**ES学习笔记之-ClusterState的学习**

前面研究过ES的get api的整体思路，作为编写ES插件时的借鉴。当时的重点在与理解整体流程，主要是shardOperation()的方法内部的调用逻辑，就弱化了shards()方法。实际上shards()方法在理解ES的结构层面，作用更大一些。我们还是从get api入手来理解shards()。

先回顾一下get api的使用流程：

添加文档到ES:

curl -XPUT 'http://localhost:9200/test1/type1/1' -d '{"name":"hello"}'

根据文档ID读取数据:

curl -XGET 'http://localhost:9200/test1/type1/1'

使用很简单。但是如果考虑到分布式，背后的逻辑就不简单了。 假如ES集群有3个节点，数据所在的索引也有3个分片，每个分片一个副本。即index的设置如下:

{

"test1" : {

"settings" : {

"index" : {

"number\_of\_replicas" : "1",

"number\_of\_shards" : "3"

}

}

}

}

那么id为1的doc该分发到那个分片呢？ 这个问题需要一篇详细的博文解答，这里我们先简单给一个结论:

默认情况下，ES会按照文档id计算一个hash值， 采用的是Murmur3HashFunction，然后根据这个id跟分片数取模。实现代码是MathUtils.mod(hash, indexMetaData.getNumberOfShards()); 最后的结果作为文档所在的分片id，所以ES的分片标号是从0开始的。

1.

不知存，焉知取。

再整理一下取数据的核心流程:

s1: 根据文档id定位到数据所在分片。由于可以设为多个副本，所以一个分片会映射到多个节点。

s2: 根据分片节点的映射信息，选择一个节点，去获取数据。 这里重点关注的是节点的选择方式，简而言之，我们需要负载均衡，不然设置副本就没有意义了。

上面两步都关联着一个核心的数据结构ClusterState, 我们可以使用\_cluster/state?pretty来查看这个数据结构:

# http://localhost:9200/\_cluster/state?pretty

{

"cluster\_name" : "elasticsearch",

"version" : 4,

"state\_uuid" : "b6B739p5SbanNLyKxTMHfQ",

"master\_node" : "KnEE25tzRjaXblFJq5jqRA",

"blocks" : { },

"nodes" : {

"KnEE25tzRjaXblFJq5jqRA" : {

"name" : "Mysterio",

"transport\_address" : "127.0.0.1:9300",

"attributes" : { }

}

},

"metadata" : {

"cluster\_uuid" : "ZIl7g86YRiGv8Dqz4DCoAQ",

"templates" : { },

"indices" : {

"test1" : {

"state" : "open",

"settings" : {

"index" : {

"creation\_date" : "1553995485603",

"uuid" : "U7v5t\_T7RG6rNU3JlGCCBQ",

"number\_of\_replicas" : "1",

"number\_of\_shards" : "1",

"version" : {

"created" : "2040599"

}

}

},

"mappings" : { },

"aliases" : [ ]

}

}

},

"routing\_table" : {

"indices" : {

"test1" : {

"shards" : {

"0" : [ {

"state" : "STARTED",

"primary" : true,

"node" : "KnEE25tzRjaXblFJq5jqRA",

"relocating\_node" : null,

"shard" : 0,

"index" : "test1",

"version" : 2,

"allocation\_id" : {

"id" : "lcSHbfWDRyOKOhXAf3HXLA"

}

}, {

"state" : "UNASSIGNED",

"primary" : false,

"node" : null,

"relocating\_node" : null,

"shard" : 0,

"index" : "test1",

"version" : 2,

"unassigned\_info" : {

"reason" : "INDEX\_CREATED",

"at" : "2019-03-31T01:24:45.845Z"

}

} ]

}

}

}

},

"routing\_nodes" : {

"unassigned" : [ {

"state" : "UNASSIGNED",

"primary" : false,

"node" : null,

"relocating\_node" : null,

"shard" : 0,

"index" : "test1",

"version" : 2,

"unassigned\_info" : {

"reason" : "INDEX\_CREATED",

"at" : "2019-03-31T01:24:45.845Z"

}

} ],

"nodes" : {

"KnEE25tzRjaXblFJq5jqRA" : [ {

"state" : "STARTED",

"primary" : true,

"node" : "KnEE25tzRjaXblFJq5jqRA",

"relocating\_node" : null,

"shard" : 0,

"index" : "test1",

"version" : 2,

"allocation\_id" : {

"id" : "lcSHbfWDRyOKOhXAf3HXLA"

}

} ]

}

}

}

整个结构比较复杂，我们慢慢拆解， 一步步逐个击破。 拆解的思路还是从使用场景入手。

IndexMetaData的学习

metaData的格式如下:

"metadata" : {

"cluster\_uuid" : "ZIl7g86YRiGv8Dqz4DCoAQ",

"templates" : { },

"indices" : {

"test1" : {

"state" : "open",

"settings" : {

"index" : {

"creation\_date" : "1553995485603",

"uuid" : "U7v5t\_T7RG6rNU3JlGCCBQ",

"number\_of\_replicas" : "1",

"number\_of\_shards" : "1",

"version" : {

"created" : "2040599"

}

}

},

"mappings" : { },

"aliases" : [ ]

}

}

}

即metadata中存储了集群中每个索引的分片和副本数量， 索引的状态， 索引的mapping, 索引的别名等。这种结构，能提供出来的功能就是根据索引名称获取索引元数据， 代码如下:

# OperationRouting.generateShardId()

IndexMetaData indexMetaData = clusterState.metaData().index(index);

if (indexMetaData == null) {

throw new IndexNotFoundException(index);

}

final Version createdVersion = indexMetaData.getCreationVersion();

final HashFunction hashFunction = indexMetaData.getRoutingHashFunction();

final boolean useType = indexMetaData.getRoutingUseType();

这里我们关注点就是clusterState.metaData().index(index)这句代码，它实现了根据索引名称获取索引元数据的功能。 通过元数据中的分片数结合文档id，我们就能定位出文档所在的分片。 这个功能在Delete, Index, Get 三类API中都是必须的。 这里我们也能理解为什么ES的索引分片数量不能修改： 如果修改了，那么hash函数就没法正确定位数据所在分片。

IndexRoutingTable的学习

"routing\_table" : {

"indices" : {

"test1" : {

"shards" : {

"0" : [ {

"state" : "STARTED",

"primary" : true,

"node" : "KnEE25tzRjaXblFJq5jqRA",

"relocating\_node" : null,

"shard" : 0,

"index" : "test1",

"version" : 2,

"allocation\_id" : {

"id" : "lcSHbfWDRyOKOhXAf3HXLA"

}

}, {

"state" : "UNASSIGNED",

"primary" : false,

"node" : null,

"relocating\_node" : null,

"shard" : 0,

"index" : "test1",

"version" : 2,

"unassigned\_info" : {

"reason" : "INDEX\_CREATED",

"at" : "2019-03-31T01:24:45.845Z"

}

} ]

}

}

}

}

routing\_table存储着每个索引的分片信息，通过这个结构，我们能清晰地了解如下的信息:

1. 索引分片在各个节点的分布

2. 索引分片是否为主分片

假如一个分片有2个副本，且都分配在不同的节点上，那么get api一共有三个数据节点可供选择， 选择哪一个呢？这里暂时不考虑带preference参数。

为了使每个节点都能公平被选择到，达到负载均衡的目的，这里用到了随机数。参考RotateShuffer

/\*\*

\* Basic {@link ShardShuffler} implementation that uses an {@link AtomicInteger} to generate seeds and uses a rotation to permute shards.

\*/

public class RotationShardShuffler extends ShardShuffler {

private final AtomicInteger seed;

public RotationShardShuffler(int seed) {

this.seed = new AtomicInteger(seed);

}

@Override

public int nextSeed() {

return seed.getAndIncrement();

}

@Override

public List<ShardRouting> shuffle(List<ShardRouting> shards, int seed) {

return CollectionUtils.rotate(shards, seed);

}

}

也就是说使用ThreadLocalRandom.current().nextInt()生成随机数作为种子， 然后取的时候依次旋转。

Collections.rotate()的效果可以用如下的代码演示:

public static void main(String[] args) {

List<String> list = Lists.newArrayList("a","b","c");

int a = ThreadLocalRandom.current().nextInt();

List<String> l2 = CollectionUtils.rotate(list, a );

List<String> l3 = CollectionUtils.rotate(list, a+1);

System.out.println(l2);

System.out.println(l3);

}

-----

[b, c, a]

[c, a, b]

比如请求A得到的节点列表是[b,c,a], 那么请求B得到的节点列表是[c,a,b]。这样就达到了负载均衡的目的。

DiscoveryNodes的学习。

由于routing\_table中存储的是节点的id, 那么将请求发送到目标节点时，还需要知道节点的ip及端口等配置信息。 这些信息存储在nodes中。

"nodes" : {

"KnEE25tzRjaXblFJq5jqRA" : {

"name" : "Mysterio",

"transport\_address" : "127.0.0.1:9300",

"attributes" : { }

}

}

通过这个nodes获取到节点信息后，就可以发送请求了，ES所有内部节点的通信都是基于transportService.sendRequest()。

总结一下，本文基于get api 梳理了一下ES的ClusterState中的几个核心结构: metadata,nodes, routing\_table。 还有一个routing\_nodes这里没有用到。后面梳理清楚使用场景后再记录。