> 压缩信息-提炼出核心观点

SparkStreaming之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原语，如：updateStateByKey()、transform()以及各种Window相关的原语。

无状态转换操作

> 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。注意，针对键值对的DStream转化操作(比如reduceByKey())要添加import StreamingContext._才能在Scala中使用。

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。

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

需求

> 统计流中每个批次的词频

代码实现

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

object Transform {

  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))

    //创建DStream
    val lineDStream: ReceiverInputDStream[String] =
ssc.socketTextStream("hadoop102", 9999)

    // 转换为RDD操作
    // 注意这里的rdd是一批的哦，一定要牢记这一点
    // 批处理就是汇聚一批数据，然后对这批数据执行逻辑
    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(_ + _)

    //这里返回的类型是RDD[(String, Int)]
    value
    })

    //打印
    wordAndCountDStream.print

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

  }

}
```

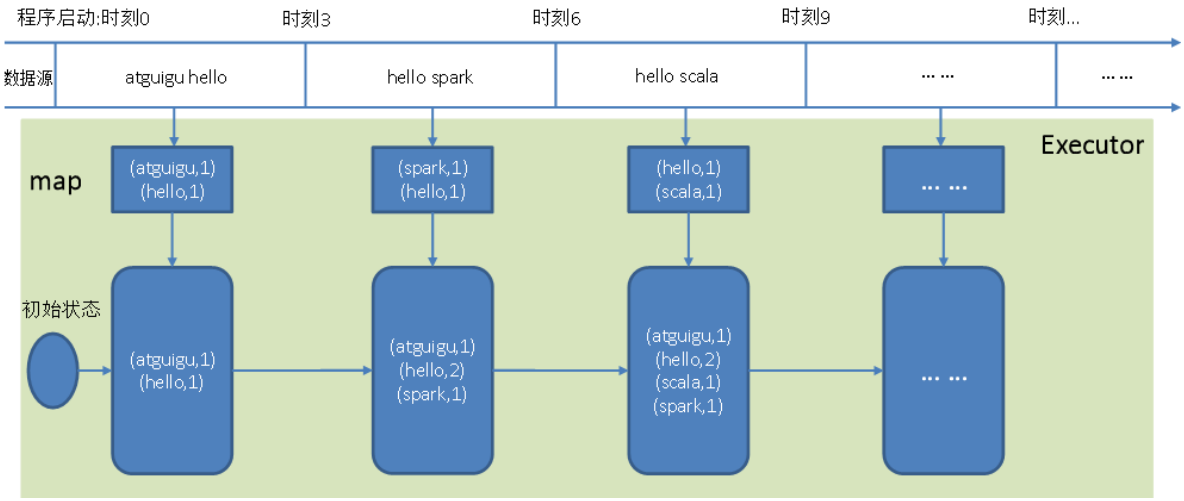
有状态转换操作

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，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。



这里的状态和Flink那里的状态编程其实是差不多的，都要维护中间状态。所以大数据中很多概念都是互通的，这也是为什么我们知识越多学习越快的原因，更好的学习方法可以搜索我的公众号：FocusBigData

代码实现

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

object WorldCount {

  def main(args: Array[String]) {

    // 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
    // scala的牛逼之处函数当作变量使用
    val updateFunc = (values: Seq[Int], state: Option[Int]) => {
      val currentCount = values.foldLeft(0)(_ + _)
      val previousCount = state.getOrElse(0)
      Some(currentCount + previousCount)
    }

    val conf = new
SparkConf().setMaster("local[*]").setAppName("NetworkWordCount")
    val ssc = new StreamingContext(conf, Seconds(3))
    ssc.checkpoint("./ck")

    // Create a DStream that will connect to hostname:port, like hadoop102:9999
    val lines = ssc.socketTextStream("hadoop102", 9999)

    // Split each line into words
    val words = lines.flatMap(_.split(" "))

    //import org.apache.spark.streaming.StreamingContext._ // not necessary
since Spark 1.3
    // Count each word in each batch
    val pairs = words.map(word => (word, 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()

    ssc.start()             // Start the computation
    ssc.awaitTermination()  // Wait for the computation to terminate
```

```
    //ssc.stop()
  }

}
```

WindowOperations

> 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Steaming的允许状态。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。

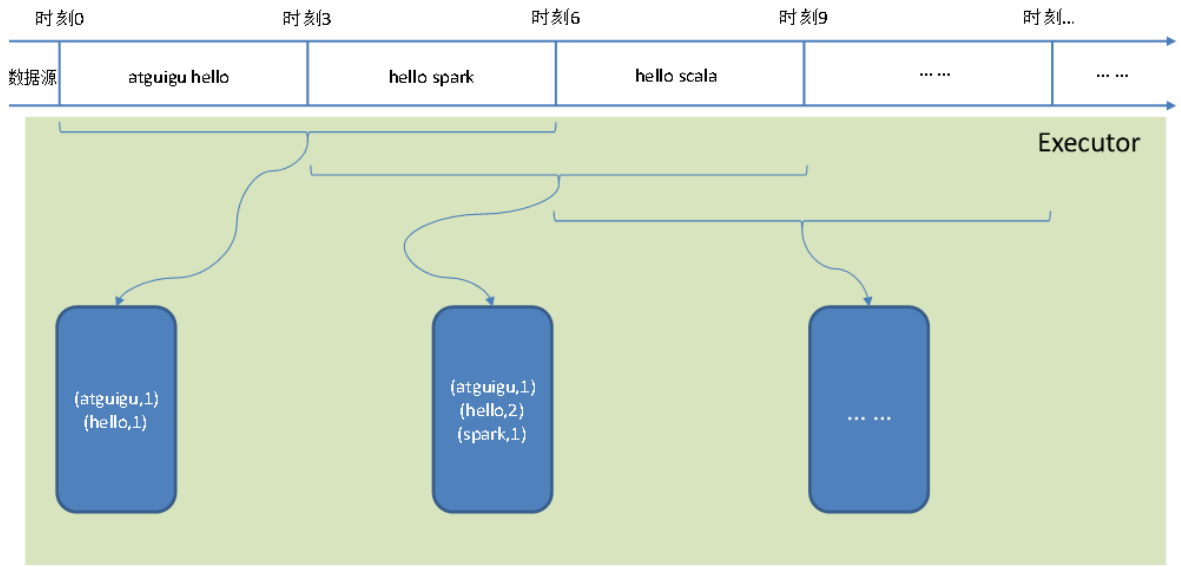
（1）窗口时长：计算内容的时间范围；

（2）滑动步长：隔多久触发一次计算。

注意：这两者都必须为批次大小的整数倍。

这里理解滑动窗口：用时间窗口举例，窗口时长10s，滑动步长5s

> 每隔5秒，计算前10秒的条数



代码实现

> 每隔5s，汇聚前10s的窗口数据

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

object WorldCount {

  def main(args: Array[String]) {

    val conf = new
SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
    val ssc = new StreamingContext(conf, Seconds(3))
    ssc.checkpoint("./ck")

    // Create a DStream that will connect to hostname:port, like localhost:9999
    val lines = ssc.socketTextStream("hadoop102", 9999)

    // Split each line into words
    val words = lines.flatMap(_.split(" "))
```

```
// Count each word in each batch
    val pairs = words.map(word => (word, 1))

    val wordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a +
b),Seconds(10), Seconds(5))

    // Print the first ten elements of each RDD generated in this DStream to the
console
    wordCounts.print()

    ssc.start()             // Start the computation
    ssc.awaitTermination()  // Wait for the computation to terminate
   }

}
```

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；

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

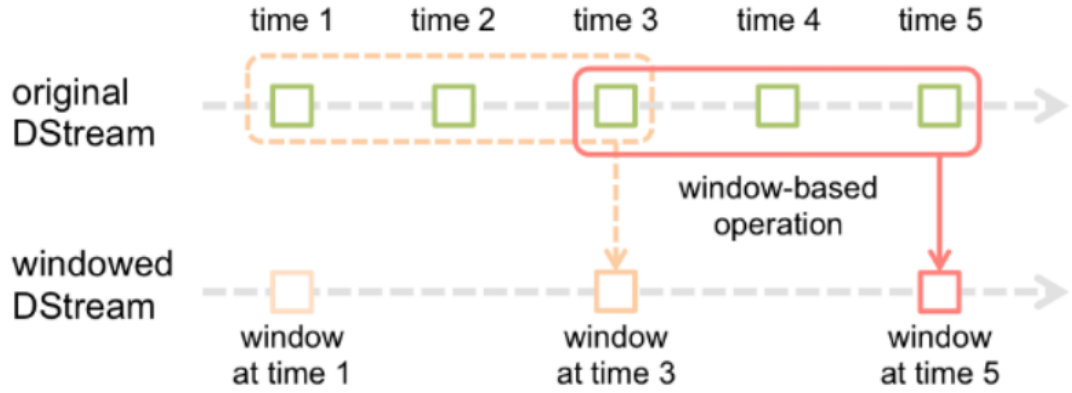
（3）reduceByWindow(func, windowLength, slideInterval): 通过使用自定义函数整合滑动区间流元素来创建一个新的单元素流；

（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。

（5）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。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。

这里来讲一下后面的两个方法，还是老样子拿wordcount举例

需求：每隔2s统计前3s词频数据并且要和后面的数据进行聚合，如下图所示



要实现这个需求很多同学直接会想到`reduceByKeyWindow(_+_,Seconds(3),Seconds(2))`，但是要考虑重叠的数据问题！！！，统计词频的时候绝不能重复统计，那么我们就可以使用它的重载方法了`reduceByKeyWindow(_+_,_-_,Seconds(3),Seconds(2))`

看看官网的描述，我觉得说的很清楚了

> 原因是，其中每个窗口的 reduce 值，使用前一个窗口的 reduce 值递增计算，得到当前前窗口的 reduce 值，然后减去 前一个窗口 失效的值

很多同学可能每个字都看得懂但连起来就看不懂了，很正常，其实原因就是概念不清晰

递增计算：就是不断叠加值

失效值：就是不再使用的值

看上面那幅图得出

window1 = time1 + time2 + time3 【这个相信大家都能看懂】

window2 = window1 + time4 + time5 - time1 - time2 【思考递增计算是啥，失效设值是啥】

= time3 + time4 + time5 【最终聚合结果】

代码实现

```
val ipDStream = accessLogsDStream.map(logEntry => (logEntry.getIpAddress(), 1))
val ipCountDStream = ipDStream.reduceByKeyAndWindow(
  {(x, y) => x + y},
  {(x, y) => x - y},
  Seconds(30),
  Seconds(10))
```

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。在Python API中，同样的操作叫print()。

（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]”。

（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]". Python中目前不可用。

（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]"。Python API 中目前不可用。

（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。

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（序列化）；

（2）如果写在foreach则每个RDD中的每一条数据都创建，得不偿失；

（3）增加foreachPartition，在分区创建（获取）。