1.概述

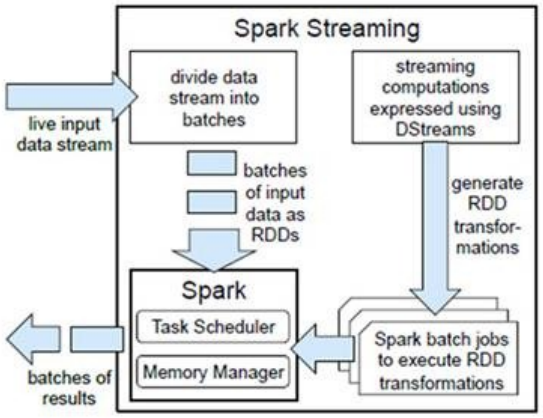
Spark Streaming 是Spark核心API的一个扩展，可以实现高吞吐量的、具备容错机制的实时流数据的处理。支持从多种数据源获取数据，包括Kafk、Flume、Twitter、ZeroMQ、Kinesis 以及TCP sockets，从数据源获取数据之后，可以使用诸如map、reduce、join和window等高级函数进行复杂算法的处理。最后还可以将处理结果存储到文件系统，数据库和现场仪表盘。

来自〈http://www.cnblogs.com/shishanyuan/p/4747735.html〉

2. Streaming架构

2.1 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。下图显示了Spark Streaming的整个流程。

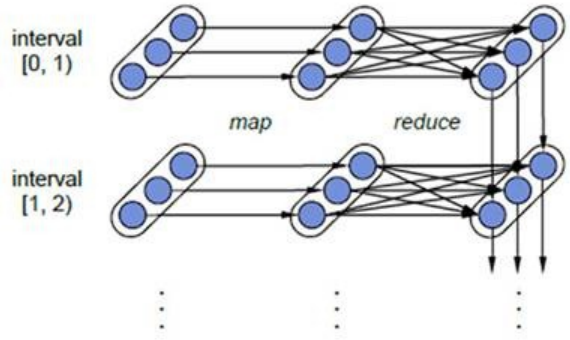


图Spark Streaming构架

2.2 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个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。



Spark Streaming中RDD的lineage关系图

2.3 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）

2.4 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。

来自〈http://www.cnblogs.com/shishanyuan/p/4747735.html〉

3.使用

3.1 基本使用(接收套接字)

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

// Create a local StreamingContext with two working thread and batch interval of 1 second.
// The master requires 2 cores to prevent from a starvation scenario.
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(5))
// 后面时间的含义:
//1. 每隔5秒切一次RDD(可能包含多个)
//2. 每隔5秒提交一个job,或者说每隔5秒执行一次
//3. 这个job需5秒内算完,不然下次就要来了,造成处理积压


// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)

// Split each line into words
val words = lines.flatMap(_.split(" "))
import org.apache.spark.streaming.StreamingContext._
// 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
```

> 使用:
> 1.在集群中执行命令: `nc -lk 9999`
> 2.然后输入内容,用空格隔开,
> 3.然后在IDE控制台就能看见单词计数结果
>
> 解释:
> **1.创建StreamingContext对象**
> 同Spark初始化需要创建SparkContext对象一样，使用Spark Streaming就需要创建StreamingContext对象。创建StreamingContext对象所需的参数与SparkContext基本一致，包括指明Master，设定名称(如NetworkWordCount)。需要注意的是参数Seconds(1)，Spark Streaming需要指定处理数据的时间间隔，如上例所示的1s，那么Spark Streaming会以1s为时间窗口进行数据处理。此参数需要根据用户的需求和集群的处理能力进行适当的设置；
> **2.创建InputDStream**
> 如同Storm的Spout，Spark Streaming需要指明数据源。如上例所示的socketTextStream，Spark Streaming以socket连接作为数据源读取数据。当然Spark Streaming支持多种不同的数据源，包括Kafka、 Flume、HDFS/S3、Kinesis和Twitter等数据源；
> **3.操作Dstream**
> 对于从数据源得到的DStream，用户可以在其基础上进行各种操作，如上例所示的操作就是一个典型的WordCount执行流程：对于当前时间窗口内从数据源得到的数据首先进行分割，然后利用Map和ReduceByKey方法进行计算，当然最后还有使用print()方法输出结果；
> **4.启动Spark Streaming**
> 之前所作的所有步骤只是创建了执行流程，程序没有真正连接上数据源，也没有对数据进行任何操作，只是设定好了所有的执行计划，当ssc.start()启动后程序才真正进行所有预期的操作。

> 来自 <http://www.cnblogs.com/shishanyuan/p/4747735.html>
> http://spark.apache.org/docs/2.2.1/streaming-programming-guide.html

3.2 HDFS接收

streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)
eg. streamingContext.fileStream("hdfs://192.168.2.101:8020/data/")

Spark Streaming将监视目录dataDirectory并处理在该目录中创建的任何文件（不支持在嵌套目录中写入的文件）。注意

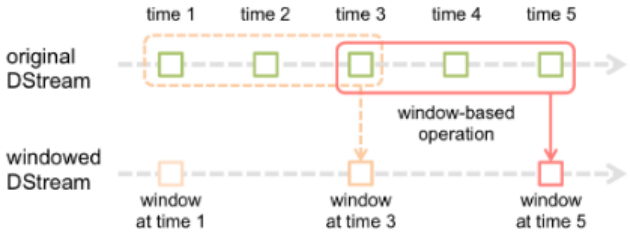
- 文件必须具有相同的数据格式。
- 必须dataDirectory通过原子地*将*文件*移动*或*重命名*到数据目录中来创建文件。
- 移动后，不得更改文件。因此，如果文件被连续追加，则不会读取新数据。

对于简单的文本文件，有一个更简单的方法streamingContext.textFileStream(dataDirectory)。文件流不需要运行接收器，因此不需要分配内核。

- **基于自定义接收器的流：可以使用通过自定义接收器接收的数据流创建DStream。有关更多详细信息，请参见《定制接收器指南》。**
- **RDD队列作为流：为了使用测试数据测试Spark Streaming应用程序，还可以使用，基于RDD队列创建DStream** streamingContext.queueStream(queueOfRDDs)**。推送到队列中的每个RDD将被视为DStream中的一批数据，并像流一样进行处理。**

3.3 windows窗口函数

（实现一阶段内的累加 ， 而不是程序启动时）



假设每隔**5s 1个batch,上图中窗口长度为15s，窗口滑动间隔10s。**

窗口长度和滑动间隔必须是**batchInterval的整数倍。如果不是整数倍会检测报错。**

优化后的**window操作要保存状态所以要设置checkpoint路径，没有优化的window操作可以不设置checkpoint路径。**

```
val sc = new SparkConf().setMaster("local[3]").setAppName("WindowsFunc")
val ssc = new StreamingContext(sc,Seconds(2)) // 每隔5秒接收一次(以前的也保留)
ssc.checkpoint("./checkPoint")
val line = ssc.socketTextStream("192.168.2.101",9999)// Receiver机制会占用一个线程
val wc = line.flatMap(_.split(" ")).map((_,1))
wc.reduceByKeyAndWindow(_+_,Seconds(8),Seconds(2)).print()// 每隔四秒算前八秒的数据
```

reduceByKeyAndWindow 函数
reduceByKeyAndWindow(func, invFunc,windowLength, slideInterval, [numTasks])

前面的func作用和上一个reduceByKeyAndWindow相同，后面的invFunc是用于处理流出rdd的。 因为窗口滑动的原因，就会造成数据重复计算(如上图的time3),使用该函数可以减少计算(会计算出重复量.再计算出增量,再合并)

在下面这个例子中，如果把3秒的时间窗口当成一个池塘，池塘每一秒都会有鱼游进或者游出，那么第一个函数表示每由进来一条鱼，就在该类鱼的数量上累加。而第二个函数是，每由出去一条鱼，就将该鱼的总数减去一。(简单这么理解,当数据出窗时,就会执行这个反向函数)

```
wc.reduceByKeyAndWindow(_+_,_-_,Seconds(8),Seconds(2)).print()// 每隔四秒算前八秒的数据(其结果与上方一直)
```

> 以后用到了再看吧
> https://www.cnblogs.com/yjd_hycf_space/p/7053722.html
> http://www.freesion.com/article/487428411/
> https://www.iteye.com/blog/humingminghz-2308138
> https://blog.csdn.net/legotime/article/details/51836040

3.4 连接kafka

```
package xkj.sparkStream

import com.alibaba.fastjson.JSON
import entity.MyPerson
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.log4j.Level
import org.apache.log4j.Logger._
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.{SaveMode, SparkSession}
import org.apache.spark.streaming.kafka010.{ConsumerStrategies, KafkaUtils, LocationStrategies}
import org.apache.spark.streaming.{Seconds, StreamingContext}


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

    getLogger("org.apache.hadoop").setLevel(Level.ERROR)
    getLogger("org.apache.zookeeper").setLevel(Level.WARN)
    getLogger("org.apache.hive").setLevel(Level.WARN)
    getLogger("org.apache").setLevel(Level.WARN)

    val spark = SparkSession.builder().master("local[3]").appName("KafkaTest").getOrCreate()
    import spark.implicits._
    val ssc = new StreamingContext(spark.sparkContext, Seconds(4))
    ssc.checkpoint("./checkPoint")
    val kafkaParams = Map[String, Object](
      "bootstrap.servers" -> "192.168.2.101:9092",
      "key.deserializer" -> classOf[StringDeserializer],
      "value.deserializer" -> classOf[StringDeserializer],
      "group.id" -> "xkj"
    )
    val topics: Set[String] = Set("mystream")
//    val stream = KafkaUtils.createDirectStream(ssc, LocationStrategies.PreferConsistent,
ConsumerStrategies.Subscribe[String, String](topics, kafkaParams))
//    stream.foreachRDD(rdd => {
//      rdd.foreach(line => {
//        println("key=" + line.key() + ",  value=" + line.value() + ", offset=" + line.offset())
//      })
//    })

    val streamPerson =
KafkaUtils.createDirectStream(ssc,LocationStrategies.PreferConsistent,ConsumerStrategies.Subscribe[String,String]
(topics,kafkaParams))
    streamPerson.foreachRDD(rdd=>{
      val v: RDD[String] = rdd.map(_.value())
      val rddP: RDD[MyPerson] = v.map(JSON.parseObject(_, classOf[MyPerson]))

      val df = rddP.toDF()
      df.show()
      df.write.mode(SaveMode.Append).json("hdfs://192.168.2.101:9000/user")
    })

    ssc.start()
```

```
    ssc.awaitTermination()

  }
}
```

特别注意版本：

pom.xml

```
<properties>
    <spark.version>2.3.0</spark.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-streaming_2.11</artifactId>
        <version>${spark.version}</version>
    </dependency>

    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-streaming-kafka-0-10_2.11</artifactId>
        <version>${spark.version}</version>
    </dependency>
</dependencies>
```

> https://blog.csdn.net/zhaolq1024/article/details/85685189
> https://blog.51cto.com/simplelife/2311296