**基于Flink的城市交通监控平台**

## 项目整体介绍

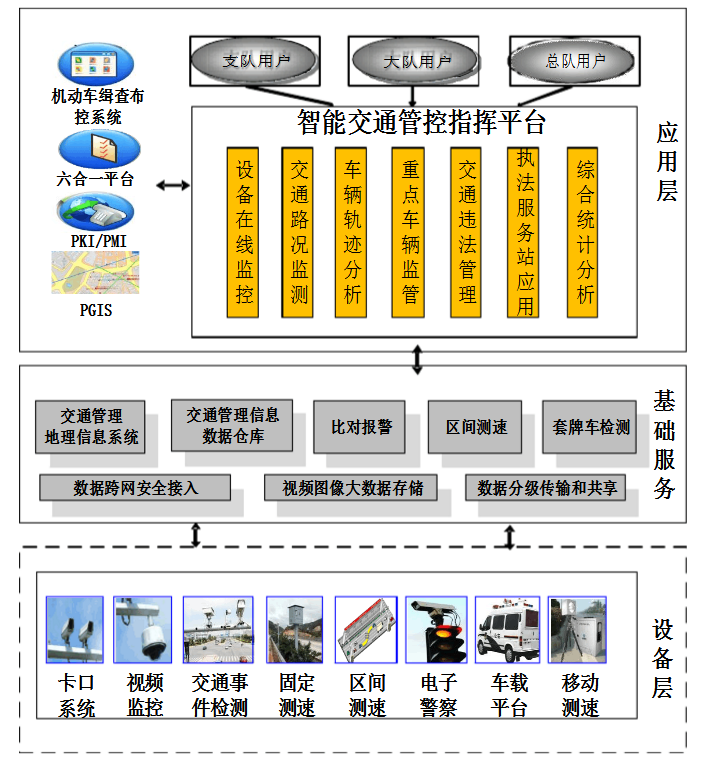
近几年来，随着国内经济的快速发展，高速公路建设步伐不断加快，全国机动车辆、驾驶员数量迅速增长，交通管理工作日益繁重，压力与日俱增。为了提高公安交通管理工作的科学化、现代化水平，缓解警力不足，加强和保障道路交通的安全、有序和畅通，减少道路交通违法和事故的发生，全国各地建设和使用了大量的“电子警察”、“高清卡口”、“固定式测速”、“区间测速”、“便携式测速”、“视频监控”、“预警系统”、“能见度天气监测系统”、“LED信息发布系统”等交通监控系统设备。尽管修建了大量的交通设施，增加了诸多前端监控设备，但交通拥挤阻塞、交通安全状况仍然十分严重。由于道路上交通监测设备种类和生产厂家繁多，目前还没有一个统一的数据采集和交换标准，无法对所有的设备、数据进行统一、高效的管理和应用，造成各种设备和管理软件混用的局面，给使用单位带来了很多不便，使得国家大量的基础建设投资未达到预期的效果。各交警支队的设备大都采用本地的数据库管理，交警总队无法看到各支队的监测设备及监测信息，严重影响对全省交通监测的宏观管理；目前网络状况为设备专网、互联网、公安网并存的复杂情况，需要充分考虑公安网的安全性，同时要保证数据的集中式管理；监控数据需要与“六合一”平台、全国机动车稽查布控系统等的数据对接，迫切需要一个全盘考虑面向交警交通行业的智慧交通管控指挥平台系统。

智慧交通管控指挥平台建成后，达到了以下效果目标：

* 交通监视和疏导：通过系统将监视区域内的现场图像传回指挥中心，使管理人员直接掌握车辆排队、堵塞、信号灯等交通状况，及时调整信号配时或通过其他手段来疏导交通，改变交通流的分布，以达到缓解交通堵塞的目的。
* 交通警卫：通过突发事件的跟踪，提高处置突发事件的能力。
* 建立公路事故、事件预警系统的指标体系及多类分析预警模型，实现对高速公路通行环境、交通运输对象、交通运输行为的综合分析和预警，建立真正意义上的分析及预警体系。
* 及时准确地掌握所监视路口、路段周围的车辆、行人的流量、交通治安情况等，为指挥人员提供迅速直观的信息从而对交通事故和交通堵塞做出准确判断并及时响应。
* 收集、处理各类公路网动静态交通安全信息，分析研判交通安全态势和事故隐患，并进行可视化展示和预警提示。
* 提供接口与其他平台信息共享和关联应用，基于各类动静态信息的大数据分析处理，实现交通违法信息的互联互通、源头监管等功能。

### 项目架构

本项目是与公安交通管理综合应用平台、机动车缉查布控系统等对接的，并且基于交通部门现有的数据平台上，进行的数据实时分析项目。



1. **相关概念**

* 卡口：道路上用于监控的某个点，可能是十字路口，也可能是高速出口等。

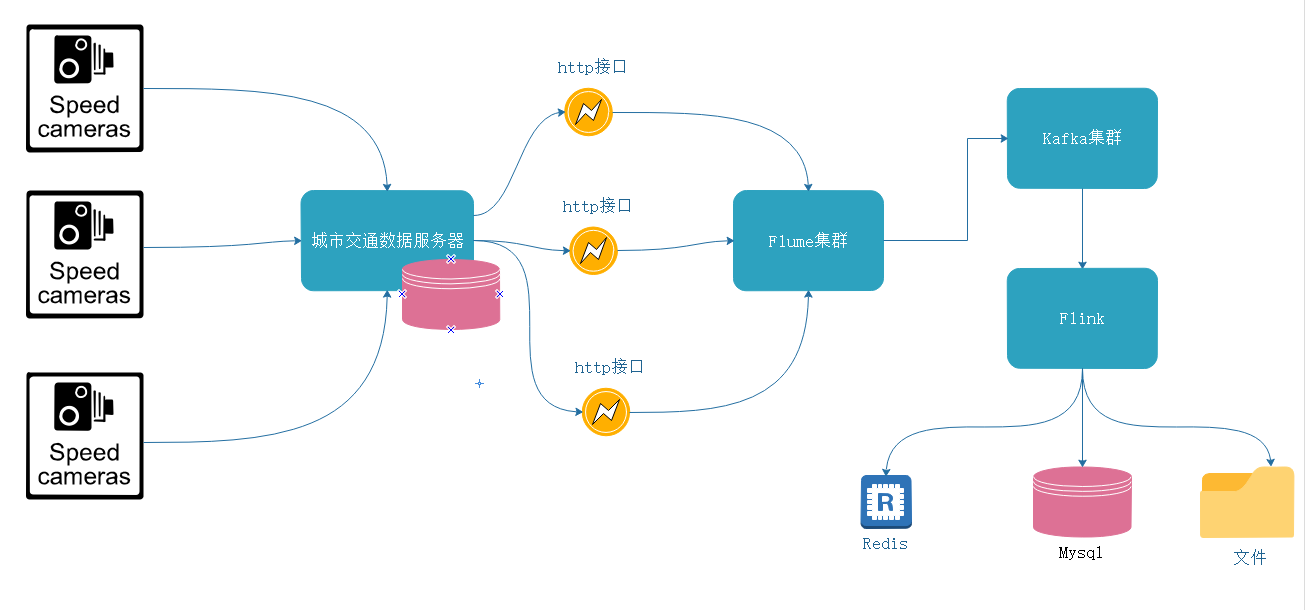


* 通道：每个卡口上有多个摄像头，每个摄像头有拍摄的方向。这些摄像头也叫通道。
* “违法王“车辆： 该车辆违法未处理超过50次以上的车。



* 摄像头拍照识别：
  + 一次拍照识别：经过卡口摄像头进行的识别，识别对象的车辆号牌信息、车辆号牌颜色信息等，基于车辆号牌和车辆颜色信息，能够实现基本的违法行为辨识、车辆黑白名单比对报警等功能。
  + 二次拍照识别：可以通过时间差和距离自动计算出车辆的速度。

### 项目数据流



实时处理流程如下：

http请求 -->数据采集接口-->数据目录--> flume监控目录[监控的目录下的文件是按照日期分的] -->Kafka [也会放在HDFS中，就是上面做的] -->Flink分析数据 --> Mysql[给运营中心使用]

### 项目主要模块

本项目的主要模块有三个方向：

1. 实时卡口监控分析：

依托卡口云管控平台达到降事故、保畅通、服务决策、引领实战的目的，最

大限度指导交通管理工作。丰富了办案手段，提高了办案效率、节省警力资源，最终达

到牵引警务模式的变革。

利用摄像头拍摄的车辆数据来分析每个卡口车辆超速监控、卡口拥堵情况监控、每个区域卡口车流量TopN统计。

1. 实时智能报警：

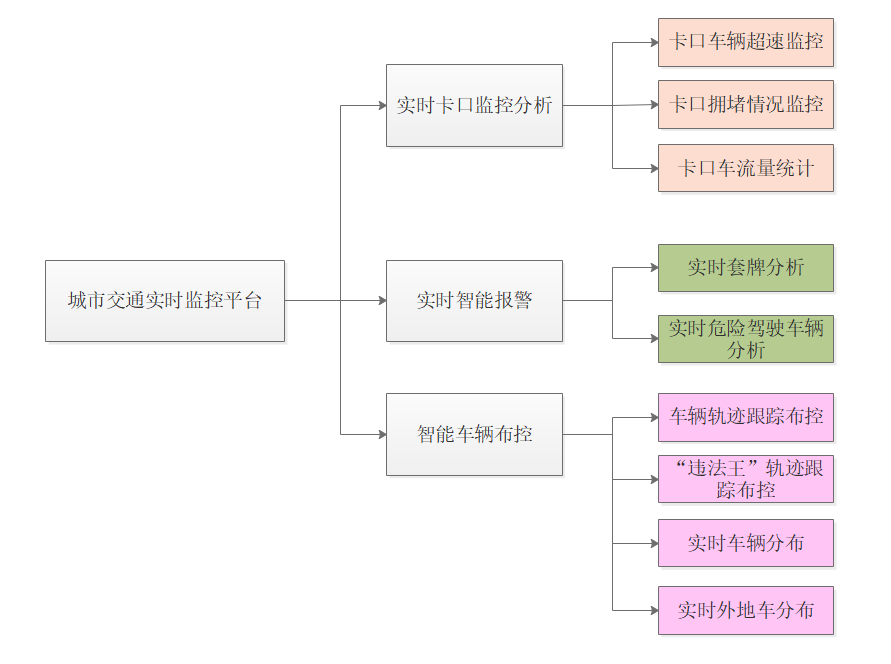
该模块主要针对路口一些无法直接用单一摄像头拍摄违章的车辆，通过海量数据分析并实时智能报警。

在一时间段内同时在 2 个区域出现的车辆记录则为可能为套牌车。这个模块包括：实时套牌分析，实时危险驾驶车辆分析。

1. 智能车辆布控：

该模块主要从整体上实时监控整个城市的车辆情况，并且对整个城市中出现“违法王”的车辆进行布控。

主要功能包括：单一车辆轨迹跟踪布控，“违法王”轨迹跟踪布控，实时车辆分布分析，实时外地车分布分析。



## 项目数据字典

### 卡口车辆采集数据

卡口数据通过Flume采集过来之后存入Kafka中，其中数据的格式为：

|  |
| --- |
| (  `action\_time` long --摄像头拍摄时间戳，精确到秒,  `monitor\_id` string --卡口号,  `camera\_id` string --摄像头编号,  `car` string --车牌号码,  `speed` double --通过卡扣的速度,  `road\_id` string --道路id,  `area\_id` string --区域id,  ) |

其中每个字段之间使用逗号隔开。

区域ID代表：一个城市的行政区域。

摄像头编号：一个卡口往往会有多个摄像头，每个摄像头都有一个唯一编号。

道路ID：城市中每一条道路都有名字，比如：蔡锷路。交通部门会给蔡锷路一个唯一编号。

### 城市交通管理数据表

Mysql数据库中有两张表是由城市交通管理平台提供的，本项目需要读取这两张表的数据来进行分析计算。

1. 城市区域表： t\_area\_info

|  |
| --- |
| DROP TABLE IF EXISTS `t\_area\_info`;  CREATE TABLE `area\_info` (  `area\_id` varchar(255) DEFAULT NULL,  `area\_name` varchar(255) DEFAULT NULL  )  --导入数据  INSERT INTO `t\_area\_info` VALUES ('01', '海淀区');  INSERT INTO `t\_area\_info` VALUES ('02', '昌平区');  INSERT INTO `t\_area\_info` VALUES ('03', '朝阳区');  INSERT INTO `t\_area\_info` VALUES ('04', '顺义区');  INSERT INTO `t\_area\_info` VALUES ('05', '西城区');  INSERT INTO `t\_area\_info` VALUES ('06', '东城区');  INSERT INTO `t\_area\_info` VALUES ('07', '大兴区');  INSERT INTO `t\_area\_info` VALUES ('08', '石景山'); |

1. 城市“违法”车辆列表：

城市“违法”车辆，一般是指需要进行实时布控的违法车辆。

|  |
| --- |
| DROP TABLE IF EXISTS `t\_violation\_list`;  CREATE TABLE `t\_violation\_list` (  `car` varchar(255) NOT NULL,  `violation` varchar(1000) DEFAULT NULL,  `create\_time` bigint(20) DEFAULT NULL,  PRIMARY KEY (`car`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  --导入数据  INSERT INTO `t\_violation\_list` VALUES ('京P88888', '违章未处理超过89次',null);  INSERT INTO `t\_violation\_list` VALUES ('京P99999', '违章未处理超过239次',null);  INSERT INTO `t\_violation\_list` VALUES ('京P77777', '违章未处理超过567次',null);  INSERT INTO `t\_violation\_list` VALUES ('京P66666', '嫌疑套牌车',null);  INSERT INTO `t\_violation\_list` VALUES ('京P55555', '嫌疑套牌车',null);  INSERT INTO `t\_violation\_list` VALUES ('京P44444', '嫌疑套牌车',null);  INSERT INTO `t\_violation\_list` VALUES ('京P33333', '违章未处理超过123次',null);  INSERT INTO `t\_violation\_list` VALUES ('京P22222', '违章未处理超过432次',null); |

1. 城市卡口限速信息表：

城市中有些卡口有限制设置，一般超过当前限速的10%要扣分。

|  |
| --- |
| DROP TABLE IF EXISTS `t\_monitor\_info`;  CREATE TABLE `t\_monitor\_info` (  `monitor\_id` varchar(255) NOT NULL,  `road\_id` varchar(255) NOT NULL,  `speed\_limit` int(11) DEFAULT NULL,  `area\_id` varchar(255) DEFAULT NULL,  PRIMARY KEY (`monitor\_id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  --导入数据  INSERT INTO `t\_monitor\_info` VALUES ('0000', '02', 60, '01');  INSERT INTO `t\_monitor\_info` VALUES ('0001', '02', 60, '02');  INSERT INTO `t\_monitor\_info` VALUES ('0002', '03', 80, '01');  INSERT INTO `t\_monitor\_info` VALUES ('0004', '05', 100, '03');  INSERT INTO `t\_monitor\_info` VALUES ('0005', '04', 0, NULL);  INSERT INTO `t\_monitor\_info` VALUES ('0021', '04', 0, NULL);  INSERT INTO `t\_monitor\_info` VALUES ('0023', '05', 0, NULL); |

### 车辆轨迹数据表

在智能车辆布控模块中，需要保存一些车辆的实时行驶轨迹，为了方便其他部门和项目方便查询获取，我们在Mysql数据库设计一张车辆实时轨迹表。

|  |
| --- |
| DROP TABLE IF EXISTS `t\_track\_info`;  CREATE TABLE `t\_track\_info` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `car` varchar(255) DEFAULT NULL,  `action\_time` bigint(20) DEFAULT NULL,  `monitor\_id` varchar(255) DEFAULT NULL,  `road\_id` varchar(255) DEFAULT NULL,  `area\_id` varchar(255) DEFAULT NULL,  `speed` double DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8; |

## 实时卡口监控分析

首先要实现的是实时卡口监控分析，由于前面课程项目中已经讲解了数据的ETL，本项目我们省略数据采集、等ETL操作。我们将读取Kafka中的数据集来进行分析。

项目主体用Scala编写，采用IDEA作为开发环境进行项目编写，采用maven作为项目构建和管理工具。首先我们需要搭建项目框架。

### 创建Maven项目

打开IDEA，创建一个maven项目，我们整个项目需要的工具的不同版本可能会对程序运行造成影响，所以应该在porm.xml文件的最上面声明所有工具的版本信息。

在pom.xml中加入以下配置：

|  |
| --- |
| <properties>  <flink.version>1.7.2</flink.version>  <scala.binary.version>2.11</scala.binary.version>  <kafka.version>0.11.0.0</kafka.version>  </properties> |

1. **添加项目依赖**

对于整个项目而言，所有模块都会用到flink相关的组件，添加Flink相关组件依赖：

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-scala\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId> <artifactId>flink-streaming-scala\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.kafka</groupId>  <artifactId>kafka\_${scala.binary.version}</artifactId>  <version>${kafka.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId> <artifactId>flink-connector-kafka\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.8.1</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-cep-scala\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  </dependencies> |

1. **添加Scala和打包插件**

|  |
| --- |
| <build>  <plugins>  <!-- 该插件用于将Scala代码编译成class文件 -->  <plugin>  <groupId>net.alchim31.maven</groupId>  <artifactId>scala-maven-plugin</artifactId>  <version>3.4.6</version>  <executions>  <execution>  <!-- 声明绑定到maven的compile阶段 -->  <goals>  <goal>testCompile</goal>  </goals>  </execution>  </executions>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-assembly-plugin</artifactId>  <version>3.0.0</version>  <configuration>  <descriptorRefs>  <descriptorRef>  jar-with-dependencies  </descriptorRef>  </descriptorRefs>  </configuration>  <executions>  <execution>  <id>make-assembly</id>  <phase>package</phase>  <goals>  <goal>single</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build> |

### 准备数据

由于在前面的课程中已经学过数据的采集和ETL，本项目不再赘述，现在我们直接随机生成数据到文件中（方便测试），同时也写入Kafka。

|  |
| --- |
| object CreateData {  def main(args: Array[String]): Unit = {  var out:FileOutputStream =null  var pw:PrintWriter =null  val r = new Random()  var producer:KafkaProducer[Nothing,String] =null  val props = new Properties()  props.setProperty("bootstrap.servers","hadoop101:9092,hadoop102:9092,hadoop103:9092")  props.setProperty("key.serializer",classOf[StringSerializer].getName)  props.setProperty("value.serializer",classOf[StringSerializer].getName)  try{  out =new FileOutputStream("E:\\monitor\_flow\_action",false)  pw =new PrintWriter(new OutputStreamWriter(out,"UTF-8"))  producer = new KafkaProducer[Nothing,String](props)  //定义车牌所在的城市，里面有一些是外地车牌  val locations = Array("鲁","京","京","京","沪","京","京","深","京","京")  var date =DateUtils.getTodayDate()  //模拟300000台车  for (i<-1 to 30000){  var car =locations(r.nextInt(10)) + (65+r.nextInt(26)).toChar + "%05d".format(r.nextInt(100000))  //得到一个随机的车辆行驶的起始时间(小时)  var baseHour ="%02d".format(r.nextInt(24))  //这里的for循环是模拟每一辆车经过不同的卡口，由不同的摄像头拍摄数据  for(j<-1 to 10+r.nextInt(300)){  //模拟每一辆车被30个摄像头拍摄之后，时间上累计加1小时，这样做数据更真实。  if(j % 30==0){  var newHour =baseHour.toInt +1  if(newHour==24){  newHour=0  }  baseHour ="%02d".format(newHour)  }  //经过卡口的时间，精确到秒，格式是yyyy-MM-dd HH:mm:ss  var actionTime=date+" "+baseHour+":"+"%02d".format(r.nextInt(60))+":"+"%02d".format(r.nextInt(60))  //得到时间戳  val actionTime2: Date = DateUtils.TIME\_FORMAT.parse(actionTime)  var monitorId="%04d".format(r.nextInt(30)) //模拟一共30多个个卡口中的随机一个  var speed =r.nextInt(260)+1 //模拟车速  var roadId ="%02d".format(r.nextInt(50)) //模拟道路ID，一共50条道路  var cameraId="%05d".format(r.nextInt(100000)) //模拟摄像头的ID  var areaId ="%02d".format(r.nextInt(8)) //模拟北京的8个区  //生成一条完整的数据  var content =actionTime2.getTime+","+monitorId+","+cameraId+","+car+","+speed+","+roadId+","+areaId  //数据写入文件中  // pw.write(content+"\n")  //数据写入Kafka中  val record = new ProducerRecord("t\_0210",content)  producer.send(record)  }  pw.flush()  }  }catch { case e:Exception=> e.printStackTrace()}  finally {  pw.close()  out.close()  producer.close()  }  }  } |

### 实时车辆超速监控

在城市交通管理数据库中，存储了每个卡口的限速信息，但是不是所有卡口都有限速信息，其中有一些卡口有限制。Flink中有广播状态流。JobManger统一管理，TaskManger中正在运行的Task不可以修改这个广播状态。只能定时更新（自定义Source）。

我们通过实时计算，需要把所有超速超过10%的车辆找出来，并写入关系型数据库中。超速结果表如下：

|  |
| --- |
| DROP TABLE IF EXISTS `t\_speeding\_info`;  CREATE TABLE `t\_speeding\_info` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `car` varchar(255) NOT NULL,  `monitor\_id` varchar(255) DEFAULT NULL,  `road\_id` varchar(255) DEFAULT NULL,  `real\_speed` double DEFAULT NULL,  `limit\_speed` int(11) DEFAULT NULL,  `action\_time` bigint(20) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8; |

在当前需求中，需要不定时的从数据库表中查询所有限速的卡口，再根据限速的卡口列表来实时的判断是否存在超速的车辆，如果找到超速的车辆，把这些车辆超速的信息保存到Mysql数据库的超速违章记录表中t\_speeding\_info。

我们把查询限速卡口列表数据作为一个事件流，车辆通行日志数据作为第二个事件流。广播状态可以用于通过一个特定的方式来组合并共同处理两个事件流。第一个流的事件被广播到另一个operator的所有并发实例，这些事件将被保存为状态。另一个流的事件不会被广播，而是发送给同一个operator的各个实例，并与广播流的事件一起处理。广播状态非常适合两个流中一个吞吐大，一个吞吐小，或者需要动态修改处理逻辑的情况。

我们对两个流使用了connect()方法，并在连接之后调用BroadcastProcessFunction接口处理两个流：

* processBroadcastElement()方法：每次收到广播流的记录时会调用。将接收到的卡口限速记录放入广播状态中；
* processElement()方法：接受到车辆通行日志流的每条消息时会调用。并能够对广播状态进行只读操作，以防止导致跨越类中多个并发实例的不同广播状态的修改。

代码如下：

|  |
| --- |
| object OutOfSpeedMonitor {  def main(args: Array[String]): Unit = {  val streamEnv: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment  //导入隐式转换  import org.apache.flink.streaming.api.scala.\_  //定义Kafka的连接属性  val props = new Properties()  props.setProperty("bootstrap.servers","hadoop101:9092,hadoop102:9092,hadoop103:9092")  props.setProperty("group.id","traffic01")  props.setProperty("key.deserializer",classOf[StringDeserializer].getName)  props.setProperty("value.deserializer",classOf[StringDeserializer].getName)  props.setProperty("auto.offset.reset","latest")  //定义KafkaSource  val stream: DataStream[TrafficMonitorLog] = streamEnv.addSource( //主流  new FlinkKafkaConsumer[String]("t\_traffic0210", new SimpleStringSchema(), props).setStartFromEarliest()  )  // val stream: DataStream[TrafficMonitorLog] = streamEnv.socketTextStream("hadoop101",9999)  .map(line => {  val arr: Array[String] = line.split(",")  new TrafficMonitorLog(arr(0).toLong, arr(1), arr(2), arr(3), arr(4).toDouble, arr(5), arr(6))  })  //广播状态的操作步骤： 1、读取广播状态数据的Source（addSource） ，  //2 、 把读取的数据广播出去  //3、 调用主流.connect算子，再调用Process底层API  //4、 定义一个底层的BroadcastProcessFunction ,实现两个方法  //获取广播状态流  val broadcastStream: BroadcastStream[MonitorInfo] = streamEnv.addSource(new JdbcReadSource("MonitorInfo"))  .map(\_.asInstanceOf[MonitorInfo])  .broadcast(GLobalConstant.MONITOR\_STATE\_DESCRIPTOR) //广播出去  stream.connect(broadcastStream)  .process(new ProcessOutOfSpeedFunction)  //写入Mysql表中  .addSink(new JdbcWriterSink[OutOfLimitSpeed](classOf[OutOfLimitSpeed]))  streamEnv.execute()  }  class ProcessOutOfSpeedFunction extends BroadcastProcessFunction[TrafficMonitorLog,MonitorInfo,OutOfLimitSpeed]{  //处理主流中的数据  override def processElement(value: TrafficMonitorLog, ctx: BroadcastProcessFunction[TrafficMonitorLog, MonitorInfo, OutOfLimitSpeed]#ReadOnlyContext, out: Collector[OutOfLimitSpeed]): Unit = {  //根据卡口ID，从广播状态中得到当前卡口的限速对象  val monitorInfo: MonitorInfo = ctx.getBroadcastState(GLobalConstant.MONITOR\_STATE\_DESCRIPTOR).get(value.monitorId)  if(monitorInfo!=null){//有限速的设置  var limitSpeed=monitorInfo.speedLimit  var realSpeed=value.speed  if(limitSpeed\* 1.1 <=realSpeed) {//超时通过卡口  val outOfLimitSpeed = new OutOfLimitSpeed(value.car,value.monitorId,value.roadId,realSpeed,limitSpeed,value.actionTime)  out.collect(outOfLimitSpeed) //只要超速就对外输出  }  }  }  //处理广播状态流中的数据,从数据流中得到数据，放入广播状态中  override def processBroadcastElement(value: MonitorInfo, ctx: BroadcastProcessFunction[TrafficMonitorLog, MonitorInfo, OutOfLimitSpeed]#Context, out: Collector[OutOfLimitSpeed]): Unit = {  val bcState: BroadcastState[String, MonitorInfo] = ctx.getBroadcastState(GLobalConstant.MONITOR\_STATE\_DESCRIPTOR)  bcState.put(value.monitorId,value)  }  }  } |

### 实时卡口拥堵情况监控

卡口的实时拥堵情况，其实就是通过卡口的车辆平均车速和通过的车辆的数量，为了统计实时的平均车速，我设定一个滑动窗口，窗口长度是为5分钟，滑动步长为1分钟。平均车速=当前窗口内通过车辆的车速之和 / 当前窗口内通过的车辆数量 ；并且在Flume采集数据的时候，我们发现数据可能出现时间乱序问题，最长迟到5秒。

实时卡口平均速度需要保存到Mysql数据库中，结果表设计为：

|  |
| --- |
| DROP TABLE IF EXISTS `t\_average\_speed`;  CREATE TABLE `t\_average\_speed` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `start\_time` bigint(20) DEFAULT NULL,  `end\_time` bigint(20) DEFAULT NULL,  `monitor\_id` varchar(255) DEFAULT NULL,  `avg\_speed` double DEFAULT NULL,  `car\_count` int(11) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8; |

完整的代码：

|  |
| --- |
| object AverageSpeedMonitor {  def main(args: Array[String]): Unit = {  val streamEnv: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment  streamEnv.setParallelism(1)  streamEnv.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)  //导入隐式转换  import org.apache.flink.streaming.api.scala.\_  //定义Kafka的连接属性  val props = new Properties()  props.setProperty("bootstrap.servers","hadoop101:9092,hadoop102:9092,hadoop103:9092")  props.setProperty("group.id","traffic01")  props.setProperty("key.deserializer",classOf[StringDeserializer].getName)  props.setProperty("value.deserializer",classOf[StringDeserializer].getName)  props.setProperty("auto.offset.reset","latest")  //定义KafkaSource  // val stream: DataStream[TrafficMonitorLog] = streamEnv.addSource( //主流  // new FlinkKafkaConsumer[String]("t\_traffic0210", new SimpleStringSchema(), props).setStartFromEarliest()  // )  val stream: DataStream[TrafficMonitorLog] = streamEnv.socketTextStream("hadoop101",9999)  .map(line => {  val arr: Array[String] = line.split(",")  new TrafficMonitorLog(arr(0).toLong, arr(1), arr(2), arr(3), arr(4).toDouble, arr(5), arr(6))  })  .assignTimestampsAndWatermarks(new BoundedOutOfOrdernessTimestampExtractor[TrafficMonitorLog](Time.seconds(5)) {  override def extractTimestamp(element: TrafficMonitorLog) = element.actionTime  })  //统计每个卡口的平均车速，设置滑动窗口，测试的时候，滑动步长为10秒，窗口长度为30秒  stream.keyBy(\_.monitorId)  .timeWindow(Time.seconds(30),Time.seconds(10))  .aggregate(new SpeedAggregate() ,new AverageSpeedFunction)  .addSink(new JdbcWriterSink[AverageSpeedInfo](classOf[AverageSpeedInfo]))  streamEnv.execute()  }  //当前需求中，累加器需要同时计算车辆的数量，还需要累加所有的车速之后,使用二元组(累加车速之后，车的数量)  class SpeedAggregate() extends AggregateFunction[TrafficMonitorLog,(Double,Long),(Double,Long)]{  override def createAccumulator(): (Double, Long) = (0.0,0)  override def add(value: TrafficMonitorLog, accumulator: (Double, Long)): (Double, Long) = {(accumulator.\_1+value.speed,accumulator.\_2+1)}  override def getResult(accumulator: (Double, Long)): (Double, Long) = accumulator  override def merge(a: (Double, Long), b: (Double, Long)): (Double, Long) = {(a.\_1+b.\_1,a.\_2+b.\_2)}  }  class AverageSpeedFunction extends WindowFunction[(Double,Long),AverageSpeedInfo,String,TimeWindow]{  //计算平均速度，然后输出  override def apply(key: String, window: TimeWindow, input: Iterable[(Double, Long)], out: Collector[AverageSpeedInfo]): Unit = {  val t: (Double, Long) = input.iterator.next()  var avg:Double=(t.\_1/t.\_2).formatted("%.2f").toDouble  out.collect(new AverageSpeedInfo(window.getStart,window.getEnd,key,avg,t.\_2.toInt))  }  }  } |

### 实时最通畅的TopN卡口

所谓的最通畅的卡口，其实就是当时的车辆数量最少的卡口，所以，其实之后在上一个功能的基础上再次开启第二个窗口操作。然后使用AllWindowFunction实现一个自定义的TopN函数Top来计算车速排名前3名的卡口，并将排名结果格式化成字符串，便于后续输出。（第二种方法：不用Flink计算，直接在Mysql数据库中计算平均车速的TopN）

完整的代码：

|  |
| --- |
| //把上一个窗口的结果数据做全量的排序计算  //根据每个窗口输出的数据（一个窗口输出多条数据） ,然后按照count降序排序。  result  // .timeWindowAll(Time.minutes(1))  .timeWindowAll(Time.seconds(5))  .process(new TopNProccessFunction(3))  .print()  streamEnv.execute()  }  /\*\*  \* 降序排序，并取topN，其中n可以通过参数传入  \* @param topN  \*/  class TopNProccessFunction(topN:Int) extends ProcessAllWindowFunction[AverageSpeed,String,TimeWindow]{  override def process(context: Context, elements: Iterable[AverageSpeed], out: Collector[String]): Unit = {  //从Iterable 得到数据按照访问的次数降序排序  val list: List[AverageSpeed] = elements.toList.sortBy(\_.avgSpeed)(Ordering.Double.reverse).take(topN)  var sb :StringBuilder =new StringBuilder()  sb.append(" 前 "+topN +"个 窗口时间："+ new Date(context.window.getEnd) +" \n")  list.foreach(t=>{  sb.append("卡口ID："+t.monitorId+",平均车速是: "+t.avgSpeed+"\n")  })  sb.append("----------------------------------\n")  out.collect(sb.toString())  }  } |

## 智能实时报警

本模块主要负责城市交通管理中，可能存在违章或者违法非常严重的行为，系统可以自动实时报警。可以实现亿级数据在线分布式计算秒级反馈。满足实战的“实时”需要，争分夺秒、聚力办案。做的真正“零”延迟的报警和出警。主要功能包括：实时套牌分析，实时危险驾驶分析等。

### 实时套牌分析

当某个卡口中出现一辆行驶的汽车，我们可以通过摄像头识别车牌号，然后在10秒内，另外一个卡口（或者当前卡口）也识别到了同样车牌的车辆，那么很有可能这两辆车之中有很大几率存在套牌车，因为一般情况下不可能有车辆在10秒内经过两个卡口。如果发现涉嫌套牌车，系统实时发出报警信息，同时这些存在套牌车嫌疑的车辆，写入Mysql数据库的结果表中，在后面的模块中，可以对这些违法车辆进行实时轨迹跟踪。

本需求可以使用CEP编程，也可以使用状态编程。我们采用状态编程。

完整的代码：

|  |
| --- |
| /\*\*  \* 实时套牌车报警,数据不考虑乱序的情况,因为一般情况下不可能有车辆在10秒内经过两个卡口  \*/  object RepetitionCarWarningAnalysis {  def main(args: Array[String]): Unit = {  val streamEnv: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment  streamEnv.setParallelism(1)  streamEnv.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)  import org.apache.flink.streaming.api.scala.\_  //读取车辆数据流  val stream: DataStream[TrafficMonitorLog] = streamEnv.socketTextStream("hadoop101", 9999)  .map(line => {  val arr: Array[String] = line.split(",")  new TrafficMonitorLog(arr(0).trim.toLong, arr(1).trim, arr(2).trim, arr(3).trim, arr(4).trim.toDouble, arr(5).trim, arr(6).trim)  }).assignAscendingTimestamps(\_.actionTime)  stream.keyBy(\_.car)  .process(new RepetitionCarWarningProcess) //必须使用底层的API，因为需要使用状态编程和触发器  .addSink(new JdbcSink[RepetitionCarWarning]("RepetitionCarWarning"))  streamEnv.execute()  }  //当前的业务，需要判断在10秒钟内，不同的卡口，或者同一个卡口，出现同一个车牌  class RepetitionCarWarningProcess extends KeyedProcessFunction[String,TrafficMonitorLog,RepetitionCarWarning]{  //使用状态保存该车辆第一次的通过卡口的时间  lazy val firstState :ValueState[TrafficMonitorLog] =getRuntimeContext.getState(new ValueStateDescriptor[TrafficMonitorLog]("first",classOf[TrafficMonitorLog]))  override def processElement(value: TrafficMonitorLog, ctx: KeyedProcessFunction[String, TrafficMonitorLog, RepetitionCarWarning]#Context, out: Collector[RepetitionCarWarning]): Unit = {  val first: TrafficMonitorLog = firstState.value()  if(first==null){//表示当前车辆第一次经过某个卡口  firstState.update(value)  }else{//当前车辆之前已经出现在某个卡口，现在要判断时间间隔是否超过10秒，超过10秒我们认为不是套牌车的概率大，如果没有超过10秒是套牌车的概率大  var nowTime =value.actionTime;  val less: Long = (nowTime-first.actionTime)/1000  if(less<=10){//可能是套牌车  var warning =new RepetitionCarWarning(value.car,first.monitorId,value.monitorId,"涉嫌套牌车",System.currentTimeMillis())  firstState.update(null)  out.collect(warning)  }else{//不是套牌车，把第二次的数据保存到状态中，以便下次判断  firstState.update(value)  }  }  }  }  } |

### 实时危险驾驶分析

在本项目中，危险驾驶是指在道路上驾驶机动车：追逐超速竞驶。我们规定：如果一辆机动车在2分钟内，超速通过卡口超过3次以上;而且每次超速的超过了规定速度的20%以上;这样的机动车涉嫌危险驾驶。系统需要实时找出这些机动车，并报警，追踪这些车辆的轨迹。注意：如果有些卡口没有设置限速值，可以设置一个城市默认限速。

这样的需求在Flink也是有两种解决思路，第一：状态编程。第二：CEP编程。但是当前的需求使用状态编程过于复杂了。所以我们采用第二种。同时还要注意：Flume在采集数据的过程中出现了数据乱序问题，一般最长延迟5秒。

涉嫌危险驾驶的车辆信息保存到Mysql数据库表（t\_violation\_list）中，以便后面的功能中统一追踪这些车辆的轨迹。

完整的代码：

|  |
| --- |
| val stream2: DataStream[OutOfLimitSpeed] = stream.map(new RichTrafficeOutOfSpeedFunction(80)) //80：没有设置限速的卡口默认是限速是80  //定义Pattern  val pattern: Pattern[OutOfLimitSpeed, OutOfLimitSpeed] = Pattern.begin[OutOfLimitSpeed]("first")  .where(t => {  (t.limitSpeed \* 1.2 <= t.realSpeed) //超速20  })  .timesOrMore(3) //超过三次以上  .greedy //尽可能多的匹配次数  .within(Time.seconds(5))  val stream3: PatternStream[OutOfLimitSpeed] = CEP.pattern(stream2.keyBy(\_.car),pattern)  val result: DataStream[(String, String, Long)] = stream3.select(new PatternSelectFunction[OutOfLimitSpeed, (String, String, Long)] {  override def select(pattern: util.Map[String, util.List[OutOfLimitSpeed]]) = {  val list: util.List[OutOfLimitSpeed] = pattern.get("first")  val first: OutOfLimitSpeed = list.get(0)  var dangeroursMsg:StringBuilder = new mutable.StringBuilder().append("涉嫌危险驾驶，经过的卡口有：")  .append(first.monitorId)  .append("->")  for (i <- 1 until list.size()) {  dangeroursMsg.append(list.get(i).monitorId).append("->")  }  (first.car, dangeroursMsg.toString(), first.actionTime)  }  })  result.print()  streamEnv.execute()  }  class RichTrafficeOutOfSpeedFunction(baseSpeed:Double) extends RichMapFunction[TrafficMonitorLog,OutOfLimitSpeed]{  var monitorMap:scala.collection.mutable.HashMap[String,MonitorInfo] =new mutable.HashMap[String,MonitorInfo]()  override def open(parameters: Configuration): Unit = {  var conn :Connection =DriverManager.getConnection("jdbc:mysql://localhost/traffic\_monitor","root","123123")  var pst :PreparedStatement=conn.prepareStatement("select monitor\_id,road\_id,speed\_limit,area\_id from t\_monitor\_info where speed\_limit>0")  val res: ResultSet = pst.executeQuery()  while(res.next()){  val m = new MonitorInfo(res.getString(1),res.getString(2),res.getInt(3),res.getString(4))  monitorMap.put(m.monitorId,m)  }  res.close()  pst.close()  conn.close()  }  override def map(value: TrafficMonitorLog): OutOfLimitSpeed = {  val monitorInfo: MonitorInfo = monitorMap.getOrElse(value.monitorId,new MonitorInfo(value.monitorId,value.roadId,baseSpeed.toInt,value.areaId))  new OutOfLimitSpeed(value.car,value.monitorId,value.roadId,value.speed,monitorInfo.speedLimit,value.actionTime)  }  } |
|  |

思考题：实时违法车辆的出警分析：

第一种，当前的违法车辆（在5分钟内）如果已经出警了。（最后输出道主流中做删除处理）。

第二种，当前违法车辆（在5分钟后）交警没有出警（发出出警的提示，在侧流中发出第一个侧流）。

第三种，有交警的出警记录，但是不是由监控平台报的警。(第二次侧流中)

### 违法车辆轨迹跟踪

城市交通中，有些车辆需要实时轨迹跟踪，这些需要跟踪轨迹的车辆，保存在城市违法表中：t\_violation\_list。系统需要实时打印这些车辆经过的卡口，并且把轨迹数据插入数据表t\_track\_info（Hbase数据库）中。根据前面所学的知识，我们应该使用Flink中的广播状态完成该功能。

完整的代码：

|  |
| --- |
| //读取车辆数据流  val stream: DataStream[TrafficMonitorLog] = streamEnv.socketTextStream("hadoop101", 9999)  .map(line => {  val arr: Array[String] = line.split(",")  new TrafficMonitorLog(arr(0).trim.toLong, arr(1).trim, arr(2).trim, arr(3).trim, arr(4).trim.toDouble, arr(5).trim, arr(6).trim)  })  //读取卡口信息流，并广播  val broadcastStream: BroadcastStream[ViolationInfo] = streamEnv.addSource(new JdbcSource("ViolationInfo")).map(\_.asInstanceOf[ViolationInfo])  .broadcast(GlobalConstant.VIOLATION\_DESCRIPTOR) //广播出去  val result: DataStream[TrackInfo] = stream.connect(broadcastStream)  .process(new BroadcastProcessFunction[TrafficMonitorLog, ViolationInfo, TrackInfo] {  override def processElement(value: TrafficMonitorLog, ctx: BroadcastProcessFunction[TrafficMonitorLog, ViolationInfo, TrackInfo]#ReadOnlyContext, out: Collector[TrackInfo]) = {  val m: ViolationInfo = ctx.getBroadcastState(GlobalConstant.VIOLATION\_DESCRIPTOR).get(value.car)  if (m != null) { //需要跟踪轨迹的车辆  val trackInfo = new TrackInfo(value.car,value.actionTime,value.monitorId,value.roadId,value.areaId,value.speed)  out.collect(trackInfo)  }  }  override def processBroadcastElement(value: ViolationInfo, ctx: BroadcastProcessFunction[TrafficMonitorLog, ViolationInfo, TrackInfo]#Context, out: Collector[TrackInfo]) = {  val bcState: BroadcastState[String, ViolationInfo] = ctx.getBroadcastState(GlobalConstant.VIOLATION\_DESCRIPTOR)  bcState.put(value.car, value)  }  }) |

## 实时车辆布控

在交警部门的指挥中心应该实时的知道整个城市的上路车辆情况，需要知道每个区一共有多少辆车？现在是否有大量的外地车进入城市等等。本模块主要是针对整个城市整体的实时车辆情况统计。

### 实时车辆分布情况

实时车辆分布情况，是指在一段时间内（比如：10分钟）整个城市中每个区多少分布多少量车。这里要注意车辆的去重，因为在10分钟内一定会有很多的车，经过不同的卡口。这些车牌相同的车，我们只统计一次。其实就是根据车牌号去重。

代码如下：

|  |
| --- |
| //读取车辆数据流  val stream: DataStream[TrafficMonitorLog] = streamEnv.readTextFile(getClass.getResource("/monitor\_flow\_action").getPath)  .map(line => {  val arr: Array[String] = line.split(",")  new TrafficMonitorLog(arr(0).trim.toLong, arr(1).trim, arr(2).trim, arr(3).trim, arr(4).trim.toDouble, arr(5).trim, arr(6).trim)  }).assignTimestampsAndWatermarks(new BoundedOutOfOrdernessTimestampExtractor[TrafficMonitorLog](Time.seconds(5)) {  override def extractTimestamp(element: TrafficMonitorLog) = element.actionTime  })  stream.keyBy(\_.areaId)  .timeWindow(Time.minutes(10))  .apply(new WindowFunction[TrafficMonitorLog,(String,Long),String,TimeWindow] {  override def apply(key: String, window: TimeWindow, input: Iterable[TrafficMonitorLog], out: Collector[(String, Long)]): Unit = {  val s: collection.mutable.Set[Long] = collection.mutable.Set()  var idSet = Set[String]()  for ( t <- input) {  idSet += t.car  }  var str:String=s"区域ID：${key},在${window.getStart} - 到 - ${window.getEnd} 的时间范围中，一共有上路车辆："  println(str)  out.collect((str,idSet.size))  }  })  .print() //写入数据库Mysql中  streamEnv.execute()  } |

### 布隆过滤器(Bloom Filter)

在上节的例子中，我们把所有数据的车牌号car都存在了窗口计算的状态里，在窗口收集数据的过程中，状态会不断增大。一般情况下，只要不超出内存的承受范围，这种做法也没什么问题；但如果我们遇到的数据量很大呢？

把所有数据暂存放到内存里，显然不是一个好注意。我们会想到，可以利用redis这种内存级k-v数据库，为我们做一个缓存。但如果我们遇到的情况非常极端，数据大到惊人呢？比如上千万级，亿级的卡口车辆数据呢？（假设）要去重计算。

如果放到redis中，假设有6千万车牌号（每个10-20字节左右的话）可能需要几G的空间来存储。当然放到redis中，用集群进行扩展也不是不可以，但明显代价太大了。

一个更好的想法是，其实我们不需要完整地存车辆的信息，只要知道他在不在就行了。所以其实我们可以进行压缩处理，用一位（bit）就可以表示一个车辆的状态。这个思想的具体实现就是布隆过滤器（Bloom Filter）。

1. 布隆过滤器的原理

本质上布隆过滤器是一种数据结构，比较巧妙的概率型数据结构（probabilistic data structure），特点是高效地插入和查询，可以用来告诉你 “某样东西一定不存在或者可能存在”。

它本身是一个很长的二进制向量，既然是二进制的向量，那么显而易见的，存放的不是0，就是1。相比于传统的 List、Set、Map 等数据结构，它更高效、占用空间更少。我们的目标就是，利用某种方法（一般是Hash函数）把每个数据，对应到一个位图的某一位上去；如果数据存在，那一位就是1，不存在则为0。

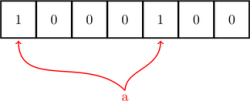
Bloom Filter是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。Bloom Filter的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省。

1. 简单的例子

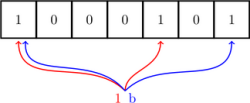
下面是一个简单的 Bloom filter 结构，开始时集合内没有元素：

IMG_256

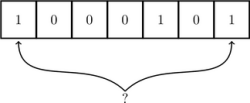
当来了一个元素 a，进行判断，这里需要一个（或者多个）哈希函数然后二进制运算（模运算），计算出对应的比特位上为 0 ，即是 a 不在集合内，将 a 添加进去：



之后的元素，要判断是不是在集合内，也是同 a 一样的方法，只有对元素哈希后对应位置上都是 1 才认为这个元素在集合内（虽然这样可能会误判）：



随着元素的插入，Bloom filter 中修改的值变多，出现误判的几率也随之变大，当新来一个元素时，满足其在集合内的条件，即所有对应位都是 1 ，这样就可能有两种情况，一是这个元素就在集合内，没有发生误判；还有一种情况就是发生误判，出现了哈希碰撞，这个元素本不在集合内。



本项目中可以采用Redis来帮我们做Bit的位图计算和判断。每个布隆过滤器对应到 Redis 的数据结构里面就是一个大型的位数组。而且Jedis为我们提供了setbit方法和getbit方法。

1. Trigger 的作用

英文单词 trigger 的意思是触发，作为名词是扳机的意思，例如枪支上的扳机就叫 trigger，所以也有开火的意思。Flink中，window操作需要伴随对窗口中的数据进行处理的逻辑，也就是窗口函数，而 Trigger 的作用就是决定何时触发窗口函数中的逻辑执行。

重写的函数：

* onElement() 方法会在窗口中每进入一条数据的时候调用一次
* onProcessingTime() 方法会在一个ProcessingTime定时器触发的时候调用
* onEventTime()方法会在一个EventTime定时器触发的时候调用
* clear()方法会在窗口清除的时候调用

TriggerResult中包含四个枚举值：

* CONTINUE：表示对窗口不执行任何操作。
* FIRE：表示对窗口中的数据按照窗口函数中的逻辑进行计算，并将结果输出。注意计算完成后，窗口中的数据并不会被清除，将会被保留。
* PURGE：表示将窗口中的数据和窗口清除。All elements in the window are cleared and the window is discarded, without evaluating the window function or emitting any elements.
* FIRE\_AND\_PURGE：表示先将数据进行计算，输出结果，然后将窗口中的数据和窗口进行清除。

代码如下：

|  |
| --- |
| object AreaDistributionAnalysisByBloomFilter {  def main(args: Array[String]): Unit = {  val streamEnv: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment  streamEnv.setParallelism(8)  // streamEnv.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)  //导入隐式转换  import org.apache.flink.streaming.api.scala.\_  //定义Kafka的连接属性  val props = new Properties()  props.setProperty("bootstrap.servers","hadoop101:9092,hadoop102:9092,hadoop103:9092")  props.setProperty("group.id","traffic09999")  props.setProperty("key.deserializer",classOf[StringDeserializer].getName)  props.setProperty("value.deserializer",classOf[StringDeserializer].getName)  props.setProperty("auto.offset.reset","latest")  //定义KafkaSource  // val stream: DataStream[TrafficMonitorLog] = streamEnv.addSource( //主流  // new FlinkKafkaConsumer[String]("t\_traffic0210", new SimpleStringSchema(), props).setStartFromEarliest()  // )  // val stream: DataStream[TrafficMonitorLog] = streamEnv.socketTextStream("hadoop101",9999)  val stream: DataStream[TrafficMonitorLog] = streamEnv.readTextFile(getClass.getResource("/traffic\_data").getPath)  .map(line => {  val arr: Array[String] = line.split(",")  new TrafficMonitorLog(arr(0).toLong, arr(1), arr(2), arr(3), arr(4).toDouble, arr(5), arr(6))  })  // .assignAscendingTimestamps(\_.actionTime)  //map集合存放每个分区的计数器  var map:mutable.HashMap[String,Long] =new mutable.HashMap[String,Long]()  stream.keyBy(\_.areaId)  // .timeWindow(Time.minutes(10))  .timeWindow(Time.hours(1))  // .timeWindow(Time.days(1))  //采用布隆过滤器处理海量数据去重的问题，同时采用Trigger来解决默认窗口状态中数据过大  .trigger(new MyTrigger) //设置窗口触发的时机和状态是否清除的问题  .process(new BloomFilterProcess(map))  .print() //数据写入mysql结果表中  streamEnv.execute()  }  //自定义Trigger  class MyTrigger extends Trigger[TrafficMonitorLog,TimeWindow]{  //当窗口中进入一条数据，直接处理，处理完之后马上删除状态。这样的话就不会把车牌信息保存状态中  override def onElement(element: TrafficMonitorLog, timestamp: Long, window: TimeWindow, ctx: Trigger.TriggerContext): TriggerResult = TriggerResult.FIRE\_AND\_PURGE  override def onProcessingTime(time: Long, window: TimeWindow, ctx: Trigger.TriggerContext): TriggerResult = TriggerResult.CONTINUE  override def onEventTime(time: Long, window: TimeWindow, ctx: Trigger.TriggerContext): TriggerResult = TriggerResult.CONTINUE  override def clear(window: TimeWindow, ctx: Trigger.TriggerContext): Unit = {}  }  //采用布隆过滤器来去重 ,ProcessWindowFunction 是全量窗口函数  class BloomFilterProcess(map:mutable.HashMap[String,Long]) extends ProcessWindowFunction[TrafficMonitorLog,String,String,TimeWindow]{  //定义redis连接  var jedis:Jedis=\_  var bloomFilter:MyBloomFilter=\_  override def open(parameters: Configuration): Unit = {  jedis=new Jedis("hadoop101",6379)  jedis.select(5)  bloomFilter =new MyBloomFilter(1<<27,2)  }  //当有一条数据进入窗口的时候就必须先使用布隆过滤器去重，然后在累加.当这条数据累加之后把它从状态中清除  override def process(key: String, context: Context, elements: Iterable[TrafficMonitorLog], out: Collector[String]): Unit = {  var windowStart =context.window.getStart  var windowEnd =context.window.getEnd  var car =elements.last.car  //1、redis数据来负责bitmap计算（返回1或者0），2、redis存储每一个区域中每个窗口的累加器  var count:Long=0  //先从redis中取得累加器的值  if(map.contains(key+"\_"+windowEnd)){ //第一次从Map集合中判断有没有这个累加器  count=map.getOrElse(key+"\_"+windowEnd,0)  }  val offsets: Array[Long] = bloomFilter.getOffsets2(car)  //为了计算准确，一区域中的一个窗口对应一个布隆过滤器  //初始化一个判断是否重复车牌  var repeated=true //默认所有车牌都重复  var loop =new Breaks  loop.breakable{  for (offset<-offsets){ //遍历下标的列表  //isContain=true（1），代表可能车牌重复，如果isContain=false(0)代表当前车牌绝对不可能重复  val isContain: lang.Boolean = jedis.getbit(key+"\_"+windowEnd,offset) //布隆过滤器的key有：区域ID+窗口时间  if(!isContain){  repeated=false  loop.break()  }  }  }  if(repeated){ //当前车牌号已经重复的，所以不用累加，直接输出  out.collect(s"区域ID：${key},在窗口起始时间:${windowStart} ---窗口结束时间:${windowEnd},一共有上路车辆为:${count}")  }else{ //当前车辆号没有出现过重复的，所以要累加  count+=1  for (offset<-offsets){ //把当前的车牌号写入布隆过滤器  jedis.setbit(key+"\_"+windowEnd,offset,true)  }  //把redis中保存的累加器更新一下  // jedis.hset(key,windowEnd.toString,count.toString)  map.put(key+"\_"+windowEnd,count)  out.collect(s"区域ID：${key},在窗口起始时间:${windowStart} ---窗口结束时间:${windowEnd},一共有上路车辆为:${count}")  }  }  }  /\*\*  \* 自定义的布隆过滤器  \* @param numBits 二进制向量的长度  \* @param numFunctions hash函数的个数  \*/  class MyBloomFilter(numBits:Long,numFunctions:Int) extends Serializable {  /\*\*  \* 自己定义的hash函数 ,采用google的hash函数  \* @param car  \* @return  \*/  def myHash(car:String) :Long={  Hashing.murmur3\_128().hashString(car,Charset.forName("UTF-8")).asLong()  }  /\*\*  \* 根据车牌，去布隆过滤器中计算得到该车牌对应的下标  \* @param car  \* @return 因为有多个hash函数，所以返回多个下标  \*/  def getOffsets(car:String) :Array[Long] ={  var firstHash=myHash(car)  // val firstHashToHash: Long = firstHash>>>16 //无符号右移位，左边不0 同时没有符号  var secondHash:Long =car.hashCode.toLong  var result:Array[Long] =new Array[Long](numFunctions) //数组的长度和hash函数的个数一样  for(i<-1 to numFunctions){ //numFunctions，两个函数  if(i==1){  // var combineHash: Long = firstHash + i \* firstHashToHash  if(firstHash<0){  firstHash= ~firstHash //取反计算  }  result(0) = firstHash % numBits //得到一个下标保存到数组  }  if(i==2){  if(secondHash<0){  secondHash= ~secondHash  }  result(1) =secondHash % numBits  }  }  result  }  /\*\*  \* 根据车牌，去布隆过滤器中计算得到该车牌对应的下标  \* @param car  \* @return 因为有多个hash函数，所以返回多个下标  \*/  def getOffsets2(car:String) :Array[Long] ={  var firstHash=myHash(car)  if(firstHash<0){  firstHash= ~ firstHash  }  var result:Array[Long] =new Array[Long](numFunctions) //数组的长度和hash函数的个数一样  result(0) =firstHash%numBits  result  }  }  } |

### 实时外地车分布情况

这个功能和前面的一样，实时统计外地车在一段时间内，整个城市的分布情况，整个城市中每个区多少分布多少量外地车。

代码如下：

|  |
| --- |
|  |