

## 今天任务

- 1、kafka的log介绍
- 2、kafka的log compaction
- 3、kafka的producer api
- 4、kafka的consumer api
- 5、kafka的adminclient api
- 6、kafka的stream
- 7、kafka的stream 实战

## 教学目标

- 1、kafka的log
- 2、kafka的log compaction
- 3、kafka的api实战
- 4、kafka的拦截器
- 5、kafka自定义分区器
- 6、kafka的拦截器
- 7、kafka的stream 实战

# 第五章 kafka的log

---

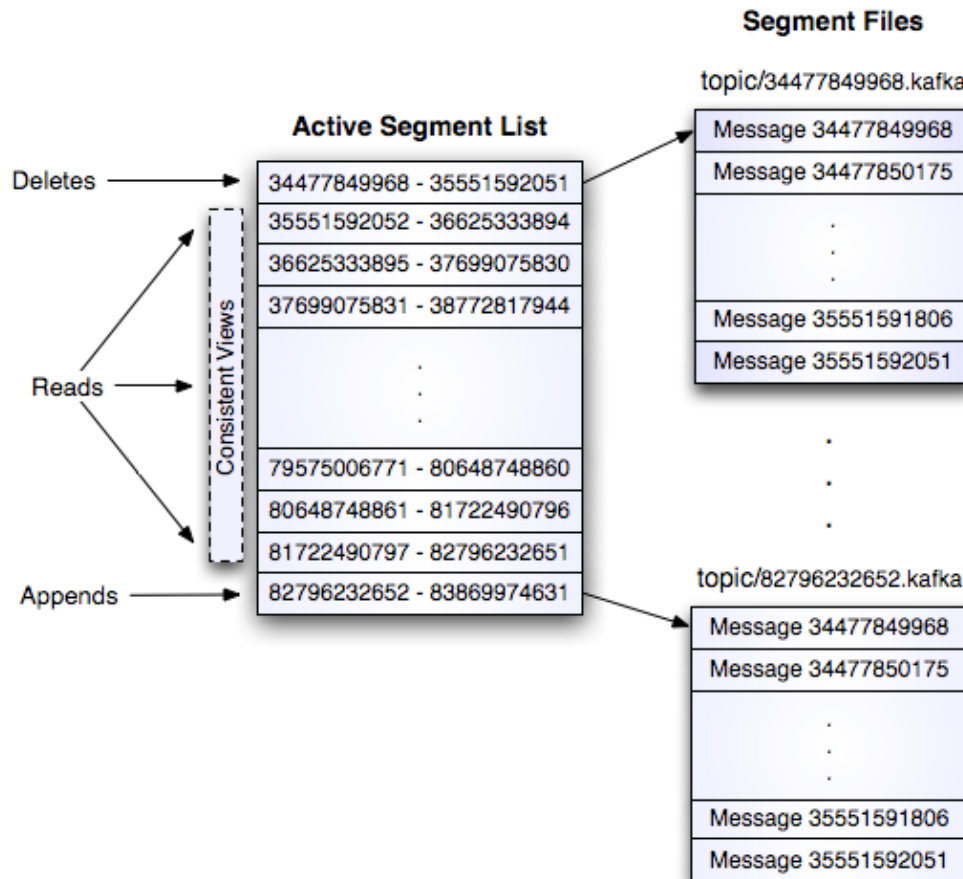
## 5.1 kafka的log

kafka的数据是以log的形式存储的,log中包含消息, 每个log文件即log entry, 也即segment。

每个log entry格式为"4个字节的数字N表示消息的长度" + "N个字节的消息内容";每个日志都有一个offset来唯一的标记一条消息,offset的值为8个字节的数字,表示此消息在此partition中所处的起始位置, 每个partition在物理存储层面,有多个log file组成(称为segment).segment就不在这儿赘述。

如下是log的实现图:

# Kafka Log Implementation



## 5.1.2 kafka的log的写

- 1、日志允许序列附加，总是附加到最后一个文件。
- 2、当该文件达到可配置的大小(比如1GB)时，就会将其刷新到一个新文件。
- 3、日志采用两个配置参数:M和S，前者给出在强制OS将文件刷新到磁盘之前要写入的消息数量(条数)，后者给出多少秒之后被强制刷新。这提供了一个持久性保证，在系统崩溃的情况下最多丢失M条消息或S秒的数据。

## 5.1.3 kafka的log读

- 1、读取的实际过程是：首先根据offset去定位数据文件中的log segment文件，然后从全局的offset值中计算指定文件offset，然后从指定文件offset读取消息。查找使用的是二分查找，每一个文件的范围都被维护到内存中。
- 2、读取是通过提供消息的64位逻辑偏移量(8字节的offset)和s字节的最大块大小来完成。
- 3、读取将返回一个迭代器包含有s字节的缓冲区，缓冲区中含有消息。S字节应该比任何单个消息都大，但是在出现异常大的消息时，可以多次重试读取，每次都应将缓冲区大小加倍，直到成功读取消息为止。
- 4、可以指定最大的消息和缓冲区大小，以使服务器拒绝的消息大于某个大小，并为客户机提供其获得完整消息所需的最大读取量。

### 5.1.4 kafka的log的删除

- 1、删除log segment即数据被删除。
- 2、日志管理器允许可插拔删除策略去选择那些最符合删除的文件来删除。
- 3、当前删除策略是修改时间超过N天以前的数据，但保留最近N GB的策略也很有用。

### 5.1.5 kafka的log的保障

- 1、日志提供了一个配置参数M，该参数控制在强制刷新磁盘之前写入的消息的最大数量(M条)。
- 2、启动日志恢复去处理最近消息在总消息中是否有效，使用crc32来校验，如果消息长度和offset总和小于文件长度且crc32和存储的消息能匹配上，则表示有效。

请注意:

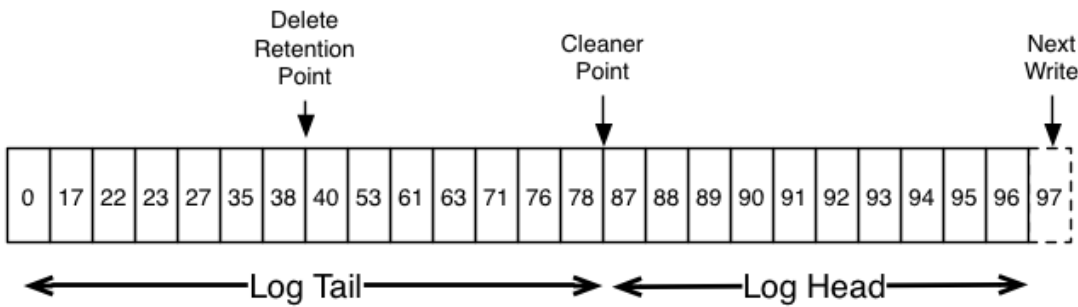
必须处理两种类型的损坏:中断(由于崩溃而丢失未写的块)和损坏(向文件添加无意义块)。

## 5.2 kafka的Log Compaction

### 5.2.1 log compaction简介

Kafka中的Log Compaction是指在默认的日志删除（Log Deletion）规则之外提供的一种清理过时数据的方式。如下图所示，Log Compaction对于有相同key的的不同value值，只保留最后一个版本。如果应用只关心key对应的最新value值，可以开启Kafka的日志清理功能，Kafka会定期将相同key的消息进行合并，只保留最新的value值。

log压缩的逻辑结构图：

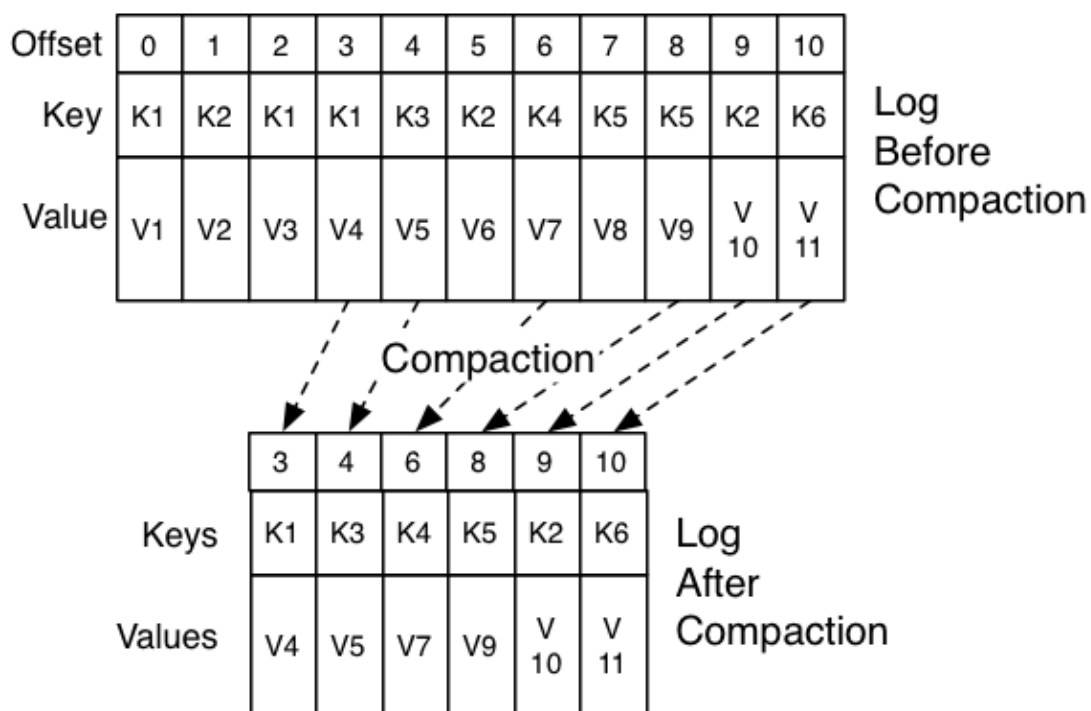


Log Head : 日志记录的头部，97代表是刚写的一条日志记录。

Log Tail : 日志记录的尾部

### 压缩日志前后

日志压缩主要是后台独立线程去定期清理log segment，将相同的key的值保留一个最新版本，压缩将不会堵塞读，并且可以进行节流，最多使用可配置的I/O吞吐量，以避免影响生产者和消费者，压缩向如下的图片一样。



### 5.2.3 log compaction保障

日志压缩保障如下：

- 1、被写在日志头部的任何一条消息对于任何一个消费者都能获取。这些消息都有一个连续的offset。主题的 `min.compaction.lag.ms` 是消息被写进来后，日志压缩的最小时间。比如，消息被写进来后，时长小于该属性的值，那么该日志不会被压缩。
- 2、消息总是被维持有序，压缩将不会对消息进行重排序，仅仅是移除。
- 3、消息的offset将绝不会被改变，这个在日志中是一个永久的标识。
- 4、消费者能够看到日志头部的所有记录的最终状态，另外，所有被删除记录的删除标记都会被看见，前提是，被提供给消费者的记录是小于主题 `delete.retention.ms` 设置的时长的所有删除标记才能被看见(默认是24小时)。换句话说：由于删除标记的删除与读取同时进行，因此，如果删除标记的延迟超过了 `delete.retention.ms`，消费者就有可能遗漏删除标记。

### 5.2.4 log compaction流程

日志cleaner(清理器)处理日志压缩，它是一个后台线程池，用于重新复制segment文件，删除其键出现在日志头部的记录。每个压缩线程的工作流程如下：

- 1、它选择具有极高比例从日志头到日志尾的日志。
- 2、为日志头的每一个key创建一个偏移量汇总。
- 3、重新负责从头到尾记录日志，删除日志中靠后出现的键。新的、干净的segment会立即交换到日志中，因此所需的额外磁盘空间只是一个额外的segment(而不是日志的完整副本)。
- 4、头部日志的汇总实质上只是一个空间紧凑的哈希表。它每个日志准确地使用24个字节。因此，若使用8GB的cleaner缓冲区，大约可以清理366 GB的头部日志(假设1k一条消息)。

### 5.2.5 log 的清空器

日志清理器默认是开启的。这将开启一个清理器线程池。对于特定主题开启略日志清理，你能添加指定的属性，该属性当主题被创建或者是主题被修改的时候所使用上。具体属性如下：

```
log.cleanup.policy=compact
```

日志清理器能配置头部日志不被压缩的最小时间。这可以通过配置compaction time lag来开启，具体属性如下：

```
log.cleaner.min.compaction.lag.ms
```

这个可以用于阻止新消息被更早的压缩，如果不设置，所有的log segment将都会被压缩，除最后一个log segment外。

如果一个segment正在被写，那么该激活的segment将不被压缩尽管它的所有消息时长都超过略压缩的最小时间。

## 第六章 kafka的实战

### 6.1 kafka的api介绍

在1.6节我们介绍过如下4类api：

- Producer API(生产者API)允许一个应用程序去推送流式记录到一个或者多个kafka的topic中。
- Consumer API(消费者API)允许一个应用程序去订阅消费一个或者多个主题，并处理生产给他们的流式记录。
- Streams API(流式API)允许应用程序作为一个流处理器，消费一个或多个主题的输入流，并生成一个或多个主题到输出流，从而有效地将输入流转换为输出流。
- Connector API(连接器API)允许构建和运行将Kafka主题连接到已经存在应用程序或数据系统的可重用生产者或消费者。例如，到关系数据库的连接器可能捕获对表的每个更改。

这儿多一种 [AdminClient](#) API，它允许管理、检查topics、broker和其他kafka对象。

kafka通过独立协议来暴露它的所有功能，该协议允许更多语言让client进行操作。java客户端是被作为kafka的一部分来维护，而其它都是kafka的一些独立开源项目。

### 6.2 Producer API

Producer API允许应用发送流数据到kafka集群中topic。要想使用Producer API，必须做如下依赖：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>edu.qf</groupId>
```

```

<artifactId>qf_kafka_demo</artifactId>
<version>1.0</version>

<!--加入kafka的相关依赖-->
<dependencies>
    <!--生产者、消费者和client需要依赖该包-->
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>1.1.1</version>
    </dependency>
    <!--stream操作需要依赖该包-->
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
        <version>1.1.1</version>
    </dependency>

    <!--adminClient需要依赖该包和client包-->
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka_2.12</artifactId>
        <version>1.1.1</version>
    </dependency>
</dependencies>

</project>

```

## 6.2.1 Producer API应用场景

- 1、网站的page View数据流实时写入
- 2、工业设备定时产生的数据需要存储到数据通道时
- 3、日志采集系统采集的数据存储到数据通道时

## 6.2.2 Producer API代码

scala版本

```

package edu.qf

import java.util.Properties
import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}

/**
 * 使用producer来循环生产10000条消息( scala版本)
 */
object ProducerAPI_Scala {
    def main(args: Array[String]): Unit = {

```

```

//实例化Properties对象，需要是java.util包中的属性
val prop = new Properties
// 指定请求的kafka集群列表。需要跟server.properties中配置的一样
//prop.put("bootstrap.servers",
"hadoop01:9092,hadoop02:9092,hadoop03:9092")
//注意：如果ip不行，则换成主机名尝试
prop.put("bootstrap.servers",
"192.167.216.111:9092,192.167.216.112:9092,192.167.216.113:9092")
// 指定响应方式
prop.put("acks", "all")
// 请求失败重试次数
prop.put("retries", "3")
// 指定key的序列化方式，key是用于存放数据对应的offset
prop.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer")
// 指定value的序列化方式
prop.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer")

// 得到KafkaProducer生产者的实例
val producer = new KafkaProducer[String, String](prop)

// 模拟一些数据并发送给kafka
for (i <- 1 to 10000) {
    val msg = s"${i}: this is kafka data"
    //生产者发送消息，消息必须被封装到KafkaProducer中，需要指定主题
    producer.send(new ProducerRecord[String, String]("test", msg))
    //每隔500ms
    Thread.sleep(500)
}
//关闭生产者实例，释放资源
producer.close()
}
}

```

先在kafka集群上启动一个消费者。

```

[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-consumer.sh --bootstrap-
server hadoop01:9092,hadoop02:9092,hadoop03:9092 --from-beginning --topic test

```

然后再运行如上的代码，验证消费者是否能消费生产者生产的消息。

## java版本代码

```

package edu.qf;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import java.util.Properties;

```

```

/**
 * 使用producer来循环生产10000条消息(java版本)
 */
public class ProducerAPI_Java {
    public static void main(String[] args) {
        //获取properties
        Properties pro = new Properties();
        //设置kafka的pro相关属性

        pro.setProperty("bootstrap.servers", "hadoop01:9092,hadoop02:9092,hadoop03:9093");

        //请求响应方式
        pro.setProperty("acks", "all");
        // 请求失败重试次数
        pro.setProperty("retries", "3");
        // 指定key的序列化方式, key是用于存放数据对应的offset

        pro.setProperty("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        // 指定value的序列化方式

        pro.setProperty("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");


        //获取kafka的producer对象
        KafkaProducer producer = new KafkaProducer(pro);

        //循环生产10000条消息
        for(int i=0; i<10000; i++){
            //组装发送的数据
            String msg = i+" kafka data of java.";
            //发送msg
            producer.send(new ProducerRecord("test",msg));
            //等待100ms
            try {
                Thread.sleep(100L);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        //关闭producer
        producer.close();
    }
}

```

运行该代码, 验证消费者是否能消费生产者生产的消息。



## 6.3 Consumer API

Consumer API允许应用去从kafka集群中读取数据流，即消费topic中的消息。

### 6.3.1 Consumer API应用场景

- 1、应用获取数据通道中数据。
- 2、流计算框架获取kafka中的数据。

### 6.3.2 Consumer API代码

scala版本

```
package edu.qf

import java.util.{Collections, Properties}

import org.apache.kafka.clients.consumer.{ConsumerRecords, KafkaConsumer}

/**
 * 使用consumer来订阅test主题，并消费里面的消息( scala版本)
 */
object ConsumerAPI_Scala {
  def main(args: Array[String]): Unit = {
    //实例化Properties对象，需要是java.util包中的属性
    val prop = new Properties
    // 指定请求的kafka集群列表。需要跟server.properties中配置的一样
    //prop.put("bootstrap.servers",
    "hadoop01:9092,hadoop02:9092,hadoop03:9092")
    //注意：如果ip不行，则换成主机名尝试
    prop.put("bootstrap.servers",
    "192.167.216.111:9092,192.167.216.112:9092,192.167.216.113:9092")
    // 指定消费者组
    prop.put("group.id", "group_test")
    // 指定offset重置策略：earliest/latest/none
    prop.put("auto.offset.reset", "earliest")
    prop.put("auto.commit.enable","false")
    // 指定key的反序列化方式，key是用于存放数据对应的offset。类也需要用反序列化的类
    prop.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer")
    // 指定value的反序列化方式。类也需要用反序列化的类
    prop.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer")

    // 得到KafkaConsumer消费者的实例
    val consumer = new KafkaConsumer[String,String](prop)

    //订阅topic
    consumer.subscribe(Collections.singletonList("test"))
  }
}
```

```

//消费数据
while(true){
    //消费；因为consumer.properties文件中auto.commit.enable设置为true，所以不能重
    复消费
    //poll一般会一直等待1000ms，直到有可用消息
    val msgs:ConsumerRecords[String,String] = consumer.poll(1000)
    //取出消费的数据放到迭代器中
    val it = msgs.iterator()
    if(it.hasNext){
        val msg = it.next() //取出一条消息
        println("partition:"+msg.partition()+" offset:"+msg.offset()
            +" key:"+msg.key()+" value:"+msg.value())
    }
}
//(一般消费者实例是一直消费，不关闭)
}
}

```

## java版本代码

```

package edu.qf;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.util.Collections;
import java.util.Iterator;
import java.util.Properties;

/**
 * 使用consumer来订阅test主题，并消费里面的消息(java版本)
 */
public class ConsumerAPI_Java {
    public static void main(String[] args) {
        //获取properties
        Properties pro = new Properties();
        //设置kafka的pro相关属性

        pro.setProperty("bootstrap.servers","hadoop01:9092,hadoop02:9092,hadoop03:909
3");

        // 指定消费者组
        pro.put("group.id", "group_test");
        // 指定消费位置: earliest/latest/none
        pro.put("auto.offset.reset", "earliest");
        pro.put("auto.commit.enable","false");
        // 指定key的反序列化方式，key是用于存放数据对应的offset。类也需要用反序列化的类
        pro.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
    }
}

```

```

// 指定value的反序列化方式。类也需要用反序列化的类
pro.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

//获取kafka的Consumer对象
KafkaConsumer consumer = new KafkaConsumer(pro);

//订阅主题
consumer.subscribe(Collections.singleton("test"));
//消费
while (true){
    ConsumerRecords msgs = consumer.poll(1000);
    Iterator<ConsumerRecord> it = msgs.iterator();
    if (it.hasNext()){
        ConsumerRecord msg = it.next();
        System.out.println("partition:"+msg.partition()+"
offset:"+msg.offset()
                        +" key:"+msg.key()+" value:"+msg.value());
    }
}
// (一般消费者实例是一直消费，不关闭)
}
}

```

运行该代码，验证消费者是否能消费生产者生产的消息。

注意:

对于已经消费的数据一般不能重复消费，但是可以指定位置消费或者通过配置属性也可以重复消费。

## 6.4 Connect API代码

待更新...

## 6.5 AdminClient API代码

Scala代码

```

import java.util.Properties

import kafka.admin.AdminUtils
import kafka.producer.{KeyedMessage, Producer, ProducerConfig}
import kafka.utils.ZkUtils
import org.I0Ittec.zkclient.ZkConnection
import org.apache.kafka.clients.consumer.KafkaConsumer

object KafkaUtilities {

```

```

def createKafkaProducer(): Producer[String, String] = {
    val props = new Properties()
    props.put("metadata.broker.list", "localhost:9092")
    props.put("serializer.class", "kafka.serializer.StringEncoder")
    // props.put("partitioner.class",
    "com.fortysevendeg.biglog.SimplePartitioner")
    // props.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer")
    // props.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer")
    props.put("producer.type", "async")
    props.put("request.required.acks", "1")

    val config = new ProducerConfig(props)
    new Producer[String, String](config)
}

def createKafkaConsumer(): KafkaConsumer[String, String] = {
    val props = new Properties()
    props.put("bootstrap.servers", "localhost:9092")
    props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer")
    props.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer")

    new KafkaConsumer[String, String](props)
}

def createTopicIntoKafka(topic: String, numPartitions: Int,
replicationFactor: Int): Unit = {
    val zookeeperConnect = "localhost:2181"
    val sessionTimeoutMs = 10 * 1000
    val connectionTimeoutMs = 8 * 1000

    val zkClient = ZkUtils.createZkClient(zookeeperConnect, sessionTimeoutMs,
connectionTimeoutMs)
    val zkUtils = new ZkUtils(zkClient, zkConnection = new
ZkConnection(zookeeperConnect), isSecure = false)
    AdminUtils.createTopic(zkUtils, topic, numPartitions, replicationFactor,
new Properties)
    zkClient.close()
}

def d(kafkaProducer: Producer[String, String], topic: String, message:
String) = {
    kafkaProducer.send(new KeyedMessage[String, String](topic, message))
}
}

```

## Java代码

```
package edu.qf;

import kafka.admin.AdminUtils;
import kafka.admin.RackAwareMode;
import kafka.utils.ZkUtils;
import java.util.Properties;

/**
 * kafka的管理客户端，可以进行主题创建、删除、添加分区等操作
 */
public class AdminClientApi {
    //定义zk的连接url
    static final String zk_url = "hadoop01:2181";
    //获取zkutil工具对象
    static final ZkUtils zk_utils = ZkUtils.apply(zk_url,6000,
        3000,false);

    //创建主题
    public static void createTopic(String topic){
        //如果主题不存在则创建
        if(!AdminUtils.topicExists(zk_utils,topic)){
            //创建主题：需主题名、分区、Properties、RackAwareMode
            AdminUtils.createTopic(zk_utils,topic,2,2,new
Properties(),RackAwareMode.Enforced$.MODULE$);
            //new RackAwareMode.Enforced$() 这种老式写法已经不行
        } else {
            System.out.println(topic+"主题已经存在");
        }
        //关闭zkutil
        zk_utils.close();
    }

    //删除主题
    public static void deleteTopic(String topic){
        //删除主题
        AdminUtils.deleteTopic(zk_utils,topic);
        //关闭zkutil
        zk_utils.close();
    }

    //测试
    public static void main(String[] args) {
        //测试创建
        //createTopic("java_topic");
    }
}
```

```

        //测试删除(先创建,再取消删除注释测试)
        deleteTopic("java_topic");
    }
}

```

测试:

```

#添加测试
[root@hadoop03 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --list --zookeeper
hadoop01:2181,hadoop02:2181,hadoop03:2181
__consumer_offsets
connect-test
java_topic    ##新增
test
test1
test3
test5
test_perf
#删除测试
[root@hadoop03 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --list --zookeeper
hadoop01:2181,hadoop02:2181,hadoop03:2181
__consumer_offsets
connect-test
test
test1
test3
test5
test_perf

```

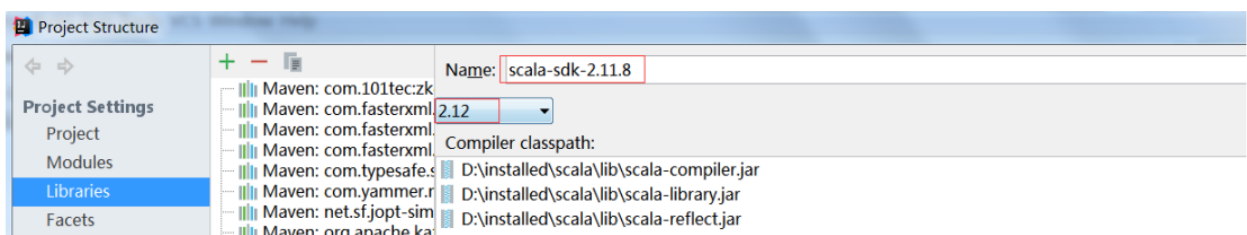
问题1:

```

Exception in thread "main" java.lang.NoSuchMethodError:
scala.Product.$init$(Lscala/Product;)V

```

解决方法:



将idea自带的2.12调整成我们安装的2.11即可。如果还不行就需要查看pom.xml中kafka依赖的scala版本:

```

<!--adminClient需要依赖该包和clients包-->
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.12</artifactId>
    <version>1.1.1</version>
</dependency>

```

kafka\_2.11

如果是kafka\_2.12,则需要修改成kafka\_2.11即可。

## 问题2:

Caused by: org.apache.zookeeper.KeeperException\$InvalidACLException:  
KeeperErrorCode = InvalidACL for /config/topics

## 解决方案:

```

//获取zkutil工具对象
static final ZkUtils zk_utils = ZkUtils.apply(zk_url, sessionTimeout: 6000,
    connectionTimeout: 3000, isZkSecurityEnabled: true);

```

false

## 6.6 kafka自定义分区

需求: 根据key的hashCode值来进行分区。如果key为空则发送到第一个分区。(根据自己的规则来发送指定消息到指定分区中去, 一般适用于生产数据时)

- 1、需要实现Partitioner ---是producer.Partitioner
- 2、核心实现partition()

## 自定义分区代码

```

package com.qianfeng.kafka;

import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;

import java.util.Map;

/**
 * kafka的自定义分区
 * 需求: 根据key的hashCode值来进行分区。如果key为空则发送到第一个分区。
 * 1、需要实现Partitioner ---是producer.Partitioner
 * 2、核心实现partition()
 */
public class MyKafkaPartition implements Partitioner {
    //分区核心实现

```

```

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

    //获取分区数
    Integer num_partitions = cluster.partitionCountForTopic(topic);
    System.out.println("分区数:"+num_partitions);
    System.out.println("key is :"+key.toString()+"key hashCode is
:"+key.hashCode());
    if (key == null || key.toString().equals("")){
        return 0; //如果key为空, 则默认返回到第一个分区
    }
    return key.toString().hashCode() % num_partitions;
}

//关闭相关资源
@Override
public void close() {
    //do nothing
}

//配置
@Override
public void configure(Map<String, ?> configs) {
    //do nothing
}
}

```

## 自定义分区的应用代码

```

package com.qianfeng.kafka;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

/**
 * 自定义分区测试
 */
public class MyKafkaPartitionTest {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers",
"hadoop01:9092,hadoop02:9092,hadoop03:9092");
        //设置key-value序列
    }
}

```



```

        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        // 设置分区器
        props.put("partitioner.class", "com.qianfeng.kafka.MyKafkaPartition");
        //获取生产者
        Producer<String, String> producer = new KafkaProducer<>(props);
        //循环生产消息
        int i = 0;
        try {
            while(i < 10) {
                ProducerRecord<String, String> record = new
ProducerRecord<String, String>("par-test",i+"",i+"value");
                //发送记录, 并获取元数据
                RecordMetadata metadata = producer.send(record).get();
                String res = "[" + record.key() + "] has been sent to
partition " + metadata.partition();
                System.out.println(res);
                Thread.sleep(500);
                i++;
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        //关闭producer
        producer.close();
    }
}

```

测试结果

分区数:3

key is :0key hashCode is :48

[0] has been sent to partition 0

分区数:3

key is :1key hashCode is :49

[1] has been sent to partition 1

分区数:3

key is :2key hashCode is :50

[2] has been sent to partition 2

分区数:3

key is :3key hashCode is :51

[3] has been sent to partition 0

分区数:3

key is :4key hashCode is :52

[4] has been sent to partition 1

分区数:3

---

Scala代码

```
class MyPartition extends Partitioner{
  override def partition(topic: String, key: Any, keyBytes: Array[Byte],
    value: Any, valueBytes: Array[Byte], cluster: Cluster): Int = {
    val num_partitions=cluster.partitionCountForTopic(topic)
    key match {
      case ""=>0
      case null=>0
      case _=>key.hashCode() % num_partitions
    }
  }

  override def close(): Unit = {
  }
}
```

```
override def configure(map: util.Map[String, _]): Unit = {  
  
}  
}
```

## 6.7 kafka自定义拦截器

kafka的拦截器分为两种，分别为：Producer Interceptor 和 Consumer Interceptor。

### 意义：

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。

producer interceptor可以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。

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器链(interceptor chain)。

### producer Intercetpor实现：

producer Intercetpor的实现接口是org.apache.kafka.clients.producer.ProducerInterceptor(消费端是org.apache.kafka.clients.consumer.ConsumerInterceptor)，producer Intercetpor需要实现的方法包括：

#### (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

### 需求：

实现由两个简单interceptor组成的拦截器链。

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异常发送消息数。

时间戳拦截器：

```
package com.qianfeng.kafka;

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

import java.util.Map;

/**
 * 时间戳拦截器
 */
public class MyTimeStampInterceptor implements
    ProducerInterceptor<String,String> {
    //封装进KafkaProducer.send方法中，即它运行在用户主线程中
    //发一条消息调用一次
    @Override
    public ProducerRecord<String, String> onSend(ProducerRecord<String,
String> record) {
        //返回一个新的record，把时间戳添加到消息体的最前部
        return new ProducerRecord(record.topic(), record.partition(),
record.timestamp(), record.key(),
            System.currentTimeMillis() + "_" + record.value().toString());
    }

    //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
    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception
exception) {
        //do nothing
    }

    //关闭interceptor时调用，主要用于执行一些资源清理工作
    @Override
    public void close() {
        //do nothing
    }

    //获取配置信息和初始化数据时调用
    @Override
    public void configure(Map<String, ?> configs) {
        //do nothing
    }
}
```

```
}
```

统计成功和异常发送消息数:

```
package com.qianfeng.kafka;

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

import java.util.Map;

/**
 * 消息发送统计拦截器： 统计成功和异常发送数据条数
 */
public class MyCounterInterceptor implements
ProducerInterceptor<String,String> {
    private int exceptions = 0;
    private int success = 0;

    //封装进KafkaProducer.send方法中，即它运行在用户主线程中
    //发一条消息调用一次
    @Override
    public ProducerRecord<String, String> onSend(ProducerRecord<String,
String> record) {
        //返回一个新的record，把时间戳添加到消息体的最前部
        return record;
    }

    //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
    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception
exception) {
        if(exception == null){
            success ++;
        } else {
            exceptions ++;
        }
    }

    //关闭interceptor时调用，主要用于执行一些资源清理工作
    @Override
    public void close() {
        //输出发送异常和成功信息
        System.out.println("成功发送数据: "+success);
        System.out.println("异常发送数据: "+exceptions);
    }

    //获取配置信息和初始化数据时调用
```

```

@Override
public void configure(Map<String, ?> configs) {
    //do nothing
}
}

```

拦截器链测试代码:

```

package com.qianfeng.kafka;

import org.apache.kafka.clients.consumer.ConsumerInterceptor;
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;

import java.util.ArrayList;
import java.util.List;
import java.util.Properties;

/**
 * 拦截器链的测试
 */
public class MyInterceptorTest {

    public static void main(String[] args) {
        //1、定义配置属性
        Properties props = new Properties();
        props.put("bootstrap.servers",
"localhost:9092,localhost:9092,localhost:9092");
        //设置key-value序列
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

        // 2 构建拦截器链
        List<String> interceptors = new ArrayList<>();
        interceptors.add("com.qianfeng.kafka.MyTimeStampInterceptor");
        interceptors.add("com.qianfeng.kafka.MyCounterInterceptor");
        props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, interceptors);
        String topic = "inter-test";
        Producer<String, String> producer = new KafkaProducer<String, String>
(props);

        // 3 循环发送消息
        for (int i = 0; i < 10; i++) {
            ProducerRecord<String, String> record = new ProducerRecord<>
(topic, "value" + i);
            producer.send(record);
        }
    }
}

```

```

    }
    // 4 关闭producer
    producer.close(); //必须关闭，不关闭将不会执行拦截器关闭方法
}
}

```

测试：

创建inter-test主题：

```

[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --create --zookeeper
hadoop01:2181,hadoop02:2181,hadoop03:2181/kafka --replication-factor 2 --
partitions 3 --topic inter-test

```

运行测试代码：

成功发送数据：10

异常发送数据：0

Process finished with exit code 0

消费inter-test主题：

```

[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-consumer.sh --bootstrap-
server hadoop01:9092,hadoop02:9092,hadoop03:9092 --from-beginning --topic
inter-test

```

```

[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-consumer.sh --bootstrap-server hadoop01:9092,hadoop02:9092,hadoop03:9092 --from-beginning --to
c inter-test
1580889669300_value1
1580889669301_value4
1580889669301_value7
1580889668911_value0
1580889669301_value3
1580889669301_value6
1580889669301_value9
1580889669301_value2
1580889669301_value5
1580889669301_value8

```

到此为止，生产端的拦截器已经实现完成。消费端的拦截器就自行编写即可。

## 第七章 kafka的Stream

### 7.1 kafka stream简介

它是一种更容易编写重要的实时应用程序和微服务。

kafak的stream是一个java客户端库，它能够构建应用程序、微服务，他们的输入和输出数据都能被存储到kafka集群中。在client端结合了简易的写、标准的java部署和scala应用程序，这更利于kafka服务端的集群技术。

## 7.2 kafka stream架构

### 7.2.1 流

流（stream）是Kafka Streams提供的最重要的抽象，它代表一个无限的、不断更新的数据集。一个流就是由一个有序的、可重放的、支持故障转移的不可变的数据记录（data record）序列，其中每个数据记录被定义成一个键值对。

### 7.2.2 流处理器

一个流处理器（stream processor）是处理拓扑中的一个节点，它代表了拓扑中的处理步骤。

一个流处理器从它所在的拓扑上游接收数据，通过Kafka Streams提供的流处理的基本方法，如map()、filter()、join()以及聚合等方法，对数据进行处理，然后将处理之后的一个或者多个输出结果发送给下游流处理器。一个拓扑中的流处理器有Source和Sink处理器两个特殊的流处理器；

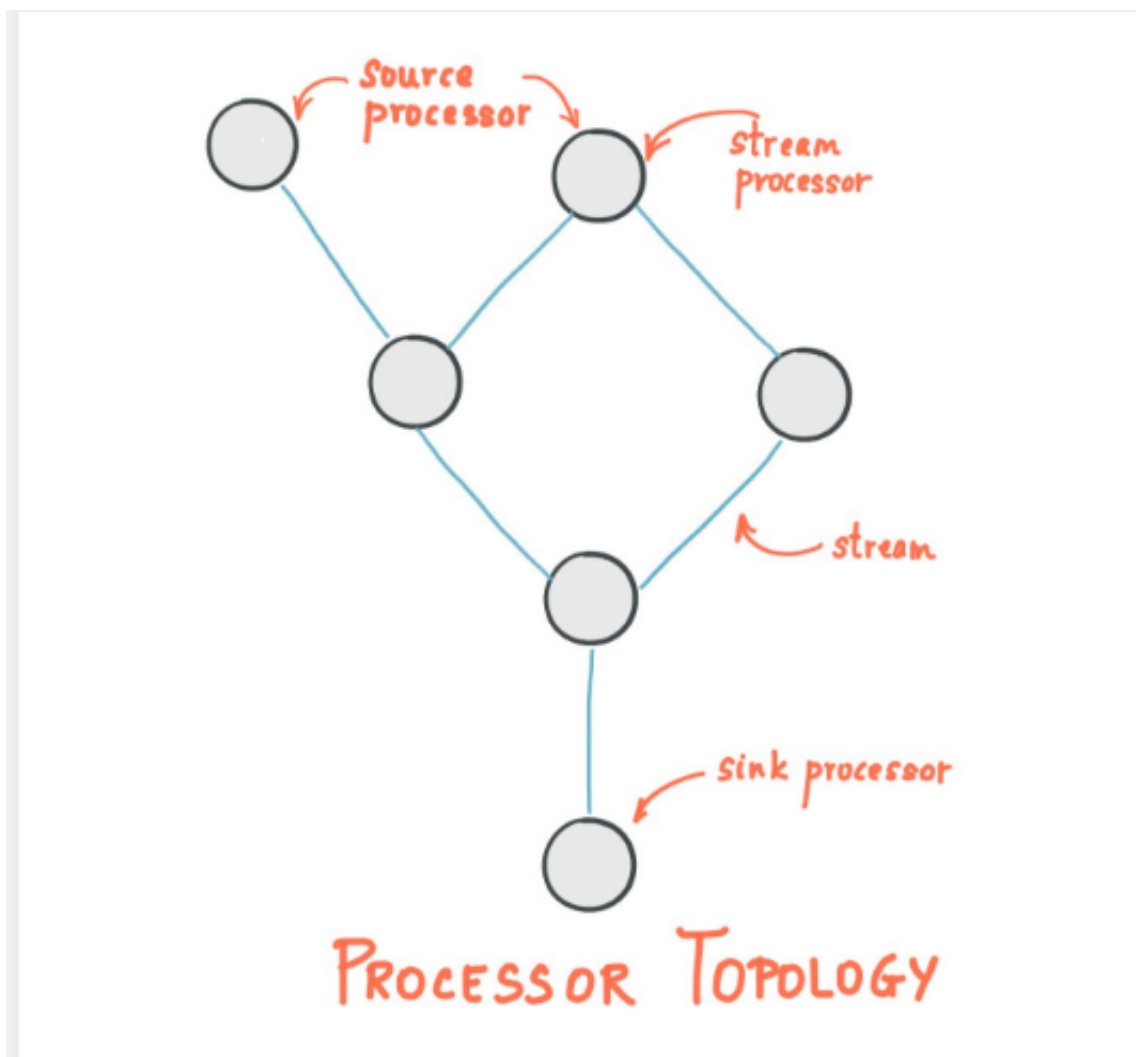
**Source处理器：**该处理器没有任何上游处理器

**Sink处理器：**该处理器没有任何下游处理器。该处理器将从上游处理器接受到的任何数据发送到指定的主题当中

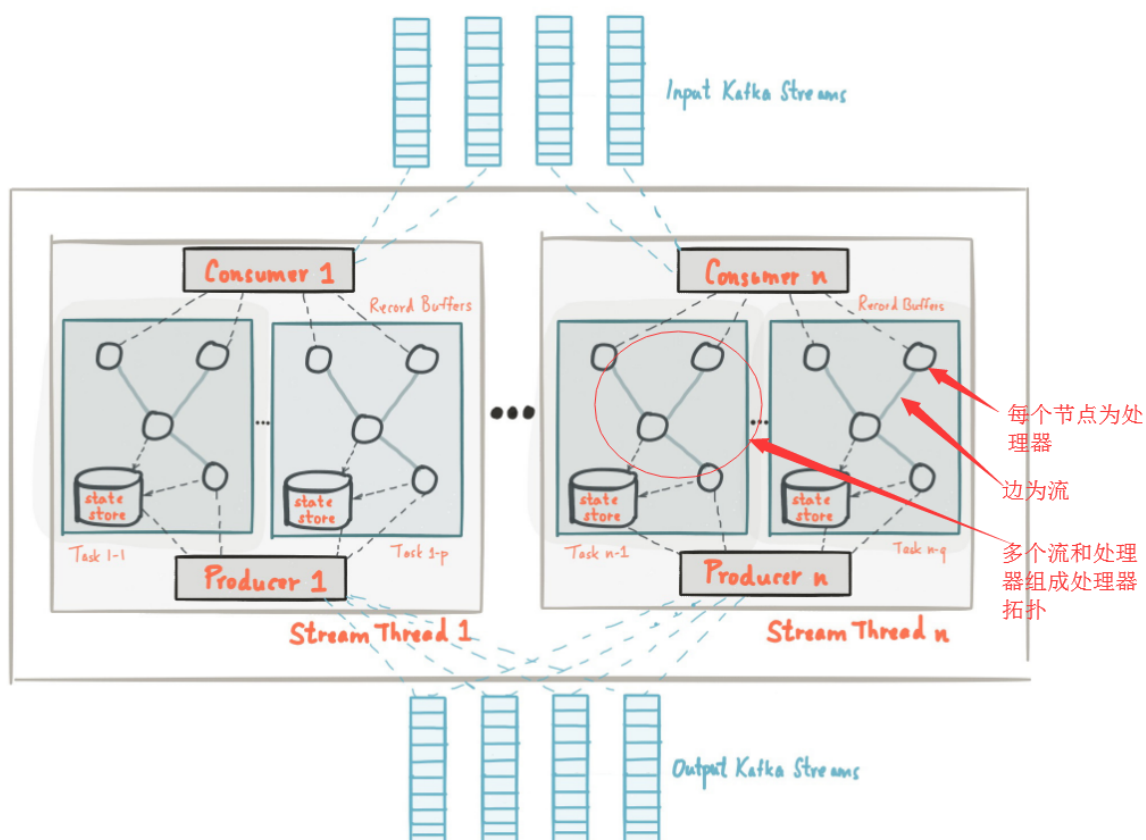
### 7.2.3 处理器拓扑

处理器拓扑（processor topology）是流处理应用程序进行数据处理的计算逻辑。一个处理器拓扑是由流处理器和相连接的流组成的有向无环图，流处理器是图的节点，流是图的边。





如下是常见kafka stream的架构图：



上图为，kafak消费者消费kafka的stream流，多为消费kafka的topic，而每个消费者经过一系列的流处理器，然后将结果数据交由生产者，然后最终将数据输出到kafak的stream流。

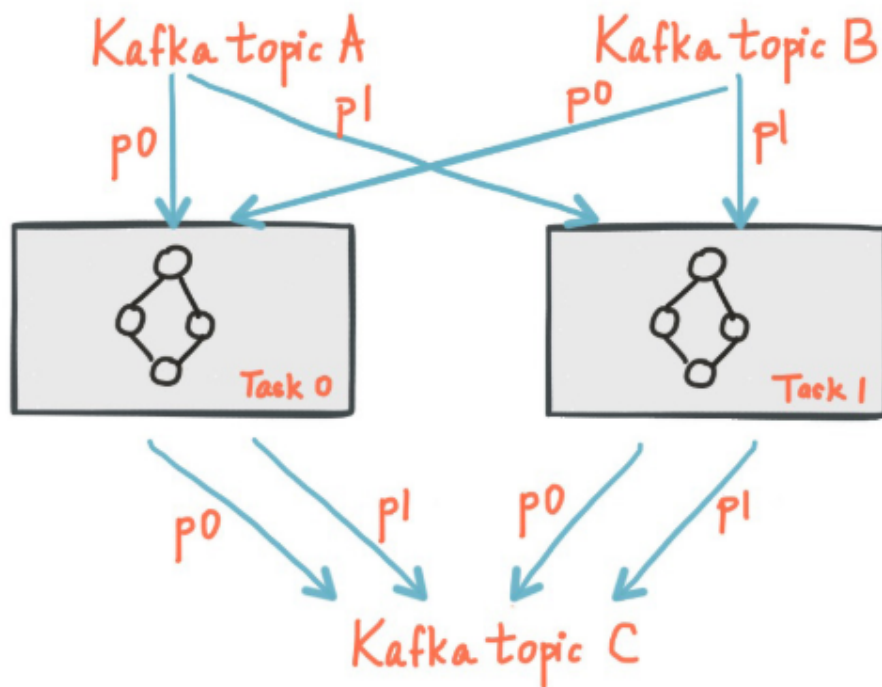
#### 7.2.4 流分区和任务

Kafka的消息层对数据进行分区以存储和传输数据。Kafka对数据进行流划分以进行处理。在这两种情况下，这种分区都支持数据本地化、弹性、可伸缩性、高性能和容错。Kafka Streams使用分区和任务的概念作为基于Kafka主题分区的并行模型的逻辑单元。在并行的背景下，Kafka流和Kafka之间有着紧密的联系：

- 每个stream partition(流分区)是数据记录的完全有序序列，并映射到Kafka主题分区。
- 流中的数据记录映射到来自该topic(主题)的Kafka消息
- 数据记录的keys(键)决定数据在Kafka和Kafka流中的划分，比如:怎样将数据路由到主题内的特定分区。

将应用程序的处理器拓扑分解为多个任务来扩展。更具体地说，Kafka流根据应用程序的输入流分区创建固定数量的任务，每个任务都从输入流分配一个分区列表(例如，kafka的topic)。分配到任务的分区不能被改变，因此每个任务都是应用程序并行性的固定单元。

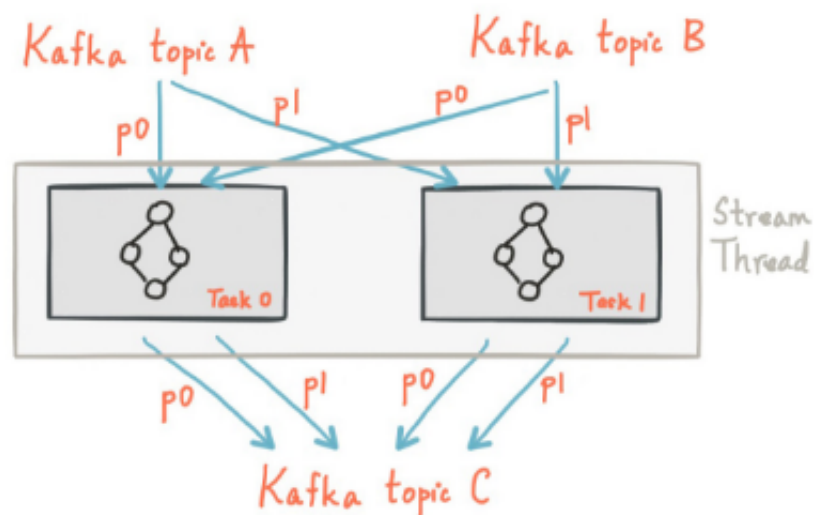
重要的是要理解Kafka流不是资源管理器，但一个流处理程序可以运行在任何能够处理流应用的地方。应用程序的多个实例可以在同一台机器上执行，也可以跨多台机器执行，库可以将任务自动分配给正在运行的应用程序实例。任务的分区分配不能被改变;如果一个应用程序实例失败，它所分配的所有任务将在其他实例上自动重新启动，并继续从相同的流分区消费。



如上图，展示两个任务，每一个任务分配输入流的一个分区，task0分为消费partition0分区，而task1任务消费partition1分区。

#### 7.2.5 线程模型

kafka stream允许用户配置线程数量用于并行处理应用程序实例。每一个线程能够执行一个或者多个任务去处理不同的拓扑。



如上图，一个流线程运行两个流任务。

开启更多流线程或应用程序实例只是相当于复制拓扑并让它处理Kafka不同分区的子集，从而能效地并行化处理。

## 7.3 kafka stream 特点

### 功能强大

稳定、高扩展性、弹性和容错

有状态和无状态的处理

基于时间时间的window、join和Aggergations

可编写标准的java应用程序

完全适合于小、中、大型应用案例

仅处理一次语义

### 轻量级

是一个库，而不是框架

无需特定的集群要求

### 完全集成

完全与kafka安全集成

易于集成到已有的应用程序

可部署于Mac、linux、windows

可部署到容器，VMS、bare metal、云

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

## 7.4 kafka stream时间

流处理的一个重要概念是时间，以及它是如何建模和集成。例如，某些操作(如windows)是基于时间边界定义。

在流里面时间一般分为：

- **事件时间 (event time)：** 事件或者记录产生的时间，即事件在源头最初创建的时间。事件时间在语义上通常要在产生的时候嵌入一个时间戳字段。  
例如：如果事件产生于汽车GPS传感器报道的位置变化，那么相关的事件时间就是GPS传感器捕捉位置变化的时间。(也就是说，这个时间通常是在流处理系统以外产生的)
- **处理时间 (processing time)：** 流处理应用程序开始处理事件的时间点（即事件进入流处理系统的时间）。这个处理时间到事件时间的间隔可能是毫秒，秒，小时，天或者更久远的时间。  
例如：假设有一个应用程序用来读取和处理来自汽车GPS传感器报告的地理位置数据，并将其呈现给车队管理仪表盘。在这里，应用程序中的处理事件可能是毫秒或者秒(例如基于Apache的Kafka和Kafka Stream流实时管道)或者小时（例如基于 Apache Hadoop或者Apache Spark的管道）。
- **摄取时间 (ingestion time)：** 数据或记录由KafkaBroker保存到 kafka topic对应分区的时间点。摄取时间类似事件时间，都是一个嵌入在数据记录中的时间戳字段。不同的是，摄取时间是由Kafka Broker附加在目标Topic上的，而不是附加在事件源上的。如果事件处理速度足够快，事件产生时间和写入Kafka的时间差就会非常小，这主要取决于具体的使用情况。因此，无法在摄取时间和事件时间之间进行二选一，两个语义是完全不同的。同时，数据还有可能没有摄取时间，比如旧版本的Kafka或者生产者不能直接生成时间戳（比如无法访问本地时钟。）

事件时间和摄取时间的选择是通过在Kafka（不是KafkaStreams）上进行配置实现的。从Kafka 0.10.X起，时间戳会被自动嵌入到Kafka的Message中，可以根据配置选择事件时间或者摄取时间。配置可以在broker或者topic中指定。Kafka Streams默认提供的时间抽取器会将这些嵌入的时间戳恢复原样。因此，应用程序的有效时间语义上依赖于这种嵌入时时间戳读取的配置。

## 7.5 kafka stream窗口

窗口能够控制如何将具有相同键的有状态操作(如group或join)记录分组到所谓的窗口。每个记录键跟踪窗口。

对流数据按照时间进行分组，也就是按照时间把流分为多个窗口（window）。窗口是流处理状态转换操作的基本条件，一个窗口相关的操作通常需要存储中间状态，根据窗口的设置旧的状态在窗口中持续时间大于窗口大小之后就会被删除；

一个窗口包括窗口大小和滑动步长两个属性：

窗口大小：一条记录在窗口中持续的时间，持续时间超过窗口大小的记录将会被删除；

滑动步长：指定了一个窗口每次相对于前一个窗口向前移动的距离。

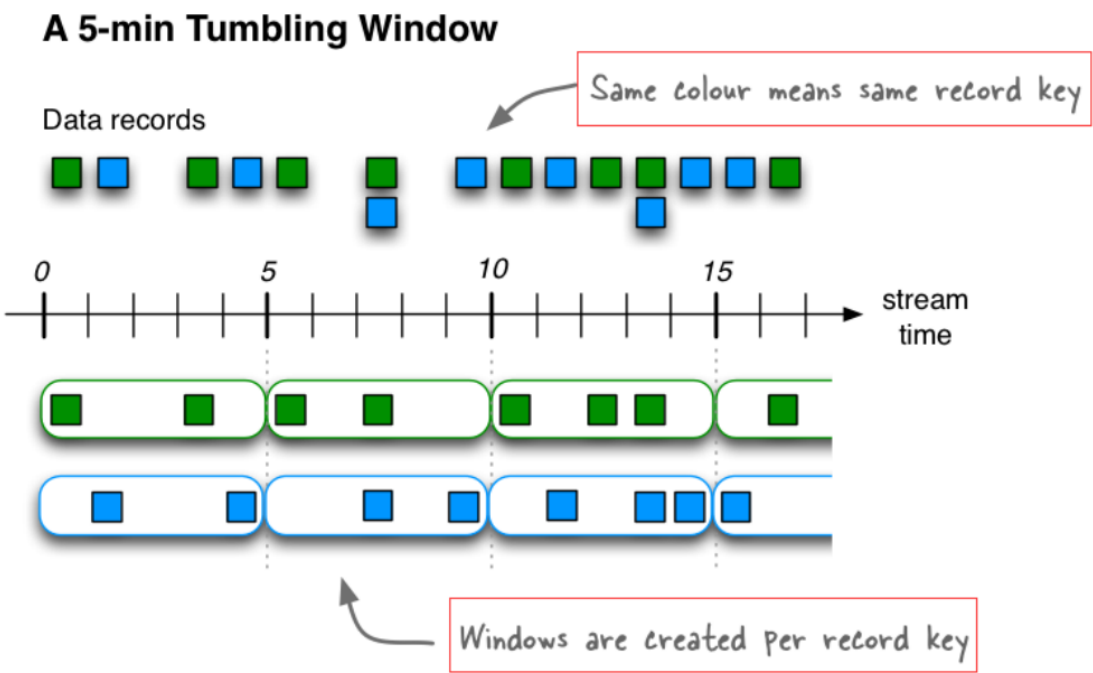
滑动步长不能超过窗口大小，如果超过窗口大小则会导致部分记录不属于任何窗口而不被处理；

根据时间窗口做聚合，是在实时计算中非常重要的功能。比如我们经常需要统计最近一段时间内的count、sum、avg等统计数据。Kafka Streams定义了四种窗口：

Window name	Behavior	Short description
<a href="#">Tumbling time window</a> (翻滚时间窗口)	基于时间	固定大小，不重叠，无间隙的窗口
<a href="#">Hopping time window</a> (跳跃时间窗口)	基于时间	固定大小,重叠窗口
<a href="#">Sliding time window</a> (滑动时间窗口)	基于时间	固定大小的重叠窗口，用于处理记录时间戳之间的差异
<a href="#">Session window</a> (会话窗口)	基于 session	动态大小，不重叠，数据驱动的窗口

7.5.1 翻滚时间窗口

翻滚时间窗口是跳跃时间窗口的一种特殊情况，就像后者一样，是基于时间间隔的窗口。他们模拟固定大小、不重叠、无间隙的窗口。滚动窗口由一个属性定义:窗口的大小。一个翻滚窗口是一个跳跃窗口，它的窗口大小等于它的前进间隔。由于翻滚窗口从不重叠，一个数据记录将只属于一个窗口。



注意:

时间默认还是按照毫秒为单位，上图标记是分钟展示，即5代表5000ms。

窗口大小是包含下边界，不包含上边界，即左开右闭，即第一个窗口是[1000,5000),第二个窗口是[5000,10000).也支持随机的数字，如[32,5032)。

翻滚时间窗口代码(不能运行):

```
import java.time.Duration;
import org.apache.kafka.streams.kstream.TimeWindows;

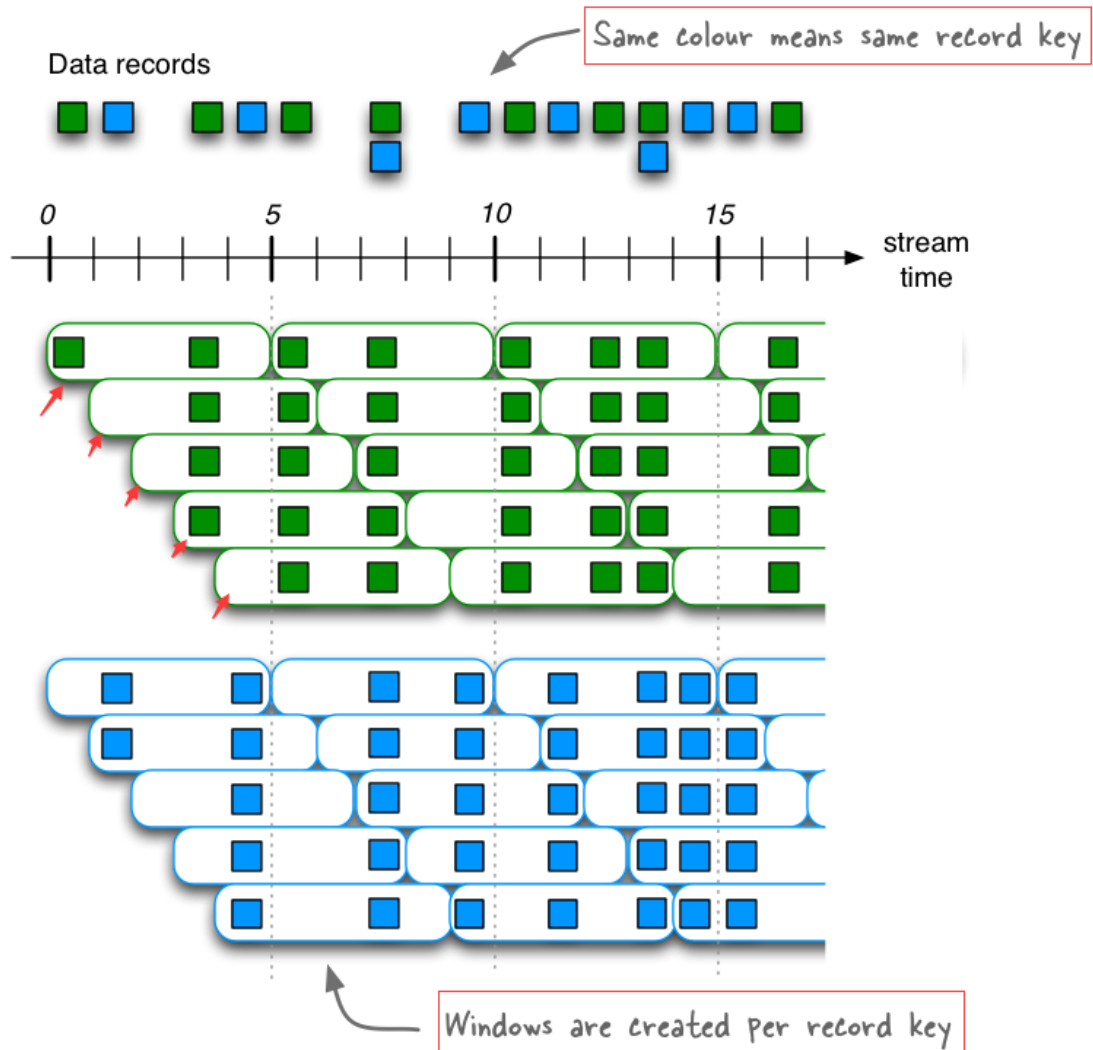
// A tumbling time window with a size of 5 minutes (and, by definition, an
implicit
// advance interval of 5 minutes).
Duration windowSizeMs = Duration.ofMinutes(5);
TimeWindows.of(windowSizeMs);

// The above is equivalent to the following code:
TimeWindows.of(windowSizeMs).advanceBy(windowSizeMs);
```

## 7.5.2 跳跃时间窗口

跳跃时间窗口是基于时间间隔的窗口。它们对固定大小(可能)重叠的窗口进行建模。一个跳跃窗口由两个属性定义:窗口的大小和它的前进间隔(又名“跳跃”)。前进间隔指定一个窗口相对于前一个窗口前进多少。例如,可以配置一个大小为5分钟、提前间隔为1分钟的跳跃窗口。由于跳跃窗口可以重叠(通常情况下确实如此),一个数据记录可能属于多个这样的窗口。

### A 5-min Hopping Window with a 1-min "hop"



上图是一个大小为5分钟、提前间隔为1分钟的跳跃窗口(绿色key相同, 蓝色key相同)。由于跳跃窗口可以重叠(通常情况下确实如此), 一个数据记录可能属于多个这样的窗口。

### 翻滚时间窗口VS跳跃时间窗口

在其他流处理工具中, 跳跃窗口有时被称为“**滑动窗口**”。kafka流遵循学术文献中的术语, 其中滑动窗口的语义不同于跳跃窗口的语义。

### 跳跃时间窗口代码(不能运行)

```
import java.time.Duration;
import org.apache.kafka.streams.kstream.TimeWindows;

// A hopping time window with a size of 5 minutes and an advance interval of 1
// minute.
// The window's name -- the string parameter -- is used to e.g. name the
// backing state store.
Duration windowSizeMs = Duration.ofMinutes(5);
Duration advanceMs = Duration.ofMinutes(1);
TimeWindows.of(windowSizeMs).advanceBy(advanceMs);
```

## 7.5.3 滑动时间窗口

滑动窗口实际上与跳跃和翻滚窗口有很大的不同。在Kafka流中, 滑动窗口**只用于join连接操作**, 可以通过JoinWindows类指定。

**滑动窗口模拟在时间轴上连续滑动的固定大小的窗口**;这里, 如果(在对称窗口的情况下)它们的时间戳的差异在窗口大小内, 那么两个数据记录被认为包含在同一个窗口中。因此, 滑动窗口不是与epoch对齐, 而是与数据记录时间戳对齐。**相对于跳跃窗口和翻滚窗口, 滑动窗口的上下窗口时间间隔边界包含在内。**

## 7.5.4 会话时间窗口

会话窗口用于将基于key的事件聚合到所谓的会话中, 这程称为会话。会话表示一段活动期间, 中间有一个确定的不活动间隙(或“空闲”)。在任何现有会话的不活动间隙内处理的任何事件将合并到现有会话中。如果一个事件落在会话间隔之外, 那么将创建一个新会话。

### 会话窗口和其它类型窗口不同:

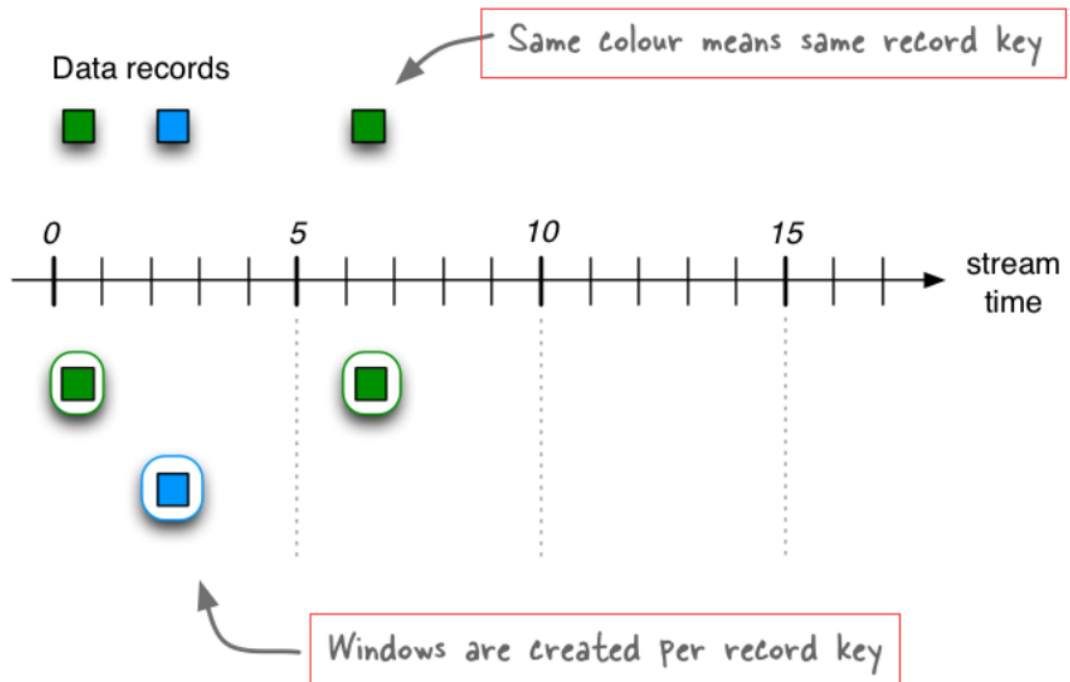
- 所有的窗口都是通过key跟踪的——例如, 不同键的窗口通常具有不同的开始和结束时间
- 会话窗口大小各不相同——即使是同一个键的窗口通常也有不同的大小

### 会话窗口应用场景:

会话窗口应用程序的主要领域是用户行为分析。基于会话的分析可以从简单的度量(例如新闻网站或社交平台上的用户访问量)到更复杂的度量(例如客户转换漏斗和事件流)。

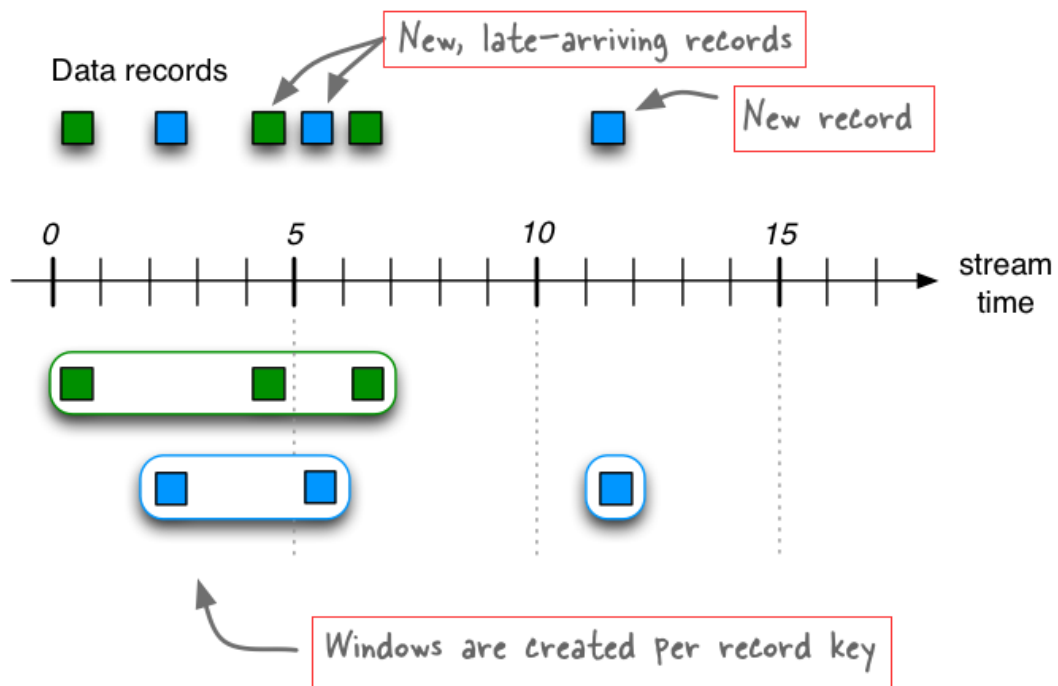


## A Session Window with a 5-min inactivity gap



时间轴上是3个记录，时间轴下是对应的3个session窗口。

## A Session Window with a 5-min inactivity gap



上图是额外增加咯3个记录，下面的session窗口又将发生变化，及在5分钟内的记录，在间隙将数据加到已有的key的session中，如果落在session窗口范围外的，将会重新创建一个新的窗口来处理。

session窗口代码



```
import java.time.Duration;
import org.apache.kafka.streams.kstream.SessionWindows;

// A session window with an inactivity gap of 5 minutes.
SessionWindows.with(Duration.ofMinutes(5));
```

## 7.6 kafka stream 的流和表

### KStream

仅仅**Kafka Streams DSL**中有KStream概念。

KStream是记录流的抽象，其中每个数据记录表示无界数据集中的独立的数据。使用table类比，数据记录在记录流里面总是被解释为一个“**插入**”。

为了说明这一点，我们假设以下两条数据记录被发送到流中：

```
("alice", 1) --> ("alice", 3)
```

如果你的流处理应用程序要对每个用户的值求和，它将为alice返回4。为什么？因为第二个数据记录不会被认为是前一个记录的更新，而是插入。

### KTable

仅仅**Kafka Streams DSL**中有KTable概念。

KTable是一个变更日志流的抽象，其中每个数据记录表示一个**更新**。更准确地说，数据记录中的值被解释为相同记录key(如果有的话)的最后一个值的“更新”(如果对应的键还不存在，则更新将被视为插入)。

为了说明这一点，我们假设以下两条数据记录被发送到流中：

```
("alice", 1) --> ("alice", 3)
```

如果你的流处理应用程序要对每个用户的值求和，它将为alice返回3。为什么？因为第二个数据记录将被认为是前一个记录的更新，而不是插入。

## 7.7 kafka stream的聚合操作

在通过groupByKey或groupBy根据key对记录进行分组，得到的表现为**KGroupedStream**或**KGroupedTable**之后，可以通过诸如reduce之类的操作对它们进行聚合。**聚合是基于键的操作**，这意味着它们总是对相同键的记录(特别是记录值)进行操作。可以对有窗口或没有窗口的数据执行聚合。

聚合操作是一种有状态的转换；

Kafka Streams定义了一个aggregate()方法，同时提供了一个Aggregator接口，我们通过该接口的apply()方法，在该方法中实现聚集函数的功能。

日志流和更新日志流都有aggregate方法，不过在进行聚合操作之前需要将Kstream或者Ktable转换为KgroupStream或者KgroupedTable，然后再调用appgregate()方法进行聚合操作；

常见聚合(暂无代码):

**Aggregate**: 无窗口的聚合, 如下是两种类型的转换。

- KGroupedStream → KTable
- KGroupedTable → KTable

**Aggregate (windowed)**: 有窗口的聚合, 如下是类型之间转换。

- KGroupedStream → KTable

**Count**: 无窗口的count, 如下是类型转换

- KGroupedStream → KTable
- KGroupedTable → KTable

**Count (windowed)**: 有窗口的count, 如下是类型转换

- KGroupedStream → KTable

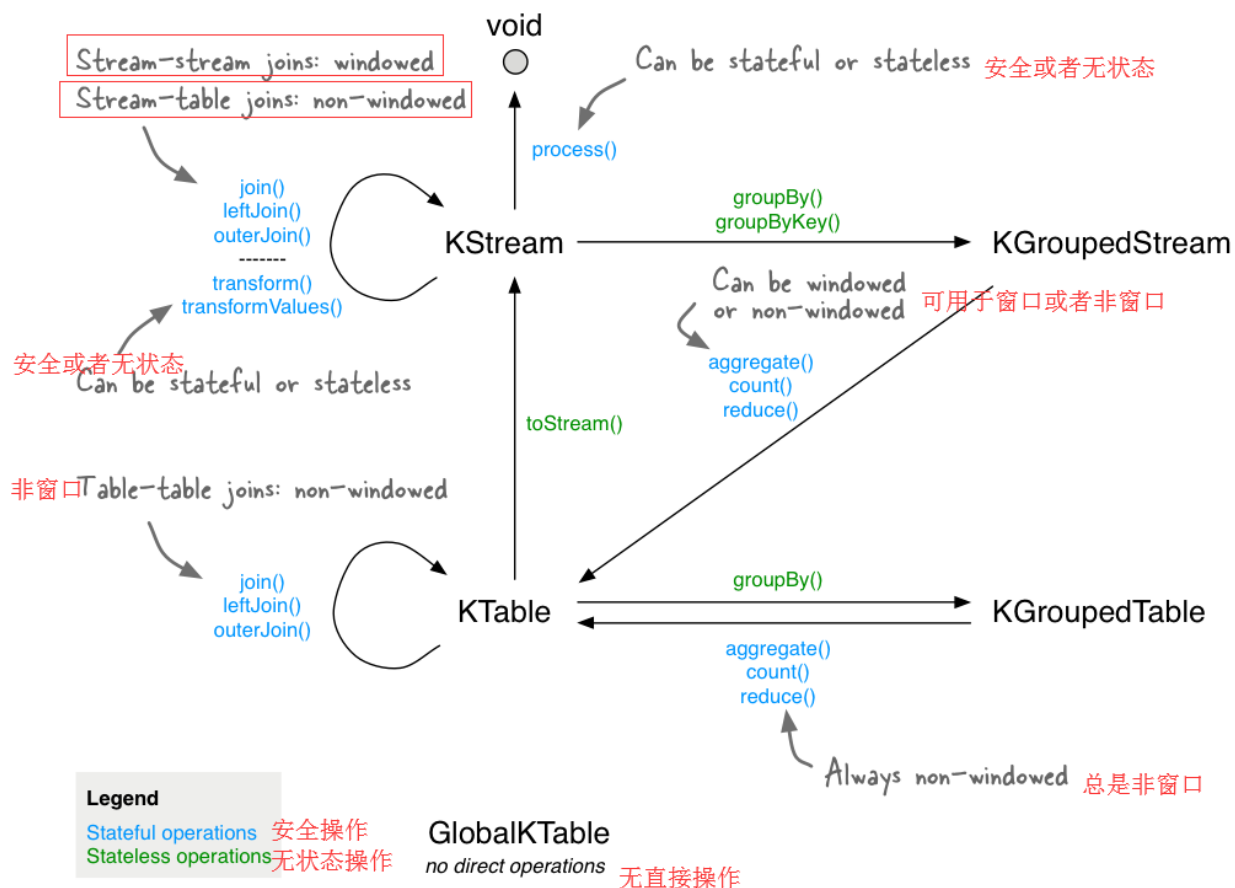
**Reduce**: 无窗口的reduce, 如下是类型转换

- KGroupedStream → KTable
- KGroupedTable → KTable

**Reduce (windowed)**: 有窗口的reduce

- KGroupedStream → KTable

状态转换的关系图:



安全不安全是看他所依赖的是否有状态存储。

参考:<http://kafka.apache.org/23/documentation/streams/developer-guide/dsl-api.html#aggregating>

## 7.8 kafka stream的状态

一些流处理应用程序不需要状态，这意味着消息的处理独立于所有其他消息的处理。然而，能够维护状态为复杂的流处理应用程序开放了许多可能性:可以加入输入流，或者对数据记录进行分组和聚合。

Kafka Streams DSL提供了许多这样的有状态操作符。

Kafka Streams提供了所谓的状态存储，流处理应用程序可以使用它来存储和查询数据。这是实现有状态操作时的一个重要功能。Kafka流中的每个任务都内嵌一个或多个状态存储，可以通过api访问这些状态存储来存储和查询处理所需的数据。这些状态存储可以是持久的键值存储、内存中的hashmap或其他方便的数据结构。Kafka流为本地状态存储提供容错和自动恢复功能。

## 7.9 kafka stream处理保障

0.11.0.0之前，Kafka只提供“至少一次”的交付保证，因此任何利用它作为后端存储的流处理系统都不能保证端到端的精确一次语义。

0.11.0.0发布以后，Kafka增加了支持，允许它的生产者以事务性和幂等性的方式向不同的主题分区发送消息，Kafka流因此通过利用这些特性增加了端到端的正好一次处理语义，**保证配置值为**`exactly_once(kafka默认值为at_least_once)`。

- At most once—Messages may be lost but are never redelivered.  
最多一次 --- 消息可能丢失，但绝不会重发。
- At least once—Messages are never lost but may be redelivered.  
至少一次 --- 消息绝不会丢失，但有可能重新发送。
- Exactly once—this is what people actually want, each message is delivered once and only once.  
正好一次 --- 这是人们真正想要的，每个消息传递一次且仅一次。

处理即消费，从消费角度考虑重复消费或者漏消费问题，其实主要是offset(位置)更新：

- **最多一次**:读取消息，然后在日志中保存它的位置，最后处理消息。在这种情况下，有可能消费者保存了位置之后，但是处理消息输出之前崩溃了。在这种情况下，接管处理的进程会在已保存的位置开始，即使该位置之前有几个消息尚未处理。这对应于“最多一次”，在消费者处理失败消息的情况下，不进行处理。
- **至少一次**: 读取消息，处理消息，最后保存消息的位置。在这种情况下，可能消费进程处理消息之后，但保存它的位置之前崩溃了。在这种情况下，当新的进程接管了它，这将接收已经被处理的前几个消息。这就符合了“至少一次”的语义。在多数情况下消息有一个主键，以便更新幂等（其任意多次执行所产生的影响均与一次执行的影响相同）。
- **正好一次**: 那么什么是“正好一次”语义（也就是你真正想要的东西）？当从Kafka主题消费并生产到另一个topic时（例如Kafka Stream），我们可以利用之前提到0.11.0.0中的生产者新事物功能。消费者的位置作为消息存储到topic中，因此我们可以与接收处理后的数据的输出topic使用相同的事务写入offset到Kafka。如果事物中断，则消费者的位置将恢复到老的值，根据其“隔离级别”，其他消费者将不会看到输出topic的生成数据，在默认的“读取未提交”隔离级别中，所有消息对消费者都是可见的，即使是被中断的事物的消息。但是在“读取提交”中，消费者将只从已提交的事物中返回消息。

## 7.10 kafka流实战

pom.xml中依赖(如果有就不需要添加):

```
<!--生产者、消费者和管理client需要依赖该包-->
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>1.1.1</version>
</dependency>
<!--stream操作需要依赖该包-->
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams</artifactId>
    <version>1.1.1</version>
</dependency>
```

### 7.10.1 Pipe案例代码

需求

1、将一个主题的数据通过管道同步到另外一个主题中。你可能会想到，怎么不写入时就写两个主题？关键是以前的代码我们不一定有或者不一定能动。

```
package edu.qf;

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

/**
 * 管道stream, 需求是将一个主题的数据通过管道到另外一个主题
 */
public class PipStream {
    public static void main(String[] args) throws Exception {
        //定义Properties对象
        Properties props = new Properties();
        //设置流应用id
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-pipe");
        //设置kafka集群的server地址
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
            "hadoop01:9092,hadoop02:9092,hadoop03:9092");
```

```

        //设置key-value的序列化类
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        //定义流处理器构建器
        final StreamsBuilder builder = new StreamsBuilder();
        //将log_topic的数据发送到view_topic
        builder.stream("log_topic").to("view_topic");
        //构建流处理器拓扑
        final Topology topology = builder.build();
        //定义kafkaStreams流(连接server地址, 去读取指定主题数据发送到指定主题)
        final KafkaStreams streams = new KafkaStreams(topology, props);
        //同步等待一个或多个线程去处理程序
        final CountDownLatch latch = new CountDownLatch(1); //计数器初始化

        //运行时钩子关闭, JVM关闭前触发
        Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-
hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown(); //计数器-1
            }
        });
        //启动流处理器, 接下来就是同步等待线程去处理程序, 异常则异常退出
        try {
            streams.start();
            //阻塞当前线程, 直到计数器为0
            latch.await();
        } catch (Throwable e) {
            System.exit(1);
        }
        //正常退出进程
        System.exit(0);
    }
}

```

## Pipe案例测试

1、先创建代码中的两个topic

```

[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --create --zookeeper
hadoop01:2181,hadoop02:2181,hadoop03:2181 --replication-factor 1 --partitions
1 --topic log_topic
Created topic "log_topic".
[root@hadoop01 kafka_2.11-1.1.1]#

```

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --create --zookeeper
hadoop01:2181,hadoop02:2181,hadoop03:2181 --replication-factor 1 --partitions
1 --topic view_topic
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-topics.sh --list --zookeeper
hadoop01:2181,hadoop02:2181,hadoop03:2181
    __consumer_offsets
connect-test
log_topic
test
test1
test3
test5
test_perf
view_topic

2、向log_topic中生产消息
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-producer.sh --broker-
list hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic log_topic
>this is pip steram
>this is pip stream

3、向view_topic中消费消息
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-consumer.sh --bootstrap-
server hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic view_topic
```

生产消息如下:

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-producer.sh --broker-list hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic log_topic
>this is pip steram
>this is pip stream!!!
```

消费消息如下:

```
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-consumer.sh --bootstrap-server hadoop01:9092,hadoop02:9092,hadoop03:9092 --from-beginning --topic view_topic
[2019-11-27 20:48:00,807] ERROR [Consumer clientId=consumer-1, groupId=console-consumer-71405] Offset commit failed on partition view_topic-0 at offset 0: The request ti
med out. (org.apache.kafka.clients.consumer.internals.ConsumerCoordinator)
[2019-11-27 20:48:26,135] ERROR [Consumer clientId=consumer-1, groupId=console-consumer-71405] Offset commit failed on partition view_topic-0 at offset 0: The request ti
med out. (org.apache.kafka.clients.consumer.internals.ConsumerCoordinator)
this is pip steram
this is pip stream!!!
```

通过如上的两个图，我们能看到，边向log\_topic生产数据，立马就能向view\_topic中消费相同数据。当停止我们的pip流处理程序后，你一边生产，另一边就不能消费数据咯。

## 7.10.2 流过滤清洗数据案例代码

```
package edu.qf;

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

```

/**
 * 过滤stream流，然后将过滤清洗后的数据发送到另外一个主题
 */
public class FilterStream {
    public static void main(String[] args) throws Exception {
        //定义Properties对象
        Properties props = new Properties();
        //设置流应用id
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-pipe");
        //设置kafka集群的server地址
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
"hadooop01:9092,hadooop02:9092,hadooop03:9092");
        //设置key-value的序列化类
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        //定义流处理器构建器
        final StreamsBuilder builder = new StreamsBuilder();
        //如下是测试，我们的value值得长度大于等于5的过滤掉，返回true的过滤出来；fase1是过
        滤掉
        builder.stream("log_topic").filter((k, v) -> v.toString().length()
<=5).to("view_topic");
        //如下的语句等价于如上的语句
        /*builder.stream("log_topic").filter((k, v) -> {
            return v.toString().length()<=5;
        }).to("view_topic");*/
        //构建流处理器拓扑
        final Topology topology = builder.build();
        //定义kafkaStreams流(连接server地址，去读取指定主题数据发送到指定主题)
        final KafkaStreams streams = new KafkaStreams(topology, props);
        //同步等待一个或多个线程去处理程序
        final CountDownLatch latch = new CountDownLatch(1);

        //运行时钩子关闭
        Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-
hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });
        //启动流处理器，接下来就是同步等待线程去处理程序，异常则异常退出
        try {
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.exit(1);
        }
    }
}

```

```

    }
    //正常退出进程
    System.exit(0);
}

}

```

运行FilterStream类。

测试发送数据：

```

[root@hadoop01 kafka_2.11-1.1.1]#
[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-producer.sh --broker-list hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic log_topic
>this is pip stream
>this is pip stream!!!
>hha
>hahhahhhhh
>hah

```

测试消费数据(value长度大于5的就被过滤略)：

```

hah

```

### 7.10.3 pv实时统计代码

需求：

不断向某一个log\_topic中生产page view数据，过滤掉非url格式数据，同时pv统计(不去重)出来发送到另外一个主题。

生产数据格式：

```

10.0.87.249 http://hadoop01.com
10.0.87.250 http://hadoop01.com
10.0.87.251 http://hadoop02.com 2312

```

代码：

```

package edu.qf;

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.KTable;
import org.apache.kafka.streams.kstream.Produced;

```



```

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

/**
 * 不断向某一个log_topic中生产page view数据, 过滤掉非url格式数据, 同时pv统计(不去重)出来
 * 发送到另外一个主题。
 */
public class PVStream {
    public static void main(String[] args) throws Exception {
        //定义Properties对象
        Properties props = new Properties();
        //设置流应用id
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-pipe");
        //设置kafka集群的server地址
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
"hadooop01:9092,hadooop02:9092,hadooop03:9092");
        //设置key-value的序列化类
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        //定义流处理器构建器
        final StreamsBuilder builder = new StreamsBuilder();
        //从log_topic的主题中消费数据
        final KStream<String,String> source = builder.stream("log_topic");
        //处理过滤
        final KTable<String, Long> counts = source.flatMapValues(value ->
Arrays.asList(value.toLowerCase(Locale.getDefault())
                .split(" "))
                .filter((k, v) -> v.startsWith("http://"))
                .groupBy((key, value) -> value)
                .count());
        /* //如下是简单的wordcount的例子的写法
        final KTable<String,Long> counts = source
                .flatMapValues(value ->
Arrays.asList(value.toLowerCase(Locale.getDefault()).split(" ")))
                .groupBy((key, value) -> value)
                .count();*/
        //counts.print(); //count后可以打印出来查看信息
        //如下是循环打印, 组装成我们想要的数据
        counts.toStream().foreach((k,v)->{
            System.out.println(k.toString()+":"+v.toString());
        });
        //转换counts为流, 并写回kafka的view_topic主题
        //counts.toStream().to("view_topic1", Produced.with(Serdes.String(),
Serdes.Long()));
    }
}

```

```

//构建流处理器拓扑
final Topology topology = builder.build();
//定义kafkaStreams流(连接server地址, 去读取指定主题数据发送到指定主题)
final KafkaStreams streams = new KafkaStreams(topology, props);
//同步等待一个或多个线程去处理程序
final CountDownLatch latch = new CountDownLatch(1);

//运行时钩子关闭
Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-
hook") {
    @Override
    public void run() {
        streams.close();
        latch.countDown();
    }
});
//启动流处理器, 接下来就是同步等待线程去处理程序, 异常则异常退出
try {
    streams.start();
    latch.await();
} catch (Throwable e) {
    System.exit(1);
}
//正常退出进程
System.exit(0);
}
}

```

### PV案例测试:

- 1、启动PVStream类。
- 2、向log\_topic中生产数据

```

[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-producer.sh --broker-
list hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic log_topic
>10.0.87.249 http://qianfeng.com
>10.0.87.250 http://qianfeng.com
>10.0.87.251 http://kafka.com 2312

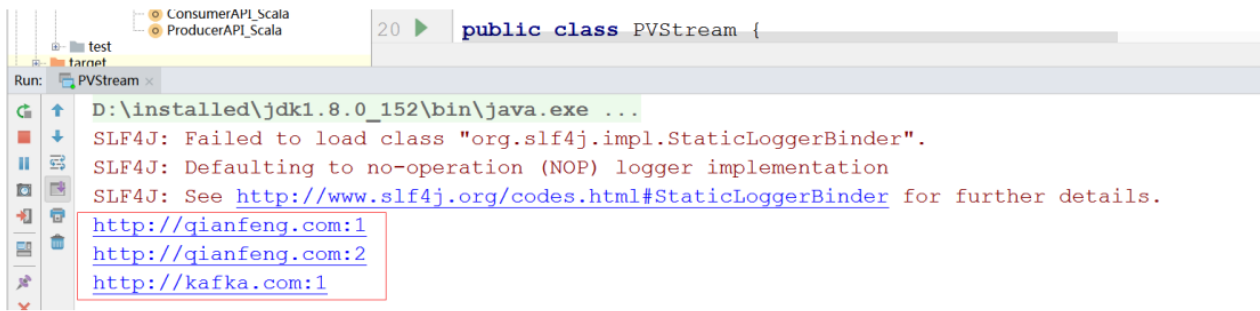
```

```

[root@hadoop01 kafka_2.11-1.1.1]# ./bin/kafka-console-producer.sh --broker-list hadoop01:9092,hadoop02:9092,hadoop03:9092 --topic log_topic
>10.0.88.249 http://qianfeng.com
>10.0.88.250 http://qianfeng.com
>10.0.88.251 http://kafka.com 2312

```

- 3、查看实时统计结果:



The screenshot shows an IDE with a project named 'test' containing two classes: 'ConsumerAPI\_Scala' and 'ProducerAPI\_Scala'. The 'PVStream' class is selected, showing a line number 20 and the code 'public class PVStream {'. Below the code editor, the 'Run' tab is active, displaying the execution command 'D:\installed\jdk1.8.0\_152\bin\java.exe ...'. The output shows SLF4J messages: 'SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".', 'SLF4J: Defaulting to no-operation (NOP) logger implementation', and 'SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.'. Three URLs are listed: 'http://qianfeng.com:1', 'http://qianfeng.com:2', and 'http://kafka.com:1', which are highlighted with a red box.

## 第八章 kafka的优化

### 8.1 kafka的配置属性优化

#### 8.1.1 enable.auto.commit

指定了消费者是否自动提交偏移量，默认值是true，为了尽量避免重复数据和数据丢失，可以把它设置为false，有自己控制合适提交偏移量，如果设置为true， 可以通过设置 auto.commit.interval.ms属性来控制提交的频率。

建议设置为false，自己控制offset。

### 8.2 kafka的常见问题

待更新...