SparkStreaming方案设计课

一 、课前准备

1. 掌握Spark知识体系
2. 掌握Kafka知识
3. 掌握SparkStreaming基础知识

二 、课堂主题

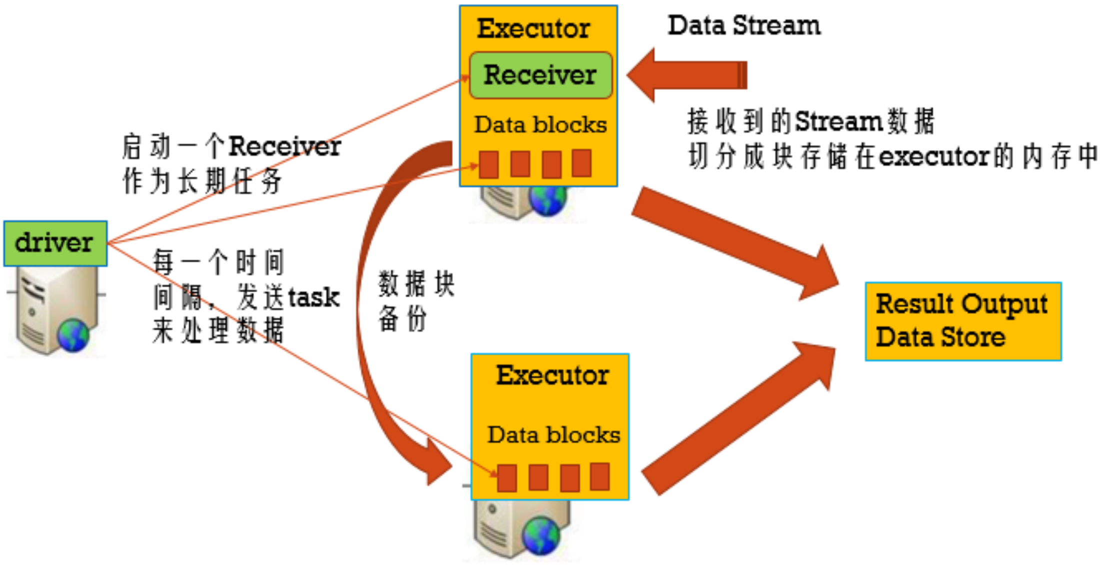
随着云计算和大数据的快速发展，在企业中大数据实时处理场景的需求越来越多。本次课程针对企业级实时处理方案进行讲解，内容包含：SparkStreaming技术的核心原理剖析，SparkStreaming项目的企业级架构设计方案，SparkStreaming实时任务的监控告警架构设计方案等。

三 、课程目标

1. 掌握SparkStreaming的容错
2. SparkStreaming和Kafka方案整合
3. SparkStreaming项目数据不丢失方案设计
4. 实时项目任务监控告警方案设计

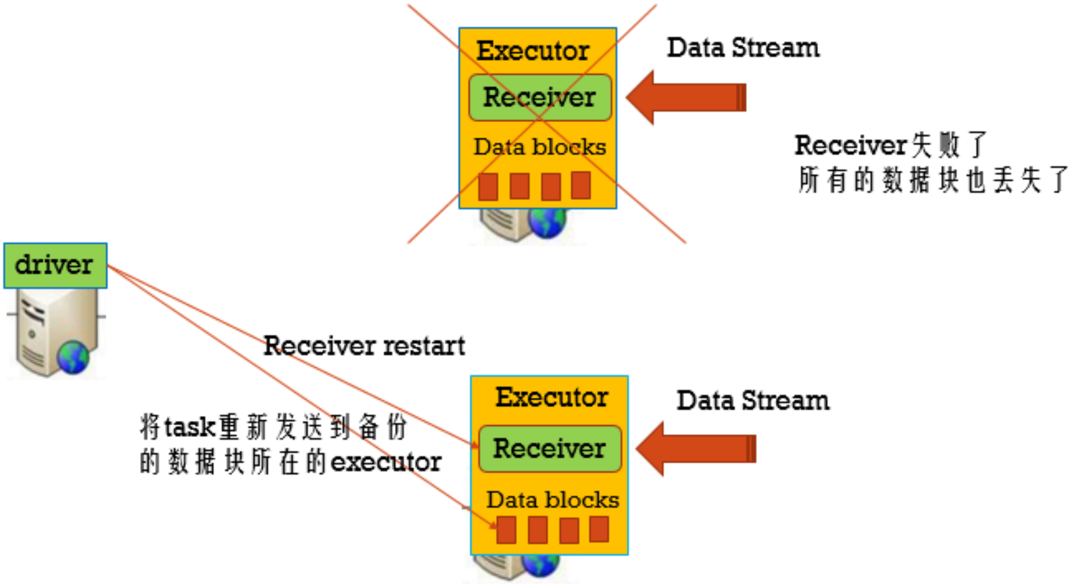
四 、知识要点

4.1 SparkStreaming运行流程回顾



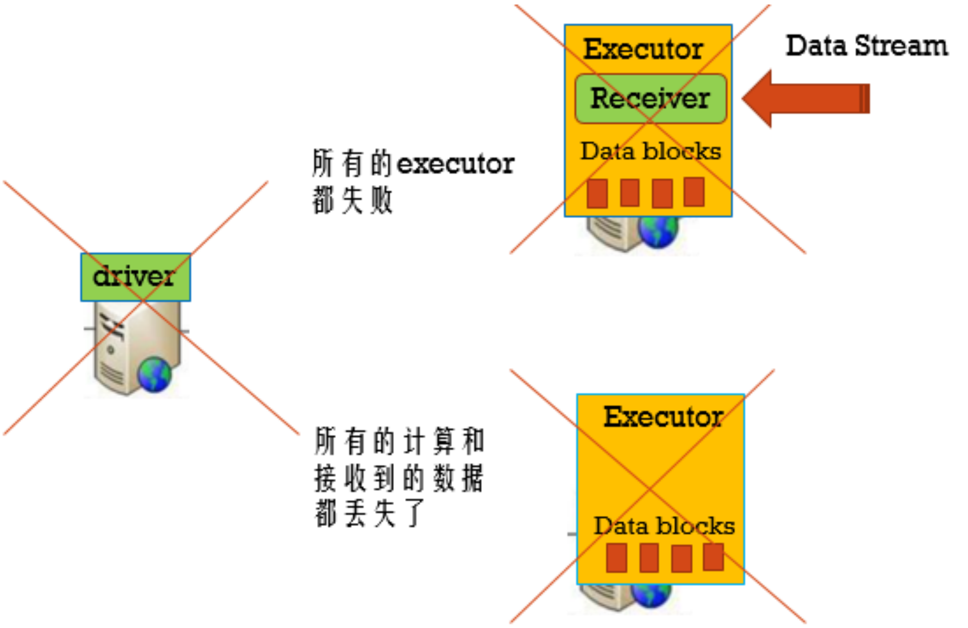
4.1 SparkStreaming容错

4.1.1 Executor失败



Tasks和Receiver自动的重启，不需要做任何的配置

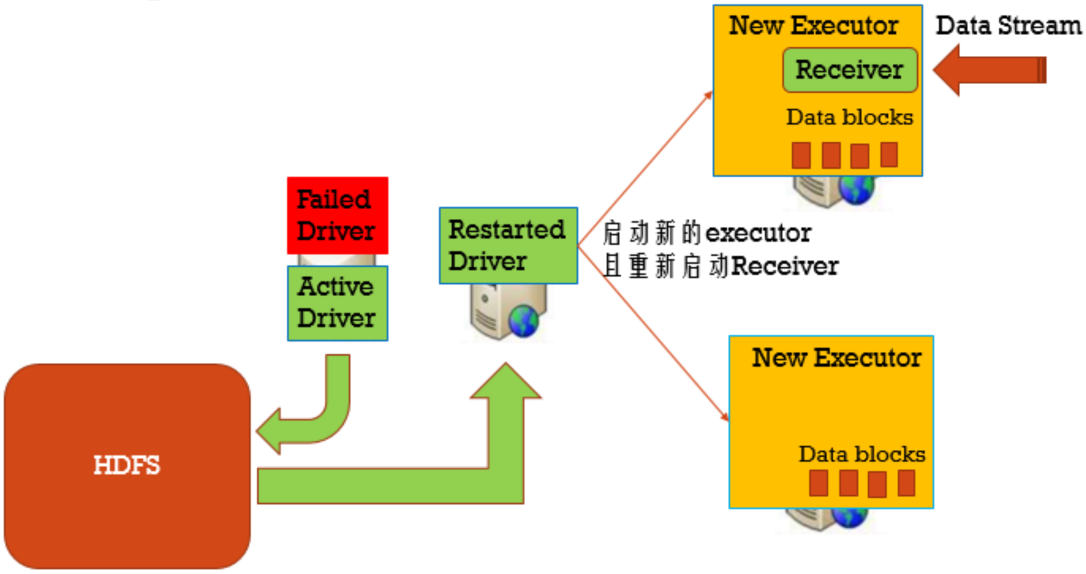
4.1.3 Driver失败(怎么容错？)



用checkpoint机制恢复失败的Driver

定期的将Driver信息写入到HDFS中。

1. checkpoint(Driver的元数据信息存储起来)
2. 自动重启



步骤一：设置自动重启Driver程序

Standalone：

在spark-submit中增加以下两个参数：

Spark的提交模式有两种：client模式，cluster模式。

--deploy-mode cluster

--supervise

Yarn：

在spark-submit中增加以下参数：

--deploy-mode cluster

在yarn配置中设置yarn.resourcemanager.am.max-attemps 重试几次？3

步骤二：设置HDFS的checkpoint目录

streamingContext.setCheckpoint(hdfsDirectory)

步骤三：代码实现

```
// Function to create and setup a new StreamingContext
def functionToCreateContext(): StreamingContext = {
  val ssc = new StreamingContext(...)   // new context
  val lines = ssc.socketTextStream(...) // create DStreams
  ...
  ssc.checkpoint(checkpointDirectory)   // set checkpoint directory
  ssc
}

// Get StreamingContext from checkpoint data or create a new one
val context = StreamingContext.getOrCreate(checkpointDirectory,
functionToCreateContext _)
```

```
// Do additional setup on context that needs to be done,
// irrespective of whether it is being started or restarted
context. ...

// Start the context
context.start()
context.awaitTermination()
```

总结：这方案如果我们升级程序，会丢数据

4.2 SparkSreaming语义

有三种语义：

1、At most once 一条记录要么被处理一次，要么没有被处理（丢数据）

2、At least once 一条记录可能被处理一次或者多次，可能会重复处理（重复消费）

3、Exactly once 一条记录只被处理一次（仅一次）

4.3 SparkStreaming与Kafka整合

SparkStreaming整合Kafka官方文档

4.3.1 方式一：Receiver-based Approach（不推荐使用）

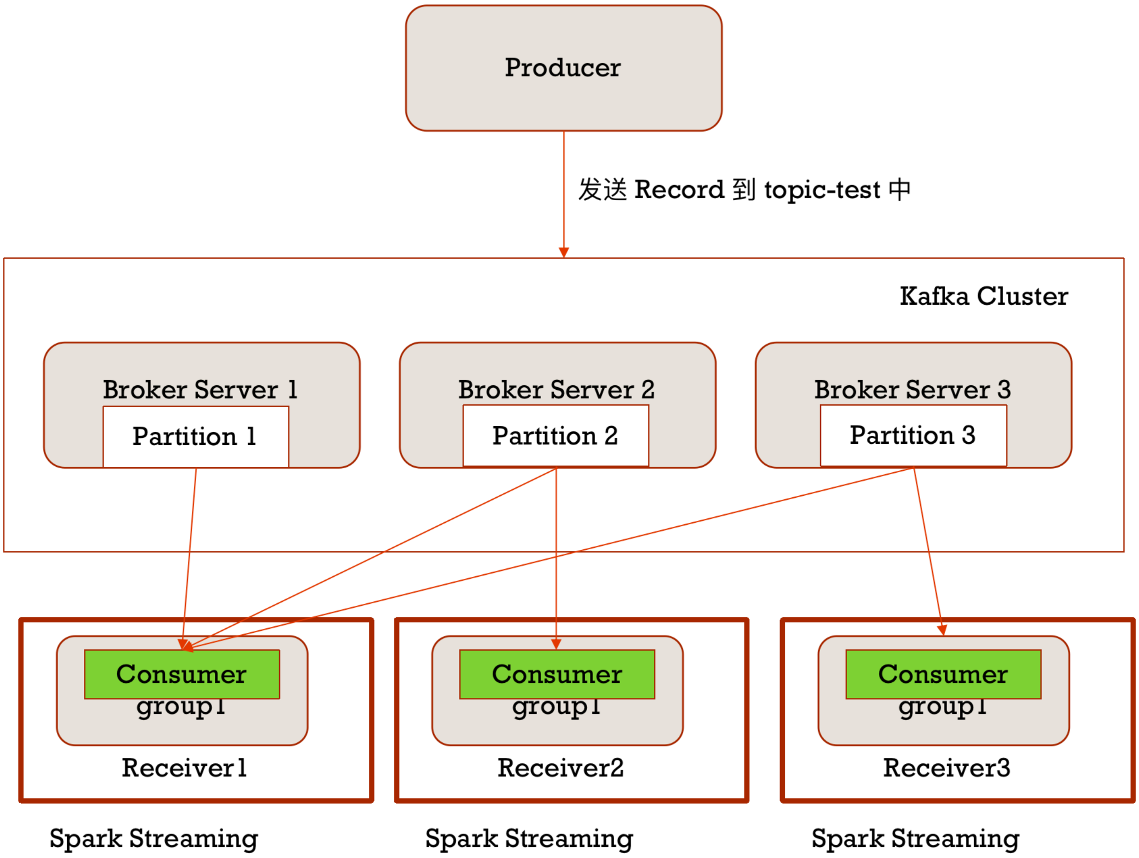
此方法使用Receiver接收数据。Receiver是使用Kafka高级消费者API实现的。与所有接收器一样，从Kafka通过Receiver接收的数据存储在Spark执行器中，然后由Spark Streaming启动的作业处理数据。但是，在默认配置下，此方法可能会在失败时丢失数据（请参阅接收器可靠性。为确保零数据丢失，必须在Spark Streaming中另外启用Write Ahead Logs（在Spark 1.2中引入）。这将同步保存所有收到的Kafka将数据写入分布式文件系统（例如HDFS）上的预写日志，以便在发生故障时可以恢复所有数据，但是性能不好。

pom.xml文件添加如下：

```
 groupId = org.apache.spark
 artifactId = spark-streaming-kafka-0-8_2.11
 version = 2.3.3
```

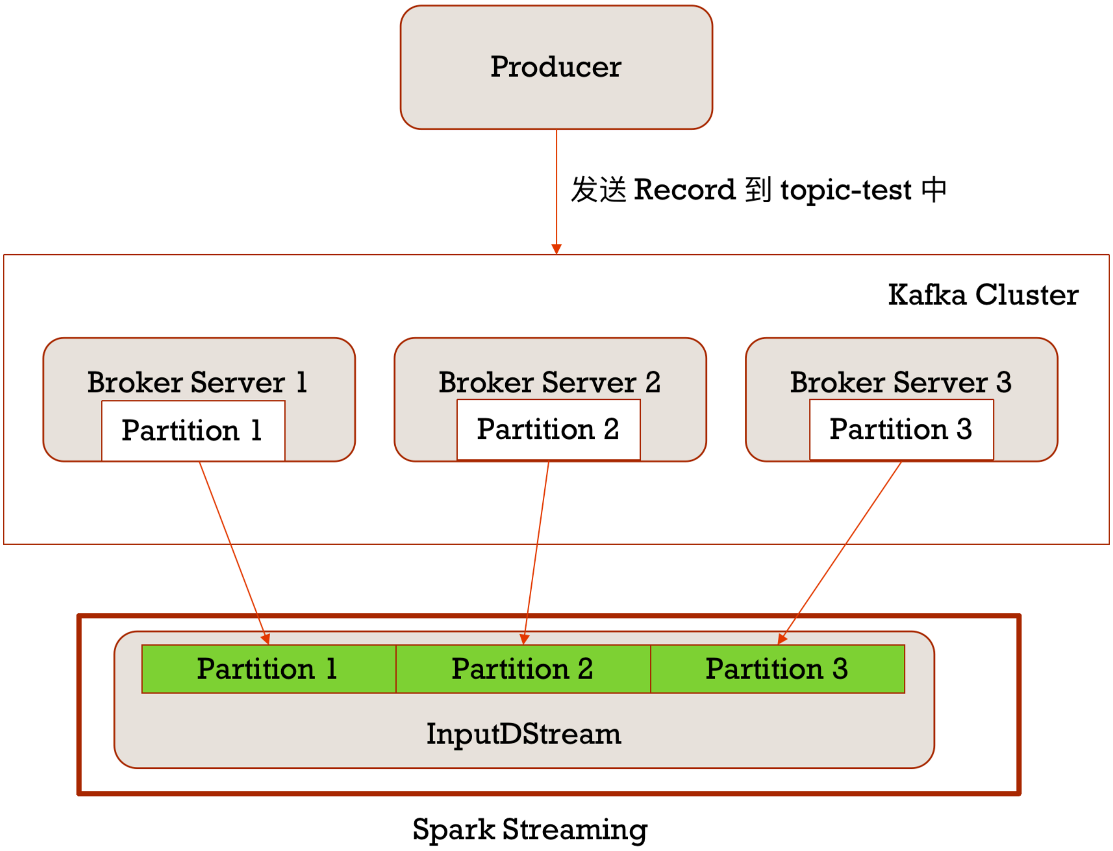
核心代码：

```
 import org.apache.spark.streaming.kafka._

 val kafkaStream = KafkaUtils.createStream(streamingContext,
     [ZK quorum], [consumer group id], [per-topic number of Kafka partitions
to consume])
```



4.3.2 方式二： Direct Approach (No Receivers)

这种新的不基于Receiver的直接方式，是在Spark 1.3中引入的，从而能够确保更加健壮的机制。替代掉使用Receiver来接收数据后，这种方式会周期性地查询Kafka，来获得每个topic+partition的最新的offset，从而定义每个batch的offset的范围。当处理数据的job启动时，就会使用Kafka的简单consumer api来获取Kafka指定offset范围的数据。 这种方式有如下优点： 1、简化并行读取：如果要读取多个partition，不需要创建多个输入DStream然后对它们进行union操作。Spark会创建跟Kafka partition一样多的RDD partition，并且会并行从Kafka中读取数据。所以在Kafka partition和RDD partition之间，有一个一对一的映射关系。 2、高性能：如果要保证零数据丢失，在基于receiver的方式中，需要开启WAL机制。这种方式其实效率低下，因为数据实际上被复制了两份，Kafka自己本身就有高可靠的机制，会对数据复制一份，而这里又会复制一份到WAL中。而基于direct的方式，不依赖Receiver，不需要开启WAL机制，只要Kafka中作了数据的复制，那么就可以通过Kafka的副本进行恢复。 3、一次且仅一次的事务机制： 基于receiver的方式，是使用Kafka的高阶API来在ZooKeeper中保存消费过的offset的。这是消费Kafka数据的传统方式。这种方式配合着WAL机制可以保证数据零丢失的高可靠性，但是却无法保证数据被处理一次且仅一次，可能会处理两次。因为Spark和ZooKeeper之间可能是不同步的。 4、降低资源。 Direct不需要Receivers，其申请的Executors全部参与到计算任务中；而Receiver-based则需要专门的Receivers来读取Kafka数据且不参与计算。因此相同的资源申请，Direct 能够支持更大的业务。 5、降低内存。 Receiver-based的Receiver与其他Exectuor是异步的，并持续不断接收数据，对于小业务量的场景还好，如果遇到大业务量时，需要提高Receiver的内存，但是参与计算的Executor并无需那么多的内存。而Direct 因为没有Receiver，而是在计算时读取数据，然后直接计算，所以对内存的要求很低。实际应用中我们可以把原先的10G降至现在的2-4G左右。 6、鲁棒性更好。 Receiver-based方法需要Receivers来异步持续不断的读取数据，因此遇到网络、存储负载等因素，导致实时任务出现堆积，但Receivers却还在持续读取数据，此种情况很容易导致计算崩溃。 Direct 则没有这种顾虑，其Driver在触发batch 计算任务时，才会读取数据并计算。队列出现堆积并不会引起程序的失败。



4.3.3 SparkStreaming与Kafka-0-8整合

支持0.8版本，或者更高的版本

pom.xml文件添加内容如下：

```
groupId = org.apache.spark
artifactId = spark-streaming-kafka-0-8_2.11
version = 2.3.3
```

代码演示：

```
import kafka.serializer.StringDecoder
import org.apache.spark.SparkConf
import org.apache.spark.streaming.kafka.KafkaUtils
import org.apache.spark.streaming.{Seconds, StreamingContext}

object KafkaDirec08 {
  def main(args: Array[String]): Unit = {
    //步骤一：初始化程序入口
    val sparkConf = new
SparkConf().setMaster("local[2]").setAppName("StreamingKafkaApp02")
    val ssc = new StreamingContext(sparkConf, Seconds(10))
    val kafkaParams =  Map[String, String](
      "metadata.broker.list"->"ruozehadoop000:9092",
          "group.id" -> "testflink"
    )
    val topics = "ruoze_kafka_streaming".split(",").toSet
   //步骤二：获取数据源
    val lines =
KafkaUtils.createDirectStream[String,String,StringDecoder,StringDecoder]
(ssc,kafkaParams,topics)
    //步骤三：业务代码处理
    lines.map(_._2).flatMap(_.split(",")).map((_,1)).reduceByKey(_+_).print()
    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }

}
```

要想保证数据不丢失，最简单的就是靠checkpoint的机制，但是checkpoint机制有个特点，入代码升级了，checkpoint机制就失效了。所以如果想实现数据不丢失，那么就需要自己管理offset。

4.3.4 SparkStreaming与Kafka-0-10整合

支持0.10版本，或者更高的版本

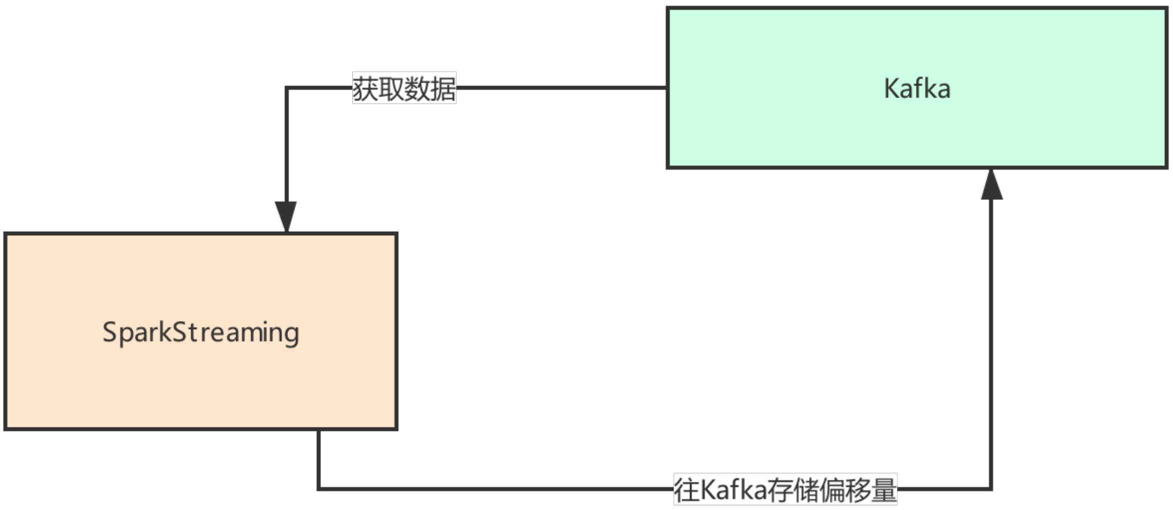
pom.xml文件添加内容如下：

```
   <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-streaming-kafka-0-10_2.11</artifactId>
        <version>${spark.version}</version>
   </dependency>
```

代码演示：

```
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.dstream.InputDStream
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.kafka010._


object KafkaDirect010 {

  def main(args: Array[String]): Unit = {
    //步骤一：获取配置信息
    val conf = new
SparkConf().setAppName("sparkstreamingoffset").setMaster("local[5]")
    conf.set("spark.streaming.kafka.maxRatePerPartition", "5")
    conf.set("spark.serializer",
"org.apache.spark.serializer.KryoSerializer");
```

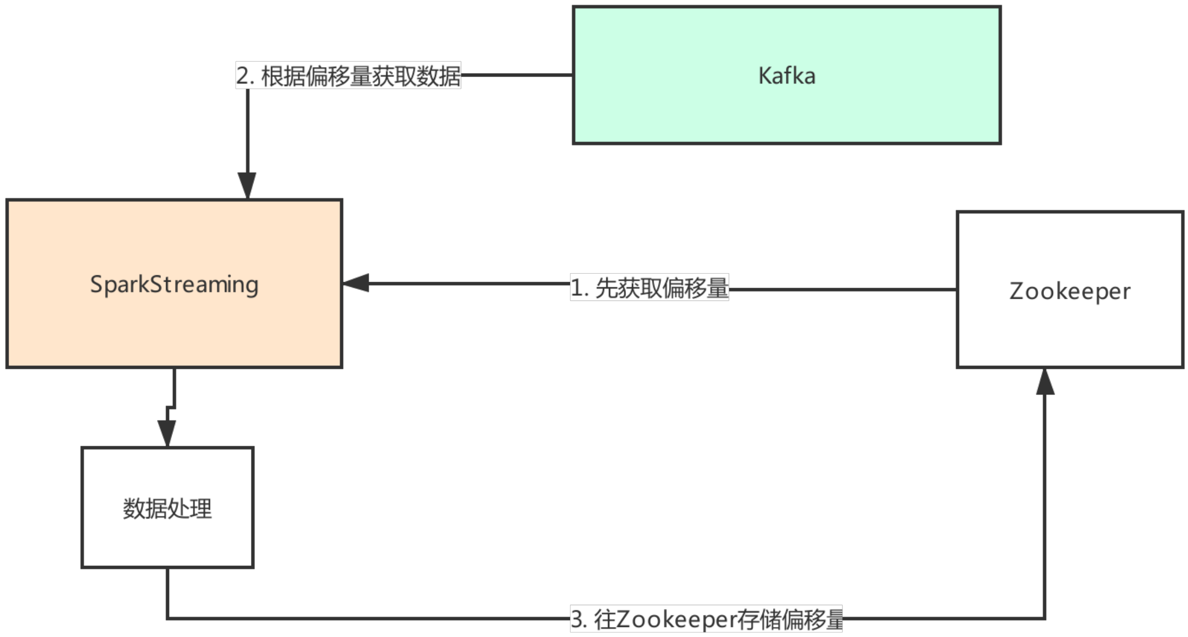
```
    val ssc = new StreamingContext(conf,Seconds(5))

    val brokers = "xxx:9092"
    val topics = "xx_openothers"
    val groupId = "xxx_consumer" //注意，这个也就是我们的消费者的名字

    val topicsSet = topics.split(",").toSet

    val kafkaParams = Map[String, Object](
      "bootstrap.servers" -> brokers,
      "group.id" -> groupId,
      "fetch.message.max.bytes" -> "209715200",
      "key.deserializer" -> classOf[StringDeserializer],
      "value.deserializer" -> classOf[StringDeserializer],
      "enable.auto.commit" -> "false"
    )

    //步骤二：获取数据源
    val stream: InputDStream[ConsumerRecord[String, String]] =
KafkaUtils.createDirectStream[String, String](
      ssc,
      LocationStrategies.PreferConsistent,
      ConsumerStrategies.Subscribe[String, String](topicsSet, kafkaParams))

    stream.foreachRDD( rdd =>{
      //步骤三：业务逻辑处理
      val newRDD: RDD[String] = rdd.map(_.value())
      newRDD.foreach( line =>{
        println(line)
      })
      //步骤四：提交偏移量信息，把偏移量信息添加到kafka里
      val offsetRanges  = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
      stream.asInstanceOf[CanCommitOffsets].commitAsync(offsetRanges)
    })


    ssc.start()
    ssc.awaitTermination()
    ssc.stop()

  }

}
```



4.4 SparkStreaming方案设计

4.4.1 解决SparkStreaming与Kafka0.8版本整合数据不丢失方案

方案设计如下：



代码一：

偏移量存入Zookeeper

```
/**
  * 自己管理offset
  */
class KafkaManager(val kafkaParams: Map[String, String]) extends Serializable
{
  private val kc = new KafkaCluster(kafkaParams)

  /**
    * 创建数据流
    */
  def createDirectStream[K: ClassTag, V: ClassTag, KD <: Decoder[K]: ClassTag, VD <: Decoder[V]: ClassTag](
                    ssc: StreamingContext,
                    kafkaParams: Map[String, String],
                    topics: Set[String]): InputDStream[(K, V)] =  {
    val groupId = kafkaParams.get("group.id").get
    // 在zookeeper上读取offsets前先根据实际情况更新offsets
    setOrUpdateOffsets(topics, groupId)

    //从zookeeper上读取offset开始消费message
    val messages = {
      val partitionsE = kc.getPartitions(topics)
      if (partitionsE.isLeft)
        throw new SparkException(s"get kafka partition failed: ${partitionsE.left.get}")
      val partitions = partitionsE.right.get
      val consumerOffsetsE = kc.getConsumerOffsets(groupId, partitions)
      if (consumerOffsetsE.isLeft)
        throw new SparkException(s"get kafka consumer offsets failed: ${consumerOffsetsE.left.get}")
      val consumerOffsets = consumerOffsetsE.right.get


      KafkaUtils.createDirectStream[K, V, KD, VD, (K, V)](
        ssc, kafkaParams, consumerOffsets, (mmd: MessageAndMetadata[K, V]) => (mmd.key, mmd.message))
    }

    messages
  }


  def createDirectStream[K, V, KD <: Decoder[K], VD <: Decoder[V]](
                                    jssc: JavaStreamingContext,
                                    keyClass: Class[K],
                                    valueClass: Class[V],
                                    keyDecoderClass: Class[KD],
                                    valueDecoderClass: Class[VD],
                                    kafkaParams: JMap[String, String],
                                    topics: JSet[String]
                                            ): JavaPairInputDStream[K, V] = {
    implicit val keyCmt: ClassTag[K] = ClassTag(keyClass)
    implicit val valueCmt: ClassTag[V] = ClassTag(valueClass)
    implicit val keyDecoderCmt: ClassTag[KD] = ClassTag(keyDecoderClass)
    implicit val valueDecoderCmt: ClassTag[VD] = ClassTag(valueDecoderClass)

    createDirectStream[K, V, KD, VD](jssc.ssc,  Map(kafkaParams.asScala.toSeq: _*),
      Set(topics.asScala.toSeq: _*));
  }


  /**
    * 创建数据流前，根据实际消费情况更新消费offsets
    * @param topics
    * @param groupId
    */
  private def setOrUpdateOffsets(topics: Set[String], groupId: String): Unit = {
    topics.foreach(topic => {
      var hasConsumed = true
      val partitionsE = kc.getPartitions(Set(topic))
      if (partitionsE.isLeft)
        throw new SparkException(s"get kafka partition failed: ${partitionsE.left.get}")
      val partitions = partitionsE.right.get
      val consumerOffsetsE = kc.getConsumerOffsets(groupId, partitions)
      if (consumerOffsetsE.isLeft) hasConsumed = false
      if (hasConsumed) {// 消费过
        /**
          * 如果streaming程序执行的时候出现kafka.common.OffsetOutOfRangeException，
          * 说明zk上保存的offsets已经过时了，即kafka的定时清理策略已经将包含该offsets的文件删除。
          * 针对这种情况，只要判断一下zk上的consumerOffsets和earliestLeaderOffsets的大小，
          * 如果consumerOffsets比earliestLeaderOffsets还小的话，说明consumerOffsets已过时，
          * 这时把consumerOffsets更新为earliestLeaderOffsets
          */
        val earliestLeaderOffsetsE = kc.getEarliestLeaderOffsets(partitions)
        if (earliestLeaderOffsetsE.isLeft)
          throw new SparkException(s"get earliest leader offsets failed: ${earliestLeaderOffsetsE.left.get}")
        val earliestLeaderOffsets = earliestLeaderOffsetsE.right.get
        val consumerOffsets = consumerOffsetsE.right.get

        // 可能只是存在部分分区consumerOffsets过时，所以只更新过时分区的consumerOffsets为earliestLeaderOffsets
        var offsets: Map[TopicAndPartition, Long] = Map()
        consumerOffsets.foreach({ case(tp, n) =>
          val earliestLeaderOffset = earliestLeaderOffsets(tp).offset
          if (n < earliestLeaderOffset) {
```

```
      println("consumer group:" + groupId + ",topic:" + tp.topic +
",partition:" + tp.partition +
       " offsets已经过时，更新为" + earliestLeaderOffset)
      offsets += (tp -> earliestLeaderOffset)
     }
    })
    if (!offsets.isEmpty) {
     kc.setConsumerOffsets(groupId, offsets)
    }
   } else {// 没有消费过
   val reset = kafkaParams.get("auto.offset.reset").map(_.toLowerCase)
    var leaderOffsets: Map[TopicAndPartition, LeaderOffset] = null
    if (reset == Some("smallest")) {
     val leaderOffsetsE = kc.getEarliestLeaderOffsets(partitions)
     if (leaderOffsetsE.isLeft)
      throw new SparkException(s"get earliest leader offsets failed:
${leaderOffsetsE.left.get}")
     leaderOffsets = leaderOffsetsE.right.get
    } else {
     val leaderOffsetsE = kc.getLatestLeaderOffsets(partitions)
     if (leaderOffsetsE.isLeft)
      throw new SparkException(s"get latest leader offsets failed:
${leaderOffsetsE.left.get}")
     leaderOffsets = leaderOffsetsE.right.get
    }
    val offsets = leaderOffsets.map {
     case (tp, offset) => (tp, offset.offset)
    }
    kc.setConsumerOffsets(groupId, offsets)
   }
  })
 }

 /**
  * 更新zookeeper上的消费offsets
  * @param rdd
  */
 def updateZKOffsets[K,V](rdd: RDD[(K, V)]) : Unit = {
  val groupId = kafkaParams.get("group.id").get
  val offsetsList = rdd.asInstanceOf[HasOffsetRanges].offsetRanges

  for (offsets <- offsetsList) {
   val topicAndPartition = TopicAndPartition(offsets.topic,
offsets.partition)
   val o = kc.setConsumerOffsets(groupId, Map((topicAndPartition,
offsets.untilOffset)))
   if (o.isLeft) {
    println(s"Error updating the offset to Kafka cluster: ${o.left.get}")
   }
```

```
        }
      }


    }
```

代码二：

这个类的目的是为了让API支持多语言

```
import scala.Tuple2;

public class TypeHelper {
    @SuppressWarnings("unchecked")
    public static   <K, V> scala.collection.immutable.Map<K, V>
toScalaImmutableMap(java.util.Map<K, V> javaMap) {
        final java.util.List<Tuple2<K, V>> list = new java.util.ArrayList<>
(javaMap.size());
        for (final java.util.Map.Entry<K, V> entry : javaMap.entrySet()) {
            list.add(Tuple2.apply(entry.getKey(), entry.getValue()));
        }
        final scala.collection.Seq<Tuple2<K, V>> seq =
scala.collection.JavaConverters.asScalaBufferConverter(list).asScala().toSeq()
;
        return (scala.collection.immutable.Map<K, V>)
scala.collection.immutable.Map$.MODULE$.apply(seq);
    }
}
```

代码三：

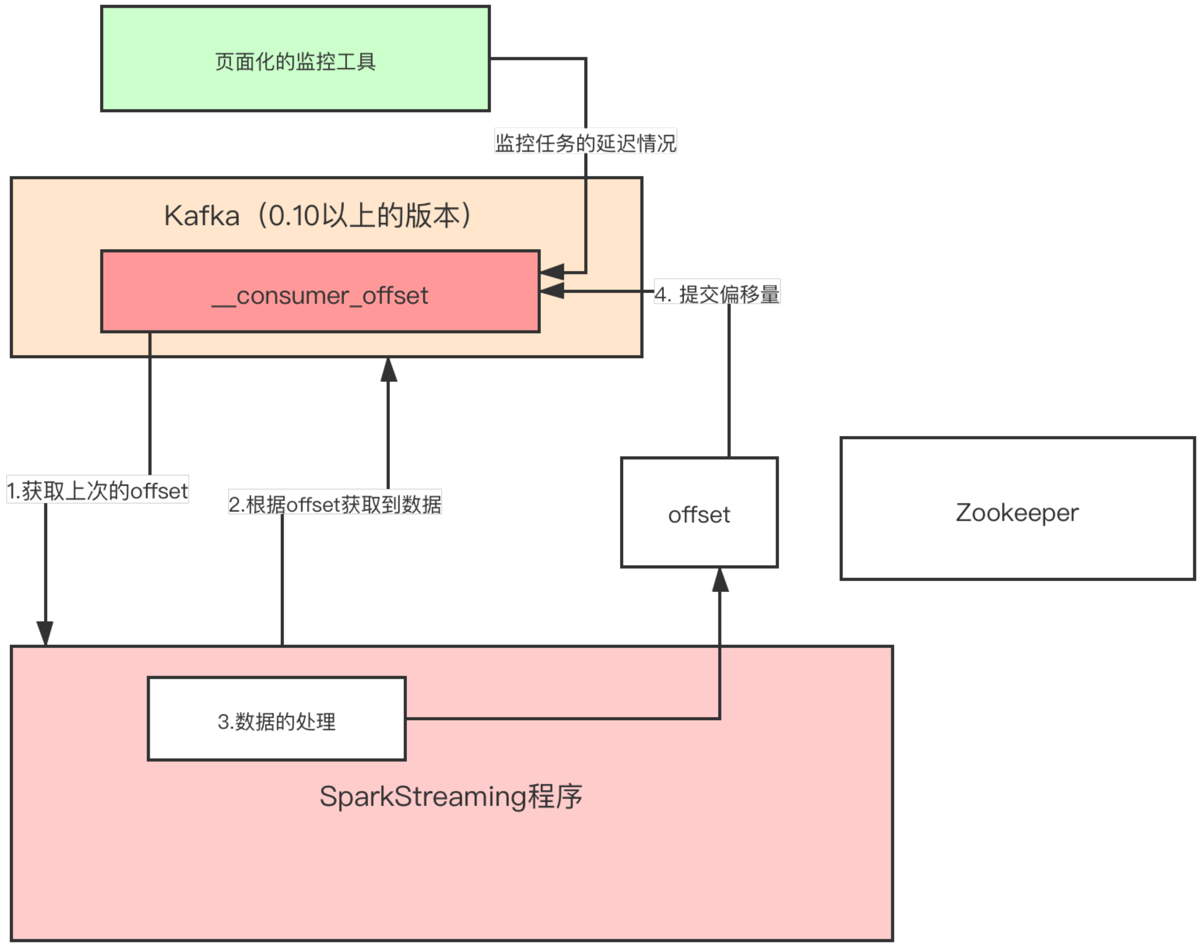
设置监听器，目的是为了让RD开发更方便。

```
import kafka.common.TopicAndPartition;
import org.apache.spark.streaming.kafka.KafkaCluster;
import org.apache.spark.streaming.kafka.OffsetRange;
import org.apache.spark.streaming.scheduler.*;
import scala.Option;
import scala.collection.JavaConversions;
import scala.collection.immutable.List;

import java.util.HashMap;
import java.util.Map;

public class MyListener implements StreamingListener {
    private KafkaCluster kc;
    public scala.collection.immutable.Map<String, String>  kafkaParams;
    public MyListener(scala.collection.immutable.Map<String, String>
kafkaParams){
        this.kafkaParams=kafkaParams;
         kc = new KafkaCluster(kafkaParams);
    }


//    @Override
//    public void onStreamingStarted(StreamingListenerStreamingStarted
streamingStarted) {
//
//    }

    @Override
    public void onReceiverStarted(StreamingListenerReceiverStarted
receiverStarted) {

    }

    @Override
    public void onReceiverError(StreamingListenerReceiverError receiverError)
{

    }

    @Override
    public void onReceiverStopped(StreamingListenerReceiverStopped
receiverStopped) {

    }

    @Override
    public void onBatchSubmitted(StreamingListenerBatchSubmitted
batchSubmitted) {

    }

    @Override
    public void onBatchStarted(StreamingListenerBatchStarted batchStarted) {

    }

    /**
     * 批次完成时调用的方法
     * @param batchCompleted
     */
    @Override
    public void onBatchCompleted(StreamingListenerBatchCompleted
batchCompleted) {
        //如果本批次里面有任务失败了，那么就终止偏移量提交
        scala.collection.immutable.Map<Object, OutputOperationInfo> opsMap =
batchCompleted.batchInfo().outputOperationInfos();
        Map<Object, OutputOperationInfo> javaOpsMap =
JavaConversions.mapAsJavaMap(opsMap);
        for (Map.Entry<Object, OutputOperationInfo> entry :
javaOpsMap.entrySet()) {
            //failureReason不等于None(是scala中的None),说明有异常，不保存offset
            if
(!"None".equalsIgnoreCase(entry.getValue().failureReason().toString())) {
                return;
            }
        }

        long batchTime =
batchCompleted.batchInfo().batchTime().milliseconds();

        /**
         * topic，分区，偏移量
         */
        Map<String, Map<Integer, Long>> offset = getOffset(batchCompleted);

        for (Map.Entry<String, Map<Integer, Long>> entry : offset.entrySet())
{
            String topic = entry.getKey();
            Map<Integer, Long> paritionToOffset = entry.getValue();

            //我只需要这儿把偏移信息放入到zookeeper就可以了。

            for(Map.Entry<Integer,Long> p2o : paritionToOffset.entrySet()){
                Map<TopicAndPartition, Object> map = new
HashMap<TopicAndPartition, Object>();

                TopicAndPartition topicAndPartition =
                        new TopicAndPartition(topic,p2o.getKey());

                map.put(topicAndPartition,p2o.getValue());

                scala.collection.immutable.Map<TopicAndPartition, Object>
                        topicAndPartitionObjectMap =
TypeHelper.toScalaImmutableMap(map);

                kc.setConsumerOffsets(kafkaParams.get("group.id").get(),
topicAndPartitionObjectMap);
            }

        }
```

```
    }

    @Override
    public void
onOutputOperationStarted(StreamingListenerOutputOperationStarted
outputOperationStarted) {

    }

    @Override
    public void
onOutputOperationCompleted(StreamingListenerOutputOperationCompleted
outputOperationCompleted) {

    }

    private Map<String, Map<Integer, Long>>
getOffset(StreamingListenerBatchCompleted batchCompleted) {
        Map<String, Map<Integer, Long>> map = new HashMap<>();

        scala.collection.immutable.Map<Object, StreamInputInfo> inputInfoMap =
                batchCompleted.batchInfo().streamIdToInputInfo();
        Map<Object, StreamInputInfo> infos =
JavaConversions.mapAsJavaMap(inputInfoMap);

        infos.forEach((k, v) -> {
            Option<Object> optOffsets = v.metadata().get("offsets");
            if (!optOffsets.isEmpty()) {
                Object objOffsets = optOffsets.get();
                if (List.class.isAssignableFrom(objOffsets.getClass())) {
                    List<OffsetRange> scalaRanges = (List<OffsetRange>)
objOffsets;

                    Iterable<OffsetRange> ranges =
JavaConversions.asJavaIterable(scalaRanges);
                    for (OffsetRange range : ranges) {
                        if (!map.containsKey(range.topic())) {
                            map.put(range.topic(), new HashMap<>());
                        }
                        map.get(range.topic()).put(range.partition(),
range.untilOffset());
                    }
                }
            }
        });

        return map;
    }
```

```
}
```

4.4.2 SparkStreaming与0.10kafka方案设计



代码实现：监听器

```
package lesson16.offset

import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.spark.streaming.dstream.InputDStream
import org.apache.spark.streaming.kafka010.{CanCommitOffsets, OffsetRange}
import org.apache.spark.streaming.scheduler._

class NxListener(var stream:InputDStream[ConsumerRecord[String, String]])
extends StreamingListener {

  override def onStreamingStarted(streamingStarted:
StreamingListenerStreamingStarted): Unit =
    super.onStreamingStarted(streamingStarted)

  override def onReceiverStarted(receiverStarted:
StreamingListenerReceiverStarted): Unit =
  super.onReceiverStarted(receiverStarted)

 override def onReceiverError(receiverError: StreamingListenerReceiverError):
Unit =
  super.onReceiverError(receiverError)

 override def onReceiverStopped(receiverStopped:
StreamingListenerReceiverStopped): Unit =
  super.onReceiverStopped(receiverStopped)

 override def onBatchSubmitted(batchSubmitted:
StreamingListenerBatchSubmitted): Unit = {



 }


 override def onBatchStarted(batchStarted: StreamingListenerBatchStarted):
Unit ={

 }

 override def onBatchCompleted(batchCompleted:
StreamingListenerBatchCompleted): Unit ={

  val oInfo: OutputOperationInfo =
batchCompleted.batchInfo.outputOperationInfos()
  if ("None".equalsIgnoreCase(oInfo.failureReason.toString())) {
   val info: Map[Int, StreamInputInfo] =
batchCompleted.batchInfo.streamIdToInputInfo

   var offsetRangesTmp:List[OffsetRange]=null;
   var offsetRanges:Array[OffsetRange]=null;

   for( k <- info){
    val offset: Option[Any] = k._2.metadata.get("offsets")

    if(!offset.isEmpty){
     try {
      val offsetValue = offset.get
      offsetRangesTmp= offsetValue.asInstanceOf[List[OffsetRange]]
      offsetRanges=offsetRangesTmp.toSet.toArray;
     } catch {
      case  e:Exception => println(e)
     }
    }
   }
```

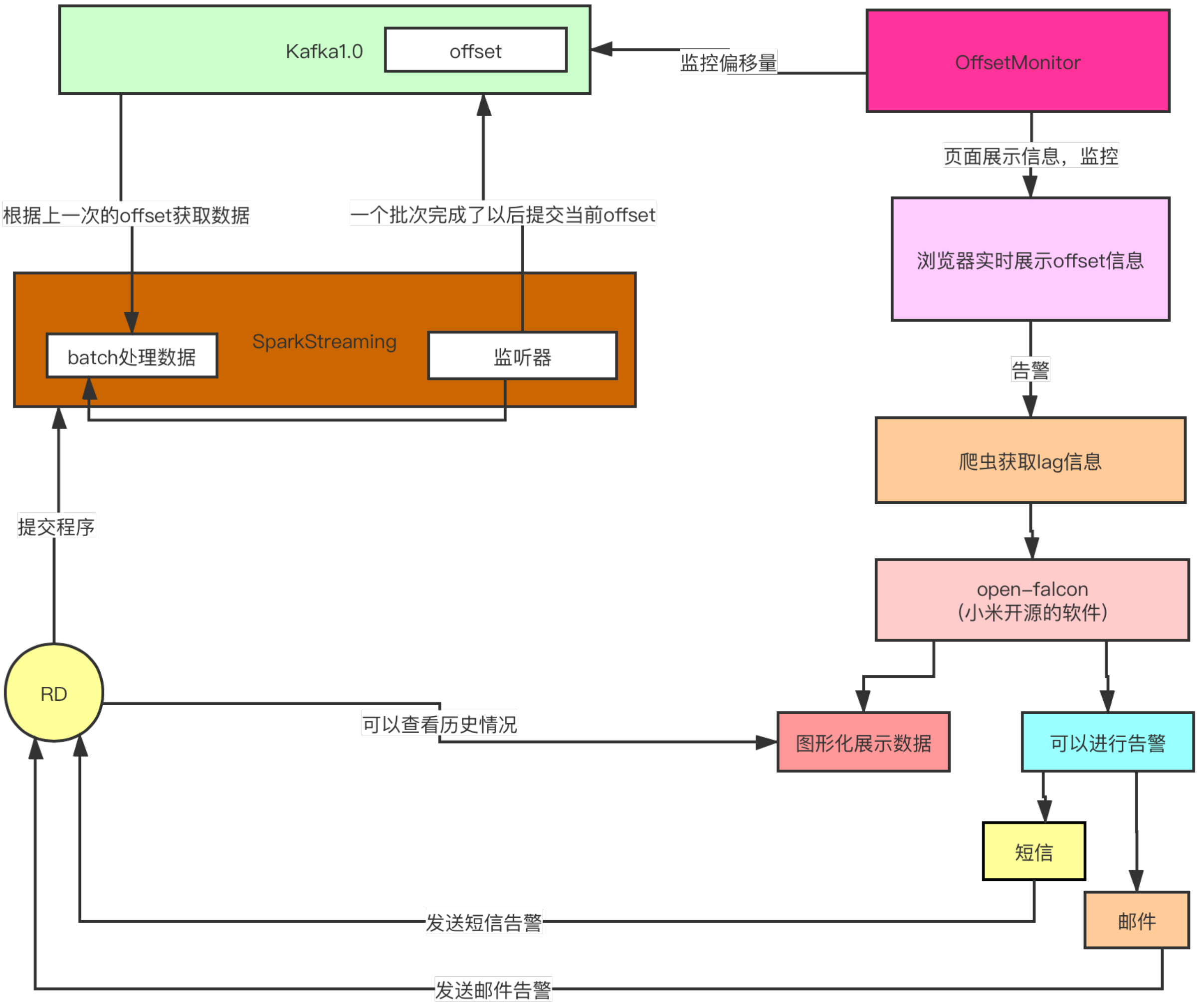
```
      if(offsetRanges != null){
        //因为我看了官网，所以我就知道这样就可以提交偏移量（Kafka）
        stream.asInstanceOf[CanCommitOffsets].commitAsync(offsetRanges);
      }


    }






  }


  override def onOutputOperationStarted(outputOperationStarted:
StreamingListenerOutputOperationStarted): Unit =
    super.onOutputOperationStarted(outputOperationStarted)

  override def onOutputOperationCompleted(outputOperationCompleted:
StreamingListenerOutputOperationCompleted): Unit =
    super.onOutputOperationCompleted(outputOperationCompleted)
}
```

案例演示：

```
import lesson16.offset.NxListener
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.SparkConf
import org.apache.spark.streaming.dstream.InputDStream
import org.apache.spark.streaming.kafka010.{ConsumerStrategies, KafkaUtils,
LocationStrategies}
import org.apache.spark.streaming.{Seconds, StreamingContext}

object DirectKafka010Kafka {
  def main(args: Array[String]): Unit = {
   // Logger.getLogger("org").setLevel(Level.ERROR)
    //步骤一：获取配置信息
    val conf = new
SparkConf().setAppName("DirectKafka010").setMaster("local[5]")
    conf.set("spark.streaming.kafka.maxRatePerPartition", "5")
    conf.set("spark.serializer",
"org.apache.spark.serializer.KryoSerializer");
    val ssc = new StreamingContext(conf,Seconds(5))
    val brokers = "192.16x.167.254:9092"
    val topics = "class3"
    val groupId = "class3_consumer2" //注意，这个也就是我们的消费者的名字

    val topicsSet = topics.split(",").toSet

    val kafkaParams = Map[String, Object](
      "bootstrap.servers" -> brokers,
      "key.deserializer" -> classOf[StringDeserializer],
      "value.deserializer" -> classOf[StringDeserializer],
      "group.id" -> groupId,
      "auto.offset.reset" -> "latest",
      "enable.auto.commit" -> (false: java.lang.Boolean)
    )


    //步骤二：获取数据源（主题里面读取offset）
    val stream: InputDStream[ConsumerRecord[String, String]] =
KafkaUtils.createDirectStream[String, String](
      ssc,
      LocationStrategies.PreferConsistent,
      ConsumerStrategies.Subscribe[String, String](topicsSet, kafkaParams))

    //设置监听器
    ssc.addStreamingListener(new NxListener(stream))

    val result = stream.map(_.value()).flatMap(_.split(","))
      .map((_, 1))
      .reduceByKey(_ + _)


    result.print()

    ssc.start()
    ssc.awaitTermination()
    ssc.stop()

  }

}
```

4.4.3 SparkStreaming任务实时告警监控方案设计

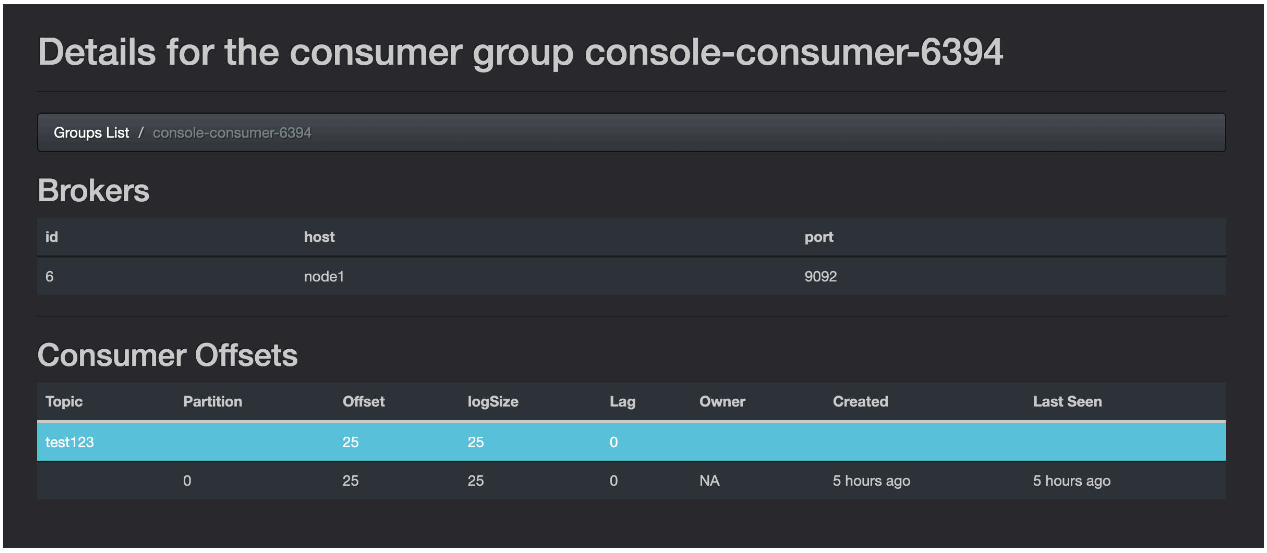
方案设计流程



offsetMonitor

http://kafkaom.netlearning.tech/

```
java -cp KafkaOffsetMonitor-assembly-0.3.0-SNAPSHOT.jar \
     com.quantifind.kafka.offsetapp.OffsetGetterWeb \
     --offsetStorage kafka \
     --zk hadoop1:2181 \
     --port 9004 \
     --refresh 15.seconds \
     --retain 2.days
```



Details for the consumer group console-consumer-6394

Groups List / console-consumer-6394

Brokers

id	host	port
6	node1	9092

Consumer Offsets

Topic	Partition	Offset	logSize	Lag	Owner	Created	Last Seen
test123		25	25	0			
	0	25	25	0	NA	5 hours ago	5 hours ago

4.5 SparkStreaming应用程序如何保证Exactly-Once?

一个流式计算如果想要保证Exactly-Once（不重不丢），那么首先要对这三个点有有要求：

（1）Source支持Replay。 （2）流计算引擎本身处理能保证Exactly-Once。 （3）Sink支持幂等或事务更新

也就是说如果要想让一个SparkSreaming的程序保证Exactly-Once,那么从如下三个角度出发：

（1）接收数据:从Source中接收数据。 （2）转换数据:用DStream和RDD算子转换。（SparkStreaming内部天然保证Exactly-Once） （3）储存数据:将结果保存至外部系统。 如果SparkStreaming程序需要实现Exactly-Once语义，那么每一个步骤都要保证Exactly-Once。

案例演示：

pom.xml添加内容如下：

```
       <dependency>
            <groupId>org.scalikejdbc</groupId>
            <artifactId>scalikejdbc_2.11</artifactId>
            <version>3.1.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.scalikejdbc/scalikejdbc-config -->
        <dependency>
            <groupId>org.scalikejdbc</groupId>
            <artifactId>scalikejdbc-config_2.11</artifactId>
            <version>3.1.0</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.39</version>
        </dependency>
```

代码：

```
import org.apache.kafka.common.TopicPartition
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession
import org.apache.spark.streaming.kafka010.{ConsumerStrategies,
HasOffsetRanges, KafkaUtils, LocationStrategies}
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.slf4j.LoggerFactory
import scalikejdbc.{ConnectionPool, DB, _}
/**
  *    SparkStreaming EOS:
  *      Input:Kafka
  *      Process:Spark Streaming
  *      Output:Mysql
  *
  *      保证EOS:
  *        1、偏移量自己管理，即enable.auto.commit=false,这里保存在Mysql中
  *        2、使用createDirectStream
  *        3、事务输出：结果存储与Offset提交在Driver端同一Mysql事务中
  */
object SparkStreamingEOSKafkaMysqlAtomic {
  @transient lazy val logger = LoggerFactory.getLogger(this.getClass)

  def main(args: Array[String]): Unit = {

    val topic="topic1"
    val group="spark_app1"

    //Kafka配置
    val kafkaParams= Map[String, Object](
      "bootstrap.servers" -> "node1:6667,node2:6667,node3:6667",
      "key.deserializer" -> classOf[StringDeserializer],
      "value.deserializer" -> classOf[StringDeserializer],
      "auto.offset.reset" -> "latest",//latest earliest
      "enable.auto.commit" -> (false: java.lang.Boolean),
      "group.id" -> group)

    //在Driver端创建数据库连接池
    ConnectionPool.singleton("jdbc:mysql://node3:3306/bigdata", "", "")

    val conf = new
SparkConf().setAppName(this.getClass.getSimpleName.replace("$",""))
    val ssc = new StreamingContext(conf,Seconds(5))

    //1)初次启动或重启时,从指定的Partition、Offset构建TopicPartition
    //2)运行过程中,每个Partition、Offset保存在内部currentOffsets =
Map[TopicPartition, Long]()变量中
```

```
    //3)后期Kafka Topic分区动扩展,在运行过程中不能自动感知
    val initOffset=DB.readOnly(implicit session=>{
      sql"select `partition`,offset from kafka_topic_offset where topic
=${topic} and `group`=${group}"
        .map(item=> new TopicPartition(topic, item.get[Int]("partition")) ->
item.get[Long]("offset"))
        .list().apply().toMap
    })

    //CreateDirectStream
    //从指定的Topic、Partition、Offset开始消费
    val sourceDStream =KafkaUtils.createDirectStream[String,String](
      ssc,
      LocationStrategies.PreferConsistent,
      ConsumerStrategies.Assign[String,String]
(initOffset.keys,kafkaParams,initOffset)
    )

    sourceDStream.foreachRDD(rdd=>{
      if (!rdd.isEmpty()){
        val offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
        offsetRanges.foreach(offsetRange=>{
          logger.info(s"Topic: ${offsetRange.topic},Group: ${group},Partition:
${offsetRange.partition},fromOffset: ${offsetRange.fromOffset},untilOffset:
${offsetRange.untilOffset}")
        })

        //统计分析
        //将结果收集到Driver端
        val sparkSession =
SparkSession.builder.config(rdd.sparkContext.getConf).getOrCreate()
        import sparkSession.implicits._
        val dataFrame = sparkSession.read.json(rdd.map(_.value()).toDS)
        dataFrame.createOrReplaceTempView("tmpTable")
        val result=sparkSession.sql(
          """
            |select
            |   --每分钟
            |   eventTimeMinute,
            |   --每种语言
            |   language,
            |   -- 次数
            |   count(1) pv,
            |   -- 人数
            |   count(distinct(userID)) uv
            |from(
            |   select *, substr(eventTime,0,16) eventTimeMinute from tmpTable
            |) as tmp group by eventTimeMinute,language
          """.stripMargin
        ).collect()

        //在Driver端存储数据、提交Offset
        //结果存储与Offset提交在同一事务中原子执行
        //这里将偏移量保存在Mysql中
        DB.localTx(implicit session=>{

          //结果存储
          result.foreach(row=>{
            sql"""
            insert into twitter_pv_uv (eventTimeMinute, language,pv,uv)
            value (
                ${row.getAs[String]("eventTimeMinute")},
                ${row.getAs[String]("language")},
                ${row.getAs[Long]("pv")},
                ${row.getAs[Long]("uv")}
                )
            on duplicate key update pv=pv,uv=uv
          """.update.apply()
          })

          //Offset提交
          offsetRanges.foreach(offsetRange=>{
            val affectedRows = sql"""
          update kafka_topic_offset set offset = ${offsetRange.untilOffset}
          where
            topic = ${topic}
            and `group` = ${group}
            and `partition` = ${offsetRange.partition}
            and offset = ${offsetRange.fromOffset}
          """.update.apply()

            if (affectedRows != 1) {
              throw new Exception(s"""Commit Kafka Topic: ${topic} Offset
Faild!""")
            }
          })
        })
      }
    })

    ssc.start()
    ssc.awaitTermination()
  }

}
```

五 、知识扩展-ScalikeJDBC（5分钟）

1、什么是ScalikeJDBC

ScalikeJDBC是一款给Scala开发者使用的简洁DB访问类库，它是基于SQL的，使用者只需要关注SQL逻辑的编写，所有的数据库操作都交给ScalikeJDBC。这个类库内置包含了JDBC API，并且给用户提供了简单易用并且非常灵活的API。并且，QueryDSL(通用查询查询框架)使你的代码类型安全的并且可重复使用。我们可以在生产环境大胆地使用这款DB访问类库。

2、IDEA项目中导入相关库

```
<!-- https://mvnrepository.com/artifact/org.scalikejdbc/scalikejdbc -->
<dependency>
    <groupId>org.scalikejdbc</groupId>
    <artifactId>scalikejdbc_2.11</artifactId>
    <version>3.1.0</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.scalikejdbc/scalikejdbc-config -->
<dependency>
    <groupId>org.scalikejdbc</groupId>
    <artifactId>scalikejdbc-config_2.11</artifactId>
    <version>3.1.0</version>
</dependency>
<!-- mysql " mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>
```

3、数据库操作

3.1 数据库连接配置信息

在IDEA的resources文件夹下创建application.conf:

```
#mysql的连接配置信息
db.default.driver="com.mysql.jdbc.Driver"
db.default.url="jdbc:mysql://localhost:3306/spark"
db.default.user="root"
db.default.password="123456"
```

scalikeJDBC默认加载default配置

或者使用自定义配置:

```
#mysql的连接配置信息
db.fred.driver="com.mysql.jdbc.Driver"
db.fred.url="jdbc:mysql://localhost:3306/spark"
db.fred.user="root"
db.fred.password="123456"
```

3.2 加载数据配置信息

```
//默认加载default配置信息
DBs.setup()
//加载自定义的fred配置信息
DBs.setup('fred)
```

3.3 查询数据库并封装数据

```
//配置mysql
DBs.setup()

//查询数据并返回单个列，并将列数据封装到集合中
val list = DB.readOnly({implicit session =>
  SQL("select content from post")
    .map(rs =>
    rs.string("content")).list().apply()
})
for(s <- list){
  println(s)
}
```

```
case class Users(id:String, name:String, nickName:String)

/**
  * 查询数据库，并将数据封装成对象，并返回一个集合
  */
//配置mysql
DBs.setup('fred)

//查询数据并返回单个列，并将列数据封装到集合中
val users = NamedDB('fred).readOnly({implicit session =>
  SQL("select * from users").map(rs =>
  Users(rs.string("id"), rs.string("name"),
    rs.string("nickName"))).list().apply()
})
for (u <- users){
  println(u)
}
```

3.4 插入数据

3.4.1 AutoCommit

```
/**
  * 插入数据，使用AutoCommit
  * @return
  */
val insertResult = DB.autoCommit({implicit session =>
  SQL("insert into users(name, nickName) values(?,?)").bind("test01",
"test01")
    .update().apply()
})
println(insertResult)
```

3.4.2 插入返回主键标识

```
/**
  * 插入数据，并返回主键
  * @return
  */
val id = DB.localTx({implicit session =>
  SQL("insert into users(name, nickName, sex) values(?,?,?)").bind("test",
"000", "male")
    .updateAndReturnGeneratedKey("nickName").apply()
})
println(id)
```

3.4.3 事务插入

```
/**
  * 使用事务插入数据库
  * @return
  */
val tx = DB.localTx({implicit session =>
  SQL("insert into users(name, nickName, sex) values(?,?,?)").bind("test",
"haha", "male").update().apply()
  //下一行会报错，用于测试
  var s = 1 / 0
  SQL("insert into users(name, nickName, sex) values(?,?,?)").bind("test01",
"haha01", "male01").update().apply()
})
println(s"tx = ${tx}")
```

3.4.4 更新数据

```
/**
  * 更新数据
  * @return
  */
DB.localTx({implicit session =>
  SQL("update users set nickName = ?").bind("xiaoming").update().apply()
})
```

六 、总结（5分钟）