ES分片和路由

代码参考了Elasticsearch5.5，大致对比了Elasticsearch2.2，路由这块代码变化不大。

分片与节点的关系

ES的分片分为主分片和副本分片，路由算法有两个:even_shard和balances，这两个的区别就是balances分片路由算法可以动态调节参数。

两者的分配方式都是Index级别的，就是说只要主分片和副本在不同的机器上，那么就认为没问题。具体到代码就是ES会起一个AllocationService的服务，通过applyStartedShards方法把index中没有分配的shards分配到节点上，如下：

```
public ClusterState applyStartedShards(ClusterState clusterState, List<ShardRoutin
g> startedShards) {
        if (startedShards.isEmpty()) {
            return clusterState;
        }
        //拿到可以路由的节点
        RoutingNodes routingNodes = getMutableRoutingNodes(clusterState);
        // shuffle the unassigned nodes, just so we won't have things like poiso
n failed shards
        //打散这些节点
        routingNodes.unassigned().shuffle();
        //初始化路由分配服务
        RoutingAllocation allocation = new RoutingAllocation(allocationDeciders, r
outingNodes, clusterState,
            clusterInfoService.getClusterInfo(), currentNanoTime(), false);
        // as starting a primary relocation target can reinitialize replica shard
s, start replicas first
        startedShards = new ArrayList<>(startedShards);
        //对分片按主备份优先级排序
        Collections.sort(startedShards, Comparator.comparing(ShardRouting::primar
y));
        //应用一些逻辑到这些shards
        applyStartedShards(allocation, startedShards);
        gatewayAllocator.applyStartedShards(allocation, startedShards);
        //把这些shards根据一些balance的规则调整shards
        reroute(allocation);
        String startedShardsAsString = firstListElementsToCommaDelimitedString(sta
rtedShards, s -> s.shardId().toString());
        return buildResultAndLogHealthChange(clusterState, allocation, "shards sta
rted [" + startedShardsAsString + "] ...");
    }
```

根据代码可以看出来，ES拿到一个index所有的主分片和备份分片，然后按照一定的规则，比如主备份分片不能在一台机器上、是否超过一台机器最大可分配分片数等规则随机的分配到打散的节点上，所以机器节点数和分片的关系最好满足以下公式：

```
节点数=主分片数*（副本数+1）
```

这样分片就会比较规则的分布在节点上。

单机上分片与数据目录(path.data)的关系

当一个分片通过路由策略找到某一个节点后，因为节点上配置了多个数据目录，所以还需要找到某一个数据目录才能存放这个分片，从代码可用看出来，基本以轮询的方式来遍历数据目录和Lucene索引目录建立对应关系，所以如果yi

- 先解析配置文件把目录信息放到dtaFiles中，代码如下所示：

```
//通过配置文件获取path.data目录
List<String> dataPaths = PATH_DATA_SETTING.get(settings);
        final ClusterName clusterName = ClusterName.CLUSTER_NAME_SETTING.get(setti
ngs);
        if (dataPaths.isEmpty() == false) {
            dataFiles = new Path[dataPaths.size()];
            dataWithClusterFiles = new Path[dataPaths.size()];
            for (int i = 0; i < dataPaths.size(); i++) {
                //把每个目录放到dataFiles中
                dataFiles[i] = PathUtils.get(dataPaths.get(i));
                //把每个目录加一个集群名字放到dataWithClusterFiles中。
                dataWithClusterFiles[i] = dataFiles[i].resolve(clusterName.value
());
            }
        } else {
            dataFiles = new Path[]{homeFile.resolve("data")};
            dataWithClusterFiles = new Path[]{homeFile.resolve("data").resolve(clu
sterName.value())};
        }
```

- 通过NodeEnvironment构造函数吧数据目录和Lucene索引目录联系起来，并且把数据目录放到NodeEnvironment的属性nodePaths中，并且对外暴露了nodeDataPaths函数，这里只截取了部分代码。

```
int maxLocalStorageNodes = MAX_LOCAL_STORAGE_NODES_SETTING.get(settings);
            for (int possibleLockId = 0; possibleLockId < maxLocalStorageNodes; po
ssibleLockId++) {
                for (int dirIndex = 0; dirIndex < environment.dataFiles().lengt
h; dirIndex++) {
                    Path dataDirWithClusterName = environment.dataWithClusterFiles
()[dirIndex];
                    Path dataDir = environment.dataFiles()[dirIndex];
                    // TODO: Remove this in 6.0, we are no longer going to read fr
om the cluster name directory
                    if (readFromDataPathWithClusterName(dataDirWithClusterName)) {
                        DeprecationLogger deprecationLogger = new DeprecationLogge
r(startupTraceLogger);
                        deprecationLogger.deprecated("ES has detected the [path.da
ta] folder using the cluster name as a folder [{}], " +
                                                     "Elasticsearch 6.0 will not allow the clus
ter name as a folder within the data path", dataDir);
                        dataDir = dataDirWithClusterName;
                    }
                    Path dir = resolveNodePath(dataDir, possibleLockId);
                    Files.createDirectories(dir);

                    try (Directory luceneDir = FSDirectory.open(dir, NativeFSLockF
actory.INSTANCE)) {
                        startupTraceLogger.trace("obtaining node lock o
n {} ...", dir.toAbsolutePath());
                        try {
                            locks[dirIndex] = luceneDir.obtainLock(NODE_LOCK_FILEN
AME);
                            nodePaths[dirIndex] = new NodePath(dir);
                            nodeLockId = possibleLockId;
```

- 真正的

```
public static ShardPath selectNewPathForShard(NodeEnvironment env, ShardId shardId, IndexSettings indexSettings,
                                              long avgShardSizeInBytes, Map<Path,Integer> dataPathToShardCount) throws IOException {

        final Path dataPath;
        final Path statePath;

        //这个index是否有自定义的数据目录，如果有，直接取出来。
        if (indexSettings.hasCustomDataPath()) {
            dataPath = env.resolveCustomLocation(indexSettings, shardId);
            statePath = env.nodePaths()[0].resolve(shardId);
        } else {
            BigInteger totFreeSpace = BigInteger.ZERO;
            //数据目录所有可用空间求和。
            for (NodeEnvironment.NodePath nodePath : env.nodePaths()) {
                totFreeSpace = totFreeSpace.add(BigInteger.valueOf(nodePath.fileStore.getUsableSpace()));
            }

            // TODO: this is a hack!!  We should instead keep track of incoming (relocated) shards since we know
            // how large they will be once they're done copying, instead of a silly guess for such cases:

            // Very rough heuristic of how much disk space we expect the shard will use over its lifetime, the max of current average
            // shard size across the cluster and 5% of the total available free space on this node:
            //因为估计一个shard的大小比较难，所以采用历史shards的评价大小和节点数据目录可用空间的5%取最大值。
            BigInteger estShardSizeInBytes = BigInteger.valueOf(avgShardSizeInBytes).max(totFreeSpace.divide(BigInteger.valueOf(20)));

            // TODO - do we need something more extensible? Yet, this does the job for now...
            final NodeEnvironment.NodePath[] paths = env.nodePaths();
            NodeEnvironment.NodePath bestPath = null;
            BigInteger maxUsableBytes = BigInteger.valueOf(Long.MIN_VALUE);
            //遍历数据目录，判断可用空间是否可用。
            for (NodeEnvironment.NodePath nodePath : paths) {
                FileStore fileStore = nodePath.fileStore;

                BigInteger usableBytes = BigInteger.valueOf(fileStore.getUsableSpace());
                assert usableBytes.compareTo(BigInteger.ZERO) >= 0;

                // Deduct estimated reserved bytes from usable space:
                Integer count = dataPathToShardCount.get(nodePath.path);
                if (count != null) {
```

```
                    usableBytes = usableBytes.subtract(estShardSizeInBytes.multipl
y(BigInteger.valueOf(count)));
                }
                if (bestPath == null || usableBytes.compareTo(maxUsableByte
s) > 0) {
                    maxUsableBytes = usableBytes;
                    bestPath = nodePath;
                }
            }

            statePath = bestPath.resolve(shardId);
            dataPath = statePath;
        }
        return new ShardPath(indexSettings.hasCustomDataPath(), dataPath, statePat
h, shardId);
    }
```

从代码中可以看出来，就是遍历了数据目录，然后看是否有可用空间，如果有可用空间，那么就把这个shard放到这个数据目录中。

总结

- 单机情况下最好一个index的分片数与数据目录的个数一致，比如parh.data配置了6个目录，那么分片数也应该是6个。
- 集群情况下，那么主分片数和备份数最好和机器数量保持一致，也就是说，如果机器有30台，备份数1份，那么shard数应该是15个。这样即使某些index数据量过多，也会比较均匀的分布在各个机器上。
- 集群情况下，这种分配方式会因为某几个index数据量过大，导致数据目录分配不均匀，但是避免了节点分配shards的不均匀，总体还是可行的。