Gmall

ElasticSearch

# 一、基本概念

## 1、Near Realtime（NRT 近实时）

Elasticsearch 是一个近实时的搜索平台。这意味着从您索引一个文档开始直到它可以被查询时会有轻微的延迟时间（通常为一秒）。

Lucence;

## 2、Cluster（集群）

cluster（集群）是一个或者多个节点的集合，它们一起保存数据并且提供所有节点联合索引以及搜索功能。**集群存在一个唯一的名字身份且默认为 “elasticsearch”**。这个名字非常重要，因为如果节点安装时通过它自己的名字加入到集群中的话，那么一个节点只能是一个集群中的一部分。

## 3、Node（节点）

node（节点）是一个单独的服务器，它是集群的一部分，存储数据，参与集群中的索引和搜索功能。像一个集群一样，一个节点通过一个在它启动时默认分配的一个随机的 UUID（通用唯一标识符）名称来识别。如果您不想使用默认名称您也可自定义任何节点名称。这个名字是要识别网络中的服务器对应这在您的 Elasticsearch 集群节点管理的目的是很重要的。

## 4、Index（索引）

动词，相当于MySQL中的insert；

名词，相当于MySQL中的Database

## 5、Type（类型）

在 Index（索引）中，可以定义一个或多个类型。

类似于MySQL中的Table；每一种类型的数据放在一起；

## 6、Document（文档）

保存在某个索引（Index）下，某种类型（Type）的一个数据（Document），文档是JSON格式的，Document就像是MySQL中的某个Table里面的内容

## 7、Shards & Replicas（分片 & 副本）

索引可以存储大量数据，可以超过单个节点的硬件限制。例如，十亿个文档占用了 1TB 的磁盘空间的单个索引可能不适合放在单个节点的磁盘上，并且从单个节点服务请求会变得很慢。

**为了解决这个问题，Elasticsearch 提供了把 Index（索引）拆分到多个 Shard（分片）中的能力。在创建索引时，您可以简单的定义 Shard（分片）的数量。每个 Shard 本身就是一个 fully-functional（全功能的）和独立的 “Index（索引）”，（Shard）它可以存储在集群中的任何节点上。**

Sharding（分片）非常重要两个理由是 :

1）水平的拆分/扩展。

2）分布式和并行跨 Shard 操作（可能在多个节点），从而提高了性能/吞吐量。

每个索引可以被拆分成多个分片，一个索引可以设置 0 个（没有副本）或多个**副本**。开启副本后，**每个索引将有主分片（被复制的原始分片）和副本分片（主分片的副本）。**分片和副本的数量在索引被创建时都能够被指定。在创建索引后，您也可以在任何时候动态的改变副本的数量，但是不能够改变分片数量。

ES的使用方式；

1）、公司搭建好了es服务器；

2）、我们只需要给es服务器发http请求即可测试es

# 二、初步检索

## 1、\_cat

GET /\_cat/nodes：查看所有节点

GET /\_cat/health：查看es健康状况

GET /\_cat/master：查看主节点

GET /\_cat/indices：查看所有索引

## 2、索引一个文档

保存一个数据，保存在哪个索引的哪个类型下，指定用哪个唯一标识

PUT customer/external/1；在customer索引下的external类型下保存1号数据为

|  |
| --- |
| PUT customer/external/1{ "name": "John Doe"} |
| PUT和POST都可以，POST新增。如果不指定id，会自动生成id。指定id就会修改这个数据，并新增版本号PUT可以新增可以修改。PUT必须指定id；由于PUT需要指定id，我们一般都用来做修改操作，不指定id会报错。 |

## 3、查询文档

|  |
| --- |
| GET customer/external/1 |

## 4、更新文档

|  |
| --- |
| POST customer/external/1/\_update{ "doc":{ "name": "John Doew" }} |
| 或者PUT customer/external/1{ "name": "John Doe"}或者POST customer/external/1{ "name": "John Doe2"} |
| * 不同：POST操作会对比源文档数据，如果相同不会有什么操作，文档version不增加

PUT操作总会将数据重新保存并增加version版本；带\_update对比元数据如果一样就不进行任何操作。看场景； 对于大并发更新，不带update； 对于大并发查询偶尔更新，带update；对比更新，重新计算分配规则。 |
| * 更新同时增加属性
 |
| POST customer/external/1/\_update{ "doc": { "name": "Jane Doe", "age": 20 }} |
| * 简单脚本更新
 |
| POST customer/external/1/\_update{ "script" : "ctx.\_source.age += 5"} |

## 5、删除文档&索引

|  |
| --- |
| DELETE customer/external/1 |
| DELETE customer  |

## 6、bulk批量API

|  |
| --- |
| POST customer/external/\_bulk{"index":{"\_id":"1"}}{"name": "John Doe" }{"index":{"\_id":"2"}}{"name": "Jane Doe" } |
| 语法格式：{ action: { metadata }}\n{ request body }\n{ action: { metadata }}\n{ request body }\n |
| 复杂实例：POST /\_bulk{ "delete": { "\_index": "website", "\_type": "blog", "\_id": "123" }} { "create": { "\_index": "website", "\_type": "blog", "\_id": "123" }}{ "title": "My first blog post" }{ "index": { "\_index": "website", "\_type": "blog" }}{ "title": "My second blog post" }{ "update": { "\_index": "website", "\_type": "blog", "\_id": "123", "\_retry\_on\_conflict" : 3} }{ "doc" : {"title" : "My updated blog post"} } |

bulk API 以此按顺序执行所有的 action（动作）。如果一个单个的动作因任何原因而失败，它将继续处理它后面剩余的动作。当 bulk API 返回时，它将提供每个动作的状态（与发送的顺序相同），所以您可以检查是否一个指定的动作是不是失败了。

## 7、样本测试数据

我准备了一份顾客银行账户信息的虚构的 JSON 文档样本。每个文档都有下列的 schema（模式）:

|  |
| --- |
| { "account\_number": 0, "balance": 16623, "firstname": "Bradshaw", "lastname": "Mckenzie", "age": 29, "gender": "F", "address": "244 Columbus Place", "employer": "Euron", "email": "bradshawmckenzie@euron.com", "city": "Hobucken", "state": "CO"} |

<https://github.com/elastic/elasticsearch/blob/master/docs/src/test/resources/accounts.json?raw=true> 导入测试数据

POST bank/account/\_bulk

测试数据

# 三、进阶检索

## 1、SearchAPI

ES支持两种基本方式检索 :

* 一个是通过使用 REST request URI 发送搜索参数（uri+检索参数）
* 另一个是通过使用 REST request body 来发送它们（uri+请求体）

### 1）、检索信息

* 一切检索从\_search开始

|  |  |
| --- | --- |
| GET bank/\_search | 检索bank下所有信息，包括type和docs |
| GET bank/\_search?q=\*&sort=account\_number:asc | 请求参数方式检索 |
| 响应结果解释：took - Elasticsearch 执行搜索的时间（毫秒）time\_out - 告诉我们搜索是否超时\_shards - 告诉我们多少个分片被搜索了，以及统计了成功/失败的搜索分片hits - 搜索结果hits.total - 搜索结果hits.hits - 实际的搜索结果数组（默认为前 10 的文档）sort - 结果的排序 key（键）（没有则按 score 排序）score 和 max\_score –相关性得分和最高得分（全文检索用） |

* **uri+请求体进行检索**

|  |
| --- |
| GET bank/\_search{ "query": { "match\_all": {} }, "sort": [ { "account\_number": { "order": "desc" } } ]} |
| HTTP客户端工具（POSTMAN），get请求不能携带请求体，我们变为post也是一样的我们 **POST** 一个 **JSON** 风格的查询请求体到 **\_search** **API**。需要了解，一旦搜索的结果被返回，**Elasticsearch** 就完成了这次请求，并且不会维护任何服务端的资源或者结果的 **cursor**（游标） |

## 2、Query DSL

### 1）、基本语法格式

**Elasticsearch** 提供了一个可以执行查询的 **Json** 风格的 **DSL**（**domain-specific language** 领域特定语言）。这个被称为 [Query DSL](https://www.elastic.co/guide/en/elasticsearch/reference/5.0/query-dsl.html)。该查询语言非常全面，并且刚开始的时候感觉有点复杂，真正学好它的方法是从一些基础的示例开始的。

* 一个查询语句 的典型结构

{

 QUERY\_NAME: {

 ARGUMENT: VALUE,

 ARGUMENT: VALUE,...

 }

}

* 如果是针对某个字段，那么它的结构如下：

{

 QUERY\_NAME: {

 FIELD\_NAME: {

 ARGUMENT: VALUE,

 ARGUMENT: VALUE,...

 }

 }

}

|  |
| --- |
| GET bank/\_search{ "query": { "match\_all": {} }, "from": 0, "size": 5, "sort": [ { "account\_number": { "order": "desc" } } ]} |
| * query 定义如何查询，
* match\_all 查询类型【代表查询所有的所有】，es中可以在query中组合非常多的查询类型完成复杂查询
* 除了 query 参数之外，我们也可以**传递其它的参数以改变查询结果**。如sort，size
* from+size限定，完成分页功能
* sort排序，多字段排序，会在前序字段相等时后续字段内部排序，否则以前序为准
 |

### 2）、返回部分字段

|  |
| --- |
| GET bank/\_search{ "query": { "match\_all": {} }, "from": 0, "size": 5, "\_source": ["age","balance"]} |

### 3）、match【匹配查询】

* 基本类型（非字符串），精确匹配

|  |
| --- |
| GET bank/\_search{ "query": { "match": { "account\_number": "20" } }} |
| match返回account\_number=20的 |

* 字符串，全文检索

|  |
| --- |
| GET bank/\_search{ "query": { "match": { "address": "mill" } }} |
| 最终查询出address中包含mill单词的所有记录match当搜索字符串类型的时候，会进行全文检索，并且每条记录有相关性得分。 |

* 字符串，多个单词（分词+全文检索）

|  |
| --- |
| GET bank/\_search{ "query": { "match": { "address": "mill road" } }} |
| 最终查询出address中包含mill或者road或者mill road的所有记录，并给出相关性得分 |

### 4）、match\_phrase【短语匹配】

将需要匹配的值当成一个整体单词（不分词）进行检索

|  |
| --- |
| GET bank/\_search{ "query": { "match\_phrase": { "address": "mill road" } }} |
| 查出address中包含mill road的所有记录，并给出相关性得分 |

### 5）、multi\_match【多字段匹配】

|  |
| --- |
| GET bank/\_search{ "query": { "multi\_match": { "query": "mill", "fields": ["state","address"] } }} |
| state或者address包含mill |

### 6）、bool【复合查询】

bool用来做复合查询：

复合语句可以合并 任何 其它查询语句，包括复合语句，了解这一点是很重要的。这就意味着，复合语句之间可以互相嵌套，可以表达非常复杂的逻辑。

* **must：必须达到must列举的所有条件**

|  |
| --- |
| GET bank/\_search{ "query": { "bool": { "must": [ { "match": { "address": "mill" } }, { "match": { "gender": "M" } } ] } }} |

* **should：应该达到should列举的条件，如果达到会增加相关文档的评分**，并不会改变查询的结果。如果query中只有should且只有一种匹配规则，那么should的条件就会被作为默认匹配条件而去改变查询结果

|  |
| --- |
| GET bank/\_search{ "query": { "bool": { "must": [ { "match": { "address": "mill" } }, { "match": { "gender": "M" } } ], "should": [ {"match": { "address": "lane" }} ] } }} |

* **must\_not必须不是指定的情况**

|  |
| --- |
| GET bank/\_search{ "query": { "bool": { "must": [ { "match": { "address": "mill" } }, { "match": { "gender": "M" } } ], "should": [ {"match": { "address": "lane" }} ], "must\_not": [ {"match": { "email": "baluba.com" }} ] } }} |
| address包含mill，并且gender是M，如果address里面有lane最好不过，但是email必须不包含baluba.com |



### 6）、filter【结果过滤】

并不是所有的查询都需要产生分数，特别是那些仅用于 “**filtering**”（过滤）的文档。为了不计算分数 **Elasticsearch** 会自动检查场景并且优化查询的执行。

|  |
| --- |
| GET bank/\_search{ "query": { "bool": { "must": [ {"match": { "address": "mill"}} ], "filter": { "range": { "balance": { "gte": 10000, "lte": 20000 } } } } }} |

### 7）、aggregations（执行聚合）

聚合提供了从数据中分组和提取数据的能力。最简单的聚合方法大致等于 **SQL GROUP BY** 和 **SQL** 聚合函数。在 **Elasticsearch** 中，您有执行搜索返回 **hits**（命中结果），并且同时返回聚合结果，把一个响应中的所有 **hits**（命中结果）分隔开的能力。这是非常强大且有效的，您可以执行查询和多个聚合，并且在一次使用中得到各自的（任何一个的）返回结果，使用一次简洁和简化的 **API** 来避免网络往返。

* **搜索address中包含mill的所有人的年龄分布以及平均年龄，但不显示这些人的详情。**

|  |
| --- |
| GET bank/\_search{ "query": { "match": { "address": "mill" } }, "aggs": { "group\_by\_state": { "terms": { "field": "age" } }, "avg\_age": { "avg": { "field": "age" } } }, "size": 0} |
| size：0 不显示搜索数据aggs：执行聚合。聚合语法如下"aggs": { "aggs\_name这次聚合的名字，方便展示在结果集中": { "AGG\_TYPE聚合的类型（avg,term,terms）": {} } }, |

复杂：

按照年龄聚合，并且请求这些年龄段的这些人的平均薪资

|  |
| --- |
| GET bank/account/\_search{ "query": { "match\_all": {} }, "aggs": { "age\_avg": { "terms": { "field": "age", "size": 1000 }, "aggs": { "banlances\_avg": { "avg": { "field": "balance" } } } } } , "size": 1000} |

<http://www.cnblogs.com/duanxz/p/6528161.html>

复杂：查出所有年龄分布，并且这些年龄段中M的平均薪资和F的平均薪资以及这个年龄段的总体平均薪资

|  |
| --- |
| GET bank/account/\_search{ "query": { "match\_all": {} }, "aggs": { "age\_agg": { "terms": { "field": "age", "size": 100 }, "aggs": { "gender\_agg": { "terms": { "field": "gender.keyword", "size": 100 }, "aggs": { "balance\_avg": { "avg": { "field": "balance" } } } }, "balance\_avg":{ "avg": { "field": "balance" } } } } } , "size": 1000} |

复杂：

查出bank中所有高于平均薪资的这些男生以及他们的最高薪资和最低薪资，女生也一样。

|  |
| --- |
|  |

## 3、Mapping

### 1）、字段类型

|  |  |
| --- | --- |
|  |  |
|  |
|  |

### 2）、映射

Mapping（映射）

**Mapping是用来定义一个文档（document），以及它所包含的属性（field）是如何存储和索引的**。比如，使用mapping来定义：

* 哪些字符串属性应该被看做全文本属性（full text fields）。
* 哪些属性包含数字，日期或者地理位置。
* 文档中的所有属性是否都能被索引（\_all 配置）。
* 日期的格式。
* 自定义映射规则来执行动态添加属性。
* 查看mapping信息：

|  |
| --- |
| GET bank/\_mapping |

* 修改mapping信息

|  |
| --- |
| 需要在创建索引的时候指定映射<https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html> PUT my\_index{ "mappings": { "user": { "\_all": { "enabled": false }, "properties": { "title": { "type": "text" }, "name": { "type": "text" }, "age": { "type": "integer" }  } }, "blogpost": { "\_all": { "enabled": false }, "properties": { "title": { "type": "text" }, "body": { "type": "text" }, "user\_id": { "type": "keyword" }, "created": { "type": "date", "format": "strict\_date\_optional\_time||epoch\_millis" } } } }} |
| 自动猜测的映射类型graphic |

## 4、分词

一个**tokenizer**（分词器）接收一个字符流，将之分割为独立的 **tokens**（词元，通常是独立的单词），然后输出 **tokens** 流。

例如，[whitespace](https://www.elastic.co/guide/en/elasticsearch/reference/5.3/analysis-whitespace-tokenizer.html) **tokenizer** 遇到空白字符时分割文本。它会将文本 "**Quick brown fox!**" 分割为 [**Quick**, **brown**, **fox!**]。

该**tokenizer**（分词器）还负责记录各个 **term**（词条）的顺序或 **position** 位置（用于 **phrase**短语和 **word** **proximity**词近邻查询），以及 **term**（词条）所代表的原始 **word**（单词）的 **start**（起始）和 **end**（结束）的 **character offsets**（字符偏移量）（用于高亮显示搜索的内容）。

**Elasticsearch** 提供了很多内置的分词器，可以用来构建 [custom analyzers](https://www.elastic.co/guide/en/elasticsearch/reference/5.3/analysis-custom-analyzer.html)（自定义分词器）。

倒排索引？

|  |
| --- |
| 红海行动红海事件 |

搜索：红海行动

name:”红海特工行动’

name:”红海行动’

name:”红海大爆炸’

name:”行动红海’

|  |  |  |  |
| --- | --- | --- | --- |
| 红海(1,2,3,4) | 1 | 1 | 1 |
| 行动(1,2,4) | 1 | 1 |  |
| 事件() |  |  |  |
| 特工(1) |  |  |  |
| 大爆炸(3) |  |  |  |
|  |  |  |  |
| 得分 | 2/3 | 1 |  |

字符串越短而且匹配的越多分就越高；

### 1）、安装ik分词器

**注意：**不能用默认elasticsearch-plugin install xxx.zip 进行自动安装

<https://github.com/medcl/elasticsearch-analysis-ik/releases?after=v6.4.2> 对应es版本安装

|  |
| --- |
| 进入es容器内部 plugins目录wget <https://github.com/medcl/elasticsearch-analysis-ik/releases/download/v5.6.11/elasticsearch-analysis-ik-5.6.11.zip> |
| unzip 下载的文件 |
| rm –rf \*.zip |
| mv elasticsearch/ ik |
| 可以确认是否安装好了分词器cd ../binelasticsearch plugin list：即可列出系统的分词器 |

### 2）、测试分词器

使用默认

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

 请观察结果

 使用分词器

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

 请观察结果

 另外一个分词器

 ik\_max\_word

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

请观察结果

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

# 四、集群原理

## 1、集群安装

|  |
| --- |
| es1配置文件*# 开启跨域，为了让es-head可以访问*http.cors.enabled: truehttp.cors.allow-origin: "\*"*# 集群的名称*cluster.name: elasticsearch*# 节点的名称*node.name: es1*# 指定该节点是否有资格被选举成为master节点，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.master: true*# 允许该节点存储数据(默认开启)*node.data: true*# 允许任何ip访问*network.host: 0.0.0.0*# 通过这个ip列表进行节点发现，我这里配置的是各个容器的ip*discovery.zen.ping.unicast.hosts: ["192.168.159.128:9300","192.168.159.128:9301","192.168.159.128:9302"]*#如果没有这种设置,遭受网络故障的集群就有可能将集群分成两个独立的集群 – 导致脑裂 - 这将导致数据丢失*discovery.zen.minimum\_master\_nodes: 2 |
| es2-es3配置文件-----------------------es2-----------------------http.cors.enabled: truehttp.cors.allow-origin: "\*"cluster.name: elasticsearchnode.name: es2network.host: 0.0.0.0node.master: truenode.data: truediscovery.zen.ping.unicast.hosts: ["192.168.159.128:9300","192.168.159.128:9301","192.168.159.128:9302"]discovery.zen.minimum\_master\_nodes: 2-----------------------es3-----------------------http.cors.enabled: truehttp.cors.allow-origin: "\*"cluster.name: elasticsearchnode.name: es3network.host: 0.0.0.0node.master: truenode.data: truediscovery.zen.ping.unicast.hosts: ["192.168.159.128:9300","192.168.159.128:9301","192.168.159.128:9302"]discovery.zen.minimum\_master\_nodes: 2 |
| 集群化实例启动实例一：docker run --name es1 -p 9200:9200 -p 9300:9300 \-e ES\_JAVA\_OPTS="-Xms256m -Xmx256m" \-v /mydata/elasticsearch/conf/elasticsearch.yml:/usr/share/elasticsearch/config/elasticsearch.yml \-v /mydata/elasticsearch/data:/usr/share/elasticsearch/data -d elasticsearch:5.6.11实例二：docker run --name es2 -p 9201:9200 -p 9301:9300 \-e ES\_JAVA\_OPTS="-Xms256m -Xmx256m" \-v /mydata/elasticsearch2/conf/elasticsearch.yml:/usr/share/elasticsearch/config/elasticsearch.yml \-v /mydata/elasticsearch2/data:/usr/share/elasticsearch/data -d elasticsearch:5.6.11实例三：docker run --name es3 -p 9202:9200 -p 9302:9300 \-e ES\_JAVA\_OPTS="-Xms256m -Xmx256m" \-v /mydata/elasticsearch3/conf/elasticsearch.yml:/usr/share/elasticsearch/config/elasticsearch.yml \-v /mydata/elasticsearch3/data:/usr/share/elasticsearch/data -d elasticsearch:5.6.11需要修改linux的进程数限制vi /etc/sysctl.confvm.max\_map\_count=655360sysctl -p |

## 2、集群、分片原理

集群的目的是为了，高可用。

### 1）、单节点



一个运行中的 Elasticsearch 实例称为一个 节点，而集群是由一个或者多个拥有相同 cluster.name 配置的节点组成， 它们共同承担数据和负载的压力。当有节点加入集群中或者从集群中移除节点时，**集群将会重新平均分布所有的数据。**

当一个节点被选举成为 主 节点时， **它将负责管理集群范围内的所有变更**，例如增加、删除索引，或者增加、删除节点等。 而**主节点并不需要涉及到文档级别的变更和搜索等操作**，所以当集群只拥有一个主节点的情况下，即使流量的增加它也不会成为瓶颈。 任何节点都可以成为主节点。我们的示例集群就只有一个节点，所以它同时也成为了主节点。

作为用户，我们可以将请求发送到 集群中的任何节点 ，包括主节点。 每个节点都知道任意文档所处的位置，并且能够将我们的请求直接转发到存储我们所需文档的节点。 无论我们将请求发送到哪个节点，它都能负责从各个包含我们所需文档的节点收集回数据，并将最终结果返回給客户端。 Elasticsearch 对这一切的管理都是透明的。

### 2）、集群健康

GET /\_cluster/health

status 字段指示着当前集群在总体上是否工作正常。它的三种颜色含义如下：

green

所有的主分片和副本分片都正常运行。

yellow

所有的主分片都正常运行，但不是所有的副本分片都正常运行。

red

有主分片没能正常运行。

**如下图：拥有一个索引的单节点集群**



### 3）、添加故障转移

当集群中只有一个节点在运行时，意味着会有一个单点故障问题——没有冗余。 幸运的是，我们只需再**启动一个节点即可防止数据丢失。**

启动第二个节点

为了测试第二个节点启动后的情况，你可以在同一个目录内，完全依照启动第一个节点的方式来启动一个新节点（参考安装并运行 Elasticsearch）。多个节点可以共享同一个目录。

当你在同一台机器上启动了第二个节点时，只要它和第一个节点有同样的 cluster.name 配置，它就会自动发现集群并加入到其中。 但是在不同机器上启动节点的时候，为了加入到同一集群，你需要配置一个可连接到的单播主机列表。 详细信息请查看最好使用单播代替组播



当第二个节点加入到集群后，3个 副本分片 将会分配到这个节点上——每个主分片对应一个副本分片。 这意味着当集群内任何一个节点出现问题时，我们的数据都完好无损。

所有新近被索引的文档都将**会保存在主分片上**，然后被并行的复制到对应的副本分片上。这就保证了我们既可以从主分片又可以从副本分片上获得文档。

cluster-health 现在展示的状态为 green ，这表示所有6个分片（包括3个主分片和3个副本分片）都在正常运行。

### 4）、水平扩容



读操作——搜索和返回数据——可以同时被主分片 或 副本分片所处理，所以当你拥有越多的副本分片时，也将拥有越高的吞吐量。

在运行中的集群上是可以动态调整副本分片数目的 ，我们可以按需伸缩集群。让我们把副本数从默认的 1 增加到 2 ：

PUT /blogs/\_settings

{

 "number\_of\_replicas" : 2

}



### 5）、应对故障



我们关闭的节点是一个主节点。而集群必须拥有一个主节点来保证正常工作，所以发生的第一件事情就是选举一个新的主节点： Node 2 。

在我们关闭 Node 1 的同时也失去了主分片 1 和 2 ，并且在缺失主分片的时候索引也不能正常工作。 如果此时来检查集群的状况，我们看到的状态将会为 red ：不是所有主分片都在正常工作。

幸运的是，在其它节点上存在着这两个主分片的完整副本， 所以新的主节点立即将这些分片在 Node 2 和 Node 3 上对应的副本分片提升为主分片， 此时集群的状态将会为 yellow（不是green是因为我们之前设置主分片存在两个副本，而现在只剩一个了） 。 这个提升主分片的过程是瞬间发生的，如同按下一个开关一般。

## 3、科普：脑裂问题

CAP：分区容器

* 脑裂问题，就是同一个集群中的不同节点，对于集群的状态，有了不一样的理解。
* 由于并发访问量的提高，导致了我们两个节点的集群（分片数默认为5，副本为1，没有固定的master，都是集群中的节点又做data又做master）状态变成了**red**，出现了大量的坏片，并且坏掉的都是主分片及其副本。分析发现，是ES集群出现了脑裂问题（俗称精神分裂），即集群中不同的节点对于master的选择出现了分歧，出现了多个master竞争，导致主分片和副本的识别也发生了分歧，对一些分歧中的分片标识为了坏片。

* “脑裂”问题可能的成因
1. 网络问题：集群间的网络延迟导致一些节点访问不到master，认为master挂掉了从而选举出新的master，并对master上的分片和副本标红，分配新的主分片
2. 节点负载：主节点的角色既为master又为data，访问量较大时可能会导致ES停止响应造成大面积延迟，此时其他节点得不到主节点的响应认为主节点挂掉了，会重新选取主节点。
3. 内存回收：data节点上的ES进程占用的内存较大，引发JVM的大规模内存回收，造成ES进程失去响应。

主不能工作了就得选。

1. 脑裂问题解决方案：
2. 减少误判：discovery.zen.ping\_timeout节点状态的响应时间，默认为3s，可以适当调大，如果master在该响应时间的范围内没有做出响应应答，判断该节点已经挂掉了。调大参数（如6s，discovery.zen.ping\_timeout:6），可适当减少误判。
3. 选举触发 discovery.zen.minimum\_master\_nodes:1。

discovery.zen.minimum\_master\_nodes（默认是1）：这个参数控制的是，一个节点需要看到的具有master节点资格的最小数量，然后才能在集群中做操作。官方的推荐值是(N/2)+1，其中N是具有master资格的节点的数量（我们的情况是3，因此这个参数设置为2，但对于只有2个节点的情况，设置为2就有些问题了，一个节点DOWN掉后，你肯定连不上2台服务器了，这点需要注意）。

增大该参数，当该值为2时，我们可以设置master的数量为3，这样，挂掉一台，其他两台都认为主节点挂掉了，才进行主节点选举。

1. 角色分离：即master节点与data节点分离，限制角色

主节点配置为：

node.master: true ##作为master节点

node.data: false ##不作为存储数据节点

从节点配置为：

node.master: false

node.data: true

1. 实际解决办法

最终考虑到资源有限，解决方案如下：

增加一台物理机，这样，一共有了三台物理机。在这三台物理机上，搭建了6个ES的节点，三个data节点，三个master节点（每台物理机分别起了一个data和一个master），3个master节点，目的是达到（n/2）+1等于2的要求，这样挂掉一台master后（不考虑data），n等于2，满足参数，其他两个master节点都认为master挂掉之后开始重新选举，

master节点上

node.master = true

node.data = false

discovery.zen.minimum\_master\_nodes = 2

data节点上

node.master = false

node.data = true

方案分析



1. 角色分离后，当集群中某一台节点的master进程意外挂掉了，或者因负载过高停止响应，终止掉的master进程很大程度上不会影响到同一台机器上的data进程，即减小了数据丢失的可能性。
2. discovery.zen.minimum\_master\_nodes设置成了2（3/2+1）当集群中两台机器都挂了或者并没有挂掉而是处于高负载的假死状态时，仅剩一台备选master节点，小于2无法触发选举行为，集群无法使用，不会造成分片混乱的情况。

而图一，两台节点假死，仅剩一台节点，选举自己为master，当真正的master苏醒后，出现了多个master，并且造成查询不同机器，查到了结果不同的情况。

以上的解决方法只能是减缓这种现象的发生，并没有从根本上杜绝，但是毕竟是有帮助的

<https://www.elastic.co/guide/cn/elasticsearch/guide/current/index.html>

# 五、JestClient

1）、9300：TCP

 spring-data-elasticsearch:transport-api.jar；

 springboot版本不同， transport-api.jar不同， 能适配es版本

2）、9200：HTTP

 JestClient、RestTemplate、HttpClient

## 1、SpringBoot整合

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/io.searchbox/jest --><dependency> <groupId>io.searchbox</groupId> <artifactId>jest</artifactId> <version>5.3.4</version></dependency> <!-- https://mvnrepository.com/artifact/org.elasticsearch/elasticsearch --><dependency> <groupId>org.elasticsearch</groupId> <artifactId>elasticsearch</artifactId> <version>5.6.11</version></dependency> |

## 2、配置

|  |
| --- |
| spring: elasticsearch: jest: uris: http://192.168.159.130:9200, http://192.168.159.130:9201 username: elastic password: 123456 read-timeout: 20000 #读取超时  connection-timeout: 20000 #连接超时 |

## 3、使用

|  |
| --- |
| @Autowired JestClient jestClient; public void search(){ //先定义一个action（增删改查），例如search  Search search = new Search.Builder(query).addIndex(index).build(); //执行action，返回结果  SearchResult result = jestClient.execute(search); //处理结果集 ...........省略 }<https://github.com/searchbox-io/Jest/tree/master/jest/src/test/java/io/searchbox/core> <https://github.com/leifengyang/spring-boot/tree/master/spring-boot-samples>  |