> 未知才有趣不是吗

SparkStreaming之DStream创建

通过RDD队列创建DStream

需求

> 循环创建几个RDD，将RDD放入队列。通过SparkStream创建Dstream，计算WordCount

代码实现

```
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.dstream.{DStream, InputDStream}
import org.apache.spark.streaming.{Seconds, StreamingContext}

import scala.collection.mutable

object RDDStream {

  def main(args: Array[String]) {

    //1.初始化Spark配置信息
    val conf = new SparkConf().setMaster("local[*]").setAppName("RDDStream")

    //2.初始化SparkStreamingContext
    val ssc = new StreamingContext(conf, Seconds(4))

    //3.创建RDD队列
    val rddQueue = new mutable.Queue[RDD[Int]]()

    //4.创建QueueInputDStream
    val inputStream = ssc.queueStream(rddQueue,oneAtATime = false)

    //5.处理队列中的RDD数据
    val mappedStream = inputStream.map((_,1))
    val reducedStream = mappedStream.reduceByKey(_ + _)

    //6.打印结果
    reducedStream.print()

    //7.启动任务
    ssc.start()

    //8.循环创建并向RDD队列中放入RDD
    for (i <- 1 to 5) {
      rddQueue += ssc.sparkContext.makeRDD(1 to 300, 10)
      Thread.sleep(2000)
    }

    ssc.awaitTermination()
```

```
  }
}
```

结果展示

```
-------------------------------------------
Time: 1539075280000 ms
-------------------------------------------
(4,60)
(0,60)
(6,60)
(8,60)
(2,60)
(1,60)
(3,60)
(7,60)
(9,60)
(5,60)

-------------------------------------------
Time: 1539075284000 ms
-------------------------------------------
(4,60)
(0,60)
(6,60)
(8,60)
(2,60)
(1,60)
(3,60)
(7,60)
(9,60)
(5,60)

-------------------------------------------
Time: 1539075288000 ms
-------------------------------------------
(4,30)
(0,30)
(6,30)
(8,30)
(2,30)
(1,30)
(3,30)
(7,30)
(9,30)
(5,30)

-------------------------------------------
Time: 1539075292000 ms
-------------------------------------------
```

通过自定义数据源创建DStream

需求

> 自定义数据源，实现监控某个端口号，获取该端口号内容，然后进行处理【有问题都可以私聊我WX：focusbigdata，或者关注我的公众号：FocusBigData，注意大小写】

代码实现

> 继承Receiver，并实现onStart、onStop方法来自定义数据源采集

```
import java.io.{BufferedReader, InputStreamReader}
import java.net.Socket
import java.nio.charset.StandardCharsets
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.receiver.Receiver

class CustomerReceiver(host: String, port: Int) extends Receiver[String]
(StorageLevel.MEMORY_ONLY) {

  //最初启动的时候，调用该方法，作用为：读数据并将数据发送给Spark
  override def onStart(): Unit = {
    new Thread("Socket Receiver") {
      override def run() {
        receive()
      }
    }.start()
  }

  //读数据并将数据发送给Spark
  def receive(): Unit = {

    //创建一个Socket
    var socket: Socket = new Socket(host, port)

    //定义一个变量，用来接收端口传过来的数据
    var input: String = null

    //创建一个BufferedReader用于读取端口传来的数据
    val reader = new BufferedReader(new InputStreamReader(socket.getInputStream,
StandardCharsets.UTF_8))

    //读取数据
    input = reader.readLine()

    //当receiver没有关闭并且输入数据不为空，则循环发送数据给Spark
    while (!isStopped() && input != null) {
      store(input)
      input = reader.readLine()
    }

    //跳出循环则关闭资源
    reader.close()
    socket.close()

    //重启任务
    restart("restart")
  }

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

> 使用自定义的数据源采集数据

```
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.dstream.DStream

object FileStream {

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

    //1.初始化Spark配置信息
val sparkConf = new SparkConf().setMaster("local[*]")
.setAppName("StreamWordCount")

    //2.初始化SparkStreamingContext
    val ssc = new StreamingContext(sparkConf, Seconds(5))

//3.创建自定义receiver的Streaming
val lineStream = ssc.receiverStream(new CustomerReceiver("hadoop102", 9999))

    //4.将每一行数据做切分，形成一个个单词
    val wordStream = lineStream.flatMap(_.split("\t"))

    //5.将单词映射成元组（word,1）
    val wordAndOneStream = wordStream.map((_, 1))

    //6.将相同的单词次数做统计
    val wordAndCountStream = wordAndOneStream.reduceByKey(_ + _)

    //7.打印
    wordAndCountStream.print()

    //8.启动SparkStreamingContext
    ssc.start()
    ssc.awaitTermination()
  }
}
```

通过Kafka数据源创建DStream

> 重点掌握，生产环境数据都是放在kafka上

需求

> 通过SparkStreaming从Kafka读取数据，并将读取过来的数据做简单计算(WordCount)，最终打印到控制台。

Maven依赖

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka-0-8_2.11</artifactId>
    <version>2.1.1</version>
</dependency>
```

代码实现

```
import kafka.common.TopicAndPartition
import kafka.message.MessageAndMetadata
import kafka.serializer.StringDecoder
import org.apache.kafka.clients.consumer.ConsumerConfig
import org.apache.spark.SparkConf
import org.apache.spark.streaming.dstream.{DStream, InputDStream}
import org.apache.spark.streaming.kafka.KafkaCluster.Err
import org.apache.spark.streaming.kafka.{HasOffsetRanges, KafkaCluster,
KafkaUtils, OffsetRange}
import org.apache.spark.streaming.{Seconds, StreamingContext}

import scala.collection.mutable

object KafkaStreaming {

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

    //创建SparkConf对象
    val sparkConf: SparkConf = new
SparkConf().setMaster("local[*]").setAppName("KafkaStreaming")

    //创建StreamingContext对象
    val ssc: StreamingContext = new StreamingContext(sparkConf, Seconds(3))

    //kafka参数声明
    val brokers = "hadoop102:9092,hadoop103:9092,hadoop104:9092"
    val topic = "first"
    val group = "bigdata"
    val deserialization =
"org.apache.kafka.common.serialization.StringDeserializer"

    //定义Kafka参数
    val kafkaPara: Map[String, String] = Map[String, String](
      ConsumerConfig.GROUP_ID_CONFIG -> group,
      ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG -> brokers,
      ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG -> deserialization,
      ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG -> deserialization
    )

    //创建KafkaCluster（维护offset）
    val kafkaCluster = new KafkaCluster(kafkaPara)

    //获取ZK中保存的offset
    val fromOffset: Map[TopicAndPartition, Long] =
getOffsetFromZookeeper(kafkaCluster, group, Set(topic))

    //读取kafka数据创建DStream
    val kafkaDStream: InputDStream[String] =
KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder,
String](ssc,
      kafkaPara,
      fromOffset,
      (x: MessageAndMetadata[String, String]) => x.message())

    //数据处理
```

```
    kafkaDStream.print

    //提交offset
    offsetToZookeeper(kafkaDStream, kafkaCluster, group)

    ssc.start()
    ssc.awaitTermination()
  }

  //从ZK获取offset
  def getOffsetFromZookeeper(kafkaCluster: KafkaCluster, kafkaGroup: String,
kafkaTopicSet: Set[String]): Map[TopicAndPartition, Long] = {

    // 创建Map存储Topic和分区对应的offset
    val topicPartitionOffsetMap = new mutable.HashMap[TopicAndPartition, Long]()

    // 获取传入的Topic的所有分区
    // Either[Err, Set[TopicAndPartition]]  : Left(Err)
Right[Set[TopicAndPartition]]
    val topicAndPartitions: Either[Err, Set[TopicAndPartition]] =
kafkaCluster.getPartitions(kafkaTopicSet)

    // 如果成功获取到Topic所有分区
    // topicAndPartitions: Set[TopicAndPartition]
    if (topicAndPartitions.isRight) {
      // 获取分区数据
      // partitions: Set[TopicAndPartition]
      val partitions: Set[TopicAndPartition] = topicAndPartitions.right.get

      // 获取指定分区的offset
      // offsetInfo: Either[Err, Map[TopicAndPartition, Long]]
      // Left[Err]  Right[Map[TopicAndPartition, Long]]
      val offsetInfo: Either[Err, Map[TopicAndPartition, Long]] =
kafkaCluster.getConsumerOffsets(kafkaGroup, partitions)

      if (offsetInfo.isLeft) {

        // 如果没有offset信息则存储0
        // partitions: Set[TopicAndPartition]
        for (top <- partitions)
          topicPartitionOffsetMap += (top -> 0L)
      } else {

        // 如果有offset信息则存储offset
        // offsets: Map[TopicAndPartition, Long]
        val offsets: Map[TopicAndPartition, Long] = offsetInfo.right.get
        for ((top, offset) <- offsets)
          topicPartitionOffsetMap += (top -> offset)
      }
    }
    topicPartitionOffsetMap.toMap
  }

  //提交offset
  def offsetToZookeeper(kafkaDstream: InputDStream[String], kafkaCluster:
KafkaCluster, kafka_group: String): Unit = {
    kafkaDstream.foreachRDD {
      rdd =>
```

```
        // 获取DStream中的offset信息
        // offsetsList: Array[OffsetRange]
        // OffsetRange: topic partition fromoffset untiloffset
        val offsetsList: Array[OffsetRange] =
rdd.asInstanceOf[HasOffsetRanges].offsetRanges

        // 遍历每一个offset信息，并更新Zookeeper中的元数据
        // OffsetRange: topic partition fromoffset untiloffset
        for (offsets <- offsetsList) {
          val topicAndPartition = TopicAndPartition(offsets.topic,
offsets.partition)
          // ack: Either[Err, Map[TopicAndPartition, Short]]
          // Left[Err]
          // Right[Map[TopicAndPartition, Short]]
          val ack: Either[Err, Map[TopicAndPartition, Short]] =
kafkaCluster.setConsumerOffsets(kafka_group, Map((topicAndPartition,
offsets.untilOffset)))
          if (ack.isLeft) {
            println(s"Error updating the offset to Kafka cluster:
${ack.left.get}")
          } else {
            println(s"update the offset to Kafka cluster: ${offsets.untilOffset}
successfully")
          }
        }
    }
  }
}
```