Elasticsearch分布式一致性原理剖析(一)-节点篇

前言

“Elasticsearch分布式一致性原理剖析”系列将会对Elasticsearch的分布式一致性原理进行详细的剖析，介绍其实现方式、原理以及其存在的问题等(基于6.2版本)。

ES目前是最流行的分布式搜索引擎系统，其使用Lucene作为单机存储引擎并提供强大的搜索查询能力。学习其搜索原理，则必须了解Lucene，而学习ES的架构，就必须了解其分布式如何实现，而一致性是分布式系统的核心之一。

本篇将介绍ES的集群组成、节点发现与Master选举，错误检测与扩缩容相关的内容。ES在处理节点发现与Master选举等方面没有选择Zookeeper等外部组件，而是自己实现的一套，本文会介绍ES的这套机制是如何工作的，存在什么问题。本文的主要内容如下：

1. ES集群构成
2. 节点发现
3. Master选举
4. 错误检测
5. 集群扩缩容
6. 与Zookeeper、raft等实现方式的比较
7. 小结

ES集群构成

首先，一个Elasticsearch集群(下面简称ES集群)是由许多节点(Node)构成的，Node可以有不同的类型，通过以下配置，可以产生四种不同类型的Node：

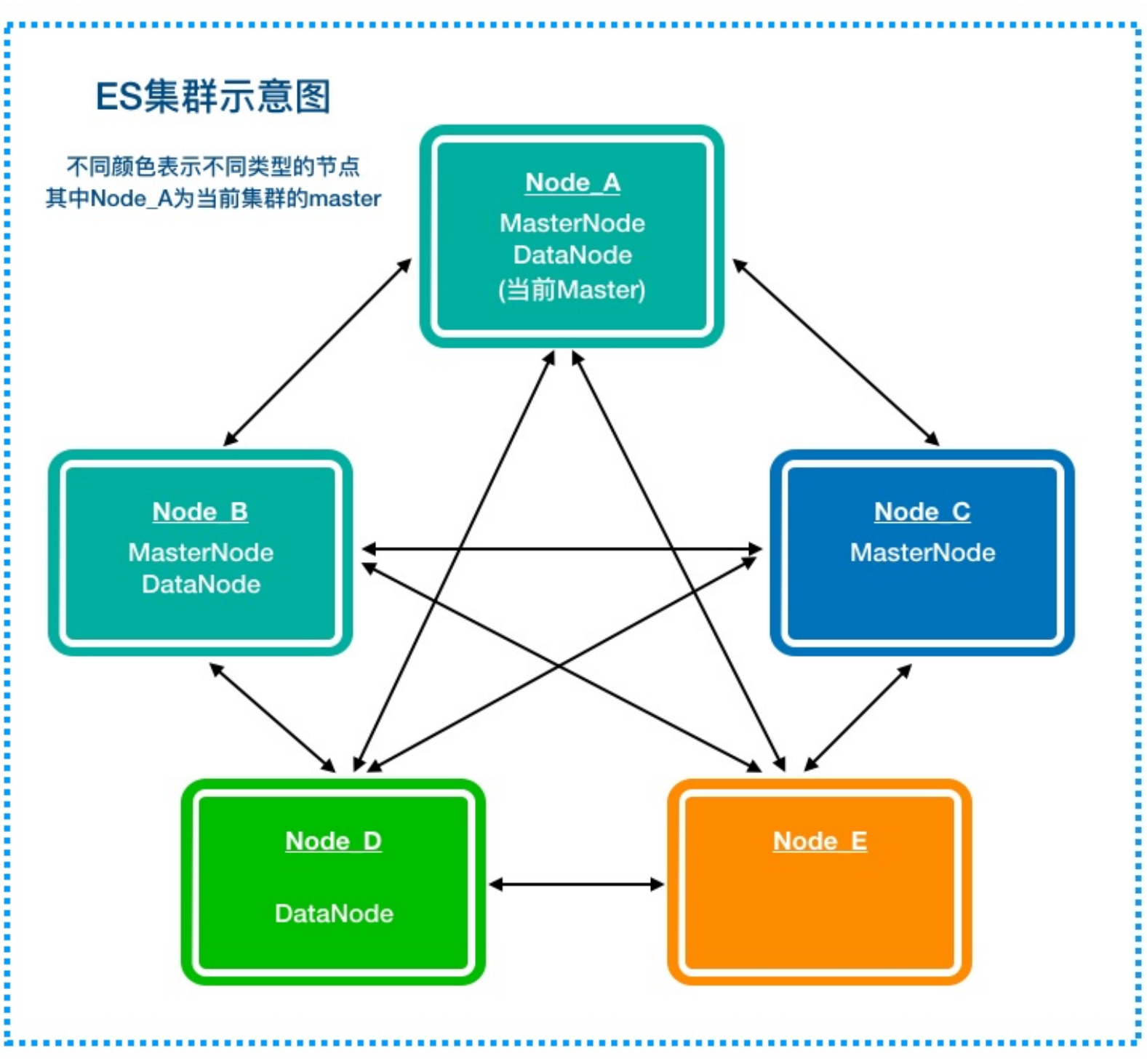
```
conf/elasticsearch.yml:
    node.master: true/false
    node.data: true/false
```

四种不同类型的Node是一个node.master和node.data的true/false的两两组合。当然还有其他类型的Node，比如IngestNode(用于数据预处理等)，不在本文讨论范围内。

当node.master为true时，其表示这个node是一个master的候选节点，可以参与选举，在ES的文档中常被称作master-eligible node，类似于MasterCandidate。ES正常运行时只能有一个master(即leader)，多于1个时会发生脑裂。

当node.data为true时，这个节点作为一个数据节点，会存储分配在该node上的shard的数据并负责这些shard的写入、查询等。

此外，任何一个集群内的node都可以执行任何请求，其会负责将请求转发给对应的node进行处理，所以当node.master和node.data都为false时，这个节点可以作为一个类似proxy的节点，接受请求并进行转发、结果聚合等。



上图是一个ES集群的示意图，其中NodeA是当前集群的Master，NodeB和NodeC是Master的候选节点，其中NodeA和NodeB同时也是数据节点(DataNode)，此外，NodeD是一个单纯的数据节点，Node_E是一个proxy节点。每个Node会跟其他所有Node建立连接。

到这里，我们提一个问题，供读者思考：一个ES集群应当配置多少个master-eligible node，当集群的存储或者计算资源不足，需要扩容时，新扩上去的节点应该设置为何种类型？

节点发现

ZenDiscovery是ES自己实现的一套用于节点发现和选主等功能的模块，没有依赖Zookeeper等工具，官方文档：

https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-discovery-zen.html

简单来说，节点发现依赖以下配置：

```
conf/elasticsearch.yml:
    discovery.zen.ping.unicast.hosts: [1.1.1.1, 1.1.1.2, 1.1.1.3]
```

这个配置可以看作是，在本节点到每个hosts中的节点建立一条边，当整个集群所有的node形成一个联通图时，所有节点都可以知道集群中有哪些节点，不会形成孤岛。

官方推荐这里设置为所有的master-eligible node，读者可以想想这样有何好处：

```
It is recommended that the unicast hosts list be maintained as the list of
```

Master选举

上面提到，集群中可能会有多个master-eligible node，此时就要进行master选举，保证只有一个当选master。如果有多个node当选为master，则集群会出现脑裂，脑裂会破坏数据的一致性，导致集群行为不可控，产生各种非预期

的影响。

为了避免产生脑裂，ES采用了常见的分布式系统思路，保证选举出的master被多数派(quorum)的master-eligible node认可，以此来保证只有一个master。这个quorum通过以下配置进行配置：

```
conf/elasticsearch.yml:
    discovery.zen.minimum_master_nodes: 2
```

这个配置对于整个集群非常重要。

1 master选举谁发起，什么时候发起?

master选举当然是由master-eligible节点发起，当一个master-eligible节点发现满足以下条件时发起选举：

1. 该master-eligible节点的当前状态不是master。
2. 该master-eligible节点通过ZenDiscovery模块的ping操作询问其已知的集群其他节点，没有任何节点连接到master。
3. 包括本节点在内，当前已有超过minimum_master_nodes个节点没有连接到master。

总结一句话，即当一个节点发现包括自己在内的多数派的master-eligible节点认为集群没有master时，就可以发起master选举。

2 当需要选举master时，选举谁?

首先是选举谁的问题，如下面源码所示，选举的是排序后的第一个MasterCandidate(即master-eligible node)。

```
public MasterCandidate electMaster(Collection<MasterCandidate> candidates)
        assert hasEnoughCandidates(candidates);
        List<MasterCandidate> sortedCandidates = new ArrayList<>(candidates
        sortedCandidates.sort(MasterCandidate::compare);
        return sortedCandidates.get(0);
    }
```

那么是按照什么排序的?

```
public static int compare(MasterCandidate c1, MasterCandidate c2) {
    // we explicitly swap c1 and c2 here. the code expects "better" is lowe
    // list, so if c2 has a higher cluster state version, it needs to come
    int ret = Long.compare(c2.clusterStateVersion, c1.clusterStateVersion);
    if (ret == 0) {
        ret = compareNodes(c1.getNode(), c2.getNode());
    }
    return ret;
}
```

如上面源码所示，先根据节点的clusterStateVersion比较，clusterStateVersion越大，优先级越高。clusterStateVersion相同时，进入compareNodes，其内部按照节点的Id比较(Id为节点第一次启动时随机生成)。

总结一下：

1. 当clusterStateVersion越大，优先级越高。这是为了保证新Master拥有最新的clusterState(即集群的meta)，避免已经commit的meta变更丢失。因为Master当选后，就会以这个版本的clusterState为基础进行更新。(一个例外是集群全部重启，所有节点都没有meta，需要先选出一个master，然后master再通过持久化的数据进行meta恢复，再进行meta同步)。
2. 当clusterStateVersion相同时，节点的Id越小，优先级越高。即总是倾向于选择Id小的Node，这个Id是节点第一次启动时生成的一个随机字符串。之所以这么设计，应该是为了让选举结果尽可能稳定，不要出现都想当master而选不出来的情况。

3 什么时候选举成功?

当一个master-eligible node(我们假设为Node_A)发起一次选举时，它会按照上述排序策略选出一个它认为的master。

- 假设Node_A选Node_B当Master：

Node_A会向Node_B发送join请求，那么此时：

(1) 如果Node_B已经成为Master，Node_B就会把Node_A加入到集群中，然后发布最新的cluster_state, 最新的cluster_state就会包含Node_A的信息。相当于一次正常情况的新节点加入。对于Node_A，等新的cluster_state发布到

Node_A的时候，Node_A也就完成join了。

⑵ 如果Node_B在竞选Master，那么Node_B会把这次join当作一张选票。对于这种情况，Node_A会等待一段时间，看Node_B是否能成为真正的Master，直到超时或者有别的Master选成功。

⑶ 如果Node_B认为自己不是Master(现在不是，将来也选不上)，那么Node_B会拒绝这次join。对于这种情况，Node_A会开启下一轮选举。

- 假设Node_A选自己当Master：

此时NodeA会等别的node来join，即等待别的node的选票，当收集到超过半数的选票时，认为自己成为master，然后变更cluster_state中的master node为自己，并向集群发布这一消息。

有兴趣的同学可以看看下面这段源码：

```
if (transportService.getLocalNode().equals(masterNode)) {
            final int requiredJoins = Math.max(0, electMaster.minimumMaster
        logger.debug("elected as master, waiting for incoming joins ([{
        nodeJoinController.waitToBeElectedAsMaster(requiredJoins, maste
                new NodeJoinController.ElectionCallback() {
                    @Override
                    public void onElectedAsMaster(ClusterState state) {
                        synchronized (stateMutex) {
                            joinThreadControl.markThreadAsDone(currentT
                        }
                    }

                    @Override
                    public void onFailure(Throwable t) {
                        logger.trace("failed while waiting for nodes to
                        synchronized (stateMutex) {
                            joinThreadControl.markThreadAsDoneAndStartN
                        }
                    }
                }

        );
    } else {
        // process any incoming joins (they will fail because we are no
        nodeJoinController.stopElectionContext(masterNode + " elected")
```

```
        // send join request
        final boolean success = joinElectedMaster(masterNode);

        synchronized (stateMutex) {
            if (success) {
                DiscoveryNode currentMasterNode = this.clusterState().g
                if (currentMasterNode == null) {
                    // Post 1.3.0, the master should publish a new clus
                    // a valid master.
                    logger.debug("no master node is set, despite of joi
                    joinThreadControl.markThreadAsDoneAndStartNew(curre
                } else if (currentMasterNode.equals(masterNode) == fals
                    // update cluster state
                    joinThreadControl.stopRunningThreadAndRejoin("maste
                }

                joinThreadControl.markThreadAsDone(currentThread);
            } else {
                // failed to join. Try again...
                joinThreadControl.markThreadAsDoneAndStartNew(currentTh
            }
        }
    }
```

按照上述流程，我们描述一个简单的场景来帮助大家理解：

假如集群中有3个master-eligible node，分别为Node_A、 Node_B、 Node_C，选举优先级也分别为Node_A、Node_B、Node_C。三个node都认为当前没有master，于是都各自发起选举，选举结果都为Node_A(因为选举时按照优先级排序，如上文所述)。于是Node_A开始等join(选票)，Node_B、Node_C都向Node_A发送join，当Node_A接收到一次join时，加上它自己的一票，就获得了两票了(超过半数)，于是Node_A成为Master。此时cluster_state(集群状态)中包含两个节点，当Node_A再收到另一个节点的join时，cluster_state包含全部三个节点。

4 选举怎么保证不脑裂?

基本原则还是多数派的策略，如果必须得到多数派的认可才能成为Master，那么显然不可能有两个Master都得到多数派的认可。

上述流程中，master候选人需要等待多数派节点进行join后才能真正成为master，就是为了保证这个master得到了多数派的认可。但是我这里想说的

是，上述流程在绝大部份场景下没问题，听上去也非常合理，但是却是有bug的。

因为上述流程并没有限制在选举过程中，一个Node只能投一票，那么什么场景下会投两票呢？比如NodeB投NodeA一票，但是NodeA迟迟不成为Master，NodeB等不及了发起了下一轮选主，这时候发现集群里多了个Node0，Node0优先级比NodeA还高，那NodeB肯定就改投Node0了。假设Node0和NodeA都处在等选票的环节，那显然这时候NodeB其实发挥了两票的作用，而且投给了不同的人。

那么这种问题应该怎么解决呢，比如raft算法中就引入了选举周期(term)的概念，保证了每个选举周期中每个成员只能投一票，如果需要再投就会进入下一个选举周期，term+1。假如最后出现两个节点都认为自己是master，那么肯定有一个term要大于另一个的term，而且因为两个term都收集到了多数派的选票，所以多数节点的term是较大的那个，保证了term小的master不可能commit任何状态变更(commit需要多数派节点先持久化日志成功，由于有term检测，不可能达到多数派持久化条件)。这就保证了集群的状态变更总是一致的。

而ES目前(6.2版本)并没有解决这个问题，构造类似场景的测试case可以看到会选出两个master，两个node都认为自己是master，向全集群发布状态变更，这个发布也是两阶段的，先保证多数派节点“接受”这次变更，然后再要求全部节点commit这次变更。很不幸，目前两个master可能都完成第一个阶段，进入commit阶段，导致节点间状态出现不一致，而在raft中这是不可能的。那么为什么都能完成第一个阶段呢，因为第一个阶段ES只是将新的cluster_state做简单的检查后放入内存队列，如果当前cluster_state的master为空，不会对新的clusterstate中的master做检查，即在接受了NodeA成为master的cluster_state后(还未commit)，还可以继续接受NodeB成为master的cluster_state。这就使NodeA和NodeB都能达到commit条件，发起commit命令，从而将集群状态引向不一致。当然，这种脑裂很快会自动恢复，因为不一致发生后某个master再次发布cluster_state时就会发现无法达到多数派条件，或者是发现它的follower并不构成多数派而自动降级为candidate等。

这里要表达的是，ES的ZenDiscovery模块与成熟的一致性方案相比，在某些特殊场景下存在缺陷，下一篇文章讲ES的meta变更流程时也会分析其他的ES无法满足一致性的场景。

错误检测

1. MasterFaultDetection与NodesFaultDetection

这里的错误检测可以理解为类似心跳的机制，有两类错误检测，一类是Master定期检测集群内其他的Node，另一类是集群内其他的Node定期检测当前集群的Master。检查的方法就是定期执行ping请求。ES文档：

```
There are two fault detection processes running. The first is by the master
```

如果Master检测到某个Node连不上了，会执行removeNode的操作，将节点从cluste_state中移除，并发布新的cluster_state。当各个模块apply新的cluster_state时，就会执行一些恢复操作，比如选择新的primaryShard或者replica，执行数据复制等。

如果某个Node发现Master连不上了，会清空pending在内存中还未commit的new cluster_state，然后发起rejoin，重新加入集群(如果达到选举条件则触发新master选举)。

2. rejoin

除了上述两种情况，还有一种情况是Master发现自己已经不满足多数派条件(>=minimumMasterNodes)了，需要主动退出master状态(退出master状态并执行rejoin)以避免脑裂的发生，那么master如何发现自己需要rejoin呢？

- 上面提到，当有节点连不上时，会执行removeNode。在执行removeNode时判断剩余的Node是否满足多数派条件，如果不满足，则执行rejoin。

```
if (electMasterService.hasEnoughMasterNodes(remainingNodesClusterSt
        final int masterNodes = electMasterService.countMasterNodes
        rejoin.accept(LoggerMessageFormat.format("not enough master
                                                 masterNodes, elect
        return resultBuilder.build(currentState);
    } else {
        return resultBuilder.build(allocationService.deassociateDea
    }
```

- 在publish新的cluster_state时，分为send阶段和commit阶段，send阶段要求多数派必须成功，然后再进行commit。如果在send阶段没有实现多数派返回成功，那么可能是有了新的master或者是无法连接到多数派个节点等，则master需要执行rejoin。

```
try {
        publishClusterState.publish(clusterChangedEvent, electMaster.mi
    } catch (FailedToCommitClusterStateException t) {
        // cluster service logs a WARN message
        logger.debug("failed to publish cluster state version [{}] (not
            newState.version(), electMaster.minimumMasterNodes());

        synchronized (stateMutex) {
            pendingStatesQueue.failAllStatesAndClear(
                new ElasticsearchException("failed to publish cluster s

            rejoin("zen-disco-failed-to-publish");
        }
        throw t;
    }
```

- 在对其他节点进行定期的ping时，发现有其他节点也是master，此时会比较本节点与另一个master节点的cluster_state的version，谁的version大谁成为master，version小的执行rejoin。

```
if (otherClusterStateVersion > localClusterState.version()) {
        rejoin("zen-disco-discovered another master with a new cluster_
    } else {
        // TODO: do this outside mutex
        logger.warn("discovered [{}] which is also master but with an o
        try {
            // make sure we're connected to this node (connect to node
            // since the network connections are asymmetric, it may be
            // in the past (after a master failure, for example)
            transportService.connectToNode(otherMaster);
            transportService.sendRequest(otherMaster, DISCOVERY_REJOIN_

                @Override
                public void handleException(TransportException exp) {
                    logger.warn((Supplier<?>) () -> new ParameterizedMe
                }
            });
        } catch (Exception e) {
            logger.warn((Supplier<?>) () -> new ParameterizedMessage("f
```

```
        }
    }
```

集群扩缩容

上面讲了节点发现、Master选举、错误检测等机制，那么现在我们可以来看一下如何对集群进行扩缩容。

1 扩容DataNode

假设一个ES集群存储或者计算资源不够了，我们需要进行扩容，这里我们只针对DataNode，即配置为：

```
conf/elasticsearch.yml:
    node.master: false
    node.data: true
```

然后需要配置集群名、节点名等其他配置，为了让该节点能够加入集群，我们把discovery.zen.ping.unicast.hosts配置为集群中的master-eligible node。

```
conf/elasticsearch.yml:
    cluster.name: es-cluster
    node.name: node_Z
    discovery.zen.ping.unicast.hosts: ["x.x.x.x", "x.x.x.y", "x.x.x.z"]
```

然后启动节点，节点会自动加入到集群中，集群会自动进行rebalance，或者通过reroute api进行手动操作。

https://www.elastic.co/guide/en/elasticsearch/reference/current/cluster-reroute.html

https://www.elastic.co/guide/en/elasticsearch/reference/current/shards-allocation.html

2 缩容DataNode

假设一个ES集群使用的机器数太多了，需要缩容，我们怎么安全的操作来保

证数据安全，并且不影响可用性呢?

首先，我们选择需要缩容的节点，注意本节只针对DataNode的缩容，MasterNode缩容涉及到更复杂的问题，下面再讲。

然后，我们需要把这个Node上的Shards迁移到其他节点上，方法是先设置allocation规则，禁止分配Shard到要缩容的机器上，然后让集群进行rebalance。

```
PUT _cluster/settings
{
  "transient" : {
    "cluster.routing.allocation.exclude._ip" : "10.0.0.1"
  }
}
```

等这个节点上的数据全部迁移完成后，节点可以安全下线。

更详细的操作方式可以参考官方文档:

https://www.elastic.co/guide/en/elasticsearch/reference/current/allocation-filtering.html

3 扩容MasterNode

假如我们想扩容一个MasterNode(master-eligible node)， 那么有个需要考虑的问题是，上面提到为了避免脑裂，ES是采用多数派的策略，需要配置一个quorum数:

```
conf/elasticsearch.yml:
    discovery.zen.minimum_master_nodes: 2
```

假设之前3个master-eligible node，我们可以配置quorum为2，如果扩容到4个master-eligible node，那么quorum就要提高到3。

所以我们应该先把discovery.zen.minimum_master_nodes这个配置改成3，再扩容master，更改这个配置可以通过API的方式:

```
curl -XPUT localhost:9200/_cluster/settings -d '{
    "persistent" : {
        "discovery.zen.minimum_master_nodes" : 3
    }
}'
```

这个API发送给当前集群的master，然后新的值立即生效，然后master会把这个配置持久化到cluster meta中，之后所有节点都会以这个配置为准。

但是这种方式有个问题在于，配置文件中配置的值和cluster meta中的值很可能出现不一致，不一致很容易导致一些奇怪的问题，比如说集群重启后，在恢复cluster meta前就需要进行master选举，此时只可能拿配置中的值，拿不到cluster meta中的值，但是cluster meta恢复后，又需要以cluster meta中的值为准，这中间肯定存在一些正确性相关的边界case。

总之，动master节点以及相关的配置一定要谨慎，master配置错误很有可能导致脑裂甚至数据写坏、数据丢失等场景。

4 缩容MasterNode

缩容MasterNode与扩容跟扩容是相反的流程，我们需要先把节点缩下来，再把quorum数调下来，不再详细描述。

与Zookeeper、raft等实现方式的比较

1. 与使用Zookeeper相比

本篇讲了ES集群中节点相关的几大功能的实现方式：

1. 节点发现
2. Master选举
3. 错误检测
4. 集群扩缩容

试想下，如果我们使用Zookeeper来实现这几个功能，会带来哪些变化？

Zookeeper介绍

我们首先介绍一下Zookeeper，熟悉的同学可以略过。

Zookeeper分布式服务框架是Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。

简单来说，Zookeeper就是用于管理分布式系统中的节点、配置、状态，并完成各个节点间配置和状态的同步等。大量的分布式系统依赖Zookeeper或者是类似的组件。

Zookeeper通过目录树的形式来管理数据，每个节点称为一个znode，每个znode由3部分组成:

- stat. 此为状态信息, 描述该znode的版本, 权限等信息.
- data. 与该znode关联的数据.
- children. 该znode下的子节点.

stat中有一项是ephemeralOwner，如果有值，代表是一个临时节点，临时节点会在session结束后删除，可以用来辅助应用进行master选举和错误检测。

Zookeeper提供watch功能，可以用于监听相应的事件，比如某个znode下的子节点的增减，某个znode本身的增减，某个znode的更新等。

怎么使用Zookeeper实现ES的上述功能

1. 节点发现：每个节点的配置文件中配置一下Zookeeper服务器的地址，节点启动后到Zookeeper中某个目录中注册一个临时的znode。当前集群的master监听这个目录的子节点增减的事件，当发现有新节点时，将新节点加入集群。
2. master选举：当一个master-eligible node启动时，都尝试到固定位置注册一个名为master的临时znode，如果注册成功，即成为master，如果注册失败则监听这个znode的变化。当master出现故障时，由于是临时znode，会自动删除，这时集群中其他的master-eligible node就会尝试再次注册。使用Zookeeper后其实是把选master变成了抢master。
3. 错误检测：由于节点的znode和master的znode都是临时znode，如果节点故障，会与Zookeeper断开session，znode自动删除。集群的master只需要监听znode变更事件即可，如果master故障，其他的候选master则会监

听到master znode被删除的事件，尝试成为新的master。
4. 集群扩缩容：扩缩容将不再需要考虑minimum_master_nodes配置的问题，会变得更容易。

使用Zookeeper的优劣点

使用Zookeeper的好处是，把一些复杂的分布式一致性问题交给Zookeeper来做，ES本身的逻辑就可以简化很多，正确性也有保证，这也是大部分分布式系统实践过的路子。而ES的这套ZenDiscovery机制经历过很多次bug fix，到目前仍有一些边角的场景存在bug，而且运维也不简单。

那为什么ES不使用Zookeeper呢，大概是官方开发觉得增加Zookeeper依赖后会多依赖一个组件，使集群部署变得更复杂，用户在运维时需要多运维一个Zookeeper。

那么在自主实现这条路上，还有什么别的算法选择吗？当然有的，比如raft。

2. 与使用raft相比

raft算法是近几年很火的一个分布式一致性算法，其实现相比paxos简单，在各种分布式系统中也得到了应用。这里不再描述其算法的细节，我们单从master选举算法角度，比较一下raft与ES目前选举算法的异同点：

相同点

1. 多数派原则：必须得到超过半数的选票才能成为master。
2. 选出的leader一定拥有最新已提交数据：在raft中，数据更新的节点不会给数据旧的节点投选票，而当选需要多数派的选票，则当选人一定有最新已提交数据。在es中，version大的节点排序优先级高，同样用于保证这一点。

不同点

1. 正确性论证：raft是一个被论证过正确性的算法，而ES的算法是一个没有经过论证的算法，只能在实践中发现问题，做bug fix，这是我认为最大的不同。

2. 是否有选举周期term：raft引入了选举周期的概念，每轮选举term加1，保证了在同一个term下每个参与人只能投1票。ES在选举时没有term的概念，不能保证每轮每个节点只投一票。
3. 选举的倾向性：raft中只要一个节点拥有最新的已提交的数据，则有机会选举成为master。在ES中，version相同时会按照NodeId排序，总是NodeId小的人优先级高。

看法

raft从正确性上看肯定是更好的选择，而ES的选举算法经过几次bug fix也越来越像raft。当然，在ES最早开发时还没有raft，而未来ES如果继续沿着这个方向走很可能最终就变成一个raft实现。

raft不仅仅是选举，下一篇介绍meta数据一致性时也会继续比较ES目前的实现与raft的异同。

小结

本篇介绍了Elasticsearch集群的组成、节点发现、master选举、故障检测和扩缩容等方面的实现，与一般的文章不同，本文对其原理、存在的问题也进行了一些分析，并与其他实现方式进行了比较。

作为Elasticsearch分布式一致性原理剖析系列的第一篇，本文先从节点入手，下一篇会介绍meta数据变更的一致性问题，会在本文的基础上对ES的分布式原理做进一步分析。

最后，我们在招人，有兴趣的可以私信联系我。