想·

尚硅谷大数据技术之SparkStreaming

版本：V3.0

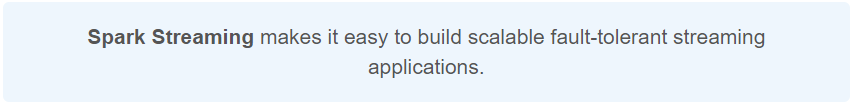


作者：尚硅谷大数据研发部

# SparkStreaming概述

## Spark Streaming是什么





Spark流使得构建可扩展的容错流应用程序变得更加容易。

Spark Streaming用于流式数据的处理。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。



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”)。所以简单来将，DStream就是对RDD在实时数据处理场景的一种封装。

## Spark Streaming的特点

* **易用**



* **容错**



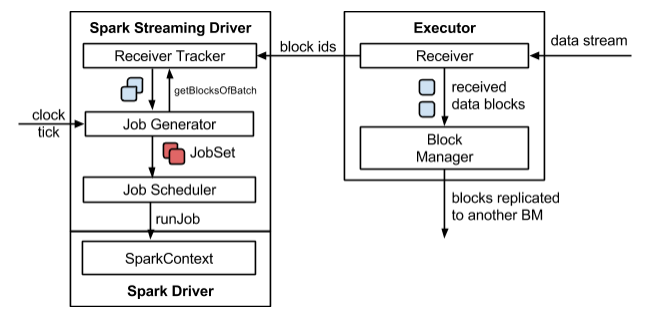
* **易整合到Spark体系**



## Spark Streaming架构

### 架构图

* **整体架构图**



* **SparkStreaming架构图**



### 1.3.2 背压机制

Spark 1.5以前版本，用户如果要限制Receiver的数据接收速率，可以通过设置静态配制参数“spark.streaming.receiver.maxRate”的值来实现，此举虽然可以通过限制接收速率，来适配当前的处理能力，防止内存溢出，但也会引入其它问题。比如：producer数据生产高于maxRate，当前集群处理能力也高于maxRate，这就会造成资源利用率下降等问题。

为了更好的协调数据接收速率与资源处理能力，1.5版本开始Spark Streaming可以动态控制数据接收速率来适配集群数据处理能力。背压机制（即Spark Streaming Backpressure）: 根据JobScheduler反馈作业的执行信息来动态调整Receiver数据接收率。

通过属性“spark.streaming.backpressure.enabled”来控制是否启用backpressure机制，默认值false，即不启用。

# 第2章 Dstream入门

## 2.1 WordCount案例实操

* 需求：使用netcat工具向9999端口不断的发送数据，通过SparkStreaming读取端口数据并统计不同单词出现的次数

1. 添加依赖

<dependency>

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

<artifactId>spark-streaming\_2.12</artifactId>

<version>2.4.5</version>

</dependency>

1. 编写代码

object StreamWordCount {

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

//1.初始化Spark配置信息

val sparkConf = new SparkConf().setMaster("local[\*]").setAppName("StreamWordCount")

//2.初始化SparkStreamingContext

val ssc = new StreamingContext(sparkConf, Seconds(3))

//3.通过监控端口创建DStream，读进来的数据为一行行

val lineStreams = ssc.socketTextStream("linux1", 9999)

//将每一行数据做切分，形成一个个单词

val wordStreams = lineStreams.flatMap(\_.split(" "))

//将单词映射成元组（word,1）

val wordAndOneStreams = wordStreams.map((\_, 1))

//将相同的单词次数做统计

val wordAndCountStreams = wordAndOneStreams.reduceByKey(\_+\_)

//打印

wordAndCountStreams.print()

//启动SparkStreamingContext

ssc.start()

ssc.awaitTermination()

}

}

1. 启动程序并通过netcat发送数据：

nc -lk 9999

hello spark

## 2.2 WordCount解析

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。



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



# 第3章 DStream创建

## 3.1 RDD队列

### 3.1.1 用法及说明

测试过程中，可以通过使用ssc.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。

### 3.1.2 案例实操

* 需求：循环创建几个RDD，将RDD放入队列。通过SparkStream创建Dstream，计算WordCount

1. 编写代码

object RDDStream {

def main(args: Array[String]) {

//1.初始化Spark配置信息

val conf = new SparkConf().setMaster("local[\*]").setAppName("RDDStream")

//2.初始化SparkStreamingContext

val ssc = new StreamingContext(conf, Seconds(4))

//3.创建RDD队列

val rddQueue = new mutable.Queue[RDD[Int]]()

//4.创建QueueInputDStream

val inputStream = ssc.queueStream(rddQueue,oneAtATime = false)

//5.处理队列中的RDD数据

val mappedStream = inputStream.map((\_,1))

val reducedStream = mappedStream.reduceByKey(\_ + \_)

//6.打印结果

reducedStream.print()

//7.启动任务

ssc.start()

//8.循环创建并向RDD队列中放入RDD

for (i <- 1 to 5) {

rddQueue += ssc.sparkContext.makeRDD(1 to 300, 10)

Thread.sleep(2000)

}

ssc.awaitTermination()

}

}

1. 结果展示

-------------------------------------------

Time: 1539075280000 ms

-------------------------------------------

(4,60)

(0,60)

(6,60)

(8,60)

(2,60)

(1,60)

(3,60)

(7,60)

(9,60)

(5,60)

-------------------------------------------

Time: 1539075284000 ms

-------------------------------------------

(4,60)

(0,60)

(6,60)

(8,60)

(2,60)

(1,60)

(3,60)

(7,60)

(9,60)

(5,60)

-------------------------------------------

Time: 1539075288000 ms

-------------------------------------------

(4,30)

(0,30)

(6,30)

(8,30)

(2,30)

(1,30)

(3,30)

(7,30)

(9,30)

(5,30)

-------------------------------------------

Time: 1539075292000 ms

-------------------------------------------

## 3.2 自定义数据源

### 3.2.1 用法及说明

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

### 3.2.2 案例实操

需求：自定义数据源，实现监控某个端口号，获取该端口号内容。

1. 自定义数据源

class CustomerReceiver(host: String, port: Int) extends Receiver[String](StorageLevel.MEMORY\_ONLY) {

//最初启动的时候，调用该方法，作用为：读数据并将数据发送给Spark

override def onStart(): Unit = {

new Thread("Socket Receiver") {

override def run() {

receive()

}

}.start()

}

//读数据并将数据发送给Spark

def receive(): Unit = {

//创建一个Socket

var socket: Socket = new Socket(host, port)

//定义一个变量，用来接收端口传过来的数据

var input: String = null

//创建一个BufferedReader用于读取端口传来的数据

val reader = new BufferedReader(new InputStreamReader(socket.getInputStream, StandardCharsets.UTF\_8))

//读取数据

input = reader.readLine()

//当receiver没有关闭并且输入数据不为空，则循环发送数据给Spark

while (!isStopped() && input != null) {

store(input)

input = reader.readLine()

}

//跳出循环则关闭资源

reader.close()

socket.close()

//重启任务

restart("restart")

}

override def onStop(): Unit = {}

}

1. 使用自定义的数据源采集数据

object FileStream {

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

//1.初始化Spark配置信息

val sparkConf = new SparkConf().setMaster("local[\*]")

.setAppName("StreamWordCount")

//2.初始化SparkStreamingContext

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

//3.创建自定义receiver的Streaming

val lineStream = ssc.receiverStream(new CustomerReceiver("hadoop102", 9999))

//4.将每一行数据做切分，形成一个个单词

val wordStream = lineStream.flatMap(\_.split("\t"))

//5.将单词映射成元组（word,1）

val wordAndOneStream = wordStream.map((\_, 1))

//6.将相同的单词次数做统计

val wordAndCountStream = wordAndOneStream.reduceByKey(\_ + \_)

//7.打印

wordAndCountStream.print()

//8.启动SparkStreamingContext

ssc.start()

ssc.awaitTermination()

}

}

## 3.3 Kafka数据源（面试、开发重点）

### 3.3.1 版本选型

ReceiverAPI：需要一个专门的Executor去接收数据，然后发送给其他的Executor做计算。存在的问题，接收数据的Executor和计算的Executor速度会有所不同，特别在接收数据的Executor速度大于计算的Executor速度，会导致计算数据的节点内存溢出。早期版本中提供此方式，当前版本不适用

DirectAPI：是由计算的Executor来主动消费Kafka的数据，速度由自身控制。

### 3.3.2 Kafka 0-8 Receiver模式（当前版本不适用）

1. 需求：通过SparkStreaming从Kafka读取数据，并将读取过来的数据做简单计算，最终打印到控制台。

2）导入依赖

<dependency>

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

<artifactId>spark-streaming-kafka-0-8\_2.11</artifactId>

<version>2.4.5</version>

</dependency>

3）编写代码

package com.atguigu.kafka

import org.apache.spark.SparkConf

import org.apache.spark.streaming.dstream.ReceiverInputDStream

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

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

object ReceiverAPI {

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

//1.创建SparkConf

val sparkConf: SparkConf = new SparkConf().setAppName("ReceiverWordCount").setMaster("local[\*]")

//2.创建StreamingContext

val ssc = new StreamingContext(sparkConf, Seconds(3))

//3.读取Kafka数据创建DStream(基于Receive方式)

val kafkaDStream: ReceiverInputDStream[(String, String)] = KafkaUtils.createStream(ssc,

"linux1:2181,linux2:2181,linux3:2181",

"atguigu",

Map[String, Int]("atguigu" -> 1))

//4.计算WordCount

kafkaDStream.map { case (\_, value) =>

(value, 1)

}.reduceByKey(\_ + \_)

.print()

//5.开启任务

ssc.start()

ssc.awaitTermination()

}

}

### 3.3.3 Kafka 0-8 Direct模式（当前版本不适用）

1）需求：通过SparkStreaming从Kafka读取数据，并将读取过来的数据做简单计算，最终打印到控制台。

2）导入依赖

<dependency>

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

<artifactId>spark-streaming-kafka-0-8\_2.11</artifactId>

<version>2.4.5</version>

</dependency>

3）编写代码（自动维护offset）

import kafka.serializer.StringDecoder

import org.apache.kafka.clients.consumer.ConsumerConfig

import org.apache.spark.SparkConf

import org.apache.spark.streaming.dstream.InputDStream

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

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

object DirectAPIAuto02 {

val getSSC1: () => StreamingContext = () => {

val sparkConf: SparkConf = new SparkConf().setAppName("ReceiverWordCount").setMaster("local[\*]")

val ssc = new StreamingContext(sparkConf, Seconds(3))

ssc

}

def getSSC: StreamingContext = {

//1.创建SparkConf

val sparkConf: SparkConf = new SparkConf().setAppName("ReceiverWordCount").setMaster("local[\*]")

//2.创建StreamingContext

val ssc = new StreamingContext(sparkConf, Seconds(3))

//设置CK

ssc.checkpoint("./ck2")

//3.定义Kafka参数

val kafkaPara: Map[String, String] = Map[String, String](

ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG -> "linux1:9092,linux2:9092,linux3:9092",

ConsumerConfig.GROUP\_ID\_CONFIG -> "atguigu"

)

//4.读取Kafka数据

val kafkaDStream: InputDStream[(String, String)] = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](ssc,

kafkaPara,

Set("atguigu"))

//5.计算WordCount

kafkaDStream.map(\_.\_2)

.flatMap(\_.split(" "))

.map((\_, 1))

.reduceByKey(\_ + \_)

.print()

//6.返回数据

ssc

}

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

//获取SSC

val ssc: StreamingContext = StreamingContext.getActiveOrCreate("./ck2", () => getSSC)

//开启任务

ssc.start()

ssc.awaitTermination()

}

}

4）编写代码（手动维护offset）

import kafka.common.TopicAndPartition

import kafka.message.MessageAndMetadata

import kafka.serializer.StringDecoder

import org.apache.kafka.clients.consumer.ConsumerConfig

import org.apache.spark.SparkConf

import org.apache.spark.streaming.dstream.{DStream, InputDStream}

import org.apache.spark.streaming.kafka.{HasOffsetRanges, KafkaUtils, OffsetRange}

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

object DirectAPIHandler {

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

//1.创建SparkConf

val sparkConf: SparkConf = new SparkConf().setAppName("ReceiverWordCount").setMaster("local[\*]")

//2.创建StreamingContext

val ssc = new StreamingContext(sparkConf, Seconds(3))

//3.Kafka参数

val kafkaPara: Map[String, String] = Map[String, String](

ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG -> "hadoop102:9092,hadoop103:9092,hadoop104:9092",

ConsumerConfig.GROUP\_ID\_CONFIG -> "atguigu"

)

//4.获取上一次启动最后保留的Offset=>getOffset(MySQL)

val fromOffsets: Map[TopicAndPartition, Long] = Map[TopicAndPartition, Long](TopicAndPartition("atguigu", 0) -> 20)

//5.读取Kafka数据创建DStream

val kafkaDStream: InputDStream[String] = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder, String](ssc,

kafkaPara,

fromOffsets,

(m: MessageAndMetadata[String, String]) => m.message())

//6.创建一个数组用于存放当前消费数据的offset信息

var offsetRanges = Array.empty[OffsetRange]

//7.获取当前消费数据的offset信息

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

offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges

rdd

}.flatMap(\_.split(" "))

.map((\_, 1))

.reduceByKey(\_ + \_)

//8.打印Offset信息

wordToCountDStream.foreachRDD(rdd => {

for (o <- offsetRanges) {

println(s"${o.topic}:${o.partition}:${o.fromOffset}:${o.untilOffset}")

}

rdd.foreach(println)

})

//9.开启任务

ssc.start()

ssc.awaitTermination()

}

}

### 3.3.4 Kafka 0-10 Direct模式

1）需求：通过SparkStreaming从Kafka读取数据，并将读取过来的数据做简单计算，最终打印到控制台。

2）导入依赖

<dependency>

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

<artifactId>spark-streaming-kafka-0-10\_2.12</artifactId>

<version>2.4.5</version>

</dependency>

3）编写代码

import org.apache.kafka.clients.consumer.{ConsumerConfig, ConsumerRecord}

import org.apache.spark.SparkConf

import org.apache.spark.streaming.dstream.{DStream, InputDStream}

import org.apache.spark.streaming.kafka010.{ConsumerStrategies, KafkaUtils, LocationStrategies}

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

object DirectAPI {

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

//1.创建SparkConf

val sparkConf: SparkConf = new SparkConf().setAppName("ReceiverWordCount").setMaster("local[\*]")

//2.创建StreamingContext

val ssc = new StreamingContext(sparkConf, Seconds(3))

//3.定义Kafka参数

val kafkaPara: Map[String, Object] = Map[String, Object](

ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG -> "linux1:9092,linux2:9092,linux3:9092",

ConsumerConfig.GROUP\_ID\_CONFIG -> "atguigu",

"key.deserializer" -> "org.apache.kafka.common.serialization.StringDeserializer",

"value.deserializer" -> "org.apache.kafka.common.serialization.StringDeserializer"

)

//4.读取Kafka数据创建DStream

val kafkaDStream: InputDStream[ConsumerRecord[String, String]] = KafkaUtils.createDirectStream[String, String](ssc,

LocationStrategies.PreferConsistent,

ConsumerStrategies.Subscribe[String, String](Set("atguigu"), kafkaPara))

//5.将每条消息的KV取出

val valueDStream: DStream[String] = kafkaDStream.map(record => record.value())

//6.计算WordCount

valueDStream.flatMap(\_.split(" "))

.map((\_, 1))

.reduceByKey(\_ + \_)

.print()

//7.开启任务

ssc.start()

ssc.awaitTermination()

}

}

### 查看Kafka消费进度

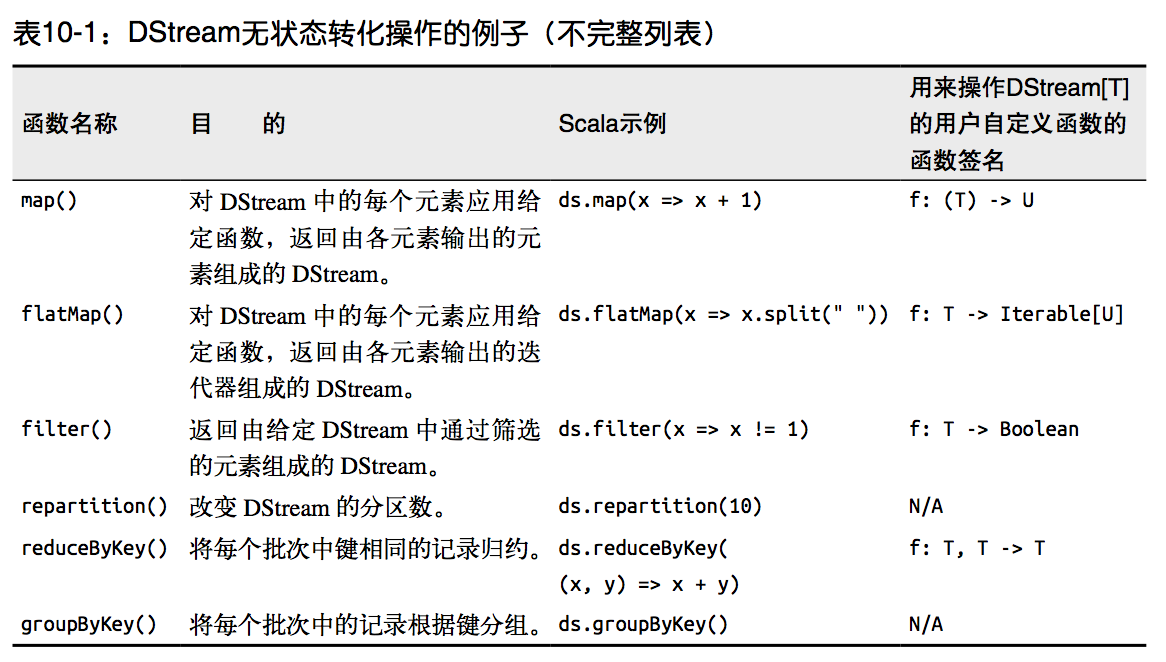
bin/kafka-consumer-groups.sh --describe --bootstrap-server linux1:9092 --group atguigu

# 第4章 DStream转换

DStream上的操作与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。

## 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。



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。

### 4.1.1 Transform

Transform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。其实也就是对DStream中的RDD应用转换。

object Transform {

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

//创建SparkConf

val sparkConf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("WordCount")

//创建StreamingContext

val ssc = new StreamingContext(sparkConf, Seconds(3))

//创建DStream

val lineDStream: ReceiverInputDStream[String] = ssc.socketTextStream("linux1", 9999)

//转换为RDD操作

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

val words: RDD[String] = rdd.flatMap(\_.split(" "))

val wordAndOne: RDD[(String, Int)] = words.map((\_, 1))

val value: RDD[(String, Int)] = wordAndOne.reduceByKey(\_ + \_)

value

})

//打印

wordAndCountDStream.print

//启动

ssc.start()

ssc.awaitTermination()

}

}

### 4.1.2 join

两个流之间的join需要两个流的批次大小一致，这样才能做到同时触发计算。计算过程就是对当前批次的两个流中各自的RDD进行join，与两个RDD的join效果相同。

import org.apache.spark.SparkConf

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

import org.apache.spark.streaming.dstream.{DStream, ReceiverInputDStream}

object JoinTest {

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

//1.创建SparkConf

val sparkConf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("JoinTest")

//2.创建StreamingContext

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

//3.从端口获取数据创建流

val lineDStream1: ReceiverInputDStream[String] = ssc.socketTextStream("linux1", 9999)

val lineDStream2: ReceiverInputDStream[String] = ssc.socketTextStream("linux2", 8888)

//4.将两个流转换为KV类型

val wordToOneDStream: DStream[(String, Int)] = lineDStream1.flatMap(\_.split(" ")).map((\_, 1))

val wordToADStream: DStream[(String, String)] = lineDStream2.flatMap(\_.split(" ")).map((\_, "a"))

//5.流的JOIN

val joinDStream: DStream[(String, (Int, String))] = wordToOneDStream.join(wordToADStream)

//6.打印

joinDStream.print()

//7.启动任务

ssc.start()

ssc.awaitTermination()

}

}

## 4.2 有状态转化操作

### 4.2.1 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，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

1. 编写代码

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[\*]").setAppName("NetworkWordCount")

val ssc = new StreamingContext(conf, Seconds(3))

ssc.checkpoint("./ck")

// Create a DStream that will connect to hostname:port, like hadoop102:9999

val lines = ssc.socketTextStream("linux1", 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()

ssc.start() // Start the computation

ssc.awaitTermination() // Wait for the computation to terminate

//ssc.stop()

}

}

1. 启动程序并向9999端口发送数据

nc -lk 9999

Hello World

Hello Scala

1. 结果展示

-------------------------------------------

Time: 1504685175000 ms

-------------------------------------------

-------------------------------------------

Time: 1504685181000 ms

-------------------------------------------

(shi,1)

(shui,1)

(ni,1)

-------------------------------------------

Time: 1504685187000 ms

-------------------------------------------

(shi,1)

(ma,1)

(hao,1)

(shui,1)

### 4.2.2 WindowOperations

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Steaming的允许状态。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。

* 窗口时长：计算内容的时间范围；
* 滑动步长：隔多久触发一次计算。

注意：这两者都必须为采集周期大小的整数倍。

WordCount第三版：3秒一个批次，窗口12秒，滑步6秒。

object WorldCount {

def main(args: Array[String]) {

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

val ssc = new StreamingContext(conf, Seconds(3))

ssc.checkpoint("./ck")

// Create a DStream that will connect to hostname:port, like localhost:9999

val lines = ssc.socketTextStream("linux1", 9999)

// Split each line into words

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

// 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

}

}

关于Window的操作还有如下方法：

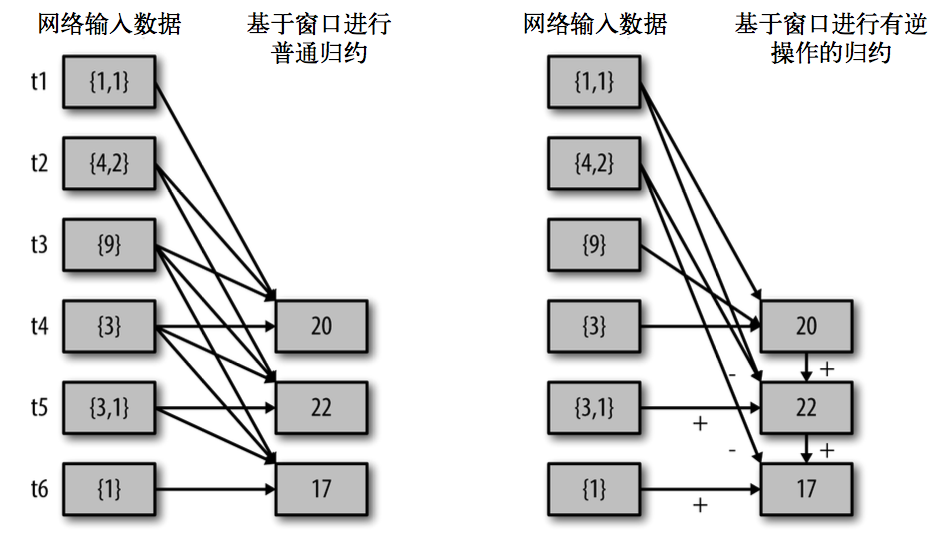
（1）window(windowLength, slideInterval): 基于对源DStream窗化的批次进行计算返回一个新的Dstream；

（2）countByWindow(windowLength, slideInterval): 返回一个滑动窗口计数流中的元素个数；

（3）reduceByWindow(func, windowLength, slideInterval): 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；

（4）reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks]): 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reduce函数来整合每个key的value值。

（5）reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks]): 这个函数是上述函数的变化版本，每个窗口的reduce值都是通过用前一个窗的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。



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))

# 第5章 DStream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。

输出操作如下：

* print()：在运行流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print()。
* saveAsTextFiles(prefix, [suffix])：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]”。
* saveAsObjectFiles(prefix, [suffix])：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。
* saveAsHadoopFiles(prefix, [suffix])：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 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。

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。

注意：

1. 连接不能写在driver层面（序列化）
2. 如果写在foreach则每个RDD中的每一条数据都创建，得不偿失；
3. 增加foreachPartition，在分区创建（获取）。

# 第6章 优雅关闭

流式任务需要7\*24小时执行，但是有时涉及到升级代码需要主动停止程序，但是分布式程序，没办法做到一个个进程去杀死，所有配置优雅的关闭就显得至关重要了。

使用外部文件系统来控制内部程序关闭。

* MonitorStop

import java.net.URI

import org.apache.hadoop.conf.Configuration

import org.apache.hadoop.fs.{FileSystem, Path}

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

class MonitorStop(ssc: StreamingContext) extends Runnable {

override def run(): Unit = {

val fs: FileSystem = FileSystem.get(new URI("hdfs://linux1:9000"), new Configuration(), "atguigu")

while (true) {

try

Thread.sleep(5000)

catch {

case e: InterruptedException =>

e.printStackTrace()

}

val state: StreamingContextState = ssc.getState

val bool: Boolean = fs.exists(new Path("hdfs://linux1:9000/stopSpark"))

if (bool) {

if (state == StreamingContextState.ACTIVE) {

ssc.stop(stopSparkContext = true, stopGracefully = true)

System.exit(0)

}

}

}

}

}

* SparkTest

import org.apache.spark.SparkConf

import org.apache.spark.streaming.dstream.{DStream, ReceiverInputDStream}

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

object SparkTest {

def createSSC(): \_root\_.org.apache.spark.streaming.StreamingContext = {

val update: (Seq[Int], Option[Int]) => Some[Int] = (values: Seq[Int], status: Option[Int]) => {

//当前批次内容的计算

val sum: Int = values.sum

//取出状态信息中上一次状态

val lastStatu: Int = status.getOrElse(0)

Some(sum + lastStatu)

}

val sparkConf: SparkConf = new SparkConf().setMaster("local[4]").setAppName("SparkTest")

//设置优雅的关闭

sparkConf.set("spark.streaming.stopGracefullyOnShutdown", "true")

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

ssc.checkpoint("./ck")

val line: ReceiverInputDStream[String] = ssc.socketTextStream("linux1", 9999)

val word: DStream[String] = line.flatMap(\_.split(" "))

val wordAndOne: DStream[(String, Int)] = word.map((\_, 1))

val wordAndCount: DStream[(String, Int)] = wordAndOne.updateStateByKey(update)

wordAndCount.print()

ssc

}

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

val ssc: StreamingContext = StreamingContext.getActiveOrCreate("./ck", () => createSSC())

new Thread(new MonitorStop(ssc)).start()

ssc.start()

ssc.awaitTermination()

}

}

# 第7章 SparkStreaming 案例实操

## 7.1 环境准备

### 7.1.1 pom文件

<dependencies>

<dependency>

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

<artifactId>spark-core\_2.12</artifactId>

<version>2.4.5</version>

</dependency>

<dependency>

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

<artifactId>spark-streaming\_2.12</artifactId>

<version>2.4.5</version>

</dependency>

<dependency>

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

<artifactId>spark-streaming-kafka-0-10\_2.12</artifactId>

<version>2.4.5</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.1.10</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.27</version>

</dependency>

</dependencies>

### 7.1.2 工具类

* PropertiesUtil

import java.io.InputStreamReader

import java.util.Properties

object PropertiesUtil {

def load(propertiesName:String): Properties ={

val prop=new Properties()

prop.load(new InputStreamReader(Thread.currentThread().getContextClassLoader.getResourceAsStream(propertiesName) , "UTF-8"))

prop

}

}

## 7.2 实时数据生成模块

* config.properties

#jdbc配置

jdbc.datasource.size=10

jdbc.url=jdbc:mysql://linux1:3306/spark2020?useUnicode=true&characterEncoding=utf8&rewriteBatchedStatements=true

jdbc.user=root

jdbc.password=000000

# Kafka配置

kafka.broker.list=linux1:9092,linux2:9092,linux3:9092

* CityInfo

/\*\*

\*

\* 城市信息表

\*

\* @param city\_id 城市id

\* @param city\_name 城市名称

\* @param area 城市所在大区

\*/

case class CityInfo (city\_id:Long,

city\_name:String,

area:String)

* RandomOptions

import scala.collection.mutable.ListBuffer

import scala.util.Random

case class RanOpt[T](value: T, weight: Int)

object RandomOptions {

def apply[T](opts: RanOpt[T]\*): RandomOptions[T] = {

val randomOptions = new RandomOptions[T]()

for (opt <- opts) {

randomOptions.totalWeight += opt.weight

for (i <- 1 to opt.weight) {

randomOptions.optsBuffer += opt.value

}

}

randomOptions

}

}

class RandomOptions[T](opts: RanOpt[T]\*) {

var totalWeight = 0

var optsBuffer = new ListBuffer[T]

def getRandomOpt: T = {

val randomNum: Int = new Random().nextInt(totalWeight)

optsBuffer(randomNum)

}

}

* MockerRealTime

import java.util.{Properties, Random}

import com.atguigu.bean.CityInfo

import com.atguigu.utils.{PropertiesUtil, RanOpt, RandomOptions}

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

import scala.collection.mutable.ArrayBuffer

object MockerRealTime {

/\*\*

\* 模拟的数据

\*

\* 格式 ：timestamp area city userid adid

\* 某个时间点 某个地区 某个城市 某个用户 某个广告

\*/

def generateMockData(): Array[String] = {

val array: ArrayBuffer[String] = ArrayBuffer[String]()

val CityRandomOpt = RandomOptions(RanOpt(CityInfo(1, "北京", "华北"), 30),

RanOpt(CityInfo(2, "上海", "华东"), 30),

RanOpt(CityInfo(3, "广州", "华南"), 10),

RanOpt(CityInfo(4, "深圳", "华南"), 20),

RanOpt(CityInfo(5, "天津", "华北"), 10))

val random = new Random()

// 模拟实时数据：

// timestamp province city userid adid

for (i <- 0 to 50) {

val timestamp: Long = System.currentTimeMillis()

val cityInfo: CityInfo = CityRandomOpt.getRandomOpt

val city: String = cityInfo.city\_name

val area: String = cityInfo.area

val adid: Int = 1 + random.nextInt(6)

val userid: Int = 1 + random.nextInt(6)

// 拼接实时数据

array += timestamp + " " + area + " " + city + " " + userid + " " + adid

}

array.toArray

}

def createKafkaProducer(broker: String): KafkaProducer[String, String] = {

// 创建配置对象

val prop = new Properties()

// 添加配置

prop.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, broker)

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

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

// 根据配置创建Kafka生产者

new KafkaProducer[String, String](prop)

}

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

// 获取配置文件config.properties中的Kafka配置参数

val config: Properties = PropertiesUtil.load("config.properties")

val broker: String = config.getProperty("kafka.broker.list")

val topic = "test"

// 创建Kafka消费者

val kafkaProducer: KafkaProducer[String, String] = createKafkaProducer(broker)

while (true) {

// 随机产生实时数据并通过Kafka生产者发送到Kafka集群中

for (line <- generateMockData()) {

kafkaProducer.send(new ProducerRecord[String, String](topic, line))

println(line)

}

Thread.sleep(2000)

}

}

}

## 7.3 需求一：广告黑名单

实现实时的动态黑名单机制：将每天对某个广告点击超过 100 次的用户拉黑。

注：黑名单保存到MySQL中。

### 7.3.1 思路分析

1）读取Kafka数据之后，并对MySQL中存储的黑名单数据做校验；

2）校验通过则对给用户点击广告次数累加一并存入MySQL；

3）在存入MySQL之后对数据做校验，如果单日超过100次则将该用户加入黑名单。

### 7.3.2 MySQL建表

创建库spark2020

1）存放黑名单用户的表

CREATE TABLE black\_list (userid CHAR(2) PRIMARY KEY);

2）存放单日各用户点击每个广告的次数

CREATE TABLE user\_ad\_count (

dt date,

userid CHAR (2),

adid CHAR (2),

count BIGINT,

PRIMARY KEY (dt, userid, adid)

);

### 7.3.3 环境准备

接下来开始实时需求的分析，需要用到SparkStreaming来做实时数据的处理，在生产环境中，绝大部分时候都是对接的Kafka数据源，创建一个SparkStreaming读取Kafka数据的工具类。

* MyKafkaUtil

import java.util.Properties

import org.apache.kafka.clients.consumer.ConsumerRecord

import org.apache.kafka.common.serialization.StringDeserializer

import org.apache.spark.streaming.StreamingContext

import org.apache.spark.streaming.dstream.InputDStream

import org.apache.spark.streaming.kafka010.{ConsumerStrategies, KafkaUtils, LocationStrategies}

object MyKafkaUtil {

//1.创建配置信息对象

private val properties: Properties = PropertiesUtil.load("config.properties")

//2.用于初始化链接到集群的地址

val broker\_list: String = properties.getProperty("kafka.broker.list")

//3.kafka消费者配置

val kafkaParam = Map(

"bootstrap.servers" -> broker\_list,

"key.deserializer" -> classOf[StringDeserializer],

"value.deserializer" -> classOf[StringDeserializer],

//消费者组

"group.id" -> "commerce-consumer-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,但是kafka宕机容易丢失数据

//如果是false，会需要手动维护kafka偏移量

"enable.auto.commit" -> (true: java.lang.Boolean)

)

// 创建DStream，返回接收到的输入数据

// LocationStrategies：根据给定的主题和集群地址创建consumer

// LocationStrategies.PreferConsistent：持续的在所有Executor之间分配分区

// ConsumerStrategies：选择如何在Driver和Executor上创建和配置Kafka Consumer

// ConsumerStrategies.Subscribe：订阅一系列主题

def getKafkaStream(topic: String, ssc: StreamingContext): InputDStream[ConsumerRecord[String, String]] = {

val dStream: InputDStream[ConsumerRecord[String, String]] = KafkaUtils.createDirectStream[String, String](ssc, LocationStrategies.PreferConsistent, ConsumerStrategies.Subscribe[String, String](Array(topic), kafkaParam))

dStream

}

}

* JdbcUtil

import java.sql.{Connection, PreparedStatement, ResultSet}

import java.util.Properties

import javax.sql.DataSource

import com.alibaba.druid.pool.DruidDataSourceFactory

object JdbcUtil {

//初始化连接池

var dataSource: DataSource = init()

//初始化连接池方法

def init(): DataSource = {

val properties = new Properties()

val config: Properties = PropertiesUtil.load("config.properties")

properties.setProperty("driverClassName", "com.mysql.jdbc.Driver")

properties.setProperty("url", config.getProperty("jdbc.url"))

properties.setProperty("username", config.getProperty("jdbc.user"))

properties.setProperty("password", config.getProperty("jdbc.password"))

properties.setProperty("maxActive", config.getProperty("jdbc.datasource.size"))

DruidDataSourceFactory.createDataSource(properties)

}

//获取MySQL连接

def getConnection: Connection = {

dataSource.getConnection

}

//执行SQL语句,单条数据插入

def executeUpdate(connection: Connection, sql: String, params: Array[Any]): Int = {

var rtn = 0

var pstmt: PreparedStatement = null

try {

connection.setAutoCommit(false)

pstmt = connection.prepareStatement(sql)

if (params != null && params.length > 0) {

for (i <- params.indices) {

pstmt.setObject(i + 1, params(i))

}

}

rtn = pstmt.executeUpdate()

connection.commit()

pstmt.close()

} catch {

case e: Exception => e.printStackTrace()

}

rtn

}

//执行SQL语句,批量数据插入

def executeBatchUpdate(connection: Connection, sql: String, paramsList: Iterable[Array[Any]]): Array[Int] = {

var rtn: Array[Int] = null

var pstmt: PreparedStatement = null

try {

connection.setAutoCommit(false)

pstmt = connection.prepareStatement(sql)

for (params <- paramsList) {

if (params != null && params.length > 0) {

for (i <- params.indices) {

pstmt.setObject(i + 1, params(i))

}

pstmt.addBatch()

}

}

rtn = pstmt.executeBatch()

connection.commit()

pstmt.close()

} catch {

case e: Exception => e.printStackTrace()

}

rtn

}

//判断一条数据是否存在

def isExist(connection: Connection, sql: String, params: Array[Any]): Boolean = {

var flag: Boolean = false

var pstmt: PreparedStatement = null

try {

pstmt = connection.prepareStatement(sql)

for (i <- params.indices) {

pstmt.setObject(i + 1, params(i))

}

flag = pstmt.executeQuery().next()

pstmt.close()

} catch {

case e: Exception => e.printStackTrace()

}

flag

}

//获取MySQL的一条数据

def getDataFromMysql(connection: Connection, sql: String, params: Array[Any]): Long = {

var result: Long = 0L

var pstmt: PreparedStatement = null

try {

pstmt = connection.prepareStatement(sql)

for (i <- params.indices) {

pstmt.setObject(i + 1, params(i))

}

val resultSet: ResultSet = pstmt.executeQuery()

while (resultSet.next()) {

result = resultSet.getLong(1)

}

resultSet.close()

pstmt.close()

} catch {

case e: Exception => e.printStackTrace()

}

result

}

//主方法,用于测试上述方法

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

}

}

### 7.3.4 代码实现

* Ads\_log

case class Ads\_log(timestamp: Long,

area: String,

city: String,

userid: String,

adid: String)

* BlackListHandler

import java.sql.Connection

import java.text.SimpleDateFormat

import java.util.Date

import com.atguigu.bean.Ads\_log

import com.atguigu.utils.JdbcUtil

import org.apache.spark.streaming.dstream.DStream

object BlackListHandler {

//时间格式化对象

private val sdf = new SimpleDateFormat("yyyy-MM-dd")

def addBlackList(filterAdsLogDSteam: DStream[Ads\_log]): Unit = {

//统计当前批次中单日每个用户点击每个广告的总次数

//1.将数据接转换结构 ads\_log=>((date,user,adid),1)

val dateUserAdToOne: DStream[((String, String, String), Long)] = filterAdsLogDSteam.map(adsLog => {

//a.将时间戳转换为日期字符串

val date: String = sdf.format(new Date(adsLog.timestamp))

//b.返回值

((date, adsLog.userid, adsLog.adid), 1L)

})

//2.统计单日每个用户点击每个广告的总次数 ((date,user,adid),1)=>((date,user,adid),count)

val dateUserAdToCount: DStream[((String, String, String), Long)] = dateUserAdToOne.reduceByKey(\_ + \_)

dateUserAdToCount.foreachRDD(rdd => {

rdd.foreachPartition(iter => {

val connection: Connection = JdbcUtil.getConnection

iter.foreach { case ((dt, user, ad), count) =>

JdbcUtil.executeUpdate(connection,

"""

|INSERT INTO user\_ad\_count (dt,userid,adid,count)

|VALUES (?,?,?,?)

|ON DUPLICATE KEY

|UPDATE count=count+?

""".stripMargin, Array(dt, user, ad, count, count))

val ct: Long = JdbcUtil.getDataFromMysql(connection, "select count from user\_ad\_count where dt=? and userid=? and adid =?", Array(dt, user, ad))

if (ct >= 30) {

JdbcUtil.executeUpdate(connection, "INSERT INTO black\_list (userid) VALUES (?) ON DUPLICATE KEY update userid=?", Array(user, user))

}

}

connection.close()

})

})

}

def filterByBlackList(adsLogDStream: DStream[Ads\_log]): DStream[Ads\_log] = {

adsLogDStream.transform(rdd => {

rdd.filter(adsLog => {

val connection: Connection = JdbcUtil.getConnection

val bool: Boolean = JdbcUtil.isExist(connection, "select \* from black\_list where userid=?", Array(adsLog.userid))

connection.close()

!bool

})

})

}

}

* RealtimeApp

import com.atguigu.bean.Ads\_log

import com.atguigu.handler.BlackListHandler

import com.atguigu.utils.MyKafkaUtil

import org.apache.kafka.clients.consumer.ConsumerRecord

import org.apache.spark.SparkConf

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

import org.apache.spark.streaming.dstream.{DStream, InputDStream}

object RealTimeApp {

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

//1.创建SparkConf

val sparkConf: SparkConf = new SparkConf().setAppName("RealTimeApp ").setMaster("local[\*]")

//2.创建StreamingContext

val ssc = new StreamingContext(sparkConf, Seconds(3))

//3.读取数据

val kafkaDStream: InputDStream[ConsumerRecord[String, String]] = MyKafkaUtil.getKafkaStream("ads\_log", ssc)

//4.将从Kafka读出的数据转换为样例类对象

val adsLogDStream: DStream[Ads\_log] = kafkaDStream.map(record => {

val value: String = record.value()

val arr: Array[String] = value.split(" ")

Ads\_log(arr(0).toLong, arr(1), arr(2), arr(3), arr(4))

})

//5.需求一：根据MySQL中的黑名单过滤当前数据集

val filterAdsLogDStream: DStream[Ads\_log] = BlackListHandler2.filterByBlackList(adsLogDStream)

//6.需求一：将满足要求的用户写入黑名单

BlackListHandler2.addBlackList(filterAdsLogDStream)

//测试打印

filterAdsLogDStream.cache()

filterAdsLogDStream.count().print()

//启动任务

ssc.start()

ssc.awaitTermination()

}

}

## 7.4 需求二：广告点击量实时统计

描述：实时统计每天各地区各城市各广告的点击总流量，并将其存入MySQL。

### 7.4.1 思路分析

1）单个批次内对数据进行按照天维度的聚合统计;

2）结合MySQL数据跟当前批次数据更新原有的数据。

### 7.4.2 MySQL建表

CREATE TABLE area\_city\_ad\_count (

dt date,

area CHAR(4),

city CHAR(4),

adid CHAR(2),

count BIGINT,

PRIMARY KEY (dt,area,city,adid)

);

### 7.4.3 代码实现

* DateAreaCityAdCountHandler

import java.sql.Connection

import java.text.SimpleDateFormat

import java.util.Date

import com.atguigu.bean.Ads\_log

import com.atguigu.utils.JdbcUtil

import org.apache.spark.streaming.dstream.DStream

object DateAreaCityAdCountHandler {

//时间格式化对象

private val sdf: SimpleDateFormat = new SimpleDateFormat("yyyy-MM-dd")

/\*\*

\* 统计每天各大区各个城市广告点击总数并保存至MySQL中

\*

\* @param filterAdsLogDStream 根据黑名单过滤后的数据集

\*/

def saveDateAreaCityAdCountToMysql(filterAdsLogDStream: DStream[Ads\_log]): Unit = {

//1.统计每天各大区各个城市广告点击总数

val dateAreaCityAdToCount: DStream[((String, String, String, String), Long)] = filterAdsLogDStream.map(ads\_log => {

//a.取出时间戳

val timestamp: Long = ads\_log.timestamp

//b.格式化为日期字符串

val dt: String = sdf.format(new Date(timestamp))

//c.组合,返回

((dt, ads\_log.area, ads\_log.city, ads\_log.adid), 1L)

}).reduceByKey(\_ + \_)

//2.将单个批次统计之后的数据集合MySQL数据对原有的数据更新

dateAreaCityAdToCount.foreachRDD(rdd => {

//对每个分区单独处理

rdd.foreachPartition(iter => {

//a.获取连接

val connection: Connection = JdbcUtil.getConnection

//b.写库

iter.foreach { case ((dt, area, city, adid), count) =>

JdbcUtil.executeUpdate(connection,

"""

|INSERT INTO area\_city\_ad\_count (dt,area,city,adid,count)

|VALUES(?,?,?,?,?)

|ON DUPLICATE KEY

|UPDATE count=count+?;

""".stripMargin,

Array(dt, area, city, adid, count, count))

}

//c.释放连接

connection.close()

})

})

}

}

* RealTimeApp

import java.sql.Connection

import com.atguigu.bean.Ads\_log

import com.atguigu.handler.{BlackListHandler, DateAreaCityAdCountHandler, LastHourAdCountHandler}

import com.atguigu.utils.{JdbcUtil, MyKafkaUtil, PropertiesUtil}

import org.apache.kafka.clients.consumer.ConsumerRecord

import org.apache.spark.SparkConf

import org.apache.spark.streaming.dstream.{DStream, InputDStream}

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

object RealTimeApp {

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

//1.创建SparkConf

val sparkConf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("RealTimeApp")

//2.创建StreamingContext

val ssc = new StreamingContext(sparkConf, Seconds(3))

//3.读取Kafka数据 1583288137305 华南 深圳 4 3

val topic: String = PropertiesUtil.load("config.properties").getProperty("kafka.topic")

val kafkaDStream: InputDStream[ConsumerRecord[String, String]] = MyKafkaUtil.getKafkaStream(topic, ssc)

//4.将每一行数据转换为样例类对象

val adsLogDStream: DStream[Ads\_log] = kafkaDStream.map(record => {

//a.取出value并按照" "切分

val arr: Array[String] = record.value().split(" ")

//b.封装为样例类对象

Ads\_log(arr(0).toLong, arr(1), arr(2), arr(3), arr(4))

})

//5.根据MySQL中的黑名单表进行数据过滤

val filterAdsLogDStream: DStream[Ads\_log] = adsLogDStream.filter(adsLog => {

//查询MySQL,查看当前用户是否存在。

val connection: Connection = JdbcUtil.getConnection

val bool: Boolean = JdbcUtil.isExist(connection, "select \* from black\_list where userid=?", Array(adsLog.userid))

connection.close()

!bool

})

filterAdsLogDStream.cache()

//6.对没有被加入黑名单的用户统计当前批次单日各个用户对各个广告点击的总次数,

// 并更新至MySQL

// 之后查询更新之后的数据,判断是否超过100次。

// 如果超过则将给用户加入黑名单

BlackListHandler.saveBlackListToMysql(filterAdsLogDStream)

//7.统计每天各大区各个城市广告点击总数并保存至MySQL中

dateAreaCityAdCountHandler.saveDateAreaCityAdCountToMysql(filterAdsLogDStream)

//10.开启任务

ssc.start()

ssc.awaitTermination()

}

}

## 7.5 需求三：最近一小时广告点击量

结果展示：

1：List[15:50->10,15:51->25,15->52->30]

2：List [15:50->10,15:51->25,15->52->30]

3：List [15:50->10,15:51->25,15->52->30]

### 7.5.1 思路分析

1）开窗确定时间范围；

2）在窗口内将数据转换数据结构为((adid,hm),count);

3）按照广告id进行分组处理，组内按照时分排序。

### 7.5.2 代码实现

* LastHourAdCountHandler

import java.text.SimpleDateFormat

import java.util.Date

import com.atguigu.bean.Ads\_log

import org.apache.spark.streaming.Minutes

import org.apache.spark.streaming.dstream.DStream

object LastHourAdCountHandler {

//时间格式化对象

private val sdf: SimpleDateFormat = new SimpleDateFormat("HH:mm")

/\*\*

\* 统计最近一小时(2分钟)广告分时点击总数

\*

\* @param filterAdsLogDStream 过滤后的数据集

\* @return

\*/

def getAdHourMintToCount(filterAdsLogDStream: DStream[Ads\_log]): DStream[(String, List[(String, Long)])] = {

//1.开窗 => 时间间隔为1个小时 window()

val windowAdsLogDStream: DStream[Ads\_log] = filterAdsLogDStream.window(Minutes(2))

//2.转换数据结构 ads\_log =>((adid,hm),1L) map()

val adHmToOneDStream: DStream[((String, String), Long)] = windowAdsLogDStream.map(adsLog => {

val timestamp: Long = adsLog.timestamp

val hm: String = sdf.format(new Date(timestamp))

((adsLog.adid, hm), 1L)

})

//3.统计总数 ((adid,hm),1L)=>((adid,hm),sum) reduceBykey(\_+\_)

val adHmToCountDStream: DStream[((String, String), Long)] = adHmToOneDStream.reduceByKey(\_ + \_)

//4.转换数据结构 ((adid,hm),sum)=>(adid,(hm,sum)) map()

val adToHmCountDStream: DStream[(String, (String, Long))] = adHmToCountDStream.map { case ((adid, hm), count) =>

(adid, (hm, count))

}

//5.按照adid分组 (adid,(hm,sum))=>(adid,Iter[(hm,sum),...]) groupByKey

adToHmCountDStream.groupByKey()

.mapValues(iter =>

iter.toList.sortWith(\_.\_1 < \_.\_1)

)

}

}

* RealTimeApp

import java.sql.Connection

import com.atguigu.bean.Ads\_log

import com.atguigu.handler.{BlackListHandler, DateAreaCityAdCountHandler, LastHourAdCountHandler}

import com.atguigu.utils.{JdbcUtil, MyKafkaUtil, PropertiesUtil}

import org.apache.kafka.clients.consumer.ConsumerRecord

import org.apache.spark.SparkConf

import org.apache.spark.streaming.dstream.{DStream, InputDStream}

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

object RealTimeApp {

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

//1.创建SparkConf

val sparkConf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("RealTimeApp")

//2.创建StreamingContext

val ssc = new StreamingContext(sparkConf, Seconds(3))

//3.读取Kafka数据 1583288137305 华南 深圳 4 3

val topic: String = PropertiesUtil.load("config.properties").getProperty("kafka.topic")

val kafkaDStream: InputDStream[ConsumerRecord[String, String]] = MyKafkaUtil.getKafkaStream(topic, ssc)

//4.将每一行数据转换为样例类对象

val adsLogDStream: DStream[Ads\_log] = kafkaDStream.map(record => {

//a.取出value并按照" "切分

val arr: Array[String] = record.value().split(" ")

//b.封装为样例类对象

Ads\_log(arr(0).toLong, arr(1), arr(2), arr(3), arr(4))

})

//5.根据MySQL中的黑名单表进行数据过滤

val filterAdsLogDStream: DStream[Ads\_log] = adsLogDStream.filter(adsLog => {

//查询MySQL,查看当前用户是否存在。

val connection: Connection = JdbcUtil.getConnection

val bool: Boolean = JdbcUtil.isExist(connection, "select \* from black\_list where userid=?", Array(adsLog.userid))

connection.close()

!bool

})

filterAdsLogDStream.cache()

//6.对没有被加入黑名单的用户统计当前批次单日各个用户对各个广告点击的总次数,

// 并更新至MySQL

// 之后查询更新之后的数据,判断是否超过100次。

// 如果超过则将给用户加入黑名单

BlackListHandler.saveBlackListToMysql(filterAdsLogDStream)

//7.统计每天各大区各个城市广告点击总数并保存至MySQL中

DateAreaCityAdCountHandler.saveDateAreaCityAdCountToMysql(filterAdsLogDStream)

//8.统计最近一小时(2分钟)广告分时点击总数

val adToHmCountListDStream: DStream[(String, List[(String, Long)])] = LastHourAdCountHandler.getAdHourMintToCount(filterAdsLogDStream)

//9.打印

adToHmCountListDStream.print()

//10.开启任务

ssc.start()

ssc.awaitTermination()

}

}