**基于Flink的城市交通监控平台**

## 项目整体介绍

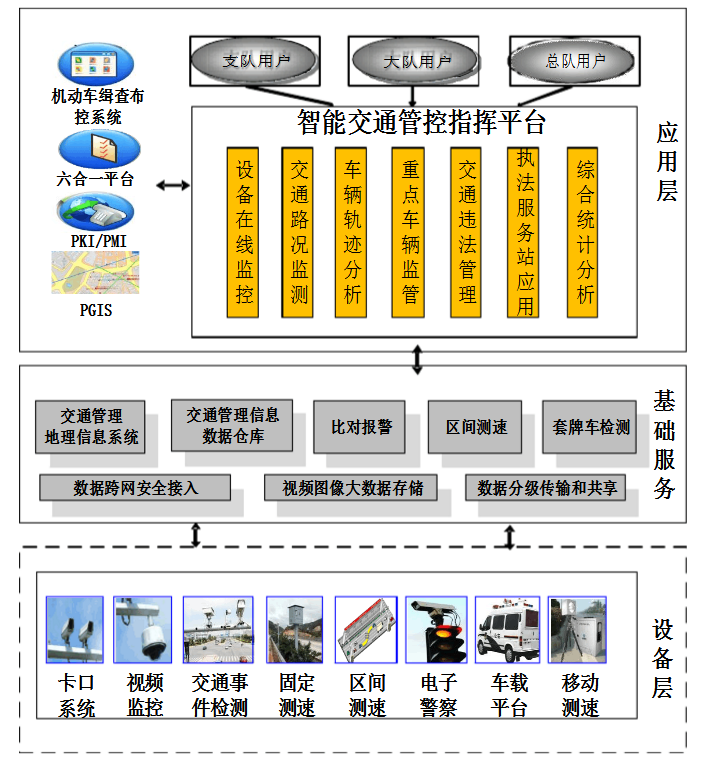
近几年来，随着国内经济的快速发展，高速公路建设步伐不断加快，全国机动车辆、驾驶员数量迅速增长，交通管理工作日益繁重，压力与日俱增。为了提高公安交通管理工作的科学化、现代化水平，缓解警力不足，加强和保障道路交通的安全、有序和畅通，减少道路交通违法和事故的发生，全国各地建设和使用了大量的“电子警察”、“高清卡口”、“固定式测速”、“区间测速”、“便携式测速”、“视频监控”、“预警系统”、“能见度天气监测系统”、“LED信息发布系统”等交通监控系统设备。尽管修建了大量的交通设施，增加了诸多前端监控设备，但交通拥挤阻塞、交通安全状况仍然十分严重。由于道路上交通监测设备种类和生产厂家繁多，目前还没有一个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。

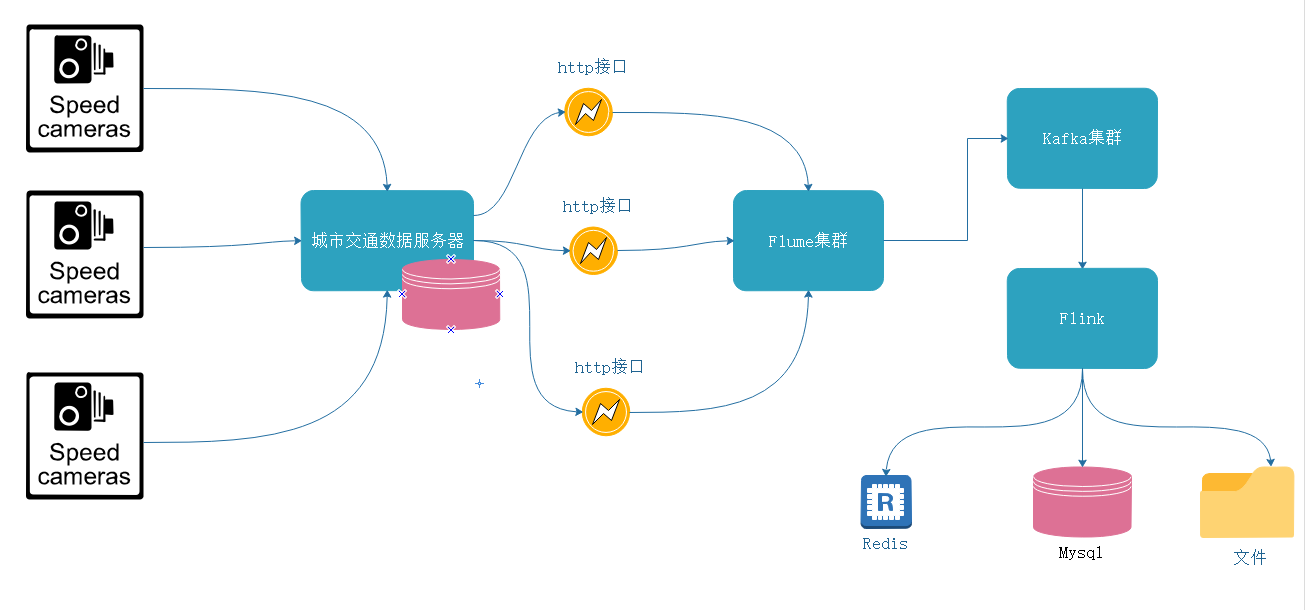


* 通道：每个卡口上有多个摄像头，每个摄像头有拍摄的方向。这些摄像头也叫通道。
* “违法王“车辆： 该车辆违法未处理超过50次以上的车。



* 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 -->Flink分析数据 --> Mysql[实时监控数据保存]

### **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：

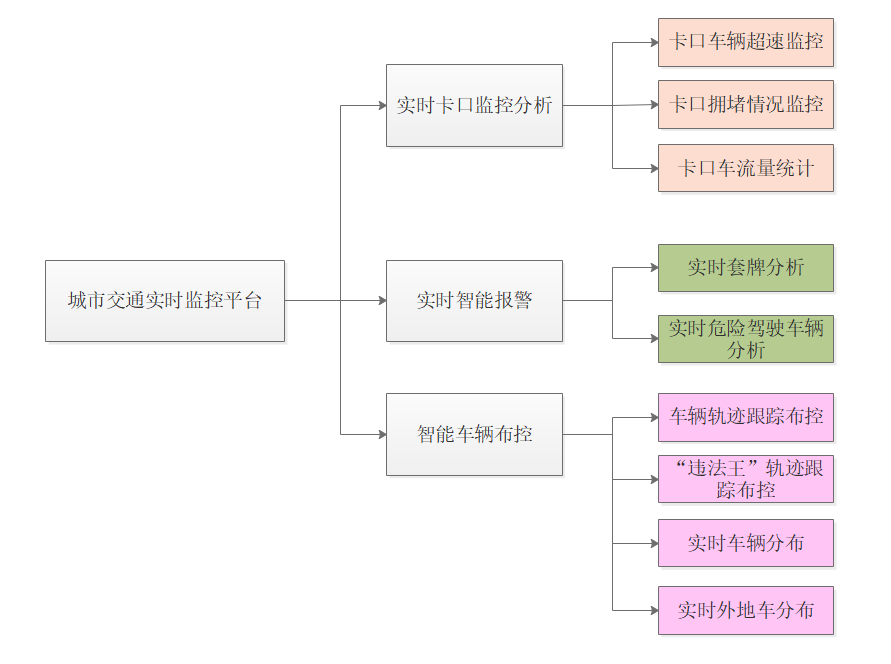
该模块主要针对路口一些无法直接用单一摄像头拍摄违章的车辆，通过海量数据分析并实时智能报警。

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` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `car` varchar(255) DEFAULT NULL,  `violation` varchar(1000) DEFAULT NULL,  `create\_time` bigint(20) DEFAULT NULL,  `detail` varchar(1000) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8; |

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` (  `area\_id` varchar(255) DEFAULT NULL,  `road\_id` varchar(255) NOT NULL,  `monitor\_id` varchar(255) NOT NULL,  `speed\_limit` int(11) DEFAULT NULL,  PRIMARY KEY (`area\_id`,`road\_id`,`monitor\_id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  --导入数据  INSERT INTO `t\_monitor\_info` VALUES ('01','10','0000','60');  INSERT INTO `t\_monitor\_info` VALUES ('02','11','0001','60');  INSERT INTO `t\_monitor\_info` VALUES ('01','12','0002','80');  INSERT INTO `t\_monitor\_info` VALUES ('03','13','0003','100'); |

### **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。如果数据量太多，需要设置在HBase中。

|  |
| --- |
| 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>1.9.1</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> |

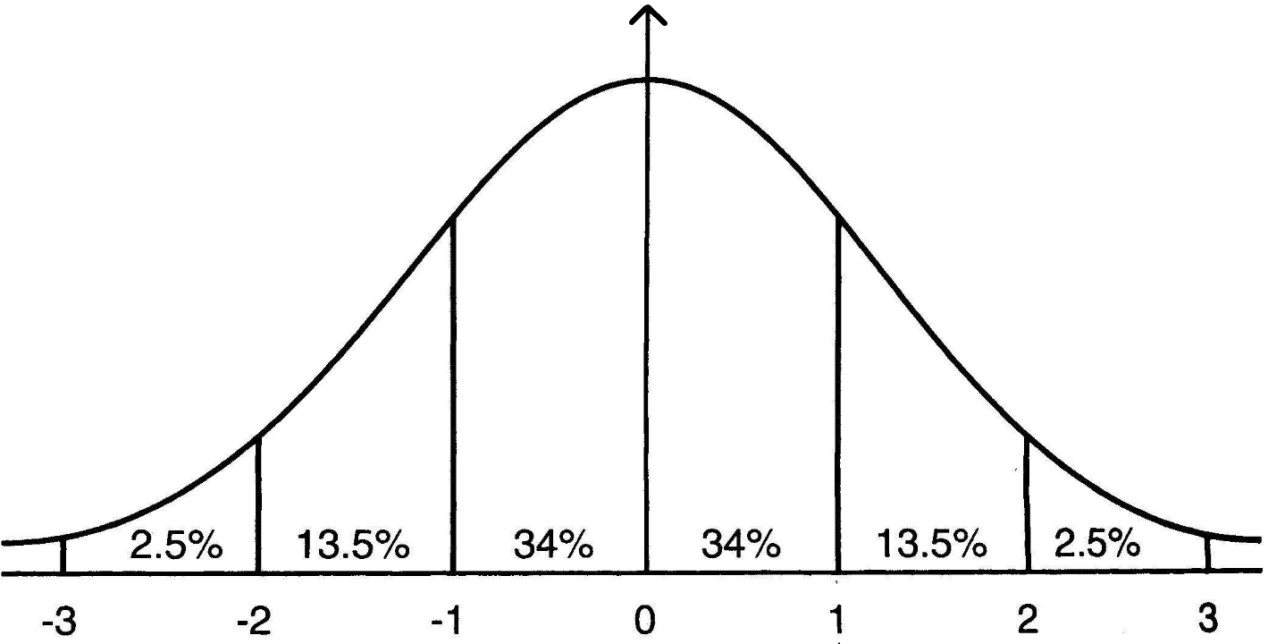
1. **添加Scala和打包插件**

|  |
| --- |
| <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。

项目中模拟车辆速度数据和车辆经过卡扣个数使用到了高斯分布，高斯分布就是正态分布。"正态分布"(Normal Distribution)可以描述所有常见的事物和现象：正常人群的身高、体重、考试成绩、家庭收入等等。这里的描述是什么意思呢？就是说这些指标背后的数据都会呈现一种中间密集、两边稀疏的特征。以身高为例，服从正态分布意味着大部分人的身高都会在人群的平均身高上下波动，特别矮和特别高的都比较少见，正态分布非常常见。



基于以上所以需要在pom.xml中导入高斯分布需要的依赖包：

|  |
| --- |
| <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-math3</artifactId>  <version>3.6.1</version>  </dependency> |

生成高斯标准分布的代码如下：

|  |
| --- |
| *//获取随机数生成器* **val** generator: JDKRandomGenerator = **new** JDKRandomGenerator() *//随机生成高斯分布的数据* **val** grg: GaussianRandomGenerator = **new** GaussianRandomGenerator(generator) *//获取标准正态分布的数据 println*(**s"随机生成数据为：$**{grg.nextNormalizedDouble()}**"**) |

模拟生成数据的代码如下：

|  |
| --- |
| */\*\*  \* 模拟生成数据,这里将数据生产到Kafka中，同时生成到文件中  \*/* **object** GeneratorData {  **def** main(args: Array[String]): Unit = {  *//创建文件流* **val** pw = **new** PrintWriter(**"./data/traffic\_data"**)   *//创建Kafka 连接properties* **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.put(**"key.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**)  props.put(**"value.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**)   **val** random = **new** Random()   *//创建Kafka produer* **val** producer = **new** KafkaProducer[String,String](props)   *//车牌号使用的地区* **val** locations = *Array*[String](**"京"**,**"津"**,**"京"**,**"鲁"**,**"京"**,**"京"**,**"冀"**,**"京"**,**"京"**,**"粤"**,**"京"**,**"京"**)   *//模拟车辆个数，这里假设每日有30万辆车信息* **for**(i <- 1 to 30000){  *//模拟每辆车的车牌号,"%05d".format(100000) %05d，d代表数字，5d代表数字长度为5位，不足位数前面补0 。 例如：京A88888* **val** car =locations(random.nextInt(12))+(65+random.nextInt(26)).toChar+**"%05d"**.format(random.nextInt(100000))   *//模拟车辆经过的卡扣数,使用高斯分布，假设正常每辆车每日经过卡扣有30个* **val** generator = **new** GaussianRandomGenerator(**new** JDKRandomGenerator())  **val** monitorThreshold: Int = 1+(generator.nextNormalizedDouble()\*30).abs.toInt *//generator.nextNormalizedDouble() 处于-1 ~ 1 之间  //模拟拍摄时间* **val** day = DateUtils.*getTodayDate*()  **var** hour = DateUtils.*getHour*()  **var** flag = 0   **for**(j <- 1 to monitorThreshold){  flag+=1   *//模拟monitor\_id ,4位长度* **val** monitorId = **"%04d"**.format(random.nextInt(9))   *//模拟camear\_id ,5为长度* **val** camearId = **"%05d"**.format(random.nextInt(100000))   *//模拟road\_id ,2为长度* **val** roadId = **"%02d"**.format(random.nextInt(50))   *//模拟area\_id ,2为长度* **val** areaId = **"%02d"**.format(random.nextInt(8))   *//模拟速度 ，使用高斯分布,速度大多位于90 左右* **val** speed = **"%.1f"**.format(60 + (generator.nextNormalizedDouble()\*30).abs)   *//模拟action\_time* **if**(flag % 30 == 0 && flag != 0 ){  hour = (hour.toInt+1).toString  }  **val** currentTime = day+**" "**+hour+**":"**+DateUtils.*getMinutesOrSeconds*()+**":"**+DateUtils.*getMinutesOrSeconds*()  *//获取action\_time 时间戳* **val** actionTime: Long = DateUtils.*getTimeStamp*(currentTime)    **var** oneInfo = **s"$**actionTime**,$**monitorId**,$**camearId**,$**car**,$**speed**,$**roadId**,$**areaId**"** *println*(**s"oneInfo = $**oneInfo**"**)   *//写入文件：* pw.write(oneInfo)  pw.println()   *//写入kafka:* producer.send(**new** ProducerRecord[String,String](**"traffic-topic"**,oneInfo))  }  }   pw.flush()  pw.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：

|  |
| --- |
| */\*\*  \* 监控超速的车辆信息  \* 思路：从mysql中读取卡扣下的限速信息，通过广播流进行广播，然后与从kafka中读取的车流量监控事件流进行connect处理  \* 广播状态操作步骤：  \* 1).读取广播流的DStream数据  \* 2).将以上DStream数据广播出去  \* 3).主流与广播流进行Connect关联，调用 process 底层API处理  \* 4).实现process方法中 BroadcastProcessFunction 类下的两个方法进行数据处理  \*/* **object** OutOfSpeedMonitor {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment   //导入隐式转换* **import** org.apache.flink.streaming.api.scala.\_   env.setParallelism(1)   **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.setProperty(**"group.id"**,**"testgroup1"**)  props.setProperty(**"key.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"value.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"auto.offset.reset"**,**"latest"**)   *//读取Kafka中的监控车辆事件流* **val** mainDStream: DataStream[TrafficLog] = env.addSource(**new** FlinkKafkaConsumer[String](**"traffic-topic"**, **new** SimpleStringSchema(), props).setStartFromEarliest()) *// val mainDStream: DataStream[TrafficLog] = env.socketTextStream("mynode5",9999)* .map(line => {  **val** arr: Array[String] = line.split(**","**)  *TrafficLog*(arr(0).toLong, arr(1), arr(2), arr(3), arr(4).toDouble, arr(5), arr(6))  })   *//广播状态流 - 卡扣限速信息* **val** broadCastStream: BroadcastStream[MonitorLimitSpeedInfo] = env.addSource(**new** JdbcReadSource(**"MonitorLimitSpeedInfo"**)).map(  one => {  one.asInstanceOf[MonitorLimitSpeedInfo]  }  ).broadcast(GlobalConstant.*MONITOR\_LIMIT\_SPEED\_DESCRIPTOR*)   **val** outOfSpeedCarInfoDStream: DataStream[OutOfSpeedCarInfo] = mainDStream.connect(broadCastStream)  .process(**new** BroadcastProcessFunction[TrafficLog, MonitorLimitSpeedInfo, OutOfSpeedCarInfo] {  *//当有车辆监控事件时会被调用* **override def** processElement(trafficLog: TrafficLog, ctx: BroadcastProcessFunction[TrafficLog, MonitorLimitSpeedInfo, OutOfSpeedCarInfo]#ReadOnlyContext, out: Collector[OutOfSpeedCarInfo]): Unit = {  *//道路\_卡扣* **val** roadMonitor = trafficLog.roadId+**"\_"**+trafficLog.monitorId  **val** info: MonitorLimitSpeedInfo = ctx.getBroadcastState(GlobalConstant.*MONITOR\_LIMIT\_SPEED\_DESCRIPTOR*).get(roadMonitor)  **if** (info != **null**) {  *//获取当前车辆真实的速度* **val** realSpeed: Double = trafficLog.speed  *//获取当前卡扣限速信息* **val** limitSpeed: Int = info.speedLimit  *//速度超过限速10% 就是超速车辆* **if** (realSpeed > limitSpeed \* 1.1) {  out.collect(*OutOfSpeedCarInfo*(trafficLog.car, trafficLog.monitorId, trafficLog.roadId, realSpeed, limitSpeed, trafficLog.actionTime))  }  }  }   *//每次收到广播流数据时，都会被调用，将接收到的卡扣限速记录放入到广播状态中* **override def** processBroadcastElement(monitorLimitSpeedInfo: MonitorLimitSpeedInfo, ctx: BroadcastProcessFunction[TrafficLog, MonitorLimitSpeedInfo, OutOfSpeedCarInfo]#Context, out: Collector[OutOfSpeedCarInfo]): Unit = {  **val** bcState: BroadcastState[String, MonitorLimitSpeedInfo] = ctx.getBroadcastState(GlobalConstant.*MONITOR\_LIMIT\_SPEED\_DESCRIPTOR*)  *//key : 道路\_卡扣 value :monitorLimitSpeedInfo* bcState.put(monitorLimitSpeedInfo.roadId+**"\_"**+monitorLimitSpeedInfo.monitorId, monitorLimitSpeedInfo)  }  })   *//将超速车辆的结果保存到 mysql 表 t\_speeding\_info 中。* **val** sink: JdbcWriteSink[OutOfSpeedCarInfo] = **new** JdbcWriteSink(**"OutOfSpeedCarInfo"**)  outOfSpeedCarInfoDStream.addSink(sink)   env.execute()   } } |

### 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，为了统计实时的平均车速，这里设定一个滑动窗口，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** MonitorAvgSpeedMonitor {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* **import** org.apache.flink.streaming.api.scala.\_  **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.setProperty(**"group.id"**,**"testgroup2"**)  props.setProperty(**"key.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"value.deserializer"**,*classOf*[StringDeserializer].getName)   *//使用时间为 事件时间* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   *//设置线程为1* env.setParallelism(1)  *// val mainDStream: DataStream[TrafficLog] = env.addSource(new FlinkKafkaConsumer[String]("traffic-topic", new SimpleStringSchema(), props))* **val** mainDStream: DataStream[TrafficLog] = env.socketTextStream(**"mynode5"**,9999)  .map(line => {  **val** arr: Array[String] = line.split(**","**)  **val** actionTime = arr(0).toLong  **val** monitorId = arr(1)  **val** cameraId = arr(2)  **val** car = arr(3)  **val** speed = arr(4).toDouble  **val** roadId = arr(5)  **val** areaId = arr(6)  *TrafficLog*(actionTime, monitorId, cameraId, car, speed, roadId, areaId)  }).assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[TrafficLog](Time.*seconds*(5)) {  **override def** extractTimestamp(element: TrafficLog): Long = element.actionTime  })   mainDStream.keyBy(\_.monitorId)  .timeWindow(Time.*minutes*(5),Time.*minutes*(1))  *//统计每个卡扣通过车辆数，统计每个卡扣下的车辆总速度和，使用增量函数* .aggregate(  **new** AggregateFunction[TrafficLog,(Int,Double),(Int,Double)] {  **override def** createAccumulator(): (Int, Double) = (0,0.0)   **override def** add(value: TrafficLog, accumulator: (Int, Double)): (Int, Double) = (accumulator.\_1+1,accumulator.\_2+value.speed)   **override def** getResult(accumulator: (Int, Double)): (Int, Double) = accumulator   **override def** merge(a: (Int, Double), b: (Int, Double)): (Int, Double) = (a.\_1+b.\_1,a.\_2+b.\_2)  },  **new** ProcessWindowFunction[(Int,Double),MonitorAvgSpeedInfo,String,TimeWindow] {  **override def** process(key: String, context: Context, elements: Iterable[(Int, Double)], out: Collector[MonitorAvgSpeedInfo]): Unit = {  **val** monitorId = key  **val** avgSpeed = (elements.last.\_2/elements.last.\_1).formatted(**"%.2f"**).toDouble  out.collect(**new** MonitorAvgSpeedInfo(context.window.getStart,context.window.getEnd,monitorId,avgSpeed,elements.last.\_1))  }  }  )  .addSink(**new** JdbcWriteSink[MonitorAvgSpeedInfo](**"MonitorAvgSpeedInfo"**))   env.execute()   } |

### 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。这里有两种实现方式，一种是基于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。另外一种是使用窗口函数，对滑动窗口内的数据全量计算并排序计算。

1. 基于上个功能基础上，完整的代码：

|  |
| --- |
| */\*\*  \* 基于 "实时卡扣拥堵情况业务" 基础之上进行统计  \*/* **object** FindTop5MonitorInfo2 {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* **import** org.apache.flink.streaming.api.scala.\_  **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.setProperty(**"group.id"**,**"testgroup2"**)  props.setProperty(**"key.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"value.deserializer"**,*classOf*[StringDeserializer].getName)   *//使用时间为 事件时间* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   *//设置线程为1* env.setParallelism(1)   **val** mainDStream: DataStream[TrafficLog] = env.addSource(**new** FlinkKafkaConsumer[String](**"traffic-topic"**, **new** SimpleStringSchema(), props).setStartFromEarliest()) *// val mainDStream: DataStream[TrafficLog] = env.socketTextStream("mynode5",9999)* .map(line => {  **val** arr: Array[String] = line.split(**","**)  **val** actionTime = arr(0).toLong  **val** monitorId = arr(1)  **val** cameraId = arr(2)  **val** car = arr(3)  **val** speed = arr(4).toDouble  **val** roadId = arr(5)  **val** areaId = arr(6)  *TrafficLog*(actionTime, monitorId, cameraId, car, speed, roadId, areaId)  }).assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[TrafficLog](Time.*seconds*(5)) {  **override def** extractTimestamp(element: TrafficLog): Long = element.actionTime  })   **val** monitorAvgSpeedDStream: DataStream[MonitorAvgSpeedInfo] = mainDStream.keyBy(\_.monitorId)  .timeWindow(Time.*minutes*(5), Time.*minutes*(1))  *//统计每个卡扣通过车辆数，统计每个卡扣下的车辆总速度和，使用增量函数* .aggregate(  **new** AggregateFunction[TrafficLog, (Int, Double), (Int, Double)] {  **override def** createAccumulator(): (Int, Double) = (0, 0.0)   **override def** add(value: TrafficLog, accumulator: (Int, Double)): (Int, Double) = (accumulator.\_1 + 1, accumulator.\_2 + value.speed)   **override def** getResult(accumulator: (Int, Double)): (Int, Double) = accumulator   **override def** merge(a: (Int, Double), b: (Int, Double)): (Int, Double) = (a.\_1 + b.\_1, a.\_2 + b.\_2)  },  **new** ProcessWindowFunction[(Int, Double), MonitorAvgSpeedInfo, String, TimeWindow] {  **override def** process(key: String, context: Context, elements: Iterable[(Int, Double)], out: Collector[MonitorAvgSpeedInfo]): Unit = {  **val** monitorId = key  **val** avgSpeed = (elements.last.\_2 / elements.last.\_1).formatted(**"%.2f"**).toDouble  out.collect(**new** MonitorAvgSpeedInfo(context.window.getStart, context.window.getEnd, monitorId, avgSpeed, elements.last.\_1))  }  }  ).assignAscendingTimestamps(masi => {  masi.endTime  })*//设置下一个窗口的时间   //这里设置一个滚动窗口，每隔1分钟，对以上所有卡扣对应的平均速度进行排序，得到对应的结果* monitorAvgSpeedDStream.timeWindowAll(Time.*minutes*(1))  .process(**new** ProcessAllWindowFunction[MonitorAvgSpeedInfo,String,TimeWindow] {  **override def** process(context: Context, elements: Iterable[MonitorAvgSpeedInfo], out: Collector[String]): Unit = {  **val** builder = **new** StringBuilder(**s"窗口起始时间：$**{context.window.getStart} **- $**{context.window.getEnd}**,最拥堵的前3个卡扣信息如下："**)  **val** infoes: List[MonitorAvgSpeedInfo] = elements.toList.sortWith((masi1,masi2)=>{masi1.avgSpeed > masi2.avgSpeed}).take(3)  **for**(masi <- infoes){  builder.append(**s"monitorId : $**{masi.monitorId}**,avgSpeed : $**{masi.avgSpeed} **|"**)  }  out.collect(builder.toString())  }  }).print()   env.execute()  } } |

1. 滑动窗口全量计算：

|  |
| --- |
| **object** FindTop5MonitorInfo1 {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment   //导入隐式转换* **import** org.apache.flink.streaming.api.scala.\_   *//设置并行度为1* env.setParallelism(1)   *//设置事件时间* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.setProperty(**"group.id"**,**"testgroup3"**)  props.setProperty(**"key.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"value.deserializer"**,*classOf*[StringDeserializer].getName)  **val** mainDStream: DataStream[TrafficLog] = env.addSource(**new** FlinkKafkaConsumer[String](**"traffic-topic"**, **new** SimpleStringSchema(), props).setStartFromEarliest()) *// val mainDStream: DataStream[TrafficLog] = env.socketTextStream("mynode5",9999)* .map(line => {  **val** arr: Array[String] = line.split(**","**)  *TrafficLog*(arr(0).toLong, arr(1), arr(2), arr(3), arr(4).toDouble, arr(5), arr(6))  }).assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[TrafficLog](Time.*seconds*(5)) {  **override def** extractTimestamp(element: TrafficLog): Long = element.actionTime  })   mainDStream  .timeWindowAll(Time.*minutes*(1))  .aggregate(  *//返回数据为 Map[String,Double] => Map[卡扣，平均速度]* **new** AggregateFunction[TrafficLog,Map[String,(Int,Double)],Map[String,Double]]{  *//初始化一个Map[卡扣，(当前卡扣对应总车辆数，当前卡扣下所有车辆总速度和)]* **override def** createAccumulator(): Map[String, (Int, Double)] = Map()   **override def** add(value: TrafficLog, accMap: Map[String, (Int, Double)]): Map[String, (Int, Double)] = {  *//获取当前一条数据的monitorID* **val** monitorId: String = value.monitorId  **if**(accMap.contains(monitorId)){*//当前map中包含此卡扣* accMap.put(monitorId,(accMap.get(monitorId).get.\_1+1,accMap.get(monitorId).get.\_2+value.speed))  }**else**{  accMap.put(monitorId,(1,value.speed))  }  accMap  }   **override def** getResult(accumulator: Map[String,(Int, Double)]): Map[String, Double] = {  accumulator.map(tp=>{  **val** monitorId: String = tp.\_1  **val** totalCarCount: Int = tp.\_2.\_1  **val** totalSpeed: Double = tp.\_2.\_2  (monitorId,(totalSpeed/totalCarCount).formatted(**"%.2f"**).toDouble)  })  }   *//合并不同线程处理的数据* **override def** merge(a: Map[String, (Int, Double)], b: Map[String, (Int, Double)]): Map[String, (Int, Double)] = {  b.foreach(tp=>{  **val** monitorId: String = tp.\_1  **val** carCount: Int = tp.\_2.\_1  **val** totalSpeed: Double = tp.\_2.\_2  **if**(a.contains(monitorId)){*//第一个map中包含当前卡扣数据* a.put(monitorId,(a.get(monitorId).get.\_1 + carCount,a.get(monitorId).get.\_2+totalSpeed))  }**else**{  *//第一个map中不包含当前卡扣数据* a.put(monitorId,tp.\_2)  }  })  a  }  },  **new** AllWindowFunction[Map[String, Double],String,TimeWindow] {  **override def** apply(window: TimeWindow, input: scala.Iterable[mutable.Map[String, Double]], out: Collector[String]): Unit = {  **val** tuples: List[(String, Double)] = input.last.toList.sortWith((tp1,tp2)=>{tp1.\_2 > tp2.\_2}).take(3)  **val** returnStr = **new** StringBuilder(**s"窗口起始时间：$**{window.getStart} **- $**{window.getEnd} **,最拥堵前3个卡扣信息 ："**)  **for**(tp <- tuples){  returnStr.append(**s"monitorId = $**{tp.\_1} **,avgSpeed = $**{tp.\_2} **|"**)  }  out.collect(returnStr.toString())  }  }  ).print()   env.execute() |

## 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：

|  |
| --- |
| **object** RepatitionCarWarning {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment  //导入scala包* **import** org.apache.flink.streaming.api.scala.\_   *//设置并行度* env.setParallelism(1)   *//设置事件时间* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.setProperty(**"group.id"**,**"test4"**)  props.setProperty(**"key.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"value.deserializer"**,*classOf*[StringDeserializer].getName)   **val** trafficDStream: DataStream[TrafficLog] = env.addSource(**new** FlinkKafkaConsumer[String](**"traffic-topic"**, **new** SimpleStringSchema(), props).setStartFromEarliest()) *// val trafficDStream: DataStream[TrafficLog] = env.socketTextStream("mynode5",9999)* .map(line => {  **val** arr: Array[String] = line.split(**","**)  *TrafficLog*(arr(0).toLong, arr(1), arr(2), arr(3), arr(4).toDouble, arr(5), arr(6))  }).assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[TrafficLog](Time.*seconds*(5)) {  **override def** extractTimestamp(element: TrafficLog): Long = element.actionTime  })   trafficDStream.keyBy(\_.car).process(**new** KeyedProcessFunction[String,TrafficLog,RepatitionCarInfo] {  **lazy private val** *valueState*: ValueState[TrafficLog] = getRuntimeContext.getState(**new** ValueStateDescriptor[TrafficLog](**"valueState"**,*classOf*[TrafficLog]))   **override def** processElement(value: TrafficLog, ctx: KeyedProcessFunction[String, TrafficLog, RepatitionCarInfo]#Context, out: Collector[RepatitionCarInfo]): Unit = {  **if**(*valueState*.value() != **null**){*//如果状态中包含当前车辆* **val** log: TrafficLog = *valueState*.value()  *//同一车辆数据，判断两次通过卡扣间隔时长* **var** dur = (log.actionTime - value.actionTime).abs  **if**(dur < 10\*1000){  out.collect(**new** RepatitionCarInfo(value.car,**"涉嫌套牌"**,System.*currentTimeMillis*(),  **s"该车辆连续两次经过的卡扣及对应时间为：$**{log.monitorId} **- $**{log.actionTime} **, $**{value.monitorId} **- $**{value.actionTime} **"**))  }  *//更新状态数据* **if**(log.actionTime < value.actionTime){  *valueState*.update(value)  }  }**else**{ *//状态中不包含当前车辆  valueState*.update(value)  }  }  })  .addSink(**new** JdbcWriteSink[RepatitionCarInfo](**"RepatitionCarInfo"**))   env.execute()  } } |

### 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。

**注意：如果要设置水位线需要设置在两个连接流连接之后。**

完整的代码：

|  |
| --- |
| **case class** newTrafficLog(actionTime:Long,monitorId:String,cameraId:String,car:String,speed:Double,roadId:String,areaId:String,monitorLimitSpeed:Int)  **object** DangerDriveCarWarning {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment   //设置并行度* env.setParallelism(1)  *//导入隐式转换* **import** org.apache.flink.streaming.api.scala.\_  **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.setProperty(**"group.id"**,**"test5"**)  props.setProperty(**"key.serializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"value.serializer"**,*classOf*[StringDeserializer].getName)   *//设置事件时间* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   *//主流 // val mainDStream: DataStream[TrafficLog] = env.addSource(new FlinkKafkaConsumer[String]("traffic-topic", new SimpleStringSchema(), props).setStartFromEarliest())* **val** mainDStream: DataStream[TrafficLog] = env.socketTextStream(**"mynode5"**,9999)  .map(line => {  **val** arr: Array[String] = line.split(**","**)  *TrafficLog*(arr(0).toLong, arr(1), arr(2), arr(3), arr(4).toDouble, arr(5), arr(6))  })   *//广播流，读取mysql中的数据，这里主要是读取卡扣限速的数据* **val** bcDStream: BroadcastStream[MonitorLimitSpeedInfo] = env.addSource(**new** JdbcReadSource(**"MonitorLimitSpeedInfo"**))  .map(  one => {  one.asInstanceOf[MonitorLimitSpeedInfo]  })  .broadcast(GlobalConstant.*MONITOR\_LIMIT\_SPEED\_DESCRIPTOR*)   *//将日志流与广播流进行整合，将道路卡扣限速信息与每条车辆运行日志数据结合* **val** trafficAllInfoDStream: DataStream[newTrafficLog] = mainDStream.connect(bcDStream).process(**new** BroadcastProcessFunction[TrafficLog, MonitorLimitSpeedInfo, newTrafficLog] {  *//处理每个日志元素* **override def** processElement(value: TrafficLog, ctx: BroadcastProcessFunction[TrafficLog, MonitorLimitSpeedInfo, newTrafficLog]#ReadOnlyContext, out: Collector[newTrafficLog]): Unit = {  *//获取状态* **val** mapState: ReadOnlyBroadcastState[String, MonitorLimitSpeedInfo] = ctx.getBroadcastState(GlobalConstant.*MONITOR\_LIMIT\_SPEED\_DESCRIPTOR*)  *//获取当前道路当前卡扣 对应的限速 ，如果没有就设置限速为80* **var** limitSpeed = 80  **if** (mapState.contains(value.roadId + **"\_"** + value.monitorId)) {  limitSpeed = mapState.get(value.roadId + **"\_"** + value.monitorId).speedLimit  }  out.collect(**new** newTrafficLog(value.actionTime, value.monitorId, value.cameraId, value.car, value.speed, value.roadId, value.areaId, limitSpeed))   }   *//处理广播元素* **override def** processBroadcastElement(value: MonitorLimitSpeedInfo, ctx: BroadcastProcessFunction[TrafficLog, MonitorLimitSpeedInfo, newTrafficLog]#Context, out: Collector[newTrafficLog]): Unit = {  *//获取状态* **val** mapState: BroadcastState[String, MonitorLimitSpeedInfo] = ctx.getBroadcastState(GlobalConstant.*MONITOR\_LIMIT\_SPEED\_DESCRIPTOR*)  *//更新当前道路当前卡扣的限速数据* mapState.put(value.roadId + **"\_"** + value.monitorId, value)  *println*(**"广播状态准备就绪"**)  }  }).assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[newTrafficLog](Time.*seconds*(0)) {  **override def** extractTimestamp(element: newTrafficLog): Long = element.actionTime  })    **val** keyDs: KeyedStream[newTrafficLog, String] = trafficAllInfoDStream.keyBy(\_.car)*//按照车辆分组    //使用CEP编程定义模式， 在1分钟内连续3次超速20%通过道路卡扣的车辆* **val** pattern:Pattern[newTrafficLog, newTrafficLog] =  Pattern.*begin*[newTrafficLog](**"start"**).where(nt=>{nt.speed > nt.monitorLimitSpeed\*1.2})  .followedBy(**"second"**).where(nt=>{nt.speed > nt.monitorLimitSpeed\*1.2})  .followedBy(**"third"**).where(nt=>{nt.speed > nt.monitorLimitSpeed\*1.2})  .within(Time.*minutes*(1))*//注意：这里的时间指的是 各个时间之间的相差时间不超过1分钟。时间采用的是事件时间* **val** patternStream: PatternStream[newTrafficLog] = CEP.*pattern*(keyDs,pattern)  **val** result: DataStream[DangerDriveCarInfo] = patternStream.select((map: Map[String, Iterable[newTrafficLog]]) => {  **val** begin: newTrafficLog = map.get(**"start"**).get.last  **val** second: newTrafficLog = map.get(**"second"**).get.last  **val** third: newTrafficLog = map.get(**"third"**).get.last  **val** builder = **s"第一次通过卡扣$**{begin.monitorId}**,当前限速:$**{begin.monitorLimitSpeed}**,通过的速度为:$**{begin.speed} **|"** +  **s"第二次通过卡扣$**{second.monitorId}**,当前限速:$**{second.monitorLimitSpeed}**,通过的速度为:$**{second.speed}**|"** +  **s"第三次通过卡扣$**{third.monitorId}**,当前限速:$**{third.monitorLimitSpeed}**,通过的速度为:$**{third.speed}**"** *DangerDriveCarInfo*(begin.car, **"危险驾驶"**, System.*currentTimeMillis*(), builder.toString)  }) *// result.print()* result.addSink(**new** JdbcWriteSink[DangerDriveCarInfo](**"DangerDriveCarInfo"**))  env.execute()  }  } |

### 出警分析

当监控到道路中有一起违法交通事故时，例如：车辆危险驾驶、车辆套牌、发生交通事故等，会有对应的交警出警处理案情。违法事故实时数据会被实时监控放入topicA，交通警察出警记录会实时上报数据被放入topicB中，这里需要对违法交通事故的出警情况进行分析并对超时未处理的警情作出对应的预警。

出警分析如下：如果在topicA中出现一条违法车辆信息，如果在5分钟内已经出警，将出警信息输出到结果库中。如果5分钟内没有出警则发出出警提示。（发出出警的提示，在侧流中发出）。

这里为了方便演示，将从socket中读取数据。

1. 使用IntervalJoin实现，这是只能输出出警信息

|  |
| --- |
| **object** PoliceAnalysis1 {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment   //导入隐式转换* **import** org.apache.flink.streaming.api.scala.\_   *//设置事件时间 // env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)* **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.setProperty(**"group.id"**,**"test6"**)  props.setProperty(**"key.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"value.deserializer"**,*classOf*[StringDeserializer].getName)   *//获取监控违法车辆信息 // val illegalDStream: DataStream[IllegalCarInfo] = env.addSource(new FlinkKafkaConsumer[String]("traffic-topic1",new SimpleStringSchema(),props))* **val** illegalDStream: DataStream[IllegalCarInfo] = env.socketTextStream(**"mynode5"**, 8888).map(line => {  **val** arr: Array[String] = line.split(**","**)  *IllegalCarInfo*(arr(0), arr(1), arr(2).toLong)  }).assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[IllegalCarInfo](Time.*seconds*(3)) {  **override def** extractTimestamp(element: IllegalCarInfo): Long = element.eventTime  })   *//获取出警信息 // val policeDStream: DataStream[String] = env.addSource(new FlinkKafkaConsumer[String]("traffic-topic2",new SimpleStringSchema(),props))* **val** policeDStream: DataStream[PoliceInfo] = env.socketTextStream(**"mynode5"**, 9999).map(line => {  **val** arr: Array[String] = line.split(**","**)  *PoliceInfo*(arr(0), arr(1), arr(2), arr(3).toLong)  }).assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[PoliceInfo](Time.*seconds*(2)) {  **override def** extractTimestamp(element: PoliceInfo): Long = element.reporTime  })   *//两个流进行 intervalJoin ，相对于join ，这里不需要设置窗口，必须后面跟上between 来以时间范围大小进行join* illegalDStream.keyBy(\_.car).intervalJoin(policeDStream.keyBy(\_.car))  *//这里假设 违法信息 illegalDStream 先出现，policeDStream数据流后出现  //between(Time.seconds(10),Time.seconds(10))相当于 illegalDStream.eventTime - 10s <= policeDStream.reporTime <= illegalDStream.eventTime + 10s  //例如 illegalDStream.eventTime 为 20:05:30 可以与 policeDStream.reporTime 为 20:05:20 - 20:05:40 范围内的数据进行匹配* .between(Time.*seconds*(-10),Time.*seconds*(10))  .process(**new** ProcessJoinFunction[IllegalCarInfo,PoliceInfo,String] {  **override def** processElement(left: IllegalCarInfo, right: PoliceInfo, ctx: ProcessJoinFunction[IllegalCarInfo, PoliceInfo, String]#Context, out: Collector[String]): Unit = {  out.collect(**s"违法车辆：$**{left.car} **已经出警，警号：$**{right.policeId}**,事故时间：$**{left.eventTime}**,出警时间：$**{right.reporTime}**"**)  }  }).print()   env.execute()  } } |

1. 使用两个流的connect,可以监测事故超时出警信息

|  |
| --- |
| **object** PoliceAnalysis2 {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment   //导入隐式转换* **import** org.apache.flink.streaming.api.scala.\_   *//设置事件时间* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   *//设置并行度为1* env.setParallelism(1)   **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.setProperty(**"group.id"**,**"test6"**)  props.setProperty(**"key.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"value.deserializer"**,*classOf*[StringDeserializer].getName)   *//获取监控违法车辆信息  // val illegalDStream: DataStream[IllegalCarInfo] = env.addSource(new FlinkKafkaConsumer[String]("traffic-topic1",new SimpleStringSchema(),props))* **val** illegalDStream: DataStream[IllegalCarInfo] = env.socketTextStream(**"mynode5"**, 8888).map(line => {  **val** arr: Array[String] = line.split(**","**)  *IllegalCarInfo*(arr(0), arr(1), arr(2).toLong)  }).assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[IllegalCarInfo](Time.*seconds*(3)) {  **override def** extractTimestamp(element: IllegalCarInfo): Long = element.eventTime  })   *//获取出警信息  // val policeDStream: DataStream[String] = env.addSource(new FlinkKafkaConsumer[String]("traffic-topic2",new SimpleStringSchema(),props))* **val** policeDStream: DataStream[PoliceInfo] = env.socketTextStream(**"mynode5"**, 9999).map(line => {  **val** arr: Array[String] = line.split(**","**)  *PoliceInfo*(arr(0), arr(1), arr(2), arr(3).toLong)  }).assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[PoliceInfo](Time.*seconds*(2)) {  **override def** extractTimestamp(element: PoliceInfo): Long = element.reporTime  })   *//定义侧流* **val** ic = **new** OutputTag[IllegalCarInfo](**"IllegalCarInfo"**)  **val** pi = **new** OutputTag[PoliceInfo](**"PoliceInfo"**)   *//以上违法记录信息 与 交警出警信息 进行关联* **val** result: DataStream[String] = illegalDStream.keyBy(\_.car).connect(policeDStream.keyBy(\_.car))  .process(**new** KeyedCoProcessFunction[String, IllegalCarInfo, PoliceInfo, String] {   *//这里每个 key 都会对应一个状态* **lazy private val** *illegalCarInfoState*: ValueState[IllegalCarInfo] = getRuntimeContext.getState(**new** ValueStateDescriptor[IllegalCarInfo](**"illegalCarInfoState"**, *classOf*[IllegalCarInfo]))  **lazy private val** *policeInfoState*: ValueState[PoliceInfo] = getRuntimeContext.getState(**new** ValueStateDescriptor[PoliceInfo](**"policeInfoState"**, *classOf*[PoliceInfo]))   *//先有违法信息* **override def** processElement1(value: IllegalCarInfo, ctx: KeyedCoProcessFunction[String, IllegalCarInfo, PoliceInfo, String]#Context, out: Collector[String]): Unit = {  *//获取当前车辆的出警信息* **val** policeInfo: PoliceInfo = *policeInfoState*.value()  **if** (policeInfo != **null**) { *//说明有对应的出警记录，说明 当前违法数据迟到了  //输出结果* out.collect(**s"违法车辆:$**{value.car} **已经出警，警号：$**{policeInfo.policeId}**,事故时间:$**{value.eventTime}**,出警时间：$**{policeInfo.reporTime}**"**)  *//删除出警状态  policeInfoState*.clear()  *//删除出警记录定时器* ctx.timerService().deleteEventTimeTimer(policeInfo.reporTime + 10000)   } **else** { *//没有对应的出警记录  //进来当前车辆的违法信息后，放入状态中  illegalCarInfoState*.update(value)  *//当前车辆有了违法记录就构建定时器，定时器设置当前时间时间后10s触发，除非10s内删除对应的定时器就不会触发* ctx.timerService().registerEventTimeTimer(value.eventTime + 10000) *//这里方便演示设置定时器时长为10s* }  }   *//后有出警状态，也有可能出警状态先到* **override def** processElement2(value: PoliceInfo, ctx: KeyedCoProcessFunction[String, IllegalCarInfo, PoliceInfo, String]#Context, out: Collector[String]): Unit = {  **val** illegalCarInfo: IllegalCarInfo = *illegalCarInfoState*.value()  **if** (illegalCarInfo != **null**) {  *//对应当前车辆的违法记录中有数据，说明这个车辆有了对应的出警记录  println*(**s"这里打印就是测试是不是一个key有一个状态： 违法车辆中的状态car 是 $**{illegalCarInfo.car} **,出警记录中的车辆是$**{value.car}**"**)  *//有对应的出警记录就正常输出数据即可：* out.collect(**s"违法车辆:$**{illegalCarInfo.car} **已经出警，警号：$**{value.policeId}**,事故时间:$**{illegalCarInfo.eventTime}**,出警时间：$**{value.reporTime}**"**)  *//清空当前车辆违法状态  illegalCarInfoState*.clear()  *//删除违法记录定时器* ctx.timerService().deleteEventTimeTimer(illegalCarInfo.eventTime + 10000) *//删除定时器* } **else** { *//有了出警记录，但是没有违法记录  //这里有了出警状态，但是没有发现当前车辆违法记录，说明 出警状态数据早到了，违法记录 迟到了  //针对这种情况，将出警记录数据放入出警状态中  policeInfoState*.update(value)   *//当前车辆有了出警就构建定时器，定时器设置当前时间时间后10s触发，除非10s内删除对应的定时器就不会触发* ctx.timerService().registerEventTimeTimer(value.reporTime + 10000) *//这里方便演示设置定时器时长为10s* }  }   *//触发定时器 定时器触发后会调用onTimer 方法 ，timestamp ： 触发器触发时间* **override def** onTimer(timestamp: Long, ctx: KeyedCoProcessFunction[String, IllegalCarInfo, PoliceInfo, String]#OnTimerContext, out: Collector[String]): Unit = {  *//获取 违法记录信息状态* **val** illegalCarInfo: IllegalCarInfo = *illegalCarInfoState*.value()   *//获取 出警记录信息状态* **val** policeInfo: PoliceInfo = *policeInfoState*.value()   **if** (illegalCarInfo != **null**) {  *//没有出警记录 ，输出到侧流* ctx.output(ic, illegalCarInfo)  }   **if** (policeInfo != **null**) { *//没有违法信息 ，输出到侧流* ctx.output(pi, policeInfo)  }  *//清空以上两种状态  illegalCarInfoState*.clear()  *policeInfoState*.clear()  }  })   result.print(**"正常流"**)   **val** illegalCarInfoDStream: DataStream[IllegalCarInfo] = result.getSideOutput(ic)  **val** policeInfoDStream: DataStream[PoliceInfo] = result.getSideOutput(pi)   illegalCarInfoDStream.print(**"没有出警记录，有违法记录的信息："**)  policeInfoDStream.print(**"有出警记录，没有违法记录车辆信息："**)    env.execute()   }  } |

### 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。

需要在hbase中创建表 t\_track\_info：create 't\_track\_info','cf1'

清空hbase表命令：truncate 't\_track\_info';

完整的代码：

|  |
| --- |
| **object** RtCarTracker {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment    //导入隐式转换* **import** org.apache.flink.streaming.api.scala.\_   *//设置事件时间* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.setProperty(**"key.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"value.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"group.id"**,**"test7"**)  *// val mainDStream: DataStream[TrafficLog] = env.addSource(new FlinkKafkaConsumer[String]("traffic-topic", new SimpleStringSchema(), props))* **val** mainDStream: DataStream[TrafficLog] = env.socketTextStream(**"mynode5"**,9999)  .map(line => {  **val** arr: Array[String] = line.split(**","**)  *TrafficLog*(arr(0).toLong, arr(1), arr(2), arr(3), arr(4).toDouble, arr(5), arr(6))  }).assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[TrafficLog](Time.*seconds*(5)) {  **override def** extractTimestamp(element: TrafficLog): Long = element.actionTime  })   **val** mapStateDescriptor = **new** MapStateDescriptor[String, IllegalCarInfo](**"MapStateDescriptor"**, *classOf*[String], *classOf*[IllegalCarInfo])  *//获取广播流* **val** bcDstream: BroadcastStream[IllegalCarInfo] = env.addSource(**new** JdbcReadSource(**"IllegalCarInfo"**)).map(pojo=>{  pojo.asInstanceOf[IllegalCarInfo]  }).broadcast(mapStateDescriptor)   *//连接两个流* **val** result: DataStream[CarThroughMonitorInfo] = mainDStream.connect(bcDstream).process(**new** BroadcastProcessFunction[TrafficLog, IllegalCarInfo, CarThroughMonitorInfo] {  **override def** processElement(value: TrafficLog, ctx: BroadcastProcessFunction[TrafficLog, IllegalCarInfo, CarThroughMonitorInfo]#ReadOnlyContext, out: Collector[CarThroughMonitorInfo]): Unit = {  **val** bcState: ReadOnlyBroadcastState[String, IllegalCarInfo] = ctx.getBroadcastState(mapStateDescriptor)  **if** (bcState.get(value.car) != **null**) {  out.collect(**new** CarThroughMonitorInfo(value.car, value.actionTime, value.monitorId, value.roadId, value.areaId))  }  }   **override def** processBroadcastElement(value: IllegalCarInfo, ctx: BroadcastProcessFunction[TrafficLog, IllegalCarInfo, CarThroughMonitorInfo]#Context, out: Collector[CarThroughMonitorInfo]): Unit = {  ctx.getBroadcastState(mapStateDescriptor).put(value.car, value)  }  })  result.countWindowAll(20).process(**new** ProcessAllWindowFunction[CarThroughMonitorInfo,util.List[Put],GlobalWindow] {  **override def** process(context: Context, elements: Iterable[CarThroughMonitorInfo], out: Collector[util.List[Put]]): Unit = {  **val** list = **new** util.ArrayList[Put]()  **for**(elem<-elements){  **val** put = **new** Put(Bytes.*toBytes*(elem.car + **"\_"** + elem.date))  put.addColumn(Bytes.*toBytes*(**"cf1"**),Bytes.*toBytes*(**"area\_id"**),Bytes.*toBytes*(elem.areaID))  put.addColumn(Bytes.*toBytes*(**"cf1"**),Bytes.*toBytes*(**"road\_id"**),Bytes.*toBytes*(elem.roadID))  put.addColumn(Bytes.*toBytes*(**"cf1"**),Bytes.*toBytes*(**"monitor\_id"**),Bytes.*toBytes*(elem.monitorId))  list.add(put)  }  out.collect(list)  }  }).addSink(**new** HBaseWriteSink())  env.execute()  }  } |

HBaseSink:

|  |
| --- |
| **class** HBaseWriteSink **extends** RichSinkFunction[java.util.List[Put]]{  *//打开HBase连接* **var** *config* :conf.Configuration = \_  **var** *conn* :Connection = \_  **override def** open(parameters: Configuration): Unit = {  *config* = HBaseConfiguration.*create*();  *config*.set(**"hbase.zookeeper.quorum"**,**"mynode3:2181,mynode4:2181,mynode5:2181"**)  *conn* = ConnectionFactory.*createConnection*(*config*)  }   **override def** close(): Unit = {  *conn*.close()  }   **override def** invoke(value: java.util.List[Put], context: SinkFunction.Context[\_]): Unit = {  *//获取HBase表,在HBase中执行 ： create 't\_track\_info','cf1'* **val** table: Table = *conn*.getTable(TableName.*valueOf*(**"t\_track\_info"**))  table.put(value)  } } |

从HBase中读取车辆轨迹api：

|  |
| --- |
| */\*\*  \* 从Hbase中扫描 rowkey 范围 查询数据  \*/* **object** GetDataFromHBase {  **def** main(args: Array[String]): Unit = {  *//获取连接* **val** conf = HBaseConfiguration.*create*();  conf.set(**"hbase.zookeeper.quorum"**, **"mynode3:2181,mynode4:2181,mynode5:2181"**);  **val** conn = ConnectionFactory.*createConnection*(conf);  *//获取表* **val** table = conn.getTable(TableName.*valueOf*(**"t\_track\_info"**));    *//设置扫描 rowkey 范围* **val** scan = **new** Scan(**"鲁A65552\_1602381959000"**.getBytes(),**"鲁A65552\_1602382000000"**.getBytes())   *//查询获取结果* **val** scanner: ResultScanner = table.getScanner(scan)   *//获取结果一条数据* **var** result :Result = scanner.next()  **while**(result != **null**){  **val** row: Array[Byte] = result.getRow  **val** cells: util.List[Cell] = result.listCells()  **import** scala.collection.JavaConverters.\_  **for** (cell <- cells.asScala) {  **val** rowKey: Array[Byte] = CellUtil.*cloneRow*(cell)  **val** family: Array[Byte] = CellUtil.*cloneFamily*(cell)  **val** qualifier: Array[Byte] = CellUtil.*cloneQualifier*(cell)  **val** value: Array[Byte] = CellUtil.*cloneValue*(cell)  *println*(**s"rowKey：$**{Bytes.*toString*(row)}**,列族名称为：$**{Bytes.*toString*(family)}**,列名称为：$**{Bytes.*toString*(qualifier)}**,列值为:$**{Bytes.*toString*(value)}**"**)  }  result = scanner.next()  }  }  } |

## 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：

|  |
| --- |
| **object** RTCarAnalysis1 {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* **import** org.apache.flink.streaming.api.scala.\_   *//设置并行度* env.setParallelism(1)   *//设置事件时间为当前时间* env.setStreamTimeCharacteristic(TimeCharacteristic.*ProcessingTime*)   **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.setProperty(**"key.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"value.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"group.id"**,**"test\_group8"**)   *//读取Kafka中的数据* **val** mainDStream: KeyedStream[TrafficLog, String] = env.addSource(**new** FlinkKafkaConsumer[String](**"traffic-topic"**, **new** SimpleStringSchema(), props).setStartFromEarliest())  .map(line => {  **val** arr: Array[String] = line.split(**","**)  *TrafficLog*(arr(0).toLong, arr(1), arr(2), arr(3), arr(4).toDouble, arr(5), arr(6))  }).keyBy(\_.areaId)  mainDStream.timeWindow(Time.*minutes*(1))  .process(**new** ProcessWindowFunction[TrafficLog,String,String,TimeWindow] {  **override def** process(key: String, context: Context, elements: Iterable[TrafficLog], out: Collector[String]): Unit = {  **val** set = scala.collection.mutable.Set[String]()  **for**(elem <- elements){  set.add(elem.car)  }  out.collect(**s"开始时间：$**{context.window.getStart} **- 结束时间：$**{context.window.getEnd}**,区域ID:$**{key}**,车辆总数 = $**{set.size}**"**)  }  }).print()   env.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。

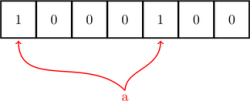
Bloom Filter是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。Bloom Filter的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省。

1. 简单的例子

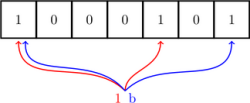
下面是一个简单的 Bloom filter 结构，开始时集合内没有元素：

IMG_256

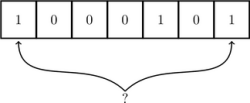
当来了一个元素 a，进行判断，这里需要一个（或者多个）哈希函数然后二进制运算（模运算），计算出对应的比特位上为 0 ，即是 a 不在集合内，将 a 添加进去：



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google 提供的BoolmFilter进行位图计算和判断：

BloomFilter.create(Funnels.stringFunnel(),100000)，Funnels.stringFunnel()指的是将对什么类型的数据使用布隆过滤器。这里我们使每个区域都对应一个布隆过滤器，位长度为100000，经过测试，可以对100万左右的数量进行去重判断，每个布隆过滤器可以认为相当于一个数组，大概占用空间为100K。

代码如下：

|  |
| --- |
| **object** RTCarAnalysis2 {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* **import** org.apache.flink.streaming.api.scala.\_   *//设置并行度 // env.setParallelism(1)   //设置事件时间为当前时间* env.setStreamTimeCharacteristic(TimeCharacteristic.*ProcessingTime*)   **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.setProperty(**"key.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"value.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"group.id"**,**"test\_group9"**)   *//读取Kafka中的数据* **val** mainDStream: KeyedStream[TrafficLog, String] = env.addSource(**new** FlinkKafkaConsumer[String](**"traffic-topic"**, **new** SimpleStringSchema(), props).setStartFromEarliest())  .map(line => {  **val** arr: Array[String] = line.split(**","**)  *TrafficLog*(arr(0).toLong, arr(1), arr(2), arr(3), arr(4).toDouble, arr(5), arr(6))  }).keyBy(\_.areaId)   *//存储 区域 - 车辆数 map* **val** map = scala.collection.mutable.Map[String,BloomFilter[CharSequence]]()   mainDStream.timeWindow(Time.*minutes*(1))  .aggregate(  **new** AggregateFunction[TrafficLog,Long,Long] {  **override def** createAccumulator(): Long = 0L   **override def** add(value: TrafficLog, accumulator: Long): Long = {  *//判断前Map中是否包含 area\_id* **if**(map.contains(value.areaId)){  *//如果包含当前区域，获取当前key对应的数值，并判断  // 车辆是否重复，* **val** bool: Boolean = map.get(value.areaId).get.mightContain(value.car)  **if**(!bool){*//如果不包含,就加1  //将当前车辆设置到布隆过滤器中* map.get(value.areaId).get.put(value.car)  accumulator + 1L  }**else**{  accumulator  }  }**else**{  *//如果不包含当前 area\_id,就设置map* map.put(value.areaId,BloomFilter.*create*(Funnels.*stringFunnel*(),100000))  *//将当前车辆设置到布隆过滤器中* map.get(value.areaId).get.put(value.car)  *//返回1* accumulator+ 1L  }  }   **override def** getResult(accumulator: Long): Long = accumulator   **override def** merge(a: Long, b: Long): Long = a+b  },  **new** WindowFunction[Long,String,String,TimeWindow] {  **override def** apply(key: String, window: TimeWindow, input: Iterable[Long], out: Collector[String]): Unit = {  out.collect(**s"窗口起始时间： $**{window.getStart} **-$**{window.getEnd} **,区域：$**{key}**,车辆总数：$**{input.last}**"**)   }  }  ).print()   env.execute()    }  } |

### 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，即统计每个区域实时外地车分布（每分钟统计一次）

代码如下：

|  |
| --- |
| **object** NonLocalCarAnalysis {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment   //导入隐式转换* **import** org.apache.flink.streaming.api.scala.\_   **val** props = **new** Properties()  props.setProperty(**"bootstrap.servers"**,**"mynode1:9092,mynode2:9092,mynode3:9092"**)  props.setProperty(**"key.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"value.deserializer"**,*classOf*[StringDeserializer].getName)  props.setProperty(**"group.id"**,**"grouptest10"**)    *//设置 BloomFilter Map* **val** map = scala.collection.mutable.Map[String,BloomFilter[CharSequence]]()   env.addSource(**new** FlinkKafkaConsumer[String](**"traffic-topic"**,**new** SimpleStringSchema(),props).setStartFromEarliest())  .map(line=>{  **val** arr: Array[String] = line.split(**","**)  *TrafficLog*(arr(0).toLong,arr(1),arr(2),arr(3),arr(4).toDouble,arr(5),arr(6))   }).filter(!\_.car.startsWith(**"京"**))  .keyBy(\_.areaId)  .timeWindow(Time.*minutes*(1))  *//apply 全量函数 ，process:全量函数，reduce 既有增量也有全量 ，aggregate 既有增量，也有全量* .aggregate(**new** AggregateFunction[TrafficLog,Long,Long] {  **override def** createAccumulator(): Long = 0L   **override def** add(value: TrafficLog, accumulator: Long): Long = {  *//判断当前区域是否在map中* **if**(map.contains(value.areaId)){*//包含当前areaID* **val** bool: Boolean = map.get(value.areaId).get.mightContain(value.car)  **if**(bool){*//布隆过滤器中包含当前车辆数据* accumulator  }**else**{*//布隆过滤器中不包含当前车辆数据* map.get(value.areaId).get.put(value.car)  accumulator +1L  }  }**else**{  map.put(value.areaId,BloomFilter.*create*(Funnels.*stringFunnel*(),100000))  map.get(value.areaId).get.put(value.car)  accumulator +1  }  }   **override def** getResult(accumulator: Long): Long = accumulator   **override def** merge(a: Long, b: Long): Long = a+b  },**new** WindowFunction[Long,String,String,TimeWindow] {  **override def** apply(key: String, window: TimeWindow, input: Iterable[Long], out: Collector[String]): Unit = {  out.collect(**s"起始时间段：$**{window.getStart} **- $**{window.getEnd}**,区域：$**{key}**,车辆数:$**{input.last}**"**)  }  }).print()  env.execute()  }  } |