大数据技术之SparkStreaming

版本：V2.0

# Spark Streaming概述

## 离线和实时概念

数据处理的延迟

* 离线计算

就是在计算开始前已知所有输入数据，输入数据不会产生变化，一般计算量级较大，计算时间也较长。例如今天早上一点，把昨天累积的日志，计算出所需结果。最经典的就是Hadoop的MapReduce方式；

* 实时计算

输入数据是可以以序列化的方式一个个输入并进行处理的，也就是说在开始的时候并不需要知道所有的输入数据。与离线计算相比，运行时间短，计算量级相对较小。强调计算过程的时间要短，即所查当下给出结果。

## 批量和流式概念

数据处理的方式

* 批：处理离线数据，冷数据。单个处理数据量大，处理速度比流慢。
* 流：在线，实时产生的数据。单次处理的数据量小，但处理速度更快。

近年来，在Web应用、网络监控、传感监测等领域，兴起了一种新的数据密集型应用——流数据，即数据以大量、快速、时变的流形式持续到达。实例：PM2.5检测、电子商务网站用户点击流。

流数据具有如下特征：

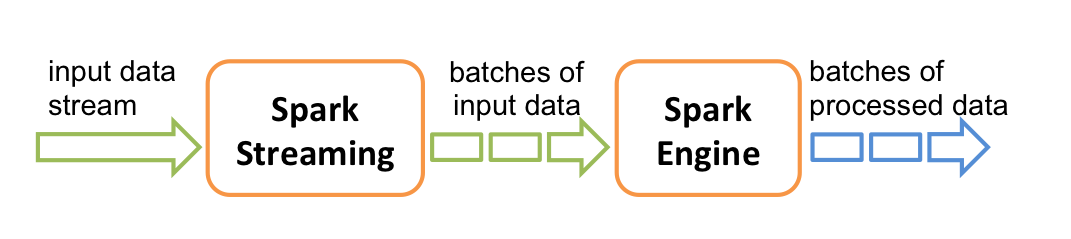
* + 数据快速持续到达，潜在大小也许是无穷无尽的；
  + 数据来源众多，格式复杂；
  + 数据量大，但是不十分关注存储，一旦经过处理，要么被丢弃，要么被归档存储；
  + 注重数据的整体价值，不过分关注个别数据；

## Spark Streaming是什么

Spark Streaming用于流式数据的处理。Spark Streaming支持的数据输入源很多，例如：Kafka、Flume、Twitter、ZeroMQ和简单的TCP套接字等等。数据输入后可以用Spark的高度抽象算子如：map、reduce、join、window等进行运算。而结果也能保存在很多地方，如HDFS，数据库等。



在 Spark Streaming 中，处理数据的单位是一批而不是单条，而数据采集却是逐条进行的，因此 Spark Streaming 系统需要设置间隔使得数据汇总到一定的量后再一并操作，这个间隔就是批处理间隔。批处理间隔是Spark Streaming的核心概念和关键参数，它决定了Spark Streaming提交作业的频率和数据处理的延迟，同时也影响着数据处理的吞吐量和性能。



和Spark基于RDD的概念很相似，Spark Streaming使用了一个高级抽象离散化流(discretized stream)，叫作DStreams。DStreams是随时间推移而收到的数据的序列。在内部，每个时间区间收到的数据都作为RDD存在，而DStreams是由这些RDD所组成的序列(因此得名“离散化”)。DStreams可以由来自数据源的输入数据流来创建, 也可以通过在其他的 DStreams上应用一些高阶操作来得到。



## Spark Streaming特点

### 易用



### 容错



### 易整合到Spark体系



### 缺点

Spark Streaming是一种“微量批处理”架构, 和其他基于“一次处理一条记录”架构的系统相比, 它的延迟会相对高一些。

## Spark Streaming架构

### 架构图



图1-1 SparkStreaming架构图

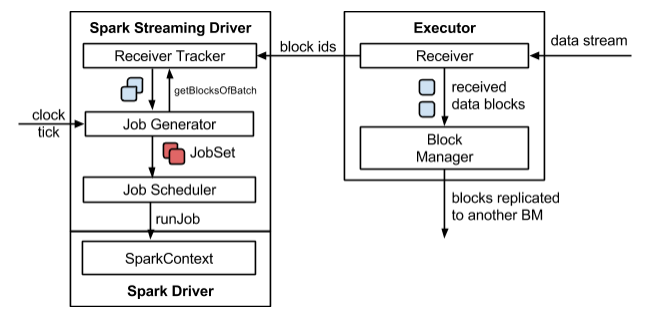


图1-2整体架构图

### 背压机制

Spark 1.5以前版本，用户如果要限制Receiver的数据接收速率，可以通过设置静态配制参数“spark.streaming.receiver.maxRate”的值来实现，此举虽然可以通过限制接收速率，来适配当前的处理能力，防止内存溢出，但也会引入其它问题。比如：producer数据生产高于maxRate，当前集群处理能力也高于maxRate，这就会造成资源利用率下降等问题。

为了更好的协调数据接收速率与资源处理能力，1.5版本开始Spark Streaming可以动态控制数据接收速率来适配集群数据处理能力。背压机制（即Spark Streaming Backpressure）: 根据JobScheduler反馈作业的执行信息来动态调整Receiver数据接收率。

通过属性“spark.streaming.backpressure.enabled”来控制是否启用backpressure机制，默认值false，即不启用。

# DStream入门

## WordCount案例实操

1. 需求：使用netcat工具向9999端口不断的发送数据，通过SparkStreaming读取端口数据并统计不同单词出现的次数
2. 添加依赖

<dependency>

<groupId>org.apache.spark</groupId>  
 <artifactId>spark-streaming\_2.11</artifactId>  
 <version>2.1.1</version>  
</dependency>

1. 编写代码

*/\*\**

*\* Author: Felix  
 \* Date: 2020/2/20  
 \* Desc: Spark Streaming WordCount案例  
 \*/*

**object** Spark01\_WordCount {

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

//创建配置文件对象 注意：Streaming程序至少不能设置为local，至少需要2个线程  
 **val** conf: SparkConf = **new** SparkConf().setAppName("Spark01\_W").setMaster("local[\*]")  
 //创建Spark Streaming上下文环境对象  
 **val** ssc = **new** StreamingContext(conf,*Seconds*(3))  
 //操作数据源-从端口中获取一行数据  
 **val** socketDS: ReceiverInputDStream[String] = ssc.socketTextStream("hadoop202",9999)  
 //对获取的一行数据进行扁平化操作  
 **val** flatMapDS: DStream[String] = socketDS.flatMap(\_.split(" "))  
 //结构转换  
 **val** mapDS: DStream[(String, Int)] = flatMapDS.map((\_,1))  
 //对数据进行聚合  
 **val** reduceDS: DStream[(String, Int)] = mapDS.reduceByKey(\_+\_)  
 //输出结果 注意：调用的是DS的print函数  
 reduceDS.print()  
 //启动采集器  
 ssc.start()  
 //默认情况下，上下文对象不能关闭  
 //ssc.stop()  
 //等待采集结束，终止上下文环境对象  
 ssc.awaitTermination()  
 }  
}

1. 启动程序并通过NetCat发送数据：

[atguigu@hadoop202 ~]$ nc -lk 9999

注意：如果程序运行时，log日志太多，可以将spark conf目录下的log4j文件里面的日志级别改成WARN。

## WordCount解析

Discretized Stream是Spark Streaming的基础抽象，代表持续性的数据流和经过各种Spark算子操作后的结果数据流。在内部实现上，DStream是一系列连续的RDD来表示,每个RDD含有一段时间间隔内的数据，对这些 RDD的转换是由Spark引擎来计算的， DStream的操作隐藏的大多数的细节, 然后给开发者提供了方便使用的高级 API如下图：

 例如：水位监控

## 几点注意

* 一旦StreamingContext已经启动, 则不能再添加新的 streaming computations
* 一旦一个StreamingContext已经停止(StreamingContext.stop()), 他也不能再重启
* 在一个 JVM 内, 同一时间只能启动一个StreamingContext
* stop() 的方式停止StreamingContext, 也会把SparkContext停掉. 如果仅仅想停止StreamingContext, 则应该这样: stop(false)
* 一个SparkContext可以重用去创建多个StreamingContext, 前提是以前的StreamingContext已经停掉,并且SparkContext没有被停掉

# DStream创建

## RDD队列(了解)

### 用法及说明

测试过程中，可以通过使用ssc.queueStream(queueOfRDDs)来创建DStream，每一个推送到这个队列中的RDD，都会作为一个DStream处理。

### 案例实操

1. 需求：循环创建几个RDD，将RDD放入队列。通过SparkStream创建Dstream，计算WordCount
2. 编写代码

*/\*\**

*\* Author: Felix  
 \* Date: 2020/2/20  
 \* Desc: 通过RDD队列的方式创建DStream  
 \*/*

**object** Spark02\_DStreamCreate\_RDDQueue {

**def** main(args: Array[String]): Unit = {  
 // 创建Spark配置信息对象  
 **val** conf = **new** SparkConf().setMaster("local[\*]").setAppName("RDDStream")  
 // 创建SparkStreamingContext  
 **val** ssc = **new** StreamingContext(conf, *Seconds*(3))  
 // 创建RDD队列  
 **val** rddQueue = **new** mutable.Queue[RDD[Int]]()  
 // 创建QueueInputDStream  
 **val** inputStream = ssc.queueStream(rddQueue,oneAtATime = **false**)  
 // 处理队列中的RDD数据  
 **val** mappedStream = inputStream.map((\_,1))  
 **val** reducedStream = mappedStream.reduceByKey(\_ + \_)  
 // 打印结果  
 reducedStream.print()  
 // 启动任务  
 ssc.start()  
 // 循环创建并向RDD队列中放入RDD  
 **for** (i <- 1 to 5) {  
 rddQueue += ssc.sparkContext.makeRDD(1 to 5, 10)  
 Thread.*sleep*(2000)  
 }  
 ssc.awaitTermination()  
 }  
}

1. 结果展示

-------------------------------------------

Time: 1582192449000 ms

-------------------------------------------

(1,1)

(2,1)

(3,1)

(4,1)

(5,1)

-------------------------------------------

Time: 1582192452000 ms

-------------------------------------------

(1,2)

(2,2)

(3,2)

(4,2)

(5,2)

-------------------------------------------

Time: 1582192455000 ms

-------------------------------------------

(1,1)

(2,1)

(3,1)

(4,1)

(5,1)

-------------------------------------------

Time: 1582192458000 ms

-------------------------------------------

(1,1)

(2,1)

(3,1)

(4,1)

(5,1)

-------------------------------------------

Time: 1582192461000 ms

-------------------------------------------

## 自定义数据源

### 用法及说明

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

### 案例实操

1. 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。
2. 自定义数据源

*/\*\**

*\* Author: Felix  
 \* Date: 2020/2/20  
 \* Desc: 自定义数据源  
 \*/*

**class** MyReceiver(host: String, port: Int) **extends** Receiver[String](StorageLevel.*MEMORY\_ONLY*) {

//创建一个Socket  
 **private var** *socket*: Socket = \_  
  
 //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") {  
 setDaemon(**true**)  
 **override def** run() { receive() }  
 }.start()  
 }  
  
 //读数据并将数据发送给Spark  
 **def** receive(): Unit = {  
 **try** {  
 *socket* = **new** Socket(host, port)  
 //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*))  
 //定义一个变量，用来接收端口传过来的数据  
 **var** input: String = **null** //读取数据 循环发送数据给Spark 一般要想结束发送特定的数据 如："==END=="  
 **while** ((input = reader.readLine())!=**null**) {  
 store(input)  
 }  
 } **catch** {  
 **case** e: ConnectException =>  
 restart(s"Error connecting to **$**host:**$**port", e)  
 **return** }  
 }  
  
 **override def** onStop(): Unit = {  
 **if**(*socket* != **null** ){  
 *socket*.close()  
 *socket* = **null** }  
 }  
}

1. 使用自定义的数据源采集数据

**object** Spark03\_CustomerReceiver {

**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** MyReceiver("hadoop202", 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()  
 }  
}

## Kafka数据源

### 用法及说明

在工程中需要引入Maven依赖spark-streaming-kafka-0-8\_2.11来使用它。包内提供的 KafkaUtils对象可以在StreamingContext和JavaStreamingContext中以你的Kafka消息创建出 DStream。

两个核心类：KafkaUtils（高级API）、KafkaCluster（低级API）

### 案例实操

1. 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。
2. 虚拟机上操作Kafka并创建相关主题

* 启动Zookeeper
* 启动Kafka

[atguigu@hadoop202 ~]$ myka start

* 查询Topic是否存在

[atguigu@hadoop202 kafka]$ cd /opt/module/kafka/

[atguigu@hadoop202 kafka]$ bin/kafka-topics.sh --zookeeper hadoop202:2181 --list

* 如果不存在创建Topic

[atguigu@hadoop202 kafka]$ bin/kafka-topics.sh --zookeeper hadoop202:2181 --create --topic my-bak --partitions 3 --replication-factor 2

Created topic "my-bak".

* 准备生产消息

[atguigu@hadoop202 kafka]$ bin/kafka-console-producer.sh --broker-list hadoop202:9092 --topic my-bak

>

1. 导入依赖

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

1. 高级API方式1

*/\*\**

*\* Author: Felix  
 \* Date: 2020/2/21  
 \* Desc: SparkStreaming读取Kafka数据 高级API方式1  
 \*/*

**object** Spark04\_Kafka\_High1 {

**def** main(args: Array[String]): Unit = {  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("HighKafka")  
 **val** ssc = **new** StreamingContext(conf, *Seconds*(3))  
  
 //kafka参数声明  
 **val** brokers = "hadoop202:9092,hadoop203:9092,hadoop204:9092"  
 **val** topic = "my-bak"  
 **val** group = "bigdata"  
 **val** deserialization = "org.apache.kafka.common.serialization.StringDeserializer"  
 **val** kafkaParams = *Map*(  
 ConsumerConfig.*GROUP\_ID\_CONFIG* -> group,  
 ConsumerConfig.*BOOTSTRAP\_SERVERS\_CONFIG* -> brokers,  
 ConsumerConfig.*KEY\_DESERIALIZER\_CLASS\_CONFIG* -> deserialization,  
 ConsumerConfig.*VALUE\_DESERIALIZER\_CLASS\_CONFIG* -> deserialization  
 )  
 //创建DS  
 **val** kafkaDS: InputDStream[(String, String)] = KafkaUtils.*createDirectStream*[String, String, StringDecoder, StringDecoder](ssc, kafkaParams, *Set*(topic))  
 //wordCount  
 **val** resDS: DStream[(String, Int)] = kafkaDS.map(\_.\_2).flatMap(\_.split(" ")).map((\_,1)).reduceByKey(\_+\_)  
 //打印输出  
 resDS.print()  
  
 ssc.start()  
 ssc.awaitTermination()  
 }  
}

运行程序，通过kafka生产者生成数据，查看效果

1. 高级API方式2

*/\*\**

*\* Author: Felix  
 \* Date: 2020/2/21  
 \* Desc: SparkStreaming读取Kafka数据 高级API方式2  
 \*/*

**object** Spark05\_Kafka\_High2 {

**def** main(args: Array[String]): Unit = {  
 **val** ssc: StreamingContext = StreamingContext.*getActiveOrCreate*("D:\\dev\\workspace\\my-bak\\spark-bak\\cp", () => *createSSC*())  
 ssc.start()  
 ssc.awaitTermination()  
 }  
 **def** createSSC(): StreamingContext = {  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("HighKafka")  
 **val** ssc = **new** StreamingContext(conf, *Seconds*(3))  
 // 偏移量保存在 checkpoint 中, 可以从上次的位置接着消费  
 ssc.checkpoint("D:\\dev\\workspace\\my-bak\\spark-bak\\cp")  
 //kafka参数声明  
 **val** brokers = "hadoop202:9092,hadoop203:9092,hadoop204:9092"  
 **val** topic = "my-bak"  
 **val** group = "bigdata"  
 **val** deserialization = "org.apache.kafka.common.serialization.StringDeserializer"  
 **val** kafkaParams = *Map*(  
 "zookeeper.connect" -> "hadoop202:2181,hadoop203:2181,hadoop204:2181",  
 ConsumerConfig.*GROUP\_ID\_CONFIG* -> group,  
 ConsumerConfig.*BOOTSTRAP\_SERVERS\_CONFIG* -> brokers,  
 ConsumerConfig.*KEY\_DESERIALIZER\_CLASS\_CONFIG* -> deserialization,  
 ConsumerConfig.*VALUE\_DESERIALIZER\_CLASS\_CONFIG* -> deserialization  
 )  
 //创建DS  
 **val** kafkaDS = KafkaUtils.*createDirectStream*[String, String, StringDecoder, StringDecoder]( ssc, kafkaParams, *Set*(topic))  
 //wordCount  
 **val** resDS: DStream[(String, Int)] = kafkaDS.map(\_.\_2).flatMap(\_.split(" ")).map((\_,1)).reduceByKey(\_+\_)  
 //打印输出  
 resDS.print()  
 ssc  
 }  
}

运行程序，通过kafka生产者生成数据，查看效果

1. 低级API方式

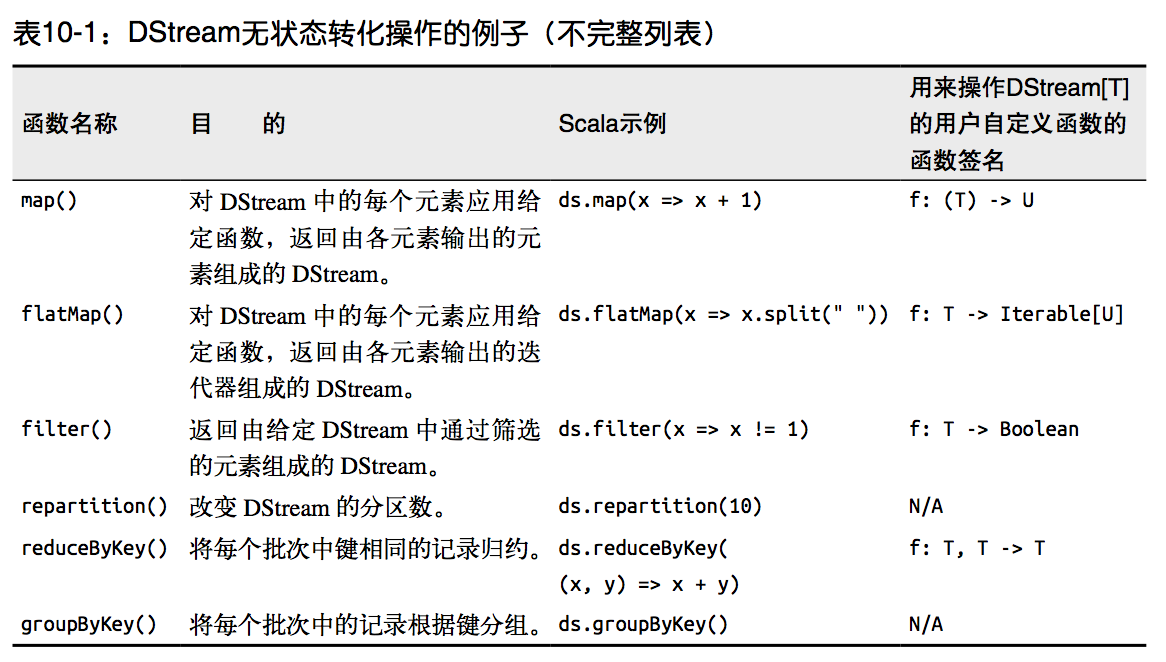
*/\*\*  
 \* Author: Felix  
 \* Date: 2020/2/21  
 \* Desc: SparkStreaming读取Kafka数据 低级API  
 \*/***object** Spark09\_Kafka\_Low {  
 **def** main(args: Array[String]): Unit = {  
  
 //创建SparkConf对象  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("KafkaStreaming")  
  
 //创建StreamingContext对象  
 **val** ssc: StreamingContext = **new** StreamingContext(conf, *Seconds*(3))  
  
 //kafka参数声明  
 **val** brokers = "hadoop202:9092,hadoop203:9092,hadoop204:9092"  
 **val** topic = "my-bak"  
 **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")  
 }  
 }  
 }  
 }  
}

# DStream转换

DStream上的操作与RDD的类似，分为Transformations（转换）和Output Operations（输出）两种，此外转换操作中还有一些比较特殊的算子，如：updateStateByKey()、transform()以及各种Window相关的算子。

## 无状态转化操作

无状态转化操作就是把简单的RDD转化操作应用到每个批次上，也就是转化DStream中的每一个RDD。部分无状态转化操作列在了下表中



需要记住的是，尽管这些函数看起来像作用在整个流上一样，但事实上每个DStream在内部是由许多RDD（批次）组成，且无状态转化操作是分别应用到每个RDD上的。

例如：reduceByKey()会归约每个时间区间中的数据，但不会归约不同区间之间的数据。在之前的wordcount程序中，我们只会统计几秒内接收到的数据的单词个数，而不会累加。

### Transform

Transform允许DStream上执行任意的RDD-to-RDD函数。即使这些函数并没有在DStream的API中暴露出来，通过该函数可以方便的扩展Spark API。该函数每一批次调度一次。其实也就是对DStream中的RDD应用转换。

*/\*\**

*\* Author: Felix  
 \* Date: 2020/2/21  
 \* Desc: 无状态转换-transform  
 \*/*

**object** Spark06\_Nostate\_Transform {

**def** main(args: Array[String]): Unit = {  
 //创建SparkConf  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("WordCount")  
 //创建StreamingContext  
 **val** ssc = **new** StreamingContext(conf, *Seconds*(3))  
 //创建DStream  
 **val** lineDStream: ReceiverInputDStream[String] = ssc.socketTextStream("hadoop202", 9999)  
 //转换为RDD操作  
 **val** wordAndCountDStream: DStream[(String, Int)] = lineDStream.transform(rdd => {  
 **val** words: RDD[String] = rdd.flatMap(\_.split(" "))  
 **val** wordAndOne: RDD[(String, Int)] = words.map((\_, 1))  
 **val** value: RDD[(String, Int)] = wordAndOne.reduceByKey(\_ + \_)  
 value  
 })

//打印  
 wordAndCountDStream.print  
 //启动  
 ssc.start()  
 ssc.awaitTermination()  
 }  
}

## 有状态转化操作

### UpdateStateByKey

UpdateStateByKey算子用于将历史结果应用到当前批次，该操作允许在使用新信息不断更新状态的同时能够保留他的状态。

有时，我们需要在DStream中跨批次维护状态(例如流计算中累加wordcount)。针对这种情况，updateStateByKey()为我们提供了对一个状态变量的访问，用于键值对形式的DStream。给定一个由(键，事件)对构成的 DStream，并传递一个指定如何根据新的事件更新每个键对应状态的函数，它可以构建出一个新的 DStream，其内部数据为(键，状态) 对。

UpdateStateByKey() 的结果会是一个新的DStream，其内部的RDD 序列是由每个时间区间对应的(键，状态)对组成的。

为使用这个功能，需要做下面两步：

1. 定义状态，状态可以是一个任意的数据类型。

2. 定义状态更新函数，用此函数阐明如何使用之前的状态和来自输入流的新值对状态进行更新。

使用updateStateByKey需要对检查点目录进行配置，会使用检查点来保存状态。

更新版的wordcount

例如：每隔一段时间景点人流量变化（从程序启动开始，在原有递增）

1. 编写代码

*/\*\**

*\* Author: Felix  
 \* Date: 2020/2/21  
 \* Desc: 有状态转换-updateStateByKey  
 \*/*

**object** Spark07\_State\_updateStateByKey {  
 **def** main(args: Array[String]): Unit = {  
 //创建SparkConf  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("WordCount")  
 //创建StreamingContext  
 **val** ssc = **new** StreamingContext(conf, *Seconds*(3))  
 //设置检查点路径 用于保存状态  
 ssc.checkpoint("D:\\dev\\workspace\\my-bak\\spark-bak\\cp")  
 //创建DStream  
 **val** lineDStream: ReceiverInputDStream[String] = ssc.socketTextStream("hadoop202", 9999)  
 //扁平映射  
 **val** flatMapDS: DStream[String] = lineDStream.flatMap(\_.split(" "))  
 //结构转换  
 **val** mapDS: DStream[(String, Int)] = flatMapDS.map((\_,1))  
 //聚合  
 // 注意：DStreasm中reduceByKey只能对当前采集周期（窗口）进行聚合操作，没有状态  
 //val reduceDS: DStream[(String, Int)] = mapDS.reduceByKey(\_+\_)  
 **val** stateDS: DStream[(String, Int)] = mapDS.updateStateByKey(  
 (seq: Seq[Int], state: Option[Int]) => {  
 *Option*(seq.sum + state.getOrElse(0))  
 }  
 )

//打印输出  
 stateDS.print()  
 //启动  
 ssc.start()  
 ssc.awaitTermination()  
 }  
}

1. 启动程序并向9999端口发送数据

[atguigu@hadoop202 ~]$ nc -lk 9999

1. 查看结果为累加

### Window Operations(窗口操作)

Spark Streaming 也提供了窗口计算, 允许执行转换操作作用在一个窗口内的数据。默认情况下, 计算只对一个时间段内的RDD进行, 有了窗口之后, 可以把计算应用到一个指定的窗口内的所有 RDD 上。一个窗口可以包含多个时间段，基于窗口的操作会在一个比StreamingContext的批次间隔更长的时间范围内，通过整合多个批次的结果，计算出整个窗口的结果。所有基于窗口的操作都需要两个参数，分别为窗口时长以及滑动步长。

* 窗口时长：计算内容的时间范围；
* 滑动步长：隔多久触发一次计算。

注意：这两者都必须为采集周期的整数倍。

如下图所示WordCount案例：窗口大小为批次的2倍，滑动步等于批次大小。

例如：一小时人流量的变化，窗口（6秒）和间隔（3秒）不一致，不一定从程序启动开始

需求：WordCount统计 3秒一个批次，窗口6秒，滑步3秒。

*/\*\*  
 \* Author: Felix  
 \* Date: 2020/2/21  
 \* Desc: 有状态转换-window相关操作  
 \*/***object** Spark08\_State\_window {  
 **def** main(args: Array[String]): Unit = {  
 //创建SparkConf  
 **val** sparkConf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("WordCount")  
 //创建StreamingContext  
 **val** ssc = **new** StreamingContext(sparkConf, *Seconds*(3))  
 //设置检查点路径 用于保存状态  
 ssc.checkpoint("D:\\dev\\workspace\\my-bak\\spark-bak\\cp")  
 //创建DStream  
 **val** lineDStream: ReceiverInputDStream[String] = ssc.socketTextStream("hadoop202", 9999)  
 //扁平映射  
 **val** flatMapDS: DStream[String] = lineDStream.flatMap(\_.split(" "))  
 //设置窗口大小，滑动的步长  
 **val** windowDS: DStream[String] = flatMapDS.window(*Seconds*(6),*Seconds*(3))  
 //结构转换  
 **val** mapDS: DStream[(String, Int)] = windowDS.map((\_,1))  
 //聚合  
 **val** reduceDS: DStream[(String, Int)] = mapDS.reduceByKey(\_+\_)  
 reduceDS.print()  
 //启动  
 ssc.start()  
 ssc.awaitTermination()  
 }  
}

### 关于Window的操作还有如下方法：

1. **window(windowLength, slideInterval)**

基于对源DStream窗化的批次进行计算返回一个新的Dstream

1. **countByWindow(windowLength, slideInterval)**

返回一个滑动窗口计数流中的元素个数

1. **countByValueAndWindow()**

返回的DStream则包含窗口中每个值的个数

1. **reduceByWindow(func, windowLength, slideInterval)**

通过使用自定义函数整合滑动区间流元素来创建一个新的单元素流

1. **reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])**

当在一个(K,V)对的DStream上调用此函数，会返回一个新(K,V)对的DStream，此处通过对滑动窗口中批次数据使用reduce函数来整合每个key的value值

1. **reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])**

这个函数是上述函数的变化版本，每个窗口的reduce值都是通过用前一个窗的reduce值来递增计算。通过reduce进入到滑动窗口数据并”反向reduce”离开窗口的旧数据来实现这个操作。如果把3秒的时间窗口当成一个池塘，池塘每一秒都会有鱼游进或者游出，那么第一个函数表示每由进来一条鱼，就在该类鱼的数量上累加。而第二个函数是，每由出去一条鱼，就将该鱼的总数减去一。

# DStream输出

输出操作指定了对流数据经转化操作得到的数据所要执行的操作(例如把结果推入外部数据库或输出到屏幕上)。与RDD中的惰性求值类似，如果一个DStream及其派生出的DStream都没有被执行输出操作，那么这些DStream就都不会被求值。如果StreamingContext中没有设定输出操作，整个context就都不会启动。

## 常用输出操作

1. **print()**

在运行流程序的驱动结点上打印DStream中每一批次数据的最开始10个元素。这用于开发和调试。在Python API中，同样的操作叫print()。

1. **saveAsTextFiles(prefix, [suffix])**

以text文件形式存储这个DStream的内容。每一批次的存储文件名基于参数中的prefix和suffix。”prefix-Time\_IN\_MS[.suffix]”。

1. **saveAsObjectFiles(prefix, [suffix])**

以Java对象序列化的方式将Stream中的数据保存为 SequenceFiles . 每一批次的存储文件名基于参数中的为"prefix-TIME\_IN\_MS[.suffix]". Python中目前不可用。

1. **saveAsHadoopFiles(prefix, [suffix])**

将Stream中的数据保存为 Hadoop files. 每一批次的存储文件名基于参数中的为"prefix-TIME\_IN\_MS[.suffix]"。Python API 中目前不可用。

1. **foreachRDD(func)**

这是最通用的输出操作，即将函数 func 用于产生于 stream的每一个RDD。其中参数传入的函数func应该实现将每一个RDD中数据推送到外部系统，如将RDD存入文件或者通过网络将其写入数据库。

通用的输出操作foreachRDD()，它用来对DStream中的RDD运行任意计算。这和transform() 有些类似，都可以让我们访问任意RDD。在foreachRDD()中，可以重用我们在Spark中实现的所有行动操作。比如，常见的用例之一是把数据写到诸如MySQL的外部数据库中，但是在使用的时候需要注意以下几点

* 连接不能写在driver层面（序列化）；
* 如果写在foreach则每个RDD中的每一条数据都创建，得不偿失；
* 增加foreachPartition，在分区创建（获取）。

# DStream 编程进阶

## 累加器和广播变量

和RDD中的累加器和广播变量的用法完全一样，RDD中怎么用, 这里就怎么用

## DataFrame and SQL Operations

你可以很容易地在流数据上使用 DataFrames 和SQL，你必须使用SparkContext来创建StreamingContext要用的SQLContext。此外，这一过程可以在驱动失效后重启。我们通过创建一个实例化的SQLContext单实例来实现这个工作。

如下例所示，我们对前例WordCount进行修改从而使用DataFrames和SQL来实现。每个RDD被转换为DataFrame，以临时表格配置并用SQL进行查询。

**val** spark = SparkSession.*builder*.config(conf).getOrCreate()  
**import** spark.implicits.\_  
mapDS.foreachRDD(rdd =>{  
 **val** df: DataFrame = rdd.toDF("word", "count")  
 df.createOrReplaceTempView("words")  
 spark.sql("select \* from words").show  
})

## Caching / Persistence

和 RDDs 类似，DStreams 同样允许开发者将流数据保存在内存中。也就是说，在DStream上使用persist()方法将会自动把DStreams中的每个RDD保存在内存中。

当DStream中的数据要被多次计算时，这个非常有用（如在同样数据上的多次操作）。对于像reduceByWindow和reduceByKeyAndWindow以及基于状态的(updateStateByKey)这种操作，保存是隐含默认的。因此，即使开发者没有调用persist()，由基于窗操作产生的DStreams也会自动保存在内存中。

# Spark Streaming项目实战

## 准备数据

本实战项目实时的分析处理用户对广告点击的行为数据。

### 数据生成方式

使用代码的方式持续的生成数据，然后写入到kafka中，然后从kafka消费数据，并对数据根据需求进行分析。

### 模拟数据生成及从kafka中读取数据

模拟出来的数据格式:

时间戳, 地区, 城市, 用户id， 广告id1566035129449, 华南, 深圳, 101, 2

1. 步骤1: 开启集群

启动 zookeeper 和 Kafka

1. 步骤2: 创建 Topic

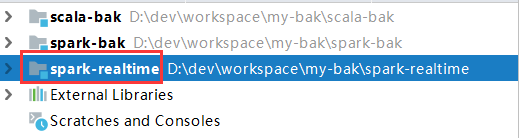
在 kafka 中创建topic: ads\_log

>bin/kafka-topics.sh --zookeeper hadoop202:2181 --list

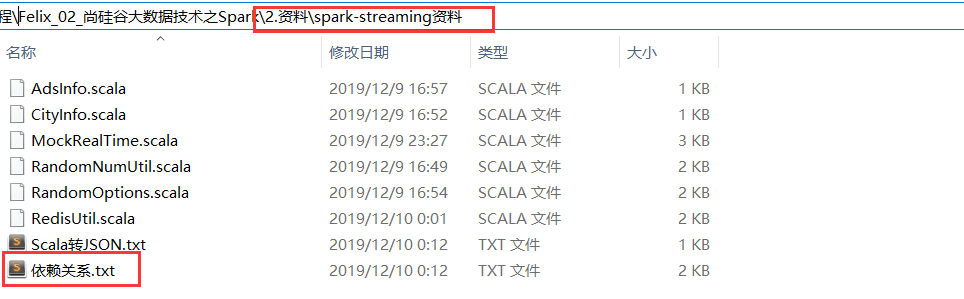
>bin/kafka-topics.sh --zookeeper hadoop202:2181 --create --topic my-ads-bak --partitions 3 --replication-factor 2

1. 步骤3: 产生循环不断的数据到指定的topic

* 创建模块spark-realtime模块



* 从资料\spark-streaming资料\目录下拷贝依赖

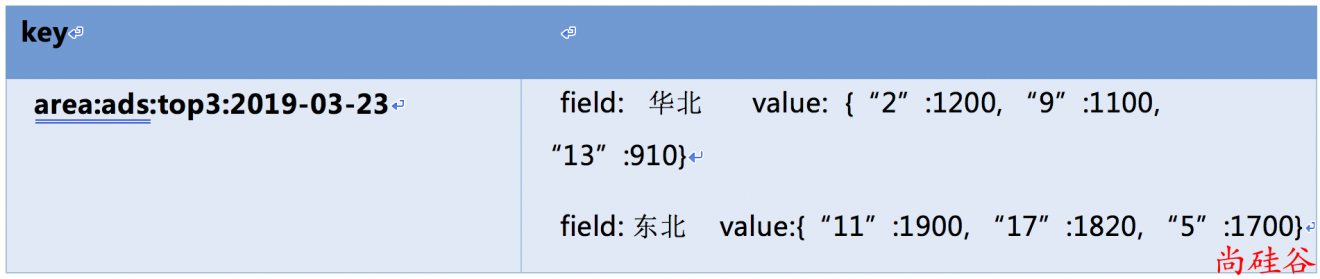


* 从资料\spark-streaming资料\目录下相关的类到Module中
* AdsInfo，样例类
* CityInfo，样例类
* RandomNumUtil，用于生成随机数
* RandomOptions，用于生成带有比重的随机选项
* MockRealTime，生成模拟数据
* RealtimeApp，测试从kafka读取数据（直接拷贝Kafka数据源高级API即可）

1. 步骤 4: 运行RealtimeApp，确认 kafka 中数据是否生成成功

注意：在测试的时候需要修改MockRealTime和RealtimeApp类中的Kafka的配置信息

## 每天每地区热门广告Top3

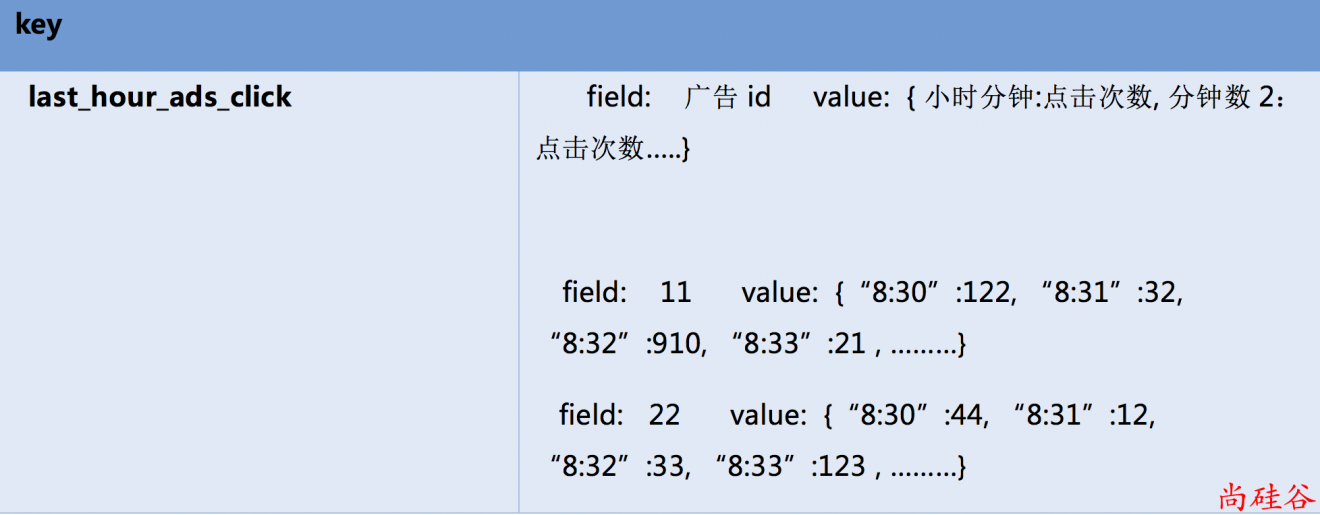
最终数据格式: 存储在 redis 中, 使用 hash 存储 

**object** RealTime\_req1 {

**def** main(args: Array[String]): Unit = {  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("HighKafka")  
 **val** ssc = **new** StreamingContext(conf, *Seconds*(3))  
 //设置检查点  
 ssc.sparkContext.setCheckpointDir("D:\\dev\\workspace\\my-bak\\spark-realtime-bak\\cp")  
  
 //kafka参数声明  
 **val** brokers = "hadoop202:9092,hadoop203:9092,hadoop204:9092"  
 **val** topic = "my-ads-bak"  
 **val** group = "bigdata"  
 **val** deserialization = "org.apache.kafka.common.serialization.StringDeserializer"  
 **val** kafkaParams = *Map*(  
 ConsumerConfig.*GROUP\_ID\_CONFIG* -> group,  
 ConsumerConfig.*BOOTSTRAP\_SERVERS\_CONFIG* -> brokers,  
 ConsumerConfig.*KEY\_DESERIALIZER\_CLASS\_CONFIG* -> deserialization,  
 ConsumerConfig.*VALUE\_DESERIALIZER\_CLASS\_CONFIG* -> deserialization  
 )  
 //创建DS  
 **val** kafkaDS: InputDStream[(String, String)] = KafkaUtils.*createDirectStream*[String, String, StringDecoder, StringDecoder](  
 ssc, kafkaParams, *Set*(topic))  
  
 //测试Kafka中消费数据  
 **val** dataDS: DStream[String] = kafkaDS.map(\_.\_2)  
  
 //打印输出  
 //dataDS.print()  
  
 //==========需求一实现： 每天每地区热门广告 msg = 1584271384370,华南,广州,100,1==========  
 //1.对获取到的Kafka中传递的原始数据  
 //2.将原始数据转换结构 (天\_地区\_广告,点击次数)  
 **val** mapDS = dataDS.map {  
 line => {  
 **val** fields = line.split(",")  
 //格式化时间戳  
 **val** timeStamp = fields(0).toLong  
 **val** day = **new** Date(timeStamp)  
 **val** sdf = **new** SimpleDateFormat("yyyy-MM-dd")  
 **val** dayStr = sdf.format(day)  
 **val** area = fields(1)  
 **val** adv = fields(4)  
 ((dayStr + "\_" + area + "\_" + adv), 1)  
 }  
 }  
 //3.将转换结构后的数据进行聚合处理 (天\_地区\_广告,点击次数sum)  
 //注意：这里要统计每天数据，所有要把每个采集周期的数据都统计，需要保存状态，使用updateStateByKey  
 **val** updateDS = mapDS.updateStateByKey(  
 (seq: Seq[Int], buffer: Option[Int]) => {  
 *Option*(buffer.getOrElse(0) + seq.sum)  
 }  
 )  
 //4.将聚合后的数据进行结构的转换 (天\_地区,(广告,点击次数sum)))  
 **val** mapDS1: DStream[(String, (String, Int))] = updateDS.map {  
 **case** (k, sum) => {  
 **val** ks: Array[String] = k.split("\_")  
 (ks(0) +"\_" +ks(1),(ks(2), sum))  
 }  
 }  
 //5.按照天\_地区对数据进行分组 (时间,Iterator[(地区,(广告,点击次数sum))])  
 **val** groupDS: DStream[(String, Iterable[(String, Int)])] = mapDS1.groupByKey()  
  
 //6.对分组后的数据降序取前三  
 **val** resDS: DStream[(String, List[(String, Int)])] = groupDS.mapValues {  
 datas => {  
 datas.toList.sortBy(-\_.\_2).take(3)  
 }  
 }  
  
 resDS.print()  
  
 ssc.start()  
 ssc.awaitTermination()  
 }  
}

## 最近1小时广告点击量实时统计

需求：统计各广告最近1小时内的点击量趋势，每6s更新一次（各广告最近1小时内各分钟的点击量）



*/\*\*  
 \* 需求二：统计各广告最近1小时内的点击量趋势，每6s更新一次（各广告最近1小时内各分钟的点击量）  
 \* 1.最近一个小时 窗口的长度为1小时  
 \* 2.每6s更新一次 窗口的滑动步长是6s  
 \* 3.各个广告每分钟的点击量 ((advId,hhmm),1)  
 \*/***object** RealTime\_req2 {  
 **def** main(args: Array[String]): Unit = {  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("HighKafka")  
 **val** ssc = **new** StreamingContext(conf, *Seconds*(3))  
  
 //设置检查点  
 ssc.sparkContext.setCheckpointDir("D:\\dev\\workspace\\my-bak\\spark-realtime-bak\\cp")  
  
 //kafka参数声明  
 **val** brokers = "hadoop202:9092,hadoop203:9092,hadoop204:9092"  
 **val** topic = "my-ads-bak"  
 **val** group = "bigdata"  
 **val** deserialization = "org.apache.kafka.common.serialization.StringDeserializer"  
 **val** kafkaParams = *Map*(  
 ConsumerConfig.*GROUP\_ID\_CONFIG* -> group,  
 ConsumerConfig.*BOOTSTRAP\_SERVERS\_CONFIG* -> brokers,  
 ConsumerConfig.*KEY\_DESERIALIZER\_CLASS\_CONFIG* -> deserialization,  
 ConsumerConfig.*VALUE\_DESERIALIZER\_CLASS\_CONFIG* -> deserialization  
 )  
 //创建DS  
 **val** kafkaDS: InputDStream[(String, String)] = KafkaUtils.*createDirectStream*[String, String, StringDecoder, StringDecoder](  
 ssc, kafkaParams, *Set*(topic))  
  
 //测试Kafka中消费数据 msg = 1584271384370,华南,广州,100,1  
 **val** dataDS: DStream[String] = kafkaDS.map(\_.\_2)  
  
 //定义窗口  
 **val** windowDS: DStream[String] = dataDS.window(*Seconds*(12),*Seconds*(3))  
  
 //转换结构为 ((advId,hhmm),1)  
 **val** mapDS: DStream[((String, String), Int)] = windowDS.map {  
 line => {  
 **val** fields: Array[String] = line.split(",")  
 **val** ts: Long = fields(0).toLong  
 **val** day: Date = **new** Date(ts)  
 **val** sdf: SimpleDateFormat = **new** SimpleDateFormat("hh:mm")  
 **val** time = sdf.format(day)  
 ((fields(4), time), 1)  
 }  
 }  
  
 //对数据进行聚合  
 **val** resDS: DStream[((String, String), Int)] = mapDS.reduceByKey(\_+\_)  
 resDS.print()  
  
 ssc.start()  
 ssc.awaitTermination()  
 }  
}