ElasticSearch学习资料

内部文件：[1.0]

颁布时间：[2014.1.21]

**目 录**

[ 文件版本说明 3](#_Toc378090310)

[ 参考资料 3](#_Toc378090311)

[ 手册目的 3](#_Toc378090312)

[ 声明 3](#_Toc378090313)

[ 名词定义和缩略语说明 3](#_Toc378090314)

[1. 总述 4](#_Toc378090315)

[1.1. 简介 4](#_Toc378090316)

[1.2. 国外的使用案例 4](#_Toc378090317)

[1.3. 基本概念解析 6](#_Toc378090318)

[1.3.1. Cluster 6](#_Toc378090319)

[1.3.2. Shards 6](#_Toc378090320)

[1.3.3. Replicas 6](#_Toc378090321)

[1.3.4. Recovery 7](#_Toc378090322)

[1.3.5. River 7](#_Toc378090323)

[1.3.6. Gateway 7](#_Toc378090324)

[1.3.7. discovery.zen 7](#_Toc378090325)

[1.3.8. Transport 7](#_Toc378090326)

[2. 服务器搭建 8](#_Toc378090327)

[2.1. 单机环境 8](#_Toc378090328)

[2.2. 服务器环境 8](#_Toc378090329)

[2.3. 中文分词集成 9](#_Toc378090330)

[2.4. 配置详解 12](#_Toc378090331)

[3. Java API 15](#_Toc378090332)

[3.1. 与集群交互 15](#_Toc378090333)

[3.1.1. Node方式 15](#_Toc378090334)

[3.1.2. TransportClient方式 16](#_Toc378090335)

[3.2. put Mapping定义索引字段属性 16](#_Toc378090336)

[3.3. 索引数据 19](#_Toc378090337)

[3.4. 删除索引数据 19](#_Toc378090338)

[3.5. 搜索 20](#_Toc378090339)

[3.6. 批量添加索引 21](#_Toc378090340)

[3.7. 与MongoDB同步数据 22](#_Toc378090341)

[3.8. 使用More like this实现基于内容的推荐 25](#_Toc378090342)

文件版本说明

表 1 版本说明

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 发布时间 | 修订章节 | 作者 |
| 1.0 | 2014.1.21 | 第一版 | 虞晶 |
|  |  |  |  |
|  |  |  |  |

参考资料

* 1. Elasticsearch官网：http://www.elasticsearch.cn/guide/

手册目的

ElasticSearch学习资料

声明

无

名词定义和缩略语说明

表 2 名词定义及缩略语说明

| 序号 | 缩写 | 说明 |
| --- | --- | --- |
| 1 | ES | Elasticsearch，一种设计用于云计算的分布式全文索引解决方案。 |
|  |  |  |
|  |  |  |

# 服务器搭建

先到https://github.com/medcl/elasticsearch-rtf/下载最新版的elasticsearch集合运行包。

## 单机环境

单机版的elasticsearch运行很简单，linux下直接 bin/elasticsearch就运行了，windows运行bin/elasticsearch.bat。如果是在局域网中运行elasticsearch集群也是很简单的，只要cluster.name设置一致，并且机器在同一网段下，启动的es会自动发现对方，组成集群。

## 服务器环境

如果是在服务器上就可以使用elasticsearch-servicewrapper这个es插件，它支持通过参数，指定是在后台或前台运行es，并且支持启动，停止，重启es服务（默认es脚本只能通过ctrl+c关闭es）。使用方法是到https://github.com/elasticsearch/elasticsearch-servicewrapper下载service文件夹，放到es的bin目录下。下面是命令集合：

bin/service/elasticsearch +

console 在前台运行es

start 在后台运行es

stop 停止es

install 使es作为服务在服务器启动时自动启动

remove 取消启动时自动启动

在service目录下有个elasticsearch.conf配置文件，主要是设置一些java运行环境参数，其中比较重要的是下面的

参数：

#es的home路径，不用用默认值就可以

set.default.ES\_HOME=<Path to ElasticSearch Home>

#分配给es的最小内存

set.default.ES\_MIN\_MEM=256

#分配给es的最大内存

set.default.ES\_MAX\_MEM=1024

# 启动等待超时时间（以秒为单位）

wrapper.startup.timeout=300

# 关闭等待超时时间（以秒为单位）

wrapper.shutdown.timeout=300

# ping超时时间(以秒为单位)

wrapper.ping.timeout=300

## 中文分词集成

elasticsearch官方只提供smartcn这个中文分词插件，效果不是很好，好在国内有medcl大神（国内最早研究es的人之一）写的两个中文分词插件，一个是ik的，一个是mmseg的，下面分别介绍下两者的用法，其实都差不多的，先安装插件，命令行：

安装ik插件：

plugin -install medcl/elasticsearch-analysis-ik/1.1.0

或者手动通过下载包安装，在github上有个最新的

<https://github.com/medcl/elasticsearch-rtf/blob/master/elasticsearch/plugins/analysis-ik/elasticsearch-analysis-ik-1.2.5.jar>

（直接用plugin --install //方式安装，这个真看人品，反正我是没装上。）

下载后用plugin --url file://path/to/plugin --install plugin-name方式安装，没问题，安装成功。

下载ik相关配置词典文件到config目录

cd config

wget http://github.com/downloads/medcl/elasticsearch-analysis-ik/ik.zip --no-check-certificate

unzip ik.zip

rm ik.zip

安装mmseg插件：

bin/plugin -install medcl/elasticsearch-analysis-mmseg/1.1.0

下载相关配置词典文件到config目录

cd config

wget http://github.com/downloads/medcl/elasticsearch-analysis-mmseg/mmseg.zip --no-check-certificate

unzip mmseg.zip

rm mmseg.zip

分词配置

ik分词配置，在elasticsearch.yml文件中加上

index:

analysis:

analyzer:

ik:

alias: [ik\_analyzer]

type: org.elasticsearch.index.analysis.IkAnalyzerProvider

或

index.analysis.analyzer.ik.type : “ik”

这两句的意义相同

mmseg分词配置，也是在在elasticsearch.yml文件中

index:

analysis:

analyzer:

mmseg:

alias: [news\_analyzer, mmseg\_analyzer]

type: org.elasticsearch.index.analysis.MMsegAnalyzerProvider

或

index.analysis.analyzer.default.type : "mmseg"

mmseg分词还有些更加个性化的参数设置如下

index:

analysis:

tokenizer:

mmseg\_maxword:

type: mmseg

seg\_type: "max\_word"

mmseg\_complex:

type: mmseg

seg\_type: "complex"

mmseg\_simple:

type: mmseg

seg\_type: "simple"

这样配置完后插件安装完成，启动es就会加载插件。

定义mapping

在添加索引的mapping时就可以这样定义分词器

{

"page":{

"properties":{

"title":{

"type":"string",

"indexAnalyzer":"ik",

"searchAnalyzer":"ik"

},

"content":{

"type":"string",

"indexAnalyzer":"ik",

"searchAnalyzer":"ik"

}

}

}

}

indexAnalyzer为索引时使用的分词器，searchAnalyzer为搜索时使用的分词器。

java mapping代码如下：

XContentBuilder content = XContentFactory.jsonBuilder().startObject()

.startObject("page")

.startObject("properties")

.startObject("title")

.field("type", "string")

.field("indexAnalyzer", "ik")

.field("searchAnalyzer", "ik")

.endObject()

.startObject("code")

.field("type", "string")

.field("indexAnalyzer", "ik")

.field("searchAnalyzer", "ik")

.endObject()

.endObject()

.endObject()

.endObject()

定义完后操作索引就会以指定的分词器来进行分词。

测试分词可用调用下面api，注意indexname为索引名，随便指定一个索引就行了

http://localhost:9200/indexname/\_analyze?analyzer=ik&text=测试elasticsearch分词器

附：

ik分词插件项目地址：https://github.com/medcl/elasticsearch-analysis-ik

mmseg分词插件项目地址：https://github.com/medcl/elasticsearch-analysis-mmseg

如果觉得配置麻烦，也可以下载个配置好的es版本，地址如下：<https://github.com/medcl/elasticsearch-rtf>

## 配置详解

elasticsearch的config文件夹里面有两个配置文 件：elasticsearch.yml和logging.yml，第一个是es的基本配置文件，第二个是日志配置文件，es也是使用log4j来记录日 志的，所以logging.yml里的设置按普通log4j配置文件来设置就行了。下面主要讲解下elasticsearch.yml这个文件中可配置的 东西。

cluster.name: elasticsearch

配置es的集群名称，默认是elasticsearch，es会自动发现在同一网段下的es，如果在同一网段下有多个集群，就可以用这个属性来区分不同的集群。

node.name: "Franz Kafka"

节点名，默认随机指定一个name列表中名字，该列表在es的jar包中config文件夹里name.txt文件中，其中有很多作者添加的有趣名字。

node.master: true

指定该节点是否有资格被选举成为node，默认是true，es是默认集群中的第一台机器为master，如果这台机挂了就会重新选举master。

node.data: true

指定该节点是否存储索引数据，默认为true。

index.number\_of\_shards: 5

设置默认索引分片个数，默认为5片。

index.number\_of\_replicas: 1

设置默认索引副本个数，默认为1个副本。

path.conf: /path/to/conf

设置配置文件的存储路径，默认是es根目录下的config文件夹。

path.data: /path/to/data

设置索引数据的存储路径，默认是es根目录下的data文件夹，可以设置多个存储路径，用逗号隔开，例：

path.data: /path/to/data1,/path/to/data2

path.work: /path/to/work

设置临时文件的存储路径，默认是es根目录下的work文件夹。

path.logs: /path/to/logs

设置日志文件的存储路径，默认是es根目录下的logs文件夹

path.plugins: /path/to/plugins

设置插件的存放路径，默认是es根目录下的plugins文件夹

bootstrap.mlockall: true

设置为true来锁住内存。因为当jvm开始swapping时es的效率会降低，所以要保证它不swap，可以把ES\_MIN\_MEM和 ES\_MAX\_MEM两个环境变量设置成同一个值，并且保证机器有足够的内存分配给es。同时也要允许elasticsearch的进程可以锁住内 存，linux下可以通过`ulimit -l unlimited`命令。

network.bind\_host: 192.168.0.1

设置绑定的ip地址，可以是ipv4或ipv6的，默认为0.0.0.0。

network.publish\_host: 192.168.0.1

设置其它节点和该节点交互的ip地址，如果不设置它会自动判断，值必须是个真实的ip地址。

network.host: 192.168.0.1

这个参数是用来同时设置bind\_host和publish\_host上面两个参数。

transport.tcp.port: 9300

设置节点间交互的tcp端口，默认是9300。

transport.tcp.compress: true

设置是否压缩tcp传输时的数据，默认为false，不压缩。

http.port: 9200

设置对外服务的http端口，默认为9200。

http.max\_content\_length: 100mb

设置内容的最大容量，默认100mb

http.enabled: false

是否使用http协议对外提供服务，默认为true，开启。

gateway.type: local

gateway的类型，默认为local即为本地文件系统，可以设置为本地文件系统，分布式文件系统，hadoop的HDFS，和amazon的s3服务器，其它文件系统的设置方法下次再详细说。

gateway.recover\_after\_nodes: 1

设置集群中N个节点启动时进行数据恢复，默认为1。

gateway.recover\_after\_time: 5m

设置初始化数据恢复进程的超时时间，默认是5分钟。

gateway.expected\_nodes: 2

设置这个集群中节点的数量，默认为2，一旦这N个节点启动，就会立即进行数据恢复。

cluster.routing.allocation.node\_initial\_primaries\_recoveries: 4

初始化数据恢复时，并发恢复线程的个数，默认为4。

cluster.routing.allocation.node\_concurrent\_recoveries: 2

添加删除节点或负载均衡时并发恢复线程的个数，默认为4。

indices.recovery.max\_size\_per\_sec: 0

设置数据恢复时限制的带宽，如入100mb，默认为0，即无限制。

indices.recovery.concurrent\_streams: 5

设置这个参数来限制从其它分片恢复数据时最大同时打开并发流的个数，默认为5。

discovery.zen.minimum\_master\_nodes: 1

设置这个参数来保证集群中的节点可以知道其它N个有master资格的节点。默认为1，对于大的集群来说，可以设置大一点的值（2-4）

discovery.zen.ping.timeout: 3s

设置集群中自动发现其它节点时ping连接超时时间，默认为3秒，对于比较差的网络环境可以高点的值来防止自动发现时出错。

discovery.zen.ping.multicast.enabled: false

设置是否打开多播发现节点，默认是true。

discovery.zen.ping.unicast.hosts: ["host1", "host2:port", "host3[portX-portY]"]

设置集群中master节点的初始列表，可以通过这些节点来自动发现新加入集群的节点。

下面是一些查询时的慢日志参数设置

index.search.slowlog.level: TRACE  
index.search.slowlog.threshold.query.warn: 10s  
index.search.slowlog.threshold.query.info: 5s  
index.search.slowlog.threshold.query.debug: 2s  
index.search.slowlog.threshold.query.trace: 500ms  
index.search.slowlog.threshold.fetch.warn: 1s  
index.search.slowlog.threshold.fetch.info: 800ms  
index.search.slowlog.threshold.fetch.debug:500ms  
index.search.slowlog.threshold.fetch.trace: 200ms

## 管理工具

### elasticsearch-head

elasticsearch-head是一个elasticsearch的集群管理工具，它是完全由html5编写的独立网页程序，你可以通过插件把它集成到es。或直接下载源码，在本地打开index.html运行它。该工具的git地址是： <https://github.com/Aconex/elasticsearch-head>

插件安装方法：

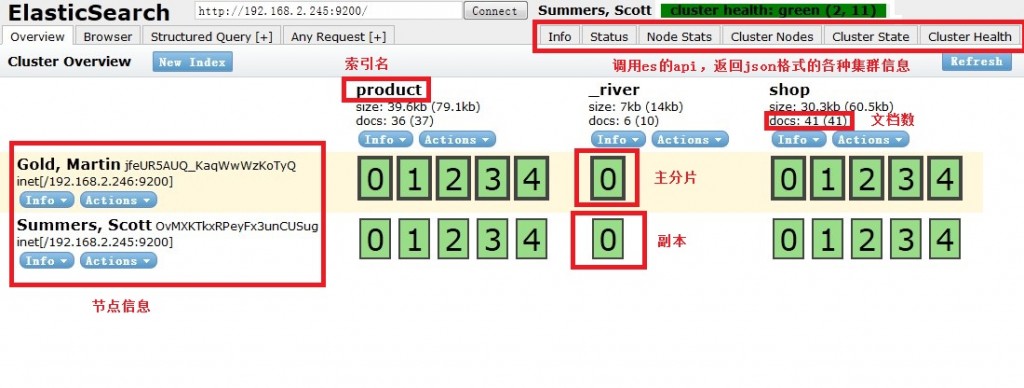
1.elasticsearch/bin/plugin -install Aconex/elasticsearch-head

2.运行es

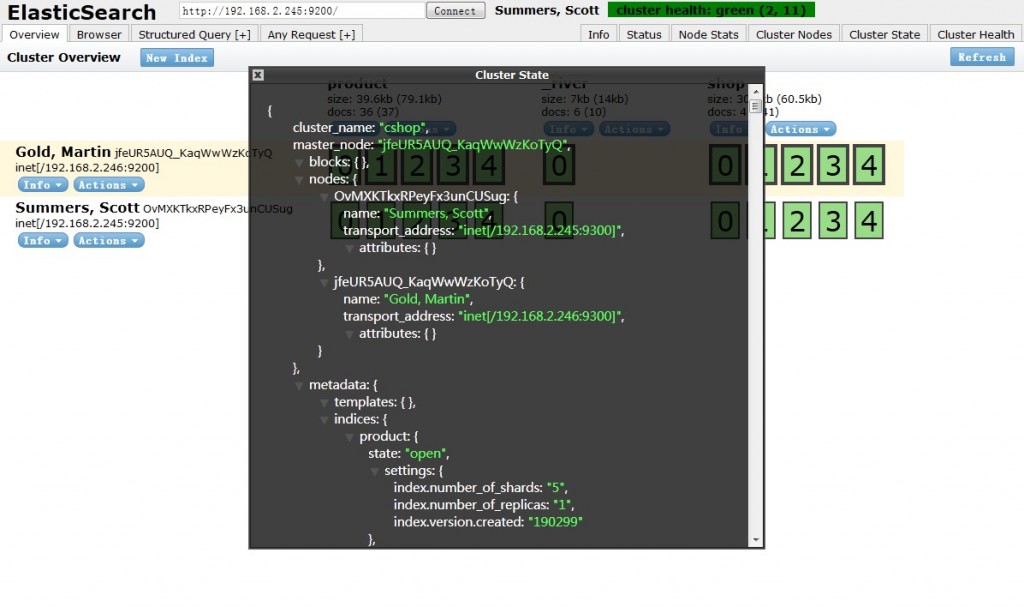
3.打开http://localhost:9200/\_plugin/head/

不想通过插件集成到es的话就可以直接在git上下载源码到本地运行。

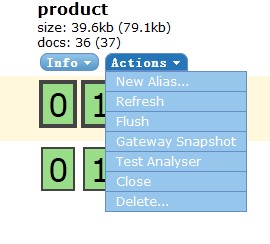
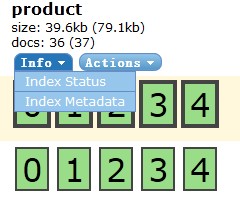
在地址栏输入es服务器的ip地址和端口点connect就可以连接到集群。下面是连接后的视图。这是主界面，在这里可以看到es集群的基本信息（如：节点情况，索引情况）。



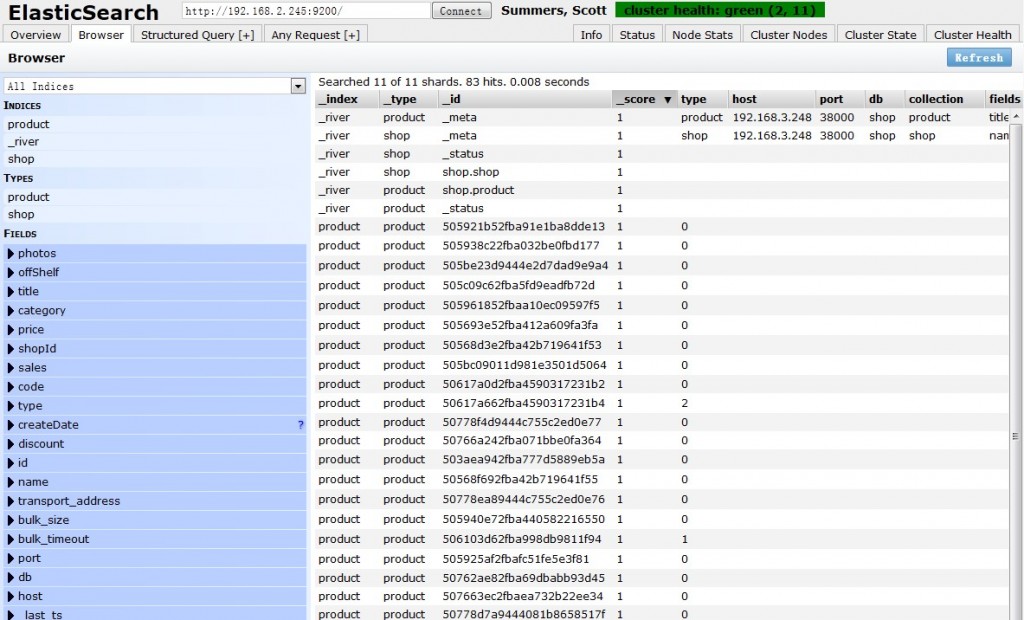
界面的右边有些按钮，如：node stats， cluster nodes，这些是直接请求es的相关状态的api，返回结果为json，如下图：



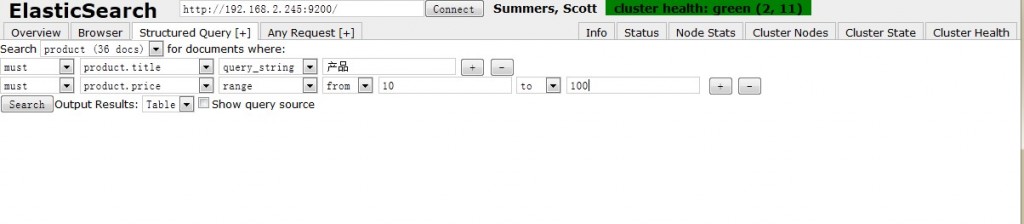
在索引下面有info和action两个按钮。info是可以查看索引的状态和mapping的定义。action是对索引进行操作，如：添加别名、刷新、关闭索引，删除索引等。



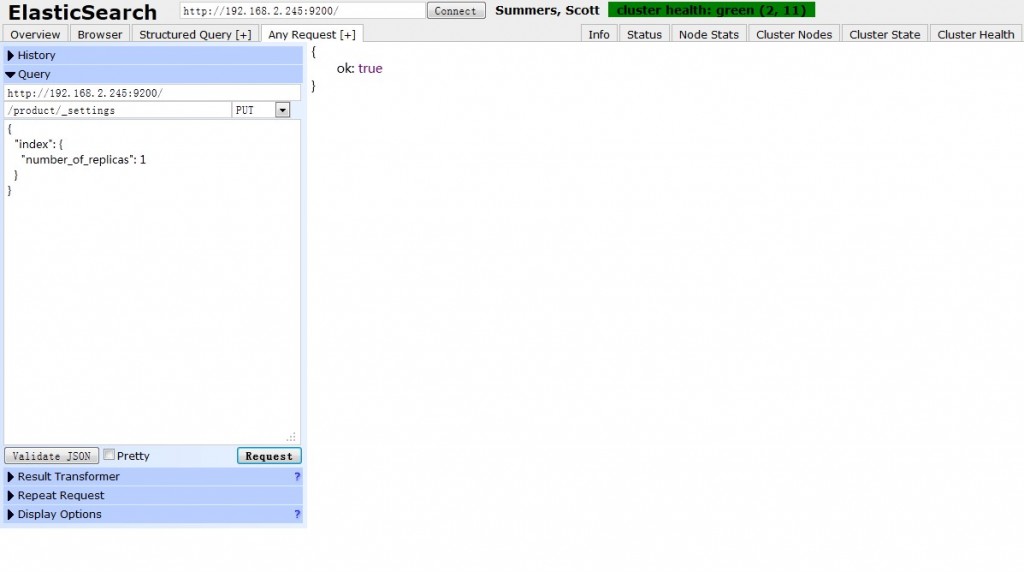
browser浏览界面，这个界面可以同时查看多个索引的数据，也可以查询指定字段的数据。



Structured Query查询界面，这个界面可以对某个索引进行一些复杂查询，如下面这个例子是查询product索引，构造boolquery，title字段里查询“产品”关键词，price范围为10到100的记录。



Any Request任意请求界面，这个界面可以说是个rest的客户端，可以通过它来对es进行一些请求操作或测试api接口，下面这个例子是把product索引的副本数设置为1，更多的api可以到es官网查询。



### elasticsearch-bigdesk

bigdesk是elasticsearch的一个集群监控工具，可以通过它来查看es集群的各种状态，如：cpu、内存使用情况，索引数据、搜索情况，http连接数等。项目git地址： <https://github.com/lukas-vlcek/bigdesk>。和head一样，它也是个独立的网页程序，使用方式和head一样。

插件安装运行：

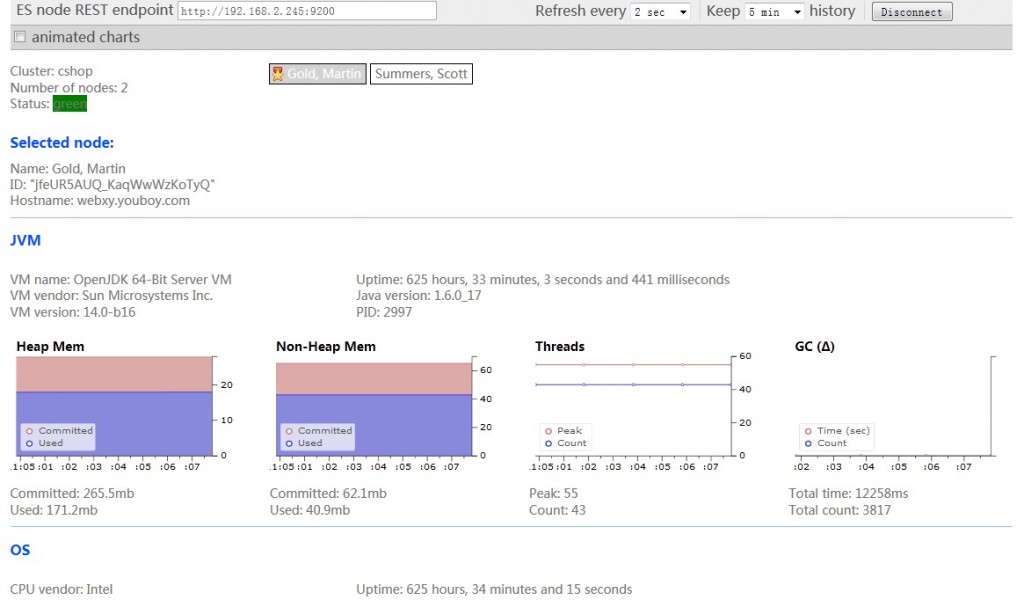
1.bin/plugin -install lukas-vlcek/bigdesk

2.运行es

3.打开http://localhost:9200/\_plugin/bigdesk/

当然，也可以直接下载源码运行index.html

同样是输入ip地址和端口后连接，界面如下。加星的表示主节点。



下面介绍下各个图表。

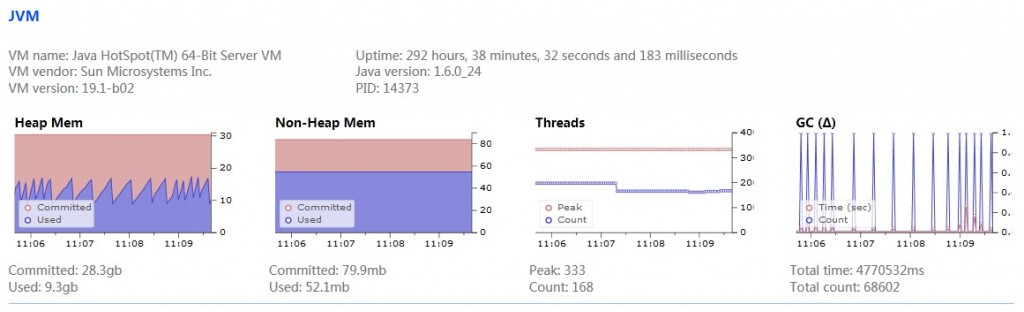
**系统监控**：

这里包含系统方面的一些状态，左起分别为：cpu，内存，交换区和平均负载的情况



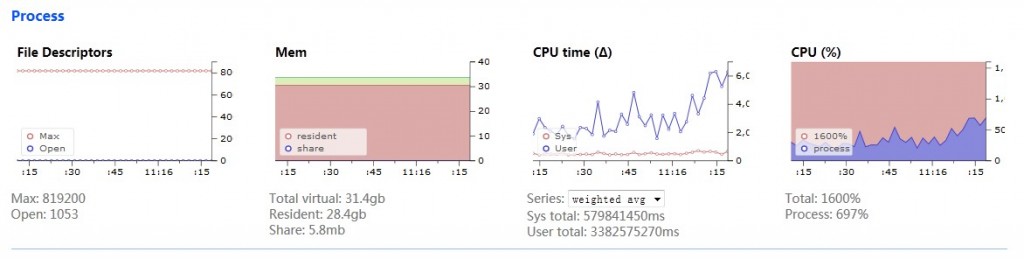
**jvm：**

显示jvm的一些状态，左起分别为：jvm heap内存使用情况，蓝色的为已使用内存；非heap使用内存；线程数；gc情况（次数和时间）；



**进程：**

下面四张图主要显示es的进程对系统资源的使用情况，左起分别为：进程打开文件数，内存使用情况，cpu时间和进程的cpu使用率



ps：

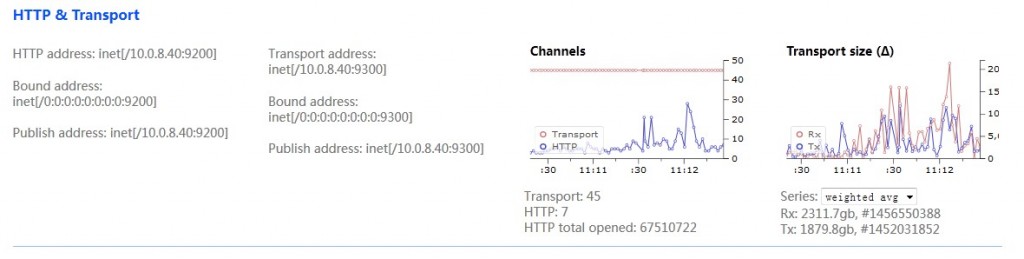
内存使用情况中的

Total virtual指linux下虚拟内存，它包括virtual memory map中的所有数据量之和。包括：程序类+程序数据+jar包空间+jre占用空间等。

resident memory指程序实际占用的物理内存。

**通讯：**

这里可以查看tcp和http链接的一些数据。

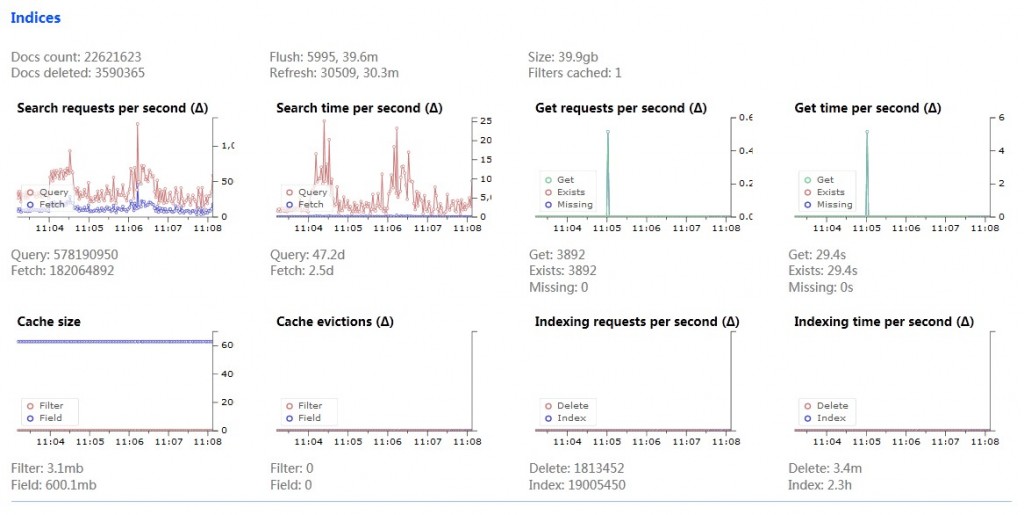


**索引：**

这里可以查看索引数据和搜索的一些情况。

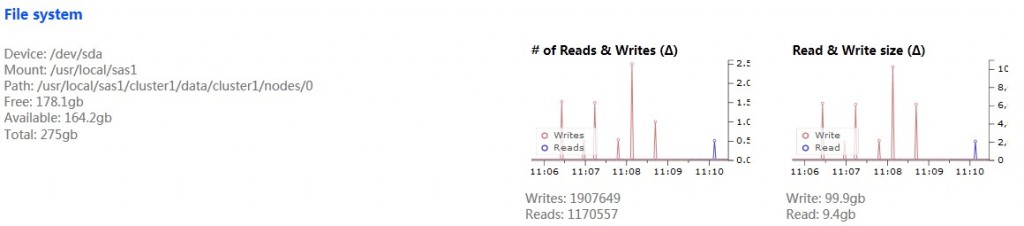
左上起：每秒索引请求，搜索时间，每秒取数据请求，取数据时间。

左下起：缓存大小，缓存失效个数，每秒索引请求，索引时间。



**文件系统：**

显示硬盘的读写情况



# Moduls

### Cluster

代表一个集群，集群中有多个节点，其中有一个为主节点，这个主节点是可以通过选举产生的，主从节点是对于集群内部来说的。es的一个概念就是去中心化，字面上理解就是无中心节点，这是对于集群外部来说的，因为从外部来看es集群，在逻辑上是个整体，你与任何一个节点的通信和与整个es集群通信是等价的。

### Shards

代表索引分片，es可以把一个完整的索引分成多个分片，这样的好处是可以把一个大的索引拆分成多个，分布到不同的节点上。构成分布式搜索。分片的数量只能在索引创建前指定，并且索引创建后不能更改。

### Replicas

代表索引副本，es可以设置多个索引的副本，副本的作用一是提高系统的容错性，当个某个节点某个分片损坏或丢失时可以从副本中恢复。二是提高es的查询效率，es会自动对搜索请求进行负载均衡。

### Recovery

代表数据恢复或叫数据重新分布，es在有节点加入或退出时会根据机器的负载对索引分片进行重新分配，挂掉的节点重新启动时也会进行数据恢复。

### River

代表es的一个数据源，也是其它存储方式（如：数据库）同步数据到es的一个方法。它是以插件方式存在的一个es服务，通过读取river中的数据并把它索引到es中，官方的river有couchDB的，RabbitMQ的，Twitter的，Wikipedia的，river这个功能将会在后面的文件中重点说到。

### Gateway

代表es索引的持久化存储方式，es默认是先把索引存放到内存中，当内存满了时再持久化到硬盘。当这个es集群关闭再重新启动时就会从gateway中读取索引数据。es支持多种类型的gateway，有本地文件系统（默认），分布式文件系统，Hadoop的HDFS和amazon的s3云存储服务。

### discovery.zen

代表es的自动发现节点机制，es是一个基于p2p的系统，它先通过广播寻找存在的节点，再通过多播协议来进行节点之间的通信，同时也支持点对点的交互。

### Transport

代表es内部节点或集群与客户端的交互方式，默认内部是使用tcp协议进行交互，同时它支持http协议（json格式）、thrift、servlet、memcached、zeroMQ等的传输协议（通过插件方式集成）。

# Java API

## 与集群交互

可以通过两种方式来连接到elasticsearch（简称es）集群，第一种是通过在你的程序中创建一个嵌入es节点（Node），使之成为es集群的一部分，然后通过这个节点来与es集群通信。第二种方式是用TransportClient这个接口和es集群通信。

### Node方式

创建嵌入节点的方式如下：

import static org.elasticsearch.node.NodeBuilder.\*;

//启动节点

Node node = nodeBuilder().node();

Client client = node.client();

//关闭节点

node.close();

当你启动一个节点，它会自动加入同网段的es集群，一个前提就是es的集群名（cluster.name）这个参数要设置一致。

默认的话启动一个节点，es集群会自动给它分配一些索引的分片，如果你想这个节点仅仅作为一个客户端而不去保存数据，你就可以设置把node.data设置成false或 node.client设置成true。下面是例子：

Node node = nodeBuilder().clusterName(clusterName).client(true).node();

还有一种情况是你并不想把节点加入集群，只想用它进行单元测试时，就要启动一个“本地”的es，这里“本地”指的是在jvm的级别下运行，即两个不同的es节点运行在同一个JVM中时会组成一个集群。它需要把节点的local参数设置成true，下面是例子：

Node node = nodeBuilder().local(true).node();

### TransportClient方式

通过TransportClient这个接口，我们可以不启动节点就可以和es集群进行通信，它需要指定es集群中其中一台或多台机的ip地址和端口，例子如下：

Client client = new TransportClient()

.addTransportAddress(new InetSocketTransportAddress("host1", 9300))

.addTransportAddress(new InetSocketTransportAddress("host2", 9300));

client.close();

如果你需要更改集群名（默认是elasticsearch），需要如下设置：

Settings settings = ImmutableSettings.settingsBuilder()

.put("cluster.name", "myClusterName").build();

Client client = new TransportClient(settings);

你可以设置client.transport.sniff为true来使客户端去嗅探整个集群的状态，把集群中其它机器的ip地址加到客户端中，这样做的好处是一般你不用手动设置集群里所有集群的ip到连接客户端，它会自动帮你添加，并且自动发现新加入集群的机器。代码实例如下：

Settings settings = ImmutableSettings.settingsBuilder()

.put("client.transport.sniff", true).build();

TransportClient client = new TransportClient(settings);

## put Mapping定义索引字段属性

Mapping,就是对索引库中索引的字段名及其数据类型进行定义，类似于关系数据库中表建立时要定义字段名及其数据类型那样，不过es的mapping比数据库灵活很多，它可以动态添加字段。一般不需要要指定mapping都可以，因为es会自动根据数据格式定义它的类型，如果你需要对某些字段添加特殊属性（如：定义使用其它分词器、是否分词、是否存储等），就必须手动添加mapping。有两种添加mapping的方法，一种是定义在配置文件中，一种是运行时手动提交mapping，两种选一种就行了。

先介绍在配置文件中定义mapping，你可以把[mapping名].json文件放到config/mappings/[索引名]目录下，这个目录要自己创建，一个mapping和一个索引对应，你也可以定义一个默认的mapping，把自己定义的default-mapping.json放到config目录下就行。json格式如下：

{

"mappings":{

"properties":{

"title":{

"type":"string",

"store":"yes"

},

"description":{

"type":"string",

"index":"not\_analyzed"

},

"price":{

"type":"double"

},

"onSale":{

"type":"boolean"

},

"type":{

"type":"integer"

},

"createDate":{

"type":"date"

}

}

}

}

接下来介绍通过请求添加mapping，下面为一个添加productIndex索引库的mapping的json格式请求。其中productIndex为索引类型，properties下面的为索引里面的字段，type为数据类型，store为是否存储，"index":"not\_analyzed"为不对该字段进行分词。

{

"productIndex":{

"properties":{

"title":{

"type":"string",

"store":"yes"

},

"description":{

"type":"string",

"index":"not\_analyzed"

},

"price":{

"type":"double"

},

"onSale":{

"type":"boolean"

},

"type":{

"type":"integer"

},

"createDate":{

"type":"date"

}

}

}

}

用java api调用的代码如下：

先创建空索引库

client.admin().indices().prepareCreate("productIndex").execute().actionGet();

put mapping：

XContentBuilder mapping = jsonBuilder()

.startObject()

.startObject("productIndex")

.startObject("properties")

.startObject("title").field("type", "string").field("store", "yes").endObject()

.startObject("description").field("type", "string").field("index", "not\_analyzed").endObject()

.startObject("price").field("type", "double").endObject()

.startObject("onSale").field("type", "boolean").endObject()

.startObject("type").field("type", "integer").endObject()

.startObject("createDate").field("type", "date").endObject()

.endObject()

.endObject()

.endObject();

PutMappingRequest mappingRequest = Requests.putMappingRequest("productIndex").type("productIndex").source(mapping);

client.admin().indices().putMapping(mappingRequest).actionGet();

## 索引数据

es索引数据非常方便，只需构建个json格式的数据提交到es就行，下面是个java api的例子

XContentBuilder doc = jsonBuilder()

.startObject()

.field("title", "this is a title!")

.field("description", "descript what?")

.field("price", 100)

.field("onSale", true)

.field("type", 1)

.field("createDate", new Date())

.endObject();

client.prepareIndex("productIndex","productType").setSource(doc).execute().actionGet();

其中productIndex为索引库名，一个es集群中可以有多个索引库。productType为索引类型，是用来区分同索引库下不同类型的数据的，一个索引库下可以有多个索引类型。

## 删除索引数据

删除api允许从特定索引通过id删除json文档。有两种方法，一是通过id删除，二是通过一个Query查询条件删除，符合这些条件的数据都会被删除。

一、通过id删除

下面的例子是删除索引名为twitter，类型为tweet，id为1的文档：

DeleteResponse response = client.prepareDelete("twitter", "tweet", "1")

.execute()

.actionGet();

二、通过Query删除

下面的例子是删除索引名为productIndex，title中包含query的所有文档：

QueryBuilder query = QueryBuilders.fieldQuery("title", "query");

client.prepareDeleteByQuery("productIndex").setQuery(query).execute().actionGet();

设置线程

当删除api在同一个节点上执行时（在一个分片中执行一个api会分配到同一个服务器上），删除api允许执行前设置线程模式（operationThreaded选项），operationThreaded这个选项是使这个操作在另外一个线程中执行，或在一个正在请求的线程（假设这个api仍是异步的）中执行。默认的话operationThreaded会设置成true，这意味着这个操作将在一个不同的线程中执行。下面是设置成false的方法：

DeleteResponse response = client.prepareDelete("twitter", "tweet", "1")

.setOperationThreaded(false)

.execute()

.actionGet();

官方文档：

http://www.elasticsearch.org/guide/reference/api/delete.html

http://www.elasticsearch.org/guide/reference/java-api/delete.html

## 搜索

elasticsearch的查询是通过执行json格式的查询条件，在java api中就是构造QueryBuilder对象，elasticsearch完全支持queryDSL风格的查询方式，QueryBuilder的构建类是QueryBuilders，filter的构建类是FilterBuilders。下面是构造QueryBuilder的例子：

import static org.elasticsearch.index.query.FilterBuilders.\*;

import static org.elasticsearch.index.query.QueryBuilders.\*;

QueryBuilder qb1 = termQuery("name", "kimchy");

QueryBuilder qb2 = boolQuery()

.must(termQuery("content", "test1"))

.must(termQuery("content", "test4"))

.mustNot(termQuery("content", "test2"))

.should(termQuery("content", "test3"));

QueryBuilder qb3 = filteredQuery(

termQuery("name.first", "shay"),

rangeFilter("age")

.from(23)

.to(54)

.includeLower(true)

.includeUpper(false)

);

其中qb1构造了一个TermQuery，对name这个字段进行项搜索，项是最小的索引片段，这个查询对应lucene本身的TermQuery。 qb2构造了一个组合查询（BoolQuery），其对应lucene本身的BooleanQuery，可以通过must、should、mustNot方法对QueryBuilder进行组合，形成多条件查询。 qb3构造了一个过滤查询，就是在TermQuery的基础上添加一个过滤条件RangeFilter，这个范围过滤器将限制查询age字段大于等于23，小于等于54的结果。除了这三个，elasticsearch还支持很多种类的查询方式，迟点写个介绍。

构造好了Query就要传到elasticsearch里面进行查询，下面是例子：

SearchResponse response = client.prepareSearch("test")

.setQuery(query)

.setFrom(0).setSize(60).setExplain(true)

.execute()

.actionGet();

这句的意思是，查询test索引，查询条件为query，从第0条记录开始，最多返回60条记录。返回结果为SearchResponse，下面解析SearchResponse：

SearchHits hits = searchResponse.hits();

for (int i = 0; i < 60; i++) {

System.out.println(hits.getAt(i).getSource().get("field"));

}

获得SearchResponse中的SearchHits，然后hits.getAt(i).getSource().get("field")获得field字段的值。

## 批量添加索引

elasticsearch支持批量添加或删除索引文档，java api里面就是通过构造BulkRequestBuilder，然后把批量的index/delete请求添加到BulkRequestBuilder里面，执行BulkRequestBuilder。下面是个例子：

import static org.elasticsearch.common.xcontent.XContentFactory.\*;

BulkRequestBuilder bulkRequest = client.prepareBulk();

bulkRequest.add(client.prepareIndex("twitter", "tweet", "1")

.setSource(jsonBuilder()

.startObject()

.field("user", "kimchy")

.field("postDate", new Date())

.field("message", "trying out Elastic Search")

.endObject()

)

);

bulkRequest.add(client.prepareIndex("twitter", "tweet", "2")

.setSource(jsonBuilder()

.startObject()

.field("user", "kimchy")

.field("postDate", new Date())

.field("message", "another post")

.endObject()

)

);

BulkResponse bulkResponse = bulkRequest.execute().actionGet();

if (bulkResponse.hasFailures()) {

//处理错误

}

## 与MongoDB同步数据

elasticsearch提供river这个模块来读取数据源中的数据到es中，es官方有提供couchDB的同步插件，因为项目用到的是mongodb，所以在找mongodb方面的同步插件，在git上找到了elasticsearch-river-mongodb。

这个插件最初是由aparo写的，最开始的功能就是读取mongodb里面的表，记录最后一条数据的id，根据时间间隔不断访问mongodb，看看有没有大于之前记录的id的数据，有的话就索引数据，这种做法的缺点就是只能同步最新的数据，修改或删除的就不能同步。后来又由richardwilly98等人修改成通过读取mongodb的oplog来同步数据。因为mongodb是通过oplog这个表来使集群中的不同机器数据同步的，这样做的话可以保证es里面的数据和mongodb里面的是一样的，因为mongodb中的数据一有改变，都会通过oplog反映到monogodb中。他们还添加了个索引mongodb gridfs里文件的功能，非常好。

但他们修改完后的插件还是有些不满意的地方。他把local库（放oplog的）和普通库的访问密码都设置成同一个，如果local库和普通库的用户名和密码不同那这个插件就不能用了。还有一个就是同步时会把mongodb的表中所有的字段都同步过去，但是有些字段我们并不想把它放到索引中，于是对这个插件再作修改，把local库和普通库的鉴权分开，添加可选字段功能。

运行环境：Elasticsearch 0.19.X

集群环境下的MongoDB 2.X

注意：该插件只支持集群环境下的mongodb，因为集群环境下的mongodb才有oplog这个表。

安装方法：

安装elasticsearch-mapper-attachments插件（用于索引gridfs里的文件）

%ES\_HOME%\bin\plugin.bat -install elasticsearch/elasticsearch-mapper-attachments/1.4.0

安装elasticsearch-river-mongodb（同步插件）

%ES\_HOME%\bin\plugin.bat -install laigood/elasticsearch-river-mongodb/laigoodv1.0.0

创建river方法：

curl方式：

$ curl -XPUT "localhost:9200/\_river/mongodb/\_meta" -d '

{

type: "mongodb",

mongodb: {

db: "test",

host: "localhost",

port: "27017",

collection: "testdb",

fields:"title,content",

gridfs: "true",

local\_db\_user: "admin",

local\_db\_password:"admin",

db\_user: "user",

db\_password:"password"

},

index: {

name: "test",

type: "type",

bulk\_size: "1000",

bulk\_timeout: "30"

}

}

db为同步的数据库名，

host mongodb的ip地址（默认为localhost），

port mongodb的端口，

collection 要同步的表名

fields 要同步的字段名（用逗号隔开，默认全部）

gridfs 是否是gridfs文件（如果collection是gridfs的话就设置成true）

local\_db\_user local数据库的用户名（没有的话不用写）

local\_db\_password local数据库的密码（没有的话不用写）

db\_user 要同步的数据库的密码（没有的话不用写）

db\_password 要同步的数据库的密码（没有的话不用写）

name 索引名（不能之前存在）

type 类型

bulk\_size 批量添加的最大数

bulk\_timeout 批量添加的超时时间

java api方式：

client.prepareIndex("\_river", "testriver", "\_meta")

.setSource(

jsonBuilder().startObject()

.field("type", "mongodb")

.startObject("mongodb")

.field("host","localhost")

.field("port",27017)

.field("db","testdb")

.field("collection","test")

.field("fields","title,content")

.field("db\_user","user")

<span style="white-space:pre"> </span>.field("db\_password","password")

.field("local\_db\_user","admin")

<span style="white-space:pre"> </span>.field("local\_db\_password","admin")

.endObject()

.startObject("index")

.field("name","test")

.field("type","test")

.field("bulk\_size","1000")

.field("bulk\_timeout","30")

.endObject()

.endObject()

).execute().actionGet();

本插件git地址：https://github.com/laigood/elasticsearch-river-mongodb

## 使用More like this实现基于内容的推荐

基于内容的推荐通常是给定一篇文档信息，然后给用户推荐与该文档相识的文档。Lucene的api中有实现查询文章相似度的接口，叫MoreLikeThis。Elasticsearch封装了该接口，通过Elasticsearch的More like this查询接口，我们可以非常方便的实现基于内容的推荐。

先看一个查询请求的json例子：

{

"more\_like\_this" : {

"fields" : ["title", "content"],

"like\_text" : "text like this one",

}

}

其中：

fields是要匹配的字段，如果不填的话默认是\_all字段

like\_text是匹配的文本。

除此之外还可以添加下面条件来调节结果

percent\_terms\_to\_match：匹配项（term）的百分比，默认是0.3

min\_term\_freq：一篇文档中一个词语至少出现次数，小于这个值的词将被忽略，默认是2

max\_query\_terms：一条查询语句中允许最多查询词语的个数，默认是25

stop\_words：设置停止词，匹配时会忽略停止词

min\_doc\_freq：一个词语最少在多少篇文档中出现，小于这个值的词会将被忽略，默认是无限制

max\_doc\_freq：一个词语最多在多少篇文档中出现，大于这个值的词会将被忽略，默认是无限制

min\_word\_len：最小的词语长度，默认是0

max\_word\_len：最多的词语长度，默认无限制

boost\_terms：设置词语权重，默认是1

boost：设置查询权重，默认是1

analyzer：设置使用的分词器，默认是使用该字段指定的分词器

下面介绍下如何用java api调用，一共有三种调用方式，不过本质上都是一样的，只不过是做了一些不同程度的封装。

MoreLikeThisRequestBuilder mlt = new MoreLikeThisRequestBuilder(client, "indexName", "indexType", "id");

mlt.setField("title");//匹配的字段

SearchResponse response = client.moreLikeThis(mlt.request()).actionGet();

这种是在查询与某个id的文档相似的文档。这个接口是直接在client那调用的，比较特殊。还有两种就是构造Query进行查询

MoreLikeThisQueryBuilder query = QueryBuilders.moreLikeThisQuery();

query.boost(1.0f).likeText("xxx").minTermFreq(10);

这里的boost、likeText方法完全和上面的参数对应的。下面这种就是把要匹配的字段作为参数传进来，参数和MoreLikeThisQueryBuilder是一样的。

MoreLikeThisFieldQueryBuilder query = QueryBuilders.moreLikeThisFieldQuery("fieldNmae");

# 高级配置

## 分片分布规则设置

分片分布是把索引分片分布到节点的过程。这个操作会在初次启动集群，副本分配，负载均衡，或增加删除节点时进行。

下面是一些与分片分布相关的设置：

cluster.routing.allocation.allow\_rebalance

设置根据集群中机器的状态来重新分配分片，可以设置为always, indices\_primaries\_active和indices\_all\_active，默认是设置成indices\_all\_active来减少集群初始启动时机器之间的交互。

cluster.routing.allocation.cluster\_concurrent\_rebalance

设置在集群中最大允许同时进行分片分布的个数，默认为2，也就是说整个集群最多有两个分片在

进行重新分布。

cluster.routing.allocation.node\_initial\_primaries\_recoveries

设置指定初始每个节点。由于多数情况下是使用local的gateway，这应该会更快，

cluster.routing.allocation.node\_concurrent\_recoveries

设置在节点中最大允许同时进行分片分布的个数，默认为2

cluster.routing.allocation.disable\_allocation

使主要分片或副本的分布失效。要知道，如果主分片不存在（那个节点挂了）那么其副本仍然会被提升为主分片，这个设置只有在动态地使用集群更新设置api调用时才生效。

cluster.routing.allocation.disable\_replica\_allocation

使副本分布失效。和上一个设置一样，只有动态地使用集群更新设置api调用时才生效。

indices.recovery.concurrent\_streams

当从一个点（peer）恢复分片时当前节点最多允许的文件读取流的个数，默认为5

自定义分片分布规则

可以通过设置分片的分布规则来人为地影响分片的分布，下面是个例子：

假设我们有几个机架。当我们启动一个节点，我们可以设置一个叫rack\_id（其它名字也可以）的属性，例如下面设置：

node.rack\_id: rack\_one

上面这个例子设置了一个属性叫rack\_id，它的值为rack\_one。现在，我们要设置rack\_id作为分片分布规则的一个属性（在所有节点都要设置）。

cluster.routing.allocation.awareness.attributes: rack\_id

上面设置意味着rack\_id会用来作为分片分布的依据。例如：我们启动两个node.rack\_id设置rack\_one的节点，然后建立一个5个分片，一个副本的索引。这个索引就会完全分布在这两个节点上。如果再启动另外两个节点，node.rack\_id设置成rack\_two，分片会重新分布，但是一个分片和它的副本不会分配到同样rack\_id值的节点上。可以为分片分布规则设置多个属性，例如：

cluster.routing.allocation.awareness.attributes: rack\_id,zone

注意：当设置了分片分布属性时，如果集群中的节点没有设置其中任何一个属性，那么分片就不会分布到这个节点中。

强制分布规则

更多的时候，我们不想更多的副本被分布到相同分布规则属性值的一群节点上，那么，我们可以强制分片规则为一个指定的值。

例如，我们有一个分片规则属性叫zone，并且我们知道有两个zone，zone1和zone2.下面是设置：

cluster.routing.allocation.awareness.force.zone.values: zone1,zone2

cluster.routing.allocation.awareness.attributes: zone

现在我们启动两个node.zone设置成zone1的节点，然后创建一个5个分片，一个副本的索引。索引建立完成后只有5个分片（没有副本），只有当我们启动node.zone设置成zone2的节点时，副本才会分配到那节点上。

分片分布过滤

允许通过include/exclude过滤器来控制分片的分布。这些过滤器可以设置在索引级别上或集群级别上。下面是个索引级别上的例子:

假如我们有四个节点，每个节点都有一个叫tag（可以是任何名字）的属性。每个节点都指定一个tag的值。如：节点一设置成node.tag: value1，节点二设置成node.tag: value2，如此类推。我们可以创建一个索引然后只把它分布到tag值为value1和value2的节点中，可以通过设置

index.routing.allocation.include.tag 为value1,value2达到这样的效果，如：

curl -XPUT localhost:9200/test/\_settings -d '{

"index.routing.allocation.include.tag" : "value1,value2"

}'

与此相反，通过设置index.routing.allocation.exclude.tag为value3，我们也可以创建一个索引让其分布在除了tag设置为value3的所有节点中，如：

curl -XPUT localhost:9200/test/\_settings -d '{

"index.routing.allocation.exclude.tag" : "value3"

}'

include或exclude过滤器的值都会使用通配符来匹配，如value\*。一个特别的属性名是\_ip，它可以用来匹配节点的ip地址。

显然，一个节点可能拥有多个属性值，所有属性的名字和值都在配置文件中配置。如，下面是多个节点的配置：

node.group1: group1\_value1

node.group2: group2\_value4

同样的方法，include和exclude也可以设置多个值，如：

curl -XPUT localhost:9200/test/\_settings -d '{

"index.routing.allocation.include.group1" : "xxx"

"index.routing.allocation.include.group2" : "yyy",

"index.routing.allocation.exclude.group3" : "zzz",

}'

上面的设置可以通过索引更新的api实时更新到索引上，允许实时移动索引分片。

集群范围的过滤器也可以定义，可以通过集群更新api实时更新到集群上。这些设置可以用来做让一些节点退出集群的操作。下面是通过ip地址去掉一个节点的操作：

curl -XPUT localhost:9200/\_cluster/settings -d '{

"transient" : {

"cluster.routing.allocation.exclude.\_ip" : "10.0.0.1"

}

}'

## 线程池设置

一个Elasticsearch节点会有多个线程池，但重要的是下面四个：

索引（index）：主要是索引数据和删除数据操作（默认是cached类型）

搜索（search）：主要是获取，统计和搜索操作（默认是cached类型）

批量操作（bulk）：主要是对索引的批量操作（默认是cached类型）

更新（refresh）：主要是更新操作（默认是cached类型）

可以通过给设置一个参数来改变线程池的类型（type），例如，把索引的线程池改成blocking类型：

min: 1

size: 30

wait\_time: 30s

下面是三种可以设置的线程池的类型

**cache**

cache线程池是一个无限大小的线程池，如果有很请求的话都会创建很多线程，下面是个例子：

threadpool:

index:

type: cached

**fixed**

fixed线程池保持固定个数的线程来处理请求队列。

size参数设置线程的个数，默认设置是cpu核心数的5倍

queue\_size可以控制待处理请求队列的大小。默认是设置为-1，意味着无限制。当一个请求到来但队列满了的时候，reject\_policy参数可以控制它的行为。默认是abort，会使那个请求失败。设置成caller会使该请求在io线程中执行。

threadpool:

index:

type: fixed

size: 30

queue: 1000

reject\_policy: caller

**blocking**

blocking线程池允许设置一个最小值（min，默认为1）和线程池大小（size，默认为cpu核心数的5倍）。它也有一个等待队列，队列的大小（queue\_size ）默认是1000，当这队列满了的时候。它会根据定好的等待时间（wait\_time，默认是60秒）来调用io线程，如果没有执行就会报错。

threadpool:

index:

type: blocking

min: 1

size: 30

wait\_time: 30s

## 虚拟机配置

引言：

今天，事情终于发生了。Java6（Mustang），是2006年早些时候出来的，至今仍然应用在众多生产环境中，现在终于走到了尽头。已经没有什么理由阻止迁移到Java7(Dolphin)上了。

这也促使我想写一篇关于在ElasticSearch上配置Java6和7的细微差异的博文。

Elasticsearch对Java虚拟机进行了预先的配置。通常情况下，因为这些配置的选择还是很谨慎的，所以你不需要太关心，并且你能立刻使用ElasticSearch。

但是，当你监视ElasticSearch节点内存时，你可能尝试修改一些配置。这些修改是否会改善你的处境？

这篇博文尝试揭开Elasticsearch配置的神秘面纱，并且讨论最常见的调整。最终，会给出一些推荐的配置调整。

Elasticsearch JVM 配置概览：

这些是Elasticsearch 0.19.11版本的默认配置。

|  |  |  |
| --- | --- | --- |
| JVM参数 | Elasticsearch默认值 | Environment变量 |
| -Xms | 256m | ES\_MIN\_MEM |
| -Xmx | 1g | ES\_MAX\_MEM |
| -Xms and -Xmx |  | ES\_HEAP\_SIZE |
| -Xmn |  | ES\_HEAP\_NEWSIZE |
| -XX:MaxDirectMemorySize |  | ES\_DIRECT\_SIZE |
| -Xss | 256k |  |
| -XX:UseParNewGC | + |  |
| -XX:UseConcMarkSweepGC | + |  |
| -XX:CMSInitiatingOccupancyFraction | 75 |  |
| -XX:UseCMSInitiatingOccupancyOnly | + |  |
| -XX:UseCondCardMark | (commented out) |  |

首先你注意到的是，Elasticsearch预留了256M到1GB的堆内存。

这个设置适用于开发和演示环境。开发人员只需要简单的解压发行包，再执行./bin/elasticsearch -f就完成了Elasticsearch的安装。当然这点对于开发来说非常棒，并且在很多场景下都能工作，但是当你需要更多内存来降低 Elasticsearch负载的时候就不行了，你需要比2GB RAM更多的可用内存。

ES\_MIN\_MEM/ES\_MAX\_MEM是控制堆大小的配置。新的ES\_HEAP\_SIZE变量是一个更为便利的选择，因为将堆的初始大小和最大值设为相同。也推荐在分配堆内存时尽可能不要用内存的碎片。内存碎片对于性能优化来说非常不利。

ES\_HEAP\_NEWSIZE是可选参数，它控制堆的子集大小，也就是新生代的大小。

ES\_DIRECT\_SIZE控制本机直接内存大小，即JVM管理NIO框架中使用的数据区域大小。本机直接内存可以被映射到虚拟地址空间上，这样在64位的机器上更高效，因为可以规避文件系统缓冲。Elasticsearch对本机直接内存没有限制(可能导致OOM)。

由于历史原因Java虚拟机有多个垃圾收集器。可以通过以下的JVM参数组合启用：

|  |  |
| --- | --- |
| JVM parameter | Garbage collector |
| -XX:+UseSerialGC | serial collector |
| -XX:+UseParallelGC | parallel collector |
| -XX:+UseParallelOldGC | Parallel compacting collector |
| -XX:+UseConcMarkSweepGC | Concurrent-Mark-Sweep (CMS) collector |
| -XX:+UseG1GC | Garbage-First collector (G1) |

UseParNewGC和UseConcMarkSweepGC组合启用垃圾收集器的并发多线程模式。UseConcMarkSweepGC自动选择UseParNewGC模式并禁用串行收集器（Serial collector）。在Java6中这是默认行为。

CMSInitiatingOccupancyFraction提炼了一种CMS（Concurrent-Mark-Sweep）垃圾收集设置；它将旧生代触发垃圾收集的阀值设为75.旧生代的大小是堆大小减去新生代大小。这告诉JVM当堆内容达到75%时启用垃圾收集。这是个估计的值，因为越小的堆可能需要越早启动GC。

UseCondCardMark将在垃圾收集器的card table使用时，在marking之前进行额外的判断，避免冗余的store操作。UseCondCardMark不影响Garbage-First收 集器。强烈推荐在高并发场景下配置这个参数（规避card table marking技术在高并发场景下的降低吞吐量的负面作用）。在ElasticSearch中，这个参数是被注释掉的。

有些配置可以参考诸如Apache Cassandra项目，他们在JVM上有类似的需求。

总而言之，ElastciSearch配置上推荐：

1. 不采用自动的堆内存配置，将堆大小默认最大值设为1GB

2.调整触发垃圾收集的阀值，比如将gc设为75%堆大小的时候触发，这样不会影响性能。

3.禁用Java7默认的G1收集器，前提是你的ElasticSearch跑在Java7u4以上的版本上。

JVM进程的内存结果

JVM内存由几部分组成：

Java代码本身：包括内部代码、数据、接口，调试和监控代理或者字节码指令

非堆内存：用于加载类

栈内存：用于为每个线程存储本地变量和操作数

堆内存：用于存放对象引用和对象本身

直接缓冲区：用于缓冲I/O数据

堆内存的大小设置非常重要，因为Java的运行依赖于合理的堆大小，并且JVM需要从操作系统那获取有限的堆内存，用于支撑整个JVM生命周期。

如果堆太小，垃圾回收就会频繁发生，发生OOM的几率会很大。

如果堆太大，垃圾回收会延迟，但是一旦回收，就需要处理大量的存活堆数据。并且，操作系统的压力也会变大，因为JVM进程需要更大的堆，产生换页的可能性就会提高。

注意，使用CMS垃圾收集器，Java不会把内存还给操作系统，因此配置合理的堆初始值和最大值就非常重要。

非堆内存由Java应用自动分配。没有什么参数控制这里的大小，这是由Java应用程序代码自己决定的。

栈内存在每个线程中分配，在Elasticsearch中，每个线程大小必须由128K增加到256K，因为Java7比Java6需要更大的栈内存 ，这是由于Java7支持新的编程语言特征来利用栈空间。比如，引入了continuations模型，编程语言的一个著名概念。Continuations模型对于

协同程序、绿色线程（green thread）、纤程（fiber）非常有用 。当实现非阻塞I/O时，一个大的优势是，代码可以根据线程实际使用情况编写，但是运行时仍然在后台采用非 阻塞I/O。Elasticsearch使用了多个线程池，因为Netty I/O框架和Guava是Elasticsearch的基础组件，因此在用Java7时，可以考虑进一步挖掘优化线程的特性。

发挥增加栈空间大小的优势还是有挑战的，因为不同的操作系统、不同的CPU架构，甚至在不同的JVM版本之间，栈空间的消耗不是容易比较的。取决于CPU 架构和操作系统，JVM的栈空间大小是内建的。他们是否在所有场景下都适合？例如Sloaris Sparc 64位的JVM Xss默认为512K，因为有更大地址指针，Sloaris X86为320K。Linux降为256K。Windows 32位Java6默认320K，Windows 64位则为1024K。

大堆的挑战

今天，几GB的内存是很常见的。但是在不久以前，系统管理员还在为多几G的内存需求泪流满面。

Java垃圾收集器是随着2006年的Java6的出现而显著改进的。从那以后，可以并发执行多任务，并且减少了GC停顿几率： stop - the - world阶段。CMS算法是革命性的，多任务，并发， 不需要移动的GC。但是不幸的是，对于堆的存活数据量来说，它是不可扩展的。Prateek Khanna 和 Aaron Morton给出了CMS垃圾收集器能够处理的堆规模的数字。

避免Stop-the-world阶段

我们已经学习了Elasticsearch如何配置CMS垃圾收集器。但这并不能组织长时间的GC停顿，它只是降低了发生的几率。CMS是一个低停顿几率 的收集器，但是仍然有一些边界情况。当堆上有MB级别的大数组，或者其他一些特殊的场景，CMS可能比预期要花费更多的时间。

MB级别数组的创建在Lucene segment-based索引合并时是很常见的。如果你希望降低CMS的额外负载，就需要调整Lucene合并阶段的段数量，使用参数index.merge.policy.segments\_per\_tier

减少换页

大堆的风险在于内存压力上。注意，如果Java JVM在处理大堆时，这部分内存对于系统其它部分来说是不可用的。如果内存吃紧，操作系统会进行换页，并且，在紧急情况下，当所有其他方式回收内存都失败 时，会强制杀掉进程。如果换页发生，整个系统的性能会下降，自然GC的性能也跟着下降。所以，不要给堆分配太多的内存。

垃圾收集器的选择

从Java JDK 7u4开始，Garbage-First（G1）收集器是Java7默认的垃圾收集器。它适用于多核的机器以及大内存。它一方面降低了停顿时间，另一方面 增加了停顿的次数。整个堆的操作，例如全局标记，是在应用线程中并发执行的。这会防止随着堆或存活数据大小的变化，中断时间也成比例的变化。

G1收集器目标是获取更高的吞吐量，而不是速度。在以下情况下，它能运行的很好：

1. 存活数据占用了超过50%的Java堆

2. 对象分配比例或者promotion会有明显的变化

3. 不希望gc或者compaction停顿时间长（超过0.5至1s）

注意，如果使用G1垃圾收集器，堆不再使用的内存可能会被归还给操作系统

G1垃圾收集器的不足是CPU使用率越高，应用性能越差。因此，如果在内存足够和CPU能力一般的情况下，CMS可能更胜一筹。

对于Elasticsearch来说，G1意味着没有长时间的stop-the-world阶段，以及更灵活的内存管理，因为buffer memory和系统I/O缓存能更充分的利用机器内存资源。代价就是小成本的最大化性能，因为G1利用了更多CPU资源。

性能调优策略

你读这篇博文因为你希望在性能调优上得到一些启示：

1. 清楚了解你的性能目标。你希望最大化速度，还是最大化吞吐量？

2. 记录任何事情（log everything），收集统计数据，阅读日志、分析事件来诊断配置

3. 选择你调整的目标（最大化性能还是最大化吞吐量）

4. 计划你的调整

5. 应用你的新配置

6. 监控新配置后的系统

7. 如果新配置没有改善你的处境，重复上面的一系列动作，反复尝试

Elasticsearch垃圾收集日志格式

Elasticsearch长时间GC下warns级别的日志如下所示：

[2012-11-26 18:13:53,166][WARN ][monitor.jvm              ] [Ectokid] [gc][ParNew][1135087][11248] duration [2.6m], collections [1]/[2.7m], total [2.6m]/[6.8m], memory [2.4gb]->[2.3gb]/[3.8gb], all\_pools {[Code Cache] [13.7mb]->[13.7mb]/[48mb]}{[Par Eden Space] [109.6mb]->[15.4mb]/[1gb]}{[Par Survivor Space] [136.5mb]->[0b]/[136.5mb]}{[CMS Old Gen] [2.1gb]->[2.3gb]/[2.6gb]}{[CMS Perm Gen] [35.1mb]->[34.9mb]/[82mb]}

JvmMonitorService类中有相关的使用方式：

|  |  |
| --- | --- |
| Logfile | Explanation |
| gc | 运行中的gc |
| ParNew | new parallel garbage collector |
| duration 2.6m | gc时间为2.6分钟 |
| collections [1]/[2.7m] | 在跑一个收集，共花2.7分钟 |
| memory [2.4gb]->[2.3gb]/[3.8gb] | 内存消耗, 开始是2.4gb, 现在是2.3gb, 共有3.8gb内存 |
| Code Cache [13.7mb]->[13.7mb]/[48mb] | code cache占用内存 |
| Par Eden Space [109.6mb]->[15.4mb]/[1gb] | Par Eden Space占用内存 |
| Par Survivor Space [136.5mb]->[0b]/[136.5mb] | Par Survivor Space占用内存 |
| CMS Old Gen [2.1gb]->[2.3gb]/[2.6gb] | CMS Old Gen占用内存 |
| CMS Perm Gen [35.1mb]->[34.9mb]/[82mb] | CMS Perm Gen占用内存 |

JvmMonitorSer

一些建议

1. 不要在Java 6u22之前的发布版本中跑Elasticsearch。有内存方面的bug。那些超过两三年的bug和缺陷会妨碍Elasticsearch的正常运行。与旧的OpenJDK 6相比，更推荐Sun/Oracle的版本，因为后者修复了很多bug。

2. 放弃Java6，转到Java7。Oracle宣称Java6更新到2013年2月结束。考虑到Elasticsearch还是一个相对新的软件，应该使用更新的技术来提升性能。尽量从JVM中挤压性能。检查操作系统的版本。在最新版本的操作系统中运行，有助于你的Java运行环境达到最佳性能。

3. 定期更新Java运行环境。平均一个季度一次。告诉sa你需要及时更新Java版本，以获取Java性能的提升。

4. 从小到大。先在Elasticsearch单节点上进行开发。但是不要忘了Elasticsearch分布式的强大功能。单节点不能模拟生产环境的特征，至少需要3个节点进行开发测试。

5. 在调整JVM之前先做一下性能测试。对你的系统建立性能基线。调整测试时候的节点数量。如果索引时候负载很高，你可能需要降低Elasticsearch索引时候占用的堆大小，通过index.merge.policy.segments\_per\_tierparameter参数调整段的合并。

6. 调整前清楚你的性能目标，然后决定是调整速度还是吞吐量。

7. 启用日志以便更好的进行诊断。在优化系统前进行小心的评估。

8. 如果使用CMS垃圾收集器，你可能需要加上合理的 -XX:CMSWaitDuration 参数。

9. 如果你的堆超过6-8GB，超过了CMS垃圾收集器设计容量，你会遇到长时间的stop-the-world阶段，你有几个方案：调整CMSInitiatingOccupancyFraction参数降低长时间GC的几率减少最大堆的大小；启用G1垃圾收集器。

10. 学习垃圾收集调优艺术。如果你想精通的话，列出可用的JVM选项，在java命令中加入java -XX:+UnlockDiagnosticVMOptions -XX:+PrintFlagsFinal -version，然后调优。

# 源码分析

## 使用Guice进行依赖注入与模块化系统

elasticsearch使用google开源的依赖注入框架guice，这个项目号称比spring快100倍，具体性能没有测试过，不过由于其代码比较简洁，比spring快很有可能，是不是快那么多就不知道了。先介绍下guice的基本使用方法。

elasticsearch是直接把guice的源码放到自己的包内（es把很多开源项目的代码都直接集成到自己项目中，省得依赖一堆的jar包，也使es的jar包达到差不多10M），在org.elasticsearch.common.inject目录下。

Guice主要是使用Module这个接口来确定各个接口和它们对应的实现。这个Module是个单例的抽象接口，通过bind(A).to(B)来绑定指定实例到这个模块中，下面看下Guice官方文档中的例子：

public class BillingModule extends AbstractModule {

@Override

protected void configure() {

bind(TransactionLog.class).to(DatabaseTransactionLog.class);

bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);

bind(BillingService.class).to(RealBillingService.class);

}

}

上面定义了一个订单模块，扩展AbstractModule这个抽象类。这个模块里面有三个实例：交易日志、支付过程和账单服务。通过bind("interface").to("implement")来使接口和实现绑定。

public class RealBillingService implements BillingService {

private final CreditCardProcessor processor;

private final TransactionLog transactionLog;

@Inject

public RealBillingService(CreditCardProcessor processor,

TransactionLog transactionLog) {

this.processor = processor;

this.transactionLog = transactionLog;

}

public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {

try {

ChargeResult result = processor.charge(creditCard, order.getAmount());

transactionLog.logChargeResult(result);

return result.wasSuccessful()

? Receipt.forSuccessfulCharge(order.getAmount())

: Receipt.forDeclinedCharge(result.getDeclineMessage());

} catch (UnreachableException e) {

transactionLog.logConnectException(e);

return Receipt.forSystemFailure(e.getMessage());

}

}

}

上面类是BillService接口的实现类。其中要注意的就是@Inject这个注释。Guice的Injector类会扫描@Inject这类注释，找到方法中传入参数的实例进行注入。如上面的CreditCardLog和TransactionLog。

public static void main(String[] args) {

Injector injector = Guice.createInjector(new BillingModule());

BillingService billingService = injector.getInstance(BillingService.class);

...

}

最后，在main方法中使用Injector进行注入与获取实例。这就是使用Guice进行依赖注入的一个简单例子。elasticsearch里面的组件基本都是用上面的方式进行模块化管理，elasticsearch对guice进行了简单的封装，通过ModulesBuilder类构建es的模块，一个es节点包括下面模块：

PluginsModule：插件模块

SettingsModule：设置参数模块

NodeModule：节点模块

NetworkModule：网络模块

NodeCacheModule：缓存模块

ScriptModule：脚本模块

JmxModule：jmx模块

EnvironmentModule：环境模块

NodeEnvironmentModule：节点环境模块

ClusterNameModule：集群名模块

ThreadPoolModule：线程池模块

DiscoveryModule：自动发现模块

ClusterModule：集群模块

RestModule：rest模块

TransportModule：tcp模块

HttpServerModule：http模块

RiversModule：river模块

IndicesModule：索引模块

SearchModule：搜索模块

ActionModule：行为模块

MonitorModule：监控模块

GatewayModule：持久化模块

NodeClientModule：客户端模块

接下来的文章会分析其中一些重要的模块。

## 索引过程源码概要分析

elasticsearch的索引逻辑简单分析，这里只是理清主要的脉络，一些细节方面以后的文章或会阐述。

假如通过java api来调用es的索引接口，先是构造成一个json串（es里表示为XContent，是对要处理的内容进行抽象），在IndexRequest里面指定要索引文档到那个索引库（index）、其类型（type）还有文档的id，如果没有指定文档的id，es会通过UUID工具自动生成一个uuid，代码在IndexRequest的process方法内。

if (allowIdGeneration) {

if (id == null) {

id(UUID.randomBase64UUID());

opType(IndexRequest.OpType.CREATE);

}

}

然后使用封装过netty的TransportService通过tcp协议发送请求到es服务器（rest的话就是通过http协议）。

服务器获得TransportAction后解析索引请求（TransportShardReplicationOperationAction）。到AsyncShardOperationAction.start()方法开始进行分片操作，先读取集群状态，把目标索引及其分片信息提取出来，根据索引数据的id、类型以及索引分片信息进行哈希取模，确定把该条数据分配到那个分片。

private int shardId(ClusterState clusterState, String index, String type, @Nullable String id, @Nullable String routing) {

if (routing == null) {

if (!useType) {

return Math.abs(hash(id) % indexMetaData(clusterState, index).numberOfShards());

} else {

return Math.abs(hash(type, id) % indexMetaData(clusterState, index).numberOfShards());

}

}

return Math.abs(hash(routing) % indexMetaData(clusterState, index).numberOfShards());

}

并找到数据要分配到的分片的主分片，先把索引请求提交到主分片处理（TransportIndexAction.shardOperationOnPrimary）。

判断是否必须要指定routing值

MappingMetaData mappingMd = clusterState.metaData().index(request.index()).mappingOrDefault(request.type());

if (mappingMd != null && mappingMd.routing().required()) {

if (request.routing() == null) {

throw new RoutingMissingException(request.index(), request.type(), request.id());

}

}

判断索引操作的类型，索引操作有两种，一种是INDEX，当要索引的文档id已经存在时，不会覆盖原来的文档，只是更新原来文档。一种是CREATE，当索引文档id存在时，会抛出该文档已存在的错误。

if (request.opType() == IndexRequest.OpType.INDEX)

调用InternalIndexShard进行索引操作

Engine.Index index = indexShard.prepareIndex(sourceToParse)

.version(request.version())

.versionType(request.versionType())

.origin(Engine.Operation.Origin.PRIMARY);

indexShard.index(index);

通过（InternalIndexShard）查找与请求索引数据类型（type）相符的mapping。对要索引的json字符串进行解析，根据mapping转换为对应的解析结果ParsedDocument 。

public Engine.Index prepareIndex(SourceToParse source) throws ElasticSearchException {

long startTime = System.nanoTime();

DocumentMapper docMapper = mapperService.documentMapperWithAutoCreate(source.type());

ParsedDocument doc = docMapper.parse(source);

return new Engine.Index(docMapper, docMapper.uidMapper().term(doc.uid()), doc).startTime(startTime);

}

最后调用RobinEngine中的相关方法(添加或修改)对底层lucene进行操作，这里是写入到lucene的内存索引中（RobinEngine.innerIndex）。

if (currentVersion == -1) {

// document does not exists, we can optimize for create

if (index.docs().size() > 1) {

writer.addDocuments(index.docs(), index.analyzer());

} else {

writer.addDocument(index.docs().get(0), index.analyzer());

}

} else {

if (index.docs().size() > 1) {

writer.updateDocuments(index.uid(), index.docs(), index.analyzer());

} else {

writer.updateDocument(index.uid(), index.docs().get(0), index.analyzer());

}

}

写入内存索引后还会写入到Translog（Translog是对索引的操作日志，会记录没有持久化的操作）中，防止flush前断电导致索引数据丢失。

Translog.Location translogLocation = translog.add(new Translog.Create(create));

主分片索引请求完就把请求发给副本进行索引操作。最后把成功信息返回给客户端。

# 问题解决

## 索引修复

在使用基于lucene的各类搜索引擎（如：elasticsearch、solr）时，有可能出现类似如下的错误：

Caused by: java.io.EOFException: read past EOF: NIOFSIndexInput(path="/usr/local/sas/escluster/data/cluster/nodes/0/indices/index/5/index/\_59ct.fdt")

这是由lucene的索引损坏了导致的，引起的原因有很多种，比如：复制不完整，硬盘损坏等。

由于elasticsearch也是基于lucene的，所以lucene的一些小工具elasticsearch也可以用到，比如luke之类的。lucene里面也有很多比较好用的工具，比如下面要介绍的CheckIndex。它在lucene-core jar包的org.apache.lucene.index目录下。它的功能是检查索引的的健康情况和修复索引。如果检查出某些segments有错误，可以通过-fix参数执行修复操作，修复的过程就是创建一个新的segments，把所有引用错误segments的索引数据删除。

使用方法,先定位到es的lib目录下

cd es\_home/lib

运行一下命令检查索引

java -cp lucene-core-3.6.1.jar -ea:org.apache.lucene... org.apache.lucene.index.CheckIndex /usr/local/sas/escluster/data/cluster/nodes/0/indices/index/5/index/

检测结果如下：

Segments file=segments\_2cg numSegments=26 version=3.6.1 format=FORMAT\_3\_1 [Lucene 3.1+] userData={translog\_id=1347536741715}

1 of 26: name=\_59ct docCount=4711242

compound=false

hasProx=true

numFiles=9

size (MB)=6,233.694

diagnostics = {mergeFactor=13, os.version=2.6.32-71.el6.x86\_64, os=Linux, lucene.version=3.6.1 1362471 - thetaphi - 2012-07-17 12:40:12, source=merge, os.arch=amd64, mergeMaxNumSegments=-1, java.version=1.6.0\_24, java.vendor=Sun Microsystems Inc.}

has deletions [delFileName=\_59ct\_1b.del]

test: open reader.........OK [3107 deleted docs]

test: fields..............OK [25 fields]

test: field norms.........OK [10 fields]

test: terms, freq, prox...OK [36504908 terms; 617641081 terms/docs pairs; 742052507 tokens]

test: stored fields.......ERROR [read past EOF: MMapIndexInput(path="/usr/local/sas/escluster/data/cluster/nodes/0/indices/index/5/index/\_59ct.fdt")]

java.io.EOFException: read past EOF: MMapIndexInput(path="/usr/local/sas/escluster/data/cluster/nodes/0/indices/index/5/index/\_59ct.fdt")

at org.apache.lucene.store.MMapDirectory$MMapIndexInput.readBytes(MMapDirectory.java:307)

at org.apache.lucene.index.FieldsReader.addField(FieldsReader.java:400)

at org.apache.lucene.index.FieldsReader.doc(FieldsReader.java:253)

at org.apache.lucene.index.SegmentReader.document(SegmentReader.java:492)

at org.apache.lucene.index.IndexReader.document(IndexReader.java:1138)

at org.apache.lucene.index.CheckIndex.testStoredFields(CheckIndex.java:852)

at org.apache.lucene.index.CheckIndex.checkIndex(CheckIndex.java:581)

at org.apache.lucene.index.CheckIndex.main(CheckIndex.java:1064)

test: term vectors........OK [0 total vector count; avg 0 term/freq vector fields per doc]

FAILED

WARNING: fixIndex() would remove reference to this segment; full exception:

java.lang.RuntimeException: Stored Field test failed

at org.apache.lucene.index.CheckIndex.checkIndex(CheckIndex.java:593)

at org.apache.lucene.index.CheckIndex.main(CheckIndex.java:1064)

WARNING: 1 broken segments (containing 4708135 documents) detected

WARNING: 4708135 documents will be lost

在检查结果中可以看到，分片5的\_59ct.fdt索引文件损坏，.fdt文件主要存储lucene索引中存储的fields，所以在检查test: stored fields时出错。

下面的警告是说有一个损坏了的segment，里面有4708135个文档。

在原来的命令基础上加上-fix参数可以进行修复索引操作（ps：在进行修改前最好对要修复的索引进行备份，不要在正在执行写操作的索引上执行修复。）

java -cp lucene-core-3.6.1.jar -ea:org.apache.lucene... org.apache.lucene.index.CheckIndex /usr/local/sas/escluster/data/cluster/nodes/0/indices/index/5/index/ -fix

执行完后会在原来检查完毕的信息后面会出现以下信息。

NOTE: will write new segments file in 5 seconds; this will remove 4708135 docs from the index. THIS IS YOUR LAST CHANCE TO CTRL+C!

5...

4...

3...

2...

1...

Writing...

OK

Wrote new segments file "segments\_2ch"

表示修复完成，移除了4708135个损坏文档。

损坏了4708135个文档，是挺多的了，对于这种损坏文档数太多的情况，一就是直接从之前备份的数据中恢复（如果有的话），还有就是读取索引，记录损坏文档的id，进行修复后重新索引。

## 其他问题

### 由gc引起节点脱离集群

因为gc时会使jvm停止工作，如果某个节点gc时间过长，master ping3次（zen discovery默认ping失败重试3次）不通后就会把该节点剔除出集群，从而导致索引进行重新分配。

解决方法：

（1）优化gc，减少gc时间。（2）调大zen discovery的重试次数（es参数：ping\_retries）和超时时间（es参数：ping\_timeout）。后来发现根本原因是有个节点的系统所在硬盘满了。导致系统性能下降。

### out of memory错误

因为默认情况下es对字段数据缓存（Field Data Cache）大小是无限制的，查询时会把字段值放到内存，特别是facet查询，对内存要求非常高，它会把结果都放在内存，然后进行排序等操作，一直使用内存，直到内存用完，当内存不够用时就有可能出现out of memory错误。

解决方法：

（1）设置es的缓存类型为Soft Reference，它的主要特点是据有较强的引用功能。只有当内存不够的时候，才进行回收这类内存，因此在内存足够的时候，它们通常不被回收。另外，这些引 用对象还能保证在Java抛出OutOfMemory 异常之前，被设置为null。它可以用于实现一些常用图片的缓存，实现Cache的功能，保证最大限度的使用内存而不引起OutOfMemory。在es的配置文件加上index.cache.field.type: soft即可。

（2）设置es最大缓存数据条数和缓存失效时间，通过设置index.cache.field.max\_size: 50000来把缓存field的最大值设置为50000，设置index.cache.field.expire: 10m把过期时间设置成10分钟。

### 无法创建本地线程问题

es恢复时报错： RecoverFilesRecoveryException[[index][3] Failed to transfer [215] files with total size of [9.4gb]]; nested: OutOfMemoryError[unable to create new native thread]; ]]

刚开始以为是文件句柄数限制，但想到之前报的是too many open file这个错误，并且也把数据改大了。查资料得知一个进程的jvm进程的最大线程数为：虚拟内存/（堆栈大小\*1024\*1024），也就是说虚拟内存越大或堆栈越小，能创建的线程越多。重新设置后还是会报那这错，按理说可创建线程数完全够用了的，就想是不是系统的一些限制。后来在网上找到说是max user processes的问题，这个值默认是1024，这个参数单看名字是用户最大打开的进程数，但看官方说明，就是用户最多可创建线程数，因为一个进程最少有一个线程，所以间接影响到最大进程数。调大这个参数后就没有报这个错了。

解决方法：

（1）增大jvm的heap内存或降低xss堆栈大小（默认的是512K）。

（2）打开/etc/security/limits.d/90-nproc.conf，把soft nproc 1024这行的1024改大就行了。

### 集群状态为黄色时并发插入数据报错

[7]: index [index], type [index], id [1569133], message [UnavailableShardsException[[index][1] [4] shardIt, [2] active : Timeout waiting for [1m], request: org.elasticsearch.action.bulk.BulkShardRequest@5989fa07]]

这是错误信息，当时集群状态为黄色，即副本没有分配。当时副本设置为2，只有一个节点，当你设置的副本大于可分配的机器时，此时如果你插入数据就有可能报上面的错，因为es的写一致性默认是使用quorum，即quorum值必须大于（副本数/2+1），我这里2/2+1=2也就是说要要至少插入到两份索引中，由于只有一个节点，quorum等于1，所以只插入到主索引，副本找不到从而报上面那个错。

解决方法：（1）去掉没分配的副本。（2）把写一致性改成one，即只写入一份索引就行。

### 设置jvm锁住内存时启动警告

当设置bootstrap.mlockall: true时，启动es报警告Unknown mlockall error 0，因为linux系统默认能让进程锁住的内存为45k。

解决方法：设置为无限制，linux命令：ulimit -l unlimited

### 错误使用api导致集群卡死

其实这个是很低级的错误。功能就是更新一些数据，可能会对一些数据进行删除，但删除时同事使用了deleteByQuery这个接口，通过构造BoolQuery把要删除数据的id传进去，查出这些数据删除。但问题是BoolQuery最多只支持1024个条件，100个条件都已经很多了，所以这样的查询一下子就把es集群卡死了。

解决方法：用bulkRequest进行批量删除操作。

*(This is the last page)*