ElasticSearch summry

ElasticSearch Basic Config

elasticSearch常用的配置文件如：elasticsearch.yml，log4j2.properties，roles.yml，user_roles，jvm.options，role_mapping.yml和users

ElasticSearch架构

ElasticSearch具有高可用性实时分布式**搜索引擎**和**分析引擎**的功能，适用于全文本搜索，结构化搜索和数据分析。

ElasticSearch集群是指一个节点或者多个ElasticSearch节点连接到一起，尽管每个节点都有自己的目的和任务，但是每个节点都可以将客户端的请求转发到适当的节点。

ElasticSearch集群中使用的节点的种类：

- 符合主节点资格的节点：
  主节点的任务主要是负责轻量级集群范围内的操作，包括创建索引或删除索引，追踪集群节点，以及确定所分配的节点的位置。默认情况下，主节点角色处于启用状态，任何符合主节点，资格的节点都可以通过节点选举过程成为主节点，可以通过elasticsearch.yml文件将node.master设置为false，来禁止该节点成为主节点。
- 数据节点：
  包含索引文档数据，处理相关的操作。例如创建、读取、更新、删除、搜索和聚合。默认情况下，数据节点角色处于启用状态。可以通过elasticsearch.yml文件将node.data设置为false。
- 提取节点
- 机器学习节点
- 转换节点
- 协调节点：如果上述角色均被禁用，则该节点仅充当执行路由请求，处理缩减搜索结果，并通过批量索引操作来分发工作的协调节点。

Elasticsearch Index

索引，可以想象成一种数据结构，使你快速的随机访问存储在索引中的关键词（关键词对应存储在索引中的字段），进而找到关键词所关联的文档。

Lucene索引原理

Lucene 是一个开源的全文检索引擎工具包。它不是一个完整的全文检索引擎，而是一个全文检索引擎的架构，它提供了完整的查询引擎和索引引擎，部分文本分析引擎。
Lucene 采用的一种称为倒排索引（Invered index）的机制，倒排索引就是说我们维护了一个

词/短语表，对于这个词/短语表，都有一个链表描述了有哪些文档包含了这个词/短语。这样在用户输入查询条件的时候，就可以非常快的得到搜索结果。简单理解来说：

1. 在术语表中找到该索引。由于术语表是按字母或拼音排序的，这一步不会很难，和查字典差不多；
2. 根据术语表中记录的页码，直接跳转到书中相应的那些位置。这一步同样也会很快；

倒排索引和这个很像，可以理解为一个哈希表：

- 哈希表中的每个键是一个词项（term），相当于术语表中的术语；
- 每个词项的对应值则是出现该词项的所有文档的编号，相当于术语表中出现该术语的所有页码位置；

Lucene索引是分段的：**为什么Lucene要对索引分段，为什么断是不可变的。**

既然倒排索引本质上就是个哈希表，存成一个文件不就好了？为什么要分段呢?
其实这也是一种无可奈何。在理想情况下，如果只需要建一次索引，那么只存为一个文件当然没问题，完全不需要分段。

但现实情况往往会更复杂，比如你已经对现有的网页建了索引，落成了索引文件，但每天都会有新的网页冒出来，旧的网页也会修改，这时候就需要更新索引。

如果索引不分段，那么就意味着你唯一的这个索引文件，在每次更新索引的时候都会被修改。

如果索引更新非常频繁（比如在 Elasticsearch 中默认是每秒），那么也就意味着你唯一的这个索引文件，每秒都在被更新。
这样会带来非常不好的影响。比如：

1. 单个文件不断被频繁写入和读取，容易产生性能瓶颈，也容易导致数据损坏；
2. 对该文件的缓存将变得毫无意义；

第二点尤为关键。例如在 Elasticsearch 集群内，为了提供更高的查询性能、保证更高的可用性，每份 Lucene 索引一般都会有几个副本，每次索引更新都会同步给所有副本。问题是，如果索引每秒都在更新，每秒钟都需要重写所有副本，效率也太低了吧！
这些问题都可以通过分段解决。分段之后，每次更新索引时会写入新的段，而之前的段则不会被修改。
比如，如果你有 10 个段，那么实际上相当于有 10 份倒排索引，每次查询某个词项时去这 10 个倒排索引中分别查询，再对结果进行汇总，这样和查一个单独的倒排索引的效果是一样的。
这样的好处是，由于之前的段都是不变的，缓存效率会很高；每个段只会被写入一次，也避免了频繁写入的问题。
可以发现，分段策略和“段的不可变性”是紧密关联的。如果段不是不可变的，那么分段的好处也就所剩无几了。

分段会带来什么问题?
分段策略非常好，但也会带来一些问题。
最大的问题是：由于之前的段是不可变的，如果希望删除之前段中的某些内容（比如，删除一

些文档），那么只能过通过“标记删除”的方式实现，即在新增的段中说明哪些段被删除了，然后在结果汇总阶段把相关结果剔除掉。
这样带来的影响就是，本来应该已经删除的内容，实际上还保留在倒排索引中，一方面会影响查询效率，另一方面也占用着存储空间。
文档更新也是同样的道理。如果某个文档的内容变化了，那么 Lucene 实际上会先把该文档删除掉（通过标记删除的方法），然后再新增一份最新版本的文档。
换句话说，文档更新会带来和文档删除一样的问题：查询效率降低、存储空间浪费。
如果你的索引更新主要是文档新增，那么还好；如果有频繁的文档删除和文档更新，那么索引就会逐渐变得非常低效。这个时候就需要“段合并”（segment merge）了。段合并操作会删除旧有段，写入合并之后的新段。
从搜索结果来看，段合并之前和之后没有任何变化。不过，段合并期间会剔除已经被删除的文档，提高查询效率、减少储存空间浪费。

对文档建立好索引之后，就可以在这些索引上面进行搜索，搜索引擎首先会对搜索的关键词进行解析，然后再在建立好的索引上面进行查找，最终返回和用户输入的关键词相关联的文档。
索引的设置可以对个别索引或者是对全局索引，基本上，全局级别的目的是避免逐个地设置索引
全局级别的索引包括：

1. 断路器
2. 字段数据缓存
3. 节点查询缓存
4. 索引缓冲区
5. 分片请求缓存
6. 恢复
7. 搜索设置
   个别索引的设置可以分为静态和动态，静态设置只能在创建索引或者关闭索引时设置，而动态设置可以随时更改。

获取索引设置

```
curl -XGET http://localhost:9200/xxxxx/_settings?pretty=true_
```

更新个别动态索引设置

```
curl -XPUT -H "Content-Type: application/json"
http://localhost:9200/xxxx/_settings?pretty=true
--data $'{"index": {"number_of_replicas": 2}' {"acknowledged": true}
```

Elasticsearch 不支持删除索引设置操作。

重置索引设置：在更新索引设置操作中指定某设置为空值(null)，则会将其重置为默认值。

索引存储原理

单个节点由于物理机硬件限制，存储的文档是有限的，如果一个索引包含海量文档，则不能在单个节点存储。ES 提供分片机制，同一个索引可以存储在不同分片（数据容器）中。

分片分为主分片 (primary shard) 以及从分片 (replica shard)。主分片会被尽可能平均地 (rebalance) 分配在不同的节点上（例如你有 2 个节点，4 个主分片（不考虑备份），那么每个节点会分到 2 个分片，后来你增加了 2 个节点，那么你这 4 个节点上都会有 1 个分片，这个过程叫 relocation，ES 感知后自动完成)。从分片只是主分片的一个副本，它用于提供数据的冗余副本，从分片和主分片不会出现在同一个节点上（防止单点故障），默认情况下一个索引创建 5 个主分片，每个主分片会有一个从分片 (5 primary + 5 replica = 10 个分片)。如果你只有一个节点，那么 5 个 replica 都无法被分配 (unassigned)，此时 cluster status 会变成 Yellow。

分片是独立的，对于一个 Search Request 的行为，每个分片都会执行这个 Request。每个分片都是一个 Lucene Index，所以一个分片只能存放 Integer.MAX_VALUE - 128 = 2,147,483,519 个 docs。

replica 的作用主要包括：

1. 容灾：primary 分片丢失，replica 分片就会被顶上去成为新的主分片，同时根据这个新的主分片创建新的 replica，集群数据安然无恙；
2. 提高查询性能：replica 和 primary 分片的数据是相同的，所以对于一个 query 既可以查主分片也可以查从分片，在合适的范围内多个 replica 性能会更优（但要考虑资源占用也会提升 [cpu/disk/heap]），另外 Index Request 只能发生在主分片上，replica 不能执行 Index Request。

注意：对于一个索引，除非重建索引否则不能调整主分片的数目 (number_of_shards)，但可以随时调整 replica 的数目 (number_of_replicas)。

文档是Elasticsearch的基本数据单位，Elasticsearch通过对文档中的词条进行标记，将文档列表并与可以找到这些词条的位置相关联累创建反向索引。
Elasticsearch中的索引由一个或者多个分片组成，分片就是lucene索引。

Elasticsearch 文档管理

在文档被索引期间，文档将与元字段（meta-fields）相关联，元字段是描述文档本身的字段，包括给定的索引名称，映射类型，文档标识符和路由等。而映射是描述文档结构的数据模型，包括指定字段，字段类型，文档之间的关系与数据转换规则等。

文档路由

无论是存储或者是搜索操作，文档都被转送到特定的分片进行。
路由routing过程如图所示：
（1）Elasticsearch客户端将文档发送到Elasticsearch集群：第一站是协调节点
（2）文档转发到路由公示确定的目标主分片所在的数据节点。
（3）分析器从文档各键值对的值生成分词并创建倒排索引，并将其存储到Lucene分段中。
（4）主分片负责在适当的时候将文档相关的索引操作转发到当前同步副本分片的每个副本进

行复制
（5）当所有副本都成功执行了复制操作并响应了主分片，主分片便会向客户端确认请求成功，而文档标识符_id将在响应正文中返回。

ElasticSearch 搜索

运行搜索的基本方法有两种：

1. 简单的URL参数的搜索请求
2. 使用查询表达式
   其中包括通配符来匹配多个字段的搜索等，符合给定的条件的文档会按照相似性进行排名，以显示查询结果的匹配程度。
   查询表达式基于JSON的搜索语言来查询特定数据集和分析数据集。基本上，查询句子分为两部分：查询操作和过滤操作。过滤操作基于搜索结果，可以继续对其进行条件匹配过滤。

ElasticSearch API Summery

- 查看ES的运行状态

```
GET /_cat/health // 返回的结果可能有三种：green yellow red

yellow的状态可能是因为当前ES上的所启动的所有节点，不满足索引所设置的shards和replicas的数目
```

- 创建名字为test的index（改索引的mapping为空）

```
PUT /test
{
        "settings": {
            //该索引在master节点上主片的数目为1，创建之后不能修改，但是可以用脚本将改节点上的数据迁移
                "number_of_shards": 1,
                "number_of_replicas": 0 // 该索引在从节点的分片数量为0，创建之后可以修改
        }
}
```

- 非结构化方式**新建/更新**(此时更新是覆盖source)索引(即请求中没有表明index中的mapping关系，即表结构中字段的类型，根据传入的字段的数据判断)

```
PUT /xxxxx/_doc/1(1这里表示id，可以替换)
{
        "name": "test",
        "age": 12
}
```

- 获取索引记录

```
GET /xxxx/_doc/1 （获取id为1的索引记录）
```

- 指定字段修改（此时更新是更新是更新source特定字段）

```
POST /xxxx/_update/1
{
        "doc": {
                "name": xxxxxx
        }
}
```

- 强制指定创建，若已经存在则失败

```
POST /xxxx/_create/1
{
        "name": 1123
        "age": 23
}
```

- 删除某个特定文档

```
DELETE /xxxx/_doc/2
```

- 查询全部文档(有默认的分页值)

```
GET /xxxx/_search
```

- 使用结构化的方式创建索引

```
PUT /xxx
{
        "setttings": {
                "number_of_shards": 1,
                "number_of_replicas": 1,
        },
        "mapping": {
                "properties": {
                        "name": {"type": "text"},
                        "age": {"age": "integer"}
                }
        }
}
```

- 不带条件查询所有记录

```
GET /xxxx/_search
{
        "query": {
                "match_all": {}
        }
}
```

- 分页查询

```
GET /xxxx/_search
{
        "query": {
                "match_all": {}
        },
        "from": 0,
        "size": 1
}
```

- 带关键字条件的查询

```
GET /xxxx/_search
{
        "query": {
                "match": {
                        "name": "兄弟" // 这里是模糊查询
                }
        }
}
```

- 带排序的查询

```
GET /xxxx/_search
{
        "query": {
                "match": {
                        "name": {"兄"}
                }
        },
        "sort": [{
                "age": {"order": "desc"}
        }]
}
```

- 带filter的查询

```
GET /xxxx/_search
{
        "query": {
                "bool": {
                        "filter": [{
                                "term":{"age": 30}
                        }]
                }
        },
}
```

- 带聚合的查询

```
GET /xxxx/_search
{
        "query": {
                "match": {
                        "name": {"兄"}
                }
        },
        "sort": [{
                "age": {"order": "desc"}
        }],
        "aggs": {
                "group_by_age": { // 这个key可以随便起名字
                        "terms": {
                                "filed": "age"
                        }
                }
        }
}
```

- 使用analyze api查看分词状态

```
GET /movie/_analyze
{
        "field": "name",
        "text": "Eating an apple,....."
}
```

ElasticSearch API Advanced

- 将查询关键字从or改为and

```
GET /movie/_search
{
        "query": {
                "match": {
                        "title": {
                                "query": "basketball with cartoom aliens",
                                "operator": "and"
                        }
                }
```

```
        }
}
```

- 最小词匹配项

```
GET /movie/_search

{
        "query": {
                "match": {
                        "title": {
                                "query": "basketball with cartoom aliens",
                                "operator": "or",
                                "minimum_should_match": 2 // 分词之后的
token必须同时出现2个
                        }
                }
        }
}
```

- 短语查询

```
GET /movie/_search
{
        "query": {
                "match_phrase": {"title": "steve zissou"} //短语匹配，搜索时
短语两次同时存在
        }
}
```

- 多字段查询

```
GET /movie/_search
{
        "query": {
                "multi_match": {
                        "query": "basketball with cartoom aliens",
                        "fields": ["title", "overview"]
                }
```

```
        }
}
```

- 优化多字段查询

```
GET /movie/_search
{
        "query": {
                "multi_match": {
                        "query": "basketball with cartoom aliens",
                        "fields": ["title^10", "overview"] // 增加系数
                        ”tie_breaker“: 0.3
                }
        }
}
```

- 如何查看score怎么计算出来

```
GET /movie/_search
{
        "explain": true,
        "query": {
                "match": {"title": "steve"}
        }
}
```

- bool查询
  - must: 必须都为true
  - must not: 必须都是false
  - should: 其中只有一个为true即可
  - 为true的越多则得分越高

```
GET /movie/_search
{
        "query": {
                "bool": {
                        "should": [
                                {"match": {"title": "xxxxxxxxx"}},
                                {"match": {"overview": "xxxxxxxx"}}
```

```
                        ]
                }
        }
}
```

- 查看分数的计算公式

```
GET /movie/_validate/query?explain
{
        "query": {
                "multi_match": {
                        "query": "xxxxx",
                        "fields": ["title^10", "overview"],
                        "type": "best_fields"
                }
        }
}
```

- multi_query其实是有不同的type
  - best_fields：默认的得分方式，取得最高的分数作为对应文档的对应分数，"最匹配模式"
  - most_fields：考虑所有的文档的字段得分相加，获得我们想要的结果
  - cross_fields：以分词为单位计算栏位的总分

```
GET /movie/_search
{
        "query": {
                "multi_match": {
                        "query": "basketball with cartoom aliens",
                        "fields": ["title", "overview"],
                        "type": "best_fields"
                }
        }
}
```

- query string
  - 方便的利用 AND OR NOT

```
GET /movie/_search
{
        "query": {
                "query_string": {
                        fields: ["title"],
                        "query": "steve OR jobs"
                }
        }
}
```

- filter过滤
  - 单条件过滤
  - 多条件过滤
  - 带match打分的filter

```
GET /movie/_search
{
        "query": {
                "query_string": {
                        fields: ["title"],
                        "query": "steve OR jobs"
                }
        }
}
{
        "query": {
                "query_string": {
                        "bool": {
                                "filter": [
                                        {"term": {"title": "xxxx"}},
                                        {"term": {"cast.name":
"asdasdas"}}
                                ]
                        }
                }
        }
}
{
        "query": {
                "query_string": {
                        "bool": {
```

```
                                "should": [
                                        {"match": {"title": "xxxxx"}}
                                ],
                                "filter": [
                                        {"term": {"title": "xxxx"}},
                                        {"term": {"cast.name":
"asdasdas"}}
                                ]
                        }
                }
        }
}
```

ElasticSearch Conception Summary

- ES中的index就相当于表
- ES上的主节点只负责写请求的路由，而不负责读请求的路由
- ES上进行search查询的过程中，如果没有指定**term**关键字，那么查询的过程中就会使用ES上带有的分词器，对于条件进行分词，然后对于对应查询字段进行分词，如果有匹配的词就会算上查询的结果（返回的结果_score会表明结果的匹配程度），如果使用term和filter关键字，则就不会进行模糊查询。
- ES的analyze（分词）过程：
  1. 字符过滤器
  2. 字符处理的过程（标准字符处理，以空格和标点符合分割内容）
  3. 分词过滤（分词转换）将所有的字符变成小写
- ES中的类型
  1. Text：被分析索引的字符串类型
  2. KeyWord：不能被分析只能被精确匹配的的字符串类型
  3. Date：日期类型，可以配合format一起的使用
  4. 数字类型：long, integer, short, double
  5. boolean：true，false
  6. Array: ["one", "two"]
  7. Object: json嵌套

- ES的名词map

Relational database	*Elasticsearch*
Database	Index
Table	Type
Row	Document
Column	Field
Schema	Mapping
Index	Everything is indexed
SQL	Query DSL
SELECT * FROM table...	GET http://...
UPDATE table SET	PUT http://...

ES7之后Type被完全废弃，即只有index(索引，等同于数据库+表定义)和Document（文档，即行记录）