服务治理与分布式

1. 什么是微服务?
2. 什么是分布式?
3. 简述CAP理论?
4. Base原则是什么?
5. 如何保证数据的最终一致性?
6. MySQL是AP还是CP?
7. Redis是AP还是CP?

一、什么是微服务?

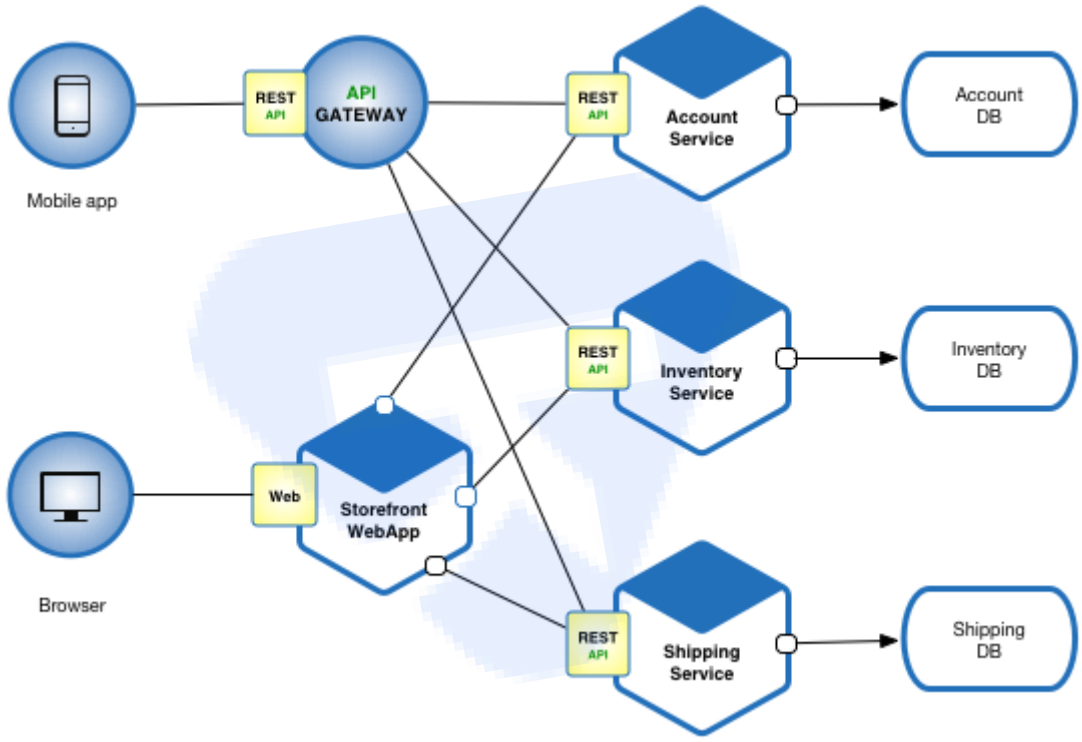
关于什么是微服务， Martin Fowler 【大神】的Paper有如下的定义:

> The term "Microservice Architecture" has sprung up over the last few years to describe a particular way of designing software applications as suites of independently deployable services. **While there is no precise definition of this architectural style**, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.
>
> **In short, the microservice architectural style is** an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

这里提到几个重要的概念：微服务架构的**风格!**

1. **一套小服务**
2. **独立进程**
3. **轻量级通信协议**
4. **可独立部署**
5. **多语言&不同储技术**

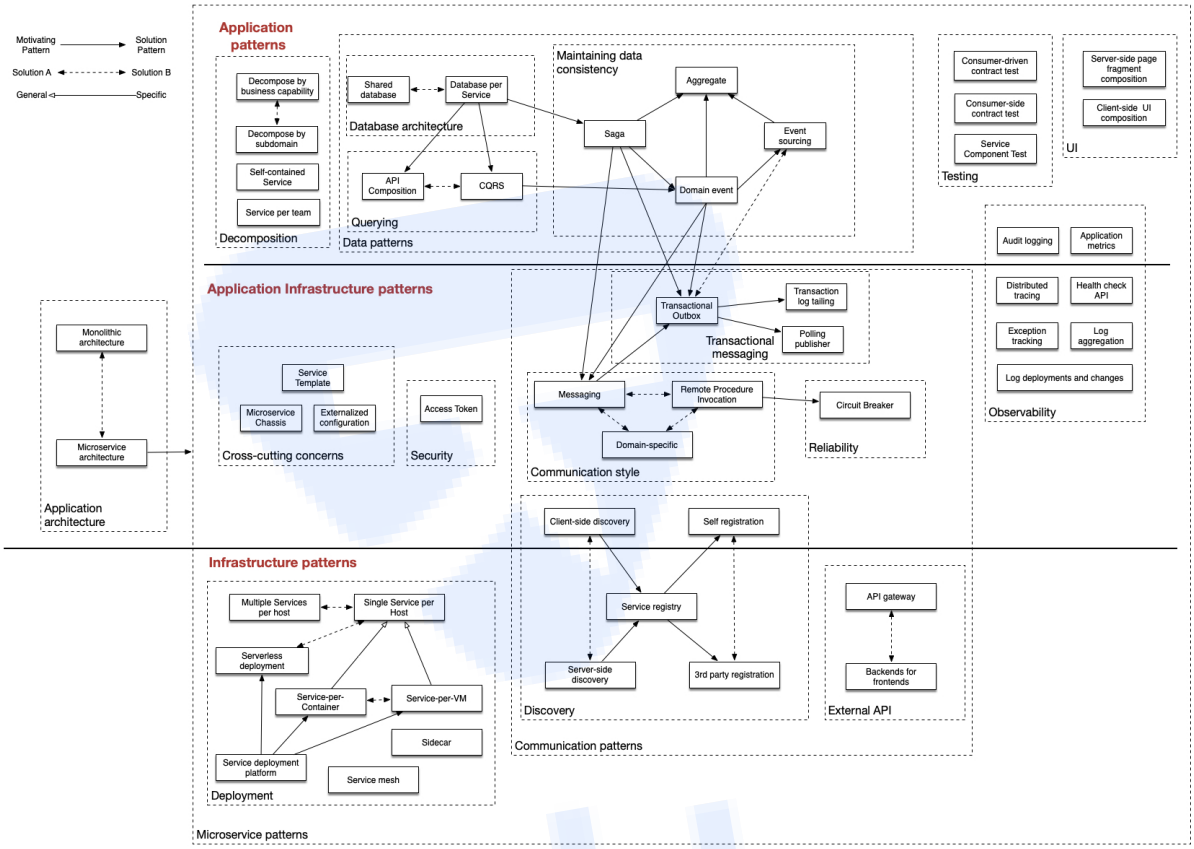
这个定义对微服务做了一个比较具象化较为易于理解的描述， 通常来说我们看到的微服务架构如下图所示： **符合了基本特征，便可以称之为微服务了!**



但是事实上， 在实际生产环境中， 微服务的架构要考虑的问题远比上面的示意图复杂的多， 主要包括但不限于如下问题：

1. **通用服务实现组件化**
2. **根据业务组织系统**
3. **简单高效的通信协议**
4. **自动化基础设施**
5. **面向失败的设计**
6. **具备进化能力的设计**

纵然有 Martin Fowler 这样的大神在前面引路， 但是我们依然认为**“ 微服务” 不是一个被设计出来的架构**， 而是在不断是尝试中总结出的一套适合在互联网行业使用的架构模式。



Learn-Build-Assess Microservices http://adopt.microservices.io

1.1 那么微服务究竟是什么?

- **符合了基本特征，便可以称之为微服务了!**
- **“ 微服务” 不是一个被设计出来的架构**
- 配置中心、注册中心、熔断器、负载均衡器、链路追踪、服务调用....

1.2 微服务并不是银弹!

架构是由什么驱动呢?

- 架构由价值驱动
-
- 由组织架构驱动
- 这个是常态了...自己的机器怎么能把服务端起起来...只能起其中一个组件...

微服务并不是一劳永逸的解决了所有的问题， 相反的， 如果不能正确的使用微服务， 则有可能被微服务自身的限制拖入另一个**泥潭**:

1. **分布式的代价**：原本在单体应用中， 很多简单的问题都会在分布式环境下被几何级的放大。 例如分布式事务、 分布式锁、 远程调用等， 不光要考虑如何实现他们， 相关场景的异常处理也是必须要考虑到的问题。
2. **协同代价**：如果你经历过一个项目上线需要发布十几个应用， 而这些应用又分别由多个团队在维护。 你就能深刻的体会到协同是一件多么痛苦的事情了。
3. **服务拆分需要很强的设计功力**： 微服务的各种优势， 其中一个重要的基础是对服务领域的正确切分。 如果使用了不合适的切分粒度， 或者是错误的切分方法， 都会让服务不能很好的实现高内聚低耦合的要求。

> 讲个案例：

二、什么是分布式?

满足了分布式特征的系统，可能是微服务架构类型。

分布式系统是多个处理机通过通信线路互联而构成的松散耦合的系统。从系统中某台处理机来看，其余的处理机和相应的资源都是远程的，只有它自己的资源才是本地的。至今，对分布式系统的定义尚未形成统一的见解。一般认为，分布式系统应具有以下四个特征：

- **分布性**：分布式系统由多台计算机组成，它们在地域上是分散的，可以散布在一个单位、一个城市、一个国家，甚至全球范围内。整个系统的功能是分散在各个节点上实现的，因而分布式系统具有数据处理的分布性。
- **自治性**：分布式系统中的各个节点都包含自己的处理机和内存，各自具有独立的处理数据的功能。通常，彼此在地位上是平等的，无主次之分，既能自治地进行工作，又能利用共享的通信线路来传送信息，协调任务处理。
- **并行性**：一个大的任务可以划分为若干个子任务，分别在不同的主机上执行。
- **全局性**：分布式系统中必须存在一个单一的、全局的进程通信机制，使得任何一个进程都能与其他进程通信，并且不区分本地通信与远程通信。同时，还应当有全局的保护机制。系统中所有机器上有统一的系统调用集合，它们必须适应分布式的环境。在所有CPU上运行同样的内核，使协调工作更加容易。

> 理解下分布式计算这个词就能对上了

> 分布式 和 集群【两个及以上的节点，都可以称之为集群】 这两个概念是什么关系

2.1 分布式共识原理

共识是分布式容错系统的基本问题。共识解决的是多个服务节点对数据达成一致的问题。

> Raft 算法动画演示：http://thesecretlivesofdata.com/raft/
>
> Raft 算法动画演示：https://raft.github.io/raftscope/index.html

为什么要对数据进行共识?

- 需要共识的数据一般不大
- ElasticSearch对什么数据进行共识呢？索引共识
- Nacos对什么数据进行共识呢？**Discovery服务列表**

2.2 CAP和BASE原则

2.2.1 CAP定理

CAP 定理：**在分布式系统中，** Consistency、 Availability、Partition tolerance。在CAP中， C、 A二者不可兼得!

对于一个分布式系统来说，CAP三项只可能满足两项，即要么 CP 强一致性，要么 AP强可用性。

对于分布式系统，出现网络分区是不可避免的，因此系统必须具备分区容错性。但其并不能同时保证一致性与可用性。

- **一致性（C）：**
  - 分布式系统中多个主机之间是否能够保持数据一致的特性。
  - 当系统数据发生更新操作后，各个主机中的数据仍然处于一致的状态。
- **可用性（A）：**
  - 系统提供的服务必须一直处于可用的状态
  - 对于用户的每一个请求，系统总是可以在有限的时间内对用户做出响应。
- **分区容错性（P）：** 分布式系统在遇到任何网络分区故障时，仍能够保证对外提供满足一致性和可用性的服务。

> 抛开事实谈论点都是耍流氓。
>
> CAP保障的是什么?

2.2.2 常见服务CAP特性分析

1. **MySQL是CP还是AP?**
   - 默认情况下：MySQL是单体
   - 主从同步：
     - 场景1：单点问题【容错】，读写(主)，从同步【既不是CP也不是AP】
     - 场景2：数据并发能力，写(主)，读是在从【AP】
   - 分库分表：扩容与性能优化
   - 可C可A，至于需要C还是需要A，看具体需求
2. **Redis是CP还是AP?**
   - 默认情况下：Redis单体
   - 主从复制：可C可A，至于需要C还是需要A，看具体需求
   - Sentinel模式：哨兵模式与CAP没有一点关系
   - 集群模式：扩容与性能优化
3. **ElasticSearch是CP还是AP?**
   - 默认情况下：与生俱来支持分布式的，单体
   - 既不是CP也不是AP：
     - 前提是针对索引库来说，因为索引的数据【TB，分片10GB】并不需要共识
     - 很多节点，ES内部服务列表【Naming List】，每个节点元数据
   - 一致性、可用性较高，低分区容错性系统【脑裂绝对是使用不当造成】
   - 可C可A，至于需要C还是需要A，看具体需求
4. **Nacos是CP还是AP?**
   - 默认是AP的，可以手动切换为CP!
   - 数据：服务列表

5. **Eureka是CP还是AP?**
    - AP
    - 数据：服务列表
6. **Zookeeper 是CP还是AP?**
    - CP
    - 数据：目录

C --> A

2.2.3 BASE 理论

BASE是Basically Available、 Soft state和 Eventually consistent三个短语的简写。 BASE理论是分布式系统中，CAP 定理对于一致性与可用性权衡的结果。

核心思想是：**即使无法做到强一致性，但每个系统都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性。**而且满足基本可用条件，及存在软状态的可能性。

1. Basically Available：
    - 基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性。响应时间的损失、功能上的损失
    - 读写不报错
2. Soft state：
    - 允许系统数据存在的中间状态，并认为该中间状态的存在不会影响系统的整体可用性
    - 允许系统主机间进行数据同步的过程存在一定延时
    - 其实就是一种灰度状态，过渡状态
3. Eventually consistent：
    - 强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。
    - 本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

2.3 解决共识问题方案：Raft 算法

- Paxos
- Bully
- Nacos自研
- ...自己实现

2.3.1 什么是Raft算法?

Raft算法是一种非常易于理解的分布式共识算法。Raft算法可以保障分布式系统数据的一致性，**解决分布式容错系统的共识问题。在容错性与性能表现方面等同于老牌的复杂的Paxos共识算法。**

Raft 算法：是一个最终一致性的算法，不是强一致性算法。

> 参考文件：RaftAlgorithm.pdf

2.3.2 角色、任期及角色转变

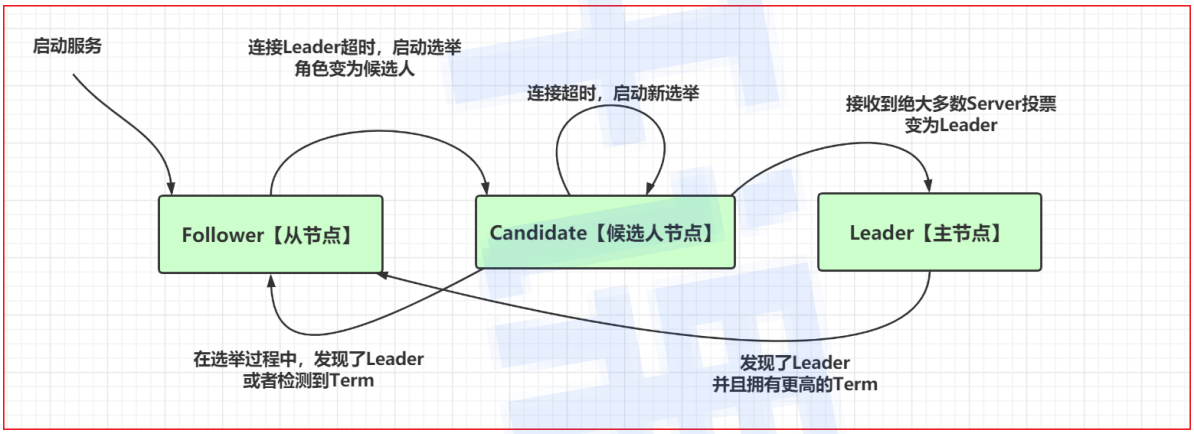
> 为什么要学主节点?

注意：主节点有写入数据的权限。对系统的所有更改，都需要经过Leader节点。从节点没有权限，从节点只有读取权限！

在 Raft 中，节点有三种角色：

- **Leader【主】**：唯一负责处理客户端**写请求**的节点；也可以处理客户端读请求；同时负责日志复制工作
- **Candidate【候选人】**：Leader 选举的候选人，其可能会成为 Leader
- **Follower【从】**：可以处理客户端读请求；负责同步来自于 Leader 的日志；当接收到其它 Cadidate 的投票请求后可以进行投票；当发现 Leader 挂了，其会转变为 Candidate 发起Leader 选举
- **Term：【任期/年号】**什么是Term呢?

2.3.3 leader选举流程



- **（1） 成为候选人**
- **（2） follower投票**
- **（3） Candidate 等待响应**
- **（4） Candidate 获取半数以上投票【活动节点】成为主节点**，可以防止脑裂的

2.3.4 Raft算法下集群脑裂情况【普遍】

> Raft算法自己就解决了网络分区的问题，基于领导选举及日志复制的机制来解决的！

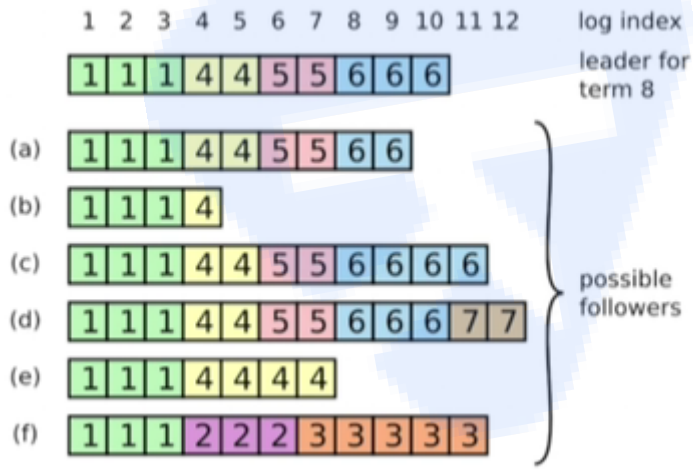
什么是脑裂?

> 在一个高可用系统中，当联系着的节点断开联系时，本来为一个整体的系统，分裂成两个独立节点，两个节点开始争抢共享资源造成系统混乱、数据损坏的现象，称为“脑裂”。网络分区会导致脑裂发生。

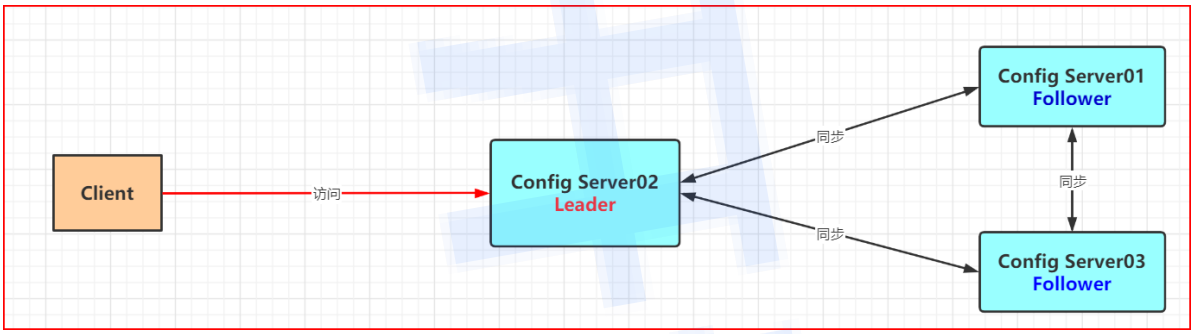
Raft 集群存在脑裂问题。在多机房部署中，由于网络连接问题，很容易形成多个分区。而多分区的形成，很容易产生脑裂，从而导致数据不一致。

Raft如何应对脑裂?

1. 只有主节点可以写入数据
2. 写入数据的条件是，半数以上节点确认，提交数据
3. 日志复制机制



2.3.5 保证数据最终的一致性



- （1）情况1：请求到达前Leader挂了
- （2）情况2：未开始同步数据前Leader挂了
- （3）情况3：同步完部分后Leader挂了
- （4）情况4：apply通知发出后Leader挂了

领导人完全原则（Leader Completeness Property）：如果一个日志条目在一个给定任期内被提交，那么这个条目一定会出现在所有任期号更大的领导人中。保证了领导人一定拥有所有已经被提交的日志条目。

> 讨论nacos

三、分布式全文检索引擎

1. 全文检索和倒排索引是什么?
2. ES集群shard和replicas有什么作用?
3. 分片应该分几片?
4. 分片数量是否可以更改?

5. ES启动的时候都做什么事情?
6. ES集群的Master选举是怎么实现的?

01-什么是全文检索

全文检索是利用**倒排索引**技术对需要搜索的数据进行处理，然后提供快速匹配的技术。其实全文检索还有另外一种专业定义，**先创建索引然后对索引进行搜索的过程**，就是全文检索。

1.1 倒排索引

倒排索引是一种存储数据的方式，与传统查找有很大区别：

- 传统查找：采用数据按行存储，查找时逐行扫描，或者根据索引查找，然后匹配搜索条件，效率较差。概括来讲是先找到文档，然后看是否匹配。查找一个10MB的word文档，大概需要3秒
- 倒排索引：首先对数据按列拆分存储，然后对文档中的数据分词，对词条进行索引，并记录词条在文档中出现的位置。这样查找时只要找到了词条，就找到了对应的文档。概括来讲是先找到词条，然后看看哪些文档包含这些词条。

1.2 创建倒排索引流程

当我们需要把这些数据创建倒排索引时，会分为两步：

1）创建文档列表

首先将数据按列进行拆分存储，类型于mysql的表存储，每一条数据，就是一个文档，形成文档列表：

文档表			
文档id	标题(title)	内容(content)	得分(Score)
1	开课吧高级架构师	牛逼的架构课程	100
2	开课吧企业开发工程师	牛逼的开发工程师课程	100
3	开课吧百万架构	开课吧百万架构牛逼	100
4	…	…	…
5	…	…	…
6	…	…	…

2）创建倒排索引列表

然后对文档中的数据进行分词，得到词条。对词条进行编号，并以词条创建索引。然后记录下包含该词条的所有文档编号（及其它信息）。

倒排索引表			
词ID	词典	倒排索引	
		文档id	文档列名
A1	开课吧	1,2,3	title
B2	开发	2	title
C3	工程师	2	title
D4	开发工程师	2	title
E5	百万架构	3	title
F6	高级架构师	1	title
G7	架构师	1	title
H8	企业开发	2	title

流程如下：

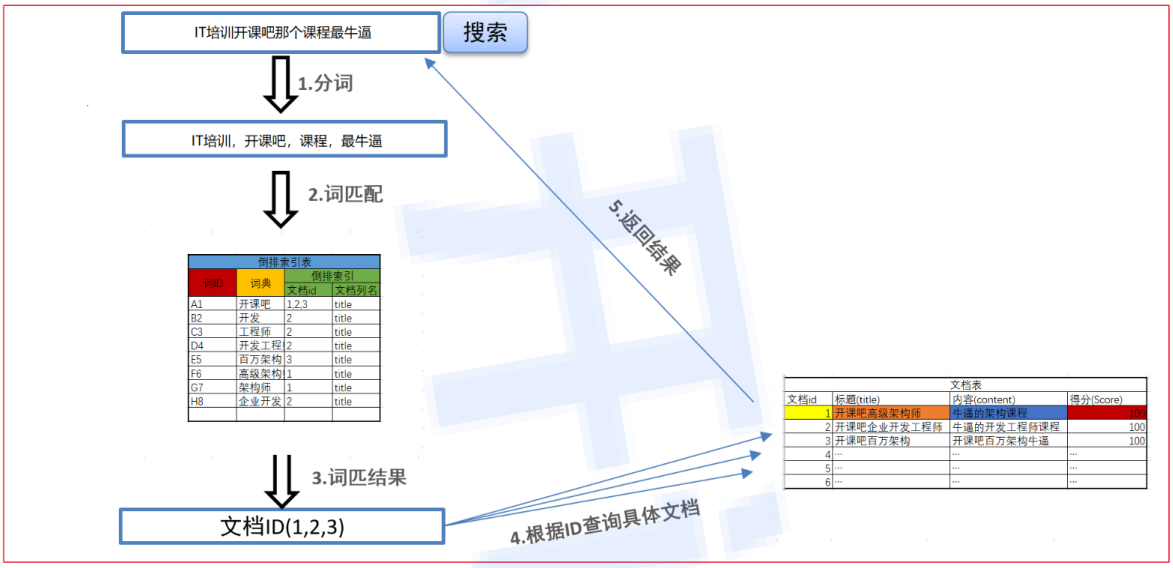
文档表			
文档id	标题(title)	内容(content)	得分(Score)
1	开课吧高级架构师	牛逼的架构课程	100
2	开课吧企业开发工程师	牛逼的开发工程师课程	100
3	开课吧百万架构	开课吧百万架构牛逼	100
4	…	…	…
5	…	…	…
6	…	…	…

倒排索引表			
词ID	词典	倒排索引	
		文档id	文档列名
A1	开课吧	1,2,3	title
B2	开发	2	title
C3	工程师	2	title
D4	开发工程	2	title
E5	百万架构	3	title
F6	高级架构	1	title
G7	架构师	1	title
H8	企业开发	2	title

1.3 搜索流程

搜索的基本流程：

- 当用户输入任意的搜索关键词时，首先对用户输入的内容进行词拆分，得到要搜索的所有词条，如用户搜索“java培训开课吧”，拆分后就是“java 、培训、开课吧”，
- 然后拿着这些拆分后的词去倒排索引列表中进行匹配。找到这些词对应的所有文档编号
- 然后根据这些编号去文档列表中找到文档



1.4 什么是ElasticSearch

- 开源的**高扩展的分布式全文检索引擎**
- 近乎实时的检索数据；
- 本身扩展性很好，可以扩展到上百台服务器，处理PB级别的数据。
- ES使用Java开发【早期是用Maven构建，后期Gradle】
- 核心是Lucene

02-ElasticSearch集群shard与replicas机制

- **分片应该分几片?**
- **分片数量是否可以更改?**

2.1 为什么分片?

在分布式系统中，单机无法存储规模巨大的数据，要依靠大规模集群处理和存储这些数据，一般通过增加机器数量来提高系统水平扩展能力。因此，需要将数据分成若干小块分配到各个机器上。然后通过某种路由策略找到某个数据块所在的位置。

2.2 为什么做副本?

Redis主从复制，MySQL主从复制

1TB -> 2TB

除了将数据分片以提高水平扩展能力，分布式存储中还会把数据复制成多个副本，放置到不同的机器中，这样一来可以增加系统可用性，同时数据副本还可以使读操作并发执行，分担集群压力。

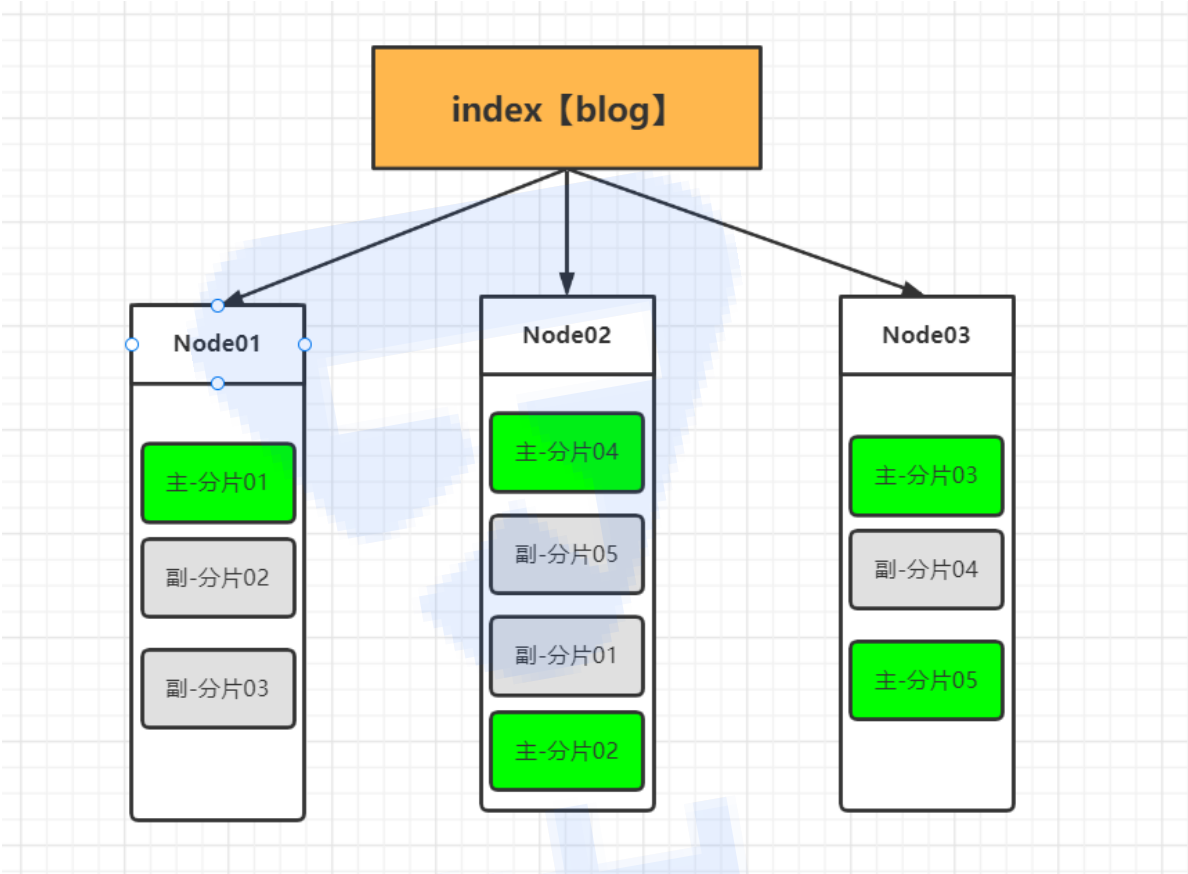
副本带来的弊病：数据的一致性问题!

但是多数据副本也带来了一致性的问题：部分副本写成功，部分副本写失败。

为了应对并发更新问题，ES将数据副本分为主从两部分，即主分片（primary shard）和副分片（replica shard）。

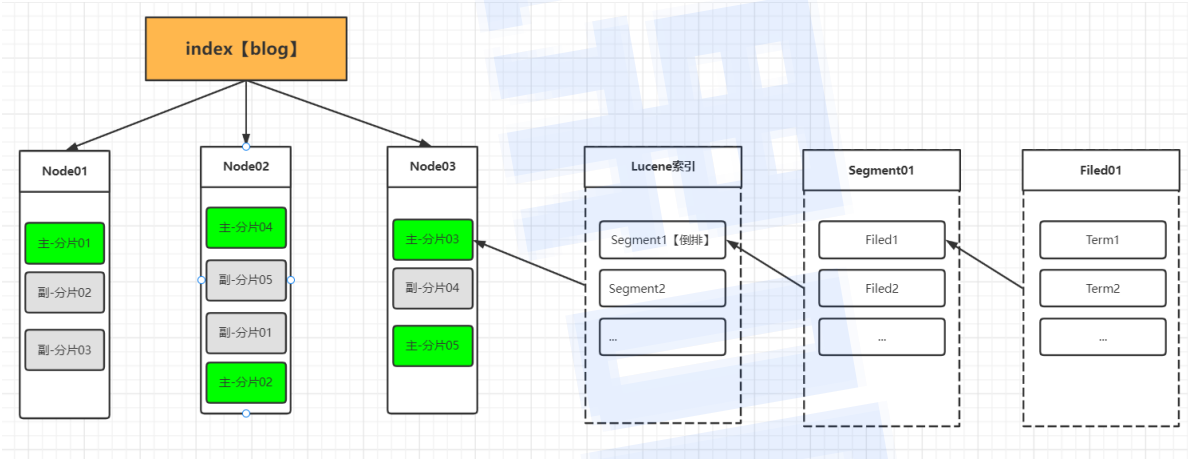
主数据作为权威数据，写过程中先写主分片，成功后再写副分片，恢复阶段以主分片为准。

2.3 索引与分片的基本结构:



- 分片（shard）是底层的基本读写单元。分片是数据容器，文档保存在分片内，不会跨分片存储
- 分片的目的是分割巨大索引：
  - 让读写可以并行操作，由多台机器共同完成，读写请求最终落到某个分片上，分片独立执行读写操作
  - ES利用分片将数据分散到集群内各处
- 当集群规模扩大或缩小时：ES 会自动在各节点中迁移分片，使数据仍然均匀分布在集群里

2.4 索引基本构成：



- 一个ES的index包含很多分片，一个分片是一个Lucene的索引，它本身就是一个完整的搜索引擎，可以独立执行建立索引和搜索任务。
- Lucene索引又由很多分段组成，每个分段都是一个倒排索引。
- ES每次“refresh”都会生成一个新的分段，其中包含若干文档的数据。
- 在每个分段内部，文档的不同字段被单独建立索引。
- 每个字段的值由若干词（Term）组成，Term是原文本内容经过分词器处理和语言处理后的最终结果（例如，去除标点符号和转换为词根）。

> Lucene分段文件组成详解：http://lucene.apache.org/core/7_3_0/core/org/apache/lucene/codecs/lucene70/package-summary.html#package.description。

2.5 分片应该怎么分?

索引建立的时候就需要确定好主分片数。

单个分片大小不要超过30GB【上限】，分片数量最好不要太多。决定分片大小的更多是基于硬件和业务场景的预估数据量和增量情况。

- 5.x 版本之前：主分片数量不可以修改，副分片数可以随时修改。
- 5.x 版本之后：支持在一定条件的限制下，对某个索引的主分片进行拆分（Split）或缩小（Shrink）。主分片数量最好是提前估算好，后期调整容易出现严重后果。
  - **分片数不够时，可以考虑新建索引**：搜索1个有着50个分片的索引与搜索50个每个都有1个分片的索引完全等价
  - **也可以使用_split API**来拆分索引（6.1版本开始支持）。

2.6 索引巨大怎么办?

虽说单个分片可以达到30GB，但在实际应用中，我们不应该向单个索引持续写数据，直到它的分片巨大无比。

- 巨大的索引的问题：数据老化后难以删除
  - 以id 为单位删除文档不会立刻释放空间，删除的 doc 只在 Lucene分段合并时才会真正从磁盘中删除
  - 即使手工触发分段合并，仍然会引起较高的 I/O 压力，并且可能因为分段巨大导致在合并过程中磁盘空间不足

一个小小的建议：周期性地创建新索引。

例如，每天创建一个。假如有一个索引ITxiongge_shop，可以将它命名为ITxiongge_shop_20211031。然后创建一个名为ITxiongge_shop的索引别名来关联这些索引。

这样，对于业务方来说，读取时使用的名称不变，当需要删除数据的时候，可以直接删除整个索引。

索引别名就像一个快捷方式或软链接，不同的是它可以指向一个或多个索引。可以用于实现索引分组，或者索引间的无缝切换。

2.7 分片数量过多怎么办?

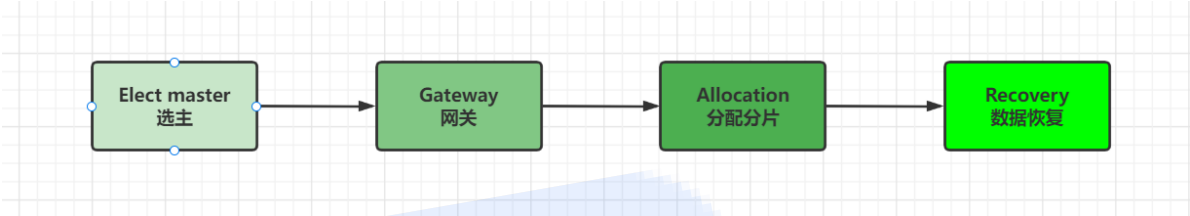
现在我们已经确定好了主分片数量，并且保证单个索引的数据量不会太大，周期性创建新索引带来的一个新问题是集群整体分片数量较多，集群管理的总分片数越多压力就越大。

在每天生成一个新索引的场景中，可能某天产生的数据量很小，实际上不需要这么多分片，甚至一个就够。这时，可以使用shrink index API来缩减主分片数量，降低集群负载。

03-ElasticSearch集群启动过程

> 理解原理对于解决或避免集群维护过程中可能遇到的脑裂、无主、恢复慢、丢数据等问题很有帮助。

ElasticSearch集群启动主要流程：



1. Elect-master：
    - 选出临时Master【参选人数过半】
    - 确定Master【得票过半】
    - 检测集群节点数【离开节点】
2. Gateway：
    - Master 索取MetaState
    - 选举元信息，发布元信息
    - 参与元信息选举的节点数过半
3. Allocation
    - 向所有节点询问shard信息
    - 磁盘且in-sync列表存在为主分片
    - 磁盘存在选为副本否则延迟分配
4. Recovery
    - 两阶段恢复
    - translog事件日志

3.1 Elect-master

集群选举，集群启动的第一件事：从已知的活跃机器列表中选择一个作为主节点。选主之后的流程由主节点触发。

选主算法：基于Bully算法进行了改进。

主要思路：对节点ID排序，取ID值最大的节点作为Master，每个节点都运行这个流程。简单来说就是基于节点ID排序的简单选举算法。

> 为什么不使用Raft算法，为什么不使用Paxos算法?

选主目的：确定唯一的主节点

3.2 Gateway

选举元信息，被选出的 Master 和集群元信息的新旧程度没有关系。因此它的第一个任务是选举元信息，让各节点把各自存储的元信息发过来，根据版本号确定最新的元信息，然后把这个信息广播下去，这样集群的所有节点都有了最新的元信息。

集群元信息的选举包括两个级别：集群级和索引级。不包含哪个shard存于哪个节点这种信息。这种信息以节点磁盘存储的为准，需要上报。为什么呢？因为读写流程是不经过Master的，Master 不知道各 shard 副本直接的数据差异。

集群元信息选举完毕后，Master发布首次集群状态，然后开始选举shard级元信息。选举shard级元信息，构建内容路由表，是在allocation模块完成的。

3.3 allocation

在初始阶段，所有的shard都处于UNASSIGNED（未分配）状态。

ES中通过**分配过程**决定哪个分片位于哪个节点，重构内容路由表。此时，首先要做的是分配主分片。

3.3.1 分配主分片

> 主分片如何分配?

Master节点向集群中所有节点询问：所有节点将分片的元信息返回给Master，根据返回的 shard 信息，配合主节点内的**分配策略**选一个分片作为主分片。

这种做法效率怎么样?

- 询问量=shard 数×节点数。所以说我们最好控制shard的总规模别太大。有了shard的分片的多份信息，剩下的就是选出主分片。

主分片选举方式:

- ES 5.x之前，通过对比shard级元信息的版本号来决定
  - 潜在问题：在多副本的情况下，考虑到如果只有一个 shard 信息汇报上来，则它一定会被选为主分片，但数据不一定是最新的。可能版本号比它大的那个shard所在节点还没启动。
- ES 5.x之后：通过集群级元信息中记录的"最新主分片列表"确定主分片【汇报信息中存在，并且这个列表中也存在】
  - 给每个 shard 都设置一个 UUID，然后在集群级的元信息中记录哪个shard是最新的。
  - 因为ES是先写主分片，再由主分片节点转发请求去写副分片，所以主分片所在节点肯定是最新的。

> 主分片数量取决于什么?

3.3.2 分配副分片

- 主分片选举完成后，从上一个过程汇总的 shard 信息中选出副本。
- 如果汇总信息中不存在，则分配一个全新副本。分配的时间依赖于延迟配置项：index.unassigned.node_left.delayed_timeout

> tip：我们的生产环境中最大的集群有100+节点，掉节点的情况并不罕见，很多时候不能第一时间处理，这个延迟我们一般配置为以天为单位。

allocation过程中允许新启动的节点加入集群。分片分配成功后进入recovery流程。

3.4 recovery

为什么需要recovery?

- 对于主分片来说，可能有一些数据没来得及刷盘
- 对于副分片来说，一是没刷盘，二是数据的不一致【主分片写完了，副分片还没来得及写，主副分片】

3.4.1 主分片recovery

将最后一次提交之后的 translog重放，建立Lucene索引，如此完成主分片的recovery

- 每次写操作都会记录translog【事务日志中记录了哪种操作以及相关数据】。
- Lucene 的一次提交就是一次 fsync 刷盘的过程

3.4.2 副分片recovery

在ES的版本迭代中，副本分片的恢复策略有过不少调整，比较复杂。

副分片需要恢复成与主分片一致，同时，恢复期间还需要允许新的索引操作。

在6.0版本中，恢复分成两个阶段执行：

- **phase1：**
  - 在主分片所在节点，获取translog保留锁，从获取保留锁开始，会保留translog不受其刷盘清空的影响。
  - 然后调用Lucene接口把已经刷磁盘中的shard做快照。然后把这些shard数据复制到副本节点。
  - 在phase1完毕前，会向副分片节点发送告知对方启动engine
  - 在phase2开始前，副分片就可以正常处理写请求了。
- **phase2：**
  - 对translog做快照，这个快照里包含从phase1开始，到执行translog快照期间的新增索引。
  - 将这些translog发送到副分片所在节点进行重放。
  - 思考：第二阶段运行期间，如果刷盘并清空tranlog怎么办?

3.4.3 恢复需重点关注4个问题

> 由于需要支持恢复期间的新增写操作（让ES的可用性更强），需要重点关注以下几个问题。

1. **分片数据完整性：**如何做到副分片不丢数据?
   - 第二阶段的 translog 快照包括第一阶段所有的新增操作。那么第一阶段执行期间如果发生“Lucene commit”（将文件系统写缓冲中的数据刷盘，并清空translog），清除translog怎么办?
     - 在ES 2.0之前：阻止了刷新操作，以此让translog都保留下来。
     - 从ES 2.0之后：为了避免这种做法产生过大的translog，引入了translog.view的概念，创建 view 可以获取后续的所有操作。
     - 从ES 6.0之后：translog.view 被移除，引入TranslogDeletionPolicy的概念，它将translog做一个快照来保持translog不被清理。这样实现了在第一阶段允许Lucene commit。
2. **数据一致性：**
   - 在ES 2.0之前：副分片恢复过程其实是三个阶段，第三阶段会阻塞新的索引操作，传输第二阶段执行期间新增的translog，这个时间很短。
   - 在ES2.0之后：第三阶段被删除，恢复期间没有任何写阻塞过程。但是在副分片节点，重放translog时，phase1和phase2之间的写操作与phase2重放操作之间的时序错误和冲突【极少】。
     - 通过写流程中异常处理，和对比版本号来过滤掉过期操作
     - 对于特定的 doc，只有最新一次操作生效，保证了主副分片的数据一致性

3. **第一阶段操作优化：**第一阶段尤其漫长，因为它需要从主分片拉取全量的数据。那可是全量数据呀！
    - 索引数据恢复是最漫长的过程。当shard总量达到十万级的时候，6.x之前的版本集群从Red变为Green的时间可能需要小时级。
    - 在ES 6.x之后：增量同步，对第一阶段再次优化，标记每个操作。
    - 在正常的写操作中，每次写入成功的操作都分配一个序号，通过对比序号就可以计算出差异范围然后同步。
    - ES 6.x中的translog恢复是一次重大改进，避免了从主分片所在节点拉取全量数据，为恢复过程节约了大量时间。
4. **集群启动过程中，集群健康值变化含义**

- 当**一个索引的主分片**分配成功后，到此分片的写操作就是允许的。
- 当**一个索引所有的主分片**都分配成功后，该索引变为Yellow。
- 当**全部索引的主分片**都分配成功后，整个集群变为Yellow。
- 当**一个索引全部分片**分配成功后，该索引变为 Green。
- 当**全部索引的索引分片**分配成功后，整个集群变为Green。

04-ElasticSearch集群Master选举机制

Discovery模块：负责发现集群中的节点，以及选择主节点。ES支持多种不同Discovery类型选择，内置的实现称为Zen Discovery。

Zen Discovery封装了**节点发现（Ping）**、**选主**等实现过程。

Master选举核心设计思想：所有分布式系统都需要以某种方式处理数据一致性问题。一般情况下，可以将策略分为两种：

1. 避免数据不一致【ES】
2. 事后处理不一致数据【事后补偿】，适用场景下非常强大，但对数据模型有比较严格的限制。

Elastic选出主节点Master，Master的职责是什么？这里采用的共识算法主要在共识什么数据？保证什么数据的一致性！

4.1 选举采用什么算法?

在主节点选举算法的选择上，基本原则是不重复造轮子。最好实现一个众所周知的算法，这样的好处是其中的优点和缺陷是已知。

1. Bully算法
    - 假定所有节点都有一个唯一的ID，使用该ID对节点进行排序。任何时候的当前Leader都是参与集群的最高ID节点。
    - **优点：易于实现**
    - 缺点：当拥有最大ID的节点处于不稳定状态的场景下会有问题。

        - 例如，Master负载过重而假死，集群拥有第二大ID的节点被选为新主，这时原来的Master恢复，再次被选为新主，然后又假死……
        - ES 通过推迟选举，直到当前的 Master 失效来解决上述问题，只要当前主节点不挂掉，就不重新选主。
        - 但是容易产生脑裂（双主），为此，再通过"法定得票人数过半"解决脑裂问题。
2. Paxos算法
    - Paxos非常强大，尤其在什么时机，以及如何进行选举方面的灵活性比简单的Bully算法有很大的优势，因为在现实生活中，存在比网络连接异常更多的故障模式。
    - Paxos 实现起来非常复杂。
3. Raft算法

4.2 选举的三条件与一配置：

三个条件：

1. **参选人数过半**：达到 quorum（多数）后就选出了临时的主
    - **为什么是临时的？**每个节点运行排序取最大值的算法，结果不一定相同。
    - 举个例子，集群有5台主机，节点ID分别是1、2、3、4、5。当产生网络分区或节点启动速度差异较大时，节点1看到的节点列表是1、2、3、4，选出4；节点2看到的节点列表是2、3、4、5，选出5。结果就不一致了怎么办？
2. **得票数需过半**：某节点被选为主节点，必须判断加入它的节点数过半，才确认Master身份。
3. **当探测到节点离开事件时，必须判断当前节点数是否过半**：
    - 如果达不到 quorum，则放弃Master身份，重新加入集群。
    - **为什么这么做？**如果不这么做，则设想以下情况：假设5台机器组成的集群产生网络分区，2台一组，3台一组，产生分区前，Master位于2台中的一个，此时3台一组的节点会重新并成功选取Master，产生双主，俗称脑裂。
    - **怎么做？怎么判断当前节点数是否过半？**集群并不知道自己共有多少个节点。quorum值从配置中读取，我们需要设置配置项：discovery.zen.minimum_master_nodes

一配置：与选主过程相关的重要配置。

discovery.zen.minimum_master_nodes：最小主节点数，这是防止脑裂、防止数据丢失的极其重要的参数。

这个参数的实际作用早已超越了其表面的含义。除了在选主时用于决定"多数"，还用于多处重要的判断，至少包含以下4个位置：

1. 触发选主 进入选主的流程之前，参选的节点数需要达到法定人数。
2. 决定Master 选出临时的Master之后，这个临时Master需要判断加入它的节点达到法定人数，才确认选主成功。
3. gateway选举元信息 向有Master资格的节点发起请求，获取元数据，获取的响应数量必须达到法定人数，也就是参与元信息选举的节点数。
4. Master发布集群状态 发布成功数量。

为了避免脑裂，它的值应该是半数以上（quorum）：（master_eligible_nodes / 2）+ 1

例如，如果有3个具备Master资格的节点，则这个值至少应该设置为（3/2）+ 1 = 2。

```
#该参数可以动态设置：
PUT /_cluster/settings
{
    "persistent" : {
    "discovery.zen.minimum_master_nodes" : 2
    }
}
```

4.3 Master选举流程

ZenDiscovery的选主过程如下：

- 每个节点计算最小的已知节点ID，该节点为临时Master。向该节点发送领导投票。
- 如果一个节点收到足够多的票数，并且该节点也为自己投票，那么它将扮演领导者的角色，开始发布集群状态。

所有节点都会参与选举，并参与投票，但是，只有有资格成为Master的节点（node.master为true）的投票才有效。获得多少选票可以赢得选举胜利，就是所谓的法定人数。在 ES 中，法定大小是一个可配置的参数。配置项：discovery.zen.minimum_master_nodes。

为了避免脑裂，最小值应该是有Master资格的节点数n/2+1。

4.4 流程详细分析

整体流程可以概括为：

- 选举临时Master
- 如果本节点当选，则等待确立Master
- 如果其他节点当选，则尝试加入集群
- 然后启动节点失效探测器

面我们具体分析每个步骤的实现。

4.4.1 选举临时Master

4.4.2 投票与得票的实现

4.4.3 确立Master或加入集群

4.4.4 节点失效检测

下次内容介绍

周日大咖分享：

- 程序员求职面试谈工资的技巧-防坑指南
- 新时代程序员生存指南

下周场景面试：

- 场景面试-在线教育
- 场景面试-电商
- 场景面试-直播平台

扩展01-讲分区容错性能的时候，能不能讲一下FLP不可能性的证明！

分片是不是相当于msyql的页？

mongo和es选型的时候怎么选？

- ElasticSearch是有自己应用场景的：搜索、数据挖掘、日志解决方案
- MongoDB