[一、基础概念 2](#_Toc11898)

[二、搭建Elasticsearch开发环境 2](#_Toc2980)

[2.1、单机单节点 2](#_Toc8325)

[2.2、为集群添加故障转移 3](#_Toc8313)

[2.3、为集群扩容 3](#_Toc22979)

[2.4、Elasticsearch的故障应对机制 4](#_Toc25505)

[2.5、Elasticsearch的主节点选举机制 4](#_Toc16707)

[2.6、增删节点 5](#_Toc8542)

[三、通过RESTful API与Elasticsearch交互 5](#_Toc5631)

[3.1、Curl请求示例 5](#_Toc14709)

[3.2、文档操作 6](#_Toc19063)

[3.3、查询操作 9](#_Toc26356)

[四、通过JAVA客户端与Elasticsearch交互 12](#_Toc7654)

[五、Elasticsearch的存储与扩容机制 12](#_Toc8960)

[5.1、查看集群的健康信息 12](#_Toc26701)

[5.2、Elasticsearch的分片机制 13](#_Toc25414)

[六、Elasticsearch的索引和文档 13](#_Toc30634)

[6.1、Elasticsearch的冲突处理机制 13](#_Toc14532)

[6.2、Elasticsearch的类型映射 14](#_Toc4115)

[6.3、Elasticsearch的文本分析过程 15](#_Toc14850)

[6.4、Elasticsearch分词 16](#_Toc28051)

[6.4.1、中文分词器插件IK Analysis 16](#_Toc15350)

[七、监控Elasticsearch 18](#_Toc24054)

[7.1、Elasticsearch提供的监控接口 18](#_Toc14122)

[7.1.1、Elasticsearch的集群监控接口 18](#_Toc31206)

[7.1.2、Elasticsearch的索引监控接口 20](#_Toc25301)

[7.2、如何监控Elasticsearch 23](#_Toc12)

[7.2.1、查询性能监控 23](#_Toc7693)

[7.2.2、索引性能监控 24](#_Toc10764)

[7.2.3、JVM垃圾回收监控 25](#_Toc16187)

[7.2.4、节点监控 25](#_Toc21144)

[7.2.5、集群健康程度监控 26](#_Toc25718)

[7.2.6、线程池监控 26](#_Toc15600)

[7.2.7、缓存监控 26](#_Toc29554)

[7.2.8、GET请求监控 27](#_Toc4489)

[7.2.9、Elasticsearch指标采集 27](#_Toc11811)

[7.2.10、Elasticsearch常见性能问题解决方案 27](#_Toc169)

[7.3、Kibana 28](#_Toc10568)

[7.4、ElasticsearchHq 28](#_Toc17219)

[7.5、Cerebro 28](#_Toc7288)

[7.6、ElasticsearchHead 28](#_Toc13145)

### 一、基础概念

1、什么是Elasticsearch：一个实时的分布式搜索分析引擎，可用于文档的分布式存储、字段索引搜索、全文搜索、结构化搜索和分析，并能够支持上百服务节点的扩展和PB级的数据。Elasticsearch建立于全文搜索引擎Lucene之上，使用Lucene做索引与搜索，并提供一套RESTful API。

2、Elasticsearch是面向文档的：Elasticsearch可以直接存储整个对象或是文档，而不必将对象或文档扁平化。Elasticsearch使用JSON作为文档额序列化格式。有别于高度结构化的关系型数据库，单个对象散布于多个表格中，文档数据库可以一次存储整个对象，并且每个对象可与其他被存储的对象不同。

3、Elasticsearch的数据组织方式：Elasticsearch中的索引(Index)类似于关系型数据库中的一个数据库，使用倒排索引来达到提升检索速度的目的。Elasticsearch中的类型(Type)类似于关系型数据库中的表格。Elasticsearch中的文档(Document)类似于关系型数据库中的一个数据行，将其存入的行为被称作索引（动词）。Elasticsearch中的字段(Field)类似于关系型数据库中的列。

4、Elasticsearch的集群组织方式：在分布式环境下，Elasticsearch由集群(cluster)、节点(node)以及分片(shard)组成。一个运行中的实例被称为一个节点，一个或多个拥有相同cluster.name配置的节点组成集群。新加入节点或是删除节点均会导致集群重新分配所有数据。集群中的主节点负责索引和其他节点的增删操作，不会涉及文档级别的数据操作和搜索。

### 二、搭建Elasticsearch开发环境

#### 2.1、单机单节点

1、执行bin/elasticsearch -d，后台运行Elasticsearch

2、访问http://localhost:9200/?pretty，测试Elasticsearch是否启动成功，若成功启动，可以收到类似结构的信息，如图2-1：

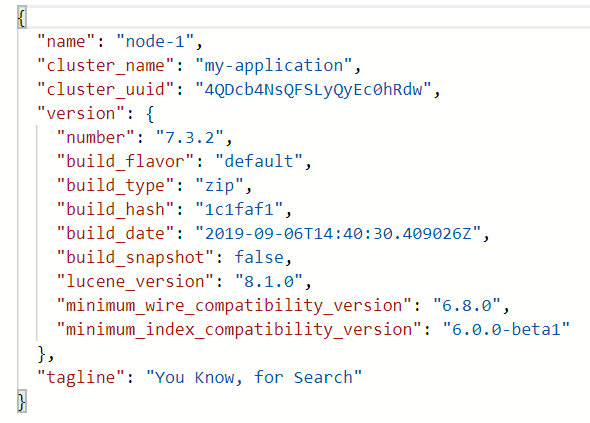


图2-1

可以在conf/elasticsearch.yml中配置相关信息。一个Elasticsearch集群中的节点拥有相同的cluster\_name。

#### 2.2、为集群添加故障转移

在同一台机器上Elasticsearch默认使用组播的方式发现新节点，集群中使用单播方式发现节点。通过修改elasticsearch.yml中的discovery.zen.ping.unicast.hosts可以修改单播列表，新节点连接单播列表中的任一成员均可获取到集群的整体状态和节点信息，之后新节点会联系master节点，并加入集群。集群的最小配置字段如图2-2所示：

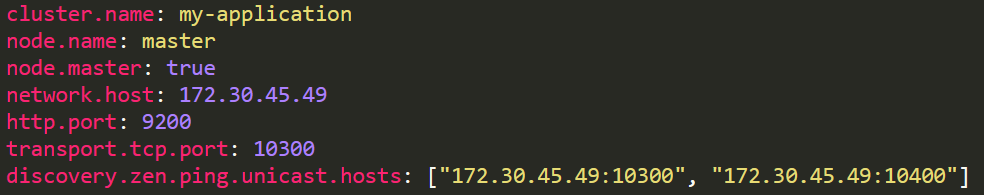


图2-2

同一集群内节点的cluster.name字段应该相同，Elasticsearch启动需要绑定两个端口（9200与9300），一个负责处理http请求，另一个负责处理来自客户端与集群其他节点的tcp连接。使用node.master字段标识主节点。

当配置并启动集群后，Elasticsearch会自动为每个节点分配主分片（P0-P2）或是副本分片（R0-R2），如图2-3，做到数据的一定程度冗余，防止出现数据丢失。若有一个新节点加入集群，那么Elasticsearch会重新调整集群上的分片分布，使其分布均匀，如图2-4所示。

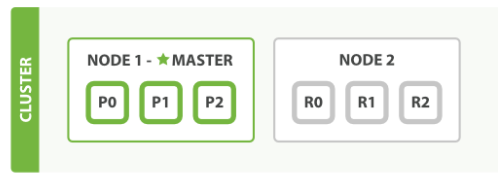


图2-3

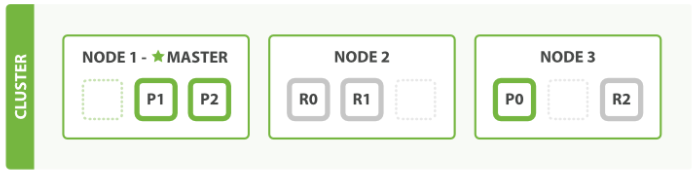


图2-4

#### 2.3、为集群扩容

在创建索引时可以指定主分片的个数，如图2-5所示，为索引megacorp指定3个主分片，每个主分片对应一个副本分片，共6个分片。6个分片最多可分布于6个节点上，每个节点上存放一个分片，若节点个数多于6个，则需要为集群扩容。

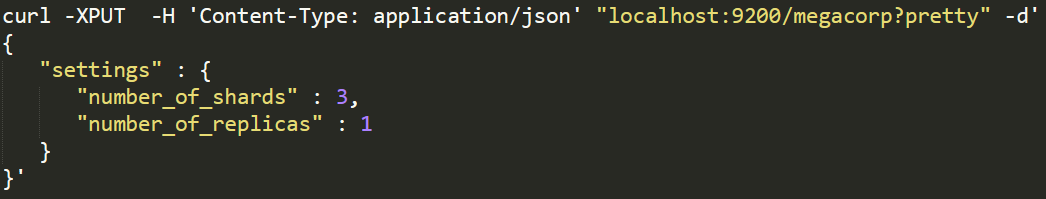


图2-5

由于索引的主分片数在创建时已经确定，可以通过调整每个主分片对应的副本分片数来动态伸缩集群，调整方式如图2-6所示。每个主分片对应2个副本分片，调整后集群一共包含9个分片，最多可扩容至9个节点上。调整后的结果如图2-7所示。

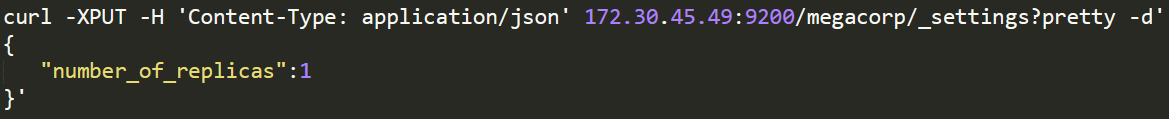


图2-6

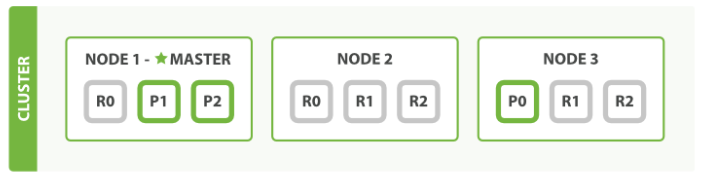


图2-7

在3个节点上存放9个分片可以保证在2个节点出故障的情况下，集群不丢失任何数据。若在3个节点上存放6个分片，如图2-4，那么最多1个节点出故障，集群才能不丢失任何数据。在完成集群扩容后，可通过如下命令查看分片状态：

curl -XGET -s 172.30.45.49:9200/\_cat/shards?h=index,shard,prirep,state,unassigned.reason

若每个节点上均存放有索引的完整数据，如图2-7，而此时仍然有新的副本分片加入，则新加入的副本分片会成为未分配的分片（unassigned replica），可通过减少副本分片数来解决。

#### 2.4、Elasticsearch的故障应对机制

若集群中的主节点发生故障，则集群会重新在剩余的节点中选举出一个新的主节点，如图2-8。并检查集群中主分片的缺失情况，若存在缺失主分片对应的副本分片，则会将这些副本分片提升为主分片。

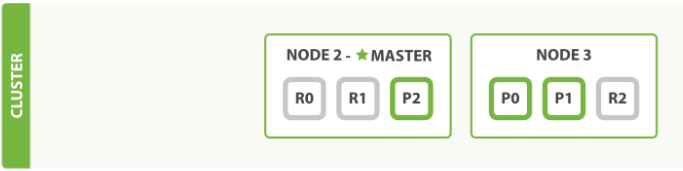


图2-8

#### 2.5、Elasticsearch的主节点选举机制

1. 集群发现阶段。集群中的每个节点会通过配置项discovery.seed\_host（或discovery.zen.ping.unicast.hosts）提供的ip地址获取整个集群的拓扑结构信息。在节点启动或是主节点因故障失联时，会启动集群发现流程，直到找到集群中的主节点或是新的主节点被选出。搜索过程可以描述为：

·依次询问seed\_host中的节点，并确认这些节点是否为候主节点选节点(可通过node.master指定)。

·依次询问seed\_host中的节点是否有连接到其他节点，且没有被询问，记录这些新节点。

·与seed\_host中的节点交换已知的候选节点列表。

·将新节点作为待询问节点，重复发现流程。

1. 主节点选举阶段。在集群的启动阶段或是已有的主节点失连，都会触发Elasticsearch的选举进程。在候选节点数目满足最小要求（quorum）后，所有的候选节点均可发起选举进程。候选节点最小数目要求可通过discovery.zen.minimum\_master\_nodes来设置，默认为集群候选节点数目一半加一，Elasticsearch使用quorum来防止集群出现Brain split问题，导致数据或状态不一致。主节点选举过程可描述为：

·一个节点发起选举进程，并依次ping集群中的其他节点，获取其他候选节点状态信息。

·从现有候选节点中选出主节点：

··若获取的候选节点中已有主节点，那么接受此选举结果。

··若候选节点中无主节点，那么将自身节点与候选节点状态信息加入待选列表中。

··若满足quorum要求，则根据节点保存的集群状态版本号、nodeId对列表排序，选出列表中第一个节点作为主节点。

·若节点自己被选举为主节点，则向集群所有节点发送状态更新消息，并等待其他节点加入。

·若其他节点被选举为主节点，则当前节点会尝试连接主节点并加入其中。

#### 2.6、增删节点

1. 增加节点。为节点配置seed\_host，节点启动时会自行向主节点发送集群加入请求。节点加入集群后，Elasticsearch会加新节点加入投票配置（voting configuration）中，此节点将会对之后的选举产生影响。
2. 若同时下线超过半数节点，那么集群将变得不可用。官方建议是一次下线一个节点，给集群调整投票配置与容错级别（fault tolerance level）的时间。
3. 若有同时下线超半数节点的需求，可通过调用如下接口手动将节点从投票配置中删除，此后这些节点将不会对集群选主造成影响。最后可以安全地同时下线这些节点。

curl -XGET /\_cluster/voting\_config\_exclusions/{node\_name}

可通过如下接口查看目前的排除列表内容：

curl -XGET /\_cluster/state?filter\_path=metadata.cluster\_coordination.voting\_config\_exclusions

可通过如下接口删除列表内容，允许节点重新拥有选主投票权限：

curl -XDELETE /\_cluster/voting\_config\_exclusions

### 三、通过RESTful API与Elasticsearch交互

#### 3.1、Curl请求示例

默认使用9200端口与Elasticsearch通信，请求示例：

curl -XGET -H "Content-Type: application/json" http://localhost:9200/\_count?pretty -d '{ "query": {"match\_all":{} } }'

向Elasticsearch插入一条数据，请求示例：

curl -X PUT -H 'Content-Type: application/json' "localhost:9200/megacorp/employee/1?pretty" -d' { "first\_name" : "John", "last\_name" : "Smith", "age" : 25, "about" : "I love to go rock climbing", "interests": [ "sports", "music" ] } '

其中megacorp为索引名，employee为类型名称，1为文档ID。

根据文档ID请求一条数据，请求示例：

curl -X GET "localhost:9200/megacorp/employee/1?pretty"

通过更换请求method为DELETE，可以删除指定数据，请求示例：

curl -X DELETE "localhost:9200/megacorp/employee/1?pretty"

可以通过使用PUT或POST方法对已存在的文档进行更新操作，请求示例：

curl -X PUT -H 'Content-Type: application/json' "localhost:9200/megacorp/employee/1?pretty" -d '{ "first\_name" : "John"}'

Elasticsearch在执行更新操作时，会先将旧文档标记为删除状态，然后添加新文档。

通过在请求路径中使用\_search端点，并配合q参数可以实现具体到字段的轻量搜索，请求示例：

curl -X GET "localhost:9200/megacorp/employee/\_search?q=last\_name:Smith&pretty"

也可以使用JSON构造一个更复杂的查询请求：

curl -X GET -H "Content-Type: application/json" "localhost:9200/megacorp/employee/\_search" -d'

{"query": {"bool": {"must": [{ "match": { "last\_name": "Smith" }},{ "match": { "age": 25 }}]}}}'

curl -X GET "localhost:9200/megacorp/employee/\_search?pretty" -H 'Content-Type: application/json' -d'

{"query" : {"bool": {"must": {"match" : {"last\_name" : "smith"}},"filter": {"range" : {"age" : { "gt" : 30 }}}}}}'

#### 3.2、文档操作

1. 索引文档，使用自定义ID

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| PUT | /{index}/{type}/{id} | {"field":"value",...} |

返回格式为：

|  |
| --- |
| {  "\_index": {index},  "\_type": {type},  "\_id": {id},  "\_version": 1,  "created": true  } |

若仅需要在文档不存在时才创建新文档，防止现有文档被覆盖，则需要在uri后附上op\_type或是\_create参数，如/{index}/{type}/{id}?op\_type=create, /{index}/{type}/{id}/\_create。若请求成功，Elasticsearch会返回文档元数据以及201的HTTP响应码，若文档已经存在，会返回409的响应码。

1. 索引文档，使用自动生成ID

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| POST | /{index}/{type}/ | {"field":"value",...} |

返回格式为：

|  |
| --- |
| {  "\_index": {index},  "\_type": {type},  "\_id": {generated\_id},  "\_version": 1,  "created": true  } |

Elasticsearch自动生成的ID为基于Base64编码的长度为20的GUID字符串。

1. 取回一个文档

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| GET | /{index}/{type}/{id} |  |

返回格式为（元数据+文档内容\_source）：

|  |
| --- |
| {  "\_index": {index},  "\_type": {type},  "\_id": {id},  "\_version": 1,  "found": true,  "\_source" : {...}  } |

若没有找到对应的文档，请求返回码将变为404，found字段将变为false。此uri接受\_source参数，如/{index}/{type}/{id}?\_source=field1,field2，则只返回field1和field2字段的查询结果。若\_source填空值，如/{index}/{type}/{id}?\_source或/{index}/{type}/{id}/\_source则返回值中不包含文档元数据。

1. 检查文档是否存在

|  |  |  |  |
| --- | --- | --- | --- |
| method | args | uri | data |
| HEAD | -I | /{index}/{type}/{id} |  |

返回格式为（只包含HTTP报头）：

|  |
| --- |
| HTTP/1.1 200 OK  Content-Type:text/plain; charset=UTF-8  Content-Length: 0 |

如文档不存在，则返回404.

1. 更新文档

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| PUT | /{index}/{type}/{id} | {"field":"value",...} |

由于Elasticsearch中的文档不能修改，只能整体替换，更新文档的API与索引文档的API相同。

返回格式为：

|  |
| --- |
| {  "\_index": {index},  "\_type": {type},  "\_id": {id},  "\_version": 2,  "created": false  } |

1. 删除文档

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| DELETE | /{index}/{type}/{id} |  |

返回格式为：

|  |
| --- |
| {  "\_index": {index},  "\_type": {type},  "\_id": {id},  "\_version": 3,  "found": true  } |

若文档不存在，则found字段为false。

1. 部分更新文档

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| POST | /{index}/{type}/{id}/\_update | {"doc":{field:xxx,...}} |

返回格式为：

|  |
| --- |
| {  "\_index": {index},  "\_type": {type},  "\_id": {id},  "\_version": 3  } |

由于Elasticsearch的文档是不可变的，因此update操作包含了检索-修改-重建索引这三个处理过程，这三个过程均发生在分片内部。由于Elasticsearch使用乐观锁来处理冲突，因此update在执行检索-修改-重建索引这三个步骤的过程中存在冲突的可能。若发生冲突，update请求失败。对于部分字段的更新，执行顺序不重要，可以通过添加retry\_on\_conflict参数来指定冲突后的重试次数，如/{index}/{type}/{id}/\_update?retry\_on\_conflict=5。

1. 批量取回文档

未指定\_index、\_type和\_id：

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| GET | /\_mget | {"docs":[  {"\_index":xx,"\_type":xx,"\_id":xx},  {"\_index":xx,"\_type":xx,"\_id":xx,"\_source":xx}  ]} |

指定\_index、\_type和\_id：

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| GET | /{index}/{type}/\_mget | {"docs":[{"\_id":xx},{"\_id":xx,"\_source":xx}]} |
| GET | /{index}/{type}/\_mget | {"ids":["xx", "xx"]} |

返回格式为：

|  |
| --- |
| {  "docs":[  {  "\_index": {index},  "\_type": {type},  "\_id": {id},  "\_version": 3  "found": true,  "\_source":{...}  },  ...  ]  } |

1. 批量更新文档

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| POST | /\_bulk | bulk\_request |
| POST | /{index}/\_bulk | bulk\_request |
| POST | /{index}/{type}/\_bulk | bulk\_request |

bulk\_request的格式为：

|  |
| --- |
| {"action01":{ doc\_metadata}}\n  {request body of action01}\n  {"action02":{ doc\_metadata}}\n  {request body of action02}\n  ...  {"actionN":{ doc\_metadata}}\n  {request body of actionN}\n |

action可以是create、index、update与delete四个操作之一，其中create操作是在文档不存在时创建，index操作是创建或替换文档。request body包含了对应操作所需的请求参数。

返回格式为：

|  |
| --- |
| {  "took":3,  "error": false,  "items":[  {"delete":{  "\_index": {index},  "\_type": {type},  "\_id": {id},  "\_version": 3  "status": 200,  "found": true  }},  ...  ]  } |

#### 3.3、查询操作

Elasticsearch的查询语言可以分为两大类：叶子查询子句与组合查询子句。叶子查询子句主要用于查询特定字段中的特定值。组合查询子句用于合并多个叶子查询子句或是嵌套合并其他组合查询子句。

一个查询请求中包含一个或多个查询子句，最终会计算出一个衡量文档与查询条件相似度的分值(\_score)出来，这个数值将被直接用于筛选查询结果。但是不同子句对最终分值的贡献并不相同，针对相似度分值计算，查询子句可以分为两类：在查询上下文中(query context)与在过滤上下文中(filter context)。在查询上下文中的子句除了会执行过滤操作外，还会计算出一个相似度分值，并用于计算最终分值，而在过滤上下文中的子句只负责根据条件过滤文档，不影响最终相似度的计算。

1、组合查询之Boolean查询(query context)

Boolean查询包含四种类型的子句，分别为：

·must(query context): 在must子句内的查询条件必须全部被满足。

·filter(filter context): 出现在filter子句内的字段会被忽略。

·should(query context): 在should子句内的查询条件需至少一个被满足。

·must\_not(filter context): 查询内容不能出现在被查询文档中。

Boolean查询通过累加策略计算最终相关度分值。

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| POST | /{index}/{type}/\_search | {content} |

{content}：

|  |
| --- |
| {  "query": {  "bool" : {  "must" : {  "term" : { "user" : "kimchy" }  },  "filter": {  "term" : { "tag" : "tech" }  },  "must\_not" : {  "range" : {  "age" : { "gte" : 10, "lte" : 20 }  }  },  "should" : [  { "term" : { "tag" : "wow" } },  { "term" : { "tag" : "elasticsearch" } }  ],  "minimum\_should\_match" : 1,  "boost" : 1.0  }  }  } |

通过minimum\_should\_match过滤掉相关度分值小于1的文档。

通过boost指定文档相关度分值系数。

2、组合查询之Boosting查询(query context)

Boosting查询包含positive、negative和negative\_boost三个子句，positive子句负责筛选文档和计算相似度，negative和negative\_boost负责衰减Boosting查询相似度。Boosting查询的最终相似度计算公式为：

\_score=I(positive\_score>0) \* (positive\_score \*(1- I(negative\_score>0)\*(1-negative\_boost))

I(positive\_score>0)表示当positive\_score>0时取1，反之取0

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| POST | /{index}/{type}/\_search | {content} |

{content}：

|  |
| --- |
| {  "query": {  "boosting" : {  "positive" : {  "term" : {  "text" : "apple"  }  },  "negative" : {  "term" : {  "text" : "pie tart fruit crumble tree"  }  },  "negative\_boost" : 0.5  }  }  } |

3、组合查询之Constant score查询(query context)

由于filter子句属于过滤上下文（filter context），因此其对文档最终相似度数值计算不会产生任何影响。若需要将filter子句放置于查询上下文中，并让其对最终相似度数值计算产生影响，则需要使用constant score查询。

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| POST | /{index}/{type}/\_search | {content} |

{content}：

|  |
| --- |
| {  "query": {  "constant\_score" : {  "filter" : {  "term" : { "user" : "kimchy"}  },  "boost" : 1.2  }  }  } |

此处filter子句会对每一个满足过滤条件的文档赋予一个1.2的相关度数值。

4、组合查询之Disjunction max查询(query context)

Disjunction max查询包含queries和tie\_breaker两个子句，queries子句中包含多个查询条件，负责过滤与计算相关度数值。Disjunction max查询的最终相关度计算公式为：

\_score=\_score\_of(best\_matched\_query) + ∑(\_score\_of(other\_matched\_query) \* tie\_breaker)

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| POST | /{index}/{type}/\_search | {content} |

{content}：

|  |
| --- |
| {  "query": {  "dis\_max" : {  "queries" : [  { "term" : { "title" : "Quick pets" }},  { "term" : { "body" : "Quick pets" }}  ],  "tie\_breaker" : 0.7  }  }  } |

5、组合查询之Function score查询(query context)

Function score查询用于以更灵活的方式更改文档的相关度分值。

|  |  |  |
| --- | --- | --- |
| method | uri | data |
| POST | /{index}/{type}/\_search | {content} |

{content}：

|  |
| --- |
| {  "query": {  "function\_score": {  "query": { "match\_all": {} },  "boost": "5",  "functions": [  {  "filter": { "match": { "test": "bar" } },  "random\_score": {},  "weight": 23  },  {  "filter": { "match": { "test": "cat" } },  "weight": 42  }  ],  "max\_boost": 42,  "score\_mode": "max",  "boost\_mode": "multiply",  "min\_score" : 42  }  }  } |

function\_score中query子句用于设定查询条件，boost参数用于指定文档相关度分值系数，functions子句用于设定相关度分值生成方式。可通过functions子句设定多个分值生成函数，并根据score\_mode进行组合。满足filter的文档才会被应用对应的分值生成函数。查询子句产生的相关度分值与functions子句产生的分值通过boost\_mode参数指定方式组合。min\_score参数指定了文档所应具备的最低分值。以上述查询语句为例，function\_score查询的相关度分值计算公式可以概括为：

\_score=I(inner\_score>=min\_score)\*inner\_score

inner\_score=(boost\*score\_of\_query)\*clip(max\_boost,max(score\_func1,score\_func2))

score\_func1=I(satisfy\_filter)\*random\_score()\*weight

score\_func2=I(satisfy\_filter)\*weight

clip(a,b)的定义为：b<a?b:a。

### 四、通过JAVA客户端与Elasticsearch交互

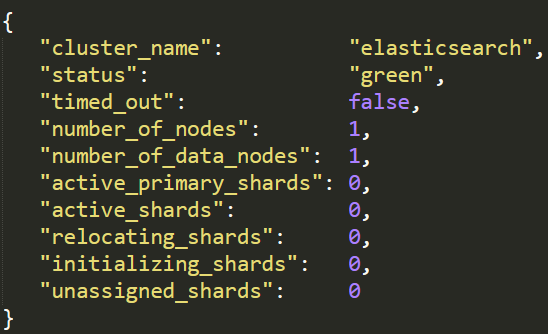
### 五、Elasticsearch的存储与扩容机制

5.1、查看集群的健康信息

请求示例：

curl -X GET "localhost:9200/\_cluster/health?pretty"

返回内容如下：

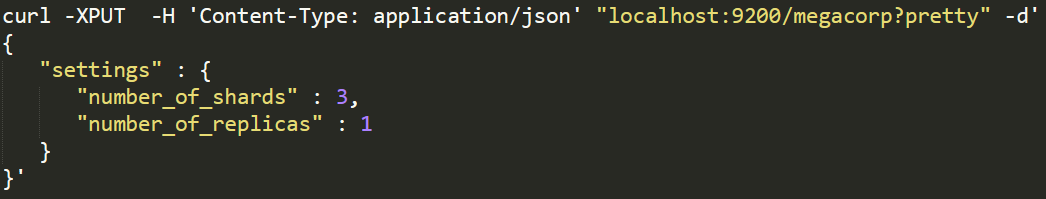


其中status字段表示集群的运行状态：green表示所有主分片和副本分片都正常运行，yellow表示所有主分片和部分副本分片正常运行，red表示部分主分片没有正常运行。

5.2、Elasticsearch的分片机制

分片是Elasticsearch的一个底层工作单元，仅保存一部分数据，一个分片即为一个完整的搜索引擎（一个Lucene实例）。当集群规模变化时，为了能够使数据均匀分布在集群里，Elasticsearch会迁移分片，利用分片将数据分发到集群各处。作为数据容器，当添加一个文档时，文档会被存储和索引到分片内。

分片可以被分为主分片和副本分片，副本分片只是主分片的拷贝，用于数据冗余备份以及提供数据读操作。在创建索引时，Elasticsearch会首先为索引分配多个主分片，每个主分片对应一个副本分片，可以通过如下方式在创建时指定索引的主分片数(number\_of\_shards)，以及每个主分片对应的副本分片数(number\_of\_replicas)：



同时也可以通过GET方法查看索引的状态信息。

### 六、Elasticsearch的索引和文档

#### 6.1、Elasticsearch的冲突处理机制

Elasticsearch使用乐观锁的方式进行并发控制。为每个文档分配一个版本号（\_version），当文档被修改时版本号依次递增，以此来确保数据变更能够以正确的方式进行。在更新文档时带上version参数，如curl -XPUT /{index}/{type}/{id}?version=1，此处version表示修改会被应用的文档版本，Elasticsearch会检测此次更新是否会有冲突，若冲突会返回409的HTTP响应码。

#### 6.2、Elasticsearch的类型映射

可以通过/{index}/\_mapping/{type}接口查看索引中指定类型的映射，映射可以理解为Elasticsearch对文档结构的一个解释。对于包含五个字段的类型示例employee的返回结果如下（7.x后的版本需要在请求中包含include\_type\_name字段，并设置为true）：

|  |
| --- |
| {  "megacorp": {  "mappings": {  "employee": {  "properties": {  "about": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "age": {  "type": "long"  },  "first\_name": {  "type": "text",  ...  },  "interests": {  "type": "text",  ...  },  "last\_name": {  "type": "text",  ...  }  }  }  }  }  } |

类型映射关系可以在创建索引时指定，若未指定则由系统自动分析生成。创建索引时自定义映射的方式如下：

curl -XPUT -H "Content-Type:application/json" /{index} -d '{content}'

{content}定义如下：

|  |
| --- |
| {  "mappings": {  "employee": {  "properties": {  "first\_name": {  "type": "text",  "analyzer": "english"  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  ...  ...  }  }  }  } |

可以通过接口/{index/\_mapping/{type}用PUT方法为映射添加新的字段，但是不能修改已存在的字段，因为这些字段的数据可能已经被索引。Elasticsearch中的字段可以大致分为两类：精确值和全文。精确值是指具有string、number、boolean或是date等类型的数据，查询时只需要做简单地对比即可，结果为匹配或是不匹配。对于全文类型的字段，查询时需要给出一个衡量待查询值与文本内容相关性的系数。E.S.分两步来处理全文类型的数据：分析与创建倒排索引。Elasticsearch支持的数据类型如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 描述 | 类型 | 描述 |
| 字符串类型 | string：5.x开始不再支持  text：全文搜索，生成倒排索引  keyword：结构化字段，精准搜索 | 整数类型 | integer  long  short  byte |
| 浮点类型 | double  float  half\_float：16位  scaled\_float：实际存储值为实际数值乘缩放因子 | 逻辑类型 | boolean：true/false，on/off，1/0 |
| 日期类型 | date：可存储字符串、毫秒数或是秒数 | 范围类型 | integer/lone\_range  float/double\_range  date\_range  ip\_range |
| 二进制类型 | binary：base64编码存储 | 数组类型 | array：如字符串、整数、json对象数组 |
| 对象类型 | object：json对象，mapping中properties属性嵌套定义 | 嵌套类型 | nested |
| 地理坐标类型 | geo\_point | 地理地图 | geo\_shape：地理形状 |
| IP类型 | ip | 范围类型 | completion |
| 令牌计数类型 | token\_count：分析字符串单词数量 | 附件类型 | attachment |
| 抽取类型 | percolator |  |  |

#### 6.3、Elasticsearch的文本分析过程

Elasticsearch的文本分析包含如下三个流程：

分析器

字符过滤

分词

Token过滤

字符过滤：由字符过滤器负责，任务是在分词前整理字符串，过滤掉无用的字符或是替换掉特殊字符。

分词：由分词器负责，任务是按照特定规则或是语言习惯将字符串拆分为单个的词条，一个简单的分词规则包括按空格或是标点对字符串进行拆分。

Token过滤：由token过滤器负责，其任务包括：改变词条，例如将单词小写化；删除词条，例如过滤掉停用词；增加词条，例如为词条增加同义词。

在增添文档或是文档查询时，均会对相应字符串执行文本分析的三流程。Elasticsearch内置多种分析器：Standard Analyzer，Simple Analyzer，Whitespace Analyzer，Stop Analyzer等（参考https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-analyzers.html）。

#### 6.4、Elasticsearch分词

6.4.1、中文分词器插件IK Analysis

1、IK Analysis安装

Elasticsearch的中文分词器插件一般选择IK Analysis，其整合了Lucene中文分词器IK Analyzer。

Github：https://github.com/medcl/elasticsearch-analysis-ik

安装方式为下在对应版本的IK Analysis，解压到Elasticsearch的plugin目录。

下载地址：https://github.com/medcl/elasticsearch-analysis-ik/releases

2、IK Analysis配置

若需要自定义字典或是停止词字典，则需要额外配置IKAnalyzer，配置文件路径为Elasticsearch配置目录{conf}/analysis-ik/config/IKAnalyzer.cfg.xml，或是插件目录{plugins}/elasticsearch-analysis-ik-\*/config/IKAnalyzer.cfg.xml。配置文件内容为：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">  <properties>  <comment>IK Analyzer 扩展配置</comment>  <!--用户可以在这里配置自己的扩展字典 -->  <entry key="ext\_dict">custom/mydict.dic;custom/single\_word\_low\_freq.dic</entry>  <!--用户可以在这里配置自己的扩展停止词字典-->  <entry key="ext\_stopwords">custom/ext\_stopword.dic</entry>  <!--用户可以在这里配置远程扩展字典 -->  <entry key="remote\_ext\_dict">location</entry>  <!--用户可以在这里配置远程扩展停止词字典-->  <entry key="remote\_ext\_stopwords">http://xxx.com/xxx.dic</entry>  </properties> |

前两个entry通过本地文件扩展分词器的词典，后两个entry可以通过配置url来热更新IKAnalyzer的字典。热更新字典时，若http请求头中Last-Modified字段或是ETag字段有一个发生了改变，则IK Analysis插件就会通过配置的url获取新的分词并更新词库。字典的内容为每行一个分词并用换行符\n结尾。

1. 测试IK Analysis插件是否生效

使用POST方法访问接口/\_analyze对如下一段话进行分词：

curl -XPOST -H 'Content-Type:application/json' 172.30.45.49:9200/\_analyze -d'

{

"tokenizer": "ik\_smart",

"text": "演示了如何使用IK Analysis 插件来实现在 Elasticsearch 中的中文分词"

}'

发送请求时需确认终端支持中文编码，部分分词结果为：

|  |
| --- |
| {  "tokens": [  {  "token": "演示",  "start\_offset": 0,  "end\_offset": 2,  "type": "CN\_WORD",  "position": 0  },  {  "token": "了",  "start\_offset": 2,  "end\_offset": 3,  "type": "CN\_CHAR",  "position": 1  },  {  "token": "如何",  "start\_offset": 3,  "end\_offset": 5,  "type": "CN\_WORD",  "position": 2  },    ...  ...    {  "token": "中文",  "start\_offset": 42,  "end\_offset": 44,  "type": "CN\_WORD",  "position": 13  },  {  "token": "分词",  "start\_offset": 44,  "end\_offset": 46,  "type": "CN\_WORD",  "position": 14  }  ]  } |

IKAnalyzer提供了两种分词器（Tokenizer），分别ik\_max\_word和ik\_smart：

ik\_max\_word：对文本做最细粒度的拆分，会穷尽各种可能的组合，适合文本索引。

ik\_smart：会做粗粒度的拆分，适合文本搜索。

### 七、监控Elasticsearch

#### 7.1、Elasticsearch提供的监控接口

7.1.1、Elasticsearch的集群监控接口

1、集群健康监控

|  |  |  |
| --- | --- | --- |
| method | uri | desc |
| GET | /\_cluster/health |  |
| GET | /\_cluster/health/index1,index2 | 查看关于索引的更详细信息 |
| GET | /\_cluster/health?level=indices |
| GET | /\_cluster/health?level=shards | 查看关于每个分片的更详细信息 |
| GET | /\_cluster/health/index1,index2?level=shards |
| GET | /\_cluster/health?wait\_for\_status=green | 阻塞，直到达到指定状态 |

返回格式为：

|  |
| --- |
| {  "cluster\_name" : "my-application",  "status" : "green",  "timed\_out" : false,  "number\_of\_nodes" : 2,  "number\_of\_data\_nodes" : 2,  "active\_primary\_shards" : 8,  "active\_shards" : 16,  "relocating\_shards" : 0,  "initializing\_shards" : 0,  "unassigned\_shards" : 0,  "delayed\_unassigned\_shards" : 0,  "number\_of\_pending\_tasks" : 0,  "number\_of\_in\_flight\_fetch" : 0,  "task\_max\_waiting\_in\_queue\_millis" : 0,  "active\_shards\_percent\_as\_number" : 100.0  } |

字段解释：

·status :green表示所有主分片和副本分片都已分配；yellow表示所有主分片已被分配，但至少一个副本分片缺失；red表示至少一个主分片缺失，其对应的副本分片也缺失（若存在副本分片，则副本分片会被提升为主分片），搜索只能返回部分数据。

·timed\_out :

·number\_of\_nodes :集群中的节点个数。

·number\_of\_data\_nodes :集群中的数据节点个数，默认所有节点都是数据节点与主节点候选节点。但是可以通过node.data: false来声明此节点不存储数据，一般用于master节点和候选节点。同时也可以通过设置node.master: false来取消节点成为主节点的资格，只存储数据。两者都为false，则节点将会成为一个专用的客户端节点，起负载均衡的作用。

·active\_primary\_shards :集群中主分片数量。

·active\_shards :集群的所有分片数量，包括副本分片。

·relocating\_shards :正在迁往其他节点的分片数量。

·initializing\_shards :当分片刚创建或是节点刚重启时，分片会短暂处于initializing状态。

·unassigned\_shards :未分配的分片数量，若分片数量过多，超过了满足集群每个节点保存完整数据的需要，就会出现未分配分片。若有节点出现故障也会出现未分配分片。

·delayed\_unassigned\_shards :

·number\_of\_pending\_tasks :

·number\_of\_in\_flight\_fetch :

·task\_max\_waiting\_in\_queue\_millis :

·active\_shards\_percent\_as\_number :

2、集群状态监控

|  |  |  |
| --- | --- | --- |
| method | uri | desc |
| GET | /\_cluster/state |  |
| GET | /\_cluster/state/{metrics}/{indices} | 查询指标和索引过滤 |

监控指标{metrics}包括了，这些指标可用逗号组合：

version：包含了集群名称，集群uuid和状态uuid等信息。

master\_node：

nodes：

routing\_table：

metadata：

blocks：

3、集群状态统计

|  |  |  |
| --- | --- | --- |
| method | uri | desc |
| GET | /\_cluster/stats | 集群范围的统计信息 |

返回内容包括了索引统计指标（分片数目、存储占用以及内存占用等信息）和集群节点（数目统计、操作系统和机器信息、jvm统计信息、插件信息、内存和cpu利用率等信息）。

1. 等待中任务信息获取

|  |  |  |
| --- | --- | --- |
| method | uri | desc |
| GET | /\_cluster/pending\_tasks | 统计能引起集群状态改变且尚未被执行的任务信息 |

返回格式为：

|  |
| --- |
| {  "tasks": [  {  "insert\_order": 101,  "priority": "URGENT",  "source": "create-index [foo\_9], cause [api]",  "time\_in\_queue\_millis": 86,  "time\_in\_queue": "86ms"  },  ...  ]  } |

5、集群节点监控

|  |  |  |
| --- | --- | --- |
| method | uri | desc |
| GET | /\_nodes/stats | 获取集群每个节点的统计数据 |
| GET | /\_nodes/nodeId1,nodeId2/stats | 获取集群指定节点的统计数据 |
| GET | /\_nodes/stats/{metrics} | 根据{metrics}过滤统计指标 |

返回格式为：

|  |
| --- |
| {  "\_nodes": {  "total": 2,  "successful": 2,  "failed": 0  },  "cluster\_name": "my-application",  "nodes": {  "82AFcZQDTzOAZc1U5i0LVQ": {  "timestamp": 1569415960231,  "name": "slave-2",  "transport\_address": "172.30.45.49:10400",  "host": "172.30.45.49",  "ip": "172.30.45.49:10400",  ...  },  "UjHGpQucRyaN6Jw2Wm7bRA": {  "timestamp": 1569415960231,  "name": "master",  ...  }  }  } |

监控指标{metrics}包括了，这些指标可用逗号组合：

indices：节点上索引的相关统计值，包括索引大小、文档数量、创建删除索引次数、搜索次数和字段缓存大小等信息。

fs：包括数据存储路径、剩余磁盘空间以及读写统计等信息。

http：http连接信息。

jvm：JVM统计信息，内存池信息，垃圾收集信息，缓存区以及类加载等信息。

os：描述整个操作系统的状况，包括负载、内存和内存交换等信息。

process：包括内存消耗、cpu占用和文件描述符占用等信息。

thread\_pool：线程池信息。

transport：网络传输数据统计。

breaker：field数据断路器相关信息。

discovery：

ingest：

7.1.2、Elasticsearch的索引监控接口

1. 获取索引详细信息

|  |  |  |
| --- | --- | --- |
| method | uri | desc |
| GET | /{index1},{index2} | 获取指定索引的详细信息 |
| GET | /{index1},{index2}/\_alias | 获取指定索引的别名设置 |
| GET | /{index1},{index2}/\_mapping | 获取指定索引的映射 |
| GET | /{index1},{index2}/\_settings | 获取指定索引的设置信息 |

返回格式为：

|  |
| --- |
| {  {index1}: {  "aliases": {  ...  },  "mappings": {  "properties": {  ...  }  },  "settings": {  "index": {  "creation\_date": "1569305098722",  "number\_of\_shards": "1",  "number\_of\_replicas": "1",  "uuid": "ylGfI0GIQe24bnLxuCvKyw",  "version": {  "created": "7030299"  },  "provided\_name": {index1}  }  }  }  {index2}：{...}  } |

返回结果包括了索引别名、索引映射信息以及索引分片等基础信息。

1. 获取索引的段（segment）信息

|  |  |  |
| --- | --- | --- |
| method | uri | desc |
| GET | /{index1},{index2}/\_segments | 获取指定索引的segment信息 |

返回格式为：

|  |
| --- |
| {  "\_shards": {  "total": 6,  "successful": 6,  "failed": 0  },  "indices": {  "{index1}": {  "shards": {  "0": [  {  "routing": {  "state": "STARTED",  "primary": false,  "node": "82AFcZQDTzOAZc1U5i0LVQ"  },  "num\_committed\_segments": 1,  "num\_search\_segments": 1,  "segments": {  "\_9": {  "generation": 9,  "num\_docs": 3,  "deleted\_docs": 0,  "size\_in\_bytes": 6748,  "memory\_in\_bytes": 3255,  "committed": true,  "search": true,  "version": "8.1.0",  "compound": false,  "attributes": {  "Lucene50StoredFieldsFormat.mode": "BEST\_SPEED"  }  }  }  },  {  "routing": {  "state": "STARTED",  "primary": true,  "node": "UjHGpQucRyaN6Jw2Wm7bRA"  },  ...  }  ],  "1": [  ...  ],  "2": [  ...  ]  }  },  "{index2}": {...}  }  } |

返回结果按照索引—分片—[主分片，副本分片1，...，副本分片N]的格式组织。常用字段包括：

generation，一个递增的整数值，每当有新的段建立时会递增1，Elasticsearch使用此值为segment命名。

num\_docs，当前段中存储的文档数量。

deleted\_docs，被标记为删除的文档数量。

size\_in\_bytes，当前段所占用的磁盘空间。

memory\_in\_bytes，当前段占用的内存空间。

committed，用于标记当前段有没有被同步到磁盘。

search，用于标记当前段是否能够被用于搜索。

version，Lucene版本。

compound，用于标记Lucene有没有把该段在磁盘上的所有文件合并为一个文件。

1. 获取索引的统计信息

|  |  |  |
| --- | --- | --- |
| method | uri | desc |
| GET | /{index1},{index2}/\_stats | 获取指定索引的统计信息 |

返回格式为：

|  |
| --- |
| {  "\_shards": {  "total": 6,  "successful": 6,  "failed": 0  },  "\_all": {  "primaries": {...},  "total": {...},  }  "indices": {  "{index1}": {  "primaries": {...},  "total": {...},  },  "{index2}":  "primaries": {...},  "total": {...},  }  }  } |

返回结果的indices中包含每个索引的单独统计信息，\_all中包含所有索引的合并统计信息。每部分的统计信息分为primaries和total，其中primaries只包含主分片的统计信息，total合并统计了主分片和副本分片的所有信息。统计数据中包含如下常用字段：

docs：文档和已删除文档数量统计。

fielddata：fielddata所占用的内存大小。

flush：索引刷入磁盘的次数、间隔和耗费时间。

get：get请求的统计数据，包括成功、失败请求数量以及耗费时间。

indexing：历史已完成和当前未完成的索引建立、删除统计数据，

merge：段（segment）合并统计数据，包括历史合并次数，正在合并段数量，涉及的文档数量。

query\_cache：查询缓存的统计数据，包括了缓存大小、缓存上限、命中/未命中次数等数据。

refresh：段刷新的统计数据，包括刷新次数、耗费时间等数据。

search：文档搜索的统计数据。

segments：段统计数据，包括数目、大小等。

store：索引所占用的磁盘空间大小。

translog：包括写入次数、大小、未提交的translog数量和大小。

#### 7.2、如何监控Elasticsearch

7.2.1、查询性能监控

Elasticsearch查询分为六个阶段：

Fetch

Query

Route

**Load&Return**

Response

MultiGet

Exec&Gather

Request

1. Request：客户端发送查询请求至集群某个节点NodeCoord。
2. Route：NodeCoord将查询请求发送至索引的一组完整分片上。
3. Execute&Gather：每个分片在本地执行查询操作并将结果传递至NodeCoord。
4. MultiGet：NodeCoord根据查询结果确定需要取回的文档和对应分片，然后向对应分片发送multi get请求。
5. Load&Return：收到请求的分片会加载文档并发送至NodeCoord。
6. Response：NodeCoord将查询结果发送至客户端。

需要监控的指标：

|  |  |
| --- | --- |
| name | desc |
| indices.search.query\_total | 查询总数 |
| indices.search.query\_time\_in\_millis | 用于查询的总时间 |
| indices.search.query\_current | 目前正在进行的查询数量 |
| indices.search.fetch\_total | 执行文档取回的次数 |
| indices.search.fetch\_time\_in\_millis | 执行文档取回的总时间 |
| indices.search.fetch\_current | 目前正在执行文档取回的任务量 |

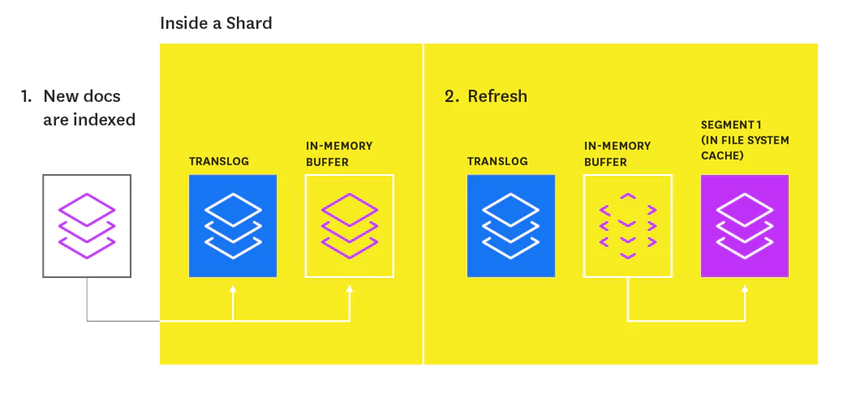
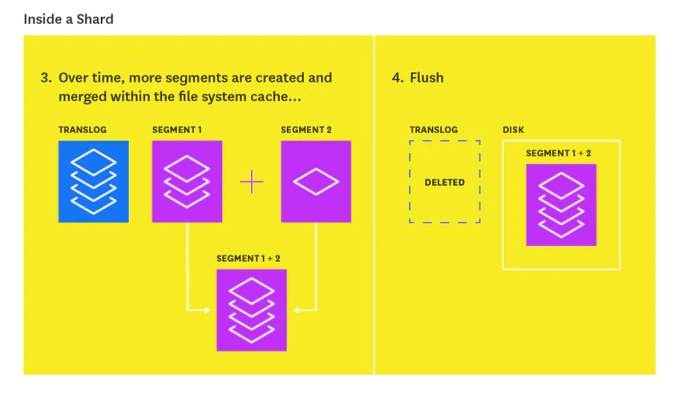
需要分析的指标：

|  |  |
| --- | --- |
| name | desc |
| query latency | 通过采样均匀时间间隔内的查询数和查询时间，计算出平均查询延迟 |
| fetch latency | 通过均匀间隔采样计算文档取回延迟 |

7.2.2、索引性能监控

每个分片内索引更新的过程分为两个阶段：refresh和flush。

1、新文档会首先被写入in-memory buffer中，同时附加在write-ahead log（translog）中。



1. 默认每隔一秒会进行一次索引刷新，使用in-memory buffer中的内容生成一个in-memory segment，然后清空in-memory buffer，此时文档可检索。
2. 每隔30分钟或是translog被写满，会触发flush操作。所有暂存于in-memory buffer中的文档会被刷新（refresh），生成新的in-memory segment。所有in-memory segment会被写入磁盘，同时清空translog。

在Elasticsearch中，每个分片由多个segment组成，每个segment是一个倒排索引，用于在词汇与文档之间建立映射关系。segment是不可变的，因此每当有文档被更改或删除，会在refresh期间将更新后的文档写入一个新的segment中，同时在旧的segment中标记该文档为删除状态。由于Elasticsearch默认每秒会生成一个新的segment，为了避免数量爆炸，会有专门的后台线程处理segment合并，小的segment会被合并为更大的segment。在合并期间，被标记为删除的文档不会合并进新的segment中。

需要监控的指标：

|  |  |
| --- | --- |
| name | desc |
| indices.indexing.index\_total | 被索引的文档总数 |
| indices.indexing.index\_time\_in\_millis | 索引文档花费的时间 |
| indices.indexing.index\_current | 正在被索引的文档数量 |
| indices.refresh.total | 索引刷新总数 |
| indices.refresh.total\_time\_in\_millis | 索引刷新花费的时间 |
| indices.flush.total | 写入到磁盘的索引数 |
| indices.flush.total\_time\_in\_millis | 索引写入磁盘花费的时间 |

需要分析的指标：

|  |  |
| --- | --- |
| name | desc |
| indexing latency | 通过index\_total与index\_time\_in\_millis可以计算出索引延迟 |
| flush latency | 数据持久化延迟 |

7.2.3、JVM垃圾回收监控

需要监控的指标：

|  |  |
| --- | --- |
| name | desc |
| jvm.gc.collectors.young.collection\_count | 年轻代垃圾回收次数 |
| jvm.gc.collectors.young.collection\_time\_in\_millis | 用于年轻代垃圾回收的总时间 |
| jvm.gc.collectors.old.collection\_count | 老年代垃圾回收次数 |
| jvm.gc.collectors.old.collection\_time\_in\_millis | 用于老年代垃圾回收的总时间 |
| jvm.mem.heap\_used\_percent | JVM堆使用百分比 |
| jvm.mem.heap\_committed\_in\_bytes | 由JVM保证能用的堆大小 |
| jvm.mem.heap\_used\_in\_bytes | 已使用的堆大小 |

需要分析的指标：

|  |  |
| --- | --- |
| name | desc |
| garbage collection duration | 通过对垃圾回收指标等间隔采样可计算出 |
| garbage collection frequency | 通过对垃圾回收指标等间隔采样可计算出 |

7.2.4、节点监控

需要监控的指标：

|  |  |
| --- | --- |
| name | desc |
| available disk space |  |
| I/O utilization | 磁盘I/O情况 |
| CPU usage |  |
| network bytes sent/received |  |
| open file descriptors | 文件描述符，用于节点间通信、客户端通信和文件操作 |
| http.current\_open | 目前打开的http连接数 |
| http.total\_opened | 历史上打开的http连接总数 |

7.2.5、集群健康程度监控

需要监控的指标：

|  |  |
| --- | --- |
| name | desc |
| cluster.health.status | green、yellow或red |
| cluster.health.number\_of\_nodes | 集群中节点总数 |
| cluster.health.initializing\_shards | 处于初始化状态的分片数，通常发生于新建索引或是节点重启后 |
| cluster.health.unassigned\_shards | 未分配的分片数 |

7.2.6、线程池监控

Elasticsearch提供了多种线程池的监控信息，其中search、merge和write线程池的信息最为常用。

需要监控的指标：

|  |  |
| --- | --- |
| name | desc |
| thread\_pool.search.queue  thread\_pool.merge.queue  thread\_pool.write.queue | 缓存于队列的任务数 |
| thread\_pool.search.rejected  thread\_pool.merge.rejected  thread\_pool.write.rejected | 被拒绝的任务数 |
| thread\_pool.search.active  thread\_pool.merge.active  thread\_pool.write.active | 正在执行的任务数 |

7.2.7、缓存监控

Elasticsearch包含了三种缓存，分别是query cache（filter cache），request cache，fielddata cache。query cache会对查询中包含了过滤器执行结果进行缓存，之后使用相同过滤器的查询可以直接使用缓存结果。request cache缓存分片查询的本地结果集，这些本地结果集之后会被发送至协调节点（NodeCoord），当查询的DSL语句一样时则命中缓存，分片每次执行完refresh操作后会被清空。字段排序与聚合会用到fielddata，其对倒排索引做了un-invert操作，可以查到每个文档包含的词语（terms），对倒排索引做逆转非常耗内存，因此Elasticsearch对fielddata做了缓存。

需要监控的指标：

|  |  |
| --- | --- |
| name | desc |
| indices.query\_cache.memory\_size\_in\_bytes  indices.request\_cache.memory\_size\_in\_bytes  indices.fielddata.memory\_size\_in\_bytes | 缓存大小 |
| indices.query\_cache.evictions  indices.request\_cache.evictions  indices.fielddata.evictions | 被移出缓存的条目数 |
| indices.query\_cache.hit\_count  indices.request\_cache.hit\_count | 缓存命中次数 |
| indices.query\_cache.miss\_count  indices.request\_cache.miss\_count | 缓存未命中次数 |

7.2.8、GET请求监控

需要监控的指标：

|  |  |
| --- | --- |
| name | desc |
| indices.get.total | GET请求总数 |
| indices.get.time\_in\_millis | 处理请求花费的总时间 |
| indices.get.exists\_total | 请求成功的GET请求总数 |
| indices.get.exists\_time\_in\_millis | 处理该类请求花费的总时间 |
| indices.get.missing\_total | 请求失败的GET请求总数 |
| indices.get.missing\_time\_in\_millis | 处理该类请求花费的总时间 |
| indices.get.current | 正在处理的GET请求数 |

7.2.9、Elasticsearch指标采集

7.2.10、Elasticsearch常见性能问题解决方案

#### 7.3、Kibana

#### 7.4、ElasticsearchHq

#### 7.5、Cerebro

#### 7.6、ElasticsearchHead