尚硅谷大数据技术之 SparkStreaming

(作者：尚硅谷大数据研发部)

版本：V1.3

第 1 章 Spark Streaming 概述

1.1 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 的高度抽象原语如：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。另外 Spark Streaming 也能和 MLlib（机器学习）以及 Graphx 完美融合。

和 Spark 基于 RDD 的概念很相似，Spark Streaming 使用离散化流(discretized stream)作为抽象表示，叫作 DStream。DStream 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 DStream 是由这些 RDD 所组成的序列(因此得名“离散化”)。

DStream 可以从各种输入源创建，比如 Flume、Kafka 或者 HDFS。创建出来的 DStream 支持两种操作，一种是转化操作(transformation)，会生成一个新的 DStream，另一种是输出操作(output operation)，可以把数据写入外部系统中。DStream 提供了许多与 RDD 所支持的操作相类似的操作支持，还增加了与时间相关的新操作，比如滑动窗口。

1.2 Spark Streaming 特点

1.易用

Ease of Use

Build applications through high-level operators.

Spark Streaming brings Spark's language-integrated API to stream processing, letting you write streaming jobs the same way you write batch jobs. It supports Java, Scala and Python.

```
TwitterUtils.createStream(...)
    .filter(_.getText.contains("Spark"))
    .countByWindow(Seconds(5))
```

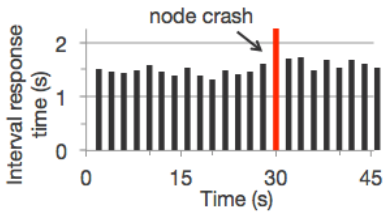
Counting tweets on a sliding window

2.容错

Fault Tolerance

Stateful exactly-once semantics out of the box.

Spark Streaming recovers both lost work and operator state (e.g. sliding windows) out of the box, without any extra code on your part.



3.易整合到 Spark 体系

Spark Integration

Combine streaming with batch and interactive queries.

By running on Spark, Spark Streaming lets you reuse the same code for batch processing, join streams against historical data, or run ad-hoc queries on stream state. Build powerful interactive applications, not just analytics.

```
stream.join(historicCounts).filter {
  case (word, (curCount, oldCount)) =>
    curCount > oldCount
}
```

Find words with higher frequency than historic data

1.3 SparkStreaming 架构

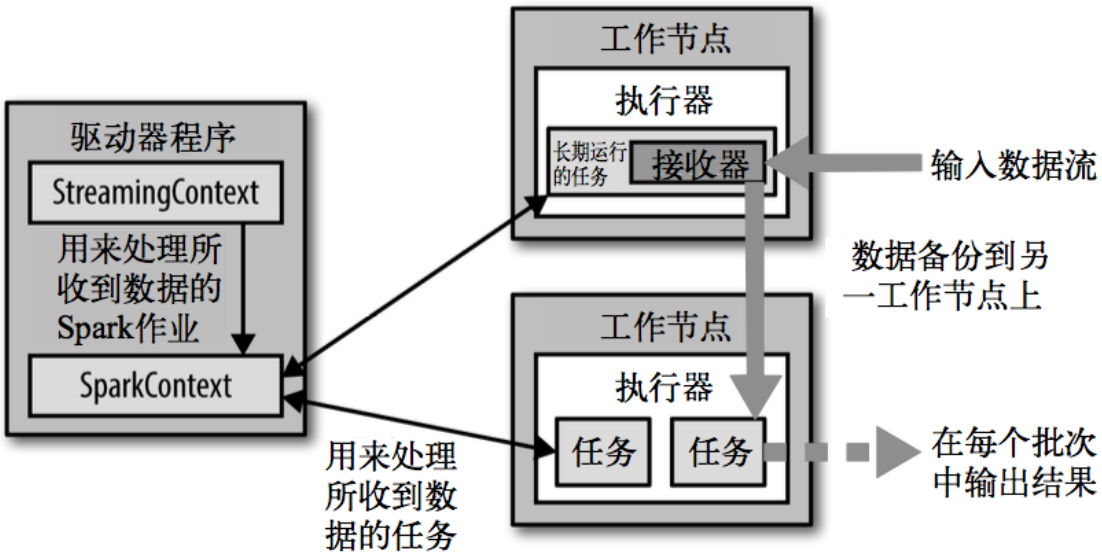


图 1-1 SparkStreaming 架构图

第 2 章 Dstream 入门

2.1 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>
```

3．编写代码

```
package com.atguigu

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

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("hadoop102", 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()
  }
}
```

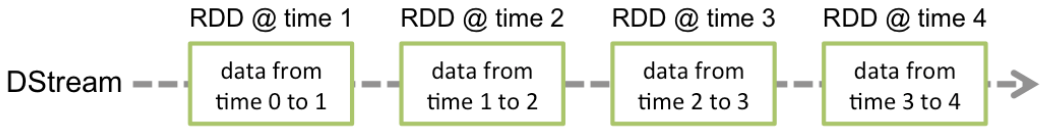
4．启动程序并通过 NetCat 发送数据：

```
[atguigu@hadoop102 spark]$ nc -lk 9999
hello atguigu
```

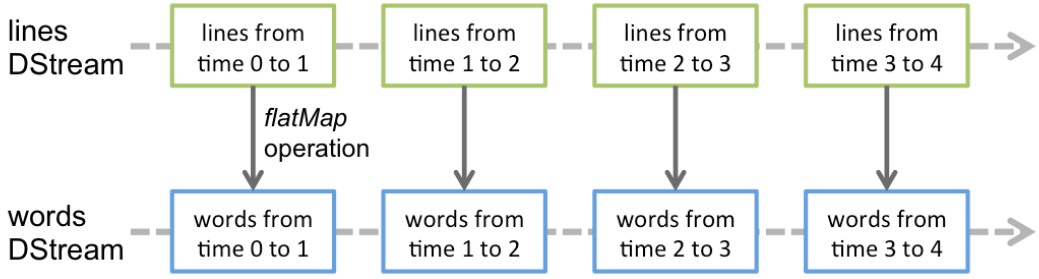
注意：如果程序运行时，log 日志太多，可以将 spark conf 目录下的 log4j 文件里面的日志级别改成 WARN。

2.2 WordCount 解析

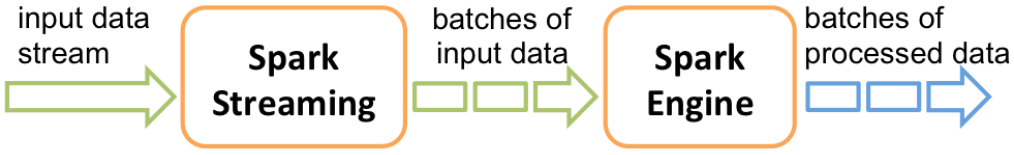
Discretized Stream 是 Spark Streaming 的基础抽象，代表持续性的数据流和经过各种 Spark 原语操作后的结果数据流。在内部实现上，DStream 是一系列连续的 RDD 来表示。每个 RDD 含有一段时间间隔内的数据，如下图：



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



第 3 章 Dstream 创建

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 或者 local[1]。

3.1 文件数据源

3.1.1 用法及说明

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或者重命名来实现；

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；

3.1.2 案例实操

（1）在 HDFS 上建好目录

```
[atguigu@hadoop102 spark]$ hadoop fs -mkdir /fileStream
```

（2）在/opt/module/data 创建三个文件

```
[atguigu@hadoop102 data]$ touch a.tsv
[atguigu@hadoop102 data]$ touch b.tsv
[atguigu@hadoop102 data]$ touch c.tsv

添加如下数据：
Helloatguigu
Hellospark
```

（3）编写代码

```
package com.atguigu

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.监控文件夹创建 DStream
    val dirStream = ssc.textFileStream("hdfs://hadoop102:9000/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()
  }
}
```

（4）启动程序并向 fileStream 目录上传文件

```
[atguigu@hadoop102 data]$ hadoop fs -put ./a.tsv /fileStream
[atguigu@hadoop102 data]$ hadoop fs -put ./b.tsv /fileStream
[atguigu@hadoop102 data]$ hadoop fs -put ./c.tsv /fileStream
```

（5）获取计算结果

```
-------------------------------------------
Time: 1539073810000 ms
-------------------------------------------


-------------------------------------------
Time: 1539073815000 ms
-------------------------------------------
(Hello,4)
(spark,2)
(atguigu,2)

-------------------------------------------
Time: 1539073820000 ms
-------------------------------------------
(Hello,2)
(spark,1)
(atguigu,1)

-------------------------------------------
Time: 1539073825000 ms
-------------------------------------------
```

3.2 RDD 队列

3.2.1 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。

3.2.2 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

```
package com.atguigu

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()

    }
}
```

3）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
-------------------------------------------
```

3.3 自定义数据源

3.3.1 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。

3.3.2 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

```
package com.atguigu

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 = {}
}
```

3）使用自定义的数据源采集数据

```
package com.atguigu

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 wordStreams = lineStream.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 wordAndCount = wordAndOneStreams.reduceByKey(_ + _)

        //7.打印
        wordAndCountStreams.print()

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

3.4 Kafka 数据源

3.4.1 用法及说明

在工程中需要引入 Maven 工件 spark- streaming-kafka_2.10 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。由于 KafkaUtils 可以订阅多个主题，因此它创建出的 DStream 由成对的主题和消息组成。要创建出一个流数据，需要使用 StreamingContext 实例、一个由逗号隔开的 ZooKeeper 主机列表字符串、消费者组的名字(唯一名字)，以及一个从主题到针对这个主题的接收器线程数的映射表来调用 createStream() 方法。

3.4.2 案例实操

1）需求 1：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。

（1）导入依赖

```
<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-streaming-kafka -->
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka_2.11</artifactId>
    <version>1.6.3</version>
</dependency>
```

（2）编写代码

```
package com.atguigu

import kafka.serializer.StringDecoder
import org.apache.kafka.clients.consumer.ConsumerConfig
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.dstream.ReceiverInputDStream
import org.apache.spark.streaming.kafka.KafkaUtils
import org.apache.spark.streaming.{Seconds, StreamingContext}

object KafkaSparkStreaming {

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

    //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 zookeeper = "hadoop102:2181,hadoop103:2181,hadoop104:2181"
    val topic = "source"
    val consumerGroup = "spark"

    //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,
        "zookeeper.connect" -> zookeeper
    )

    //4.通过 KafkaUtil 创建 kafkaDSteam
    val kafkaDSteam: ReceiverInputDStream[(String, String)] = KafkaUtils.createStream[String, String, StringDecoder, StringDecoder](
        ssc,
        kafkaParam,
        Map[String, Int](topic -> 3),
        StorageLevel.MEMORY_ONLY
    )

    //5.对 kafkaDSteam 做计算（WordCount）
    kafkaDSteam.foreachRDD {
        rdd => {
            val word: RDD[String] = rdd.flatMap(_._2.split(" "))
            val wordAndOne: RDD[(String, Int)] = word.map((_, 1))
            val wordAndCount: RDD[(String, Int)] = wordAndOne.reduceByKey(_ + _)
            wordAndCount.collect().foreach(println)
        }
    }

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

第 4 章 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。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。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 1 秒内接收到的数据的单词个数，而不会累加。

无状态转化操作也能在多个 DStream 间整合数据，不过也是在各个时间区间内。例如，键 值对 DStream 拥有和 RDD 一样的与连接相关的转化操作，也就是 cogroup()、join()、leftOuterJoin()等。我们可以在 DStream 上使用这些操作，这样就对每个批次分别执行了对应的 RDD 操作。

我们还可以像在常规的 Spark 中一样使用 DStream 的 union() 操作将它和另一个 DStream 的内容合并起来，也可以使用 StreamingContext.union()来合并多个流。

4.2 有状态转化操作

4.2.1 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：

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：

（1）编写代码

```
package com.atguigu.streaming

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
    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[2]").setAppName("NetworkWordCount")
    val ssc = new StreamingContext(conf, Seconds(3))
    ssc.checkpoint(".")

    // 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()

    //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
    //ssc.stop()
  }

}
```

（2）启动程序并向 9999 端口发送数据

```
[atguigu@hadoop102 kafka]$ nc -lk 9999
```

```
ni shi shui
ni hao ma
```

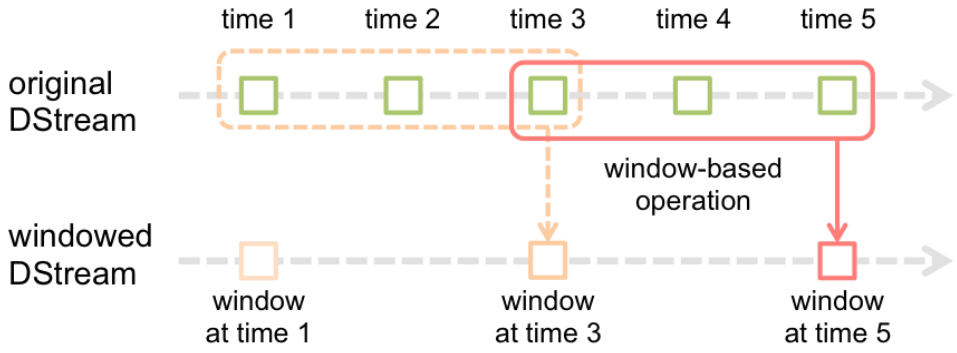
（3）结果展示

```
-------------------------------------------
Time: 1504685175000 ms
-------------------------------------------
-------------------------------------------
Time: 1504685181000 ms
-------------------------------------------
(shi,1)
(shui,1)
(ni,1)
-------------------------------------------
Time: 1504685187000 ms
-------------------------------------------
(shi,1)
(ma,1)
(hao,1)
(shui,1)
(ni,2)
```

4.2.2 Window Operations

Window Operations 有点类似于 Storm 中的 State，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。

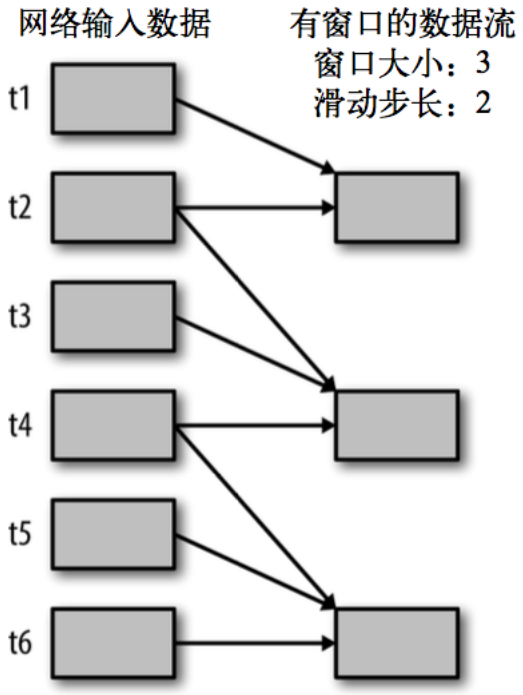
基于窗口的操作会在一个比 StreamingContext 的批次间隔更长的时间范围内，通过整合多个批次的结果，计算出整个窗口的结果。



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，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。

假设，你想拓展前例从而每隔十秒对持续 30 秒的数据生成 word count。为做到这个，我们需要在持续 30 秒数据的(word,1)对 DStream 上应用 reduceByKey。使用操作 reduceByKeyAndWindow.

```
# reduce last 30 seconds of data, every 10 second
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x -y, 30, 20)
```



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。

（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。

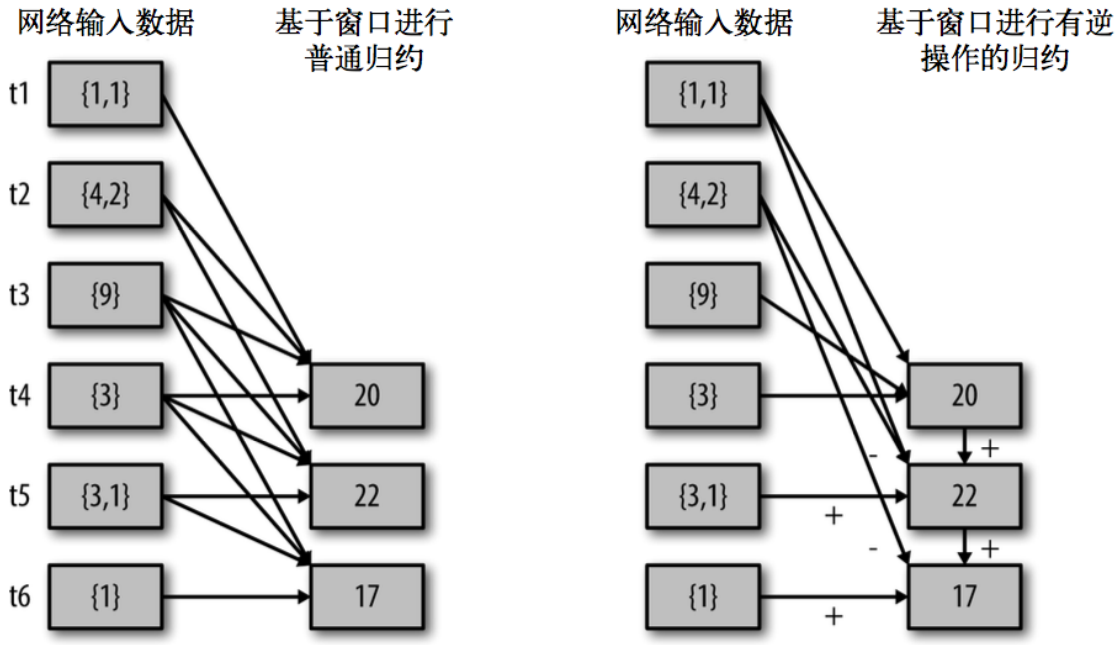
（4）reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])：当在一个(K,V)对的 DStream 上调用此函数，会返回一个新(K,V)对的 DStream，此处通过对滑动窗口中批次数据使用 reduce 函数来整合每个 key 的 value 值。Note:默认情况下，这个操作使用 Spark 的默认数量并行任务(本地是 2)，在集群模式中依据配置属性(spark.default.parallelism)来做 grouping。你可以通过设置可选参数 numTasks 来设置不同数量的 tasks。

（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},
   {(x, y) => x - y},
   Seconds(30),
   Seconds(10))
   //加上新进入窗口的批次中的元素  //移除离开窗口的老批次中的元素  //窗口时长// 滑动步长
```

countByWindow() 和 countByValueAndWindow() 作为对数据进行计数操作的简写。countByWindow()返回一个表示每个窗口中元素个数的 DStream，而 countByValueAndWindow()返回的 DStream 则包含窗口中每个值的个数。

```
val ipDStream = accessLogsDStream.map{entry => entry.getIpAddress()}
val ipAddressRequestCount = ipDStream.countByValueAndWindow(Seconds(30), Seconds(10))
val requestCount = accessLogsDStream.countByWindow(Seconds(30), Seconds(10))
```

WordCount 第三版：3 秒一个批次，窗口 12 秒，滑步 6 秒。

```
package com.atguigu.streaming
```

```
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
    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[2]").setAppName("NetworkWordCount")
    val ssc = new StreamingContext(conf, Seconds(3))
    ssc.checkpoint(".")

    // 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(" "))

    //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))

    val wordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b),Seconds(12), Seconds(6))

    // 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
    //ssc.stop()
  }

}
```

4.3 其他重要操作

4.3.1 Transform

Transform 原语允许 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。

比如下面的例子，在进行单词统计的时候，想要过滤掉 spam 的信息。

```
val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...) // RDD containing spam information

val cleanedDStream = wordCounts.transform { rdd =>
  rdd.join(spamInfoRDD).filter(...) // join data stream with spam information to do data cleaning
  ...
}
```

4.3.2 Join

连接操作（leftOuterJoin, rightOuterJoin, fullOuterJoin 也可以），可以连接 Stream-Stream，windows-stream to windows-stream、stream-dataset

Stream-Stream Joins

```
val stream1: DStream[String, String] = ...
val stream2: DStream[String, String] = ...
val joinedStream = stream1.join(stream2)

val windowedStream1 = stream1.window(Seconds(20))
val windowedStream2 = stream2.window(Minutes(1))
val joinedStream = windowedStream1.join(windowedStream2)
```

Stream-dataset joins

```
val dataset: RDD[String, String] = ...
val windowedStream = stream.window(Seconds(20))...
val joinedStream = windowedStream.transform { rdd => rdd.join(dataset) }
```

第 5 章 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 Python 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。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。

第 6 章 DStream 编程进阶

6.1 累加器和广播变量

累加器(Accumulators)和广播变量(Broadcast variables)不能从 Spark Streaming 的检查点中恢复。如果你启用检查并也使用了累加器和广播变量，那么你必须创建累加器和广播变量的延迟单实例从而在驱动因失效重启后他们可以被重新实例化。如下例述：

```
object WordBlacklist {

  @volatile private var instance: Broadcast[Seq[String]] = null

  def getInstance(sc: SparkContext): Broadcast[Seq[String]] = {
    if (instance == null) {
      synchronized {
        if (instance == null) {
          val wordBlacklist = Seq("a", "b", "c")
          instance = sc.broadcast(wordBlacklist)
        }
      }
    }
    instance
  }
}

object DroppedWordsCounter {

  @volatile private var instance: LongAccumulator = null

  def getInstance(sc: SparkContext): LongAccumulator = {
    if (instance == null) {
      synchronized {
        if (instance == null) {
          instance = sc.longAccumulator("WordsInBlacklistCounter")
        }
      }
    }
    instance
  }
}
wordCounts.foreachRDD { (rdd: RDD[(String, Int)], time: Time) =>
  // Get or register the blacklist Broadcast
  val blacklist = WordBlacklist.getInstance(rdd.sparkContext)
  // Get or register the droppedWordsCounter Accumulator
  val droppedWordsCounter = DroppedWordsCounter.getInstance(rdd.sparkContext)
```

```
  // Use blacklist to drop words and use droppedWordsCounter to count them
  val counts = rdd.filter { case (word, count) =>
    if (blacklist.value.contains(word)) {
      droppedWordsCounter.add(count)
      false
    } else {
      true
    }
  }.collect().mkString("[", ", ", "]")
  val output = "Counts at time " + time + " " + counts
})
```

6.2 DataFrame ans 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 word count 进行修改从而使用 DataFrames 和 SQL 来产生 word counts。每个 RDD 被转换为 DataFrame，以临时表格配置并用 SQL 进行查询。

```
val words: DStream[String] = ...

words.foreachRDD { rdd =>

  // Get the singleton instance of SparkSession
  val spark = SparkSession.builder.config(rdd.sparkContext.getConf).getOrCreate()
  import spark.implicits._

  // Convert RDD[String] to DataFrame
  val wordsDataFrame = rdd.toDF("word")

  // Create a temporary view
  wordsDataFrame.createOrReplaceTempView("words")

  // Do word count on DataFrame using SQL and print it
  val wordCountsDataFrame =
  spark.sql("select word, count(*) as total from words group by word")
  wordCountsDataFrame.show()
}
```

你也可以从不同的线程在定义于流数据的表上运行 SQL 查询（也就是说，异步运行 StreamingContext）。仅确定你设置 StreamingContext 记住了足够数量的流数据以使得查询操作可以运行。否则，StreamingContext 不会意识到任何异步的 SQL 查询操作，那么其就会在查询完成之后删除旧的数据。例如，如果你要查询最后一批次，但是你的查询会运行 5 分钟，那么你需要调用 streamingContext.remember(Minutes(5))(in Scala, 或者其他语言的等价操作)。

6.3 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 会自动保存在内存中。