

SparkStreaming 应用解析

教案

文档修订记录

文件状态： [] 草稿 [√] 正式发布		当前版本：	V1.1			
		作 者：	武玉飞			
		审 核 人：				
		发布日期：	20170518			
编制日期	版本	状态	简要说明	作者	审核者	审核日期
20170518	V1.1	A				

说明：

1. 按修改时间先后倒序排列，最新修改的排在第一行。
2. 版本栏中填入版本编号或者更改记录编号。
3. 状态分为三种状态：A——增加；M——修改；D——删除。
4. 在简要说明栏中填写变更的内容和变更的范围。
5. 表中所有日期格式为：YYYYMMDD。

目 录

第 1 章	Spark Streaming 概述	2
1.1	什么是 Spark Streaming	2
1.2	为什么要学习 Spark Streaming	3
1.3	Spark 与 Storm 的对比	4
第 2 章	运行 Spark Streaming	4
2.1	IDEA 编写程序	4
第 3 章	架构与抽象	6
第 4 章	Spark Streaming 解析	7
4.1	初始化 StreamingContext	7
4.2	什么是 DStreams	8
4.3	DStreams 输入	9
4.3.1	基本数据源	9
4.3.2	高级数据源	17
4.4	DStreams 转换	27
4.4.1	无状态转化操作	29
4.4.2	有状态转化操作	30
4.4.3	重要操作	39
4.5	DStreams 输出	40
4.6	累加器和广播变量	42
4.7	DataFrame ans SQL Operations	43
4.8	Caching / Persistence	44
4.9	7x24 不间断运行	44
4.9.1	检查点机制	44
4.9.2	驱动器程序容错	46
4.9.3	工作节点容错	46
4.9.4	接收器容错	46
4.9.5	处理保证	47
4.10	性能考量	48

第1章 Spark Streaming 概述

1.1 什么是 Spark Streaming



[Download](#) [Libraries](#) [Documentation](#) [Examples](#) [Community](#) [FAQ](#)

Spark Streaming makes it easy to build scalable fault-tolerant streaming applications.

Spark Streaming 类似于 Apache Storm，用于流式数据的处理。根据其官方文档介绍，Spark Streaming 有高吞吐量和容错能力强等特点。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 为什么要学习 Spark Streaming

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))
```

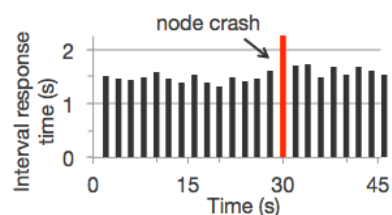
Counting tweets on a sliding window

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

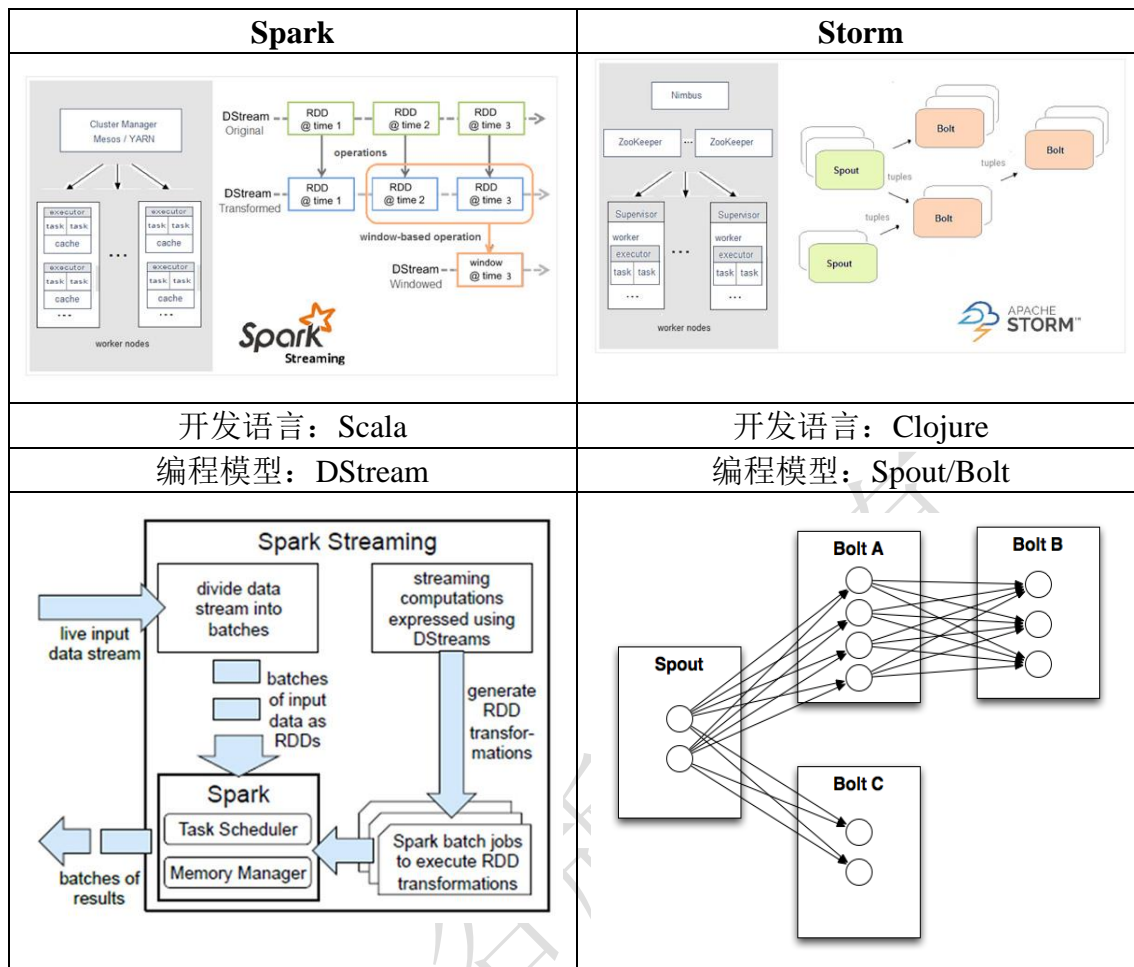
Combine streaming with batch and interactive queries.

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 Spark 与 Storm 的对比



第2章 运行 Spark Streaming

2.1 IDEA 编写程序

Pom.xml 加入以下依赖:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-streaming_2.11</artifactId>
  <version>${spark.version}</version>
  <scope>provided</scope>
</dependency>
```

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

```
import org.apache.spark.SparkConf
```

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

/**
 * Created by wuyufei on 06/09/2017.
 */
object WorldCount {
  def main(args: Array[String]) {

    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("master01", 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.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
  }
}
```

按照 Spark Core 中的方式进行打包，并将程序上传到 Spark 机器上。并运行：

```
bin/spark-submit --class com.atguigu.streaming.worldCount ~/wordcount-jar-with-
dependencies.jar
```

通过 Netcat 发送数据：

```
# TERMINAL 1:
# Running Netcat

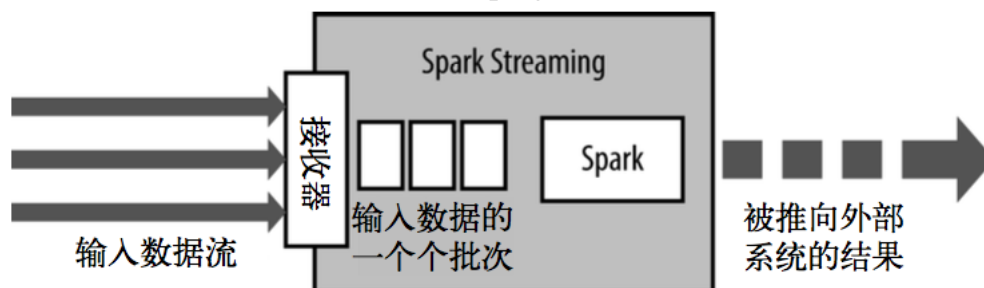
$ nc -lk 9999

hello world
```

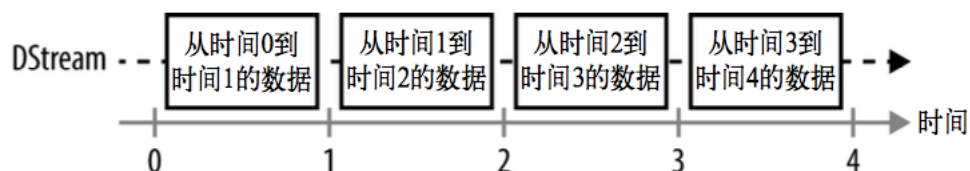
如果程序运行时，log 日志太多，可以将 spark conf 目录下的 log4j 文件里面的日志级别改成 WARN。

第3章 架构与抽象

Spark Streaming 使用“微批次”的架构，把流式计算当作一系列连续的小规模批处理来对待。Spark Streaming 从各种输入源中读取数据，并把数据分组为小的批次。新的批次按均匀的时间间隔创建出来。在每个时间区间开始的时候，一个新的批次就创建出来，在该区间内收到的数据都会被添加到这个批次中。在时间区间结束时，批次停止增长。时间区间的大小是由批次间隔这个参数决定的。批次间隔一般设在 500 毫秒到几秒之间，由应用开发者配置。每个输入批次都形成一个 RDD，以 Spark 作业的方式处理并生成其他的 RDD。处理的结果可以以批处理的方式传给外部系统。高层次的架构如图

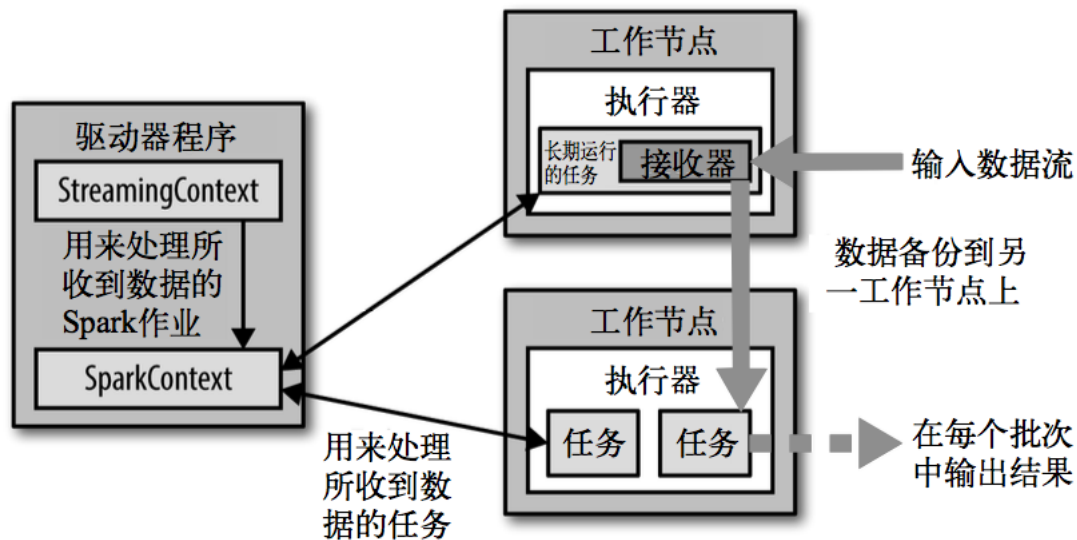


Spark Streaming 的编程抽象是离散化流，也就是 DStream。它是一个 RDD 序列，每个 RDD 代表数据流中一个时间片内的数据。



Spark Streaming 在 Spark 的驱动程序—工作节点的结构执行过程如下图所示。Spark Streaming 为每个输入源启动对应的接收器。接收器以任务的形式运行在应用的执行器进程中，从输入源收集数据并保存为 RDD。它们收集到输入数据后会吧数据复制到另一个执行器进程来保障容错性(默认行

为)。数据保存在执行器进程的内存中，和缓存 RDD 的方式一样。驱动器程序中的 StreamingContext 会周期性地运行 Spark 作业来处理这些数据，把数据与之前时间区间中的 RDD 进行整合。



第4章 Spark Streaming 解析

4.1 初始化 StreamingContext

```
import org.apache.spark._
import org.apache.spark.streaming._

val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
// 可以通过 ssc.sparkContext 来访问 SparkContext
// 或者通过已经存在的 SparkContext 来创建 StreamingContext
import org.apache.spark.streaming._

val sc = ... // existing SparkContext
val ssc = new StreamingContext(sc, Seconds(1))
```

初始化完 Context 之后：

- 1) 定义消息输入源来创建 DStreams.
- 2) 定义 DStreams 的转化操作和输出操作。
- 3) 通过 streamingContext.start()来启动消息采集和处理.

- 4) 等待程序终止，可以通过 `streamingContext.awaitTermination()` 来设置
- 5) 通过 `streamingContext.stop()` 来手动终止处理程序。

StreamingContext 和 SparkContext 什么关系？

```
import org.apache.spark.streaming._

val sc = ...                // existing SparkContext

val ssc = new StreamingContext(sc, Seconds(1))
```

注意：

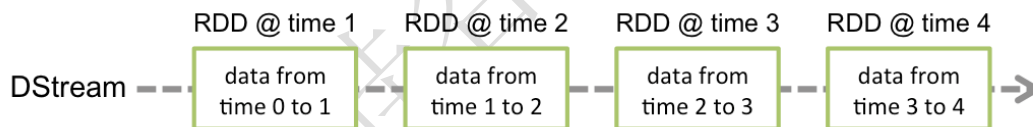
StreamingContext 一旦启动，对 DStreams 的操作就不能修改了。

在同一时间一个 JVM 中只有一个 StreamingContext 可以启动

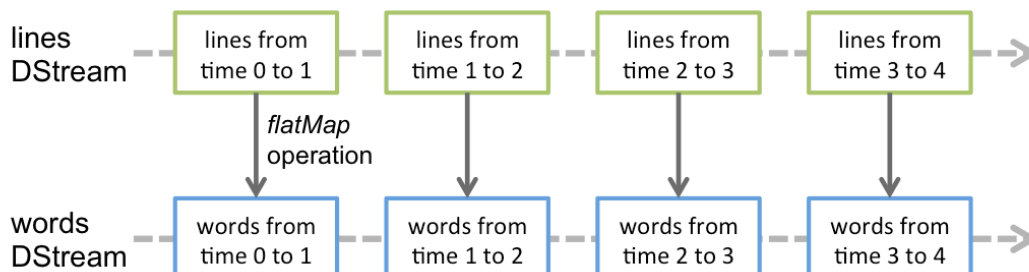
`stop()` 方法将同时停止 SparkContext，可以传入参数 `stopSparkContext` 用于只停止 StreamingContext

4.2 什么是 DStreams

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



4.3 DStreams 输入

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]。

4.3.1 基本数据源

4.3.1.1 文件数据源

Socket 数据流前面的例子已经看到过。

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

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

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。

- 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。

如果文件比较简单，则可以使用 `streamingContext.textFileStream(dataDirectory)` 方法来读取文件。文件流不需要接收器，不需要单独分配 CPU 核。

Hdfs 读取实例：

提前需要在 HDFS 上建好目录。

```
scala> import org.apache.spark.streaming._
import org.apache.spark.streaming._

scala> val ssc = new StreamingContext(sc, Seconds(1))
ssc: org.apache.spark.streaming.StreamingContext =
org.apache.spark.streaming.StreamingContext@4027edeb

scala> val lines = ssc.textFileStream("hdfs://master01:9000/data/")
lines: org.apache.spark.streaming.dstream.DStream[String] =
org.apache.spark.streaming.dstream.MappedDStream@61d9dd15

scala> val words = lines.flatMap(_.split(" "))
words: org.apache.spark.streaming.dstream.DStream[String] =
org.apache.spark.streaming.dstream.FlatMappedDStream@1e084a26

scala> val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts: org.apache.spark.streaming.dstream.DStream[(String, Int)] =
org.apache.spark.streaming.dstream.ShuffledDStream@8947a4b

scala> wordCounts.print()

scala> ssc.start()
```

上传文件上去：

```
[bigdata@master01 hadoop-2.7.3]$ ls
bin data etc include lib libexec LICENSE.txt logs NOTICE.txt README.txt sbin sdata
share

[bigdata@master01 hadoop-2.7.3]$ bin/hdfs dfs -put ./LICENSE.txt /data/
[bigdata@master01 hadoop-2.7.3]$ bin/hdfs dfs -put ./README.txt /data/
```

获取计算结果：

```
-----
Time: 1504665716000 ms
-----
```

```
-----
Time: 1504665717000 ms
-----
```

```
-----
Time: 1504665718000 ms
-----
```

```
(227.7202-1,2)
```

```
(created,2)
```

```
(offer, 8)
(BUSINESS, 11)
(agree, 10)
(hereunder, , 1)
( "control" , 1)
(Grant, 2)
(2. 2. , 2)
(include, 11)
...

-----

Time: 1504665719000 ms

-----

Time: 1504665739000 ms

-----

Time: 1504665740000 ms

-----

(under, 1)
(Technology, 1)
(distribution, 2)
(http://hadoop. apache. org/core/, 1)
(Unrestricted, 1)
(740. 13), 1)
(check, 1)
(have, 1)
(policies, 1)
(uses, 1)
...

-----

Time: 1504665741000 ms

-----
```

4. 3. 1. 2 自定义数据源

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

```
class CustomReceiver(host: String, port: Int)
  extends Receiver[String](StorageLevel.MEMORY_AND_DISK_2) with Logging {

  def onStart() {
    // Start the thread that receives data over a connection

    new Thread("Socket Receiver") {
      override def run() { receive() }
    }
  }
}
```

```
    }.start()
  }

  def onStop() {
    // There is nothing much to do as the thread calling receive()
    // is designed to stop by itself if isStopped() returns false
  }

  /** Create a socket connection and receive data until receiver is stopped */
  private def receive() {
    var socket: Socket = null
    var userInput: String = null
    try {
      // Connect to host:port
      socket = new Socket(host, port)

      // Until stopped or connection broken continue reading
      val reader = new BufferedReader(
        new InputStreamReader(socket.getInputStream(), StandardCharsets.UTF_8))
      userInput = reader.readLine()
      while(!isStopped && userInput != null) {
        store(userInput)
        userInput = reader.readLine()
      }
      reader.close()
      socket.close()

      // Restart in an attempt to connect again when server is active again
      restart("Trying to connect again")
    } catch {
      case e: java.net.ConnectException =>
        // restart if could not connect to server
        restart("Error connecting to " + host + ":" + port, e)
      case t: Throwable =>
        // restart if there is any other error
        restart("Error receiving data", t)
    }
  }
}
```

可以通过 `streamingContext.receiverStream(<instance of custom receiver>)`
来使用自定义的数据采集源

```
// Assuming ssc is the StreamingContext
val customReceiverStream = ssc.receiverStream(new CustomReceiver(host, port))
val words = lines.flatMap(_.split(" "))
...
```

模拟 Spark 内置的 Socket 链接:

```
package com.atguigu.streaming

import java.io.{BufferedReader, InputStreamReader}
import java.net.Socket
import java.nio.charset.StandardCharsets

import org.apache.spark.SparkConf
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.receiver.Receiver

/**
 * Created by wuyufei on 06/09/2017.
 */

class CustomReceiver (host: String, port: Int) extends
Receiver[String](StorageLevel.MEMORY_AND_DISK_2) {
  override def onStart(): Unit = {
    // Start the thread that receives data over a connection
    new Thread("Socket Receiver") {
      override def run() { receive() }
    }.start()
  }

  override def onStop(): Unit = {
    // There is nothing much to do as the thread calling receive()
    // is designed to stop by itself if isStopped() returns false
  }

  /** Create a socket connection and receive data until receiver is stopped */
  private def receive() {
    var socket: Socket = null
    var userInput: String = null
    try {
      // Connect to host:port
      socket = new Socket(host, port)
    }
  }
}
```

```
// Until stopped or connection broken continue reading
val reader = new BufferedReader(new
InputStreamReader(socket.getInputStream(), StandardCharsets.UTF_8))

    userInput = reader.readLine()
    while(!isStopped && userInput != null) {

        // 传送出来
        store(userInput)

        userInput = reader.readLine()
    }
    reader.close()
    socket.close()

    // Restart in an attempt to connect again when server is active again
    restart("Trying to connect again")
} catch {
    case e: java.net.ConnectException =>
        // restart if could not connect to server
        restart("Error connecting to " + host + ":" + port, e)
    case t: Throwable =>
        // restart if there is any other error
        restart("Error receiving data", t)
}
}

object CustomReceiver {
    def main(args: Array[String]) {

        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.receiverStream(new CustomReceiver("master01", 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.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()

}

}
```

```
[bigdata@master01 spark-2.1.1-bin-hadoop2.7]$ bin/spark-submit --class com.atguigu.streaming.CustomReceiver ~/customwo
rdcount-jar-with-dependencies.jar
17/09/05 23:10:32 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java
classes where applicable
-----
Time: 1504667435000 ms
-----
Time: 1504667436000 ms
-----
Time: 1504667437000 ms
-----
Time: 1504667438000 ms
-----
Time: 1504667439000 ms
-----
17/09/05 23:10:39 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
17/09/05 23:10:39 WARN BlockManager: Block input-0-1504667439400 replicated to only 0 peer(s) instead of 1 peers
Time: 1504667440000 ms
-----
(sdsfsf,1)
```

```
[bigdata@master01 hadoop-2.7.3]$ nc -lk 9999
sdsfsf
sdf
fdsfs
sdf
sdf
```

4.3.1.3 RDD 队列

测试过程中，可以通过使用 `streamingContext.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。

```
package com.atguigu.streaming

import org.apache.spark.SparkConf
```

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

import scala.collection.mutable

object QueueRdd {

  def main(args: Array[String]) {

    val conf = new SparkConf().setMaster("local[2]").setAppName("QueueRdd")
    val ssc = new StreamingContext(conf, Seconds(1))

    // Create the queue through which RDDs can be pushed to
    // a QueueInputDStream
    //创建 RDD 队列
    val rddQueue = new mutable.SynchronizedQueue[RDD[Int]]()

    // Create the QueueInputDStream and use it do some processing
    // 创建 QueueInputDStream
    val inputStream = ssc.queueStream(rddQueue)

    //处理队列中的 RDD 数据
    val mappedStream = inputStream.map(x => (x % 10, 1))
    val reducedStream = mappedStream.reduceByKey(_ + _)

    //打印结果
    reducedStream.print()

    //启动计算
    ssc.start()

    // Create and push some RDDs into
    for (i <- 1 to 30) {
      rddQueue += ssc.sparkContext.makeRDD(1 to 300, 10)
      Thread.sleep(2000)

      //通过程序停止 StreamingContext 的运行
      //ssc.stop()
    }
  }
}
```

```
[bigdata@master01 spark-2.1.1-bin-hadoop2.7]$ bin/spark-submit --class  
com.atguigu.streaming.QueueRdd ~/queueRdd-jar-with-dependencies.jar  
17/09/05 23:28:03 WARN NativeCodeLoader: Unable to load native-hadoop library for your  
platform... using builtin-java classes where applicable
```

```
-----  
Time: 1504668485000 ms  
-----
```

```
(4, 30)  
(0, 30)  
(6, 30)  
(8, 30)  
(2, 30)  
(1, 30)  
(3, 30)  
(7, 30)  
(9, 30)  
(5, 30)
```

```
-----  
Time: 1504668486000 ms  
-----
```

```
-----  
Time: 1504668487000 ms  
-----
```

```
(4, 30)  
(0, 30)  
(6, 30)  
(8, 30)  
(2, 30)  
(1, 30)  
(3, 30)  
(7, 30)  
(9, 30)  
(5, 30)
```

4.3.2 高级数据源

除核心数据源外，还可以用附加数据源接收器来从一些知名数据获取系统

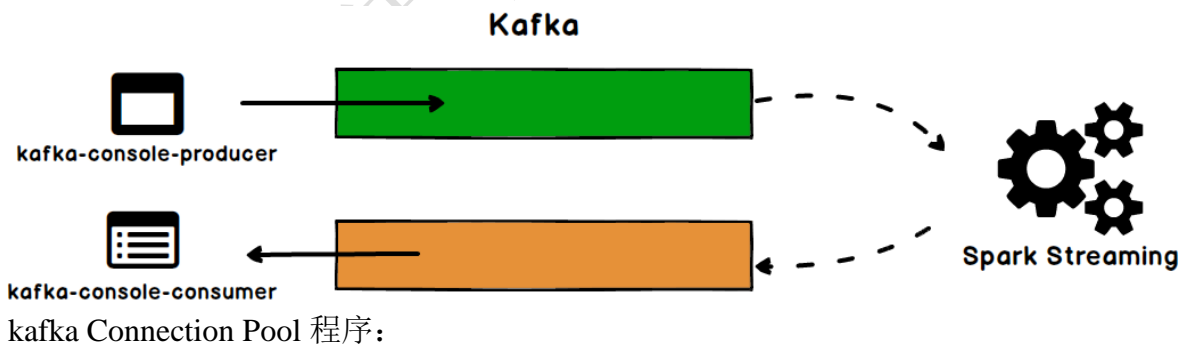
中接收的数据，这些接收器都作为 Spark Streaming 的组件进行独立打包了。它们仍然是 Spark 的一部分，不过你需要在构建文件中添加额外的包才能使用它们。现有的接收器包括 Twitter、Apache Kafka、Amazon Kinesis、Apache Flume，以及 ZeroMQ。可以通过添加与 Spark 版本匹配的 Maven 工件 spark-streaming-[projectname]_2.10 来引入这些附加接收器。

4.3.2.1 Apache Kafka

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

```
import org.apache.spark.streaming.kafka._ ... // 创建一个从主题到接收器线程数的映射表
val topics = List(("pandas", 1), ("logs", 1)).toMap
val topicLines = KafkaUtils.createStream(ssc, zkQuorum, group, topics)
topicLines.map(_._2)
```

下面我们进行一个实例，演示 SparkStreaming 如何从 Kafka 读取消息，如果通过连接池方法把消息处理完成后再写会 Kafka：



```
package com.atguigu.streaming
import java.util.Properties
import org.apache.commons.pool2.impl.DefaultPooledObject
import org.apache.commons.pool2.{BasePooledObjectFactory, PooledObject}
import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}

/**
 * Created by wuyufei on 06/09/2017.
```

```
*/  
  
case class KafkaProducerProxy(brokerList: String,  
                               producerConfig: Properties = new Properties,  
                               defaultTopic: Option[String] = None,  
                               producer: Option[KafkaProducer[String, String]] = None)  
{  
  
  type Key = String  
  type Val = String  
  
  require(brokerList == null || !brokerList.isEmpty, "Must set broker list")  
  
  private val p = producer.getOrElse {  
  
    var props: Properties = new Properties();  
    props.put("bootstrap.servers", brokerList);  
    props.put("key.serializer",  
              "org.apache.kafka.common.serialization.StringSerializer");  
    props.put("value.serializer",  
              "org.apache.kafka.common.serialization.StringSerializer");  
  
    new KafkaProducer[String, String](props)  
  }  
  
  private def toMessage(value: Val, key: Option[Key] = None, topic:  
Option[String] = None): ProducerRecord[Key, Val] = {  
    val t = topic.getOrElse(defaultTopic.getOrElse(throw new  
IllegalArgumentExcpion("Must provide topic or default topic")))  
    require(!t.isEmpty, "Topic must not be empty")  
    key match {  
      case Some(k) => new ProducerRecord(t, k, value)  
      case _ => new ProducerRecord(t, value)  
    }  
  }  
  
  def send(key: Key, value: Val, topic: Option[String] = None) {  
    p.send(toMessage(value, option(key), topic))  
  }  
  
  def send(value: Val, topic: Option[String]) {
```

```
    send(null, value, topic)
  }

  def send(value: Val, topic: String) {
    send(null, value, Option(topic))
  }

  def send(value: Val) {
    send(null, value, None)
  }

  def shutdown(): Unit = p.close()
}

abstract class KafkaProducerFactory(brokerList: String, config: Properties,
topic: Option[String] = None) extends Serializable {

  def newInstance(): KafkaProducerProxy
}

class BaseKafkaProducerFactory(brokerList: String,
                                config: Properties = new Properties,
                                defaultTopic: Option[String] = None)
  extends KafkaProducerFactory(brokerList, config, defaultTopic) {

  override def newInstance() = new KafkaProducerProxy(brokerList, config,
defaultTopic)
}

class PooledKafkaProducerAppFactory(val factory: KafkaProducerFactory)
  extends BasePooledObjectFactory[KafkaProducerProxy] with Serializable {

  override def create(): KafkaProducerProxy = factory.newInstance()

  override def wrap(obj: KafkaProducerProxy): PooledObject[KafkaProducerProxy]
= new DefaultPooledObject(obj)

  override def destroyObject(p: PooledObject[KafkaProducerProxy]): Unit = {
```

```
p.getObject.shutdown()
super.destroyObject(p)
}
}
```

KafkaStreaming main:

```
package com.atguigu.streaming

import org.apache.commons.pool2.impl.{GenericObjectPool,
GenericObjectPoolConfig}
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.SparkConf
import org.apache.spark.api.java.function.VoidFunction
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.kafka010.{ConsumerStrategies, KafkaUtils,
LocationStrategies}
import org.apache.spark.streaming.{Seconds, StreamingContext}

/**
 * Created by wuyufei on 06/09/2017.
 */
object createKafkaProducerPool{

  def apply(brokerList: String, topic: String):
GenericObjectPool[KafkaProducerProxy] = {
    val producerFactory = new BaseKafkaProducerFactory(brokerList, defaultTopic
= Option(topic))
    val pooledProducerFactory = new
PooledKafkaProducerAppFactory(producerFactory)
    val poolConfig = {
      val c = new GenericObjectPoolConfig
      val maxNumProducers = 10
      c.setMaxTotal(maxNumProducers)
      c.setMaxIdle(maxNumProducers)
      c
    }
    new GenericObjectPool[KafkaProducerProxy](pooledProducerFactory,
poolConfig)
  }
}
```

```
object KafkaStreaming{

    def main(args: Array[String]) {

        val conf = new
SparkConf().setMaster("local[4]").setAppName("NetworkWordCount")
        val ssc = new StreamingContext(conf, Seconds(1))

        //创建 topic
        val brokers =
"172.16.148.150:9092,172.16.148.151:9092,172.16.148.152:9092"
        val sourcetopic="source";
        val targettopic="target";

        //创建消费者组
        var group="con-consumer-group"
        //消费者配置
        val kafkaParam = Map(
            "bootstrap.servers" -> brokers, //用于初始化链接到集群的地址
            "key.deserializer" -> classOf[StringDeserializer],
            "value.deserializer" -> classOf[StringDeserializer],
            //用于标识这个消费者属于哪个消费团体
            "group.id" -> group,
            //如果没有初始化偏移量或者当前的偏移量不存在任何服务器上，可以使用这个配置属性
            //可以使用这个配置，latest 自动重置偏移量为最新的偏移量
            "auto.offset.reset" -> "latest",
            //如果是 true，则这个消费者的偏移量会在后台自动提交
            "enable.auto.commit" -> (false: java.lang.Boolean)
        );

        //ssc.sparkContext.broadcast(pool)

        //创建 DStream，返回接收到的输入数据
        var stream=KafkaUtils.createDirectStream[String,String](ssc,
LocationStrategies.PreferConsistent,ConsumerStrategies.Subscribe[String,String]
](Array(sourcetopic),kafkaParam))

        //每一个 stream 都是一个 ConsumerRecord
        stream.map(s =>("id:" + s.key(), ">>>:" + s.value())).foreachRDD(rdd => {
```



```
rdd.foreachPartition(partitionOfRecords => {  
    // Get a producer from the shared pool  
    val pool = createKafkaProducerPool(brobrokers, targettopic)  
    val p = pool.borrowObject()  
  
    partitionOfRecords.foreach {message =>  
System.out.println(message._2);p.send(message._2,option(targettopic))}  
  
    // Returning the producer to the pool also shuts it down  
    pool.returnObject(p)  
  
    })  
})  
  
ssc.start()  
ssc.awaitTermination()  
  
}  
}
```

程序部署：

1、启动 zookeeper 和 kafka。

```
bin/kafka-server-start.sh -daemon ./config/server.properties
```

2、创建两个 topic，一个为 source，一个为 target

```
bin/kafka-topics.sh --create --zookeeper  
172.16.148.150:2181,172.16.148.151:2181,172.16.148.152:2181 --replication-factor 2 --  
partitions 2 --topic source
```

```
bin/kafka-topics.sh --create --zookeeper  
172.16.148.150:2181,172.16.148.151:2181,172.16.148.152:2181 --replication-factor 2 --  
partitions 2 --topic target
```

3、启动 kafka console producer 写入 source topic

```
bin/kafka-console-producer.sh --broker-list  
172.16.148.150:9092,172.16.148.151:9092,172.16.148.152:9092 --topic source
```

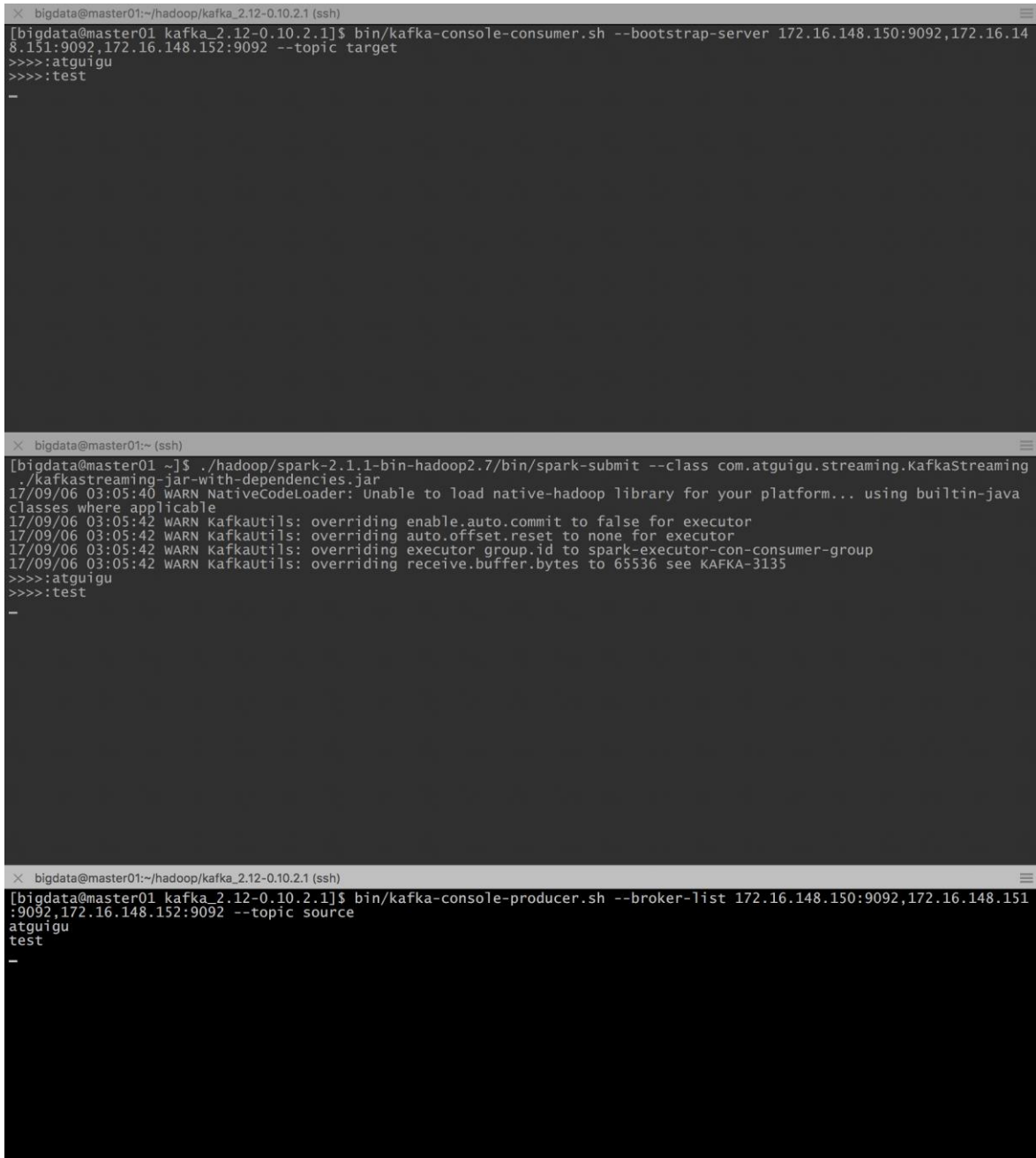
4、启动 kafka console consumer 监听 target topic

```
bin/kafka-console-consumer.sh --bootstrap-server  
172.16.148.150:9092,172.16.148.151:9092,172.16.148.152:9092 --topic source
```

5、启动 kafkaStreaming 程序：

```
[bigdata@master01 ~]$ ./hadoop/spark-2.1.1-bin-hadoop2.7/bin/spark-submit --class  
com.atguigu.streaming.KafkaStreaming ./kafkastreaming-jar-with-dependencies.jar
```

6、程序运行截图：



```
bigdata@master01:~/hadoop/kafka_2.12-0.10.2.1 (ssh)
[bigdata@master01 kafka_2.12-0.10.2.1]$ bin/kafka-console-consumer.sh --bootstrap-server 172.16.148.150:9092,172.16.148.151:9092,172.16.148.152:9092 --topic target
>>>>:atguigu
>>>>:test
-

bigdata@master01:~/hadoop/kafka_2.12-0.10.2.1 (ssh)
[bigdata@master01 ~]$ ./hadoop/spark-2.1.1-bin-hadoop2.7/bin/spark-submit --class com.atguigu.streaming.KafkaStreaming ./kafkastreaming-jar-with-dependencies.jar
17/09/06 03:05:40 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/09/06 03:05:42 WARN kafkautils: overriding enable.auto.commit to false for executor
17/09/06 03:05:42 WARN kafkautils: overriding auto.offset.reset to none for executor
17/09/06 03:05:42 WARN kafkautils: overriding executor.group.id to spark-executor-con-consumer-group
17/09/06 03:05:42 WARN kafkautils: overriding receive.buffer.bytes to 65536 see KAFKA-3135
>>>>:atguigu
>>>>:test
-

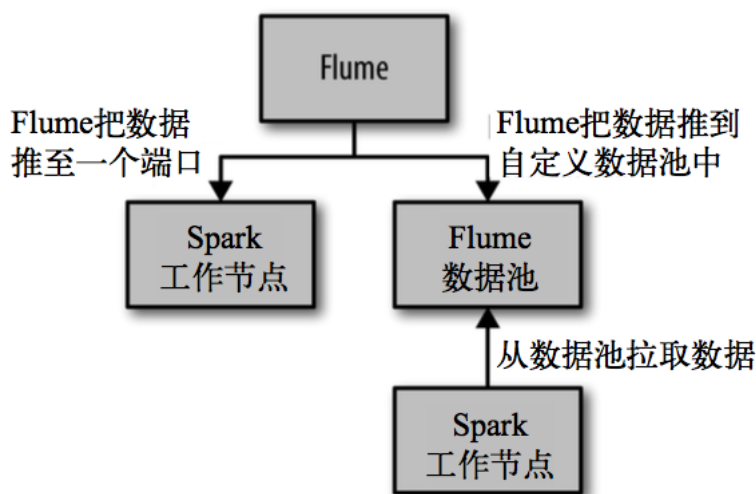
bigdata@master01:~/hadoop/kafka_2.12-0.10.2.1 (ssh)
[bigdata@master01 kafka_2.12-0.10.2.1]$ bin/kafka-console-producer.sh --broker-list 172.16.148.150:9092,172.16.148.151:9092,172.16.148.152:9092 --topic source
atguigu
test
-
```

4.3.2.2 Flume-ng

Spark 提供两个不同的接收器来使用 Apache Flume(<http://flume.apache.org/>，见图 10-8)。两个接收器简介如下。

- 推式接收器[55]该接收器以 Avro 数据池的方式工作，由 Flume 向其中推数据。
- 拉式接收器[56]该接收器可以从自定义的中间数据池中拉数据，而其他进程可以使用 Flume 把数据推进 该中间数据池。

两种方式都需要重新配置 Flume，并在某个节点配置的端口上运行接收器(不是已有的 Spark 或者 Flume 使用的端口)。要使用其中任何一种方法，都需要在工程中引入 Maven 工件 `spark-streaming-flume_2.10`。



推式接收器的方法设置起来很容易，但是它不使用事务来接收数据。在这种方式中，接收器以 Avro 数据池的方式工作，我们需要配置 Flume 来把数据发到 Avro 数据池(见例 10-34)。我们提供的 `FlumeUtils` 对象会把接收器配置在一个特定的工作节点的主机名及端口号上(见例 10-35 和例 10-36)。这些设置必须和 Flume 配置相匹配。

例 10-34: Flume 对 Avro 池的配置

```

a1.sinks = avroSink
a1.sinks.avroSink.type = avro
a1.sinks.avroSink.channel = memoryChannel
a1.sinks.avroSink.hostname = receiver-hostname
a1.sinks.avroSink.port = port-used-for-avro-sink-not-spark-port
  
```

例 10-35: Scala 中的 FlumeUtils 代理

```

val events = FlumeUtils.createStream(ssc, receiverHostname, receiverPort)
  
```

虽然这种方式很简洁，但缺点是没有事务支持。这会增加运行接收器的工作

作节点发生错误 时丢失少量数据的几率。不仅如此, 如果运行接收器的工作节点发生故障, 系统会尝试从 另一个位置启动接收器, 这时需要重新配置 Flume 才能将数据发给新的工作节点。这样配 置会比较麻烦。

较新的方式是拉式接收器(在 Spark 1.1 中引入), 它设置了一个专用的 Flume 数据池供 Spark Streaming 读取, 并让接收器主动从数据池中拉取数据。这种方式的优点在于弹性较 好, Spark Streaming 通过事务从数据池中读取并复制数据。在收到事务完成的通知前, 这 些数据还保留在数据池中。

我们需要先把自定义数据池配置为 Flume 的第三方插件。安装插件的最新方法请参考 Flume 文档的相关部分 (<https://flume.apache.org/FlumeUserGuide.html#installing-third-party-plugins>)。由于插件是用 Scala 写的, 因此需要把插件本身以及 Scala 库都添加到 Flume 插件 中。Spark 1.1 中对应的 Maven 索引如例 10-37 所示。

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-streaming-flume-sink_2.11</artifactId>
  <version>1.2.0</version>
</dependency>
<dependency>
  <groupId>org.scala-lang</groupId>
  <artifactId>scala-library</artifactId>
  <version>2.11.11</version>
</dependency>
```

当你把自定义 Flume 数据池添加到一个节点上之后, 就需要配置 Flume 来把数据推送到这个数据池中,

```
al.sinks = spark
al.sinks.spark.type = org.apache.spark.streaming.flume.sink.SparkSink
al.sinks.spark.hostname = receiver-hostname
al.sinks.spark.port = port-used-for-sync-not-spark-port
al.sinks.spark.channel = memoryChannel
```

等到数据已经在数据池中缓存起来, 就可以调用 FlumeUtils 来读取数据了

例 10-39: 在 Scala 中使用 FlumeUtils 读取自定义数据池

```
val events = FlumeUtils.createPollingStream(ssc, receiverHostname, receiverPort)
```

4.4 DStreams 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。

Transformation	Meaning
map(func)	将源 DStream 中的每个元素通过一个函数 func 从而得到新的 DStreams。
flatMap(func)	和 map 类似，但是每个输入的项可以被映射为 0 或更多项。
filter(func)	选择源 DStream 中函数 func 判为 true 的记录作为新 DStreams
repartition(numPartitions)	通过创建更多或者更少的 partition 来改变此 DStream 的并行级别。
union(otherStream)	联合源 DStreams 和其他 DStreams 来得到新 DStream
count()	统计源 DStreams 中每个 RDD 所含元素的个数得到单元素 RDD 的新 DStreams。
reduce(func)	通过函数 func(两个参数一个输出)来整合源 DStreams 中每个 RDD 元素得到单元素 RDD 的 DStreams。这个函数需要关联从而可以被并行计算。
countByValue()	对于 DStreams 中元素类型为 K 调用此函数，得到包含(K,Long)对的新 DStream，其中 Long 值表明相应的 K 在源 DStream 中每个 RDD 出现的频率。
reduceByKey(func, [numTasks])	对(K,V)对的 DStream 调用此函数，返回同样 (K,V)对的新 DStream，但是新 DStream 中的对应 V 为使用 reduce 函数整合而来。 <i>Note</i> ：默认情况下，这个操作使用

	Spark 默认数量的并行任务（本地模式为 2，集群模式中的数量取决于配置参数 <code>spark.default.parallelism</code> ）。你也可以传入可选的参数 <code>numTaska</code> 来设置不同数量的任务。
<code>join(otherStream, [numTasks])</code>	两 DStream 分别为(K,V)和(K,W)对，返回(K,(V,W))对的新 DStream。
<code>cogroup(otherStream, [numTasks])</code>	两 DStream 分别为(K,V)和(K,W)对，返回(K,(Seq[V],Seq[W]))对新 DStreams
<code>transform(func)</code>	将 RDD 到 RDD 映射的函数 <code>func</code> 作用于源 DStream 中每个 RDD 上得到新 DStream。这个可用于在 DStream 的 RDD 上做任意操作。
<code>updateStateByKey(func)</code>	得到”状态” DStream，其中每个 key 状态的更新是通过将给定函数用于此 key 的上一个状态和新值而得到。这个可用于保存每个 key 值的任意状态数据。

DStream 的转化操作可以分为无状态(stateless)和有状态(stateful)两种。

- 在无状态转化操作中，每个批次的处理不依赖于之前批次的数据。常见的 RDD 转化操作，例如 `map()`、`filter()`、`reduceByKey()` 等，都是无状态转化操作。

- 相对地，有状态转化操作需要使用之前批次的数据或者是中间结果来计算当前批次的数据。有状态转化操作包括基于滑动窗口的转化操作和追踪状态变化的转化操作。

4.4.1 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。

表10-1: DStream无状态转化操作的例子（不完整列表）

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。

4.4.2 有状态转化操作

特殊的 Transformations

4.4.2.1 追踪状态变化 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`：

```
package com.atguigu.streaming

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

/**
 * Created by wuyufei on 06/09/2017.
 */
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("master01", 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()
}

}
```

启动 nc -lk 9999

```
[bigdata@master01 ~]$ nc -lk 9999

ni shi shui

ni hao ma
```

启动统计程序:

```
[bigdata@master01 ~]$ ./hadoop/spark-2.1.1-bin-hadoop2.7/bin/spark-submit --class
com.atguigu.streaming.WorldCount ./statefulwordcount-jar-with-dependencies.jar
```

```
17/09/06 04:06:09 WARN NativeCodeLoader: Unable to load native-hadoop library for your
platform... using builtin-java classes where applicable
```

```
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)
```

```
[bigdata@master01 ~]$ ls
```

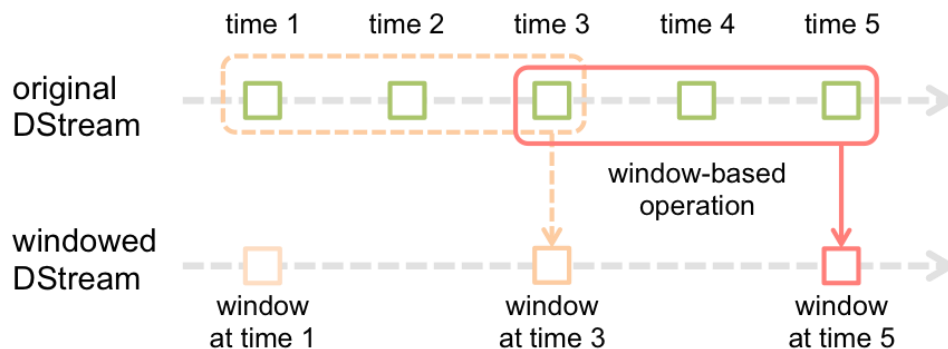
```
2df8e0c3-174d-401a-b3a7-f7776c3987db  checkpoint-1504685205000    data
backup                                checkpoint-1504685205000.bk  debug.log
checkpoint-1504685199000                checkpoint-1504685208000    hadoop
checkpoint-1504685199000.bk             checkpoint-1504685208000.bk receivedBlockMetadata
checkpoint-1504685202000                checkpoint-1504685211000    software
checkpoint-1504685202000.bk             checkpoint-1504685211000.bk statefulwordcount-jar-with-
dependencies.jar
```

4. 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。

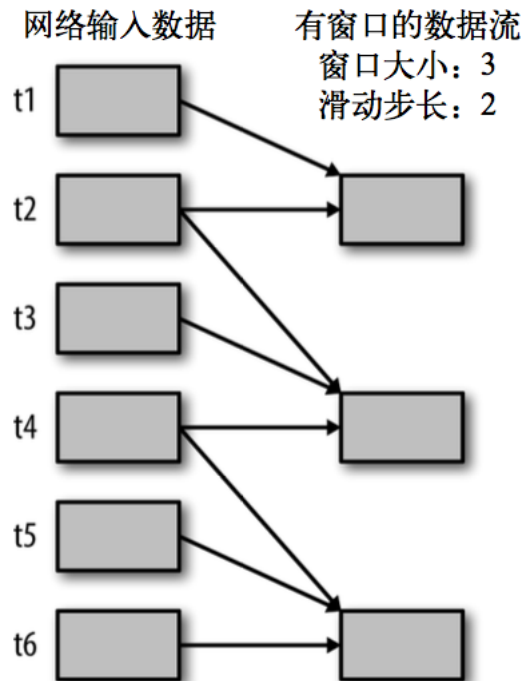


所有基于窗口的操作都需要两个参数，分别为窗口时长以及滑动步长，两者都必须是 `StreamContext` 的批次间隔的整数倍。窗口时长控制每次计算最近的多少个批次的数据，其实就是最近的 `windowDuration/batchInterval` 个批次。如果有一个以 10 秒为批次间隔的源 `DStream`，要创建一个最近 30 秒的时间窗口(即最近 3 个批次)，就应当把 `windowDuration` 设为 30 秒。而滑动步长的默认值与批次间隔相等，用来控制对新的 `DStream` 进行计算的间隔。如果源 `DStream` 批次间隔为 10 秒，并且我们只希望每两个批次计算一次窗口结果，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

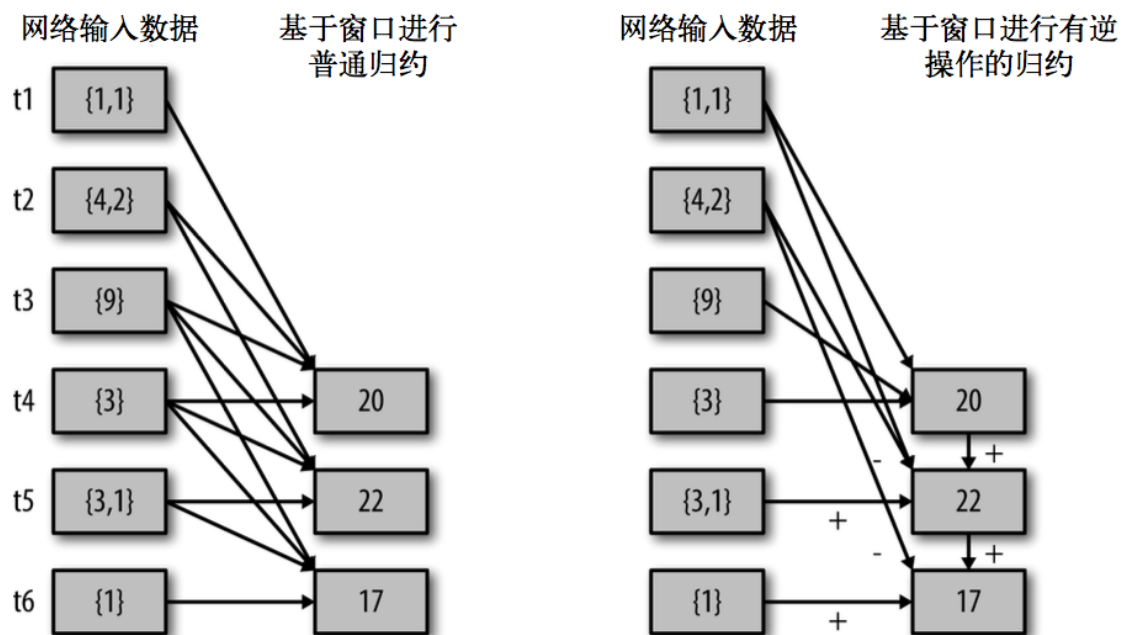
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda
x, y: x - y, 30, 20)
```



Transformation	Meaning
<code>window(windowLength, slideInterval)</code>	基于对源 DStream 窗化的批次进行计算返回一个新的 DStream
<code>countByWindow(windowLength, slideInterval)</code>	返回一个滑动窗口计数流中的元素。
<code>reduceByWindow(func, windowLength, slideInterval)</code>	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。
<code>reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])</code>	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 <code>reduce</code> 函数来整合每个

	<p>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。</p>
<code>reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])</code>	<p>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口的 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。</p>
<code>countByValueAndWindow(windowLength, slideInterval, [numTasks])</code>	<p>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。</p>

`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}

/**
 * Created by wuyufei on 06/09/2017.
 */
object WordCount {

  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("master01", 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()
}

}
```

4.4.3 重要操作

4.4.3.1 Transform Operation

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。

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。

其实也就是对 DStream 中的 RDD 应用转换。

```
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
  ...
}
```

4.4.3.2 Join 操作

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) }
```

4.5 DStreams 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。

Output Operation	Meaning
<code>print()</code>	在运行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 <code>pprint()</code> 。
<code>saveAsTextFiles(prefix, [suffix])</code>	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]”。
<code>saveAsObjectFiles(prefix, [suffix])</code>	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。
<code>saveAsHadoopFiles(prefix, [suffix])</code>	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。
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。

需要注意的：

- 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
- 4) 可以考虑使用连接池优化

```
dstream.foreachRDD { rdd =>
    // error val connection = createNewConnection() // executed at the driver 序列化错误

    rdd.foreachPartition { partitionOfRecords =>
        // ConnectionPool is a static, lazily initialized pool of connections
        val connection = ConnectionPool.getConnection()
        partitionOfRecords.foreach(record => connection.send(record) // executed at the worker
        )
        ConnectionPool.returnConnection(connection) // return to the pool for future reuse
    }
}
```

4.6 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
})
```

4.7 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 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)。

4.8 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 `reduceByWindowed` 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。

4.9 7x24 不间断运行

4.9.1 检查点机制

检查点机制是我们在 Spark Streaming 中用来保障容错性的主要机制。与应用程序逻辑无关的错误（即系统错位，JVM 崩溃等）有迅速恢复的能力。

它可以使 Spark Streaming 阶段性地把应用数据存储到诸如 HDFS 或 Amazon S3 这样的可靠存储系统中，以供恢复时使用。具体来说，检查点机制主要为以下两个目的服务。

- 1) 控制发生失败时需要重算的状态数。SparkStreaming 可以通过转化图的谱系图来重算状态，检查点机制则可以控制需要在转化图中回溯多远。
- 2) 提供驱动器程序容错。如果流计算应用中的驱动器程序崩溃了，你可以重启驱动器程序并让驱动器程序从检查点恢复，这样 Spark Streaming 就可以读取之前运行的程序处理数据的进度，并从那里继续。 [1]

了实现这个，Spark Streaming 需要为容错存储系统 *checkpoint* 足够的信息从而使得其可以从失败中恢复过来。有两种类型的数据设置检查点。

Metadata checkpointing: 将定义流计算的信息存入容错的系统如 HDFS。元数据包括：

配置 - 用于创建流应用的配置。

DStreams 操作 - 定义流应用的 DStreams 操作集合。

不完整批次 - 批次的工作已进行排队但是并未完成。

Data checkpointing: 将产生的 RDDs 存入可靠的存储空间。对于在多批次间合并数据的状态转换，这个很有必要。在这样的转换中，RDDs 的产生基于之前批次的 RDDs，这样依赖链长度随着时间递增。为了避免在恢复期这种无限的时间增长（和链长度成比例），状态转换中间的 RDDs 周期性写入可靠地存储空间（如 HDFS）从而切短依赖链。

总而言之，元数据检查点在由驱动失效中恢复是首要需要的。而数据或者 RDD 检查点甚至在使用了状态转换的基础函数中也是必要的。

出于这些原因，检查点机制对于任何生产环境中的流计算应用都至关重要。你可以通过向 `ssc.checkpoint()` 方法传递一个路径参数(HDFS、S3 或者本地路径均可)来配置检查点机制,同时你的应用应该能够使用检查点的数据

1. 当程序首次启动，其将创建一个新的 `StreamingContext`，设置所有的流并调用 `start()`。

2. 当程序在失效后重启，其将依据检查点目录的检查点数据重新创建一个 `StreamingContext`。通过使用 `StreamingContext.getOrCreate` 很容易获得这个性能。

```
ssc.checkpoint("hdfs://...")

# 创建和设置一个新的 StreamingContext
def functionToCreateContext():
    sc = SparkContext(...) # new context
    ssc = new StreamingContext(...)
    lines = ssc.socketTextStream(...) # create DStreams
    ...
    ssc.checkpoint(checkpointDirectory) # 设置检查点目录
    return ssc

# 从检查点数据中获取 StreamingContext 或者重新创建一个
context = StreamingContext.getOrCreate(checkpointDirectory,
functionToCreateContext)

# 在需要完成的 context 上做额外的配置
# 无论其有没有启动
context ...

# 启动 context
context.start()
context.awaitTermination()
```


如果检查点目录(checkpointDirectory)存在, 那么 context 将会由检查点数据重新创建。如果目录不存在(首次运行), 那么函数 functionToCreateContext 将会被调用来创建一个新的 context 并设置 DStreams。

注意 RDDs 的检查点引起存入可靠内存的开销。在 RDDs 需要检查点的批次里, 处理的时间会因此而延长。所以, 检查点的间隔需要很仔细地设置。在小尺寸批次(1 秒钟)。每一批次检查点会显著减少操作吞吐量。反之, 检查点设置的过于频繁导致“血统”和任务尺寸增长, 这会有很不好的影响对于需要 RDD 检查点设置的状态转换, 默认间隔是批次间隔的乘数一般至少为 10 秒钟。可以通过 `dstream.checkpoint(checkpointInterval)`。通常, 检查点设置间隔是 5-10 个 DStream 的滑动间隔。

4.9.2 驱动器程序容错

驱动器程序的容错要求我们以特殊的方式创建 StreamingContext。我们需要把检查点目录提供给 StreamingContext。与直接调用 `new StreamingContext` 不同, 应该使用 `StreamingContext.getOrCreate()` 函数。

```
def createStreamingContext() = {  
    ...  
    val sc = new SparkContext(conf) // 以 1 秒作为批次大小创建 StreamingContext  
    val ssc = new StreamingContext(sc, Seconds(1))  
    ssc.checkpoint(checkpointDir)  
}  
...  
val ssc = StreamingContext.getOrCreate(checkpointDir, createStreamingContext _)
```

4.9.3 工作节点容错

为了应对工作节点失败的问题, Spark Streaming 使用与 Spark 的容错机制相同的方法。所有从外部数据源中收到的数据都在多个工作节点上备份。所有从备份数据转化操作的过程中创建出来的 RDD 都能容忍一个工作节点的失败, 因为根据 RDD 谱系图, 系统可以把丢失的数据从幸存的输入数据备份中重算出来。

4.9.4 接收器容错

运行接收器的工作节点的容错也是很重要的。如果这样的节点发生错误, Spark Streaming 会在集群中别的节点上重启失败的接收器。然而, 这种情况不会导致数据的丢失取决于数据源的行为(数据源是否会重发数据)以及接收器的实现(接收器是否会向数据源确认收到数据)。举个例子, 使用 Flume 作

为数据源时，两种接收器的主要区别在于数据丢失时的保障。在“接收器从数据池中拉取数据”的模型中，Spark 只有在数据已经在集群中备份时才会从数据池中移除元素。而在“向接收器推数据”的模型中，如果接收器在数据备份之前失败，一些数据可能就会丢失。总的来说，对于任意一个接收器，你必须同时考虑上游数据源的容错性(是否支持事务)来确保零数据丢失。

总的来说，接收器提供以下保证。

- 所有从可靠文件系统中读取的数据(比如通过 `StreamingContext.hadoopFiles` 读取的)都是可靠的，因为底层的文件系统是有备份的。Spark Streaming 会记住哪些数据存放到了检查点中，并在应用崩溃后从检查点处继续执行。

- 对于像 Kafka、推式 Flume、Twitter 这样的不可靠数据源，Spark 会把输入数据复制到其他节点上，但是如果接收器任务崩溃，Spark 还是会丢失数据。在 Spark 1.1 以及更早的版本中，收到的数据只被备份到执行器进程的内存中，所以一旦驱动程序崩溃(此时所有的执行器进程都会丢失连接)，数据也会丢失。在 Spark 1.2 中，收到的数据被记录到诸如 HDFS 这样的可靠的文件系统中，这样即使驱动程序重启也不会导致数据丢失。

综上所述，确保所有数据都被处理的最佳方式是使用可靠的数据源(例如 HDFS、拉式 Flume 等)。如果你还要在批处理作业中处理这些数据，使用可靠数据源是最佳方式，因为这种方式确保了你的批处理作业和流计算作业能读取到相同的数据，因而可以得到相同的结果。

4.9.5 处理保证

由于 Spark Streaming 工作节点的容错保障，Spark Streaming 可以为所有的转化操作提供“精确一次”执行的语义，即使一个工作节点在处理部分数据时发生失败，最终的转化结

果(即转化操作得到的 RDD)仍然与数据只被处理一次得到的结果一样。

然而，当把转化操作得到的结果使用输出操作推入外部系统中时，写结果的任务可能因故障而执行多次，一些数据可能也就被写了多次。由于这引入了外部系统，因此我们需要专门针对各系统的代码来处理这样的情况。我们可以使用事务操作来写入外部系统(即原子化地将一个 RDD 分区一次写入)，或者设计幂等的更新操作(即多次运行同一个更新操作仍生成相同的结果)。比如 Spark Streaming 的 `saveAs...File` 操作会在一个文件写完时自动将其原子化地移动到最终位置上，以此确保每个输出文件只存在一份。

4. 10 性能考量

最常见的问题是 Spark Streaming 可以使用的最小批次间隔是多少。总的来说, 500 毫秒已经被证实为对许多应用而言是比较好的最小批次大小。寻找最小批次大小的最佳实践是从一个比较大的批次大小(10 秒左右)开始, 不断使用更小的批次大小。如果 Streaming 用户界面中显示的处理时间保持不变, 你就可以进一步减小批次大小。如果处理时间开始增加, 你可能已经达到了应用的极限。

相似地, 对于窗口操作, 计算结果的间隔(也就是滑动步长)对于性能也有巨大的影响。当计算代价巨大并成为系统瓶颈时, 就应该考虑提高滑动步长了。减少批处理所消耗时间的常见方式还有提高并行度。有以下三种方式可以提高并行度:

- 增加接收器数目 有时如果记录太多导致单台机器来不及读入并分发的话, 接收器会成为系统瓶颈。这时 你就需要通过创建多个输入 DStream(这样会创建多个接收器)来增加接收器数目, 然后使用 union 来把数据合并为一个数据源。

- 将收到的数据显式地重新分区 [5] 如果接收器数目无法再增加, 你可以通过使用 DStream.repartition 来显式重新分区输入流(或者合并多个流得到的数据流)来重新分配收到的数据。

- 提高聚合计算的并行度 对于像 reduceByKey() 这样的操作, 你可以在第二个参数中指定并行度, 我们在介绍 RDD 时提到过类似的手段。