# SparkStreaming

1、Spark Streaming 简介

1.1、概述

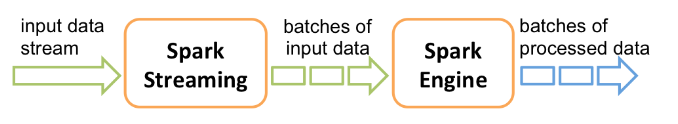
Spark Streaming 是 Spark 核心 API 的一个扩展，可以实现高吞吐量的、具备容错机制的实时流数据的处理。支持从多种数据源获取数据，包括 Kafk、Flume、Twitter、ZeroMQ、Kinesis以及TCP sockets，从数据源获取数据之后，可以使用诸如 map、reduce、join 和 window 等高级函数进行复杂算法的处理。最后还可以将处理结果存储到文件系统，数据库和现场仪表盘。

在“One Stack rule them all”的基础上，还可以使用 Spark 的其他子框架，如集群学习、图计算等，对流数据进行处理。

Spark Streaming 处理的数据流图：



Spark 的各个子框架，都是基于Spark Core的，Spark Streaming在内部的处理机制是，Spark Streaming接受实时传入的数据流，然后将数据按批次（batch）进行划分，然后再将这部分数据交由Spark引擎进行处理，处理完成后将结果输出到外部文件。



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

1.2、术语定义

* **离散流（discretized stream）或DStream：**这是 Spark Streaming 对内部持续的实时数据流的抽象描述，即我们处理的一个实时数据流，在 Spark Streaming 中对应于一个DStream 实例。
* **批数据（batch data）：**这是化整为零的第一步，将实时流数据以时间片为单位进行分批，将流处理转化为时间片数据的批处理。随着持续时间的推移，这些处理结果就形成了对应的结果数据流了。
* **时间片或批处理时间间隔（batch interval）：**这是人为地对流数据进行定量的标准，以时间片作为我们拆分流数据的依据。一个时间片的数据对应一个 RDD 实例。
* **窗口长度（window length）：**一个窗口覆盖的流数据的时间长度。必须是批处理时间间隔的倍数，
* **滑动时间间隔：**前一个窗口到后一个窗口所经过的时间长度。必须是批处理时间间隔的倍数
* **Input DStream :**一个input DStream是一个特殊的DStream，将 Spark Streaming 连接到一个外部数据源来读取数据。

1.3、Storm 与 Spark Streming 比较

* **处理模型以及延迟**

虽然两框架都提供可扩展性(scalability)和可容错性(fault tolerance)，但是它们的处理模型从根本上说是不一样的。Storm 可以实现亚秒级时延的处理，而每次只处理一条 event，而Spark Streaming 可以在一个短暂的时间窗口里面处理多条(batches)Event。所以说 Storm 可以实现亚秒级时延的处理，而Spark Streaming 则有一定的时延。

* **容错和数据保证**

然而两者的代价都是容错时候的数据保证，Spark Streaming 的容错为有状态的计算提供了更好的支持。在 Storm 中，每条记录在系统的移动过程中都需要被标记跟踪，所以 Storm 只能保证每条记录最少被处理一次，但是允许从错误状态恢复时被处理多次。这就意味着可变更的状态可能被更新两次从而导致结果不正确。

任一方面，Spark Streaming 仅仅需要在批处理级别对记录进行追踪，所以他能保证每个批处理记录仅仅被处理一次，即使是 node 节点挂掉。虽然说 Storm 的 Trident library 可以保证一条记录被处理一次，但是它依赖于事务更新状态，而这个过程是很慢的，并且需要由用户去实现。

* **实现和编程 API**

Storm 主要是由 Clojure 语言实现，Spark Streaming 是由 Scala 实现。如果你想看看这两个框架是如何实现的或者你想自定义一些东西你就得记住这一点。Storm 是由 BackType 和Twitter 开发，而 Spark Streaming 是在 UC Berkeley 开发的。

Storm 提供了 Java API，同时也支持其他语言的 API。 Spark Streaming 支持 Scala 和 Java语言(其实也支持 Python)。

* **批处理框架集成**

Spark Streaming 的一个很棒的特性就是它是在 Spark 框架上运行的。这样你就可以想使用其他批处理代码一样来写 Spark Streaming 程序，或者是在 Spark 中交互查询。这就减少了单独编写流批量处理程序和历史数据处理程序。

* **生产支持**

Storm 已经出现好多年了，而且自从 2011 年开始就在 Twitter 内部生产环境中使用，还有其他一些公司。而 Spark Streaming 是一个新的项目，并且在 2013 年仅仅被 Sharethrough使用(据作者了解)。

Storm 是 Hortonworks Hadoop 数据平台中流处理的解决方案，而 Spark Streaming 出现在 MapR 的分布式平台和 Cloudera 的企业数据平台中。除此之外，Databricks 是为 Spark提供技术支持的公司，包括了 Spark Streaming。

虽然说两者都可以在各自的集群框架中运行，但是 Storm 可以在 Mesos 上运行, 而 SparkStreaming 可以在 YARN 和 Mesos 上运行。

Storm/JStorm

完全的实时流计算框架；

一条数据就处理一条数据，数据的延迟性比较低；在高并发、大数据量的情况下，对机器的性能要求比较高。

SparkStreaming

准实时/微观的流式数据计算框架/微批量计算；

一个批次一个批次的进行数据处理，只有当前批次处理完成后，下一个批次才会开始进行处理；如果当前批次执行的时间太长，就会导致后续的批次延迟执行。

相比Storm的来讲：

**优点：**

Spark的体系结构比较完备.

对于机器的性能要求相比于Storm而言要低一点

在2w/s的数据量情况下，一般的配置机器(8cpu+32g)处理的速度也还行.

**缺点：**

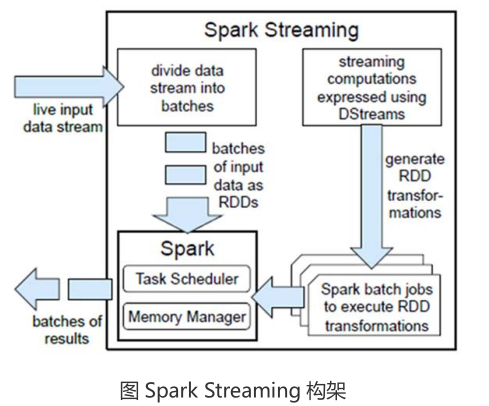
没法实现完全实时数据处理.

在高并发、大数据量的情况下，相比于Storm的延迟性会高一点.

2、运行原理

2.1、Streaming 架构

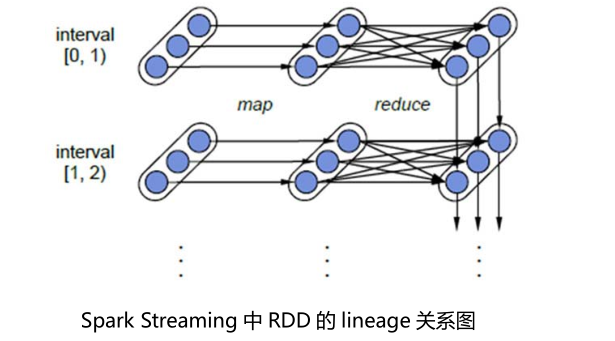
SparkStreaming 是一个对实时数据流进行高通量、容错处理的流式处理系统，可以对多种数据源（如 Kafka、Flume、Twitter、Zero 和 TCP 套接字）进行类似 Map、Reduce 和 Join等复杂操作，并将结果保存到外部文件系统、数据库或应用到实时仪表盘。



 **计算流程：**Spark Streaming 是将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是 Spark Core，也就是把 Spark Streaming 的输入数据按照 batch size（如 1 秒）分成一段一段的数据（Discretized Stream），每一段数据都转换成 Spark 中的 RDD（Resilient Distributed Dataset），然后将 Spark Streaming 中对 DStream 的 Transformation 操作变为针对 Spark 中对 RDD 的 Transformation 操作，将 RDD 经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加或者存储到外部设备。上图显示了 Spark Streaming 的整个流程。

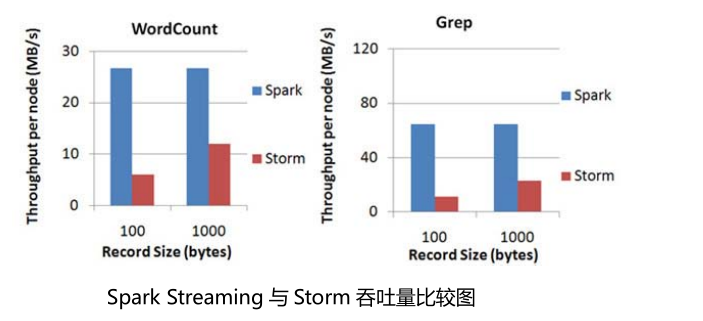
**容错性：**对于流式计算来说，容错性至关重要。首先我们要明确一下Spark 中 RDD的容错机制。每一个RDD都是一个不可变的分布式可重算的数据集，其记录着确定性的操作继承关系（lineage），所以只要输入数据是可容错的，那么任意一个RDD的分区（Partition）出错或不可用，都是可以利用原始输入数据通过转换操作而重新算出的。

对于Spark Streaming 来说，其RDD的传承关系如下图所示，图中的每一个椭圆形表示一个RDD，椭圆形中的每个圆形代表一RDD中的一个Partition，图中的每一列的多个RDD表示一个DStream（图中有三个 DStream），而每一行最后一个RDD则表示每一个Batch Size所产生的中间结果RDD。我们可以看到图中的每一个 RDD都是通过lineage相连接的，由于 Spark Streaming 输入数据可以来自于磁盘，例如HDFS（多份拷贝）或是来自于网络的数据流（Spark Streaming 会将网络输入数据的每一个数据流拷贝两份到其他的机器）都能保证容错性，所以RDD中任意的Partition出错，都可以并行地在其他机器上将缺失的Partition计算出来。这个容错恢复方式比连续计算模型（如 Storm）的效率更高。



**实时性：**对于实时性的讨论，会牵涉到流式处理框架的应用场景。Spark Streaming 将流式计算分解成多个 Spark Job，对于每一段数据的处理都会经过 Spark DAG 图分解以及 Spark的任务集的调度过程。对于目前版本的 Spark Streaming 而言，其最小的 Batch Size 的选取在 0.5~2 秒钟之间（Storm 目前最小的延迟是 100ms 左右），所以 Spark Streaming 能够满足除对实时性要求非常高（如高频实时交易）之外的所有流式准实时计算场景。

**扩展性与吞吐量：**Spark 目前在 EC2 上已能够线性扩展到 100 个节点（每个节点 4Core），可以以数秒的延迟处理 6GB/s 的数据量（60M records/s），其吞吐量也比流行的 Storm 高2～5 倍，图中是 Berkeley 利用 WordCount 和 Grep 两个用例所做的测试，在 Grep 这个测试中，Spark Streaming 中的每个节点的吞吐量是 670k records/s，而 Storm 是 115k records/s。



### 2.2、SparkStreaming的处理流程

**spark-core：RDD、 spark-sql：DataFrame、 Spark-Stream：Dstram**

**1）读取数据形成DStream**

读取外部的数据形成DStream，eg: KAFKA、Flume...

**2）数据处理**

DStream的相关API进行处理

DStream的API内部的实际执行是RDD对应的API的执行

**3）数据的输出（打印和保存）**

数据保存到外部的存储系统或者返回给driver

(1) 返回给**driver**

dstream.print dstream.foreachRDD

(2) 存储到外部的存储系统

dstream.saveXXX dstream.foreachRDD

**主要的存储系统：**

--1. Redis、MongoDB --2. RDBMs

--3. HBase、Hive、HDFS **--4. Kafka**

// 1、上下文构建

val conf = new SparkConf()

.setMaster("local[10]")

.setAppName("streaming-wordcount")

val sc = SparkContext.getOrCreate(conf)

/\*\*

\* Seconds(10): 指定的是DStream的批次产生间隔时间，

\* 也就是每隔10s指定的时间后，就产生一个待执行的批次

\*/

val ssc = new StreamingContext(sc, Seconds(10))

// 2、DStream的构建

val dstream = ssc.socketTextStream("ip 或 hostname", 9999)

// 3、DStream的数据处理

val resultDStream = dstream

.flatMap(line => line.split(" "))

.filter(word => word.nonEmpty)

.map((\_, 1))

.reduceByKey(\_ + \_)

// 4、DStream结果输出

//4. 1. 输出到driver

resultDStream.print()

// 4.2. 存储到外部的存储系统中

/\*\*

\* saveAsTextFiles：底层调用RDD的saveAsTextFile API，将DStream中的数据

\* 写出到HDFS上的指定文件夹中，指定的文件夹路径为:

\* ${prefix}-batchTime.${suffix}; 每个批次的结果保存一个文件夹

\*/

resultDStream.saveAsTextFiles(prefix = "result/streaming/wc",

suffix = "170505")

// 5、友好的关闭方式

// 5.1. 设置参数:spark.streaming.stopGracefullyOnShutdown为true，

// 表示jvm退出的时候进行关闭操作

// 5.2. 使用后台的线程进行关闭操作

/\*new Thread(new Runnable {

override def run(): Unit = {

// TODO: 根据外部的变量决定是否进行关闭操作

// 1. 连接mysql

// 2. 从mysql中获取变量值

// 3. 判断变量之是否是运行状态，如果是，休息一段时间后，

// 继续做2的操作；否则进入4

// 4. 关闭

/\*var isRunning = true

while(isRunning) {

val flag:Boolean = conn.xxxx

if (flag) {

isRunning = false

} else {

Thread.sleep(60000)

}

}\*/

var isRunning = true

while(isRunning) {

Thread.sleep(60000)

isRunning = false

}

// 4. 关闭操作

ssc.stop()

}

}).start()\*/

// 6、启动SparkStreaming开启数据处理

ssc.start()

ssc.awaitTermination() // 阻塞，等待程序的遇到中断等操作

// 7、关闭sparkstreaming的程序运行

ssc.stop()

}

}

### **2.3、SparkStreaming的执行流程(使用数据接收器的形式)**

1）数据接收器(Receiver)

Receiver接收数据的输入，并将输入数据以Block块的形式存储在内存或者磁盘中(会进行备份)

**默认情况下200ms会产生一个block块**，由参数:

spark.streaming.blockInterval控制

一般情况下设置为1~2s，一般为:

**batchDuration** / (executor num \* cpu cores pre executor \* (0.8~2))

在最终的运行过程中，**一个Block块就对应一个RDD的分区**，也就是一个RDD的task只会处理一个Block块的数据 ==> 一个批次中有多少个block块，就会有多少个task任务(读取block块数据的第一个stage)

Receiver就会将Block块的存储信息汇报给Driver中的StreamingContext

2）batch/批次/rdd的产生

def this(conf: SparkConf, batchDuration: Duration)

每隔batchDuration指定的间隔时间后，就会产生一个批次，实质上一个批次其实就是一个RDD；这一个RDD负责处理的数据就是这个批次中所接收到的所有数据；**最好要求batchDuration是blockInterval的整数倍200ms的整数倍**

每个批次运行的其实就是RDD中的DAG图 ==> DAG图的构建其实是DStream API调用的结果，当DStream调用非输出API的时候，其实就是在底层构建DAG执行图

RDD中的每个Task对应一个Block块(指的是第一个读取Block块数据的Stage中的Task)

3）批次的执行/RDD action算子的触发

当调用DStream的输出类型的API的时候，底层就会触发rdd的action类型算子的调用

DStream的批次执行其实就是DStream中产生的RDD的执行

一个DStream中每个批次只会有一个RDD生成；但是一个批次中一个DStream可以触发多次数据输出API，也就是可以有多个RDD Job的运行；

**3、程序入口：Streaming SqlContext及核心抽象DStream**

**3.1、Streaming SqlContext**

Spark-core -> SparkContext

Spark-SQL -> SqlContext HiveContet

Spark-Streaming -> StreamingContext

SparkStreaming应用的上下文，依赖SparkContext。在构建的过程中，需要给定SparkContext对象以及批次产生间隔时间(batchDuration)。

val conf = new SparkConf()

.setMaster("local[5]")

.setAppName("KafkaReceiverWordCount")

.set("spark.streaming.blockInterval", "5s") // 5s产生一个block块

val sc = SparkContext.getOrCreate(conf)

val ssc = new StreamingContext(sc, Seconds(10))

**因为存在一个数据接收器Job, 需要占用一个task的执行名额，所以在本地运行的时候至少给定local[2]，在集群中总的executor的cpu-cores数量至少为2 ===> 在本地模式下，数据接收器的task宕机后，资源不会释放；集群不用考虑。**

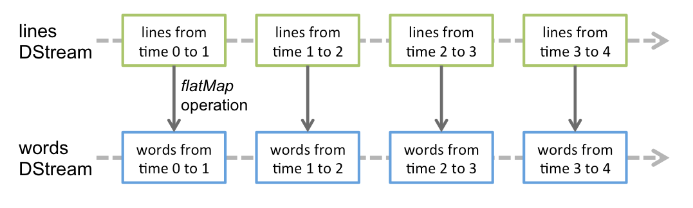
**3.2、DStream**

3.2.1、什么是Dstream

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



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



3.2.2、DStream底层

**DStream**底层是由一系列的RDD和批次时间组成的一个集合，所以DStream中的数据也是分区存在而且不可变的；每个RDD均包含对应批次的数据(数据源+数据处理函数)，DStream底层的API实质上就是RDD API的调用：

\* - A list of other DStreams that the DStream depends on

DStream具有和RDD类型的依赖关系，除了第一个DStream外(InputDStream)，其它的DStream均具有父DStream

当DStream调用数据处理API的时候，就会构建出一个DStream的依赖关系图(DAG执行图)；实质上来讲，DStream的依赖关系图其实就是最终每个批次RDD执行时候的RDD DAG图的构建

\* - A time interval at which the DStream generates an RDD

每个DStream具有一个时间间隔参数，每隔这个时间就会产生一个批次/RDD

**InputDStream的时间间隔默认为构建StreamingContext的时候给定的batchDuration的值**

子DStream的批次产生间隔时间默认就等于父DStream的间隔时间；**除了Window类型的DStream，window类型的DStream的批次产生间隔时间是父DStream的间隔时间的整数倍**

\* - A function that is used to generate an RDD after each time interval

DStream中对于每个批次来讲，均有一个函数用来创建该批次对应的RDD: compute

3.2.3、RDD的销毁/Block的销毁

**默认情况下，只会销毁数据，下下一个批次执行完成后，会将当前批次创建的RDD以及RDD读取的Block块的原始数据(Executor中保存的Block块信息)进行销毁操作**

删除的前提：被删除的RDD、Block块数据已经被处理，而且在之后的批次中不可能再被处理(RDD不会被其它RDD(后续可能会运行)依赖)

可以通过streaming的API来进行修改RDD的生命周期的长度：

ssc.remember(Seconds(30)) ==> 要求给定的参数的时间间隔必须比原始的批次产生间隔大

备注：在SparkStreaming程序中，必须设置相关参数进行元数据的删除(eg: RDD、State、Job.....)

spark.cleaner.ttl:1d ==> 表示元数据最多存储1天，超过1天的元数据会被删除

**4、SparkStreaming的数据源类别**

4.1、Basic Sources

基于StreamingContext的提供的API进行创建构建而成的DStream对象。一般情况下，是基于数据接收器（Receiver）形式的一个DStream。

**常用的API：**

ssc.socketTextStream: 以文本的形式读取TCP端口的数据；

ssc.socketStream: 以给定的数据转换方式读取TCP端口的数据；

ssc.receiverStream: 基于一个给定的数据接收器对象进行数据接收并形成DStream。

**4.2、Advanced Sources 外部工具类**

基于外部数据的API读取数据(非Streaming模块的API，非StreamingContext的API)。

eg：Kafka、Flume.....

一般分为两种数据读取方式：

--a. Use Receiver：使用数据接收器

--b. Direct Approach：直接的数据读取方式

**4.2.1、Use Receiver**

在正常的job task之外，启动一个一直运行的task专门用于接收数据，并将接收的数据以block块的形式保存到本地的内存/磁盘, 这个task就叫做receiver task；

在这种情况下，每个批次产生的RDD中的初始分区数量默认为一个批次中所接收数据最终形成的Block块的数量，一个Block块就是一个分区，也就是对应一个最终task。

val conf = new SparkConf()  
 .setMaster("local[2]")  
 .setAppName("streaming-kafka")  
 val sc = SparkContext.getOrCreate(conf)  
 val ssc = new StreamingContext(sc,Seconds(5))  
 val kafkaParams = Map[String,String](  
 "zookeeper.connect" -> "192.168.204.111:2181",  
 "group.id"->"streaming1",  
 "auto.offset.reset"->"largest",  
 "zookeeper.connection.timeout.ms"->"10000"  
 )  
 val topics =Map[String,Int]("testdemo"->4)  
 val msg = KafkaUtils.createStream[String,String,StringDecoder,StringDecoder]

(ssc,kafkaParams,topics,StorageLevel.MEMORY\_AND\_DISK\_2)  
 val wordcount =msg.map(\_.\_2).

flatMap(\_.split(" "))

.filter(\_.nonEmpty).

map((\_,1)).

reduceByKey(\_+\_)  
 wordcount.print()  
 ssc.start()  
 ssc.awaitTermination()

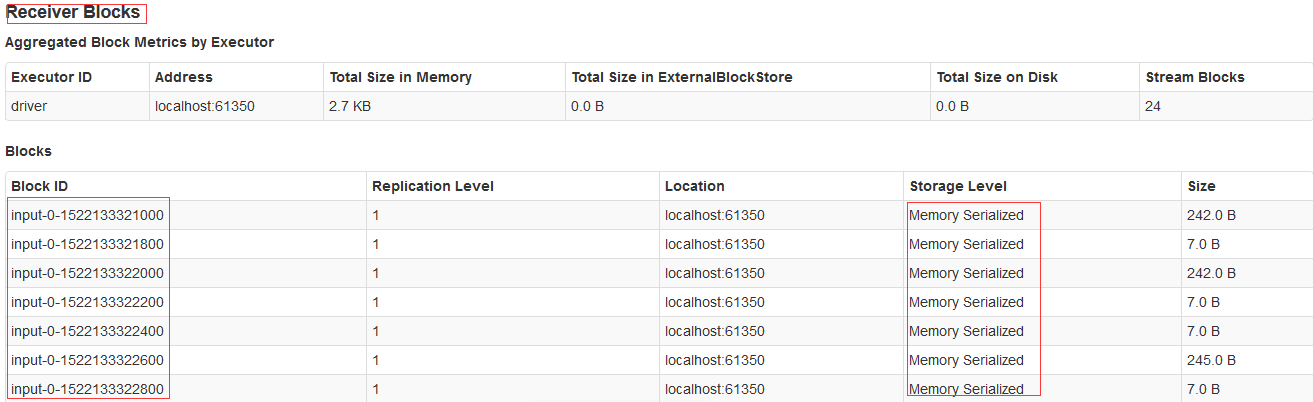


图-1 executor的receiver接收的block

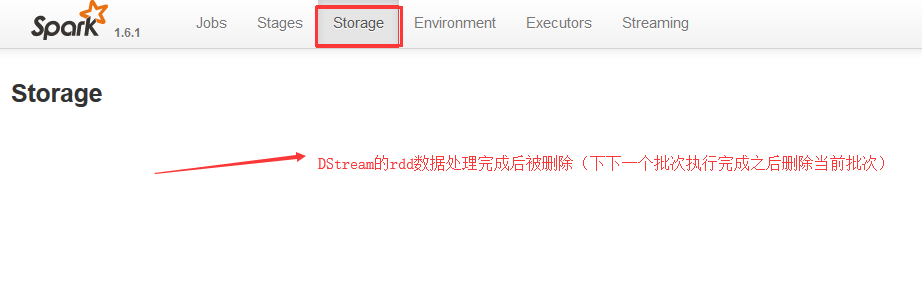


图-2 数据处理完成删除数据

4.2.2、Direct Approace（直接数据读取）

**不会启动额外的receiver task任务，也就是不会形成Block块**

在每个批次产生的RDD中，存储的是这个RDD处理的原始数据存储的位置信息(eg: kafka集成中存储的就是hostname、topic、partition、offset...)，然后直到RDD真正的运行的时候，才会从数据源基于RDD中保存的相关元数据读取数据源中的对应数据，然后对这些数据进行后续的dstream上的转换操作 ==> 数据不会形成Block块，也就不会将Block块保存到Executor中。

RDD的初始分区数量由具体的集成代码决定(和集成代码的实现方式以及数据源有关系)。

val conf = new SparkConf()  
 .setMaster("local[2]")  
 .setAppName("KafkaDirectWordCount")  
 val sc = SparkContext.getOrCreate(conf)  
 val ssc = new StreamingContext(sc, Seconds(5))  
 // 二、DStream的构建  
 // kafka的Simple consumer API的连接参数， 只有两个  
 // metadata.broker.list: 给定Kafka的服务器路径信息  
 // auto.offset.reset：给定consumer的偏移量的值，largest表示设置为最大值，

// smallest表示设置为最小值(最大值&最小值指的是对应的分区中的日志数据的偏移量的值) // ==> 每次启动都生效  
 val kafkaParams = Map[String, String](  
 "metadata.broker.list" -> "192.168.204.111:9092,192.168.204.111:9093,

192.168.204.111:9094,192.168.204.111:9095",  
 "auto.offset.reset" -> "smallest" )  
 // 给定一个由topic名称组成的set集合  
 val topics = Set("testdemo")  
 // 构建DStream  
 val dstream = KafkaUtils.createDirectStream[String, String, StringDecoder,

StringDecoder](ssc, kafkaParams, topics).map(\_.\_2)  
 // 三、DStream的数据处理  
 val resultDStream = dstream  
 .flatMap(line => line.split(" "))  
 .filter(word => word.nonEmpty)  
 .map(word => (word, 1))  
 .reduceByKey(\_ + \_)  
 // 四、DStream结果输出  
 // 1. 输出到driver  
 resultDStream.print()  
 // 六、启动SparkStreaming开启数据处理  
 ssc.start()  
 ssc.awaitTermination() // 阻塞，等待程序的遇到中断等操作  
 // 七、关闭sparkstreaming的程序运行  
 ssc.stop()

5、SparkStreaming 和kafka集成

5.1、添加pom依赖

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-streaming-kafka\_2.10</artifactId>

<version>1.6.1</version>

<scope>compile</scope>

</dependency>

5.2、Receiver-based Approach（基于数据接收器的集成方式）

这种方法使用Receiver来接收数据。Receiver是使用**Kafka高级消费者API**实现的。与所有接收方一样，通过Receiver从Kafka接收的数据存储在Spark执行程序中，然后由Spark Streaming启动的作业处理数据。此时SparkStreaming程序中的receiver task相当于是kafka的一个consumer，源源不断的将kafka的数据接收过来并以block块的形式保存到executor中。

最终形成的Block中的数据不会保持Kafka分区数据有序的特性 ==> 后续的数据处理过程中，没法保证数据的处理有序(局部有序)。

底层使用Kafka的High Level Consumer API, 所以基本的数据读取、offset管理等操作均使用consumer的参数来进行控制。会将consumer的offset偏移量默认提交到Zookeeper中进行保存。RDD的初始分区数就是对应批次中接收的所有black块的数量。

但是，在默认配置下，这种方法可能会在失败时丢失数据。为确保零数据丢失，您必须另外启用Spark Streaming中的预写日志（在Spark 1.2中引入），同时保存所有收到的Kafka数据写入分布式文件系统（例如HDFS）的预先写入日志中，以便在发生故障时恢复所有数据。

**需要注意的是：**　　1、Kafka中Topic的分区和Spark Streaming生成的RDD中分区不是一个概念。所以，在KafkaUtils.createStream()增加特定主题分区数仅仅是增加一个receiver中消费Topic的线程数。并不增加Spark并行处理数据的数量；

　　2、对于不同的Group和tpoic我们可以使用多个receivers创建不同的DStreams来并行接收数据；

3、如果你启用了WAL，这些接收到的数据将会被持久化到日志中，因此，我们需要将storage level 设置为StorageLevel.MEMORY\_AND\_DISK\_SER ,也就是：KafkaUtils.createStream(..., StorageLevel.MEMORY\_AND\_DISK\_SER)。

5.3、Direct Approach (No Receivers) native Offsets

直接kafka的相关API进行数据的读取(根据给定的offset偏移量)，然后进行数据的处理操作，这个数据默认情况下不会保存到executor中。

最终形成的RDD中，RDD的初始分区数量就是对应的Kafka的Topic的分区数量，也就是一个Kafka的Topic分区就对应一个RDD的分区，一个Kafak的分区数据由一个Task进行处理 ==> 读取数据的第一个Stage可以保证数据的有序性(Kafak内分区内部数据有序)。

内部使用Kafka的Simple Consumer API进行数据的读取 ==> 要求在调用API形成DStream的时候必须给定kafka的集群位置以及读取数据的偏移量信息 ===> 默认情况下，consumer的消费offset偏移量不会保存到zk中 ==> 需要你在代码中进行offset偏移量的保存操作，native Offsets，“消费者本地保存自己的offset”。

和基于Receiver方式相比，这种方式主要有一些几个优点：  
　　（1）**简化并行。**我们不需要创建多个Kafka 输入流，然后union他们。而使用directStream，Spark Streaming将会创建和Kafka分区一样的RDD分区个数，而且会从Kafka并行地读取数据，也就是说Spark分区将会和Kafka分区有一一对应的关系，这对我们来说很容易理解和使用；

　　（2）**高效。**第一种实现零数据丢失是通过将数据预先保存在WAL中，这将会复制一遍数据，这种方式实际上很不高效，因为这导致了数据被拷贝两次：一次是被Kafka复制；另一次是写到WAL中。但是本文介绍的方法因为没有Receiver，从而消除了这个问题，所以不需要WAL日志；

　　（3）**恰好一次语义（Exactly-once semantics）。**通过使用Kafka高层次的API把偏移量写入Zookeeper中，这是读取Kafka中数据的传统方法。虽然这种方法可以保证零数据丢失，但是还是存在一些情况导致数据会丢失，因为在失败情况下通过Spark Streaming读取偏移量和Zookeeper中存储的偏移量可能不一致。而此处提到的方法是通过Kafka低层次的API，并没有使用到Zookeeper，偏移量仅仅被Spark Streaming保存在Checkpoint中。这就消除了Spark Streaming和Zookeeper中偏移量的不一致，而且可以保证每个记录仅仅被Spark Streaming读取一次，即使是出现故障。

5.4、优化方式

-1. Use Receiver

--a. 将参数spark.streaming.blockInterval进行参数调整

--b. 可以考虑使用多个Receiver的形式，然后将DStream进行union合并即可

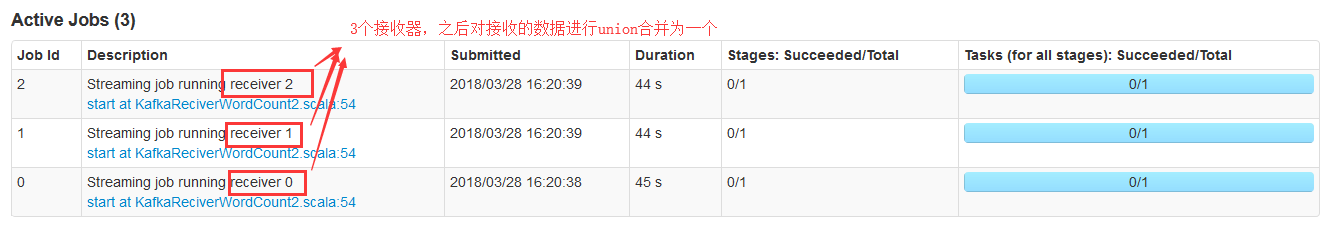
要求：

合并的DStream的ssc属性是同一个(属于同一个StreamingContext对象)

合并的两个DStream的批次产生间隔时间一致

val conf = new SparkConf()  
 .setMaster("local[\*]") //多个接收器每个都需要一个线程，这里需要设置多个线程来处理  
 .setAppName("StreamingWordCount2")  
//调整每个批次中，block生成的间隔时间  
 conf.set("spark.streaming.blockInterval","500ms")

val msg1: DStream[String] = KafkaUtils.createStream[String,String,StringDecoder,StringDecoder](ssc,kafkaParams,topics,StorageLevel.MEMORY\_AND\_DISK\_SER).map(\_.\_2)  
 val msg2: DStream[String] = KafkaUtils.createStream[String,String,StringDecoder,StringDecoder](ssc,kafkaParams,topics,StorageLevel.MEMORY\_AND\_DISK\_SER).map(\_.\_2)  
 val msg3: DStream[String] = KafkaUtils.createStream[String,String,StringDecoder,StringDecoder](ssc,kafkaParams,topics,StorageLevel.MEMORY\_AND\_DISK\_SER).map(\_.\_2)  
  
 //合并接收器形成的input DStream  
val msg = msg1.union(msg2).union(msg3)



-2. Direct

--a. 开启Spark的动态executor调度机制(解决访问高峰和低谷的问题)

--b. 开启streaming的动态数据调整处理机制。

功能：限制每个批次的最多数据处理量，可以防止出现意料之外的 执行时间超过批次间隔时间的情况。

参数：

spark.streaming.backpressure.enabled:false，当设置为true的时候， 表示开启动态调整机制。

spark.streaming.receiver.maxRate: 给定当使用receiver模式的情 况下，一个批次中receiver最多的数据接收量

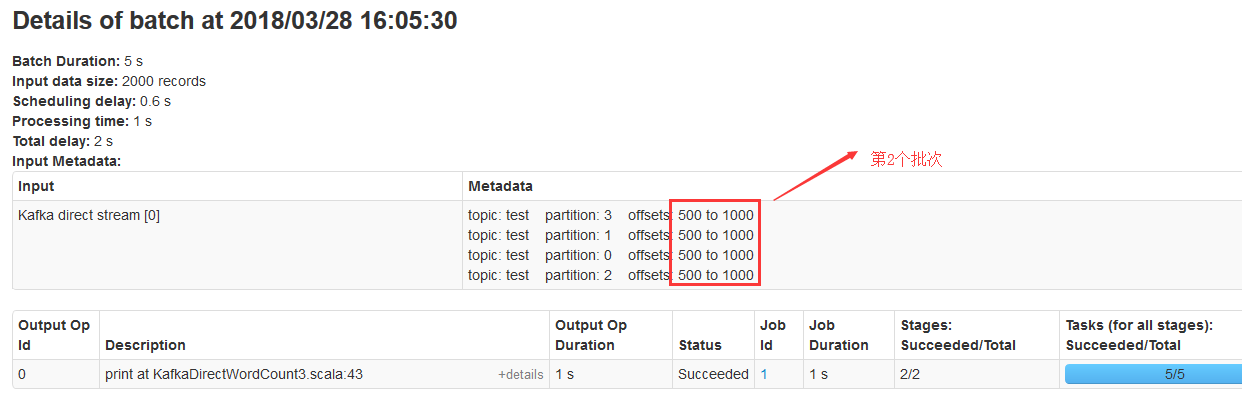
(在sparkstreaming和kafka的集成中该参数无用)

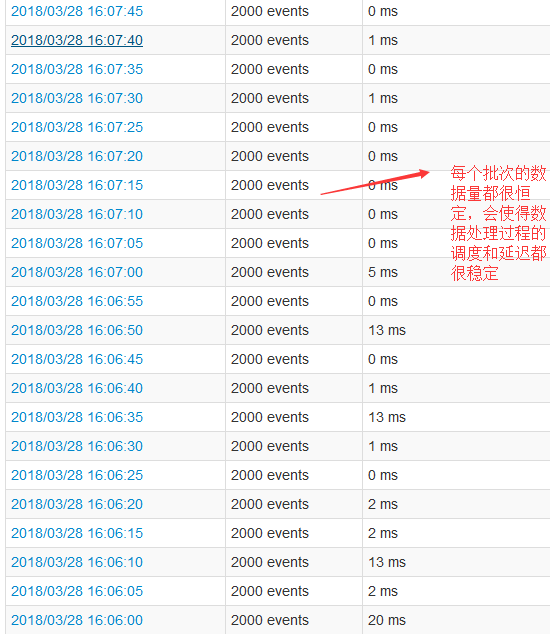
spark.streaming.kafka.maxRatePerPartition：给定streaming和kafka 集成模式为direct的情况下，每秒每个rdd分区最多处理的数据 量 (number of records per second)

val conf = new SparkConf()  
 .setMaster("local[2]") //如果要做SparkSteaming，那么这里必须是至少两个线程  
 .setAppName(getClass.getSimpleName)  
 .set("spark.streaming.backpressure.enabled","true") //开启backpressure机制  
 .set("spark.streaming.kafka.maxRatePerPartition","100")//限制每s从每个分区读取的最大条数









### 5.5、数据恢复机制

1）Use Receiver

默认情况下，Kafka的框架会**每隔60s**将consumer的**offset偏移量提交到zk中进行保存**，下一次重启consumer的时候，会自动从zk中加载这个偏移量信息。

**数据处理过程：首先将接收的数据保存Block块,然后Block块被业务处理。**

-a. 只要形成Block块，就可以认为这个数据已经被consumer消费了，然后当60s后，就会在zk中更新offset偏移量值 ==> 如果offset已经更新，那么表示这个offset对应的数据已经在executor中以block块的形式存储了 ==> 只能修改提交间隔时间来减少恢复时候的重复数据量。

val kafkaParams =Map[String,String](  
 //指定consumer（sparkstreaming）连接的kafaka集群  
 "zookeeper.connect" -> "bigdata.server1:2181",  
 //给定消费组的名称，这是必须给定  
 "group.id" -> "streaming1",  
 "auto.offset.reset"-> "largest",  
 "auto.commit.interval.ms"->"2000", //将默认60s的offset更新缩减到2s  
 "zookeeper.connection.timeout.ms" -> "10000"  
)

-b. block块的恢复

将参数spark.streaming.receiver.writeAheadLog.enable设置为true表示开启日志功能，可用于sparkstreaming应用的block块恢复操作

val conf = new SparkConf()  
 .setMaster("local[\*]") //如果要做SparkSteaming，那么这里必须是至少两个线程  
 .setAppName("StreamingWordCount")  
 .set("spark.streaming.receiver.writeAheadLog.enable","true") //开启预写式日志

2）Direct

-a.基于自定义的messageHandler函数,可以进行offset偏移量的管理

val messageHandler:MessageAndMetadata[String,String] => String = {  
 (msg:MessageAndMetadata[String,String]) =>{  
 //元数据信息  
 // msg.partition， //  
 //msg.topic,  
 //msg.offset,  
 //数据本身，包含了key和value  
 //msg.key(),  
 msg.message()  
 }  
}

-b.利用streaming的HA机制

DStream在构建RDD的过程中，会将RDD的相关元数据写入checkpoint文件夹中(hdfs上的一个文件夹)，当streaming应用宕机进行恢复后，支持从checkpoint文件夹中恢复对应的数据 ==> 可以达到offset的一个恢复策略。

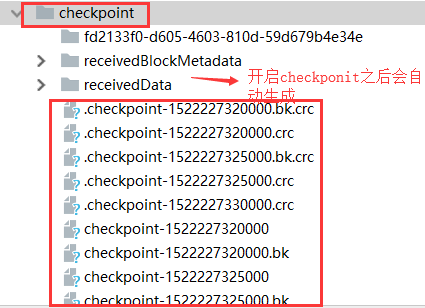
SparkStreaming HA机制：是SparkStreaming提供的一个元数据恢复机制，通过将计算过程中涉及到的所有元数据以及状态数据保存到checkpoint文件夹中(只会保存最近几个批次)，然后当streaming的程序宕机后，可以基于这些元数据进行恢复操作

要求：

1. 恢复的streaming程序中的DStream的最终Job执行DAG图(RDD DAG图)必须没有发生变化

2. checkpoint文件夹在第一次运行的时候要求不存在

ssc.checkpoint("./checkpoint") //表示再当前项目所在目录下进行checkpoint



<http://spark.apache.org/docs/1.6.1/streaming-programming-guide.html#checkpointing>

**6、DStream的相关API**

6.1、transform API

直接将DStream的操作转换为RDD的操作，通过该API，可以将DStream每个批次的执行操作转换为批次内RDD的转换操作，该API要求返回一个新的RDD。

注意：不管什么类型的DStream，一个批次就只有一个RDD。

// 一、上下文构建

val conf = new SparkConf()

.setMaster("local[10]")

.setAppName("StreamingWordCountOfTransformAndForeachRDDAPIDemo")

val sc = SparkContext.getOrCreate(conf)

val ssc = new StreamingContext(sc, Seconds(10))

// 二、DStream的构建

val dstream = ssc.socketTextStream("hadoop-senior01.ibeifeng.com", 9999)

// 三、DStream的数据处理

val resultDStream = dstream

.flatMap(line => line.split(" "))

.filter(word => word.nonEmpty)

.map(word => (word, 1))

.reduceByKey(\_ + \_)

val dstream2: DStream[(String, Int)] = dstream.transform(rdd => {

// rdd就是一个批次中对应的那个rdd， 这里对rdd的操作其实就是

// 对批次中的数据的一个操作

rdd

.flatMap(line => line.split(" "))

.filter(word => word.nonEmpty)

.map(word => (word, 1))

.reduceByKey(\_ + \_)

.sortBy(t => t.\_2, ascending = false)

})

dstream2.print()

/\*dstream.transform((rdd, time) => {

// rdd是批次中的数据，time是批次时间

})\*/

dstream2.foreachRDD(rdd => {

// 对RDD的数据进行输出操作

rdd.foreachPartition(iter => {

iter.foreach(println)

})

})

// 四、DStream结果输出

// 1. 输出到driver

resultDStream.print()

// 六、启动SparkStreaming开启数据处理

ssc.start()

ssc.awaitTermination() // 阻塞，等待程序的遇到中断等操作

// 七、关闭sparkstreaming的程序运行

ssc.stop()

6.2、foreachRDD API：DStream的数据输出

1）调用foreachRDD API将DStream的数据输出转换为RDD的数据输出(每个批次就有一个rdd的数据输出操作)。

foreachRDD API类似transform API，区别：foreachRDD是数据输出API，会触发job的执行，底层是rdd的action算子；foreachRDD没有返回值。

2）调用dstream的相关api进行数据输出

dstream.print dstream.savexxx

====> 这些API的底层还是foreachRDD

备注：既然transform和foreachRDD API可以将DStream的操作转换为RDD的操作，那么也就是可以将DStream -> RDD -> DataFrame，可以使用SQLContext对DStream中的数据进行处理操作。

6.3、updateStateByKey API ==> 累加求解(累加多个批次的结果)

1）updateStateByKey一般情况下和Streaming的HA机制一起使用，要求

必须给定checkpoint文件夹路径 ===> 需要保存之前批次执行的

结果值(状态值)。

应用场景：需要对数据进行累加统计的情况下就使用该API

2）updateStateByKey 解释:   
 以DStream中的数据进行按key做reduce操作，然后对各个批次的数据 进行累加。在有新的数据信息进入或更新时，updateStateByKey操作允 许您在使用新的信息持续更新时保持任意状态。 要使用这个，你将不得 不做两个步骤。

(1) 定义状态：能够是随意数据类型   
 (2) 定义状态更新函数：使用函数指定如何使用上一个状态更新状态，

并从输入流中指定新值。

/\*\*

\* def updateStateByKey[S: ClassTag](

\* updateFunc: (Seq[V], Option[S]) => Option[S]

\* ): DStream[(K, S)]

\* 功能：按照key进行分组后，对所有的value数据以及上一个批次对应key的执行结果数据进行一个更新状态值的操作(聚合操作)

\* updateFunc：给定的一个聚合函数/更新状态值函数

\* Seq[V]: 输入参数， 表示的是当前批次中，同一组中的所有value数据组成的一个序列集合

\* Option[S]: 输入参数，表示的是上一个批次/之前批次执行完成后的状态值，当为None，表示对应的key是第一次出现，Some表示有状态值

\* Option[S]: 返回值，表示的是经过当前批次的数据聚合后，对应key的新的状态值；如果返回None，表示这个key不保存状态值了，否则返回Some，表示保存新的状态值

\*/

val resultDStream: DStream[(String, Long)] = dstream

.flatMap(line => line.split(" "))

.filter(word => word.nonEmpty)

.map(word => (word, 1))

.reduceByKey(\_ + \_) // 最好保留

.updateStateByKey(

(seq: Seq[Int], state: Option[Long]) => {

// 1. 获取当前批次中的所有value的聚合值

val sum = seq.sum

// 2. 获取上一个批次的状态值

val preState = state.getOrElse(0L)

// 3. 更新状态值并返回

// TODO: 当返回None的时候表示清空对应key的状态值，

返回Some表示保存对应的状态值

Some(preState + sum)

}

)

6.4、window类型的API：

计算最近一段时间的统计值(计算最近几个批次的累加统计值)

eg：reduceByKeyAndWindow 功能就是将父DStream的多个批次进行合并 结果数据，然后一起进行统计操作

应用场景：计算最近一段时间的数据的一个统计指标值

xxxByWindow这个API在有一些的场景下需要必须给定checkpoint文件夹路径

/\*\*

\* def updateStateByKey[S: ClassTag](

\* updateFunc: (Seq[V], Option[S]) => Option[S]

\* ): DStream[(K, S)]

\* 功能：按照key进行分组后，对所有的value数据以及上一个批次对应key的执行结果数据进行一个更新状态值的操作(聚合操作)

\* updateFunc：给定的一个聚合函数/更新状态值函数

\* Seq[V]: 输入参数， 表示的是当前批次中，同一组中的所有value数据组成的一个序列集合

\* Option[S]: 输入参数，表示的是上一个批次/之前批次执行完成后的状态值，当为None，表示对应的key是第一次出现，Some表示有状态值

\* Option[S]: 返回值，表示的是经过当前批次的数据聚合后，对应key的新的状态值；如果返回None，表示这个key不保存状态值了，否则返回Some，表示保存新的状态值

\*/

val resultDStream: DStream[(String, Long)] = dstream

.flatMap(line => line.split(" "))

.filter(word => word.nonEmpty)

.map(word => (word, 1))

.reduceByKey(\_ + \_) // 最好保留

.updateStateByKey(

(seq: Seq[Int], state: Option[Long]) => {

// 1. 获取当前批次中的所有value的聚合值

val sum = seq.sum

// 2. 获取上一个批次的状态值

val preState = state.getOrElse(0L)

// 3. 更新状态值并返回

// TODO: 当返回None的时候表示清空对应key的状态值，返回Some表示保存对应的状态值

Some(preState + sum)

}

)