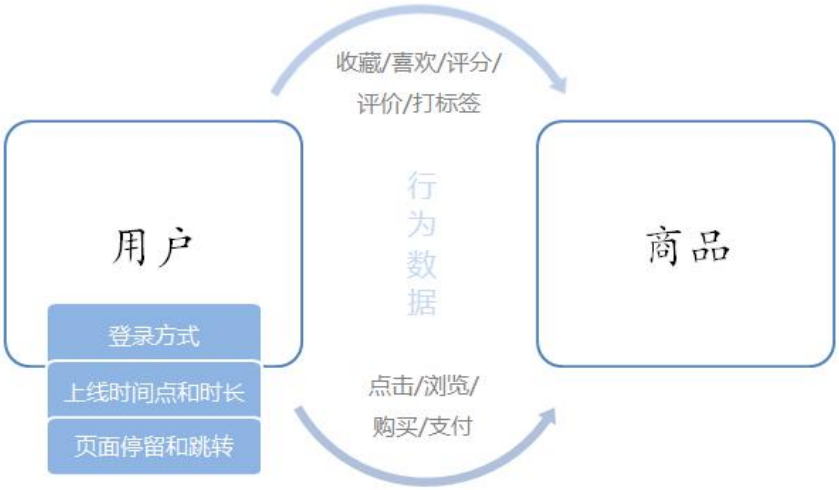
尚硅谷大数据技术之电商用户行为分析

第 1 章 项目整体介绍

1.1 电商的用户行为

电商平台中的用户行为频繁且较复杂，系统上线运行一段时间后，可以收集到大量的用户行为数据，进而利用大数据技术进行深入挖掘和分析，得到感兴趣的商业指标并增强对风险的控制。

电商用户行为数据多样，整体可以分为用户行为习惯数据和业务行为数据两大类。用户的行为习惯数据包括了用户的登录方式、上线的时间点及时长、点击和浏览页面、页面停留时间以及页面跳转等等，我们可以从中进行流量统计和热门商品的统计，也可以深入挖掘用户的特征；这些数据往往可以从 web 服务器日志中直接读取到。而业务行为数据就是用户在电商平台中针对每个业务（通常是某个具体商品）所作的操作，我们一般会在业务系统中相应的位置埋点，然后收集日志进行分析。业务行为数据又可以简单分为两类：一类是能够明显地表现出用户兴趣的行为，比如对商品的收藏、喜欢、评分和评价，我们可以从中对数据进行深入分析，得到用户画像，进而对用户给出个性化的推荐商品列表，这个过程往往会用到机器学习相关的算法；另一类则是常规的业务操作，但需要着重关注一些异常状况以做好风控，比如登录和订单支付。



1.2 项目主要模块

基于对电商用户行为数据的基本分类，我们可以发现主要有以下三个分析方向：

1. 热门统计

利用用户的点击浏览行为，进行流量统计、近期热门商品统计等。

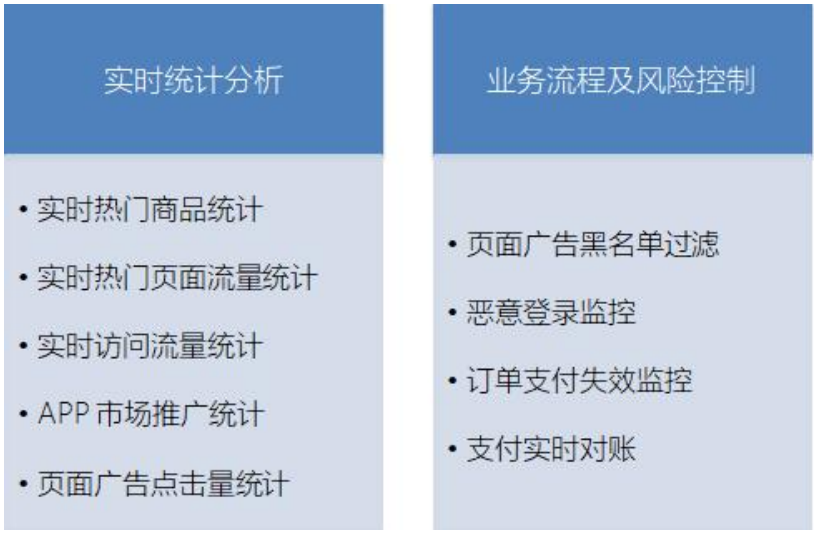
2. 偏好统计

利用用户的偏好行为，比如收藏、喜欢、评分等，进行用户画像分析，给出个性化的商品推荐列表。

3. 风险控制

利用用户的常规业务行为，比如登录、下单、支付等，分析数据，对异常情况进行报警提示。

本项目限于数据，我们只实现热门统计和风险控制中的部分内容，将包括以下五大模块：实时热门商品统计、实时流量统计、市场营销商业指标统计、恶意登录监控和订单支付失效监控，其中细分为以下 9 个具体指标：



由于对实时性要求较高，我们会用 flink 作为数据处理的框架。在项目中，我们将综合运用 flink 的各种 API，基于 EventTime 去处理基本的业务需求，并且灵活地使用底层的 processFunction，基于状态编程和 CEP 去处理更加复杂的情形。

1.3 数据源解析

我们准备了一份淘宝用户行为数据集，保存为 csv 文件。本数据集包含了淘宝上某一天随机一百万用户的所有行为（包括点击、购买、收藏、喜欢）。数据集的每一行表示一条用户行为，由用户 ID、商品 ID、商品类目 ID、行为类型和时间戳组成，并以逗号分隔。关于数据集中每一列的详细描述如下：

字段名	数据类型	说明
userId	Long	加密后的用户 ID
itemId	Long	加密后的商品 ID
categoryId	Int	加密后的商品所属类别 ID
behavior	String	用户行为类型，包括('pv', ''buy, 'cart', 'fav')
timestamp	Long	行为发生的时间戳，单位秒

另外，我们还可以拿到 web 服务器的日志数据，这里以 apache 服务器的一份 log 为例，每一行日志记录了访问者的 IP、userId、访问时间、访问方法以及访问的 url，具体描述如下：

字段名	数据类型	说明
ip	String	访问的 IP
userId	Long	访问的 user ID
eventTime	Long	访问时间
method	String	访问方法 GET/POST/PUT/DELETE
url	String	访问的 url

由于行为数据有限，在实时热门商品统计模块中可以使用 UserBehavior 数据集，而对于恶意登录监控和订单支付失效监控，我们只以示例数据来做演示。

第 2 章 实时热门商品统计

首先要实现的是实时热门商品统计，我们将会基于 UserBehavior 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。

2.1 创建 Maven 项目

2.1.1 项目框架搭建

打开 IDEA，创建一个 maven 项目，命名为 UserBehaviorAnalysis。由于包含了多个模块，我们可以以 UserBehaviorAnalysis 作为父项目，并在其下建一个名为 HotItemsAnalysis 的子项目，用于实时统计热门 top N 商品。

在 UserBehaviorAnalysis 下新建一个 maven module 作为子项目，命名为 HotItemsAnalysis。

父项目只是为了规范化项目结构，方便依赖管理，本身是不需要代码实现的，所以 UserBehaviorAnalysis 下的 src 文件夹可以删掉。

2.1.2 声明项目中工具的版本信息

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，所以应该在最外层的 UserBehaviorAnalysis 中声明所有子模块共用的版本信息。

在 pom.xml 中加入以下配置：

UserBehaviorAnalysis/pom.xml

```
<properties>
    <flink.version>1.10.1</flink.version>
    <scala.binary.version>2.12</scala.binary.version>
```

```
    <kafka.version>2.2.0</kafka.version>
</properties>
```

2.1.3 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，所以我们在 UserBehaviorAnalysis 中引入公有依赖：

UserBehaviorAnalysis/pom.xml

```
<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>
</dependencies>
```

同样，对于 maven 项目的构建，可以引入公有的插件：

```
<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>4.4.0</version>
            <executions>
                <execution>
                    <!-- 声明绑定到 maven 的 compile 阶段 -->
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>3.3.0</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>
```

在 HotItemsAnalysis 子模块中，我们并没有引入更多的依赖，所以不需要改动 pom 文件。

2.1.4 数据准备

在 src/main/目录下，可以看到已有的默认源文件目录是 java，我们可以将其改名为 scala。将数据文件 UserBehavior.csv 复制到资源文件目录 src/main/resources 下，我们将从这里读取数据。

至此，我们的准备工作都已完成，接下来可以写代码了。

2.2 模块代码实现

我们将实现一个“实时热门商品”的需求，可以将“实时热门商品”翻译成程序员更好理解的需求：每隔 5 分钟输出最近一小时内点击量最多的前 N 个商品。将这个需求进行分解我们大概要做这么几件事情：

- 抽取出业务时间戳，告诉 Flink 框架基于业务时间做窗口
- 过滤出点击行为数据
- 按一小时的窗口大小，每 5 分钟统计一次，做滑动窗口聚合（Sliding Window）
- 按每个窗口聚合，输出每个窗口中点击量前 N 名的商品

2.2.1 程序主体

在 src/main/scala 下创建 HotItems.scala 文件，新建一个单例对象。定义样例类 UserBehavior 和 ItemViewCount，在 main 函数中创建 StreamExecutionEnvironment 并做配置，然后从 UserBehavior.csv 文件中读取数据，并包装成 UserBehavior 类型。代码如下：

HotItemsAnalysis/src/main/scala/HotItems.scala

```
case class UserBehavior(userId: Long, itemId: Long, categoryId: Int, behavior: String,
timestamp: Long)
case class ItemViewCount(itemId: Long, windowEnd: Long, count: Long)


object HotItems {
  def main(args: Array[String]): Unit = {
    // 创建一个 StreamExecutionEnvironment
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // 设定 Time 类型为 EventTime
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    // 为了打印到控制台的结果不乱序，我们配置全局的并发为 1，这里改变并发对结果正确性没有影响
    env.setParallelism(1)
    val stream = env
      // 以 window 下为例，需替换成自己的路径
      .readTextFile("YOUR_PATH\\resources\\UserBehavior.csv")
      .map(line => {
        val linearray = line.split(",")
        UserBehavior(linearray(0).toLong, linearray(1).toLong, linearray(2).toInt,
linearray(3), linearray(4).toLong)
      })
      // 指定时间戳和 watermark
      .assignAscendingTimestamps(_.timestamp * 1000)
```

```
    env.execute("Hot Items Job")
  }
```

这里注意，我们需要统计业务时间上的每小时的点击量，所以要基于 EventTime 来处理。那么如果让 Flink 按照我们想要的业务时间来处理呢？这里主要有两件事情要做。

第一件是告诉 Flink 我们现在按照 EventTime 模式进行处理，Flink 默认使用 ProcessingTime 处理，所以我们要显式设置如下：

```
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

第二件事情是指定如何获得业务时间，以及生成 Watermark。Watermark 是用来追踪业务事件的概念，可以理解成 EventTime 世界中的时钟，用来指示当前处理到什么时刻的数据了。由于我们的数据源的数据已经经过整理，没有乱序，即事件的时间戳是单调递增的，所以可以将每条数据的业务时间就当做 Watermark。这里我们用 assignAscendingTimestamps 来实现时间戳的抽取和 Watermark 的生成。

注：真实业务场景一般都是乱序的，所以一般不用 assignAscendingTimestamps，而是使用 BoundedOutOfOrdernessTimestampExtractor。

```
    .assignAscendingTimestamps(_.timestamp * 1000)
```

这样我们就得到了一个带有时间标记的数据流了，后面就能做一些窗口的操作。

2.2.2 过滤出点击事件

在开始窗口操作之前，先回顾下需求“每隔 5 分钟输出过去一小时内点击量最多的前 N 个商品”。由于原始数据中存在点击、购买、收藏、喜欢各种行为的数据，但是我们只需要统计点击量，所以先使用 filter 将点击行为数据过滤出来。

```
    .filter(_.behavior == "pv")
```

2.2.3 设置滑动窗口，统计点击量

由于要每隔 5 分钟统计一次最近一小时每个商品的点击量，所以窗口大小是一小时，每隔 5 分钟滑动一次。即分别要统计[09:00, 10:00), [09:05, 10:05), [09:10, 10:10)…等窗口的商品点击量。是一个常见的滑动窗口需求（Sliding Window）。

```
    .keyBy("itemId")
```

```
    .timeWindow(Time.minutes(60), Time.minutes(5))
    .aggregate(new CountAgg(), new WindowResultFunction());
```

我们使用*.keyBy("itemId")*对商品进行分组，使用*.timeWindow(Time size, Time slide)*对每个商品做滑动窗口（1 小时窗口，5 分钟滑动一次）。然后我们使用*.aggregate(AggregateFunction af, WindowFunction wf)*做增量的聚合操作，它能使用 AggregateFunction 提前聚合掉数据，减少 state 的存储压力。较之*.apply(WindowFunction wf)*会将窗口中的数据都存储下来，最后一起计算要高效地多。这里的 CountAgg 实现了 AggregateFunction 接口，功能是统计窗口中的条数，即遇到一条数据就加一。

```
// COUNT 统计的聚合函数实现，每出现一条记录就加一
class CountAgg extends AggregateFunction[UserBehavior, Long, Long] {
  override def createAccumulator(): Long = 0L
  override def add(userBehavior: UserBehavior, acc: Long): Long = acc + 1
  override def getResult(acc: Long): Long = acc
  override def merge(acc1: Long, acc2: Long): Long = acc1 + acc2
}
```

聚合操作*.aggregate(AggregateFunction af, WindowFunction wf)*的第二个参数 WindowFunction 将每个 key 每个窗口聚合后的结果带上其他信息进行输出。我们这里实现的 WindowResultFunction 将<主键商品 ID，窗口，点击量>封装成了 ItemViewCount 进行输出。

```
// 商品点击量(窗口操作的输出类型)
case class ItemViewCount(itemId: Long, windowEnd: Long, count: Long)
```

代码如下：

```
// 用于输出窗口的结果
class WindowResultFunction extends WindowFunction[Long, ItemViewCount, Tuple,
TimeWindow] {
  override def apply(key: Tuple, window: TimeWindow, aggregateResult: Iterable[Long],
                     collector: Collector[ItemViewCount]) : Unit = {
```

```
    val itemId: Long = key.asInstanceOf[Tuple1[Long]].f0
    val count = aggregateResult.iterator.next
    collector.collect(ItemViewCount(itemId, window.getEnd, count))
  }
}
```

现在我们就得到了每个商品在每个窗口的点击量的数据流。

2.2.4 计算最热门 Top N 商品

为了统计每个窗口下最热门的商品，我们需要再次按窗口进行分组，这里根据 ItemViewCount 中的 windowEnd 进行 keyBy()操作。然后使用 ProcessFunction 实现一个自定义的 TopN 函数 TopNHotItems 来计算点击量排名前 3 名的商品，并将排名结果格式化成字符串，便于后续输出。

```
    .keyBy("windowEnd")
    .process(new TopNHotItems(3));  // 求点击量前 3 名的商品
```

ProcessFunction 是 Flink 提供的一个 low-level API，用于实现更高级的功能。它主要提供了定时器 timer 的功能（支持 EventTime 或 ProcessingTime）。本案例中我们将利用 timer 来判断何时收齐了某个 window 下所有商品的点击量数据。由于 Watermark 的进度是全局的，在 processElement 方法中，每当收到一条数据 ItemViewCount，我们就注册一个 windowEnd+1 的定时器（Flink 框架会自动忽略同一时间的重复注册）。windowEnd+1 的定时器被触发时，意味着收到了 windowEnd+1 的 Watermark，即收齐了该 windowEnd 下的所有商品窗口统计值。我们在 onTimer() 中处理将收集的所有商品及点击量进行排序，选出 TopN，并将排名信息格式化成字符串后进行输出。

这里我们还使用了 ListState<ItemViewCount>来存储收到的每条 ItemViewCount 消息，保证在发生故障时，状态数据的不丢失和一致性。ListState 是 Flink 提供的类似 Java List 接口的 State API，它集成了框架的 checkpoint 机制，自动做到了 exactly-once 的语义保证。

```
// 求某个窗口中前 N 名的热门点击商品，key 为窗口时间戳，输出为 TopN 的结果字符串
class TopNHotItems(topSize: Int) extends KeyedProcessFunction[Tuple, ItemViewCount,
String] {
```

```
    private var itemState : ListState[ItemViewCount] = _


    override def open(parameters: Configuration): Unit = {
      super.open(parameters)
      // 命名状态变量的名字和状态变量的类型
      val itemsStateDesc = new ListStateDescriptor[ItemViewCount]("itemState-state",
classOf[ItemViewCount])
      // 定义状态变量
      itemState = getRuntimeContext.getListState(itemsStateDesc)
    }


    override def processElement(input: ItemViewCount, context:
KeyedProcessFunction[Tuple, ItemViewCount, String]#Context, collector:
Collector[String]): Unit = {
      // 每条数据都保存到状态中
      itemState.add(input)
      // 注册 windowEnd+1 的 EventTime Timer, 当触发时，说明收齐了属于windowEnd 窗口的所
有商品数据
      // 也就是当程序看到windowend + 1 的水位线watermark 时，触发onTimer 回调函数
      context.timerService.registerEventTimeTimer(input.windowEnd + 1)
    }


    override def onTimer(timestamp: Long, ctx: KeyedProcessFunction[Tuple,
ItemViewCount, String]#OnTimerContext, out: Collector[String]): Unit = {
      // 获取收到的所有商品点击量
      val allItems: ListBuffer[ItemViewCount] = ListBuffer()
      import scala.collection.JavaConversions._
      for (item <- itemState.get) {
        allItems += item
```

```
    }
    // 提前清除状态中的数据，释放空间
    itemState.clear()
    // 按照点击量从大到小排序
    val sortedItems = allItems.sortBy(_.count)(Ordering.Long.reverse).take(topSize)
    // 将排名信息格式化成 String，便于打印
    val result: StringBuilder = new StringBuilder
    result.append("====================================\n")
    result.append("时间: ").append(new Timestamp(timestamp - 1)).append("\n")

    for(i <- sortedItems.indices){
      val currentItem: ItemViewCount = sortedItems(i)
      // e.g.  No1:   商品 ID=12224  浏览量=2413
      result.append("No").append(i+1).append(":")
            .append("  商品 ID=").append(currentItem.itemId)
            .append("  浏览量=").append(currentItem.count).append("\n")
    }
    result.append("====================================\n\n")
    // 控制输出频率，模拟实时滚动结果
    Thread.sleep(1000)
    out.collect(result.toString)
  }
}
```

最后我们可以在 main 函数中将结果打印输出到控制台，方便实时观测：

```
.print();
```

至此整个程序代码全部完成，我们直接运行 main 函数，就可以在控制台看到不断输出的各个时间点统计出的热门商品。

2.2.5 完整代码

最终完整代码如下：

```
case class UserBehavior(userId: Long, itemId: Long, categoryId: Int, behavior: String,
timestamp: Long)


case class ItemViewCount(itemId: Long, windowEnd: Long, count: Long)


object HotItems {


  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.setParallelism(1)
    val stream = env
      .readTextFile("YOUR_PATH\\resources\\UserBehavior.csv")
      .map(line => {
        val linearray = line.split(",")
        UserBehavior(linearray(0).toLong, linearray(1).toLong, linearray(2).toInt,
linearray(3), linearray(4).toLong)
      })
      .assignAscendingTimestamps(_.timestamp * 1000)
      .filter(_.behavior=="pv")
      .keyBy("itemId")
      .timeWindow(Time.minutes(60), Time.minutes(5))
      .aggregate(new CountAgg(), new WindowResultFunction())
      .keyBy(1)
      .process(new TopNHotItems(3))
      .print()
```

```
    env.execute("Hot Items Job")
  }

  // COUNT 统计的聚合函数实现，每出现一条记录加一
  class CountAgg extends AggregateFunction[UserBehavior, Long, Long] {
    override def createAccumulator(): Long = 0L
    override def add(userBehavior: UserBehavior, acc: Long): Long = acc + 1
    override def getResult(acc: Long): Long = acc
    override def merge(acc1: Long, acc2: Long): Long = acc1 + acc2
  }
  // 用于输出窗口的结果
  class WindowResultFunction extends WindowFunction[Long, ItemViewCount, Tuple,
TimeWindow] {
    override def apply(key: Tuple, window: TimeWindow, aggregateResult: Iterable[Long],
                       collector: Collector[ItemViewCount]) : Unit = {
      val itemId: Long = key.asInstanceOf[Tuple1[Long]].f0
      val count = aggregateResult.iterator.next
      collector.collect(ItemViewCount(itemId, window.getEnd, count))
    }
  }

// 求某个窗口中前 N 名的热门点击商品，key 为窗口时间戳，输出为 TopN 的结果字符串
  class TopNHotItems(topSize: Int) extends KeyedProcessFunction[Tuple, ItemViewCount,
String] {
    private var itemState : ListState[ItemViewCount] = _

    override def open(parameters: Configuration): Unit = {
      super.open(parameters)
      // 命名状态变量的名字和状态变量的类型
```

```
      val itemsStateDesc = new ListStateDescriptor[ItemViewCount]("itemState-state",
classOf[ItemViewCount])
      // 从运行时上下文中获取状态并赋值
      itemState = getRuntimeContext.getListState(itemsStateDesc)
    }

    override def processElement(input: ItemViewCount, context:
KeyedProcessFunction[Tuple, ItemViewCount, String]#Context, collector:
Collector[String]): Unit = {
      // 每条数据都保存到状态中
      itemState.add(input)
      // 注册 windowEnd+1 的 EventTime Timer, 当触发时，说明收齐了属于windowEnd 窗口的所
有商品数据
      // 也就是当程序看到windowend + 1 的水位线watermark 时，触发onTimer 回调函数
      context.timerService.registerEventTimeTimer(input.windowEnd + 1)
    }

    override def onTimer(timestamp: Long, ctx: KeyedProcessFunction[Tuple,
ItemViewCount, String]#OnTimerContext, out: Collector[String]): Unit = {
      // 获取收到的所有商品点击量
      val allItems: ListBuffer[ItemViewCount] = ListBuffer()
      import scala.collection.JavaConversions._
      for (item <- itemState.get) {
        allItems += item
      }
      // 提前清除状态中的数据，释放空间
      itemState.clear()
      // 按照点击量从大到小排序
      val sortedItems = allItems.sortBy(_.count)(Ordering.Long.reverse).take(topSize)
```

```
    // 将排名信息格式化成 String, 便于打印
    val result: StringBuilder = new StringBuilder
    result.append("====================================\n")
    result.append("时间: ").append(new Timestamp(timestamp - 1)).append("\n")

    for(i <- sortedItems.indices){
      val currentItem: ItemViewCount = sortedItems(i)
      // e.g.  No1:   商品 ID=12224  浏览量=2413
      result.append("No").append(i+1).append(":")
            .append("  商品 ID=").append(currentItem.itemId)
            .append("  浏览量=").append(currentItem.count).append("\n")
    }
    result.append("====================================\n\n")
    // 控制输出频率，模拟实时滚动结果
    Thread.sleep(1000)
    out.collect(result.toString)
  }
 }
}
```

2.2.6 更换 Kafka 作为数据源

实际生产环境中，我们的数据流往往是从 Kafka 获取到的。如果要让代码更贴近生产实际，我们只需将 source 更换为 Kafka 即可：

```
val properties = new Properties()
properties.setProperty("bootstrap.servers", "localhost:9092")
properties.setProperty("group.id", "consumer-group")
properties.setProperty("key.deserializer",
                       "org.apache.kafka.common.serialization.StringDeserializer")
properties.setProperty("value.deserializer",
```

```
                    "org.apache.kafka.common.serialization.StringDeserializer")
properties.setProperty("auto.offset.reset", "latest")

val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
env.setParallelism(1)

val stream = env
  .addSource(new FlinkKafkaConsumer[String]("hotitems", new SimpleStringSchema(),
properties))
```

当然，根据实际的需要，我们还可以将 Sink 指定为 Kafka、ES、Redis 或其它存储，这里就不一一展开实现了。

第 3 章 实时流量统计

3.1 模块创建和数据准备

在 UserBehaviorAnalysis 下新建一个 maven module 作为子项目，命名为 NetworkFlowAnalysis。在这个子模块中，我们同样并没有引入更多的依赖，所以也不需要改动 pom 文件。

在 src/main/目录下，将默认源文件目录 java 改名为 scala。将 apache 服务器的日志文件 apache.log 复制到资源文件目录 src/main/resources 下，我们将从这里读取数据。

当然，我们也可以仍然用 UserBehavior.csv 作为数据源，这时我们分析的就不是每一次对服务器的访问请求了，而是具体的页面浏览（“pv”）操作。

3.2 基于服务器 log 的热门页面浏览量统计

我们现在要实现的模块是 “实时流量统计”。对于一个电商平台而言，用户登录的入口流量、不同页面的访问流量都是值得分析的重要数据，而这些数据，可以简单地从 web 服务器的日志中提取出来。

我们在这里先实现“热门页面浏览数”的统计，也就是读取服务器日志中的每一行 log，统计在一段时间内用户访问每一个 url 的次数，然后排序输出显示。

具体做法为：每隔 5 秒，输出最近 10 分钟内访问量最多的前 N 个 URL。可以看出，这个需求与之前“实时热门商品统计”非常类似，所以我们完全可以借鉴此前的代码。

在 src/main/scala 下创建 NetworkFlow.scala 文件，新建一个单例对象。定义样例类 ApacheLogEvent，这是输入的日志数据流；另外还有 UrlViewCount，这是窗口操作统计的输出数据类型。在 main 函数中创建 StreamExecutionEnvironment 并做配置，然后从 apache.log 文件中读取数据，并包装成 ApacheLogEvent 类型。

需要注意的是，原始日志中的时间是“dd/MM/yyyy:HH:mm:ss”的形式，需要定义一个 DateTimeFormat 将其转换为我们需要的时间戳格式：

```
.map(line => {
    val linearray = line.split(" ")
    val sdf = new SimpleDateFormat("dd/MM/yyyy:HH:mm:ss")
    val timestamp = sdf.parse(linearray(3)).getTime
    ApacheLogEvent(linearray(0), linearray(2), timestamp,
                   linearray(5), linearray(6))
})
```

完整代码如下：

NetworkFlowAnalysis/src/main/scala/NetworkFlow.scala

```
case class ApacheLogEvent(ip: String, userId: String, eventTime: Long, method: String,
url: String)
case class UrlViewCount(url: String, windowEnd: Long, count: Long)


object NetworkFlow{


  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
```

```
    env.setParallelism(1)
    val stream = env
      // 以window下为例，需替换成自己的路径
      .readTextFile("YOUR_PATH\\resources\\apache.log")
      .map(line => {
        val linearray = line.split(" ")
        val simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy:HH:mm:ss")
        val timestamp = simpleDateFormat.parse(linearray(3)).getTime
        ApacheLogEvent(linearray(0), linearray(2), timestamp, linearray(5),
linearray(6))
      })
      .assignTimestampsAndWatermarks(new
          BoundedOutOfOrdernessTimestampExtractor[ApacheLogEvent]
          (Time.milliseconds(1000)) {
              override def extractTimestamp(t: ApacheLogEvent): Long = {
                  t.eventTime
          }
      })
      .filter( data => {
          val pattern = "^((?!\\.(css|js)$).)*$".r
          (pattern findFirstIn data.url).nonEmpty
      } )
      .keyBy("url")
      .timeWindow(Time.minutes(10), Time.seconds(5))
      .aggregate(new CountAgg(), new WindowResultFunction())
      .keyBy(1)
      .process(new TopNHotUrls(5))
      .print()
    env.execute("Network Flow Job")
```

```
  }


  class CountAgg extends AggregateFunction[ApacheLogEvent, Long, Long] {
    override def createAccumulator(): Long = 0L
    override def add(apacheLogEvent: ApacheLogEvent, acc: Long): Long = acc + 1
    override def getResult(acc: Long): Long = acc
    override def merge(acc1: Long, acc2: Long): Long = acc1 + acc2
  }


  class WindowResultFunction extends WindowFunction[Long, UrlViewCount, Tuple,
TimeWindow] {
    override def apply(key: Tuple, window: TimeWindow, aggregateResult: Iterable[Long],
collector: Collector[UrlViewCount]) : Unit = {
      val url: String = key.asInstanceOf[Tuple1[String]].f0
      val count = aggregateResult.iterator.next
      collector.collect(UrlViewCount(url, window.getEnd, count))
    }
  }


  class TopNHotUrls(topsize: Int) extends KeyedProcessFunction[Tuple, UrlViewCount,
String] {
    private var urlState : ListState[UrlViewCount] = _


    override def open(parameters: Configuration): Unit = {
      super.open(parameters)
      val urlStateDesc = new ListStateDescriptor[UrlViewCount]("urlState-state",
classOf[UrlViewCount])
      urlState = getRuntimeContext.getListState(urlStateDesc)
    }
```

```
    override def processElement(input: UrlViewCount, context:
KeyedProcessFunction[Tuple, UrlViewCount, String]#Context, collector:
Collector[String]): Unit = {
      // 每条数据都保存到状态中
      urlState.add(input)
      context.timerService.registerEventTimeTimer(input.windowEnd + 1)
    }

    override def onTimer(timestamp: Long, ctx: KeyedProcessFunction[Tuple, UrlViewCount,
String]#OnTimerContext, out: Collector[String]): Unit = {
      // 获取收到的所有 URL 访问量
      val allUrlViews: ListBuffer[UrlViewCount] = ListBuffer()
      import scala.collection.JavaConversions._
      for (urlView <- urlState.get) {
        allUrlViews += urlView
      }
      // 提前清除状态中的数据，释放空间
      urlState.clear()
      // 按照访问量从大到小排序
      val sortedUrlViews = allUrlViews.sortBy(_.count)(Ordering.Long.reverse)
                                      .take(topSize)
      // 将排名信息格式化成 String, 便于打印
      var result: StringBuilder = new StringBuilder
      result.append("====================================\n")
      result.append("时间: ").append(new Timestamp(timestamp - 1)).append("\n")

      for (i <- sortedUrlViews.indices) {
        val currentUrlView: UrlViewCount = sortedUrlViews(i)
```

```
        // e.g.  No1:  URL=/blog/tags/firefox?flav=rss20  流量=55
        result.append("No").append(i+1).append(":")
                .append("  URL=").append(currentUrlView.url)
                .append("  流量=").append(currentUrlView.count).append("\n")
      }
      result.append("===================================\n\n")
      // 控制输出频率，模拟实时滚动结果
      Thread.sleep(1000)
      out.collect(result.toString)
    }
  }
}
```

3.3 基于埋点日志数据的网络流量统计

我们发现，从 web 服务器 log 中得到的 url，往往更多的是请求某个资源地址（/*.js、/*.css），如果要针对页面进行统计往往还需要进行过滤。而在实际电商应用中，相比每个单独页面的访问量，我们可能更加关心整个电商网站的网络流量。这个指标，除了合并之前每个页面的统计结果之外，还可以通过统计埋点日志数据中的“pv”行为来得到。

3.3.1 网站总浏览量（PV）的统计

衡量网站流量一个最简单的指标，就是网站的页面浏览量（Page View，PV）。用户每次打开一个页面便记录 1 次 PV，多次打开同一页面则浏览量累计。一般来说，PV 与来访者的数量成正比，但是 PV 并不直接决定页面的真实来访者数量，如同一个来访者通过不断的刷新页面，也可以制造出非常高的 PV。

我们知道，用户浏览页面时，会从浏览器向网络服务器发出一个请求（Request），网络服务器接到这个请求后，会将该请求对应的一个网页（Page）发送给浏览器，从而产生了一个 PV。所以我们的统计方法，可以是从 web 服务器的日志中去提取对应的页面访问然后统计，就向上一节中的做法一样；也可以直接从埋点日志中提取用户发来的页面请求，从而统计出总浏览量。

所以，接下来我们用 UserBehavior.csv 作为数据源，实现一个网站总浏览量的统计。我们可以设置滚动时间窗口，实时统计每小时内的网站 PV。

在 src/main/scala 下创建 PageView.scala 文件，具体代码如下：

NetworkFlowAnalysis/src/main/scala/PageView.scala

```
case class UserBehavior(userId: Long, itemId: Long, categoryId: Int, behavior: String,
timestamp: Long)


object PageView {
  def main(args: Array[String]): Unit = {
    val resourcesPath = getClass.getResource("/UserBehaviorTest.csv")
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.setParallelism(1)


    val stream = env.readTextFile(resourcesPath.getPath)
      .map(data => {
        val dataArray = data.split(",")
        UserBehavior(dataArray(0).toLong, dataArray(1).toLong, dataArray(2).toInt,
dataArray(3), dataArray(4).toLong)
      })
      .assignAscendingTimestamps(_.timestamp * 1000)
      .filter(_.behavior == "pv")
      .map(x => ("pv", 1))
      .keyBy(_._1)
      .timeWindow(Time.seconds(60 * 60))
      .sum(1)
      .print()


    env.execute("Page View Job")
  }
}
```

3.3.2 网站独立访客数（UV）的统计

在上节的例子中，我们统计的是所有用户对页面的所有浏览行为，也就是说，同一用户的浏览行为会被重复统计。而在实际应用中，我们往往还会关注，在一段时间内到底有多少不同的用户访问了网站。

另外一个统计流量的重要指标是网站的独立访客数（Unique Visitor，UV）。UV 指的是一段时间（比如一小时）内访问网站的总人数，1 天内同一访客的多次访问只记录为一个访客。通过 IP 和 cookie 一般是判断 UV 值的两种方式。当客户端第一次访问某个网站服务器的时候，网站服务器会给这个客户端的电脑发出一个 Cookie，通常放在这个客户端电脑的 C 盘当中。在这个 Cookie 中会分配一个独一无二的编号，这其中会记录一些访问服务器的信息，如访问时间，访问了哪些页面等等。当你下次再访问这个服务器的时候，服务器就可以直接从你的电脑中找到上一次放进去的 Cookie 文件，并且对其进行一些更新，但那个独一无二的编号是不会变的。

当然，对于 UserBehavior 数据源来说，我们直接可以根据 userId 来区分不同的用户。

在 src/main/scala 下创建 UniqueVisitor.scala 文件，具体代码如下：

NetworkFlowAnalysis/src/main/scala/UniqueVisitor.scala

```
case class UvCount(windowEnd: Long, count: Long)


object UniqueVisitor {
  def main(args: Array[String]): Unit = {
    val resourcesPath = getClass.getResource("/UserBehaviorTest.csv")
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.setParallelism(1)
    val stream = env
      .readTextFile(resourcesPath.getPath)
      .map(line => {
        val linearray = line.split(",")
        UserBehavior(linearray(0).toLong, linearray(1).toLong, linearray(2).toInt,
linearray(3), linearray(4).toLong)
```

```
      })
      .assignAscendingTimestamps(_.timestamp * 1000)
      .filter(_.behavior == "pv")
      .timeWindowAll(Time.seconds(60 * 60))
      .apply(new UvCountByWindow())
      .print()

    env.execute("Unique Visitor Job")
  }
}

class UvCountByWindow extends AllWindowFunction[UserBehavior, UvCount, TimeWindow] {
  override def apply(window: TimeWindow,
                     input: Iterable[UserBehavior],
                     out: Collector[UvCount]): Unit = {

    val s: collection.mutable.Set[Long] = collection.mutable.Set()
    var idSet = Set[Long]()

    for ( userBehavior <- input) {
      idSet += userBehavior.userId
    }

    out.collect(UvCount(window.getEnd, idSet.size))
  }
}
```

3.3.3 使用布隆过滤器的 UV 统计

在上节的例子中，我们把所有数据的 userId 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？比如上亿级的用户，要去重计算 UV。

如果放到 redis 中，亿级的用户 id（每个 20 字节左右的话）可能需要几 G 甚至几十 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储用户 ID 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）就可以表示一个用户的状态。这个思想的具体实现就是布隆过滤器（Bloom Filter）。

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，但是缺点是其返回的结果是概率性的，而不是确切的。

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。

接下来我们就来具体实现一下。

注意这里我们用到了 redis 连接存取数据，所以需要加入 redis 客户端的依赖：

```
<dependencies>
    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>2.8.1</version>
    </dependency>
</dependencies>
```

在 src/main/scala 下创建 UniqueVisitor.scala 文件，具体代码如下：

NetworkFlowAnalysis/src/main/scala/UvWithBloom.scala

```
object UvWithBloomFilter {

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

    val env = StreamExecutionEnvironment.getExecutionEnvironment

    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    env.setParallelism(1)


    val resourcesPath = getClass.getResource("/UserBehaviorTest.csv")

    val stream = env

      .readTextFile(resourcesPath.getPath)

      .map(data => {

      val dataArray = data.split(",")

      UserBehavior(dataArray(0).toLong, dataArray(1).toLong, dataArray(2).toInt,
dataArray(3), dataArray(4).toLong)

    })

      .assignAscendingTimestamps(_.timestamp * 1000)

      .filter(_.behavior == "pv")

      .map(data => ("dummyKey", data.userId))

      .keyBy(_._1)

      .timeWindow(Time.seconds(60 * 60))

      .trigger(new MyTrigger())    // 自定义窗口触发规则

      .process(new UvCountWithBloom())    // 自定义窗口处理规则


    stream.print()

    env.execute("Unique Visitor with bloom Job")

  }

}


// 自定义触发器

class MyTrigger() extends Trigger[(String, Long), TimeWindow] {
```

```
  override def onEventTime(time: Long, window: TimeWindow, ctx: Trigger.TriggerContext):
TriggerResult = {
    TriggerResult.CONTINUE
  }

  override def onProcessingTime(time: Long, window: TimeWindow, ctx:
Trigger.TriggerContext): TriggerResult = {
    TriggerResult.CONTINUE
  }

  override def clear(window: TimeWindow, ctx: Trigger.TriggerContext): Unit = {
  }

  override def onElement(element: (String, Long), timestamp: Long, window: TimeWindow,
ctx: Trigger.TriggerContext): TriggerResult = {
    // 每来一条数据，就触发窗口操作并清空
    TriggerResult.FIRE_AND_PURGE
  }
}

// 自定义窗口处理函数
class UvCountWithBloom() extends ProcessWindowFunction[(String, Long), UvCount, String,
TimeWindow] {
  // 创建 redis 连接
  lazy val jedis = new Jedis("localhost", 6379)
  lazy val bloom = new Bloom(1 << 29)

  override def process(key: String, context: Context, elements: Iterable[(String, Long)],
```

```
out: Collector[UvCount]): Unit = {
    val storeKey = context.window.getEnd.toString
    var count = 0L
    if (jedis.hget("count", storeKey) != null) {
      count = jedis.hget("count", storeKey).toLong
    }

    val userId = elements.last._2.toString
    val offset = bloom.hash(userId, 61)

    val isExist = jedis.getbit(storeKey, offset)
    if (!isExist) {
        jedis.setbit(storeKey, offset, true)
        jedis.hset("count", storeKey, (count + 1).toString)
        out.collect(UvCount(storeKey.toLong, count + 1))
    } else {
        out.collect(UvCount(storeKey.toLong, count))
    }
  }
}

// 定义一个布隆过滤器
class Bloom(size: Long) extends Serializable {
  private val cap = size

  def hash(value: String, seed: Int): Long = {
    var result = 0
    for (i <- 0 until value.length) {
      // 最简单的hash算法，每一位字符的ascii码值，乘以seed之后，做叠加
```

```
      result = result * seed + value.charAt(i)
    }
    (cap - 1) & result
  }
}
```

第 4 章 市场营销商业指标统计分析

4.1 模块创建和数据准备

继续在 UserBehaviorAnalysis 下新建一个 maven module 作为子项目，命名为 MarketAnalysis。

这个模块中我们没有现成的数据，所以会用自定义的测试源来产生测试数据流，或者直接用生成测试数据文件。

4.2 APP 市场推广统计

随着智能手机的普及，在如今的电商网站中已经有越来越多的用户来自移动端，相比起传统浏览器的登录方式，手机 APP 成为了更多用户访问电商网站的首选。对于电商企业来说，一般会通过各种不同的渠道对自己的 APP 进行市场推广，而这些渠道的统计数据（比如，不同网站上广告链接的点击量、APP 下载量）就成了市场营销的重要商业指标。

首先我们考察分渠道的市场推广统计。在 src/main/scala 下创建 AppMarketingByChannel.scala 文件。由于没有现成的数据，所以我们需要自定义一个测试源来生成用户行为的事件流。

4.2.1 自定义测试数据源

定义一个源数据的样例类 MarketingUserBehavior，再定义一个 SourceFunction，用于产生用户行为源数据，命名为 SimulatedEventSource：

```
case class MarketingUserBehavior(userId: Long, behavior: String, channel: String,
timestamp: Long)
```

```
class SimulatedEventSource extends RichParallelSourceFunction[MarketingUserBehavior]
{

  var running = true
  val channelSet: Seq[String] = Seq("AppStore", "XiaomiStore", "HuaweiStore", "weibo",
"wechat", "tieba")
  val behaviorTypes: Seq[String] = Seq("BROWSE", "CLICK", "PURCHASE", "UNINSTALL")
  val rand: Random = Random

  override def run(ctx: SourceContext[MarketingUserBehavior]): Unit = {
    val maxElements = Long.MaxValue
    var count = 0L

    while (running && count < maxElements) {
      val id = UUID.randomUUID().toString.toLong
      val behaviorType = behaviorTypes(rand.nextInt(behaviorTypes.size))
      val channel = channelSet(rand.nextInt(channelSet.size))
      val ts = System.currentTimeMillis()

      ctx.collectWithTimestamp(MarketingUserBehavior(id, behaviorType, channel, ts),
ts)
      count += 1
      TimeUnit.MILLISECONDS.sleep(5L)
    }
  }
```

```
  override def cancel(): Unit = running = false
}
```

4.2.2 分渠道统计

另外定义一个窗口处理的输出结果样例类 MarketingViewCount，并自定义 ProcessWindowFunction 进行处理，代码如下：

```
case class MarketingCountView(windowStart: Long, windowEnd: Long, channel: String,
behavior: String, count: Long)


object AppMarketingByChannel {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.setParallelism(1)
    val stream: DataStream[MarketingUserBehavior] = env.addSource(new
SimulatedEventSource)
      .assignAscendingTimestamps(_.timestamp)


    stream
      .filter(_.behavior != "UNINSTALL")
      .map(data => {
        ((data.channel, data.behavior), 1L)
      })
      .keyBy(_._1)
      .timeWindow(Time.hours(1), Time.seconds(1))
      .process(new MarketingCountByChannel())
      .print()
```

```
    env.execute(getClass.getSimpleName)
  }
}
    class MarketingCountByChannel() extends ProcessWindowFunction[((String, String),
Long), MarketingViewCount, (String, String), TimeWindow] {
  override def process(key: (String, String),
                       context: Context,
                       elements: Iterable[((String, String), Long)],
                       out: Collector[MarketingViewCount]): Unit = {
    val startTs = context.window.getStart
    val endTs = context.window.getEnd
    val channel = key._1
    val behaviorType = key._2
    val count = elements.size
    out.collect( MarketingViewCount(formatTs(startTs), formatTs(endTs), channel,
behaviorType, count) )
  }
  private def formatTs (ts: Long) = {
    val df = new SimpleDateFormat ("yyyy/MM/dd-HH:mm:ss")
    df.format (new Date (ts) )
  }
}
```

4.2.3 不分渠道（总量）统计

同样我们还可以考察不分渠道的市场推广统计，这样得到的就是所有渠道推广的总量。在 src/main/scala 下创建 AppMarketingStatistics.scala 文件，代码如下：

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

```
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.setParallelism(1)
    val stream: DataStream[MarketingUserBehavior] = env.addSource(new
SimulatedEventSource)
      .assignAscendingTimestamps(_.timestamp)

    stream
      .filter(_.behavior != "UNINSTALL")
      .map(data => {
        ("dummyKey", 1L)
      })
      .keyBy(_._1)
      .timeWindow(Time.hours(1), Time.seconds(1))
      .process(new MarketingCountTotal())
      .print()

    env.execute(getClass.getSimpleName)
  }
}

class MarketingCountTotal() extends ProcessWindowFunction[(String, Long),
MarketingViewCount, String, TimeWindow]{
  override def process(key: String, context: Context, elements: Iterable[(String, Long)],
out: Collector[MarketingViewCount]): Unit = {
    val startTs = context.window.getStart
    val endTs = context.window.getEnd
    val count = elements.size
    out.collect( MarketingViewCount(formatTs(startTs), formatTs(endTs), "total",
```

```
"total", count) )
  }
  private def formatTs (ts: Long) = {
    val df = new SimpleDateFormat ("yyyy/MM/dd-HH:mm:ss")
    df.format (new Date (ts) )
  }
}
```

4.3 页面广告分析

电商网站的市场营销商业指标中，除了自身的 APP 推广，还会考虑到页面上的广告投放（包括自己经营的产品和其它网站的广告）。所以广告相关的统计分析，也是市场营销的重要指标。

对于广告的统计，最简单也最重要的就是页面广告的点击量，网站往往需要根据广告点击量来制定定价策略和调整推广方式，而且也可以借此收集用户的偏好信息。更加具体的应用是，我们可以根据用户的地理位置进行划分，从而总结出不同省份用户对不同广告的偏好，这样更有助于广告的精准投放。

4.3.1 页面广告点击量统计

接下来我们就进行页面广告按照省份划分的点击量的统计。在 src/main/scala 下创建 AdStatisticsByGeo.scala 文件。同样由于没有现成的数据，我们定义一些测试数据，放在 AdClickLog.csv 中，用来生成用户点击广告行为的事件流。

在代码中我们首先定义源数据的样例类 AdClickLog，以及输出统计数据的样例类 CountByProvince。主函数中先以 province 进行 keyBy，然后开一小时的时间窗口，滑动距离为 5 秒，统计窗口内的点击事件数量。具体代码实现如下：

```
case class AdClickLog(userId: Long, adId: Long, province: String, city: String, timestamp:
Long)

case class CountByProvince(windowEnd: String, province: String, count: Long)
```

```
object AdStatisticsByGeo {

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

    val env = StreamExecutionEnvironment.getExecutionEnvironment

    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    env.setParallelism(1)

    val adLogStream: DataStream[AdClickLog] =
env.readTextFile("YOURPATH\\resources\\AdClickLog.csv")

      .map(data => {

        val dataArray = data.split(",")

        AdClickLog(dataArray(0).toLong, dataArray(1).toLong, dataArray(2),
dataArray(3), dataArray(4).toLong)

      })

      .assignAscendingTimestamps(_.timestamp * 1000L)


    val adCountStream = adLogStream

      .keyBy(_.province)

      .timeWindow(Time.minutes(60), Time.seconds(5))

      .aggregate(new CountAgg(), new CountResult())

      .print()


    env.execute("ad statistics job")

  }

}


class CountAgg() extends AggregateFunction[AdClickLog, Long, Long]{

  override def add(value: AdClickLog, accumulator: Long): Long = accumulator + 1L

  override def createAccumulator(): Long = 0L

  override def getResult(accumulator: Long): Long = accumulator

  override def merge(a: Long, b: Long): Long = a + b
```

```
}


class CountResult() extends WindowFunction[Long, CountByProvince, String, TimeWindow]{
  override def apply(key: String, window: TimeWindow, input: Iterable[Long], out:
Collector[CountByProvince]): Unit = {
    out.collect(CountByProvince(formatTs(window.getEnd), key, input.iterator.next()))
  }
  private def formatTs (ts: Long) = {
    val df = new SimpleDateFormat ("yyyy/MM/dd-HH:mm:ss")
    df.format (new Date (ts) )
  }
}
```

4.3.2 黑名单过滤

上节我们进行的点击量统计，同一用户的重复点击是会叠加计算的。在实际场景中，同一用户确实可能反复点开同一个广告，这也说明了用户对广告更大的兴趣；但是如果用户在一段时间非常频繁地点击广告，这显然不是一个正常行为，有刷点击量的嫌疑。所以我们可以对一段时间内（比如一天内）的用户点击行为进行约束，如果对同一个广告点击超过一定限额（比如 100 次），应该把该用户加入黑名单并报警，此后其点击行为不应该再统计。

具体代码实现如下：

```
case class AdClickLog(userId: Long, adId: Long, province: String, city: String, timestamp:
Long)
case class CountByProvince(windowEnd: String, province: String, count: Long)
case class BlackListWarning(userId: Long, adId: Long, msg: String)


object AdStatisticsByGeo {


  val blackListOutputTag = new OutputTag[BlackListWarning]("blacklist")
```

```
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.setParallelism(1)
    val adLogStream: DataStream[AdClickLog] =
env.readTextFile("D:\\Projects\\BigData\\UserBehaviorAnalysis\\MarketAnalysis\\src\
\main\\resources\\AdClickLog.csv")
      .map(data => {
        val dataArray = data.split(",")
        AdClickLog(dataArray(0).toLong, dataArray(1).toLong, dataArray(2),
dataArray(3), dataArray(4).toLong)
      })
      .assignAscendingTimestamps(_.timestamp * 1000L)

    val filterBlackListStream = adLogStream
      .keyBy(logData => (logData.userId, logData.adId))
      .process(new FilterBlackListUser(100))

    val adCountStream = filterBlackListStream
      .keyBy(_.province)
      .timeWindow(Time.minutes(60), Time.seconds(5))
      .aggregate(new countAgg(), new countResult())
      .print()

    filterBlackListStream.getSideOutput(blackListOutputTag)
      .print("black list")

    env.execute("ad statistics job")
  }
```

```
  class FilterBlackListUser(maxCount: Long) extends KeyedProcessFunction[(Long, Long),
AdClickLog, AdClickLog] {
    // 保存当前用户对当前广告的点击量
    lazy val countState: ValueState[Long] = getRuntimeContext.getState(new
ValueStateDescriptor[Long]("count-state", classOf[Long]))
    // 标记当前（用户，广告）作为 key 是否第一次发送到黑名单
    lazy val firstSent: ValueState[Boolean] = getRuntimeContext.getState(new
ValueStateDescriptor[Boolean]("firstsent-state", classOf[Boolean]))
    // 保存定时器触发的时间戳，届时清空重置状态
    lazy val resetTime: ValueState[Long] = getRuntimeContext.getState(new
ValueStateDescriptor[Long]("resettime-state", classOf[Long]))

    override def processElement(value: AdClickLog, ctx: KeyedProcessFunction[(Long,
Long), AdClickLog, AdClickLog]#Context, out: Collector[AdClickLog]): Unit = {
      val curCount = countState.value()
      // 如果是第一次处理，注册一个定时器，每天 00：00 触发清除
      if( curCount == 0 ){
        val ts = (ctx.timerService().currentProcessingTime() / (24*60*60*1000) + 1) *
(24*60*60*1000)
        resetTime.update(ts)
        ctx.timerService().registerProcessingTimeTimer(ts)
      }
      // 如果计数已经超过上限，则加入黑名单，用侧输出流输出报警信息
      if( curCount > maxCount ){
        if( !firstSent.value() ){
          firstSent.update(true)
          ctx.output( blackListOutputTag, BlackListWarning(value.userId, value.adId,
"Click over " + maxCount + " times today.") )
```

```
        }
        return
      }
      // 点击计数加1
      countState.update(curCount + 1)
      out.collect( value )
    }

    override def onTimer(timestamp: Long, ctx: KeyedProcessFunction[(Long, Long),
AdClickLog, AdClickLog]#OnTimerContext, out: Collector[AdClickLog]): Unit = {
      if( timestamp == resetTime.value() ){
        firstSent.clear()
        countState.clear()
      }
    }
  }
}


class countAgg() extends AggregateFunction[AdClickLog, Long, Long] {
  override def add(value: AdClickLog, accumulator: Long): Long = accumulator + 1L
  override def createAccumulator(): Long = 0L
  override def getResult(accumulator: Long): Long = accumulator
  override def merge(a: Long, b: Long): Long = a + b
}

class countResult() extends WindowFunction[Long, CountByProvince, String, TimeWindow]
{
  override def apply(key: String, window: TimeWindow, input: Iterable[Long], out:
```

```
Collector[CountByProvince]): Unit = {
    out.collect(CountByProvince(formatTs(window.getEnd), key, input.iterator.next()))
  }

  private def formatTs(ts: Long) = {
    val df = new SimpleDateFormat("yyyy/MM/dd-HH:mm:ss")
    df.format(new Date(ts))
  }
}
```

第 5 章 恶意登录监控

5.1 模块创建和数据准备

继续在 UserBehaviorAnalysis 下新建一个 maven module 作为子项目，命名为 LoginFailDetect。在这个子模块中，我们将会用到 flink 的 CEP 库来实现事件流的模式匹配，所以需要在 pom 文件中引入 CEP 的相关依赖：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-cep-scala_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
</dependency>
```

同样，在 src/main/目录下，将默认源文件目录 java 改名为 scala。

5.2 代码实现

对于网站而言，用户登录并不是频繁的业务操作。如果一个用户短时间内频繁登录失败，就有可能是出现了程序的恶意攻击，比如密码暴力破解。因此我们考虑，应该对用户的登录失败动作进行统计，具体来说，如果同一用户（可以是不同 IP）在 2 秒之内连续两次登录失败，就认为存在恶意登录的风险，输出相关的信息进行

报警提示。这是电商网站、也是几乎所有网站风控的基本一环。

5.2.1 状态编程

由于同样引入了时间，我们可以想到，最简单的方法其实与之前的热门统计类似，只需要按照用户 ID 分流，然后遇到登录失败的事件时将其保存在 ListState 中，然后设置一个定时器，2 秒后触发。定时器触发时检查状态中的登录失败事件个数，如果大于等于 2，那么就输出报警信息。

在 src/main/scala 下创建 LoginFail.scala 文件，新建一个单例对象。定义样例类 LoginEvent，这是输入的登录事件流。登录数据本应该从 UserBehavior 日志里提取，由于 UserBehavior.csv 中没有做相关埋点，我们从另一个文件 LoginLog.csv 中读取登录数据。

代码如下：

LoginFailDetect/src/main/scala/LoginFail.scala

```
case class LoginEvent(userId: Long, ip: String, eventType: String, eventTime: Long)


object LoginFail {


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


    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.setParallelism(1)


    val loginEventStream = env.readTextFile("YOUR_PATH\\resources\\LoginLog.csv")
      .map( data => {
        val dataArray = data.split(",")
        LoginEvent(dataArray(0).toLong, dataArray(1), dataArray(2),
dataArray(3).toLong)
      })
      .assignTimestampsAndWatermarks(new
                BoundedOutOfOrdernessTimestampExtractor[ApacheLogEvent]
```

```
            (Time.milliseconds(3000)) {
          override def extractTimestamp(element: ApacheLogEvent): Long ={
            element.eventTime * 1000L
          }
      })
    .keyBy(_.userId)
    .process(new MatchFunction())
    .print()

  env.execute("Login Fail Detect Job")
}

class MatchFunction extends KeyedProcessFunction[Long, LoginEvent, LoginEvent] {

  // 定义状态变量
  lazy val loginState: ListState[LoginEvent] = getRuntimeContext.getListState(
    new ListStateDescriptor[LoginEvent]("saved login", classOf[LoginEvent]))

  override def processElement(login: LoginEvent,
                              context: KeyedProcessFunction[Long, LoginEvent,
                              LoginEvent]#Context, out: Collector[LoginEvent]): Unit = {

    if (login.eventType == "fail") {
      loginState.add(login)
    }
    // 注册定时器，触发事件设定为2秒后
    context.timerService.registerEventTimeTimer(login.eventTime * 1000 + 2 * 1000)
  }
```

```
    override def onTimer(timestamp: Long,
                         ctx: KeyedProcessFunction[Long, LoginEvent,
                         LoginEvent]#OnTimerContext, out: Collector[LoginEvent]): Unit = {


      val allLogins: ListBuffer[LoginEvent] = ListBuffer()
      import scala.collection.JavaConversions._
      for (login <- loginState.get) {
        allLogins += login
      }
      loginState.clear()

      if (allLogins.length > 1) {
        out.collect(allLogins.head)
      }
    }
  }
}
```

5.2.2 状态编程的改进

上一节的代码实现中我们可以看到，直接把每次登录失败的数据存起来、设置定时器一段时间后再读取，这种做法尽管简单，但和我们开始的需求还是略有差异的。这种做法只能隔 2 秒之后去判断一下这期间是否有多次失败登录，而不是在一次登录失败之后、再一次登录失败时就立刻报警。这个需求如果严格实现起来，相当于要判断任意紧邻的事件，是否符合某种模式。

于是我们可以想到，这个需求其实可以不用定时器触发，直接在状态中存取上一次登录失败的事件，每次都做判断和比对，就可以实现最初的需求。

上节的代码 MatchFunction 中删掉 onTimer，processElement 改为：

```
override def processElement(value: LoginEvent, ctx: KeyedProcessFunction[Long,
LoginEvent, Warning]#Context, out: Collector[Warning]): Unit = {
```

```
    // 首先按照 type 做筛选，如果 success 直接清空，如果 fail 再做处理
    if ( value.eventType == "fail" ){
      // 如果已经有登录失败的数据，那么就判断是否在两秒内
      val iter = loginState.get().iterator()
      if ( iter.hasNext ){
        val firstFail = iter.next()
        // 如果两次登录失败时间间隔小于 2 秒，输出报警
        if ( value.eventTime < firstFail.eventTime + 2 ){
          out.collect( Warning( value.userId, firstFail.eventTime, value.eventTime,
"login fail in 2 seconds." ) )
        }
        // 把最近一次的登录失败数据，更新写入 state 中
        val failList = new util.ArrayList[LoginEvent]()
        failList.add(value)
        loginState.update( failList )
      } else {
        // 如果 state 中没有登录失败的数据，那就直接添加进去
        loginState.add(value)
      }
    } else
      loginState.clear()
  }
```

5.2.3 CEP 编程

上一节我们通过对状态编程的改进，去掉了定时器，在 process function 中做了更多的逻辑处理，实现了最初的需求。不过这种方法里有很多的条件判断，而我们目前仅仅实现的是检测“连续 2 次登录失败”，这是最简单的情形。如果需要检测更多次，内部逻辑显然会变得非常复杂。那有什么方式可以方便地实现呢？

很幸运，flink 为我们提供了 CEP（Complex Event Processing，复杂事件处理）库，用于在流中筛选符合某种复杂模式的事件。接下来我们就基于 CEP 来完成这个

模块的实现。

在 src/main/scala 下继续创建 LoginFailWithCep.scala 文件，新建一个单例对象。样例类 LoginEvent 由于在 LoginFail.scala 已经定义，我们在同一个模块中就不需要再定义了。

代码如下：

LoginFailDetect/src/main/scala/LoginFailWithCep.scala

```
    object LoginFailWithCep {


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


    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.setParallelism(1)


    val loginEventStream = env.readTextFile("YOUR_PATH\\resources\\LoginLog.csv")
      .map( data => {
        val dataArray = data.split(",")
        LoginEvent(dataArray(0).toLong, dataArray(1), dataArray(2),
dataArray(3).toLong)
      })
      .assignTimestampsAndWatermarks(new
                BoundedOutOfOrdernessTimestampExtractor[ApacheLogEvent]
                (Time.milliseconds(3000)) {
            override def extractTimestamp(element: ApacheLogEvent): Long ={
                element.eventTime * 1000L
            }
      })
    // 定义匹配模式
    val loginFailPattern = Pattern.begin[LoginEvent]("begin")
      .where(_.eventType == "fail")
```

```
      .next("next")
      .where(_.eventType == "fail")
      .within(Time.seconds(2))

    // 在数据流中匹配出定义好的模式
    val patternStream = CEP.pattern(loginEventStream.keyBy(_.userId), loginFailPattern)

    // .select 方法传入一个 pattern select function，当检测到定义好的模式序列时就会调用
    val loginFailDataStream = patternStream
      .select((pattern: Map[String, Iterable[LoginEvent]]) => {
        val first = pattern.getOrElse("begin", null).iterator.next()
        val second = pattern.getOrElse("next", null).iterator.next()
        (second.userId, second.ip, second.eventType)
      })
    // 将匹配到的符合条件的事件打印出来
    loginFailDataStream.print()
    env.execute("Login Fail Detect Job")
  }
}
```

第 6 章 订单支付实时监控

在电商网站中，订单的支付作为直接与营销收入挂钩的一环，在业务流程中非常重要。对于订单而言，为了正确控制业务流程，也为了增加用户的支付意愿，网站一般会设置一个支付失效时间，超过一段时间不支付的订单就会被取消。另外，对于订单的支付，我们还应保证用户支付的正确性，这可以通过第三方支付平台的交易数据来做一个实时对账。在接下来的内容中，我们将实现这两个需求。

6.1 模块创建和数据准备

同样地，在 UserBehaviorAnalysis 下新建一个 maven module 作为子项目，命名

为 OrderTimeoutDetect。在这个子模块中，我们同样将会用到 flink 的 CEP 库来实现事件流的模式匹配，所以需要在 pom 文件中引入 CEP 的相关依赖：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-cep-scala_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
</dependency>
```

同样，在 src/main/目录下，将默认源文件目录 java 改名为 scala。

6.2 代码实现

在电商平台中，最终创造收入和利润的是用户下单购买的环节；更具体一点，是用户真正完成支付动作的时候。用户下单的行为可以表明用户对商品的需求，但在现实中，并不是每次下单都会被用户立刻支付。当拖延一段时间后，用户支付的意愿会降低。所以为了让用户更有紧迫感从而提高支付转化率，同时也为了防范订单支付环节的安全风险，电商网站往往会对订单状态进行监控，设置一个失效时间（比如 15 分钟），如果下单后一段时间仍未支付，订单就会被取消。

6.2.1 使用 CEP 实现

我们首先还是利用 CEP 库来实现这个功能。我们先将事件流按照订单号 orderId 分流，然后定义这样的一个事件模式：在 15 分钟内，事件“create”与“pay”非严格紧邻：

```
val orderPayPattern = Pattern.begin[OrderEvent]("begin")
  .where(_.eventType == "create")
  .followedBy("follow")
  .where(_.eventType == "pay")
  .within(Time.seconds(5))
```

这样调用.select 方法时，就可以同时获取到匹配出的事件和超时未匹配的事件了。

在 src/main/scala 下继续创建 OrderTimeout.scala 文件，新建一个单例对象。定义样例类 OrderEvent，这是输入的订单事件流；另外还有 OrderResult，这是输出显示的订单状态结果。订单数据也本应该从 UserBehavior 日志里提取，由于

UserBehavior.csv 中没有做相关埋点，我们从另一个文件 OrderLog.csv 中读取登录数据。

完整代码如下：

OrderTimeoutDetect/src/main/scala/OrderTimeout.scala

```
case class OrderEvent(orderId: Long, eventType: String, eventTime: Long)
case class OrderResult(orderId: Long, eventType: String)

object OrderTimeout {

  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setParallelism(1)
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    val orderEventStream = env.readTextFile("YOUR_PATH\\resources\\OrderLog.csv")
      .map( data => {
        val dataArray = data.split(",")
        OrderEvent(dataArray(0).toLong, dataArray(1), dataArray(3).toLong)
      })
      .assignAscendingTimestamps(_.eventTime * 1000)

    // 定义一个带匹配时间窗口的模式
    val orderPayPattern = Pattern.begin[OrderEvent]("begin")
      .where(_.eventType == "create")
      .followedBy("follow")
      .where(_.eventType == "pay")
      .within(Time.minutes(15))

    // 定义一个输出标签
```

```
    val orderTimeoutOutput = OutputTag[OrderResult]("orderTimeout")
    // 订单事件流根据 orderId 分流，然后在每一条流中匹配出定义好的模式
    val patternStream = CEP.pattern(orderEventStream.keyBy("orderId"), orderPayPattern)

    val completedResult = patternStream.select(orderTimeoutOutput) {
      // 对于已超时的部分模式匹配的事件序列，会调用这个函数
      (pattern: Map[String, Iterable[OrderEvent]], timestamp: Long) => {
        val createOrder = pattern.get("begin")
        OrderResult(createOrder.get.iterator.next().orderId, "timeout")
      }
    } {
      // 检测到定义好的模式序列时，就会调用这个函数
      pattern: Map[String, Iterable[OrderEvent]] => {
        val payOrder = pattern.get("follow")
        OrderResult(payOrder.get.iterator.next().orderId, "success")
      }
    }
    // 拿到同一输出标签中的 timeout 匹配结果（流）
    val timeoutResult = completedResult.getSideOutput(orderTimeoutOutput)

    completedResult.print()
    timeoutResult.print()

    env.execute("Order Timeout Detect Job")
  }
}
```

6.2.2 使用 Process Function 实现

我们同样可以利用 Process Function，自定义实现检测订单超时的功能。为了简化问题，我们只考虑超时报警的情形，在 pay 事件超时未发生的情况下，输出超时

报警信息。

一个简单的思路是，可以在订单的 create 事件到来后注册定时器，15 分钟后触发；然后再用一个布尔类型的 Value 状态来作为标识位，表明 pay 事件是否发生过。如果 pay 事件已经发生，状态被置为 true，那么就不再需要做什么操作；而如果 pay 事件一直没来，状态一直为 false，到定时器触发时，就应该输出超时报警信息。

具体代码实现如下：

OrderTimeoutDetect/src/main/scala/OrderTimeoutWithoutCep.scala

```
object OrderTimeoutWithoutCep {

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

    val env = StreamExecutionEnvironment.getExecutionEnvironment

    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    env.setParallelism(1)


    val orderEventStream = env.readTextFile("YOUR_PATH\\resources\\OrderLog.csv")

      .map( data => {

        val dataArray = data.split(",")

        OrderEvent(dataArray(0).toLong, dataArray(1), dataArray(3).toLong)

      })

      .assignAscendingTimestamps(_.eventTime * 1000)

      .keyBy(_.orderId)


    // 自定义一个 process function，进行 order 的超时检测，输出超时报警信息

    val timeoutWarningStream = orderEventStream

      .process(new OrderTimeoutAlert)


    timeoutWarningStream.print()


    env.execute()

  }
```

```
  class OrderTimeoutAlert extends KeyedProcessFunction[Long, OrderEvent, OrderResult]
{
    lazy val isPayedState: ValueState[Boolean] = getRuntimeContext.getState(new
ValueStateDescriptor[Boolean]("ispayed-state", classOf[Boolean]))

    override def processElement(value: OrderEvent, ctx: KeyedProcessFunction[Long,
OrderEvent, OrderResult]#Context, out: Collector[OrderResult]): Unit = {
      val isPayed = isPayedState.value()

      if (value.eventType == "create" && !isPayed) {
        ctx.timerService().registerEventTimeTimer(value.eventTime * 1000L + 15 * 60 *
1000L)
      } else if (value.eventType == "pay") {
        isPayedState.update(true)
      }
    }
    override def onTimer(timestamp: Long, ctx: KeyedProcessFunction[Long, OrderEvent,
OrderResult]#OnTimerContext, out: Collector[OrderResult]): Unit = {
      val isPayed = isPayedState.value()
      if (!isPayed) {
        out.collect(OrderResult(ctx.getCurrentKey, "order timeout"))
      }
      isPayedState.clear()
    }
  }
}
```

6.3 来自两条流的订单交易匹配

对于订单支付事件，用户支付完成其实并不算完，我们还得确认平台账户上是

否到账了。而往往这会来自不同的日志信息，所以我们要同时读入两条流的数据来做合并处理。这里我们利用 connect 将两条流进行连接，然后用自定义的 CoProcessFunction 进行处理。

具体代码如下：

TxMatchDetect/src/main/scala/TxMatch

```
case class OrderEvent( orderId: Long, eventType: String, txId: String, eventTime: Long )


case class ReceiptEvent( txId: String, payChannel: String, eventTime: Long )


object TxMatch {


  val unmatchedPays = new OutputTag[OrderEvent]("unmatchedPays")

  val unmatchedReceipts = new OutputTag[ReceiptEvent]("unmatchedReceipts")


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

    val env = StreamExecutionEnvironment.getExecutionEnvironment

    env.setParallelism(1)

    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)


    val orderEventStream = env.readTextFile("YOUR_PATH\\resources\\OrderLog.csv")

      .map( data => {

        val dataArray = data.split(",")

        OrderEvent(dataArray(0).toLong, dataArray(1), dataArray(2),
dataArray(3).toLong)

      })


      .filter(_.txId != "")

      .assignAscendingTimestamps(_.eventTime * 1000L)

      .keyBy(_.txId)
```

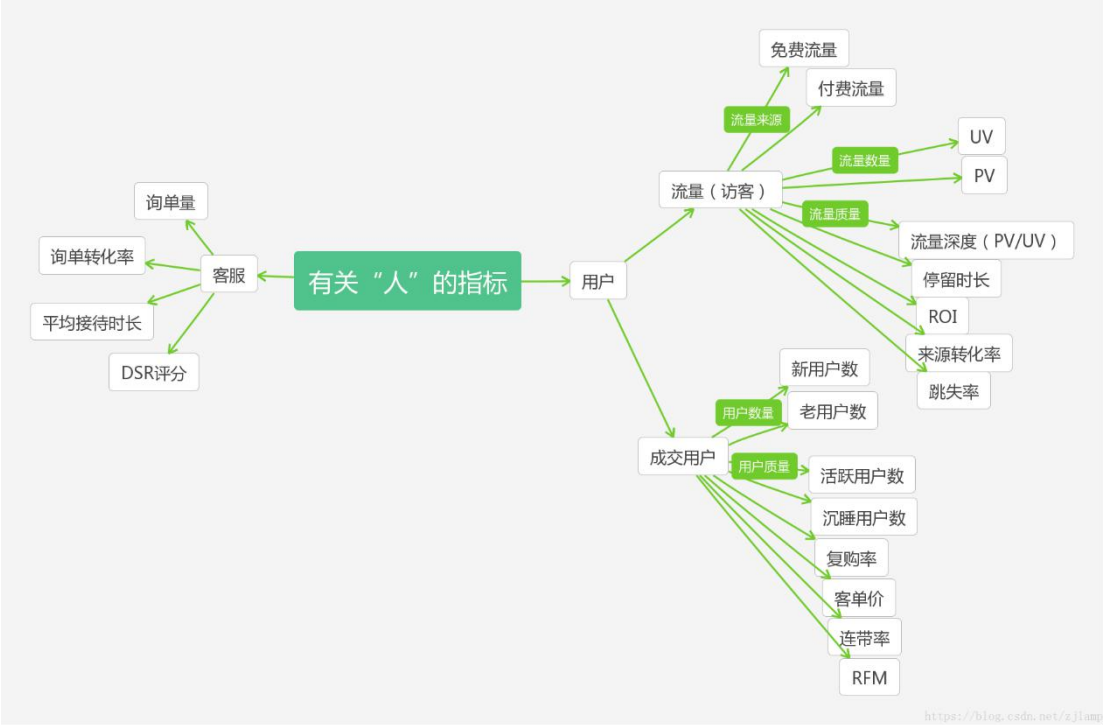
```
  val receiptEventStream = env.readTextFile("YOUR_PATH\\resources\\ReceiptLog.csv")
    .map( data => {
      val dataArray = data.split(",")
      ReceiptEvent(dataArray(0), dataArray(1), dataArray(2).toLong)
    })
    .assignAscendingTimestamps(_.eventTime * 1000L)
    .keyBy(_.txId)

  val processedStream = orderEventStream
    .connect(receiptEventStream)
    .process(new TxMatchDetection)

  processedStream.getSideOutput(unmatchedPays).print("unmatched pays")
  processedStream.getSideOutput(unmatchedReceipts).print("unmatched receipts")

  processedStream.print("processed")

  env.execute()
}

class TxMatchDetection extends CoProcessFunction[OrderEvent, ReceiptEvent,
(OrderEvent, ReceiptEvent)]{
  lazy val payState: ValueState[OrderEvent] = getRuntimeContext.getState(new
ValueStateDescriptor[OrderEvent]("pay-state",classOf[OrderEvent]) )
  lazy val receiptState: ValueState[ReceiptEvent] = getRuntimeContext.getState(new
ValueStateDescriptor[ReceiptEvent]("receipt-state", classOf[ReceiptEvent]) )

  override def processElement1(pay: OrderEvent, ctx: CoProcessFunction[OrderEvent,
```

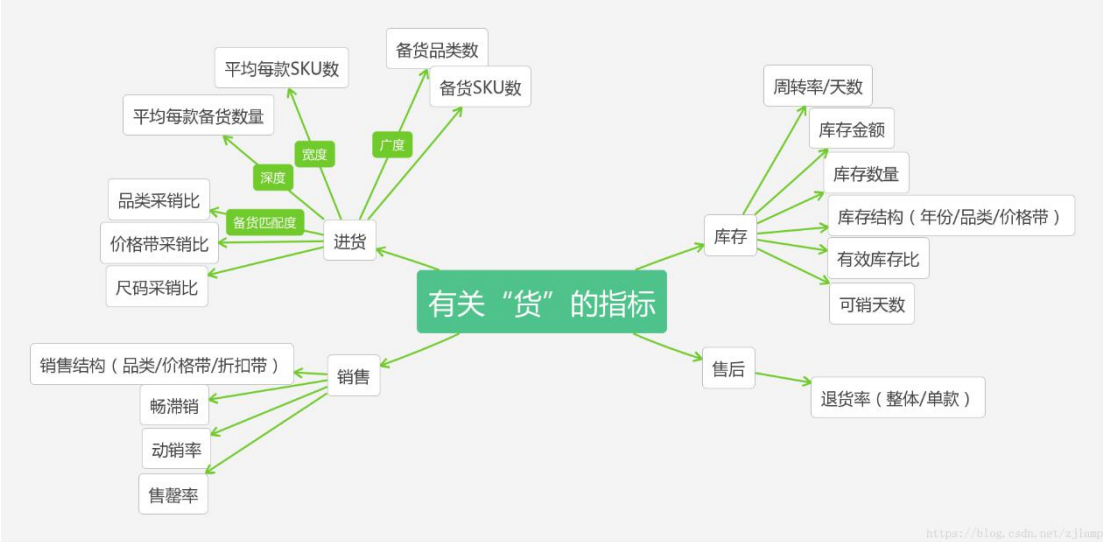
```
ReceiptEvent, (OrderEvent, ReceiptEvent)]#Context, out: Collector[(OrderEvent,
ReceiptEvent)]): Unit = {
      val receipt = receiptState.value()

      if( receipt != null ){
        receiptState.clear()
        out.collect((pay, receipt))
      } else{
        payState.update(pay)
        ctx.timerService().registerEventTimeTimer(pay.eventTime * 1000L)
      }
    }

    override def processElement2(receipt: ReceiptEvent, ctx:
CoProcessFunction[OrderEvent, ReceiptEvent, (OrderEvent, ReceiptEvent)]#Context, out:
Collector[(OrderEvent, ReceiptEvent)]): Unit = {
      val payment = payState.value()

      if( payment != null ){
        payState.clear()
        out.collect((payment, receipt))
      } else{
        receiptState.update(receipt)
        ctx.timerService().registerEventTimeTimer(receipt.eventTime * 1000L)
      }
    }

    override def onTimer(timestamp: Long, ctx: CoProcessFunction[OrderEvent,
ReceiptEvent, (OrderEvent, ReceiptEvent)]#OnTimerContext, out: Collector[(OrderEvent,
```

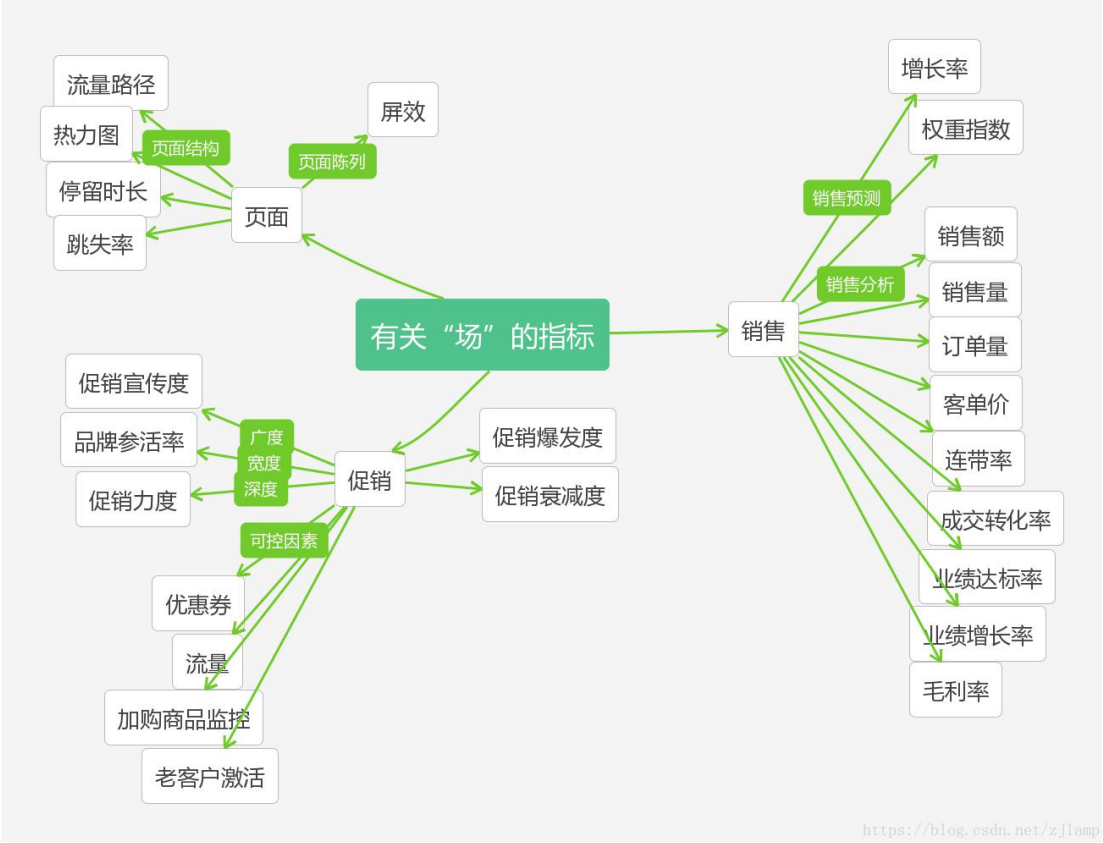
```
ReceiptEvent)]): Unit = {
      if ( payState.value() != null ){
        ctx.output(unmatchedPays, payState.value())
      }
      if ( receiptState.value() != null ){
        ctx.output(unmatchedReceipts, receiptState.value())
      }
      payState.clear()
      receiptState.clear()
    }
  }
}
```

附录 电商常见指标汇总

1. 电商指标整理







现在的电子商务：

1、大多买家通过搜索找到所买物品，而非电商网站的内部导航，搜索关键字更为重要；

2、电商商家通过推荐引擎来预测买家可能需要的商品。推荐引擎以历史上具有类似购买记录的买家数据以及用户自身的购买记录为基础，向用户提供推荐信息；

3、电商商家时刻优化网站性能，如 A/B Test 划分来访流量，并区别对待来源不同的访客，进而找到最优的产品、内容和价格；

4、购买流程早在买家访问网站前，即在社交网络、邮件以及在线社区中便已开始，即长漏斗流程（以一条推文、一段视频或一个链接开始，以购买交易结束）。

相关数据指标：关键词和搜索词、推荐接受率、邮件列表/短信链接点入率

2. 电商 8 类基本指标

1）总体运营指标：从流量、订单、总体销售业绩、整体指标进行把控，起码对运营的电商平台有个大致了解，到底运营的怎么样，是亏是赚。

2）站流量指标：即对访问你网站的访客进行分析，基于这些数据可以对网页进行改进，以及对访客的行为进行分析等等。

3）销售转化指标：分析从下单到支付整个过程的数据，帮助你提升商品转化率。也可以对一些频繁异常的数据展开分析。

4）客户价值指标：这里主要就是分析客户的价值，可以建立 RFM 价值模型，找出那些有价值的客户，精准营销等等。

5）商品类指标：主要分析商品的种类，那些商品卖得好，库存情况，以及可以建立关联模型，分析那些商品同时销售的几率比较高，而进行捆绑销售，有点像啤酒和尿布的故事。

6）市场营销活动指标，主要监控某次活动给电商网站带来的效果，以及监控广告的投放指标。

7）风控类指标：分析卖家评论，以及投诉情况，发现问题，改正问题

8）市场竞争指标：主要分析市场份额以及网站排名，进一步进行调整