# Filebeat原理

Logstash输出使用Lumberjack协议将事件直接发送到Logstash，该协议在TCP上运行。Logstash允许对生成的事件进行附加处理和路由

compression\_level

gzip压缩级别。将此值设置为0将禁用压缩。压缩级别必须在1(最佳速度)到9(最佳压缩)的范围内。

提高压缩级别将减少网络使用率，但会增加CPU使用率。

默认值为3。

**close\_\* 系列**

log input 中有一系列以 close\_开头配置，这些配置决定了 Harvester 何时结束对文件的读取。

1. close\_eof  
   如果读取到了 EOF(即文件末尾)，是否要结束读取。如果为 true，则读取到文件末尾就结束读取，否则 Harvester 将会继续工作。默认只为 false。
2. close\_inactive  
   如果配置了 close\_eof 为 false，则 Harvester 即使读取到了文件末尾也不会终止。close\_inactive 决定了最长没有读到新消息的时长，默认为 5m(即五分钟)。如果超过了 close\_inactive 规定的时间依然没有新消息，则 Harvester 退出。
3. close\_timeout  
   决定了一个 Harvester 的最长工作时间，如果 Harvester 工作了一段时间后依然没有停止，则强行停止 Harvester。默认为 0，表示不强行停止 Harvester。
4. close\_renamed  
   文件更名时是否退出，默认为 false。文件更名一般发生在日志轮替的场景下。
5. close\_removed  
   表示当文件被删除时 Harvester 是否要继续。默认为 true，表示当文件被删除时，Harvester 即刻停止工作。

不过即使 Harvester 关闭了也关系不大。因为根据 filebeat 会定时扫描文件，如果关闭后又有了新增内容，filebeat 依然是可以检查出来的。

**clean\_\* 系列**

clean\_开头的一系列配置用来清理 Registrar 中的文件状态，同时也可以起到减小 Registrar 文件大小、防止 inode 复用等作用。

1. clean\_inactive  
   表示一个时间段。用于移除已经一长段时间没有新产生内容的日志文件，默认为 0，表示禁用该功能。
2. clean\_removed  
   在 Registrar 中移除那些已经不存在的文件。默认为 true，表示当文件不存在时，则从 Registrar 中移除。

# Logstash原理

# Elasticsearch原理

term index在内存中是以FST（finite state transducers）的形式保存的，其特点是非常节省内存。Term dictionary在磁盘上是以分block的方式保存的，一个block内部利用公共前缀压缩，比如都是Ab开头的单词就可以把Ab省去。这样term dictionary可以比b-tree更节约磁盘空间。

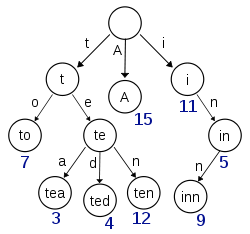
Log

Term Dictionary

Elasticsearch为了能快速找到某个term，将所有的term排个序，二分法查找term，logN的查找效率，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的方式类似啊，为什么说比B-Tree的查询快呢？

Term Index

B-Tree通过减少磁盘寻道次数来提高查询性能，Elasticsearch也是采用同样的思路，直接通过内存查找term，不读磁盘，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。

索引结构

　　Lucene现在采用的数据结构为FST，它的特点就是：

　　1、词查找复杂度为O(len(str))

　　2、共享前缀、节省空间

　　3、内存存放前缀索引、磁盘存放后缀词块

　　这跟我们前面说到的词典结构三要素是一致的：1. 查询速度。2. 内存占用。3. 内存+磁盘结合。我们往索引库里插入四个单词abd、abe、acf、acg,看看它的索引文件内容。**TraceId 生成规则**

SOFATracer 通过 TraceId 来将一个请求在各个服务器上的调用日志串联起来，TraceId 一般由接收请求经过的第一个服务器产生，产生规则是： 服务器 IP + 产生 ID 时候的时间 + 自增序列 + 当前进程号 ，比如：

0ad1348f1403169275002100356696

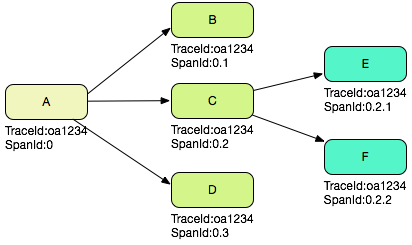
前 8 位 0ad1348f 即产生 TraceId 的机器的 IP，这是一个十六进制的数字，每两位代表 IP 中的一段，我们把这个数字，按每两位转成 10 进制即可得到常见的 IP 地址表示方式 10.209.52.143，大家也可以根据这个规律来查找到请求经过的第一个服务器。 后面的 13 位 1403169275002 是产生 TraceId 的时间。 之后的 4 位 1003 是一个自增的序列，从 1000 涨到 9000，到达 9000 后回到 1000 再开始往上涨。 最后的 5 位 56696 是当前的进程 ID，为了防止单机多进程出现 TraceId 冲突的情况，所以在 TraceId 末尾添加了当前的进程 ID。

TraceId 目前的生成的规则参考了阿里的鹰眼组件。

**SpanId 生成规则**

SOFATracer 中的 SpanId 代表本次调用在整个调用链路树中的位置，假设一个 Web 系统 A 接收了一次用户请求，那么在这个系统的 SOFATracer MVC 日志中，记录下的 SpanId 是 0，代表是整个调用的根节点，如果 A 系统处理这次请求，需要通过 RPC 依次调用 B，C，D 三个系统，那么在 A 系统的 SOFATracer RPC 客户端日志中，SpanId 分别是 0.1，0.2 和 0.3，在 B，C，D 三个系统的 SOFATracer RPC 服务端日志中，SpanId 也分别是 0.1，0.2 和 0.3；如果 C 系统在处理请求的时候又调用了 E，F 两个系统，那么 C 系统中对应的 SOFATracer RPC 客户端日志是 0.2.1 和 0.2.2，E，F 两个系统对应的 SOFATracer RPC 服务端日志也是 0.2.1 和 0.2.2。根据上面的描述，我们可以知道，如果把一次调用中所有的 SpanId 收集起来，可以组成一棵完整的链路树。

我们假设一次分布式调用中产生的 TraceId 是 0a1234（实际不会这么短），那么根据上文 SpanId 的产生过程，有下图：



SpanId 目前的生成的规则参考了阿里的鹰眼组件。