# Spark Streaming

在介绍Spark Streaming之前，首先需要了解一些数据方面的知识。数据总体上可以分为静态数据和流数据。对静态数据和流数据的处理，对应着两种截然不同的计算模式：批量计算和实时计算。批量计算：一个定义了一系列的彼此相关或者先后执行或者并行执行的的作业的处理流程，这些作业的输出最终会合并起来产生最终结果，对于批量计算来说，输入数据是在一个时间段内分批收集的，它也叫海量数据的不连续处理，以海量静态数据数据为对象，在很充裕的时间内进行处理，相应时间是否够快不是重点。实时计算：持续受到变化的数据（如空管相关数据、旅行预订系统等），并足够地处理这些信息，流数据必须采用实时计算，实时计算最重要的一个需求是能够实时得到计算结果，一般要求响应时间为毫秒级（甚至微秒级）。Spark Streaming是构建在Spark上的实时计算框架，它扩展了Spark处理大规模流式数据的能力。

## 基础知识

### Spark Streaming工作原理

Spark Streaming属于核心Spark API的扩展，支持实时数据流的可扩展、高吞吐、容错的流处理。可以接受来自Kafka、Flume、ZeroMQ、Kinesis、Twitter或者TCP套接数字的数据源，也可以用map、reduce、join、window等高级函数表示的复杂算法进行处理，处理的结果可以输出到文件系统、数据库、现场dashboards等，也可以直接使用内置的机器学习算法、图形处理算法处理数据。

基本的数据输入/输出的示意图如下所示，基本的输入源（HDFS\TCP套接字、Akka Actor等）和高级输入源（Kafka，Flume、ZeroMQ、Kinesis、Twitter等）均可以直接对Spark Streaming进行输入，经过处理的数据可以存储到文件系统（如HDFS）、数据库（如HBase）、其他输出（如Dashboards）等。

Spark Streaming在内部的处理机制是：先接收实时流的数据，并根据一定的时间间隔拆分成一批一批的数据，然后把数据传输给Spark Engine进行处理，最终得到处理后的一批一批的结果数据。通俗点讲，把流数据分成一批一批，然后将它们通过一个先进先出的队列，然后Spark核心作业从该队列中依次取出一个个批数据，把批数据封装成一个个RDD，然后进行处理，这是一个典型的生产者消费者模型，它的工作原理也在下图如下图所示：

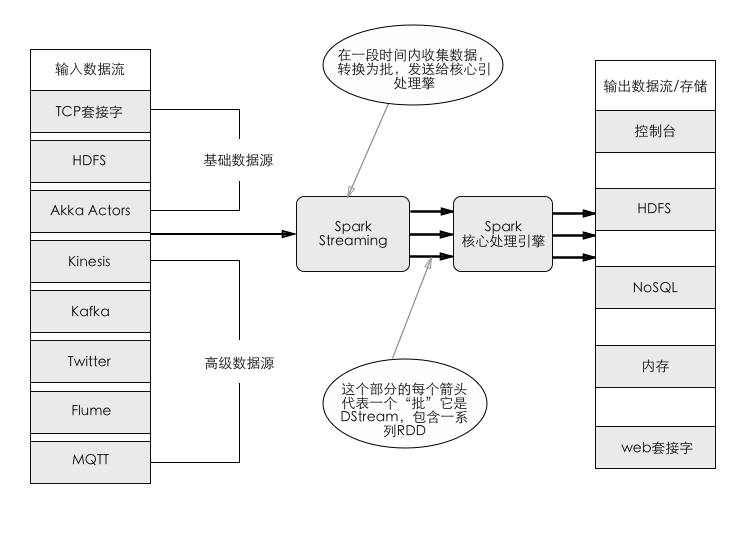


图 8-1 Spark数据输入/输出图

### **Spark Streaming整体架构**

前面对Spark Streaming应用的外部表象做了介绍，接下来介绍Spark Streaming的整体架构，在介绍这部分的时候首先要讲解一下DStream（Discretized Stream，离散流），它是Spark Streaming提供的高级抽象连续数据流。DStream

本质上是对RDD的一层封装，DStream中的每个RDD都包含来自一个时间间隔的数据，如下图所示：



图 8-2 DStream与RDD的关系

DStream是一个没有边界的集合，也就是它没有大小的限制，他代表的是一个时空的概念，上图中就能体现出多个时间段。具体要到那个时间段，就是空间的操作，也就是对时间间隔的对应批次的数据的处理。Spark Streaming应用程序中，除了使用数据源产生的数据流来创建DStream，也会在已有的DStream上使用某种操作创建新的DStream，下图显示了对DStream做flatMap操作的示意图：

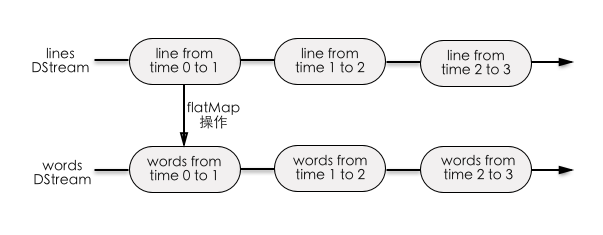


图 8-3 利用flatMap操作把行DStream转换为单词DStream的过程

当对DStream进行的各种操作让它们之间构建起依赖关系，当遇到DStream使用输出操作的时候，这些依赖关系以及它们之间的操作会被记录到一个名为DStream Graph的对象中表示一个作业。这些作业注册到DStream Graph并不会立即运行，而是等到Spark Streaming启动后，到达批处理时间，才会根据DStream Graph生成作业处理该批处理时间内接收到的数据。从程序到DStream Graph的转换图如下所示，代码首先用来生成DStream Graph前面的操作不会马上执行，而是从每个foreach()开始都会进行回溯。从后往前回溯这些操作之间的依赖关系。

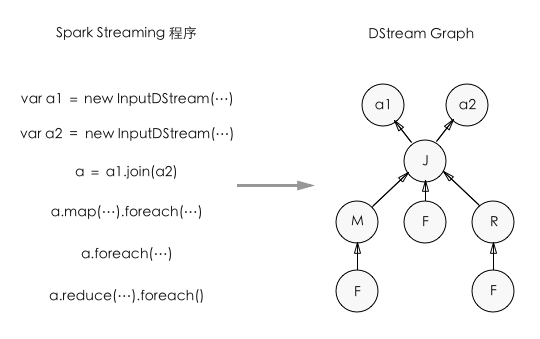


图 8-4 Spark Streaming代码与DStream Graph关系

下面来对Spark Streaming整体进行分析，经过了对于DStream的理解之后，知道了Spark Streaming将流式计算分解成一系列短小的批处理作业。这里批处理引擎是Spark Core，也就是把Spark Streaming的输入数据按照Batch Interval（批次间隔）分成一段一段的数据（DStream），每段数据都转换成Spark中的RDD，然后将Spark Streaming中对于DStream的转换操作变成针对Spark中对RDD的转换操作，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将结果存储到外部设备。

### **Spark Streaming的特点**

1. **流式处理**

Spark Streaming是将流计算分解成一系列短小的批处理作业。这里批处理引擎是Spark Core，也就是把Spark Streaming的输入数据按照批处理间隔（如0.5秒）分解成一段一段数据（DStream，离散流），每一段数据都转换成Spark中的RDD，然后将Spark Streaming中对DStream的Transformation操作变成针对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需求可以对中间的结果进行叠加或者存储到外部 设备。

1. **高容错**

对于流式计算来说，容错性至关重要，Spark Streaming的一大优势在于它提供的强大容错性保障。只要输入数据存储在可靠的系统中，Spark Streaming就可以根据输入计算出正确的结果，提供“精确一次”执行的语义（就好象所有的数据都是在没有任何节点失败的情况下处理的一样），即使工作节点或者驱动程序发了失败。Spark Streaming的容错机制如下：

1. 检查点机制

检查点机制是Spark Streaming中用来保障容错性的主要机制。它可以使Spark Streaming阶段性地把应用数据存储到如HDFS或者Amazon S3这样的可靠的存储系统中，以供恢复使用（一般来说，处理5-10个批次的数据保存一次）。具体来说检查点机制主要以下面的两个主要目的服务。

* 控制发生失败时需要重算状态数，Spark Streaming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。
* 提供驱动器容错，如果应用程序崩溃了，可以重启驱动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，并在那里继续执行。

1. 工作节点容错

为了对应工作节点失败的问题，Spark Streaming使用与Spark的容错机制相同的办法，所有从外部数据源中收到的数据都在多个节点上备份。所有从备份数据的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。

1. 接收器容错

当使用网络接收数据的数据源（如Kafka、Flume）时，为了获取相同容错性，接收到的数据会被复制到集群中的节点的内存中，根据失败的场景和接收器的类型 进行容错，主要有两种类型：可靠的接收器，这些接收器确保收到的数据备份并获得认可，如果一个接收器失败了，数据源不会发送数据，只有在接收器重启的时候，数据源才会重新进行数据发送，不会导致数据丢失；不可靠的接收器，当工作节点或者驱动器节点失败时，该接收器可能会导致数据丢失。

数据会不会丢失取决于用什么类型的接收器，当工作节点失败时，用可靠的接收器是不会产生数据丢失，如果是不可靠的接收器，接收数据没有备份，可能会有丢失，当驱动器节点失败，除了接收数据丢失之外，所有接收和备份在内存中的历史数据全部都丢失。为了避免数据丢失，在Spark1.2开始，接收数据进行容错存储，并提前写入日志（write ahead log），用来实现零数据丢失。如果“写入日志”启用了，所有接收到的数据会保存到一个日志文件中去（HDFS），这样保存接收数据的持久性，此外，只有在数据写入到日志中之后接收器才向数据源确认，这样drive重启后那些保存在内存中但是没有写入到日志中的数据将会重新发送，这两点保证的数据的无丢失。

1. **低延迟**

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的任务调度的过程。对于目前版本的Spark Streaming而言，最小的Batch Size（批大小）选取在0.5s-2s之间，所以，Spark Streaming可以满足除了对实时性要求非常高（高频实时交易）之外的所有流式准实时计算场景。

1. **吞吐量大**

Spark目前在EC2上已经能够线性扩展到100个节点，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高了2-5倍。

## DStream操作

### **DStream输入源**

在Spark Streaming中所有的操作都是基于流，而输入源是这一系列操作的起点，Spark Streaming提供基础和高级两种输入源。

1. 基础输入源

基础输入源是指能够直接应用于StreamingContext API的输入源。其中核心Spark Streaming API提供了文件，而套接字和Akka Actor创建DStream作为输入源的方法。

* 文件流：文件流用于从兼容于HDFS API的文件系统（如HDFS、S3、NFS）中

读取文件中的数据，DStream创建方法如下：

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

Spark Streaming将监控dataDirectory目录，并处理在该目录中创建的任何文件对于简单的文本文件而言，还有一个更简单的办法：

StreamingContext.textFileStream(dataDirectory)。文件流并不需要运行接收器，因此不需要分配核。所有的DStream的计算都是使用 streamingContext.start() 来处理数据，使用 streamingContext.awaitTermination() 等待处理被终止（手动或者由于任何错误），使用 streamingContext.stop() 来手动的停止处理。

* 套接字流：套接字流是通过监听Socket端口接收的数据，创建方法如下：

//在本地创建一个StreamingContext，分三个线程，批次间隔为2s

val conf = new SparkConf().setMaster(“local[3]”).setAppName(“WordCount”)

val ssc = new StreamingContext(conf, Seconds(2))

//创建一个DStream，连接hostname:port，类似于localhost:9999

val lines = ssc.socketTextStream(“localhost”, 9999)

* RDD队列流：使用streamingContext.queueStream(RDDqueue)创建基于RDD

队列的DStream，一般是用来调试Spark Streaming应用程序的，一下是它的创建方法的代码：

//基本配置

val conf = new SparkConf().setMaster(local[3]).setAppName(“yourqueue”)

//创建StreamingContext

val ssc = new StreamingContext(conf, Seconds(2))

//创建一个队列

val queue = new SynchronizedQueue[RDD[Int]]()

//创建RDD的队列流，并对其做操作

val inputStream = ssc.queueStream(rddQueue)

val mapStream = inputStream.map(y => (y%10, 1))

val reducebukeyStream = mapStream.reduceByKey(\_+\_)

reducedStream.print()

//开始ssc

ssc.start()

//创建并push这些RDDs

for(i <- 1 to 40) {

queue += ssc.sparkContext.makeRDD(1 to 1000, 10)

Thread.sleep(1000)

}

ssc.stop()

1. 高级输入源

这一类的来源需要外部的接口，其中一些有复杂的依赖关系（如Kafka核Flume），因此，通过这些源来创建DStream需要明确其依赖。例如，想使用Kafka数据源

创建的DStream的数据流，需要按照以下步骤来做：

1. 首先在SBT或者Maven工程中添加spark-streaming-kafka\_2.11依赖。

<dependency>

<groupId>org.apache.spark</groupId> <artifactId>spark-streaming-kafka-0-8\_2.11</artifactId> <version>2.2.0</version> </dependency>

1. 导入KafkaUtils类并创建基于Kafka的DStream。

Import org.apache.spark.streaming.kafka.\_

Val kafkaStream = KafkaUtils.createStream(streamingContext,[zookeeperQuorum],

[consumer group id],[per-topic number of Kafka partitions to consume])

需要注意的是，这些高级的来源一般在Spark Shell中不可用，因此基于这些高级来源的应用不能再Spark Shell中进行测试，若需要在Spark Shell中使用，需要下载相应的Maven工程Jar依赖，并添加到类路径中。

表 8-1 常见的输入源和对应的项目

|  |  |  |
| --- | --- | --- |
| 输入源名称 | 项目名称 |  |
| Kafka | Spark-streaming-kafka\_2.11 | 可从Kafka中接收数据流 |
| Flume | Spark-streaming-flume\_2.11 | 可从Flume中接收数据流 |
| Twitter | Spark-streaming-twitter\_2.11 | 可从TwitterUtils工具类调用Twitter4j，可得到公众的流 |
| MQTT | Spark-streaming-mqtt\_2.11 | 可从MQTT消息队列中接收流数据 |
| ZeroMQ | Spark-streaming-zeromq\_2.11 | 可从ZeroMQ消息队列中接收流数据 |

### **DStream转化操作**

DStream的转化操作可以分为无状态和有状态两种，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，这样的操作与之前讲到的RDD的转化操作类似，例如map()、filter()、reduceByKey()等，都属于无状态转化操作。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。

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中的每个RDD。部分无状态转化操作列在下表中，在图8-3中能看到filter()的例子。

表 8-2 无状态转化操作

|  |  |
| --- | --- |
| 转换操作 | 含义 |
| map(func) | 源DStream每个元素通过函数func返回一个新的DStream |
| flatMap(func) | 类似map，但是每一个输入元素可以被映射成0或多个输出元素 |
| filter(func) | 返回一个新的DStream，由经过func函数计算后返回值为true的输入元素组成 |
| repartition(numPartitions) | 通过输入的参数numPartitions的值改变DStream的分区大小 |
| union(otherStream) | 返回一个新的DStream，包含源DStream和其他DStream元素 |
| count() | 对源DStream内部的所有含有的RDD元素数量进行计数 |
| reduce(func) | 使用函数func（有两个输入参数，一个输出结果）将源DStream中每个RDD的元素进行聚合，返回一个内部所包含的RDD只有一个元素的新的DStream |
| countByValue() | 计算DStream中每个RDD内的元素出现的频次并返回新的DStream[(K,Long)]，其中K是RDD中元素的类型，Long是元素出现的频次 |
| reduceByKey(func,[numTasks]) | 当调用一个（K，V）键值对类型的DStream时，返回一个新的（K，V）键值对类型的DStream，这里键（K）不发生变化，而新的值（V）则是由reduce函数（func）聚合后得到的，这里的numTasks是可选参数，用来设置任务数量 |
| join(otherStream,[numTasks]) | 当调用（K，V）键值对类型和（K，W）键值对类型的两个DStream时，返回一个（K，（V，W））键值对类型的DStream |
| cogroup(otherStream,[numTasks]) | 当调用一个（K，V）键值对类型和（K，W）键值对类型的两个DStream时，返回一个（K，Seq[V]，Seq[W]）元组的新DStream |
| transform(func) | 通过对源DStream的每个RDD应用RDD-to-RDD函数返回一个新的DStream，这里可以用来在DStream做任意RDD操作 |

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。

1. 有状态转化操作

DStream的有状态转化操作是跨时间区间跟踪数据的操作，也就是说，一些先前批次的数据也被用来在新的批次中计算结果。主要的两种类型是滑动窗口和updateStateByKey()，前者以一个时间阶段为滑动窗口进行操作，后者则用来跟踪每个键的状态变化（例如构建一个代表用户会话的对象）。

* 基于窗口的操作

Spark Streaming提供了基于Window的计算，允许通过滑动窗口对数据进行转换。下图表示窗口的滑动计算过程当窗口在DStream中按照定义的时间间隔滑动时，落入窗口的RDD被视为一个个窗口化的DStream。因此，任何窗口操作都需要制定两个重要参数，这两个参数分别为窗口长度（窗口的持续时间），滑动步长（执行基于窗口操作计算的时间间隔），特别注意的时，这两个参数必须是源DStream批处理时间间隔的倍数。

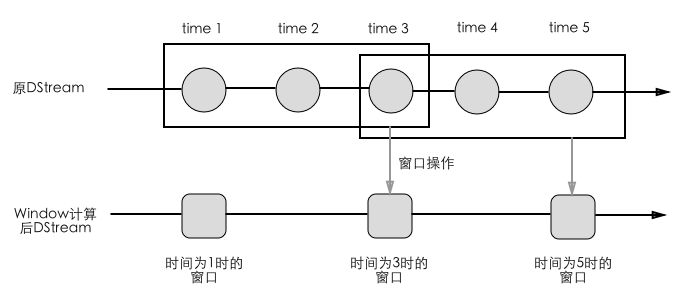


图 8-5 滑动窗口计算图

对DStream可以用的最简单的窗口操作是window()，它返回一个新的DStream来表示所请求的窗口操作的结果数据，换句话说，window()生成的DStream中每个RDD会包含多个批次的数据，可对这些数据进行count()、transform()等操作，对滑动窗口的计算，假设批处理时间间隔为10秒，要创建一个最近30秒的时间间隔的窗口（最近3个批次），设置每两个批次计算一次窗口结果，应该把滑动步长设置为20秒，代码如下：

//统计Stream中RDD的元素个数

val inputStreamWindow = inputStream.window(Seconds(30), Seconds(20))

val windowCount = inputStreamWindow.count()

Spark Streaming还提供了其他基于窗口的操作，如下表所示：

表 8-3 窗口转化操作

|  |  |
| --- | --- |
| 转换 | 含义 |
| window(windowLength,slideInterval) | 返回一个基于源DStream的窗口批次计算后得到新的DStream |
| countByWindow(windowLength,slideInterval) | 返回基于滑动窗口的DStream中元素的数量 |
| reduceByWindow(func,windowLength,  slideInterval) | 返回基于窗口对源DStream中元素进行聚合操作，得到新的DStream |
| reduceByKeyAndWindow(func,windowLength,  slideInterval,[numTasks]) | 基于滑动窗口对（K，V）键值对类型的DStream中的值按照K使用聚合函数func进行聚合操作，得到新的DStream |
| reduceByKeyAndWindow(func,invFunc,  windowLength,slideInterval,[numTasks]) | 一个更高效的reduceByKeyAndWindow()的实现版本，每个窗口的reduce值是通过先前窗口的reduce值增量计算得到的，通过减少进入滑动窗口的新数据，并对离开窗口的老数据进行“逆减少”操作实现，可以使用键的“加”和“减”数目作为滑动窗口的数量，但只能用于“可逆的reduce函数”，即那些reduce函数都有一个对应的“逆reduce函数” |
| countByValueAndWindow(windowLength,  slideInterval,[numTasks]) | 基于滑动窗口计算源DStream中每个RDD内每个元素出现的频次并返回DStream[(K,Long)]，其中K是RDD中元素的类型，Long是元素频次，与countByValue一样，reduce任务的数量可以通过一个可选参数配置 |

这里简单介绍一下窗口操作的例子演示，利用套接字流实现，套接字流的具体流程参考8.4章节。

1. Window操作

//导入所需的包

import org.apache.spark.SparkConf

import org.apache.spark.storage.StorageLevel

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

object Spark\_Streaming {

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

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

//设置监听间隔为10s

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

//用套接字流指定ip端口

val lines = ssc.socketTextStream("yourIP",9999)

val words = lines.flatMap(\_.split(" "))

指定窗口大小为30s，滑动为10s

val windowwords = words.window(Seconds(30),Seconds(10))

windowwords.print()

ssc.start()

ssc.awaitTermination()

}

}

该代码可以将每组输入数按照空格分隔输出，由于窗口的大小为30s，滑动为10s正好是窗口大小的三倍，所以它会把最近30s内的输入数据处理输出，所以每次的数据会被监听三次，结果图如下：



图 8-6 端口输入图

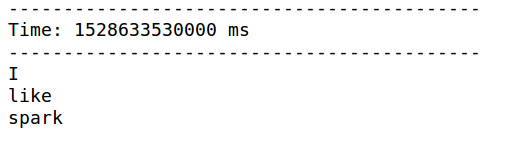


图 8-7 监听结果图1

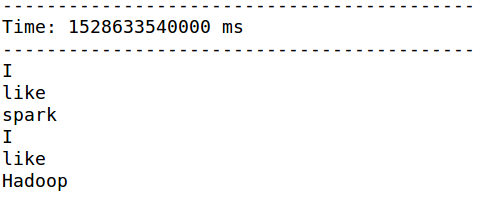


图 8-8 监听结果图2

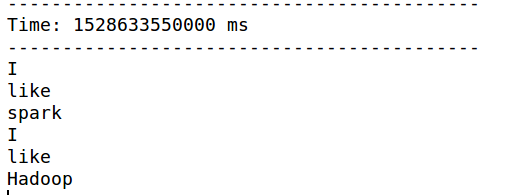


图 8-9 监听结果图3

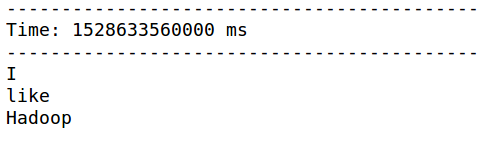


图 8-10 监听结果图4

1. countByWindow操作

//导入所需的包

import org.apache.spark.SparkConf

import org.apache.spark.storage.StorageLevel

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

object Spark\_Streaming2 {

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

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

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

//设置检查点

ssc.checkpoint("/home/xiongfan123/checkpoint")

val lines = ssc.socketTextStream("192.168.28.133", 9999)

val words = lines.flatMap(\_.split(" "))

//根据窗口大小统计DStream元素个数

val windowwords = words.countByWindow(Seconds(30), Seconds(10))

windowwords.print()

ssc.start()

ssc.awaitTermination()

}

}

该代码的意思是每组数据按空格拆分，由于窗口的大小为30s，滑动为10s正好是窗口大小的三倍，所以它会把最近30s内的输入数据组成一个大的DStream，并统计其内部元素个数，进行输出，结果图如下：

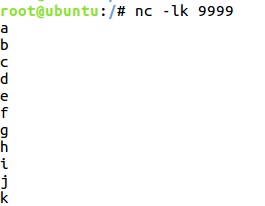


图 8-11 端口输入图

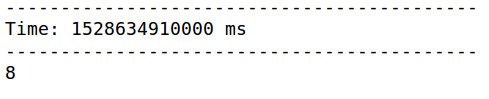


图 8-12 监听结果图1

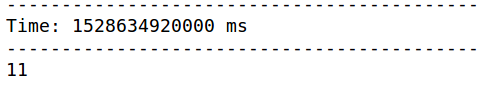


图 8-13 监听结果图2

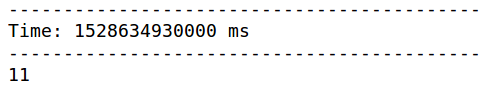


图 8-14 监听结果图3

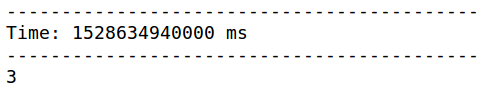


图 8-15 监听结果图4

* updateStateByKey转化操作

当我们需要在跨批次之间维护状态时，就必须使用updateStateByKey操作。该操作为我们提供了一个状态变量的访问，用于键值对的形式的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的状态函数，它可以构建出一个新的DStream，其内部数据为（键，状态）对。举一个单词词频统计的例子来说明上述情况，达能处理每个批次的DStream时，如果时无状态的转换操作，单词处理时只对本批次内的单词进行词频统计，不会考虑之前的批次的单词，不同批次的单词词频是独立的，而对于updateStateByKey转化操作，本批次的词频统计会在之前的批次的词频统计基础上进行不断累加，最终得到的词频是所有批次单词的总的词频统计结果。

操作示例代码如下：

import org.apache.spark.SparkConf

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

object Streaming\_four {

def main(args: Array[String]) {

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

.setAppName("UpdateStateByKeyDemo")

val ssc = new StreamingContext(conf,Seconds(20))

//要使用updateStateByKey方法，必须设置Checkpoint

ssc.checkpoint("/home/xiongfan123/checkpoint")

val socketLines = ssc.socketTextStream("192.168.28.133",9999)

//设置updateStateByKey的参数，有当前值和以往值，当前最新值加上以往值就是与之前值相加的总值

socketLines.flatMap(\_.split(",")).map(word=>(word,1))

.updateStateByKey(

(currValues:Seq[Int],preValue:Option[Int]) =>{

val currValue = currValues.sum

Some(currValue + preValue.getOrElse(0))

}).print()

ssc.start()

ssc.awaitTermination()

ssc.stop()

}

}

该代码的意思是进行wordcount的监听，然后它会记录所有的DStream的count的值而不仅仅是这一个DStream的count的值。

监听的结果图如下：

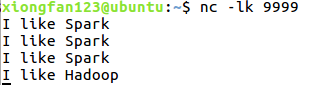


图 8-16 端口输入图

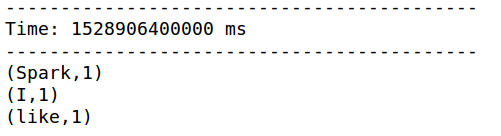


图 8-17 监听结果图1

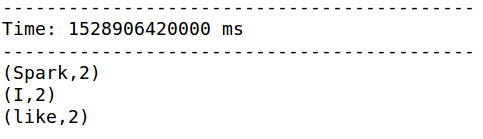


图 8-18 监听结果图2

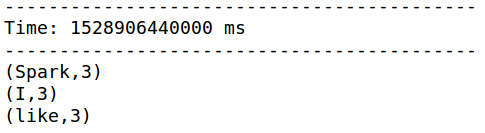


图 8-19 监听结果图3

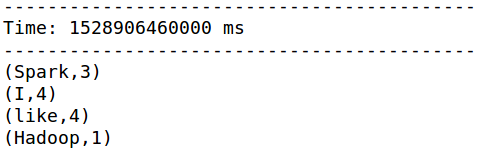


图 8-20 监听结果图4

### **DStream输出操作**

输出操作是将DStream的数据输出到外部的系统，比如说数据库或者文件系统。实际上，输出操作作用于DStream后，外部系统才能使用这些数据，触发所有的DStream变换实际执行，这一点于RDD执行（Action）类似。下表列出了目前主要的输出操作。

表 8-4 输出操作

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

举例saveAsTextFiles(prefix,[suffix])

import org.apache.spark.SparkConf

import org.apache.spark.storage.StorageLevel

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

object Spark\_Streaming3 {

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

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

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

val lines = ssc.socketTextStream("192.168.28.133", 9999)

//保存到路径下，会自动生成test+“监听时间”+.txt文件

lines.saveAsTextFiles("/home/xiongfan123/sparktest/test","txt")

ssc.start()

ssc.awaitTermination()

}

}

该代码会每隔十秒钟生成一个文件夹到指定路径下，文件夹名称为test+“监听时间”+.txt的文件，在端口输入I like Spark后结果图如下：

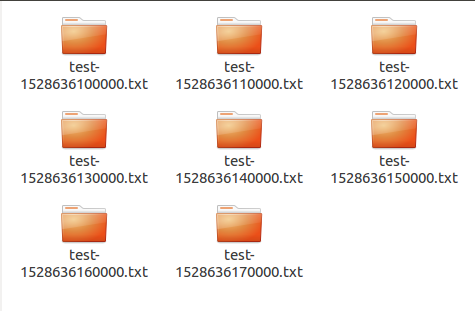


图 8-21 文件夹监听结果

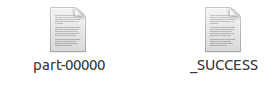


图 8-22 文件夹监听到的文件



图 8-23 文件的内容

## 性能调优

如果想要使Spark Streaming应用程序在集群中获得最佳的性能实践，则需要对一些性能参数进行调优。这里主要考虑下面两个方面：

* 有效使用集群资源，减少批处理所消耗的时间。
* 设置合理的窗口的大小，使数据尽可能快速得到处理。

### **减少批处理时间**

优化运行时间可以降低每个批次数据的处理时间，主要包括：提升数据接收和处理的并行度，减少序列化和反序列化负担，优化内存的使用，减少任务提交与分享的开销。

1. 提高数据的接收并行度

如果是通过网络接收数据的话（比如Kafka、Flume、套接字等），则需要把数据反序列化并存储在Spark上，若数据接收成为瓶颈，则需要并行接收数据。采用的办法主要是提升Receiver的并发度和调整Receiver的RDD数据分区时间隔。

提升Receiver的并发度：在Worker节点上对每个输入的DStream创建一个Receiver并运行，以接收一个数据流。通过创建多个DStream并配置从数据接收源接收不同的分区的数据流。例如，一个单Kafka输入DStream接收两个主题的数据可以分成两个Kafka输入流，每个仅仅接收一个主题。输入DStream运行在两个Worker节点接收器上，从而能够并行接受，提高整体吞吐量。多个DStream也可以通过联合在一起创建一个DStream，这样一些应用在一个输入DStream的转换操作便可以用在联合后的DStream上。

调整Receiver的RDD数据分区时的间隔：这由configuration parameter （配置参数） 的 spark.streaming.blockInterval 决定。对于大多数的Receiver，接收到的数据要合并成大的数据块，然后存储在Spark的内存中。对于大多数的Receiver，接收到的数据要合并成大的数据块，然后存储在Spark的内存中。每个批次的数量决定任务的数量，这些任务用来处理那些接收到的数据，即进行类“Map”操作，每个Receiver每批次任务数目大约为（批时间间隔/块时间间隔），例如在100毫秒的块时间间隔将会在2秒的批次中创建20个任务。若任务太少，会导致有的核闲置，没有用来处理数据，会使效率降低很多。针对一个给定时间间隔的情况，如果要提升任务数，则需要降低每一小块的时间间隔。推荐的块时间间隔最小为50毫秒。

1. 提升数据处理并行度

若在任务执行阶段使用并行的任务数量不多，会造成集群资源利用低下的问题。例如分布式Reduce操作，如reduceByKey和reduceByKeyAndWindow，并行任务数量决定是在spark.default.parallelism中配置的。要确保均衡使用整个集群的资源，而不是把任务都集中在几个特定的节点上，对包含Shuffle的操作，增加并行度以确保更为充分使用集群资源。

1. 数据序列化

数据序列化的开销很大，特别是要实现亚秒级批次的大小，数据序列化主要包括下面两个方面。RDD数据的序列化，默认情况下RDD被保存为序列化子节数组，来减少GC停顿；输入数据序列化，将数据的外部数据插入Spark，接收到的数据为子节型，需要反序列化为Spark的序列化格式。因此，输入数据的反序列化开销会成为一个瓶颈。Spark Streaming默认将接收到的数据序列化存储，以减少内存的使用。序列化和反序列化需要更多的CPU时间，更加高效的序列化方式（Kryo）和自定义的序列化接口可以更高效地使用CPU。

### **设置正确的批次间隔**

设置合适的批处理大小，首先要了解几个关键词。

* 批处理时间：每个批次的数据的处理时间。
* 批次间隔时间：两个批次处理的时间间隔。
* 数据速率：数据在集群上的处理速率。

为了保证Spark Streaming应用程序能在集群上稳定运行，系统应该能够尽可能快地处理接收到的数据。换言之，处理数据的速度要跟上数据输入速度。处理数据速度对应批次处理时间，批次间隔时间设置数据流入速度，批次间隔时间应该大于批处理时间。根据Streaming流计算性质来看，批次间隔时间可用一组固定集群资源对应用程序持续影响，批次间隔时间设置要充分考虑预期数据速率是否稳定。下面说说如何设置一个合理的批次大小。先设置Batch（批次大小）为5-10秒和一个很低的数据输入速度。确定系统能跟上数据输入的速度，可以根据经验调整批次大小，通过查看日志获知Total delay（总延迟）。如果Delay稍微小于批处理时间，那么系统稳定，如果Delay一直在增加，说明系统处理速度跟不上数据输入速度。

### **优化内存使用**

针对Spark应用程序的内存使用和GC（垃圾回收）行为，这一部分讲了一些自定义Spark Streaming应用的调优参数，来优化内存使用。

1. 设置合理的DStream存储的级别

与RDD不同，RDD默认持久化级别是MEMORY\_ONLY，而DStream默认持久化级别是MEMORY\_ONLY\_SER，尽管保持数据序列化会带来更高序列化、反序列化开销，但是大大减少GC出现停顿的现象。

1. 及时清理持久化的RDD

Streaming会将接收到的数据全部存储于可以用的内存中，因此对已经完成处理的数据应该及时清理，以确保Streaming有足够的内存，默认情况下，所有Streaming持久化RDD的清理会使用内置的内存清理策略LRU（Least Recently Used）；通过设置spark.cleaner.ttl的值，Streaming就能自动地定期清除旧的内容。通过设置spark.streaming.unpersist属性启用内存清理，减少Spark RDD内存的使用，提升GC性能。

1. 并发垃圾收集

GC会影响任务的正常运行，任务执行时间的延长，会引起一系列不可预料的问问题，采用不同的GC策略可以进一步减小GC对Job运行的影响。例如，使用并行mark-and-sweep GC能够减少GC的突然暂停情况，另外也可以降低系统吞吐量为代价来获得最短GC停顿（大部分GC执行需要将应用程序停止以便内存保持一个一致的状态来进行回收）。

## 代码实例

### **文件流实例**

首先新建一个文件夹用来接收数据文件，取名为file

root@ubuntu:~# cd Streaming/

root@ubuntu:~/Streaming# mkdir file

root@ubuntu:~/Streaming# cd file/

root@ubuntu:~/Streaming/file#

接着在IDEA中新建一个scala的object，命名为Streaming\_one，例子是wordcount

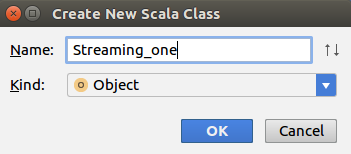


图 8-24 文件命名

在文件中输入代码：

import org.apache.spark.SparkConf

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

object Streaming\_one {

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

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

val ssc = new StreamingContext(conf, Seconds(30))//设置监听时间间隔为30s

val lines = ssc.textFileStream("/home/xiongfan123/Streaming/file")//设置监听的文件夹

//监听的文件执行wordcount

val words = lines.flatMap(\_.split(" "))

val wordCounts = words.map(x=>(x,1)).reduceByKey(\_+\_)

wordCounts.print()

ssc.start()

ssc.awaitTermination()

}

}

在没有输入文件时的监听状态

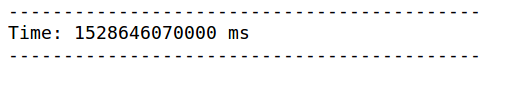


图 8-25 无输入监听状态

在有数据输入时的监听状态，即在路径下的文件夹中新建文本文件，并在其中写入

I love bigdata

I like my friend

I like sport

执行监听的结果如下：

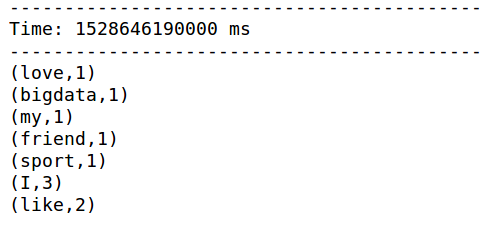


图 8-26 有输入监听状态

发现文件已经执行了wordcount。

### **RDD队列流**

在IDEA中新建一个scala的object，命名为Streaming\_two，例子是wordcount

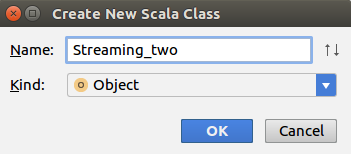


图 8-27 文件命名

在文件中输入代码：

import org.apache.spark.SparkConf

import org.apache.spark.rdd.RDD

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

object Streaming\_two {

def main(args:Array[String]){

val sparkConf = new SparkConf().setAppName("RDDQueue").setMaster("local[2]")

val ssc = new StreamingContext(sparkConf,Seconds(4)) //设置每四秒监听一次

val rddQueue = new scala.collection.mutable.SynchronizedQueue[RDD[Int]]()//创建RDD队列

val queueStream = ssc.queueStream(rddQueue) //创建输入的队列数据流

val mapStream = queueStream.map(r=>(r%10,1)) //处理队列中的RDD数据

val reduceStream = mapStream.reduceByKey(\_+\_)

reduceStream.print()

ssc.start()

//创建和向队列推入RDD

for(i<-1 to 10){

rddQueue+=ssc.sparkContext.makeRDD(1 to 100,2)

Thread.sleep(1000)

}

ssc.stop()

}

}

监听到的结果如图所示：

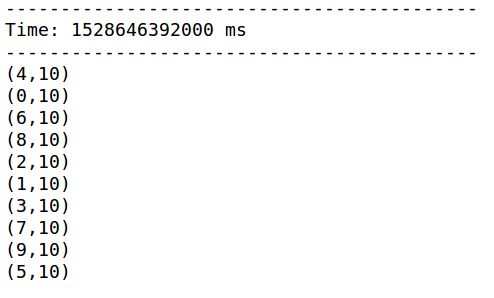


图 8-28 监听状态

### **套接字流实例**

在IDEA中新建一个scala的object，命名为Streaming\_three，例子是wordcount

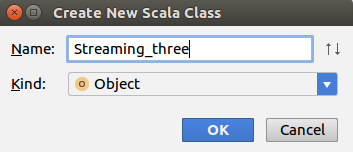


图 8-29 文件命名

在文件中输入代码：

import org.apache.spark.SparkConf

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

object Streaming\_three {

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

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

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

val lines = ssc.socketTextStream("192.168.28.133", 9999)//第一个参数为主机IP，第二个为端口

val words = lines.flatMap(\_.split(" "))

val wordCounts = words.map(x => (x, 1)).reduceByKey(\_ + \_)

wordCounts.print()

ssc.start()

ssc.awaitTermination()

}

}

接着在终端窗口中输入：

nc -lk 9999

进入9999端口，然后可以在这个端口输入下面：

I love bigdata

I like my friend

I like sport

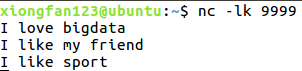


图 8-30 端口输入

监听到的结果如下图所示：

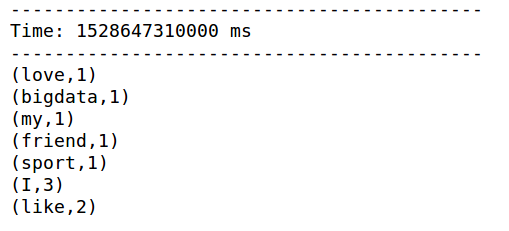


图 8-31 监听状态

### **Kafka消息队列流**

请在使用kafka的时候先保证kafka与sbt的正确安装，安装步骤在本章后面。这一小节的例子是把kafka作为生产者，并且每秒发送三条消息，每个消息包含5个0~9的随机数，Spark Streaming作为消费者，使用reduceByKeyAndWindow()方法进行统计并显示结果。

这里需要下载spark-streaming-kafka-0-8\_2.11.jar包，下载方法在命令行输入：

wget <http://centram.maven.org/maven2/org/apache/spark/spark-streaming-kafka_2.11/1.6.3/sp>

ark-streaming-kafka\_2.11-1.6.3.jar

把这里下载的jar与kafka中libs目录下的jar文件全都复制到spark目录下的jars目录下，按照如下代码类似方法：

cp spark-streaming-kafka-0-8\_2.11-2.1.0.jar /home/xiongfan123/spark-2.3.0-bin-hadoop2.7/jars

前面目录为要复制的文件，后面为复制到的目录，复制kafka的libs目录下文件代码如下：

cd /home/xiongfan123/kafka\_2.11-1.0.1/libs

cp ./\* /home/xiongfan123/spark-2.3.0-bin-hadoop2.7/jars

复制完毕后新建一个kafka的工程目录，目录建成如下：

/kafka/src/main/scala

其中在scala目录下新建一个scala文件，KafkaWordProducer程序，用它来产生数据，输入命令如下：

cd kafka/src/main/scala

vim KafkaWordProducer.scala

在其中输入如下代码：

package org.apache.spark.examples.streaming

import java.util.HashMap

import org.apache.kafka.clients.producer.{KafkaProducer,ProducerConfig,ProducerRecord}

import org.apache.spark.SparkConf

import org.apache.spark.streaming.\_

import org.apache.spark.streaming.kafka.\_

//生成1-100的随机数

object KafkaWordProducer{

def main(args:Array[String]){

//验证参数，少于4个则退出

if(args.length<4){

System.err.println("Usage:KafkaWordCountProducer <metadataBrokerList>

<topic>"+"<messagesPerSec> <wordsPerMessage>")

System.exit(1)

}

val Array(brokers,topic,messagesPerSec,wordsPerMessage)=args

//zookeeper的连接属性，通过第一个传入的参数获取brokers地址信息

val props = new HashMap[String,Object]()

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG,brokers)

props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG,"org.apache.kafka.common.serialization.StringSerializer")

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG,"org.apache.kafka.common.serialization.StringSerializer")

//创建kafka实例，通过参数确定每秒发送的信息条数，每个消息包含0~9随机数

val producer = new KafkaProducer[String,String](props)

while(true){

(1 to messagesPerSec.toInt).foreach{messageNum => val str = (1 to wordsPer

Message.toInt).map(x=>scala.util.Random.nextInt(10).toString).mkString(" ")

print(str)

println()

val message = new ProducerRecord[String,String](topic,null,str)

producer.send(message)

}

Thread.sleep(1000)

}

}

}

在这个目录下创建KafkaWordCount.scala文件，它可以把KafkaWordProducer发送的单词进行词频统计，代码如下：

package org.apache.spark.examples.streaming

import org.apache.spark.\_

import org.apache.spark.SparkConf

import org.apache.spark.streaming.\_

import org.apache.spark.streaming.kafka.\_

import org.apache.spark.streaming.StreamingContext.\_

import org.apache.spark.streaming.kafka.KafkaUtils

object KafkaWordCount{

def main(args:Array[String]){

val sc = new SparkConf().setAppName("KafkaWordCount").setMaster("local[2]")

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

ssc.checkpoint("/home/xiongfan123/kafka/checkpoint") //设置检查点

val zkQuorum = "localhost:2181" //设置zookeeper服务器地址

val group = "1"

val topics = "wordsender" //topic的名称

val numThreads = 1 //每个topic分区数

//通过zookeeper连接属性获取kafka的组和主题信息，并创建连接获取数据流

val topicMap = topics.split(",").map((\_,numThreads.toInt)).toMap

val lineMap = KafkaUtils.createStream(ssc,zkQuorum,group,topicMap)

val lines = lineMap.map(\_.\_2)

//获取的数据流进行分词

val words = lines.flatMap(\_.split(" "))

val pair = words.map(x=>(x,1))

//窗口大小为2分钟，每10s输出一次结果

val wordCounts = pair.reduceByKeyAndWindow(\_+\_,\_-\_,Minutes(2),Seconds(10),2)

wordCounts.print()

ssc.start()

ssc.awaitTermination()

}

}

接着再把之前的Spark目录下的examples/src/main/scala/org/apache/spark/examples/streaming目录下的StreamingExamples.scala文件拷贝到该目录。然后对该工程进行打包，进入该工程文件夹，新建一个simple.sbt文件，输入下列代码：

name:="My four Project"

version:="1.0"

scalaVersion:="2.11.8"

libraryDependencies += "org.apache.spark"%%"spark-core"%"2.3.0"

libraryDependencies += "org.apache.spark"%"spark-streaming\_2.11"%"2.3.0"

libraryDependencies += "org.apache.spark"%"spark-streaming-kafka-0-8\_2.11"%"2.3.0"

保存成功后，就可以打包，进入该工程文件：

cd kafka

sbt

package

打包成功后，需要先开启zookeeper、kafka两个服务，开启步骤如下：

首先开启zookeeper服务，先进入kafka目录，输入代码：

/home/xiongfan123/kafka\_2.11-1.0.1/bin/zookeeper-server-start.sh /home/xiongfan123/kafka\_2.11-1.0.1/config/zookeeper.properties

（两个目录分别为zookeeper-server-start.sh、zookeeper.properties安装目录，中间有空格）

成功之后显示如下：

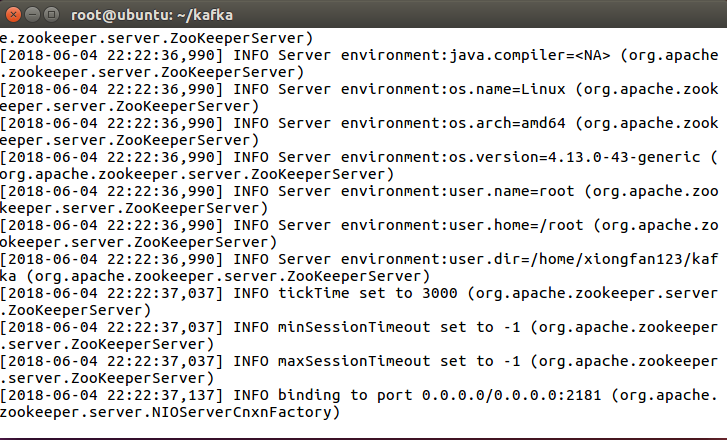


图 8-32 进入zookeeper服务成功

接着开启kafka服务，另起一终端，进入kafka目录，输入代码：

/home/xiongfan123/kafka\_2.11-1.0.1/bin/kafka-server-start.sh /home/xiongfan123/kafka\_2.11-1.0.1/config/server.properties

成功之后显示如下：

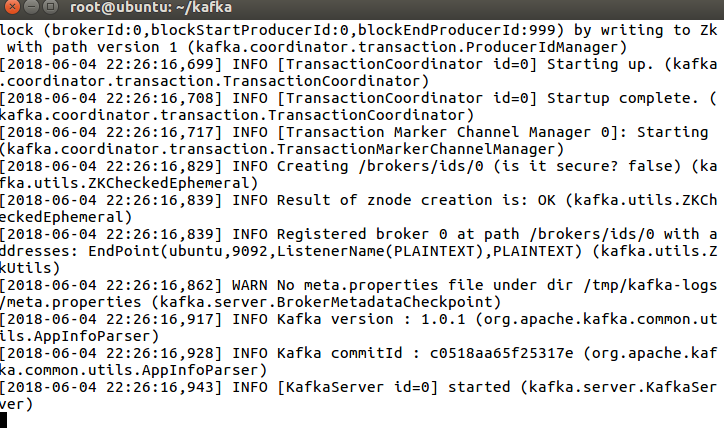


图 8-33 进入kafka服务成功

最后，另起终端执行产生数据命令，生成一堆整数：

cd kafka

/home/xiongfan123/spark-2.3.0-bin-hadoop2.7/bin/spark-submit --class "org.apache.spark.examples.streaming.KafkaWordProducer" /home/xiongfan123/kafka/target/scala-2.11/my-four-project\_2.11-1.0.jar localhost:9092 wordsender 3 5（这里master表示生产者的地址与端口，wordsender表示kafka主题，3表示每秒发送3条消息，5表示每条消息中包含5个单词）

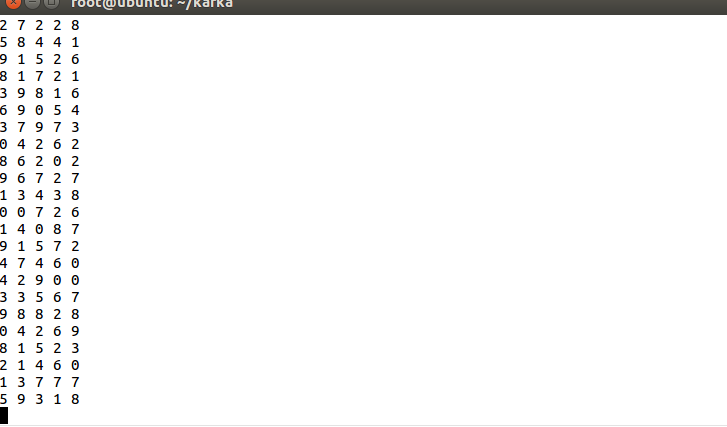


图 8-34 数据产生成功

上面的窗口作为数据来源的kafka消息队列，然后执行新终端，接收数据源数据，进行词频统计：

cd kafka

/home/xiongfan123/spark-2.3.0-bin-hadoop2.7/bin/spark-submit --class "org.apache.spark.examples.streaming.KafkaWordCount" /home/xiongfan123/kafka/target/scala-2.11/my-four-project\_2.11-1.0.jar

可以看到监听信息出现的情况：

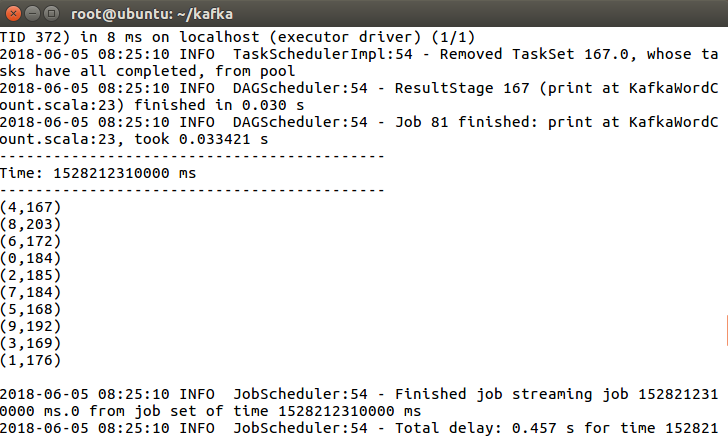


图 8-35 数据统计执行监听

### **sbt的安装（附录）**

首先需要在官网上下载sbt安装包，在linux命令行中输入如下命令：

wget <https://sbt-downloads.cdnedge.bluemix.net/releases/v1.1.6/sbt-1.1.6.tgz>

下载sbt安装包，下载完毕后，对其进行解压，在命令行输入

tar zxvf sbt-1.1.6.tgz

解压完毕之后，在sbt安装的目录下，新建一个名为sbt的文本文件，并在文件下输入：

BT\_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256M"

java $SBT\_OPTS -jar /home/xiongfan123/sbt/bin/sbt-launch.jar "$@"

其中的目录为sbt-launch.jar的存放目录，一般放在sbt下的bin目录下。

接着修改sbt的权限，先用命令行进入刚建立的sbt文本文件同级目录下，然后执行命令：

chmod u+x sbt

下一步配置环境变量，在命令行输入命令：

vim ~/.bashrc

点击insert，然后在文件末尾添加代码：

#SBT\_HOME

export SBT\_HOME=/home/xiongfan123/sbt

export PATH=${SBT\_HOME}/bin:$PATH

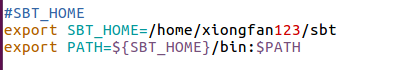


图 8-36 环境变量配置图

最后点击esc，然后依次输入“:wq”进行保存和退出，并执行命令：

source ~/.bashrc

使环境变量生效。

要测试安装成不成功，可以在命令行键入（注：这里需要进入管理员模式，sudo -s即可）：

sbt

第一次的时候这一步要下很多东西，所以要等很久，这里要特别提醒，不是你错了，是真的慢。成功之后会出现如下图所示：

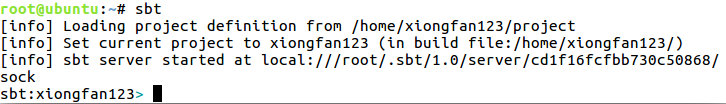


图 8-37 sbt安装执行成功

在这里输入sbtVersion可以查看sbt版本：



图 8-38 查看sbt版本

### **Kafka安装（附录）**

首先在官网上下载kafka的安装包，在linux命令行中输入如下命令：

wget <http://mirrors.shu.edu.cn/apache/kafka/1.1.0/kafka_2.11-1.1.0.tgz>

下载kafka安装包完毕后，对其进行解压，输入如下命令：

tar zxvf sbt-1.1.6.tgz

解压完毕后，设置环境变量，在命令行输入：

vim ~/.bashrc

然后点击insert，并在文件最后输入下面代码：

#KAFKA\_HOME

export KAFKA\_HOME=/home/kafka\_2.11-1.0.1

export PATH=$PATH:$KAFKA\_HOME/bin

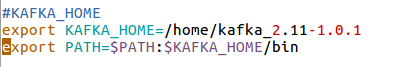


图 8-39 kafka环境变量

为了验证kafka的安装情况，可以在命令行中输入下面代码：

/home/xiongfan123/kafka\_2.11-1.0.1/bin/zookeeper-server-start.sh /home/xiongfan123/kafka\_2.11-1.0.1/config/zookeeper.properties

（注：这里前后两个目录分别为你的kafka安装目录下的zookeeper-server-start.sh文件和zookeeper.properties文件所在目录，而且它们中间有空格）

启动成功会出现如下图片所示界面：

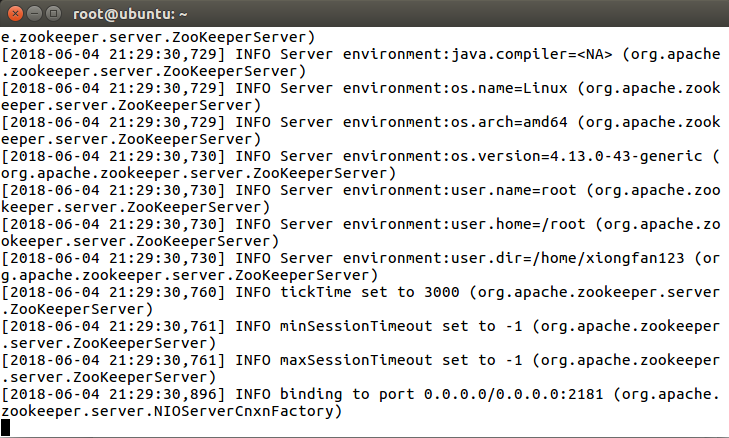


图 8-40 执行zookeeper服务成功

## 本章小结

本章节主要学习了如何使用DStream操作数据流。由于DStream是由RDD组成的，所以之前的章节学习到的技术和知识仍然适用于流式计算与实时应用。

DStream与Spark其他扩展一样使用相同的执行引擎，因此能与批处理和交互式查询无缝集成。