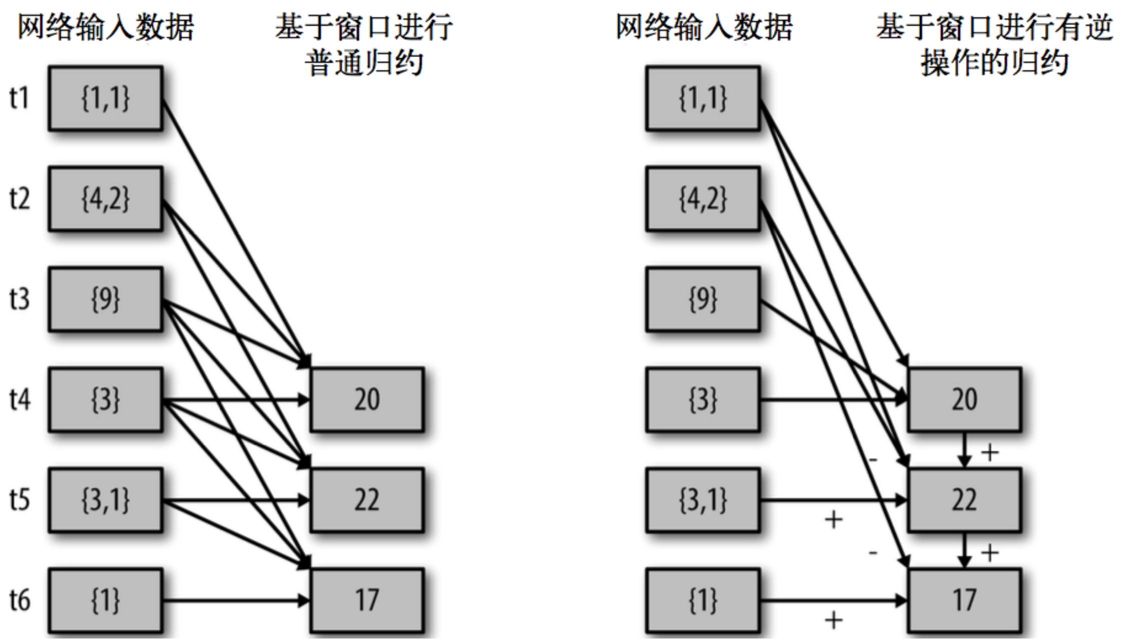
可逆的 reduce 函数”，也就是这些 reduce 函数有相应的”反 reduce”函数(以参数 invFunc 形式传入)。如前述函数，reduce 任务的数量通过可选参数来配置。



```
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>3.0.0</version>
    </dependency>

    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-streaming_2.12</artifactId>
        <version>3.0.0</version>
    </dependency>

    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-streaming-kafka-0-10_2.12</artifactId>
        <version>3.0.0</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>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.10.1</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.getResourceAsStrea
m(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=utf
8&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(1) 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 varchar(255),
  userid CHAR (1),
  adid CHAR (1),
  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 VARCHAR(255),
  area VARCHAR(255),
  city VARCHAR(255),
  adid VARCHAR(255),
  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()
  }
}
```