# **Spark Streaming**

## 常用的流处理框架

* Apache Storm
* Spark Streaming
* Apache Samza
* Apache Flink

## **Spark Streaming概述**

Spark Streaming 是Spark核心API的一个扩展，可以实现高吞吐量的、具备容错机制的实时流数据的处理。支持从多种数据源获取数据，包括Kafk、Flume、Twitter、ZeroMQ、Kinesis 以及TCP sockets，从数据源获取数据之后，可以使用诸如map、reduce、join和window等高级函数进行复杂算法的处理。最后还可以将处理结果存储到文件系统，数据库和现场仪表盘。在“One Stack rule them all”的基础上，还可以使用Spark的其他子框架，如集群学习、图计算等，对流数据进行处理。

Spark Streaming处理的数据流图：

[](http://images0.cnblogs.com/blog/107289/201508/211408290197622.jpg)

Spark的各个子框架，都是基于核心Spark的，Spark Streaming在内部的处理机制是，接收实时流的数据，并根据一定的时间间隔拆分成一批批的数据，然后通过Spark Engine处理这些批数据，最终得到处理后的一批批结果数据。

## 对应的批数据，在Spark内核对应一个RDD实例，因此，对应流数据的DStream可以看成是一组RDDs，即RDD的一个序列。通俗点理解的话，在流数据分成一批一批后，通过一个先进先出的队列，然后 Spark Engine从该队列中依次取出一个个批数据，把批数据封装成一个RDD，然后进行处理，这是一个典型的生产者消费者模型，对应的就有生产者消费者模型的问题，即如何协调生产速率和消费速率。

## **Spark Streaming入门**

作为构建于Spark之上的应用框架，Spark Streaming承袭了Spark的编程风格，对于已经了解Spark的用户来说能够快速地上手。接下来以Spark Streaming官方提供的WordCount代码为例来介绍Spark Streaming的使用方式。

|  |
| --- |
| import org.apache.spark.\_  import org.apache.spark.streaming.\_  import org.apache.spark.streaming.StreamingContext.\_    // Create a local StreamingContext with two working thread and batch interval of 1 second.  // The master requires 2 cores to prevent from a starvation scenario.  val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")  val ssc = new StreamingContext(conf, Seconds(1))    // Create a DStream that will connect to hostname:port, like localhost:9999  val lines = ssc.socketTextStream("localhost", 9999)    // Split each line into words  val words = lines.flatMap(\_.split(" "))  import org.apache.spark.streaming.StreamingContext.\_  // Count each word in each batch  val pairs = words.map(word => (word, 1))  val wordCounts = pairs.reduceByKey(\_ + \_)    // 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 |

**1.创建StreamingContext对象** 同Spark初始化需要创建SparkContext对象一样，使用Spark Streaming就需要创建StreamingContext对象。创建StreamingContext对象所需的参数与SparkContext基本一致，包括指明Master，设定名称(如NetworkWordCount)。需要注意的是参数Seconds(1)，Spark Streaming需要指定处理数据的时间间隔，如上例所示的1s，那么Spark Streaming会以1s为时间窗口进行数据处理。此参数需要根据用户的需求和集群的处理能力进行适当的设置；

**2.创建InputDStream**如同Storm的Spout，Spark Streaming需要指明数据源。如上例所示的socketTextStream，Spark Streaming以socket连接作为数据源读取数据。当然Spark Streaming支持多种不同的数据源，包括Kafka、 Flume、HDFS/S3、Kinesis和Twitter等数据源；

**3.操作DStream**对于从数据源得到的DStream，用户可以在其基础上进行各种操作，如上例所示的操作就是一个典型的WordCount执行流程：对于当前时间窗口内从数据源得到的数据首先进行分割，然后利用Map和ReduceByKey方法进行计算，当然最后还有使用print()方法输出结果；

**4.启动Spark Streaming**之前所作的所有步骤只是创建了执行流程，程序没有真正连接上数据源，也没有对数据进行任何操作，只是设定好了所有的执行计划，当ssc.start()启动后程序才真正进行所有预期的操作。

至此对于Spark Streaming的如何使用有了一个大概的印象，在后面的章节我们会通过源代码深入探究一下Spark Streaming的执行流程。

我们先来一个小程序

|  |
| --- |
| **import** org.apache.spark.storage.StorageLevel **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.spark.{SparkConf, SparkContext}  **object** NetWorkWordCount {   **def** main(args: Array[String]): Unit = {  **if**(args.length<2){  System.*out*.println(**"Usage:NetworkWordCount<hostname><port>"**)  System.*exit*(1)  }  **val** conf = **new** SparkConf().setAppName(**"NetWorkWordCount"**).setMaster(**"local[2]"**)  *//val sc = new SparkContext(conf)* **val** ssc = **new** StreamingContext(conf,*Seconds*(30))  **val** lines=ssc.socketTextStream(args(0),args(1).toInt,StorageLevel.*MEMORY\_AND\_DISK\_SER*)  **val** words=lines.flatMap(\_.split(**" "**))  **val** wordCounts = words.map(x=>(x,1)).reduceByKey(\_+\_)  wordCounts.print()  ssc.start()  ssc.awaitTermination()  } } |

首先，在IDEA中配置两个输入参数分别为：hostname和port 。配置好参数后运行我们的程序。

然后，在控制台中输入以下内容：

$nc -lk 9999 #netcat(linux小工具)作为我们数据服务器

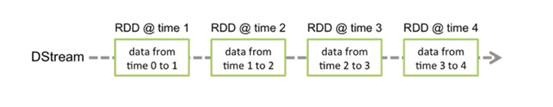
Hello world

Hadoop hbase hive linux

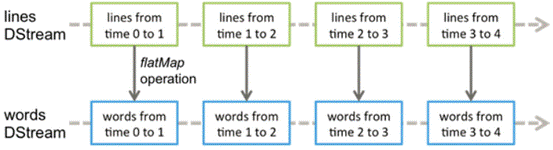
## **Spark Streaming Dstream**

### 1、编程模型

DStream（Discretized Stream）作为Spark Streaming的基础抽象，它代表持续性的数据流。这些数据流既可以通过外部输入源赖获取，也可以通过现有的Dstream的transformation操作来获得。在内部实现上，DStream由一组时间序列上连续的RDD来表示。每个RDD都包含了自己特定时间间隔内的数据流。如下图所示。

[](http://images0.cnblogs.com/blog/107289/201508/211408350358425.jpg)

对DStream中数据的各种操作也是映射到内部的RDD上来进行的，如下图所示，对Dtream的操作可以通过RDD的transformation生成新的DStream。这里的执行引擎是Spark。

[](http://images0.cnblogs.com/blog/107289/201508/211408358168298.gif)

### 2、DStream的输入源

在Spark Streaming中所有的操作都是基于流的，而输入源是这一系列操作的起点。输入 DStreams 和 DStreams 接收的流都代表输入数据流的来源，在Spark Streaming 提供两种内置数据流来源：

* **基础来源** 在 StreamingContext API 中直接可用的来源。例如：文件系统、Socket（套接字）连接和 Akka actors；
* **高级来源** 如 Kafka、Flume、Kinesis、Twitter 等，可以通过额外的实用工具类创建。

#### **（1）基础来源**

在前面分析怎样使用Spark Streaming的例子中我们已看到ssc.socketTextStream()方法，可以通过 TCP 套接字连接，从从文本数据中创建了一个 DStream。除了套接字，StreamingContext 的API还提供了方法从文件和 Akka actors 中创建 DStreams作为输入源。

Spark Streaming提供了streamingContext.fileStream(dataDirectory)方法可以从任何文件系统(如：HDFS、S3、NFS 等）的文件中读取数据，然后创建一个DStream。Spark Streaming 监控 dataDirectory 目录和在该目录下任何文件被创建处理(不支持在嵌套目录下写文件)。需要注意的是：读取的必须是具有相同的数据格式的文件；创建的文件必须在 dataDirectory 目录下，并通过自动移动或重命名成数据目录；文件一旦移动就不能被改变，如果文件被不断追加,新的数据将不会被阅读。对于简单的文本文，可以使用一个简单的方法streamingContext.textFileStream(dataDirectory)来读取数据。

Spark Streaming也可以基于自定义 Actors 的流创建DStream ，通过 Akka actors 接受数据流，使用方法streamingContext.actorStream(actorProps, actor-name)。Spark Streaming使用 streamingContext.queueStream(queueOfRDDs)方法可以创建基于 RDD 队列的DStream，每个RDD 队列将被视为 DStream 中一块数据流进行加工处理。

#### **（2）高级来源**

这一类的来源需要外部 non-Spark 库的接口，其中一些有复杂的依赖关系(如 Kafka、Flume)。因此通过这些来源创建 DStreams 需要明确其依赖。例如，如果想创建一个使用 Twitter tweets 的数据的DStream 流，必须按以下步骤来做：

1）在 SBT 或 Maven工程里添加 spark-streaming-twitter\_2.10 依赖。

2）开发：导入 TwitterUtils 包，通过 TwitterUtils.createStream 方法创建一个DStream。

3）部署：添加所有依赖的 jar 包(包括依赖的spark-streaming-twitter\_2.10 及其依赖)，然后部署应用程序。

需要注意的是，这些高级的来源一般在Spark Shell中不可用，因此基于这些高级来源的应用不能在Spark Shell中进行测试。如果你必须在Spark shell中使用它们，你需要下载相应的Maven工程的Jar依赖并添加到类路径中。

其中一些高级来源如下：

* **Twitter** Spark Streaming的TwitterUtils工具类使用Twitter4j，Twitter4J 库支持通过任何方法提供身份验证信息，你可以得到公众的流，或得到基于关键词过滤流。
* **Flume** Spark Streaming可以从Flume中接受数据。
* **Kafka** Spark Streaming可以从Kafka中接受数据。
* **Kinesis** Spark Streaming可以从Kinesis中接受数据。

需要重申的一点是在开始编写自己的 SparkStreaming 程序之前，一定要将高级来源依赖的Jar添加到SBT 或 Maven 项目相应的artifact中。常见的输入源和其对应的Jar包如下图所示。

[](http://images0.cnblogs.com/blog/107289/201508/211408388948072.jpg)

另外，输入DStream也可以创建自定义的数据源，需要做的就是实现一个用户定义的接收器。

### 3、DStream的操作

与RDD类似，DStream也提供了自己的一系列操作方法，这些操作可以分成三类：普通的转换操作、窗口转换操作和输出操作。

#### **（1）普通的转换操作**

普通的转换操作如下表所示：

|  |  |
| --- | --- |
| **转换** | **描述** |
| **map**(func) | 源 DStream的每个元素通过函数func返回一个新的DStream。 |
| **flatMap**(func) | 类似与map操作，不同的是每个输入元素可以被映射出0或者更多的输出元素。 |
| **filter**(func) | 在源DSTREAM上选择Func函数返回仅为true的元素,最终返回一个新的DSTREAM 。 |
| **repartition**(numPartitions) | 通过输入的参数numPartitions的值来改变DStream的分区大小。 |
| **union**(otherStream) | 返回一个包含源DStream与其他 DStream的元素合并后的新DSTREAM。 |
| **count**() | 对源DStream内部的所含有的RDD的元素数量进行计数，返回一个内部的RDD只包含一个元素的DStreaam。 |
| **reduce**(func) | 使用函数func（有两个参数并返回一个结果）将源DStream 中每个RDD的元素进行聚 合操作,返回一个内部所包含的RDD只有一个元素的新DStream。 |
| **countByValue**() | 计算DStream中每个RDD内的元素出现的频次并返回新的DStream[(K,Long)]，其中K是RDD中元素的类型，Long是元素出现的频次。 |
| **reduceByKey**(func, [numTasks]) | 当一个类型为（K，V）键值对的DStream被调用的时候,返回类型为类型为（K，V）键值对的新 DStream,其中每个键的值V都是使用聚合函数func汇总。注意：默认情况下，使用 Spark的默认并行度提交任务（本地模式下并行度为2，集群模式下位8），可以通过配置numTasks设置不同的并行任务数。 |
| **join**(otherStream, [numTasks]) | 当被调用类型分别为（K，V）和（K，W）键值对的2个DStream时，返回类型为（K，（V，W））键值对的一个新 DSTREAM。 |
| **cogroup**(otherStream, [numTasks]) | 当被调用的两个DStream分别含有(K, V) 和(K, W)键值对时,返回一个(K, Seq[V], Seq[W])类型的新的DStream。 |
| **transform**(func) | 通过对源DStream的每RDD应用RDD-to-RDD函数返回一个新的DStream，这可以用来在DStream做任意RDD操作。 |
| **updateStateByKey**(func) | 返回一个新状态的DStream,其中每个键的状态是根据键的前一个状态和键的新值应用给定函数func后的更新。这个方法可以被用来维持每个键的任何状态数据。 |

在上面列出的这些操作中，transform()方法和updateStateByKey()方法值得我们深入的探讨一下：

* **transform(func)操作**

该transform操作（转换操作）连同其其类似的 transformWith操作允许DStream 上应用任意RDD-to-RDD函数。它可以被应用于未在DStream API 中暴露任何的RDD操作。例如，在每批次的数据流与另一数据集的连接功能不直接暴露在DStream API 中，但可以轻松地使用transform操作来做到这一点，这使得DStream的功能非常强大。例如，你可以通过连接预先计算的垃圾邮件信息的输入数据流（可能也有Spark生成的），然后基于此做实时数据清理的筛选，如下面官方提供的伪代码所示。事实上，也可以在transform方法中使用机器学习和图形计算的算法。

* **updateStateByKey操作**

该 updateStateByKey 操作可以让你保持任意状态，同时不断有新的信息进行更新。要使用此功能，必须进行两个步骤 ：

（1）  定义状态 - 状态可以是任意的数据类型。

（2）  定义状态更新函数 - 用一个函数指定如何使用先前的状态和从输入流中获取的新值 更新状态。

让我们用一个例子来说明，假设你要进行文本数据流中单词计数。在这里，正在运行的计数是状态而且它是一个整数。我们定义了更新功能如下：

[](http://images0.cnblogs.com/blog/107289/201508/211408411135716.jpg)

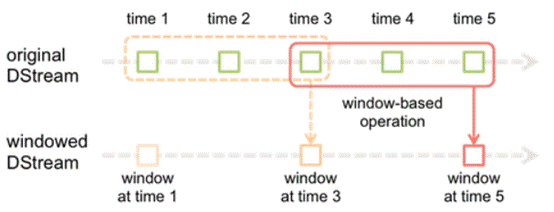
此函数应用于含有键值对的DStream中（如前面的示例中，在DStream中含有（word，1）键值对）。它会针对里面的每个元素（如wordCount中的word）调用一下更新函数，newValues是最新的值，runningCount是之前的值。

[clip_image018](http://images0.cnblogs.com/blog/107289/201508/211408421605902.jpg)

#### **（2）窗口转换操作**

Spark Streaming 还提供了窗口的计算，它允许你通过滑动窗口对数据进行转换，窗口转换操作如下：

|  |  |
| --- | --- |
| **转换** | **描述** |
| **window**(windowLength, slideInterval) | 返回一个基于源DStream的窗口批次计算后得到新的DStream。 |
| **countByWindow**(windowLength,slideInterval) | 返回基于滑动窗口的DStream中的元素的数量。 |
| **reduceByWindow**(func, windowLength,slideInterval) | 基于滑动窗口对源DStream中的元素进行聚合操作，得到一个新的DStream。 |
| **reduceByKeyAndWindow**(func,windowLength,slideInterval, [numTasks]) | 基于滑动窗口对（K，V）键值对类型的DStream中的值按K使用聚合函数func进行聚合操作，得到一个新的DStream。 |
| **countByValueAndWindow**(windowLength,slideInterval, [numTasks]) | 基于滑动窗口计算源DStream中每个RDD内每个元素出现的频次并返回DStream[(K,Long)]，其中K是RDD中元素的类型，Long是元素频次。与countByValue一样，reduce任务的数量可以通过一个可选参数进行配置。 |

[](http://images0.cnblogs.com/blog/107289/201508/211408429889788.gif)

批处理间隔示意图

在Spark Streaming中，数据处理是按批进行的，而数据采集是逐条进行的，因此在Spark Streaming中会先设置好批处理间隔（batch duration），当超过批处理间隔的时候就会把采集到的数据汇总起来成为一批数据交给系统去处理。

对于窗口操作而言，在其窗口内部会有N个批处理数据，批处理数据的大小由窗口间隔（window duration）决定，而窗口间隔指的就是窗口的持续时间，在窗口操作中，只有窗口的长度满足了才会触发批数据的处理。除了窗口的长度，窗口操作还有另一个重要的参数就是滑动间隔（slide duration），它指的是经过多长时间窗口滑动一次形成新的窗口，滑动窗口默认情况下和批次间隔的相同，而窗口间隔一般设置的要比它们两个大。在这里必须注意的一点是滑动间隔和窗口间隔的大小一定得设置为批处理间隔的整数倍。

如批处理间隔示意图所示，批处理间隔是1个时间单位，窗口间隔是3个时间单位，滑动间隔是2个时间单位。对于初始的窗口time 1-time 3，只有窗口间隔满足了才触发数据的处理。这里需要注意的一点是，初始的窗口有可能流入的数据没有撑满，但是随着时间的推进，窗口最终会被撑满。当每个2个时间单位，窗口滑动一次后，会有新的数据流入窗口，这时窗口会移去最早的两个时间单位的数据，而与最新的两个时间单位的数据进行汇总形成新的窗口（time3-time5）。

对于窗口操作，批处理间隔、窗口间隔和滑动间隔是非常重要的三个时间概念，是理解窗口操作的关键所在。

**窗口函数实验：**

**通过nc -lk 9999在Linux系统上发送数据。**

**执行命令：**

|  |
| --- |
| **spark-submit --executor-memory 215m /root/SparkTest.jar 0.0.0.0 9999 /root/spark\_checkpoint** |

* **Windows**

|  |
| --- |
| **import** org.apache.log4j.{Level, Logger} **import** org.apache.spark.storage.StorageLevel **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.spark.{SparkConf, SparkContext}  **object** windowOnStreaming {  **def** main(args: Array[String]) {  */\*\*  \* this is test of Streaming operations-----window  \*/* Logger.*getLogger*(**"org.apache.spark"**).setLevel(Level.*ERROR*)  Logger.*getLogger*(**"org.eclipse.jetty.Server"**).setLevel(Level.*OFF*)   **val** conf = **new** SparkConf().setAppName(**"the Window operation of SparK Streaming"**).setMaster(**"local[2]"**)  **val** sc = **new** SparkContext(conf)  **val** ssc = **new** StreamingContext(sc,*Seconds*(2))    *//set the Checkpoint directory* ssc.checkpoint(args(2))   *//get the socket Streaming data* **val** socketStreaming = ssc.socketTextStream(args(0),args(1).toInt,StorageLevel.*MEMORY\_AND\_DISK\_SER*)   **val** data = socketStreaming.map(x =>(x,1))  *//def window(windowDuration: Duration): DStream[T]* **val** getedData1 = data.window(*Seconds*(6),*Seconds*(4))  *println*(**"windowDuration only : "**)  getedData1.print()  *//ssame as  // def window(windowDuration: Duration, slideDuration: Duration): DStream[T]  //val getedData2 = data.window(Seconds(9),Seconds(3))  //println("Duration and SlideDuration : ")  //getedData2.print()* ssc.start()  ssc.awaitTermination()  }  } |

* **countByWindow**

|  |
| --- |
| **import** org.apache.log4j.{Level, Logger} **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.spark.{SparkConf, SparkContext}  */\*\*  \* Created by root on 6/23/16.  \*/* **object** countByWindow {  **def** main(args: Array[String]) {   */\*\*  \* this is test of Streaming operations-----countByWindow  \*/* Logger.*getLogger*(**"org.apache.spark"**).setLevel(Level.*ERROR*)  Logger.*getLogger*(**"org.eclipse.jetty.Server"**).setLevel(Level.*OFF*)   **val** conf = **new** SparkConf().setAppName(**"the reduceByWindow operation of SparK Streaming"**).setMaster(**"local[2]"**)  **val** sc = **new** SparkContext(conf)  **val** ssc = **new** StreamingContext(sc,*Seconds*(2))  *//set the Checkpoint directory* ssc.checkpoint(args(2))   *//get the socket Streaming data* **val** socketStreaming = ssc.socketTextStream(args(0),args(1).toInt)   **val** data = socketStreaming.countByWindow(*Seconds*(6),*Seconds*(2))    *println*(**"countByWindow: count the number of elements"**)  data.print()    ssc.start()  ssc.awaitTermination()   } } |

* **reduceByWindow**

|  |
| --- |
| **import** org.apache.log4j.{Level, Logger} **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.spark.{SparkConf, SparkContext}   **object** reduceByWindow {  **def** main(args: Array[String]) {  */\*\*  \* this is test of Streaming operations-----reduceByWindow  \*/* Logger.*getLogger*(**"org.apache.spark"**).setLevel(Level.*ERROR*)  Logger.*getLogger*(**"org.eclipse.jetty.Server"**).setLevel(Level.*OFF*)   **val** conf = **new** SparkConf().setAppName(**"the reduceByWindow operation of SparK Streaming"**)  **val** sc = **new** SparkContext(conf)  **val** ssc = **new** StreamingContext(sc,*Seconds*(2))  *//set the Checkpoint directory* ssc.checkpoint(args(2))   *//get the socket Streaming data* **val** socketStreaming = ssc.socketTextStream(args(0),args(1).toInt)   *//val data = socketStreaming.reduceByWindow(\_+\_,Seconds(6),Seconds(2))* **val** data = socketStreaming.map(\_.toInt).reduceByWindow(\_-\_,*Seconds*(6),*Seconds*(2))    *println*(**"reduceByWindow: count the number of elements"**)  data.print()    ssc.start()  ssc.awaitTermination()   } } |

* **reduceByKeyAndWindow**

|  |
| --- |
| **package** org.apache.spark.examples.streaming  **import** org.apache.log4j.{Level, Logger} **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.spark.{SparkConf, SparkContext}   **object** reduceByKeyAndWindow {  **def** main(args: Array[String]) {  */\*\*  \* this is test of Streaming operations-----reduceByWindow  \*/* Logger.*getLogger*(**"org.apache.spark"**).setLevel(Level.*ERROR*)  Logger.*getLogger*(**"org.eclipse.jetty.Server"**).setLevel(Level.*OFF*)   **val** conf = **new** SparkConf().setAppName(**"the reduceByWindow operation of SparK Streaming"**)  **val** sc = **new** SparkContext(conf)  **val** ssc = **new** StreamingContext(sc,*Seconds*(2))  *//set the Checkpoint directory* ssc.checkpoint(args(2))   *//get the socket Streaming data* **val** socketStreaming = ssc.socketTextStream(args(0),args(1).toInt)   *//val data = socketStreaming.reduceByWindow(\_+\_,Seconds(6),Seconds(2))* **val** data = socketStreaming.flatMap(\_.toString.split(**" "**)).map(x=>(x,1)).reduceByKeyAndWindow((x:Int,y:Int)=>(x+y),*Seconds*(6),*Seconds*(4))    *//println("reduceByWindow: count the number of elements")* data.print()    ssc.start()  ssc.awaitTermination()   } } |

* **countByValueAndWindow**

|  |
| --- |
| **import** org.apache.log4j.{Level, Logger} **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.spark.{SparkConf, SparkContext}   **object** countByValueAndWindow {  **def** main(args: Array[String]) {  */\*\*  \* this is test of Streaming operations-----reduceByWindow  \*/* Logger.*getLogger*(**"org.apache.spark"**).setLevel(Level.*ERROR*)  Logger.*getLogger*(**"org.eclipse.jetty.Server"**).setLevel(Level.*OFF*)   **val** conf = **new** SparkConf().setAppName(**"the reduceByWindow operation of SparK Streaming"**)  **val** sc = **new** SparkContext(conf)  **val** ssc = **new** StreamingContext(sc,*Seconds*(2))  *//set the Checkpoint directory* ssc.checkpoint(args(2))   *//get the socket Streaming data* **val** socketStreaming = ssc.socketTextStream(args(0),args(1).toInt)   *//val data = socketStreaming.reduceByWindow(\_+\_,Seconds(6),Seconds(2))* **val** data = socketStreaming.flatMap(\_.toString.split(**" "**)).countByValueAndWindow(*Seconds*(6),*Seconds*(4))    *//println("reduceByWindow: count the number of elements")* data.print()    ssc.start()  ssc.awaitTermination()   } } |

#### **（3）输出操作**

Spark Streaming允许DStream的数据被输出到外部系统，如数据库或文件系统。由于输出操作实际上使transformation操作后的数据可以通过外部系统被使用，同时输出操作触发所有DStream的transformation操作的实际执行（类似于RDD操作）。以下表列出了目前主要的输出操作：

|  |  |
| --- | --- |
| **转换** | **描述** |
| **print**() | 在Driver中打印出DStream中数据的前10个元素。 |
| **saveAsTextFiles**(prefix, [suffix]) | 将DStream中的内容以文本的形式保存为文本文件，其中每次批处理间隔内产生的文件以prefix-TIME\_IN\_MS[.suffix]的方式命名。 |
| **saveAsObjectFiles**(prefix, [suffix]) | 将DStream中的内容按对象序列化并且以SequenceFile的格式保存。其中每次批处理间隔内产生的文件以prefix-TIME\_IN\_MS[.suffix]的方式命名。 |
| **saveAsHadoopFiles**(prefix, [suffix]) | 将DStream中的内容以文本的形式保存为Hadoop文件，其中每次批处理间隔内产生的文件以prefix-TIME\_IN\_MS[.suffix]的方式命名。 |
| **foreachRDD**(func) | 最基本的输出操作，将func函数应用于DStream中的RDD上，这个操作会输出数据到外部系统，比如保存RDD到文件或者网络数据库等。需要注意的是func函数是在运行该streaming应用的Driver进程里执行的。 |

dstream.foreachRDD是一个非常强大的输出操作，它允将许数据输出到外部系统。但是 ，如何正确高效地使用这个操作是很重要的，下面展示了如何去避免一些常见的错误。

通常将数据写入到外部系统需要创建一个连接对象（如 TCP连接到远程服务器），并用它来发送数据到远程系统。出于这个目的，开发者可能在不经意间在Spark driver端创建了连接对象，并尝试使用它保存RDD中的记录到Spark worker上，如下面代码：

[](http://images0.cnblogs.com/blog/107289/201508/211408441289003.jpg)

这是不正确的，这需要连接对象进行序列化并从Driver端发送到Worker上。连接对象很少在不同机器间进行这种操作，此错误可能表现为序列化错误（连接对不可序列化），初始化错误（连接对象在需要在Worker 上进行需要初始化） 等等，正确的解决办法是在 worker上创建的连接对象。

通常情况下，创建一个连接对象有时间和资源开销。因此，创建和销毁的每条记录的连接对象可能招致不必要的资源开销，并显著降低系统整体的吞吐量 。一个更好的解决方案是使用rdd.foreachPartition方法创建一个单独的连接对象，然后使用该连接对象输出的所有RDD分区中的数据到外部系统。

 这缓解了创建多条记录连接的开销。最后，还可以进一步通过在多个RDDs/ batches上重用连接对象进行优化。一个保持连接对象的静态池可以重用在多个批处理的RDD上将其输出到外部系统，从而进一步降低了开销。

 需要注意的是，在静态池中的连接应该按需延迟创建，这样可以更有效地把数据发送到外部系统。另外需要要注意的是：DStreams延迟执行的，就像RDD的操作是由actions触发一样。默认情况下，输出操作会按照它们在Streaming应用程序中定义的顺序一个个执行。

## **Spark Streaming编程实战**

### 使用Spark Streaming监控目录上的数据

文件数据流：可以从任何兼容HDFS API（包括HDFS、S3、NFS等）的文件系统，创建方式如下：

streamingContext.fileStream[KeyClass,ValueClass,InputFormatClass](dataDirectory)

SparkStreaming将监控该dataDirectory目录，并处理该目录下任何新建的文件（目前还不支持嵌套目录）。注意：

1. 各个文件数据格式必须一致。
2. dataDirectory中的文件必须通过moving或者renaming来创建。
3. 一旦文件移进dataDirectory之后，就不能再改动，所以如果这个文件后续还有写入，这些新写入数据不会被读取。
4. 另外，文件数据流不是基于接收器的，所以不需要为其单独分配一个CPU core。

对于简单的文本文件，更简单的方式是调用streamingContextFileStream（dataDirectory）。

需求：统计HDFS文件的词频

代码如下：

|  |
| --- |
| **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.spark.{SparkConf, SparkContext}  **object** HdfsWordCount {  **def** main(args: Array[String]): Unit = {  *//两个参数分别为路径和运行模式* **if**(args.length!=3){  System.*out*.println(**"Usage:HdfsWordCount<directory>"**)  System.*exit*(1)  }  **val** input:String = args(0)  **val** master:String = args(1)  **val** timewin:Int = args(2).toInt  **val** conf = **new** SparkConf().setAppName(**"HdfsWordCount"**).setMaster(master)  **val** sc = **new** SparkContext(conf)  *//create streamingContext* **val** streamC = **new** StreamingContext(sc,*Seconds*(timewin))  *//create FileInputDStream read* **val** lines = streamC.textFileStream(input)  **val** line = lines.flatMap(\_.split(**" "**)).map(x=>(x,1)).reduceByKey(\_+\_)  line.print()    streamC.start()  streamC.awaitTermination()  } } |

参数：D:\\甲骨文资料\\Spark师资培训\\data\\input //监控路径

local[2] //应用运行方式

30 //时间窗口

等进程拉起来后，在D:\\甲骨文资料\\Spark师资培训\\data\\input目录里放入文件进行测试。

### 使用Spark Streaming处理带状态的数据

### Spark Streaming整合Flume

Flume是我们大数据中常用的数据采集工具，能够采集各种数据，并能把采集到的数据接入到各种大数据存储和处理软件中。我们今天的实验是把采集到数据接入到我们的SparkStreaming中。

开发及测试步骤如下：

1. Flume配置文件配置（文件名：flume.conf）

|  |
| --- |
| a1.sources = r1  a1.sinks = k1  a1.channels = c1  # Describe/configure the source  a1.sources.r1.type= netcat  a1.sources.r1.channels = c1  a1.sources.r1.bind = 127.0.0.1  a1.sources.r1.port = 9998  # Describe the sink  a1.sinks.k1.type= avro  a1.sinks.k1.hostname = 192.168.85.149  a1.sinks.k1.port = 9999  # Use a channel which buffers events in memory  a1.channels.c1.type= memory  a1.channels.c1.capacity = 1000  a1.channels.c1.transactionCapacity = 100  # Bind the source and sink to the channel  a1.sources.r1.channels = c1  a1.sinks.k1.channel = c1 |

1. Spark Streaming代码

|  |
| --- |
| **import** org.apache.spark.SparkConf **import** org.apache.spark.sql.catalyst.expressions.Second **import** org.apache.spark.storage.StorageLevel **import** org.apache.spark.streaming.flume.FlumeUtils **import** org.apache.spark.streaming.{Seconds, StreamingContext}  **object** FlumePushWordCount {  **def** main(args: Array[String]): Unit = {   **if**(args.length!=4){  System.*out*.println(**"Usage:FlumePushWordCount<host><port><master><output>"**)  System.*exit*(1)  }  **val** host=args(0)  **val** port = args(1)  **val** master = args(2)  **val** output = args(3)  **val** conf = **new** SparkConf().setAppName(**"FlumePushWordCount"**).setMaster(master)  **val** streamC = **new** StreamingContext(conf,*Seconds*(20))  **val** flumeStream = FlumeUtils.*createStream*(streamC,host,port.toInt,StorageLevel.*MEMORY\_AND\_DISK\_SER\_2*)  flumeStream.map(x=>**new** String(x.*event*.getBody.array()).trim).flatMap(x=>x.split(**" "**)).map((\_,1)).reduceByKey(\_+\_).print  streamC.start()  streamC.awaitTermination()   } } |

1. 启动Spark Streaming进程

|  |
| --- |
| spark-submit --master yarn --deploy-mode client --executor-memory 216m --driver-memory 216m /home/cloudera/Desktop/SparkTest.jar 192.168.85.149 9999 |

注：在启动前先调整内存的大小。

1. 启动Flume进程

|  |
| --- |
| flume-ng agent --name a1 --conf /etc/flume-ng/conf --conf-file /etc/flume-ng/conf/flume.conf -Dflume.root.logger=INFO,console& |

1. 启动telnet进程后，输入数据，观察效果

|  |
| --- |
| Telnet 127.0.0.1 9998 |