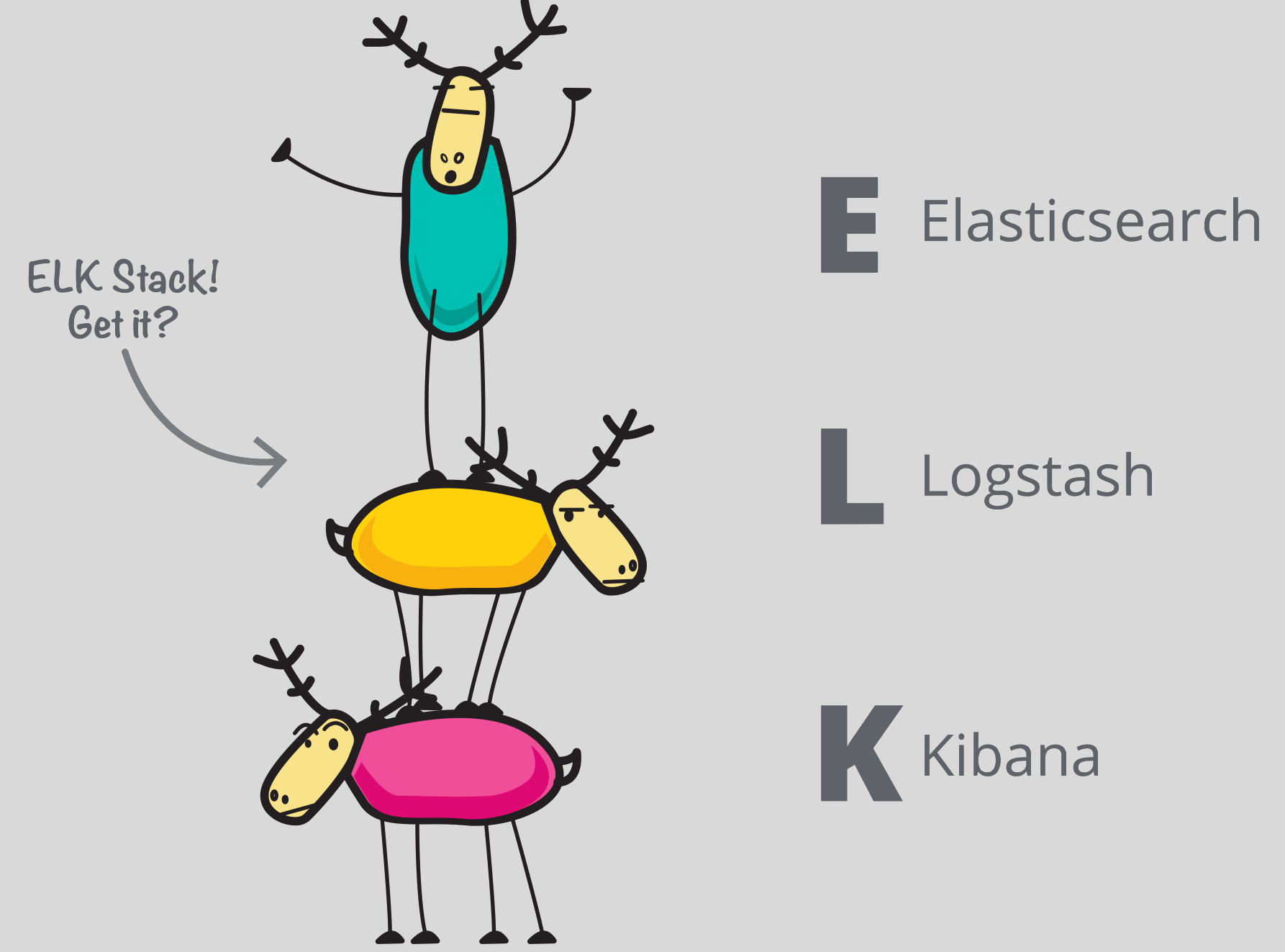
**ELK调研文档**

# 概述

## 介绍

ELK由ElasticSearch、Logstash和Kiabana三个开源工具组成，用于集中化管理日志、分析、可视化。

1. Elasticsearch是个开源分布式搜索引擎，它的特点有：分布式，零配置，自动发现，索引自动分片，索引副本机制，restful风格接口，多数据源，自动搜索负载等。
2. Logstash是一个完全开源的工具，他可以对你的日志进行收集、过滤，并将其存储供以后使用（如，搜索）。
3. Kibana 也是一个开源和免费的工具，它Kibana可以为 Logstash 和 ElasticSearch 提供的日志分析友好的 Web 界面，可以帮助您汇总、分析和搜索重要数据日志。



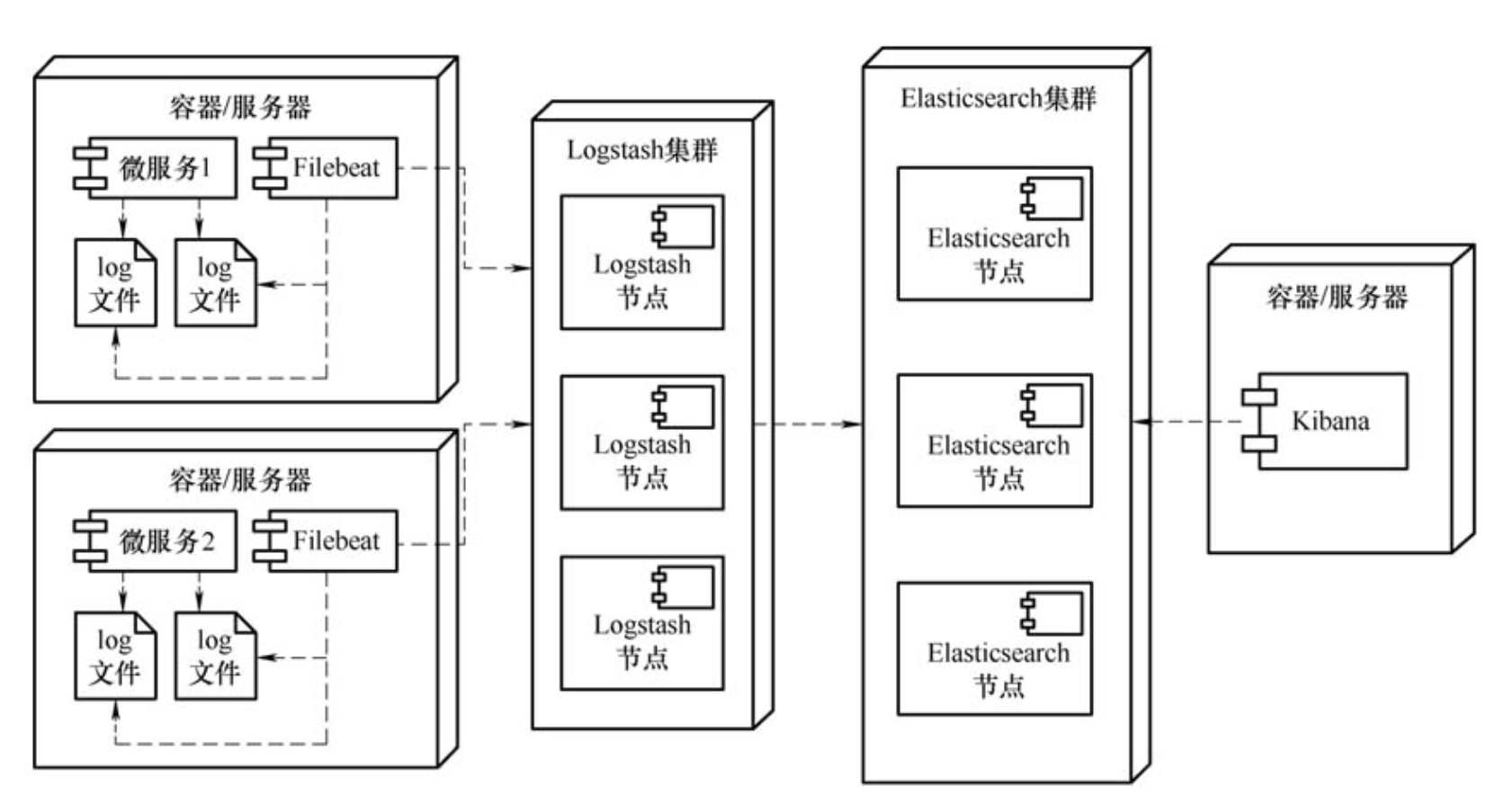
根据不同日志类型，有不同应用。日志有三种分类，包括：系统日志、应用程序日志、安全日志。系统运维和开发人员可以通过日志了解服务器软硬件信息、检查配置过程中的错误及错误发生的原因。经常分析日志可以了解服务器的负荷，性能安全性，从而及时采取措施纠正错误。

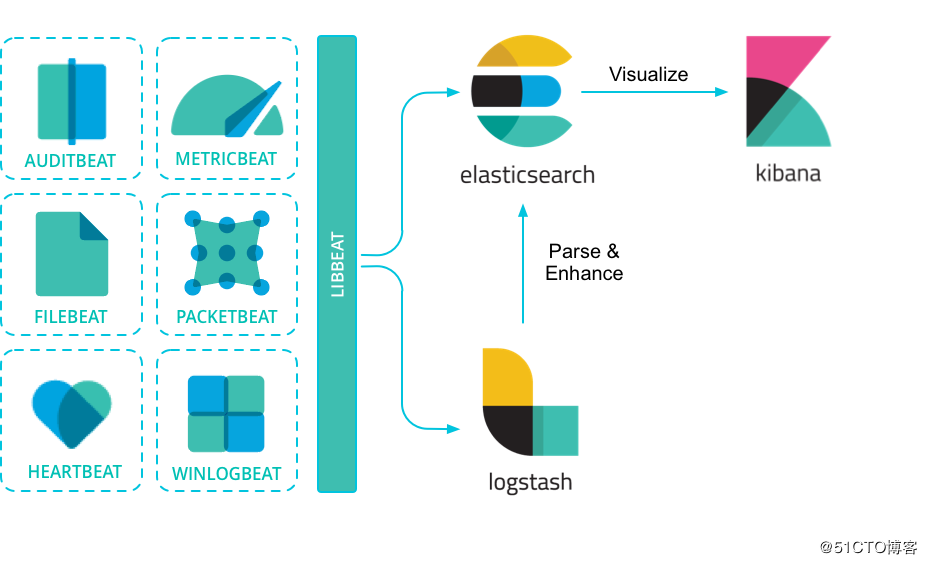
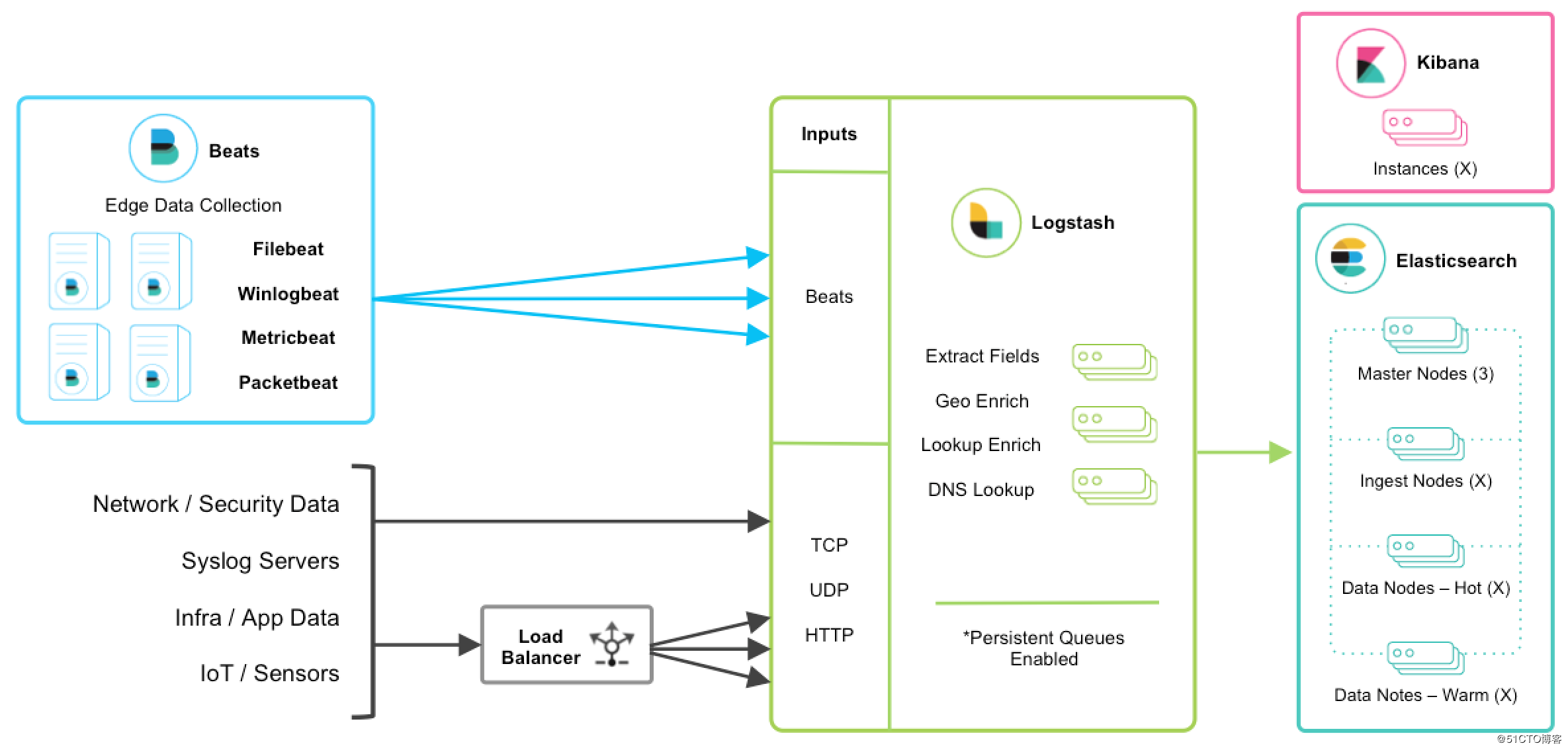
ELK 不是一款软件，而是 Elasticsearch、Logstash 和 Kibana 三种软件产品的首字母缩写。这三者都是开源软件，通常配合使用，而且又先后归于 Elastic.co 公司名下，所以被简称为 ELK Stack。根据 Google Trend 的信息显示，ELK Stack 已经成为目前最流行的集中式日志解决方案。

1. Elasticsearch：分布式搜索和分析引擎，具有高可伸缩、高可靠和易管理等特点。基于 Apache Lucene 构建，能对大容量的数据进行接近实时的存储、搜索和分析操作。通常被用作某些应用的基础搜索引擎，使其具有复杂的搜索功能；
2. Logstash：数据收集引擎。它支持动态的从各种数据源搜集数据，并对数据进行过滤、分析、丰富、统一格式等操作，然后存储到用户指定的位置；
3. Kibana：数据分析和可视化平台。通常与 Elasticsearch 配合使用，对其中数据进行搜索、分析和以统计图表的方式展示；
4. Filebeat：ELK 协议栈的新成员，一个轻量级开源日志文件数据搜集器，基于 Logstash-Forwarder 源代码开发，是对它的替代。在需要采集日志数据的 server 上安装 Filebeat，并指定日志目录或日志文件后，Filebeat 就能读取数据，迅速发送到 Logstash 进行解析，亦或直接发送到 Elasticsearch 进行集中式存储和分析。

Elastic Stack授权分为四级，即开源、基础级、黄金级和白金版，它们支持的功能依次增加。其中，开源和基础级授权所提供的功能可免费使用，而黄金级和白金版的功能则属于商业授权。

应用场景由于同属一个家族，Elastic Stack四大组件放在一起使用当然最为方便。Beats组件从分布式环境中的主机节点上采集数据发送给Logstash，而Logstash根据配置将数据过滤、清洗后再发送给Elasticsearch并编入索引，最后在Kibana中配置仪表盘、画布等并从Elasticsearch中读取数据将它们可视化。最典型的应用场景就是使用Elastic Stack从文件中收集日志，图1-1所示的就是一个在分布式环境中收集日志数据的部署图。





Elastic Stack中组件的作用：

● Elasticsearch：用于数据存储和数据检索，天然支持数据分片和复制，可以轻松实现扩容，在很多情况下也被当成NoSQL数据库独立使用。

● Logstash：用于数据清洗、数据传输，可以适配多种输入和输出数据源，同时还提供了丰富的过滤器插件。

● Kibana：用于数据可视化和数据分析，可以看作是Elasticsearch的界面。

●Filebeat：Beats组件中的一种，是一种安装于宿主机的轻量级组件，核心作用就是将宿主机上指定文件的内容提取出来并发送到指定的目的地。

# Elasticsearch

## 介绍

Elasticsearch是一个基于Apache Lucene全文搜索引擎开发的分布式的 RESTful 风格的的实时搜索与数据分析引擎，它比Lucene更强大，并且是开源的。Elasticsearch是面向文档型数据库，一条数据就是一个文档，和数据结构mongoDB类似，文档序列化之后是JSON格式，相当于关系型数据库中的一张表中的一条记录。

Apache Lucene是Apache软件基金会一个开放源代码的全文检索引擎工具包，一个全文检索引擎的架构，提供了完整的查询引擎和索引引擎、部分文本分析引擎。2012年Elastic把Lucene基础库包成了一个更好用的软件，并且在2015年推出ELK Stack（Elastic Logstash Kibana）解决集中式日志采集、存储和查询问题。

### 基本特征

1. 实现了用于全文检索的倒排索引，实现了用于存储数值数据和位置数据的 BKD 树， 以及用于分析的列存储。
2. 将每个字段编入索引，使其可搜索，提高搜索速度。
3. 实时分析的分布式引擎，确保故障时仍安全可用。
4. 可以在承载了 PB级数据的成百上千台服务器上运行。
5. 可处理多种数据类型，数字、文本、地理位置、结构化、非结构化。

### 解决问题

Elasticsearch能够解决的问题：在你的应用或网站中实现全文检索，存储和分析你的应用的日志，使用Elasticsearch作为存储引擎自动化业务工作流程，使用Elasticsearch作为地理信息系统（GIS）管理，集成和分析空间信息等。

使用Elasticsearch主要是用的强大的检索信息的能力。我们有很多的查询信息的需求，在数据量足够大的情况下，如果我们还是使用，关系型数据库的模糊查询语法进行数据信息检索的话，可想而知那是多么痛苦的一件事情。所以Es解决的问题就是信息的搜索。

### 核心概念

Elasticsearch 的基础。核心的概念学习一门新技术的时候，作者习惯将它和已经学会的知识进行映射对比，这样有助于理解新的技术。那么既然Elasticsearch能存储数据，能够搜索数据，那么理解它的一些概念的时候，我们不妨以我们熟悉的关系型数据库来对比这理解。

* 节点：一台Elasticsearch服务就是一个节点。多个节点进行简单的配置就能搭建一个集群
* 集群：Elasticsearch是一个天然的分布式系统，只需要简单的配置就能搭建一个es集群
* 索引：又叫索引库，数据可以存储在不同的索引中，可以理解为：传统的数据库， 不同的数据可以存储在不同的数据库中
* 文档：可以理解为传统数据库中的一行数据，由多个字段组成，是索引的基本存储单元
* 映射：可以理解为传统数据库的表结构，每个文档的的每个字段是什么类型，通过映射来定义
* 分片：Es在存储数据的时候，会将一个索引的分配到不同的数据分片中，然后将分片分散在集群中的不同节点上
* 副本：索引的备份，提高数据的安全性，防止数据的丢失，及时恢复数据
* 近实时：Elasticsearch从写入数据到查询到数据只需要1s的时间
* 字段：对应传统数据库的字段

## 索引

ES 中的数据可以概括为两类：精确值和全文。精确值有大小之分，检索中很少对全文类型的域做精确匹配。相反，在文本类型的域中搜索。不仅判断文档和给定查询条件的相关性，还希望搜索能够理解检索的意图

### 倒排索引

倒排索引先将文档中包含的关键字全部提取出来，然后再将关键字与文档的对应关系保存起来，最后再对关键字本身做索引排序。用户在检索某一关键字时，可以先对关键字的索引进行查找，再通过关键字与文档的对应关系找到所在文档。这类似于查字典一样，字典的拼音表和部首表就是关键字索引，而拼音表和部首表中的内容就是关键字与文档的对应关系。



在Elasticsearch中，添加或更新文档时最重要的动作是将它们编入倒排索引，未被编入倒排索引的文档将不能被检索。也就是说，Elasticsearch中所有数据的检索都必须要通过倒排索引来检索，离开了倒排索引文档就相当于不存在。所以从检索的角度来看，文档以倒排索引的形式表现其存在性。

因为文档存储前的分析和索引过程比较耗资源，所以为了提升性能，文档在添加到Elasticsearch中时并不会立即被编入索引。在默认情况下，Elasticsearch会每隔1s统一处理一次新加入的文档，可以通过index.refresh\_interval参数修改。为了提升性能，在Elasticsearch 7中还添加了index.search.idle.after参数，它的默认值是30s。其大体含义是，如果索引在一段时间内没有收到检索数据的请求，那么它至少要等30s后才会刷新索引数据。所以，从这两个参数的作用来看，Elasticsearch实际上是准实时的（Near Realtime，NRT）。也就是说，新添加到索引中的文档，有可能在一段时间内不能被检索到。如果的确需要立即检索到文档，Elasticsearch也提供了强制刷新到索引的方式，包括使用\_refresh接口和在操作文档时使用refresh参数。

### 不变性

倒排索引被写入磁盘后是不可改变的（它永远不会修改）。如果你需要让一个新的文档可被搜索，你需要重建整个索引。这要么对一个索引所能包含的数据量造成了很大的限制，要么对索引可被更新的频率造成了很大的限制。但不变性有重要的价值：

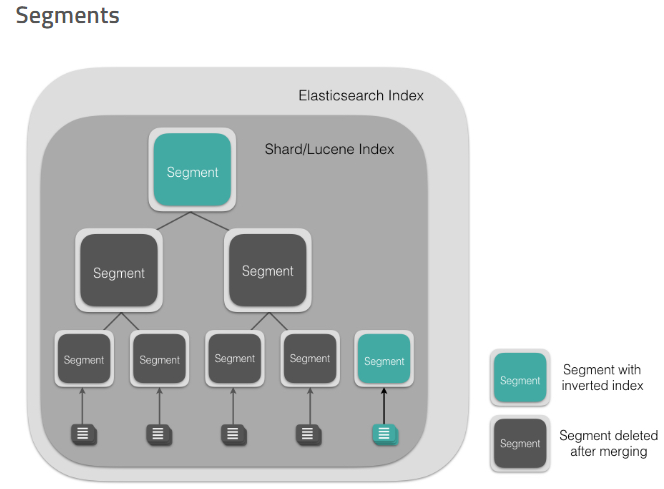
不需要锁，无并发问题；

提升性能，一旦索引被读入内核的文件系统缓存，只要有足够的空间，大部分请求直接请求内存，不会命中磁盘；

写入单个大的倒排索引允许数据被压缩，减少磁盘 I/O 和需要被缓存到内存的索引的使用量。

### 分片原理

在ES中，基本的存储单元是shard(分片)，但是在更底层的Lucene上稍微有点不同，ES的每一个shard是Lucene的一个index(索引)，Lucene的索引由多个segment组成，每个segment就是ES文档的倒序索引，里面包含了一些term(词)的mapping(映射)。

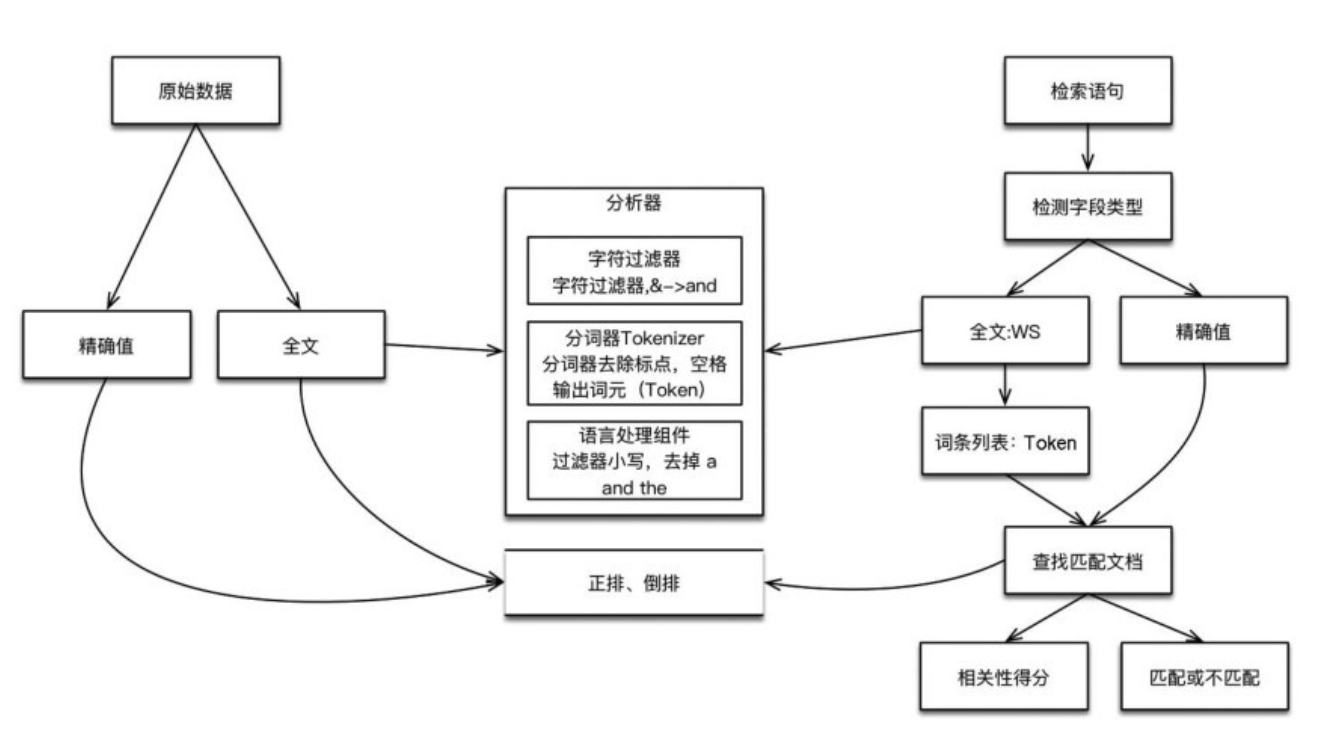


Segment 直接提供了搜索功能的，ES 的一个 Shard （Lucene Index）中是由大量的 Segment 文件组成的，且每一次 fresh 都会产生一个新的 Segment 文件，这样一来 Segment 文件有大有小，相当碎片化。ES 内部则会开启一个线程将小的 Segment 合并（Merge）成大的 Segment，减少碎片化，降低文件打开数，提升 I/O 性能。

Segment 文件是不可变更的。当一个 Document 更新的时候，实际上是将旧的文档标记为删除，然后索引一个新的文档。在 Merge 的过程中会将旧的 Document 删除掉。具体到文件系统来说，文档 A 是写入到 .cfs 文件里的，删除文档 A 实际上是在.del文件里标记某个 document 已被删除，那么下次查询的时候则会跳过这个文档，是为逻辑删除。当归并（Merge）的时候，老的 segment 文件将会被删除，合并成新的 segment 文件，这个时候也就是物理删除了。

### 分析器与规整器

在Elasticsearch中，文档编入索引时会从全文数据中提取词项，这个过程被称为文档分析（Analysis）。文档分析不仅存在于文档索引时，也存在于文档检索时。文档分析会从查询条件的全文数据中提取词项，然后再根据这些词项检索文档。文档分析器（Analyzer）是Elasticsearch中用于文档分析的组件，通常由字符过滤器（Character Filter）、分词器（Tokenizer）和分词过滤器（Token Filter）三部分组成。它们就像是连接在一起的管道，共同完成对全文数据的词项提取工作。



## 集群

### 节点类型

ES使用主从模式,主从模式可以简化系统设计，Master作为权威节点，部分操作仅由Master执行，并负责维护集群元信息。缺点是Master节点存在单点故障，需要解决灾备问题，并且集群规模会受限于Master节点的管理能力。

### 集群健康状态

从数据完整性的角度划分，集群健康状态分为三种：

1. Green，所有的主分片和副分片都正常运行。
2. Yellow，所有的主分片都正常运行，但不是所有的副分片都正常运行。这意味着存在单点故障风险。
3. Red，有主分片没能正常运行。

### 集群状态

集群状态元数据是全局信息，元数据包括内容路由信息、配置信息等，其中最重要的是内容路由信息，它描述了“哪个分片位于哪个节点”这种信息。集群状态由主节点负责维护，如果主节点从数据节点接收更新，则将这些更新广播到集群的其他节点，让每个节点上的集群状态保持最新。

### 集群扩容

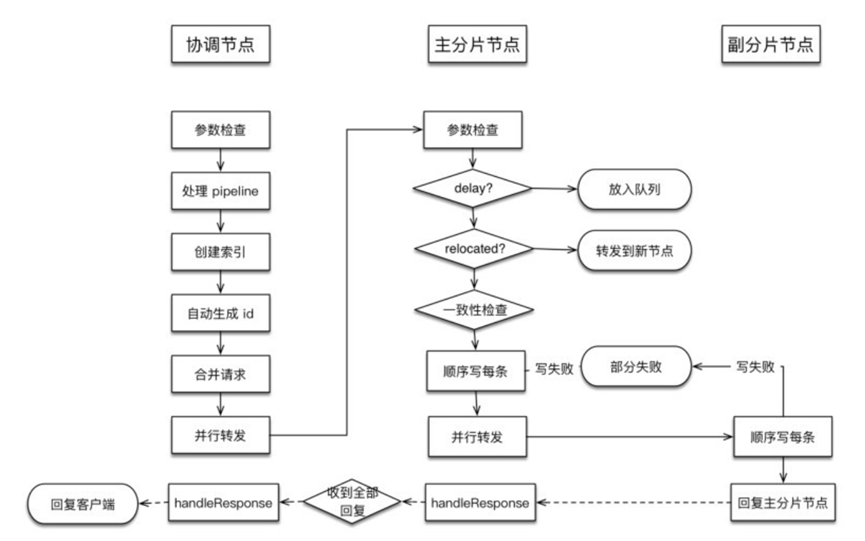
当扩容集群、添加节点时，分片会均衡地分配到集群的各个节点，从而对索引和搜索过程进行负载均衡，这些都是系统自动完成的。分片副本实现了数据冗余，从而防止硬件故障导致的数据丢失。

分片分配过程中除了让节点间均匀存储，还要保证不把主分片和副分片分配到同一节点，避免单个节点故障引起数据丢失。分布式系统中难免出现故障，当节点异常时，ES会自动处理节点异常。当主节点异常时，集群会重新选举主节点。当某个主分片异常时，会将副分片提升为主分片。

## 写入过程

每个索引操作首先会使用routing参数解析到副本组，通常基于文档ID。一旦确定副本组，就会内部转发该操作到分片组的主分片中。主分片负责验证操作和转发它到其他副分片。ES维护一个可以接收该操作的分片的副本列表。这个列表叫作同步副本列表（in-sync copies），并由Master节点维护。正如它的名字，这个“好”分片副本列表中的分片，都会保证已成功处理所有的索引和删除操作，并给用户返回 ACK。主分片负责维护不变性（各个副本保持一致），因此必须复制这些操作到这个列表中的每个副本。

写入过程遵循以下基本流程：



（1）请求到达协调节点，协调节点先验证操作，如果有错就拒绝该操作。然后根据当前集群状态，请求被路由到主分片所在节点。

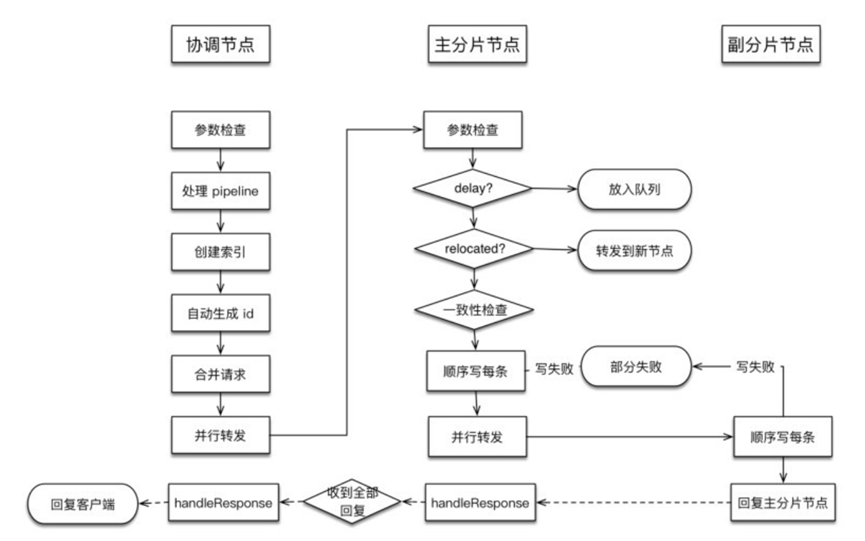
（2）该操作在主分片上本地执行，例如，索引、更新或删除文档。这也会验证字段的内容，如果未通过就拒绝操作（例如，字段串的长度超出Lucene定义的长度）。

（3）操作成功执行后，转发该操作到当前in-sync 副本组的所有副分片。如果有多个副分片，则会并行转发。

（4）一旦所有的副分片成功执行操作并回复主分片，主分片会把请求执行成功的信息返回给协调节点，协调节点返回给客户端。

## 读取过程

通过 ID 读取是非常轻量级的操作，而一个巨大的复杂的聚合查询请求需要消耗大量 CPU和内存资源。主从模式的一个好处是保证所有的分片副本都是一致的（正在执行的操作例外）。因此，单个in-sync中的某个副本也可以提供服务。当一个读请求被协调节点接收，这个节点负责转发它到其他涉及相关分片的节点，并整理响应结果发送给客户端。接收用户请求的这个节点称为协调节点。基本流程如下：



（1）把读请求转发到相关分片。因为大多数搜索都会发送到一个或多个索引，通常需要从多个分片中读取，每个分片都保存这些数据的一部分。

（2）从副本组中选择一个相关分片的活跃副本。它可以是主分片或副分片。默认情况下， ES会简单地循环遍历这些分片。

（3）发送分片级的读请求到被选中的副本。

（4）合并结果并给客户端返回响应。注意，针对通过ID查找的get请求，会跳过这个步骤，因为只有一个相关的分片。

## 集群中的分析与检索

ES 中的每个索引都会被拆分为多个分片，并且每个分片都有多个副本。这些副本称为replication group（副本组，与PacificA中的副本组概念一致），并且在删除或添加文档的时候，各个副本必须同步。否则，从不同副本中读取的数据会不一致。我们把保持分片副本之间的同步，以及从中读取的过程称为数据副本模型（data replication model）。ES的数据副本模型基于主备模式（primary-backup model），主分片是所有索引操作的入口，它负责验证索引操作是否有效。一旦主分片接受一个索引操作，主分片的副分片也会接受该操作。下面讨论数据副本模型在写操作和读操作时如何交互。

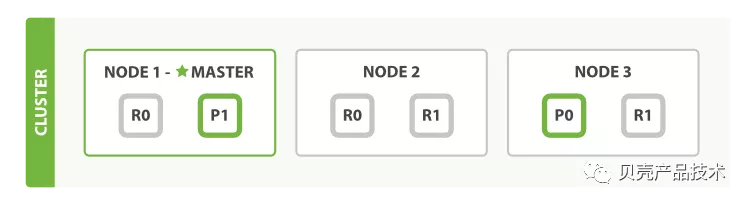
### 路由一个文档到一个分片中

公式：shard = hash(routing) % number\_of\_primary\_shards

根据公式创建文档的时候确定一个文档存放在哪个一个分片中。routing 是一个可变值，默认是文档的 \_id，也可以设置成一个自定义的值。 routing 通过 hash 函数生成一个数字，然后这个数字再除以 number\_of\_primary\_shards（主分片的数量）后得到余数 。这个分布在 0 到 number\_of\_primary\_shards - 1 之间的余数，就是我们所寻求的文档所在分片的位置。

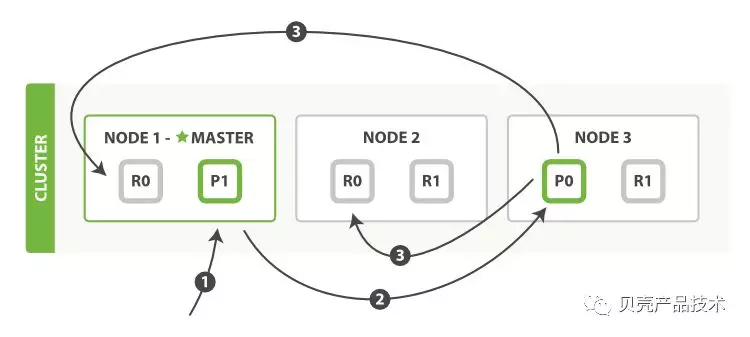
### 主分片和副分片的交互

以官网的例子进行分析，从图中能看出一个集群由三个节点组成，有两个分片，两个副本。



上图中集群中的任意一个节点有能力处理任意的请求，每个节点都知道集群中任意一个文档所处的位置。

### 写一个文档操作



上图中写操作必须在主分片上面完成之后，才能被复制到其他节点作为分片副本，新建、索引和删除请求都是写操作。

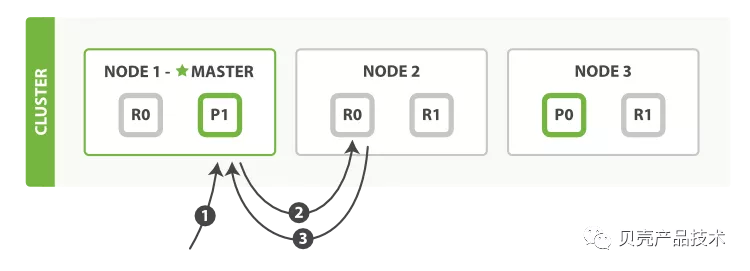
1）客户端向 master 发送写入请求，该节点作为协调节点；

2）根据文档的 \_id 确定分片, 图中请求文档属于分片 0，协调节点请求转到节点的主分片；

3）在节点 3 上执行请求，成功之后，节点 3 根据副本数将请求并行转到副本分片对应节点，一旦副本分片执行完成，都向节点 3 报告成功，节点 3 将向协调节点报告成功，协调节点再向客户端报告成功。

客户端收到成功响应时，则变更操作是安全的。这个过程中有些请求参数影响效率。

### 取回一个文档



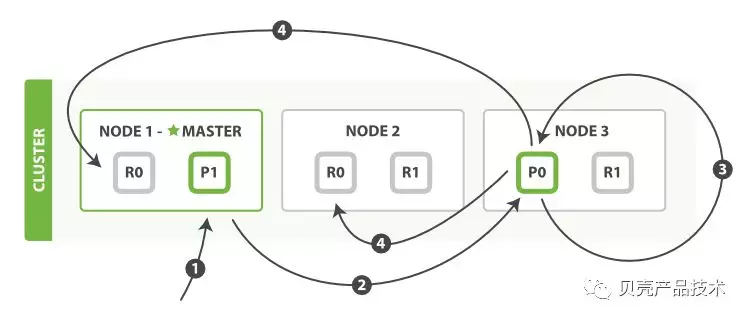
1）客户端向 master 发送获取请求，该节点作为协调节点；

2）根据文档的 \_id 确定分片, 图中请求文档属于分片 0，分片 0 的主副分片都在三个节点上，这儿将请求转到节点 2；

3）节点 2 将文档返回给节点 1，然后将文档返回给客户端。

在文档被检索时，已经被索引的文档可能已经存在于主分片上，但是还没有复制到副本分片。 在这种情况下，副本分片可能会报告文档不存在，但是主分片可能成功返回文档。 一旦索引请求成功返回给用户，文档在主分片和副本分片都是可用的。

### 局部更新文档



1）客户端向 master 发送更新请求，该节点作为协调节点；

2）根据文档的 \_id 确定分片, 图中请求文档属于分片 0，协调节点将请求转到节点 3；

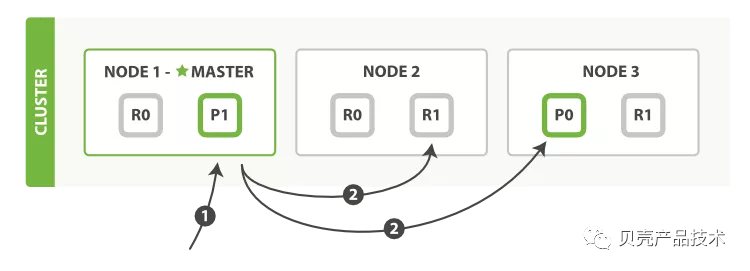
3）节点 3 从主分片检索文档，修改 \_source 字段中的 JSON ，并且尝试重新索引主分片的文档。如果文档已经被另一个进程修改，它会重试步骤 3，超过 retry\_on\_conflict 次后放弃；

4）如果节点 3 成功地更新文档，它将新版本的文档并行转发到其他节点上的副本分片，重新建立索引。一旦所有副本分片都返回成功，节点 3 向协调节点也返回成功，协调节点向客户端返回成功。

### 多文档模式

mget 和 bulk API 的模式类似于单文档模式。区别在于协调节点知道每个文档存在于哪个分片中。 它将整个多文档请求分解成 每个分片 的多文档请求，并且将这些请求并行转发到每个参与节点。协调节点一旦收到来自每个节点的应答，就将每个节点的响应收集整理成单个响应，返回给客户端。

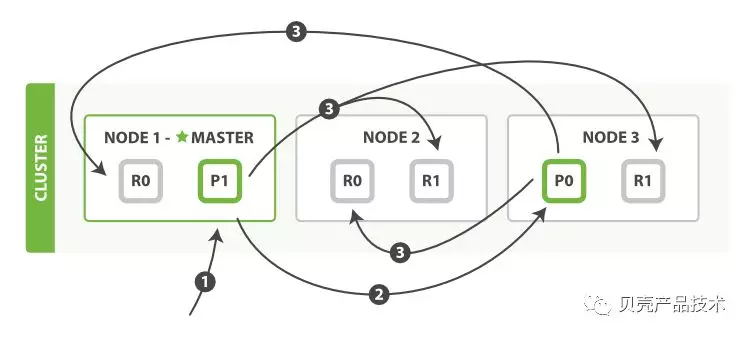
使用 mget 取回多个文档



1）客户端向 master 发送 mget 请求，该节点作为协调节点；

2）节点 1 为每个分片构建多文档获取请求，然后并行转发这些请求到托管在每个所需的主分片或者副本分片的节点上。一旦收到所有答复，节点 1 构建响应并将其返回给客户端。

使用 bulk 修改多个文档



1）客户端向 master 发送 bluk 请求，该节点作为协调节点；

2）节点 1 为每个节点创建一个批量请求，并将这些请求并行转发到每个包含主分片的节点主机；

3）主分片一个接一个按顺序执行每个操作。当每个操作成功时，主分片并行转发新文档（或删除）到副本分片，然后执行下一个操作。一旦所有的副本分片报告所有操作成功，该节点将向协调节点报告成功，协调节点将这些响应收集整理并返回给客户端。

# Kibana

·Discover：用于分析数据，可以访问每个字段并附上默认的时间。

·Visualize：用于创建对Elasticsearch索引中数据的可视化展示。可以接下来创建仪表板，展示相关的可视化内容。

·Dashboard：用于展示一系列已保存的可视化成果。

·Timelion：这是一个基于时间序列的数据可视化器，可以在一个可视化中合并展示完全不相关的数据源的内容。它也是基于简单的表示语言的。

·Management：在这里对运行时的Kibana进行配置，包括初始化安装和对索引模式的运行时配置，以及一些高级设置，可以调节Kibana自身和已保存的可视化内容的行为。

·Dev Tool：包含基于Sense插件的终端，可以在一个标签中写Elasticsearch命令，在另一个标签中看命令结果。

## Kibana文档发现

文档发现（Discover），它提供了交互式检索文档的接口，用户可以在这里提交查询条件、设置过滤器并查看检索结果。在文档发现中的查询条件还可以保存起来，这些保存起来的查询条件称为查询对象，可以在文档可视化和仪表盘功能中使用，也可以CSV或链接的形式分享出来。

### 索引模式

在使用文档发现功能检索文档之前，首先要告诉Kibana要检索Elasticsearch的哪些索引，这在Kibana中是通过定义索引模式来实现的。没有被索引模式包含进来的索引不能在文档发现、文档可视化和仪表盘等功能中使用，本节主要讲解如何创建索引模式。

索引模式是一种对Elasticsearch中索引的模式匹配，以定义哪些索引将被包含到这个模式中。它以索引名称为基础，可以匹配单个索引也可以使用星号“\*”匹配多个索引。

索引模式的管理功能位于Management菜单中，如图所示。

### 时间范围与过滤器

文档发现就是要将满足条件的文档检索出来，Kibana提供了多种方式设置查询条件。这包括通过时间范围过滤文档、使用过滤器过滤文档，还可以通过Lucene或KQL查询语言过滤文档。无论使用哪一种方式过滤文档，它们最终都会以DSL查询语言的形式传递给底层的Elasticsearch。

时间范围和过滤器这两种方式，它们最终会以must子句的形式组合进bool查询。

### 使用查询语言

时间范围和过滤器设置的查询条件都是以逻辑与的形式组合在一起的，如果需要设置更复杂的查询条件就需要在查询栏中输入查询条件以检索文档。目前Kibana文档发现中支持Lucene和KQL（Kibana Query Language）两种查询语言，前者可以认为就是Elasticsearch中的DSL，而后者则是Kibana提供的一种新查询语言。

### 文档展示与字段过滤

通过时间范围、过滤器和查询语言发现的文档最终会以柱状图和表格两种形式展示，但只有表格可以通过CSV或链接形式分享出来，而柱状图更多地是为了方便在界面中选取文档。

索引模式是Kibana中读取索引数据的基本单元，只有与索引模式关联起来的索引才能在Kibana中使用。索引模式通过定义名称模式的形式匹配一个或多个索引名称，同时还可以通过源过滤器过滤掉不需要的字段，或通过脚本添加额外的字段。对于包含有日期类型字段的文档，还可以给索引模式定义时间过滤器。文档发现功能以界面交互的形式设置查询条件，可使用时间栏、过滤器栏设置简单的查询条件；也可以通过Lucene和KQL两种查询语言设置复杂的查询条件。Lucene本质上就是DSL，而KQL则是Kibana自定义的语言，使用起来更简单且具有强大的提示功能。

文档发现中设置的查询条件可以保存起来供可视化或仪表盘使用，也可以通过CSV或链接形式分享出去。

文档发现仍然是以简单的表格展示文档，接下来两章将会介绍Kibana中丰富的图表功能。

## Kibana文档可视化

Kibana可视化功能以图表形式展示Elasticsearch中的文档数据，能够让用户以最直观的形式了解数据变化的趋势、峰谷值或形成对比。这些图表根据查询条件从索引中提取文档，查询条件可以在文档可视化界面中定制，也可以使用在文档发现中保存的查询对象。文档可视化生成的图表也可以保存，本书后续章节称这些保存起来的可视化图表为可视化对象。在进入文档可视化界面时将会列出Kibana中所有的可视化对象，如果已经将Kibana提供的样例数据导入，则在这个页面上将列出几十个不同的可视化对象。读者可以逐一点开查看这些可视化对象，它们是学习文档可视化的好素材。在列表上方有搜索框，可根据名称搜索文档可视化对象

Kibana提供的可视化图表分为二维坐标图、圆形圆弧、热度等几大类

## Kibana综合展示

尽管查询对象和可视化对象都可以单独分享出来供其他系统使用，但它们展示出来的数据和图像比较单薄，只能反映系统的一个侧面。在实际应用中，人们更希望能够在一个页面中体现数据的各方面情况。Kibana提供的仪表盘和画布满足了这种需求，它们可以将多个查询对象和可视化对象组合到一起，共同将数据包含的信息展示出来。

除了可以展示Elasticsearch文档数据，Kibana还提供对ElasticStack组件监控数据的可视化功能，这就是Kibana的监控功能。此外，在Kibana版本7中还新增加了地图功能，本节也将一并介绍。

仪表盘仪表盘（Dashboard）是Kibana提供的综合展示数据的功能，在Kibana中保存的查询对象和可视化对象可以在仪表盘中组合起来共同展示。

画布仪表盘将查询对象和可视化对象组合在一起，可以通过图表的形式直观地展示一组数据的各个方面。但仪表盘只能使用Kibana中已有的图表形式，而不支持使用自定义图片等其他更为生动的形式展示数据。同时由于仪表盘中包含的图表专业性比较强，更适合于开发和运维人员监控系统或排查错误，而不适合商务场合下展示，所以许多商务背景下的数据展示还是通过其他方式单独制作。为了提供更为生动更为灵活的数据展示方式，Kibana从版本6.5开始引入了另一项功能——画布（Canvas），以类似幻灯片的形式展示数据。

监控在分布式环境下，监控无疑是一个非常重要的话题，系统越重要，对这个系统的监控也就越重要。Elastic Stack组件本身可以用于对其他系统做监控，比如可以使用Metricbeat收集其他系统的指标信息，然后再在Kibana中展示出来就实现了对该系统的监控。监控往往与报警联系在一起，监控收集数据但不理解数据；而报警则通过一些预警指标理解监控数据，并在系统指标数据异常时向管理人员发出警报。

Elastic Stack监控可以收集Elasticsearch、Logstash和Kibana运行数据，监控数据会被存储到Elasticsearch中，然后再通过Kibana监控功能将这些数据可视化。Elasticsearch在整个Elastic Stack监控体系中处于核心地位，它不仅要收集自身产生的监控数据，还要负责收集Logstash和Kibana产生的监控数据。这些监控数据会被一种称为导出器（Exporter）的组件保存在本地Elasticsearch索引中，或者通过HTTP协议发送给其他的Elasticsearch集群。所以要想监控Elastic Stack组件，首先要开启Elasticsearch的监控功能。

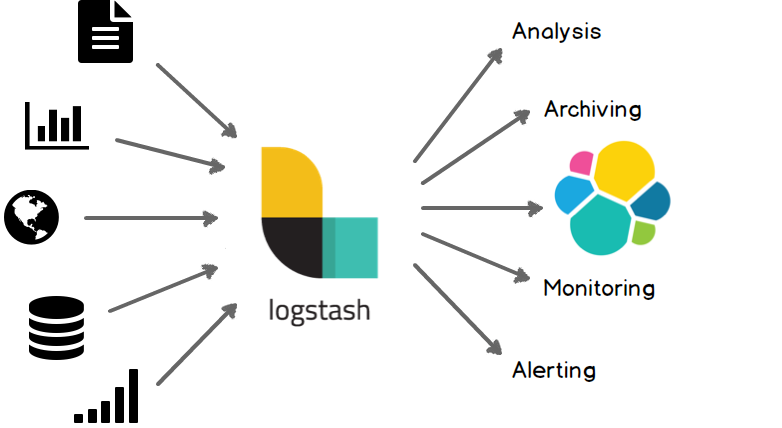
仪表盘是对查询对象和可视化对象的综合展示，这些对象在文档发现功能和文档可视化功能中创建并保存。仪表盘提供了对这些对象的布局管理，还可以通过控件筛选要展示的数据。仪表盘可以直接在Kibana中通过浏览器查看，也可以通过iFrame或链接的形式分享出去。

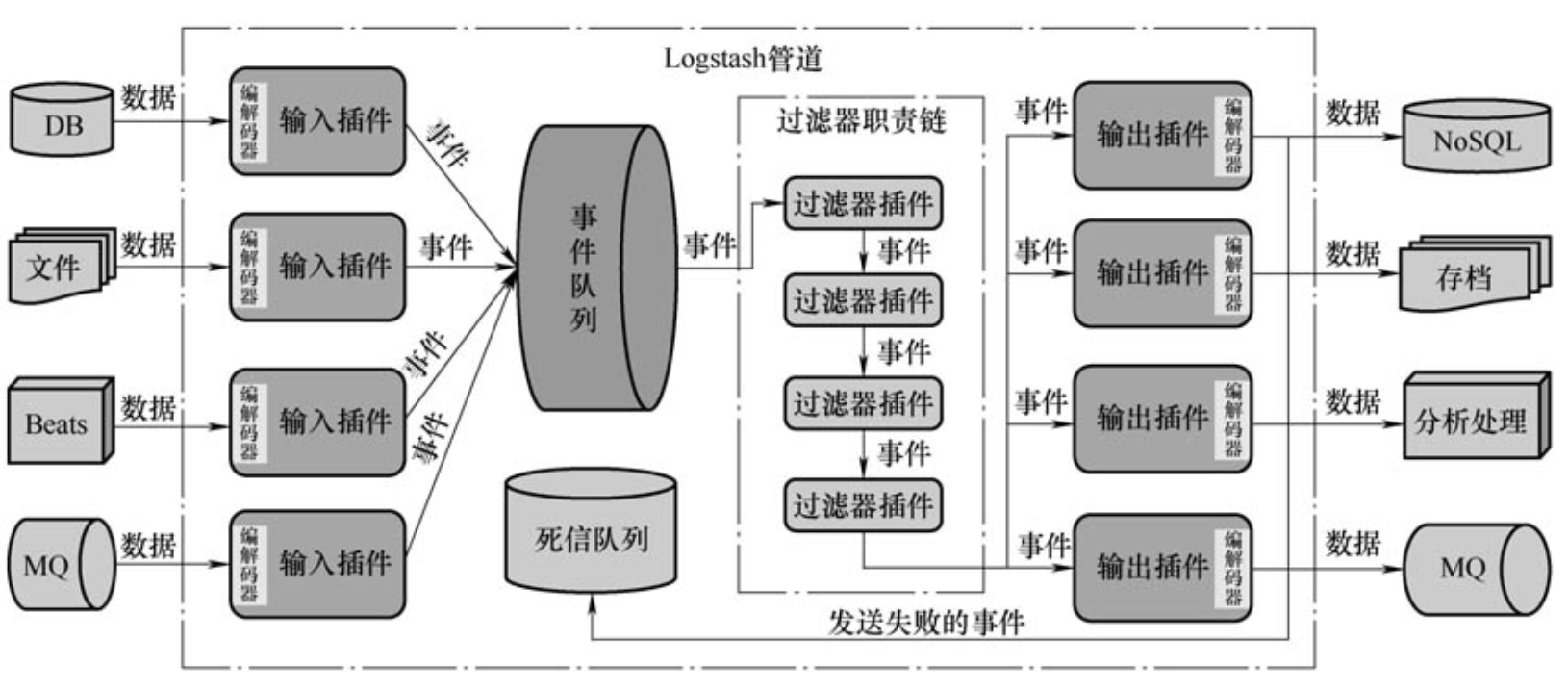
画布是另一种更为灵活的综合数据展示形式，它的一个显著特点就是在画布中可以添加自定义的图片。制作画布的单元称为workpad，这类似于在桌面应用中使用的PPT。画布由于更灵活更生动，比较适合商务场景下的数据展示，所以画布也提供了以PDF格式导出的功能。监控是Kibana提供的对Elastic Stack组件监控数据可视化的功能，可用于监控Elasticsearch、Logstash和Kibana自身。在开启了Elasticsearch监控数据收集后，Elasticsearch和Kibana的监控信息会在监控界面中展示出来。有关Logstash和Beats组件的监控将在后续章节中介绍。

# Logstash

Logstash是一个类似实时流水线的开源数据传输引擎，它像一个两头连接不同数据源的数据传输管道，将数据实时地从一个数据源传输到另一个数据源中。在数据传输的过程中，Logstash还可以对数据进行清洗、加工和整理，使数据在到达目的地时直接可用或接近可用，为更复杂的数据分析、处理以及可视化做准备。

Logstash的核心价值就在于它将业务系统与数据处理系统隔离开来，屏蔽了各自系统变化对彼此的影响，使系统之间的依赖降低并可独自进化发展。





## Logstash工作原理：

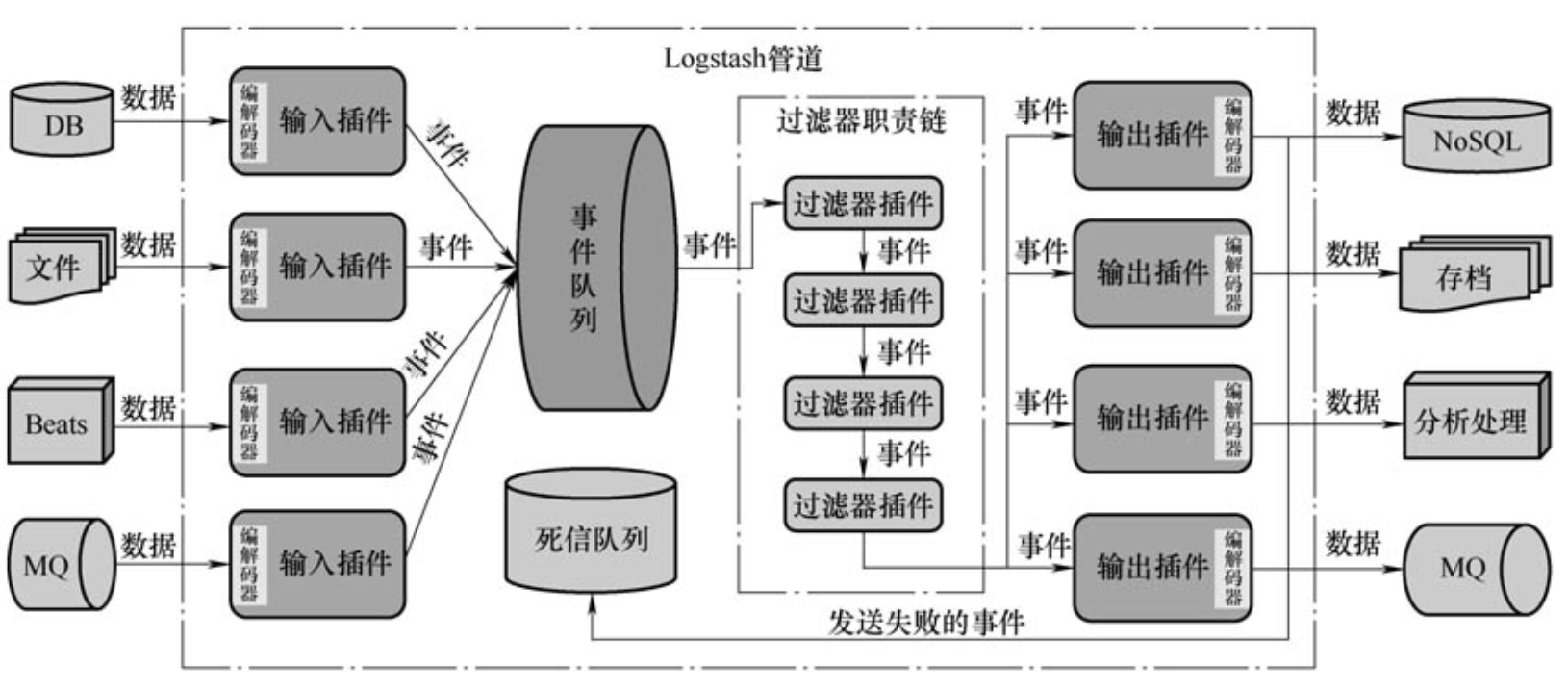
Logstash数据传输管道所具备的流水线特征，体现在数据传输过程分为三个阶段——输入、过滤和输出。这三个阶段按顺序依次相连，像一个加工数据的流水线。在实现上，它们分别由三种类型的插件实现，即输入插件、过滤器插件和输出插件，并可通过修改配置文件实现快速插拔。除了这三种类型的插件以外，还有一种称为编解码器（Codec）的插件。编解码器插件用于在数据进入和离开管道时对数据做解码和编码，所以它一般都是与具体的输入插件或输出插件结合起来使用。

事件（Event）是Logstash中另一个比较重要的概念，它是对Logstash处理数据的一种面向对象的抽象。如果将Logstash比喻为管道，那么事件就是流淌在管道中的涓涓细流。事件由输入插件在读入数据时产生，不同输入插件产生事件的属性并不完全相同，但其中一定会包含有读入的原始数据。过滤器插件会对这些事件做进一步处理，处理的方式主要体现在对事件属性的访问、添加和修改。最后，输出插件会将事件转换为目标数据源的数据格式并将它们存储到目标数据源中。

事件（Event）是Logstash中另一个比较重要的概念，它是对Logstash处理数据的一种面向对象的抽象。如果将Logstash比喻为管道，那么事件就是流淌在管道中的涓涓细流。事件由输入插件在读入数据时产生，不同输入插件产生事件的属性并不完全相同，但其中一定会包含有读入的原始数据。过滤器插件会对这些事件做进一步处理，处理的方式主要体现在对事件属性的访问、添加和修改。最后，输出插件会将事件转换为目标数据源的数据格式并将它们存储到目标数据源中。

事件（Event）是Logstash中另一个比较重要的概念，它是对Logstash处理数据的一种面向对象的抽象。如果将Logstash比喻为管道，那么事件就是流淌在管道中的涓涓细流。事件由输入插件在读入数据时产生，不同输入插件产生事件的属性并不完全相同，但其中一定会包含有读入的原始数据。过滤器插件会对这些事件做进一步处理，处理的方式主要体现在对事件属性的访问、添加和修改。最后，输出插件会将事件转换为目标数据源的数据格式并将它们存储到目标数据源中。

事件（Event）是Logstash中另一个比较重要的概念，它是对Logstash处理数据的一种面向对象的抽象。如果将Logstash比喻为管道，那么事件就是流淌在管道中的涓涓细流。事件由输入插件在读入数据时产生，不同输入插件产生事件的属性并不完全相同，但其中一定会包含有读入的原始数据。过滤器插件会对这些事件做进一步处理，处理的方式主要体现在对事件属性的访问、添加和修改。最后，输出插件会将事件转换为目标数据源的数据格式并将它们存储到目标数据源中。



插件插件（Plugin）最吸引人的特征就是它的可插拔性，通过简单的配置就可接入系统并增强系统功能。Logstash提供了丰富的输入、过滤器、输出和编解码器插件，它们也都是通过配置接入系统并增强Logstash在某一方面的功能。输入插件的作用是使数据进入管道并生成数据传输事件，过滤器插件则对进入管道的事件进行修改、清洗等预处理，而输出插件则将过滤器处理好的事件发送到目标数据源。输入插件和输出插件都可以使用编解码器在数据进入或退出管道时对数据进行编码或解码，使数据以用户期望的格式进入或退出管道。

事件Logstash事件由一组属性组成，包括数据本身、事件产生时间、版本等。不同输入插件产生事件的属性各不相同，这些事件属性可以在过滤器插件和输出插件中访问、添加、修改或是删除。由于事件本身由输入插件产生，所以在输入插件中不能访问事件及其属性。这里所说的对事件及其属性的访问是指在Logstash管道配置中的访问，比如过滤器插件配置中根据事件属性执行不同的过滤等。

队列在互联网时代，许多活动或突发事件会导致应用访问量在某一时间点瞬间呈几何式增长。在这种情况下，应用产生的数据也会在瞬间爆发，而类似Logstash这样的数据管道要搬运的数据也会突然增加。如果没有应对这种瞬间数据爆炸的机制，轻则导致应用数据丢失，重则直接导致系统崩溃，甚至引发雪崩效应将其他应用一并带垮。应对瞬间流量爆炸的通用机制是使用队列，将瞬间流量先缓存起来再交由后台系统处理。后台系统能处理多少就从队列中取出多少，从而避免了因流量爆炸导致的系统崩溃。Logstash输入插件对接的事件队列实际上就是应对瞬间流量爆炸、提高系统可用性的机制，它利用队列先进先出的机制平滑事件流量的峰谷，起到了削峰填谷的重要作用。

除了输入插件使用的事件队列，输出插件还有一个死信队列。这个队列将会保存输出插件没有成功发送出去的事件，它的作用不是削峰填谷而是容错。

Logstash事件处理有三个阶段：inputs → filters → outputs。是一个接收，处理，转发日志的工具。支持系统日志，webserver日志，错误日志，应用日志，总之包括所有可以抛出来的日志类型。

## 执行模型

Logstash事件处理管道协调输入、过滤器和输出的执行。

Logstash管道中的每个输入阶段都在自己的线程中运行。输入将事件写入内存（默认）或磁盘上的中央队列。然后通过一个已配置的事件的批处理运行该批处理的事件的输出。批处理的大小和管道工作线程的数量是可配置的（请参阅调整和分析Logstash性能）。

默认情况下，Logstash在管道阶段（input→filter and filter→output）之间使用内存限制队列来缓冲事件。如果Logstash不安全地终止，则存储在内存中的所有事件都将丢失。为了帮助防止数据丢失，可以启用Logstash将正在运行的事件持久化到磁盘。有关详细信息，请参阅持久队列。

## Input：输入数据到logstash。

一些常用的输入为：

file：从文件系统的文件中读取，类似于tail -f命令

syslog：在514端口上监听系统日志消息，并根据RFC3164标准进行解析

redis：从redis service中读取

beats：从filebeat中读取

## Filters：数据中间处理，对数据进行操作。

一些常用的过滤器为：

grok：解析任意文本数据，Grok 是 Logstash 最重要的插件。它的主要作用就是将文本格式的字符串，转换成为具体的结构化的数据，配合正则表达式使用。内置120多个解析语法。

官方提供的grok表达式：https://github.com/logstash-plugins/logstash-patterns-core/tree/master/patterns

grok在线调试：https://grokdebug.herokuapp.com/

mutate：对字段进行转换。例如对字段进行删除、替换、修改、重命名等。

drop：丢弃一部分events不进行处理。

clone：拷贝 event，这个过程中也可以添加或移除字段。

geoip：添加地理信息(为前台kibana图形化展示使用)

Outputs：outputs是logstash处理管道的最末端组件。一个event可以在处理过程中经过多重输出，但是一旦所有的outputs都执行结束，这个event也就完成生命周期。

## 一些常见的outputs为：

elasticsearch：可以高效的保存数据，并且能够方便和简单的进行查询。

file：将event数据保存到文件中。

graphite：将event数据发送到图形化组件中，一个很流行的开源存储图形化展示的组件。

Codecs：codecs 是基于数据流的过滤器，它可以作为input，output的一部分配置。Codecs可以帮助你轻松的分割发送过来已经被序列化的数据。

## 一些常见的codecs：

json：使用json格式对数据进行编码/解码。

multiline：将汇多个事件中数据汇总为一个单一的行。比如：java异常信息和堆栈信息。

# Filebeat

## Beats原理与结构

Beats这个单词在英文中有跳动的意思，而在Elastic Stack中，它指代的是一种轻量级的组件。Beats这个名称源于计算机领域中的“心跳（Heartbeat）”概念，即设备或服务每隔一段时间向监控服务发送一小段数据以表明自身存活。由于“心跳”只是存活标识而与业务无关，所以“心跳”本身应尽可能少地消耗计算资源。与普通的“心跳”不同，Beats组件主要的功能不是标识存活而是收集数据，所以它们一般都会安装到业务服务所在的宿主机上，并且按一定的时间间隔或周期采集特定数据。由于Beats组件会与业务服务运行在同一计算资源上，所以Beats组件也必须要尽可能少地消耗计算资源，以避免由于消耗过多计算资源而导致业务服务不可用。所以Beats组件的基本特征是需要安装于数据产生的宿主机，并以一定的时间间隔或周期轻量地收集数据。

Beats组件的轻量级特征是它和Logstash组件共存的一个重要原因。虽然它们都有数据收集、传输的功能，但Beats主要用于数据收集而Logstash主要用于数据传输。以Filebeat为例，它的主要作用是收集宿主机上的文件内容；而Logstash本身也包含一个文件输入类型，主要作用也是收集宿主机上的文件内容。但不同的是，Logstash本身由Java语言开发，是重量级的组件；而Filebeat则是由Go语言开发，是轻量级的组件。Logstash在启动、运行时会消耗大量内存、CPU等计算资源，如果与业务服务安装在一起就有可能会影响到业务服务的运行。但是正是由于Logstash的重量级特性，又使得Logstash运行稳定且包含完备的数据传输和数据过滤功能，而这又是Beats组件不具备的。Beats组件的轻量性要求，决定了它不可能具备复杂的数据加工能力，而这些就只能留给Logstash。

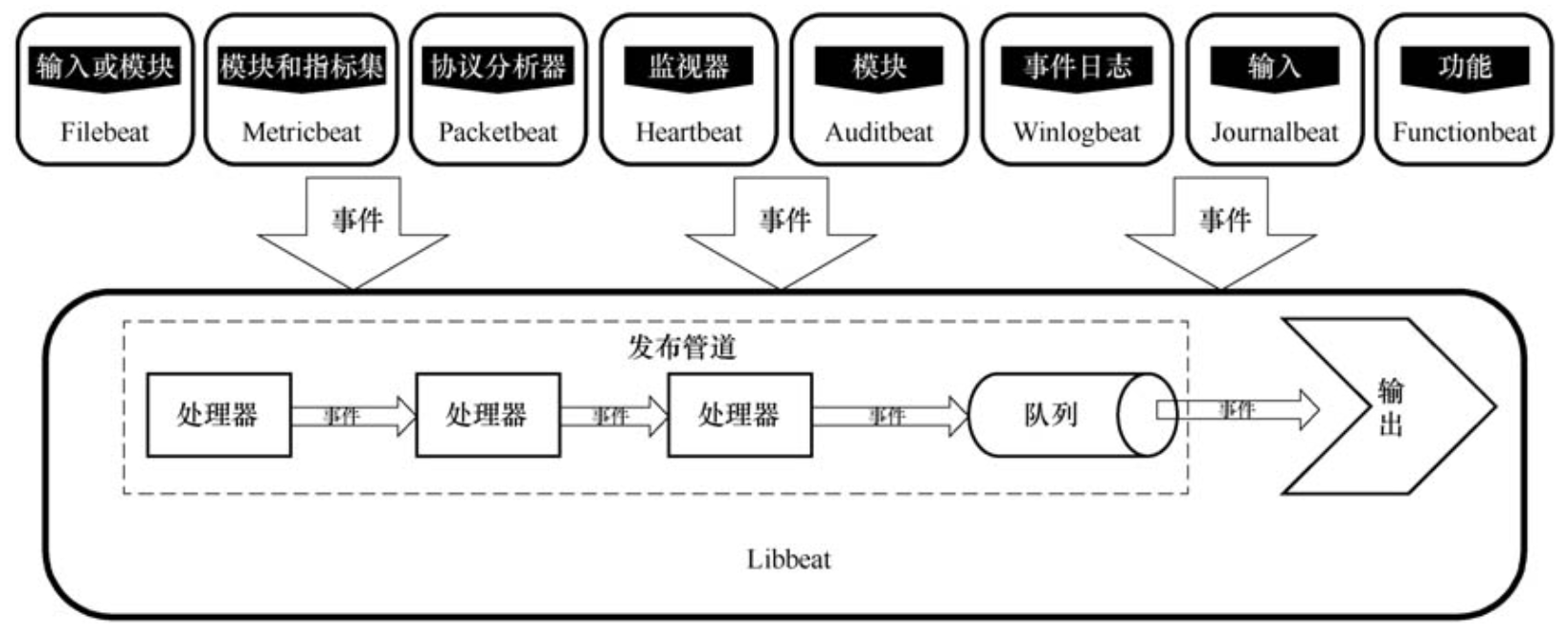
Beats处理器组件、输出组件等由于通过Libbeat实现，所以在所有Beats组件中基本上都是通用的。但Beats输入组件由具体Beats实例单独实现，所以它们之间存在着巨大差异，也是不同种类Beats之间的核心区别。本章就来介绍这些输入组件的使用方法。

Filebeat一般用于采集宿主机本地保存的文件，然后将它们的内容发送到指定的目的地，以实现对文件内容的收集、归档和分析等后期处理。Filebeat最典型的应用场景是收集日志文件，它能够在不侵入现有业务系统的情况下实现日志数据搬运，比如将日志数据搬运到Elasticsearch中以供检索分析使用。这对于采集第三方应用产生的日志来说尤为重要，因为多数第三方应用的日志只提供了本地文件日志。比如MySQL、Tomcat等第三方应用，它们一般只会在指定的目录中保存日志文件，而没有提供将日志写到其他地方的功能。所以如果想对它们的日志做统一分析和处理，就必须在日志文件所在主机安装代理程序做搬运。这种能够实现搬运功能的组件除了Filebeat以外还有很多，比如Flume、Scribe等。

Filebeat输入组件的配置相对其他Beats来说更为灵活，可以通过两种方式实现，它们对于学习后面的Beats组件配置也有一定的借鉴意义。第一种方式是通过输入类型指定输入组件，另一种则是通过开启模块指定输入组件。前者根据输入类型和用户自定义的配置收集数据，数据从哪里收集以及如何展示都需要用户自定义；而后者则预先定义了一组通用系统的日志收集配置，每一组配置就是一个模块并针对一种第三方应用。所以相对来说，输入类型在配置上更自由更灵活，而模块则更简便更快捷。下面就来看看这两种方式是如何配置和使用的。

尽管有多种类型的Beats组件，但它们都基于Libbeat框架开发，所以具有相同的体系结构。Libbeat是Beats组件的基础框架，包含了所有Beats组件共有特征。应该说，Beats组件的体系结构与Logstash有些相似，同样包含了与数据输入、过滤和输出相关的组件，只是它们在名称、结构以及实现上与Logstash存在着一些差异。具体来说，Beats整体上是由输入组件、发布管道和输出组件三部分组成。输入组件从不同数据源采集数据，采集进来的数据在Beats中也称为事件（Event）。Beats事件通常以JSON格式表示，为了与Elasticsearch索引字段相区别，Beats事件中的JSON属性将统一称为属性。由于不同数据源采用的数据采集方法肯定不一样，所以输入组件是在具体的Beats类型中实现，它们的名称往往也是不一样的。比如，在Filebeat中输入组件是输入（Input）和模块（Module），Metricbeat是模块（Module）和指标集（Metricset），而Packetbeat则是协议分析器（Protocol Analyzer）。

发布管道（Publisher Pipeline）由多个处理器（Processor）和一个队列（Queue）组成，它们会对输入组件采集进来的事件做加工处理，然后发送到队列中缓存。输出组件则负责将事件发送到最终的目标数据源，它会从队列中读取事件并将它们发送出去。发布管道和输出组件由Libbeat实现，它们在所有Beats类型中都是一样的。Beats组件的整体结构如图15-1所示。



## 共有组件

由于发布管道由处理器和队列组成，所以由Libbeat框架实现的Beats共有组件可以细化为处理器、队列和输出组件三种。其中，处理器负责对事件做加工处理，而队列则负责缓存事件以提升效率，输出组件则负责最终的事件发送。

### 处理器

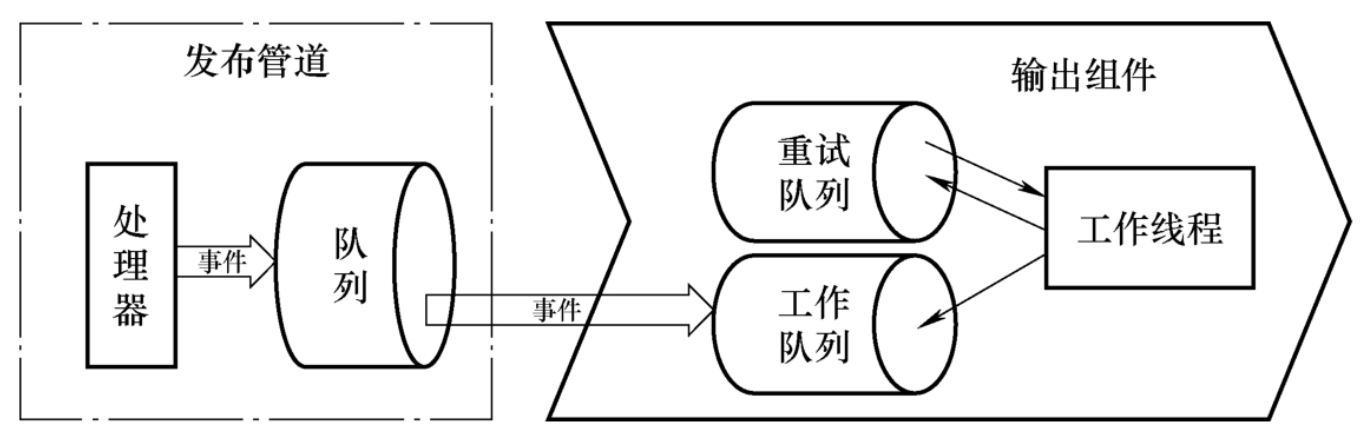
处理器与Logstash中的过滤器插件类似，负责对事件做额外的加工和处理，比如向事件中添加、删除或更新属性等。Beats发布管道可以定义多个处理器，它们最终会形成一个职责链并依次对事件做处理。

### 队列

为了提升Beats管道处理事件的性能，Beats组件会将读取到的事件缓存到队列中，队列可以是基于内存的也可以是基于文件的。默认情况下，Beats管道使用基于内存的队列缓存事件，并在队列缓存事件数量达到2048时将事件一次性传递给输出组件。如果在超过1s时事件数量仍然没有达到2048，队列也会将事件刷出到输出组件。也就是说，队列刷出事件有两个阈值，一个是事件数量另一个是超时时间。基于内存的队列使用queue.mem参数配置，包括一组子参数

### 输出组件

Beats输出组件一般由三部分组成，它们是工作队列、重试队列和一组工作线程。管道组件发布出来的事件会进入输出组件的工作队列，工作线程再从工作队列中取出事件做处理并发送。如果发送失败则事件将进入重试队列，并在一定时间间隔后再次由工作线程处理。输出组件也是所有Beats组件都具有的组件，由Libbeat实现。将这些共有组件整合起来，它们的结构和关系如图15-2所示。



### Beats输出组件

Beats输出组件由Libbeat框架实现，所以它们的类型、配置对所有Beats组件也都相同。由于同属Elastic家族，Elasticsearch（elasticsearch）和Logstash（logstash）是Beats输出的主要类型。除此之外还支持Kafka消息中间件（kafka）、Redis缓存（redis）、文件（file）、控制台（console）、Elastic云服务（cloud）等几种输出的类型。上述输出类型后面的括号中，是Beats配置这些输出类型时使用的关键字。此外，Functionbeat比较特殊，它仅支持Elasticsearch和Cloud两种输出组件。

## 资源管理

Filebeat采集数据的功能由两类组件构成，它们是输入组件（Input）和收集组件（Harvester）。输入组件负责管理一个或多个收集组件，它会根据配置找到所有需要收集的文件并创建专门的收集组件。每个收集组件对应一个要处理的文件，除了负责从文件中收集数据以外，它还要负责文件的打开与关闭以保证文件被合理使用和释放。Filebeat输入组件通过扫描文件感知文件变化，扫描的时间间隔由scan\_frequency参数设置，它的默认值为10s。在扫描过程中，输入组件除了要感知是否有新文件加入被监控路径以外，还要感知没有被收集的旧文件是否发生了变化。如果发现有新文件加入，或是旧文件发生了变化则会为它们创建新的收集组件以收集数据。在扫描文件变化时，Filebeat输入组件会按文件在操作系统中的顺序依次扫描，但可以通过scan.sort和scan.order两个参数自定义扫描文件的次序。scan.sort参数的可选值为modtime和filename，前者定义扫描时按文件的修改时间顺序扫描，后者则是按文件名称排序。scan.order参数则定义扫描排序是升序还是降序，所以可选值为asc和desc。

默认情况下，如果收集组件在5min内没有从文件中收集到数据，收集组件就会被关闭以释放资源，这是Filebeat释放收集组件和文件的最主要方式。如果文件在收集组件关闭后又发生了变化，输入组件会再次创建新的收集组件来收集数据。但由于输入组件默认情况下每隔10s才扫描一次文件变化，所以在最坏情况下Filebeat要在10s以后才能收集到变化的数据。所以为了保证收集文件数据的实时性，应尽量将关闭收集组件的时间间隔设置得高于文件更新的频率。比如日志文件更新频度如果是秒级别的，则应该通过close\_inactive参数将关闭文件的时间间隔设置为分钟级别，以保证日志在持续更新时不会关闭文件。

除了close\_inactive以外，log输入类型还提供了close\_removed、close\_renamed、close\_eof和close\_timeout几个参数用于控制对收集组件的关闭时机。close\_removed默认情况下开启，输入组件会在文件被删除时关闭对应的收集组件。close\_renamed和close\_eof默认情况下则是关闭的，前者会在文件重命名时关闭收集组件，而后者则在读取到文件结尾时关闭收集组件。close\_timeout参数比较特殊，它给每个收集组件定义了一个生存周期。超过生存周期时，不管收集组件是否已经收集结束都会被关闭。默认情况下close\_timeout值为0，也就是没有给收集组件定义生存周期。

## 读取状态

Filebeat通过注册文件（Registry File）保存收集组件读取每个文件的状态，具体来说就是收集组件读取文件的偏移量。注册文件保存的读取状态可以保证文件中所有的内容都可以被读取出来，这与Logstash中文件输入插件SinceDB的作用类似。默认情况下，注册文件会被保存到Filebeat数据路径下的registry目录中，而Filebeat数据路径一般位于Filebeat安装路径下的data目录。注册文件存储路径可以通过filebeat.registry.path参数来修改，它的默认值为${path.data}/registry。如果设置了新的注册文件目录并重启Filebeat，那么已经保存的读取状态将全部丢失。

当Filebeat因意外崩溃，重启后会通过注册文件获取文件的读取状态以保证不会重复读取数据。同时由于只有事件被发送后偏移量才会保存到注册文件中，所以Filebeat可以保证收集的数据至少被发送一次。比如当事件发送的目的地不可用，Filebeat除了会尝试重新发送事件以外，读取状态的偏移量也不会更新到注册文件中。显然，如果Filebeat发送成功而偏移量不能及时更新到注册文件时，有可能会导致数据被重复发送。在默认情况下，Filebeat会在发送成功后立即更新注册文件，更新的时间间隔可通过filebeat.registry.flush参数设置，默认值为0s。

如果Filebeat监控的文件非常多，那么注册文件要保存的条目也就非常多，这将导致注册文件变得越来越大。log输入类型提供了clean\_inactive参数，可以将超过该参数指定时限的注册条目删除。另一个参数clean\_removed参数，可以将已经删除文件的注册条目删除。这个参数与close\_removed参数相关联，如果close\_removed没有开启，则clean\_removed也必须要关闭。

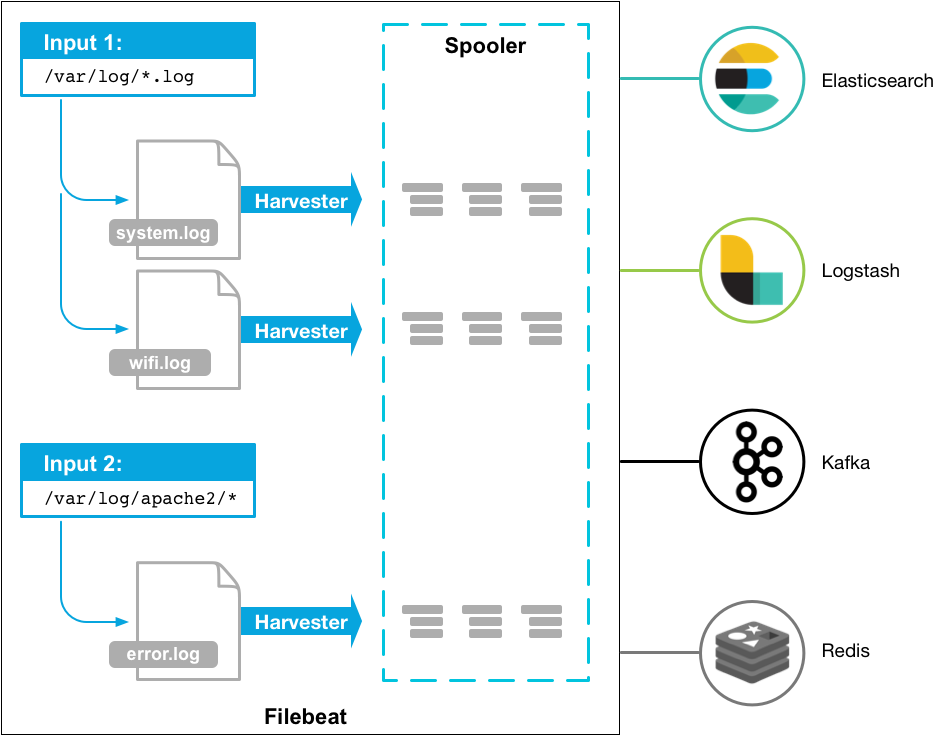
FileBeat是用于转发和集中日志数据的轻量级发送器。Filebeat作为代理安装在您的服务器上，它监视您指定的日志文件或位置，收集日志事件，并将它们转发到Elasticsearch或Logstash进行索引。

FileBeat的工作原理如下：当您启动FileBeat时，它会启动一个或多个在您为日志数据指定的位置查找的输入。对于Filebeat找到的每个日志，Filebeat都会启动一个收割器。每个收割器都读取新内容的单个日志，并将新的日志数据发送到libeat，后者聚合事件并将聚合的数据发送到您为Filebeat配置的输出。

## Filebeat工作原理：

Filebeat由两个主要组件组成：prospectors 和 harvesters。这两个组件协同工作将文件变动发送到指定的输出中。

Filebeat 主要由两个组件构成：prospector（探测器）和 harvester（收集器），这两类组件一起协作完成 Filebeat 的工作。



Harvester（收割机）：负责读取单个文件内容。每个文件会启动一个Harvester，每个Harvester会逐行读取各个文件，并将文件内容发送到制定输出中。Harvester负责打开和关闭文件，意味在Harvester运行的时候，文件描述符处于打开状态，如果文件在收集中被重命名或者被删除，Filebeat会继续读取此文件。所以在Harvester关闭之前，磁盘不会被释放。默认情况filebeat会保持文件打开的状态，直到达到close\_inactive（如果此选项开启，filebeat会在指定时间内将不再更新的文件句柄关闭，时间从harvester读取最后一行的时间开始计时。若文件句柄被关闭后，文件发生变化，则会启动一个新的harvester。关闭文件句柄的时间不取决于文件的修改时间，若此参数配置不当，则可能发生日志不实时的情况，由scan\_frequency参数决定，默认10s。Harvester使用内部时间戳来记录文件最后被收集的时间。例如：设置5m，则在Harvester读取文件的最后一行之后，开始倒计时5分钟，若5分钟内文件无变化，则关闭文件句柄。默认5m）。

Prospector（勘测者）：负责管理Harvester并找到所有读取源。

filebeat.prospectors:

- input\_type: log

paths:

- /apps/logs/\*/info.log

Prospector会找到/apps/logs/\*目录下的所有info.log文件，并为每个文件启动一个Harvester。Prospector会检查每个文件，看Harvester是否已经启动，是否需要启动，或者文件是否可以忽略。若Harvester关闭，只有在文件大小发生变化的时候Prospector才会执行检查。只能检测本地的文件。

## Filebeat如何记录文件状态：

将文件状态记录在文件中（默认在/var/lib/filebeat/registry）。此状态可以记住Harvester收集文件的偏移量。若连接不上输出设备，如ES等，filebeat会记录发送前的最后一行，并再可以连接的时候继续发送。Filebeat在运行的时候，Prospector状态会被记录在内存中。Filebeat重启的时候，利用registry记录的状态来进行重建，用来还原到重启之前的状态。每个Prospector会为每个找到的文件记录一个状态，对于每个文件，Filebeat存储唯一标识符以检测文件是否先前被收集。

## Filebeat如何保证事件至少被输出一次：

Filebeat之所以能保证事件至少被传递到配置的输出一次，没有数据丢失，是因为filebeat将每个事件的传递状态保存在文件中。在未得到输出方确认时，filebeat会尝试一直发送，直到得到回应。若filebeat在传输过程中被关闭，则不会再关闭之前确认所有时事件。任何在filebeat关闭之前为确认的时间，都会在filebeat重启之后重新发送。这可确保至少发送一次，但有可能会重复。可通过设置shutdown\_timeout 参数来设置关闭之前的等待事件回应的时间（默认禁用）。

4.1Filebeat 如何保持文件的状态？

Filebeat 保存每个文件的状态并经常将状态刷新到磁盘上的注册文件中。该状态用于记住 harvester 正在读取的最后偏移量，并确保发送所有日志行。如果输出（例如 Elasticsearch 或 Logstash）无法访问，Filebeat 会跟踪最后发送的行，并在输出再次可用时继续读取文件。

在 Filebeat 运行时，每个 prospector 内存中也会保存文件状态信息，当重新启动 Filebeat 时，将使用注册文件的数据来重建文件状态，Filebeat 将每个 harvester 在从保存的最后偏移量继续读取。

每个 prospector 为它找到的每个文件保留一个状态。由于文件可以被重命名或移动，因此文件名和路径不足以识别文件。对于每个文件，Filebeat 存储唯一标识符以检测文件是否先前已被采集过。

如果你使用的案例涉及每天创建大量新文件，你可能会发现注册文件增长过大。请参阅注册表文件太大？编辑有关你可以设置以解决此问题的配置选项的详细信息。

4.2Filebeat 如何确保至少一次交付？

Filebeat 保证事件至少会被传送到配置的输出一次，并且不会丢失数据。 Filebeat 能够实现此行为，因为它将每个事件的传递状态存储在注册文件中。

在输出阻塞或未确认所有事件的情况下，Filebeat 将继续尝试发送事件，直到接收端确认已收到。如果 Filebeat 在发送事件的过程中关闭，它不会等待输出确认所有收到事件。

发送到输出但在 Filebeat 关闭前未确认的任何事件在重新启动 Filebeat 时会再次发送。这可以确保每个事件至少发送一次，但最终会将重复事件发送到输出。

也可以通过设置 shutdown\_timeout 选项来配置 Filebeat 以在关闭之前等待特定时间。

注意：Filebeat 的至少一次交付保证包括日志轮换和删除旧文件的限制。如果将日志文件写入磁盘并且写入速度超过 Filebeat 可以处理的速度，或者在输出不可用时删除了文件，则可能会丢失数据。

在 Linux 上，Filebeat 也可能因 inode 重用而跳过行。有关 inode 重用问题的更多详细信息，请参阅 filebeat 常见问题解答。

4.3Filebeat 如何保证在日志文件被切割 (或滚动 rolling) 时依然正确读取文件？

Logback 日志切割用的是 JDK 里 File#renameTo() 方法。如果该方法失败，就再尝试使用复制数据的方式切割日志。查找该方法相关资料得知，只有当源文件和目标目录处于同一个文件系统、同 volumn(即 windows 下的 C, D 盘) 下该方法才会成功，切不会为重命名的后的文件分配新的 inode 值。也就是说，如果程序里一直保存着该文件的描述符，那么当程序再写日志时，就会向重命名后的文件中写。那么问题来了，filebeat 是会一直打开并保存文件描述符的，那么它是怎么得知日志被切割这件事的呢？

如果只用当前文件描述符一路监控到天黑的话，那么当 logback 把日志重命名后，filebeat 仍然会监控重命名后的日志，新创建的日志文件就看不到了。实际上，filebeat 是通过 close\_inactive 和 scan\_frequency 两个参数 (机制) 来应对这种情况的：

(1)close\_inactive

该参数指定当被监控的文件多长时间没有变化后就关闭文件句柄 (file handle)。官方建议将这个参数设置为一个比文件最大更新间隔大的值。比如文件最长 5s 更新一次，那就设置成 1min。默认值为 5min。

(2)scan\_frequency

该参数指定 Filebeat 搜索新文件的频率 (时间间隔)。当发现新的文件被创建时， Filebeat 会为它再启动一个 harvester 进行监控，默认为 10s。

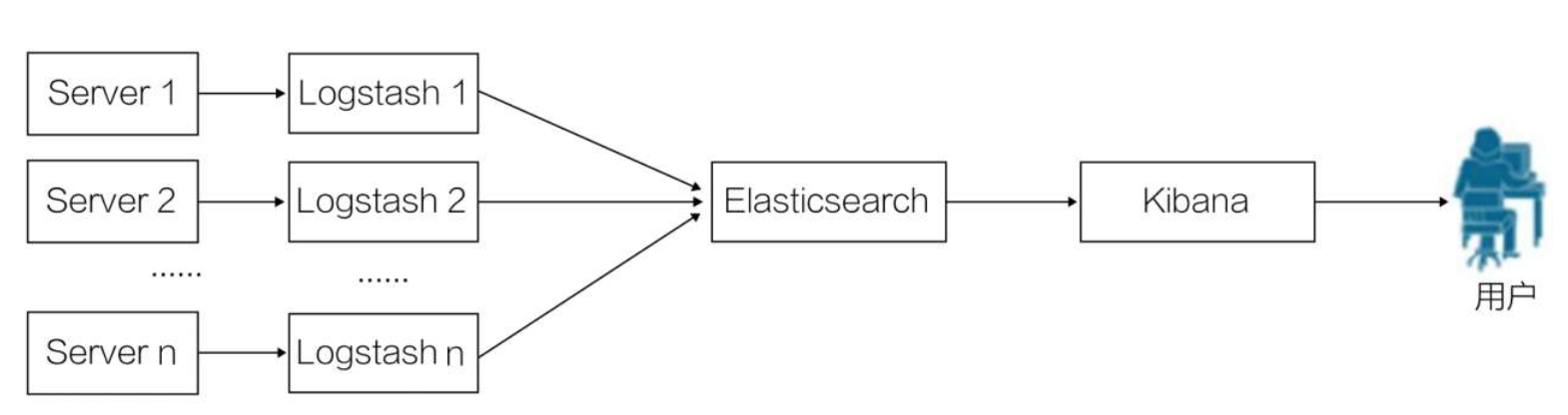
综合以上两个机制，当 logback 完成日志切割后 (即重命名)，此时老的 harvester 仍然在监控重命名后的日志文件，但是由于该文件不会再更新，因此会在 close\_inactive 时间后关闭这个文件的 harvester。当 scan\_frequency 时间过后，Filebeat 会发现目录中出现了新文件，于是为该文件启动 harvester 进行监控。这样就保证了切割日志时也能不丢不重的传输数据。(不重是通过为每个日志文件保存 offset 实现的)

# 日志场景应用

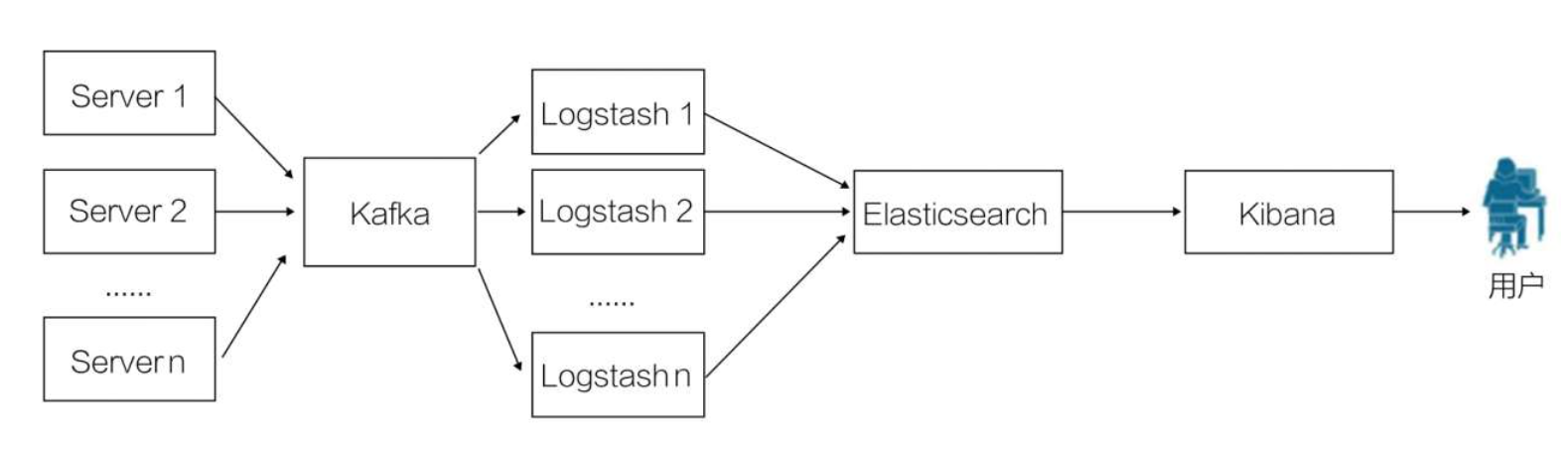
ELK的三个组件是如何分工协作的呢？首先，我们使用Logstash进行日志的搜集、分析和过滤。一般工作方式为C/S架构，Client端会被安装在需要收集日志的主机上，Server端则负责收集的各节点的日志数据，并进行过滤、修改和分析等操作，预处理过的数据会一并发到Elasticsearch上。随后将Kibana接入Elasticsearch，并为Logstash和Elasticsearch提供日志分析友好的Web界面，帮助用户汇总、分析和搜索重要数据的日志。

首先由分布于各个服务节点上的Logstash搜集相关日志和数据，经过Logstash的分析和过滤后发送给远端服务器上的Elasticsearch进行存储。Elasticsearch将数据以分片的形式压缩存储，并提供多种API供用户进行查询操作。用户还可以通过配置Kibana Web Portal对日志进行查询，并根据数据生成报表。

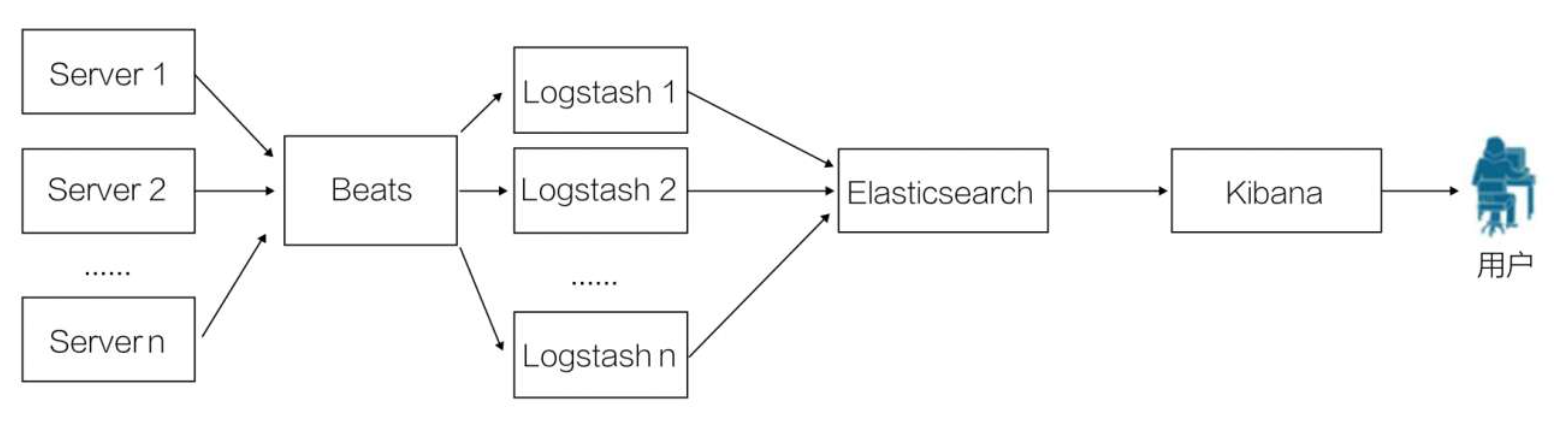
ELK架构为数据分布式存储、可视化查询和日志解析创建了一个功能强大的管理链。ELK架构为用户建立了集中式日志收集系统，将所有节点上的日志统一收集、管理和访问。三者相互配合，取长补短，共同完成分布式大数据处理工作。当前官方推荐的ELK部署架构并非一步到位，而是经过迭代演进发展而来的。下面简单介绍ELK架构的发展历程。最简单的一种ELK部署架构方式如图10-2所示。

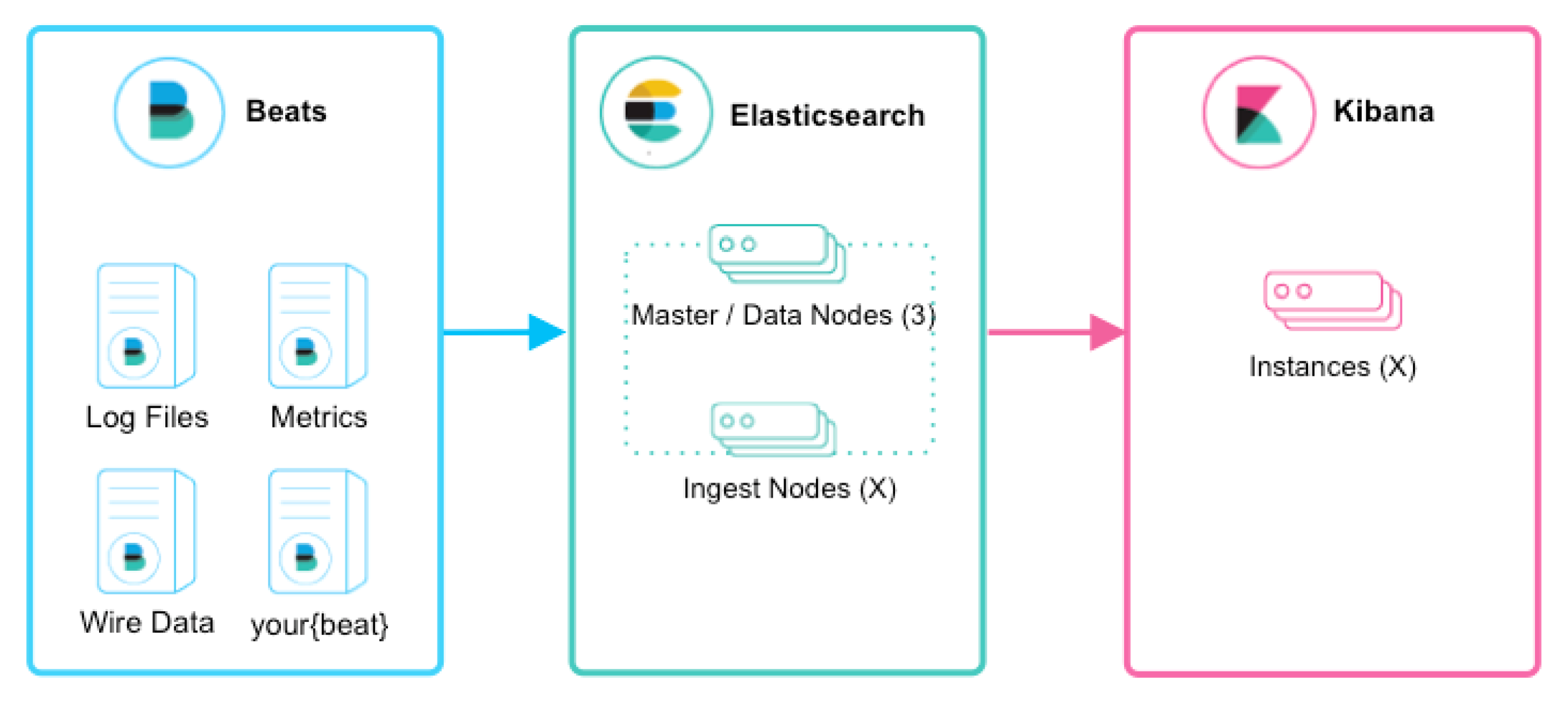


基于第一种ELK部署架构的优缺点，第二种架构引入了消息队列机制，如图10-3所示。

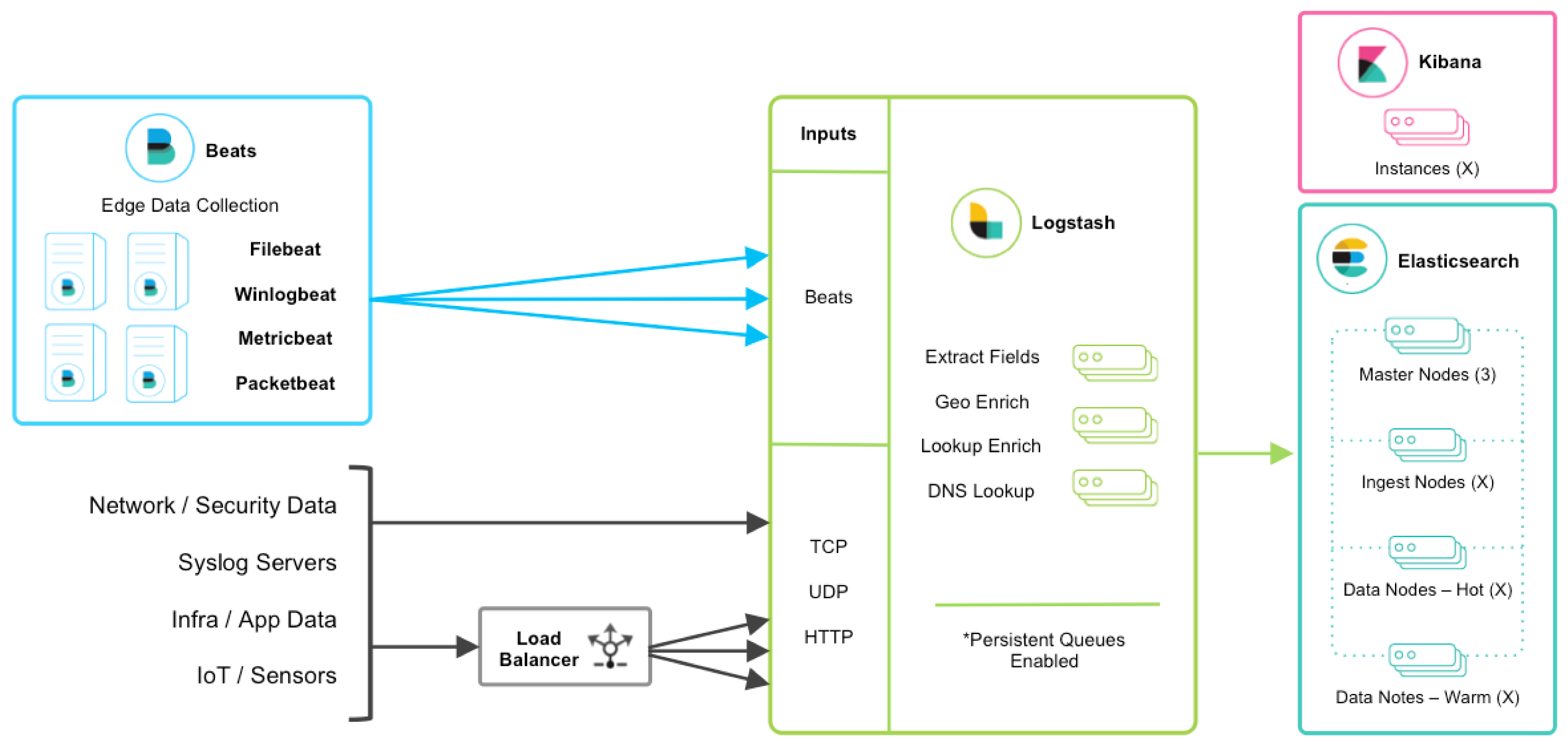
 [插图]图10-3位于各个节点上的Logstash客户端先将数据和日志等内容传递给Kafka，当然，也可以用其他消息机制，如各类MQ（Message Queue）和Redis等。Kafka会将队列中的消息和数据传递给Logstash，经过Logstash的过滤和分析等处理后，传递给Elasticsearch进行存储。最后由Kibana将日志和数据呈现给用户。在该部署架构中，Kafka的引入使得即使远端Logstash因故障而停止运行，数据也会被存储下来，从而避免数据丢失。

随着Beats组件引入ELK Stack，第四种部署架构应运而生，如图10-5所示。[插图]图10-5在实际使用中，Beats平台在满负荷状态时所耗系统资源和Logstash-forwarder相当，但其扩展性和灵活性更好。Beats平台目前包含Packagebeat、Topbeat和Filebeat三个产品，均为Apache 2.0 License。同时用户可以根据需要进行二次开发。与前面三个部署架构相比，显然第四种架构更灵活，可扩展性更强。









# 总结

Elasticsearch 技术分析（九）：全文搜索引擎Elasticsearch，这篇文章给讲透了！

之前已经分享过Elasticsearch的使用和原理的知识，由于近期在公司内部做了一次分享，所以本篇主要是基于之前的博文的一个总结，希望通过这篇文章能让读者大致了解Elasticsearch是做什么的以及它的使用和基本原理。

生活中的数据#

搜索引擎是对数据的检索，所以我们先从生活中的数据说起。

我们生活中的数据总体分为两种：结构化数据 和 非结构化数据。

结构化数据： 也称作行数据，是由二维表结构来逻辑表达和实现的数据，严格地遵循数据格式与长度规范，主要通过关系型数据库进行存储和管理。指具有固定格式或有限长度的数据，如数据库，元数据等。

非结构化数据： 又可称为全文数据，不定长或无固定格式，不适于由数据库二维表来表现，包括所有格式的办公文档、XML、HTML、word文档，邮件，各类报表、图片和咅频、视频信息等。

说明：如果要更细致的区分的话，XML、HTML可划分为 半结构化数据。因为它们也具有自己特定的标签格式，所以既可以根据需要按结构化数据来处理，也可抽取出纯文本按非结构化数据来处理。

根据两种数据分类，搜索也相应的分为两种：结构化数据搜索和非结构化数据搜索。

对于结构化数据，因为它们具有特定的结构，所以我们一般都是可以通过关系型数据库（mysql，oracle等）的 二维表（table）的方式存储和搜索，也可以建立索引。

对于非结构化数据，也即对全文数据的搜索主要有两种方法：顺序扫描法，全文检索。

顺序扫描：通过文字名称也可了解到它的大概搜索方式，即按照顺序扫描的方式查询特定的关键字。

例如给你一张报纸，让你找到该报纸中“平安”的文字在哪些地方出现过。你肯定需要从头到尾把报纸阅读扫描一遍然后标记出关键字在哪些版块出现过以及它的出现位置。

这种方式无疑是最耗时的最低效的，如果报纸排版字体小，而且版块较多甚至有多份报纸，等你扫描完你的眼睛也差不多了。

全文搜索：对非结构化数据顺序扫描很慢，我们是否可以进行优化？把我们的非结构化数据想办法弄得有一定结构不就行了吗？将非结构化数据中的一部分信息提取出来，重新组织，使其变得有一定结构，然后对此有一定结构的数据进行搜索，从而达到搜索相对较快的目的。

这种方式就构成了全文检索的基本思路。这部分从非结构化数据中提取出的然后重新组织的信息，我们称之索引。这种方式的主要工作量在前期索引的创建，但是对于后期搜索却是快速高效的。

先说说Lucene#

通过对生活中数据的类型作了一个简短了解之后，我们知道关系型数据库的SQL检索是处理不了这种非结构化数据的。这种非结构化数据的处理需要依赖全文搜索，而目前市场上开放源代码的最好全文检索引擎工具包就属于 apache 的 Lucene了。

但是 Lucene 只是一个工具包，它不是一个完整的全文检索引擎。Lucene的目的是为软件开发人员提供一个简单易用的工具包，以方便的在目标系统中实现全文检索的功能，或者是以此为基础建立起完整的全文检索引擎。

目前以 Lucene 为基础建立的开源可用全文搜索引擎主要是 Solr 和 Elasticsearch。

Solr 和 Elasticsearch 都是比较成熟的全文搜索引擎，能完成的功能和性能也基本一样。但是 ES 本身就具有分布式的特性和易安装使用的特点，而Solr的分布式需要借助第三方来实现，例如通过使用ZooKeeper来达到分布式协调管理。

不管是 Solr 还是 Elasticsearch 底层都是依赖于 Lucene，而 Lucene 能实现全文搜索主要是因为它实现了倒排索引的查询结构。

如何理解倒排索引呢？假如现有三份数据文档，文档的内容如下分别是：

Java is the best programming language.

PHP is the best programming language.

Javascript is the best programming language.

为了创建倒排索引，我们通过分词器将每个文档的内容域拆分成单独的词（我们称它为词条或 Term），创建一个包含所有不重复词条的排序列表，然后列出每个词条出现在哪个文档。结果如下所示：

Term Doc\_1 Doc\_2 Doc\_3

-------------------------------------

Java | X | |

is | X | X | X

the | X | X | X

best | X | X | X

programming | x | X | X

language | X | X | X

PHP | | X |

Javascript | | | X

-------------------------------------

这种结构由文档中所有不重复词的列表构成，对于其中每个词都有一个文档列表与之关联。这种由属性值来确定记录的位置的结构就是倒排索引。带有倒排索引的文件我们称为倒排文件。

我们将上面的内容转换为图的形式来说明倒排索引的结构信息，如下图所示，

其中主要有如下几个核心术语需要理解：

词条(Term)：索引里面最小的存储和查询单元，对于英文来说是一个单词，对于中文来说一般指分词后的一个词。

词典(Term Dictionary)：或字典，是词条Term的集合。搜索引擎的通常索引单位是单词，单词词典是由文档集合中出现过的所有单词构成的字符串集合，单词词典内每条索引项记载单词本身的一些信息以及指向“倒排列表”的指针。

倒排表(Post list)：一个文档通常由多个词组成，倒排表记录的是某个词在哪些文档里出现过以及出现的位置。每条记录称为一个倒排项(Posting)。倒排表记录的不单是文档编号，还存储了词频等信息。

倒排文件(Inverted File)：所有单词的倒排列表往往顺序地存储在磁盘的某个文件里，这个文件被称之为倒排文件，倒排文件是存储倒排索引的物理文件。

从上图我们可以了解到倒排索引主要由两个部分组成：词典和倒排文件。词典和倒排表是Lucene中很重要的两种数据结构，是实现快速检索的重要基石。词典和倒排文件是分两部分存储的，词典在内存中而倒排文件存储在磁盘上。

核心概念#

一些基础知识的铺垫之后我们正式进入今天的主角Elasticsearch的介绍， ES是使用Java编写的一种开源搜索引擎，它在内部使用Lucene做索引与搜索，通过对Lucene的封装，隐藏了Lucene的复杂性，取而代之的提供一套简单一致的 RESTful API。

然而，Elasticsearch 不仅仅是 Lucene，并且也不仅仅只是一个全文搜索引擎。 它可以被下面这样准确的形容：

一个分布式的实时文档存储，每个字段可以被索引与搜索。

一个分布式实时分析搜索引擎。

能胜任上百个服务节点的扩展，并支持 PB 级别的结构化或者非结构化数据。

官网对Elasticsearch的介绍是Elasticsearch 是一个分布式、可扩展、近实时的搜索与数据分析引擎。 我们通过一些核心概念来看下Elasticsearch 是如何做到分布式，可扩展和近实时搜索的。

集群（Cluster）#

ES的集群搭建很简单，不需要依赖第三方协调管理组件，自身内部就实现了集群的管理功能。ES集群由一个或多个Elasticsearch节点组成，每个节点配置相同的 cluster.name 即可加入集群，默认值为 “elasticsearch”。确保不同的环境中使用不同的集群名称，否则最终会导致节点加入错误的集群。

一个Elasticsearch服务启动实例就是一个节点（Node）。节点通过node.name来设置节点名称，如果不设置则在启动时给节点分配一个随机通用唯一标识符作为名称。

发现机制#

那么有一个问题，ES内部是如何通过一个相同的设置cluster.name 就能将不同的节点连接到同一个集群的？答案是Zen Discovery。

Zen Discovery是Elasticsearch的内置默认发现模块（发现模块的职责是发现集群中的节点以及选举master节点）。它提供单播和基于文件的发现，并且可以扩展为通过插件支持云环境和其他形式的发现。Zen Discovery 与其他模块集成，例如，节点之间的所有通信都使用Transport模块完成。节点使用发现机制通过Ping的方式查找其他节点。

Elasticsearch 默认被配置为使用单播发现，以防止节点无意中加入集群。只有在同一台机器上运行的节点才会自动组成集群。

如果集群的节点运行在不同的机器上，使用单播，你可以为 Elasticsearch 提供一些它应该去尝试连接的节点列表。 当一个节点联系到单播列表中的成员时，它就会得到整个集群所有节点的状态，然后它会联系 master 节点，并加入集群。

这意味着单播列表不需要包含集群中的所有节点， 它只是需要足够的节点，当一个新节点联系上其中一个并且说上话就可以了。如果你使用 master 候选节点作为单播列表，你只要列出三个就可以了。 这个配置在 elasticsearch.yml 文件中：

discovery.zen.ping.unicast.hosts: ["host1", "host2:port"]

节点启动后先 ping ，如果discovery.zen.ping.unicast.hosts 有设置，则 ping 设置中的 host ，否则尝试 ping localhost 的几个端口， Elasticsearch 支持同一个主机启动多个节点， Ping 的 response 会包含该节点的基本信息以及该节点认为的 master 节点。 选举开始，先从各节点认为的 master 中选，规则很简单，按照 id 的字典序排序，取第一个。 如果各节点都没有认为的 master ，则从所有节点中选择，规则同上。

这里有个限制条件就是 discovery.zen.minimum\_master\_nodes ，如果节点数达不到最小值的限制，则循环上述过程，直到节点数足够可以开始选举。 最后选举结果是肯定能选举出一个 master ，如果只有一个 local 节点那就选出的是自己。 如果当前节点是 master ，则开始等待节点数达到 discovery.zen.minimum\_master\_nodes，然后提供服务。 如果当前节点不是 master ，则尝试加入 master 。 Elasticsearch 将以上服务发现以及选主的流程叫做 ZenDiscovery 。

由于它支持任意数目的集群（ 1- N ），所以不能像 Zookeeper 那样限制节点必须是奇数，也就无法用投票的机制来选主，而是通过一个规则，只要所有的节点都遵循同样的规则，得到的信息都是对等的，选出来的主节点肯定是一致的。但分布式系统的问题就出在信息不对等的情况，这时候很容易出现脑裂（ Split-Brain ）的问题，大多数解决方案就是设置一个 quorum 值，要求可用节点必须大于 quorum （一般是超过半数节点），才能对外提供服务。而 Elasticsearch 中，这个 quorum 的配置就是 discovery.zen.minimum\_master\_nodes 。

节点的角色#

每个节点既可以是候选主节点也可以是数据节点，通过在配置文件../config/elasticsearch.yml中设置即可，默认都为true。

node.master: true //是否候选主节点

node.data: true //是否数据节点

数据节点负责数据的存储和相关的操作，例如对数据进行增、删、改、查和聚合等操作，所以数据节点（data节点）对机器配置要求比较高，对CPU、内存和I/O的消耗很大。通常随着集群的扩大，需要增加更多的数据节点来提高性能和可用性。

候选主节点可以被选举为主节点（master节点），集群中只有候选主节点才有选举权和被选举权，其他节点不参与选举的工作。主节点负责创建索引、删除索引、跟踪哪些节点是群集的一部分，并决定哪些分片分配给相关的节点、追踪集群中节点的状态等，稳定的主节点对集群的健康是非常重要的。

一个节点既可以是候选主节点也可以是数据节点，但是由于数据节点对CPU、内存核I/0消耗都很大，所以如果某个节点既是数据节点又是主节点，那么可能会对主节点产生影响从而对整个集群的状态产生影响。

因此为了提高集群的健康性，我们应该对Elasticsearch集群中的节点做好角色上的划分和隔离。可以使用几个配置较低的机器群作为候选主节点群。

主节点和其他节点之间通过Ping的方式互检查，主节点负责Ping所有其他节点，判断是否有节点已经挂掉。其他节点也通过Ping的方式判断主节点是否处于可用状态。

虽然对节点做了角色区分，但是用户的请求可以发往任何一个节点，并由该节点负责分发请求、收集结果等操作，而不需要主节点转发，这种节点可称之为协调节点，协调节点是不需要指定和配置的，集群中的任何节点都可以充当协调节点的角色。

脑裂现象#

同时如果由于网络或其他原因导致集群中选举出多个Master节点，使得数据更新时出现不一致，这种现象称之为脑裂，即集群中不同的节点对于master的选择出现了分歧，出现了多个master竞争。

“脑裂”问题可能有以下几个原因造成：

网络问题：集群间的网络延迟导致一些节点访问不到master，认为master挂掉了从而选举出新的master，并对master上的分片和副本标红，分配新的主分片

节点负载：主节点的角色既为master又为data，访问量较大时可能会导致ES停止响应（假死状态）造成大面积延迟，此时其他节点得不到主节点的响应认为主节点挂掉了，会重新选取主节点。

内存回收：主节点的角色既为master又为data，当data节点上的ES进程占用的内存较大，引发JVM的大规模内存回收，造成ES进程失去响应。

为了避免脑裂现象的发生，我们可以从原因着手通过以下几个方面来做出优化措施：

适当调大响应时间，减少误判

通过参数discovery.zen.ping\_timeout设置节点状态的响应时间，默认为3s，可以适当调大，如果master在该响应时间的范围内没有做出响应应答，判断该节点已经挂掉了。调大参数（如6s，discovery.zen.ping\_timeout:6），可适当减少误判。

选举触发

我们需要在候选集群中的节点的配置文件中设置参数discovery.zen.munimum\_master\_nodes的值，这个参数表示在选举主节点时需要参与选举的候选主节点的节点数，默认值是1，官方建议取值(master\_eligibel\_nodes/2) + 1，其中master\_eligibel\_nodes为候选主节点的个数。这样做既能防止脑裂现象的发生，也能最大限度地提升集群的高可用性，因为只要不少于discovery.zen.munimum\_master\_nodes个候选节点存活，选举工作就能正常进行。当小于这个值的时候，无法触发选举行为，集群无法使用，不会造成分片混乱的情况。

角色分离

即是上面我们提到的候选主节点和数据节点进行角色分离，这样可以减轻主节点的负担，防止主节点的假死状态发生，减少对主节点“已死”的误判。

分片（Shards）#

ES支持PB级全文搜索，当索引上的数据量太大的时候，ES通过水平拆分的方式将一个索引上的数据拆分出来分配到不同的数据块上，拆分出来的数据库块称之为一个分片。

这类似于MySql的分库分表，只不过Mysql分库分表需要借助第三方组件而ES内部自身实现了此功能。

在一个多分片的索引中写入数据时，通过路由来确定具体写入哪一个分片中，所以在创建索引的时候需要指定分片的数量，并且分片的数量一旦确定就不能修改。

分片的数量和下面介绍的副本数量都是可以通过创建索引时的settings来配置，ES默认为一个索引创建5个主分片, 并分别为每个分片创建一个副本。

PUT /myIndex

{

"settings" : {

"number\_of\_shards" : 5,

"number\_of\_replicas" : 1

}

}

ES通过分片的功能使得索引在规模上和性能上都得到提升，每个分片都是Lucene中的一个索引文件，每个分片必须有一个主分片和零到多个副本。

副本（Replicas）#

副本就是对分片的Copy，每个主分片都有一个或多个副本分片，当主分片异常时，副本可以提供数据的查询等操作。主分片和对应的副本分片是不会在同一个节点上的，所以副本分片数的最大值是 n -1（其中n为节点数）。

对文档的新建、索引和删除请求都是写操作，必须在主分片上面完成之后才能被复制到相关的副本分片，ES为了提高写入的能力这个过程是并发写的，同时为了解决并发写的过程中数据冲突的问题，ES通过乐观锁的方式控制，每个文档都有一个 \_version （版本）号，当文档被修改时版本号递增。一旦所有的副本分片都报告写成功才会向协调节点报告成功，协调节点向客户端报告成功。

从上图可以看出为了达到高可用，Master节点会避免将主分片和副本分片放在同一个节点上。

假设这时节点Node1服务宕机了或者网络不可用了，那么主节点上主分片S0也就不可用了。幸运的是还存在另外两个节点能正常工作，这时ES会重新选举新的主节点，而且这两个节点上存在我们的所需要的S0的所有数据，我们会将S0的副本分片提升为主分片，这个提升主分片的过程是瞬间发生的。此时集群的状态将会为 yellow。

为什么我们集群状态是 yellow 而不是 green 呢？ 虽然我们拥有所有的2个主分片，但是同时设置了每个主分片需要对应两份副本分片，而此时只存在一份副本分片。 所以集群不能为 green 的状态。如果我们同样关闭了 Node2 ，我们的程序依然可以保持在不丢任何数据的情况下运行，因为Node3 为每一个分片都保留着一份副本。

如果我们重新启动Node1 ，集群可以将缺失的副本分片再次进行分配，那么集群的状态又将恢复到原来的正常状态。 如果Node1依然拥有着之前的分片，它将尝试去重用它们，只不过这时Node1节点上的分片不再是主分片而是副本分片了，如果期间有更改的数据只需要从主分片上复制修改的数据文件即可。

小结:

1、将数据分片是为了提高可处理数据的容量和易于进行水平扩展，为分片做副本是为了提高集群的稳定性和提高并发量。

2、副本是乘法，越多消耗越大，但也越保险。分片是除法，分片越多，单分片数据就越少也越分散。

3、副本越多，集群的可用性就越高，但是由于每个分片都相当于一个Lucene的索引文件，会占用一定的文件句柄、内存及CPU，并且分片间的数据同步也会占用一定的网络带宽，所以索引的分片数和副本数也不是越多越好。

映射（Mapping）#

映射是用于定义ES对索引中字段的存储类型、分词方式和是否存储等信息，就像数据库中的 schema ，描述了文档可能具有的字段或属性、每个字段的数据类型。只不过关系型数据库建表时必须指定字段类型，而ES对于字段类型可以不指定然后动态对字段类型猜测，也可以在创建索引时具体指定字段的类型。

对字段类型根据数据格式自动识别的映射称之为动态映射（Dynamic mapping），我们创建索引时具体定义字段类型的映射称之为静态映射或显示映射（Explicit mapping）。

在讲解动态映射和静态映射的使用前，我们先来了解下ES中的数据有哪些字段类型？之后我们再讲解为什么我们创建索引时需要建立静态映射而不使用动态映射。

ES（v6.8）中字段数据类型主要有以下几类：

类别 数据类型

核心类型 text, keywords, long, integer, short, double, data, boolean等等

复杂类型 Object, Nested

地理类型 geo\_point, geo\_shape

特殊类型 ip, completion, token\_count, join等等

....... ...

text 用于索引全文值的字段，例如电子邮件正文或产品说明。这些字段是被分词的，它们通过分词器传递 ，以在被索引之前将字符串转换为单个术语的列表。分析过程允许Elasticsearch搜索单个单词中每个完整的文本字段。文本字段不用于排序，很少用于聚合。

keyword 用于索引结构化内容的字段，例如电子邮件地址，主机名，状态代码，邮政编码或标签。它们通常用于过滤，排序，和聚合。keyword字段只能按其确切值进行搜索。

通过对字段类型的了解我们知道有些字段需要明确定义的，例如某个字段是text类型还是keword类型差别是很大的，时间字段也许我们需要指定它的时间格式，还有一些字段我们需要指定特定的分词器等等。如果采用动态映射是不能精确做到这些的，自动识别常常会与我们期望的有些差异。

所以创建索引给的时候一个完整的格式应该是指定分片和副本数以及Mapping的定义，如下：

PUT my\_index

{

"settings" : {

"number\_of\_shards" : 5,

"number\_of\_replicas" : 1

}

"mappings": {

"\_doc": {

"properties": {

"title": { "type": "text" },

"name": { "type": "text" },

"age": { "type": "integer" },

"created": {

"type": "date",

"format": "strict\_date\_optional\_time||epoch\_millis"

}

}

}

}

}

基本使用#

在决定使用 Elasticsearch 的时候首先要考虑的是版本问题，Elasticsearch （排除 0.x 和 1.x）目前有如下常用的稳定的主版本：2.x，5.x，6.x，7.x（current）。你可能会发现没有 3.x 和 4.x，ES 从 2.4.6 直接跳到了 5.0.0。

其实是为了ELK（ElasticSearch, logstash, kibana）技术栈的版本统一，免的给用户带来混乱。在 Elasticsearch 是 2.x （2.x 的最后一版 2.4.6 的发布时间是 July 25, 2017） 的情况下，kibana 已经是 4.x（Kibana 4.6.5 的发布时间是 July 25, 2017），那么在 kibana 的下一主版本肯定是 5.x 了，所以 Elasticsearch 直接将自己的主版本发布为 5.0.0 了。统一之后，我们选版本就不会犹豫困惑了，我们选定 elasticsearch 的版本后再选择相同版本的 kibana 就行了，不用担忧版本不兼容的问题。

Elasticsearch是使用Java构建，所以除了注意 ELK 技术的版本统一，我们在选择 Elasticsearch 的版本的时候还需要注意 JDK的版本。因为每个大版本所依赖的 JDK版本也不同，目前7.2版本已经可以支持 jdk11。

安装使用#

1、下载和解压Elasticsearch，无需安装解压后即可用，解压后目录如下。

bin：二进制系统指令目录，包含启动命令和安装插件命令等。

config： 配置文件目录。

data： 数据存储目录。

lib：依赖包目录。

logs：日志文件目录。

modules：模块库，例如x-pack的模块。

plugins：插件目录。

2、安装目录下运行 bin/elasticsearch来启动 ES。

3、默认在9200端口运行，请求curl http://localhost:9200/ 或者浏览器输入http://localhost:9200，得到一个 JSON 对象，其中包含当前节点、集群、版本等信息。

{

"name" : "U7fp3O9",

"cluster\_name" : "elasticsearch",

"cluster\_uuid" : "-Rj8jGQvRIelGd9ckicUOA",

"version" : {

"number" : "6.8.1",

"build\_flavor" : "default",

"build\_type" : "zip",

"build\_hash" : "1fad4e1",

"build\_date" : "2019-06-18T13:16:52.517138Z",

"build\_snapshot" : false,

"lucene\_version" : "7.7.0",

"minimum\_wire\_compatibility\_version" : "5.6.0",

"minimum\_index\_compatibility\_version" : "5.0.0"

},

"tagline" : "You Know, for Search"

}

集群健康状态#

要检查群集运行状况，我们可以在 Kibana 控制台中运行以下命令 GET /\_cluster/health，得到如下信息：

{

"cluster\_name" : "wujiajian",

"status" : "yellow",

"timed\_out" : false,

"number\_of\_nodes" : 1,

"number\_of\_data\_nodes" : 1,

"active\_primary\_shards" : 9,

"active\_shards" : 9,

"relocating\_shards" : 0,

"initializing\_shards" : 0,

"unassigned\_shards" : 5,

"delayed\_unassigned\_shards" : 0,

"number\_of\_pending\_tasks" : 0,

"number\_of\_in\_flight\_fetch" : 0,

"task\_max\_waiting\_in\_queue\_millis" : 0,

"active\_shards\_percent\_as\_number" : 64.28571428571429

}

集群状态通过 绿，黄，红 来标识

绿色：集群健康完好，一切功能齐全正常，所有分片和副本都可以正常工作。

黄色：预警状态，所有主分片功能正常，但至少有一个副本是不能正常工作的。此时集群是可以正常工作的，但是高可用性在某种程度上会受影响。

红色：集群不可正常使用。某个或某些分片及其副本异常不可用，这时集群的查询操作还能执行，但是返回的结果会不准确。对于分配到这个分片的写入请求将会报错，最终会导致数据的丢失。

当集群状态为红色时，它将会继续从可用的分片提供搜索请求服务，但是你需要尽快修复那些未分配的分片。

机制原理#

ES的基本概念和基本操作介绍完了之后我们可能还有很多疑惑，它们内部是如何运行的？主分片和副本分片是如何同步的？创建索引的流程是什么样的？ES如何将索引数据分配到不同的分片上的？以及这些索引数据是如何存储的？为什么说ES是近实时搜索引擎而文档的 CRUD (创建-读取-更新-删除) 操作是实时的？以及Elasticsearch 是怎样保证更新被持久化在断电时也不丢失数据？还有为什么删除文档不会立刻释放空间？带着这些疑问我们进入接下来的内容。

写索引原理#

下图描述了3个节点的集群，共拥有12个分片，其中有4个主分片（S0、S1、S2、S3）和8个副本分片（R0、R1、R2、R3），每个主分片对应两个副本分片，节点1是主节点（Master节点）负责整个集群的状态。

写索引是只能写在主分片上，然后同步到副本分片。这里有四个主分片，一条数据ES是根据什么规则写到特定分片上的呢？

这条索引数据为什么被写到S0上而不写到S1或S2上？那条数据为什么又被写到S3上而不写到S0上了？

首先这肯定不会是随机的，否则将来要获取文档的时候我们就不知道从何处寻找了。实际上，这个过程是根据下面这个公式决定的：

shard = hash(routing) % number\_of\_primary\_shards

routing 是一个可变值，默认是文档的 \_id ，也可以设置成一个自定义的值。 routing 通过 hash 函数生成一个数字，然后这个数字再除以 number\_of\_primary\_shards （主分片的数量）后得到余数 。这个在 0 到 number\_of\_primary\_shards-1 之间的余数，就是我们所寻求的文档所在分片的位置。

这就解释了为什么我们要在创建索引的时候就确定好主分片的数量并且永远不会改变这个数量：因为如果数量变化了，那么所有之前路由的值都会无效，文档也再也找不到了。

由于在ES集群中每个节点通过上面的计算公式都知道集群中的文档的存放位置，所以每个节点都有处理读写请求的能力。在一个写请求被发送到某个节点后，该节点即为前面说过的协调节点，协调节点会根据路由公式计算出需要写到哪个分片上，再将请求转发到该分片的主分片节点上。

假如此时数据通过路由计算公式取余后得到的值是 shard = hash(routing) % 4 = 0，则具体流程如下：

客户端向ES1节点（协调节点）发送写请求，通过路由计算公式得到值为0，则当前数据应被写到主分片S0上。

ES1节点将请求转发到S0主分片所在的节点ES3，ES3接受请求并写入到磁盘。

并发将数据复制到两个副本分片R0上，其中通过乐观并发控制数据的冲突。一旦所有的副本分片都报告成功，则节点ES3将向协调节点报告成功，协调节点向客户端报告成功。

存储原理#

上面介绍了在ES内部索引的写处理流程，这个流程是在ES的内存中执行的，数据被分配到特定的分片和副本上之后，最终是存储到磁盘上的，这样在断电的时候就不会丢失数据。具体的存储路径可在配置文件../config/elasticsearch.yml中进行设置，默认存储在安装目录的data文件夹下。建议不要使用默认值，因为若ES进行了升级，则有可能导致数据全部丢失。

path.data: /path/to/data //索引数据

path.logs: /path/to/logs //日志记录

分段存储#

索引文档以段的形式存储在磁盘上，何为段？索引文件被拆分为多个子文件，则每个子文件叫作段， 每一个段本身都是一个倒排索引，并且段具有不变性，一旦索引的数据被写入硬盘，就不可再修改。在底层采用了分段的存储模式，使它在读写时几乎完全避免了锁的出现，大大提升了读写性能。

段被写入到磁盘后会生成一个提交点，提交点是一个用来记录所有提交后段信息的文件。一个段一旦拥有了提交点，就说明这个段只有读的权限，失去了写的权限。相反，当段在内存中时，就只有写的权限，而不具备读数据的权限，意味着不能被检索。

段的概念提出主要是因为：在早期全文检索中为整个文档集合建立了一个很大的倒排索引，并将其写入磁盘中。如果索引有更新，就需要重新全量创建一个索引来替换原来的索引。这种方式在数据量很大时效率很低，并且由于创建一次索引的成本很高，所以对数据的更新不能过于频繁，也就不能保证时效性。

索引文件分段存储并且不可修改，那么新增、更新和删除如何处理呢？

新增，新增很好处理，由于数据是新的，所以只需要对当前文档新增一个段就可以了。

删除，由于不可修改，所以对于删除操作，不会把文档从旧的段中移除而是通过新增一个.del文件，文件中会列出这些被删除文档的段信息。这个被标记删除的文档仍然可以被查询匹配到， 但它会在最终结果被返回前从结果集中移除。

更新，不能修改旧的段来进行反映文档的更新，其实更新相当于是删除和新增这两个动作组成。会将旧的文档在.del文件中标记删除，然后文档的新版本被索引到一个新的段中。可能两个版本的文档都会被一个查询匹配到，但被删除的那个旧版本文档在结果集返回前就会被移除。

段被设定为不可修改具有一定的优势也有一定的缺点，优势主要表现在：

不需要锁。如果你从来不更新索引，你就不需要担心多进程同时修改数据的问题。

一旦索引被读入内核的文件系统缓存，便会留在哪里，由于其不变性。只要文件系统缓存中还有足够的空间，那么大部分读请求会直接请求内存，而不会命中磁盘。这提供了很大的性能提升。

其它缓存(像filter缓存)，在索引的生命周期内始终有效。它们不需要在每次数据改变时被重建，因为数据不会变化。

写入单个大的倒排索引允许数据被压缩，减少磁盘 I/O 和 需要被缓存到内存的索引的使用量。

段的不变性的缺点如下：

当对旧数据进行删除时，旧数据不会马上被删除，而是在.del文件中被标记为删除。而旧数据只能等到段更新时才能被移除，这样会造成大量的空间浪费。

若有一条数据频繁的更新，每次更新都是新增新的标记旧的，则会有大量的空间浪费。

每次新增数据时都需要新增一个段来存储数据。当段的数量太多时，对服务器的资源例如文件句柄的消耗会非常大。

在查询的结果中包含所有的结果集，需要排除被标记删除的旧数据，这增加了查询的负担。

延迟写策略#

介绍完了存储的形式，那么索引是写入到磁盘的过程是这怎样的？是否是直接调 fsync 物理性地写入磁盘？

答案是显而易见的，如果是直接写入到磁盘上，磁盘的I/O消耗上会严重影响性能，那么当写数据量大的时候会造成ES停顿卡死，查询也无法做到快速响应。如果真是这样ES也就不会称之为近实时全文搜索引擎了。

为了提升写的性能，ES并没有每新增一条数据就增加一个段到磁盘上，而是采用延迟写的策略。

每当有新增的数据时，就将其先写入到内存中，在内存和磁盘之间是文件系统缓存，当达到默认的时间（1秒钟）或者内存的数据达到一定量时，会触发一次刷新（Refresh），将内存中的数据生成到一个新的段上并缓存到文件缓存系统 上，稍后再被刷新到磁盘中并生成提交点。

这里的内存使用的是ES的JVM内存，而文件缓存系统使用的是操作系统的内存。新的数据会继续的被写入内存，但内存中的数据并不是以段的形式存储的，因此不能提供检索功能。由内存刷新到文件缓存系统的时候会生成了新的段，并将段打开以供搜索使用，而不需要等到被刷新到磁盘。

在 Elasticsearch 中，写入和打开一个新段的轻量的过程叫做 refresh （即内存刷新到文件缓存系统）。 默认情况下每个分片会每秒自动刷新一次。这就是为什么我们说 Elasticsearch 是近实时搜索，因为文档的变化并不是立即对搜索可见，但会在一秒之内变为可见。我们也可以手动触发 refresh，POST /\_refresh 刷新所有索引，POST /nba/\_refresh刷新指定的索引。

Tips： 尽管刷新是比提交轻量很多的操作，它还是会有性能开销。 当写测试的时候， 手动刷新很有用，但是不要在生产> 环境下每次索引一个文档都去手动刷新。而且并不是所有的情况都需要每秒刷新。可能你正在使用 Elasticsearch 索引大量的日志文件， 你可能想优化索引速度而不是> 近实时搜索， 这时可以在创建索引时在settings中通过调大refresh\_interval = "30s" 的值 ， 降低每个索引的刷新频率，设值时需要注意后面带上时间单位，否则默认是毫秒。当refresh\_interval = -1时表示关闭索引的自动刷新。

虽然通过延时写的策略可以减少数据往磁盘上写的次数提升了整体的写入能力，但是我们知道文件缓存系统也是内存空间，属于操作系统的内存，只要是内存都存在断电或异常情况下丢失数据的危险。

为了避免丢失数据，Elasticsearch添加了事务日志（Translog），事务日志记录了所有还没有持久化到磁盘的数据。添加了事务日志后整个写索引的流程如下图所示。

一个新文档被索引之后，先被写入到内存中，但是为了防止数据的丢失，会追加一份数据到事务日志中。不断有新的文档被写入到内存，同时也都会记录到事务日志中。这时新数据还不能被检索和查询。

当达到默认的刷新时间或内存中的数据达到一定量后，会触发一次 refresh，将内存中的数据以一个新段形式刷新到文件缓存系统中并清空内存。这时虽然新段未被提交到磁盘，但是可以提供文档的检索功能且不能被修改。

随着新文档索引不断被写入，当日志数据大小超过512M或者时间超过30分钟时，会触发一次 flush。内存中的数据被写入到一个新段同时被写入到文件缓存系统，文件系统缓存中数据通过 fsync 刷新到磁盘中，生成提交点，日志文件被删除，创建一个空的新日志。

通过这种方式当断电或需要重启时，ES不仅要根据提交点去加载已经持久化过的段，还需要工具Translog里的记录，把未持久化的数据重新持久化到磁盘上，避免了数据丢失的可能。

段合并#

由于自动刷新流程每秒会创建一个新的段 ，这样会导致短时间内的段数量暴增。而段数目太多会带来较大的麻烦。 每一个段都会消耗文件句柄、内存和cpu运行周期。更重要的是，每个搜索请求都必须轮流检查每个段然后合并查询结果，所以段越多，搜索也就越慢。

Elasticsearch通过在后台定期进行段合并来解决这个问题。小的段被合并到大的段，然后这些大的段再被合并到更大的段。段合并的时候会将那些旧的已删除文档从文件系统中清除。被删除的文档不会被拷贝到新的大段中。合并的过程中不会中断索引和搜索。

段合并在进行索引和搜索时会自动进行，合并进程选择一小部分大小相似的段，并且在后台将它们合并到更大的段中，这些段既可以是未提交的也可以是已提交的。合并结束后老的段会被删除，新的段被 flush 到磁盘，同时写入一个包含新段（已排除旧的被合并的段）的新提交点，新的段被打开可以用来搜索。

段合并的计算量庞大， 而且还要吃掉大量磁盘 I/O，段合并会拖累写入速率，如果任其发展会影响搜索性能。Elasticsearch在默认情况下会对合并流程进行资源限制，所以搜索仍然有足够的资源很好地执行。

性能优化#

存储设备#

磁盘在现代服务器上通常都是瓶颈。Elasticsearch 重度使用磁盘，你的磁盘能处理的吞吐量越大，你的节点就越稳定。这里有一些优化磁盘 I/O 的技巧：

使用 SSD。就像其他地方提过的， 他们比机械磁盘优秀多了。

使用 RAID 0。条带化 RAID 会提高磁盘 I/O，代价显然就是当一块硬盘故障时整个就故障了。不要使用镜像或者奇偶校验 RAID 因为副本已经提供了这个功能。

另外，使用多块硬盘，并允许 Elasticsearch 通过多个 path.data 目录配置把数据条带化分配到它们上面。

不要使用远程挂载的存储，比如 NFS 或者 SMB/CIFS。这个引入的延迟对性能来说完全是背道而驰的。

如果你用的是 EC2，当心 EBS。即便是基于 SSD 的 EBS，通常也比本地实例的存储要慢。

内部索引优化#

Elasticsearch为了能快速找到某个term，先将所有的term排个序，然后根据二分法查找term，时间复杂度为logN，就像通过字典查找一样，这就是Term Dictionary。现在再看起来，似乎和传统数据库通过B-Tree的方式类似。

但是如果term太多，term dictionary也会很大，放内存不现实，于是有了Term Index，就像字典里的索引页一样，A开头的有哪些term，分别在哪页，可以理解term index是一颗树。这棵树不会包含所有的term，它包含的是term的一些前缀。通过term index可以快速地定位到term dictionary的某个offset，然后从这个位置再往后顺序查找。

在内存中用FST方式压缩term index，FST以字节的方式存储所有的term，这种压缩方式可以有效的缩减存储空间，使得term index足以放进内存，但这种方式也会导致查找时需要更多的CPU资源。演示地址：Build your own FST

对于存储在磁盘上的倒排表同样也采用了压缩技术减少存储所占用的空间，更多可以阅读 Frame of Reference and Roaring Bitmaps。

调整配置参数#

给每个文档指定有序的具有压缩良好的序列模式ID，避免随机的UUID-4 这样的 ID，这样的ID压缩比很低，会明显拖慢 Lucene。

对于那些不需要聚合和排序的索引字段禁用Doc values。Doc Values是有序的基于document => field value的映射列表；

不需要做模糊检索的字段使用 keyword类型代替 text 类型，这样可以避免在建立索引前对这些文本进行分词。

如果你的搜索结果不需要近实时的准确度，考虑把每个索引的 index.refresh\_interval 改到 30s 。如果你是在做大批量导入，导入期间你可以通过设置这个值为 -1 关掉刷新，还可以通过设置 index.number\_of\_replicas: 0关闭副本。别忘记在完工的时候重新开启它。

避免深度分页查询建议使用Scroll进行分页查询。普通分页查询时，会创建一个from + size的空优先队列，每个分片会返回from + size 条数据，默认只包含文档id和得分score给协调节点，如果有n个分片，则协调节点再对（from + size）× n 条数据进行二次排序，然后选择需要被取回的文档。当from很大时，排序过程会变得很沉重占用CPU资源严重。

减少映射字段，只提供需要检索，聚合或排序的字段。其他字段可存在其他存储设备上，例如Hbase，在ES中得到结果后再去Hbase查询这些字段。

创建索引和查询时指定路由routing值，这样可以精确到具体的分片查询，提升查询效率。路由的选择需要注意数据的分布均衡。

JVM调优#

确保堆内存最小值（ Xms ）与最大值（ Xmx ）的大小是相同的，防止程序在运行时改变堆内存大小。

Elasticsearch 默认安装后设置的堆内存是 1 GB。可通过../config/jvm.option文件进行配置，但是最好不要超过物理内存的50%和超过32GB。

GC 默认采用CMS的方式，并发但是有STW的问题，可以考虑使用G1收集器。

ES非常依赖文件系统缓存（Filesystem Cache），快速搜索。一般来说，应该至少确保物理上有一半的可用内存分配到文件系统缓存。