EleasticSearch核心原理

一、ES集群的相关概念

ES集群是一个 P2P类型(使用 gossip 协议)的分布式系统，除了集群状态管理以外，其他所有的请求都可以发送到集群内任意一台节点上，这个节点可以自己找到需要转发给哪些节点，并且直接跟这些节点通信。所以，从网络架构及服务配置上来说，构建集群所需要的配置极其简单。在 Elasticsearch 2.0 之前，无阻碍的网络下，所有配置了相同 cluster.name 的节点都自动归属到一个集群中。2.0 版本之后，基于安全的考虑避免开发环境过于随便造成的麻烦，从 2.0 版本开始，默认的自动发现方式改为了单播(unicast)方式。配置里提供几台节点的地址，ES 将其视作 gossip router 角色，借以完成集群的发现。由于这只是 ES 内一个很小的功能，所以 gossip router 角色并不需要单独配置，每个 ES 节点都可以担任。所以，采用单播方式的集群，各节点都配置相同的几个节点列表作为 router 即可。

集群中节点数量没有限制，一般大于等于2个节点就可以看做是集群了。一般处于高性能及高可用方面来考虑一般集群中的节点数量都是3个及3个以上。

1 集群 cluster

一个集群就是由一个或多个节点组织在一起，它们共同持有整个的数据，并一起提供索引和搜索功能。一个集群由一个唯一的名字标识，这个名字默认就是“elasticsearch”。这个名字是重要的，因为一个节点只能通过指定某个集群的名字，来加入这个集群

2 节点 node

一个节点是集群中的一个服务器，作为集群的一部分，它存储数据，参与集群的索引和搜索功能。和集群类似，一个节点也是由一个名字来标识的，默认情况下，这个名字是一个随机的漫威漫画角色的名字，这个名字会在启动的时候赋予节点。这个名字对于管理工作来说挺重要的，因为在这个管理过程中，你会去确定网络中的哪些服务器对应于Elasticsearch集群中的哪些节点。

一个节点可以通过配置集群名称的方式来加入一个指定的集群。默认情况下，每个节点都会被安排加入到一个叫做“elasticsearch”的集群中，这意味着，如果你在你的网络中启动了若干个节点，并假定它们能够相互发现彼此，它们将会自动地形成并加入到一个叫做“elasticsearch”的集群中。

在一个集群里，只要你想，可以拥有任意多个节点。而且，如果当前你的网络中没有运行任何Elasticsearch节点，这时启动一个节点，会默认创建并加入一个叫做“elasticsearch”的集群。

3 分片和副本 shards&replicas

一个索引可以存储超出单个结点硬件限制的大量数据。比如，一个具有10亿文档的索引占据1TB的磁盘空间，而任一节点都没有这样大的磁盘空间；或者单个节点处理搜索请求，响应太慢。为了解决这个问题，Elasticsearch提供了将索引划分成多份的能力，这些份就叫做分片。当你创建一个索引的时候，你可以指定你想要的分片的数量。每个分片本身也是一个功能完善并且独立的“索引”，这个“索引”可以被放置到集群中的任何节点上。分片很重要，主要有两方面的原因：
1）允许你水平分割/扩展你的内容容量。
2）允许你在分片（潜在地，位于多个节点上）之上进行分布式的、并行的操作，进而提高性能/吞吐量。

至于一个分片怎样分布，它的文档怎样聚合回搜索请求，是完全由Elasticsearch管理的，对于作为用户的你来说，这些都是透明的。

在一个网络/云的环境里，失败随时都可能发生，在某个分片/节点不知怎么的就处于离线状态，或者由于任何原因消失了，这种情况下，有一个故障转移机制是非常有用并且是强烈推荐的。为此目的，Elasticsearch允许你创建分片的一份或多份拷贝，这些拷贝叫做副本分片，或者直接叫副本。

副本之所以重要，有两个主要原因： 在分片/节点失败的情况下，提供了高可用性。因为这个原因，注意到复制分片从不与原/主要（original/primary）分片置于同一节点上是非常重要的。扩展你的搜索量/吞吐量，因为搜索可以在所有的复制上并行运行。总之，每个索引可以被分成多个分片。一个索引也可以被复制0次（意思是没有复制）或多次。一旦复制了，每个索引就有了主分片（作为复制源的原来的分片）和副本分片（主分片的拷贝）之别。分片和副本的数量可以在索引创建的时候指定。在索引创建之后，你可以在任何时候动态地改变副本的数量，但是你事后不能改变分片的数量。

默认情况下，Elasticsearch中的每个索引被分片5个主分片和1个副本，这意味着，如果你的集群中至少有两个节点，你的索引将会有5个主分片和另外5个副本分片（1个完全拷贝），这样的话每个索引总共就有10个分片。

二、集群搭建及使用

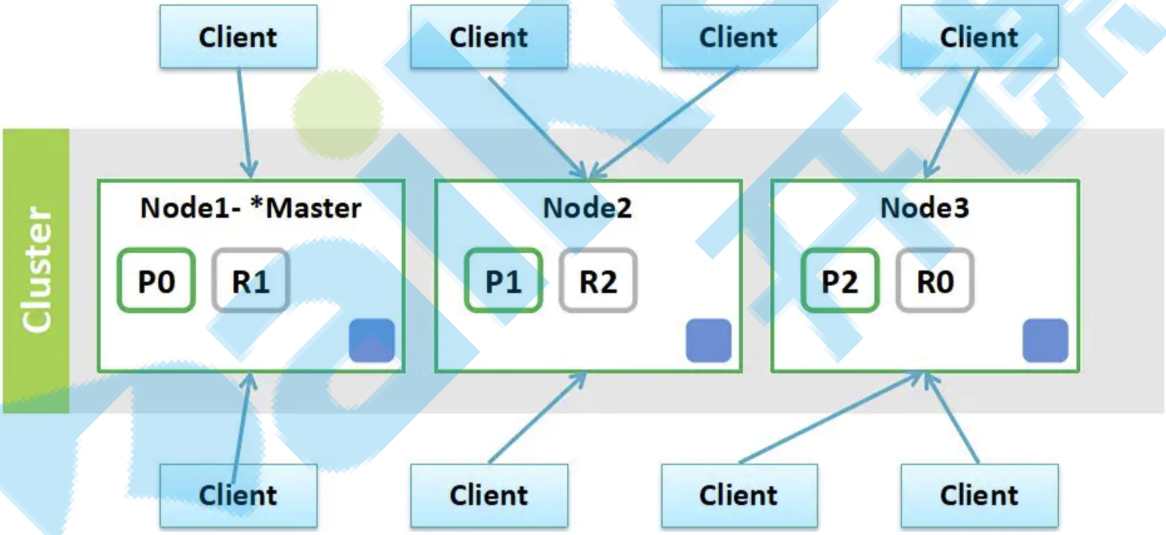
1、集群搭建

略

2、集群的使用

使用方法同单机版

三、ES集群核心原理



1、节点类型

1）master节点

master节点特点：

```
1 整个集群只会有一个master节点，它将负责管理集群范围内的所有变更，例如增加、删除索引；或者增加、删除节点等。而master节点并不需要涉及到文档级别的变更和搜索等操作，所以当集群只拥有一个master节点的情况下，即使流量的增加它也不会成为瓶颈。
2 master节点需要从众多候选master节点中选择一个。
```

master节点的作用：

```
负责集群节点上下线，shard分片的重新分配。
创建、删除索引。
负责接收集群状态(cluster state)的变化，并推送给所有节点。集群节点都各有一份完整的cluster state，只是master node负责维护。
利用自身空闲资源，协调创建索引请求或者查询请求，将请求分发到相关node服务器。
```

master节点的配置如下（elasticsearch.yml）

```
node.master: true
node.data: false
```

2）数据节点

```
负责存储数据，提供建立索引和搜索索引的服务。
data节点消耗内存和磁盘IO的性能比较大。
```

data节点的配置如下（elasticsearch.yml）

```
node.master: false
node.data: true
```

3）协调节点

```
不会被选作主节点，也不会存储任何索引数据。主要用于查询负载均衡。将查询请求分发给多个node服务器，并对结果进行汇总处理。协调节点的作用：
第一：诸如搜索请求或批量索引请求之类的请求可能涉及保存在不同数据节点上的数据。 例如，搜索请求在两个阶段中执行（query 和fetch），这两个阶段由接收客户端请求的节点 - 协调节点协调。
第二：在请求阶段，协调节点将请求转发到保存数据的数据节点。 每个数据节点在本地执行请求并将其结果返回给协调节点。 在收集fetch阶段，协调节点将每个数据节点的结果汇集为单个全局结果集。
第三：每个节点都隐式地是一个协调节点。 这意味着将所有三个node.master，node.data设置为false的节点作为仅用作协调节点，无法禁用该节点。 结果，这样的节点需要具有足够的内存和CPU以便处理收集阶段。

协调节点最好不要分离，跟数据节点在一起就行。
```

协调节点的配置如下（elasticsearch.yml）

```
node.master: false
node.data: false
```

> **注意：**
>
> 一个节点可以充当一个或多个角色。默认 3个角色都有。

2、Cluster State

1）什么是CLUSTER STATE

Cluster State是指集群中的各种状态和元数据(meta data)信息。其中，索引(Index)的mappings、settings配置、持久化状态等信息，以及集群的一些配置信息都属于元数据(meta data)。
元数据非常重要，标识了集群中节点状态、索引的配置与状态信息等；假如记录某个index的元数据丢失，那么集群就认为这个index不再存在。ES集群中的每个节点都会保存一份这样的元数据信息，这样在增删索引、节点场景下能够极大方便集群选主流程、集群的管理操作。

2)CLUSTER STATE的主要内容

Cluster State中存储的主要内容如下：

```
long version：当前版本号，每次更新加1，即便集群重启version仍会增加
String stateUUID：该state对应的唯一id
ClusterName clusterName：集群名称
DiscoveryNodes nodes：当前集群全部节点信息
RoutingTable routingTable：集群中所有index的路由表，即集群中全部索引的各个分片在集群节点上的分布信息
MetaData metaData：集群的meta数据，主要包括所有索引的mappings和settings配置
ClusterBlocks blocks：集群级的限制设定，用于屏蔽某些操作
ImmutableOpenMap<String, Custom> customs：自定义配置，如snapshots，可用插件扩展
```

ES提供了Cluster state API，可通过`_cluster/state`来获取集群中的全部状态信息

3）CLUSTER STATE的更新流程

ClusterState相关特性如下：

1. ClusterState是不可变对象，每次状态变更都会产生新的ClusterState，版本号随之更新
2. 在ES中, 集群状态由Master节点维护，并且只能由Master节点更新集群状态
3. Master节点一次处理一批集群状态更新，计算所需的更改并将更新后的新版集群状态发布到集群中的所有其他节点

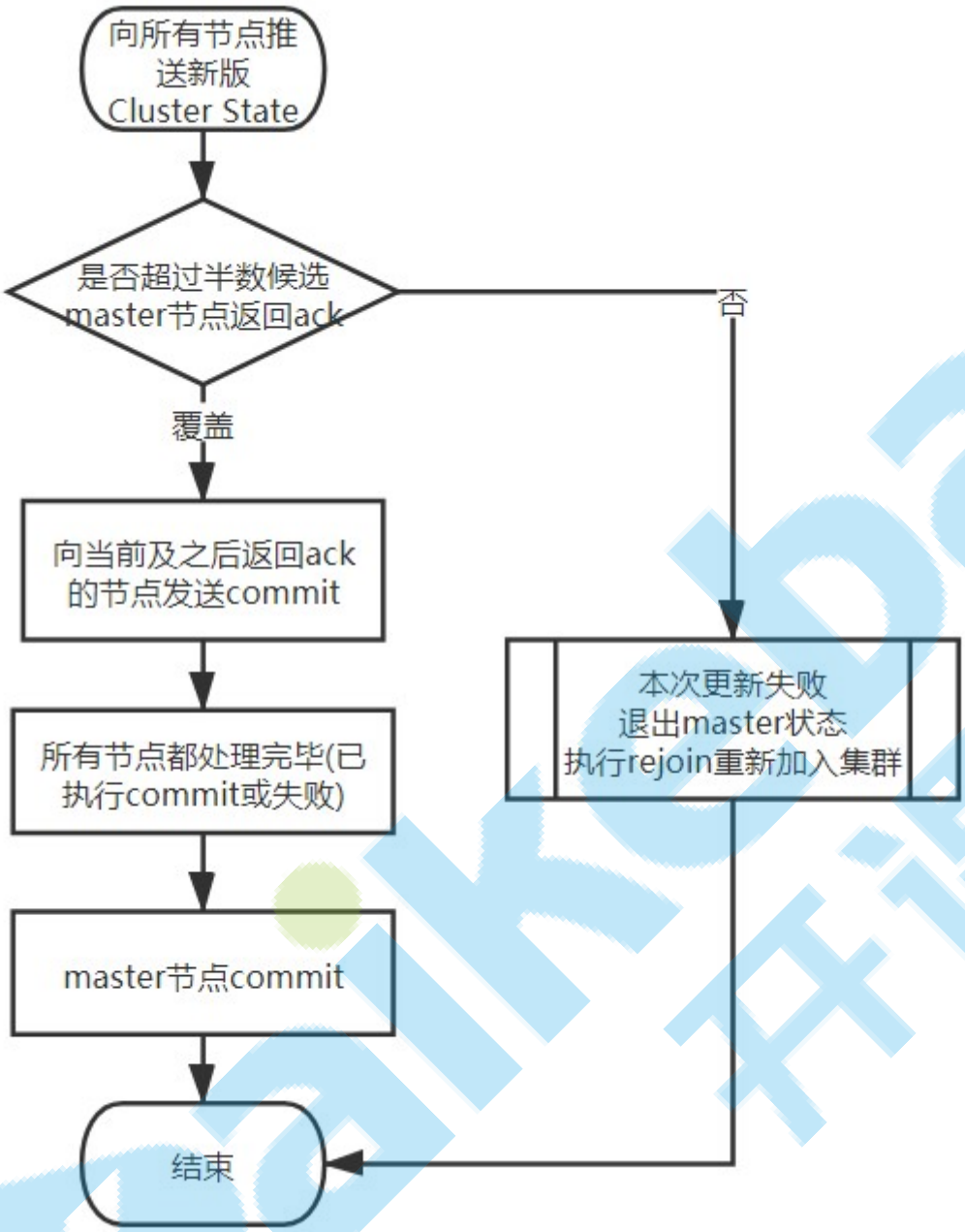
Cluster State的更新流程，实现了原子性和一致性，确保Cluster State能够反映集群中最新且真实的状态，为Master选举流程、错误检测、集群扩缩容提供了高度的一致性保障。下面来详细分析：

1. 原子性保证：master节点进程内的不同线程更改ClusterState时，每次需提交一个Task给MasterService，MasterService中只使用一个线程来串行处理这些Task，每次处理时把当前的ClusterState作为Task中execute函数的参数，即保证了所有的Task都是在currentClusterState的基础上进行更改，然后不同的Task是串行执行的。
2. 一致性保证：

   `一致性`是为了解决这样一个问题，我们知道，新的集群状态一旦在某个节点上commit，那么这个节点就会执行相应的操作，比如删除某个Shard等，这样的操作是不可回退的。而假如此时Master节点挂掉了，新产生的Master一定要在新的集群配置信息上进行更改，不能出现回退，否则就会出现集群配置数据回退了但是操作无法回退的情况；ES使用两阶段提交方式(Add two phased commit to Cluster State publishing)实现`一致性`。

   所谓的两阶段提交，是把Master节点发布ClusterState分成两步，第一步是向所有节点推送最新的ClusterState，当有超过半数的master节点返回ack请求时，再发送commit请求，要求节点commit接收到的新版ClusterState。如果没有超过半数的节点返回ack，则认为本次发布失败，同时退出master状态，执行rejoin重新加入集群。

基于原子性和一致性更新保证，分析Cluster State的更新流程如下图所示：



每次发布时，

1. 首先由Master节点向集群中的所有节点广播新版本的集群状态；
2. 其他节点接收到新版集群状态后，均向Master节点发送ack确认请求，但此时并不立即应用新接收的集群状态。
3. 一旦Master节点收到足够多的候选Master节点的确认请求，就认为新版集群状态*已提交（committed）*，继而广播commit请求，指示其他节点应用现在已提交的集群状态。
4. 每个节点接收到commit请求后，会把新版ClusterState发给该节点上相关的各个模块，各个模块根据新版ClusterState判断是否要做什么操作，如创建Shard等；应用新版集群状态后，再将第二个ack确认请求发送回Master节点。
5. 在Master节点开始处理并发布下一版本集群状态更新前，它一直处于等待状态，直到到达超时时间或它收到集群中每个节点已应用新版集群状态的确认请求。其中，新版集群状态完全发布到所有节点的超时时间，由`cluster.publish.timeout`这一配置设置，默认为从发布开始算起的30s。如果在提交新版集群状态之前已达到该超时时间，则集群状态更改失败，并且Master节点认为退出Master状态，并重新加入集群，开始尝试选举新的Master节点。如果新版集群状态在到达`cluster.publish.timeout`指定的超时时间前*已提交*，则Master节点认为更改*已成功*。
6. 如果没有收到某些节点确认已应用当前新版集群状态的请求，则这些节点被认为*滞后节点*，因为它们的集群状态已落后于Master节点的最新状态。Master节点等待滞后节点再追赶

`cluster.follower_lag.timeout`设定的时间，默认为90s。如果节点在该时间段内仍未成功应用集群状态更新，则认为该节点已失败并将之从集群中删除。

3、集群选举

1）选举的时机

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。并且该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。
2. 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）选举的过程

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)。

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。

4、脑裂问题

1. 什么是脑裂现象

由于部分节点网络断开，集群分成两部分，且这两部分都有master选举权。就成形成一个与原集群一样名字的集群，这种情况称为集群脑裂（split-brain）现象。这个问题非常危险，因为两个新形成的集群会同时索引和修改集群的数据。

2. 解决方案

```
# 决定选举一个master最少需要多少master候选节点。默认是1。
# 这个参数必须大于等于为集群中master候选节点的quorum数量，也就是大多数。
# quorum算法：master候选节点数量 / 2 + 1
# 例如一个有3个节点的集群，minimum_master_nodes 应该被设置成 3/2 + 1 = 2（向下取整）
discovery.zen.minimum_master_nodes:2
# 等待ping响应的超时时间,默认值是3秒。如果网络缓慢或拥塞，会造成集群重新选举，建议略微调大这个值。
# 这个参数不仅仅适应更高的网络延迟，也适用于在一个由于超负荷而响应缓慢的节点的情况。
discovery.zen.ping.timeout:10s
# 当集群中没有活动的Master节点后，该设置指定了哪些操作（read、write）需要被拒绝（即阻塞执行）。有两个设置值：all和write，默认为wirte。
discovery.zen.no_master_block : write
```

3.场景分析

一个生产环境的es集群，至少要有3个节点，同时将discovery.zen.minimum_master_nodes设置为2，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，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。

但是因为ES集群是可以动态增加和下线节点的，所以可能随时会改变`quorum`。所以这个参数也是可以通过api随时修改的，特别是在节点上线和下线的时候，都需要作出对应的修改。而且一旦修改过后，这个配置就会持久化保存下来。

```
PUT /_cluster/settings { "persistent" : {
"discovery.zen.minimum_master_nodes" : 2 } }
```

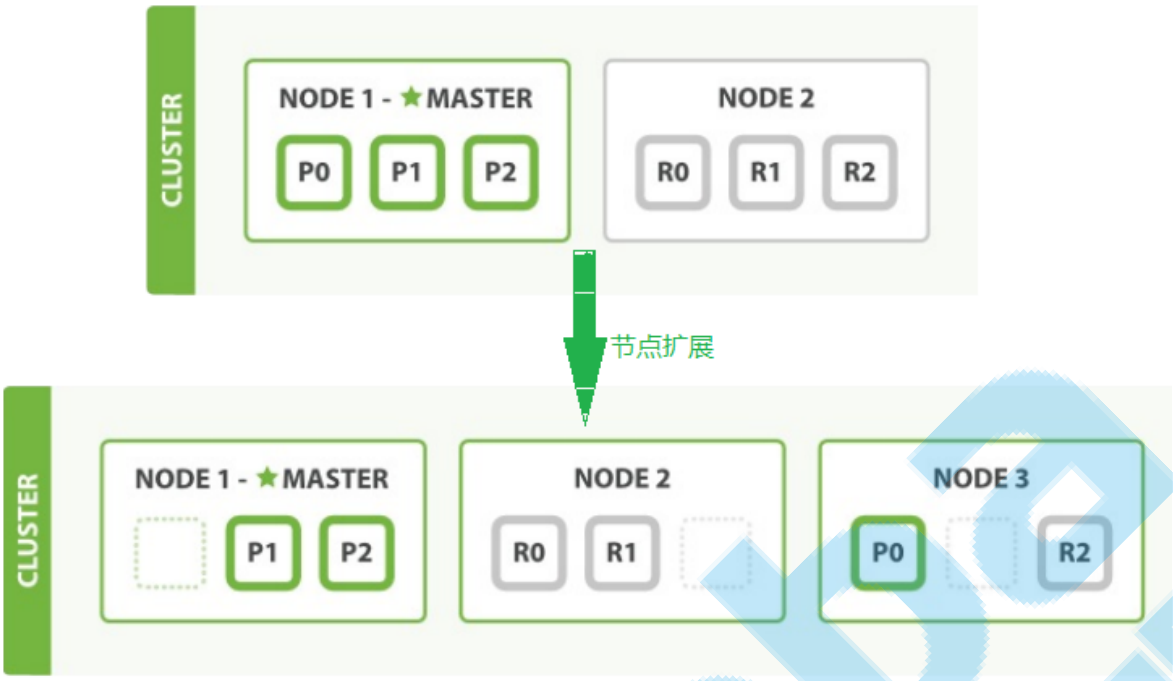
5、集群扩展

1）横向扩展

一个索引可以存储超出单个结点硬件限制的大量数据。

```
# 问题：随着应用需求的增长，我们该如何扩展？
如果我们启动第三个节点，我们的集群会重新组织自己。分片已经被重新分配以平衡负载

Node3 包含了分别来自 Node 1 和 Node 2 的一个分片，这样每个节点就有两个分片，和之前相比少了一个，这意味着每个节点上的分片将获得更多的硬件资源（CPU、RAM、I/O）
```



注意：分片本身就是一个完整的搜索引擎，它可以使用单一节点的所有资源。我们拥有6个分片（3个主分片和三个复制分片），最多可以扩展到6个节点，每个节点上有一个分片，每个分片可以100%使用这个节点的资源。

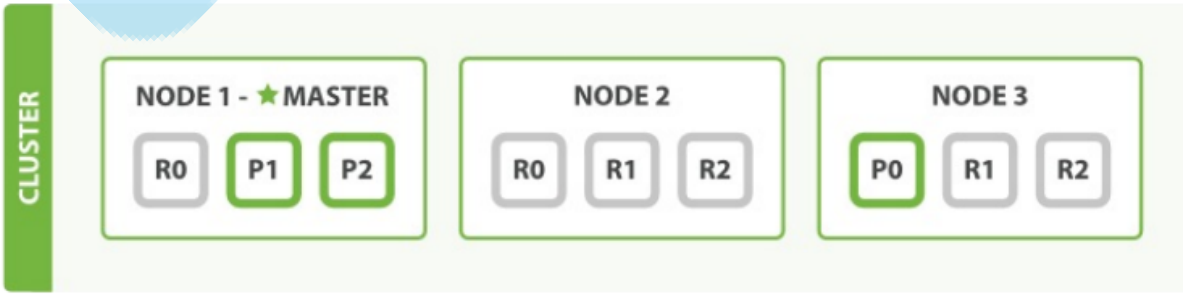
2）继续扩展

如果我们要扩展到6个以上的节点，要怎么做?

```
1    主分片的数量在创建索引时已经确定。实际上，这个数量定义了能存储到索引里数据的最大数量（实际的数量取决于你的数据、硬件和应用场景）。
2    然而，主分片或者复制分片都可以处理读请求——搜索或文档检索，所以数据的冗余越多，我们能处理的搜索吞吐量就越大。复制分片的数量可以在运行中的集群中动态地变更，这允许我们可以根据需求扩大或者缩小规模。让我们把复制分片的数量从原来的 1 增加到 2 ：
```

更新副本数量:

```
1  # 更新副本数量
2  PUT /blogs/_settings
3  {
4  "number_of_replicas" : 2
5  }
```



从图中可以看出， 索引现在有9个分片：3个主分片和6个复制分片。这意味着我们能够扩展到9个节点，再次变成每个节点一个分片。这样使我们的搜索性能相比原始的三节点集群增加三倍。

当然，在同样数量的节点上增加更多的复制分片并不能提高性能，因为这样做的话平均每个分片的所占有的硬件资源就减少了（大部分请求都聚集到了分片少的节点，导致一个节点吞吐量太大，反而降低性能），你需要增加硬件来提高吞吐量。不过这些额外的复制节点使我们有更多的冗余：通过以上对节点的设置，我们能够承受两个节点故障而不丢失数据。

6、故障转移

ES有两种集群故障探查机制：

> 1. 通过master进行的，master会ping集群中所有的其他node，确保它们是否是存活着的。
> 2. 每个node都会去ping master来确保master是存活的，否则会发起一个选举过程。

注意：相同的分片不会放在同一个节点上

从图可知：

1）每个索引被分成了5个分片；

2）每个分片有一个副本；

3）5个分片基本均匀分布在3个dataNode上；

注意分片的边框（border）有粗有细，具体区别是：

粗边框代表：primary（true）

细边框代表：replica

演示负载均衡效果： 关闭集群中其中一个节点，剩余分片将会重新进行均衡分配。

四、ES数据存储

1、存储流程

为了将数据添加到Elasticsearch，我们需要索引(index)——一个存储关联数据的地方。实际上，索引只是一个用来指向一个或多个分片(shards)的“逻辑命名空间(logical namespace)”.一个分片(shard)是一个最小级别“工作单元(worker unit)”,它只是保存了索引中所有数据的一部分。

当一个写请求发送到 es 后，es 将数据写入 `memory buffer` 中，并添加事务日志（`translog`）。如果每次一条数据写入内存后立即写到硬盘文件上，由于写入的数据肯定是离散的，因此写入硬盘的操作也就是随机写入了。硬盘随机写入的效率相当低，会严重降低es的性能。

因此 es 在设计时在 `memory buffer` 和硬盘间加入了 Linux 的页面高速缓存（`File system cache`）来提高 es 的写效率。

当写请求发送到 es 后，es 将数据暂时写入 `memory buffer` 中，此时写入的数据还不能被查询到。默认设置下，es 每1秒钟将 `memory buffer` 中的数据 `refresh` 到 Linux 的 `File system cache`，并清空 `memory buffer`，此时写入的数据就可以被查询到了。

但 `File system cache` 依然是内存数据，一旦断电，则 `File system cache` 中的数据全部丢失。默认设置下，es 每30分钟调用 `fsync` 将 `File system cache` 中的数据 `flush` 到硬盘。因此需要通过 `translog` 来保证即使因为断电 `File system cache` 数据丢失，es 重启后也能通过日志回放找回丢失的数据。

`translog` 默认设置下，每一个 `index`、`delete`、`update` 或 `bulk` 请求都会直接 `fsync` 写入硬盘。为了保证 `translog` 不丢失数据，在每一次请求之后执行 `fsync` 确实会带来一些性能问题。对于一些允许丢失几秒钟数据的场景下，可以通过设置 `index.translog.durability` 和 `index.translog.sync_interval` 参数让 `translog` 每隔一段时间才调用 `fsync` 将事务日志数据写入硬盘。

2、动态更新索引

以在线动态服务的层面看，要做到实时更新条件下数据的可用和可靠，就需要在倒排索引的基础上，再做一系列更高级的处理。

其实总结一下 Lucene 的处理办法，很简单，就是一句话：**新收到的数据写到新的索引文件里。**

Lucene 把每次生成的倒排索引，叫做一个段(segment)。然后另外使用一个 commit 文件，记录索引内所有的 segment。而生成 segment 的数据来源，则是内存中的 buffer。也就是说，动态更新过程如下：

1. 当前索引有 3 个 segment 可用。索引状态如图 2-1；

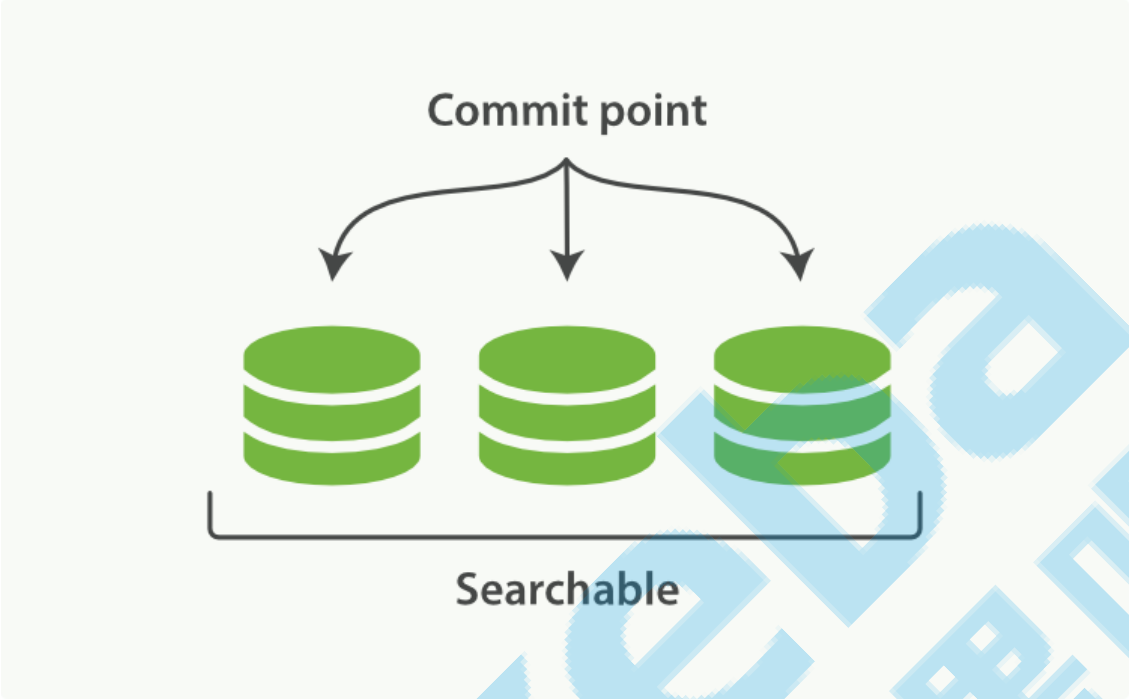


图 2-1

2. 新接收的数据进入内存 buffer。索引状态如图 2-2；

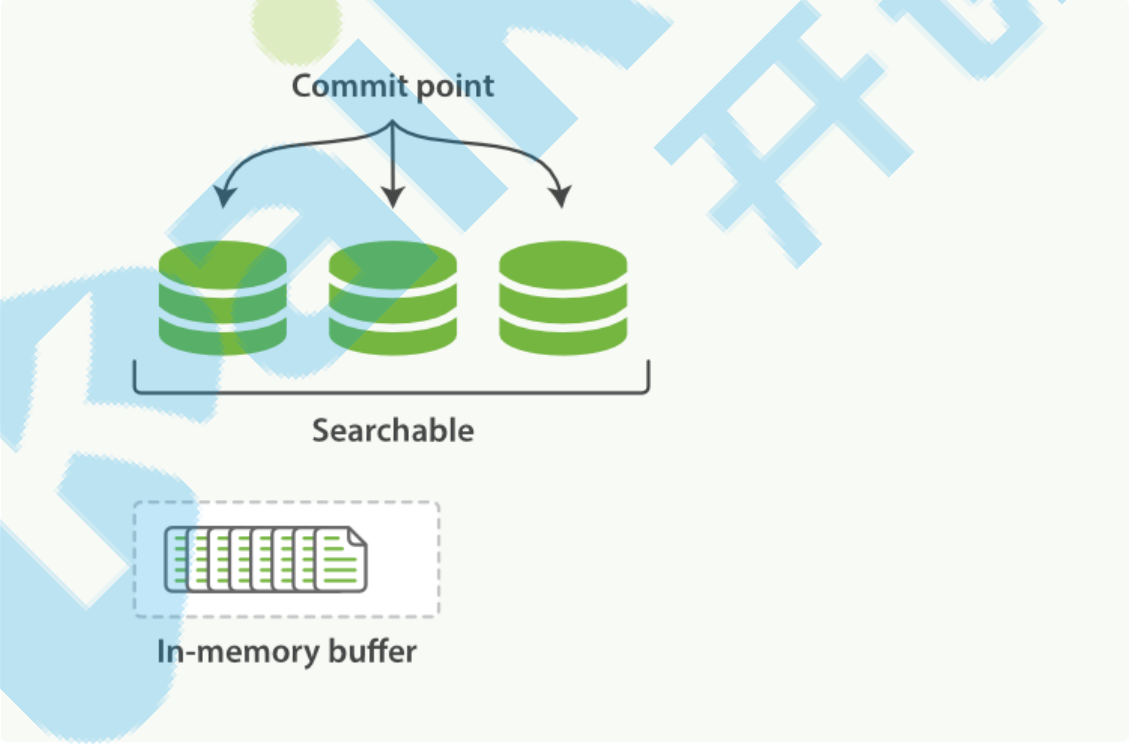


图 2-2

3. 内存 buffer 刷到磁盘，生成一个新的 segment，commit 文件同步更新。索引状态如图 2-3。

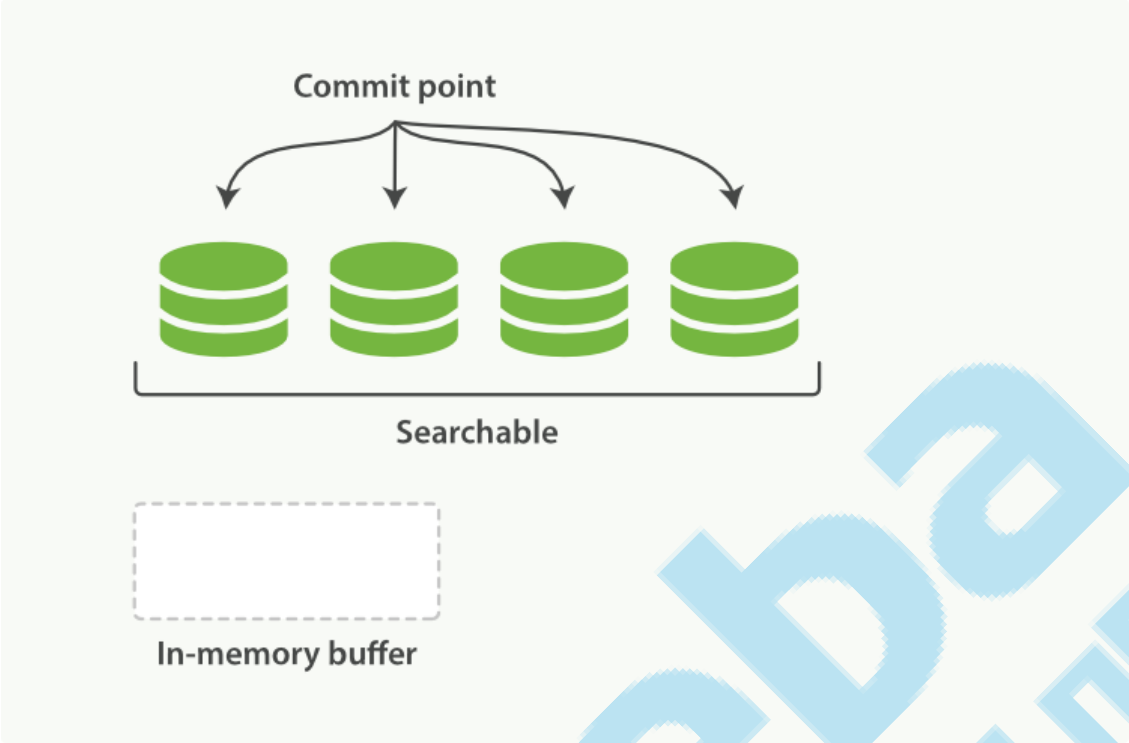


图 2-3

3、利用磁盘缓存实现的准实时检索

既然涉及到磁盘，那么一个不可避免的问题就来了：磁盘太慢了！对我们要求实时性很高的服务来说，这种处理还不够。所以，在第 3 步的处理中，还有一个中间状态：

1. 内存 buffer 生成一个新的 segment，刷到文件系统缓存中，Lucene 即可检索这个新 segment。索引状态如图 2-4。

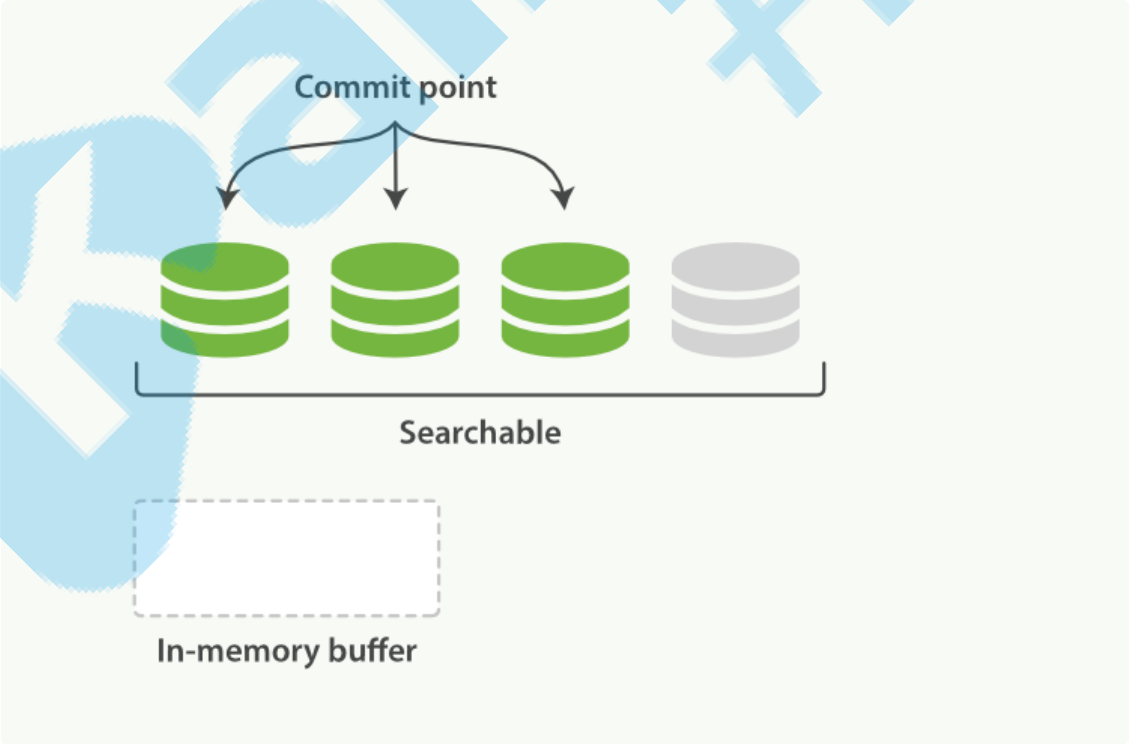


图 2-4

2. 文件系统缓存真正同步到磁盘上，commit 文件更新。达到图 2-3 中的状态。

这一步刷到文件系统缓存的步骤，在 Elasticsearch 中，是默认设置为 1 秒间隔的，对于大多数应用来说，几乎就相当于是实时可搜索了。Elasticsearch 也提供了单独的 `/_refresh` 接口，用户如果对 1 秒间隔还不满意的，可以主动调用该接口来保证搜索可见。

注：5.0 中还提供了一个新的请求参数：`?refresh=wait_for`，可以在写入数据后不强制刷新但一直等到刷新才返回。

不过对于 Elastic Stack 的日志场景来说，恰恰相反，我们并不需要如此高的实时性，而是需要更快的写入性能。所以，一般来说，我们反而会通过 `/_settings` 接口或者定制 template 的方式，加大 `refresh_interval` 参数：

```
# curl -XPOST http://127.0.0.1:9200/logstash-2015.06.21/_settings -d'
{ "refresh_interval": "10s" }
'
```

如果是导入历史数据的场合，那甚至可以先完全关闭掉：

```
# curl -XPUT http://127.0.0.1:9200/logstash-2015.05.01 -d'
{
  "settings" : {
    "refresh_interval": "-1"
  }
}'
```

在导入完成以后，修改回来或者手动调用一次即可：

```
# curl -XPOST http://127.0.0.1:9200/logstash-2015.05.01/_refresh
```

4、translog 提供的磁盘同步控制

既然 refresh 只是写到文件系统缓存，那么第 4 步写到实际磁盘又是有什么来控制的？如果这期间发生主机错误、硬件故障等异常情况，数据会不会丢失？

这里，其实有另一个机制来控制。Elasticsearch 在把数据写入到内存 buffer 的同时，其实还另外记录了一个 translog 日志。也就是说，第 2 步并不是图 2-2 的状态，而是像图 2-5 这样：

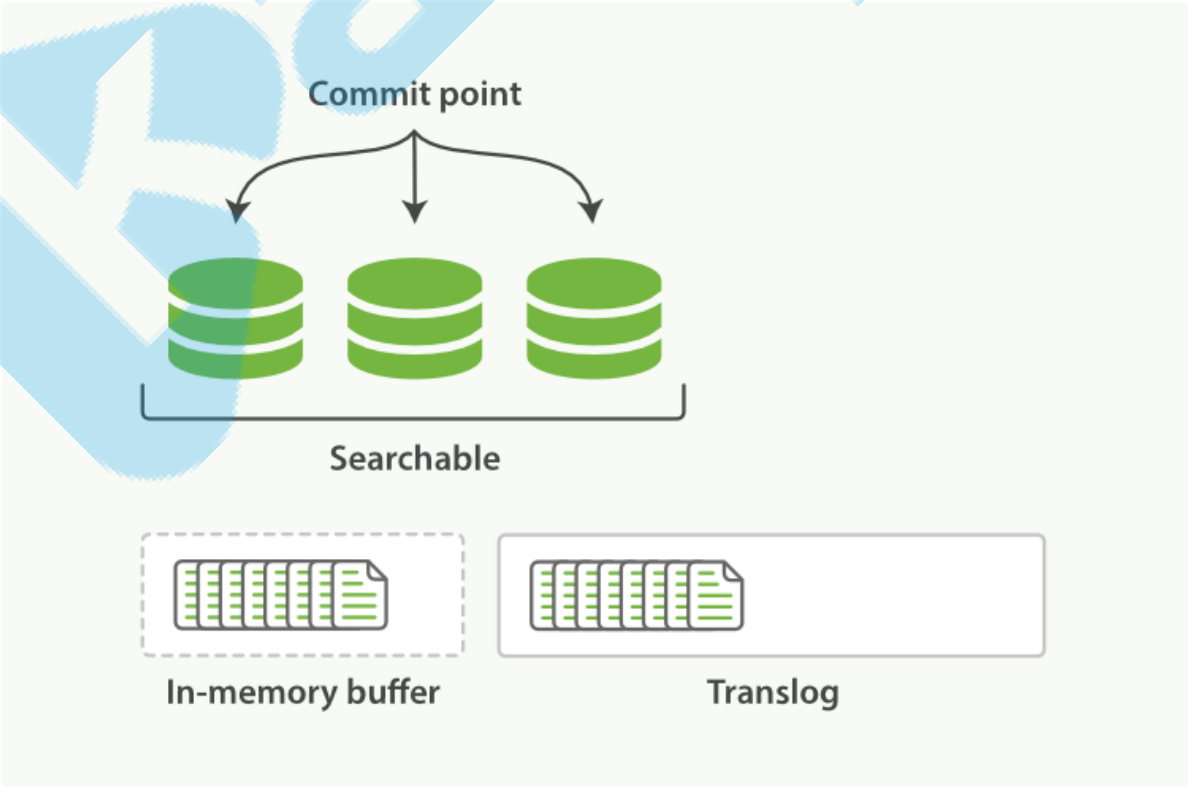


图 2-5

在第 3 和第 4 步，refresh 发生的时候，translog 日志文件依然保持原样，如图 2-6：

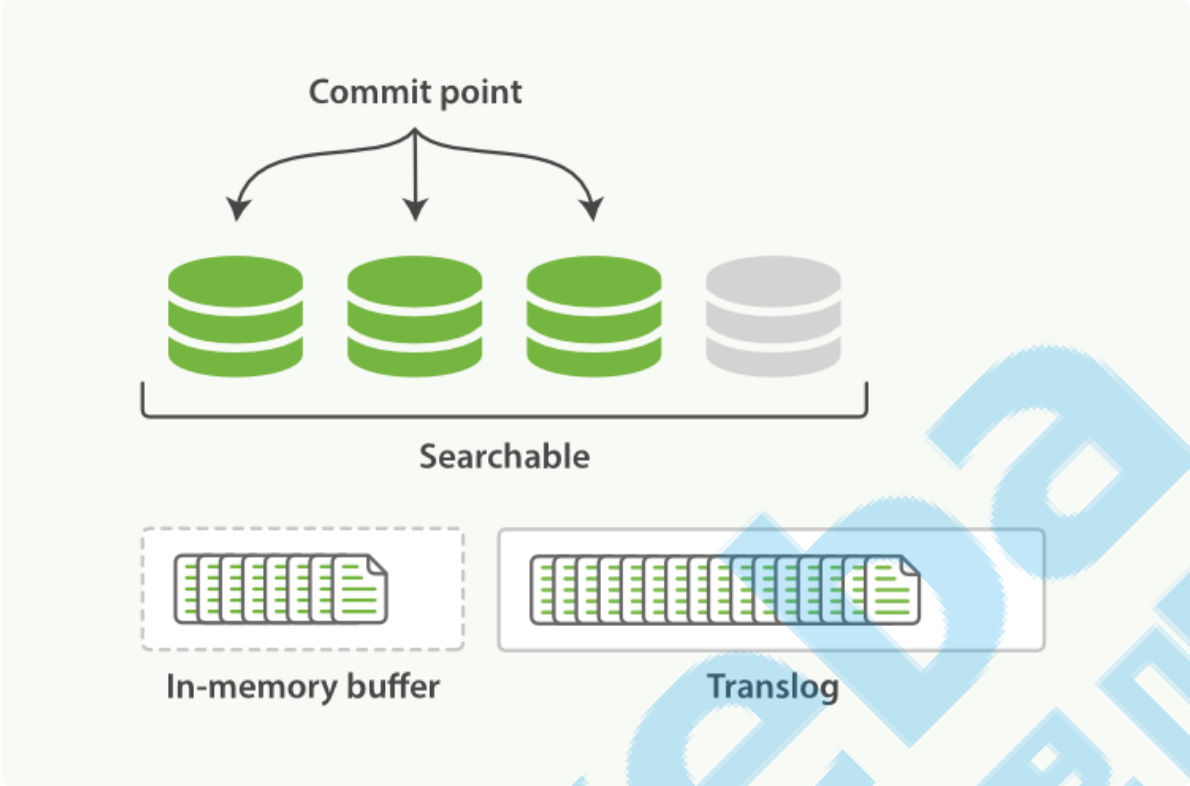


图 2-6

也就是说，如果在这期间发生异常，Elasticsearch 会从 commit 位置开始，恢复整个 translog 文件中的记录，保证数据一致性。

等到真正把 segment 刷到磁盘，且 commit 文件进行更新的时候， translog 文件才清空。这一步，叫做 flush。同样，Elasticsearch 也提供了 `/_flush` 接口。

对于 flush 操作，Elasticsearch 默认设置为：每 30 分钟主动进行一次 flush，或者当 translog 文件大小大于 512MB (老版本是 200MB)时，主动进行一次 flush。这两个行为，可以分别通过 `index.translog.flush_threshold_period` 和 `index.translog.flush_threshold_size` 参数修改。

如果对这两种控制方式都不满意，Elasticsearch 还可以通过 `index.translog.flush_threshold_ops` 参数，控制每收到多少条数据后 flush 一次。

5、translog 的一致性

索引数据的一致性通过 translog 保证。那么 translog 文件自己呢?

默认情况下，Elasticsearch 每 5 秒，或每次请求操作结束前，会强制刷新 translog 日志到磁盘上。

后者是 Elasticsearch 2.0 新加入的特性。为了保证不丢数据，每次 insert、bulk、delete、update 完成的时候，一定触发刷新 translog 到磁盘上，才给请求返回 200 OK。这个改变在提高数据安全性的同时当然也降低了一点性能。

如果你不在意这点可能性，还是希望性能优先，可以在 index template 里设置如下参数：

```
{
    "index.translog.durability": "async"
}
```

6、段的合并

通过上节内容，我们知道了数据怎么进入 ES 并且如何才能让数据更快的被检索使用。其中用一句话概括了 Lucene 的设计思路就是"开新文件"。从另一个方面看，开新文件也会给服务器带来负载压力。因为默认每 1 秒，都会有一个新文件产生，每个文件都需要有文件句柄，内存，CPU 使用等各种资源。一天有 86400 秒，设想一下，每次请求要扫描一遍 86400 个文件，这个响应性能绝对好不了！

为了解决这个问题，ES 会不断在后台运行任务，主动将这些零散的 segment 做数据归并，尽量让索引内只保有少量的，每个都比较大的，segment 文件。这个过程是有独立的线程来进行的，并不影响新 segment 的产生。归并过程中，索引状态如图 2-7，尚未完成的较大的 segment 是被排除在检索可见范围之外的：

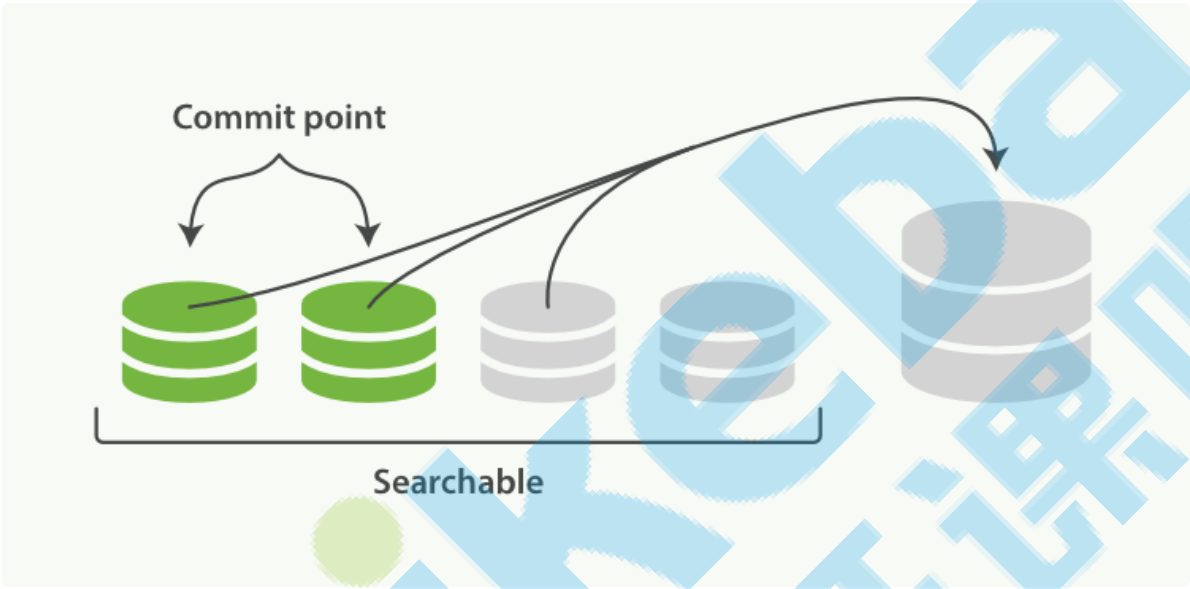
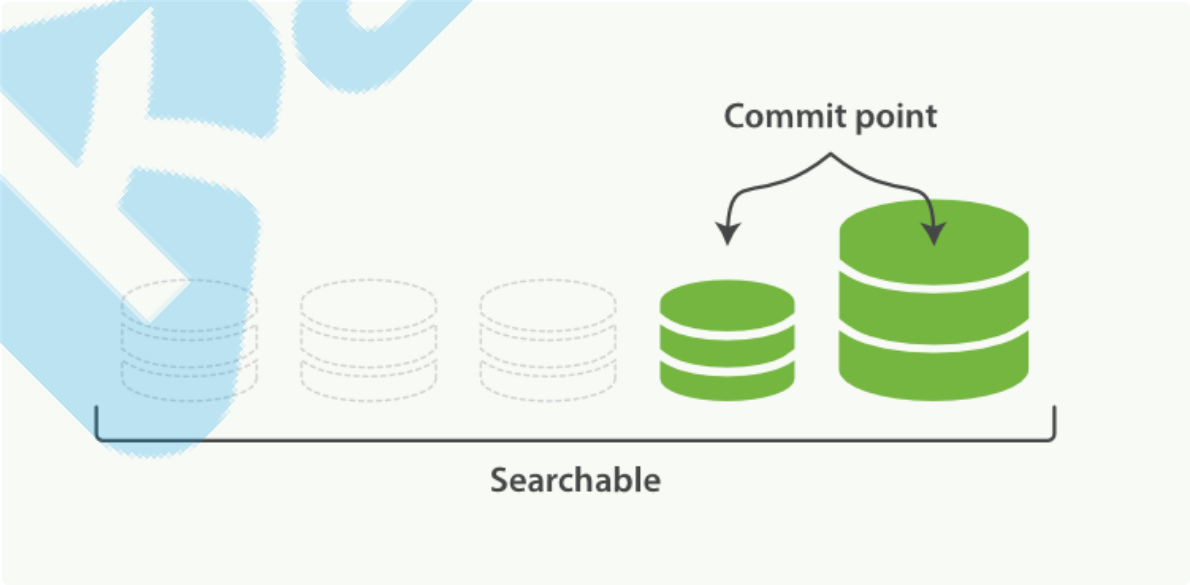


图 2-7

当归并完成，较大的这个 segment 刷到磁盘后，commit 文件做出相应变更，删除之前几个小 segment，改成新的大 segment。等检索请求都从小 segment 转到大 segment 上以后，删除没用的小 segment。这时候，索引里 segment 数量就下降了，状态如图 2-8 所示：



7、增删改原理

1）增加文档

增加文档就是在新的索引段中增加一个文档，并且新的文档的增加会产生新的索引段。

2）删除文档

由于段是不可修改的删除文档，所以删除文档只需要在被删除的文档上打上一个删除标记，等合并索引段的时候将这个文档删除。

3）修改文档

修改文档也是在旧的文档上打上删除标记，然后增加一个新的文档。等合并索引段的时候将这个文档删除。

8、 文档路由

1）document路由到shard分片上，就叫做文档路由 ,如何路由?

2）路由算法

算法公式：shard = hash(routing)%number_of_primary_shards

例子：

一个索引index ,有3个primary shard : p0,p1,p2

增删改查 一个document文档时候，都会传递一个参数 routing number, 默认就是document文档 _id, (也可以手动指定)

Routing = _id，假设： _id = 1

算法：

Hash(1) = 21 % 3 = 0 表示 请求被 路由到 p0分片上面。

3）自定义路由

请求：

PUT /index/item/id?routing = _id （默认）

PUT /index/item/id?routing = user_id（自定义路由）--自定义分片key

4）primary shard不可变原因

即使加服务器也不能改变主分片的数量

五、实战的几个问题

1、索引应该设置多少个分片?

> 1）避免分片过大，因为这样会对集群从故障中恢复造成不利影响。尽管并没有关于分片大小的固定限值，但是人们通常将 50GB 作为分片上限，而且这一限值在各种用例中都已得到验证。分片过小会导致段过小，进而致使开销增加。您要尽量将分片的平均大小控制在至少几 GB 到几十 GB 之间。对时序型数据用例而言，分片大小通常介于 20GB 至 40GB 之间。

> 2）每个节点上可以存储的分片数量与可用的堆内存大小成正比关系，但是 Elasticsearch 并未强制规定固定限值。这里有一个很好的经验法则：确保对于节点上已配置的每个 GB，将分片数量保持在 20 以下。如果某个节点拥有 30GB 的堆内存，那其最多可有 600 个分片，但是在此限值范围内，您设置的分片数量越少，效果就越好。一般而言，这可以帮助集群保持良好的运行状态

注意：分片数量一旦设置，就不可更改 （primary shard）

2、分片应该设置多少个副本?

一般情况：

1-2个即可。

集群规模没有变化，副本分片过多?

浪费资源，占用资源，影响性能。

3、我们需要多少台服务器

根据分片数量计算，单节点索引分片数建议不要超过 3 个，每个索引分片推荐 20-40GB 大小。