**filebeat的安装和使用**

| | | | |
|---|---|---|---|
| **笔记本：** | ElasticSearch | | |
| **创建时间：** | 2020/6/4 16:18 | **更新时间：** | 2020/6/4 16:18 |
| **作者：** | jin.zhou@definesys.com | | |

@[TOC]

# 什么是filebeat

Filebeat是本地文件的日志数据采集器，是使用golang语言编写的，可监控日志目录或特定日志文件（tail file），并将它们转发给Elasticsearch或Logstatsh进行索引、kafka等。带有内部模块（auditd，Apache，Nginx，System和MySQL），可通过一个指定命令来简化通用日志格式的收集，解析和可视化。

# 安装filebeat

这里我们使用可以两种方式安装filebeat 7.1.1
1、使用rpm包管理器安装

```
curl -L -O
https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-7.1.1-
x86_64.rpm
sudo rpm -vi filebeat-7.1.1-x86_64.rpm
```

2、解压安装

```
curl -L -O
https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-7.1.1-
darwin-x86_64.tar.gz
tar xzvf filebeat-7.1.1-darwin-x86_64.tar.gz
cd filebeat-7.1.1-darwin-x86_64/
```

安装好之后可以进入目录 /etc/filebeat/ 看到配置文件 filebeat.yml

# 配置文件解读

filebeat的配置文件分为一下几个部分

1. inputs
   输入部分，在这里面可以配置日志文件的输入，以及可以做一下特殊处理
   例如这段配置：
   其中 `- type: log` 为一个类型的日志读取的开始，可以通过`fields`字段添加额外的字段以供我们来使用，更多的解释看代码里面的注释

```
filebeat.inputs:
# Each - is an input. Most options can be set at the input level, so
# you can use different inputs for various configurations.
# Below are the input specific configurations.
- type: log ## 固定
  enabled: true #是否启用
  paths: #日志路径
    - /data/test.log
  tags: ["test","mylog",'1'] #标记，选填
  tail_files: true #从尾部开始读取日志
  encoding: UTF-8 #编码格式
  fields:  #附加字段
    partition: "1"
    log_topic: "testLogPlatform"
  fields_under_root: true  #把附加字段作为一级字段
  # 多行合并的配置，比如异常信息合并为一行
  multiline.pattern: '^[[:space:]]+(at|\.{3})\b|^Caused by:'
  multiline.negate: false
  multiline.match: after
```

2. modules
   模块配置，这里不多做解释
3. Elasticsearch template setting
   ElasticSearch输出模板配置
4. General
   通用配置
5. Dashboards
   仪表盘配置
6. Kibana
   kibana相关配置
7. Elastic Cloud
   elasticSearch公有云配置
8. Outputs
   输出属性配置

# 输出日志到kafka

在filebeat里面我们只需要在output模块 配置一下kafka的输出配置就可以数据发送到kafka

```
#------------------------------ Kafka output ------------------------------
----
output.kafka:
  hosts:
["172.16.161.51:9002","172.16.161.51:9003","172.16.161.51:9004"]
  topic: 'testTopic'
  partition.hash:
    reachable_only: false
```

# 如何输出到不同的主题

根据上面的配置我们是把topic固定为testTopic了，无法输出到其他的主题里面去，但是我们希望不同类型的数据输出到不同的主题里面去，这时候上面定义的附加字段就可以发挥作用了，如下面的例子，我们在上面的配置中定义了附加字段 `log_topic: "testLogPlatform"` 那么我们这里可以使用表达式`'%{[log_topic]}'` 来动态的输出到不同的主题

```
#------------------------------ Kafka output ------------------------------
----
output.kafka:
  hosts:
["172.16.161.51:9002","172.16.161.51:9003","172.16.161.51:9004"]
  topic: '%{[log_topic]}'
  partition.hash:
    reachable_only: false
```

# 如何输出到指定的分区

filebeat的分区分发方式一共有3种
1、随机分发
2、轮询分发
3、hash分发
上面说到了如何输出到指定的主题，但是我们除了需要输出到指定的主题之外有可能还需要输出到指定的分区，但是filebeat没有为我们提供输出到指定分区的配置，苦思冥想，filebeat为我们提供了hash的方式来做分区分发，而且还可以指定字段作为求hash值的字段。
于是上面的附加字段又发挥作用了，例如下面的配置：
使用了附加字段`partition` 求hash值然后进行分发

```
#------------------------------ Kafka output ------------------------------
----
output.kafka:
  hosts:
["172.16.161.51:9002","172.16.161.51:9003","172.16.161.51:9004"]
  topic: '%{[log_topic]}'
  partition.hash:
    reachable_only: false
    hash: ['partition']
```

关于附加字段 topic和partition 的值如何确定会在日志平台后台的使用上说明。

# 如何获取真实的分区

上面使用了hash的方式之后由于我们还是不知道这个字段经过hash之后他得分区到底是哪一个，因此我们不得不翻开了filebeat的源码。

```
func cfgHashPartitioner(log *logp.Logger, config *common.Config) (func()
partitioner, error) {
        cfg := struct {
                Hash   []string `config:"hash"`
                Random bool     `config:"random"`
        }{
                Random: true,
        }
        if err := config.Unpack(&cfg); err != nil {
                return nil, err
        }

        if len(cfg.Hash) == 0 {
                return makeHashPartitioner, nil
        }

        return func() partitioner {
                // 1、根据指定的字段使用hash方式获取分区
                return makeFieldsHashPartitioner(log, cfg.Hash,
!cfg.Random)
        }, nil
}

func makeHashPartitioner() partitioner {
        generator := rand.New(rand.NewSource(rand.Int63()))
```

```
        hasher := fnv.New32a()

        return func(msg *message, numPartitions int32) (int32, error) {
                if msg.key == nil {
                        return
int32(generator.Intn(int(numPartitions))), nil
                }

                hash := msg.hash
                if hash == 0 {
                        hasher.Reset()
                        if _, err := hasher.Write(msg.key); err != nil
{
                                return -1, err
                        }
                        msg.hash = hasher.Sum32()
                        hash = msg.hash
                }

                // create positive hash value
                return hash2Partition(hash, numPartitions)
        }
}

func makeFieldsHashPartitioner(log *logp.Logger, fields []string,
dropFail bool) partitioner {
        generator := rand.New(rand.NewSource(rand.Int63()))
        hasher := fnv.New32a()

        return func(msg *message, numPartitions int32) (int32, error) {
                hash := msg.hash
                if hash == 0 {
                        hasher.Reset()

                        var err error
                        for _, field := range fields {
                                // 2、验证字段hash是否报错，一般不会报
错
                                err = hashFieldValue(hasher,
msg.data.Content.Fields, field)
                                if err != nil {
                                        break
                                }
                        }

                        if err != nil {
                                if dropFail {
```

```
                                        log.Errorf("Hashing partition
key failed: %+v", err)
                                        return -1, err
                                }

                                msg.hash = generator.Uint32()
                        } else {
                        //3、上面验证之后在进行一次hash求和
                                msg.hash = hasher.Sum32()
                        }
                        hash = msg.hash
                }
                //4、根据hash值使用分区数目求模，获取最终的分区编号
                return hash2Partition(hash, numPartitions)
        }
}

func hash2Partition(hash uint32, numPartitions int32) (int32, error) {
        p := int32(hash)
        if p < 0 {
                p = -p
        }
        return p % numPartitions, nil
}

func hashFieldValue(h hash.Hash32, event common.MapStr, field string)
error {
        type stringer interface {
                String() string
        }

        type hashable interface {
                Hash32(h hash.Hash32) error
        }

        v, err := event.GetValue(field)
        if err != nil {
                return err
        }

        switch s := v.(type) {
        case hashable:
                err = s.Hash32(h)
        case string:
                _, err = h.Write([]byte(s))
        case []byte:
                _, err = h.Write(s)
```

```
        case stringer:
                _, err = h.Write([]byte(s.String()))
        case int8, int16, int32, int64, int,
                uint8, uint16, uint32, uint64, uint:
                err = binary.Write(h, binary.LittleEndian, v)
        case float32:
                tmp := strconv.FormatFloat(float64(s), 'g', -1, 32)
                _, err = h.Write([]byte(tmp))
        case float64:
                tmp := strconv.FormatFloat(s, 'g', -1, 32)
                _, err = h.Write([]byte(tmp))
        default:
                // try to hash using reflection:
                err = binary.Write(h, binary.LittleEndian, v)
                if err != nil {
                        err = fmt.Errorf("can not hash key '%v' of
unknown type", field)
                }
        }
        return err
}
```

通过阅读上面最主要的方法`func makeFieldsHashPartitioner(log *logp.Logger, fields []string, dropFail bool) partitioner` 可以把整个hash的过程简化成下面这样，因此我们可以通过下面这个方法获取我们在filebeat里面配置的hash字段的值对应的分区了

```
func testHash(val string,partitionNum int32) int32 {
        hasher := fnv.New32a()
        hasher.Write([]byte(val))
        hash := hasher.Sum32()
        p := int32(hash)
        if p < 0 {
                p = -p
        }
        i := p % partitionNum
        fmt.Println(i)
        return i;
}
```

# 如何解决hash冲突

由于存在hash冲突的情况发生，也就是不同的值却出现了相同的hash值，为了解决这个问题，我们可以使用数字作为hash的key去求hash值，同时判断这个hash值是否已经使用了，如果使用了那么就使用下一个key再来求hash值，知道出现一个未使用的为止。