Spark Streaming

一 、Spark Streaming概述

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、ZeroMQ和简单的TCP套接字等等。数据输入后可以用Spark的高度抽象原语(等同于RDD算子)如：map、reduce、window等进行运算。而结果也能保存在很多地方，如HDFS，数据库等。

批处理 VS 流处理区别

	数据形式	数据量	计算延迟
批处理 （离线处理）	静态数据 [在计算之前已经落实的数据]	数据量级大 - GB+	延迟高 分钟|小时
流处理 （实时处理）	动态数据 [实时的数据]	数据量级小 - Byte+	延迟低 ms | s

目前主流流处理框架：Kafka Streaming | Storm | Spark Streaming | Flink

> kafka 消息队列 ---> kafka自带的流处理计算 kafka streaming
>
> Strom ---> 上一代流处理框架
>
> Spark ---> 批量处理框架RDD Spark SQL 、基于RDD构建一个上层的流处理技术 Spark Streaming
>
> 为什么批量框架可以完成流处理，当单个处理批次的数据量足够小、处理速度足够快其实就模拟出来了流处理的特点
>
> Flink ---> 新一代流处理框架

二、DStream入门

> 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(wordCount)

1. 添加依赖

```
 <dependency>
     <groupId>org.apache.spark</groupId>
     <artifactId>spark-core_2.11</artifactId>
     <version>2.4.3</version>
</dependency>

<dependency>
     <groupId>org.apache.spark</groupId>
     <artifactId>spark-streaming_2.11</artifactId>
     <version>2.4.3</version>
</dependency>
```

2. 编写代码

```
object StreamWordCount {

  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.通过监控端口创建DStream，读进来的数据为一行行
    val lineStreams = ssc.socketTextStream("spark0", 9999)

    //将每一行数据做切分，形成一个个单词
    val wordStreams = lineStreams.flatMap(_.split(" "))

    //将单词映射成元组（word,1)
    val wordAndOneStreams = wordStreams.map((_, 1))

    //将相同的单词次数做统计
    val wordAndCountStreams = wordAndOneStreams.reduceByKey(_+_)

    //打印
    wordAndCountStreams.print()

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

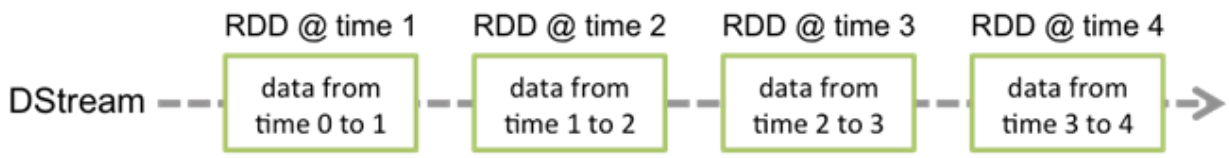
3. 启动程序并通过NetCat发送数据

```
[root@spark0 spark-2.4.3-bin-hadoop2.7]# nc -lk 9999
hello hello hello world
```

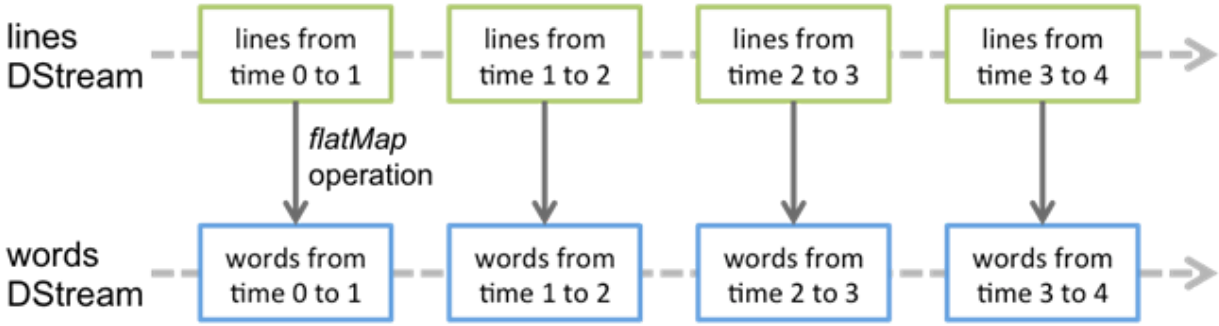
> 需要安装nectcat插件。yum install -y nc
>
> 启动插件 nc -lk 9999

三、WordCount程序解析

Discretized Stream(简称：DStream)是Spark Streaming的基础抽象，代表持续性的数据流和经过各种Spark原语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，如下图：



对数据的操作也是按照RDD为单位来进行的



计算过程由Spark engine来完成

四、DStream创建

开发步骤

- 构建StreamingContext(sparkconf,Seconds(1)).
- 设置数据的Receiver(Basic|Advance)
- 使用DStream（Macro Batch RDD）转换算子
- 启动流计算`ssc.start()`
- 等待系统关闭流计算`ssc.awaitTermination()`

Spark Streaming原生支持一些不同的数据源。一些"核心"数据源已经被打包到Spark Streaming 的 Maven 工件中，而其他的一些则可以通过 spark-streaming-kafka 等附加工件获取。每个接收器都以 Spark 执行器程序中一个长期运行的任务的形式运行，因此会占据分配给应用的 CPU 核心。此外，我们还需要有可用的 CPU 核心来处理数据。这意味着如果要运行多个接收器，就必须至少有和接收器数目相同的核心数，还要加上用来完成计算所需要的核心数。例如，如果我们想要在流计算应用中运行 10 个接收器，那么至少需要为应用分配 11 个 CPU 核心。所以如果在本地模式运行，不要使用local[1]。

1.文件数据源

文件数据流：能够读取所有HDFS API兼容的文件系统文件，通过fileStream方法进行读取，Spark Streaming 将会监控 dataDirectory 目录并不断处理移动进来的文件，记住目前不支持嵌套目录。

```
streamingContext.textFileStream(dataDirectory)
```

注意事项：

1）文件需要有相同的数据格式；

2）文件进入 dataDirectory的方式需要通过移动来实现；

3）一旦文件移动进目录，则不能再修改，即便修改了也不会读取新数据；

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

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

//3.监控文件夹创建DStream
val dirStream = ssc.textFileStream("file:///f:/fileStream")

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

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

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

//7.打印
wordAndCountStreams.print()

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

2.Kafka数据源

> 需求：通过Spark Streaming从Kafka读取数据，并将读取过来的数据做简单计算(Word Count)，最终打印到控制台

1) 添加依赖

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka-0-10_2.11</artifactId>
    <version>2.4.3</version>
</dependency>

<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.0</version>
</dependency>
```

2) 编写代码

```
//1.创建SparkConf并初始化SSC
    val sparkConf: SparkConf = new
SparkConf().setMaster("local[*]").setAppName("KafkaSparkStreaming")
    val ssc = new StreamingContext(sparkConf, Seconds(5))

    //2.定义kafka参数
    val brokers = "kafka1:9092,kafka2:9092,kafka3:9092"
    val consumerGroup = "g1"

    //3.将kafka参数映射为map
    val kafkaParam: Map[String, String] = Map[String, String](
      ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG ->
"org.apache.kafka.common.serialization.StringDeserializer",
      ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG ->
"org.apache.kafka.common.serialization.StringDeserializer",
      ConsumerConfig.GROUP_ID_CONFIG -> consumerGroup,
      ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG -> brokers
    )

    //4.读取kafka中的数据
    val records: InputDStream[ConsumerRecord[String, String]] =
KafkaUtils.createDirectStream(ssc,
      LocationStrategies.PreferConsistent,
      ConsumerStrategies.Subscribe[String, String](List("topica"), kafkaParam))


    //5.对读取的消息进行WordCount处理
    records.map(record=>record.value)
      .flatMap(line=>line.split(" "))
      .map(word => (word, 1))
      .reduceByKey(_ + _)
      .print()

    //6.启动SparkStreaming
    ssc.start()
    ssc.awaitTermination()
```

> ① LocationStrategies.PreferConsistent策略
>
> SparkStreaming读取数据使用 LocationStrategies.PreferConsistent 这种策略， 这种策略会将kafka的分区均匀的分布在集群的Executor之间。【推荐】
>
> ② LocationStrategies.PreferBrokers
>
> 如果Executor在kafka 集群中的某些节点上，可以使用 LocationStrategies.PreferBrokers这种策略 那么当前这个Executor 中的数据会来自当前broker节点。
>
> 如果Executor和Kafka Broker在同一主机,则可使用此策略。
>
> ③ LocationStrategies.PreferFixed
>
> 如果节点之间的分区有明显的分布不均，可以使用 LocationStrategies.PreferFixed 这种策略, 可以通过一个map 指定将topic分区分布在哪些节点中。

五、DStream转换

DStream上的原语与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原语，如：updateStateByKey()、transform()以及各种Window相关的原语。

4.1 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

函数名称	目　　的	Scala示例	用来操作DStream[T]的用户自定义函数的函数签名
map()	对 DStream 中的每个元素应用给定函数，返回由各元素输出的元素组成的 DStream。	ds.map(x => x + 1)	f: (T) -> U
flatMap()	对 DStream 中的每个元素应用给定函数，返回由各元素输出的迭代器组成的 DStream。	ds.flatMap(x => x.split(" "))	f: T -> Iterable[U]
filter()	返回由给定 DStream 中通过筛选的元素组成的 DStream。	ds.filter(x => x != 1)	f: T -> Boolean
repartition()	改变 DStream 的分区数。	ds.repartition(10)	N/A
reduceByKey()	将每个批次中键相同的记录归约。	ds.reduceByKey((x, y) => x + y)	f: T, T -> T
groupByKey()	将每个批次中的记录根据键分组。	ds.groupByKey()	N/A

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。

举个例子，在之前的wordcount程序中，我们只会统计5秒内接收到的数据的单词个数，而不会累加。

4.2 有状态转化操作

1) UpdateStateByKey

UpdateStateByKey原语用于记录历史记录，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。

updateStateByKey操作使得我们可以在用新信息进行更新时保持任意的状态。为使用这个功能，你需要做下面两步：

- 定义状态，状态可以是一个任意的数据类型。
- 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。

使用UpdateStateByKey更新word count代码，实现数据累加统计

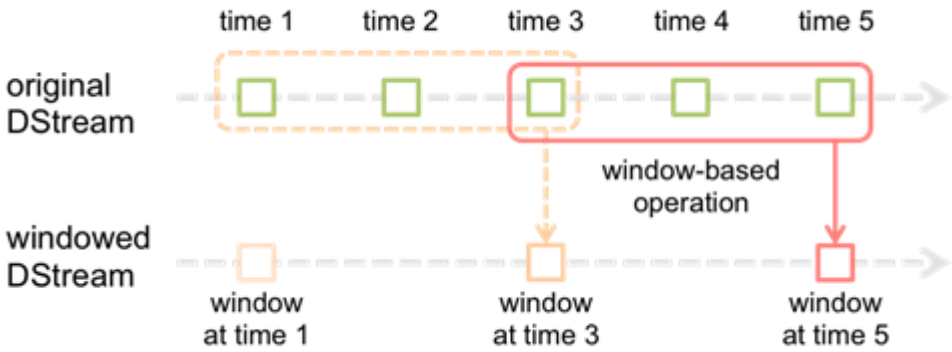
```
// 定义更新状态方法，参数values为当前批次单词频度，state为以往批次单词频度
val updateFunc = (values: Seq[Int], state: Option[Int]) => {
      val currentCount = values.foldLeft(0)(_ + _)
      val previousCount = state.getOrElse(0)
      Some(currentCount + previousCount)
}

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

//2.初始化SparkStreamingContext
val ssc = new StreamingContext(sparkConf, Seconds(5))
ssc.checkpoint("file:///f:/streamCheck")

//3.通过监控端口创建DStream，读进来的数据为一行行
val lines  = ssc.socketTextStream("spark0", 9999)

//将每一行数据做切分，形成一个个单词
val words  = lines.flatMap(_.split(" "))

//将单词映射成元组（word,1)
val pairs  = words.map((_, 1))

// 使用updateStateByKey来更新状态，统计从运行开始以来单词总的次数
val stateDstream = pairs.updateStateByKey[Int](updateFunc)
stateDstream.print()

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

2) Window Operations

前言：通过scala中的`sliding`体会窗口和滑动的概念

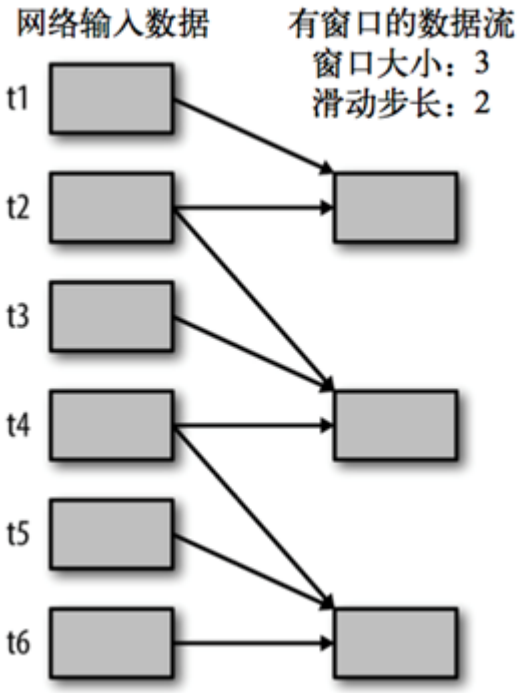
```
//sliding(size: Int)                size代表窗口显示的元素个数
//sliding(size: Int, step: Int)     step代表窗口滑动步长
val iter: Iterator[List[Int]] = List(1,2,3,4,5,6).sliding(3,2)
for (elem <- iter) {
   println(elem)
}
List(1,2,3)
List(3,4,5)
List(5,6)
```

Window Operations可以设置窗口的大小和滑动窗口的间隔来动态的获取当前Streaming的【允许】状态。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。



注意：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，两者都必须是 StreamContext 的批次间隔的整数倍。

窗口时长控制每次计算最近的多少个批次的数据，其实就是最近的 windowDuration/batchInterval 个批次。如果有一个以 10 秒为批次间隔的源 DStream，要创建一个最近 30 秒的时间窗口(即最近 3 个批次)，就应当把 windowDuration 设为 30 秒。而滑动步长的默认值与批次间隔相等，用来控制对新的 DStream 进行计算的间隔。如果源 DStream 批次间隔为 10 秒，并且我们只希望每两个批次计算一次窗口结果， 就应该把滑动步长设置为 20 秒。



WordCount第三版：5秒一个批次，窗口时长10秒，步长5秒。

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

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

//3.通过监控端口创建DStream，读进来的数据为一行行
val lines  = ssc.socketTextStream("spark0", 9999)

//5秒一个批次，窗口10秒，滑步5秒。
val value: DStream[String] = lines.window(Seconds(10),Seconds(5))

//将每一行数据做切分，形成一个个单词
val words  = value.flatMap(_.split(" "))

//将单词映射成元组（word,1)
val pairs  = words.map((_, 1))

val wordCounts =pairs.reduceByKey(_+_)

wordCounts.print()

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

关于Window的操作有如下原语：

（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

（2）countByWindow(windowLength, slideInterval)：返回一个滑动窗口计数流中的元素。

```
countByWindow(窗口大小，滑动步长) = .widnow(窗口大小，滑动步长)  + .count()
```

（3）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。

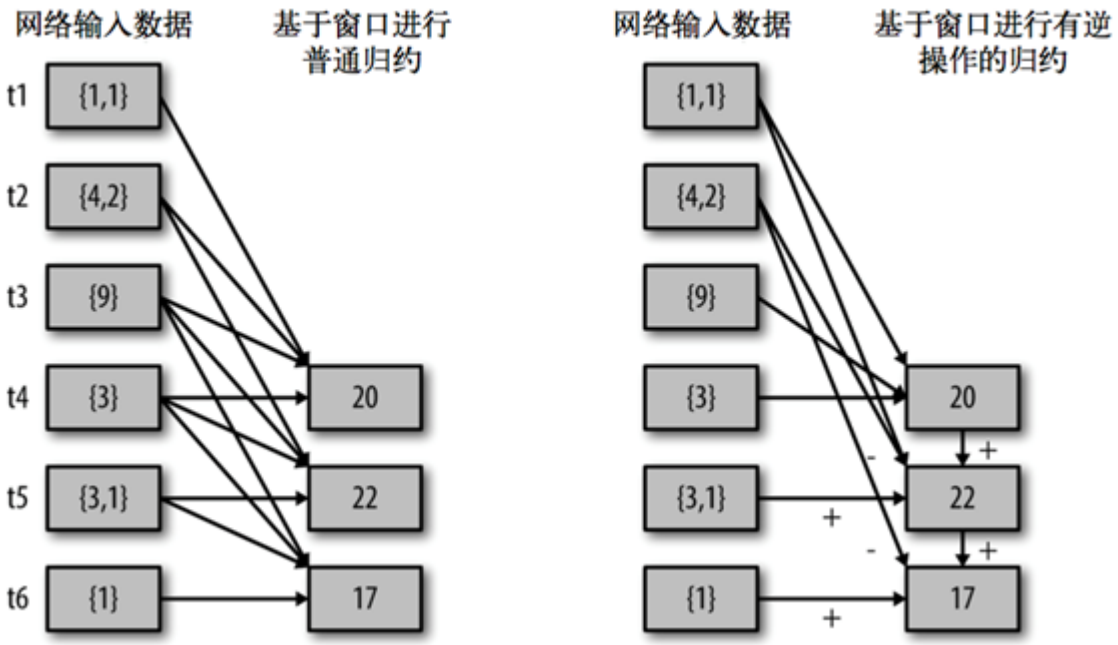
```
reduceByWindow()  = .window()  +  .reduce()
```

（4）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。Note:默认情况下，这个操作使用Spark的默认数量并行任务(本地是2)，在集群模式中依据配置属性(spark.default.parallelism)来做grouping。你可以通过设置可选参数numTasks来设置不同数量的tasks。

（5）reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])：这个函数是上述函数的更高效版本，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。一个例子是随着窗口滑动对keys的“加”“减”计数。通过前边介绍可以想到，这个函数只适用于”可逆的reduce函数”，也就是这些reduce函数有相应的”反reduce”函数(以参数invFunc形式传入)。如前述函数，reduce任务的数量通过可选参数来配置。注意：为了使用这个操作，检查点必须可用。

（6）countByValueAndWindow(windowLength,slideInterval, [numTasks])：对(K,V)对的DStream调用，返回(K,Long)对的新DStream，其中每个key的值是其在滑动窗口中频率。如上，可配置reduce任务数量。

reduceByWindow() 和 reduceByKeyAndWindow() 让我们可以对每个窗口更高效地进行归约操作。它们接收一个归约函数，在整个窗口上执行，比如 +。除此以外，它们还有一种特殊形式，通过只考虑新进入窗口的数据和离开窗口的数据，让 Spark 增量计算归约结果。这种特殊形式需要提供归约函数的一个逆函数，比 如 + 对应的逆函数为 -。对于较大的窗口，提供逆函数可以大大提高执行效率




```
val ipDStream = accessLogsDStream.map(logEntry => (logEntry.getIpAddress(), 1))
val ipCountDStream = ipDStream.reduceByKeyAndWindow(
  {(x, y) => x + y},            // 加上新进入窗口的批次中的元素
  {(x, y) => x - y},            // 移除离开窗口的老批次中的元素
  Seconds(30),                  // 窗口时长
  Seconds(10))                  // 滑动步长
```

4.3 Transform操作

Transform操作允许任意RDD-to-RDD类型的函数被应用在一个DStream上。通过它可以在DStream上使用任何没有在DStream API中暴露的任意RDD操作。

比如说，DStream API中，并没有提供将一个DStream中的每个batch，与一个特定的RDD进行join的操作。但是我们自己就可以使用transform操作来实现该功能。

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

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

val blackList: RDD[(String, Boolean)] = ssc.sparkContext.makeRDD(List(("zhangsan",true),
("lisi",true)))

//3.通过监控端口创建DStream，读进来的数据为一行行
val dStream: ReceiverInputDStream[String] = ssc.socketTextStream("spark0", 9999)
```

```
//模拟用户日志 zhangsan#20
dStream.map(v=>(v.split("#")(0),v)).transform(rdd=>{
    val rdd2: RDD[(String, (String, Option[Boolean]))] = rdd.leftOuterJoin(blackList)
    rdd2.filter(v=>{
            v._2._2.getOrElse(false) == false
        }).map(x => (x._1,x._2._1))
}).print()

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

六、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。

输出操作如下：

（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。

（2）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]”.

```
例如：res.saveAsTextFiles("file:///f:/result/a")
生成的目录为 a-时间戳
```

（3）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]".

（4）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]"。

（5）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。注意：函数func在运行流应用的驱动中被执行，同时其中一般函数RDD操作从而强制其对于流RDD的运算。

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。 注意：

（1）连接不能写在driver层面；

（2）如果写在foreach则每个RDD都创建，得不偿失；

（3）增加foreachPartition，在分区创建。

```
create table t_words(
   id     int primary key auto_increment,
   words  varchar(50),
   c      int,
   time   timestamp
)
```

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

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

val dStream: ReceiverInputDStream[String] = ssc.socketTextStream("spark0",9999)

dStream.window(Seconds(15),Seconds(10))

val res: DStream[(String, Int)] = dStream.flatMap(_.split(" "))
      .map((_, 1))
      .reduceByKey(_ + _)

res.foreachRDD((rdd,time)=>{
    rdd.foreachPartition(f=>{
        val conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test1",
                                               "root","123456")
        f.foreach(v=>{
           val ps: PreparedStatement = conn.prepareStatement("insert into t_words
values(null,?,?,?)")
           ps.setString(1,v._1)
           ps.setInt(2,v._2)
           ps.setTimestamp(3,new Timestamp(time.milliseconds))
           ps.executeUpdate()
           ps.close()
        })
    })
})

ssc.start()
ssc.awaitTermination()
```