五、Kafka功能扩展

-- 楼兰

这一章节主要分享一些在开发过程中用得比较多的Kafka相关扩展功能。

这一部分的内容非常依赖真实项目的实践经验，需要多理解解决各种相关问题的思路，而不是关注每一项具体功能的使用技巧。因为你要面对的问题是多样的，这些相关的小技巧是无尽的。

一、Kafka性能压测

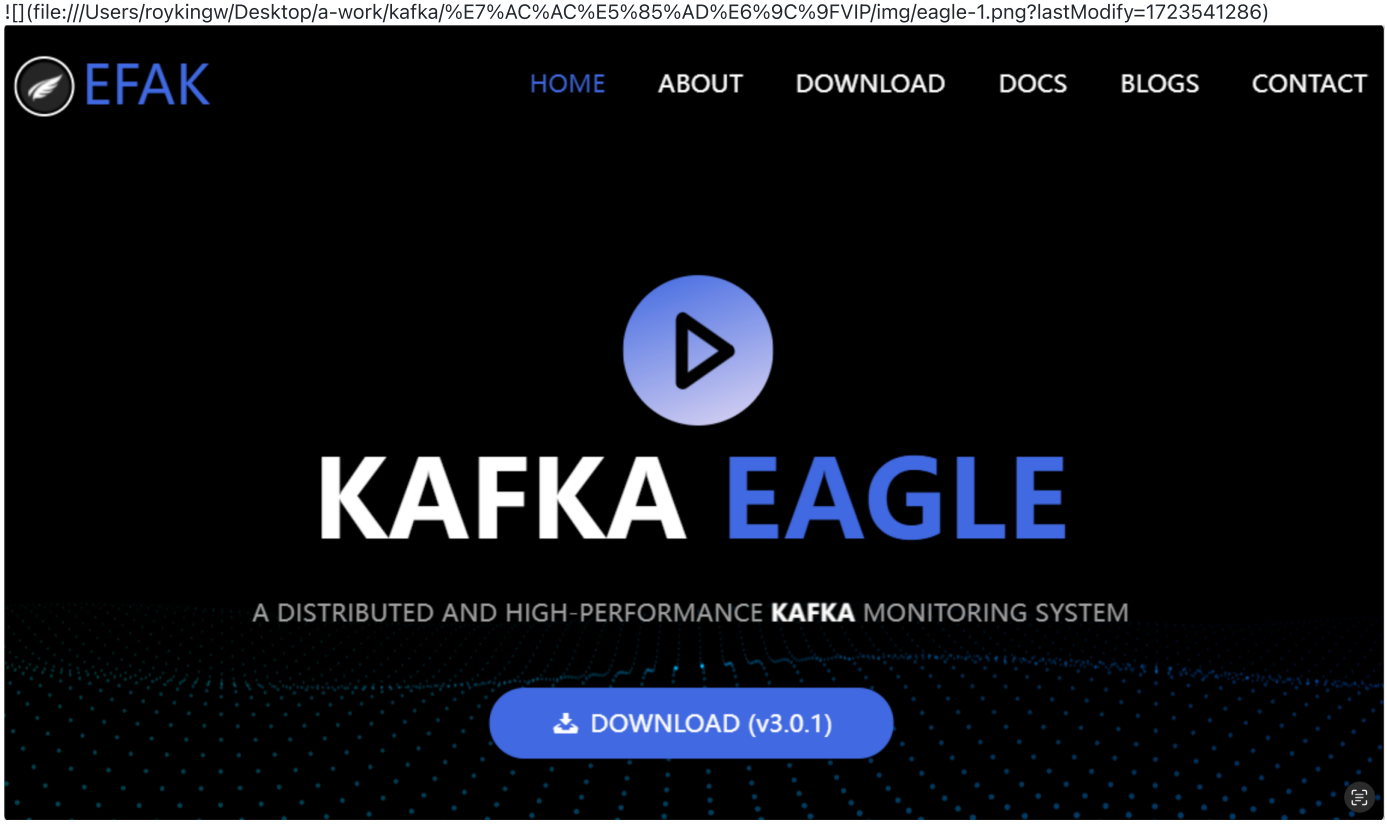
Kafka提供了一个性能压测的脚本，可以用来衡量集群的整体性能。

```
[root@192-168-65-112 kafka_2.13-3.8.0]# bin/kafka-producer-perf-test.sh --topic test --num-record 1000000 --record-size 1024 --
throughput -1 --producer-props bootstrap.servers=worker1:9092 acks=1
212281 records sent, 42456.2 records/sec (41.46 MB/sec), 559.9 ms avg latency, 1145.0 ms max latency.
463345 records sent, 92669.0 records/sec (90.50 MB/sec), 229.6 ms avg latency, 946.0 ms max latency.
1000000 records sent, 80560.702489 records/sec (78.67 MB/sec), 237.82 ms avg latency, 1145.00 ms max latency, 145 ms 50th, 699 ms
95th, 959 ms 99th, 1123 ms 99.9th.
```

通常会以此作为Kafka的基准测试。由此来衡量对Kafka的服务端配置是否充足。

二、搭建Kafka监控平台

生产环境通常会对Kafka搭建监控平台。而Kafka-eagle就是一个可以监控Kafka集群整体运行情况的框架，在生产环境经常会用到。官网地址：https://www.kafka-eagle.org/ 以前叫做Kafka-eagle，现在用了个简写，EFAK（Eagle For Apache Kafka）



环境准备：

在官网的DownLoad页面可以下载EFAK的运行包，efak-web-3.0.2-bin.tar.gz。

另外，EFAK需要依赖的环境主要是Java和数据库。其中，数据库支持本地化的SQLLite以及集中式的MySQL。生产环境建议使用MySQL。在搭建EFAK之前，需要准备好对应的服务器以及MySQL数据库。

> 略过MySQL服务搭建过程。
>
> 数据库不需要初始化，EFAK在执行过程中会自己完成初始化。

安装过程：以Linux服务器为例。

1、将efak压缩包解压。

```
tar -zxvf efak-web-3.0.2-bin.tar.gz -C /app/kafka/eagle
```

2、修改efak解压目录下的conf/system-config.properties。 这个文件中提供了完整的配置，下面只列出需要修改的部分。

```
######################################
# multi zookeeper & kafka cluster list
# Settings prefixed with 'kafka.eagle.' will be deprecated, use 'efak.' instead
######################################
# 指向Zookeeper地址
efak.zk.cluster.alias=cluster1
cluster1.zk.list=worker1:2181,worker2:2181,worker3:2181

######################################
# zookeeper enable acl
######################################
# Zookeeper权限控制
cluster1.zk.acl.enable=false
cluster1.zk.acl.schema=digest
#cluster1.zk.acl.username=test
#cluster1.zk.acl.password=test123

######################################
# kafka offset storage
######################################
# offset选择存在kafka中。
cluster1.efak.offset.storage=kafka
#cluster2.efak.offset.storage=zk

######################################
# kafka mysql jdbc driver address
######################################
#指向自己的MySQL服务。库需要提前创建
efak.driver=com.mysql.cj.jdbc.Driver
efak.url=jdbc:mysql://192.168.65.212:3306/ke?useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull
efak.username=root
efak.password=root
```

3、配置EFAK的环境变量

```
vi ~/.bash_profile
-- 配置KE_HOME环境变量，并添加到PATH中。
  export KE_HOME=/app/kafka/eagle/efak-web-3.0.2
  PATH=$PATH:#KE_HOME/bin:$HOME/.local/bin:$HOME/bin
--让环境变量生效
source ~/.bash_profile
```

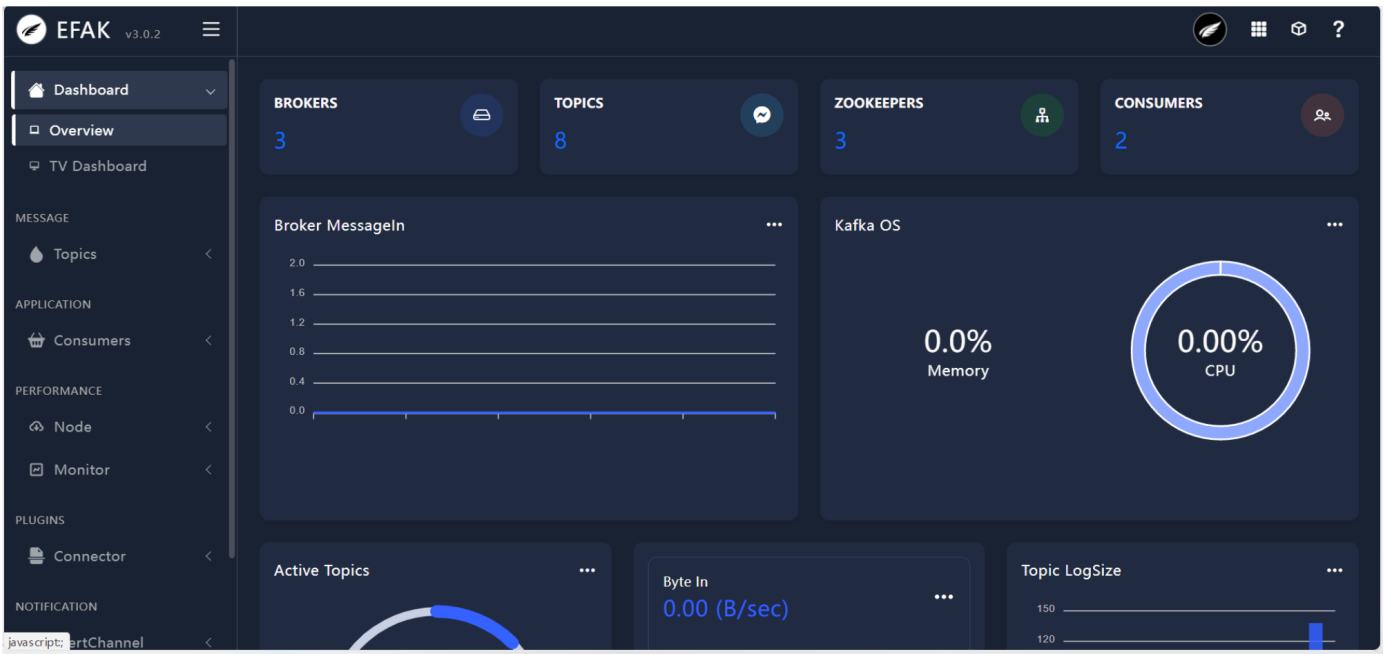
4、启动EFAK

配置完成后，先启动Zookeeper和Kafka服务，然后调用EFAK的bin目录下的ke.sh脚本启动服务

```
[oper@worker1 bin]$ ./ke.sh start
-- 日志很长，看到以下内容表示服务启动成功
[2023-06-28 16:09:43] INFO: [Job done!]
Welcome to
    ______    ______    ___     __ __
   / ____/   / ____/   /   |   / //_/
  / __/     / /_      / /| |  / ,<
 / /___    / __/     / ___ | / /| |
/_____/   /_/       /_/  |_|/_/ |_|
( Eagle For Apache Kafka® )

Version v3.0.2 -- Copyright 2016-2022
**********************************************************************
* EFAK Service has started success.
* Welcome, Now you can visit 'http://192.168.232.128:8048'
* Account:admin ,Password:123456
**********************************************************************
* <Usage> ke.sh [start|status|stop|restart|stats] </Usage>
* <Usage> https://www.kafka-eagle.org/ </Usage>
**********************************************************************
```

5、访问EFAK管理页面

接下来就可以访问EFAK的管理页面。http://192.168.232.128:8048。 默认的用户名是admin ， 密码是123456



关于EFAK更多的使用方式，比如EFAK服务如何集群部署等，可以参考官方文档。

三、Kraft集群

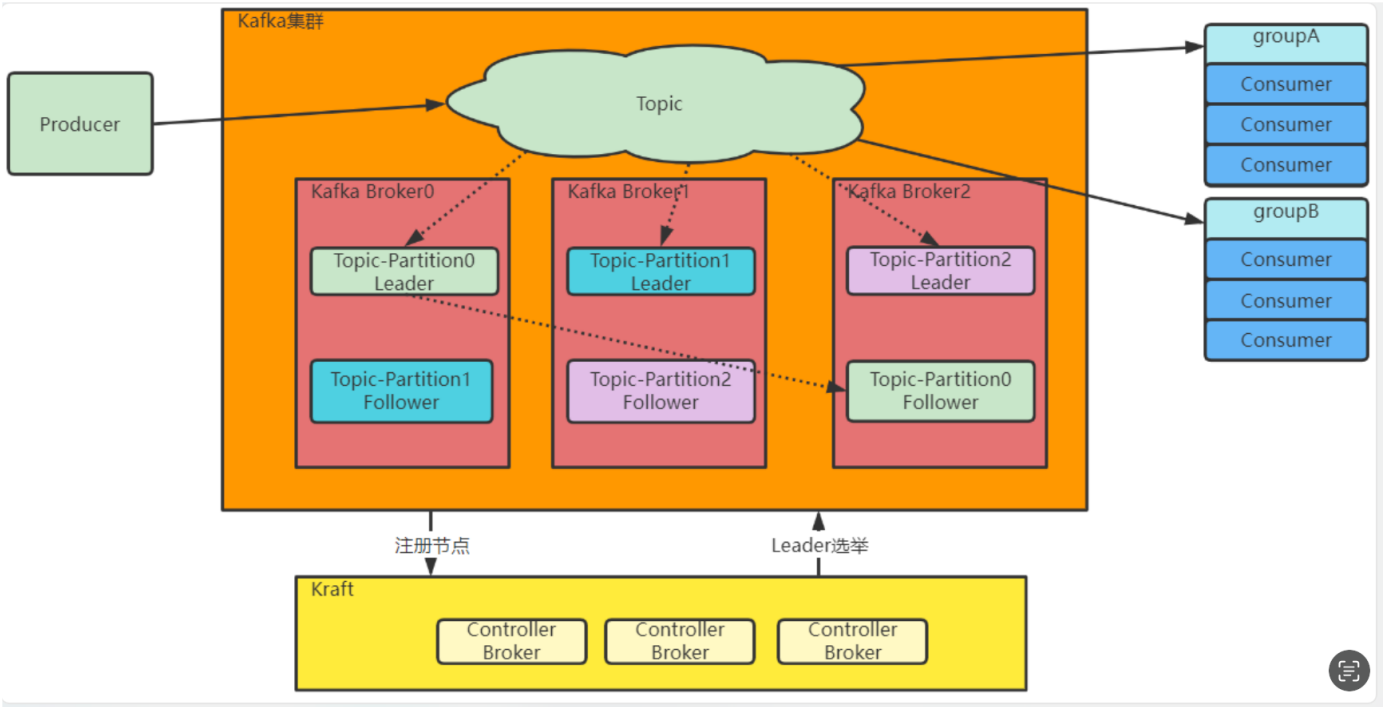
1、Kraft集群简介

Kraft是Kafka从2.8.0版本开始支持的一种新的集群架构方式。其目的主要是为了摆脱Kafka对Zookeeper的依赖。因为以往基于Zookeeper搭建的集群，增加了Kafka演进与运维的难度，逐渐开始成为Kakfa拥抱云原生的一种障碍。使用Kraft集群后，Kafka集群就不再需要依赖Zookeeper，将之前基于Zookeeper管理的集群数据，转为由Kafka集群自己管理。

> 虽然官方规划会在未来完全使用Kraft模式代替现有的Zookeeper模式，但是目前来看，Kraft集群还是没有Zookeeper集群稳定，所以现在大部分企业还是在使用Zookeeper集群。
>
> 2022年10月3日发布的3.3.1版本才开始将KRaft标注为准备用于生产。KIP-833: Mark KRaft as Production Ready。 这离大规模使用还有比较长的距离。
>
> 实际上，Kafka摆脱Zookeeper是一个很长的过程。在之前的版本迭代过程中，Kafka就已经在逐步减少Zookeeper中的数据。在Kafka的bin目录下的大量脚本，早期都是要指定zookeeper地址，后续长期版本更迭过程中，逐步改为通过--bootstrap-server参数指定Kafka服务地址。到目前版本，基本所有脚本都已经抛弃了--zookeeper参数了。

传统的Kafka集群，会将每个节点的状态信息统一保存在Zookeeper中，并通过Zookeeper动态选举产生一个Controller节点，通过Controller节点来管理Kafka集群，比如触发Partition的选举。而在Kraft集群中，会固定配置几台Broker节点来共同担任Controller的角色，各组Partition的Leader节点就会由这些Controller选举产生。原本保存在Zookeeper中的元数据也转而保存到Controller节点中。

> Raft协议是目前进行去中心化集群管理的一种常见算法，类似于之前的Paxos协议，是一种基于多数同意，从而产生集群共识的分布式算法。Kraft则是Kafka基于Raft协议进行的定制算法。



新的Kraft集群相比传统基于Zookeeper的集群，有一些很明显的好处：

- Kafka可以不依赖于外部框架独立运行。这样减少Zookeeper性能抖动对Kafka集群性能的影响，同时Kafka产品的版本迭代也更自由。
- Controller不再由Zookeeper动态选举产生，而是由配置文件进行固定。这样比较适合配合一些高可用工具来保持集群的稳定性。
- Zookeeper的产品特性决定了他不适合存储大量的数据，这对Kafka的集群规模(确切的说应该是Partition规模)是极大的限制。摆脱Zookeeper后，集群扩展时元数据的读写能力得到增强。

不过，由于分布式算法的复杂性。Kraft集群和同样基于Raft协议定制的RocketMQ的Dledger集群一样，都还在不太稳定，在真实企业开发中，用得相对还是比较少。

2、配置Kraft集群

在Kafka的config目录下，提供了一个kraft的文件夹，在这里面就是Kraft协议的参考配置文件。在这个文件夹中有三个配置文件，broker.properties,controller.properties,server.properties，分别给出了Kraft中三种不同角色的示例配置。

- broker.properties: 数据节点
- controller.properties: Controller控制节点
- server.properties: 即可以是数据节点，又可以是Controller控制节点。

这里同样列出几个比较关键的配置项，按照自己的环境进行定制即可。

```
#配置当前节点的角色。Controller相当于Zookeeper的功能，负责集群管理。Broker提供具体的消息转发服务。
process.roles=broker,controller
#配置当前节点的id。与普通集群一样，要求集群内每个节点的ID不能重复。
node.id=1
#配置集群的投票节点。其中@前面的是节点的id，后面是节点的地址和端口，这个端口跟客户端访问的端口是不一样的。通常将集群内的所有Controllor节点都配置进去。
controller.quorum.voters=1@worker1:9093,2@worker2:9093,3@worker3:9093
#Broker对客户端暴露的服务地址。基于PLAINTEXT协议。
advertised.listeners=PLAINTEXT://worker1:9092
#Controller服务协议的别名。默认就是CONTROLLER
controller.listener.names=CONTROLLER
#配置监听服务。不同的服务可以绑定不同的接口。这种配置方式在端口前面是省略了一个主机IP的，主机IP默认是使用的
java.net.InetAddress.getCanonicalHostName()
listeners=PLAINTEXT://:9092,CONTROLLER://:9093
#数据文件地址。默认配置在/tmp目录下。
log.dirs=/app/kafka/kraft-log
#topic默认的partition分区数。
num.partitions=2
```

> controller.quorum.voters 表示进行选举的Controller节点。@符号前面的表示节点ID，需要与node.id对应。后面的表示节点的协议地址。
>
> 在配合Kraft集群时，如果一个节点只是broker，也就是不参与投票，那么他的node.id就不能包含在controller.quorum.voters包含的节点ID中。

将配置文件分发，并修改每个服务器上的node.id属性和advertised.listeners属性。

由于Kafka的Kraft集群对数据格式有另外的要求，所以在启动Kraft集群前，还需要对日志目录进行格式化。

```
[root@192-168-65-112 kafka_2.13-3.8.0]$ bin/kafka-storage.sh random-uuid
j8XGPOrcR_yX4F7ospFkTA
[root@192-168-65-112 kafka_2.13-3.8.0]$ bin/kafka-storage.sh format -t j8XGPOrcR_yX4F7ospFkTA -c config/kraft/server.properties
Formatting /app/kafka/kraft-log with metadata.version 3.4-IV0.
```

> -t 表示集群ID，三个服务器上可以使用同一个集群ID。

接下来就可以指定配置文件，启动Kafka的服务了。 例如，在Worker1上，启动Broker和Controller服务。

```
[root@192-168-65-112 kafka_2.13-3.8.0]$ bin/kafka-server-start.sh -daemon config/kraft/server.properties
[root@192-168-65-112 kafka_2.13-3.8.0]$ jps
10993 Jps
10973 Kafka
```

等三个服务都启动完成后，就可以像普通集群一样去创建Topic，并维护Topic的信息了。

四、Kafka与流式计算

Kafka还有一个很大的用处，是作为流式计算的数据源。

1、批量计算与流式计算

在针对大量数据进行计算时，通常有两种计算方式，批量计算和流式计算。

批量计算通常针对静态数据。也就是每次拿一批完整的数据进行计算。例如通过SQL语句，从数据库中查出一批完整的数据，进行计算。这就是典型的批量计算。还有我们之前使用Kafka的消费者时，每次都是从Kafka中poll出一批数据，然后进行计算。计算完了再拿下一批。这都是典型的批量计算。 批量计算需要关注的是系统中的全套数据，这些数据大部分是静态的。一般进行一些大型的离线计算。

而流式计算则通常指关注系统中当前传输的数据。这些数据是实时动态产生的。流式计算一般用于对实时性要求更高的计算。例如统计网站的实时PV，UV值。这些实时的值是根据每一次用户的请求动态变化的。以累加的方式计算到实时的PV，UV值上。这样计算的实时性显然比拿一批用户的访问记录再做批量计算要更好。

流式计算时一个很重要的计算分支，在处理海量数据时优势非常明显。业界有很多大型的流式计算框架，来支持这种海量数据下的流式计算场景。如Spark Streaming，Flink等。我们通常写的一些针对MQ产品的消费端程序，其实整体也更希望是这种来一条处理一条的流式计算场景。只不过通常情况下，为了尽量保证数据传输的性能，常见的MQ产品都是以小批量的形式，一次传输一批数据。这样其实数据的实时性是有所降低的。而Kafka针对这种情况，也推出了一套Stream流式计算的API。

> 其实关于流式计算的API，RocketMQ，RabbitMQ等其他MQ产品也在陆续推出。只不过，由于Kafka的数据吞吐量，处理性能最为强大，所以Kafka天生就适合作为流式计算的一个重要数据源。而围绕kafka的大数据流式计算技术生态也是最为完整的。

2、一个简单的流式计算示例

下面实现一个基于Kafka的最为基本的word count计算来接触一下流式计算。

> word count计算也就是实时统计Kafka中每个单词传递的次数。这个word count计算是流式计算中最为基础的一个实现案例，类似于Java的Hello world。

首先首先需要引入Kafka Streams的Maven依赖

```
<dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams</artifactId>
            <version>3.8.0</version>
        </dependency>
```

然后，就可以使用Kafka Streams的API，构建自己的流式计算

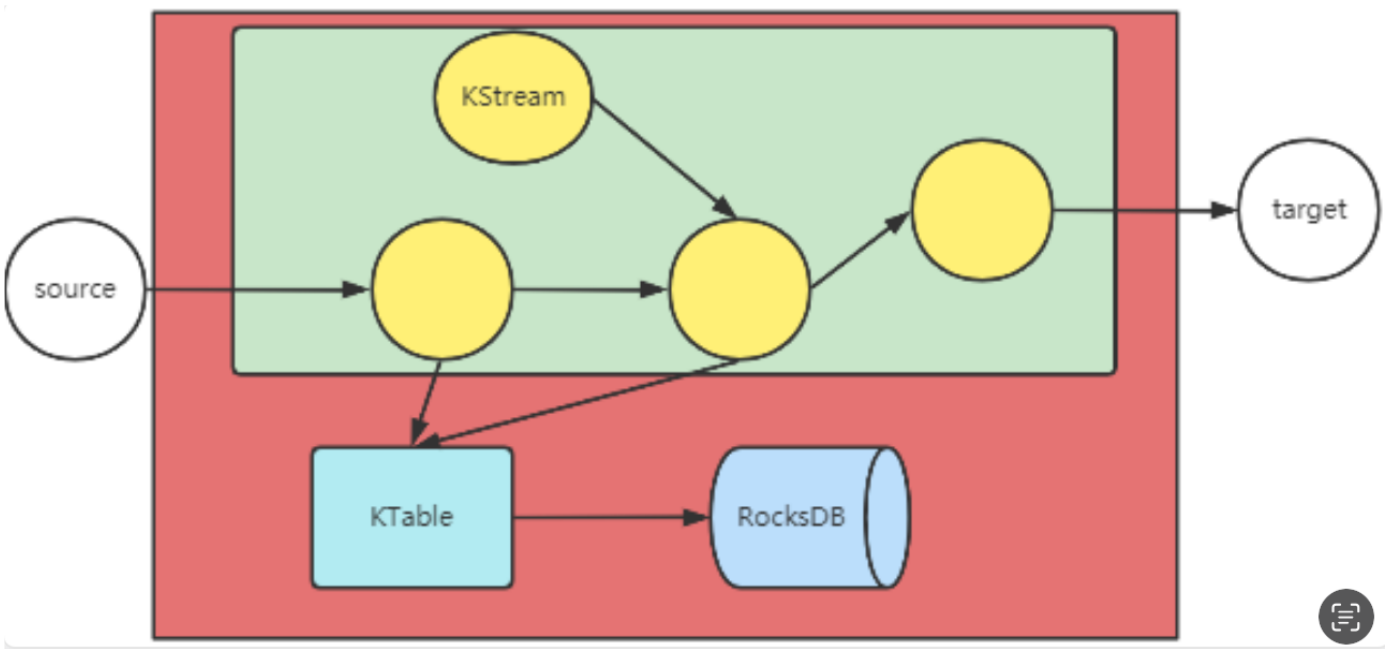
```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.Produced;

import java.util.Arrays;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;

/**
 * Author:  roy
 * Description：使用HighLevel构建Topology
 **/
public class WordCountStream {

    private static final String INPUT_TOPIC = "inputTopic";
    private static final String OUTPUT_TOPIC = "outputTopic";

    public static void main(String[] args) {
        Properties props = new Properties();
        props.putIfAbsent(StreamsConfig.APPLICATION_ID_CONFIG, "streams-wordcount");
        props.putIfAbsent(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "192.168.65.112:9092");
        props.putIfAbsent(StreamsConfig.STATESTORE_CACHE_MAX_BYTES_CONFIG, 0);
        props.putIfAbsent(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.StringSerde.class);
        props.putIfAbsent(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.StringSerde.class);

        props.putIfAbsent(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");

        KafkaStreams streams = new KafkaStreams(buildTopology(), props);
        final CountDownLatch latch = new CountDownLatch(1);

        // 优雅关闭。streams需要调用close才会清除本地缓存
        Runtime.getRuntime().addShutdownHook(new Thread("streams-wordcount-shutdown-hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (final Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }

    private static Topology buildTopology() {
        StreamsBuilder streamsBuilder = new StreamsBuilder();
        KStream<Object, String> source = streamsBuilder.stream(WordCountStream.INPUT_TOPIC);
        //flatMapValues：对每个值(如果Value是Collection，也会解析出每个值)执行一个函数，返回一个或多个值
        // 将字符串转换为小写，并使用空格分隔符分割字符串
        source.flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
                // 将每个单词作为key，进行分组
                .groupBy((key, value) -> value)
                // 对每个分组进行计数，结果为一个KTable，可以理解为一个中间结果集
                .count()
                // 转换成为KStream数据流
                .toStream()
                // 输出到指定Topic
                .to(OUTPUT_TOPIC, Produced.with(Serdes.String(),Serdes.String()));
        return streamsBuilder.build();
    }
}
```

通过这个案例，就可以统计出INPUT_TOPIC下每个单词出现的次数，并将结果输出到OUTPUT_TOPIC下。这种处理方式，就是一条一条处理消息。

3、流式计算的基本构成方式

Kafka Streams的核心构成其实就是通过构建一个包含了数据处理链路的Topology来处理数据。然后只要Source端有数据，就会经过Topology一条条处理。这个过程，就好比是把一个水管接到水龙头上。水管接好后，只要水龙头有水，就可以立即进行处理。

这中间有两个核心的概念， KStream代表的是一个数据流，而KTable则代表一个中间的结果集。

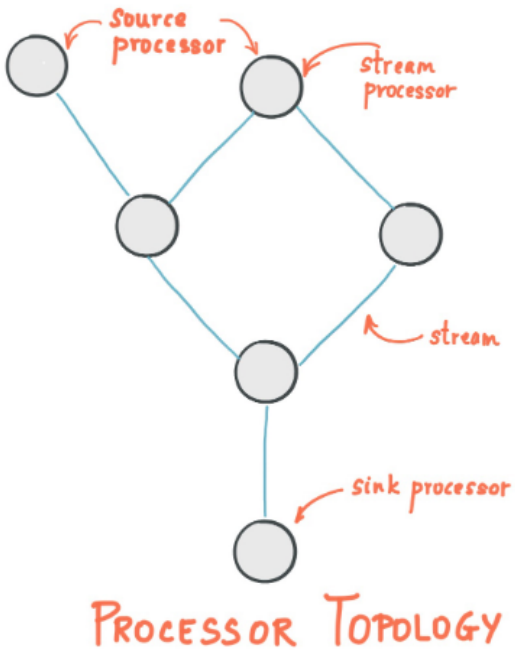


如果你觉得KStream和KTable太过抽象，那么Kafka Streams还提供了另一层LowLevel API，能够更自由的构建复杂的Topology。

```
package com.roy.kfk.stream;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.IntegerSerializer;
import org.apache.kafka.common.serialization.LongSerializer;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.common.serialization.StringSerializer;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.processor.PunctuationType;
import org.apache.kafka.streams.processor.api.Processor;
import org.apache.kafka.streams.processor.api.ProcessorContext;
import org.apache.kafka.streams.processor.api.ProcessorSupplier;
import org.apache.kafka.streams.processor.api.Record;
import org.apache.kafka.streams.state.KeyValueIterator;
import org.apache.kafka.streams.state.KeyValueStore;
import org.apache.kafka.streams.state.StoreBuilder;
import org.apache.kafka.streams.state.Stores;

import java.io.IOException;
import java.time.Duration;
import java.util.Collections;
import java.util.Properties;
import java.util.Set;
import java.util.concurrent.CountDownLatch;

/**
 * Author:  roy
 * Description：将INPUT_TOPIC中每个单词出现的次数。 使用LowLevel API构建Topology。自由度更高。
 **/
public class WordCountProcessorDemo {
    private static final String INPUT_TOPIC = "inputTopic";
    private static final String OUTPUT_TOPIC = "outputTopic";

    public static void main(String[] args) {
        Properties props = new Properties();
        props.putIfAbsent(StreamsConfig.APPLICATION_ID_CONFIG, "word");
        props.putIfAbsent(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "192.168.65.112:9092,192.168.65.170:9092,192.168.65.193:9092");
        props.putIfAbsent(StreamsConfig.STATESTORE_CACHE_MAX_BYTES_CONFIG, 0);
        props.putIfAbsent(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.StringSerde.class);
        props.putIfAbsent(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.StringSerde.class);

        props.putIfAbsent(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");

        KafkaStreams streams = new KafkaStreams(buildTopology(), props);
        final CountDownLatch latch = new CountDownLatch(1);

        // 优雅关闭。streams需要调用close才会清除本地缓存
        Runtime.getRuntime().addShutdownHook(new Thread("streams-wordcount-shutdown-hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (final Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }

    private static Topology buildTopology(){
        Topology topology = new Topology();

        topology.addSource("source",WordCountProcessorDemo.INPUT_TOPIC);

        topology.addProcessor("process", new MyWCProcessor(),"source");

        topology.addSink("sink",OUTPUT_TOPIC, new StringSerializer(),new LongSerializer(),"process");
        return topology;
    }
    //输入和输出的key与value都必须是相同类型，否则无法序列化。--只能设置一个KEY_SERDE_CLASS 和一个 VALUE_SERDE_CLASS
    // 用highlevel API就可以转。但是用lowlevel API的话，暂不知道怎么处理。
    static class MyWCProcessor implements ProcessorSupplier<String,String,String,Long> {
        @Override
        public Processor<String, String, String, Long> get() {
```

```
            return new Processor<String, String, String, Long>() {
                private KeyValueStore<String,Long> kvstore;
                @Override
                public void init(ProcessorContext<String, Long> context) {
                    context.schedule(Duration.ofSeconds(1),PunctuationType.STREAM_TIME,timestamp -> {
                        try(KeyValueIterator<String, Long> iter = kvstore.all()){
                            System.out.println("=======+timestamp+======");
                            while (iter.hasNext()){
                                KeyValue<String, Long> entry = iter.next();
                                System.out.println("["+entry.key+","+entry.value+"]");
                                context.forward(new Record<>(entry.key, entry.value, timestamp));
                            }
                        }
                    });
                    this.kvstore = context.getStateStore("counts");
                }

                @Override
                public void process(Record<String, String> record) {
                    System.out.println(">>>>>"+record.value());
                    String[] words = record.value().toLowerCase().split("\\W+");
                    for (String word : words) {
                        Long count = this.kvstore.get(word);
                        if(null == count){
                            this.kvstore.put(word,1);
                        }else{
                            this.kvstore.put(word,count+1);
                        }
                    }
                }
            };
        }

        @Override
        public Set<StoreBuilder<?>> stores() {
            return Collections.singleton(Stores.keyValueStoreBuilder(
                    Stores.inMemoryKeyValueStore("counts"),
                    Serdes.String(),
                    Serdes.Long()));
        }
    }
}
```

从这里就能看到， 实际上Kafka Streams是通过一系列处理节点Processor来构建数据的处理链条。整个过程跟工厂流水线很类似。



这里面有三种不同的Processor处理节点。

- Source Processor代表数据的起点，直接读取一个或多个Topic中的数据，往下游的Processor传递。
- 普通Processor代表数据的一个处理节点。从上游Processor读取数据，经过处理后往下游Processor传递。
- Sink Processor代表数据的终点。从上游Processor读取数据后，输出到一个或多个Topic中。

而这种处理方式，就是流式计算的一种标准处理方式。当然，Kafak Streams其实是围绕Kafka产品本身构建的一套流式数据处理框架。后续，如果需要对接更多的Source和Sink，或者进行更大规模甚至集群化的数据计算，那就需要其他更大型的流式计算框架来处理了。例如Spark Streaming，Flink等。这些大型框架功能更丰富，性能更强，但是基础的流式计算思路和Kafka Streams是一脉相承的。

五、Kafka课程总结

1、Kafka的强大之处，远不是这些客户端API所能展现的。通过我们这个课程，你能看到Kafka产品是如何把SpringBoot就已经包含了的消息驱动功能提升到能够单机承载每秒几十万TPS水平的成熟产品的。他提供的很多解决三高问题的思想，都是最有价值的财富。

2、Kafka的生态非常强大。实际上，kafka是从传统Java开发迈向大数据的最好的工具。而由大数据进一步发展出来的机器学习，AI人工智能，有多强大，这个就不用我来介绍了。所以，未来无限，好好享受吧。

> 【有道云笔记】五、Kafka功能扩展.md
> https://note.youdao.com/s/dQrFkSjS