

尚硅谷大数据项目之 Spark 实时(数据采集)

(作者：尚硅谷研究院)

版本：v2.0

第 1 章 简介

1.1 离线计算

离线计算一般指通过**批处理**的方式计算已知的所有输入数据，输入数据不会发生变化，一般**计算量级较大**，**计算时间较长**。

例如今天凌晨一点，把昨天累积的日志，计算出所需结果。最经典的就是 Hadoop 的 MapReduce 方式：

一般需要根据前一日的数据生成报表，虽然统计指标、报表繁多，但是对时效性不敏感。

1.1.1 离线计算的特点

- (1) 数据在计算前已经全部就位，不会发生变化
- (2) 数据量大且保存时间长
- (3) 在大量数据上进行复杂的批量运算
- (4) 方便的查看批量计算的结果

1.2 实时计算

实时计算一般是指通过**流处理**方式计算**当日的数据**都算是实时计算。

也会有一些准实时计算，利用离线框架通过批处理完成（小时、10 分钟级）的计算，一般为过渡产品，不能算是实时计算。

1.2.1 实时计算特点

1) 局部计算

1

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

每次计算以输入的**每条数据**，或者**微批次、小窗口**的数据范围进行计算，没法像离线数据一样能够基于当日全部数据进行统计排序分组。

2) 开发成本较高

相比离线的批处理 SQL，实时计算需要通过**代码**，往往需要对接多种数据容器完成，相对**开发较为复杂**。

3) 资源成本较高

实时计算虽然单位时间内数据量不如批处理，但是需要**24 小时**不停进行运行，一旦计算资源投入就无法释放，所以每个任务都要合理分配资源。

4) 时效性

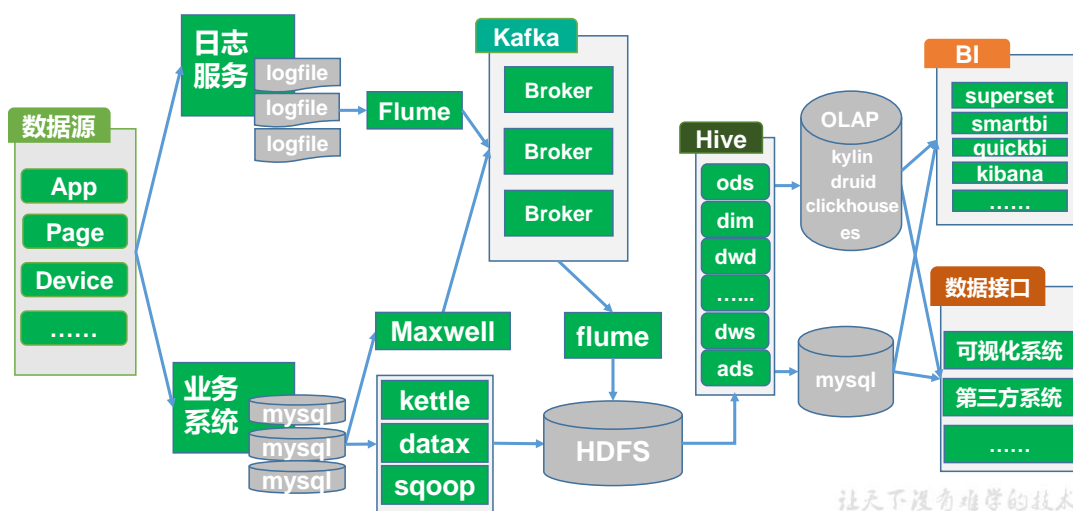
实时计算往往对时效性有一定的要求，所以要尽量优化整个计算链，减少计算过程中的中间环节。

5) 可视化性

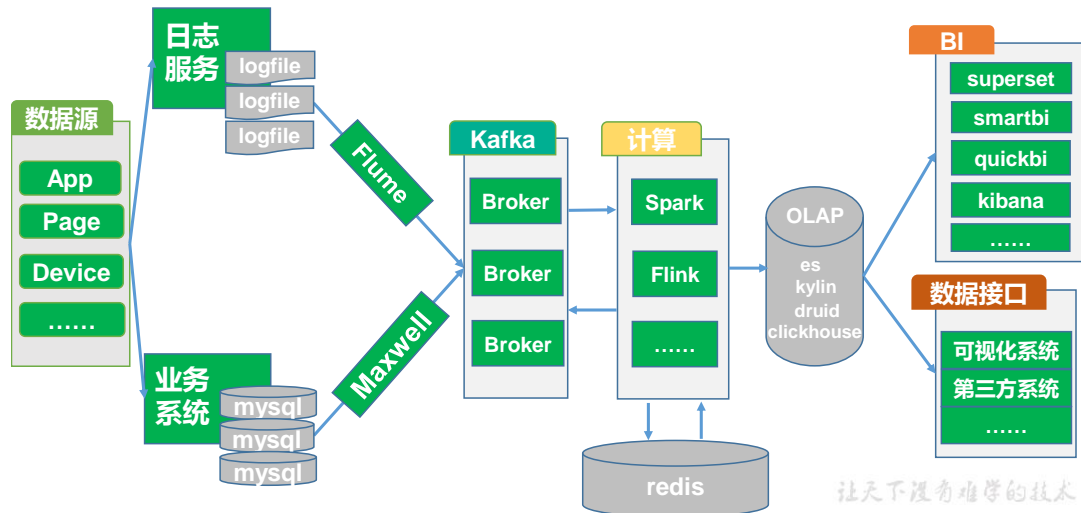
因为数据是不断变化的，所以相对于严谨整齐的离线报表，实时数据更寄希望通过图形化的手段能够及时的观察到**数据趋势**。

1.3 数仓架构设计

1.3.1 离线架构

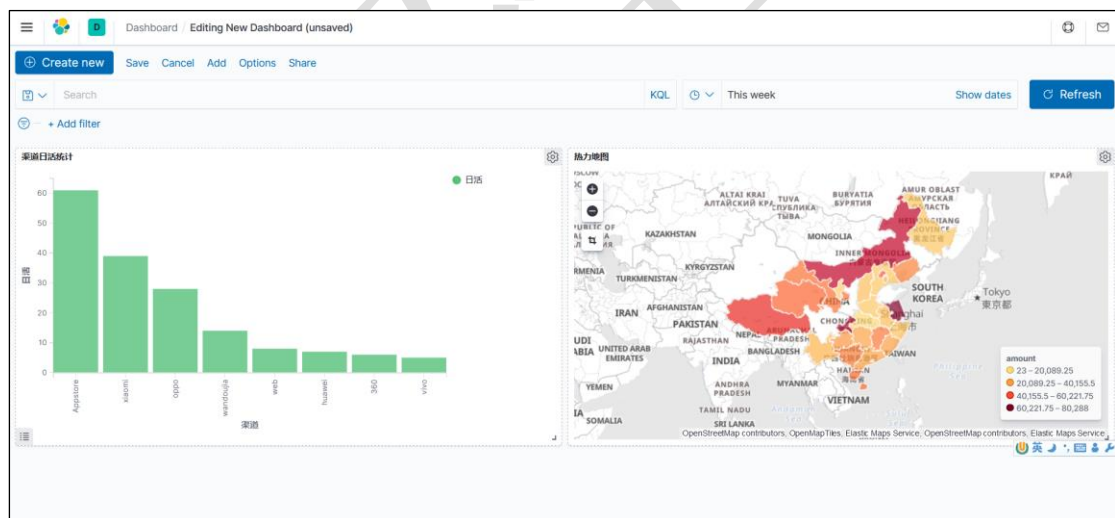


1.3.2 实时架构



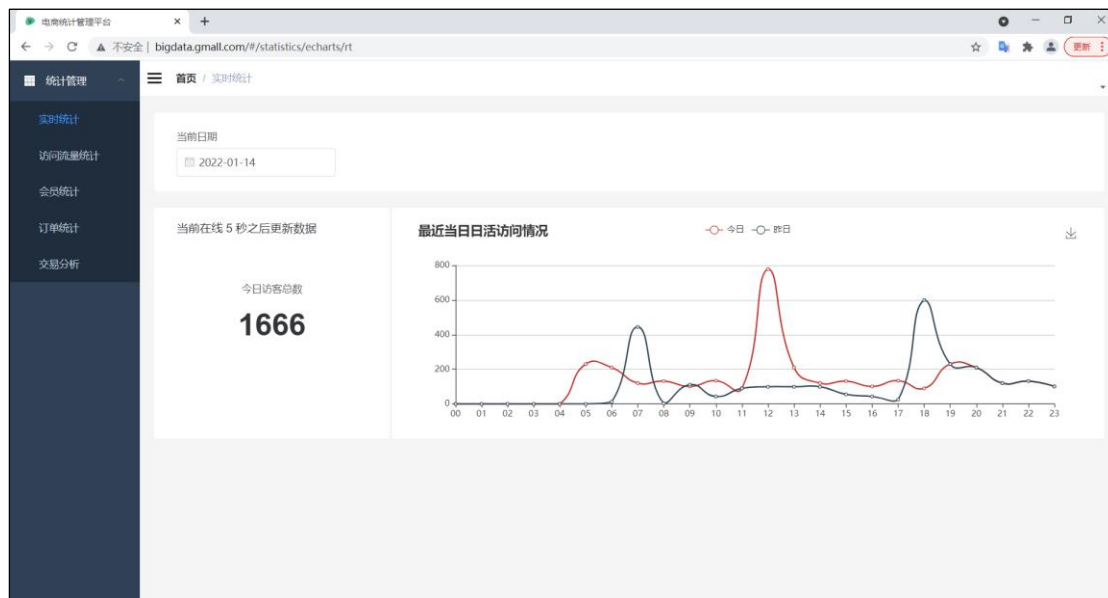
1.4 项目需求

1.4.1 BI 分析

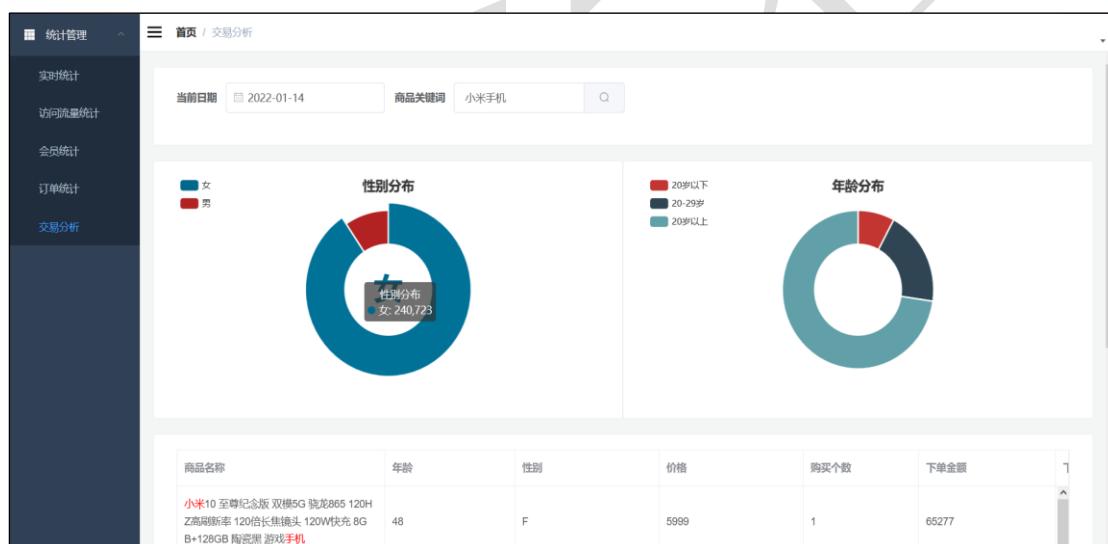


1.4.2 数据接口应用

1) 日活实时统计

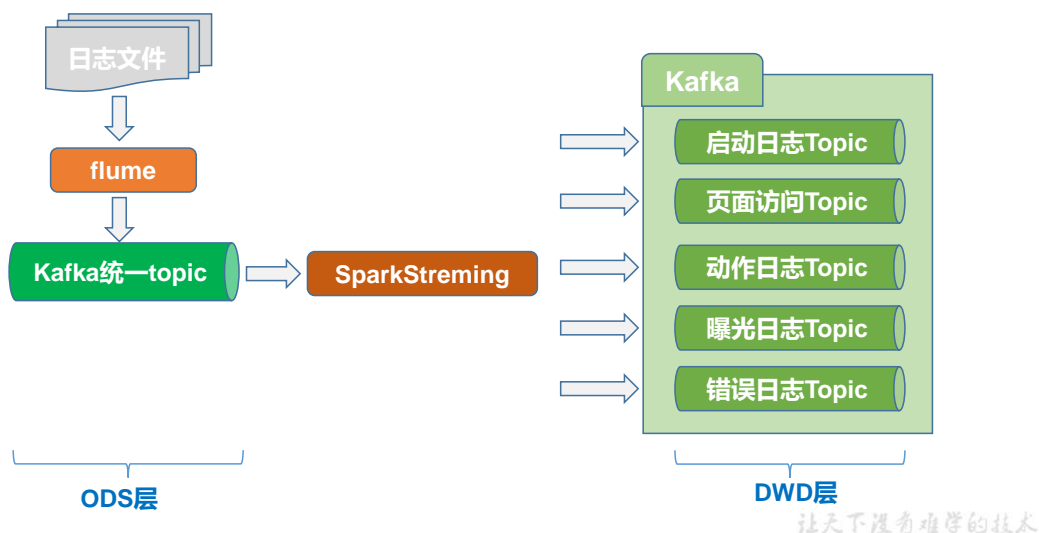


2) 订单交易分析



第 2 章 日志数据采集和分流

2.1 整体架构



2.2 采集日志数据

1) 上传模拟日志数据生成器到/opt/module/applog 目录

```

[atguigu@hadoop102 applog]$ ll
总用量 15288
-rw-r--r--. 1 atguigu atguigu 979 9月 18 09:24 application.yml
-rw-r--r--. 1 atguigu atguigu 15640308 2月 1 2021
gmall2020-mock-log-2021-01-22.jar
-rw-r--r--. 1 atguigu atguigu 1039 2月 1 2021 logback.xml
-rw-r--r--. 1 atguigu atguigu 565 2月 1 2021 path.json
  
```

2) 根据实际需要修改 application.yml 的如下配置

```

#业务日期
mock.date: "2020-06-20"

#模拟数据发送模式
mock.type: "kafka"

#kafka 模式下，发送的地址
mock:
  kafka-server: "hadoop102:9092,hadoop103:9092,hadoop104:9092"
  kafka-topic: "ODS_BASE_LOG"
  
```

3) 生成数据测试

```

[atguigu@hadoop102 applog]$ java -jar
gmall2020-mock-log-2021-01-22.jar
  
```

4) 消费数据测试

```
[atguigu@hadoop102 applog]$ kafka-console-consumer.sh
--bootstrap-server=hadoop102:9092,hadoop103:9092,hadoop104:9092 --
topic ODS_BASE_LOG
```

2.3 辅助脚本

1) 模拟生成数据脚本 log.sh

```
#!/bin/bash
#根据传递的日期参数修改配置文件的日期
if [ $# -ge 1 ]
then
    sed -i "/mock.date/c mock.date: $1"
/opt/module/applog/application.yml
fi
cd /opt/module/applog;java -jar
gmall2020-mock-log-2021-11-29.jar >/dev/null 2>&1 &
```

2) Kafka 脚本

```
#!/bin/bash
if [ $# -lt 1 ]
then
    echo "Usage: kf.sh {start|stop|kc [topic]|kp [topic] |list |delete
[topic] |describe [topic]}"
    exit
fi
case $1 in
start)
    for i in hadoop102 hadoop103 hadoop104
    do
        echo "=====> START $i KF <====="
        ssh $i kafka-server-start.sh -daemon
/opt/module/kafka_2.11-2.4.1/config/server.properties
    done
    ;;
stop)
    for i in hadoop102 hadoop103 hadoop104
    do
        echo "=====> STOP $i KF <====="
        ssh $i kafka-server-stop.sh
    done
    ;;
kc)
    if [ $2 ]
    then
        kafka-console-consumer.sh --bootstrap-server
hadoop102:9092,hadoop103:9092,hadoop104:9092 --topic $2
    else
        echo "Usage: kf.sh {start|stop|kc [topic]|kp [topic] |list
|delete [topic] |describe [topic]}"
    fi
    ;;
kp)

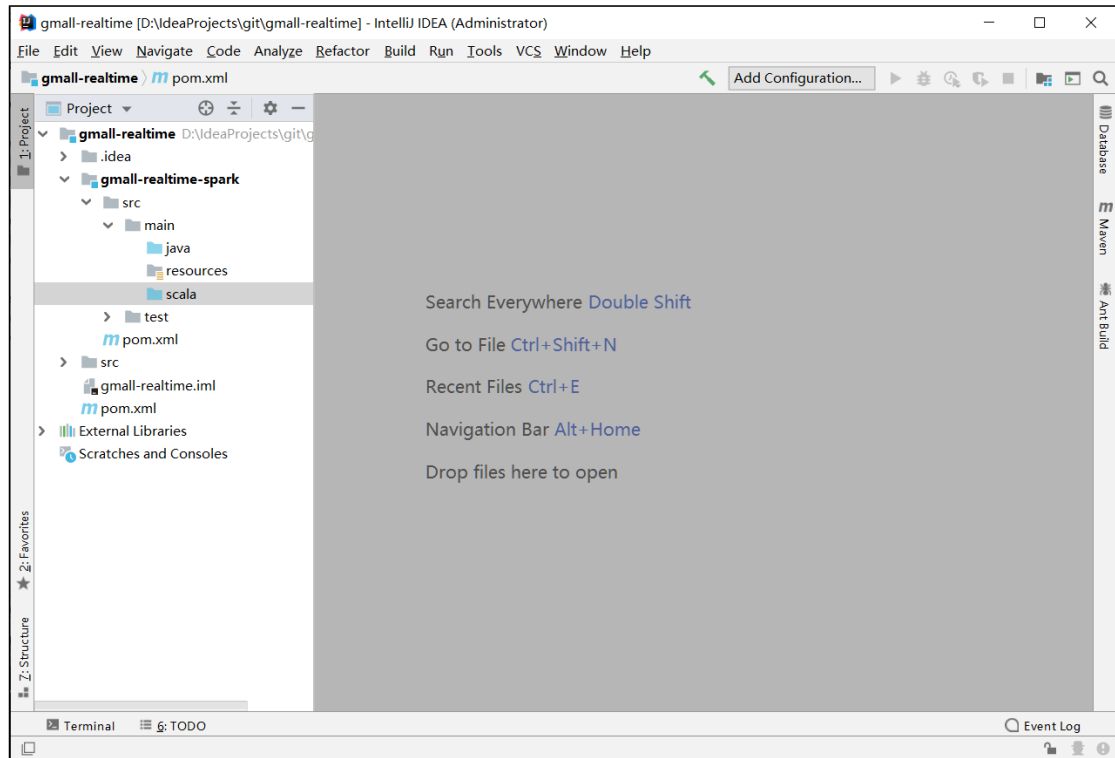
```

```
if [ $2 ]
then
    kafka-console-producer.sh          --broker-list
hadoop102:9092,hadoop103:9092,hadoop104:9092 --topic $2
else
    echo "Usage: kf.sh {start|stop|kc [topic]|kp [topic] |list
|delete [topic] |describe [topic]}"
fi
;;

list)
    kafka-topics.sh          --list          --bootstrap-server
hadoop102:9092,hadoop103:9092,hadoop104:9092
;;
describe)
    if [ $2 ]
    then
        kafka-topics.sh          --describe          --bootstrap-server
hadoop102:9092,hadoop103:9092,hadoop104:9092 --topic $2
    else
        echo "Usage: kf.sh {start|stop|kc [topic]|kp [topic] |list
|delete [topic] |describe [topic]}"
    fi
;;
delete)
    if [ $2 ]
    then
        kafka-topics.sh          --delete          --bootstrap-server
hadoop102:9092,hadoop103:9092,hadoop104:9092 --topic $2
    else
        echo "Usage: kf.sh {start|stop|kc [topic]|kp [topic] |list
|delete [topic] |describe [topic]}"
    fi
;;
*)
    echo "Usage: kf.sh {start|stop|kc [topic]|kp [topic] |list |delete
[topic] |describe [topic]}"
    exit
;;
esac
```

2.4 准备开发环境

2.4.1 创建工程



2.4.2 添加依赖

```
<properties>
  <spark.version>3.0.0</spark.version>
  <scala.version>2.12.11</scala.version>
  <kafka.version>2.4.1</kafka.version>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</project>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.62</version>
  </dependency>

  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.12</artifactId>
```



```
        <version>${spark.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-streaming_2.12</artifactId>
        <version>${spark.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>${kafka.version}</version>

    </dependency>
    <dependency>
        <groupId>org.apache.spark</groupId>
<artifactId>spark-streaming-kafka-0-10_2.12</artifactId>
        <version>${spark.version}</version>
    </dependency>

    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.12</artifactId>
        <version>${spark.version}</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.11.0</version>
    </dependency>

    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-to-slf4j</artifactId>
        <version>2.11.0</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>

    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>3.3.0</version>
    </dependency>

    <dependency>
        <groupId>org.elasticsearch</groupId>
        <artifactId>elasticsearch</artifactId>
        <version>7.8.0</version>
    </dependency>
```

```
<dependency>
  <groupId>org.elasticsearch.client</groupId>
<artifactId>elasticsearch-rest-high-level-client</artifactId>
  <version>7.8.0</version>
</dependency>

<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
  <version>4.5.10</version>
</dependency>

</dependencies>

<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>compile</goal>
            <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>
```

2.4.3 添加配置文件

1) 添加 config.properties 文件

```
# Kafka 配置
kafka.broker.list=hadoop102:9092,hadoop103:9092,hadoop104:9092
```

2) 添加 log4j.properties

```
log4j.appender.atguigu.MyConsole=org.apache.log4j.ConsoleAppender
log4j.appender.atguigu.MyConsole.target=System.out
log4j.appender.atguigu.MyConsole.layout=org.apache.log4j.PatternLayout
log4j.appender.atguigu.MyConsole.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %10p (%c:%M) - %m%n
log4j.rootLogger =error,atguigu.MyConsole
```

2.4.4 添加工具类

1) properties 工具类

```
package com.atguigu.gmall.realtime.util
import java.util.ResourceBundle

/**
 * 配置解析类
 */
object PropertiesUtils {

    private val bundle: ResourceBundle =
ResourceBundle.getBundle("config")

    def apply(key : String ):String ={
        bundle.getString(key)
    }

    def main(args: Array[String]): Unit = {
        println(PropertiesUtils("kafka.broker.list"))
    }
}
```

2) Kafka 工具类

```
package com.atguigu.gmall.realtime.util

import java.util.Properties

import org.apache.kafka.clients.consumer.{ConsumerConfig,
ConsumerRecord}
import org.apache.kafka.clients.producer
import org.apache.kafka.clients.producer.{KafkaProducer,
ProducerConfig, ProducerRecord}
import org.apache.kafka.common.TopicPartition
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.dstream.InputDStream
import org.apache.spark.streaming.kafka010.{ConsumerStrategies,
KafkaUtils, LocationStrategies}
```

```
import scala.collection.mutable

/**
 * Kafka 工具类，用于生产和消费
 */
object MyKafkaUtils {

    //kafka 消费配置
    private val consumerConfig: mutable.Map[String, String] =
mutable.Map(
    ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG ->
PropertiesUtils("kafka.bootstrap.servers"),
    ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG ->
"org.apache.kafka.common.serialization.StringDeserializer",
    ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG ->
"org.apache.kafka.common.serialization.StringDeserializer",
    ConsumerConfig.GROUP_ID_CONFIG -> "gmall",
    ConsumerConfig.AUTO_OFFSET_RESET_CONFIG -> "latest",
    ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG -> "true"
)

    /**
     * 默认 offsets 位置消费
     */
    def getKafkaDStream(topic:String,ssc:
StreamingContext,groupId:String) :
InputDStream[ConsumerRecord[String, String]] ={
        consumerConfig(ConsumerConfig.GROUP_ID_CONFIG) = groupId
        val dStream: InputDStream[ConsumerRecord[String, String]] =
KafkaUtils.createDirectStream[String, String](
            ssc,
            LocationStrategies.PreferConsistent,
            ConsumerStrategies.Subscribe[String, String](Array(topic),
consumerConfig)
        )
        dStream
    }

    /**
     * 指定 offsets 位置消费
     */
    def getKafkaDStream(topic:String,ssc: StreamingContext,offsets:
Map[TopicPartition,Long],groupId : String ) :
InputDStream[ConsumerRecord[String, String]] ={
        consumerConfig(ConsumerConfig.GROUP_ID_CONFIG) = groupId
        val dStream: InputDStream[ConsumerRecord[String, String]] =
KafkaUtils.createDirectStream[String, String](
            ssc,
            LocationStrategies.PreferConsistent,
            ConsumerStrategies.Subscribe[String, String](Array(topic),
consumerConfig,offsets)
        )
        dStream
    }
}
```

```
/**
 * 创建 Kafka 生产者对象
 */
def createKafkaProducer():KafkaProducer[String,String] = {
  //Kafka 生产配置
  val props = new Properties()

  props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, PropertiesUtils
    ("kafka.bootstrap.servers"))

  props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.
    kafka.common.serialization.StringSerializer")

  props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "org.apach
    e.kafka.common.serialization.StringSerializer")
  props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true")

  val producer:KafkaProducer[String,String] = new
    KafkaProducer[String,String](props)
  producer
}

private var producer : KafkaProducer[ String , String ] =
  createKafkaProducer()

/**
 * 生产
 */
def send(topic:String, msg :String):Unit={
  producer.send(new ProducerRecord[String,String](topic,msg))
}

/**
 * 生产 指定 key
 */
def send(topic:String, key: String , msg :String):Unit = {
  producer.send(new ProducerRecord[String,String](topic,key,
    msg))
}

/**
 * 刷写缓冲区
 */

def flush(): Unit = {
  if(producer != null ) producer.flush()
}

/**
 * 关闭生产者对象
 */
def close():Unit = {
  if(producer != null ) producer.close()
}
```

```
}
```

2.5 日志数据消费分流

2.5.1 相关实体 bean

1) 页面日志 Bean

```
case class PageLog(  
    mid :String,  
    user_id:String,  
    province_id:String,  
    channel:String,  
    is_new:String,  
    model:String,  
    operate_system:String,  
    version_code:String,  
    page_id:String ,  
    last_page_id:String,  
    page_item:String,  
    page_item_type:String,  
    during_time:Long,  
    ts:Long  
) {  
  
}
```

2) 启动日志 Bean

```
case class StartLog(  
mid :String,  
    user_id:String,  
    province_id:String,  
    channel:String,  
    is_new:String,  
    model:String,  
    operate_system:String,  
    version_code:String,  
    entry:String,  
    open_ad_id:String,  
    loading_time_ms:Long,  
    open_ad_ms:Long,  
    open_ad_skip_ms:Long,  
    ts:Long) {  
  
}
```

3) 页面动作日志 Bean

```
case class PageActionLog(  
    mid :String,  
    user_id:String,  
    province_id:String,  
    channel:String,  
    is_new:String,  
    model:String,  
    operate_system:String,
```

```

        version_code:String,
        page_id:String ,
        last_page_id:String,
        page_item:String,
        page_item_type:String,
        during_time:Long,
        action_id:String,
        action_item:String,
        action_item_type:String,
        ts:Long
    ) {
}

```

4) 页面曝光日志 Bean

```

case class PageDisplayLog (
    mid :String,
    user_id:String,
    province_id:String,
    channel:String,
    is_new:String,
    model:String,
    operate_system:String,
    version_code:String,
    page_id:String ,
    last_page_id:String,
    page_item:String,
    page_item_type:String,
    during_time:Long,
    display_type:String,
    display_item: String,
    display_item_type:String,
    display_order:String ,
    display_pos_id:String,
    ts:Long
)

```

2.5.2 消费分流

1) 代码

```

package com.atguigu.gmall.realtime.app

import com.alibaba.fastjson.serializer.SerializeConfig
import com.alibaba.fastjson.{JSON, JSONArray, JSONObject}
import com.atguigu.gmall.realtime.bean.{PageActionLog,
PageDisplayLog, PageLog, StartLog}
import com.atguigu.gmall.realtime.util.MyKafkaUtils
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.dstream.{DStream, InputDStream}
import org.apache.spark.streaming.{Seconds, StreamingContext}

/**
 * 消费分流

```

```
* 1. 接收 Kafka 数据流
*
* 2. 转换数据结构:
*     通用的数据结构: Map 或者 JsonObject
*     专用的数据结构: bean
*
* 3. 分流 : 将数据拆分到不同的主题中
*     启动主题: DWD_START_LOG
*     页面访问主题: DWD_PAGE_LOG
*     页面动作主题: DWD_PAGE_ACTION
*     页面曝光主题: DWD_PAGE_DISPLAY
*     错误主题: DWD_ERROR_INFO
*/
object BaseLogApp {

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

    //创建配置对象
    val sparkConf: SparkConf = new
SparkConf().setAppName("base_log_app").setMaster("local[4]")
    val ssc = new StreamingContext(sparkConf, Seconds(5))

    //原始主题
    val ods_base_topic : String = "ODS_BASE_LOG"
    //启动主题
    val dwd_start_log : String = "DWD_START_LOG"
    //页面访问主题
    val dwd_page_log : String = "DWD_PAGE_LOG"
    //页面动作主题
    val dwd_page_action : String = "DWD_PAGE_ACTION"
    //页面曝光主题
    val dwd_page_display : String = "DWD_PAGE_DISPLAY"
    //错误主题
    val dwd_error_info : String = "DWD_ERROR_INFO"

    //消费组
    val group_id : String = "ods_base_log_group"

    //1. 接收 Kafka 数据流
    val kafkaDStream: InputDStream[ConsumerRecord[String, String]] =
      MyKafkaUtils.getKafkaDStream(ods_base_topic, ssc, group_id)

    //kafkaDStream.map(_._value()).print(3)

    //2. 转换数据结构
    val jsonDStream: DStream[JsonObject] = kafkaDStream.map(
      record => {

        val value: String = record.value()
        println(value)
        val jsonObject: JsonObject = JSON.parseObject(value)
      }
    )
  }
}
```



```
        jsonObject
    }
)

//3.切分数据分流
jsonDStream.foreachRDD(
    rdd => {
        rdd.foreach(
            jsonObj => {
                //TODO 错误日志
                //分流错误日志 判断是否是错误日志, 如果是错误日志, 直接发送
                val errorObj: JSONObject = jsonObj.getJSONObject("err")
                if(errorObj != null){
                    MyKafkaUtils.send(dwd_error_info, jsonObj.toJSONString)
                }else{
                    //提取公共信息
                    val commonObj: JSONObject = jsonObj.getJSONObject("common")
                    val mid: String = commonObj.getString("mid")
                    val uid: String = commonObj.getString("uid")
                    val ar: String = commonObj.getString("ar")
                    val ch: String = commonObj.getString("ch")
                    val os: String = commonObj.getString("os")
                    val md: String = commonObj.getString("md")
                    val vc: String = commonObj.getString("vc")
                    val isNew: String = commonObj.getString("is_new")
                    val ts: Long = jsonObj.getLong("ts")

                    //分流页面日志
                    val pageObj: JSONObject = jsonObj.getJSONObject("page")
                    if(pageObj != null ){
                        //提取字段
                        val pageId: String = pageObj.getString("page_id")
                        val pageItem: String = pageObj.getString("item")
                        val pageItemType: String = pageObj.getString("item_type")
                        val lastPageId: String = pageObj.getString("last_page_id")
                        val duringTime: Long = pageObj.getLong("during_time")

                        //封装 bean
                        val pageLog = PageLog(mid, uid, ar, ch, isNew, md, os, vc, pageId, lastPageId, pageItem, pageItemType, duringTime, ts)
                        //发送 kafka
                        MyKafkaUtils.send(dwd_page_log, JSON.toJSONString(pageLog, new SerializeConfig(true)))

                        //分流动作日志
                        val actionArrayObj: JSONArray = jsonObj.getJSONArray("actions")
                        if(actionArrayObj != null && actionArrayObj.size() > 0 ){
                            for(i <- 0 until actionArrayObj.size() ){
```

```
        val actionObj: JSONObject =
actionArrayObj.getJSONObject(i)
        val actionId: String =
actionObj.getString("action_id")
        val actionItem: String =
actionObj.getString("item")
        val actionItemType: String =
actionObj.getString("item_type")
        //TODO actions
        val actionTs: Long = actionObj.getLong("ts")

        //封装 Bean
        val pageActionLog =
            PageActionLog(mid, uid, ar, ch, isNew, md, os, vc,
pageId, lastPageId, pageItem, pageItemType,
duringTime, actionId, actionItem, actionItemType, actionTs, ts)

        //发送 Kafka

MyKafkaUtils.send(dwd_page_action, JSON.toJSONString(pageActionLog
,new SerializeConfig(true)))
    }
}

//分流曝光日志
val displayArrayObj: JSONArray =
jsonObj.getJSONArray("displays")
if(displayArrayObj != null && displayArrayObj.size() >
0 ){
    for(i <- 0 until displayArrayObj.size()){
        val displayObj: JSONObject =
displayArrayObj.getJSONObject(i)
        val displayType: String =
displayObj.getString("display_type")
        val displayItem: String =
displayObj.getString("item")
        val displayItemType: String =
displayObj.getString("item_type")
        val displayOrder: String =
displayObj.getString("order")
        val displayPosId: String =
displayObj.getString("pos_id")

        //封装 Bean
        val displayLog = PageDisplayLog(mid, uid, ar, ch,
isNew, md, os, vc, pageId, lastPageId, pageItem, pageItemType,
duringTime, displayType, displayItem, displayItemType, displayOrder, d
isplayPosId, ts)

        //发送 Kafka

MyKafkaUtils.send(dwd_page_display, JSON.toJSONString(displayLog, n
ew SerializeConfig(true)))
    }
}
```

```

    }

    //分流启动日志
    val startObj: JSONObject =
jsonObj.getJSONObject("start")
    if(startObj != null ){
        val entry: String = startObj.getString("entry")
        val loadingTimeMs: Long =
startObj.getLong("loading_time_ms")
        val openAdId: String =
startObj.getString("open_ad_id")
        val openAdMs: Long = startObj.getLong("open_ad_ms")
        val openAdSkipMs: Long =
startObj.getLong("open_ad_skip_ms")

        //封装 Bean
        val startLog =
StartLog(mid,uid,ar,ch,isNew,md,os,vc,entry,openAdId,loadingTimeM
s,openAdMs,openAdSkipMs,ts)
        //发送 Kafka

MyKafkaUtils.send(dwd_start_log,JSON.toJSONString(startLog,new
SerializeConfig(true)))
    }
}
}
)
}
)

ssc.start()
ssc.awaitTermination()
}
}

```

2.6 优化 - 精确一次消费

2.6.1 相关语义

1) 至少一次消费 (at least once)

主要是保证数据不会丢失，但有可能存在数据重复问题。

2) 最多一次消费 (at most once)

主要是保证数据不会重复，但有可能存在数据丢失问题。

3) 精确一次消费 (Exactly-once)

指消息一定会被处理且只会被处理一次。不多不少就一次处理。如果达不到精确一次消费，可能会达到另外两种情况。

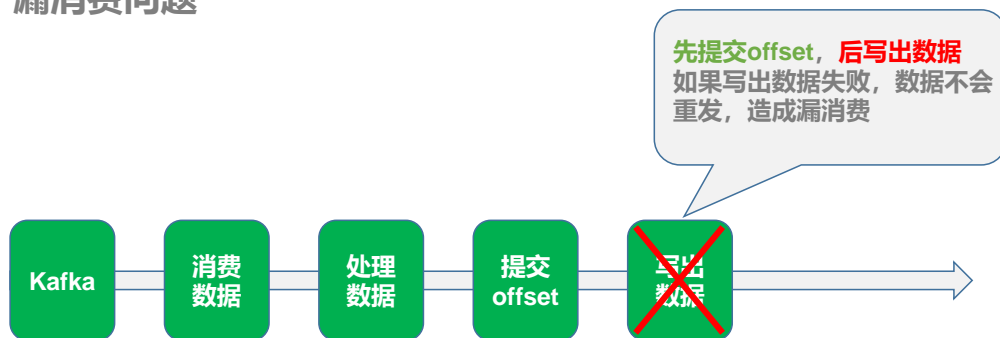
2.6.2 消费问题

1) 漏消费（丢失数据）

比如实时计算任务进行计算，到数据结果存盘之前，进程崩溃，假设在进程崩溃前 kafka 调整了偏移量，那么 kafka 就会认为数据已经被处理过，即使进程重启，kafka 也会从新的偏移量开始，所以之前没有保存的数据就被丢失掉了。



漏消费问题



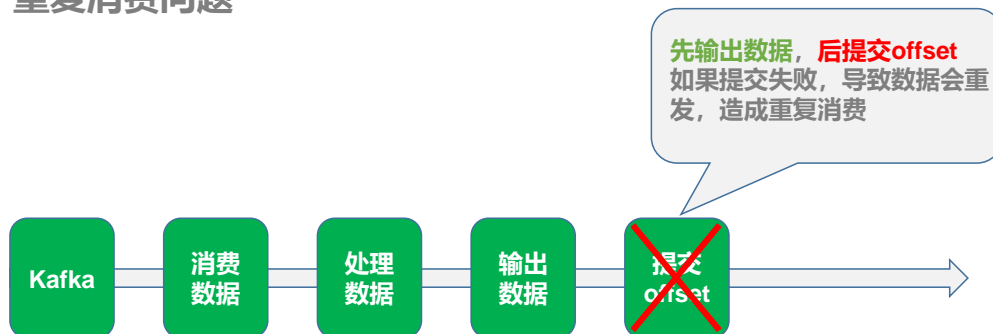
让天下没有难学的技术

2) 重复消费（重复计算）

如果数据计算结果已经存盘了，在 kafka 调整偏移量之前，进程崩溃，那么 kafka 会认为数据没有被消费，进程重启，会重新从旧的偏移量开始，那么数据就会被 2 次消费，又会被存盘，数据就被存了 2 遍，造成数据重复。



重复消费问题



让天下没有难学的技术

3) 如果同时解决了数据丢失和数据重复的问题, 那么就实现了精确一次消费的语义了。

目前 Kafka 默认每 5 秒钟做一次自动提交偏移量, 这样并不能保证精准一次消费。

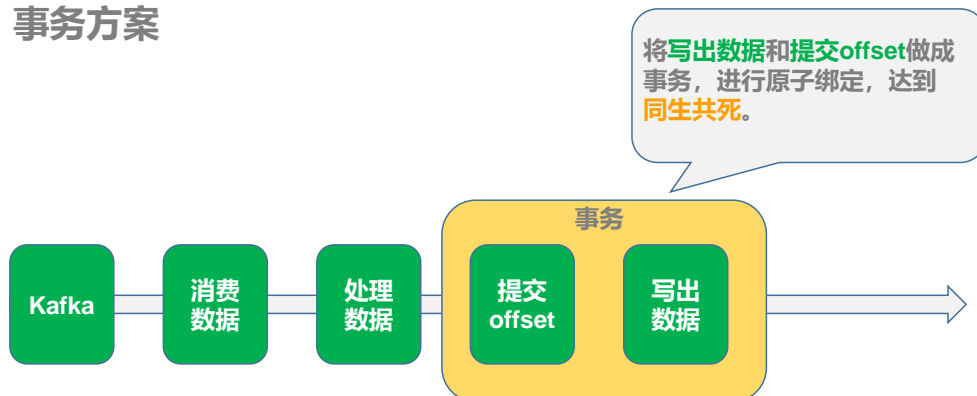
`enable.auto.commit` 的默认值是 `true`; 就是默认采用自动提交的机制。

`auto.commit.interval.ms` 的默认值是 5000, 单位是毫秒。

2.6.3 问题解决-策略一



事务方案



让天下没有难学的技术

1) 策略: 利用关系型数据库的事务进行处理

出现丢失或者重复的问题, 核心就是偏移量的提交与数据的保存, 不是原子性的。如果能做成要么数据保存和偏移量都成功, 要么两个失败, 那么就不会出现丢失或者重复了。

这样的话可以把存数据和修改偏移量放到一个事务里。这样就做到前面的成功, 如果后

面做失败了,就回滚前面那么就达成了原子性,这种情况先存数据还是先修改偏移量没影响。

2) 好处

事务方式能够保证精准一次性消费

3) 问题与限制

(1) 数据必须都要放在一个关系型数据库中,无法使用其他功能强大的 nosql 数据库

(2) 事务本身性能不好

(3) 如果保存的数据量较大一个数据库节点不够,多个节点的话,还要考虑分布式事务的问题。分布式事务会带来管理的复杂性,一般企业不选择使用,有的企业会把分布式事务变成本地事务,例如把 Executor 上的数据通过 `rdd.collect` 算子提取到 Driver 端,由 Driver 端统一写入数据库,这样会将分布式事务变成本地事务的单线程操作,降低了写入的吞吐量。

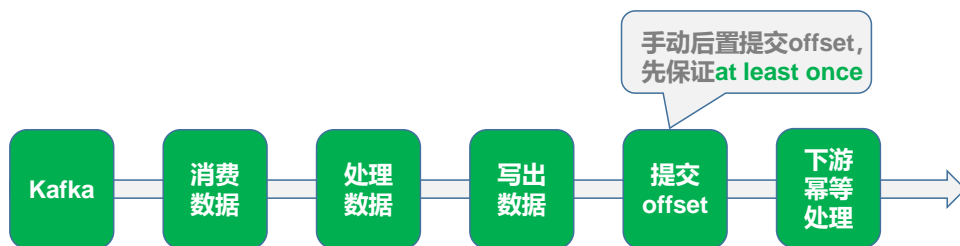
4) 使用场景

数据足够少(通常经过聚合后的数据量都比较小,明细数据一般数据量都比较大),并且支持事务的数据库。

2.6.4 问题解决-策略二



后置提交offset + 幂等方案



让天下没有难学的技术

1) 策略:手动提交偏移量 + 幂等性处理

我们知道如果能够同时解决数据丢失和数据重复问题,就等于做到了精确一次消费。那就各个击破。

首先解决数据丢失问题，办法就是要等数据保存成功后再提交偏移量，所以就必须手工来控制偏移量的提交时机。

但是如果数据保存了，没等偏移量提交进程挂了，数据会被重复消费。怎么办？那就要把数据的保存做成幂等性保存。即同一批数据反复保存多次，数据不会翻倍，保存一次和保存一百次的效果是一样的。如果能做到这个，就达到了幂等性保存，就不用担心数据会重复了。

2) 难点

话虽如此，在实际的开发中手动提交偏移量其实不难，难的是幂等性的保存，有的时候并不一定能保证，这个需要看使用的数据库，如果数据库本身不支持幂等性操作，那只能优先保证的数据不丢失，数据重复难以避免，即只保证了至少一次消费的语义。

一般有主键的数据库都支持幂等性操作 upsert。

3) 使用场景

处理数据较多，或者数据保存在不支持事务的数据库上。

2.6.5 手动提交偏移流程

1) 偏移量保存在哪

kafka 0.9 版本以后 consumer 的偏移量是保存在 kafka 的 __consumer_offsets 主题中。但是如果用这种方式管理偏移量，有一个限制就是在提交偏移量时，数据流的元素结构不能发生转变，即提交偏移量时数据流，必须是 `InputDStream[ConsumerRecord[String, String]]` 这种结构。但是在实际计算中，数据难免发生转变，或聚合，或关联，一旦发生转变，就无法在利用以下语句进行偏移量的提交：

```
xxDstream.asInstanceOf[CanCommitOffsets].commitAsync(offsetRanges)
```

因为 offset 的存储于 `HasOffsetRanges`，只有 `kafkaRDD` 继承了他，所以假如我们对 `KafkaRDD` 进行了转化之后，其它 `RDD` 没有继承 `HasOffsetRanges`，所以就无法再获取 offset 了。

所以实际生产中通常会利用 ZooKeeper, Redis, Mysql 等工具手动对偏移量进行保存

2) 流程



offset管理方案



让天下没有难学的技术

2.6.6 代码实现

1) 在 config.properties 中添加 redis 的连接配置

```
# Kafka 通用配置
kafka.bootstrap.servers=hadoop102:9092,hadoop103:9092,hadoop104:9092

#Redis 配置
redis.host=hadoop102
redis.port=6379
```

2) 添加 Redis 工具类

```
package com.atguigu.gmall.realtime.util

import redis.clients.jedis.{Jedis, JedisPool, JedisPoolConfig}

object MyRedisUtils {

  var jedisPool : JedisPool = null

  def getJedisClient : Jedis = {
    if(jedisPool == null ){
      var host : String = PropertiesUtils("redis.host")
      var port : String = PropertiesUtils("redis.port")
      val jedisPoolConfig = new JedisPoolConfig()
      jedisPoolConfig.setMaxTotal(100) //最大连接数
      jedisPoolConfig.setMaxIdle(20) //最大空闲
      jedisPoolConfig.setMinIdle(20) //最小空闲
      jedisPoolConfig.setBlockWhenExhausted(true) //忙碌时是否等待
      jedisPoolConfig.setMaxWaitMillis(5000) //忙碌时等待时长 毫秒
      jedisPoolConfig.setTestOnBorrow(true) //每次获得连接的进行测试

      jedisPool = new JedisPool(jedisPoolConfig,host,port.toInt)
    }
  }
```



```
jedisPool.getResource
}

}
```

3) 添加偏移量工具类

```
object OffsetManagerUtil {

    /**
     * 从 Redis 中读取偏移量
     * Redis 格式 : type=>Hash [key=>offset:topic:groupId
     field=>partitionId value=>偏移量值] expire 不需要指定
     *
     * @param topicName 主题名称
     * @param groupId 消费者组
     * @return 当前消费者组中, 消费的主题对应的分区的偏移量信息
     * KafkaUtils.createDirectStream 在读取数据的时候封装了
     Map[TopicPartition, Long]
     */
    def getOffset(topicName: String, groupId: String):
    Map[TopicPartition, Long] = {
        //获取 Redis 客户端
        val jedis: Jedis = RedisUtil.getJedisClient
        //拼接 Reids 中存储偏移量的 key
        val offsetKey: String = "offset:" + topicName + ":" + groupId
        //根据 key 从 Reids 中获取数据
        val offsetMap: java.util.Map[String, String] =
        jedis.hgetAll(offsetKey)
        //关闭客户端
        jedis.close()
        //将 Java 的 Map 转换为 Scala 的 Map, 方便后续操作
        import scala.collection.JavaConverters._
        val kafkaOffsetMap: Map[TopicPartition, Long] =
        offsetMap.asScala.map {
            case (partitionId, offset) => {
                println("读取分区偏移量: " + partitionId + ":" + offset)
                //将 Redis 中保存的分区对应的偏移量进行封装
                (new TopicPartition(topicName, partitionId.toInt),
                offset.toLong)
            }
        }.toMap
        kafkaOffsetMap
    }

    /**
     * 向 Redis 中保存偏移量
     * Reids 格式 : type=>Hash [key=>offset:topic:groupId
     field=>partitionId value=>偏移量值] expire 不需要指定
     *
     * @param topicName 主题名
     * @param groupId 消费者组
     * @param offsetRanges 当前消费者组中, 消费的主题对应的分区的偏移量起
     始和结束信息
    
```

```

*/
def saveOffset(topicName: String, groupId: String, offsetRanges:
Array[OffsetRange]): Unit = {

    //定义 Java 的 map 集合, 用于向 Reids 中保存数据
    val offsetMap: java.util.HashMap[String, String] = new
java.util.HashMap[String, String]()
    //对封装的偏移量数组 offsetRanges 进行遍历
    for (offset <- offsetRanges) {
        //获取分区
        val partition: Int = offset.partition
        //获取结束点
        val untilOffset: Long = offset.untilOffset
        //封装到 Map 集合中
        offsetMap.put(partition.toString, untilOffset.toString)
        //打印测试
        println("保存分区:" + partition + ":" + offset.fromOffset +
"--->" + offset.untilOffset)
    }

    //拼接 Reids 中存储偏移量的 key
    val offsetKey: String = "offset:" + topicName + ":" + groupId

    //如果需要保存的偏移量不为空 执行保存操作
    if (offsetMap != null && offsetMap.size() > 0) {
        //获取 Redis 客户端
        val jedis: Jedis = RedisUtil.getJedisClient

        //保存到 Redis 中
        jedis.hmset(offsetKey, offsetMap)
        //关闭客户端
        jedis.close()
    }
}
}

```

4) 修补主程序的代码

```

package com.atguigu.gmall.realtime.app

import com.alibaba.fastjson.serializer.SerializeConfig
import com.alibaba.fastjson.{JSON, JSONArray, JSONObject}
import com.atguigu.gmall.realtime.bean.{PageActionLog,
PageDisplayLog, PageLog, StartLog}
import com.atguigu.gmall.realtime.util.{MyKafkaUtils,
MyOffsetUtils}
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.TopicPartition
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.dstream.{DStream, InputDStream}
import org.apache.spark.streaming.kafka010.{HasOffsetRanges,
OffsetRange}
import org.apache.spark.streaming.{Seconds, StreamingContext}

```

```
/**
 * 消费分流
 * 1. 接收 Kafka 数据流
 *
 * 2. 转换数据结构:
 *     通用的数据结构: Map 或者 JsonObject
 *     专用的数据结构: bean
 *
 * 3. 分流 : 将数据拆分到不同的主题中
 *     启动主题: DWD_START_LOG
 *     页面访问主题: DWD_PAGE_LOG
 *     页面动作主题: DWD_PAGE_ACTION
 *     页面曝光主题: DWD_PAGE_DISPLAY
 *     错误主题: DWD_ERROR_INFO
 */
object BaseLogApp {
  def main(args: Array[String]): Unit = {
    //创建配置对象
    val sparkConf: SparkConf = new
    SparkConf().setAppName("base_log_app").setMaster("local[4]")
    val ssc = new StreamingContext(sparkConf, Seconds(5))

    //原始主题
    val ods_base_topic : String = "ODS_BASE_LOG"
    //启动主题
    val dwd_start_log : String = "DWD_START_LOG"
    //页面访问主题
    val dwd_page_log : String = "DWD_PAGE_LOG"
    //页面动作主题
    val dwd_page_action : String = "DWD_PAGE_ACTION"
    //页面曝光主题
    val dwd_page_display : String = "DWD_PAGE_DISPLAY"
    //错误主题
    val dwd_error_info : String = "DWD_ERROR_INFO"

    //消费组
    val group_id : String = "ods_base_log_group"

    //补充:
    // 从 redis 中读取偏移量
    val offsets: Map[TopicPartition, Long] =
    MyOffsetUtils.getOffset(ods_base_topic, group_id)

    // 判断是否能读取到
    var kafkaDStream : DStream[ConsumerRecord[String, String]] =
    null
    if(offsets != null && offsets.nonEmpty){
      //redis 中有记录 offset
      //1. 接收 Kafka 数据流
      kafkaDStream =
```

```
MyKafkaUtils.getKafkaDStream(ods_base_topic,ssc,offsets,group_id)
}else{
    //redis 中没有记录 offset
    //1. 接收 Kafka 数据流
    kafkaDStream =
        MyKafkaUtils.getKafkaDStream(ods_base_topic,ssc,group_id)
}

//在数据转换前,提取本次流中 offset 的结束点,
var offsetRanges: Array[OffsetRange] = null // driver
kafkaDStream = kafkaDStream.transform( // 每批次执行一次
    rdd => {
        println(rdd.getClass.getName)
        offsetRanges
    }
)

rdd.asInstanceOf[HasOffsetRanges].offsetRanges
rdd
}
)

//2. 转换数据结构
val jsonDStream: DStream[JSONObject] = kafkaDStream.map(
    record => {

        val value: String = record.value()
        println(value)
        val jsonObject: JSONObject = JSON.parseObject(value)

        jsonObject
    }
)

//3.切分数据分流
jsonDStream.foreachRDD(
    rdd => {
        rdd.foreach(
            jsonObj => {
                //TODO 错误日志
                //分流错误日志 判断是否是错误日志,如果是错误日志,直接发送
                val errorObj: JSONObject = jsonObj.getJSONObject("err")
                if(errorObj !=null){
                    MyKafkaUtils.send(dwd_error_info,jsonObj.toJSONString)
                }else{
                    //提取公共信息
                    val commonObj: JSONObject =
                        jsonObj.getJSONObject("common")
                    val mid: String = commonObj.getString("mid")
                    val uid: String = commonObj.getString("uid")
                    val ar: String = commonObj.getString("ar")
                    val ch: String = commonObj.getString("ch")
                    val os: String = commonObj.getString("os")
                    val md: String = commonObj.getString("md")
                    val vc: String = commonObj.getString("vc")
                    val isNew: String = commonObj.getString("is_new")
                }
            }
        )
    }
)
```

```
        val ts: Long = jsonObj.getLong("ts")

        //分流页面日志
        val pageObj: JSONObject = jsonObj.getJSONObject("page")
        if(pageObj != null ){
            //提取字段
            val pageId: String = pageObj.getString("page_id")
            val pageItem: String = pageObj.getString("item")
            val pageItemType: String = pageObj.getString("item_type")
            val lastPageId: String = pageObj.getString("last_page_id")
            val duringTime: Long = pageObj.getLong("during_time")

            //封装 bean
            val pageLog =
                PageLog(mid, uid, ar, ch, isNew, md, os, vc, pageId,
                    lastPageId, pageItem, pageItemType, duringTime, ts)
            //发送 kafka

            MyKafkaUtils.send(dwd_page_log, JSON.toJSONString(pageLog, new
                SerializeConfig(true)))

            //分流动作日志
            val actionArrayObj: JSONArray = jsonObj.getJSONArray("actions")
            if(actionArrayObj != null && actionArrayObj.size() > 0 ){
                for(i <- 0 until actionArrayObj.size() ){
                    val actionObj: JSONObject = actionArrayObj.getJSONObject(i)
                    val actionId: String = actionObj.getString("action_id")
                    val actionItem: String = actionObj.getString("item")
                    val actionItemType: String = actionObj.getString("item_type")
                    //TODO actions
                    val actionTs: Long = actionObj.getLong("ts")

                    //封装 Bean
                    val pageActionLog =
                        PageActionLog(mid, uid, ar, ch, isNew, md, os, vc,
                            pageId, lastPageId, pageItem, pageItemType,
                            duringTime, actionId, actionItem, actionItemType, actionTs, ts)

                    //发送 Kafka

                    MyKafkaUtils.send(dwd_page_action, JSON.toJSONString(pageActionLog,
                        new SerializeConfig(true)))

                }
            }

            //分流曝光日志
```

```
        val displayArrayObj: JSONArray =
jsonObj.getJSONArray("displays")
        if(displayArrayObj != null && displayArrayObj.size() >
0 ){
            for(i <- 0 until displayArrayObj.size()){
                val displayObj: JSONObject =
displayArrayObj.getJSONObject(i)
                val displayType: String =
displayObj.getString("display_type")
                val displayItem: String =
displayObj.getString("item")
                val displayItemType: String =
displayObj.getString("item_type")
                val displayOrder: String =
displayObj.getString("order")
                val displayPosId: String =
displayObj.getString("pos_id")

                //封装 Bean
                val displayLog = PageDisplayLog(mid, uid, ar, ch,
isNew, md, os, vc, pageId, lastPageId, pageItem, pageItemType,
duringTime,displayType,displayItem,displayItemType,displayOrder,d
isplayPosId,ts)

                //发送 Kafka

MyKafkaUtils.send(dwd_page_display,JSON.toJSONString(displayLog,new
SerializeConfig(true)))
            }
        }

        //分流启动日志
        val startObj: JSONObject =
jsonObj.getJSONObject("start")
        if(startObj != null ){
            val entry: String = startObj.getString("entry")
            val loadingTimeMs: Long =
startObj.getLong("loading_time_ms")
            val openAdId: String =
startObj.getString("open_ad_id")
            val openAdMs: Long = startObj.getLong("open_ad_ms")
            val openAdSkipMs: Long =
startObj.getLong("open_ad_skip_ms")

            //封装 Bean
            val startLog =
StartLog(mid,uid,ar,ch,isNew,md,os,vc,entry,openAdId,loadingTimeM
s,openAdMs,openAdSkipMs,ts)

            //发送 Kafka

MyKafkaUtils.send(dwd_start_log,JSON.toJSONString(startLog,new
SerializeConfig(true)))
        }
```

```

    }
    // foreach 里边 executor 中执行，一条数据执行一次
  }

  )
  //补充：
  //foreachRDD 里边 driver 中执行，一个批次执行一次
  //提交偏移量

MyOffsetUtils.saveOffset(ods_base_topic,group_id,offsetRanges)
}
)

// foreachRDD 外面 driver 中执行，一次启动执行一次

ssc.start()
ssc.awaitTermination()
}
}

```

2.6.7 幂等性操作

目前处理完的数据写到了 kafka,如果程序出现宕机重试，kafka 是没有办法通过唯一性标识实现幂等性识别，但是也没有关系，因为 kafka 中的数据只是用于中间存储,并不会进行统计，所以只要保证不丢失即可，重复数据的幂等性处理可以交给下游处理，只要保证最终统计结果是不会有重复即可。

2.7 优化 - kafka 消息发送问题

2.7.1 缓冲区问题

Kafka 消息的发送分为同步发送和异步发送。Kafka 默认使用异步发送的方式。Kafka 的生产者将消息进行发送时，会先将消息发送到缓冲区中，待缓冲区写满或者到达指定的时间，才会真正的将缓冲区的数据写到 Broker。

假设消息发送到缓冲区中还未写到 Broker，我们认为数据已经成功写给了 Kafka，接下来会手动的提交 offset，如果 offset 提交成功，但此刻 Kafka 集群突然出现故障。缓冲区的数据会丢失，最终导致的问题就是数据没有成功写到 Kafka，而 offset 已经提交，此部分的数据就会被漏掉。

2.7.2 问题解决 – 策略一

将消息的发送修改为同步发送，保证每条数据都能发送到 Broker。但带来的问题就是消息是一条一条写给 Broker,会牺牲性能，一般不推荐。

2.7.3 问题解决 – 策略二

1) 策略: 在手动提交 offset 之前, 强制将缓冲区的数据 flush 到 broker

Kafka 的生产者对象提供了 flush 方法, 可以强制将缓冲区的数据刷到 Broker。

2) 修补主程序代码

```
package com.atguigu.gmall.realtime.app

import com.alibaba.fastjson.serializer.SerializeConfig
import com.alibaba.fastjson.{JSON, JSONArray, JSONObject}
import com.atguigu.gmall.realtime.bean.{PageActionLog,
PageDisplayLog, PageLog, StartLog}
import com.atguigu.gmall.realtime.util.{MyKafkaUtils,
MyOffsetUtils}
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.TopicPartition
import org.apache.spark.SparkConf
import org.apache.spark.streaming.dstream.{DStream, InputDStream}
import org.apache.spark.streaming.kafka010.{HasOffsetRanges,
OffsetRange}
import org.apache.spark.streaming.{Seconds, StreamingContext}

/**
 * 消费分流
 * 1. 接收 Kafka 数据流
 *
 * 2. 转换数据结构:
 *     通用的数据结构: Map 或者 JsonObject
 *     专用的数据结构: bean
 *
 * 3. 分流 : 将数据拆分到不同的主题中
 *     启动主题: DWD_START_LOG
 *     页面访问主题: DWD_PAGE_LOG
 *     页面动作主题: DWD_PAGE_ACTION
 *     页面曝光主题: DWD_PAGE_DISPLAY
 *     错误主题: DWD_ERROR_INFO
 */
object BaseLogApp {
  def main(args: Array[String]): Unit = {
    //创建配置对象
    val sparkConf: SparkConf = new
SparkConf().setAppName("base_log_app").setMaster("local[4]")
    val ssc = new StreamingContext(sparkConf, Seconds(5))

    //原始主题
    val ods_base_topic : String = "ODS_BASE_LOG"
    //启动主题
    val dwd_start_log : String = "DWD_START_LOG"
    //页面访问主题
    val dwd_page_log : String = "DWD_PAGE_LOG"
```



```
//页面动作主题
val dwd_page_action : String = "DWD_PAGE_ACTION"
//页面曝光主题
val dwd_page_display : String = "DWD_PAGE_DISPLAY"
//错误主题
val dwd_error_info : String = "DWD_ERROR_INFO"

//消费者组
val group_id : String = "ods_base_log_group"

//补充:
// 从 redis 中读取便宜量
val offsets: Map[TopicPartition, Long] =
MyOffsetUtils.getOffset(ods_base_topic, group_id)

// 判断是否能读取到
var kafkaDStream : DStream[ConsumerRecord[String, String]] =
null
if(offsets != null && offsets.nonEmpty){
  //redis 中有记录 offset
  //1. 接收 Kafka 数据流
  kafkaDStream =
MyKafkaUtils.getKafkaDStream(ods_base_topic, ssc, offsets, group_id)
}else{
  //redis 中没有记录 offset
  //1. 接收 Kafka 数据流
  kafkaDStream =
    MyKafkaUtils.getKafkaDStream(ods_base_topic, ssc, group_id)
}

//在数据转换前, 提取本次流中 offset 的结束点,
var offsetRanges: Array[OffsetRange] = null // driver
kafkaDStream = kafkaDStream.transform( // 每批次执行一次
  rdd => {
    println(rdd.getClass.getName)
    offsetRanges
  }
)

//2. 转换数据结构
val jsonDStream: DStream[JSONObject] = kafkaDStream.map(
  record => {
    val value: String = record.value()
    val jsonObject: JSONObject = JSON.parseObject(value)
    jsonObject
  }
)

//3. 切分数据分流
jsonDStream.foreachRDD(
```

```
rdd => {
  rdd.foreachPartition(
    iter => {
      for (jsonObj <- iter) {
        //TODO 错误日志
        //分流错误日志 判断是否是错误日志, 如果是错误日志, 直接发送
        val errorObj: JSONObject = jsonObj.getJSONObject("err")
        if(errorObj !=null){

MyKafkaUtils.send(dwd_error_info,jsonObj.toJSONString)
        }else{
          //提取公共信息
          val commonObj: JSONObject =
jsonObj.getJSONObject("common")
          val mid: String = commonObj.getString("mid")
          val uid: String = commonObj.getString("uid")
          val ar: String = commonObj.getString("ar")
          val ch: String = commonObj.getString("ch")
          val os: String = commonObj.getString("os")
          val md: String = commonObj.getString("md")
          val vc: String = commonObj.getString("vc")
          val isNew: String = commonObj.getString("is_new")
          val ts: Long = commonObj.getLong("ts")

          //分流页面日志
          val pageObj: JSONObject =
jsonObj.getJSONObject("page")
          if(pageObj != null ){
            //提取字段
            val pageId: String = pageObj.getString("page_id")
            val pageItem: String = pageObj.getString("item")
            val pageItemType: String =
pageObj.getString("item_type")
            val lastPageId: String =
pageObj.getString("last_page_id")
            val duringTime: Long = pageObj.getLong("during_time")

            //封装 bean
            val pageLog =
              PageLog(mid, uid, ar, ch, isNew, md, os, vc, pageId,
lastPageId, pageItem, pageItemType, duringTime, ts)
            //发送 kafka

MyKafkaUtils.send(dwd_page_log,JSON.toJSONString(pageLog,new
SerializeConfig(true)))

            //分流动作日志
            val actionArrayObj: JSONArray =
jsonObj.getJSONArray("actions")
            if(actionArrayObj != null && actionArrayObj.size() >
0 ){
              for(i <- 0 until actionArrayObj.size() ){
                val actionObj: JSONObject =
actionArrayObj.getJSONObject(i)
                val actionId: String =
```

```
actionObj.getString("action_id")
        val actionItem: String =
actionObj.getString("item")
        val actionItemType: String =
actionObj.getString("item_type")
        //TODO actions
        val actionTs: Long = actionObj.getLong("ts")

        //封装 Bean
        val pageActionLog =
            PageActionLog(mid, uid, ar, ch, isNew, md, os,
vc, pageId, lastPageId, pageItem, pageItemType,
duringTime, actionId, actionItem, actionItemType, actionTs, ts)

        //发送 Kafka

MyKafkaUtils.send(dwd_page_action, JSON.toJSONString(pageActionLog
,new SerializeConfig(true)))

    }
}

//分流曝光日志
val displayArrayObj: JSONArray =
jsonObj.getJSONArray("displays")
if(displayArrayObj != null && displayArrayObj.size() >
0 ){
    for(i <- 0 until displayArrayObj.size()){
        val displayObj: JSONObject =
displayArrayObj.getJSONObject(i)
        val displayType: String =
displayObj.getString("display_type")
        val displayItem: String =
displayObj.getString("item")
        val displayItemType: String =
displayObj.getString("item_type")
        val displayOrder: String =
displayObj.getString("order")
        val displayPosId: String =
displayObj.getString("pos_id")

        //封装 Bean
        val displayLog = PageDisplayLog(mid, uid, ar, ch,
isNew, md, os, vc, pageId, lastPageId, pageItem, pageItemType,
duringTime, displayType, displayItem, displayItemType, displayOrder, d
isplayPosId, ts)

        //发送 Kafka

MyKafkaUtils.send(dwd_page_display, JSON.toJSONString(displayLog, n
ew SerializeConfig(true)))
    }
}

}
```

```
//分流启动日志
val startObj: JSONObject =
jsonObj.getJSONObject("start")
if(startObj != null ){
    val entry: String = startObj.getString("entry")
    val loadingTimeMs: Long =
startObj.getLong("loading_time_ms")
    val openAdId: String =
startObj.getString("open_ad_id")
    val openAdMs: Long = startObj.getLong("open_ad_ms")
    val openAdSkipMs: Long =
startObj.getLong("open_ad_skip_ms")

    //封装 Bean
    val startLog =
StartLog(mid,uid,ar,ch,isNew,md,os,vc,entry,openAdId,loadingTimeM
s,openAdMs,openAdSkipMs,ts)
    //发送 Kafka

MyKafkaUtils.send(dwd_start_log,JSON.toJSONString(startLog,new
SerializeConfig(true)))
}

}

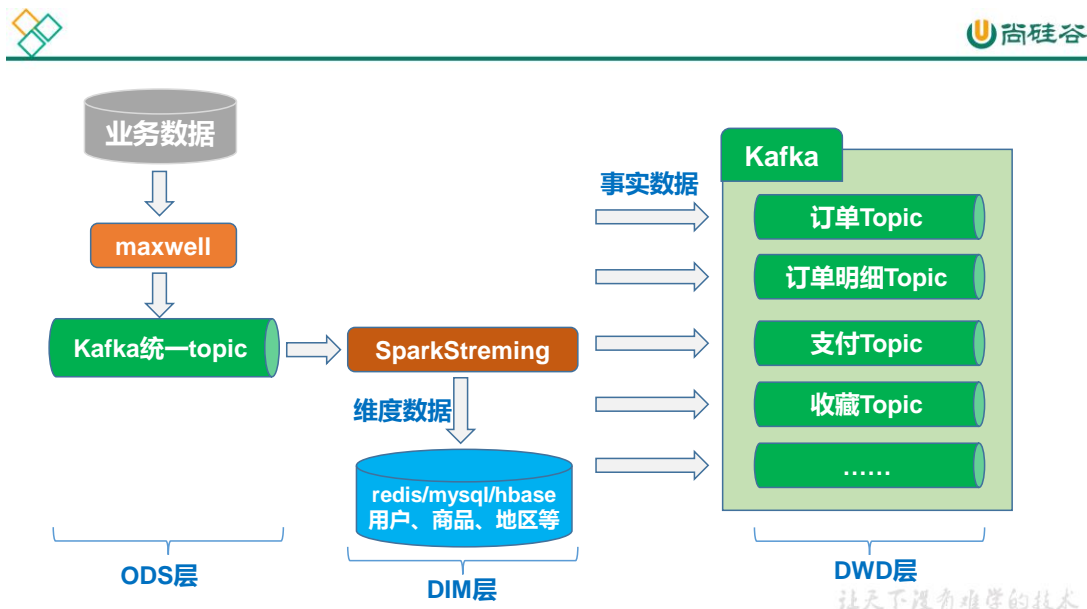
// 一个批次一个分区提交一次
MyKafkaUtils.flush()
}

//foreachRDD 里边 driver 中执行，一个批次执行一次
//提交偏移量

MyOffsetUtils.saveOffset(ods_base_topic,group_id,offsetRanges)
}
)
//foreachRDD 外面 driver 中执行，启动应用程序执行一次。
ssc.start()
ssc.awaitTermination()
}
}
```

第3章 业务数据采集和分流

3.1 整体架构



业务数据库的采集主要是基于对数据库的变化的实时监控。目前市面上的开源产品主要是 **Canal** 和 **Maxwell**。要利用这些工具实时采集数据到 **kafka**，以备后续处理。

Maxwell 采集的 **日志数据**，默认是放在一个统一的 **kafka** 的 **Topic** 中，为了后续方便处理要进行以表为单位拆分到不同 **kafka** 的 **Topic** 中。

针对 **维度数据**，要单独保存。通常考虑用 **redis**、**hbase**、**mysql**、**kudu** 等通过唯一键查询性能较快的数据库中。

3.2 Maxwell 简介

3.2.1 什么是 Maxwell

Maxwell 是由美国 **Zendesk** 公司开源，用 **Java** 编写的 **MySQL** 变更数据抓取软件。它会实时监控 **MySQL** 数据库的数据变更操作（包括 **insert**、**update**、**delete**），并将变更数据以 **JSON** 格式发送给 **Kafka**、**Kinesis** 等流数据处理平台。

官网地址：<http://maxwells-daemon.io/>

3.2.2 MySQL 主从复制

1) 主从复制的应用场景如下

- (1) 做数据库的热备：主数据库服务器故障后，可切换到从数据库继续工作。

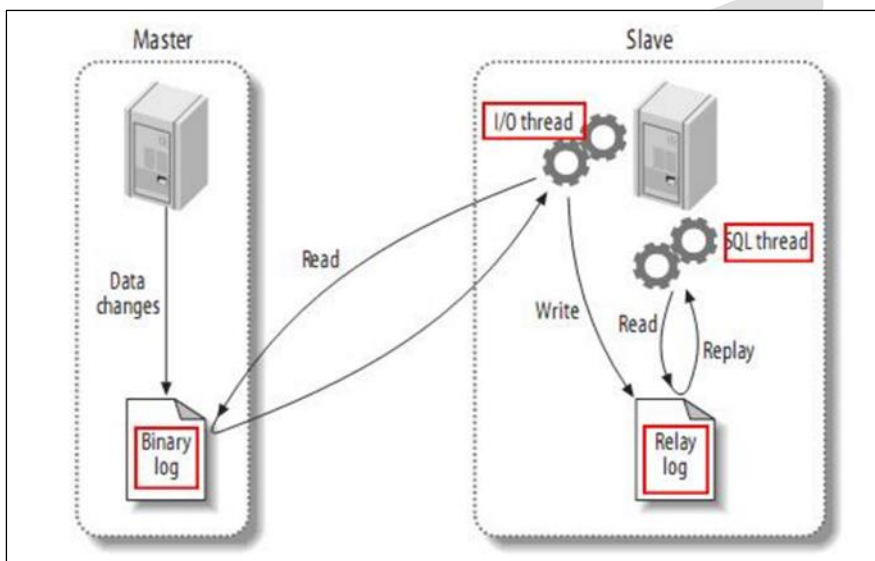
(2) 分离：主数据库只负责业务数据的写入操作，而多个从数据库只负责业务数据的查询工作，在读多写少场景下，可以提高数据库工作效率

2) MySQL 主从复制工作原理

(1) Master 主库将数据变更记录，写到二进制日志(binary log)中

(2) Slave 从库向 mysql master 发送 dump 协议，将 master 主库的 binary log events 拷贝到它的中继日志(relay log)

(3) Slave 从库读取并回放中继日志中的事件，将改变的数据同步到自己的数据库。



3) binlog

(1) 什么是 binlog

MySQL 的二进制日志可以说 MySQL 最重要的日志了，它记录了所有的 DDL 和 DML(除了数据查询语句)语句，以事件形式记录，还包含语句所执行的消耗的时间，MySQL 的二进制日志是事务安全型的。

一般来说开启二进制日志大概会有 1%的性能损耗。

二进制日志包括两类文件：二进制日志索引文件（文件名后缀为.index）用于记录所有的二进制文件，二进制日志文件（文件名后缀为.00000*）记录数据库所有的 DDL 和 DML(除了数据查询语句)语句事件。

(2) binlog 的分类设置

mysql binlog 的格式有三种，分别是 STATEMENT,MIXED,ROW。

```
binlog_format= statement|mixed|row
```

➤ statement

语句级，binlog 会记录每次一执行写操作的语句。相对 row 模式节省空间，但是可能产生不一致性，比如 `update tt set create_date=now()` 如果用 binlog 日志进行恢复，由于执行时间不同可能产生的数据就不同。

优点： 节省空间

缺点： 有可能造成数据不一致。

➤ row

行级， binlog 会记录每次操作后每行记录的变化。

优点：保持数据的绝对一致性。因为不管 sql 是什么，引用了什么函数，他只记录执行后的效果。

缺点： 占用较大空间。

➤ mixed

statement 的升级版，一定程度上解决了因为一些情况而造成的 statement 模式不一致问题。默认还是 statement，在某些情况下譬如：当函数中包含 `UUID()` 时；包含 `AUTO_INCREMENT` 字段的表被更新时；执行 `INSERT DELAYED` 语句时；用 UDF 时；会按照 ROW 的方式进行处理

优点：节省空间，同时兼顾了一定的一致性。

缺点：还有些极个别情况依旧会造成不一致，另外 statement 和 mixed 对于需要对 binlog 的监控的情况都不方便。

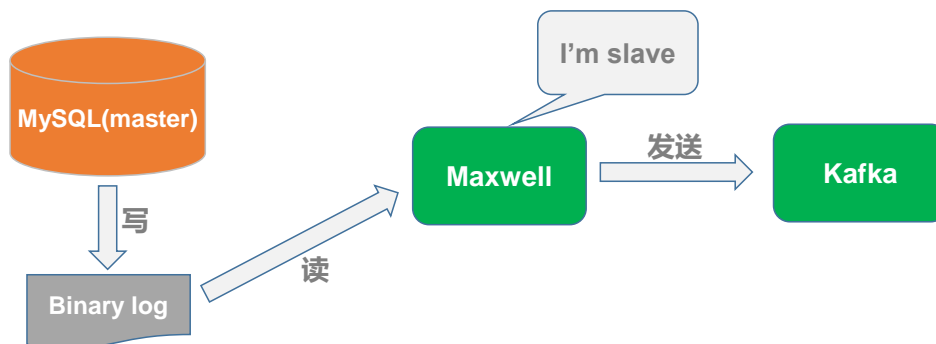
综合上面对比，Maxwell 想做监控分析，选择 row 格式比较合适。

3.2.3 Maxwell 工作原理

很简单，就是将自己伪装成 slave，并遵循 MySQL 主从复制的协议，从 master 同步数据。



Maxwell工作原理



让天下没有难学的技术

3.2.4 Maxwell 安装

1) 将安装包上传到 hadoop102 节点的/opt/software 目录

2) 解压到/opt/module/目录下

```
[atguigu@hadoop102 software]$ tar -zxvf maxwell-1.29.2.tar.gz -C /opt/module/
```

3) 修改名称

```
[atguigu@hadoop102 module]$ mv maxwell-1.29.2/ maxwell
```

3.3 采集业务数据

3.3.1 MySQL 部分

1) 安装 MySQL (略)

2) 创建业务数据库 (直接使用离线数仓的即可)

3) 导入数据表 (直接使用离线数仓的即可)

4) 修改/etc/my.cnf 文件, 开启 binlog

```
[atguigu@hadoop102 module]$ sudo vim /etc/my.cnf
```

```
server-id=1
log-bin=mysql-bin
binlog_format=row
binlog-do-db=gmall
```

注意: binlog-do-db 根据自己的情况进行修改, 指定具体要同步的数据库。

5) 重启 MySQL 使配置生效

```
[atguigu@hadoop102 ~]$ sudo systemctl restart mysqld
```

40

更多 Java -大数据 -前端 -python 人工智能资料下载, 可百度访问: 尚硅谷官网

到/var/lib/mysql 目录下查看是否生成 binlog 文件

```
-rw-r-----. 1 mysql mysql      154 12月 28 17:40 mysql-bin.000001
-rw-r-----. 1 mysql mysql      19 12月 28 17:40 mysql-bin.index
```

3.3.2 模拟数据

1) 上传 jar 和 properties 文件上传到/opt/module/db_log 目录下

```
[atguigu@hadoop102 db_log]$
[atguigu@hadoop102 db_log]$ ll
总用量 14780
-rw-r--r--. 1 atguigu atguigu      1601  9 月      25  16:24
application.properties
-rw-r--r--. 1 atguigu atguigu 15127607  2 月      1  2021
gmall2020-mock-db-2021-01-22.jar
```

2) 修改 application.properties 中数据库连接信息

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://hadoop102:3306/gmall?character
Encoding=utf-8&useSSL=false&serverTimezone=GMT%2B8
spring.datasource.username=root
spring.datasource.password=123456
```

3) 模拟生成数据

```
[atguigu@hadoop102 db_log]$ java -jar
gmall2020-mock-db-2021-01-22.jar
```

4) 再次到/var/lib/mysql 目录下, 查看 binlog 的改动

```
-rw-r-----. 1 mysql mysql 345632 12月 28 17:53 mysql-bin.000001
-rw-r-----. 1 mysql mysql      19 12月 28 17:40 mysql-bin.index
```

3.3.3 赋权限

1) 给 maxwell 创建用户并赋权限

```
mysql> CREATE USER 'maxwell'@'%' IDENTIFIED BY 'maxwell';
mysql> GRANT SELECT, REPLICATION CLIENT, REPLICATION SLAVE ON *.* TO
'maxwell'@'%';
```

3.3.4 Maxwell 部分

1) 创建数据库, 用于存储 Maxwell 运行过程中的一些数据,包括 binlog 同步的断点位置等

```
mysql> create database maxwell;
mysql> GRANT ALL ON maxwell.* TO 'maxwell'@'%';
```

2) 修改 Maxwell 配置文件名字

```
[atguigu@hadoop102 maxwell]$ mv config.properties.example
config.properties
```

3) 修改 Maxwell 配置文件内容

```
[atguigu@hadoop102 maxwell]$ vim config.properties

#Maxwell 数据发送目的地, 可选配置有
stdout|file|kafka|kinesis|pubsub|sqs|rabbitmq|redis
```

```
producer=kafka
#目标 Kafka 集群地址
kafka.bootstrap.servers=hadoop102:9092,hadoop103:9092
#目标 Kafka topic, 可静态配置, 例如:maxwell, 也可动态配置, 例如:
#{database}_{table}
kafka_topic=ODS_BASE_DB_M

#MySQL 相关配置
host=hadoop102
user=maxwell
password=maxwell
jdbc_options=useSSL=false&serverTimezone=Asia/Shanghai
```

4) 启动 Maxwell

```
[atguigu@hadoop102 maxwell]$ bin/maxwell --config config.properties
--daemon
```

5) 查看进程

```
[atguigu@hadoop102 bin]$ jps
2428 Maxwell
```

6) 启动 Kafka 消费客户端测试, 查看消费情况

```
[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh
--bootstrap-server hadoop102:9092 --topic ODS_BASE_DB_M
```

7) 模拟生成数据, 查看是否能消费到数据

```
[atguigu@hadoop102 db_log]$ java -jar
gmall2020-mock-db-2021-01-22.jar
```

3.4 业务数据消费分流

3.4.1 Maxwell 数据格式

```
{"database":"gmall","table":"order_info","type":"update","ts":164
0862565,"xid":3159,"commit":true,"data":{"id":5319,"consignee":"王
洁
1111","consignee_tel":"13559267642","total_amount":10130.00,"orde
r_status":"1004","user_id":12,"payment_way":null,"delivery_addres
s":"第 3 大街第 29 号楼 8 单元 674 门","order_comment":"描述
314649","out_trade_no":"481626686333178","trade_body":"Redmi 10X 4G
Helio G85 游戏芯 4800 万超清四摄 5020mAh 大电量 小孔全面屏 128GB 大存储
8GB+128GB 明月灰 游戏智能手机 小米 红米等 4 件商品
","create_time":"2021-09-24 19:05:45","operate_time":"2021-09-24
19:05:45","expire_time":"2021-09-24
19:20:45","process_status":null,"tracking_no":null,"parent_order_
id":null,"img_url":"http://img.gmall.com/844142.jpg","province_id
":13,"activity_reduce_amount":0.00,"coupon_reduce_amount":0.00,"o
riginal_total_amount":10120.00,"feight_fee":10.00,"feight_fee_red
uce":null,"refundable_time":null},"old":{"consignee":"王洁"}}
```

3.4.2 消费分流

Maxwell 会追踪整个数据库的变更, 把所有的数据变化都发到一个 topic 中了, 但是为

了后续处理方便，应该将事实表的数据分流到不同的 topic 中，将维度表的数据写入到 redis 中。

1) 分流业务代码

```
package com.atguigu.gmall.realtime.app

import java.util

import com.alibaba.fastjson.{JSON, JSONArray, JSONObject}
import com.atguigu.gmall.realtime.util.{MyKafkaUtils, MyOffsetUtils, MyRedisUtils}
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.TopicPartition
import org.apache.spark.SparkConf
import org.apache.spark.broadcast.Broadcast
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.streaming.kafka010.{HasOffsetRanges, OffsetRange}
import org.apache.spark.streaming.{Seconds, StreamingContext}
import redis.clients.jedis.Jedis

/**
 * 业务数据分流
 * 1. 读取偏移量
 * 2. 接收 kafka 数据
 * 3. 提取偏移量结束点
 * 4. 转换结构
 * 5. 分流处理
 *   5.1 事实数据分流-> kafka
 *   5.2 维度数据分流-> redis
 * 6. 提交偏移量
 */
object BaseDBApp_maxwell {
  def main(args: Array[String]): Unit = {

    val sparkConf: SparkConf = new SparkConf().setAppName("base_db_app").setMaster("local[4]")
    val ssc = new StreamingContext(sparkConf, Seconds(5))

    val topic = "ODS_BASE_DB_M"
    val groupId = "base_db_group"

    //1.读取偏移量
    val offsets: Map[TopicPartition, Long] = MyOffsetUtils.getOffset(topic, groupId)

    //2. 接收 kafka 数据
    var kafkaDStream : DStream[ConsumerRecord[String, String]] = null
    if(offsets != null && offsets.nonEmpty){
      kafkaDStream =
```

```
MyKafkaUtils.getKafkaDStream(topic,ssc,offsets,groupId)
    }else{
        kafkaDStream = MyKafkaUtils.getKafkaDStream(topic,ssc,groupId)
    }

    //3. 提取偏移量结束点
    var offsetRanges: Array[OffsetRange] = null
    kafkaDStream = kafkaDStream.transform(
        rdd => {
            offsetRanges
        }
    )

    //4. 转换结构
    val jsonObjDstream: DStream[JSONObject] = kafkaDStream.map(
        record => {
            val jsonString: String = record.value()
            val jsonObject: JSONObject = JSON.parseObject(jsonString)
            jsonObjDstream
        }
    )

    //5. 分流

    // 5.1 事实数据
    // 5.2 维度数据

    jsonObjDstream.foreachRDD(
        rdd => {
            val jedis: Jedis = MyRedisUtils.getJedisClient
            val dimTableKey : String = "DIM:TABLES"
            val factTableKey : String = "FACT:TABLES"
            //从 redis 中读取表清单
            val dimTables: util.Set[String] = jedis.smembers(dimTableKey)
            val factTables: util.Set[String] = jedis.smembers(factTableKey)
            println("检查维度表: " + dimTables)
            println("检查事实表: " + factTables)
            //做成广播变量
            val dimTablesBC: Broadcast[util.Set[String]] = ssc.sparkContext.broadcast(dimTables)
            val factTablesBC: Broadcast[util.Set[String]] = ssc.sparkContext.broadcast(factTables)

            jedis.close()

            //只要有一个批次执行一次,且在 executor 中执行的代码,就需要用
            foreachPartition
            rdd.foreachPartition(
                jsonObjIter => {
```

```
//获取 redis 连接
val jedis: Jedis = MyRedisUtils.getJedisClient
for (jsonObj <- jsonObjIter) {
    //提取表名
    val tableName: String = jsonObj.getString("table")
    //提取操作类型
    val optType: String = jsonObj.getString("type")
    val opt: String = optType match {
        case "insert" => "I"
        case "update" => "U"
        case "delete" => "D"
        case _ => null // DDL 操作, 例如: CREATE ALTER
        TRUNCATE ....
    }
    if(opt != null ){

        //提取修改后的数据
        val jsonArray: JSONArray = jsonObj.getJSONObject("data")
        //事实表数据
        if(factTablesBC.value.contains(tableName)){
            // 拆分到指定的主题
            // topic => DWD_[TABLE_NAME]_[I/U/D]
            val topicName = s"DWD_${tableName.toUpperCase()}_${opt}"

            val key: String = jsonArray.getString("id")
            //发送 kafka
            MyKafkaUtils.send(topicName, key,
            jsonArray.toJSONString)

        }

        //维度表处理
        if(dimTablesBC.value.contains(tableName)){
            //val jedis: Jedis = MyRedisUtils.getJedisClient
            // 存储类型的选择: String 、 set 、 hash ?
            // key : DIM:[table_name]:[主键]
            // value : 整条数据的 json 串
            val id: String = jsonArray.getString("id")
            val redisKey : String = s"DIM:${tableName.toUpperCase()}:$id"
            val redisValue : String = jsonArray.toJSONString
            jedis.set(redisKey, redisValue)

            //jedis.close()
        }
    }
    jedis.close()
    MyKafkaUtils.flush()
}
//6.提交偏移量
```

```

        MyOffsetUtils.saveOffset(topic, groupId, offsetRanges)
    }
)
ssc.start()
ssc.awaitTermination()
}
}

```

3.4.3 历史维度数据初始引导

1) Maxwell 提供了 bootstrap 功能来进行历史数据的全量同步, 命令如下:

```

[atguigu@hadoop102 maxwell]$ bin/maxwell-bootstrap --config
config.properties --database gmall --table user_info

```

2) Bootstrap 数据格式

```

{
  "database": "fooDB",
  "table": "barTable",
  "type": "bootstrap-start",
  "ts": 1450557744,
  "data": {}
}
{
  "database": "fooDB",
  "table": "barTable",
  "type": "bootstrap-insert",
  "ts": 1450557744,
  "data": {
    "txt": "hello"
  }
}
{
  "database": "fooDB",
  "table": "barTable",
  "type": "bootstrap-insert",
  "ts": 1450557744,
  "data": {
    "txt": "bootstrap!"
  }
}
{
  "database": "fooDB",
  "table": "barTable",
  "type": "bootstrap-complete",
  "ts": 1450557744,
  "data": {}
}

```

注意事项:

第一条 type 为 bootstrap-start 和最后一条 type 为 bootstrap-complete 的数据, 是 bootstrap 开始和结束的标志, 不包含数据, 中间的 type 为 bootstrap-insert 的数据才包含数据。

一次 bootstrap 输出的所有记录的 ts 都相同, 为 bootstrap 开始的时间。

1) 修改分流代码

```
package com.atguigu.gmall.realtime.app

import java.util

import com.alibaba.fastjson.{JSON, JSONArray, JSONObject}
import com.atguigu.gmall.realtime.util.{MyKafkaUtils, MyOffsetUtils, MyRedisUtils}
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.TopicPartition
import org.apache.spark.SparkConf
import org.apache.spark.broadcast.Broadcast
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.streaming.kafka010.{HasOffsetRanges, OffsetRange}
import org.apache.spark.streaming.{Seconds, StreamingContext}
import redis.clients.jedis.Jedis

/**
 * 业务数据分流
 * 1. 读取偏移量
 * 2. 接收 kafka 数据
 * 3. 提取偏移量结束点
 * 4. 转换结构
 * 5. 分流处理
 *   5.1 事实数据分流-> kafka
 *   5.2 维度数据分流-> redis
 * 6. 提交偏移量
 */
object BaseDBApp_maxwell {
  def main(args: Array[String]): Unit = {

    val sparkConf: SparkConf = new SparkConf().setAppName("base_db_app").setMaster("local[4]")
    val ssc = new StreamingContext(sparkConf, Seconds(5))

    val topic = "ODS_BASE_DB_M"
    val groupId = "base_db_group"

    //1. 读取偏移量
    val offsets: Map[TopicPartition, Long] = MyOffsetUtils.getOffset(topic, groupId)

    //2. 接收 kafka 数据
    var kafkaDStream: DStream[ConsumerRecord[String, String]] = null
    if (offsets != null && offsets.nonEmpty) {
      kafkaDStream = MyKafkaUtils.getKafkaDStream(topic, ssc, offsets, groupId)
    } else {
      kafkaDStream = MyKafkaUtils.getKafkaDStream(topic, ssc, groupId)
    }
  }
}
```

```
//3. 提取偏移量结束点
var offsetRanges: Array[OffsetRange] = null
kafkaDStream = kafkaDStream.transform(
  rdd => {
    offsetRanges
  }
)

//4. 转换结构
val jsonObjDstream: DStream[JSONObject] = kafkaDStream.map(
  record => {
    val jsonString: String = record.value()
    val jsonObject: JSONObject = JSON.parseObject(jsonString)
    jsonObject
  }
)

//5. 分流
// 5.1 事实数据
// 5.2 维度数据

jsonObjDstream.foreachRDD(
  rdd => {
    val jedis: Jedis = MyRedisUtils.getJedisClient
    val dimTableKey : String = "DIM:TABLES"
    val factTableKey : String = "FACT:TABLES"
    //从 redis 中读取表清单
    val dimTables: util.Set[String] = jedis.smembers(dimTableKey)
    val factTables: util.Set[String] = jedis.smembers(factTableKey)
    println("检查维度表: " + dimTables)
    println("检查事实表: " + factTables)
    //做成广播变量
    val dimTablesBC: Broadcast[util.Set[String]] = ssc.sparkContext.broadcast(dimTables)
    val factTablesBC: Broadcast[util.Set[String]] = ssc.sparkContext.broadcast(factTables)

    jedis.close()

    //只要有一个批次执行一次, 且在 executor 中执行的代码, 就需要用 foreachPartition
    rdd.foreachPartition(
      jsonObjIter => {
        //获取 redis 连接
        val jedis: Jedis = MyRedisUtils.getJedisClient
        for (jsonObj <- jsonObjIter) {
          //提取表名
          val tableName: String = jsonObj.getString("table")
        }
      }
    )
  }
)
```



```

//提取操作类型
val optType: String = jsonObj.getString("type")
val opt: String = optType match {
    case "bootstrap-insert" => "I"
    case "insert" => "I"
    case "update" => "U"
    case "delete" => "D"
    case _ => null // DDL 操作, 例如: CREATE ALTER
TRUNCATE ....
}
if(opt != null ){

    //提取修改后的数据
    val jsonArray: JSONArray = jsonObj.getJSONObject("data")
    //事实表数据
    if(factTablesBC.value.contains(tableName)){
        // 拆分到指定的主题
        // topic => DWD_[TABLE_NAME]_[I/U/D]
        val topicName = s"DWD_${tableName.toUpperCase()}_${opt}"

        val key: String = jsonArray.getString("id")
        //发送 kafka
        MyKafkaUtils.send(topicName, key,
        jsonArray.toJSONString)

    }

    //维度表处理
    if(dimTablesBC.value.contains(tableName)){
        //val jedis: Jedis = MyRedisUtils.getJedisClient
        // 存储类型的选择: String、set、hash?
        // key : DIM:[table_name]:[主键]
        // value : 整条数据的 json 串
        val id: String = jsonArray.getString("id")
        val redisKey: String = s"DIM:${tableName.toUpperCase()}:$id"
        val redisValue: String = jsonArray.toJSONString
        jedis.set(redisKey, redisValue)

        //jedis.close()
    }
    jedis.close()
    MyKafkaUtils.flush()
}
//6.提交偏移量
MyOffsetUtils.saveOffset(topic, groupId, offsetRanges)
}
)
ssc.start()

```

```
ssc.awaitTermination()
}
}
```

2) 测试引导历史数据

```
[atguigu@hadoop102 maxwell]$ bin/maxwell-bootstrap --config
config.properties --database gmall --table user_info
```

3.5 数据处理顺序性

在实时计算中，对业务数据的计算，要考虑到数据处理的顺序，即能否依照数据改变的顺序进行处理。

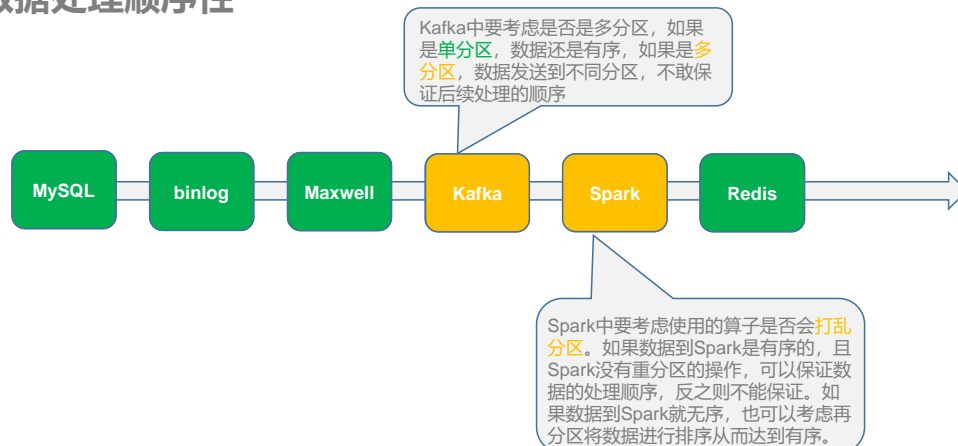
假设一个场景，如果将某个用户数据的姓名字段进行多次更改，由原先的 A 改为 B 再改为 C，在数据库层面最终的结果为 C，但是我们能否保证数据经过实时处理后，在 DIM 层存储的结果也为 C，可不可能存储的结果为 B。

我们依次审视一下，在实时处理的各个环节中，是否能保证数据的顺序？如果不能保证，是在哪个环节出的问题，最终导致存储的结果不正确。

3.5.1 分析



数据处理顺序性



让天下没有难学的技术

3.5.2 解决

通过分析，目前我们的计算过程中，只有可能在 **Kafka** 环节出现数据乱序，导致最终存储的结果不正确。如果要想保证数据处理的顺序性，我们可以将同一条数据的修改发往 **topic** 的同一个分区中。需要修改 **maxwell** 的配置文件，指定发送数据到 **kafka** 时要使用分区键。

1) 修改 config.properties 文件中的如下配置:

50

更多 **Java - 大数据 - 前端 - python 人工智能** 资料下载，可百度访问：[尚硅谷官网](http://www.shangguigu.com)

```
producer_partition_by=column  
producer_partition_columns=id  
producer_partition_by_fallback=table
```

第4章 总结