# 一. Elasticsearch架构原理

## 1、Elasticsearch的节点类型

在Elasticsearch主要分成两类节点，一类是Master，一类是DataNode。

### 1.1 Master节点

在Elasticsearch启动时，会选举出来一个Master节点。当某个节点启动后，然后使用Zen Discovery机制找到集群中的其他节点，并建立连接。

discovery.seed_hosts: ["192.168.21.130", "192.168.21.131", "192.168.21.132"]

并从候选主节点中选举出一个主节点。

cluster.initial_master_nodes: ["node1", "node2","node3"]

Master节点主要负责：

管理索引（创建索引、删除索引）、分配分片

维护元数据

管理集群节点状态

不负责数据写入和查询，比较轻量级

一个Elasticsearch集群中，只有一个Master节点。在生产环境中，内存可以相对小一点，但机器要稳定。

### 1.2 DataNode节点

在Elasticsearch集群中，会有N个DataNode节点。DataNode节点主要负责：

数据写入、数据检索，大部分Elasticsearch的压力都在DataNode节点上

在生产环境中，内存最好配置大一些

### 1.3 补充

1）集群状态

如何快速了解集群的健康状况？green、yellow、red?

green：每个索引的primary shard和replica shard都是active状态的

yellow：每个索引的primary shard都是active状态的，但是部分replica shard不是active状态，处于不可用的状态

red：不是所有索引的primary shard都是active状态的，部分索引有数据丢失了

集群什么情况会处于一个yellow状态?

假设现在就一台linux服务器，就启动了一个es进程，相当于就只有一个node。现在es中有一个index，就是kibana自己内置建立的index。由于默认的配置是给每个index分配1个primary shard和1个replica shard，而且primary shard和replica shard不能在同一台机器上（为了容错）。现在kibana自己建立的index是1个primary shard和1个replica shard。当前就一个node，所以只有1个primary shard被分配了和启动了，但是一个replica shard没有第二台机器去启动。

测试：启动第二个es进程，就会在es集群中有2个node，然后那1个replica shard就会自动分配过去，然后cluster status就会变成green状态。

2）不同节点介绍

主节点：node.master：true

数据节点: node.data: true

-1. 客户端节点

当主节点和数据节点配置都设置为false的时候，该节点只能处理路由请求，处理搜索，分发索引操作等，从本质上来说该客户节点表现为智能负载平衡器。

独立的客户端节点在一个比较大的集群中是非常有用的，他协调主节点和数据节点，客户端节点加入集群可以得到集群的状态，根据集群的状态可以直接路由请求。

-2. 数据节点

数据节点主要是存储索引数据的节点，主要对文档进行增删改查操作，聚合操作等。数据节点对cpu，内存，io要求较高， 在优化的时候需要监控数据节点的状态，当资源不够的时候，需要在集群中添加新的节点。

-3. 主节点

主资格节点的主要职责是和集群操作相关的内容，如创建或删除索引，跟踪哪些节点是群集的一部分，并决定哪些分片分配给相关的节点。稳定的主节点对集群的健康是非常重要的，默认情况下任何一个集群中的节点都有可能被选为主节点，索引数据和搜索查询等操作会占用大量的cpu，内存，io资源，为了确保一个集群的稳定，分离主节点和数据节点是一个比较好的选择。

在一个生产集群中我们可以对这些节点的职责进行划分，建议集群中设置3台以上的节点作为master节点，这些节点只负责成为主节点，维护整个集群的状态。再根据数据量设置一批data节点，这些节点只负责存储数据，后期提供建立索引和查询索引的服务，这样的话如果用户请求比较频繁，这些节点的压力也会比较大，所以在集群中建议再设置一批client节点(node.master: false node.data: false)，这些节点只负责处理用户请求，实现请求转发，负载均衡等功能

## 二 、分片和副本机制

### 2.1 分片（Shard）

Elasticsearch是一个分布式的搜索引擎，索引的数据也是分成若干部分，分布在不同的服务器节点中

分布在不同服务器节点中的索引数据，就是分片（Shard）。Elasticsearch会自动管理分片，如果发现分片分布不均衡，就会自动迁移

一个索引（index）由多个shard（分片）组成，而分片是分布在不同的服务器上的

### 2.2 副本

为了对Elasticsearch的分片进行容错，假设某个节点不可用，会导致整个索引库都将不可用。所以，需要对分片进行副本容错。每一个分片都会有对应的副本。

在Elasticsearch中，默认创建的索引为1个分片、每个分片有1个主分片和1个副本分片。

每个分片都会有一个Primary Shard（主分片），也会有若干个Replica Shard（副本分片）

Primary Shard和Replica Shard不在同一个节点上

### 2.3 指定分片、副本数量

```
// 创建指定分片数量、副本数量的索引
PUT /job_idx_shard_temp
{
"mappings":{
"properties":{
"id":{"type":"long","store":true},
"area":{"type":"keyword","store":true},
"exp":{"type":"keyword","store":true},
"edu":{"type":"keyword","store":true},
"salary":{"type":"keyword","store":true},
"job_type":{"type":"keyword","store":true},
```

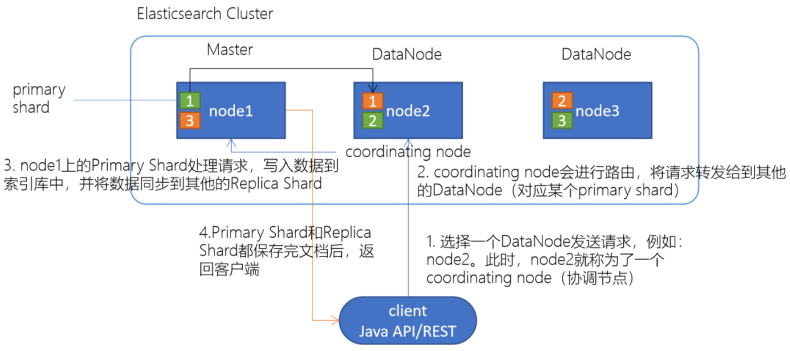
```
"cmp":{"type":"keyword","store":true},
"pv":{"type":"keyword","store":true},
"title":{"type":"text","store":true},
"jd":{"type":"text"}

}
},
"settings":{
"number_of_shards":3,
"number_of_replicas":2
}
}

// 查看分片、主分片、副本分片
GET /_cat/indices?v
```

## 三、Elasticsearch重要工作流程

### 3.1 Elasticsearch文档写入原理




1.选择任意一个DataNode发送请求，例如：node2。此时，node2就成为一个coordinating node（协调节点）

2.计算得到文档要写入的分片

`shard = hash(routing) % number_of_primary_shards`

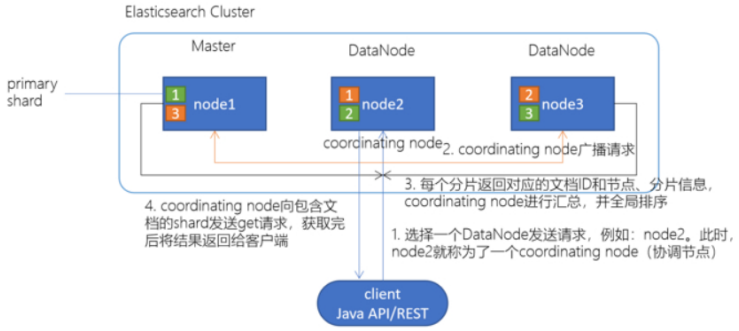
routing 是一个可变值，默认是文档的 _id

3.coordinating node会进行路由，将请求转发给对应的primary shard所在的DataNode（假设primary shard在node1、replica shard在node2)

4.node1节点上的Primary Shard处理请求，写入数据到索引库中，并将数据同步到Replica shard

5.Primary Shard和Replica Shard都保存好了文档，返回client

### 3.2 Elasticsearch检索原理




client发起查询请求，某个DataNode接收到请求，该DataNode就会成为协调节点（Coordinating Node）

协调节点（Coordinating Node）将查询请求广播到每一个数据节点，这些数据节点的分片会处理该查询请求

每个分片进行数据查询，将符合条件的数据放在一个优先队列中，并将这些数据的文档ID、节点信息、分片信息返回给协调节点

协调节点将所有的结果进行汇总，并进行全局排序

协调节点向包含这些文档ID的分片发送get请求，对应的分片将文档数据返回给协调节点，最后协调节点将数据返回给客户端

## 四、Elasticsearch准实时索引实现

### 4.1 溢写到文件系统缓存

当数据写入到ES分片时，会首先写入到内存中，然后通过内存的buffer生成一个segment，并刷到文件系统缓存中，数据可以被检索（注意不是直接刷到磁盘）

ES中默认1秒，refresh一次

### 4.2 写translog保障容错

在写入到内存中的同时，也会记录translog日志，在refresh期间出现异常，会根据translog来进行数据恢复

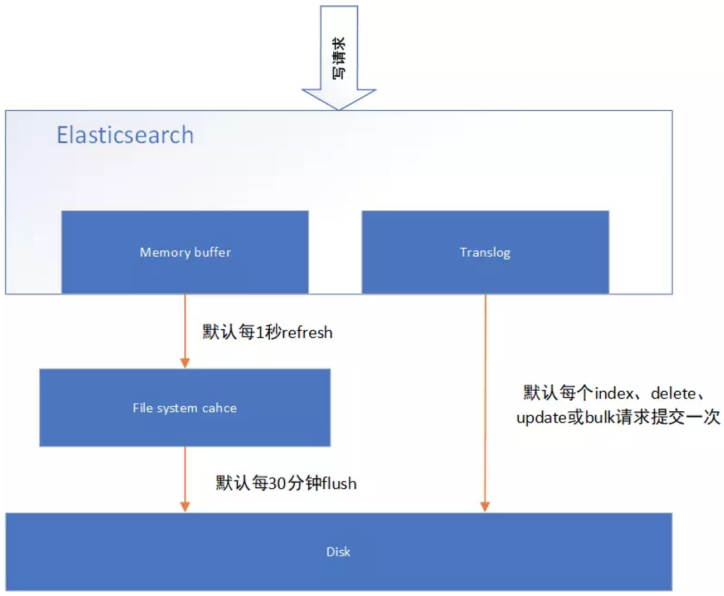
等到文件系统缓存中的segment数据都刷到磁盘中，清空translog文件

### 4.3 flush到磁盘

ES默认每隔30分钟会将文件系统缓存的数据刷入到磁盘

### 4.4 segment合并

Segment太多时，ES定期会将多个segment合并成为大的segment，减少索引查询时IO开销，此阶段ES会真正的物理删除（之前执行过的delete的数据）




## 五、Elasticsearch集群脑裂及解决办法

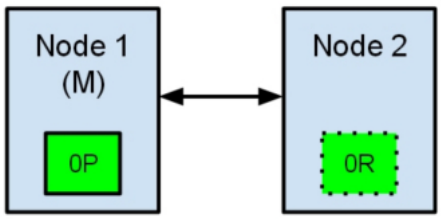
集群脑裂是什么?

所谓脑裂问题，就是同一个集群中的不同节点，对于集群的状态有了不一样的理解，比如集群中存在两个master

如果因为网络的故障，导致一个集群被划分成了两片，每片都有多个node，以及一个master，那么集群中就出现了两个master了。

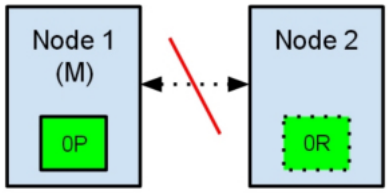
但是因为master是集群中非常重要的一个角色，主宰了集群状态的维护，以及shard的分配，因此如果有两个master，可能会导致数据异常。

如:




节点1在启动时被选举为主节点并保存主分片标记为0P，而节点2保存复制分片标记为0R

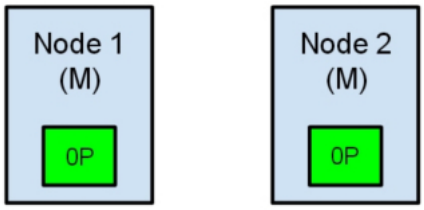
现在，如果在两个节点之间的通讯中断了，会发生什么？由于网络问题或只是因为其中一个节点无响应,这是有可能发生的。




两个节点都相信对方已经挂了。节点1不需要做什么，因为它本来就被选举为主节点。但是节点2会自动选举它自己为主节点，因为它相信集群的一部分没有主节点了。

在elasticsearch集群，是有主节点来决定将分片平均的分布到节点上的。节点2保存的是复制分片，但它相信主节点不

可用了。所以它会自动提升Node2节点为主节点。




现在我们的集群在一个不一致的状态了。打在节点1上的索引请求会将索引数据分配在主节点，同时打在节点2的请求会将索引数据放在分片上。在这种情况下，分片的两份数据分开了，如果不做一个全量的重索引很难对它们进行重排序。在更坏的情况下，一个对集群无感知的索引客户端（例如，使用REST接口的）,这个问题非常透明难以发现，无论哪个节点被命中索引请求仍然在每次都会成功完成。问题只有在搜索数据时才会被隐约发现：取决于搜索请求命中了哪个节点，结果都会不同。

那么那个参数的作用，就是告诉es直到有足够的master候选节点时，才可以选举出一个master，否则就不要选举出一个master。这个参数必须被设置为集群中master候选节点的quorum数量，也就是大多数。至于quorum的算法，就是：master候选节点数量 / 2 + 1。

比如我们有10个节点，都能维护数据，也可以是master候选节点，那么quorum就是10 / 2 + 1 = 6。

如果我们有三个master候选节点，还有100个数据节点，那么quorum就是3 / 2 + 1 = 2

如果我们有2个节点，都可以是master候选节点，那么quorum是2 / 2 + 1 = 2。此时就有问题了，因为如果一个node挂掉了，那么剩下一个master候选节点，是无法满足quorum数量的，也就无法选举出新的master，集群就彻底挂掉了。此时就只能将这个参数设置为1，但是这就无法阻止脑裂的发生了。

2个节点，discovery.zen.minimum_master_nodes分别设置成2和1会怎么样

综上所述，一个生产环境的es集群，至少要有3个节点，同时将这个参数设置为quorum，也就是2。
discovery.zen.minimum_master_nodes设置为2，如何避免脑裂呢?

那么这个是参数是如何避免脑裂问题的产生的呢？比如我们有3个节点，quorum是2.现在网络故障，1个节点在一个网络区域，另外2个节点在另外一个网络区域，不同的网络区域内无法通信。这个时候有两种情况情况：

（1）如果master是单独的那个节点，另外2个节点是master候选节点，那么此时那个单独的master节点因为没有指定数量的候选master node在自己当前所在的集群内，因此就会取消当前master的角色，尝试重新选举，但是无法选举成功。然后另外一个网络区域内的node因为无法连接到master，就会发起重新选举，因为有两个master候选节点，满足了quorum，因此可以成功选举出一个master。此时集群中就会还是只有一个master。

（2）如果master和另外一个node在一个网络区域内，然后一个node单独在一个网络区域内。那么此时那个单独的node因为连接不上master，会尝试发起选举，但是因为master候选节点数量不到quorum，因此无法选举出master。而另外一个网络区域内，原先的那个master还会继续工作。这也可以保证集群内只有一个master节点。

综上所述，集群中master节点的数量至少3台，三台主节点通过在elasticsearch.yml中配置
discovery.zen.minimum_master_nodes: 2，就可以避免脑裂问题的产生。