ES从使用到原理

Prompt(目的)

1. FP 遇到典型 ES 问题有哪些，分哪些类别，归纳出需要回答/关注的一些问题
2. ES 资源使用是否合理
   a. 多少规模数据，使用多少内存、CPU是合理的。（拿 FP/RIO 来举例）
   b. 推理方案是什么，底层机制是什么。
3. 查询：流程、影响（对负载）、如何分析
   a. 历史问题：ES 的查询优化，mget, AIOS 做过的。可以和 彭志昂 交流
4. 异常恢复
   a. 多副本情况下，其中一份数据异常，比如某个节点挂掉。在系统恢复正常后，es能否自动恢复数据。恢复数据如果出问题应该如何处理。
5. 常见的一些问题排查方案：比如健康检查、慢查询、资源异常
   a. 再扣回典型问题，经过这次调研和分享，排查能优化么？能避免么？

背景

Elasticsearch（ES）是基于Apache Lucene的开源搜索引擎。它提供了一个分布式、实时的全文搜索和分析引擎，构建在Lucene库之上，同时还提供了更多的功能和工具来简化搜索应用的开发和管理。

了解其架构、存储方式、查询链路会有利于我们优化相关的CRUD操作，提高FP的性能。

这里将淡化分布式相关的概念，强调单机上的链路以及性能。

下面将从下面几个概念进行调研

1. **核心概念**
2. **查询与分析**
3. **数据存储与管理**
4. **集群管理与优化**

FP/AIOS相关问题整理

这里我们整理一下当前工单中遇到过的一些和ES相关的问题，以及当初的解决方案

相关链接：

[图像围栏][工程设计][RIO] ES MGET 稳定性

[图像围栏][版本][2.3.z RC][测试][FP]GPU集群es内存规格测试

2023.2.20-2.22 es性能测试

1. 线上问题分类

这里整理了20个ES直接关联的问题

1. FP使用问题(bug) 占 **40 %**
2. 配置导致异常 占 **25%**

评论（7）

Prompt(目的)

冯竣伥 5月23日 10:58
@蒋海波 @彭志昂 @柴森鹏

FP 遇到典型 ES 问题有哪些，分哪些类别…

狄宇奕 5月23日 21:14
从线上问题单子里找了 es 相关的问题，进行分类归纳

背景

冯竣伥 5月23日 10:31
这文章首先会有目的，类似给 GPT 的 Prompt。我觉得这个先对齐，这样对你调研结果的预期也会比较明确。

围绕几个关键的问题。可以是线上的问题里整理。找 @蒋海波 输入

我想到的几个：
1. ES 资源使用是否合理
多少数据，使用多少内存、CPU是合理的。推理是什么，机制是什么。
2. 查询的影响是什么。
3. 最后输出，常见的一些问题排查方案：比如健康检查、慢查询、资源异常
展开

冯竣伥 5月23日 10:32
@蒋海波 @彭志昂 @柴森鹏
可以输入

狄宇奕 5月23日 10:41
比较实用的建议，准备从这里 18-22 章节抄几个，基本上都能覆盖到
https://weread.qq.com/web/bookDetail/f9c32dc07184876ef9cdeb6

3. 启动失败 占 **35 %**

1和2大部分可以提前避免，3基本上和机房断电、数据盘损坏等物理因素有关，偶然因素较大。

问题类型	具体问题	处理方式	出现频率	总个数	工单链接
FP使用问题(bug)	es请求body过大	• extra_meta下的数据查询时只填extra_meta, 不再精细到具体的field http://gitlab.yitu-inc.com/ficus/product/face_platform/-/commit/d49a8f603d212e70ab899ac5e9e7ab6d31c89dc2 • 结构化搜图选择全量设备会因为传递的camera_id数量太多，导致查询es失败(上海市局定制) • 修改limit和 jvm配置	5	8	https://jira.yitu-inc.com/browse/SD-21086 https://jira.yitu-inc.com/browse/SD-20621 https://jira.yitu-inc.com/browse/SD-20852 https://jira.yitu-inc.com/browse/SD-20060 https://jira.yitu-inc.com/browse/SD-19921
	大量冗余的index	删除所有空的camera index http://gitlab.yitu-inc.com/ficus/product/face_platform/-/commit/1e744e504c653efed501db473019f8e271fc1a43	2		https://jira.yitu-inc.com/browse/SD-20393
	字段未设置为keyword	修改mapping	1		https://jira.yitu-inc.com/browse/SDD-4658
配置导致异常	gc频繁	修改limit和 jvm配置	3	5	https://jira.yitu-inc.com/browse/SD-21466 https://jira.yitu-inc.com/browse/SD-20232 https://jira.yitu-inc.com/browse/SD-19158
	慢查询	limit调整为20Gi, jvm配置调整为16g	1		https://jira.yitu-inc.com/browse/SD-21790
	删数据报错	增大thread_pool配置和队列长度	1		https://jira.yitu-inc.com/browse/SD-21466
启动失败	内存锁定	memory_lock全都改成false ./fp stop，./fp uninstall,./fp install解决内存锁定问题	1	7	https://jira.yitu-inc.com/browse/SD-20552

线上问题分类

冯竣倬 6月3日 20:21
这里的分类/分析是不是可以合到一个表里？ 是不是都是同样的问题？

狄宇奕 6月3日 20:23
分析列出了具体解决方法，可以合并一下

总个数

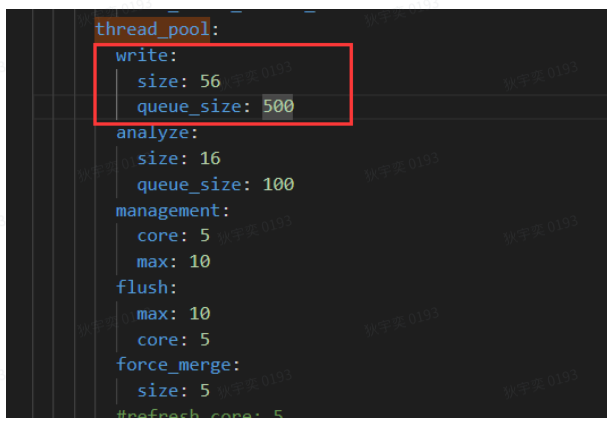
冯竣倬 6月3日 20:19
加一列，“RC 是否已修复”，“修复方案”

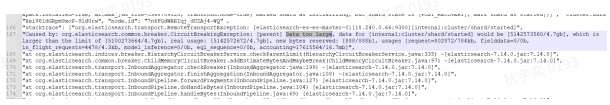
冯竣倬 6月3日 20:20
“根因是否定位”

	有大量unassigned分片		1		https://jira.yitu-inc.com/browse/SD-20232
	master not discovered	kubectl delete pods elastic-operator-0 -n elastic-system ./fp start storage	2		https://jira.yitu-inc.com/browse/SD-20170 https://jira.yitu-inc.com/browse/SD-15785
	数据损坏	工具修复	3		https://jira.yitu-inc.com/browse/SD-10231 https://jira.yitu-inc.com/browse/SD-3826 https://jira.yitu-inc.com/browse/SD-9371

2. 线上问题分析

工单	问题描述	问题原因	类别	解决方法
https://jira.yitu-inc.com/browse/SD-20552	启动异常	内存锁定	es配置	middlewares/elasticsearch/config/elasticsearch_ha.yaml，middlewares/elasticsearch/config/elasticsearch_bak.yaml，middlewares/elasticsearch/config/elasticsearch_ha_stop.yaml这三个文件里的memory_lock全都改成false ./fp stop，./fp uninstall,./fp install解决内存锁定问题
https://jira.yitu-inc.com/browse/SD-20232	启动异常	readiness失败导致pod起不来 /_cat/shards?v看到有大量unassigned分片	es配置	注释掉readiness
https://jira.yitu-inc.com/browse/SD-20170 https://jira.yitu-inc.com/browse/SD-15785	启动异常	master not discovered的报错		先执行./fp stop all 停掉所有服务 执行下 kubectl delete pods elastic-operator-0 -n elastic-system 然后重启下fp的storage ./fp start storage ./fp start ragnaros ./fp start product
https://jira.yitu-inc.com/browse/SD-10231	数据损坏	es master的数据损坏	异常恢复	1.停止es 2.备份runtime/data/elasticsearch/es1 3.删除runtime/data/elasticsearch/es1 4.重启es
		数据损坏		es red 的情况，可以先用 es工具修复

https://jira.yitu-inc.com/browse/SD-3826	数据损坏		异常恢复	es_status.py
https://jira.yitu-inc.com/browse/SD-9371	数据损坏	数据损坏	异常恢复	服务器意外关机，重启修复磁盘，其他正常，ES启动报错，根据日志提示可能是数据损坏 清理ES损坏文件，重启ES。 1. 停下es (然后等1min) 2. 把 fp_run/config/elasticsearch 重命名 3. 把 fp_run/data/elasticsearch 重命名 4. 在fp_deploy/console/下　执行一下 ./cluster_update_all.sh 5. 重启下es
https://jira.yitu-inc.com/browse/SD-21790	ES查询慢	内存占用限制	查询性能 es配置	ES的limit调整为20Gi, jvm配置调整为16g
https://jira.yitu-inc.com/browse/SD-21466 https://jira.yitu-inc.com/browse/SD-20232 https://jira.yitu-inc.com/browse/SD-19158	频繁gc	es报错 which is larger than the limit of xxxx	性能 es配置	修改limit和 jvm配置
https://jira.yitu-inc.com/browse/SD-21466		删除部分数据时报错。发现是thread_pool配置的线程池太小了，增大配置	性能	修改thread_pool中队列长度 
https://jira.yitu-inc.com/browse/SD-21086	es请求body过大	es请求body过大	FP bug	extra_meta下的数据查询时只填extra_meta, 不再精细到具体的field http://gitlab.yitu-inc.com/ficus/product/face_platform/-/comm

https://jira.yitu-inc.com/browse/SD-20621 https://jira.yitu-inc.com/browse/SD-20852 https://jira.yitu-inc.com/browse/SD-20060				it/d49a8f603d212e70ab899ac5e9e7ab6d31c89dc2 结构化搜图选择全量设备会因为传递的camera_id数量太多，导致查询es失败 (上海市局定制)	
https://jira.yitu-inc.com/browse/SD-18560 https://jira.yitu-inc.com/browse/SD-20393	大量冗余的index	大量冗余的index	FP bug	删除所有空的camera index http://gitlab.yitu-inc.com/ficus/product/face_platform/-/commit/1e744e504c653efed501db473019f8e271fc1a43	
https://jira.yitu-inc.com/browse/SDD-4658	rec_hat_color无法被查询到	字段未设置为keyword	es配置	．middlewares/elasticsearch/action/init.py这个文件中增加"rec_hat_color": { "type" : "keyword" } ,，一共两处 ７．bash init.sh ８.执行kubectl get svc -n middlewares|grep elas 获取es http的ip，curl -X GET 10.98.140.134:9200/_cat/indices?v|grep track_meta|sort -k3 -r查看最新的track_meta,然后执行 curl --location --request PUT 'http://esip:9200/track_meta19341/_mapping' \ --header 'Content-Type: application/json' \ --data-raw '{ "properties": { "rec_hat_color": { "type": "keyword" } } }'让es改动在当前indice生效	
https://jira.yitu-inc.com/browse/SD-19921	查询失败		es配置 FP bug	es 到20G后 结构化检索可以检索 后面有调整到40G es的日志有一些请求失败的原因是http请求的数据量太大，超过4.7g 	

序号	Top N问题	频率	影响客户	涉及版本	工单号
543	ES内存爆了，导致结构化搜索返回失败	8	广东惠州市局	2.2.6	SD-2008
			广东汕头市公安	2.2.6	SD-1994
			广东潮州市公安局科信支队视频大队	2.2.10	SD-2008 SD-2041
			江苏省南京市公	2.2.8	SD-1992
			福建省厦门市公	9	SD-1978
			山东滨州市公安局科信处	2.2.8	SD-1968
			浙江温州市局	2.1.7	SD-1954
			浙江杭州拱墅	2.2.8	SD-2039
32	ES的过期数据未能清理，导致业务系统瘫痪（2次）	1	海外迪拜警局Oyoon	2.1.6海外版	——

3. FP/RIO资源评估方法

1. 通用评估准则

和ES紧密相关的三个资源：磁盘、内存、CPU

下面是一个通用的评估方法：

	细分类型	需求量描述	计算公式	示例计算	依据
磁盘空间					
	原始数据量	每天生成的数据量	20GB * 365天	7300GB	
	索引开销	索引开销（假设20%）	7300GB * 0.2	1460GB	通常，索引开销大约在原始数据量的10%到20%之间
	副本数	副本数（假设1个副本）	(7300GB + 1460GB) * (1 + 1)	17520GB	
总磁盘空间		原始数据量 + 索引开销 + 副本数	7300GB * (1 + 0.2) * (1 + 1)	17520GB	
内存					
	堆内存	每个节点的堆内存	推荐总内存的一半，但不超过32GB	31GB	ES不建议为JVM配置超过32GB的内存，超过32GB时，Java内存指针压缩失效，浪费一些内存，降低了CPU性能，GC压力也较大。因此推荐设置为31GB： -Xmx31g -Xms31g 确保堆内存最小值（Xms）与最大值（Xmx）大小相同，防止程序在运行时动态改变堆内存大小，这是很耗系统资源的过程。
	操作系统缓存	操作系统缓存需等于或多于堆内存	堆内存 * 1	31GB	搜索操作很依赖对系统cache的命中，标准的建议是把50%的可用内存作为ES的堆内存，为Lucene保留剩下的50%，用作系统cache。
总内存		每个节点总内存	堆内存 * 2	62GB	
CPU					
	单节点CPU数	每个节点的虚拟CPU	4到8个vCPU	假设3个节点，每个节点4到8个vCPU	**小型集群推荐** • 节点数：3个 • vCPU/节点：4个 • 总vCPU数：12个
总CPU		总节点数	节点数 * (4到8个vCPU)	3个节点 * (4到8个vCPU) = 12到24个vCPU	

Tips:关于Java指针压缩

> 通常，索引开销大约在原始数据量的10%…
>
> 冯竣倬 6月3日 20:23
>
> 这个拿实际数据对过么？ 是不是还和索引数量相关。通常多少索引的情况下，空间占到 20%？

在 Java 中，所有的对象都分配在堆上，并通过一个指针进行引用。 普通对象指针（OOP）指向这些对象，通常为 CPU 字长 的大小：32 位或 64 位，取决于你的处理器。指针引用的就是这个 OOP 值的字节位置。

对于 32 位的系统，意味着堆内存大小最大为 4 GB。对于 64 位的系统， 可以使用更大的内存，但是 64 位的指针意味着更大的浪费，因为你的指针本身大了。更糟糕的是， 更大的指针在主内存和各级缓存（例如 LLC，L1 等）之间移动数据的时候，会占用更多的带宽。

Java 使用一个叫作 内存指针压缩（compressed oops）的技术来解决这个问题。 它的指针不再表示对象在内存中的精确位置，而是表示 偏移量 。这意味着 32 位的指针可以引用 40 亿个 对象 ， 而不是 40 亿个字节。最终， 也就是说堆内存增长到 32 GB 的物理内存，也可以用 32 位的指针表示。

一旦你越过那个神奇的 ~32 GB 的边界，指针就会切回普通对象的指针。 每个对象的指针都变长了，就会使用更多的 CPU 内存带宽，也就是说你实际上失去了更多的内存。事实上，当内存到达 40–50 GB 的时候，有效内存才相当于使用内存对象指针压缩技术时候的 32 GB 内存。

这段描述的意思就是说：**即便你有足够的内存，也尽量不要 超过 32 GB。因为它浪费了内存，降低了 CPU 的性能，还要让 GC 应对大内存。**

备注

- **磁盘空间需求**：根据数据增长情况进行预估，注意考虑索引开销和副本数。
- **内存需求**：堆内存设置为总内存的一半，操作系统缓存也需要足够的内存。
- **CPU需求**：根据节点数和并发量进行合理分配，通常每个节点4到8个vCPU。

2. FP怎么评估(TODO)

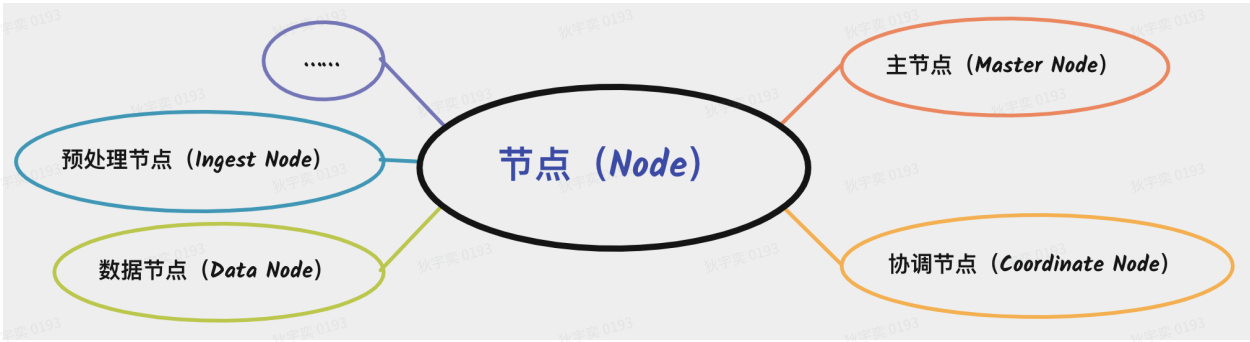
3. AIOS怎么评估(TODO)

核心概念

ES的核心概念。

- **索引（Index）**：存储、管理和查询文档的地方，相当于数据库中的表。
- **类型（Type）**：Elasticsearch 6.x 及以前版本中的分类概念，从 7.x 开始废弃。
- **文档（Document）**：存储的数据单元，是一个 JSON 对象，相当于数据库表中的一行。
- **节点（Node）**：运行 Elasticsearch 实例的基本单元，可以有多种角色。
- **集群（Cluster）**：由多个节点组成的集合，协同工作来存储数据和处理请求。
- **分片（Shard）**：索引的基本存储单元，分为主分片和副本分片。
- **副本（Replica）**：主分片的复制，用于提高数据的可用性和查询性能。

1. 节点（Node）与集群（Cluster）



- **集群**是有**多个节点**组成的，在上图中可以看到集群中有多个不同种类型的节点。
- **节点**是一个Elasticsearch的实例，本质上是一个Java进程。
  - 每个节点上面都**保存**着集群的**状态信息**，包括所有的**节点信息**、所有的**索引**和相关的**Mapping、Setting**信息和**分片的路由信息**等。
  - 节点按照角色可以划分为主节点、数据节点、协调节点和预处理节点等

节点类型	职责与功能
主节点	• 管理集群状态信息，包括处理创建、删除索引等请求。 • 负责维护和更新集群状态，包括分片路由信息，决定分片被分配到哪个节点。 • 是唯一能够修改集群状态信息的节点，负责将更改同步给其他节点。
协调节点	• 接收客户端的请求，并将请求路由到合适的节点。 • 负责汇集来自不同节点的结果，将结果汇总返回给客户端。
数据节点	• 保存实际的数据。 • 解决水平扩展和数据单点问题，增加数据节点可以增加存储容量和分担负载。 • 执行实际的数据读写操作。
预处理节点	• 数据前置处理转换的节点，支持 pipeline 管道设置。 • 可以对数据进行过滤、转换等操作，为数据处理提供灵活性和扩展性。 • 可以在数据传输前对数据进行加工处理，满足不同的业务需求。

在小型 Elasticsearch 集群中，为了节省资源，可以让主节点兼具数据节点的角色，即在主节点上保存实际的数据。在某些场景下，也可以让协调节点兼具数据节点的角色，即让协调节点保存实际的数据。

在我们的fp部署目录中middlewares/elasticsearch/config/es_config.yaml定义了我们使用es的一些配置

下面这个是我在44.58上调用健康检查接口拿到的集群状态信息

```
curl -X GET "http://10.102.178.118:9200/_cluster/health?pretty"
{
  "cluster_name" : "elasticsearch",
  "status" : "green",
```

```
  "timed_out" : false,
  "number_of_nodes" : 3,
  "number_of_data_nodes" : 3,
  "active_primary_shards" : 1,
  "active_shards" : 2,
  "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
}
```

含义整理如下：从绿色的表格中可以看出我们的部署方式是1个master&data + 2个data节点

字段	值	描述
cluster_name	elasticsearch	Elasticsearch 集群的名称。
status	green	集群的健康状态。可能的值包括：green（绿色）、yellow（黄色）、red（红色）。
timed_out	false	指示请求是否在完成之前超时。
number_of_nodes	3	集群中节点的总数。
number_of_data_nodes	3	集群中数据节点的数量。
active_primary_shards	1	活动的主分片数量。
active_shards	2	活动分片的总数（包括主分片和副本分片）。
relocating_shards	0	当前正在重新定位的分片数量。
initializing_shards	0	当前正在初始化的分片数量。
unassigned_shards	0	未分配的分片数量。
delayed_unassigned_shards	0	延迟未分配的分片数量。
number_of_pending_tasks	0	集群中挂起的任务数量。
number_of_in_flight_fetch	0	索引数据提取任务的数量。
task_max_waiting_in_queue_millis	0	在队列中等待的任务的最长等待时间（毫秒）。
active_shards_percent_as_number	100.0	活动分片的百分比。

有兴趣的可以再观察一下 `curl -X GET "http://10.102.178.118:9200/_nodes"` 可以获取到详细的nodes信息.这里贴一下拿到的json数据：

https://jsonhero.io/j/O3nVp3puaM9j?path=%24.nodes.Gal8kewGQ-KEbTkM3AOPSQ.roles

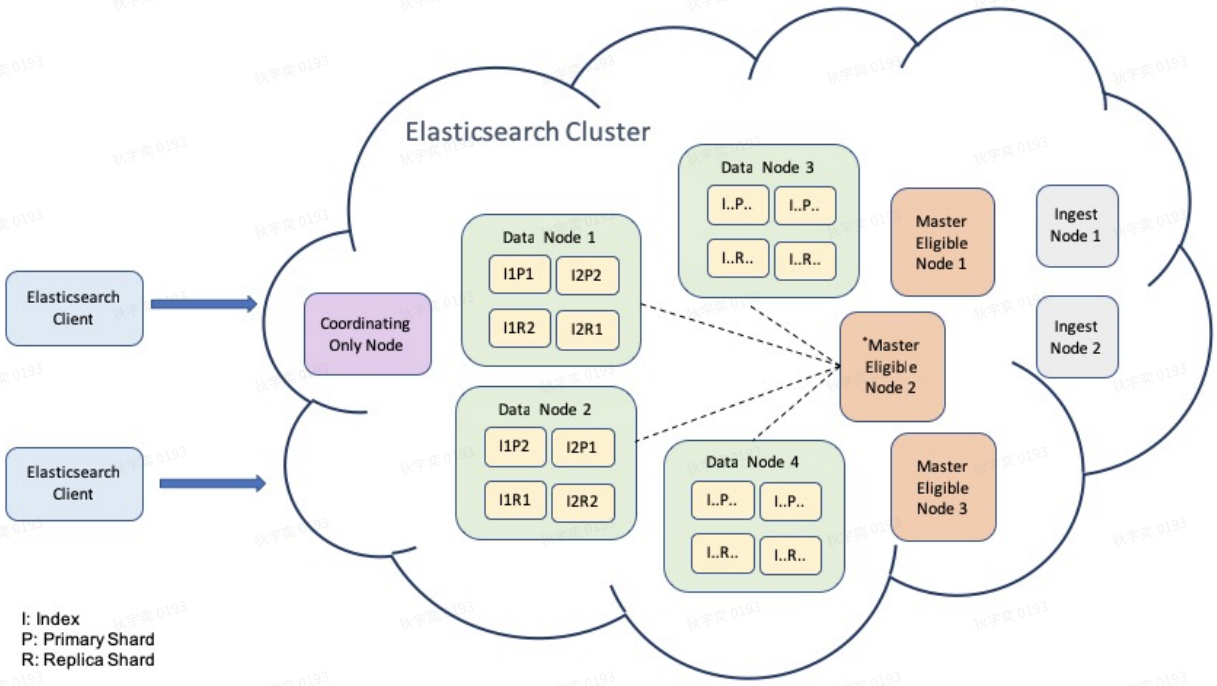
jsonhero.io

从roles字段可以看出3个节点的role信息分别为

master节点	data节点	备注
["ingest", "master", "data",	["ingest", "data", "ml"	• 都具备预处理、存储和索引数据、以及机器学习的功能

"ml"]	]	

2. 分片（Shard）与副本（Replica）



分片

- **定义**: 分片是 Elasticsearch 中索引的基本存储单元，一个索引可以被拆分成多个分片。
- **类型**: 分片有两种类型：主分片（Primary Shard）和副本分片（Replica Shard）。
- **配置**: 分片数量在索引创建时指定，主分片数量创建后不可更改。
- **分配**: Elasticsearch 会自动将分片分配到集群中的不同节点上，以均衡负载和提高可用性。

副本

- **定义**: 副本是主分片的复制，用于提高数据的可用性和容错能力。
- **数量**: 每个主分片可以有多个副本分片，副本数量可以在索引创建后动态调整。
- **角色**: 副本不仅在主分片失效时接管工作，还可以分担查询负载，提升查询性能。

简单来说，一个表太大了可能会降低查询效率，所以需要分成多个shard，可以支持并行查询以及存储的扩展伸缩(数据越多分片越多)

出于容灾的考虑，一个shard又得做多个副本。

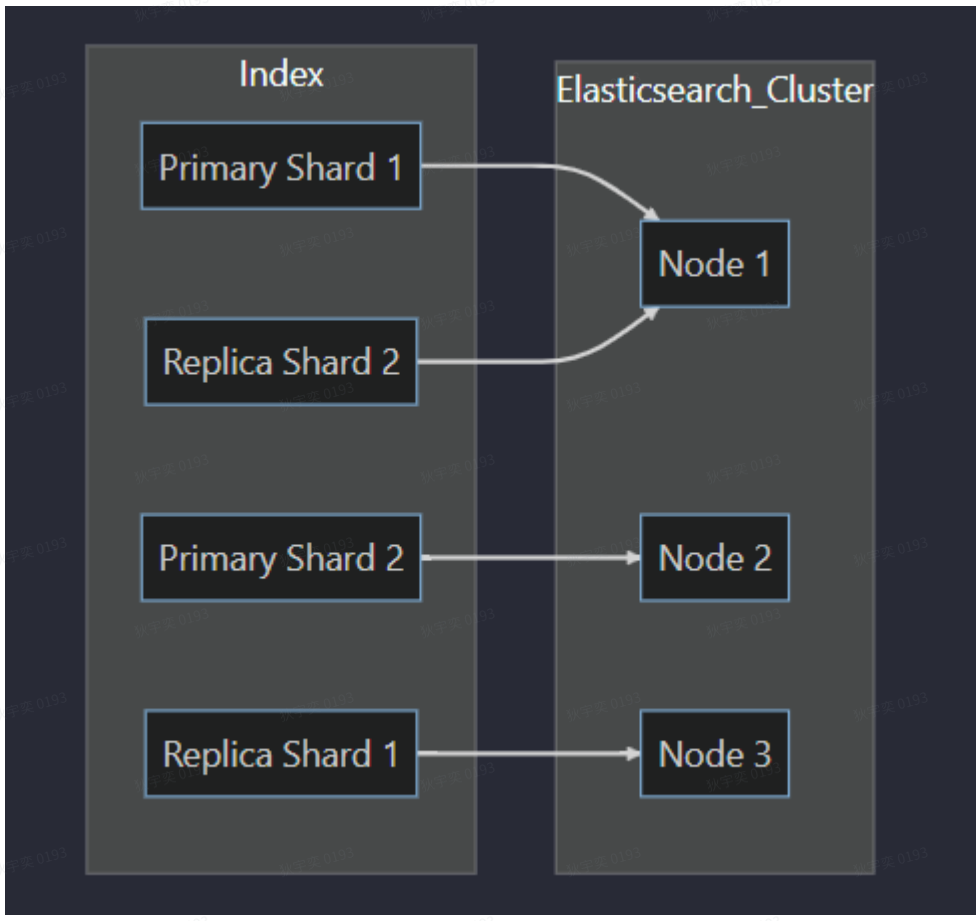
这里给出一个分片、节点、集群的关系

```
graph LR
    subgraph Elasticsearch_Cluster
        A[Node 1]
        B[Node 2]
        C[Node 3]
    end

    subgraph Index
        D[Primary Shard 1]
        E[Primary Shard 2]
        F[Replica Shard 1]
        G[Replica Shard 2]
    end

    D --> A
    E --> B
```

```
    F --> C
    G --> A
```



主分片 1 存储在 Node 1，主分片 2 存储在 Node 2。这符合 Elasticsearch 的默认行为，即**每个主分片都会存储在不同的节点上，以提高数据的容错性和性能**。

副本分片 1 存储在 Node 3，副本分片 2 存储在 Node 1。这里的副本分片会根据你的配置和集群状态动态分配到不同的节点上，以提供数据的冗余备份。

针对于某个具体的index，我们可以这样查询分片个数以及在node上的分布

```
curl -X GET "http://10.110.192.50:9200/_cat/indices?v"
```

```
health status index           uuid                   pri rep
docs.count docs.deleted store.size pri.store.size
```

```
green  open   track_meta19864 X3YLAIUbTs6qh9hbXVf67Q   3   0
   16            0    123.6kb        123.6kb
```

```
green  open   repository      8lj1oObOT4K2VwJvO8rrLg   1   1
    2            0     19.1kb          9.5kb
```

- pri：表示主分片的数量。
- rep：表示副本分片的数量。
- docs.count：表示索引中文档的数量。
- docs.deleted：表示已删除的文档数量。
- store.size：表示索引占用的存储空间大小。
- pri.store.size：表示主分片占用的存储空间大小。

这里我们可以观察出track_meta19864 这个index，有 3 个主分片，没有副本分片。

```
curl -X GET "http://10.110.192.50:9200/_cat/shards/track_meta19864?v"

index           shard prirep state   docs  store ip          node

track_meta19864 2     p      STARTED    8 61.7kb 18.240.0.80 elasticsearch-es-es-data-1

track_meta19864 1     p      STARTED    2 20.6kb 18.240.0.81 elasticsearch-es-es-master-0

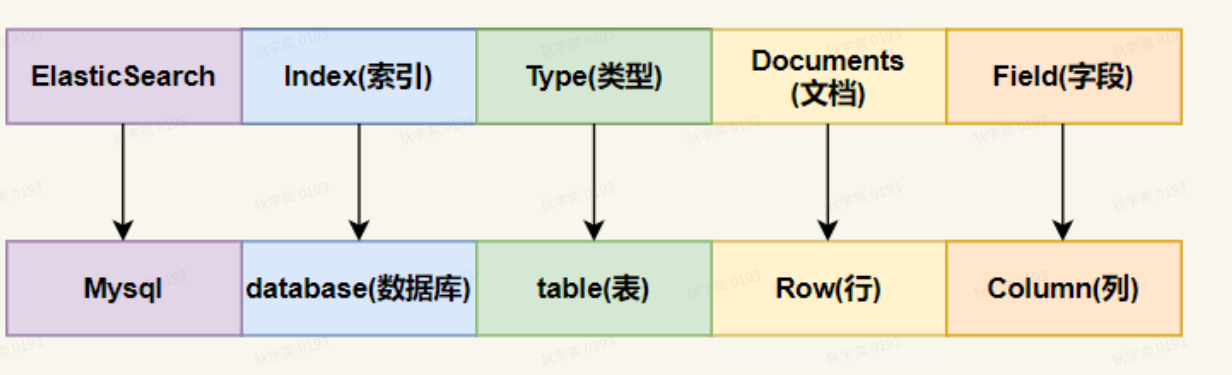
track_meta19864 0     p      STARTED    6 41.3kb 18.240.0.82 elasticsearch-es-es-data-0
```

这里我们可以观察出track_meta_19864 这个index的3个分片分别分布在3个node上。

侧面说明如果某个node挂了，那么我们的路人数据能检索到的数据就会少1/3。

另外使用_cluster/allocation/explain可以用于解释为什么某个分片（shard）没有被分配到某个节点或集群中的任何节点上

3. 索引（Index）、类型（Type）、文档（Document）



索引（Index）

- **定义**: 索引是 Elasticsearch 中存储、管理和查询文档的地方。可以把索引看作是一个数据库中的表。
- **结构**: 每个索引包含多个类型（在 Elasticsearch 6.x 及以前版本）和文档。
- **创建**: 索引在创建时定义映射（Mapping）和设置（Settings）。映射定义了**字段类型和字段属性**，而设置包括**分片和副本的数量等配置**。
- **命名**: 索引名称必须是小写，不能包含空格和一些特殊字符。

类型（Type）

- **定义**: 类型是 Elasticsearch 6.x 及以前版本中的概念，用于将索引内的文档分类。类型相当于数据库表中的列族。
- **结构**: 每个类型包含多个文档。不同类型可以有不同的字段。
- **废弃**: 从 Elasticsearch 7.x 开始，类型概念被废弃，一个索引中只能包含一个类型，以简化数据模型和避免冲突。

文档（Document）

- **定义**: 文档是 Elasticsearch 中存储的数据单元，相当于数据库表中的一行。每个文档是一个 JSON 对象。
- **结构**: 文档包含多个字段，每个字段都有数据类型，如字符串、数字、日期等。
- **唯一标识**: 每个文档有一个唯一的 `_id` 字段，作为其唯一标识符。

这里只需要关心索引在创建时定义映射（Mapping）和设置（Settings），这里关系到CRUD的性能，具体定义在：http://gitlab.yitu-inc.com/ficus/product/face_platform/-/blob/release/2.3.4/eng/deploy/middlewares/elasticsearch/action/init.py

这里就存放着track_meta_mapping，为一个es的索引模板，用于定义索引的配置和映射。

这里我们放一下简化的mapping

```
track_meta_mapping = """
{
    "index_patterns": ["track_meta*"],
    "settings": {
        "number_of_shards": %d,
        "number_of_replicas": 0,
        "index" : {
          "sort.field" : "timestamp",
          "sort.order" : "desc" ,
        "translog.durability" : "async",
        "translog.sync_interval" : "30s",
        "refresh_interval" : "2s"
        }
    },
    "mappings": {
    "dynamic": false,
    "_source": {
      "enabled": true
    },
    "properties" : {
      "id" : {
        "type" : "keyword"
      },
      "camera_id" : {
        "type" : "keyword"
      },
      ...
    }
  }
}
"""
```

参数	描述	
index_patterns	["track_meta*"]	指定了适用于该模板的索引名称模式，这里指定了所有以 `track_meta` 开头的索引都会应用这个模板。
settings		指定了索引的设置，包括主要的配置参数
- number_of_shards	根据代码中的动态填充	索引的主分片数量

- number_of_replicas	0（表示没有副本分片）	索引的副本分片数量
- index.sort.field	"timestamp"	索引的默认排序字段和排序顺序
- index.sort.order	"desc"	
- translog.durability	"async"	指定了事务日志的持久性和同步间隔 `index.translog.durability`:默认是request，每个请求都落盘；设置成async，可异步写入 `index.translog.sync_interval`：默认5s，不能小于100ms
- translog.sync_interval	"30s"	
- refresh_interval	"2s"	索引的刷新间隔，控制了新写入数据对搜索的可见性
mappings		指定了索引的映射，定义了索引中字段的类型和属性
- dynamic	false	表示禁用动态映射，即不允许自动创建新字段
- _source.enabled	true	设置为 `true`，表示启用 `_source` 字段，允许存储文档的原始 JSON 内容。
- properties		定义了索引中的字段和其属性，比如 `id` 和 `camera_id`，它们的类型都是 `keyword`，表示这些字段是关键字类型，通常用于精确匹配和聚合操作。
- id	类型：keyword	
- camera_id	类型：keyword	
- ...	其他字段及其属性	

这里有几个参数translog、refresh_interval在后续讲到分片具体实现细节时会涉及到。

4. ES和Lucene之间的关系

其实就是就是单机到分布式的扩展

1. **基于关系**：
   - Elasticsearch 是在 Lucene 基础上构建的搜索引擎。它利用 Lucene 的核心功能，并添加了分布式特性、RESTful API、数据聚合能力等功能。
2. **功能扩展**：
   - Elasticsearch 在 Lucene 的基础上提供了更高级的搜索和分析功能，例如集群管理、索引管理、数据复制、故障转移等功能。
3. **易用性**：

- Elasticsearch 提供了更友好的 RESTful API 和 JSON 格式的交互方式，使得开发者更容易使用和集成 Elasticsearch 进行搜索应用的开发。

4. **分布式特性**：
   - Elasticsearch 强调分布式搜索与分析，通过水平扩展来处理大规模数据，并提供高可用性、容错性等功能。Lucene 作为其搜索核心，支持单机搜索。

5. 内部模块

```
Cluster --> allocation
   |             |
   V             V
Discovery <-- gateway
   |          /  |
   V         V   V
Indices     HTTP  Transport
    \       /
      Engine
```

模块名称	功能描述
ClusterCluster	- 管理集群状态，发布新生成的集群状态到所有节点 - 触发分片分配，决策分配到哪个节点 - 迁移分片保持数据平衡
allocation	- 封装分片分配功能和策略 - 主分片和副分片的分配 - 处理创建新索引和集群完全重启等过程
Discovery	- 负责发现集群中的节点，选举主节点 - 对节点加入或退出进行相应行动
gateway	- 持久化存储Master广播下来的集群状态数据 - 在集群完全重启时恢复这些数据
Indices	- 全局级索引设置管理 - 索引数据恢复功能 - 主分片和副分片恢复在集群启动阶段实现
HTTP	- 允许通过 JSON over HTTP 访问 ES 的 API - 完全异步通信，解决 C10k 问题
Transport	- 集群内节点间的内部通信模块 - 使用 TCP 通信，维持 TCP 长连接
Engine	- 封装对 Lucene 的操作和 translog 的调用 - 提供一个分片读写操作的最终提供者

查询/写入与分析

1. 一条查询请求的链路

1. **搜索请求链路**

搜索请求步骤:

1. **客户端发送请求**: 客户端通过REST API发送搜索请求到集群中的任意一个节点（称为协调节点）。
2. **协调节点解析请求**: 协调节点解析搜索请求，将其转化为Lucene查询。
3. **分发查询到分片**: 协调节点将查询请求分发到相关的主分片和副本分片上。
4. **分片执行查询**: 每个分片在本地执行查询，将结果返回给协调节点。
5. **协调节点合并结果**: 协调节点收集并合并所有分片的查询结果。
6. **返回结果给客户端**: 协调节点将最终结果返回给客户端。

搜索请求流程图:

```
flowchart TD
    A[客户端发送搜索请求] --> B[协调节点]
    B --> C[解析搜索请求]
    C --> D[分发查询到相关分片]
    D --> E1[主分片]
    D --> E2[副本分片]
    E1 --> F[执行查询]
    E2 --> F[执行查询]
    F --> G[协调节点合并结果]
    G --> H[返回结果给客户端]
```



2. 一条写入请求的链路

存储请求步骤：

1. **客户端发送请求**: 客户端通过REST API发送存储请求（例如索引文档请求）到集群中的任意一个节点（称为协调节点）。
2. **协调节点解析请求**: 协调节点解析请求并确定目标索引。
3. **分片路由**: 协调节点使用路由算法确定请求应发送到哪个主分片。
4. **写入主分片**: 请求被转发到目标主分片，主分片执行写入操作。
5. **同步到副本分片**: 主分片成功写入后，将变更同步到所有副本分片。
6. **确认写入成功**: 当所有副本分片确认写入成功后，主分片返回成功信息给协调节点。
7. **返回结果给客户端**: 协调节点将成功信息返回给客户端。

```
flowchart TD
    A[客户端发送存储请求] --> B[协调节点]
    B --> C[解析请求]
    C --> D[分片路由]
    D --> E[目标主分片]
    E --> F[写入数据]
    F --> G[同步到副本分片]
    G --> H1[副本分片1]
```

```
    G --> H2[副本分片2]
    H1 --> I[确认写入成功]
    H2 --> I[确认写入成功]
    I --> J[主分片确认]
    J --> K[协调节点返回结果]
    K --> L[返回结果给客户端]
```



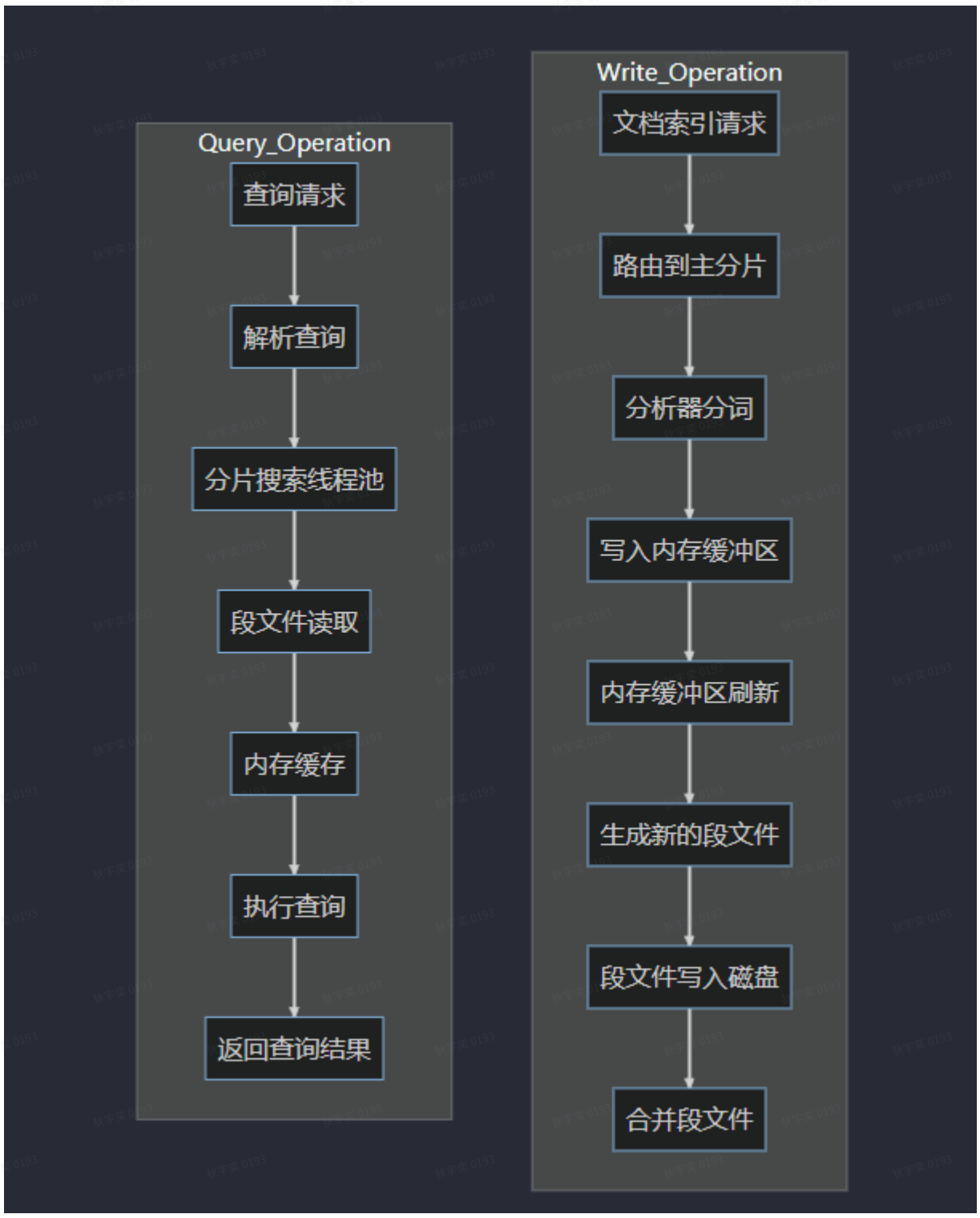
3. Elasticsearch 分片内部执行逻辑

从上面我们可以窥探查询链路，那么具体到某个分片，内部的查询/写入逻辑又是如何的?

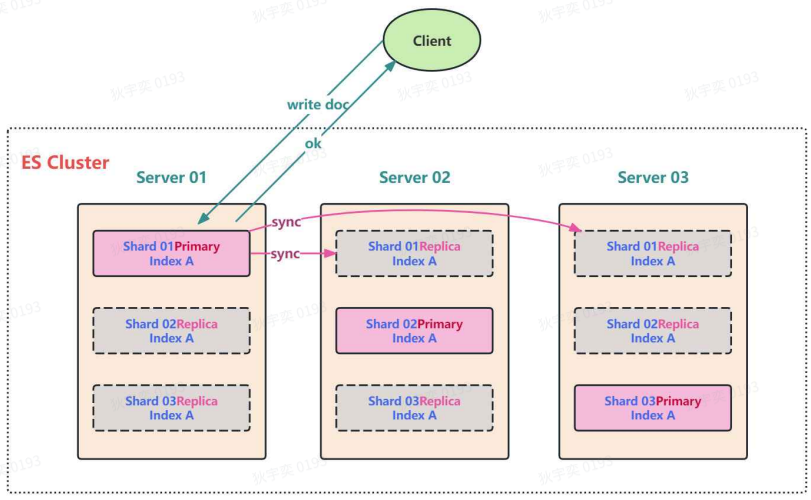
```
flowchart TD
    subgraph Write_Operation
        A[文档索引请求] --> B[路由到主分片]
        B --> C[分析器分词]
        C --> D[写入内存缓冲区]
        D --> E[内存缓冲区刷新]
        E --> F[生成新的段文件]
        F --> G[段文件写入磁盘]
```

```
        G --> H[合并段文件]
    end

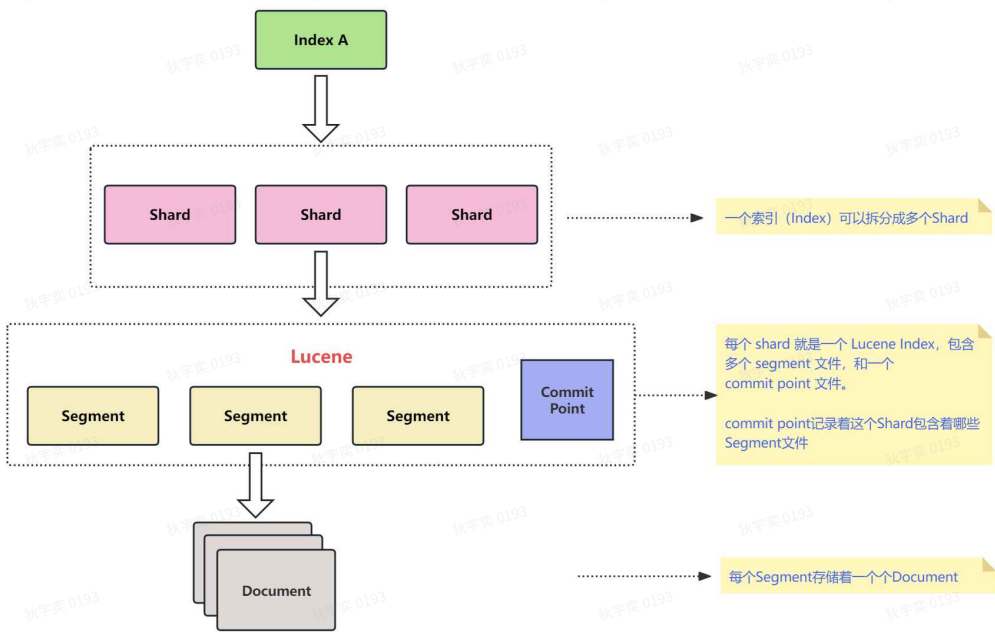
    subgraph Query_Operation
        I[查询请求] --> J[解析查询]
        J --> K[分片搜索线程池]
        K --> L[段文件读取]
        L --> M[内存缓存]
        M --> N[执行查询]
        N --> O[返回查询结果]
    end
```



3.1 写入逻辑



ES集群整体架构
1、ES 集群里面有多个 Server 节点，一个 ES Index 有多个 shard 分片，每个 shard 有多个副本
2、其中有一个 primary 主副本，负责写入，其他副本为 replica，不能写，只能同步 primary 的数据，但可以处理读请求。
3、ES 收到写请求后，会将请求路由到目标shard的primary 副本。

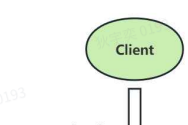


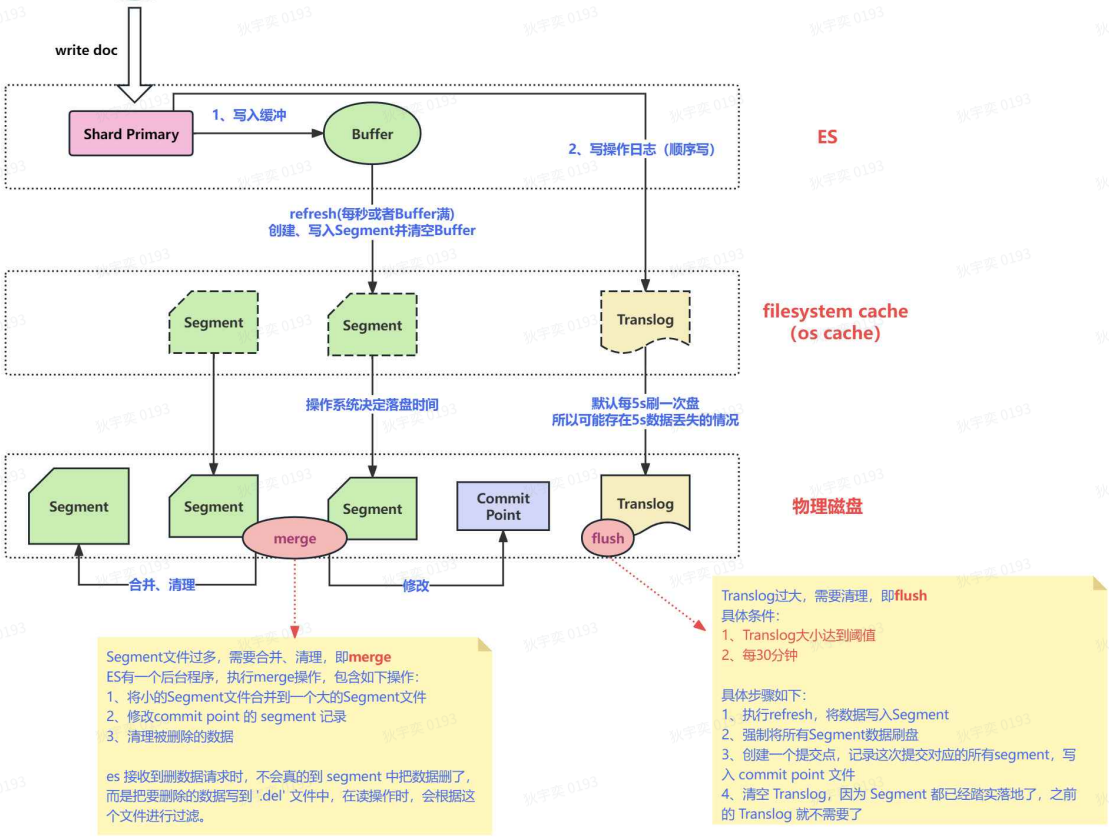
一个索引（Index）可以拆分成多个Shard

每个 shard 就是一个 Lucene Index，包含多个 segment 文件，和一个 commit point 文件。

commit point记录着这个Shard包含着哪些Segment文件

每个Segment存储着一个个Document





Segment文件过多，需要合并、清理，即**merge**
ES有一个后台程序，执行merge操作，包含如下操作：
1、将小的Segment文件合并到一个大的Segment文件
2、修改commit point 的 segment 记录
3、清理被删除的数据

es 接收到删数据请求时，不会真的到 segment 中把数据删了，而是把要删除的数据写到 '.del' 文件中，在读操作时，会根据这个文件进行过滤。

Translog过大，需要清理，即**flush**
具体条件：
1、Translog大小达到阈值
2、每30分钟

具体步骤如下：
1、执行refresh，将数据写入Segment
2、强制将所有Segment数据刷盘
3、创建一个提交点，记录这次提交对应的所有segment，写入 commit point 文件
4、清空 Translog，因为 Segment 都已经踏实落地了，之前的 Translog 就不需要了

当有新数据写入的时候，ES首先写入Index Buffer区域，此时是无法检索的。

默认情况下ES每秒钟进行一次Refresh操作，将Index Buffer中的index刷新到文件系统缓存，在文件系统缓存中，是以Segment进行存储的，而且是可以被搜索到的，这就是ES实现近实时搜索：文档的更改无法立即被搜索到，但是在一定时间会变得可见。

Refresh 触发的情况有3种：

1. 按照时间频率触发，默认情况是每 1 秒触发 1 次 Refresh，可通过 `index.refresh_interval` 设置；**FP这边设置的是2s**

2. 当Index Buffer 被占满的时候，会触发 Refresh，Index Buffer 的大小默认值是 JVM 所占内存容量的 10%；(FP默认给的是8G，也就是820MB)
3. 手动调用调用Refresh API。

由于Refresh操作默认间隔为1s，因此会产生大量的小Segment，ES查询时会同时查询所有的Segment，并对结果进行汇总，大量小Segment会使性能变差。因此ES会对小Segment进行段合并（Merge），合并操作会丢弃掉重复的键，并只保留每个键最近的更新。段合并之后搜索请求可以直接访问合并之后的Segment，从而提高搜索性能。

Merge触发的情况有2种：

1. ES自动启动后台任务进行Merge；
2. 手动调用_forcemerge API主动触发。

在段合并完成之后，ES会将Segment文件Flush到磁盘中，并创建一个Commit Point文件，用来标识被Flush到磁盘的Segment。Commit Point其实是记录所有的Segment信息，关于移除的Segment的信息会记录在“.del”文件中，查询结果后会从该文件中进行过滤。

Flush操作是将Segment从文件系统缓存写入到磁盘进行持久化，在执行 Flush 的时候会依次执行下面操作：

1. 清空Index Buffer
2. 记录 Commit Point
3. 刷新Segment到磁盘
4. 删除translog

小结一下，**ES的存储最终的形式是segment**。

3.2 查询逻辑

经济基础决定上层建筑，ES的分片存储决定了搜索机制。

搜索需要遍历分片所有的Lucene分段，因此合并Lucene分段对搜索性能有好处。

3.2.1 GET/MGET与Search

ES的读取分为GET和Search两种操作，这两种读取操作有较大的差异

操作类型	描述	实现方式	影响因素和特点	case

GET/MGET	通过指定三元组（_index、_type、_id）直接获取文档内容	实时读取，依赖于translog（5.x之前）或refresh机制（5.x之后）	实时性高，对写入速度有负面影响	`GET /my_index/_mget`
				`{`
				`  "docs": [`
				`  {`
				`  "_id": "1"`
				`  },`
				`  {`
				`  "_id": "2"`
				`  }`
				`  ]`
				`}`

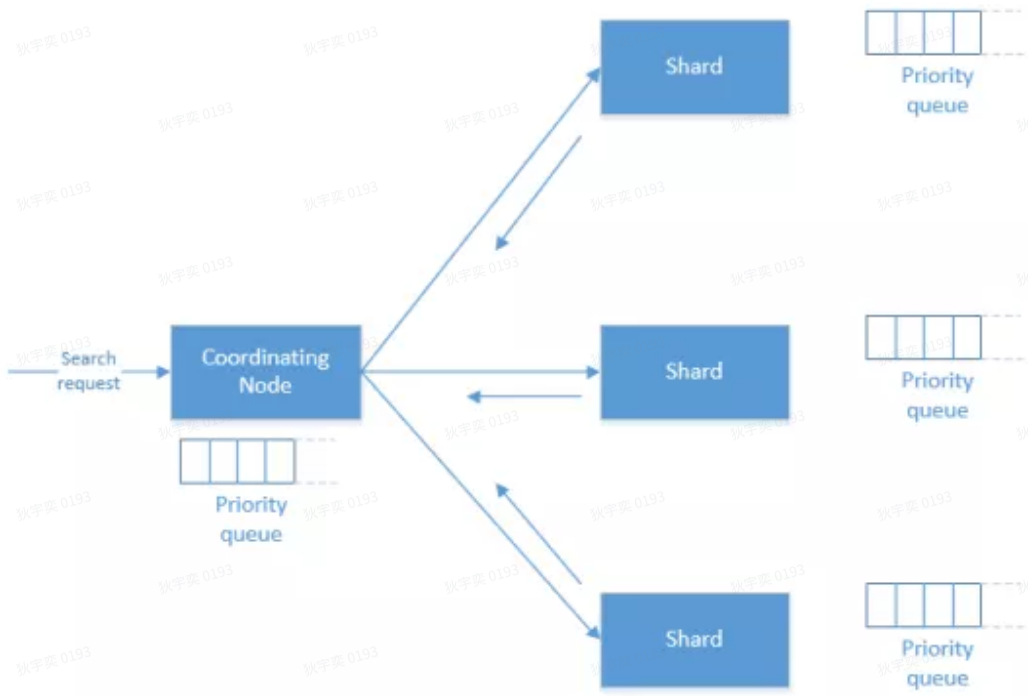
Search	根据关键词从倒排索引中获取内	倒排索引	需要复杂的查询模型，可能命中多个文档	`GET /my_index/_search` `{`

	容，不需要指定文档ID			

```
  "query": {
    "match": {
      "field1": "value1"
    }
  }
}
```

操作类型	描述	实现方式	影响因素和特点
GET/MGET	通过指定三元组（_index、_type、_id）直接获取文档内容。这些操作是针对单个或多个已知ID的文档的精确查询。	在Elasticsearch 5.x之前，依赖于translog实现实时读取。在5.x及之后的版本中，依赖于refresh机制。默认情况下，GET API是实时的，这意味着文档写入后可以立刻读取。	由于GET和MGET是实时操作，它们对系统的写入速度有负面影响，因为每次写入后必须刷新以确保数据可被实时读取。GET操作仅能对单个文档进行处理，指定的三元组唯一确定文档。
Search	根据关键词从倒排索引中获取内容。搜索操作不需要指定文档ID，而是根据查询条件在倒排索引中查找匹配的文档。搜索操作支持复杂的查询模型，允许在多个字段上进行关键词搜索、过滤和聚合。	搜索操作使用倒排索引进行查找，这种方式允许在大量文档中快速查找匹配关键词的文档。	搜索操作的复杂性高于GET，因为**搜索需要处理和解析查询条件**，并在倒排索引中查找匹配文档。搜索操作可能命中多个文档，因此搜索的性能和结果质量与查询模型和索引结构密切相关。
实时性比较	GET/MGET操作默认是实时的，这意味着写入文档后可以立即读取。但这种实时性仅限于GET和MGET操作，不包括搜索操作。	GET/MGET操作在5.x之前通过translog实现实时性，在5.x及之后通过refresh机制实现。	Search操作不是实时的，写入后的文档在下一次refresh之前不可见。这意味着写入操作和搜索操作之间存在一定的延迟。
写入速度影响	GET/MGET的实时读取特性对系统的写入速度有负面影响。因为为了确保文档可以实时读取，系统在写入后需要进行刷新操作。	写入后需要刷新以保证GET/MGET的实时性。	实时读取的要求会对写入性能产生负面影响，因为每次写入后都需要进行refresh操作。搜索操作对写入速度的影响较小，因为搜索不需要实时性，每次refresh后文档才会对搜索可见。

Elasticsearch的分布式搜索的运行机制称为 `Query-then-Fetch` 。具体分为Query和Fetch两个阶段



Query阶段

用户发出搜索请求到达ES节点。节点收到请求后，会以协调节点（Coordinating Node）的身份，在6个主副分片中随机选择3个分片，发送查询请求。

被选中的节点，进行排序（根据score值进行排序）。然后每个分片都返回 From+size 个排序后的文档id和排序值给协调节点。 注意这里返回的是**文档id**。

在数据节点响应query请求的时候已经完成了**Query慢查询的日志统计、查询结果的打分、纠错和聚合**

Fetch阶段

Coordinating节点将Query阶段从每个分片获取的排序的文档id列表重新进行排序，选取 From 到 From+size 个文档的id。**以 `multi get` 请求的方式，到相应的分片获取详细的文档数据**。

因为Elasticsearch在查询的时候不知道文档位于哪个分片，因此索引的所有分片都要参与搜索，然后协调节点将结果合并，在根据文档ID获取文档内容。例如现在有5个分片，需要查询匹配度Top10的数据，那么每个分片都要查询出当前分片的Top10的数据，协调节点将5×10个结果再次进行排序，返回Top10的结果给客户端。

Query-then-Fetch存在问题分为两方面，一个是性能问题，一个是相关性算分问题。

性能问题主要表现为深度分页的问题。

Elasticsearch数据是分片存储的，数据分布在多台机器上。有这样一个场景，如何获取前1000个文档？当获取从990-1000的文档时候，会在每个分片上面都先获取1000个文档，然后再由协调节点聚合所有分片的结果在排序选取前1000个文档。

页数越深，处理文档越多，占用内存越多，耗时越长。所以要尽量避免深度分页。

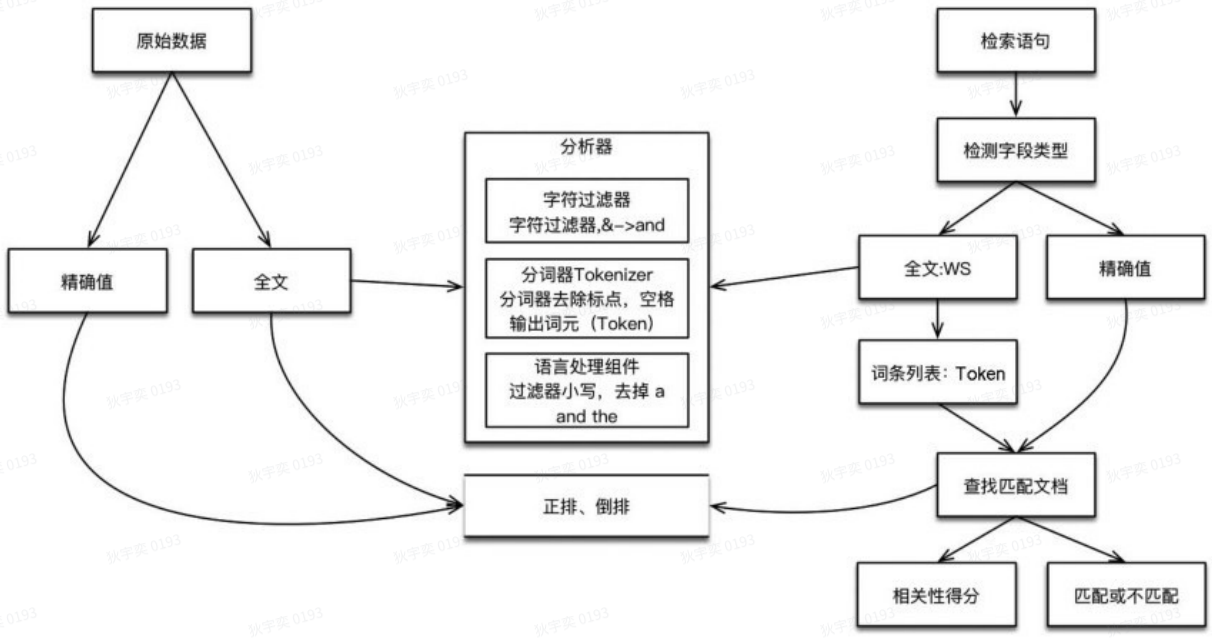
3.2.2 关于score计算

ES中的数据可以分为两类：精确值和全文。

· 精确值，比如日期和用户id、IP地址等。

· 全文，指文本内容，比如一条日志，或者邮件的内容。

这两种类型的数据在查询时是不同的：对精确值的比较是二进制的，查询要么匹配，要么不匹配；全文内容的查询无法给出“有”还是“没有”的结果，它只能找到结果是“看起来像”你要查询的东西，因此把查询结果按相似度排序，评分越高，相似度越大。



如果是全文数据，则对文本内容进行分析，这项工作在 ES 中由分析器实现。对上一步得到的Token做一些和语言相关的处理，例如，转为小写，以及将单词转换为词根的形式。语言处理组件输出的结果称为词（Term）。

分析完毕后，将分析器输出的词（Term）传递给索引组件，**生成倒排和正排索引**，再存储到文件系统中。

ES中正排索引（forward index）和倒排索引（inverted index）是两种不同的索引结构：

1. **正排索引（forward index）**：
   - 正排索引是文档到词项的映射，也可以理解为将文档按照顺序进行索引。每个文档包含了一个或多个词项及其相关信息，如位置、频率等。
   - **在搜索时，需要通过正排索引来定位文档**，然后再进行进一步处理。
2. **倒排索引（inverted index）**：
   - 倒排索引是词项到文档的映射，它是通过词项来查找包含该词项的所有文档。
   - 在倒排索引中，每个词项都会记录包含该词项的文档列表，以及文档内该词项出现的位置等信息。
   - 当用户执行搜索时，**搜索引擎会首先查找倒排索引，确定包含关键词的文档列表，然后进一步筛选和排序结果。**

因此，正排索引与倒排索引的主要区别在于数据的组织方式。正排索引是从文档到词项的映射，而倒排索引是从词项到文档的映射。在Elasticsearch等搜索引擎中，倒排索引被广泛应用以加快搜索速度和改善搜索性能。

3.2.3 关于分析器

它将输入文本转换为可以索引和搜索的词条（tokens）。分析器在索引和搜索两个阶段都会用到，确保输入文本和查询语句都经过一致的处理。

假设我们有一个自定义分析器，它包含以下组件：

- **字符过滤器**：`html_strip`
- **分词器**：`standard`
- **词条过滤器**：`lowercase` 和 `stop`

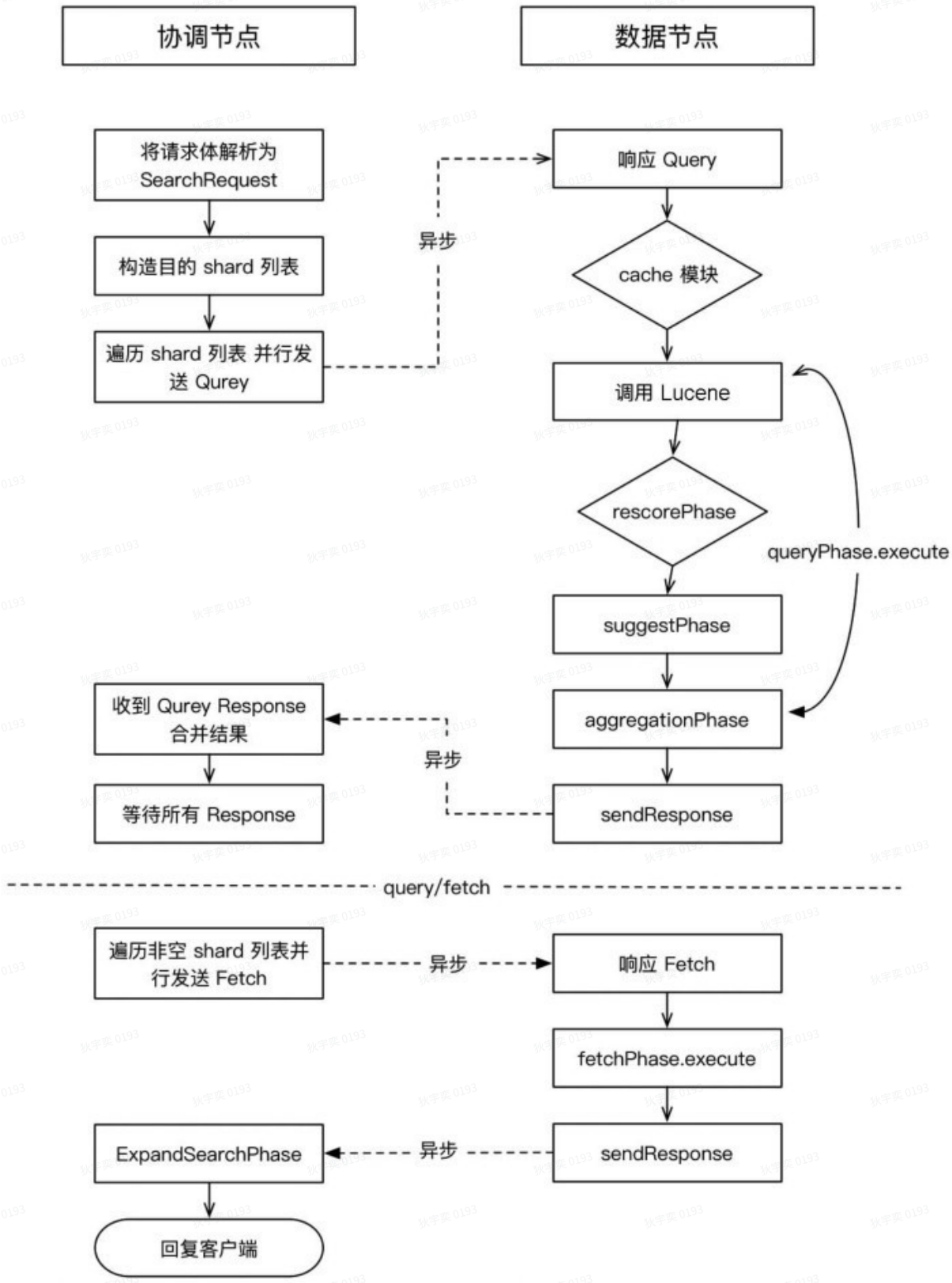
输入文本：“ Elasticsearch is a distributed, RESTful search and analytics engine. ”

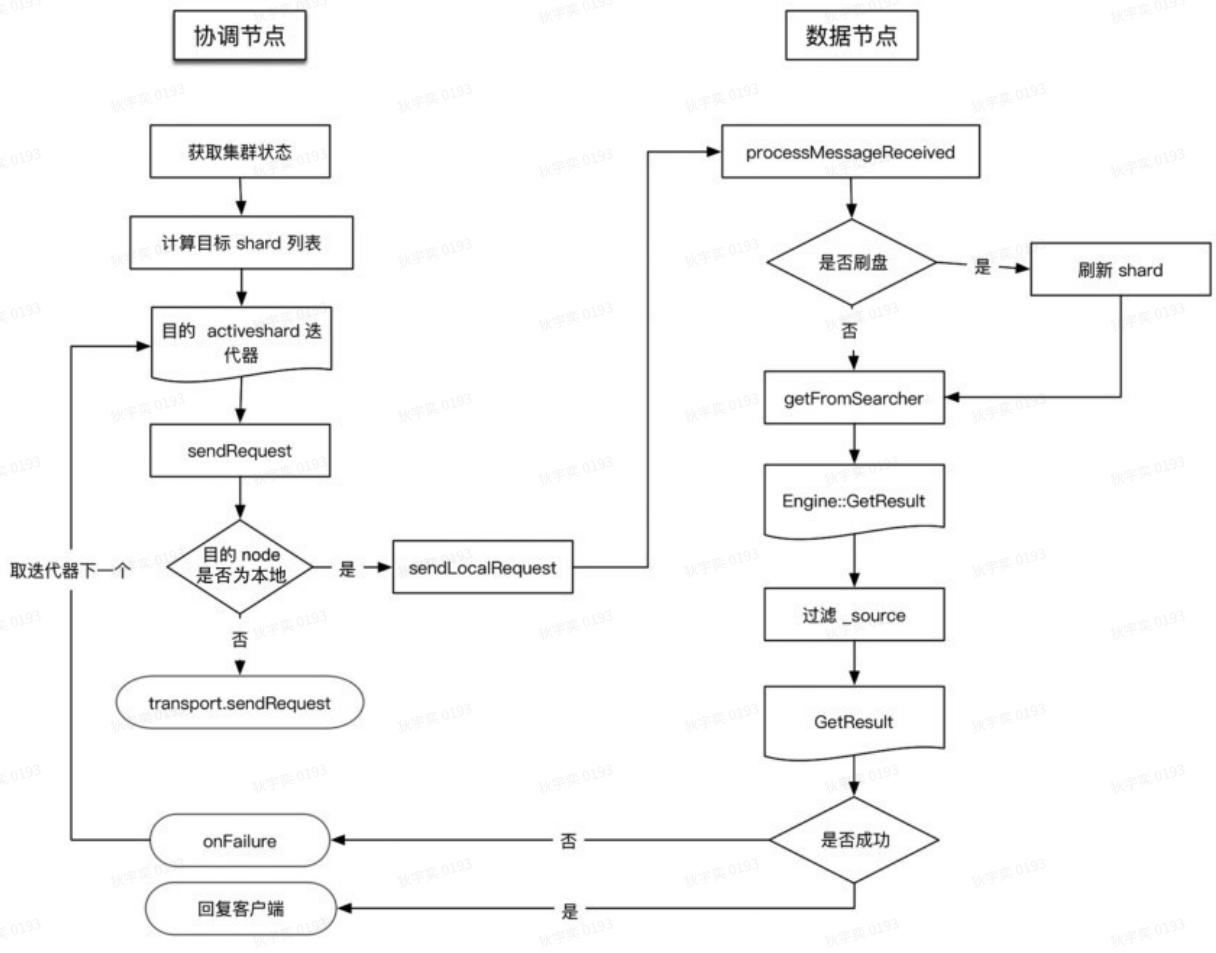
1. **字符过滤器（html_strip）**：
   - 输入：`<p>Elasticsearch is a distributed, RESTful search and analytics engine.</p>`

- 输出：`Elasticsearch is a distributed, RESTful search and analytics engine.`

2. **分词器（standard）**：
   - 输入：`Elasticsearch is a distributed, RESTful search and analytics engine.`
   - 输出：`["Elasticsearch", "is", "a", "distributed", "RESTful", "search", "and", "analytics", "engine"]`
3. **词条过滤器（lowercase 和 stop）**：
   - lowercase 输入：`["Elasticsearch", "is", "a", "distributed", "RESTful", "search", "and", "analytics", "engine"]`
   - lowercase 输出：`["elasticsearch", "is", "a", "distributed", "restful", "search", "and", "analytics", "engine"]`
   - stop 输入：`["elasticsearch", "is", "a", "distributed", "restful", "search", "and", "analytics", "engine"]`
   - stop 输出：`["elasticsearch", "distributed", "restful", "search", "analytics", "engine"]`（去掉了停用词“is”、“a”、“and”）

4. 源码


协调节点
数据节点
将请求体解析为 SearchRequest
构造目的 shard 列表
遍历 shard 列表 并行发送 Qurey
异步
响应 Query
cache 模块
调用 Lucene
rescorePhase
queryPhase.execute
suggestPhase
aggregationPhase
收到 Qurey Response 合并结果
等待所有 Response
异步
sendResponse
query/fetch
遍历非空 shard 列表并行发送 Fetch
异步
响应 Fetch
fetchPhase.execute
ExpandSearchPhase
异步
sendResponse
回复客户端


协调节点
数据节点
获取集群状态
计算目标 shard 列表
目的 activeshard 迭代器
sendRequest
目的 node 是否为本地
是
sendLocalRequest
取迭代器下一个
否
transport.sendRequest
processMessageReceived
是否刷盘
是
刷新 shard
否
getFromSearcher
Engine::GetResult
过滤 _source
GetResult
是否成功
否
onFailure
是
回复客户端

数据存储与管理

1. 磁盘存储的文件格式

- segment : lucene内部的数据是由一个个segment组成的，写入lucene的数据并不直接落盘，而是先写在内存中，经过了refresh间隔，lucene才将该时间段写入的全部数据refresh成一个segment，segment多了之后会进行merge成更大的segment。lucene查询时会遍历每个segment完成。由于lucene* 写入的数据是在内存中完成，所以写入效率非常高。但是也存在丢失数据的风险，所以Elasticsearch基于此现象实现了translog，只有在segment数据落盘后，Elasticsearch才会删除对应的translog。
- doc : doc表示lucene中的一条记录
- field： field表示记录中的字段概念，一个doc由若干个field组成。
- term： term是lucene中索引的最小单位，某个field对应的内容如果是全文检索类型，会将内容进行分词，分词的结果就是由term组成的。如果是不分词的字段，那么该字段的内容就是一个term。
- 倒排索引（inverted index）: lucene索引的通用叫法，即实现了term到doc list的映射。
- 正排数据：搜索引擎的通用叫法，即原始数据，可以理解为一个doc list。
- docvalues :Elasticsearch中的列式存储的名称，Elasticsearch除了**存储原始存储、倒排索引**，还存储了一份docvalues，用作分析和排序。

每个segment会包含如下的文件。

Name	Extension	Brief Description
Segment Info	.si	segment的元数据文件
Compound File	.cfs, .cfe	一个segment包含了如下表的各个文件，为减少打开文件的数量，在segment的时候，segment的所有文件内容都保存在cfs文件中，cfe文件保存了lucene各文件在cfs文件中的位置信息
Fields	.fnm	保存了fields的相关信息
Field Index	.fdx	正排存储文件的元数据信息
Field Data	.fdt	存储了正排存储数据，写入的原文存储在这
Term Dictionary	.tim	倒排索引字典数据文件
Term Index	.tip	倒排索引文件，存储了所有的倒排索引数据
Frequencies	.doc	保存了每个term的doc id列表和term在doc中的词频
Positions	.pos	Stores position information about where a term occurs in the index 全文索引的字段，会有该文件，保存了term在doc中的位置
Payloads	.pay	Stores additional per-position metadata information such as character offsets and user payloads 全文索引的字段，使用了一些像payloads的高粒性会有该文件，保存了term在doc中的一些高粒性
Norms	.nvd, .nvm	文件保存索引字段加权数据

在fp中，es的数据目录放在/usr/share/el…

狄宇奕 6月3日 20:37

@冯竣倬 这里有验证过，97G的路人 json 数据，约 30G 的磁盘占用

Per-Document Values	.dvd, .dvm	lucene的docvalues文件，即数据的列式存储，用作聚合和排序
Term Vector Data	.tvx, .tvd, .tvf	Stores offset into the document data file 保存索引字段的矢量信息，用在对term进行高亮，计算文档相关性中使用
Live Documents	.liv	记录了segment中删除的doc

在fp中，es的数据目录放在/usr/share/elasticsearch/data，这里我们以176上的master节点为例，这里我们选择track_meta19841作为观察对象，大概3kw的过人量。

使用track_result/fetch_total拉取100条数据，约347.51k，对应这么多docs，实际的track_meta的json数据占用量为：102,566,186K = 97G。存储下来实际耗费30G，应该是有数据压缩.

health	status	index	uuid	pri	rep	docs.count	docs.deleted	store.size	pri.store.size
green	open	track_meta19841	qIvBoK1uTsWYvBK05Nwb4g	3	0	29514600	374015	30gb	30gb

这是指示Elasticsearch索引状态的表格。其中包含了相关信息，如健康状况、状态、索引名、UUID、主分片数、副本数、文档数量、已删除文档数量、存储大小和主分片存储大小。

```
/usr/share/elasticsearch/data/nodes/0/indices/qIvBoK1uTsWYvBK05Nwb4g/0/index
```

可以观察到单个shard上的index数据空间10G，乘上分片数3，正好等于store.size

```
/usr/share/elasticsearch/data/nodes/0/indices/qIvBoK1uTsWYvBK05Nwb4g/2
du -h --max-depth=1 | sort -h


8.0K    ./_state


8.0K    ./translog


10G     .


10G     ./index
```

后缀	Total size	占比	作用	命令
fdm	12k			find . -type f -name "*.fdm" -exec du -ch {} +
fdt	8.6G	86%	存储了正排存储数据，写入的原文存储在这	find . -type f -name "*.fdt" -exec du -ch {} +
fdx	180k			find . -type f -name "*.fdx" -exec du -ch {} +
fnm	48k			find . -type f -name "*.fnm" -exec du -ch {} +
kdd	74M			find . -type f -name "*.kdd" -exec du -ch {} +
kdi	264K			find . -type f -name "*.kdi" -exec du -ch {} +
kdm	12k			find . -type f -name "*.kdm" -exec du -ch {} +
dvd	211M	2%	lucene的docvalues文件，即数据的列式存储，用作聚合和排序	find . -type f -name "*.dvd" -exec du -ch {} +
dvm	36K			find . -type f -name "*.dvm" -exec du -ch {} +
si	100K			find . -type f -name "*.si" -exec du -ch {} +
cfe	88K			find . -type f -name "*.cfe" -exec du -ch {} +
cfs	938M	9.1%	segment的所有文件内容都保存在cfs文件中	find . -type f -name "*.cfs" -exec du -ch {} +
segments_2x	4k			find . -type f -name "segments_2x" -exec du -ch {} +
total	10G			
raw_data total	97 / 3 = 32G			

```
data/
└── nodes/  // 所有节点的数据目录
    └── 0/  // 特定节点的ID，每个节点都有一个唯一的ID
        ├── node.lock  // 锁定文件，防止多个进程同时访问同一个节点数据
        ├── snapshot_cache/  // 快照缓存目录，存储快照数据
        │   ├── segments_f  // 快照中使用的段文件
        │   └── write.lock  // 写入锁定文件
        ├── _state/  // 节点状态目录
        │   ├── _cc.fdm  // 索引的字段映射文件
        │   ├── _cc.fdt  // 索引的字段数据文件
        │   ├── _cc.fdx  // 索引的字段索引文件
        │   ├── _cc.fnm  // 索引的字段名称映射文件
        │   ├── _cc_Lucene84_0.doc  // Lucene 8.4 版本的文档存储文件
```

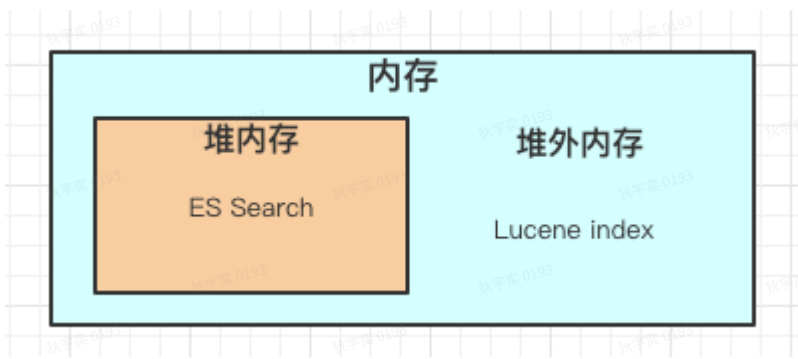
```
        │   ├── _cc_Lucene84_0.tim  // Lucene 8.4 版本的术语字典文件
        │   ├── _cc_Lucene84_0.tip  // Lucene 8.4 版本的术语索引文件
        │   ├── _cc_Lucene84_0.tmd  // Lucene 8.4 版本的术语字典元数据文件
        │   ├── _cc.si  // 索引的存储信息文件
        │   ├── manifest-14.st  // 索引的清单文件
        │   ├── node-14.st  // 节点状态文件
        │   ├── segments_9a  // 索引的段文件
        │   └── write.lock  // 写入锁定文件
        └── indices/  // 存储索引数据的目录
            └── qIvBoK1uTsWYvBK05Nwb4g/  // 索引的唯一标识符
                ├── _state/  // 索引状态目录
                │   └── state-9.st  // 索引状态文件
                └── 2/  // 索引的分片ID
                    ├── index/  // 分片的索引文件
                    │   // 以下是分片中包含的不同类型的文件：
                    // 文件名以 _ 开头的通常是Lucene的内部文件，用于存储索引数据和元数据。
                    // 例如：
                    // _15b4.fdm, _15b4.fdt, _15b4.fdx, _15b4.fnm - 索引的字段映射和数据文件
                    // _15b4.kdd, _15b4.kdi, _15b4.kdm - 索引的Kryo压缩数据文件
                    // _15b4_Lucene80_0.dvd, _15b4_Lucene80_0.dvm - Lucene 8.0 版本的文档存储和元数据文件
                    // _15b4.si - 索引的存储信息文件
                    // _19oo.cfe, _19oo.cfs, _19oo.si - 索引的压缩文件和存储信息文件
                    │   // 其他文件遵循类似的命名约定，代表不同的数据和索引结构
                    ├── _state/  // 分片状态目录
                    │   ├── retention-leases-2834.st  // 保留租约状态文件
                    │   └── state-2.st  // 分片状态文件
                    └── translog/  // 事务日志目录
                        ├── translog-772.tlog  // 事务日志文件，用于记录操作以便在故障时恢复
                        └── translog.ckp  // 事务日志检查点文件，标记已持久化的操作
```

2. 内存机制

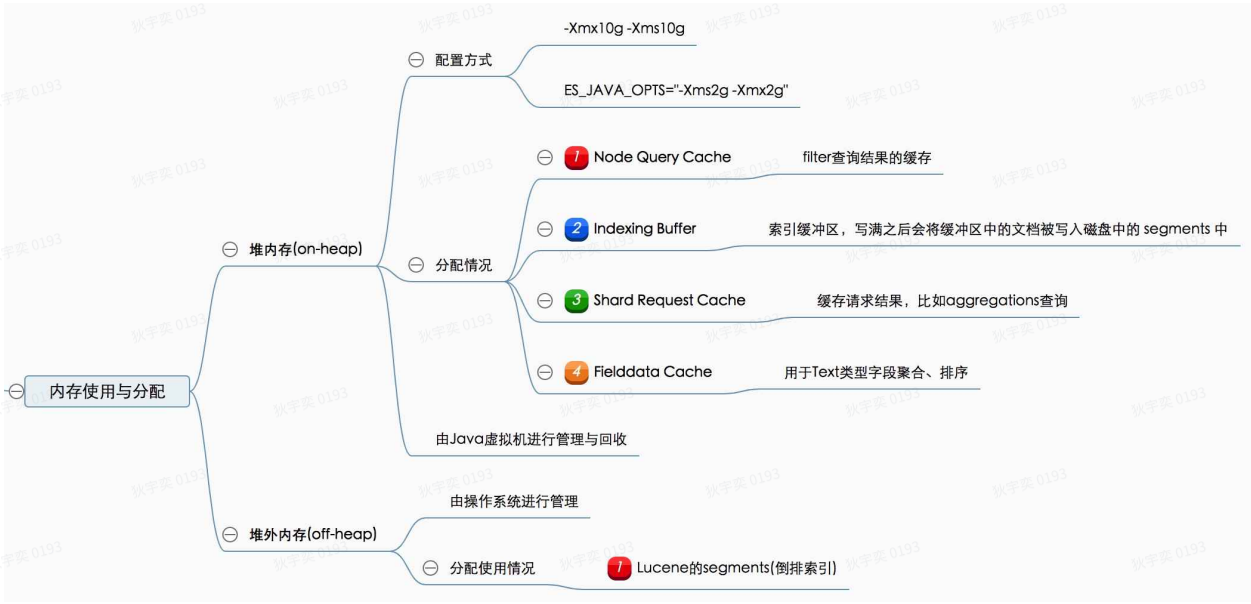
Elasticsearch 内存主要分为两部分

系统内存 也叫 **堆外内存：直接收操作系统管理的内存属于堆外内存（off-heap memory），通过把内存对象分配在堆外内存中，可以保持一个较小的堆，可以减少垃圾回收对应用的影响**
JVM 内存 也叫 **堆内存：**

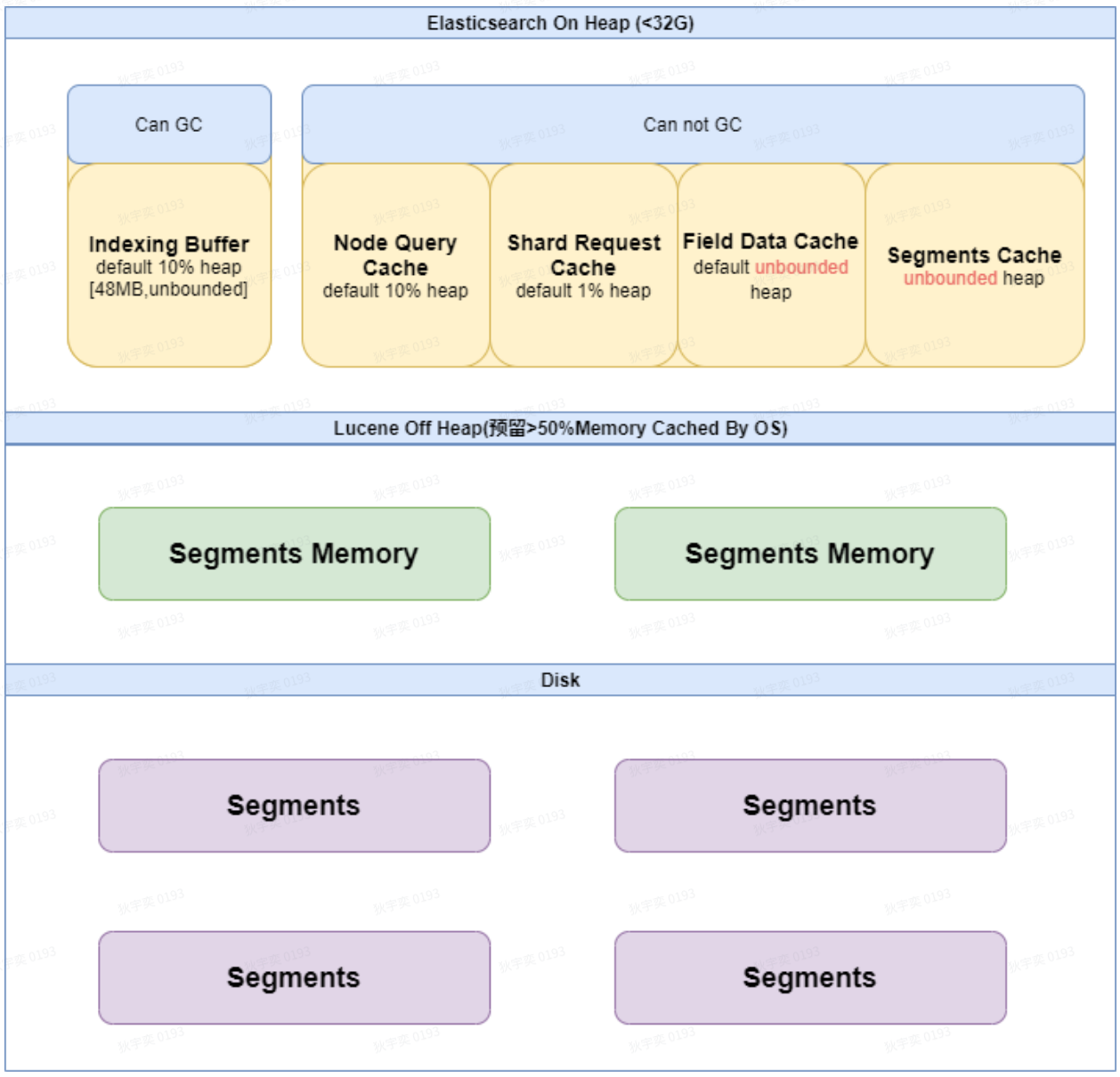
系统内存主要给**底层 Lucene 使用**。
JVM 内存提供给 **ES 查询时各类缓存使用**。



具体的思维导图见：



这里



2.1 jvm内存

ES的JVM heap按使用场景分为**可GC部分**和**常驻部分**。 可GC部分内存会随着GC操作而被回收； 常驻部分不会被GC，通常使用LRU策略来进行淘汰；

下面整理

内存管理方式	类目	层级	默认占比	是否常驻	淘汰策略(在控制大小情况下)	作用	控制参数	其他补充

非GC	query cache	node-level	**10%**	是	LRU	能够被当前节点的所有shard所共享，用于缓存filter的查询结果	• **indices.queries.cache.size** 用来控制缓存的内存大小，默认是10%，属于节点级别配置。支持百分数，也支持大小精确值。该配置属于静态配置，更改配置需要重启节点 • **index.queries.cache.enabled** 用来控制具体索引是否启用缓存，默认是开启的。属于index级别配置。只用在索引创建时候或者关闭的索引上设置 • **indices.queries.cache.all_segments** 用于控制是否在所有Segment上启用缓存，默认是false,即不会对文档数小于100000或者小于整个索引大小的3%的Segment进行缓存	
request cache	shard-level	**1%**	是	LRU	当进行索引搜索时候，每个相关的shard在本地执行搜索，并将其本地结果返回给协调节点(Coordinating Node)，协调节点将这些分片级别的结果合并为一个“全	• **indices.requests.cache.size** 设置请求缓存大小，默认是1%。该配置是静态配置。 • 索引的请求缓存默认是开启的，该配置可以动态开启或关闭 `1 PU`	我们可以通过以`_stat` API查看分片请求缓存大小 `1`	

局”结果集。

- 仅缓存 `size=0` 的搜索请求，即不反悔具体文档内容，**只返回统计信息**
- 缓存 hits.total
- 缓存 aggregations
- 缓存 suggestions

```
PUT /my-index/_settings
{
  "index.requests.cache.enable": true
}
```

- 需要在那些需要缓存的搜索请求上加上 `request_cache=true` 这个参数才能使我们的查询请求被缓存

```
curl 'localhost:9200/index_name/_search?request_cach
```

<table>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>
<pre><code>e
=
t
r
u
e
'
-
d
'
</code></pre>
</td>
<td></td>
</tr>
<tr>
<td>fielddata cache</td>
<td>segment-level</td>
<td>无限制(ES默认禁用)</td>
<td>是</td>
<td>LRU</td>
<td>

Text类型的字段，如果要对其进行聚合和排序，则需要打开字段的 <code>Fileddata</code> 属性
当对该字段进行聚合，排序时候，ES会把Field Data中加载到内存中，构建成 <code>Field data Cache</code>
整个segment生命周期内都存在，是无法GC的。

</td>
<td>

indices.fielddata.cache.size
用来控制索引的fileddata cache大小。默认是 <code>unbounded</code> 。支持百分数，也支持大小精确值。该参数是静态配置类型。
大小默认是没有限制的原因是fielddata不是临时性的cache，它能够极大地提升性能，而且构建fielddata又比较耗时的操作，所以需要一直cache。
如果没有足够的内存保存fielddata时，Elastisearch会不断地从磁盘加载数据到内存，并剔除掉旧的内存数据。剔除操作会造成严重的磁盘I/O，并且引发大量的GC，会严重影响Elastisearch的性能。
indices.breaker.fielddata.limit

</td>
<td>
我们可以通过以下API查看fielddata大小：
<pre><code>1
</code></pre>
</td>
</tr>
</table>

		用来设置Fielddata断路器限制大小，**默认是JVM heap大小的40%**。当加载的内存中fielddata数据超过该限制大小，会发生异常，目的为了防止发生JVM OOM。该值可以动态设置，建议配置该项的值： • **indices.breaker.fielddata.overhead** 用来设置fielddata过载值，默认值是**1.03**	2 3

4
5

	segment memory		无限制(需要监控)	是	无	segments FST(Finite State Transducer)数据的缓存，为倒排索引的二级索引。 FST 永驻堆内内存，无法被 GC 回收。该部分内存无法设置大小，长期占用 50% ~ 70% 的堆内存，只能通过 **delete index，close index以及force-merge index**释放内存。**从7.3版本开始，FST由堆外内存来存储**	不能通过参数控制	
GC	Index buffer	node-level	**10%**	否	GC	当文档创建时候，会先保存在IndexBuffer中，然后每隔 `index.refresh_interval` 或者索引缓存	• **indices.memory.index_buffer_size** 用于控制 index buffer 大小，属于节点级别配置。默认是10%，即允许分配到	

						满的时候进行refresh操作，将文档的段存在系统缓存中，此时文档才能搜索到。索引缓冲是可以通过GC被反复利用的。	整个堆大小的10%。支持百分数，也支持大小精确值。 • **indices.memory.min_index_buffer_size** 用于控制index buffer大小的最小值，当index_buffer_size为百分数时候，才生效。默认是`48mb` • **indices.memory.max_index_buffer_size** 用于控制index buffer大小的最大值，当index_buffer_size为百分数时候，才生效。默认是**unbounded**（无上限的）	

2.2 堆外内存

服务器总内存除过给jvm配置的其余都给了lucene，占用page cache内存，page cache保存对文件数据segment（segment是shard数据的一部分内容）的缓存。缓存索引的部分数据，用于提升查询响应（segment越大查询的名字率就越高查询性能越高，这也就是es查询响应ms级和s的主要差别所在就是内存）

服务器执行：free -g可查看内存使用，es节点只有es服务，基本cache就是缓存的segment。

我们可以通过`cat segments`查看索引的segment使用内存的情况：

```
1 GET /_cat/segments?v
```

清除全部的缓存：

```
1 POST /_cache/clear
```

清除特定索引的缓存：

```
1 POST /my_index/_cache/clear
```

```
POST /my_index1,my_index2/_cache/clear
```

清除特定类型缓存：

通过设置 `fielddata`，`query`，`request` 参数为 `true` 来清除特定类型的缓存

```
POST /my-index/_cache/clear?fielddata=true
POST /my-index/_cache/clear?query=true
POST /my-index/_cache/clear?request=true
```

3. 节点负载观察

节点负载详解

```
curl -s "10.108.138.231:9200/_cat/nodes?h=ip,hp,rp,rc,rm,fm,sm,sc,qcm,sqti,dt,du&v"
ip:node.ip
cpu：cpu使用率
hp：堆内存使用
rp：ram.percent,总内存使用量百分比，保持在75%以下最好
rc：Used total memory,使用总内存大小
rm：ram.max：总内存大小
fm：fielsdata.sizeMemoey:字段缓存
sm：segment.memory：segment使用的内存
sc：segments.count：segment的数量
qcm：query_cache.memory_size：查询缓存的内存使用
sqti：search.query_time:查询总用时
dt：disk.total：所有磁盘空间
du：disk.used：使用的磁盘空间

```

这里查看一下86.176上的节点数据存储相关的信息

好的，以下是将名称转换为中文后的表格：

IP	堆内存使用	总内存使用量百分比	使用总内存大小	总内存大小	字段缓存	Segment使用的内存	Segment的数量	查询缓存的内存使用	查询总用时	所有磁盘空间	使用的磁盘空间
18.240.0.240	40	100	9.9gb	10gb	0b	9.7mb	1429	123.7mb	11.2h	12.2tb	5.7tb
18.240.0.242	70	100	9.9gb	10gb	0b	9.4mb	1411	112.2mb	10.8h	12.2tb	5.7tb
18.240.0.241	37	100	9.9gb	10gb	0b	9.6mb	1427	153.3mb	10.4h	12.2tb	5.7tb

接下来使用监视 API 查看堆内存的详细信息

```
curl -s "10.108.138.231:9200/_nodes/stats/jvm?human"
```

结果如下：https://jsonhero.io/j/Pw9NwMuLR613?path=%24.nodes.26Ife_H_RQG9AOqf0IXO3A.jvm

内存类型	使用量	最大可用量	使用百分比	峰值使用量
堆内存	3.9GB	8GB	49%	-
非堆内存	197.4MB	204.5MB	-	-
Young代内存	1.6GB	-	-	4.7GB
Old代内存	2.2GB	8GB	-	2.4GB
Survivor代内存	87.3MB	-	-	272.9MB

，我们可以用结果向GPT询问情况是否正常

4. 数据恢复流程(TODO)

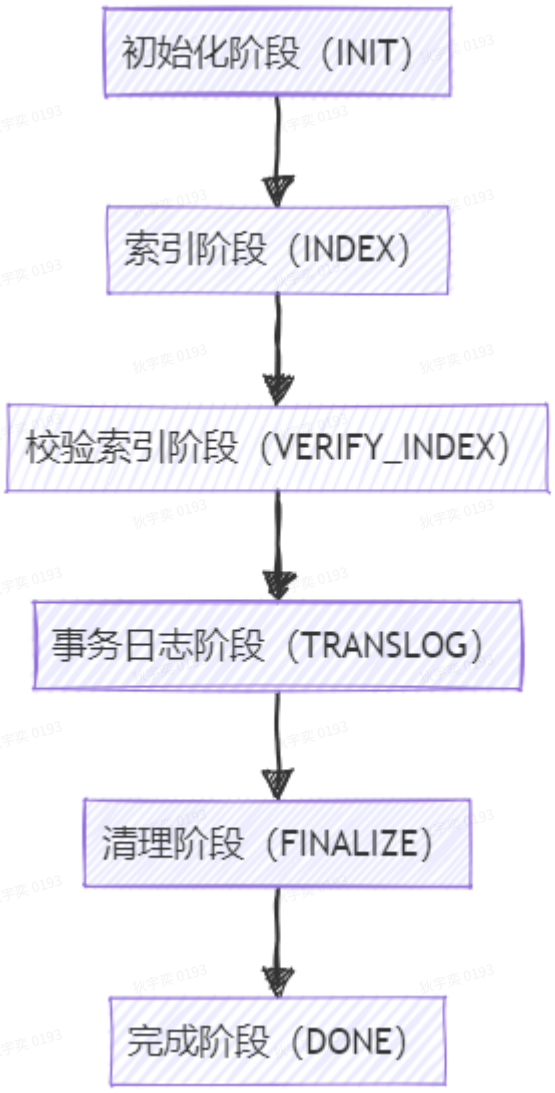
Elasticsearch的数据恢复流程是确保数据完整性和集群稳定性的重要机制。

该流程主要发生在集群启动或分片重新分配时。以下是ES数据恢复流程的概述

1. **初始化阶段（INIT）：**
    - 此阶段是恢复过程的开始，集群尚未开始数据恢复。在这个阶段，Elasticsearch准备恢复所需的资源和状态。
    - 准备恢复所需的配置和参数。
    - 初始化相关的数据结构和内存缓存。
2. **索引阶段（INDEX）：**
    - 在索引阶段，Elasticsearch从本地存储或其他节点复制Lucene文件以恢复索引数据。
    - 检查并复制数据分片以确保数据的完整性和一致性。
3. **校验索引阶段（VERIFY_INDEX）（可选）：**
    - 如果配置了`index.shard.check_on_startup`，则会执行此检查，用于检查索引文件是否损坏。
    - 如果发现损坏的索引文件，可能会触发修复或重建索引的操作。
        - 如果索引文件的损坏程度较小，并且仅影响了部分数据，Elasticsearch可能会尝试修复索引。
            - 检测索引文件的损坏部分。
            - 尝试修复或恢复损坏的部分数据。
            - 如果修复成功，则继续索引恢复流程。
            - 如果修复失败，则可能需要手动介入进行进一步的修复或重建索引。
        - 如果索引文件的损坏程度较为严重，或者索引修复失败，Elasticsearch可能会选择重建索引
            - 删除损坏的索引文件。
            - 根据原始数据重新构建索引。
            - 重新索引数据并建立新的索引文件。
            - 更新集群状态以反映新的索引状态。
4. **事务日志阶段（TRANSLOG）：**

- 在此阶段，启动索引引擎并开始重放事务日志（translog）。
- 重放事务日志是为了重建那些已经写入但尚未被flush到磁盘的数据，确保数据的完整性和一致性。

5. **清理阶段（FINALIZE）**：
   - 在这个阶段，Elasticsearch执行一些清理工作，以确保所有数据都已写入到segment中，并执行一些必要的操作以准备集群和索引的正常运行。
   - 刷新索引以确保所有数据都写入到了segment中。
   - 打开GcDeletes以允许删除操作。
6. **完成阶段（DONE）**：
   - 恢复过程结束，更新集群状态以反映分片的新状态。
   - 此时，集群已经成功完成数据恢复，并且可以正常对外提供服务。



可以通过下面的操作进行手动数据备份与恢复：https://help.aliyun.com/zh/es/user-guide/create-manual-snapshots-and-restore-data-from-manual-snapshots?spm=a2c4g.11186623.0.0.66493087uXyj5U

集群管理与优化

1. 集群监控

一个 Elasticsearch 集群至少包括一个节点和一个索引。或者它可能有一百个数据节点、三个单独的主节点，以及一小打客户端节点——这些共同操作一千个索引（以及上万个分片）。

不管集群扩展到多大规模，你都会想要一个快速获取集群状态的途径。Cluster Health API 充当的就是这个角色。

```
GET _cluster/health
```

response: 返回集群概览信息

```
{
   "cluster_name": "elasticsearch_zach",
   "status": "green",
   "timed_out": false,
   "number_of_nodes": 1,
   "number_of_data_nodes": 1,
   "active_primary_shards": 10,
   "active_shards": 10,
   "relocating_shards": 0,
   "initializing_shards": 0,
   "unassigned_shards": 0
}
```

响应信息中最重要的一个指标 status 字段。状态可能是下列三个值之一：

- **green**

所有的主分片和副本分片都已分配。你的集群是 100% 可用的。

- **yellow**

所有的主分片已经分片了，但至少还有一个副本是缺失的。不会有数据丢失，所以搜索结果依然是完整的。不过，你的高可用性在某种程度上被弱化。如果 更多的 分片消失，你就会丢数据了。把 yellow 想象成一个需要及时调查的警告。

- **red**

至少一个主分片（以及它的全部副本）都在缺失中。这意味着你在缺少数据：搜索只能返回部分数据，而分配到这个分片上的写入请求会返回一个异常。

其他信息输出

指标	描述
number_of_nodes	集群中的节点数量。
number_of_data_nodes	数据节点的数量。
active_primary_shards	主分片的数量，涵盖了所有索引的汇总值。
active_shards	所有分片的数量，包括主分片和副本分片的汇总值。
relocating_shards	正在从一个节点迁往其他节点的分片数量。通常情况下为 0，当集群发现不均衡时可能会增加。
initializing_shards	刚创建的分片数量。通常是临时事件，分片不应该长期停留在 initializing 状态。
unassigned_shards	已经存在于集群状态中但找不到的分片数量。通常来源于未分配的副本，可能导致集群处于 red 状态。

如果发现status 变为red ，我们就需要进一步排查具体原因；定位具体那个分片的索引有问题；执行以下API

```
GET _cluster/health?level=indices
```

response 有关每个索引的细节（状态、分片数、未分配分片数等等）；

```
{
   "cluster_name": "elasticsearch_zach",
   "status": "red",
   "timed_out": false,
   "number_of_nodes": 8,
   "number_of_data_nodes": 8,
   "active_primary_shards": 90,
   "active_shards": 180,
   "relocating_shards": 0,
   "initializing_shards": 0,
   "unassigned_shards": 20
   "indices": {
      "v_1": {
         "status": "green",
         "number_of_shards": 10,
         "number_of_replicas": 1,
         "active_primary_shards": 10,
         "active_shards": 20,
         "relocating_shards": 0,
         "initializing_shards": 0,
         "unassigned_shards": 0
      },
      "v_2": {
         "status": "red",
         "number_of_shards": 10,
         "number_of_replicas": 1,
         "active_primary_shards": 0,
         "active_shards": 0,
         "relocating_shards": 0,
         "initializing_shards": 0,
         "unassigned_shards": 20
      },
      "v_3": {
         "status": "green",
         "number_of_shards": 10,
         "number_of_replicas": 1,
         "active_primary_shards": 10,
         "active_shards": 20,
         "relocating_shards": 0,
         "initializing_shards": 0,
         "unassigned_shards": 0
      },
      ....
   }
}
```

```
GET _cluster/health?level=shards
```

`shards` 选项会提供一个详细得多的输出，列出每个索引里每个分片的状态和位置。这个输出有时候很有用，但是由于太过详细会比较难用。

2. 监控关键指标

ElasticSearch 提供了大量的指标，可以用于监控各类故障现象：

- 搜索性能指标
- 索引性能指标
- 内存使用和垃圾回收指标
- 主机级别网络系统指标
- 集群健康和节点可用性指标
- 资源饱和度和错误

2.1 搜索性能指标

序号	指标	类型	详情
1	elasticsearch_indices_search_query_total	counter	查询总数 吞吐量
2	elasticsearch_indices_search_query_time_seconds	counter	查询总时间 性能
3	elasticsearch_indices_search_fetch_total	counter	提取总数
4	elasticsearch_indices_search_fetch_time_seconds	counter	花费在提取上的总时间
5	elasticsearch_indices_get_time_seconds	counter	GET请求总时间
6	elasticsearch_indices_get_missing_total	counter	丢失的文件的GET请求总数
7	elasticsearch_indices_get_missing_time_seconds	counter	花费在文档丢失的GET请求上的总时间
8	elasticsearch_indices_get_exists_time_seconds	counter	花费在文档存在的GET请求上的总时间
9	elasticsearch_indices_get_exists_total	counter	存在的文件的GET请求总数
10	elasticsearch_indices_get_total	counter	GET请求总次数

2.2 索引性能指标

对于一个写入较为频繁的系统，refresh 和 flush 操作相关的指标较为重要，merge 相关的指标也需要关注，同时整个写入的耗时，搜索文档总数也需要关注，故而需要关注的写入相关指标为：

序号	指标	类型	详情
1	elasticsearch_indices_indexing_index_total	counter	被索引的文档总数
2	elasticsearch_indices_indexing_index_time_seconds_total	counter	索引文档花费的总时间

3	elasticsearch_indices_refresh_total	counter	索引refresh的总数
4	elasticsearch_indices_refresh_time_seconds_total	counter	refresh索引总共花费的时间
5	elasticsearch_indices_flush_total	counter	flush索引到磁盘的总数
6	elasticsearch_indices_flush_time_seconds	counter	flush索引到磁盘的总花费时间
7	elasticsearch_indices_merges_total	counter	Total merges（merge操作总次数）
8	elasticsearch_indices_merges_total_time_seconds_total	counter	索引merge操作总花费时间
9	elasticsearch_indices_merges_Docs_total	counter	merge的文档总数
10	elasticsearch_indices_merges_total_size_bytes_total	counter	merge的数据量合计
11	elasticsearch_indices_indexing_delete_total	counter	索引的文件删除总数
12	elasticsearch_indices_indexing_delete_time_seconds_total	counter	索引的文件删除总时间

2.3 内存使用和垃圾回收指标

ES 是使用 Java 进行编写，运行在 JVM 之上，故而 ES 会以两种方式使用节点上的全部可用内存：JVM 堆和文件系统缓存(file system cache)，因此，在 ES 运行期间，整个 JVM 的垃圾回收持续时间和频率将很值得监控。

序号	指标	类型	详情
1	elasticsearch_jvm_gc_collection_seconds_sum	counte	JVM GC垃圾回收时间
2	elasticsearch_jvm_gc_collection_seconds_count	counter	JVM GC 垃圾搜集数
3	elasticsearch_jvm_memory_committed_bytes	gauge	JVM最大使用内存限制
4	elasticsearch_jvm_memory_max_bytes	gauge	配置的最大jvm值
5	elasticsearch_jvm_memory_pool_max_bytes	counter	JVM内存最大池数
6	elasticsearch_jvm_memory_pool_peak_max_bytes	counter	最大的JVM内存峰值
7	elasticsearch_jvm_memory_pool_peak_used_bytes	counter	池使用的JVM内存峰值
8	elasticsearch_jvm_memory_pool_used_bytes	gauge	目前使用的JVM内存池

9	elasticsearch_jvm_memory_used_bytes	gauge	

2.4 主机级别网络系统指标

主机级别资源及网络使用情况也需要关注。

序号	指标	类型
1	elasticsearch_process_cpu_percent	gauge
2	elasticsearch_filesystem_data_free_bytes	gauge
3	elasticsearch_process_open_files_count	gauge
4	elasticsearch_transport_rx_packets_total	counter
5	elasticsearch_transport_tx_packets_total	counter

2.5 集群健康和节点可用性指标

ES 是按集群部署的，需要关注集群相关指标。

序号	指标	类型	详情
1	elasticsearch_cluster_health_status	gauge	集群状态，gre 常运行）red
2	elasticsearch_cluster_health_number_of_data_nodes	gauge	node节点的数
3	elasticsearch_cluster_health_number_of_in_flight_fetch	gauge	正在进行的碎
4	elasticsearch_cluster_health_number_of_nodes	gauge	集群内所有的
5	elasticsearch_cluster_health_number_of_pending_tasks	gauge	尚未执行的集
6	elasticsearch_cluster_health_initializing_shards	gauge	正在初始化的
7	elasticsearch_cluster_health_unassigned_shards	gauge	未分配分片数
8	elasticsearch_cluster_health_active_primary_shards	gauge	活跃的主分片
9	elasticsearch_cluster_health_active_shards	gauge	活跃的分片总
10	elasticsearch_cluster_health_relocating_shards	gauge	当前节点正在
11	elasticsearch_breakers_tripped	counter	熔断发生次数
12	elasticsearch_breakers_limit_size_bytes	gauge	熔断内存限制
13	elasticsearch_breakers_estimated_size_bytes	gauge	熔断器内存大

2.6 资源饱和度和错误

ElasticSearch 节点使用线程池来管理线程如何消耗内存和 CPU，所以需要监控线程相关的指标。

序号	指标	类型
1	elasticsearch_thread_pool_completed_count	gauge
2	elasticsearch_thread_pool_active_count	gauge
3	elasticsearch_thread_pool_largest_count	gauge
4	elasticsearch_thread_pool_queue_count	gauge
5	elasticsearch_thread_pool_active_count	gauge
6	elasticsearch_indices_fielddata_memory_size_bytes	gauge
7	elasticsearch_indices_fielddata_evictions	

更多如何看面板，详见：https://cloud.tencent.com/developer/article/2396903，**这里不展开了**

常见的一些问题排查方案

1.使用 `explain` 分析查询

Elasticsearch 提供了对于查询的 `explain` 功能，可以帮助我们辅助分析执行计划。

`explain` 功能可以提供查询的详细执行计划，显示每个部分是如何匹配和评分的。

```
GET /<index>/_search
{
  "query": {
    "match": {
      "field": "value"
    }
  },
  "explain": true
}
```

`explain` 的输出包含了每个分片的详细信息，包括每个文档的评分解释。这有助于理解哪些部分耗时较长，或者为什么某些文档得分较高。

分析慢查询

Elasticsearch 提供了慢查询日志（slow log）功能，帮助识别和分析慢查询。以下是启用和查看慢查询日志的步骤。

在 Elasticsearch 配置文件（`elasticsearch.yml`）中设置慢查询日志阈值。例如：

```
index.search.slowlog.threshold.query.warn: 10s
index.search.slowlog.threshold.query.info: 5s
index.search.slowlog.threshold.query.debug: 2s
index.search.slowlog.threshold.query.trace: 500ms
index.indexing.slowlog.threshold.index.warn: 10s
index.indexing.slowlog.threshold.index.info: 5s
index.indexing.slowlog.threshold.index.debug: 2s
index.indexing.slowlog.threshold.index.trace: 500ms
```

上述配置将记录超过指定时间的慢查询日志。

慢查询日志通常保存在 Elasticsearch 日志目录中（例如 `/var/log/elasticsearch`）。可以查看相应的日志文件：

- 搜索慢查询日志：`elasticsearch_index_search_slowlog.log`
- 索引慢查询日志：`elasticsearch_index_indexing_slowlog.log`

慢查询日志记录了详细的查询信息，包括查询内容、耗时、分片信息等。这有助于识别哪些查询比较慢，以及是哪些分片导致的。

优化查询性能

根据 `explain` 和慢查询日志的分析结果，可以采取以下措施优化查询性能：

1. **优化查询语句**: 简化查询条件，避免使用复杂的正则表达式和大量的 `wildcard` 查询。

2. **合理设计索引**: 使用适当的映射和分词器，创建必要的索引和别名。
3. **分页查询**: 对于大结果集，使用 `scroll` API 进行分页查询，避免一次性返回所有数据。
4. **缓存**: 使用 Elasticsearch 提供的缓存功能，如结果缓存和请求缓存，提高查询响应速度。
5. **增加资源**: 如果查询量大且资源紧张，可以考虑增加节点和分片，平衡负载。

示例：慢查询优化

假设我们有以下慢查询：

```
GET /my_index/_search
{
  "query": {
    "match": {
      "description": "some long and complex query"
    }
  }
}
```

通过 `explain` 和慢查询日志发现，`match` 查询在 `description` 字段上耗时较长。优化方案可能包括：

1. **优化字段映射**: 确保 `description` 字段的映射和分词器适合查询需求。
2. **使用更高效的查询类型**: 如果适用，改用 `term` 查询而不是 `match` 查询。
3. **减少查询范围**: 使用过滤器（如 `bool` 查询中的 `filter`）来减少需要查询的数据量。

优化后的查询可能是：

```
GET /my_index/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "term": {
            "status": "active"
          }
        }
      ],
      "must": [
        {
          "match": {
            "description": "some long and complex query"
          }
        }
      ]
    }
  }
}
```

通过合理使用 `filter` 可以减少查询的计算量，因为过滤器不会影响评分且可以被缓存。

通过 `explain` 功能和慢查询日志，可以详细分析 Elasticsearch 的查询性能，识别和优化慢查询。合理设计索引结构和查询语句，使用缓存和增加资源，可以有效提升查询效率。

2. 使用profile api

lasticsearch的Profile API用于分析和优化查询的性能。它可以帮助你了解查询的执行过程以及每个阶段花费的时间，从而找出瓶颈和优化点。

使用Profile API的步骤

1. 构建查询：构建你需要分析的查询。
2. 添加 `profile` 参数：在查询请求中添加 `profile` 参数，并将其设置为 `true` 。
3. 执行查询：执行查询并查看返回的结果。
4. 分析结果：查看返回的结果中的 `profile` 部分，分析查询的执行情况。

假设我们这里有个结构化查询语句,这个是它初始语句(在176上)

```
curl --location --request POST '10.108.138.231:9200/track_meta,track_meta19869,track_meta19870,track_meta19871,track_meta19872/_search?ignore_unavailable=true' \
--header 'Content-Type: application/json' \
--data-raw '{
  "query": {
    "bool": {
      "must": [
        {
          "terms": {
            "camera_id": [
              23812,
              24353,
              29938,
              16367,
              9161
            ]
          }
        },
        {
          "term": {
            "is_set_face_data": true
          }
        },
        {
          "range": {
            "timestamp": {
              "gt": 1716825600,
              "lt": 1716912000
            }
          }
        }
      ]
    }
  },
  "size": 2,
  "sort": [
    {
```

```
      "timestamp": {
        "order": "desc"
      }
    }
  ]
}
'
```

接下来我们加入profile参数

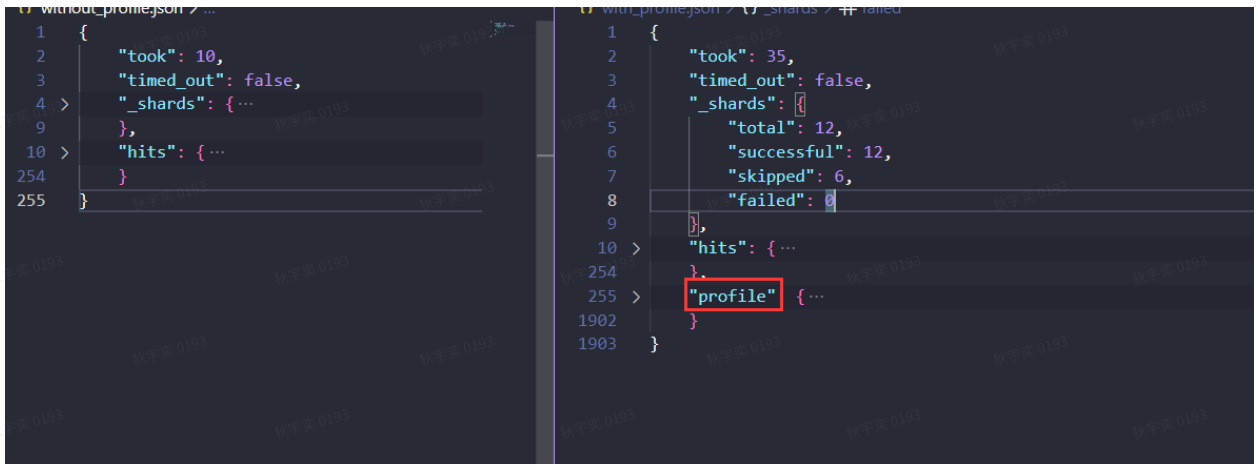
```
curl --location --request POST '10.108.138.231:9200/track_meta,track_meta19869,track_meta19870,track_meta19871,track_meta19872/_search?ignore_unavailable=true' \
--header 'Content-Type: application/json' \
--data-raw '{
  "profile": true,
  "query": {
    "bool": {
      "must": [
        {
          "terms": {
            "camera_id": [
              23812,
              24353,
              29938,
              16367,
              9161
            ]
          }
        },
        {
          "term": {
            "is_set_face_data": true
          }
        },
        {
          "range": {
            "timestamp": {
              "gt": 1716825600,
              "lt": 1716912000
            }
          }
        }
      ]
    }
  },
  "size": 2,
  "sort": [
    {
      "timestamp": {
        "order": "desc"
      }
    }
  ]
}
'
```

输出的数据比较多，这里直接放上文件

with_profile.json

without_profile.json

会发现with_profile中增加了一个profile字段



返回结果中包含了一个 `profile` 字段，里面详细描述了每个分片的查询执行情况。以下是几个关键部分：

- `type`：查询的类型，例如 `BooleanQuery`。
- `description`：查询的描述，例如 `title:elasticsearch`。
- `time_in_nanos`：查询花费的时间（纳秒）。
- `breakdown`：每个阶段的时间分解，例如创建权重时间、构建评分器时间、匹配文档时间等。

通过这些信息，你可以了解查询的各个阶段花费的时间，找出性能瓶颈，并进行相应的优化。

3. 使用explain api分析unassigned shards

使用Elasticsearch的Explain API分析unassigned shards的具体步骤如下：

1. **检查集群状态**：首先，查看集群状态以识别是否有未分配的分片。
2. **使用Explain API**：对未分配的分片使用Explain API进行分析。
3. **查看分片分配说明**：解释未分配分片的原因。
4. **解决问题**：根据解释的原因采取相应的措施解决未分配分片的问题。

以下是详细步骤：

详细步骤

1. **检查集群状态**
   - 使用 `GET /_cluster/health` 检查集群健康状态，确认是否存在未分配的分片。

```
GET /_cluster/health
```

2. **列出未分配的分片**
   - 使用 `GET /_cat/shards` 列出所有分片并找到未分配的分片。

```
GET /_cat/shards
```

3. **使用Explain API**

- 对于每个未分配的分片，使用Explain API进行分析。例如，假设索引为 `my_index` ，分片ID为 `0` ：

```
POST /_cluster/allocation/explain
{
  "index": "my_index",
  "shard": 0,
  "primary": true
}
```

4. **查看分片分配说明**
   - 查看Explain API的响应，分析未分配的原因。

```
{
  "index": "my_index",
  "shard": 0,
  "primary": true,
  "current_state": "unassigned",
  "unassigned_info": {
    "reason": "INDEX_CREATED",
    "at": "2023-06-03T00:00:00.000Z",
    "last_allocation_status": "no_attempt"
  },
  "can_allocate": "no",
  "allocate_explanation": "cannot allocate because allocation is not permitted to any of the nodes"
}
```

5. **解决问题**
   - 根据解释的原因采取相应措施。例如，如果是因为没有足够的磁盘空间，则需要增加节点的磁盘空间。

4. 节点CPU使用率高

原因	描述
查询负载增加	Elasticsearch 集群承受查询负载增加会导致 CPU 使用率升高，通常出现在索引大量新数据或搜索流量增加的情况下。
索引负载增加	Elasticsearch 集群承受索引负载增加会导致 CPU 使用率升高，通常发生在索引大量新数据的情况下。
GC（垃圾回收）	Elasticsearch Java 进程进行垃圾回收时，CPU 使用率上升。GC 清理 Java 堆内存中不再使用的对象，频率和时间随对象数量增加而增加。
机器性能不足	Elasticsearch 部署的机器性能不足会导致 CPU 使用率升高，如慢速 CPU 处理器、内存不足、磁盘 I/O 较慢等。
插件/脚本等性能问题	插件或脚本存在性能问题会导致 CPU 使用率升高。某些插件或脚本可能影响 Elasticsearch 性能，如复杂脚本或调用较慢的第三方库。

Elasticsearch集群的CPU使用率升高时，可以考虑调整哪些参数

参数	描述	建议值范围
indices.store.throttle.max_bytes_per_sec	限制每秒写入的数据量，减缓写入速度。过高的写入速度可能导致CPU使用率升高。最优值取决于硬件配置和负载情况，一般设置为每秒写入速率的80%到90%。	40MB/s 到 45MB/s
indices.memory.index_buffer_size	调整索引缓冲区大小，避免频繁IO操作增加CPU负载。建议设置为可用内存的20%到30%。	20GB 到 30GB
indices.fielddata.cache.size	增大字段数据缓存大小，减少磁盘读取操作。建议设置为可用内存的20%到30%。	20GB 到 30GB
indices.queries.cache.size	增大查询缓存大小，减少频繁查询操作增加CPU负载。建议设置为查询缓存占用可用内存的20%到30%。	20GB 到 30GB
indices.recovery.max_bytes_per_sec	限制每秒恢复的数据量，减缓恢复速度。快速恢复可能导致CPU使用率升高。建议设置为每秒恢复速率的80%到90%。	40MB/s 到 45MB/s
indices.search.slowlog.threshold.query.warn	调整查询慢日志告警阈值，快速发现查询慢的问题。建议设置为查询执行时间的90%到95%。	900ms 到 950ms

在这个github的issues中有一些关于ES的建议配置:**https://github.com/elastic/elasticsearch/issues/4288**

以下是将上述配置项转换成表格形式的输出：

类别	配置项名称	配置值	描述
内存设置	bootstrap.mlockall	true	强制JVM将所有内存锁定在物理内存中，防止交换到磁盘。
线程池设置	threadpool.search.type	fixed	使用固定大小的线程池。
	threadpool.search.size	20	搜索线程池的线程数为20。
	threadpool.search.queue_size	200	搜索线程池的队列大小为200。
	threadpool.bulk.type	fixed	使用固定大小的线程池。
	threadpool.bulk.size	60	批量处理线程池的线程数为60。
	threadpool.bulk.queue_size	3000	批量处理线程池的队列大小为3000。
	threadpool.index.type	fixed	使用固定大小的线程池。
	threadpool.index.size	20	索引线程池的线程数为20。
	threadpool.index.queue_size	1000	索引线程池的队列大小为1000。
		30%	

索引设置	indices.memory.index_buffer_size		分配给索引缓冲区的内存比例为30%。
	indices.memory.min_shard_index_buffer_size	12mb	每个分片的最小索引缓冲区大小为12MB。
	indices.memory.min_index_buffer_size	96mb	索引缓冲区的最小大小为96MB。
缓存大小设置	indices.fielddata.cache.size	30%	字段数据缓存的大小为总内存的30%。
	indices.cache.filter.size	30%	过滤缓存的大小为总内存的30%。
写操作的索引设置	index.refresh_interval	30s	索引的刷新间隔时间为30秒。
	index.translog.flush_threshold_period	5m	事务日志刷新阈值时间为5分钟。
	index.merge.scheduler.max_thread_count	1	合并调度器的最大线程数为1。

排查的时候还可以通过API查看hot_threads:

```
GET _nodes/hot_threads

::: {test}{ikKuXkFvRc-qFCqG99smGg}{VE-uqoiARoONJwomfPwRBw}{127.0.0.1}{127.0.0.1:9300}{ml.machine_memory=8481566720, ml.max_open_jobs=20, ml.enabled=true}
   Hot threads at 2018-04-09T15:58:21.117Z, interval=500ms, busiestThreads=3, ignoreIdleThreads=true:

    0.0% (0s out of 500ms) cpu usage by thread 'Attach Listener'
     unique snapshot
     unique snapshot
     unique snapshot
     unique snapshot
     unique snapshot
     unique snapshot
     unique snapshot
     unique snapshot
     unique snapshot
     unique snapshot
```

5. 节点内存使用率高

ES内存使用率过高，可以考虑调整哪些参数

参数	描述	建议值范围
indices.memory.index_buffer_size	控制索引模块使用的内存缓冲区大小。降低该值可减少内存占用，但可能影响性能。建议设置为每个索引的可用堆内存的5%-10%。	5%-10%的可用堆内存
indices.fielddata.cache.size	控制字段数据缓存大小。降低该值可减少内存占用，但可能影响查询性能。建议设置为10%-30%的可用堆内存。	10%-30%的可用堆内存

indices.queries.cache.size	控制查询缓存大小。降低该值可减少内存占用，但可能影响查询性能。建议设置为10%-30%的可用堆内存。	10%-30%的可用堆内存
indices.breaker.*	控制熔断器阈值，用于保护内存和磁盘资源。调整阈值可避免过度使用内存。	适应具体情况调整
indices.recovery.max_bytes_per_sec	控制恢复速度，降低该值可减少内存占用。建议根据网络带宽进行调整，设置为网络带宽的70%-80%。	网络带宽的70%-80%

6. 节点GC频率过高

原因	描述
JVM堆内存不足	当JVM堆内存不足时，会导致GC频繁触发，通常由于索引数据量增加或查询负载增加等原因导致。
索引数据过多	当索引数据量过多时，会占用大量内存空间，导致JVM堆内存不足，进而引发GC频繁触发。
查询压力过大	查询压力过大会导致Elasticsearch集群需要处理大量查询请求，可能引起JVM堆内存不足，导致GC频繁触发。
代码逻辑问题	存在代码逻辑问题（如内存泄漏）可能导致JVM堆内存占用过高，诱发GC频繁触发。

解决方法	描述	具体demo
增加JVM堆内存大小	增加JVM堆内存大小可以有效降低GC频率，应设置为应用程序需求的最小值加一定余量以确保系统有足够内存。	`java -Xms31G -Xmx31G -jar your-application.jar`
优化查询和索引操作	通过优化查询和索引操作来减少GC频率，例如使用filter替代query、避免过度聚合、减少脚本使用等。	`json GET /_search { "query": { "bool": { "filter": [ { "term": { "status": "active" } } ] } } }`
使用合适的垃圾回收器	选择适合的垃圾回收器可以降低GC频率。CMS适合实时处理，G1适合大型堆内存应用程序。	`java -XX:+UseG1GC -jar your-application.jar`
使用合适的JVM参数	合适的JVM参数对GC频率影响重大，可通过设置堆内存大小、调整垃圾回收器参数等方式来减少GC频率。	`java -Xms31G -Xmx31G -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -jar your-application.jar`

7. 节点磁盘使用率过高

原因	描述
数据量	Elasticsearch的磁盘使用率与数据量直接相关，随着数据量增加，磁盘使用率也会相应增加。
索引设置	

	分片和副本设置会影响数据在磁盘上的存储方式和占用空间大小。分片数量多意味着每个分片占用空间小，但可能增加额外磁盘空间占用。
索引更新频率	索引频繁更新会导致写入更多数据到磁盘，增加磁盘使用率。
删除操作	删除文档不会立即释放磁盘空间，而通过后台段合并操作来回收空间。频繁删除文档可能需定期执行段合并操作以避免持续增加磁盘使用率。
数据复制	支持副本机制导致每个副本需要额外磁盘空间。
磁盘类型	不同类型磁盘对性能和空间占用影响不同，如固态硬盘比机械硬盘更快但成本更高，机械硬盘适合低成本应用。

处理方案	描述
添加更多节点	通过添加更多节点扩展集群存储能力，减轻单节点负担。
删除旧或不需要数据	删除旧或不需要数据释放磁盘空间，确保数据不再需要，因为删除是不可逆操作。
压缩索引	Elasticsearch提供工具压缩索引，在不降低性能情况下减小索引大小。
增加磁盘空间	若磁盘使用率升高由磁盘空间不足导致，可考虑增加磁盘空间。
调整文档存储方式	调整文档存储方式减小磁盘使用率，例如设置字段为不索引或压缩存储。

8. 分词导致的检索不到结果

'

参考文档

https://www.cnblogs.com/sgh1023/p/15691061.html

https://wiki.yitu-inc.com/pages/viewpage.action?pageId=604883858

ES的节点：https://www.elastic.co/guide/en/elasticsearch/reference/7.10/modules-node.html

https://elasticsearch.cn/article/6178

https://juejin.cn/post/7222440107214274597

https://www.cyub.vip/2020/09/25/Elasticsearch%E5%86%85%E5%AD%98%E5%8D%A0%E7%94%A8%E5%88%86%E6%9E%90/

https://www.shenyanchao.cn/blog/2018/12/04/lucene-fst/

使用到的命令

```
curl -X GET "http://10.102.178.118:9200/_nodes"
curl -X GET "http://10.102.178.118:9200/_cluster/state"
curl -X GET "http://10.108.138.231:9200/_cat/indices?v"
curl -X GET "http://10.110.192.50:9200/_cat/indices/repository?v"
curl -X GET "http://10.101.162.184:9200/_cat/shards?v"
curl -X GET "http://10.110.192.50:9200/_cat/shards/track_meta19864?v"
```