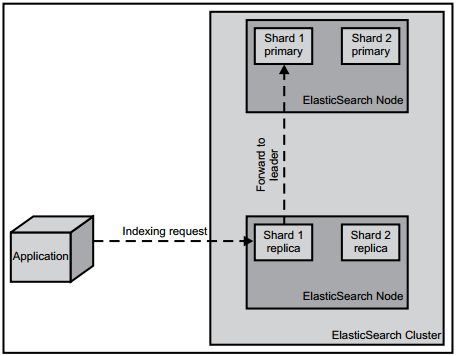
curl -XPUT http://localhost:9200/blog/article/1 -d '{

"title": "New version of Elastic Search released!", "content": "...",

"tags": ["announce", "elasticsearch", "release"]

}'

有一点需要注意，索引数据的操作只会发生在主分片(primary shard)上，而不会发生在分片副本(Replica)上。如果索引数据的请求发送到的节点没有合适的分片或者分片是副本，那么请求会被转发到含有主分片的节点。

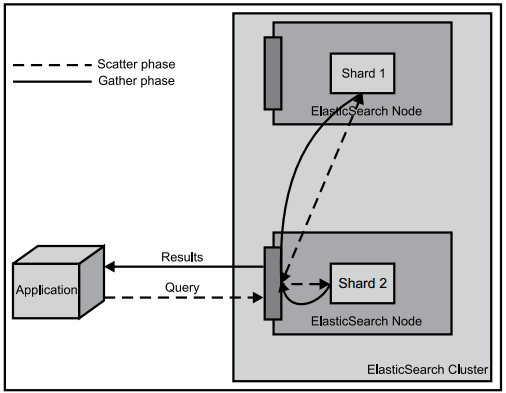


## 数据查询

查询API在Elasticsearch中有着很大的比重。通过使用Query DSL（基于JSON，用来构建复杂查询的语言），使用各种类型的查询方式，包括：简单的关键词查询(termquery) 、短语(phrase)、区间(range)、布尔(boolean)、模糊(fuzzy)、跨度(span)、通配符(wildcard)、地理位置(spatial)等其它类型的查询方式。

* 通过组合简单查询构建出复杂的查询
* 过滤文档，去除不符合标准的文档而且不影响打分排序
* 查找给定文档的相似文档
* 查找给定短语的搜索建议和查询短语修正
* 通过faceting构建动态的导航和数据统计
* 使用prospective search而且找到匹配写定文档的查询语句。（关于prospective search，似乎是一种推送方式。即把用户的查询语句存储到索引中，如果新的文档添加到索引中，就把文档关联到匹配的查询语句中。这种查询适合于新闻、博客等会定时更新的应用场景）

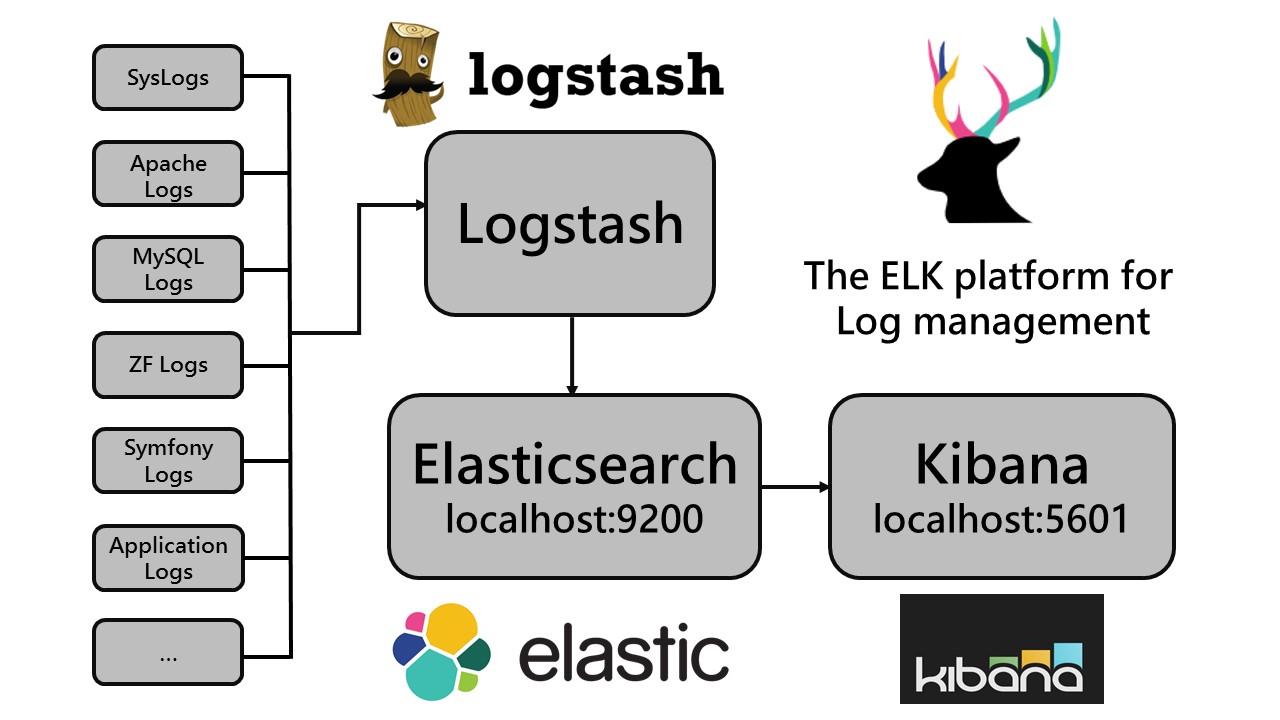
关于数据查询，其核心点在于查询过程不是一个简单、单一的流程。通常这个过程分为两个阶段：查询分发阶段和结果汇总阶段。在查询分发阶段，会从各个分片中查询数据；在结果汇总阶段，会把从各个分片上查询到的结果进行合并、排序等其它处理过程，然后返回给用户。



# 通用应用方案

## Elasticsearch + Logstash + Kibana方案

Elasticsearch + Logstash + Kibana(ELK)是一套开源的日志管理方案，主要针对分布式集群或者数据量级较大时会显得心有余而力不足的局面。下面对这三种工具逐一做简单介绍。



Logstash是一个用来搜集，分析，过滤日志的工具。它支持几乎任何类型的日志，包括系统日志、错误日志和自定义应用程序日志。它可以从许多来源接收日志，这些来源包括syslog、消息队列和JMX，它能够以多种方式输出数据，包括电子邮件、websockets和Elasticsearch。

Elasticsearch是实时全文搜索和分析引擎，提供搜集，分析，存储数据三大功能；是一套开放REST和Java API等结构提供高效搜索功能，可扩展的分布式系统。它构建于Apache Lucene搜索引擎库之上。

Kibana是一个基于Web的图形界面，用于搜索、分析和可视化存储在 Elasticsearch指标中的日志数据。它利用Elasticsearch的REST接口来检索数据，不仅允许用户创建他们自己的数据的定制仪表板视图，还允许他们以特殊的方式查询和过滤数据。

在不同的应用场景，ELK系统的构架略有不同，比如说有的场景运用到了Redis或者Kafka来做消息队列，以减轻Logstash的压力，以防数据丢失。

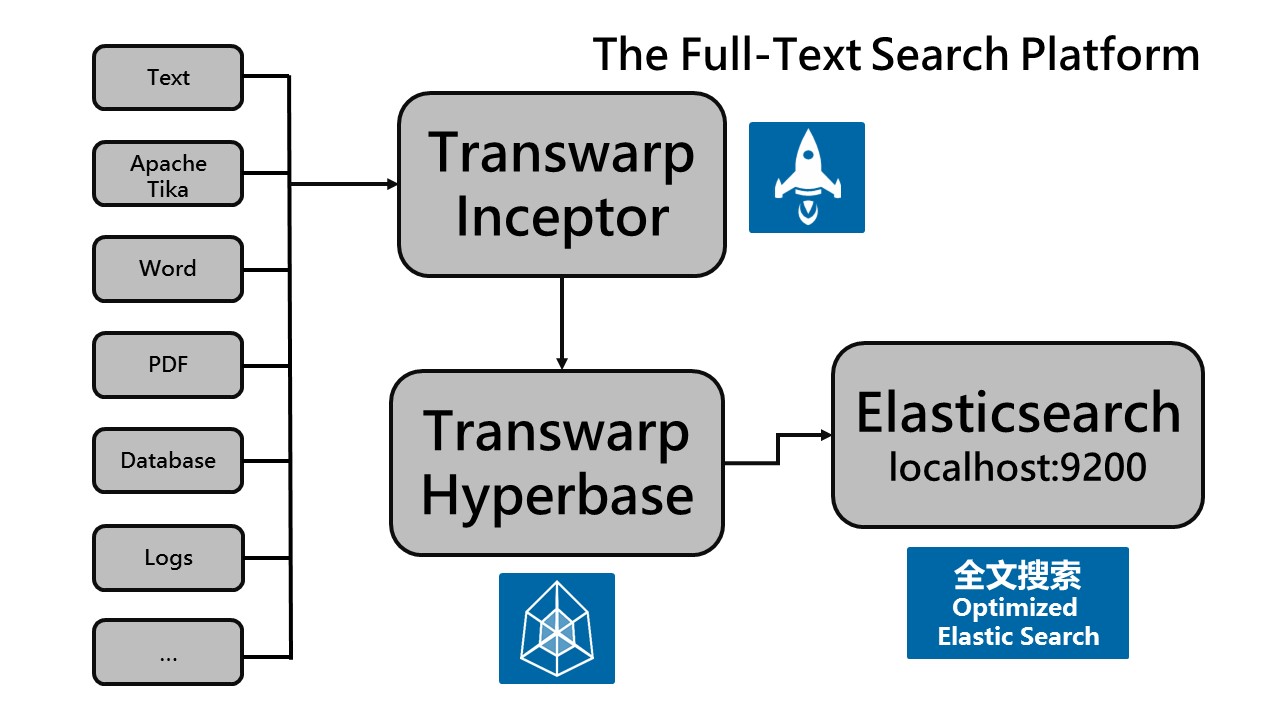
## Transwarp Hyperbase + Elasticsearch方案

全文索引功能集成在Transwarp Hyperbase中，可在Hyperbase上建立索引任务，通过Spark分布式计算框架把子任务分发给节点服务器，把需求作为查询条件的列建立一个映射到主数据的索引，利用平台的处理能力，结合Big Table实时创建索引并进行搜索，快速返回搜索结果。全文索引通过建立词库的方式记录词的出现位置及次数，以加速数据查询。创建全文索引需要在创建表时同时指定需要对哪些column families的哪些columns创建全文索引。

基于SQL接口的支持，用户在TDH上进行关键字搜索的门槛变得很低，整个关键字搜索的实现过程是：

* 用户可以根据业务场景选择将全量数据存入Elasticsearch，还是将原始数据存入Hyperbase索引数据存入Elasticsearch，两种方式对应不同的建表操作。如果是选择使用第二种方式，需要在建立Hyperbase表时定义fulltext index、是否启用分析和对分词建立索引
* 用户可以利用非常简便的使用insert语法插入数据，在数据入库过程中建立索引，也可以使用build index语法对已经入库的数据创建索引，该过程均是利用集群所有计算节点的资源分布式完成。每分钟可以处理几十GB的数据
* 数据在入Elasticsearch时会进行分词操作，TDH支持常见的中英文分词器，如IKAnalyzer，根据应用需求可以扩展分词器。目前平台支持的语言包括英文，简体/繁体中文
* 用户可以利用SQL语法进行任意模糊的关键字搜索
* 用户可以自由选择将检索结果导出文件，或者是保存到其他存储介质，或者是参与其他数据统计和数据分析

Transwarp Inceptor + Transwarp Hyperbase + Transwarp Elasticsearch的搭建方便简单，只需在TDH管理界面添加相应的服务，并且点击组件间的相互依赖，即可完成部署。因此不在场景方案实现那一节中阐述。



# 操作示例

所有其他语言可以使用RESTful API通过端口9200和Elasticsearch进行通信，甚至可以使用curl命令来和Elasticsearch交互。Elasticsearch为以下语言提供了官方客户端Groovy、JavaScript、.NET、PHP、Perl、Python和Ruby，还有很多社区提供的客户端和插件，所有这些都可以在Elasticsearch Clients中找到。

一个Elasticsearch请求和任何HTTP请求一样由若干相同的部件组成：

curl -X<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY\_STRING>' -d '<BODY>'

被< >标记的部件：

VERB：适当的HTTP方法或谓词：GET、POST、PUT、HEAD或者DELETE

PROTOCOL：http或者https（如果你在Elasticsearch前面有一个https代理）

HOST：Elasticsearch集群中任意节点的主机名，或者用localhost代表本地机器上的节点

PORT：运行Elasticsearch HTTP服务的端口号，默认是9200

PATH：API的终端路径（例如\_count将返回集群中文档数量）。Path可能包含多个组件，例如：\_cluster/stats和\_nodes/stats/jvm

QUERY\_STRING：任意可选的查询字符串参数（例如?pretty将格式化地输出JSON返回值，使其更容易阅读）

BODY：一个JSON格式的请求体（如果请求需要的话）

## 基础API

本章我们搭建一个员工信息Index（Index名为employee），并通过这个Index来演示一些常见操作。对这个Index我们将进行如下设计

• Index下的每个Document对应一名员工的信息

• Index下的Type按部门分类，分为 dev（研发部）、finance（财务部）和 sales（销售部）

### 编入Document

目前，employee Index还不存在，但是我们无需专门创建它，只需直接将Document放入(PUT)对应的路径下，路径的形式为/<index>/<type>/<id>，Elasticsearch会在PUT时自动创建employee Index。例如，下面我们将员工Zhang San的Document编入employee下的dev：

curl -XPUT 'localhost:9200/employee/dev/1?pretty' -d '{

"firstname": "San",

"lastname": "Zhang",

"age": 26,

"on\_board\_date": "2015-10-31",

"hometown": "Beijing",

"school": "Nanjing University",

"married": false,

"about": "I love Beijing Opera"

}'

路径/employee/dev/1包含了三条信息：

• employee 为Index名字

• dev 为Type名字

• 1 为这条Document的ID。

Elasticsearch会输出下面信息，说明Document插入成功：

{

"\_index":"employee",

"\_type":"dev",

"\_id":"1",

"\_version":1,

"created":true

}

这条信息包含了一个Document的身份元数据(identity metadata)：\_index、\_type、\_id和\_version分别对应该Document的Index、Type、ID和版本号。Elasticsearch中每个Document都有一个版本号，而每次对该Document进行修改（包括更新、插入和删除）都会让版本号加1。created项的值为true说明该Document是第一次创建。下面我们再插入两个Document：

curl -XPUT 'localhost:9200/employee/dev/2' -d '{

"firstname": "Si",

"lastname": "Li",

"age": 28,

"on\_board\_date": "2014-09-16",

"hometown": "Nanjing",

"school": "Beijing University",

"married": true,

"about": "cooking, mountain climbing"

}'

curl -XPUT 'localhost:9200/employee/dev/3' -d '{

"firstname": "Wu",

"lastname": "Wang",

"age": 24,

"on\_board\_date": "2016-01-05",

"hometown": "Shanghai",

"school": "Fudan University",

"married": false,

"about": "My favorite writer is Mo Yan."

}'

curl -XPUT 'localhost:9200/employee/sales/4' -d '{

"firstname": "Qui",

"lastname": "Li",

"age": 35,

"on\_board\_date": "1205-09-16",

"hometown": "unknown",

"school": "Home Schooled",

"married": false,

"about": "I am very strong."

}'

### 获取整个Document

要获取一个完整的Document，使用GET并指定Document的路径，路径的形式为/<index>/<type>/<id>：

curl -XGET 'localhost:9200/employee/dev/1?pretty'

这里，我们用?pretty指定JSON格式化输出。输出如下：

{

"\_index" : "employee",

"\_type" : "dev",

"\_id" : "1",

"\_version" : 1,

"found" : true,

"\_source":{

"firstname": "San",

"lastname": "Zhang",

"age": 26,

"on\_board\_date": "2015-10-31",

"hometown": "Beijing",

"school": "Nanjing University",

"married": false,

"about": "I love Beijing Opera"

}}

我们看到，GET的结果包含了/employee/dev/1的元数据以及一个\_source字段，\_source字段中存储的是Document内部的信息。

### 获取部分Document

默认情况下，GET会打印Document完整的\_source。在GET时加上\_source参数可以获取Document中的指定字段，如果一次指定多个字段，字段名称之间用“,”隔开：

curl -XGET 'localhost:9200/employee/dev/1?\_source=firstname,lastname,age&pretty'

Elasticsearch会只输出name和age字段：

{

"\_index" : "employee",

"\_type" : "dev",

"\_id" : "1",

"\_version" : 1,

"found" : true,

"\_source":{"age":26,"lastname":"Zhang","firstname":"San"}

}

如果不想Elasticsearch输出元数据，可以将\_source作为一个endpoint使用：

curl -XGET 'localhost:9200/employee/dev/1/\_source?pretty'

Elasticsearch将只输出\_source中的数据：

{

"firstname": "San",

"lastname": "Zhang",

"age": 26,

"on\_board\_date": "2015-10-31",

"hometown": "Beijing",

"school": "Nanjing University",

"married": false,

"about": "I love Beijing Opera"

}

### 查看Document是否存在

查看某个路径下的Document是否存在使用HEAD，curl要加上-i选项打印HTTP header：

curl -i -XHEAD 'localhost:9200/employee/dev/1'

输出为：

HTTP/1.1 200 OK

Content-Type: text/plain; charset=UTF-8

Content-Length: 0

### 删除Document

删除指定路径下的Document使用XDELETE：删除/employee/dev/1下的Document

curl -XDELETE 'localhost:9200/employee/dev/1?pretty'

Elasticsearch会输出下面信息，说明删除成功：

{

"found":true,

"\_index":"employee",

"\_type":"dev",

"\_id":"1",

"\_version":2

}

注意，删除操作让版本号变为了2。我们可以再一次用XHEAD查看/employee/dev/1下是否存在Document，Elasticsearch会返回404 Not Found：

curl -i -XHEAD 'localhost:9200/employee/dev/1'

HTTP/1.1 404 Not Found

Content-Type: text/plain; charset=UTF-8

Content-Length: 0

### 更新Document

Elasticsearch中的Document不可更改，只能更新。操作和编入Document相同——对想要更新的Document路径

执行PUT，Elasticsearch会将路径下的Document更新并对版本号加1。下面我们对/employee/dev/2下的Document进行更新（将age从原来的28改为30）：

curl -XPUT 'localhost:9200/employee/dev/2' -d '{

"firstname": "Si",

"lastname": "Li",

"age": 30,

"on\_board\_date": "2014-09-16",

"hometown": "Nanjing",

"school": "Beijing University",

"married": true,

"about": "cooking, mountain climbing"

}'

Elasticsearch会输出下面信息，注意版本号变成了2。另外，created值为false说明这次PUT并没有新建该Document。

{

"\_index" : "employee",

"\_type" : "dev",

"\_id" : "2",

"\_version" : 2,

"created" : false

}

### 新建一个Document

我们看到PUT可能新建一个Document（例如编入Document中介绍的），也可能覆盖原有的Document（例如更新Document中介绍的）。那么如何确保我们在PUT的时候不会覆盖已有的Document呢？ 特定的/<index>/<type>下的Document靠它们的ID区分，所以我们需要确保新增的Document的ID是唯一的即可，使用POST可以让Elasticsearch为新增的Document自动生成一个唯一的ID：向/employee/sales下编入一个新Document

curl -XPOST 'localhost:9200/employee/sales?pretty' -d '{

"firstname": "Lei",

"lastname": "Li",

"age": 28,

"on\_board\_date": "2013-10-03",

"hometown": "Hangzhou",

"school": "Zhejiang University",

"married": true,

"about": "I appear in your English textbook."

}'

注意，使用POST只需给出/<index>/<type>/。Elasticsearch的输出如下：

{

"\_index" : "employee",

"\_type" : "sales",

"\_id" : "aKWYjab5Se-nt7gyYNJsGg",

"\_version" : 1,

"created" : true

}

### 轻量检索

在获取整个Document和获取部分Document中，我们看到GET可以获取Document信息。使用GET是将\_search作为endpoint可以进行检索。下面我们执行一个最简单的检索：  
curl -XGET 'localhost:9200/employee/dev/\_search?pretty'  
这个检索指令会将/employee/dev/下所有的Document返回。轻量检索指通过在GET请求的URI中直接提供检索串(query-string)来进行检索，可以用于快速简单的检索请求。例如：在employee Index中查找firstname为Si的Document  
curl -XGET 'localhost:9200/employee/\_search?pretty&q=firstname:Si'  
Elasticsearch的输出为：

{  
"took" : 13,  
"timed\_out" : false,  
"\_shards" : {  
"total" : 10,  
"successful" : 10,  
"failed" : 0  
},  
"hits" : {  
"total" : 1,  
"max\_score" : 0.30685282,  
"hits" : [ {  
"\_index" : "employee",  
"\_type" : "dev",  
"\_id" : "2",  
"\_score" : 0.30685282,  
"\_source":{  
"firstname": "Si",  
"lastname": "Li",  
"age": 30,  
"on\_board\_date": "2014-09-16",  
"hometown": "Nanjing",  
"school": "Beijing University",  
"married": true,  
"about": "cooking, mountain climbing"  
}  
} ]  
}  
}  
用类似的请求我们可以检索/employee/dev下age为26的员工信息：检索age为26的员工信息  
curl -XGET 'localhost:9200/employee/dev/\_search?pretty&q=age:26'  
Elasticsearch的输出为：  
{  
"took" : 17,  
"timed\_out" : false,  
"\_shards" : {  
"total" : 10,  
"successful" : 10,  
"failed" : 0  
},  
"hits" : {  
"total" : 1,  
"max\_score" : 1.0,  
"hits" : [ {  
"\_index" : "employee",  
"\_type" : "dev",  
"\_id" : "1",  
"\_score" : 1.0,  
"\_source":{  
"firstname": "San",  
"lastname": "Zhang",  
"age": 26,  
"on\_board\_date": "2015-10-31",  
"hometown": "Beijing",  
"school": "Nanjing University",  
"married": false,  
"about": "I love Beijing Opera"  
}  
} ]  
}  
}

## URI检索

在URI检索中，我们通过直接将检索参数包含在URI中发出检索请求。URI检索的功能较少，适用于快速的即席检索。它的格式为：

curl -XGET  
'<HOST>:9200/<PATH>/\_search?q=<QUERY\_STRING>[&<PARAMTERS>=<VALUE>&<PARAMTERS>=<VALUE>...]'  
其中，q参数用于接收查询(<QUERY\_STRING>),<QUERY\_STRING>需要符合一定的书写规范。URI中还可以包含一些其他参数用于控制Elasticsearch输出的形式。其中较为常见的有：  
 • df：<QUERY\_STRING>中不指定查询字段时默认使用的查询字段。它的默认值为\_all字段。  
 • analyzer：用于指定为<QUERY\_STRING>分词的分词器  
 • explain：为每一个匹配的Document解释该Document的关联分\_score是如何计算的  
 • default\_operator：默认的运算符，可以设为AND或OR。它的默认值为 OR  
 • \_source：可以通过将值设为false做到不获取Document的\_source字段  
 • \_source\_include：指定获取某个或某几个字段  
 • \_source\_exclude：指定不获取某个或某几个字段  
 • sort：指定排序方式。写法为sort=<fieldName>:<asc|desc>,<fieldName>可以是Document中实际的字段名，也可以是\_score字段。sort可以接受多个参数，参数间用“,”隔开。排序键出现的顺序会影响性能  
 • timeout：超时时间。默认为没有超时时间。设置后，Elasticsearch将在 timeout内返回已查询到的结果  
 • from：指定从第几个结果开始输出，默认值为0  
 • size：指定输出多少个匹配结果，默认值为10  
 • lowercase\_expanded\_terms：设置是否将查询中的词项(term)全部变为小写，默认为true  
 • analyze\_wildcard：是否为wildcard query和prefix query分词，默认为 false  
用df指定默认查询字段  
curl -XGET 'localhost:9200/employee/\_search?q=Beijing&df=school'

将返回结果排序  
curl -XGET 'localhost:9200/employee/\_search?q=\*&sort=lastname:asc&pretty'

### 输出结果分页

默认情况下，一次查询返回前十条匹配结果。使用from和size参数可以用于对返回结果分页。

输出前2条结果

curl -XGET 'localhost:9200/employee/\_search?q=\*&size=2'

从第3条起输出结果

curl -XGET 'localhost:9200/employee/\_search?q=\*&from=3'

从第3条起输出2条结果

curl -XGET 'localhost:9200/employee/\_search?q=\*&from=3&size=2'

### Query String语法

#### 检索字段的指定

检索时，可以用<fieldName>:<value>的方式指定在哪个字段内搜索value。

在指定字段中检索，下面的查询返回employee Index中name字段包含Li的Document：

curl -XGET 'localhost:9200/employee/\_search?pretty&q=name:li'

#### 逻辑运算符

在同一个字段中指定多个value时，value之间可以用AND或OR运算符连接。一个Document的指定字段中必须包含 所有AND连接的value才能算匹配；一个Document的字段中只要包含OR连接的value中的一个就能算匹配。另外，AND和OR也可以用于连接多个<fieldName>:<value>条件。条件之间用AND连接时，Document必须满足全部条件才能算匹配；条件之间用OR连接时，Document只需满足条件之一便能算匹配。不指定运算符则代表value或 <fieldName>:<value> 条件之间用默认运算符连接。例如：

• hometown:(Nanjing OR Beijing)

• school:(Beijing AND University)

• school:(Beijing University)

• lastname:li AND school:beijing

使用OR运算符连接value

curl -XGET 'localhost:9200/employee/\_search?q=hometown:(Nanjing%20OR%20Beijing)'

使用AND运算符连接value

curl -XGET 'localhost:9200/employee/\_search?q=school:(Beijing%20AND%20University)'

不指定运算符连接value

curl -XGET 'localhost:9200/employee/\_search?q=school:(Beijing%20University)'

使用AND运算符连接<fieldName>:<value>条件

curl -XGET 'localhost:9200/employee/\_search?q=(lastname:li)%20AND%20(school:beijing)'

#### 偏好运算符

默认情况下，fieldName对应的所有value都是可选的，例如school字段只要包含beijing或university便可匹配school:beijing university。我们在逻辑运算符中看到，用AND运算符可以指定哪些value是必选的。Elasticsearch还提供两个偏好运算符：+（必须匹配）和-（必须不匹配）。前面没有偏好运算符的value则为可选项——它们为Document的关联性加分。

• school:(-university)

• school:+nanjing+beijing

-运算符

curl -XGET 'localhost:9200/employee/\_search?q=school:(-university)'

+运算符

curl -XGET 'localhost:9200/employee/\_search?q=school:+nanjing+beijing'

#### 括号的使用

可以通过使用括号为<QUERY\_STRING>分组，使表达更清晰。括号还可以用于写子查询，例如school=(beijing AND university) OR (nanjing AND university)。

curl -XGET 'localhost:9200/employee/\_search?q=school=(beijing%20AND%20university)%20OR%20(nanjing%20AND%20university)'

#### 通配符

<QUERY\_STRING>中支持使用通配符：?匹配任意一个字符，\*匹配任意多个字符。例如：在URI检索中使用通配符

curl -XGET 'localhost:9200/employee/\_search?q=??d\*'

#### 正则表达式

<QUERY\_STRING>中支持使用正则表达式。正则表达式需要放在/中，例如：/.\*y/。

curl -XGET 'localhost:9200/employee/\_search?q=/.\*y/'

#### 范围查询

对date类型、数值类型和string类型的字段查询时可以用>，>=，<，<=来在指定的范围内进行匹配，例如on\_board\_date:>2014-01-01 AND <2015-12-31。

curl -XGET 'localhost:9200/employee/\_search?q=on\_board\_date:(>2014-01-01%20AND%20<2015-12-31)'

#### 保留字符

如果需要在<QUERY\_STRING>中包含以下Elasticsearch保留字符，您需要使用\进行转译：

+ - = && || > < ! ( ) { } [ ] ^ " ~ \* ? : \ /

#### 复杂查询

当想要进行一些复杂的查询时，<QUERY\_STRING>会变得非常难以理解和维护，同时会非常容易出错。所以我们建议只用URI检索进行非常简单的查询。

## Request Body检索

使用Request Body检索时，检索参数会通过HTTP Request Body传递：

curl -XPOST '<HOST>:9200/[<PATH>]/\_search?<PARAMETERS>' -d '{ ①

"query" : {<BODY>} ②

"<SEARCH\_PARAMETER>" : {<VALUE>} ③

}'

① 这里的HTTP方法可以是POST也可以是GET，使用方法没有区别。有些客户端不支持GET和Request Body合用，这时就要使用POST

② 查询通过<BODY>来表达。<BODY>的书写使用Elasticsearch的Query DSL语法

③ Request Body中还可以包含其他的检索参数，例如排序参数sort

Request body检索相对URI检索表达能力、可读性都更强。URI检索能做到的查询Request body检索都能做到

### Query DSL

Query DSL是用于构造查询体<BODY>的语言。本节我们介绍Query DSL查询体的结构。为了表述简洁，除非另外指出，本节给出的语法和示例都默认包含在BODY中。在实际操作中，您应当将curl指令以及Request Body的其他部分补齐。例如，当我们下面查询时：

{

"match": {

"lastname": "li"

}

}

实际操作中需要执行：

curl -XPOST '<HOST>:9200/[<PATH>]/\_search?<PARAMETERS>' -d '{

"query" : {

"match": {

"lastname": "li"

}

}

}'

### 查询语句

查询体由查询语句构成，Query DSL中的语句一般有如下形式：

{

"<QUERY\_NAME>": { ①

"<ARGUMENT>": <VALUE>, ②

"<ARGUMENT>": <VALUE>,

...

}

}

1. <QUERY\_NAME>为查询语句的种类，见常见Elasticsearch查询语句

② <ARGUMENT>为该语句接受的一些参数

如果查询语句使用字段名，那么参数要放在字段名下一级：

{

"<QUERY\_NAME>": {

"<FIELD\_NAME>": {

"<ARGUMENT>": <VALUE>,

"<ARGUMENT>": <VALUE>,...

}

}

}

### 查询语句的组合和嵌套

一个查询中可以包含多个查询语句，语句还可以互相嵌套。如果把一个查询看做一个抽象语法树(AST)，那么一个查询中有两类语句：  
 • 叶语句：语句中将一个或多个字段和一个查询串做比较。叶语句中不嵌套其他语句，例如：  
{  
"match": {  
"lastname": "li"  
}  
}  
 • 复合语句：嵌套了其他语句的语句。例如：  
{  
"bool" : {  
"must" : {  
"term" : { "lastname" : "li" }  
},  
"must\_not" : {  
"range" : {  
"on\_board\_date" : { "from" : 2014-01-01, "to" : 2015-12-31 }  
}  
}  
}  
Elasticsearch的语句可以多层嵌套，用于构造非常复杂的查询。

### Query和Filter

Elasticsearch中的查询语句分两种：query和filter。一些同名查询语句既是query也是filter，例如term，terms等等。同一查询语句作为query和filter时使用形式相似，但是表现不同：  
 • 作为query，查询语句回答的问题是“查询串和字段的匹配程度如何”，例如找出和I love Beijing Opera最匹配的Document，所以query会计算关联分 \*\_score\*来衡量Document的关联度。query常用于做全文检索  
 • 作为filter，查询语句回答的问题是“查询串是否和字段匹配”，所以filter的回答是二元的：是或不是。filter不计算关联分。下面是两个filter回答的问题：  
 • 是否在2014-01-01和2015-12-31之间入职（on\_board\_date字段是否在 2014-01-01 和\*2015-12-31\*之间）？  
 • Status是否是active（status\_active字段是否为true）？

filter用于做精确匹配查询时，"query" 对应的查询语句为query：  
"query": {  
查询语句在这里为query  
}  
filter对应的查询语句为filter  
"filter":{  
查询语句在这里为filter  
}  
同一个查询中，filter和query可以互相结合。要结合filter和query，filter和query都要放在filtered查询  
{  
"filtered" : {  
"query" : { ... },  
"filter" : { ... }  
}  
}  
{  
"filtered" :{  
"query" : {  
match : {"lastname" : "li"}  
},  
"filter" : {  
term : { "married" : true }  
}  
}  
}

## 映射与分词基础

映射(mapping)机制用于进行字段类型确认，将每个字段匹配为一种确定的数据类型。

分词(analysis)机制用于进行全文文本(Full Text)的分词，以建立供搜索用的反向索引。

### 数据类型差异

当在索引中处理数据时，我们注意到一些奇怪的事，有些东西似乎被破坏了。下面Index中有12个tweets，只有一个包含日期2014-09-15，但是我们看看查询中的total hits：

GET /\_search?q=2014 # 12 results

GET /\_search?q=2014-09-15 # 12 results !

GET /\_search?q=date:2014-09-15 # 1 result

GET /\_search?q=date:2014 # 0 results !

为什么全日期的查询返回所有的tweets，而针对date字段进行年度查询却什么都不返回？为什么我们的结果会因为查询\_all字段（默认所有字段中进行查询）或date字段而变得不同？可以想到，这是因为我们的数据在\_all字段的索引方式和在date字段的索引方式不同而导致。让我们看看Elasticsearch在对gb索引中的tweet类型进行映射后是如何解读我们的文档结构的：

GET /gb/\_mapping/tweet

返回：

{

"gb": {

"mappings": {

"tweet": {

"properties": {

"date": {

"type": "date",

"format": "strict\_date\_optional\_time||epoch\_millis"

},

"name": {

"type": "string"

},

"tweet": {

"type": "string"

},

"user\_id": {

"type": "long"

}

}

}

}

}

}

Elasticsearch为对字段类型进行猜测，动态生成了字段和类型的映射(mapping)关系。返回的信息显示了date字段被识别为date类型。\_all因为是默认字段所以没有在此显示，不过我们知道它是string类型。date类型的字段和string类型的字段的索引方式是不同的，因此导致查询结果的不同。你可能会期望每一种核心数据类型（strings, numbers, booleans及dates）以不同的方式进行索引，事实也正是这样：在Elasticsearch中他们是被区别对待的。

但是更大的区别在于确切值(exact values)（比如string类型）及全文文本(full text)之间。这两者的区别才真的很重要——这是区分搜索引擎和其他数据库的关键所在。

### 确切值(Exact values) vs.全文文本(Full text)

Elasticsearch中的数据可以大致分为两种类型：确切值(exact values)及全文文本(full text)确切值是确定的，正如它的名字一样。比如一个date或用户ID，也可以包含更多的字符串比如username或email地址。确切值"Foo"和"foo"就并不相同。确切值2014和2014-09-15也不相同。全文文本，从另一个角度来说是文本化的数据（常常以人类的语言书写），比如一篇Twitter文章或邮件正文。

确切值是很容易查询的，因为结果是二进制的——要么匹配，要么不匹配。这种查询很容易以SQL表达：

WHERE name = "John Smith"

AND user\_id = 2

AND date > "2014-09-15"

而对于全文数据的查询来说，却有些微妙。我们不会去询问这篇文档是否匹配查询要求 ，而会询问这篇文档和查询的匹配程度如何？也就是说，对于查询条件，这篇文档的相关性有多高？

我们很少确切地匹配整个全文文本。我们想在全文中查询包含查询文本的部分。不仅如此，我们还期望搜索引擎能理解我们的意图：

• 一个针对UK的查询将返回涉及United Kingdom的文档

• 一个针对jump的查询同时能够匹配jumped, jumps, jumping甚至leap

• johnny walker也能匹配 Johnnie Walker, johnnie depp及Johnny Depp

• fox news hunting能返回有关hunting on Fox News的故事，而fox hunting news也能返回关于fox hunting的新闻故事。

为了方便在全文文本字段中进行这些类型的查询，Elasticsearch首先对文本分词(analyze)，然后使用结果建立一个倒排索引(inverted index)。我们下面讨论倒排索引及分词过程。

### 倒排索引(inverted index)

Elasticsearch使用一种叫做倒排索引(inverted index)的结构来做快速的全文搜索。倒排索引由在文档中出现的唯一的单词列表，以及对于每个单词在文档中的位置组成。

例如，我们有两个文档，每个文档content字段包含：

1. The quick brown fox jumped over the lazy dog

2. Quick brown foxes leap over lazy dogs in summer

为了创建倒排索引，我们首先切分每个文档的content字段为单独的单词（我们把它们叫做terms或者tokens），把所有的唯一词放入列表并排序，结果是这个样子的：

Term Doc\_1 Doc\_2

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

Quick | | X

The | X |

brown | X | X

dog | X |

dogs | | X

fox | X |

foxes | | X

in | | X

jumped | X |

lazy | X | X

leap | | X

over | X | X

quick | X |

summer | | X

the | X |

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

现在，如果我们想搜索quick brown，我们只需要找到每个词在哪个文档中出现即可：

Term Doc\_1 Doc\_2

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

brown | X | X

quick | X |

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

Total | 2 | 1

两个文档都匹配，但是第一个比第二个有更多的匹配项。如果我们加入简单的相似度算法(similarity algorithm)，计算匹配单词的数目，这样我们就可以说第一个文档比第二个匹配度更高——对于我们的查询具有更多相关性。

但是在我们的倒排索引中还有些问题：

• Quick和quick被认为是不同的单词，但是用户可能认为它们是相同的

• fox和foxes很相似——它们都是同根词

• jumped和leap不是同根词，但意思相似——它们是同义词

上面的索引中，搜索Quick +fox\*不会匹配任何文档（记住，前缀\*表示单词必须匹配到）。只有Quick和fox都在同一文档中才可以匹配查询。如何才能合理地让两个文档都匹配查询？

如果我们将词为统一为标准格式，这样就可以找到不是确切匹配查询，但是足以相似，从而可以关联的文档。例如：

• Quick可以转为小写成为quick

• foxes可以被转为根形式fox；同理dogs可以被转为dog

• jumped和leap同义就可以只索引为单个词jump

现在的索引：

Term Doc\_1 Doc\_2

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

brown | X | X

dog | X | X

fox | X | X

in | | X

jump | X | X

lazy | X | X

over | X | X

quick | X | X

summer | | X

the | X | X

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

但这样还不够。我们的搜索+Quick +fox依旧失败，因为Quick的确切值已经不在索引里。如果我们使用相同的标准化规则处理查询字符串的content字段，查询将变成+quick +fox，这样就可以匹配到两个文档。

### 分词与分词器

分词是这样一个过程：首先，将一个文本块拆分为单独的词(term)，然后标准化这些词为标准形式，提高它们的“可搜索性”或“查全率”，而后构建倒排索引。而这些工作就是由分词器完成的。

#### 内建的分词器

Elasticsearch附带了一些预装的分词器，你可以直接使用它们。下面我们列出了重要的几个分词器，来演示这个字符串分词后的表现差异：  
Set the shape to semi-transparent by calling set\_trans(5)  
 • 标准分词器(Standard analyzer)  
 标准分词器是Elasticsearch默认使用的分词器。对于文本分词，如果没有什么特殊要求，它是够用的。它根据Unicode Consortium的定义的单词边界(word boundaries)来切分文本，然后去掉大部分标点符号。最后，把所有词转为小写。产生的结果为：  
set, the, shape, to, semi, transparent, by, calling, set\_trans, 5  
 • 简单分词器(Simple analyzer)  
 简单分词器将非单个字母的文本切分，然后把每个词转为小写。产生的结果为：  
set, the, shape, to, semi, transparent, by, calling, set, trans  
 • 空格分词器(Whitespace analyzer)  
 空格分词器依据空格切分文本。它不转换小写。产生结果为：  
Set, the, shape, to, semi-transparent, by, calling, set\_trans(5)

• 语言分词器(Language analyzer)

特定语言分词器适用于很多语言。它们能够考虑到特定语言的特性。例如，english分词器自带一套英语停用词库——像and或the这些与语义无关的通用词。这些词被移除后，因为语法规则的存在，英语单词的主体含义依旧能被理解。english分词器将会产生以下结果：  
set, shape, semi, transpar, call, set\_tran, 5  
注意transparent、calling和set\_trans是如何转为词干的。

#### 分词器的使用

当我们索引(index)一个文档，全文字段会被分词为单独的词来创建倒排索引。不过，当我们在全文字段搜索(search)时，我们要让查询字符串经过同样的分词流程处理，以确保这些词在索引中存在。全文查询我们将在稍后讨论。只有理解每个字段是如何定义的，这样才可以得到想要的结果：  
 • 当你查询全文(full text)字段，查询将使用相同的分词器来将查询字符串分词，以产生正确的词列表  
 • 当你查询一个确切值(exact value)字段，查询不将查询字符串分词，但是你可以自己指定。现在你可以明白为什么映射和分词的开头会产生那种结果：  
 • date字段包含一个确切值：单独的一个词2014-09-15  
 • \_all字段是一个全文字段，所以分词过程将日期转为三个词：2014、09和15  
当我们在\_all字段查询2014，它一个匹配到12条推文，因为这些推文都包含词 2014：  
GET /\_search?q=2014

# 12 results  
当我们在\_all字段中查询2014-09-15，首先将查询字符串分词，产生匹配任一词 2014、09或15的  
查询语句，它依旧匹配12个推文，因为它们都包含词2014。  
GET /\_search?q=2014-09-15

# 12 results !  
当我们在date字段中查询2014-09-15，它查询一个确切的日期，然后只找到一条推文：  
GET /\_search?q=date:2014-09-15

# 1 result  
当我们在date字段中查询2014，没有找到文档，因为没有文档包含那个确切的日期：  
GET /\_search?q=date:2014

# 0 results !

#### 测试分词器

尤其当你是Elasticsearch新手时，对于如何分词以及存储到索引中理解起来比较困难。为了更好的理解如何进行，你可以使用analyze API来查看文本是如何被分词的。在查询字符串参数中指定要使用的分词器，被分词的文本作为请求体：

GET /\_analyze

{

"analyzer": "standard",

"text": "Text to analyze"

}

结果中每个节点在代表一个词：

{

"tokens": [

{

"token": "text",

"start\_offset": 0,

"end\_offset": 4,

"type": "<ALPHANUM>",

"position": 1

},

{

"token": "to",

"start\_offset": 5,

"end\_offset": 7,

"type": "<ALPHANUM>",

"position": 2

},

{

"token": "analyze",

"start\_offset": 8,

"end\_offset": 15,

"type": "<ALPHANUM>",

"position": 3

}

]

}

token是一个实际被存储在索引中的词；position指明词在原文本中是第几个出现的；start\_offset和end\_offset表示词在原文本中占据的位置。

#### 指定分词器

当Elasticsearch在你的文档中探测到一个新的字符串字段，它将自动设置它为全文string字段并用标准分词器分词。

你不可能总是想要这样做。也许你想使用一个更适合这个数据的语言分词器。或者，你只想把字符串字段当作一个普通的字段——不做任何分词，只存储确切值，就像字符串类型的用户ID或者内部状态字段或者标签。为了达到这种效果，我们必须通过mapping人工设置这些字段。

### 映射(mapping)

为了能够把日期字段处理成日期，把数字字段处理成数字，把字符串字段处理成全文本(Full-text)或精确的字符串值，Elasticsearch需要知道每个字段里面都包含了什么类型。这些类型和字段的信息存储在mapping中。

索引中每个文档都从属于一个类型(type),每个type拥有自己的mapping。一个mapping在type中定义了字段，每个字段的数据类型，以及字段被Elasticsearch处理的方式。Mapping还用于设置关联到类型上的元数据。

#### 核心简单字段类型

Elasticsearch支持以下简单字段类型：

• String: string

• Whole number: byte, short, integer, long

• Floating-point: float, double

• Boolean: boolean

• Date: date

#### 查看mapping

我们可以使用\_mapping后缀来查看Elasticsearch中的mapping。在本章开始我们已经找到索引gb类型tweet中的mapping：

GET /gb/\_mapping/tweet

这展示给了我们字段的mapping（叫做属性properties），这些mapping是Elasticsearch在创建索引时动态生成的：

{  
"gb": {  
"mappings": {  
"tweet": {  
"properties": {  
"date": {  
"type": "date",  
"format": "strict\_date\_optional\_time||epoch\_millis"  
},  
"name": {  
"type": "string"  
},  
"tweet": {  
"type": "string"  
},  
"user\_id": {  
"type": "long"  
}  
}  
}  
}  
}  
}

#### 自定义字段mapping

虽然大多数情况下基本数据类型已经能够满足，但你也会经常需要自定义一些特殊类型(fields)，特别是字符串字段类型。自定义类型可以使你完成一下几点：

• 区分全文字符串字段和准确字符串字段（即分词与不分词，全文的一般要分词，准确的就不需要分词）。

• 使用特定语言的分词器（中文、英文、阿拉伯语，不同文字的断字、断词方式的差异）

• 优化部分匹配字段

• 指定自定义日期格式（例如英文的Feb,12,2016和中文的2016年2月12日）

• 以及更多

mapping中最重要的字段参数是type。除了string类型的字段，你可能很少需要指定除type外的其他值：

{

"number\_of\_clicks": {

"type": "integer"

}

}

string 类型的字段，默认的，考虑到包含全文本，它们的值在索引前要经过分词器分词，并且在全文搜索此字段前要把查询语句做分词处理。

对于string字段，两个最重要的mapping参数是index和analyer。

•index

index参数控制字符串以何种方式被索引。它包含以下三个值当中的一个：

analyzed

首先将字符串分词，然后索引。换言之，以全文形式索引此字段。

not\_analyzed

索引这个字段，使之可以被搜索，但是索引内容和指定值一样。不对此字段分词。

no

不索引这个字段。这个字段不能为搜索到。

string 类型字段默认值是analyzed。如果我们想映射字段为确切值，我们需要设置它为not\_analyzed :

{

"tag": {

"type": "string",

"index": "not\_analyzed"

}

}

• analyzer

对于analyzed类型的字符串字段，使用analyzer参数来指定在搜索和索引的时候使用哪一种分词器。默认的是Elasticsearch使用标准分词器，但是你可以通过指定一个内建的分词器来更改它，例如whitespace、simple或english。

{

"tweet": {

"type": "string",

"analyzer": "english"

}

}

#### 更新mapping

你可以在第一次创建索引的时候指定mapping的类型。此外，你也可以之后为新类型添加mapping（或者为已有的类型更新mapping）。我们可以更新一个mapping来增加一个新字段，但是不能把已有字段的类型那个从analyzed改到not\_analyzed。为了演示两个指定的mapping方法，让我们首先删除索引gb：

DELETE /gb

然后创建一个新索引，指定tweet字段的分词器为english：

curl -XPUT 'localhost:9200/gb' -d '{

"mappings": {

"tweet" : {

"properties" : {

"tweet" : {

"type" : "string",

"analyzer": "english"

},

"date" : {

"type" : "date"

},

"name" : {

"type" : "string"

},

"user\_id" : {

"type" : "long"

}

}

}

}

}'

1. 这将创建包含mappings的索引，mapping在请求体中指定。

再后来，我们决定在tweet的mapping中增加一个新的not\_analyzed类型的文本字段，叫做tag，使用\_mapping后缀:

curl -XPUT 'localhost:9200/gb/\_mapping/tweet' -d '{

"properties" : {

"tag" : {

"type" : "string",

"index": "not\_analyzed"

}

}

}'

注意到我们不再需要列出所有的已经存在的字段，因为我们没法修改他们。我们的新字段已经被合并至存在的那个mapping中。

#### 测试mapping

你可以通过名字使用analyze API测试字符串字段的mapping。对比这两个请求的输出：  
curl -XGET 'localhost:9200/gb/\_analyze' -d '{

"field": "tweet",

"text": "Black-cats"

}'  
curl -XGET 'localhost:9200/gb/\_analyze' -d '{

"field": "tag",

"text": "Black-cats"

}'  
tweet字段产生两个词，black和cat, tag字段产生单独的一个词Black-cats。换言之，我们的mapping工作正常。

### 复合核心字段类型

除了之前提到的简单的标量类型，JSON还有null值，数组和对象，所有这些Elasticsearch都支持。

#### 多值字段

我们很可能想让 tag 字段包含多个字段。我们可以索引一个标签数组来代替单一字符串：{ "tag": [ "search", "nosql" ]}

对于数组不需要特殊的mapping。任何一个字段可以包含零个、一个或多个值，同样对于全文字段将被分词处理并产生多个词。言外之意，这意味着数组中所有值必须为同一类型。你不能把日期和字符串混合。如果你创建一个新字段，这个字段索引了一个数组，Elasticsearch将使用第一个值的类型来确定这个新字段的类型。

#### 空字段

当然数组可以是空的。这等价于有零个值。事实上，Lucene没法存放null值，所以一个null值的字段被认为是空字段。

这三个字段将被识别为空字段而不被索引：

"null\_value": null,

"empty\_array": [],

"array\_with\_null\_value": [ null ]

#### 多层对象

我们需要讨论的最后一个自然JSON数据类型是对象(object)——在其它语言中叫做hash、hashmap、dictionary或者associative array。

内部对象(inner objects)经常用于在另一个对象中嵌入一个实体或对象。例如，作为在tweet文档中user\_name和user\_id的替代，我们可以这样写：

{

"tweet": "Elasticsearch is very flexible",

"user": {

"id": "@johnsmith",

"gender": "male",

"age": 26,

"name": {

"full": "John Smith",

"first": "John",

"last": "Smith"

}

}

}

#### 内部对象的mapping

Elasticsearch会动态的检测新对象的字段，并且映射它们为object类型，将每个字段加到properties字段下：

{

"gb": {

"tweet": { ①

"properties": {

"tweet": { "type": "string" },

"user": { ②

"type": "object",

"properties": {

"id": { "type": "string" },

"gender": { "type": "string" },

"age": { "type": "long" },

"name": { ②

"type": "object",

"properties": {

"full": { "type": "string" },

"first": { "type": "string" },

"last": { "type": "string" }

}

}

}

}

}

}

}

}

1. 根对象

② 内部对象

对user和name字段的mapping与tweet类型自己很相似。事实上，type mapping只是object mapping的一种特殊类型，我们将object称为根对象。它与其他对象一模一样，除非它有一些特殊的顶层字段，比如\_source , \_all等等。

#### 内部对象是怎样被索引的

Lucene并不了解内部对象。一个Lucene文件包含一个键-值对应的扁平表单。为了让Elasticsearch可以有效的索引内部对象，将文件转换为以下格式：

{

"tweet": [elasticsearch, flexible, very],

"user.id": [@johnsmith],

"user.gender": [male],

"user.age": [26],

"user.name.full": [john, smith],

"user.name.first": [john],

"user.name.last": [smith]

}

内部栏位可被归类至name，如first。为了区别两个相同名字的栏位，可以使用完整路径，user.name.first，甚至类型名称加上路径：tweet.user.name.first。

#### 内部对象数组

最后，一个包含内部对象的数组如何索引。我们有个数组如下所示：

{

"followers": [

{ "age": 35, "name": "Mary White"},

{ "age": 26, "name": "Alex Jones"},

{ "age": 19, "name": "Lisa Smith"}

]

}

此文件会如我们以上所说的被扁平化，但其结果会像如此：

{

"followers.age": [19, 26, 35],

"followers.name": [alex, jones, lisa, smith, mary, white]

}

{age: 35}与{name: Mary White}之间的关联会消失，因每个多值的栏位会变成一个值集合，而非有序的阵列。这让我们可以知道：

• 是否有26岁的追随者？但我们无法取得准确的资料

• 是否有26岁的追随者且名字叫Alex Jones？关联内部对象可解决此类问题，我们称之为嵌套对象

## Mapping操作

Mapping定义了Elasticsearch如何对一个Index中的Document进行保存并建倒排索引。Mapping可以定义的内容包括：  
 • 各个字段的数据类型  
 • String类型字段应该做为全文本(full text)还是精确值(exact values)处理  
 • 是否保存某字段的原始数据  
 • 日期值的格式  
Mapping可以在创建Index时显式定义，Mapping也可以不显式定义。如果您不显式定义Mapping，Elasticsearch的动态Mapping功能会根据编入Index的Document的数据自动分辨各自段数据类型，并按照Elasticsearch的默认配置定义Mapping。然而Elasticsearch的默认配置并不总能满足您的检索需求，这时您需要对Mapping进行显式定义。Mapping的显式定义需要在编入Elasticsearch Index中的第一个Document之前完成，而且Elasticsearch Index中有数据后，建议不要修改Mapping。由于Mapping直接影响检索结果，无论是显式定义还是动态定义，我们都建议在检索前了解Index的Mapping。

### 显式定义Mapping

显式定义Mapping需要在创建Index时进行，格式如下：  
curl -XPUT '<HOST>:9200/<INDEX>?pretty' -d ' ①  
{  
"mappings" : {  
"default\_type\_" : {  
<META-FIELD\_MAPPING>, ②  
<META-FIELD\_MAPPING>,  
...  
"properties" : { ③  
<FIELD\_MAPPING>,  
<FIELD\_MAPPING>,  
...  
}  
}  
}  
}

① 创建名为<INDEX>的Index  
② Mapping中可以包含零到多个元数据字段的Mapping(Meta-field Mapping)。元数据字段都以下划线开头，例如\_source, \_all等。用于定义Index元数据字段的表现，详情请参考定义字段Mapping(META-FIELD\_MAPPING)  
③ properties描述Index中的各个Document字段的Mapping(Field Mapping)，每个<FIELD\_MAPPING>对应一个字段的Mapping。字段Mapping包含字段的数据类型、分词器等信息，详情请参考定义字段Mapping(FIELD\_MAPPING)  
下面是一个显式定义Mapping的示例：

curl -XPUT 'localhost:9200/sample\_index?pretty' -d '  
{  
"mappings" : {  
"default\_type\_" : {  
"\_all" : { ①  
"enabled" : false  
},  
"\_source" : { ②  
"enabled" : true  
},  
"properties" : {  
"d:C1" : { ③  
"type" : "string",  
"index" : "not\_analyzed",  
"store" : true  
},  
"d:C2" : { ④  
"type" : "long",  
"store" : true  
}  
}  
}  
}  
}'  
① \_all元数据字段  
② \_source元数据字段  
③ 名为d:C1的字段的Mapping。该字段类型为string("type" : "string")，不分词("index" :"not\_analyzed")并且存储原始数据("store" : true)  
④ 名为d:C2 的字段的Mapping。该字段类型为long("type" : "long")并且存储原始数据("store" :true)

### 定义元数据字段Mapping

Document不仅有字段(Field)，还有一系列元数据字段(Meta-Field)。Mapping可以控制Document元数据字段的行为。本节选取几个最重要的Document元数据字段，介绍它们的概念以及如何使用Mapping来修改其行为。  
Document中较常用的Meta-Field包括（但不限于）：  
 • \_id：Document的ID字符串  
 • \_type：Document所属Type  
 • \_index：Document所属的Index  
 • \_uid：\_type和\_id相连：“type#id”  
 • \_source：Document中的原始数据。默认状况下，\_source字段是存储的。但是\_source字段的存储会占用磁盘空间，所以在有需求的情况下，您可以通过下面的方式关闭\_source的存储：

{  
"mapping" : {  
"default\_type\_" : {  
"\_source" : { "enabled" : false }  
}  
}  
}'  
• \_all：Document所有字段中数据首尾相连形成的字段。例如：  
{  
"firstname": "Si",  
"lastname": "Li",  
"age": 30,  
"on\_board\_date": "2014-09-16",  
"hometown": "Nanjing",  
"school": "Beijing University",  
"married": true,  
"about": "cooking, mountain climbing"  
}  
的\_all字段为：  
Si Li 30 2014-09-16 Nanjing Beijing University true cooking mountain climbing  
在检索时如果不指定检索字段，Elasticsearch会在\_all字段中进行匹配。通过Mapping可以设置关闭\_all字段（Elasticsearch将不为Document生成\_all字段）：  
{  
"mapping" : {  
"default\_type\_" : {  
"\_all" : { "enabled" : false}  
}  
}  
}'  
Mapping还可以设置\_all字段的分词器：  
{  
"mapping" : {  
"default\_type\_" : {  
"\_all" : { "analyzer" : "<analyzerName>"}  
}  
}  
}

### 定义字段Mapping

我们已经在显式定义Mapping中见过字段Mapping。字段的Mapping定义一般具有如下格式：  
"<fieldName>" : {  
"<parameter>" : <value>,  
...  
}  
其中比较重要的参数有

• type：字段的数据类型。Elasticsearch支持的数据类型有：string, date, byte, short, integer, long, float, double和boolean。不显式定义Index Mapping，动态Mapping中会做根据Document字段中的内容自动做判断  
 • index：可以是analyzed、not\_analyzed或no。如果设为no，则该字段不能被查询。analyzed/not\_analyzed指定是否分词（仅对string类型字段有效，默认为analyzed）  
 • analyzer：字段使用的分词器（仅对string类型字段有效）  
 • store：可以是true或false。是否存储字段元数据  
 • doc\_values：可以是true或false。在Elasticsearch 2.0.0中默认值为true。如果该字段被用于排序或者聚合，我们推荐将doc\_values设置为true

"d:C5" : {  
"type" : "string", ①  
"store" : true, ②  
"index" : "analyzed", ③  
"analyzer" : "ik" ④  
}  
① 字段类型为string  
② 存储字段原始数据  
③ 为该字段分词  
④ 使用ik分词器

## 常见查询语句

Query DSL提供一系列查询语句，用于完成不同类型的检索。例如下面的match\_all查询匹配所有的Document，也就是做空检索：  
match\_all查询  
{  
"match\_all":{}  
}  
Elasticsearch中的查询语句分为三个大类：Full Text查询语句，Term-level查询语句和复合查询语句。每个大类下又分别有多种不同的查询语句。为了文档简洁明了，除非另外指出，本章的所有JSON文本都默认包含在查询体中。在实际操作中，您需要补上检索的其他元素，例如上面的match\_all查询对应的实际命令行操作应当为：  
curl -XPOST '<HOST>:9200/<PAHT>/\_search' -d '{  
"query": {  
"match\_all":{}  
}  
}'

### Full Text查询语句

全文检索查询语句用于文本字段，例如邮件内容。全文检索查询语句会使用被查询字段的分词器对查询串进行分词后再执行查询。全文检索查询语句中最常见的是match查询语句。match查询用于在指定的检索字段中检索文本、数值和日期类型的查询串。  
{  
"match" : {  
"<FIELD\_NAME>" : { ①  
"query" : "<QUERY\_STRING>", ②  
"<PARAMETER>" : <VALUE>, ③  
....  
}  
}  
}

① 指定检索字段 FIELD\_NAME  
② 提供检索串 QUERY\_STRING  
③ 设置一些可选参数

如果不需要设置可选参数，match查询可以简写成：  
{  
"match" : {  
"<FIELD\_NAME>" : "<QUERY\_STRING>"  
}  
}  
match查询是布尔型的，意思是Elasticsearch会先将<queryString>分词，然后用分词得到的词项构造一个bool查询，该bool查询的默认逻辑运算符为or，要修改逻辑运算符可以使用operator参数，如下：  
{  
"match" : {  
"<FIELD\_NAME>" : {  
"query" : "<QUERY\_STRING>",  
"operator" : "and"  
}  
}  
}  
match查询对queryString分词默认使用的分词器是fieldName的分词器。可以通过analyzer手动设置分词器：  
{  
"match" : {  
"<FIELD\_NAME>" : {  
"query" : "<QUERY\_STRING>",  
"analyzer" : "<ANALYZER\_NAME>"  
}  
}  
}  
查询about字段中有  
{  
"match":{  
"about":"beijing is great"  
}  
}

### Term-level查询语句

不像全文检索查询语句会对查询串分词，Term级别查询语句不对查询串分词，而是将其和倒排索引中的数据进行 精确匹配。Term级别的查询语句通常用于检索结构化数据，例如数字、日期、布尔型数据等等。

#### term查询语句

term 查询语句的形式为：  
{  
"term" : { "<FIELD\_NAME>" : <TERM>}  
}  
term查询返回FIELD\_NAME字段包含和TERM完全相同词项的Document。

{  
"term" : {  
"married" : true  
}  
}

#### terms查询语句

terms 查询可以在指定字段中精确匹配多个词项，形式如下：  
{  
"terms" : {  
"<FIELD\_NAME>" : [<TERM1>, <TERM2>, ...] ①  
}  
}

① 用数组指定要匹配的多个词项。

{  
"terms" : {  
"school" : ["nanjing", "beijing"]  
}  
}

#### range查询语句

range 查询返回字段中词项在指定范围内的Document。

{

"range" : {

"<FIELD\_NAME>" : {

"gt"|"gte" : <LOWER\_BOUND>, ①

"lt"|"lte" : <UPPER\_BOUND> ②

}

}

}

1. 指定范围的下限，gt表示大于，gte表示大于等于

② 指定范围的上限，lt表示小于，lte表示小于等于

{

"range" : {

"on\_board\_date" : {

"gt" : "2013-12-31",

"lte" : "2015-12-31"

}

}

}

#### wildcard查询语句

wildcard 查询允许在查询串中使用通配符\*和?。\\*用于匹配任意多个任意字符（包括0个字符）；?用于匹配任意的单个字符。

{

"wildcard" : { "school" : "bei\*"}

}

### 复合查询语句

复合查询语句有其他的查询语句嵌套在内。复合查询语句可以用于叠加不同语句产生的关联分、改变语句的表现或者在查询语境和过滤语境之间转换。

#### bool查询语句

bool查询由逻辑运算符连接的bool子句构成，每个子句都有成立类型(occurrence type)，包括以下几种：

• must：该子句必须成立，并且它的成立会为\_score加分

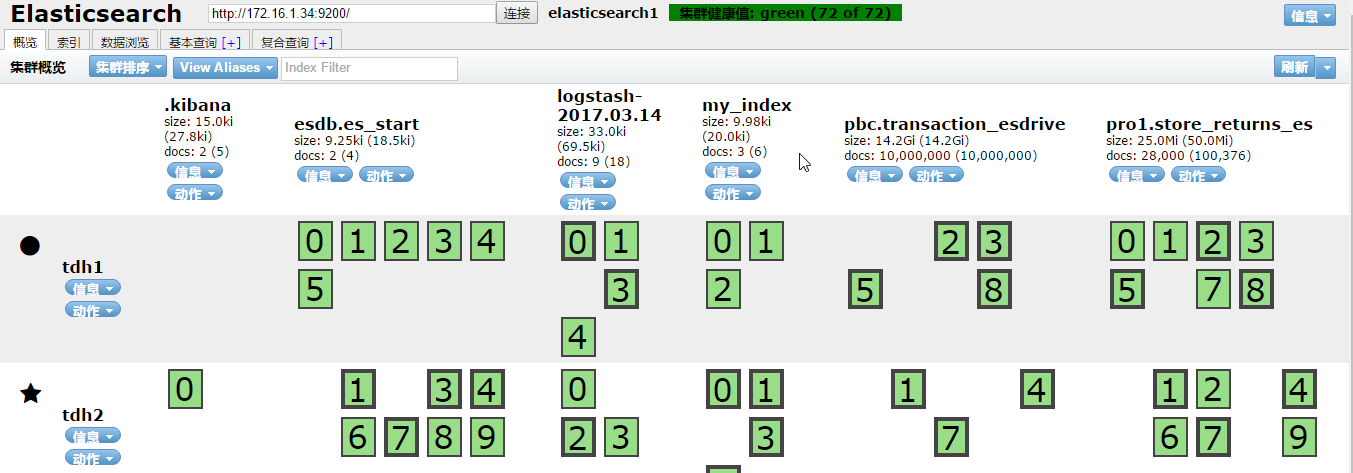
• filter：该子句必须成立，但是它的成立不会为\_score加分  
 • should：该子句应该成立，并且它的成立会为\_score加分。如果一个bool查询中没有must或filter子句，至少一个should子句应该有匹配的Document，您也可以通过minimum\_should\_match参数来设置至少有几个should子句应该有匹配的Document  
 • must\_not：该子句必须不成立  
bool查询的一般形式如下：  
{  
"bool" : {  
"must" : {  
<QUERY>  
},  
"filter" : {  
<QUERY>  
},  
"should" : {  
<QUERY>  
},  
"must\_not" : {  
<QUERY>  
},  
"minimum\_should\_match" : <n>  
}  
}  
must、should、filter和must\_not中有多个查询语句时，用数组表示：  
{  
"bool" : {  
“must” : [{<QUERY>}, {<QUERY>}, ...],  
“filter”: [{<QUERY>}, {<QUERY>}, ...],  
"should": [{<QUERY>}, {<QUERY>}, ...],  
"must\_not" : [{<QUERY>}, {<QUERY>}, ...],  
"minimum\_should\_match" : <n>  
}  
}  
对于bool查询来说，匹配越多越好，所以should和must子句为同一个Document贡献的关联分会被相加算进Document的\_score。  
{  
"bool": {  
"must": {  
"term": {"\_all": "beijing"}  
},  
"should": {  
"term": {"about": "opera"}  
},  
"must\_not": {  
"range": {  
"on\_board\_date":{"gte": "2013-01-01","lte": "2014-12-31" }  
}  
},  
"filter" : {  
"married" : true  
}  
}  
}

# 场景方案实现

## ELK搭建

Elasticsearch安装：

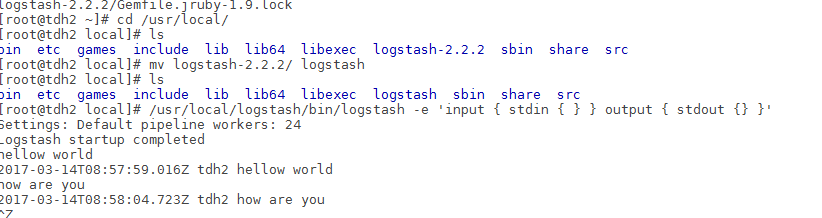
安装TDH，安装Elasticsearch，以172.16.1.34节点为例，浏览器进<http://172.16.1.34:9200/_plugin/head/>看到界面，代表安装正常。



Logstash安装：

下载logstash-2.2.2.tar.gz

tar zxvf logstash-2.2.2.tar.gz -C /usr/local/

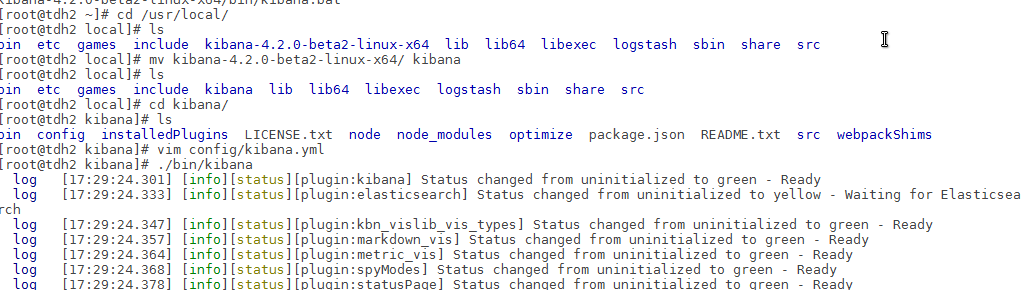


出现，Setting：Default pipeline workers代表正常，然后可以输入数据。

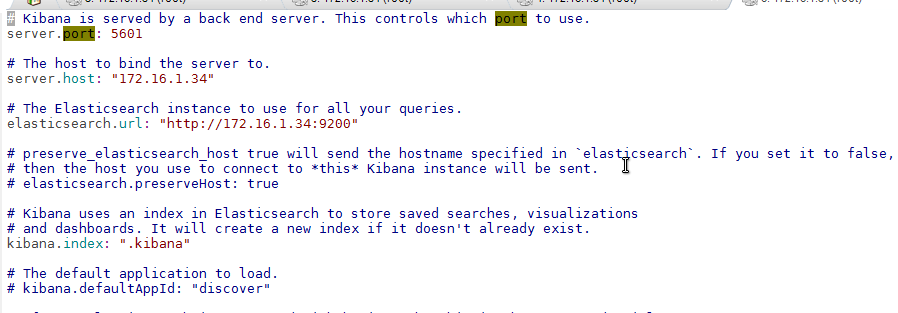
Kibana安装：

下载kibana-4.2.0-beta2-linux-x64.tar.gz

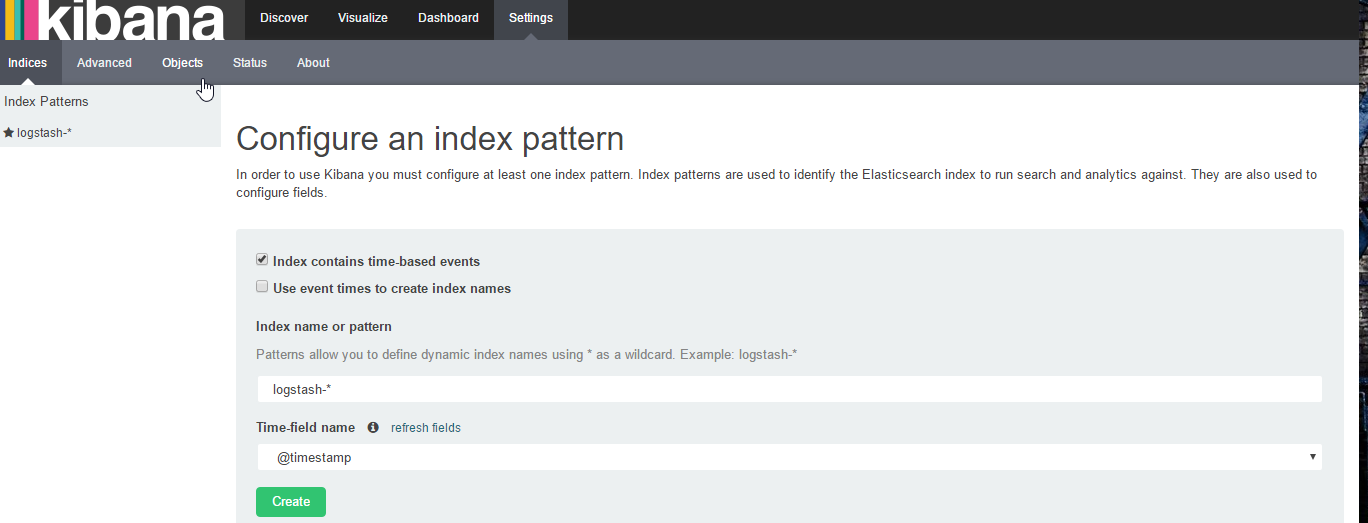
tar zxvf kibana-4.2.0-beta2-linux-x64.tar.gz -C /usr/local



其中配置kibana.yml中，注意高亮的四处取消注释并修改



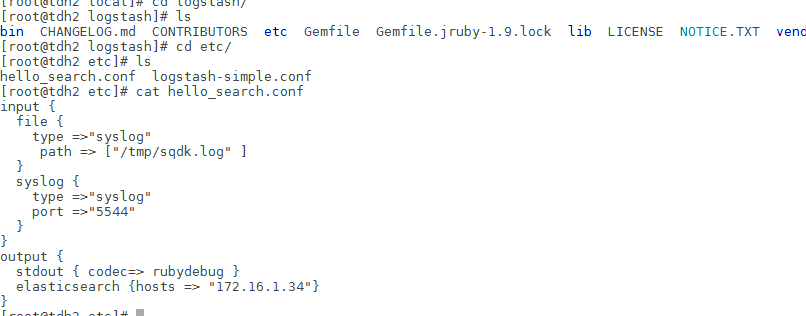
在启动kibana时，浏览器进<http://172.16.1.34:5601/app/kibana>



默认create，kibana安装成功

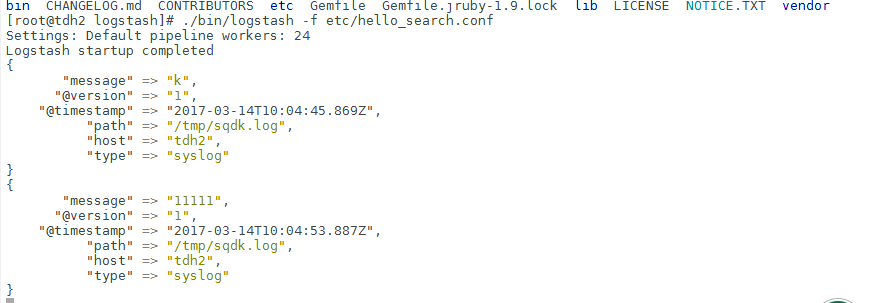
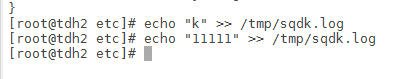
ELK实战读取日志文件：

在Logstash中新建conf文件

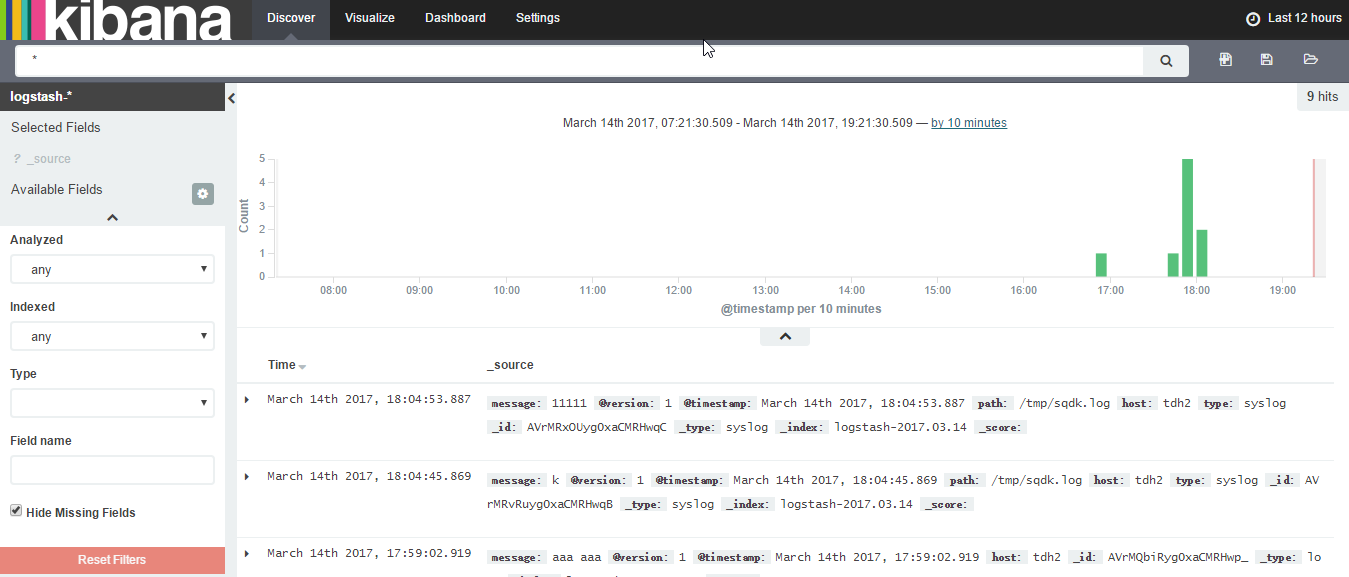


新建/tmp/sqdk.log文件

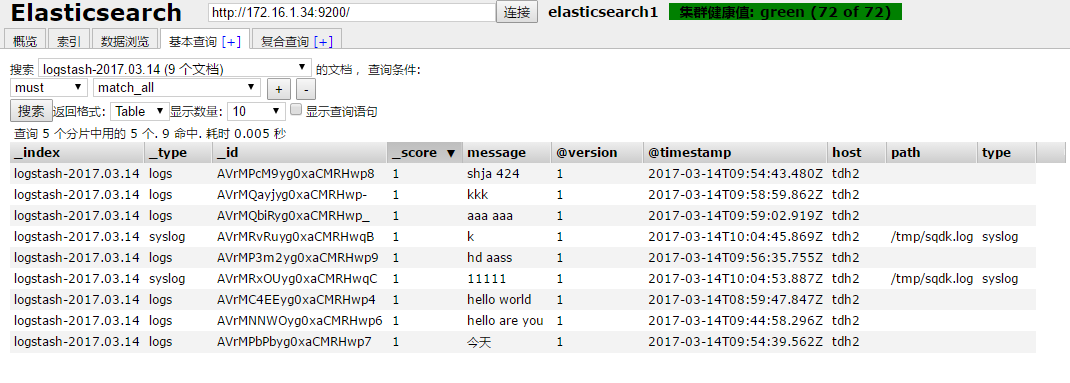
启动Logstash，并往sqdk.log中插入数据



然后在Kibana界面中的Discover中



看到插入的数据，同时在Elasticsearch的界面中能够看到Logstash中的数据



# 源码编译

**编译版本信息**

Elasticsearch版本：2.0.0

**编译环境信息**

操作系统：Win10

Maven版本：3.39

**编译步骤**

下载源码<https://github.com/elastic/elasticsearch/tree/v2.0.0>，解压后进入目录，执行命令mvn clean package -DskipTests。

**验证方法**

各个子目录的target文件夹生成对应的jar包。

# 对应UML图

org包UML：



org.apache.lucene包UML：



org.elasticsearch包UML：



org.elasticsearch.action包UML：



org.elasticsearch.bootstrap包UML：



org.elasticsearch.client包UML：



org.elasticsearch.cluster包UML：



org.elasticsearch.common包UML：



org.elasticsearch.index包UML：



org.elasticsearch.indices包UML：



org.elasticsearch.node包UML：



org.elasticsearch.search包UML：



org.elasticsearch.transport包UML：



# 二次开发介绍

## Elasticsearch的Java API的统计查询示例

**public static void** statsQuery() {  
 **try** {  
 Settings settings = Settings.*settingsBuilder*().put(**"cluster.name"**, **"elasticsearch1"**).build();  
 TransportClient transportClient = TransportClient.*builder*().  
 settings(settings).build().addTransportAddress(  
 **new** InetSocketTransportAddress(InetAddress.*getByName*(**"172.16.2.95"**), 9300));  
 SearchRequestBuilder searchRequestBuilder = transportClient.prepareSearch(**"sqd.es\_start"**);  
  
 TermsBuilder termsBuilder = AggregationBuilders.*terms*(**"my\_fieldA"**).field(**"fielda"**).size(100);  
 termsBuilder.subAggregation(AggregationBuilders.*sum*(**"my\_sum\_fieldB"**).field(**"fieldb"**));  
 termsBuilder.subAggregation(AggregationBuilders.*avg*(**"my\_avg\_fieldB"**).field(**"fieldb"**));  
 termsBuilder.subAggregation(AggregationBuilders.*max*(**"my\_max\_fieldB"**).field(**"fieldb"**));  
  
 searchRequestBuilder.setQuery(QueryBuilders.*matchAllQuery*()).addAggregation(termsBuilder);  
 SearchResponse searchResponse = searchRequestBuilder.execute().actionGet();  
  
 Terms terms = searchResponse.getAggregations().get(**"my\_fieldA"**);  
 **for** (Terms.Bucket bucket : terms.getBuckets()) {  
 StringBuilder stringBuilder = **new** StringBuilder();  
 String key = (String) bucket.getKey();  
 **long** count = bucket.getDocCount();  
 Sum sumAgg = bucket.getAggregations().get(**"my\_sum\_fieldB"**);  
 **long** sum = (**long**) sumAgg.getValue();  
 Avg avgAgg = bucket.getAggregations().get(**"my\_avg\_fieldB"**);  
 **double** avg = avgAgg.getValue();  
 Max maxAgg = bucket.getAggregations().get(**"my\_max\_fieldB"**);  
 **double** max = maxAgg.getValue();  
 stringBuilder.append(**"key is "** + key + **", count is "** + count +  
 **", sum is "** + sum + **", avg is "** + avg + **", max is "** + max);  
 System.***out***.println(stringBuilder.toString());  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
}

## Elasticsearch的Java API的模糊查询示例

**public static void** wildcardQuery() {  
 **try** {  
 Settings settings = Settings.*settingsBuilder*().put(**"cluster.name"**, **"elasticsearch1"**).build();  
 TransportClient transportClient = TransportClient.*builder*().  
 settings(settings).build().addTransportAddress(  
 **new** InetSocketTransportAddress(InetAddress.*getByName*(**"172.16.2.94"**), 9300));  
 SearchRequestBuilder searchRequestBuilder = transportClient.prepareSearch(**"sqd.es\_start"**);SearchResponse searchResponse = searchRequestBuilder.  
 setQuery(QueryBuilders.*boolQuery*()  
 .must(QueryBuilders.*orQuery*(QueryBuilders.*wildcardQuery*(**"content"**,**"\*mysql\*"**),  
 QueryBuilders.*wildcardQuery*(**"content"**,**"\*oracle\*"**)))  
 .must(QueryBuilders.*termQuery*(**"tbool"**,**"false"**)))  
 .setFrom(0).setSize(100).setExplain(**true**).execute().actionGet();  
 SearchHits searchHits = searchResponse.getHits();  
 System.***out***.println();  
 System.***out***.println(**"Total Hits is "** + searchHits.totalHits());  
 System.***out***.println();  
 **for** (**int** i = 0; i < searchHits.getHits().**length**; ++i) {  
 System.***out***.println(**"content is "** + searchHits.getHits()[i].getSource().get(**"content"**));  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
}

## Elasticsearch的Java API的多条件查询示例

**public static void** multisearch() {  
 **try** {  
 Settings settings = Settings.*settingsBuilder*().put(**"cluster.name"**, **"elasticsearch1"**).build();  
 TransportClient transportClient = TransportClient.*builder*().  
 settings(settings).build().addTransportAddress(  
 **new** InetSocketTransportAddress(InetAddress.*getByName*(**"172.16.2.93"**), 9300));  
 SearchRequestBuilder searchRequestBuilder = transportClient.prepareSearch(**"service2"**,**"clients"**);  
 SearchResponse searchResponse = searchRequestBuilder.  
 setQuery(QueryBuilders.*boolQuery*()  
 .should(QueryBuilders.*termQuery*(**"id"**,**"5"**))  
 .should(QueryBuilders.*prefixQuery*(**"content"**,**"oracle"**)))  
 .setFrom(0).setSize(100).setExplain(**true**).execute().actionGet();  
 SearchHits searchHits = searchResponse.getHits();  
 System.***out***.println();  
 System.***out***.println(**"Total Hits is "** + searchHits.totalHits());  
 System.***out***.println();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
}

## Elasticsearch的Java API的JSON查询

**public static void** jsonquery() {  
 **try** {  
 Settings settings = Settings.*settingsBuilder*().put(**"cluster.name"**, **"elasticsearch1"**).build();  
 TransportClient transportClient = TransportClient.*builder*().  
 settings(settings).build().addTransportAddress(  
 **new** InetSocketTransportAddress(InetAddress.*getByName*(**"172.16.2.93"**), 9300));  
 SearchRequestBuilder searchRequestBuilder = transportClient.prepareSearch(**"service2"**);  
 SearchResponse searchResponse = searchRequestBuilder.setSource(**"{\n"** +  
 **"\"query\": {\n"** +  
 **"\"bool\": {\n"** +  
 **"\"must\": [\n"** +  
 **"{\n"** +  
 **"\"prefix\": {\n"** +  
 **"\"content\": \"oracle\"\n"** +  
 **"}\n"** +  
 **"}\n"** +  
 **"],\n"** +  
 **"\"must\_not\": [ ],\n"** +  
 **"\"should\": [ ]\n"** +  
 **"}\n"** +  
 **"},\n"** +  
 **"\"from\": 0,\n"** +  
 **"\"size\": 10,\n"** +  
 **"\"sort\": [ ],\n"** +  
 **"\"aggs\": { }\n"** +  
 **"}"**)  
 .get();  
 SearchHits searchHits = searchResponse.getHits();  
 System.***out***.println();  
 System.***out***.println(**"Total Hits is "** + searchHits.totalHits());  
 System.***out***.println();  
 **for** (**int** i = 0; i < searchHits.getHits().**length**; ++i) {  
 System.***out***.println(**"content is "** + searchHits.getHits()[i].getSource().get(**"content"**));  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
}

## Elasticsearch的Java API的创建索引

**public static void** createByJson() {  
 **try** {  
 Settings settings = Settings.*settingsBuilder*().put(**"cluster.name"**, **"elasticsearch1"**).build();  
 TransportClient client = TransportClient.*builder*().  
 settings(settings).build().addTransportAddress(  
 **new** InetSocketTransportAddress(InetAddress.*getByName*(**"172.16.2.93"**), 9300));  
 XContentBuilder mapping = *jsonBuilder*()  
 .startObject()  
 .startObject(**"general"**)  
 .startObject(**"properties"**)  
 .startObject(**"message"**)  
 .field(**"type"**, **"string"**)  
 .field(**"index"**, **"not\_analyzed"**)  
 .endObject()  
 .startObject(**"source"**)  
 .field(**"type"**,**"string"**)  
 .endObject()  
 .endObject()  
 .endObject()  
 .endObject();  
 client.admin().indices().prepareCreate(**"t2"**)  
 .setSettings(Settings.*builder*()  
 .put(**"index.number\_of\_shards"**, 3)  
 .put(**"index.number\_of\_replicas"**, 2)  
 )  
 .get();  
 client.admin().indices().preparePutMapping(**"t2"**)  
 .setType(**"general"**)  
 .setSource(mapping)  
 .execute().actionGet();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
}