## Flink多语言开发

## 课程目标

* 了解PyFlink并掌握官方示例
* 掌握Scala语音编写Flink程序
* 实现电商点击流日志分析

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/scala_api_extensions.html>

## Scala-Flink

### 需求

使用Flink从Kafka接收对电商点击流日志数据并进行实时处理:

1.数据预处理:对数据进行拓宽处理,也就是将数据变为宽表,方便后续分析

2.分析实时频道热点

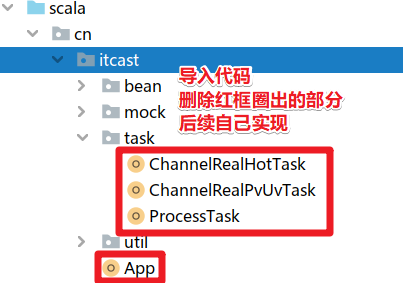
3.分析实时频道PV/UV

### 准备工作

* kafka

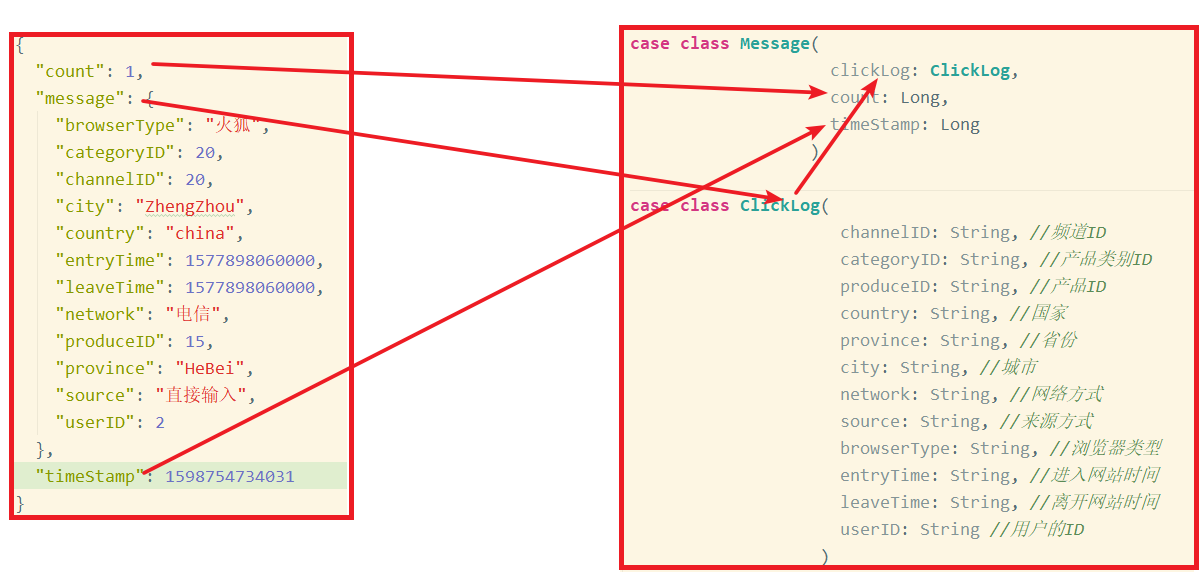
|  |
| --- |
| 查看主题:  /export/servers/kafka/bin/kafka-topics.sh --list --zookeeper node01:2181  创建主题:  /export/servers/kafka/bin/kafka-topics.sh --create --zookeeper node01:2181 --replication-factor 2 --partitions 3 --topic pyg  再次查看主题:  /export/servers/kafka/bin/kafka-topics.sh --list --zookeeper node01:2181  启动控制台消费者  /export/servers/kafka/bin/kafka-console-consumer.sh --bootstrap-server node01:9092 --from-beginning --topic pyg  删除主题--不需要执行  /export/servers/kafka/bin/kafka-topics.sh --delete --zookeeper node01:2181 --topic pyg |

* 导入准备骨架代码



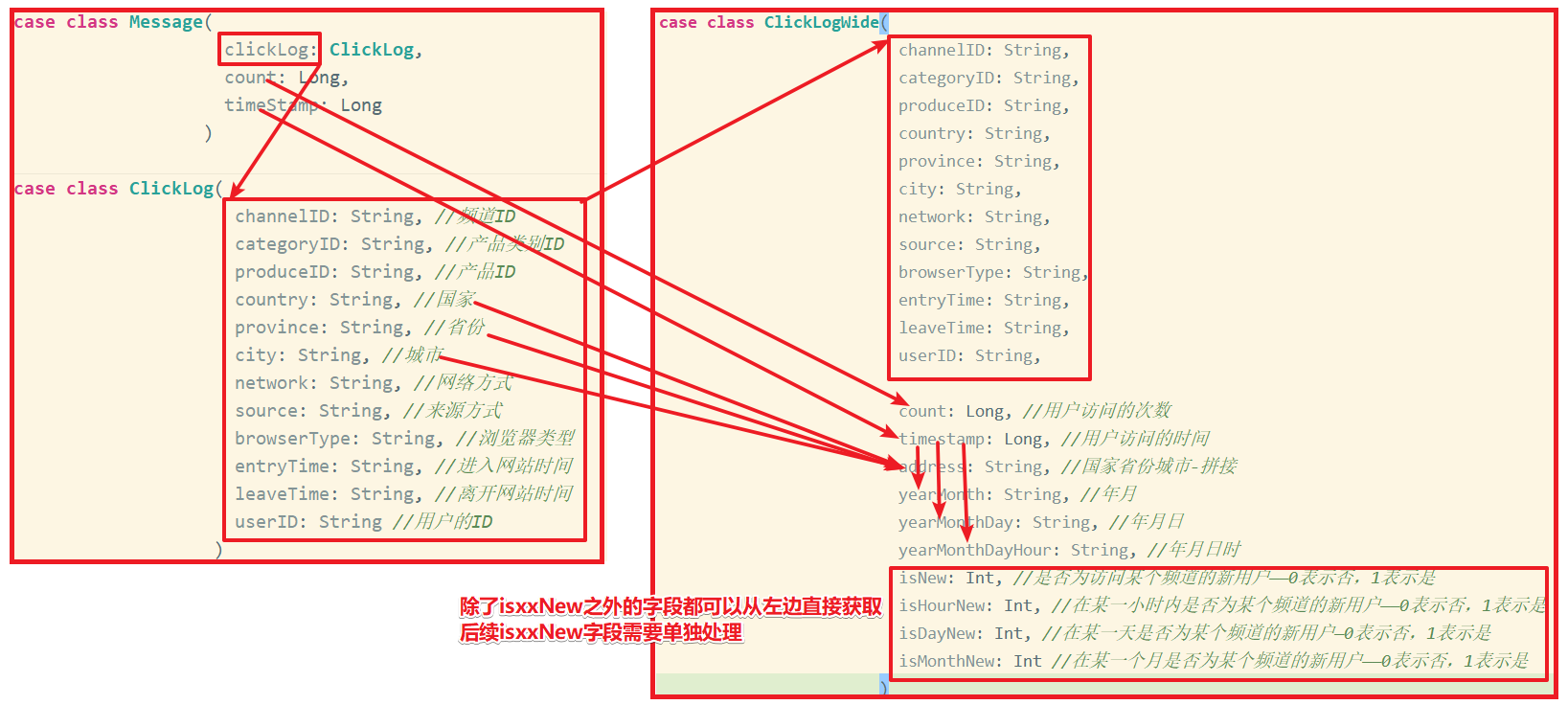
### 代码实现

#### 入口类-数据解析



|  |
| --- |
| **package** com.nhbd.flink.ch7  **import** java.lang **import** java.util.Properties **import** java.util.concurrent.TimeUnit  **import** com.nhbd.flink.ch7.bean.{ClickLog, ClickLogWide, Message} **import** com.nhbd.flink.ch7.task.{ChannelRealHotTask, ChannelRealPvUvTask, ProcessTask} **import** com.alibaba.fastjson.{JSON, JSONObject} **import** org.apache.flink.api.common.restartstrategy.RestartStrategies **import** org.apache.flink.api.common.serialization.SimpleStringSchema **import** org.apache.flink.runtime.state.filesystem.FsStateBackend **import** org.apache.flink.streaming.api.environment.CheckpointConfig.ExternalizedCheckpointCleanup **import** org.apache.flink.streaming.api.functions.timestamps.BoundedOutOfOrdernessTimestampExtractor **import** org.apache.flink.streaming.api.scala.\_ **import** org.apache.flink.streaming.api.{CheckpointingMode, TimeCharacteristic} **import** org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer **import** org.apache.kafka.clients.CommonClientConfigs  */\*\*  \*   \* Desc 编写Flink实时流处理-入口程序(这都是通用模版代码,这里写一遍,明天再做抽取,以后开发都可以直接使用)  \*/* **object** App {  **def** main(args: Array[String]): Unit = {  *//注意:TODO在开发中表示该步骤未完成,后续需要补全  //在这里仅仅为了使用不同的颜色区分步骤  //****TODO 1.准备环境StreamExecutionEnvironment* val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment  //****TODO 2.设置环境参数(Checkpoint/重启策略/是否使用事件时间...)*** *//=================建议必须设置的===================  //设置Checkpoint-State的状态后端为FsStateBackend,本地测试时使用本地路径,集群测试时使用传入的HDFS的路径* **if**(args.length<1){  env.setStateBackend(**new** FsStateBackend(**"file:///D:/ckp"**))  }**else**{  env.setStateBackend(**new** FsStateBackend(args(0)))*//后续集群测试时传入hdfs://node01:8020/flink-checkpoint/checkpoint* }  *//设置Checkpointing时间间隔为1000ms,意思是做 2 个 Checkpoint 的间隔为1000ms。Checkpoint 做的越频繁，恢复数据时就越简单，同时 Checkpoint 相应的也会有一些IO消耗。* env.enableCheckpointing(1000)*//(默认情况下如果不设置时间checkpoint是没有开启的)  //设置两个Checkpoint 之间最少等待时间,如设置Checkpoint之间最少是要等 500ms(为了避免每隔1000ms做一次Checkpoint的时候,前一次太慢和后一次重叠到一起去了)  //如:高速公路上,每隔1s关口放行一辆车,但是规定了两车之前的最小车距为500m* env.getCheckpointConfig.setMinPauseBetweenCheckpoints(500)*//默认是0  //设置如果在做Checkpoint过程中出现错误，是否让整体任务失败：true是 false不是* env.getCheckpointConfig.setFailOnCheckpointingErrors(**false**)*//默认是true  //设置是否清理检查点,表示 Cancel 时是否需要保留当前的 Checkpoint，默认 Checkpoint会在作业被Cancel时被删除  //ExternalizedCheckpointCleanup.DELETE\_ON\_CANCELLATION：true,当作业被取消时，删除外部的checkpoint(默认值)  //ExternalizedCheckpointCleanup.RETAIN\_ON\_CANCELLATION：false,当作业被取消时，保留外部的checkpoint* env.getCheckpointConfig.enableExternalizedCheckpoints(ExternalizedCheckpointCleanup.*RETAIN\_ON\_CANCELLATION*)  *//=================建议必须设置的===================   //=================直接使用默认的即可===============  //设置checkpoint的执行模式为EXACTLY\_ONCE(默认),注意:得需要外部支持,如Source和Sink的支持* env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.*EXACTLY\_ONCE*)  *//设置checkpoint的超时时间,如果 Checkpoint在 60s内尚未完成说明该次Checkpoint失败,则丢弃。* env.getCheckpointConfig.setCheckpointTimeout(60000)*//默认10分钟  //设置同一时间有多少个checkpoint可以同时执行* env.getCheckpointConfig.setMaxConcurrentCheckpoints(1)*//默认为1  //=================直接使用默认的即可===============    //======================配置重启策略==============  //1.如果配置了Checkpoint,而没有配置重启策略,那么代码中出现了非致命错误时,程序会无限重启  //2.配置无重启策略  //env.setRestartStrategy(RestartStrategies.noRestart())  //3.固定延迟重启策略--开发中使用  //如下:如果有异常,每隔10s重启1次,最多3次* env.setRestartStrategy(RestartStrategies.*fixedDelayRestart*(  3, *// 最多重启3次数* org.apache.flink.api.common.time.Time.*of*(10, TimeUnit.*SECONDS*) *// 重启时间间隔* ))  *//4.失败率重启策略--开发偶尔使用  //如下:5分钟内,最多重启3次,每次间隔10  /\*env.setRestartStrategy(RestartStrategies.failureRateRestart(  3, // 每个测量时间间隔最大失败次数  Time.of(5, TimeUnit.MINUTES), //失败率测量的时间间隔  Time.of(10, TimeUnit.SECONDS) // 两次连续重启的时间间隔  ))\*/  //======================配置重启策略==============   //****TODO 3.Source-Kafka* val** topic: String = **"pyg"  val** schema = **new** SimpleStringSchema()  **val** props:Properties = **new** Properties()  props.setProperty(CommonClientConfigs.*BOOTSTRAP\_SERVERS\_CONFIG*,**"node1:9092"**)  props.setProperty(**"group.id"**,**"flink"**)  props.setProperty(**"auto.offset.reset"**,**"latest"**)*//如果有记录偏移量从记录的位置开始消费,如果没有从最新的数据开始消费* props.setProperty(**"flink.partition-discovery.interval-millis"**,**"5000"**)*//动态分区检测,开一个后台线程每隔5s检查Kafka的分区状态* **val** kafkaSource: FlinkKafkaConsumer[String] = **new** FlinkKafkaConsumer[String](topic,schema,props)  kafkaSource.setCommitOffsetsOnCheckpoints(**true**)*//在执行Checkpoint的时候,会提交offset(一份在Checkpoint中,一份在默认主题)* **val** jsonStrDS: DataStream[String] = env.addSource(kafkaSource)  *//jsonStrDS.print()  // {"count":1,"message":"{\"browserType\":\"火狐\",\"categoryID\":20,\"channelID\":20,\"city\":\"ZhengZhou\",\"country\":\"china\",\"entryTime\":1577898060000,\"leaveTime\":1577898060000,\"network\":\"电信\",\"produceID\":15,\"province\":\"HeBei\",\"source\":\"直接输入\",\"userID\":2}","timeStamp":1598754734031}   //****TODO 4.解析jsonStr数据为样例类Message* val** messageDS: DataStream[Message] = jsonStrDS.map(jsonStr => {  **val** jsonObj: JSONObject = JSON.*parseObject*(jsonStr)  **val** count: lang.Long = jsonObj.getLong(**"count"**)  **val** timeStamp: lang.Long = jsonObj.getLong(**"timeStamp"**)  **val** clickLogStr: String = jsonObj.getString(**"message"**)  **val** clickLog: ClickLog = JSON.*parseObject*(clickLogStr, *classOf*[ClickLog])  *Message*(clickLog, count, timeStamp)  *//不能使用下面偷懒的办法  //val message: Message = JSON.parseObject(jsonStr,classOf[Message])* })  *//messageDS.print()  //Message(ClickLog(10,10,3,china,HeBei,ZhengZhou,电信,360搜索跳转,谷歌浏览器,1577876460000,1577898060000,15),1,1598754740100)   //****TODO 5.给数据添加Watermaker(或者放在第6步)*** env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  env.getConfig.setAutoWatermarkInterval(200)  **val** watermakerDS: DataStream[Message] = messageDS.assignTimestampsAndWatermarks(  **new** BoundedOutOfOrdernessTimestampExtractor[Message](org.apache.flink.streaming.api.windowing.time.Time.*seconds*(5)) {  **override def** extractTimestamp(element: Message): Long = element.timeStamp  }  )   *//****TODO 6.数据预处理*** *//为了方便后续的指标统计,可以对上面解析处理的日志信息Message进行预处理,如拓宽字段  //预处理的代码可以写在这里,也可以单独抽取出一个方法来完成,也可以单独抽取一个object.方法来完成  //把DataStream[Message]拓宽为DataStream[ClickLogWide]* **val** clickLogWideDS: DataStream[ClickLogWide] = ProcessTask.*process*(watermakerDS)  clickLogWideDS.print()  *//ClickLogWide(18,9,10,china,HeNan,LuoYang,移动,百度跳转,谷歌浏览器,1577887260000,1577898060000,15,1,1598758614216,chinaHeNanLuoYang,202008,20200830,2020083011,0,0,0,0)    //****TODO 7.实时指标统计分析-直接sink结果到HBase*** *//实时指标统计分析-实时频道热点* ChannelRealHotTask.*process*(clickLogWideDS)  *//实时指标统计分析-实时频道分时段PV/UV* ChannelRealPvUvTask.*process*(clickLogWideDS)    *//****TODO 8.execute*** env.execute()  } } |

#### 数据预处理



为了方便后续分析，我们需要对点击流日志，使用Flink进行实时预处理。在原有点击流日志的基础上添加一些字段，方便进行后续业务功能的统计开发。

以下为Kafka中消费得到的原始点击流日志字段：

|  |  |
| --- | --- |
| 字段名 | 说明 |
| channelID | 频道ID |
| categoryID | 产品类别ID |
| produceID | 产品ID |
| country | 国家 |
| province | 省份 |
| city | 城市 |
| network | 网络方式 |
| source | 来源方式 |
| browserType | 浏览器类型 |
| entryTime | 进入网站时间 |
| leaveTime | 离开网站时间 |
| userID | 用户的ID |

我们需要在原有点击流日志字段基础上，再添加以下字段：

|  |  |
| --- | --- |
| 字段名 | 说明 |
| count | 用户访问的次数 |
| timestamp | 用户访问的时间 |
| address | 国家省份城市（拼接） |
| yearMonth | 年月 |
| yearMonthDay | 年月日 |
| yearMonthDayHour | 年月日时 |
| isNew | 是否为访问某个频道的新用户 |
| isHourNew | 在某一小时内是否为某个频道的新用户 |
| isDayNew | 在某一天是否为某个频道的新用户 |
| isMonthNew | 在某一个月是否为某个频道的新用户 |

我们不能直接从点击流日志中，直接计算得到上述后4个字段的值。而是需要在hbase中有一个历史记录表，来保存用户的历史访问状态才能计算得到。

该历史记录表(user\_history表)结构如下：

|  |  |  |
| --- | --- | --- |
| 列名 | 说明 | 示例 |
| rowkey | 用户ID:频道ID | 10:220 |
| userid | 用户ID | 10 |
| channelid | 频道ID | 220 |
| lastVisitedTime | 最后访问时间（时间戳） | 1553653555 |

|  |
| --- |
| **package** com.nhbd.flink.ch7.task  **import** com.nhbd.flink.ch7.bean.{ClickLogWide, Message} **import** com.nhbd.flink.ch7.util.{HBaseUtil, TimeUtil} **import** org.apache.commons.lang3.StringUtils **import** org.apache.flink.streaming.api.scala.DataStream  */\*\*  \*   \* Desc 数据预处理模块业务任务  \*/* **object** ProcessTask {  *//将添加了水印的原始的用户行为日志数据根据需求转为宽表ClickLogWide并返回  //将DataStream[Message]转为DataStream[ClickLogWide]* **def** process(watermakerDS: DataStream[Message]): DataStream[ClickLogWide] = {  **import** org.apache.flink.api.scala.\_  **val** clickLogWideDS: DataStream[ClickLogWide] = watermakerDS.map(message => {  **val** address: String = message.clickLog.country + message.clickLog.province + message.clickLog.city  **val** yearMonth: String = TimeUtil.*parseTime*(message.timeStamp, **"yyyyMM"**)  **val** yearMonthDay: String = TimeUtil.*parseTime*(message.timeStamp, **"yyyyMMdd"**)  **val** yearMonthDayHour: String = TimeUtil.*parseTime*(message.timeStamp, **"yyyyMMddHH"**)   **val** (isNew, isHourNew, isDayNew, isMonthNew) = *getIsNew*(message)   **val** clickLogWide = *ClickLogWide*(  message.clickLog.channelID,  message.clickLog.categoryID,  message.clickLog.produceID,  message.clickLog.country,  message.clickLog.province,  message.clickLog.city,  message.clickLog.network,  message.clickLog.source,  message.clickLog.browserType,  message.clickLog.entryTime,  message.clickLog.leaveTime,  message.clickLog.userID,   message.count, *//用户访问的次数* message.timeStamp, *//用户访问的时间* address, *//国家省份城市-拼接* yearMonth, *//年月* yearMonthDay, *//年月日* yearMonthDayHour, *//年月日时* isNew, *//是否为访问某个频道的新用户——0表示否，1表示是* isHourNew, *//在某一小时内是否为某个频道的新用户——0表示否，1表示是* isDayNew, *//在某一天是否为某个频道的新用户—0表示否，1表示是* isMonthNew *//在某一个月是否为某个频道的新用户——0表示否，1表示是* )  clickLogWide  })  clickLogWideDS  }   */\*如:某用户,2020-08-30-11,第一次访问该频道  那么这条日志  isNew=1  isHourNew=1  isDayNew=1  isMonthNew=1  该用户2020-08-30-11,再次访问  那么这条日志:  isNew=0  isHourNew=0  isDayNew=0  isMonthNew=0  该用户2020-08-30-12,再次访问  isNew=0  isHourNew=1  isDayNew=0  isMonthNew=0  该用户2020-08-31-09,再次访问  isNew=0  isHourNew=1  isDayNew=1  isMonthNew=0\*/* **def** getIsNew(msg: Message):(Int,Int,Int,Int) = {  **var** isNew: Int = 0 *//是否为访问某个频道的新用户——0表示否，1表示是* **var** isHourNew: Int = 0 *//在某一小时内是否为某个频道的新用户——0表示否，1表示是* **var** isDayNew: Int = 0 *//在某一天是否为某个频道的新用户—0表示否，1表示是* **var** isMonthNew: Int = 0*//在某一个月是否为某个频道的新用户——0表示否，1表示是   //如何判断该用户是该频道的各个isxxNew?  //可以把上次 该用户 访问 该频道 的 访问时间 记录在外部介质中,如HBase中  //进来一条日志,先去HBase查该用户该频道的lastVisitTime  //没有结果--isxxNew全是1  //有结果--把这次访问时间和lastVisitTime进行比较   //1.定义一些HBase的常量,如表名,列族名,字段名* **val** tableName = **"user\_history"  val** columnFamily = **"info"  val** rowkey = msg.clickLog.userID + **":"** + msg.clickLog.channelID  **val** queryColumn = **"lastVisitTime"** *//2.根据该用户的该频道去查lastVisitTime  //注意:记得修改resources/hbase-site.xml中的主机名,还得启动HBase* **val** lastVisitTime: String = HBaseUtil.*getData*(tableName,rowkey,columnFamily,queryColumn)   *//3.判断lastVisitTime是否有值* **if**(StringUtils.*isBlank*(lastVisitTime)){  *//如果lastVisitTime为空,说明该用户之前没有访问过该频道,全设置为1即可* isNew = 1  isHourNew = 1  isDayNew = 1  isMonthNew = 1  }**else**{  *//如果lastVisitTime不为空,说明该用户之前访问过该频道,那么isxxNew给根据情况来赋值  //如:lastVisitTime为2020-08-30-11,当前这一次访问时间为:2020-08-30-12,那么isHourNew=1,其他的为0  //如:lastVisitTime为2020-08-30,当前这一次访问时间为:2020-08-31,那么isDayNew=1,其他的为0  //如:lastVisitTime为2020-08,当前这一次访问时间为:2020-09,那么isMonthNew=1,其他的为0* isNew = 0  isHourNew = TimeUtil.*compareDate*(msg.timeStamp,lastVisitTime.toLong,**"yyyyMMddHH"**)  isDayNew = TimeUtil.*compareDate*(msg.timeStamp,lastVisitTime.toLong,**"yyyyMMdd"**)  isMonthNew = TimeUtil.*compareDate*(msg.timeStamp,lastVisitTime.toLong,**"yyyyMM"**)  }  *//不要忘了把这一次的访问时间作为lastVisitTime存入HBase* HBaseUtil.*putData*(tableName,rowkey,columnFamily,queryColumn,msg.timeStamp.toString)   (isNew,isHourNew,isDayNew,isMonthNew)   *//注意:  /\*  测试时先启动hbase  /export/servers/hbase/bin/start-hbase.sh  再登入hbase shell  ./hbase shell  查看hbase表  list  运行后会生成表,然后查看表数据  scan "user\_history",{LIMIT=>10}  \*/* } } |

#### 实时频道热点

频道热点，就是要统计频道被访问（点击）的数量。

分析得到以下的数据：

|  |  |
| --- | --- |
| 频道ID | 访问数量 |
| 频道ID1 | 128 |
| 频道ID2 | 401 |
| 频道ID3 | 501 |

需要将历史的点击数据进行累加

|  |
| --- |
| **package** com.nhbd.flink.ch7.task  **import** com.nhbd.flink.ch7.bean.ClickLogWide **import** com.nhbd.flink.ch7.util.HBaseUtil **import** org.apache.commons.lang3.StringUtils **import** org.apache.flink.streaming.api.functions.sink.SinkFunction **import** org.apache.flink.streaming.api.scala.DataStream **import** org.apache.flink.streaming.api.windowing.time.Time  */\*\*  \*   \* Desc 统计频道实时热点指标任务模块  \* 需求:每隔10s统计一次各个频道的访问次数  \*/* **object** ChannelRealHotTask {   *//定义一个样例类,用来封装频道id和访问次数* **case class** ChannelRealHot(channelId: String, visited: Long)   *//根据传入的用户行为日志宽表,进行频道的访问次数统计分析,并将结果保存到HBase* **def** process(clickLogWideDS: DataStream[ClickLogWide]) = {  **import** org.apache.flink.api.scala.\_  *//1.取出我们需要的字段channelID和count,并封装为样例类* **val** result: DataStream[ChannelRealHot] = clickLogWideDS  .map(clickLogWide => {  *ChannelRealHot*(clickLogWide.channelID, clickLogWide.count)  })  *//2.分组* .keyBy(\_.channelId)  *//3.窗口  //ize: Time, slide: Time  //需求:每隔10s统计一次各个频道的访问次数* .timeWindow(Time.*seconds*(10))  *//4.聚合* .reduce((c1, c2) => {  *ChannelRealHot*(c2.channelId, c1.visited + c2.visited)  })  *//5.结果存入HBase* result.addSink(**new** SinkFunction[ChannelRealHot] {  **override def** invoke(value: ChannelRealHot, context: SinkFunction.Context): Unit = {  *//在这里调用HBaseUtil将每条结果(每个频道的访问次数),保存到HBase  //-1.先查HBase该频道的上次的访问次数* **val** tableName = **"channel\_realhot"  val** columnFamily = **"info"  val** queryColumn = **"visited"  val** rowkey = value.channelId  **val** historyValueStr: String = HBaseUtil.*getData*(tableName, rowkey, columnFamily, queryColumn)   **var** currentFinalResult = 0L   *//-2.判断并合并结果* **if** (StringUtils.*isBlank*(historyValueStr)) {  *//如果historyValueStr为空,直接让本次的次数作为本次最终的结果并保存* currentFinalResult = value.visited  } **else** {  *//如果historyValueStr不为空,本次的次数+历史值 作为本次最终的结果并保存* currentFinalResult = value.visited + historyValueStr.toLong  }   *//-3.存入本次最终的结果* HBaseUtil.*putData*(tableName, rowkey, columnFamily, queryColumn, currentFinalResult.toString)  }  })  } } |

#### 实时频道PV/UV

**PV(访问量)** 即Page View，页面刷新一次算一次。

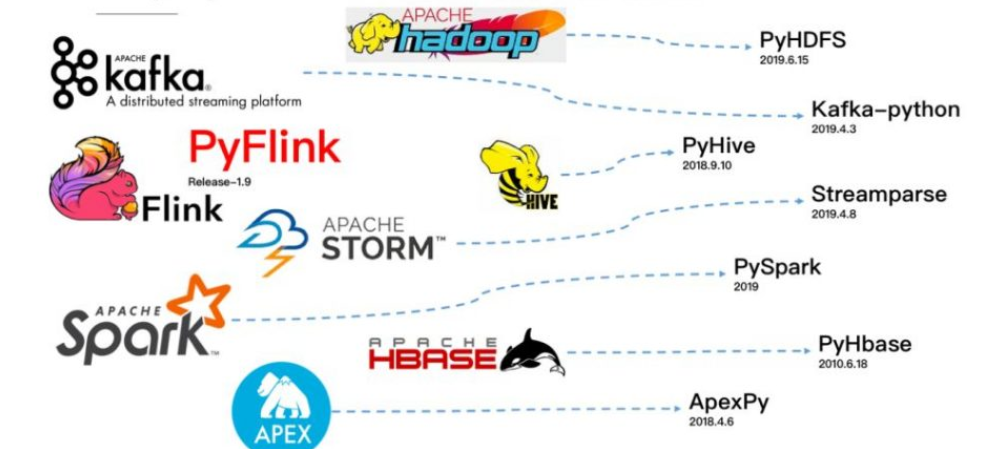
**UV(独立访客)** 即Unique Visitor，指定时间内相同的客户端只被计算一次

统计分析后得到的数据如下所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 频道ID | 时间 | PV | UV |
| 频道1 | 2017010116 | 1230 | 350 |
| 频道1 | 20170101 | 4251 | 530 |
| 频道1 | 201701 | 5512 | 610 |

|  |
| --- |
| **package** com.nhbd.flink.ch7.task  **import** com.nhbd.flink.ch7.bean.ClickLogWide **import** com.nhbd.flink.ch7.util.HBaseUtil **import** org.apache.flink.streaming.api.functions.sink.SinkFunction **import** org.apache.flink.streaming.api.scala.DataStream **import** org.apache.flink.streaming.api.windowing.time.Time  */\*\*  \*   \* Desc 实时频道分时段PV/UV统计,结果示例如下:  \* 频道ID 时间 PV UV  \* 频道1 202101 5512 610  \* 频道1 20210101 4251 530  \* 频道1 2021010116 1230 350  \* 注意:  \* pv可以使用条数统计  \* uv可以借助isxxNew字段  \*/* **object** ChannelRealPvUvTask {   **case class** ChannelRealPvUv(channelId: String, monthDayHour: String, pv: Long, uv: Long)   **def** process(clickLogWideDS: DataStream[ClickLogWide]) = {  **import** org.apache.flink.api.scala.\_  *//注意:  // 每条宽表日志都有: yearMonth,yearMonthDay,yearMonthDayHour这3个字段,  // 根据需求我们需要把1条日志根据这3个字段,变成3条数据,方便后面统计分时段PV/UV  // 也就是说现在要将每1条数据变为3条数据!  //使用flatMap  //中国北京昌平张三  // -->  //中国,张三  //中国北京,张三  //中国北京昌平,张三   //1.数据转换* **val** result: DataStream[ChannelRealPvUv] = clickLogWideDS.flatMap(clickLogWide => {  *List*(  *ChannelRealPvUv*(clickLogWide.channelID, clickLogWide.yearMonth, clickLogWide.count, clickLogWide.isMonthNew),  *ChannelRealPvUv*(clickLogWide.channelID, clickLogWide.yearMonthDay, clickLogWide.count, clickLogWide.isDayNew),  *ChannelRealPvUv*(clickLogWide.channelID, clickLogWide.yearMonthDayHour, clickLogWide.count, clickLogWide.isHourNew)  )  })  *//2.分组* .keyBy(**"channelId"**, **"monthDayHour"**)  *//3.窗口* .timeWindow(Time.*seconds*(10))  *//4.聚合* .reduce((c1, c2) => {  *ChannelRealPvUv*(c2.channelId, c2.monthDayHour, c1.pv + c2.pv, c1.uv + c2.uv)  })  *//5.结果保存到HBase  //注意:如果课下测试的时候,HBase性能跟不上,可以直接print打印能看到结果即可,下面的sink能看懂就行!  //result.print()* result.addSink(**new** SinkFunction[ChannelRealPvUv] {   **override def** invoke(value: ChannelRealPvUv, context: SinkFunction.Context): Unit = {  *//-1.查* **val** tableName = **"channel\_pvuv"  val** columnFamily = **"info"  val** queryColumn1 = **"pv"  val** queryColumn2 = **"uv"  val** rowkey = value.channelId + **":"** + value.monthDayHour   **val** map: Map[String, String] = HBaseUtil.*getMapData*(tableName,rowkey,columnFamily,*List*(queryColumn1,queryColumn2))   */\* val pvhistoryValueStr: String = map.getOrElse(queryColumn1,null)  val uvhistoryValueStr: String = map.getOrElse(queryColumn2,null)   //-2.合  var currentFinalPv = 0L  var currentFinalUv = 0L   if(StringUtils.isBlank(pvhistoryValueStr)){  //如果pvhistoryValueStr为空,直接将本次该频道该时段的pv 作为 该频道该时段的本次最终的结果  currentFinalPv = value.pv  }else{  //如果pvhistoryValueStr不为空,将本次该频道该时段的pv + pvhistoryValueStr 作为 该频道该时段的本次最终的结果  currentFinalPv = value.pv + pvhistoryValueStr.toLong  }   if(StringUtils.isBlank(uvhistoryValueStr)){  //如果uvhistoryValueStr为空,直接将本次该频道该时段的uv 作为 该频道该时段的本次最终的结果  currentFinalUv = value.uv  }else{  //如果uvhistoryValueStr不为空,将本次该频道该时段的uv + uvhistoryValueStr 作为 该频道该时段的本次最终的结果  currentFinalUv = value.uv + uvhistoryValueStr.toLong  }\*/* **val** pvhistoryValueStr: String = map.getOrElse(queryColumn1,**"0"**)  **val** uvhistoryValueStr: String = map.getOrElse(queryColumn2,**"0"**)   **val** currentFinalPv = value.pv + pvhistoryValueStr.toLong  **val** currentFinalUv = value.uv + uvhistoryValueStr.toLong   *//-3.存* HBaseUtil.*putMapData*(tableName,rowkey,columnFamily,  *Map*(  (queryColumn1,currentFinalPv),  (queryColumn2,currentFinalUv)  )  )  }  })  } } |

## PyFlink-略



### 环境准备

pip install apache-flink

需要在网络环境好的条件下安装,估计用时2小时左右,因为需要下载很多其他的依赖

### 官方文档

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/python/datastream_tutorial.html>

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/python/table_api_tutorial.html>

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/api/python/>

### 代码示例

|  |
| --- |
| **from** pyflink.common.serialization **import** SimpleStringEncoder **from** pyflink.common.typeinfo **import** Types **from** pyflink.datastream **import** StreamExecutionEnvironment **from** pyflink.datastream.connectors **import** StreamingFileSink   **def** tutorial():  env = StreamExecutionEnvironment.get\_execution\_environment()  env.set\_parallelism(1)  ds = env.from\_collection(  collection=[**"hadoop spark flink"**,**"hadoop spark"**,**"hadoop"**],  type\_info=Types.STRING()  )   ds.print()   result = ds.flat\_map(**lambda** line: line.split(**" "**), result\_type=Types.STRING())\  .map(**lambda** word: (word, 1),output\_type=Types.ROW([Types.STRING(), Types.INT()]))\  .key\_by(**lambda** x: x[0],key\_type\_info=Types.STRING())\  .reduce(**lambda** a, b: a + b)   result.print()   result.add\_sink(StreamingFileSink  .for\_row\_format(**'data/output/result1'**, SimpleStringEncoder())  .build())   env.execute(**"tutorial\_job"**)   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  tutorial() |

|  |
| --- |
| **from** pyflink.dataset **import** ExecutionEnvironment **from** pyflink.table **import** TableConfig, DataTypes, BatchTableEnvironment **from** pyflink.table.descriptors **import** Schema, OldCsv, FileSystem **from** pyflink.table.expressions **import** lit  exec\_env = ExecutionEnvironment.get\_execution\_environment() exec\_env.set\_parallelism(1) t\_config = TableConfig() t\_env = BatchTableEnvironment.create(exec\_env, t\_config)  t\_env.connect(FileSystem().path(**'data/input'**)) \  .with\_format(OldCsv()  .field(**'word'**, DataTypes.STRING())) \  .with\_schema(Schema()  .field(**'word'**, DataTypes.STRING())) \  .create\_temporary\_table(**'mySource'**)  t\_env.connect(FileSystem().path(**'/tmp/output'**)) \  .with\_format(OldCsv()  .field\_delimiter(**'\t'**)  .field(**'word'**, DataTypes.STRING())  .field(**'count'**, DataTypes.BIGINT())) \  .with\_schema(Schema()  .field(**'word'**, DataTypes.STRING())  .field(**'count'**, DataTypes.BIGINT())) \  .create\_temporary\_table(**'mySink'**)  tab = t\_env.from\_path(**'mySource'**) tab.group\_by(tab.word) \  .select(tab.word, lit(1).count) \  .execute\_insert(**'mySink'**).wait() |