

硅谷大数据技术之 ElasticSearch

(作者：尚硅谷研究院)

版本：V2.0

第 1 章 Elasticsearch 简介

ElasticSearch 是一个基于 Lucene 的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于 RESTful web 接口。Elasticsearch 是用 Java 开发的，并作为 Apache 许可条款下的开放源码发布，是当前流行的企业级搜索引擎。

1.1 ElasticSearch 的使用场景

- (1) 为用户提供按关键字查询的全文搜索功能。
- (2) 实现企业海量数据的处理分析的解决方案。大数据领域的重要一份子，如著名的 ELK 框架(ElasticSearch,Logstash,Kibana)。
- (3) 作为 OLAP 数据库，对数据进行统计分析。

1.2 与其他数据存储进行比较

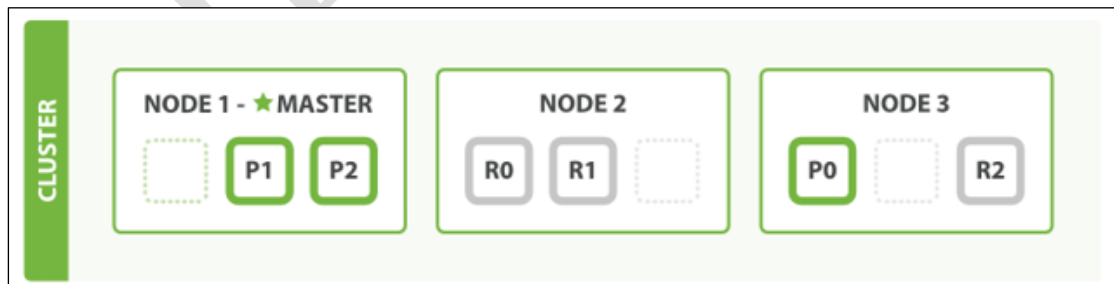
	redis	mysql	elasticsearch	hbase	hadoop/hive
容量/容量扩展	低	中	较大	海量	海量
查询时效性	极高	较高 (需要索引优	较高	较高 (rowkey 方式) 较低(scan 方式)	低

		化)			
查询灵活性	最差 k-v 模式	非常好，支持 sql	较好，关联查询较弱，但是可以全文检索，DSL 语言可以处理过滤、匹配、排序、聚合等各种操作	较差，主要靠 rowkey，scan 的话性能不行，或者安装 phoenix 插件来实现 sql 及二级索引	非常好，支持 sql
写入速度	极快	中等 (同步写入)	较快 (异步写入)	较快 (异步写入)	慢
一致性、事务	弱	强	弱	弱	弱

1.3 Elasticsearch 的特点

1.3.1 天然的分布式数据库

ES 把数据分成多个 shard (分片)，下图中的 P0-P2，多个 shard 可以组成一份完整的数据，这些 shard 可以分布在集群中的各个机器节点中。随着数据的不断增加，集群可以增加多个分片，把多个分片放到多个机子上，已达到负载均衡，横向扩展。



1.3.2 天然索引之倒排索引

ES 所有数据都是默认进行索引的，这点和 mysql 正好相反，mysql 是默认不加索引，要加索引必须特别说明，ES 只有不加索引才需要说明。

而 ES 使用的是倒排索引和 Mysql 的 B+Tree 索引不同。

1) 传统关系性数据库索引

(1) 传统的保存数据的方式是 记录→单词



(2) 弊端:

对于传统的关系性数据库对于关键词的查询，只能逐字逐行的匹配，性能非常差。匹配方式不合理，比如搜索“小密手机”，如果用 like 进行匹配，根本匹配不到。但是考虑使用者的用户体验的话，除了完全匹配的记录，还应该显示一部分近似匹配的记录，至少应该匹配到“手机”。

2) 倒排索引

全文搜索引擎目前主流的索引技术就是倒排索引的方式。

(1) 传统的保存数据的方式是 记录 → 单词

id	标题
1	红海行动影评
2	红海事件始末
3	湄公河行动导演
4

(2) 倒排索引的保存数据的方式是 单词→记录

分词	ids
红海	1,2
行动	1,3
影评	1
事件	2
始末	2
湄公河	3
导演	3

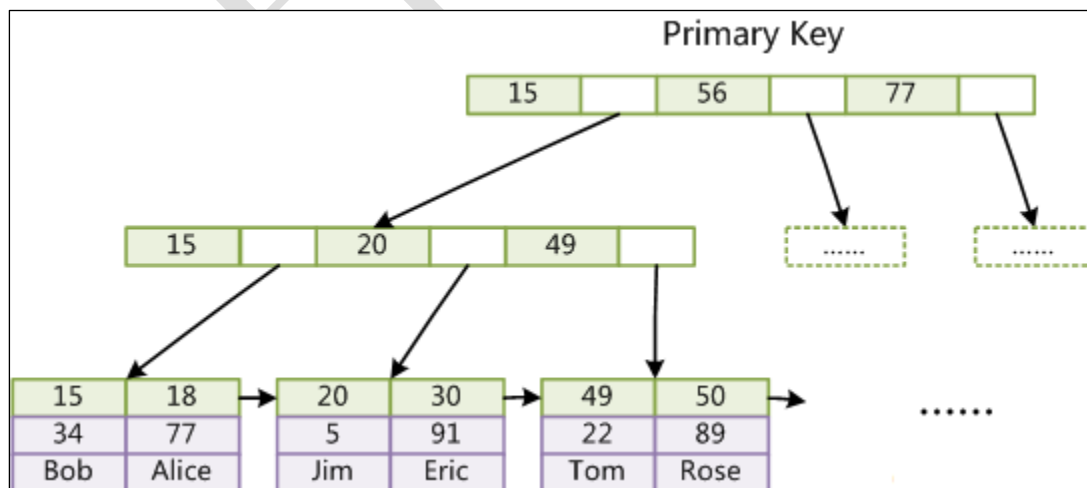
(3) 基于分词技术构建倒排索引：

首先每个记录保存数据时，都不会直接存入数据库。系统先会对数据进行分词，然后以倒排索引结构保存。然后等到用户搜索的时候，会把搜索的关键词也进行分词，会把“红海行动”分词分成：红海和行动两个词。这样的话，先用红海进行匹配，得到 id 为 1 和 2 的记录编号，再用行动匹配可以迅速定位 id 为 1,3 的记录。

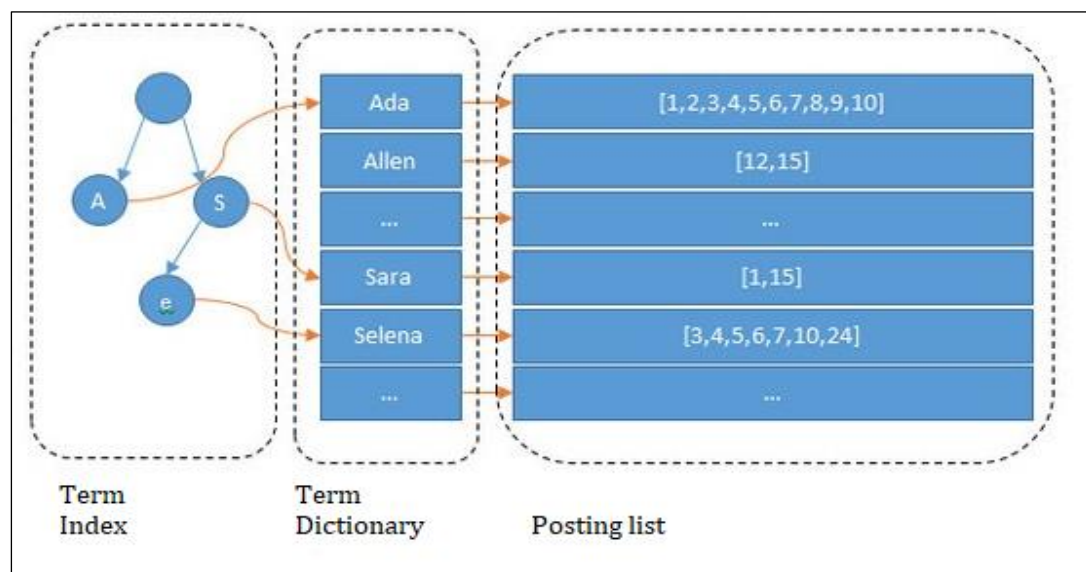
那么全文索引通常，还会根据匹配程度进行打分，显然 1 号记录能匹配的次数更多。所以显示的时候以评分进行排序的话，1 号记录会排到最前面。而 2、3 号记录也可以匹配到。

3) 索引结构对比

(1) B+Tree



(2) lucene 倒排索引结构



可以看到 lucene 为倒排索引(Term Dictionary)部分又增加一层 Term Index 结构, 用于快速定位, 而这 Term Index 是缓存在内存中的, 但 mysql 的 B+tree 不在内存中, 所以整体来看 ES 速度更快, 但同时也更消耗资源 (内存、磁盘)。

Term index → Term dictionary → posting list

1.3.3 天然索引之正排索引(Doc Value 列式存储)

倒排索引在搜索包含指定词条的文档时非常高效, 但是在相反的操作时表现很差: 查询一个文档中包含哪些词条。具体来说, 倒排索引在搜索时最为高效, 但在排序、聚合等与指定字段相关的操作时效率低下, 需要用 doc_values。

在 Elasticsearch 中, Doc Values 就是一种列式存储结构, 默认情况下每个字段的 Doc Values 都是激活的。

索引中某个字段的存储结构如下:

1	Doc	Terms
2	-----	
3	Doc_1	100
4	Doc_2	1000
5	Doc_3	1500
6	Doc_4	1200
7	Doc_5	300
8	Doc_6	1900
9	Doc_7	4200
10	-----	

列式存储结构非常适合排序、聚合以及字段相关的脚本操作。而且这种存储方式便于压缩，尤其是数字类型。压缩后能够大大减少磁盘空间，提升访问速度。

1.4 lucene 与 Elasticsearch 的关系

咱们之前讲的处理分词，构建倒排索引，等等，都是这个叫 lucene 的做的。那么能不能说这个 lucene 就是搜索引擎呢？

还不能。lucene 只是一个提供全文搜索功能类库的核心工具包，而真正使用它还需要一个完善的服务框架搭建起来的应用。

好比 lucene 是类似于发动机，而搜索引擎软件（ES,Solr）就是汽车。

目前市面上流行的搜索引擎软件，主流的就两款，elasticsearch 和 solr,这两款都是基于 lucene 的搭建的，可以独立部署启动的搜索引擎服务软件。由于内核相同，所以两者除了服务器安装、部署、管理、集群以外，对于数据的操作，修改、添加、保存、查询等等都十分类似。就好像都是支持 sql 语言的两种数据库软件。只要学会其中一个另一个很容易上手。

从实际企业使用情况来看，elasticSearch 的市场份额逐步在取代 solr，国内百度、京东、新浪都是基于 elasticSearch 实现的搜索功能。国外就更多了 像维基百科、GitHub、Stack Overflow 等等也都是基于 ES 的。

第 2 章 Elasticsearch 安装

2.1 下载地址

本课程选择的版本是 elasticsearch-7.8.0

Elasticsearch 官网：

<https://www.elastic.co/products/elasticsearch>

<https://www.elastic.co/cn/downloads/past-releases/elasticsearch-7-8-0>

2.2 修改操作系统参数

因为默认 elasticsearch 是单机访问模式，就是只能自己访问自己。但是我们会设置成允许应用服务器通过网络方式访问，而且生产环境也是这种方式。这时，Elasticsearch 就会因为嫌弃单机版的低端默认配置而报错，甚至无法启动。所以我们在这里就要把服务器的一些限制打开，能支持更多并发。

1) 问题: max file descriptors [4096] for elasticsearch process likely too low, increase to at least [65536] elasticsearch

修改系统允许 Elasticsearch 打开的最大文件数需要修改成 65536。

```
[atguigu@hadoop102 es7]$ sudo vim /etc/security/limits.conf
#在文件最后添加如下内容:
* soft nofile 65536
* hard nofile 131072
* soft nproc 2048
* hard nproc 65536

#分发文件
[atguigu@hadoop102 es7]$ scp -r /etc/security/limits.conf
root@hadoop103:/etc/security/
[atguigu@hadoop102 es7]$ scp -r /etc/security/limits.conf
root@hadoop104:/etc/security/
```

2) 问题: max virtual memory areas vm.max_map_count [65530] likely too low, increase to at least [262144]

修改一个进程可以拥有的虚拟内存区域的数量。

```
[atguigu@hadoop102 es7]$ sudo vim /etc/sysctl.conf
#在文件最后添加如下内容
vm.max_map_count=262144

#分发文件
[atguigu@hadoop102 es7]$ scp -r /etc/sysctl.conf
root@hadoop103:/etc/
[atguigu@hadoop102 es7]$ scp -r /etc/sysctl.conf
root@hadoop104:/etc/
```

3) 问题: max number of threads [1024] for user [judy2] likely too low, increase to at least [4096] (CentOS7.x 不用改)

修改允许最大线程数为 4096

```
[atguigu@hadoop102 es7]$ sudo vim /etc/security/limits.d/20-
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
nproc.conf
#修改如下内容
* soft nproc 4096

#分发文件
[atguigu@hadoop102 es7]$ scp -r /etc/security/limits.d/20-
nproc.conf root@hadoop103:/etc/security/limits.d/
[atguigu@hadoop102 es7]$ scp -r /etc/security/limits.d/20-
nproc.conf root@hadoop104:/etc/security/limits.d/
```

4) 重启 linux 使配置生效

2.3 安装 ES

1) 上传安装包, 将 elasticsearch 安装包上传到 opt/software/目录下

```
[atguigu@hadoop102 software]$ ll
-rw-r--r--. 1 atguigu atguigu 319112561 4月 12 2021 elasticsearch-
7.8.0-linux-x86_64.tar.gz
```

2) 将 ES 解压到/opt/module 目录下

```
[atguigu@hadoop102 software]$ tar -zxvf elasticsearch-7.8.0.tar.gz
-C /opt/module/
```

3) 在/opt/module 目录下对 ES 重命名

```
[atguigu@hadoop102 module]$ mv elasticsearch-7.8.0 es7
```

4) 修改 ES 配置文件 elasticsearch.yml

```
[atguigu@hadoop102 es7]$ cd config/
[atguigu@hadoop102 config]$ vim elasticsearch.yml
```

(1) 集群名称, 同一集群名称必须相同

```
cluster.name: my-es
```

(2) 单个节点名称, 不同节点名称不能一样, 例如:node-1,node-2,node-3

```
node.name: node-1
```

(3) 把 bootstrap 自检程序关掉

```
bootstrap.memory_lock: false
```

(4) 网络部分

```
# 允许任意 ip 访问
network.host: 0.0.0.0
# 数据服务端口
http.port: 9200
# 集群间通信端口
transport.tcp.port: 9301
```

(5) 自发现配置: 新节点向集群报到的主机名

```
#集群的"介绍人"节点
discovery.seed_hosts: ["hadoop102:9301", "hadoop103:9301"]
#默认候选 master 节点
cluster.initial_master_nodes: ["node-1", "node-2", "node-3"]
#集群检测的超时时间和次数
```

8

更多 Java -大数据 -前端 -python 人工智能资料下载, 可百度访问: 尚硅谷官网


```
discovery.zen.fd.ping_timeout: 1m
discovery.zen.fd.ping_retries: 5
```

(6) 修改 yml 配置的注意事项:

每行必须顶格, 不能有空格

“:” 后面必须有一个空格

5) 教学环境启动优化

ES 是用在 Java 虚拟机中运行的, 虚拟机默认启动占用 1G 内存。但是如果是装在 PC 机学习用, 实际用不了 1 个 G。所以可以改小一点内存; 但生产环境一般 31G 内存是标配, 这个时候需要将这个内存调大。

```
[atguigu@hadoop102 config]$ vim jvm.options
-Xms512m
-Xmx512m
```

6) 分发 ES

```
[atguigu@hadoop102 module]$ xsync es7
```

7) 修改 hadoop103 和 hadoop104 上的节点名

```
#在 hadoop103 进行修改
[atguigu@hadoop103 config]$ vim elasticsearch.yml
node.name: node-2

#在 hadoop104 进行修改
[atguigu@hadoop104 config]$ vim elasticsearch.yml
node.name: node-3
```

2.4 启动测试

2.4.1 单台启动测试

1) 在 hadoop102 上执行启动命令

```
[atguigu@hadoop102 es7]$ bin/elasticsearch
.....
[2022-01-10T13:48:36,099][INFO ][o.e.n.Node                ] [node-1]
started
.....
```

2) 查看进程

```
[atguigu@hadoop102 ~]$ jps
1204 Elasticsearch
```

3) 命令行进行测试

```
[atguigu@hadoop102 config]$ curl http://hadoop102:9200
{
  "name" : "node-1",
```

```
"cluster_name" : "my-es",
"cluster_uuid" : "_na_",
"version" : {
  "number" : "7.8.0",
  "build_flavor" : "default",
  "build_type" : "tar",
  "build_hash" : "757314695644ea9a1dc2fec26d1a43856725e65",
  "build_date" : "2020-06-14T19:35:50.234439Z",
  "build_snapshot" : false,
  "lucene_version" : "8.5.1",
  "minimum_wire_compatibility_version" : "6.8.0",
  "minimum_index_compatibility_version" : "6.0.0-beta1"
},
"tagline" : "You Know, for Search"
}
```

4) 浏览器进行测试

在浏览器地址栏中输入: <http://hadoop102:9200> 进行访问

```
{
  "name" : "node-1",
  "cluster_name" : "my-es",
  "cluster_uuid" : "_na_",
  "version" : {
    "number" : "7.8.0",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "757314695644ea9a1dc2fec26d1a43856725e65",
    "build_date" : "2020-06-14T19:35:50.234439Z",
    "build_snapshot" : false,
    "lucene_version" : "8.5.1",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

2.4.2 集群启动

1) 集群启动脚本, 在/home/atguigu/bin 目录下创建 es.sh

```
#!/bin/bash

es_home=/opt/module/es7
kibana_home=/opt/module/kibana7

if [ $# -lt 1 ]
then
    echo "USAGE:es.sh {start|stop}"
    exit
fi

case $1 in
"start")
    #启动ES
```

10

更多 Java -大数据 -前端 -python 人工智能资料下载, 可百度访问: 尚硅谷官网

```

for i in hadoop102 hadoop103 hadoop104
do
    ssh $i "source /etc/profile;nohup
${es_home}/bin/elasticsearch >/dev/null 2>&1 &"
done
;;
"stop")
#停止 ES
for i in hadoop102 hadoop103 hadoop104
do
    ssh $i "ps -ef|grep $es_home |grep -v grep|awk '{print
\${2}}'|xargs kill" >/dev/null 2>&1
done

;;
*)
    echo "USAGE:es.sh {start|stop}"
    exit
;;
esac

```

2) 授权

```

[atguigu@hadoop102 bin]$ chmod u+x es.sh
[atguigu@hadoop102 bin]$ ll
-rwxrw-r--. 1 atguigu atguigu 430 1月 10 14:12 es.sh

```

3) 执行脚本启动 ES 集群

```

[atguigu@hadoop102 es7]$ es.sh start

```

4) 查看进程

```

[atguigu@hadoop102 bin]$ myjps.sh
=====> hadoop102 JPS <=====
2969 Elasticsearch
=====> hadoop103 JPS <=====
1684 Elasticsearch
=====> hadoop104 JPS <=====
1690 Elasticsearch

```

5) 命令行测试

```

[atguigu@hadoop102 es7]$ curl http://hadoop102:9200/_cat/nodes?v
ip                heap.percent ram.percent cpu load_1m load_5m load_15m
node.role master name
192.168.202.102    28          50      1    0.01    0.08    0.08
dilmrt *          node-1
192.168.202.103    42          42      1    0.02    0.07    0.07
dilmrt -          node-2
192.168.202.104    28          42      1    0.01    0.08    0.08
dilmrt -          node-3

```

6) 浏览器测试

在浏览器地址栏中输入 http://hadoop102:9200/_cat/nodes?v 进行访问。

```

ip                heap.percent ram.percent cpu load_1m load_5m load_15m

```

11

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

node.role	master	name					
192.168.202.102	31	50	0	0.00	0.07	0.08	
dilmrt	*	node-1					
192.168.202.103	16	42	0	0.04	0.08	0.07	
dilmrt	-	node-2					
192.168.202.104	29	42	0	0.00	0.07	0.07	
dilmrt	-	node-3					

第 3 章 Kibana 安装

3.1 Kibana 简介

Elasticsearch 提供了一套全面和强大的 REST API，我们可以通过这套 API 与 ES 集群进行交互。

例如：我们可以通过 API: GET /_cat/nodes?v 获取 ES 集群节点情况，要想访问这个 API，我们需要使用 curl 命令工具来访问 Elasticsearch 服务：curl http://hdp1:9200/_cat/nodes?v，当然也可以使用任何其他 HTTP/REST 调试工具，例如浏览器、POSTMAN 等。

Kibana 是为 Elasticsearch 设计的开源分析和可视化平台。你可以使用 Kibana 来搜索，查看存储在 Elasticsearch 索引中的数据并与之交互。你可以很容易实现高级的数据分析和可视化，以图表的形式展现出来。

3.2 安装 Kibana

1) 上传安装包将 Kibana 安装包上传到 opt/software/目录下

```
[atguigu@hadoop102 software]$ ll
-rw-r--r--. 1 atguigu atguigu 334236568 10 月 29 16:34 kibana-7.8.0-linux-x86_64.tar.gz
```

2) 解压到/opt/module 目录下

```
[atguigu@hadoop102 software]$
[atguigu@hadoop102 software]$ tar -zxvf kibana-7.8.0-linux-x86_64.tar.gz -C /opt/module/
```

3) 重命名

```
[atguigu@hadoop102 module]$ mv kibana-7.8.0-linux-x86_64/ kibana7
```

4) 修改 Kibana 配置文件

```
[atguigu@hadoop102 kibana7]$ cd config/
[atguigu@hadoop102 config]$ vim kibana.yml
```

5) 授权远程访问

```
server.host: "0.0.0.0"
```

6) 指定 Elasticsearch 地址（可以指定多个，多个地之间用逗号分隔）

```
elasticsearch.hosts: ["http://hadoop102:9200",
```

```
"http://hadoop103:9200"]
```

7) Kibana 本身只是一个工具，不需要分发，不涉及集群，访问并发量也不会很大

3.3 启动测试

1) 启动 Kibana

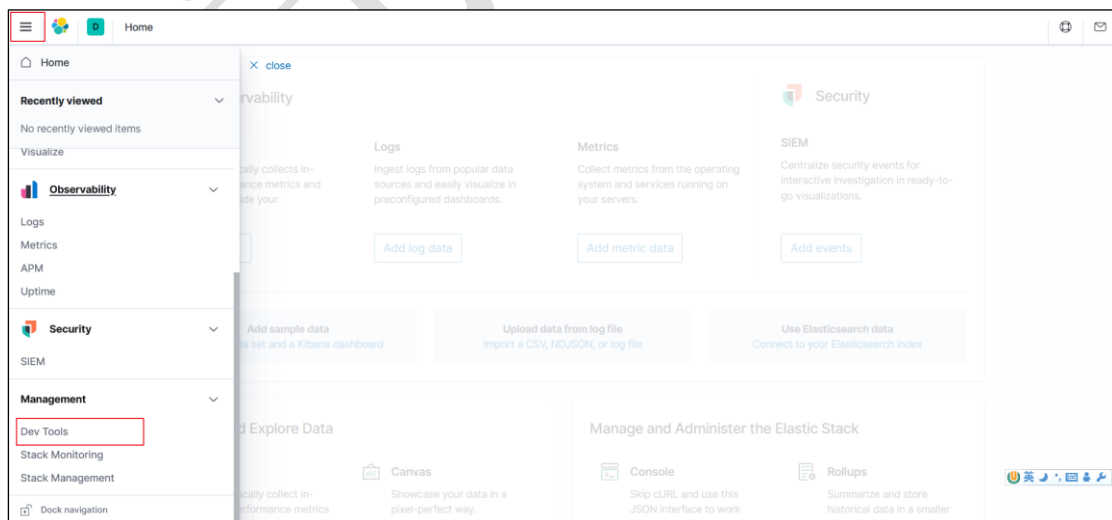
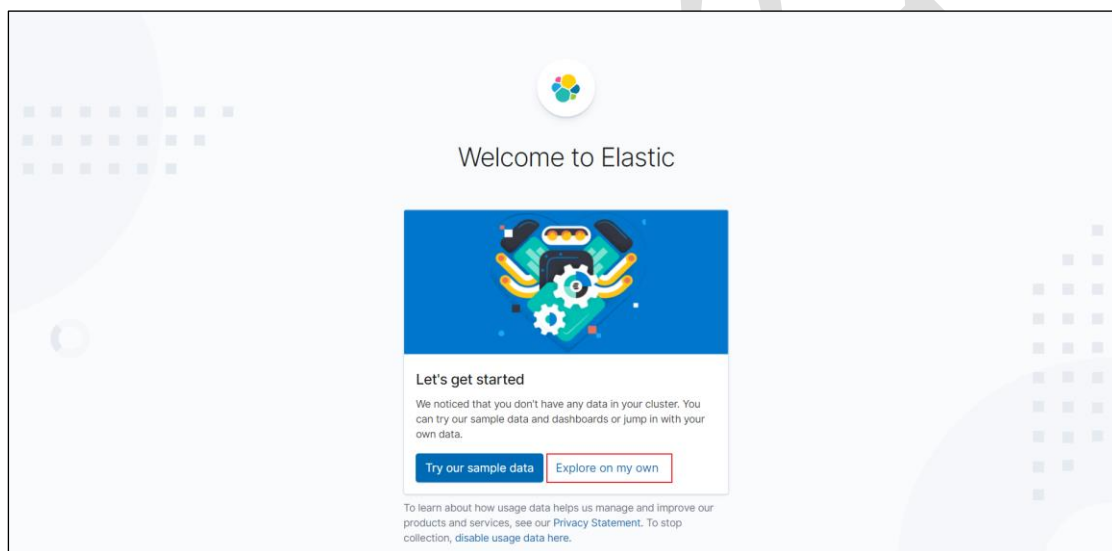
```
[atguigu@hadoop102 kibana7]$ bin/kibana
```

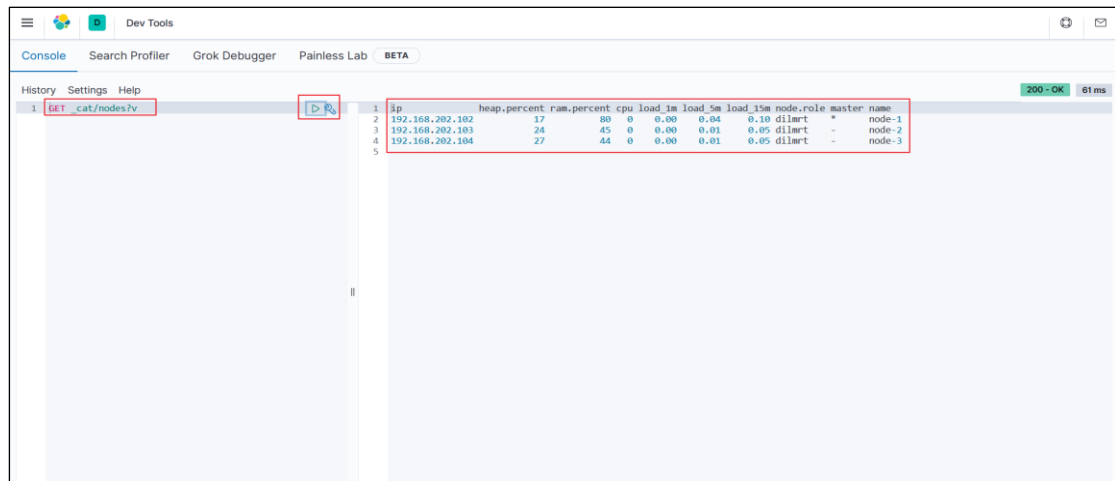
2) 查看进程

```
[atguigu@hadoop102 bin]$ sudo netstat -nltp | grep 5601
tcp        0      0 0.0.0.0:5601          0.0.0.0:*
LISTEN      3281/bin/../../node/bi
```

3) 浏览器访问

浏览器访问 <http://hadoop102:5601/>





ip	heap.percent	ram.percent	cpu	load_1m	load_5m	load_15m	node.role	master	name
192.168.202.102	17	80	0	0.00	0.04	0.10	dilmrt	*	node-1
192.168.202.103	24	45	0	0.00	0.01	0.05	dilmrt	-	node-2
192.168.202.104	27	44	0	0.00	0.01	0.05	dilmrt	-	node-3

4) 最终集群脚本

```
#!/bin/bash

es_home=/opt/module/es7
kibana_home=/opt/module/kibana7

if [ $# -lt 1 ]
then
    echo "USAGE:es.sh {start|stop}"
    exit
fi

case $1 in
"start")
    #启动 ES
    for i in hadoop102 hadoop103 hadoop104
    do
        ssh $i "source /etc/profile;nohup
${es_home}/bin/elasticsearch >/dev/null 2>&1 &"
    done
    #启动 Kibana
    nohup ${kibana_home}/bin/kibana > ${kibana_home}/logs/kibana.log
2>&1 &
;;
"stop")
    #停止 Kibana
    sudo netstat -nlt | grep 5601 | awk '{print $7}' | awk -F /
'{print $1}' | xargs kill
    #停止 ES
    for i in hadoop102 hadoop103 hadoop104
    do
        ssh $i "ps -ef|grep $es_home |grep -v grep|awk '{print
\$2}'|xargs kill" >/dev/null 2>&1
    done
;;
*)

```

14

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
echo "USAGE:es.sh {start|stop}"
exit
;;
esac
```

第 4 章 elasticsearch DSL

DSL 全称 Domain Specific language，即特定领域专用语言。

4.1 名词解释

cluster	整个 elasticsearch 默认就是集群状态，整个集群是一份完整、互备的数据。
node	集群中的一个节点，一般指一个进程就是一个 node
shard	分片，即使是一个节点中的数据也会通过 hash 算法，分成多个片存放，7.x 默认是 1 片,之前版本默认 5 片。
index	index 相当于 table
type	一种逻辑分区，7.x 版本已经废除，用固定的 _doc 占位替代。
document	类似于 rdbms 的 row、面向对象里的 object
field	相当于字段、属性

4.2 服务状态查询

4.2.1 服务整体状态查询

GET /_cat/health?v	
epoch	从标准时间(1970-01-01 00:00:00)以来的秒数
timestamp	时分秒，utc 时区
cluster	集群名称
status	<p>集群状态</p> <p>green:代表健康，主分片都正常并且至少有一个副本</p> <p>yellow:分配了所有主分片，至少缺少一个副本，集群数据仍旧完整</p> <p>red: 代表部分主分片不可用，可能已经丢失数据</p>

node.total	在线的节点总数量
node.data	在线的数据节点总数量
shards	存活的分片数量
pri	存活的主分片数量
relo	迁移中的分片数量
init	初始化中的分片数量
unassign	未分配的分片数量
pending_tasks	准备中的任务，例如迁移分片等
max_task_wait_time	任务最长等待时间
active_shards_percent	存活的分片百分比

4.2.2 查询各个节点状态

GET /_cat/nodes?v	
ip	节点 ip
heap.percent	堆内存占用百分比
ram.percent	内存占用百分比
cpu	cpu 占用百分比
load_1m	1 分钟的系统负载
load_5m	5 分钟的系统负载
node.role	节点的角色
master	是否是 master 节点
name	节点名称

4.2.3 查询各个索引状态

GET _cat/indices?v	
health	索引的健康状态
status	索引的开启状态
index	索引名

uuid	索引的 uuid
pri	索引主分片数量
rep	索引副本分片数量
docs.count	索引中文档总数
docs.deleted	索引中删除状态的文档数
store.size	索引主分片与副本分片总占用存储空间
pri.store.size	索引主分片占用空间

4.2.4 查询某个索引的分片情况

GET /_cat/shards/xxxx	
index	索引名称
shard	shard 分片序号
prirep	主分片 or 副本分片, p 表示主分片, r 表示副本分片
state	分片状态
docs	该分片存放的文档数
store	该分片占用的存储空间
ip	该分片所在的服务器 ip
node	该分片所在的节点名称

4.2.5 显示各个节点分片情况

GET _cat/allocation?v	
shards	节点承载的分片数量
disk.indices	索引占用的空间大小
disk.used	节点所在机器已使用的磁盘空间大小
disk.avail	节点可用空间大小
disk.total	节点总空间大小
disk.percent	节点磁盘占用百分比
host	节点 host

ip	节点 ip
node	节点名称

4.2.6 显示索引文档总数

GET _cat/count?v	
epoch	自标准时间(1970-01-01 00:00:00) 以来的秒数
timestamp	时分秒, utc 时区
count	文档总数

4.2.7 参数说明

?参数	
v	显示表头
help	显示命令返回的参数说明
h	指定要显示的列, GET _cat/count?h=epoch
format	设置要返回的内容格式 json,yml,txt 等, GET _cat/count?format=json
s	指定排序, GET _cat/indices?v&s=docs.count:desc
&	拼接多个参数, GET _cat/count?v&h=epoch

4.3 对数据的操作

4.3.1 es 中保存的数据结构

1) 有如下对象关系, 如果使用关系型数据库保存, 会被拆分成多张表

```
public class Movie {
    String id;
    String name;
    Double doubanScore;
    List<Actor> actorList;
}

public class Actor{
    String id;
    String name;
}
```

2) 在 elasticsearch 是用一个 json 来表示一个 document, 所以他保存到 es 中如下

```
{
```

18

更多 Java -大数据 -前端 -python 人工智能资料下载, 可百度访问: 尚硅谷官网

```
"id": "1",
"name": "operation red sea",
"doubanScore": "8.5",
"actorList": [
  {"id": "1", "name": "zhangyi"},
  {"id": "2", "name": "haiqing"},
  {"id": "3", "name": "zhanghanyu"}
]
```

4.3.2 创建索引

- 1) 创建索引不指定字段，按照第一条数据自动推断

```
PUT /movie_index
```

4.3.3 查看索引结构

```
GET movie_index/_mapping
```

```
{
  "movie_index" : {
    "mappings" : {
      "properties" : {
        "actorList" : {
          "properties" : {
            "id" : {
              "type" : "long"
            },
            "name" : {
              "type" : "text", //text 文本类型, 会做倒排索引
              "fields" : {
                "keyword" : { //子字段
                  "type" : "keyword", // 标准列式存储, 用于过滤 分组 聚合
                  "ignore_above" : 256
                }
              }
            },
            "doubanScore" : {
              "type" : "float"
            },
            "id" : {
              "type" : "long"
            },
            "name" : {
              "type" : "text", 倒排索引
              "fields" : {
                "keyword" : {
                  "type" : "keyword", 列式存储
                  "ignore_above" : 256
                }
              }
            }
          }
        }
      }
    }
  }
}
```

```
    }  
  }  
}  
}
```

4.3.4 删除索引

```
DELETE /movie_index
```

4.3.5 新增文档(幂等)

如果 index 不存在， 在新增文档时会自动创建 index。

```
PUT /movie_index/_doc/1  
{  
  "id":1,  
  "name":"operation red sea",  
  "doubanScore":8.5,  
  "actorList":[  
    {"id":1,"name":"zhang yi"},  
    {"id":2,"name":"hai qing"},  
    {"id":3,"name":"zhang han yu"}  
  ]  
}  
  
PUT /movie_index/_doc/2  
{  
  "id":2,  
  "name":"operation meigong river",  
  "doubanScore":8.0,  
  "actorList":[  
    {"id":3,"name":"zhang han yu"}  
  ]  
}  
  
PUT /movie_index/_doc/3  
{  
  "id":3,  
  "name":"incident red sea",  
  "doubanScore":5.0,  
  "actorList":[  
    {"id":4,"name":"atguigu"}  
  ]  
}
```

4.3.6 新增文档 (非幂等)

重复执行会新增重复数据，_id 会随机生成。

```
POST /movie_index/_doc  
{  
  "id":3,  
  "name":"incident red sea",  
  "doubanScore":5.0,  
  "actorList":[
```

```
{ "id": 4, "name": "atguigu" }
]
```

4.3.7 修改

1) 整体替换，和新增没有区别，最终只保留指定的字段

```
PUT /movie_index/_doc/3
{
  "id": "3",
  "name": "incident red sea",
  "doubanScore": 5.0,
  "actorList": [
    { "id": "1", "name": "shangguigu" }
  ]
}
```

2) 仅修改指定字段，其他字段保留原样

```
POST movie_index/_update/3
{
  "doc": {
    "doubanScore": 7.0
  }
}
```

4.3.8 删除一个 document

```
DELETE movie_index/_doc/3
```

4.3.9 查询 index 所有数据

```
GET movie_index/_search

{
  "took" : 3, // 耗费时间（毫秒）
  "timed_out" : false, // 是否超时
  "_shards" : {
    "total" : 1, // 发送给全部 1 个分片
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 12, // 命中多少条数据
      "relation" : "eq"
    },
    "max_score" : 1.0, // 最大评分
    "hits" : [ // 结果
      {
        "_index" : "movie_index",
        "_type" : "_doc",
        "_id" : "1",
        "_score" : 1.0,
```

```
"_source" : {
  "id" : 1,
  "name" : "operation red sea",
  "doubanScore" : 8.5,
  "actorList" : [
    {
      "id" : 1,
      "name" : "zhang yi"
    },
    {
      "id" : 2,
      "name" : "hai qing"
    },
    {
      "id" : 3,
      "name" : "zhang han yu"
    }
  ]
},
.....
```

4.3.10 查询一个 document

```
GET movie_index/_doc/3
```

4.3.11 按条件查询(全部)

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

4.3.12 按分词查询

1) 搜索 “operation red sea”

```
GET movie_index/_search
{
  "query": {
    "match": {
      "name": "operation red sea"
    }
  }
}
```

```
=====分析=====
----- operation red sea 分词-----
operation
red
sea
```

```
-----3 条示例数据倒排索引-----
operation 1 2
red 1 3
sea 1 3
meigong 3
river 2
incident 3

-----条件查询命中分词结果-----
operation 1 2
red 1 3
sea 1 3

1 号 doc 命中 3 次
3 号 doc 命中 2 次
2 号 doc 命中 1 次

-----结果展示-----
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 3,
      "relation" : "eq"
    },
    "max_score" : 1.4100108,
    "hits" : [
      {
        "_index" : "movie_index",
        "_type" : "_doc",
        "_id" : "1",
        "_score" : 1.4100108,
        "_source" : {
          "id" : 1,
          "name" : "operation red sea",
          "doubanScore" : 8.5,
          "actorList" : [
            {
              "id" : 1,
              "name" : "zhang yi"
            },
            {
              "id" : 2,
              "name" : "hai qing"
            }
          ]
        }
      }
    ]
  }
}
```

```

        {
            "id" : 3,
            "name" : "zhang han yu"
        }
    ]
},
{
    "_index" : "movie_index",
    "_type" : "_doc",
    "_id" : "3",
    "_score" : 0.9400072,
    "_source" : {
        "id" : 3,
        "name" : "incident red sea",
        "doubanScore" : 5.0,
        "actorList" : [
            {
                "id" : 4,
                "name" : "atguigu"
            }
        ]
    }
},
{
    "_index" : "movie_index",
    "_type" : "_doc",
    "_id" : "2",
    "_score" : 0.4700036,
    "_source" : {
        "id" : 2,
        "name" : "operation meigong river",
        "doubanScore" : 8.0,
        "actorList" : [
            {
                "id" : 3,
                "name" : "zhang han yu"
            }
        ]
    }
}
]
}
}

```

2) 评分

评分公式（不同版本有差异）：

$$\text{score}(q,d) = \text{coord}(q,d) \cdot \text{queryNorm}(q) \cdot \sum_{t \in q} (\text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot \text{t.getBoost}() \cdot \text{norm}(t,d))$$

根据匹配的程度，会影响数据的相关度评分，而相关度评分会影响默认排名。

正向因素：

命中次数、命中长度比例。

负面因素：

关键词在该字段的其他词条中出现的次数。

4.3.13 按分词子属性查询

要根据具体的字段，判断使用倒排还是列式存储

```
GET movie_index/_search
{
  "query": {
    "match": {
      "actorList.name": "zhang yi"
    }
  }
}
```

"actorList.name.keyword": "zhang yi" 能直接取到 zhang yi 这条数据

4.3.14 按 match_phrase（短语）查询

按短语查询，不再利用分词技术，直接用短语在原始数据中匹配。

```
GET movie_index/_search
{
  "query": {
    "match_phrase": {
      "name": "operation red"
    }
  }
}
```

4.3.15 过滤 - 值等判断

es 在存储字符串时，都会保留两种方式存储：

一种是倒排索引方式(text 类型)，用于分词匹配。

一种是标准列式存储(keyword 类型)，用于过滤，分组，聚合，排序....，需要加 keyword。

1) 条件过滤 name="operation red sea" （值等判断）

```
GET movie_index/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "term": {
            "name.keyword": "operation red sea"
          }
        }
      ]
    }
  }
}
```

```

    }
  }
}

```

2) 分词匹配"red sea", 条件过滤 actorList.name="zhang han yu"

```

GET movie_index/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "name": "red sea"
          }
        }
      ],
      "filter": [
        {
          "term": {
            "actorList.name.keyword": "zhang han yu"
          }
        }
      ]
    }
  }
}

```

filter 必须满足
 must 在 filter 的基础上, 必须满足, 满足后, 才会进到结果集
 should 在 filter 的基础上, 不是必须满足, 若能满足就会打分, 不满足, 就不会打分, 但都会出现在结果集上

4.3.16 过滤 - 范围过滤

```

GET movie_index/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "range": { //范围过滤
            "doubanScore": { // 查询 doubanScore 大于等于 7 小于等于 8
              "gte": 7,
              "lte": 8
            }
          }
        }
      ]
    }
  }
}

```

4.3.17 过滤 - 符合查询

```

GET movie_index/_search
{
  "query": {
    "bool": {
      "filter": [

```

```

        {"term": {
          "actorList.id": "3"
        }
      }
    ],
    "should": [
      {
        "match": {
          "name": "red"
        }
      }
    ]
  }
}
}

```

must 和 should 的区别： must 是必须有，should 在有其他条件命中的情况下，should 命中给分并显示，如果 should 未命中则只显示不给分。

4.3.18 过滤 - 修改

```

POST movie_index/_update_by_query
{
  "query": {
    "term": {
      "actorList.name": {
        "value": "atguigu"
      }
    }
  },
  "script": {
    "source": "ctx._source['actorList'][0]['name']=params.newName",
    "params": {
      "newName": "shangguigu"
    },
    "lang": "painless"
  }
}

```

4.3.19 过滤 - 删除

```

POST movie_index/_delete_by_query
{
  "query": {
    "term": {
      "actorList.name": "atguigu"
    }
  }
}

```

4.3.20 排序

```
GET movie_index/_search
```

27

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
{
  "query": {
    "bool": {
      "filter": [
        {
          "range": {
            "doubanScore": {
              "gte": 7,
              "lte": 10
            }
          }
        }
      ]
    }
  },
  "sort": [
    {
      "doubanScore": {
        "order": "desc"
      }
    }
  ]
}
```

4.3.21 分页查询

```
GET movie_index/_search
{
  "query": {
    "match_all": {}
  },
  "from": 2 // 从哪行开始, 需要通过页码进行计算 (pageNum - 1) * size
  "size": 2
}
```

4.3.22 高亮

用于在网页中通过特殊标签来高亮显示结果中的关键字。

```
GET movie_index/_search
{
  "query": {
    "match": {
      "name": "red"
    }
  },
  "highlight": {
    "fields": {
      "name": {} //默认标签
    }
  }
}
```

```
=====结果=====
"highlight" : {
  "name" : [
    "operation <em>red</em> sea"
  ]
}
```

也可以自定义高亮标签（一般由前端来负责显示样式）。

```
GET movie_index/_search
{
  "query": {
    "match": {
      "name": "red"
    }
  },
  "highlight": {
    "fields": {
      "name": {
        "pre_tags": "<font color='red'>",
        "post_tags": "</font>"
      }
    }
  }
}

=====结果=====
"highlight" : {
  "name" : [
    "operation <font color='red'>red</font> sea"
  ]
}
```

4.3.23 聚合

1) 取出每个演员共参演了多少部电影

```
GET movie_index/_search
{
  "aggs": {
    "groupby_actor": { 自己取的名字
      "terms": {
        "field": "actorList.name.keyword", 列式存储字段
        "size": 10000
      }
    }
  }
}

=====默认会送 count 结果=====
"aggregations" : {
  "groupby_actor" : {
    "doc_count_error_upper_bound" : 0,
    "sum_other_doc_count" : 0,
```

```
"buckets" : [
  {
    "key" : "zhang han yu",
    "doc_count" : 2
  },
  {
    "key" : "atguigu",
    "doc_count" : 1
  },
  {
    "key" : "hai qing",
    "doc_count" : 1
  },
  {
    "key" : "zhang yi",
    "doc_count" : 1
  }
]
}
```

2) 每个演员参演电影的平均分是多少，并按平均分排序

```
GET movie_index/_search
{
  "aggs": {
    "groupby_actor": {
      "terms": {
        "field": "actorList.name.keyword",
        "size": 100000
      },
      "order": {
        "avg_score": "desc"
      }
    },
    "avg_score": {
      "avg": {
        "field": "doubanScore"
      }
    }
  },
  "size": 0
}
```

=====结果=====

```
"aggregations" : {
  "groupby_actor" : {
    "doc_count_error_upper_bound" : 0,
    "sum_other_doc_count" : 0,
    "buckets" : [
      {
```

```

    "key" : "hai qing",
    "doc_count" : 1,
    "avg_score" : {
      "value" : 8.5
    }
  },
  {
    "key" : "zhang yi",
    "doc_count" : 1,
    "avg_score" : {
      "value" : 8.5
    }
  },
  {
    "key" : "zhang han yu",
    "doc_count" : 2,
    "avg_score" : {
      "value" : 8.25
    }
  },
  {
    "key" : "atguigu",
    "doc_count" : 1,
    "avg_score" : {
      "value" : 5.0
    }
  }
]
}
}

```

4.4 SQL 的使用

1) 说明

ElasticSearch SQL 是 6.3 版本以后的功能，能够支持一些最基本的 SQL 查询语句。

<https://www.elastic.co/guide/en/elasticsearch/reference/6.6/sql-functions.html>

目前的一些情况：

只支持 select 操作，insert，update，delete 一律不支持；6.3 以前的版本无法支持；

SQL 比 DSL 有丰富的函数；不支持窗口函数；SQL 少一些特殊功能，比如高亮。

2) 每个演员参演电影的平均分是多少，并按评分排序

```

GET _sql?format=txt
{
  "query":
  """
  SELECT
    actorList.name.keyword, avg (doubanScore) ads
  FROM
    movie_index

```

```
where
  match(name, 'red') 单引号
group by
  actorList.name.keyword
order by
  ads
desc
limit
  10
  """
}
```

4.5 中文分词

4.5.1 测试默认分词

1) 创建 index, put 数据

```
PUT /movie_index_cn/_doc/1
{ "id":1,
  "name":"红海行动",
  "doubanScore":8.5,
  "actorList":[
    {"id":1,"name":"张译"},
    {"id":2,"name":"海清"},
    {"id":3,"name":"张涵予"}
  ]
}

PUT /movie_index_cn/_doc/2
{
  "id":2,
  "name":"湄公河行动",
  "doubanScore":8.0,
  "actorList":[
    {"id":3,"name":"张涵予"}
  ]
}

PUT /movie_index_cn/_doc/3
{
  "id":3,
  "name":"红海事件",
  "doubanScore":5.0,
  "actorList":[
    {"id":4,"name":"尚硅谷"}
  ]
}
```

2) 执行查询 1

```
GET movie_index_cn/_search
{
  "query": {
    "match": {
```



```
    "name": "红海战役"
  }
}
```

如上查询可以查询到数据,但是正确吗?

3) 执行查询 2

```
GET movie_index_cn/_search
{
  "query": {
    "match": {
      "name": "渤海银行"
    }
  }
}
```

如上查询可以查询到数据,但是正确吗?

4) 查看默认中文分词效果

```
GET movie_index/_analyze
{
  "text": "红海行动"
}

=====结果=====
{
  "tokens" : [
    {
      "token" : "红",
      "start_offset" : 0,
      "end_offset" : 1,
      "type" : "<IDEOGRAPHIC>",
      "position" : 0
    },
    {
      "token" : "海",
      "start_offset" : 1,
      "end_offset" : 2,
      "type" : "<IDEOGRAPHIC>",
      "position" : 1
    },
    {
      "token" : "行",
      "start_offset" : 2,
      "end_offset" : 3,
      "type" : "<IDEOGRAPHIC>",
      "position" : 2
    },
    {
      "token" : "动",
      "start_offset" : 3,
      "end_offset" : 4,

```

```
    "type" : "<IDEOGRAPHIC>",
    "position" : 3
  }
]
```

elasticsearch 本身自带的中文分词，就是单纯把中文一个字一个字的分开，根本没有词汇的概念。但是实际应用中，用户都是以词汇为条件，进行查询匹配的，如果能够把文章以词汇为单位切分开，那么与用户的查询条件能够更贴切的匹配上，查询速度也更加快速。

分词器下载网址：<https://github.com/medcl/elasticsearch-analysis-ik>

4.5.2 安装 ik 分词

1) 上传分词包到/opt/software 目录下

```
[atguigu@hadoop102 software]$ ll
-rw-r--r--. 1 atguigu atguigu 4504403 4月 12 2021 elasticsearch-
analysis-ik-7.8.0.zip
```

2) 解压到 es 安装目录的 plugins/ik 目录下，路径一定要正确

```
[atguigu@hadoop102 plugins]$ pwd
/opt/module/es7/plugins
[atguigu@hadoop102 plugins]$ mkdir ik
[atguigu@hadoop102 software]$ pwd
/opt/software
[atguigu@hadoop102 software]$ unzip elasticsearch-analysis-ik-
7.8.0.zip -d /opt/module/es7/plugins/ik
```

3) 分发到每个 es 节点

```
[atguigu@hadoop102 plugins]$ xsync.sh ik
```

4) 重启 es

```
[atguigu@hadoop102 plugins]$ es.sh stop
[atguigu@hadoop102 plugins]$ es.sh start
```

4.5.3 测试 ik 分词

1) 使用 ik_smart 分词器

```
GET movie_index/_analyze
{
  "text": "我是中国人"
  ,
  "analyzer": "ik_smart"
}
```

=====结果=====

```
{
  "tokens" : [
    {
      "token" : "我",
      "start_offset" : 0,
```

```
    "end_offset" : 1,
    "type" : "CN_CHAR",
    "position" : 0
  },
  {
    "token" : "是",
    "start_offset" : 1,
    "end_offset" : 2,
    "type" : "CN_CHAR",
    "position" : 1
  },
  {
    "token" : "中国人",
    "start_offset" : 2,
    "end_offset" : 5,
    "type" : "CN_WORD",
    "position" : 2
  }
]
```

2) 使用 ik_max_word 分词器

```
GET movie_index/_analyze
{
  "text": "我是中国人"
  ,
  "analyzer": "ik_max_word"
}
```

=====结果=====

```
{
  "tokens" : [
    {
      "token" : "我",
      "start_offset" : 0,
      "end_offset" : 1,
      "type" : "CN_CHAR",
      "position" : 0
    },
    {
      "token" : "是",
      "start_offset" : 1,
      "end_offset" : 2,
      "type" : "CN_CHAR",
      "position" : 1
    },
    {
      "token" : "中国人",
      "start_offset" : 2,
      "end_offset" : 5,
      "type" : "CN_WORD",
      "position" : 2
    }
  ],
}
```

```
{
  "token" : "中国",
  "start_offset" : 2,
  "end_offset" : 4,
  "type" : "CN_WORD",
  "position" : 3
},
{
  "token" : "国人",
  "start_offset" : 3,
  "end_offset" : 5,
  "type" : "CN_WORD",
  "position" : 4
}
]
```

4.6 关于 mapping

4.6.1 查看 mapping

GET movie_index/_mapping

=====结果=====

```
{
  "movie_index" : {
    "mappings" : {
      "properties" : {
        "actorList" : {
          "properties" : {
            "id" : {
              "type" : "long"
            },
            "name" : {
              "type" : "text",
              "fields" : {
                "keyword" : {
                  "type" : "keyword",
                  "ignore_above" : 256
                }
              }
            }
          }
        },
        "doubanScore" : {
          "type" : "float"
        },
        "id" : {
          "type" : "long"
        },
        "name" : {
          "type" : "text",
          "fields" : {
```

```

        "keyword" : {
          "type" : "keyword",
          "ignore_above" : 256
        }
      }
    }
  }
}
}
}

```

实际上每个 type 中的字段是什么数据类型，由 mapping 定义。

但是如果没有设定 mapping 系统会自动，根据一条数据来推断出应该的数据格式：

- ① true/false → boolean
- ② 1020 → long
- ③ 20.1 → float
- ④ “2018-02-01” → date
- ⑤ “hello world” → text + keyword

默认只有 text 会进行分词，keyword 是不会分词的字符串。

mapping 除了自动定义，还可以手动定义，但是只能对新加的、没有数据的字段进行定义。一旦有了数据就无法再做修改了。

注意：虽然每个 Field 的数据放在不同的 type 下,但是同一个名字的 Field 在一个 index 下只能有一种 mapping 定义。

4.6.2 基于中文分词搭建索引

1) 创建 index，手动指定 type

```

PUT movie_index_cn
{
  "settings": {
    "number_of_shards": 1 //分片数
  },
  "mappings": {
    "properties": {
      "id": {
        "type": "long"
      },
      "name": {
        "type": "text"
        , "analyzer": "ik_smart" //分词器
      },
      "doubanScore": {
        "type": "double"
      }
    }
  }
}

```

```
    },
    "actorList": {
      "properties": {
        "id": {
          "type": "long"
        },
        "name": {
          "type": "keyword"
        }
      }
    }
  }
}
```

2) 插入数据

```
PUT /movie_index_cn/_doc/1
{ "id":1,
  "name":"红海行动",
  "doubanScore":8.5,
  "actorList":[
    {"id":1,"name":"张译"},
    {"id":2,"name":"海清"},
    {"id":3,"name":"张涵予"}
  ]
}
PUT /movie_index_cn/_doc/2
{
  "id":2,
  "name":"湄公河行动",
  "doubanScore":8.0,
  "actorList":[
    {"id":3,"name":"张涵予"}
  ]
}
PUT /movie_index_cn/_doc/3
{
  "id":3,
  "name":"红海事件",
  "doubanScore":5.0,
  "actorList":[
    {"id":4,"name":"尚硅谷"}
  ]
}
```

3) 执行查询 1

```
GET /movie_index_cn/_search
{
  "query": {
    "match": {
      "name": "红海战役"
    }
  }
}
```

```
}
}
```

可以正常查询到数据。

4) 执行查询 2

```
GET /movie_index_cn/_search
{
  "query": {
    "match": {
      "name": "渤海银行"
    }
  }
}
```

查询不到数据。

4.7 分割索引

4.7.1 介绍

ES 不允许对索引结构进行修改,如果业务发生变化,字段类型需要进行修改,ES 如何应对呢?

分割索引是企业中常用的一种应对策略.实际就是根据时间间隔把一个业务索引切分成多个索引。

例如将某个业务存储数据使用到的索引,设计成以小时、天、周等分割后的多个索引。这样,每次分割都可以应对一次字段的变更。

原索引	分割索引
order_info	order_info_20210101
	order_info_20210102
	order_info_20210103

4.7.2 好处

1) 查询范围优化

因为一般情况并不会查询全部时间周期的数据,那么通过切分索引,物理上减少了扫描数据的范围,也是对性能的优化。类似于大家学过的 Hive 中的分区表。

2) 结构变化的灵活性

因为 elasticsearch 不允许对数据结构进行修改。但是实际使用中索引的结构和配置难免

变化，那么只要对下一个间隔的索引进行修改，原来的索引位置原状。这样就有了一定的灵活性。

4.8 索引别名

4.8.1 介绍

索引别名就像一个快捷方式或软连接，可以指向一个或多个索引，也可以给任何一个需要索引名的 API 来使用。别名带给我们极大的灵活性，允许我们做下面这些：

1) 给多个索引分组

分割索引可以解决数据结构变更的场景，但是分割的频繁，如果想要统计一个大周期的数据(例如季度、年)，数据是分散到不同的索引中的，统计比较麻烦。我们可以将分割的索引取相同的别名，这样，我们在统计时直接指定别名即可。

2) 给索引的一个子集创建视图

将一个索引中的部分数据(基于某个条件)创建别名，查询此部分数据时，可以直接使用别名

对频繁使用的数据

3) 在运行的集群中可以无缝的从一个索引切换到另一个索引

如果涉及到索引的切换，我们可以在程序中指定别名，而不是索引。当需要切换索引时，我们可以直接从一个索引上减去别名，在另一个索引上加上别名（减与加为原子操作）实现无缝切换。

4.8.2 创建索引别名

1) 建表时直接声明

```
PUT movie_index_1
{
  "aliases": {
    "movie_index_1_20210101": {}
  },
  "mappings": {
    "properties": {
      "id": {
        "type": "long"
      },
      "name": {
        "type": "text",
        "analyzer": "ik_smart"
      },
      "doubanScore": {
        "type": "double"
      }
    }
  }
}
```



```

    },
    "actorList": {
      "properties": {
        "id": {
          "type": "long"
        },
        "name": {
          "type": "keyword"
        }
      }
    }
  }
}
}
}

```

2) 为已存在的索引增加别名

```

POST _aliases
{
  "actions": [
    { "add": { "index": "movie_index", "alias": "movie_index_2021" } }
  ]
}

POST _aliases
{
  "actions": [
    { "add": { "index": "movie_index_cn", "alias": "movie_index_2021" } }
  ]
}

```

3) 通过加过滤条件缩小查询范围，建立一个子集视图

```

POST _aliases
{
  "actions": [
    { "add": {
      "index": "movie_index",
      "alias": "movie_index_query_zhy",
      "filter": {
        "term": {
          "actorList.name.keyword": "zhang han yu"
        }
      }
    }
  ]
}

```

4) 查询别名，与使用普通索引没有区别

```
GET movie_index_2021/_search
```

5) 删除某个索引的别名

```
POST _aliases
{
  "actions": [
    { "remove": {
      "index": "movie_index",
      "alias": "movie_index_2021" }}
  ]
}
```

6) 为某个别名进行无缝切换

```
POST /_aliases
{
  "actions": [
    { "remove": {
      "index": "movie_index",
      "alias": "movie_index_2021" }},
    { "add": {
      "index": "movie_index_cn",
      "alias": "movie_index_2021" }}
  ]
}
```

7) 查询别名列表

```
GET _cat/aliases?v
```

4.9 索引模板

Index Template 索引模板，顾名思义，就是创建索引的模具，其中可以定义一系列规则来帮助我们构建符合特定业务需求的索引的 mappings 和 settings，通过使用 Index Template 可以让我们的索引具备可预知的一致性。

4.9.1 创建模板

```
PUT _template/template_movie2021
{
  "index_patterns": ["movie_test*"],
  "settings": {
    "number_of_shards": 1
  },
  "aliases" : {
    "{index}-query": {},
    "movie_test-query": {} 基于此模板的统一别名
  },
  "mappings": {
    "properties": {
      "id": {
        "type": "keyword"
      },
      "movie_name": {
        "type": "text",
        "analyzer": "ik_smart"
      }
    }
  }
}
```

```

    }
  }
}
}

```

其中 "index_patterns": ["movie_test*"], 的含义就是凡是往 movie_test 开头的索引写入数据时, 如果索引不存在, 那么 es 会根据此模板自动建立索引。

在 "aliases" 中用 {index} 表示, 获得真正的创建的索引名。

4.9.2 测试模板

1) 创建索引

```

POST movie_test_20210101/_doc
{
  "id": "111",
  "movie_name": "zhang111"
}

POST movie_test_20210102/_doc
{
  "id": "222",
  "movie_name": "zhang222"
}

```

2) 查看索引 mapping

```

GET movie_test_20210101/_mapping

{
  "movie_test_20210101" : {
    "mappings" : {
      "properties" : {
        "id" : {
          "type" : "keyword"
        },
        "movie_name" : {
          "type" : "text",
          "analyzer" : "ik_smart"
        }
      }
    }
  }
}

GET movie_test_20210102/_mapping

{
  "movie_test_20210102" : {
    "mappings" : {
      "properties" : {
        "id" : {
          "type" : "keyword"

```

```

    },
    "movie_name" : {
      "type" : "text",
      "analyzer" : "ik_smart"
    }
  }
}
}
}
}

```

4.9.3 查看系统中已有的模板清单

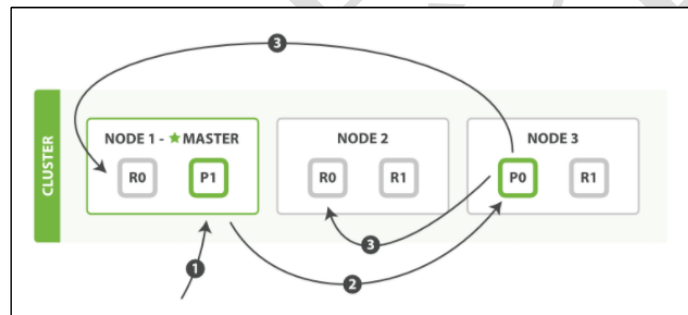
```
GET _cat/templates?v
```

4.9.4 查看某个模板详情

```
GET _template/template_movie2021
```

第 5 章 分布式读写原理

5.1 写流程（基于_id）



- 写操作必须在主分片上面完成之后才能被复制到相关的副本分片
- 客户端向 Node 1 发送写操作请求，此时 Node1 为协调节点（接收客户端请求的节点）。

(3) Node1 节点使用文档的_id 通过公式计算确定文档属于分片 0 。请求会被转发到 Node 3，因为分片 0 的主分片目前被分配在 Node 3 上。分片计算公式：

```
shard = hash(routing) % number_of_primary_shards
```

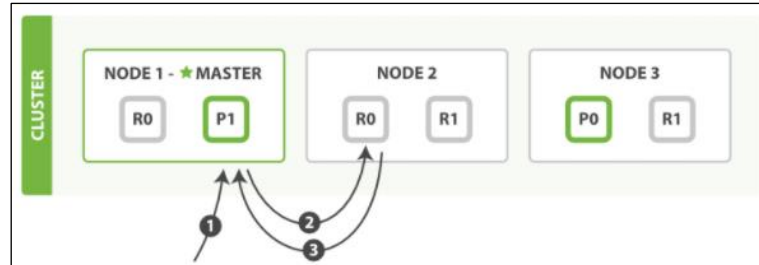
routing 是一个可变值，默认是文档的_id，也可以设置成一个自定义的值。routing 通过 hash 函数生成一个数字，然后这个数字再除以 number_of_primary_shards（主分片的数量）后得到余数。这个分布在 0 到 number_of_primary_shards-1 之间的余数，就是我们所寻求的文档所在分片的位置。

- Node 3 在主分片上面执行请求。如果成功了，它将请求并行转发到 Node 1 和 Node 2 的副本分片上。一旦所有的副本分片都报告成功，Node 3 将向协调节点报告成功，协调节点向客户端报告成功。

(5) 那么如果 shard 的数量变化，是不是数据就要重新 rehash 呢？

不会，因为一个 index 的 shards 数量是不能改变的。

5.2 读流程（基于_id）



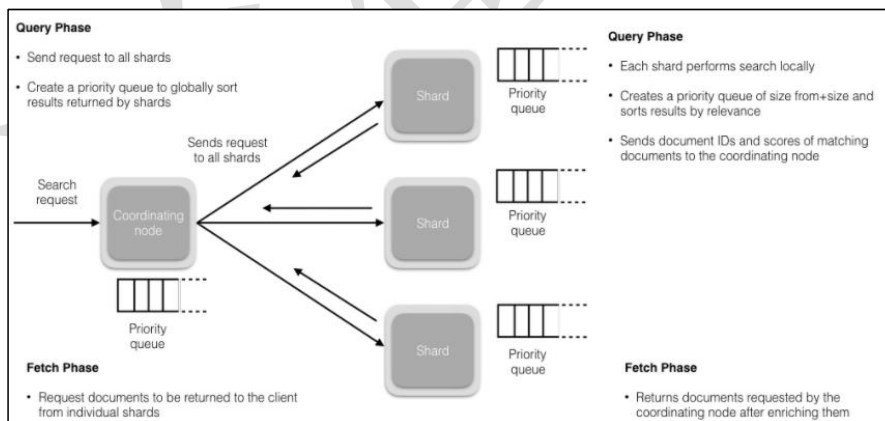
(1) 读操作可以从主分片或者从其它任意副本分片检索文档

(2) 客户端向 Node 1 发送读请求，Node1 为协调节点

(3) 节点使用文档的 `_id` 来确定文档属于分片 0。分片 0 的主副分片存在于所有的三个节点上。协调节点在每次请求的时候都会通过轮询的方式将请求打到不同的节点上来达到负载均衡，假设本次它将请求转发到 Node 2。

(4) Node 2 将文档返回给 Node 1，Node1 然后将文档返回给客户端。

5.3 搜索流程（_search）



(1) 搜索被执行成一个两阶段过程，我们称之为 Query Then Fetch

(2) 在初始查询阶段时，查询会广播到索引中每一个分片（主分片或者副本分片）。每个分片在本地执行搜索并构建一个匹配文档的大小为 `from + size` 的优先队列。

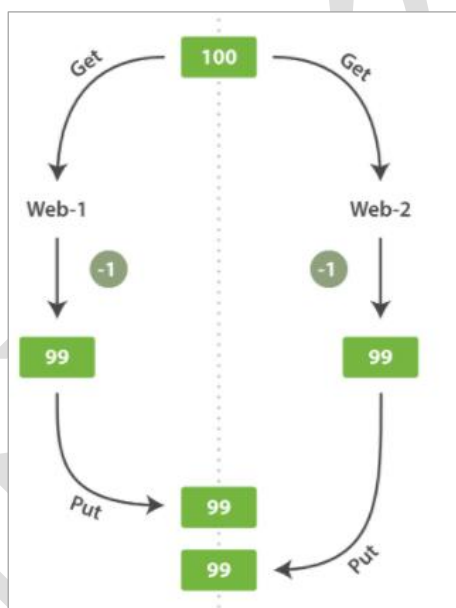
PS: 在搜索的时候是会查询 Filesystem Cache 的，但是有部分数据还在 Memory Buffer，所以搜索是近实时的。

(3) 每个分片返回各自优先队列中 所有文档的 ID 和排序值 给协调节点，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。

(4) 接下来就是取回阶段，协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。每个分片加载并丰富文档，接着返回文档给协调节点。一旦所有的文档都被取回了，协调节点返回结果给客户端。

5.4 文档的修改和并发控制

ElasticSearch 中的全部文档数据都是不可变的，数据不能修改，只能通过版本号的方式不断增加。这样做的主要目的是解决更新过程中的并发冲突问题。



1) 悲观并发控制

这种方法被关系型数据库广泛使用，它假定有变更冲突可能发生，因此阻塞访问资源以防止冲突。一个典型的例子是读取一行数据之前先将其锁住，确保只有获取锁的线程才能够对这行数据进行修改。

2) 乐观并发控制

Elasticsearch 中使用的这种方法假定冲突是不可能发生的，并且不会阻塞正在尝试的操作。然而，如果源数据在读写当中被修改，更新将会失败。应用程序接下来将决定该如何解决冲突。例如，可以重试更新、使用新的数据、或者将相关情况报告给用户。

5.5 删除方式

如果进行删除文档操作，也不会直接物理删除，而是通过给文档打删除标记，进行逻辑删除，至到该索引触发段合并时，才物理删除，释放存储空间。

5.6 shard 与段

由于索引一般是以天为单位进行建立，如果业务线很多，每个索引又不注意控制分片，日积月累下来一个集群几万到几十万个分片也是不难见到的。

5.6.1 shard 太多带来的危害

每个分片都有 Lucene 索引，这些索引都会消耗 cpu 和内存。同样的数据，分片越多，额外消耗的 cpu 和内存就越多，会出现“1+1”>2 的情况。

shard 的目的是为了负载均衡让每个节点的硬件充分发挥，但是如果分片多，在单个节点上的多个 shard 同时接受请求，并对本节点的资源形成了竞争，实际上反而造成了内耗。

5.6.2 如何规划 shard 数量

1) 根据每日数据量规划 shard 数量

以按天划分索引情况为例，单日数据评估低于 10G 的情况可以只使用一个分片，高于 10G 的情况，单一分片也不能太大不能超过 30G。所以一个 200G 的单日索引大致划分 7-10 个分片。

2) 根据堆内存规划 shard 数量

另外从堆内存角度，一个 Elasticsearch 服务官方推荐的最大堆内存是 32G。一个 10G 分片，大致会占用 30-80M 堆内存，那么对于这个 32G 堆内存的节点，最好不要超过 1000 个分片。

```
There is another reason to not allocate enormous heaps to Elasticsearch. As it turns out, the HotSpot JVM uses a trick to compress object pointers when heaps are less than around 32 GB. Once you cross that magical ~32 GB boundary, the pointers switch back to ordinary object pointers. The size of each pointer grows, more CPU-memory bandwidth is used, and you effectively lose memory. In fact, it takes until around 40-50 GB of allocated heap before you have the same effective memory of a heap just under 32 GB using compressed oops.
```

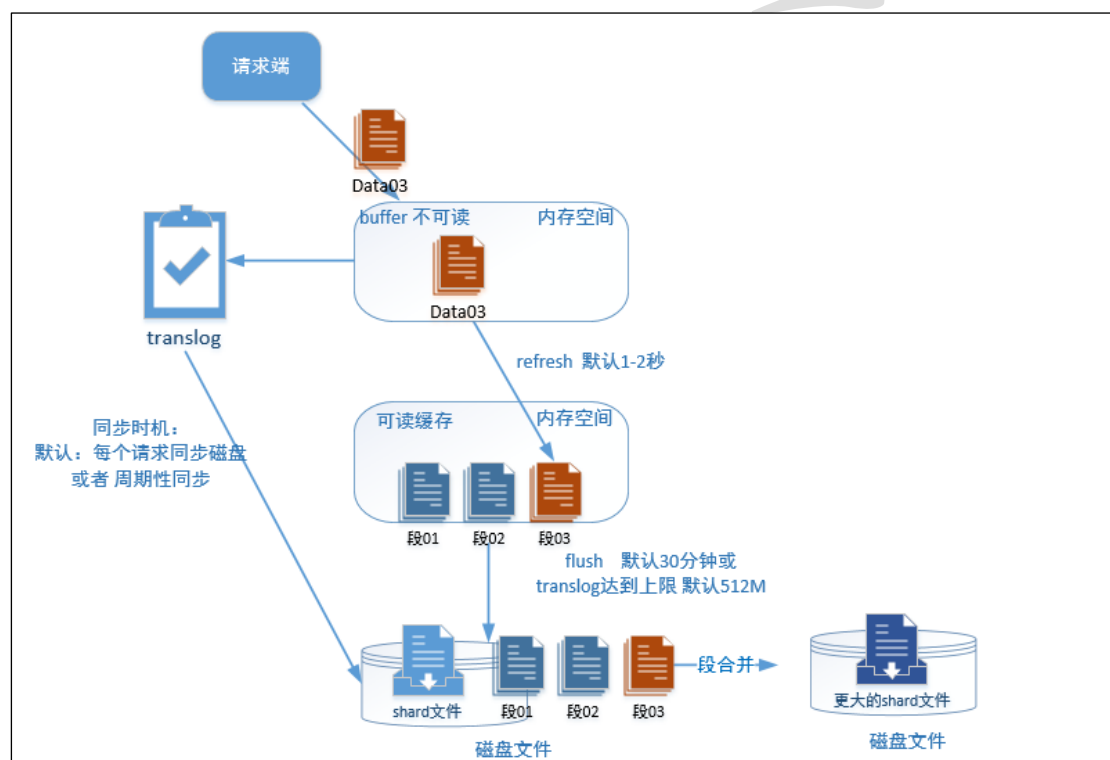
简意就是 JVM 在 32G 以下时会使用一种压缩技术，这种压缩技术对于内存使用性价比高，如果超过 32GB，大概可能要 50 个 G 的内存才能达到压缩方案的 32G 。

另外还有一个原因就是，除了 JVM 的堆内存，lucene 引擎会使用大量的堆外内存，用于比如：可读缓存等等。所以建议预留堆内存的一倍空间给操作系统。

5.6.3 shard 优化

- (1) 及时归档冷数据。
- (2) 降低单个分片占用的资源消耗，具体方法就是：合并分片中多个 segment（段）。

5.6.4 数据的物理提交流程



- (1) 首先由请求端提交到 buffer 中，此时数据不可写。同时写入 translog(内存中),然后 translog 根据默认设置同步磁盘文件。此时会返回给请求端处理成功。
- (2) 1-2 秒后执行 refresh 操作把 buffer 中的数据写入整理成为一个段，并提交到可读缓存。
- (3) 可读缓存会存在相当一段时间直到达到 flush 条件写入磁盘（flush 条件：默认 30 分钟或者 translog 达到默认上限 512M ）。
- (4) 落盘到磁盘文件后多个段以文件形式保存，后台周期性或手动进行段合并，把段中的数据合并到 shard 文件中。

5.6.5 Segment（段）优化

由于 es 的异步写入机制，后台每一次把写入到内存的数据 refresh（默认每秒）到磁盘，都会在对应的 shard 上产生一个 segment。

1) segment 太多的危害

这个 segment 上的数据有着独立的 Lucene 索引。日积月累，如果一个 shard 是由成千上万的 segment 组成，那么性能一定非常不好，而且消耗内存也大。

2) 官方优化

如果尽可能的把小的 segment 合并成为大 segment，这样既节省内存又提高查询性能。于是 Es 后台实际上会周期性的执行合并 segment 的任务。

但是由于 ES 担心这种合并操作会占用资源，影响搜索性能（事实上也确实很有影响尤其是在有写操作的时候）。所以有很多内置的限制和门槛设置的非常保守，导致很久不会触发合并，或者合并效果不理想。

3) 手动优化

生产环境中如果是 **以天为单位** 切割索引的话，我们其实是可以明确的知道只要某个索引过了当天，就几乎不会再有什么写操作了，这个时候其实是一个主动进行合并的好时机。

可以利用如下语句主动触发合并操作：

```
POST movie_index/_forcemerge?max_num_segments=1
```

查看索引的段情况

```
GET _cat/indices/?s=segmentsCount:desc&v&h=index,segmentsCount,segmentsMemory,memoryTotal,storeSize,p,r
```

5.7 关于 master

master 并不负责实际的数据处理，主要是负责对如下信息的维护工作。

集群状态；用户参数配置；数据的索引、别名、分析器的相关设置；分片所在节点位置等。

虽然这些信息每个节点都有，但是只有 master 节点可以发起变更，然后同步给其他节点。

第 6 章 Java 程序中的应用

6.1 ES 客户端

目前市面上有两类客户端。

一类是 `TransportClient` 为代表的 ES 原生客户端，不能执行原生 `dsl` 语句，必须使用它的 Java api 方法。

另外一种是以 `RestApi` 为主的 `missing client`，最典型的就是 `jest`。这种客户端可以直接使用 `dsl` 语句拼成的字符串，直接传给服务端，然后返回 `json` 字符串再解析。

两种方式各有优劣，但是最近 `elasticsearch` 官网，宣布计划在 7.0 以后的版本中废除 `TransportClient`，以 `RestClient` 为主。目前 `RestClient` 类型的 es 客户端有很多种，比如 `Jest`、`High level Rest Client`、`ElasticsearchRestTemplate` 等。

We plan on deprecating the `TransportClient` in Elasticsearch 7.0 and removing it completely in 8.0. Instead, you should be using the Java High Level REST Client, which executes HTTP requests rather than serialized Java requests. The migration guide describes all the steps needed to migrate.

The Java High Level REST Client currently has support for the more commonly used APIs, but there are a lot more that still need to be added. You can help us prioritise by telling us which missing APIs you need for your application by adding a comment to this issue: [Java high-level REST client completeness](#).

Any missing APIs can always be implemented today by using the low level Java REST Client with JSON request and response bodies.

6.2 ES 开发

6.2.1 准备环境

1) 创建工程

2) 在 `pom.xml` 中加入 es 所需依赖

```
<dependency>
  <groupId>org.elasticsearch</groupId>
  <artifactId>elasticsearch</artifactId>
  <version>7.8.0</version>
</dependency>
<!-- elasticsearch 的客户端 -->
<dependency>
  <groupId>org.elasticsearch.client</groupId>
  <artifactId>elasticsearch-rest-high-level-client</artifactId>
  <version>7.8.0</version>
```

50

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
</dependency>

<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
  <version>4.5.10</version>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.62</version>
</dependency>
```

6.2.2 写入数据

1) 在 ES 中创建 index

```
PUT movie_test_20210103
```

2) 单条数据写入

```
object EsTest {

  //声明 es 客户端
  var esClient : RestHighLevelClient = build()

  def main(args: Array[String]): Unit = {

    val movie = Movie("100", "梅艳芳")
    save(movie, "movie_test20210103")

    destory()
  }

  /**
   * 销毁
   */
  def destory(): Unit = {
    esClient.close()
    esClient = null
  }

  /**
   * 创建 es 客户端对象
   */
  def build(): RestHighLevelClient = {
    val builder: RestClientBuilder = RestClient.builder(new
    HttpHost("hadoop102", 9200))
    val esClient = new RestHighLevelClient(builder)
    esClient
  }
}
```

3) 单条数据幂等写入

```
def main(args: Array[String]): Unit = {
```

```
val source: (String, Movie) = ("101", Movie("101", "功夫"))
saveIdempotent(source, "movie_test20210103")
destory()
}

/**
 * 单条数据幂等写入
 * 通过指定 id 实现幂等
 */
def saveIdempotent(source: (String, AnyRef), indexName: String): Unit = {
    val indexRequest = new IndexRequest()
    indexRequest.index(indexName)
    val movieJsonStr: String = JSON.toJSONString(source._2, new
SerializeConfig(true))
    indexRequest.source(movieJsonStr, XContentType.JSON)
    indexRequest.id(source._1)
    esClient.index(indexRequest, RequestOptions.DEFAULT)
}
```

4) 批量数据写入

```
def main(args: Array[String]): Unit = {
    val movies = List(Movie("102", "速度与激情"), Movie("103", "八佰
") )
    bulkSave(movies, "movie_test20210103")
    destory()
}

/**
 * 批量数据写入
 */
def bulkSave(sourceList: List[AnyRef], indexName: String): Unit = {
    // BulkRequest 实际上就是由多个单条 IndexRequest 的组合
    val bulkRequest = new BulkRequest()

    for (source <- sourceList) {
        val indexRequest = new IndexRequest()
        indexRequest.index(indexName)
        val movieJsonStr: String = JSON.toJSONString(source, new
SerializeConfig(true))
        indexRequest.source(movieJsonStr, XContentType.JSON)
        bulkRequest.add(indexRequest)
    }

    esClient.bulk(bulkRequest, RequestOptions.DEFAULT)
}
```

5) 批量数据幂等写入

```
def main(args: Array[String]): Unit = {

    val movies = List(("104", Movie("104", "红海行动")), ("105",
Movie("105", "湄公河行动")))
}
```

```
bulkSaveIdempotent(movies,"movie_test20210103")

destory()
}

/**
 * 批量数据幂等写入
 * 通过指定 id 实现幂等
 */
def bulkSaveIdempotent(sourceList: List[(String, AnyRef)], indexName: String): Unit = {
  // BulkRequest 实际上就是由多个单条 IndexRequest 的组合
  val bulkRequest = new BulkRequest()

  for ((docId, sourceObj) <- sourceList) {
    val indexRequest = new IndexRequest()
    indexRequest.index(indexName)
    val movieJsonStr: String = JSON.toJSONString(sourceObj, new
SerializeConfig(true))
    indexRequest.source(movieJsonStr, XContentType.JSON)
    indexRequest.id(docId)
    bulkRequest.add(indexRequest)
  }

  esClient.bulk(bulkRequest, RequestOptions.DEFAULT)
}
```

6.2.3 修改数据

1) 根据 docid 更新字段

```
def main(args: Array[String]): Unit = {

  val source= ("101", "movie_name", "功夫功夫功夫")
  update(source,"movie_test20210103")

}

/**
 * 根据 docid 更新字段
 */
def update( source :( String, String , String ) ,
indexName:String): Unit = {
  val updateRequest = new UpdateRequest()
  updateRequest.index(indexName)
  val sourceJson: String = JSON.toJSONString(source._2,new
SerializeConfig(true))
  updateRequest.id(source._1)
  updateRequest.doc(XContentType.JSON,source._2, source._3)
  esClient.update(updateRequest,RequestOptions.DEFAULT)
}
```

2) 根据查询条件修改

```
def main(args: Array[String]): Unit = {

    updateByQuery("movie_name", " 八佰 " , " 八佰 good" ,
"movie_test20210103")

    destory()
}

/**
 * 根据条件修改
 * 将 指定字段 中包含有 srcValue 的统一修改为 newValue
 */
def updateByQuery( fieldName : String ,   srcValue: String ,
newValue : String ,   indexName:String ): Unit ={
    val updateByQueryRequest = new UpdateByQueryRequest()

    updateByQueryRequest.indices(indexName)
    val queryBuilder = QueryBuilders.matchQuery(fieldName,srcValue)

    val map = new util.HashMap[String,AnyRef]()
    map.put("newName",newValue)

    val script =
        new Script(ScriptType.INLINE ,   Script.DEFAULT_SCRIPT_LANG,
"ctx._source['movie_name']=params.newName" ,   map )

    updateByQueryRequest.setQuery(queryBuilder)
    updateByQueryRequest.setScript(script)

    esClient.updateByQuery(updateByQueryRequest
RequestOptions.DEFAULT)
}
```

6.2.4 删除数据

```
def main(args: Array[String]): Unit = {

    delete("101","movie_test20210103")

    destory()
}

/**
 * 删除
 */
def delete(docid : String ,   indexName : String ): Unit ={
    val deleteRequest = new DeleteRequest()
    deleteRequest.index(indexName)
    deleteRequest.id(docid)
    esClient.delete(deleteRequest,RequestOptions.DEFAULT)
}
```

6.2.5 查询数据

1) 条件查询

```
/**
 * search :
 *
 * 查询 doubanScore>=5.0 关键词搜索 red sea
 * 关键词高亮显示
 * 显示第一页，每页 20 条
 * 按 doubanScore 从大到小排序
 */

def search() : Unit = {
    val searchRequest = new SearchRequest()
    searchRequest.indices("movie_index")
    val searchSourceBuilder = new SearchSourceBuilder()
    //过滤匹配
    val boolQueryBuilder = new BoolQueryBuilder()
    val rangeQueryBuilder = new RangeQueryBuilder("doubanScore").gte("5.0")
    boolQueryBuilder.filter(rangeQueryBuilder)
    val matchQueryBuilder = new MatchQueryBuilder("name", "red sea")
    boolQueryBuilder.must(matchQueryBuilder)
    searchSourceBuilder.query(boolQueryBuilder)
    //高亮
    val highlightBuilder = new HighlightBuilder("name")
    searchSourceBuilder.highlighter(highlightBuilder)
    //分页
    searchSourceBuilder.from(0)
    searchSourceBuilder.size(20)
    //排序
    searchSourceBuilder.sort("doubanScore", SortOrder.DESC)

    searchRequest.source(searchSourceBuilder)
    val searchResponse = esClient.search(searchRequest, RequestOptions.DEFAULT)

    //处理结果
    //总数:
    val total: Long = searchResponse.getHits.getTotalHits.value
    println(s"共查询到 $total 条数据")
    //明细
    val hits: Array[SearchHit] = searchResponse.getHits.getHits
    for (searchHit <- hits) {
        val sourceJson : String = searchHit.getSourceAsString
        //高亮
        val fields: util.Map[String, HighlightField] = searchHit.getHighlightFields
        val field: HighlightField = fields.get("name")
        val fragments: Array[Text] = field.getFragments
    }
}
```

```
    val highlightText: Text = fragments(0)

    println(sourceJson)
    println(highlightText)
  }
}
```

2) 聚合查询

```
/**
 * 查询每位演员参演的电影的平均分，倒叙排序
 */
def searchAgg(): Unit = {

  val searchRequest = new SearchRequest()
  searchRequest.indices("movie_index")
  val searchSourceBuilder = new SearchSourceBuilder()
  //聚合
  //分组
  val termsAggregationBuilder: TermsAggregationBuilder =
    AggregationBuilders.terms("groupByActor").
      field("actorList.name.keyword").
      size(100).
      order(BucketOrder.aggregation("avg_score", false))
  //avg
  val avgAggregationBuilder: AvgAggregationBuilder =
    AggregationBuilders.avg("avg_score").
      field("doubanScore")

  termsAggregationBuilder.subAggregation(avgAggregationBuilder)
  searchSourceBuilder.aggregation(termsAggregationBuilder)

  searchRequest.source(searchSourceBuilder)
  val searchResponse: SearchResponse =
    esClient.search(searchRequest, RequestOptions.DEFAULT)
  //处理结果
  val groupByActorTerms: ParsedStringTerms =
    searchResponse.getAggregations.get[ParsedStringTerms]("groupByActor")
  val buckets: util.List[_] <: Terms.Bucket =
    groupByActorTerms.getBuckets
  import scala.collection.JavaConverters._
  for (bucket <- buckets.asScala) {
    val actorName: AnyRef = bucket.getKey
    val movieCount: Long = bucket.getDocCount
    val aggregations: Aggregations = bucket.getAggregations
    val parsedAvg: ParsedAvg =
      aggregations.get[ParsedAvg]("avg_score")
    val avgScore: Double = parsedAvg.getValue
    println(s"$actorName 共参演了 $movieCount 部电影， 平均评分为 $avgScore")
  }
}
```



```
}
```