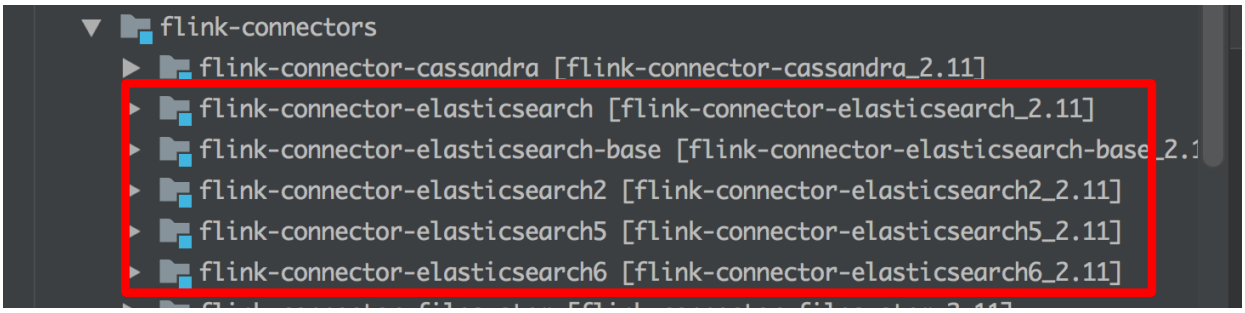
## 前言

在上一节 Flink 读取 Kafka 数据处理后批量写入到 MySQL 的总结中，我讲到我们生产环境中的数据量会很大，而对于 MySQL 是承受不了这种数据量写入，所以可以考虑换其他的存储中间件，比如 ES、HBase 等。不过到底存储在哪里，还是得看我们的业务需求和自己公司的情况，选择最适合自己公司的才是最好的。

我们这一节就是讲解一下如何 Flink 如何读取 Kafka 数据处理后写入到 ElasticSearch?

## 准备条件

因为在文章 Flink 环境准备 我已经讲过 ElasticSearch 的安装，这里就不做过多的重复，需要注意的一点就是 Flink 的 ElasticSearch Connector 是区分版本号的。




所以添加依赖的时候要区分一下，根据你安装的 ElasticSearch 来选择不一样的版本依赖，另外就是不同版本的 ElasticSearch 还会导致下面的数据写入到 ElasticSearch 中出现一些不同，我们这里使用的版本是 ElasticSearch6，如果你使用的是其他的版本可以参考官网的实现。

## 添加依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-elasticsearch6_${scala.binary.version}
</artifactId>
    <version>${flink.version}</version>
</dependency>
```

上面这依赖版本号请自己根据使用的版本对应改变下。

下面所有的代码都没有把 import 引入到这里来，如果需要查看更详细的代码，请查看我的 GitHub 仓库地址：

https://github.com/zhisheng17/flink-learning/tree/master/flink-learning-connectors/flink-learning-connectors-es6

> 这个 module 含有本文的所有代码实现，后期可能因为代码做了部分重构，所以如果有代码改变很正常，请直接查看全部项目代码。

## ElasticSearchSinkUtil 工具类

这个工具类是自己封装的，getEsAddresses 方法将传入的配置文件 es 地址解析出来，可以是域名方式，也可以是 ip + port 形式。addSink 方法是利用了 Flink 自带的 ElasticsearchSink 来封装了一层，传入了一些必要的调优参数和 es 配置参数，下面章节还会再讲些其他的配置。

ElasticSearchSinkUtil.java

```
1  public class ElasticSearchSinkUtil {
2
3      /**
4       * es sink
5       *
6       * @param hosts es hosts
7       * @param bulkFlushMaxActions bulk flush size
8       * @param parallelism 并行数
9       * @param data 数据
10      * @param func
11      * @param <T>
12      */
13     public static <T> void addSink(List<HttpHost> hosts, int
    bulkFlushMaxActions, int parallelism,
14                                    SingleOutputStreamOperator<T> data,
    ElasticsearchSinkFunction<T> func) {
15         ElasticsearchSink.Builder<T> esSinkBuilder = new
    ElasticsearchSink.Builder<>(hosts, func);
16         esSinkBuilder.setBulkFlushMaxActions(bulkFlushMaxActions);
17         data.addSink(esSinkBuilder.build()).setParallelism(parallelism);
18     }
19
20     /**
21      * 解析配置文件的 es hosts
22      *
23      * @param hosts
24      * @return
25      * @throws MalformedURLException
26      */
27     public static List<HttpHost> getEsAddresses(String hosts) throws
    MalformedURLException {
28         String[] hostList = hosts.split(",");
29         List<HttpHost> addresses = new ArrayList<>();
30         for (String host : hostList) {
31             if (host.startsWith("http")) {
32                 URL url = new URL(host);
33                 addresses.add(new HttpHost(url.getHost(), url.getPort()));
34             } else {
35                 String[] parts = host.split(":", 2);
36                 if (parts.length > 1) {
37                     addresses.add(new HttpHost(parts[0],
    Integer.parseInt(parts[1])));
38                 } else {
```

```
                    throw new MalformedURLException("invalid elasticsearch hosts format");
                }
            }
        }
        return addresses;
    }
}
```

## Main 启动类

Main.java

```
public class Main {
    public static void main(String[] args) throws Exception {
        //获取所有参数
        final ParameterTool parameterTool = ExecutionEnvUtil.createParameterTool(args);
        //准备好环境
        StreamExecutionEnvironment env = ExecutionEnvUtil.prepare(parameterTool);
        //从kafka读取数据
        DataStreamSource<Metrics> data = KafkaConfigUtil.buildSource(env);

        //从配置文件中读取 es 的地址
        List<HttpHost> esAddresses = ElasticSearchSinkUtil.getEsAddresses(parameterTool.get(ELASTICSEARCH_HOSTS));
        //从配置文件中读取 bulk flush size，代表一次批处理的数量，这个可是性能调优参数，特别提醒
        int bulkSize = parameterTool.getInt(ELASTICSEARCH_BULK_FLUSH_MAX_ACTIONS, 40);
        //从配置文件中读取并行 sink 数，这个也是性能调优参数，特别提醒，这样才能够更快的消费，防止 kafka 数据堆积
        int sinkParallelism = parameterTool.getInt(STREAM_SINK_PARALLELISM, 5);

        //自己再自带的 es sink 上一层封装了下
        ElasticSearchSinkUtil.addSink(esAddresses, bulkSize, sinkParallelism, data,
                (Metrics metric, RuntimeContext runtimeContext, RequestIndexer requestIndexer) -> {
                    requestIndexer.add(Requests.indexRequest()
                            .index(ZHISHENG + "_" + metric.getName()) //es 索引名
                            .type(ZHISHENG) //es type
                            .source(GsonUtil.toJSONBytes(metric), XContentType.JSON));
                });
        env.execute("flink learning connectors es6");
    }
}
```

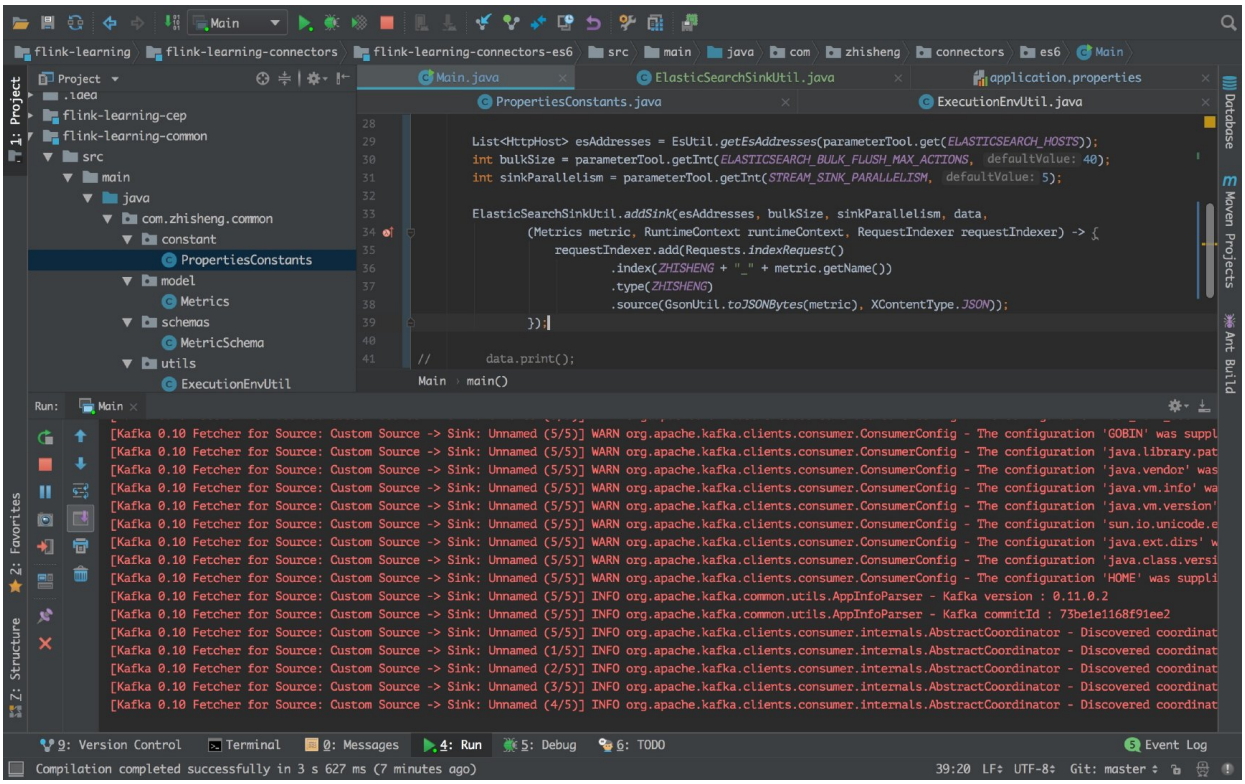
## 配置文件

配置都支持集群模式填写，注意用 ， 分隔！

```
kafka.brokers=localhost:9092
kafka.group.id=zhisheng-metrics-group-test
kafka.zookeeper.connect=localhost:2181
metrics.topic=zhisheng-metrics
stream.parallelism=5
stream.checkpoint.interval=1000
stream.checkpoint.enable=false
elasticsearch.hosts=localhost:9200
elasticsearch.bulk.flush.max.actions=40
stream.sink.parallelism=5
```

## 运行结果

执行 Main 类的 main 方法，我们的程序是只打印 Flink 的日志，没有打印存入的日志（因为我们这里没有打日志）：




所以看起来不知道我们的 Sink 是否有用，数据是否从 Kafka 读取出来后存入到 ES 了。

你可以查看下本地起的 ES 终端或者服务器的 ES 日志就可以看到效果了。

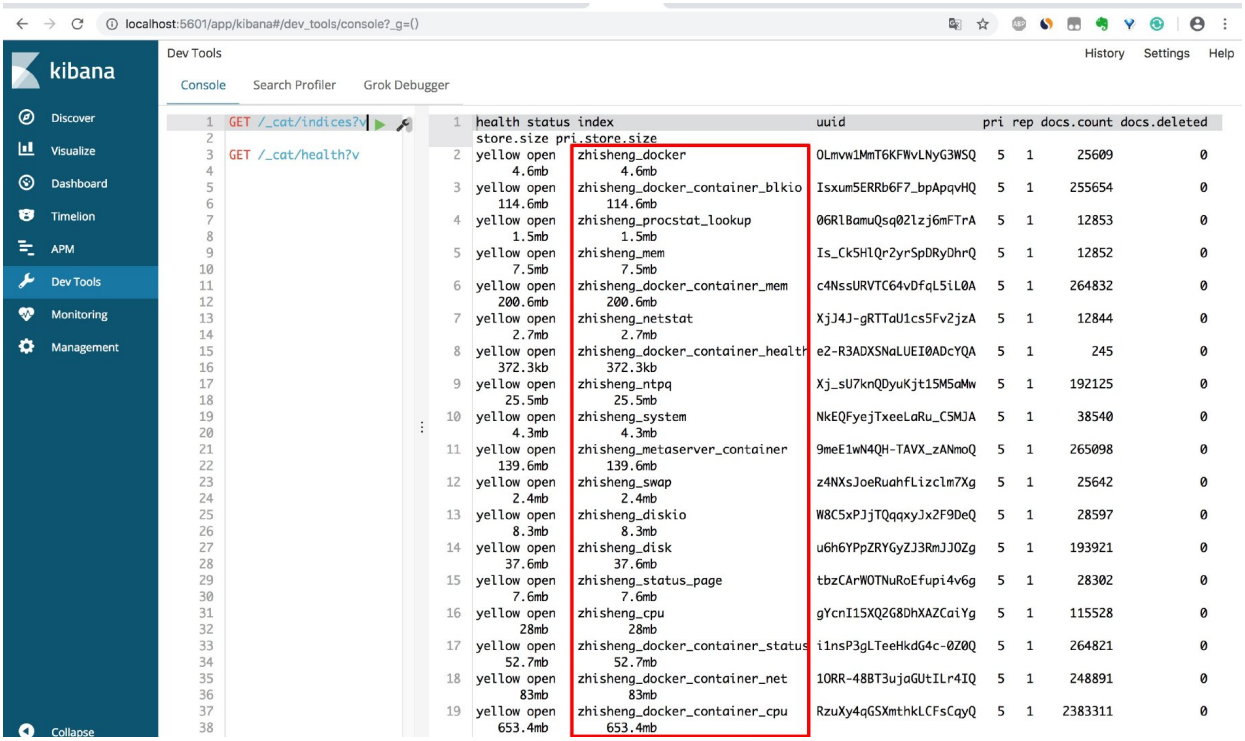
ES 日志如下：

```
2019-01-01T21:53:30,276][INFO ][o.e.i.IndexingMemoryController] [node-1] now throttling indexing for shard [[zhisheng_metaserver_container][0]
: segment writing can't keep up
2019-01-01T21:53:30,276][INFO ][o.e.i.IndexingMemoryController] [node-1] now throttling indexing for shard [[zhisheng_metaserver_container][3]
: segment writing can't keep up
2019-01-01T21:53:30,276][INFO ][o.e.i.IndexingMemoryController] [node-1] now throttling indexing for shard [[zhisheng_docker_container_cpu][0]
: segment writing can't keep up
2019-01-01T21:53:30,276][INFO ][o.e.i.IndexingMemoryController] [node-1] now throttling indexing for shard [[zhisheng_docker_container_cpu][3]
: segment writing can't keep up
2019-01-01T21:53:30,276][INFO ][o.e.i.IndexingMemoryController] [node-1] now throttling indexing for shard [[zhisheng_docker_container_mem][0]
: segment writing can't keep up
2019-01-01T21:53:30,276][INFO ][o.e.i.IndexingMemoryController] [node-1] now throttling indexing for shard [[zhisheng_docker_container_mem][3]
: segment writing can't keep up
2019-01-01T21:53:30,276][INFO ][o.e.i.IndexingMemoryController] [node-1] now throttling indexing for shard [[zhisheng_docker_container_mem][1]
: segment writing can't keep up
2019-01-01T21:53:30,276][INFO ][o.e.i.IndexingMemoryController] [node-1] now throttling indexing for shard [[zhisheng_docker_container_mem][4]
: segment writing can't keep up
2019-01-01T21:53:31,461][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_metaserver_container][0
]
2019-01-01T21:53:31,461][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_procstat][2]]
2019-01-01T21:53:31,461][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_procstat][0]]
2019-01-01T21:53:31,461][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_metaserver_container][3
]
2019-01-01T21:53:31,462][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_docker_container_cpu][0
]
2019-01-01T21:53:31,462][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_docker_container_mem][0
]
2019-01-01T21:53:31,462][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_procstat][3]]
2019-01-01T21:53:31,462][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_procstat][4]]
2019-01-01T21:53:31,462][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_metaserver_container][1
]
2019-01-01T21:53:31,462][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_docker_container_mem][4
]
2019-01-01T21:53:31,462][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_metaserver_container][2
]
2019-01-01T21:53:31,462][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_docker_container_mem][1
]
2019-01-01T21:53:31,462][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_docker_container_mem][3
]
2019-01-01T21:53:31,462][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_docker_container_cpu][3
]
2019-01-01T21:53:31,463][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_procstat][1]]
2019-01-01T21:53:31,463][INFO ][o.e.i.IndexingMemoryController] [node-1] stop throttling indexing for shard [[zhisheng_metaserver_container][4
]
```

上图是我本地 Mac 电脑终端的 ES 日志，可以看到我们的索引了。

如果还不放心，你也可以在你的电脑装个 Kibana，然后更加的直观查看下 ES 的索引情况（或者直接敲 ES 的命令）。

我们用 Kibana 查看存入 ES 的索引如下：



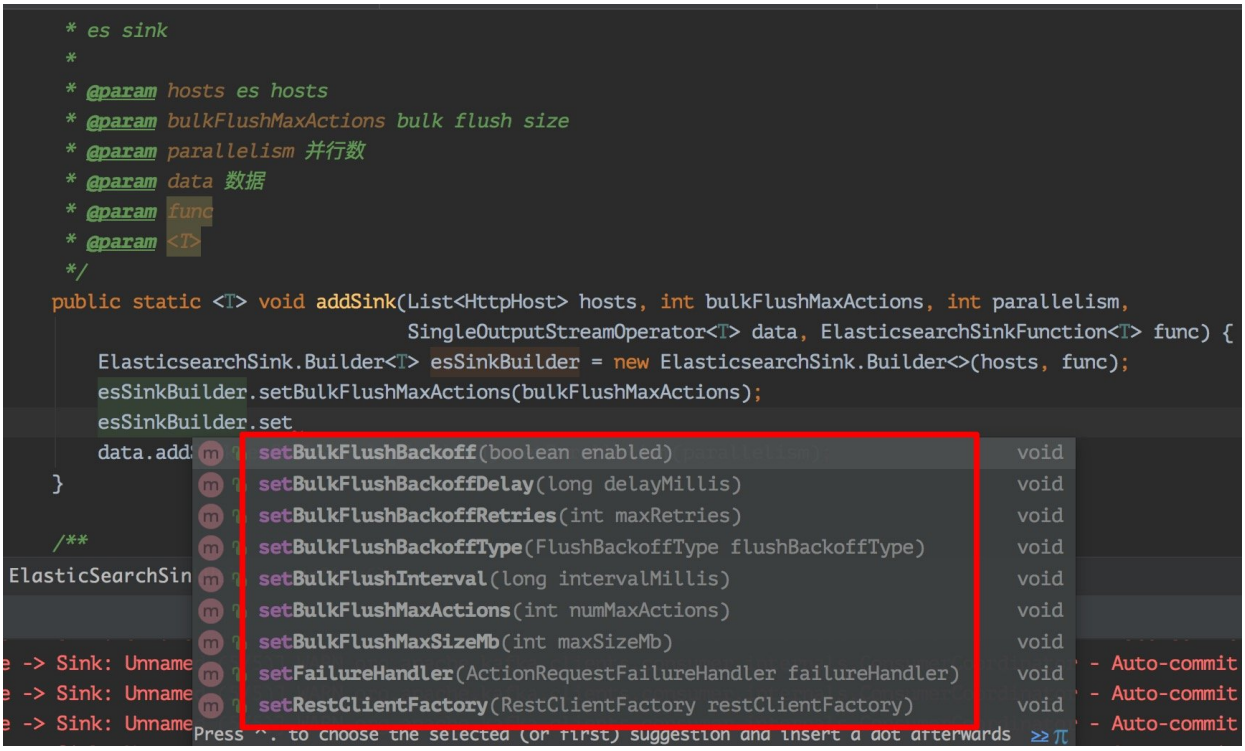

程序执行了一会，存入 ES 的数据量就很大了。

## 扩展配置

上面代码已经可以实现你的大部分场景了，但是如果你的业务场景需要保证数据的完整性（不能出现丢数据的情况），那么就需要添加一些重试策略，因为在我们的生产环境中，很有可能会因为某些组件不稳定性导致各种问题，所以这里我们就要在数据存入失败的时候做重试操作，这个 Flink 自带的 es sink 就支持了，常用的失败重试配置有：

```
1、bulk.flush.backoff.enable 用来表示是否开启重试机制

2、bulk.flush.backoff.type 重试策略，有两种：EXPONENTIAL 指数型（表示多次重试之间的时间间隔按照指数方式进行增长）、CONSTANT 常数型（表示多次重试之间的时间间隔为固定常数）

3、bulk.flush.backoff.delay 进行重试的时间间隔

4、bulk.flush.backoff.retries 失败重试的次数

5、bulk.flush.max.actions: 批量写入时的最大写入条数

6、bulk.flush.max.size.mb: 批量写入时的最大数据量

7、bulk.flush.interval.ms: 批量写入的时间间隔，配置后则会按照该时间间隔严格执行，无视上面的两个批量写入配置
```

看下，就是如下这些配置了，如果你需要的话，可以在这个地方配置扩充。




## FailureHandler 失败处理器

写入 ES 的时候会有这些情况会导致写入 ES 失败。

1、ES 集群队列满了，报如下错误：

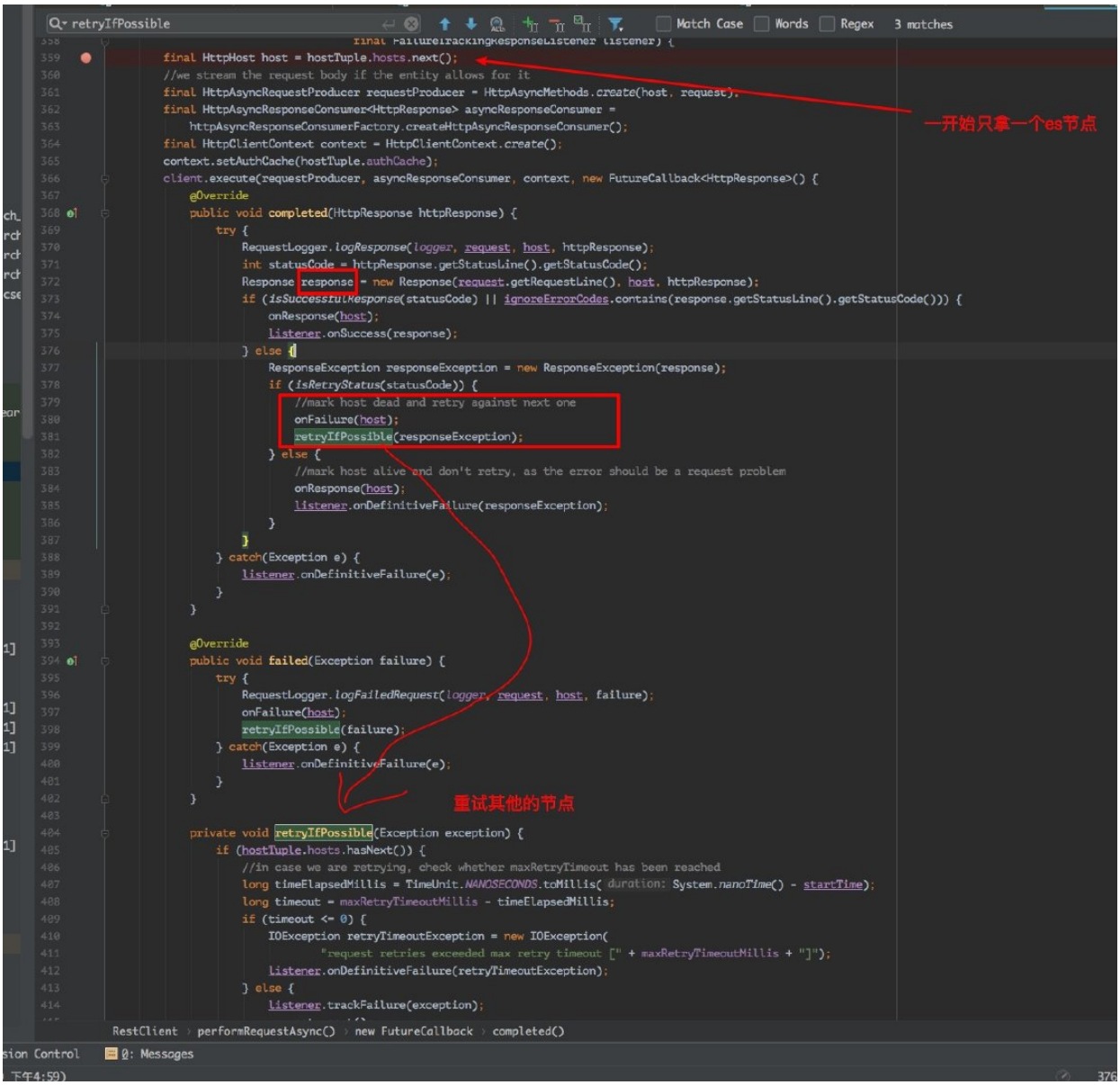
```
12:08:07.326 [I/O dispatcher 13] ERROR o.a.f.s.c.e.ElasticsearchSinkBase -
Failed Elasticsearch item request: ElasticsearchException[Elasticsearch
exception [type=es_rejected_execution_exception, reason=rejected execution
of org.elasticsearch.transport.TransportService$7@566c9379 on
EsThreadPoolExecutor[name = node-1/write, queue capacity = 200,
org.elasticsearch.common.util.concurrent.EsThreadPoolExecutor@f00b373[Runni
ng, pool size = 4, active threads = 4, queued tasks = 200, completed tasks
= 6277]]]]
```

是这样的，我电脑安装的 ES 队列容量默认应该是 200，我没有修改过。我这里如果配置的 bulk flush size * 并发 Sink 数量 这个值如果大于这个 queue capacity ，那么就很容易导致出现这种因为 ES 队列满了而写入失败。

当然这里你也可以通过调大点 es 的队列。参考：https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-threadpool.html

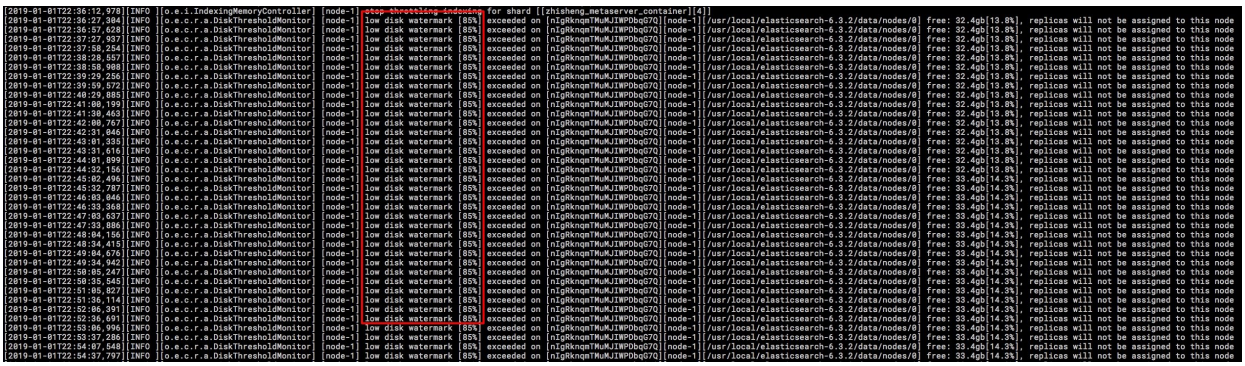
2、ES 集群某个节点挂了

这个就不用说了，肯定写入失败的。跟过源码可以发现 RestClient 类里的 performRequestAsync 方法一开始会随机的从集群中的某个节点进行写入数据，如果这台机器掉线，会进行重试在其他的机器上写入，那么当时写入的这台机器的请求就需要进行失败重试，否则就会把数据丢失！

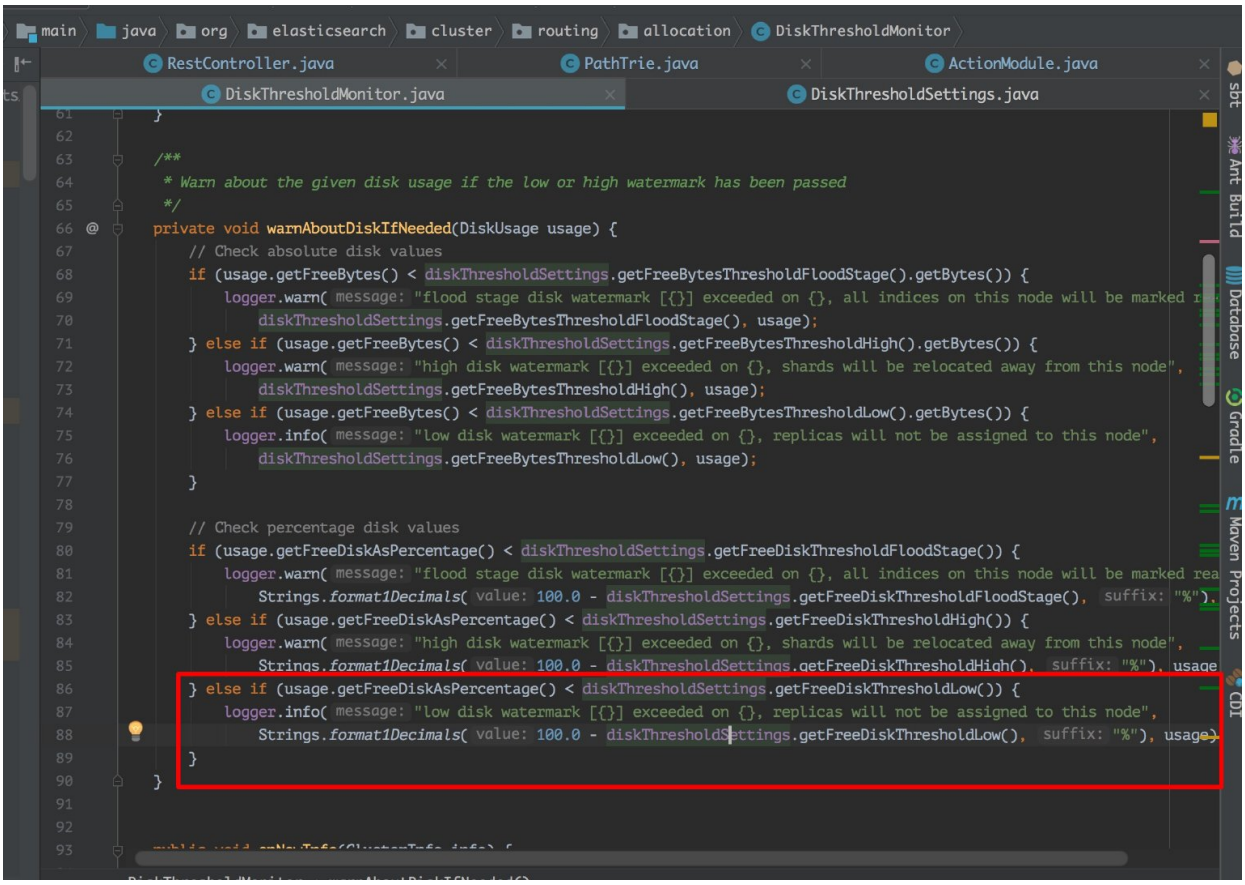


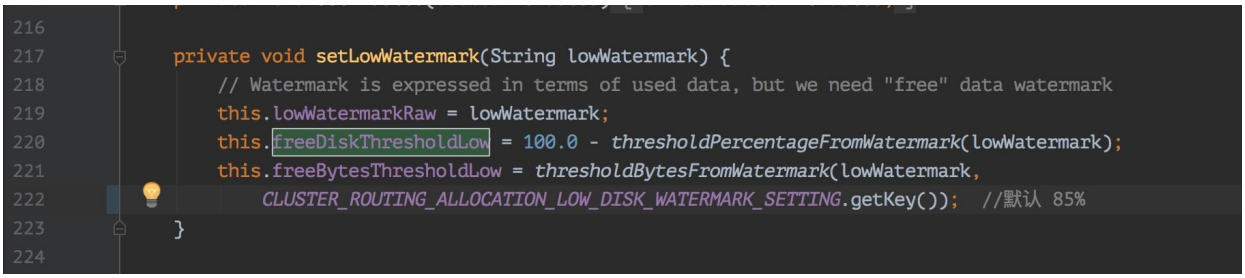

3、ES 集群某个节点的磁盘满了

这里说的磁盘满了，并不是磁盘真的就没有一点剩余空间的，是 ES 会在写入的时候检查磁盘的使用情况，在 85% 的时候会打印日志警告。



这里我看了下源码如下图：




如果你想继续让 ES 写入的话就需要去重新配一下 ES 让它继续写入，或者你也可以清空些不必要的数据腾出磁盘空间来。

## 解决方法

```
DataStream<String> input = ...;

input.addSink(new ElasticsearchSink<>(
    config, transportAddresses,
    new ElasticsearchSinkFunction<String>() {...},
    new ActionRequestFailureHandler() {
        @Override
        void onFailure(ActionRequest action,
                Throwable failure,
                int restStatusCode,
                RequestIndexer indexer) throw Throwable {

            if (ExceptionUtils.containsThrowable(failure,
EsRejectedExecutionException.class)) {
                //队列满了，重新添加用于索引的 document
```

```
                indexer.add(action);
            } else if (ExceptionUtils.containsThrowable(failure,
ElasticsearchParseException.class)) {
                // 对于有问题的 document，删除该请求，没有额外的错误处理逻辑
            } else {
                //对于抛出其他的异常错误，直接就当成 sink 失败，向外抛出异常，你也可
以抛出自定义的异常
                throw failure;
            }
        }
}));
```

如果仅仅只是想做失败重试，也可以直接使用官方提供的默认的 RetryRejectedExecutionFailureHandler ，该处理器会对 EsRejectedExecutionException 导致到失败写入做重试处理。如果你没有设置失败处理器（failure handler），那么就会使用默认的 NoOpFailureHandler 来简单处理所有的异常。

## 总结

本文写了如何利用 Flink 的 ElasticSearch connector 将 Kafka 中的数据读取并存储到 ElasticSearch 中，讲了如何封装自带的 Sink，还有一些扩展配置以及 FailureHandler 情况下（这个问题可是线上很容易遇到的）要怎么处理。

## Github 代码仓库

https://github.com/zhisheng17/flink-learning/tree/master/flink-learning-connectors/flink-learning-connectors-es6