

# SparkStreaming

## 回顾

- 1、Kafka的重要组件
- 2、Kafka的负载均衡
- 3、Kafka的存储机制

## 今天内容

- 1、SparkStreaming概述
- 2、DStream
- 3、案例实战
- 4、Spark On Yarn

## 教学目标

- 1、了解SparkStreaming的应用场景
- 2、熟悉DStream
- 3、通过案例掌握SparkStreaming的开发流程
- 4、Spark On Yarn使用场景及任务提交方式

## 1、SparkStreaming概述

### 1.1、SparkStreaming概念

Spark Streaming类似于Apache Storm，用于流式数据的处理。

知识点1：

流式大数据

流式大数据从这个角度看，可以把大数据分成两个：一个是批式大数据，另一个是流式大数据。

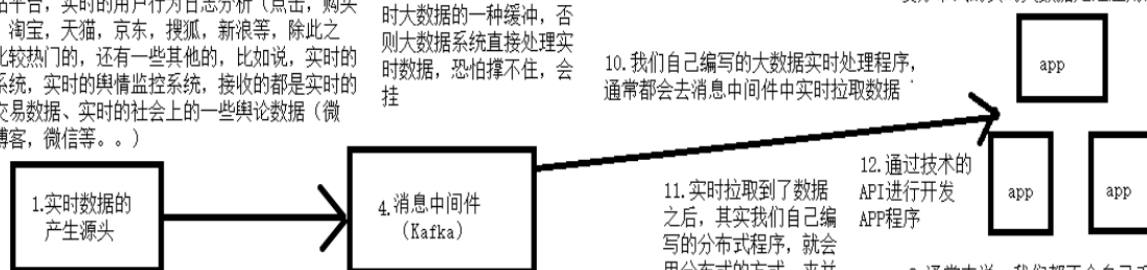
举个例子来说

我们把数据当成水库的话，水库里面存在的水就是批式大数据，进来的水是流式大数据。

3. 现在企业中，以及现在的这个世界和社会中，主要的实时数据产生的源头，有哪些呢？最基本的，是各大网站平台，实时的用户行为日志分析（点击，购买等）、淘宝，天猫，京东，搜狐，新浪等，除此之外，比较热门的，还有一些其他的，比如说，实时的金融交易系统，实时的舆情监控系统，接收的都是实时的金融交易数据、实时的社会上的一些舆论数据（微博，博客，微信等。。）

6. 其实，核心就是作为实时大数据的一种缓冲，否则大数据系统直接处理实时数据，恐怕撑不住，会挂

7. 我们学习的，以及我们要做的，其实就是开发分布式的实时大数据处理应用/系统



2. 实时数据，现在是在大数据领域里的一种非常热门的场景和应用，而且技术有相当的难度，应该是比sparkCore以及MR实现的离线批处理，以及Hive和sparkSQL可以实现的大数据交互查询，都要难的多。

5. 消息中间件：一般，实时的数据，都是发送到消息中间件里面去的，比如说，网站上的实时点击，可以通过很多方式，传送到消息中间件里面去，说一个最土的，比如，大家每点击一次，js脚本就发送一次Ajax请求到后台的kafka中，常见的还有，比如Nginx日志，还有Flume收集日志等，最后都发送到kafka中去，实时计算领域，最常见的消息队列/消息中间件，就是kafka

10. 我们自己编写的大数据实时处理程序，通常都会去消息中间件中实时拉取数据

11. 实时拉取到了数据之后，其实我们自己编写的分布式程序，就会用分布式的方式，来并行处理，实时的大数据，每个节点可能就处理一部分实时的数据，这样，多个节点同时处理就可以增强我们的大数据实时计算的能力，提高处理的速度。

12. 通过技术的API进行开发APP程序

8. 通常来说，我们都不会自己手动去开发基础的分布式实时计算平台/框架，而是使用现有的，开源框架/平台，比如storm, spark-streaming，它们其实就是一种分布式实时平台，其进程，可以部署多个节点，从而进行大数据的分布式实时处理，而我们自己编写的基于某种平台的大数据实时计算程序，就会以并行的方式，运行在这些平台之上。

9. 其实，就是我们所说的，storm/spark-streaming。

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。

## Overview

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data streams.



Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.



Flume是Cloudera提供的一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统，Flume支持在日志系统中定制各类数据发送方，用于收集数据；同时，Flume提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力。

Amazon Kinesis 可让您轻松收集、处理和分析实时流数据，以便您及时获得见解并对新信息快速做出响应。

dashboard是一种实时连续桌面信息检索引擎,以窗口形式出现在桌面上,可以提供实时天气情况、股票报价、航班时间等信息。

dashboard是商业智能仪表盘（business intelligence dashboard，BI dashboard）的简称，它是一般商业智能都拥有的实现数据可视化的模块，是向企业展示度量信息和关键业务指标（KPI）现状的数据虚拟化工具 [1] 。

dashboard在一个简单屏幕上联合并整理数字、公制和绩效记分卡。它们调整适应特定角色并展示为单一视角或部门指定的度量。

dashboard关键的特征是从多种数据源获取实时数据，并且是定制化的交互式界面。

dashboard以丰富的，可交互的可视化界面为数据提供更好的使用体验

## 1.2、为什么要学习Spark Streaming

易用

### Ease of Use

Build applications through high-level operators.

Spark Streaming brings Apache 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))
```

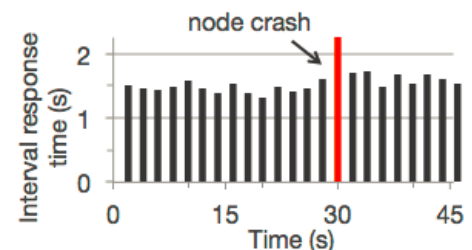
Counting tweets on a sliding window

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.



易整合到Spark体系

### Spark Integration

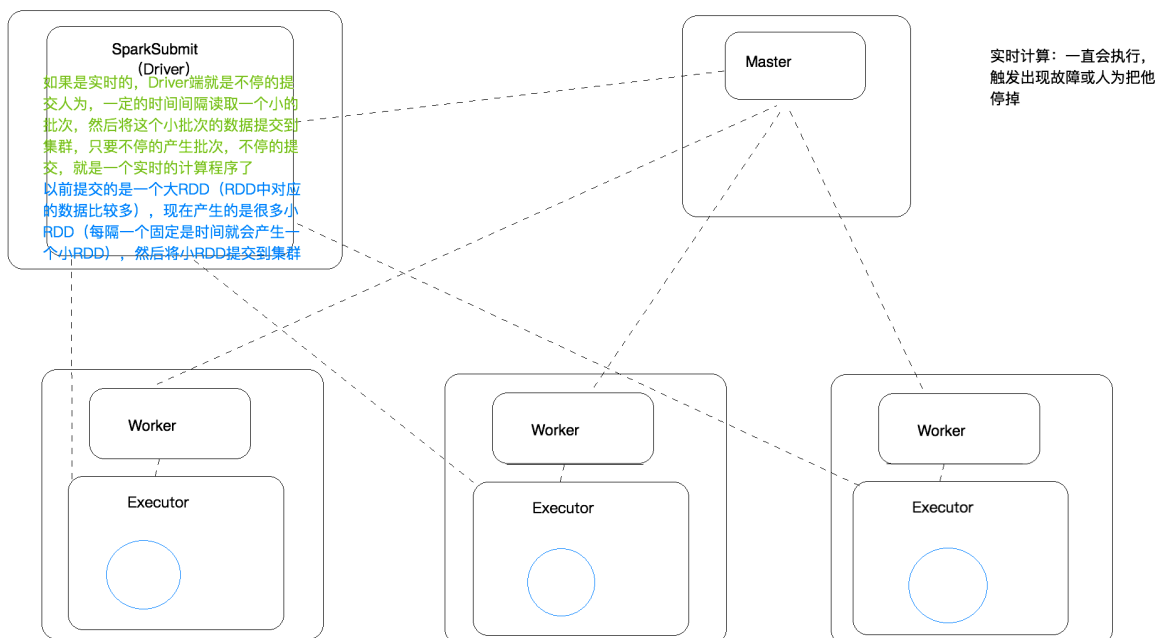
Combine streaming with batch and interactive queries.

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原理



## 1.4、应用场景

股票交易

交通状况

广告推荐

阿里双11

SparkStreaming 微批处理（类似于电梯），它并不是纯的批处理 优点：吞吐量大，可以做复杂的业务逻辑(保证每个job的处理小于batch interval) 缺点：数据延迟度较高

公司中为什么选用SparkStreaming要多一些？ 1.秒级别延迟，通常应用程序是可以接受的， 2.可以应用机器学习，SparkSQL...可扩展性比较好，数据吞吐量较高

## 1.5、实时计算框架对比

实时计算相关技术 Strom / JStrom Spark Streaming Flink

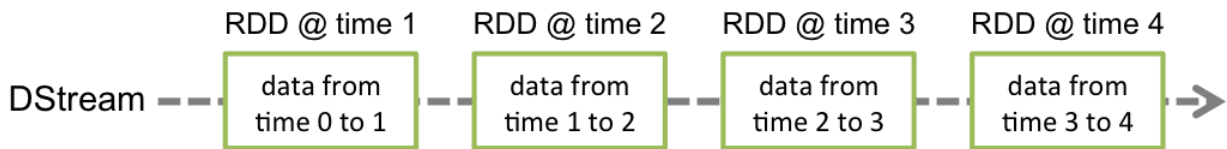
实时性高 有延迟 实时性高 吞吐量较低 吞吐量高 吞吐量高 只能实时计算 离线+实时 离线+实时 算子比较少 算子丰富 算子丰富 没有 机器学习 没有 没有 图计算 没有 使用比较少 非常火 一般

## 2、DStream

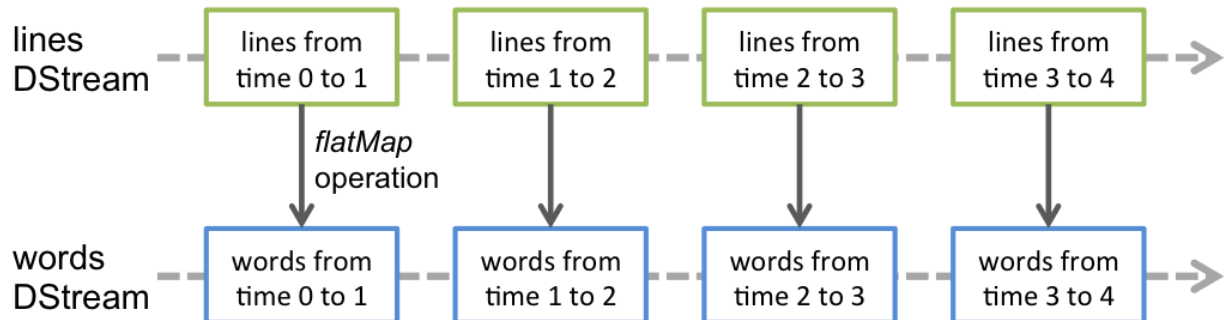
官网：

### 2.1、什么是DStream

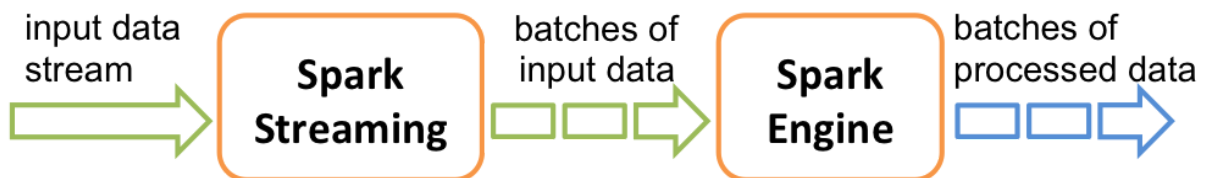
Discretized Stream是Spark Streaming的基础抽象，代表持续性的数据流和经过各种Spark原语操作后的结果数据流。在内部实现上，DStream是一系列连续的RDD来表示。每个RDD含有一段时间间隔内的数据，如下图：



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



## 2.2 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。Note：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spark.default.parallelism）。你也可以传入可选的参数numTaska来设置不同数量的任务。
join(otherStream, [numTasks])	两DStream分别为(K,V)和(K,W)对，返回(K,(V,W))对的新DStream。
cogroup(otherStream, [numTasks])	两DStream分别为(K,V)和(K,W)对，返回(K,(Seq[V],Seq[W]))对新DStreams
transform(func)	将RDD到RDD映射的函数func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。
updateStateByKey(func)	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。

### 2.2.1 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。

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。

## 2.2.2 有状态转化操作

### 特殊的Transformations

#### 2.2.2.1 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。

#### 2.2.2.2 Window Operations

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

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

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。

`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

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，

### 2.3 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
<b>print()</b>	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pprint()。
<b>saveAsTextFiles</b> ( <i>prefix</i> , <i>[suffix]</i> )	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]"。
<b>saveAsObjectFiles</b> ( <i>prefix</i> , <i>[suffix]</i> )	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。
<b>saveAsHadoopFiles</b> ( <i>prefix</i> , <i>[suffix]</i> )	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。
<b>foreachRDD</b> ( <i>func</i> )	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

### 3、案例实战

netcat是网络工具中的瑞士军刀，它通过TCP和UDP在网络中读写数据。通过与其他工具结合和重定向，你可以在脚本中以多种方式使用它。使用netcat命令所能完成的事情令人惊讶。

netcat所做的就是在两台电脑之间建立链接并返回两个数据流，在这之后所能做的事就看你的想像力了。你能建立一个服务器，传输文件，与朋友聊天，传输流媒体或者用它作为其它协议的独立客户端。



```
nc -lk 8888
```

在服务器 A 执行以上命令，将会把 nc 绑定到 9090 端口，并开始监听请求。-l 代表 netcat 将以监听模式运行；-k 表示 nc 在接收完一个请求后不会立即退出，而是会继续监听其他请求。这时就可以请求该接口了，nc 会把请求报文输出到标准输出。

### 3.1、SparkStreamingWordCount

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

    val conf = new SparkConf().setAppName("SparkStreamingWC").setMaster("local[2]")
    val sc = new SparkContext(conf)
    // 创建SparkStreaming的上下文对象
    //创建sparkStreaming的执行入口，也就是Streaming对象
    //我们设置的时间间隔，也就是每一批数据，叫batch
    val ssc: StreamingContext = new StreamingContext(sc, Seconds(5))

    // 首先，创建输入DStream，代表了一个从数据源（比如kafka、socket）来的持续不断的实时数据流
    // socketTextStream()方法接收两个基本参数，第一个是监听哪个主机上的ip，第二个是监听哪个端口
    // 从NetCat服务里获取数据
    val dStream: ReceiverInputDStream[String] = ssc.socketTextStream("node01", 8888)
    //到这里为止，你可以理解为DStream，每秒进行封装一次RDD，也就是这5秒的RDD
    //RDD的元素是一行一行的文本
    //接下来，开始接受数据，并执行计算，使用SparkCore算子，执行操作在DStream中完成
    //其实底层就在DStream中产生新的RDD
    // 调用DStream里的api进行计算
    val res: DStream[(String, Int)] = dStream.flatMap(_.split(" ")).map((_, 1)).reduceByKey(_+_ )

    res.print()

    // 提交任务到集群
    ssc.start()

    // 线程等待，等待处理任务
    ssc.awaitTermination()

  }
}
```

### 3.2、按批次累加updateStateByKey

```
/**
 * 实现按批次累加功能，需要调用updateStateByKey
 * 其中需要自定义一个函数，该函数是对历史结果数据和当前批次数据的操作过程
 * 该函数中第一个参数代表每个单词
 * 第二个参数代表当前批次单词出现的次数：Seq(1,1,1,1)
 * 第三个参数代表之前批次累加的结果，可能有值，也可能没有值，所以在获取的时候要用getOrElse方法
 */
```

```

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

    val conf = new SparkConf().setAppName("SparkStreamingACCWC").setMaster("local[2]")
    val ssc = new StreamingContext(conf, Milliseconds(5000))

    // 设置检查点目录
    // ssc.checkpoint("hdfs://node01:9000/cp-20180306-1")
    ssc.checkpoint("c://cp-20180306-1")

    // 获取数据
    val dStream = ssc.socketTextStream("node01", 8888)
    val tup: DStream[(String, Int)] = dStream.flatMap(_.split(" ")).map((_, 1))
    val res: DStream[(String, Int)] =
      tup.updateStateByKey(func, new HashPartitioner(ssc.sparkContext.defaultParallelism), true)

    res.print()

    ssc.start()
    ssc.awaitTermination()
  }

  val func = (it: Iterator[(String, Seq[Int], Option[Int])]) => {
    it.map(x => {
      (x._1, x._2.sum + x._3.getOrElse(0))
    })
  }
}

```

### 3.3 transform

transform操作，应用在DStream上时，可以用于执行任意的RDD到RDD的转换操作。它可以用于实现，DStream API中所没有提供的操作。比如说，DStream API中，并没有提供将一个DStream中的每个batch，与一个特定的RDD进行join的操作。但是我们自己就可以使用transform操作来实现该功能。

DStream.join()，只能join其他DStream。在DStream每个batch的RDD计算出来之后，会去跟其他DStream的RDD进行join。

```

object BlackListFilter {
  def main(args: Array[String]): Unit = {
    val sparkConf = new SparkConf().setAppName("BlackListFilter").setMaster("local[2]")
    val ssc = new StreamingContext(sparkConf, Seconds(5))
    //设置黑名单
    val blackList = Array(("jack", true), ("tom", true))
    //设置并行度
    val blackListRDD = ssc.sparkContext.parallelize(blackList, 5)
    //使用socketTextStream来监听端口，也就是接收到的实时数据
    val socketData = ssc.socketTextStream("192.168.14.131", 9999)
    //接下来将接收到的数据转换成和我们黑名单对应得形式数据格式
    //返回的数据格式就是(username, date username)
    val users = socketData.map(line => (line.split(" ")(1), line))

    //使用Spark Streaming中的transform算子操作，实现过滤
  }
}

```

```

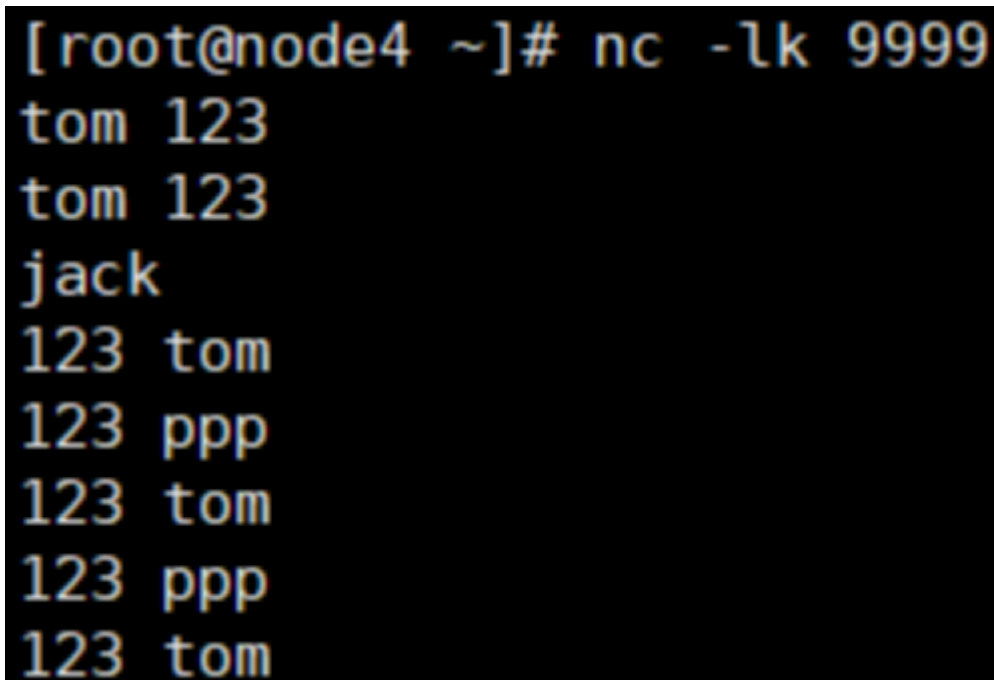
val validRddDS = users.transform(user=>{

    val joinRDD = user.leftOuterJoin(blackListRDD)
    //接下来调用filter算子实现join后的数据过滤
    val filterRDDs = joinRDD.filter(tuple=>{

        if(tuple._2._2.getOrElse(false)){
            false
        }else{
            true
        }
    })
    //我们将黑名单过滤后，将处理真正的白名单数据
    val validRDD = filterRDDs.map(tuple=>tuple._2._1)
    validRDD
})
//进行数据的显示打印
validRddDS.print()
ssc.start()
ssc.awaitTermination()
}
}

```

接下来启动nc



```

[root@node4 ~]# nc -lk 9999
tom 123
tom 123
jack
123 tom
123 ppp
123 tom
123 ppp
123 tom

```

### 3.4、获取Kafka数据并按批次累加

```

object LoadKafkaDataAndWC {
    def main(args: Array[String]): Unit = {
        LoggerLevels.setStreamingLogLevels()

        val conf = new SparkConf().setAppName("LoadKafkaDataAndWC").setMaster("local[2]")
    }
}

```

```

val ssc = new StreamingContext(conf, Seconds(5))

// 设置请求kafka的几个参数
val Array(zkQuorum, group, topics, numTheads) = args

// 设置检查点
ssc.checkpoint("c://cp-20180306-2")

// 获取每一个topic并放到一个Map里
val topicMap: Map[String, Int] = topics.split(",").map((_, numTheads.toInt)).toMap

// 调用KafkaUtils工具类获取kafka的数据
val data: ReceiverInputDStream[(String, String)] =
    KafkaUtils.createStream(ssc, zkQuorum, group, topicMap)

// 因为DStream里的key是offset值, 把DStream里的value数据取出来
val lines: DStream[String] = data.map(_._2)
val tup = lines.flatMap(_.split(" ")).map((_, 1))
val res: DStream[(String, Int)] = tup.updateStateByKey(func, new
HashPartitioner(ssc.sparkContext.defaultParallelism), true)

res.print()

ssc.start()
ssc.awaitTermination()
}

val func = (it: Iterator[(String, Seq[Int], Option[Int])]) => {
    it.map{
        case (x, y, z) => {
            (x, y.sum + z.getOrElse(0))
        }
    }
}
}
}

```

### 3.5、窗口操作

```

object WindowOperationWC {
    def main(args: Array[String]): Unit = {
        LoggerLevels.setStreamingLogLevels()

        val conf = new SparkConf().setAppName("WindowOperationWC").setMaster("local[2]")
        val ssc = new StreamingContext(conf, Seconds(5))

        ssc.checkpoint("c://cp-20180306-3")

        val dStream = ssc.socketTextStream("node01", 8888)

        val tup = dStream.flatMap(_.split(" ")).map((_, 1))

        // 调用窗口操作来计算数据的聚合。批次间隔是5秒, 设置窗口长度是10, 滑动间隔是10秒
    }
}

```

```
val res: DStream[(String, Int)] =  
  tup.reduceByKeyAndWindow((x: Int, y: Int) => (x + y), Seconds(10), Seconds(10))  
  
res.print()  
  
ssc.start()  
ssc.awaitTermination()  
}  
}
```