kafka stream及interceptor

stream

演示：

1、创建一个输入topic和一个输出topic

```
bin/kafka-topics.sh --create --bootstrap-server 192.168.18.22:9092 --topic stream-plaintext-input --partitions 1 --replication-factor 1
```

```
bin/kafka-topics.sh --create --bootstrap-server 192.168.18.22:9092 --topic stream-wordcount-output --partitions 1 --replication-factor 1 --config cleanup.policy=compact
```

2、运行WordCount程序

```
#代码在源码
```

启动生产者：

```
./bin/kafka-console-producer.sh --bootstrap-server 192.168.18.22:9092 --topic stream-plaintext-input
```

启动消费者：

```
./bin/kafka-console-consumer.sh --bootstrap-server 192.168.18.22:9092 --topic stream-wordcount-output --from-beginning --formatter kafka.tools.DefaultMessageFormatter --property print.key=true --property print.value=true --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
```

1. Kafka Streams

1.1. 概述

1.1.1. Kafka Streams

Kafka Streams是一个客户端库，用于构建任务关键型实时应用程序和微服务，其中输入和/或输出数据存储在Kafka集群中。Kafka Streams结合了在客户端编写和部署标准Java和Scala应用程序的简单性以及Kafka服务器端集群技术的优势，使这些应用程序具有高度可扩展性，弹性，容错性，分布式等等。

1.1.2. Kafka Streams特点

1）功能强大

高扩展性，弹性，容错

2）轻量级

无需专门的集群

一个库，而不是框架

3）完全集成

100%的Kafka 0.10.0版本兼容

易于集成到现有的应用程序

4）实时性

毫秒级延迟

并非微批处理

窗口允许乱序数据

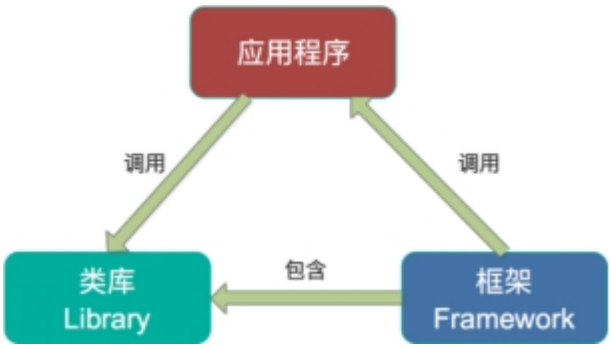
允许迟到数据

1.1.3. 为什么要有Kafka Streams

当前已经有非常多的流式处理系统，最知名且应用最多的开源流式处理系统有Spark Streaming和Apache Storm。Apache Storm发展多年，应用广泛，提供记录级别的处理能力，当前也支持SQL on Stream。而Spark Streaming基于Apache Spark，可以非常方便与图计算，SQL处理等集成，功能强大，对于熟悉其它Spark应用开发的用户而言使用门槛低。另外，目前主流的Hadoop发行版，如Cloudera和Hortonworks，都集成了Apache Storm和Apache Spark，使得部署更容易。

既然Apache Spark与Apache Storm拥用如此多的优势，那为何还需要Kafka Stream呢？主要有如下原因。

第一，Spark和Storm都是流式处理框架，而Kafka Stream提供的是一个基于Kafka的流式处理类库。框架要求开发者按照特定的方式去开发逻辑部分，供框架调用。开发者很难了解框架的具体运行方式，从而使得调试成本高，并且使用受限。而Kafka Stream作为流式处理类库，直接提供具体的类给开发者调用，整个应用的运行方式主要由开发者控制，方便使用和调试。



第二，虽然Cloudera与Hortonworks方便了Storm和Spark的部署，但是这些框架的部署仍然相对复杂。而Kafka Stream作为类库，可以非常方便的嵌入应用程序中，它对应用的打包和部署基本没有任何要求。

第三，就流式处理系统而言，基本都支持Kafka作为数据源。例如Storm具有专门的kafka-spout，而Spark也提供专门的spark-streaming-kafka模块。事实上，Kafka基本上是主流的流式处理系统的标准数据源。换言之，大部分流式系统中都已部署了Kafka，此时使用Kafka Stream的成本非常低。

第四，使用Storm或Spark Streaming时，需要为框架本身的进程预留资源，如Storm的supervisor和Spark on YARN的node manager。即使对于应用实例而言，框架本身也会占用部分资源，如Spark Streaming需要为shuffle和storage预留内存。但是Kafka作为类库不占用系统资源。

第五，由于Kafka本身提供数据持久化，因此Kafka Stream提供滚动部署和滚动升级以及重新计算的能力。

第六，由于Kafka Consumer Rebalance机制，Kafka Stream可以在线动态调整并行度。

1.2. 单词统计案例

以下是WordCountDemo示例代码的要点（为了方便阅读，使用的是java8 lambda表达式）。

步骤：

1.启动zk和kafka

> bin/zookeeper-server-start.sh config/zookeeper.properties

> bin/kafka-server-start.sh config/server.properties

\2. 准备输入主题并启动生产者

创建名为**streams-plaintext-input**的输入主题和名为**streams-wordcount-output**的输出主题：

```
> bin/kafka-topics.sh --create \
  --zookeeper localhost:2181 \
  --replication-factor 1 \
  --partitions 1 \
  --topic streams-plaintext-input
Created topic "streams-plaintext-input".
```

注意：我们创建输出主题并启用压缩，因为输出流是更改日志流

```
> bin/kafka-topics.sh --create \
  --zookeeper localhost:2181 \
  --replication-factor 1 \
  --partitions 1 \
  --topic streams-wordcount-output \
  --config cleanup.policy=compact
Created topic "streams-wordcount-output".
```

使用相同的**kafka-topics**工具描述创建的主题：

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --describe
```

\3. 启动Wordcount应用程序

```
> bin/kafka-run-class.sh org.apache.kafka.streams.examples.wordcount.WordCountDemo
```

演示应用程序将从输入主题**stream-plaintext-input**读取，对每个读取消息执行WordCount算法的计算，并将其当前结果连续写入输出主题**streams-wordcount-output**。

\4. 处理数据

开启一个生产者终端：

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
 streams-plaintext-input
all streams lead to kafka
```

开启一个消费者终端：

```
> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
  --topic streams-wordcount-output \
  --from-beginning \
  --formatter kafka.tools.DefaultMessageFormatter \
  --property print.key=true \
  --property print.value=true \
  --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
--property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
all   1
streams 1
lead  1
to   1
kafka  1
```

这里，第一列是java.lang.String格式的Kafka消息键，表示正在计数的单词，第二列是java.lang.Long格式的消息值，表示单词的最新计数。

2. Kafka producer拦截器(interceptor)

2.1. 拦截器原理

Producer拦截器(interceptor)是在Kafka 0.10版本被引入的，主要用于实现clients端的定制化控制逻辑。

对于producer而言，interceptor使得用户在消息发送前以及producer回调逻辑前有机会对消息做一些定制化需求，比如修改消息等。同时，producer允许用户指定多个interceptor按序作用于同一条消息从而形成一个拦截链(interceptor chain)。Intercetpor的实现接口是org.apache.kafka.clients.producer.ProducerInterceptor，其定义的方法包括：

（1）configure(configs)

获取配置信息和初始化数据时调用。

（2）onSend(ProducerRecord)：

该方法封装进KafkaProducer.send方法中，即它运行在用户主线程中。Producer确保在消息被序列化以及计算分区前调用该方法。用户可以在该方法中对消息做任何操作，但最好保证不要修改消息所属的topic和分区，否则会影响目标分区的计算

（3）onAcknowledgement(RecordMetadata, Exception)：

该方法会在消息被应答或消息发送失败时调用，并且通常都是在producer回调逻辑触发之前。onAcknowledgement运行在producer的IO线程中，因此不要在该方法中放入很重的逻辑，否则会拖慢producer的消息发送效率

（4）close：

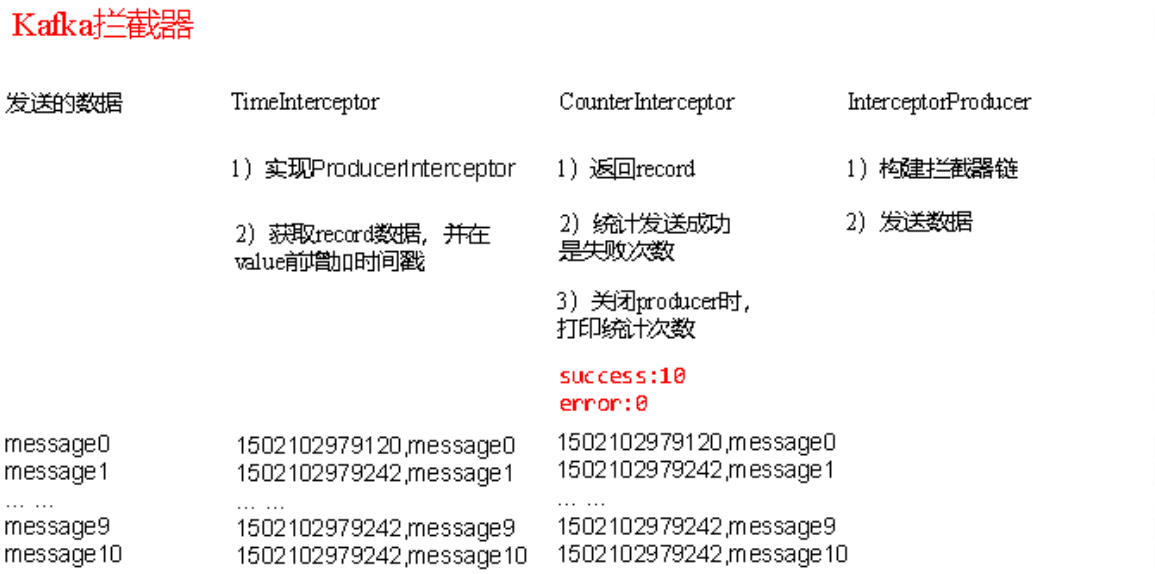
关闭interceptor，主要用于执行一些资源清理工作

如前所述，interceptor可能被运行在多个线程中，因此在具体实现时用户需要自行确保线程安全。另外倘若指定了多个interceptor，则producer将按照指定顺序调用它们，并仅仅是捕获每个interceptor可能抛出的异常记录到错误日志中而非在向上传递。这在使用过程中要特别留意。

2.2. 拦截器案例

1）需求：

实现一个简单的双interceptor组成的拦截链。第一个interceptor会在消息发送前将时间戳信息加到消息value的最前部；第二个interceptor会在消息发送后更新成功发送消息数或失败发送消息数。



2）案例实操

（1）增加时间戳拦截器

```
import java.util.Map;

import org.apache.kafka.clients.producer.ProducerInterceptor;

import org.apache.kafka.clients.producer.ProducerRecord;

import org.apache.kafka.clients.producer.RecordMetadata;
```

```
public class TimeInterceptor implements ProducerInterceptor<String, String> {


    @Override
    public void configure(Map<String, ?> configs) {


    }


    @Override
    public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {
        // 创建一个新的record，把时间戳写入消息体的最前部
        return new ProducerRecord(record.topic(), record.partition(), record.timestamp(),
record.key(),
                System.currentTimeMillis() + "," + record.value().toString());
    }


    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {


    }


    @Override
    public void close() {


    }
}
```

（2）统计发送消息成功和发送失败消息数，并在producer关闭时打印这两个计数器

```
import java.util.Map;
import org.apache.kafka.clients.producer.ProducerInterceptor;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

public class CounterInterceptor implements ProducerInterceptor<String, String>{
  private int errorCounter = 0;
  private int successCounter = 0;
```

```
    @Override
    public void configure(Map<String, ?> configs) {

    }

    @Override
    public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {
        return record;
    }

    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {
       // 统计成功和失败的次数
    if (exception == null) {
      successCounter++;
    } else {
      errorCounter++;
    }
    }

    @Override
    public void close() {
    // 保存结果
    System.out.println("Successful sent: " + successCounter);
    System.out.println("Failed sent: " + errorCounter);
    }
}
```

（3）producer主程序

```
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
```

```
import org.apache.kafka.clients.producer.ProducerRecord;

public class InterceptorProducer {

    public static void main(String[] args) throws Exception {
        // 1 设置配置信息
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("acks", "all");
        props.put("retries", 0);
        props.put("batch.size", 16384);
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        // 2 构建拦截链
        List interceptors = new ArrayList<>();
        interceptors.add("com.root.kafka.interceptor.TimeInterceptor");
interceptors.add("com.root.kafka.interceptor.CounterInterceptor");
        props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, interceptors);

        String topic = "first";
        Producer<String, String> producer = new KafkaProducer<>(props);

        // 3 发送消息
        for (int i = 0; i < 10; i++) {

          ProducerRecord<String, String> record = new ProducerRecord<>(topic, "message" + i);
          producer.send(record);
        }

        // 4 一定要关闭producer，这样才会调用interceptor的close方法
        producer.close();
    }
```

}

3）测试

（1）在kafka上启动消费者，然后运行客户端java程序。

> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic first

1501904047034,message0

1501904047225,message1

1501904047230,message2

1501904047234,message3

1501904047236,message4

1501904047240,message5

1501904047243,message6

1501904047246,message7

1501904047249,message8

1501904047252,message9

（2）观察java平台控制台输出数据如下：

Successful sent: 10

Failed sent: 0

3. Kafka 自定义分区器

在调用Kafka的Producer API时，如果没有指定分区器，那么数据将会根据默认分区器的算法均分到各个分区。然而实际的生产环境中，可能Kafka的分区数不止一个(官方建议：Kafka的分区数量应该是Broker数量的整数倍！)，所以这时需要我们自定义分区器。

3.1. 默认分区器DefaultPartitioner

默认分区器：

org.apache.kafka.clients.producer.internals.DefaultPartitioner

默认分区器获取分区：

如果消息的 key 为 null，此时 producer 会使用默认的 partitioner 分区器将消息随机分布到 topic 的可用 partition 中。

如果 key 不为 null，并且使用了默认的分区器，kafka 会使用自己的 hash 算法对 key 取 hash 值，使用 hash 值与 partition 数量取模，从而确定发送到哪个分区。注意：此时 key 相同的消息会发送到相同的分区(只要 partition 的数量不变化)。

```
 /**
  * Compute the partition for the given record.
  * 计算partition
  * @param topic The topic name
  * @param key The key to partition on (or null if no key)
  * @param keyBytes serialized key to partition on (or null if no key)
  * @param value The value to partition on or null
  * @param valueBytes serialized value to partition on or null
  * @param cluster The current cluster metadata
  */
  public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes,
Cluster cluster) {
    //获取指定topic的partitions
    List partitions = cluster.partitionsForTopic(topic);
    int numPartitions = partitions.size();
    //key=null
    if (keyBytes == null) {
      int nextValue = nextValue(topic);
      //可用分区
      List availablePartitions = cluster.availablePartitionsForTopic(topic);
      if (availablePartitions.size() > 0) {
        //消息随机分布到topic可用的partition中
        int part = Utils.toPositive(nextValue) % availablePartitions.size();
        return availablePartitions.get(part).partition();
      } else {
        // no partitions are available, give a non-available partition
        return Utils.toPositive(nextValue) % numPartitions;
      }
      //如果 key 不为 null，并且使用了默认的分区器，kafka 会使用自己的 hash 算法对 key 取 hash 值
    } else {//通过hash获取partition
      // hash the keyBytes to choose a partition
      return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
    }
  }
```

3.2. 自定义分区器

查看源码可以发现：

1、DefaultPartitioner实现了Partitioner接口

2、分区算法的实现在这个方法中：

public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster){.....}

3、如果我们需要实现自己的分区器，那么可以有2种方法

(1)新建一个包路径和DefaultPartitioner所在的路径一致，然后更改

public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster){.....}

方法体的内容，更改为我们自己的算法即可。

(2)新建一个类，实现Partitioner接口

```
public class MySamplePartitioner implements Partitioner {
  private final AtomicInteger counter = new AtomicInteger(new Random().nextInt());
  private Random random = new Random();

  //我的分区器定义
  @Override
  public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {
    List partitioners = cluster.partitionsForTopic(topic);
    int numPartitions = partitioners.size();

    /**
     * 由于我们按key分区，在这里我们规定：key值不允许为null。
     * 在实际项目中，key为null的消息*，可以发送到同一个分区,或者随机分区。
     */
    int res = 1;
    if (keyBytes == null) {
      System.out.println("value is null");
      res = random.nextInt(numPartitions);
    } else {
//      System.out.println("value is " + value + "\n hashcode is " + value.hashCode());
      res = Math.abs(key.hashCode()) % numPartitions;
    }
```

```
    System.out.println("data partitions is " + res);
    return res;
  }
}
```

3.3. 分区器使用

1、如果重载默认分区器，那么不用在Producer中做修改

2、如果是实现Partitioner接口的方式，那么需要在Producer中添加一个属性：

props.put("partitioner.class","com.study.kafka.MySamplePartitioner");//我的自定义分区器

partitioner.class值为自定义分区类的完整包名，这样生产者就会选择自定义的分区策略。

说明：

1.客户端测试环境中，自定义分区类跟生产者类在一个项目中，不需要其他操作；

2.想要自定义的分区放到kafka的服务器端环境时，需要将自定义的分区类生成jar包放到kafka环境的lib下，同样配置文件中指定完整包名。

当然我们还可以在不使用默认分区和自定义分区器的情况下直接指定分区直，具体做法：

在创建ProducerRecord时，指定消息的分区值。

```
 int partition = 0;
 if(key<100){
   partition = 0;
 }else if(key<200){
   partition = 1;
 }else{
   partition = 2;
```

```
  }

  ProducerRecord<String,String> records = new ProducerRecord<String,String>
(TOPIC,partition,key,value);

  kafkaProducer.send(records);
```