1. **【数仓模块-ID-MAPPING】**
   1. **id-mapping 概述**

在后续的数仓等模块开发中，我们都需要对每一条行为日志数据标记用户的唯一标识！

* 简单的方案  
  将这条数据中的 uid/imei码/imsi码/mac/androidid/uuid 这些字段（标识字段）按优先级取一个标识，作为这条数据的用户唯一标识！  
  这个方案有严重的漏洞
* 现实的无奈

在现实的日志数据中，由于，用户可能使用各种各样的设备，有着各种各样的前端入口，甚至同一个 用户拥有多个设备以及使用多种前端入口，就会导致，日志数据中对同一个人，不同时间段所收集到的日志数据中，可能取到的标识个数、种类各不相同；

比如：

用户可能使用各种各样的设备：

1. 手机、平板电脑
2. 安卓手机、ios 手机、winphone 手机

3）安卓系统有各种版本 （ 5.0 6.0 7.0 8.0 9.0 ）

4）ios 系统也有各种版本（3.x 4.x 5.x 6.x 7.x .... 12.x ）

产生问题：

用户设备的标识，没办法轻易定制一个规则来取某个作为唯一标识：

mac：手机网卡物理地址， 若干早期版本的 ios，winphone，android 可取到

imei(入网许可证序号)：安卓系统可取到，若干早期版本的 ios，winphone 可取到，运营商可取到

imsi(手机 SIM 卡序号)：安卓系统可取到，若干早期版本的 ios，winphone 可取到，运营商可取

到

androidid ：安卓系统 id

openuuid(app 自己生成的序号) ：卸载重装 app 就会变更**idfa**（广告跟踪码）

deviceid(app 日志采集埋点开发人员自己定义一种逻辑 id，可能取自 android,imei,openudid

等)：逻辑上的 id

|  |
| --- |
| * 从而导致： |

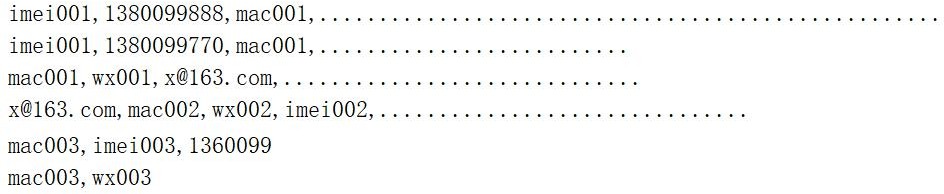
有一些数据中，用户有登录账号，而有些没有；

有一些数据中，有 imei 码，mac 地址；而有些则有 mac 地址和 android；

前一日的数据中，有 uid，android，而后一日数据中有 android，mac 地址

在这些情况中，如果按照之前的方案来生成数据的唯一标识，显然错漏百出！

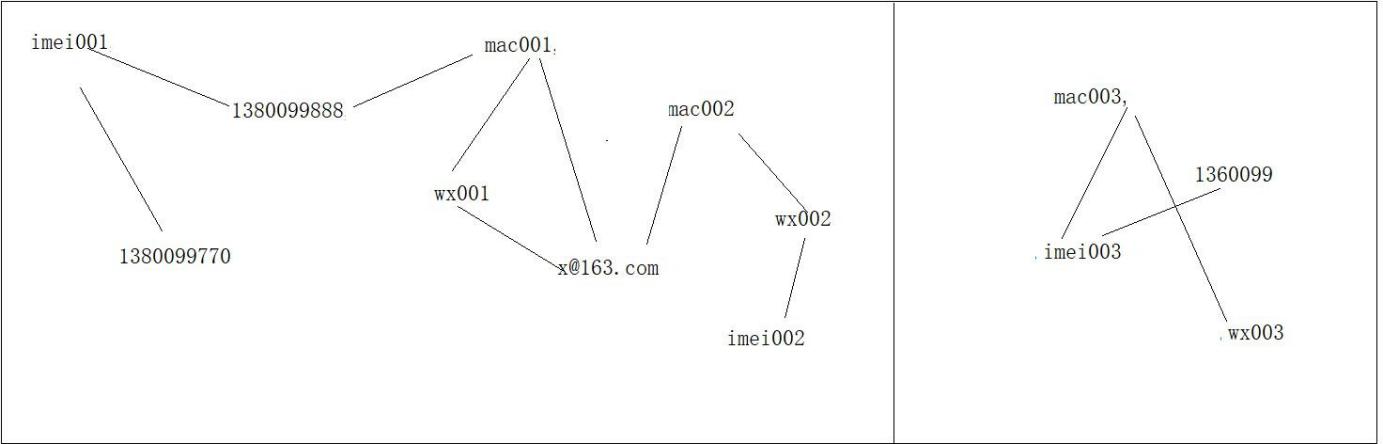
如下图：



要从这些纷繁复杂的各类 id 中，分辨出哪些 id 属于同一个受众（设备），用普通的“where x=y”这种简单条件逻辑很难实现。

* 1. **id-mapping 技术方案**
  2. **id-mapping 技术手段 2：借助图计算**

采用图计算手段，来找到各种 id 标识之间的关联关系，从而识别出哪些 id 标识属于同一个人；



图计算的核心思想：

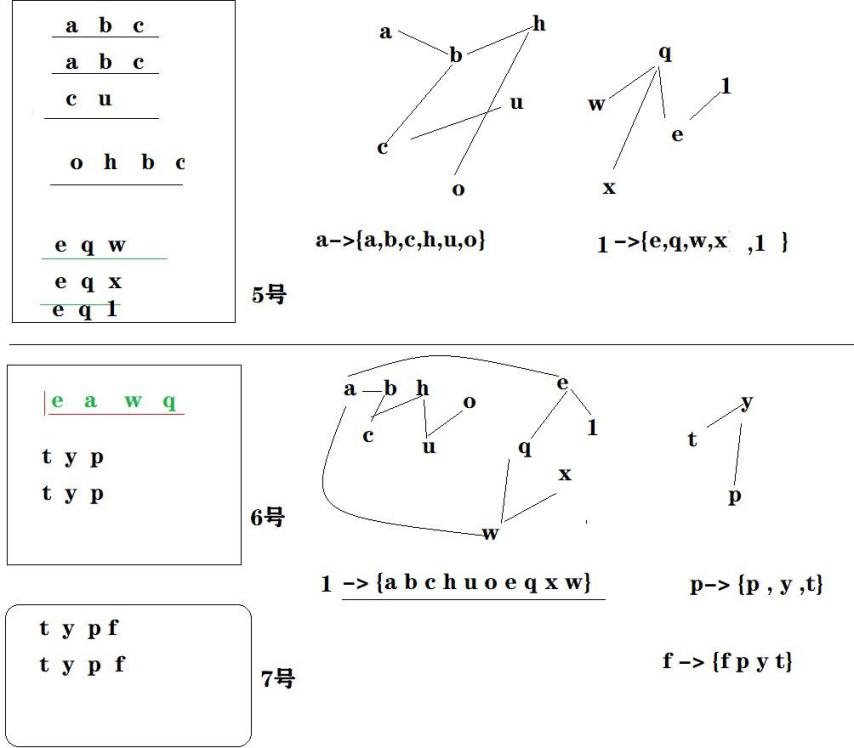
将数据表达成“点”，点和点之间可以通过某种业务含义建立“边” 然后，我们就可以从点、边上找出各种类型的数据关系：

比如连通性；

比如最短路径规划；

|  |  |  |
| --- | --- | --- |
| id\_mapping(id 打通)的最后目标，就是形成一个 **id 映射字典**： | | |
| id | ---- | guid |
| idx01 -> gid01 | | |
| idy01-> gid01 | | |
| idz01 -> gid01 | | |
| idx02 -> gid01 | | |
| idx02,idy2,idz02,idx13 -> gid02 | | |

|  |
| --- |
| 整体流程：   1. 将当日数据中的所有用户**标识字段**，及**标志字段**之间的关联，生成点集合 、边集合 2. 将上一日的 ids->guid 的映射关系，也生成点集合、边集合 3. 将上面两类点集合、边集合合并到一起生成一个图 4. 再对上述的图执行“**最大连通子图**”算法，得到一个连通子图结果 5. 在从结果图中取到哪些 id 属于同一组，并生成一个唯一标识 6. 将上面步骤生成的唯一标识去比对前日的 ids->guid 映射表（如果一个人已经存在 guid，则沿用原来的 guid） |



* 1. **id-mapping 开发实现**
     1. **开发目标**

将（pc 端埋点日志、app 端埋点日志、小程序端埋点日志）数据中的各种 id 标识字段提取出来，用图计算的连通子图算法，求出哪些 id 标识属于同一个人；

目标结果形式：

id-mapping 字典：

gid01 : [imei01 , imei03 ,13888997766 ,andorid03,mac03 ]

* + 1. **整体逻辑**

1. 抽取 3 类数据中的各种 id 字段，映射成“点”集合和“边”集合
2. （可选）过滤掉出现次数低于某个阈值（10 次）的“边”
3. 在结合 T-1 日的 idmapping 字典，构造一个图模型
4. 求最大连通子图，得到 T 日的临时结果
5. 整理临时结果（去 T-1 日的 idmapping 中匹配已经存在的 gid）以便保持 gid 的延续性！

*gid -> imei01,idfa01,mac01*

图计算算法应用基本步骤跟前面章节入门案例基本一致

(抽各 id，得到一个 id 数据集合，然后再映射出点集合、边集合，再构造图模型，再调连通子图算法， 取结果加工)；

## 重点细节问题

有一些现实情况需要考虑，比如：

1. 假如一个人偶尔用他朋友的手机登录过一次，则会出现他的 uid 跟他朋友的各设备 id 之间产生关联。

13800001100,zss,imei000,android000 13800001100,zss,imei000,android000 13800001100,zss,imei000,android000 13800001100,zss,imei000,android000 13800001100,zss,imei000,android000 13800001100,zss,imei000,android000

13800002211,tqq,imei002,android002 13800002211,tqq,imei002,android002 13800002211,tqq,imei002,android002

（去掉弱关联，保留强关联）

13800002211,zss,imei002,android002

13800002211,zss,imei002,android002

1. id-mapping 计算，需要逐日滚动进行，这就涉及到 T 日的数据与 T-1 日的 id-mapping 字典数据之间的合并问题（难在需要将之前生成好的“图 id”保持一贯）。
2. 一个用户的某些 id 标识，如果持续不再“出现”，应该持续衰减其权重值，权重值低于指定阈值后可以考虑过滤去除（本需求暂时放入下期迭代时实现）

## 滚动计算方案

day=1：

1. 抽取当天各类数据中的 id，并映射出“点集合”和“边集合”
2. 对“边集合”进行聚合计数，过滤掉(频次<阈值)的边（作用：过滤偶然性“噪点”干扰）
3. 用过滤后的边构造图

4）用图计算连通子图算法，对 id 标识聚合打通，得到“gid<=>各 id 标识” id-mapping 映射字典

day=2：

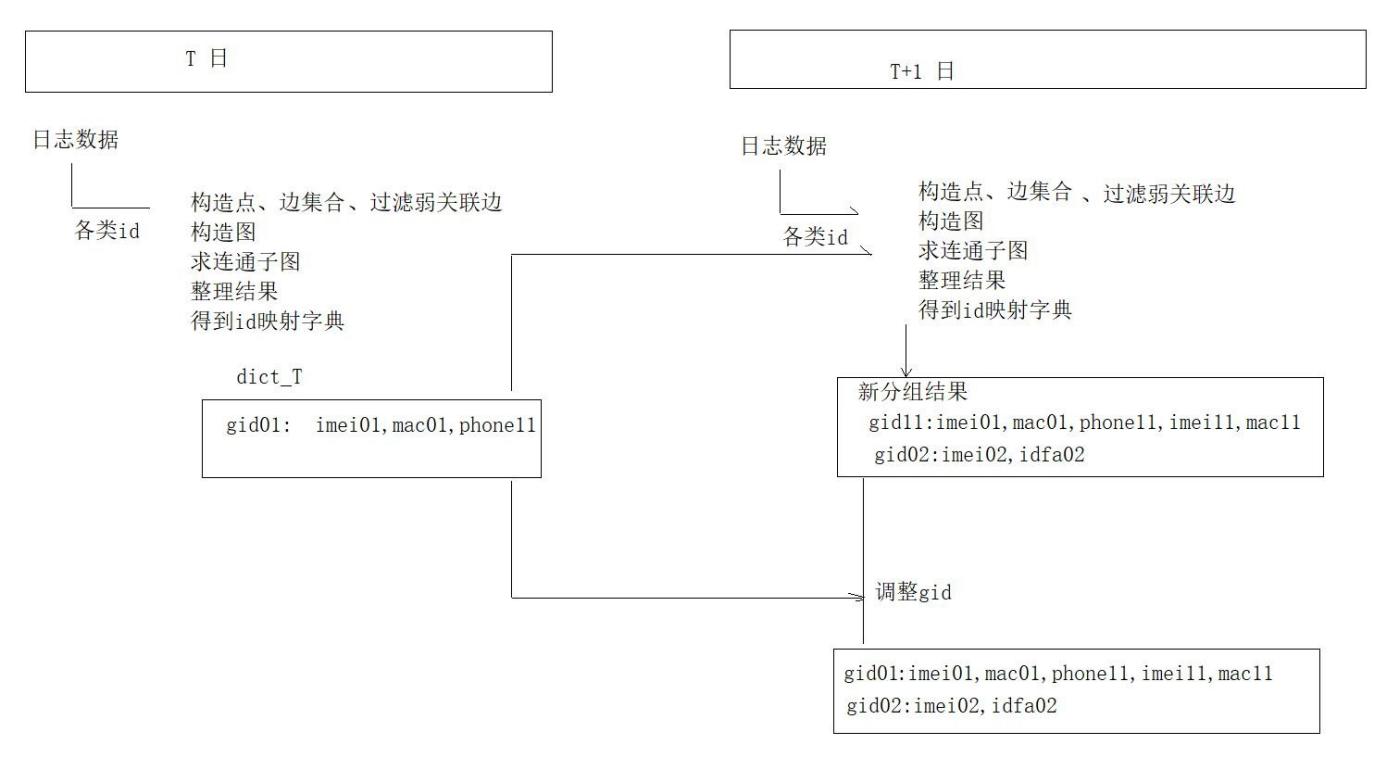
1. 抽取当天各类数据中的 id，并映射出“点集合 VertextC1”和“边集合”
2. 对当天“边集合”进行聚合计数，过滤掉(频次<阈值)的边，得到“边集合”EdgeC1 3）对 T-1 日的 id-mapping 结果，重新映射成“点 VertextC2、边集合 EdgeC2”
3. 将上述两个“点、边集合”合并，并构造图
4. 对合并图执行连通子图计算，得到新的 id-mapping 结果
5. 将新 id-mapping 跟旧的 id-mapping 进行合并操作(可能有新增组，可能有旧组新增 id)

（考虑情况：新 id 新组；新 id 旧组）

day>=3：重复 day\_2 的流程

## 实现流程图

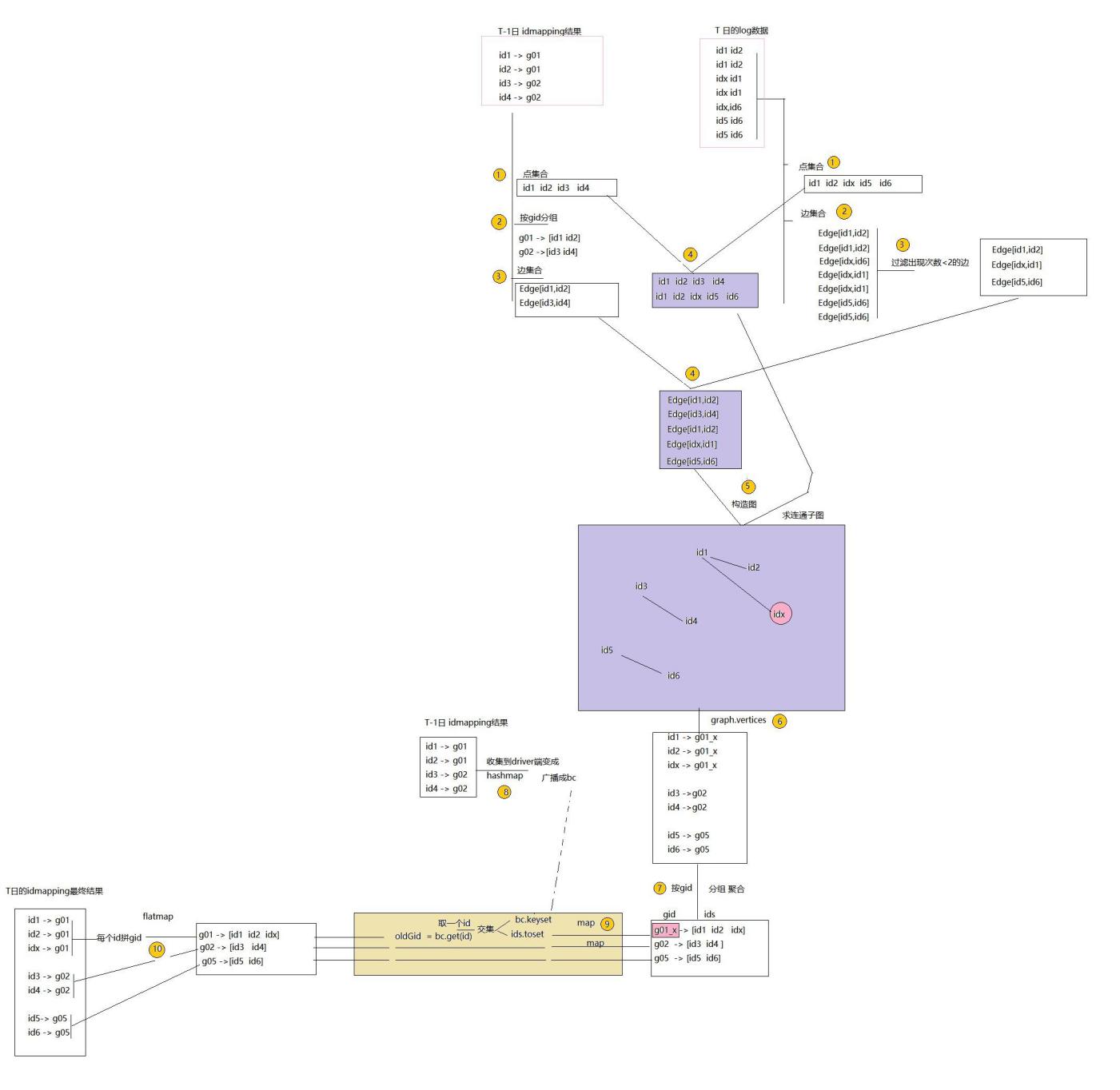
* 主体流程



相关说明：

|  |
| --- |
|  |
| * 细节流程 |





## 代码实现

多种来源的数据：

* web 端埋点日志
* app 端埋点日志
* 微信小程序埋点日志



步骤：

1. 加载当日的这几类数据
2. 提取出每类数据中的每一行中的各种用户标识（uid,imei,mac,androidid,uuid,imsi ）
3. 根据提取出来的这些标识，生成图计算中的 vertex 集合，以及生成图计算中的 Edge 集合
4. 然后将点集合和边集合，构造一张图 Graph
5. 然后调用图的算法（最大连通子图算法），得到结果图
6. 然后从结果图上，取所有的点集合（就是映射结果）
7. 整理结果，成为我们想要的形式

**代码放到文末**

## 最终结果形式

|  |
| --- |
| (id 映射字典—idmapping 字典)： |
| imei01:gid01 mac01:gid01 idfa01:gid01 uuid01:gid01 138002:gid02  imei02:gid02  android02:gid02 |

1. object IdMapFunction {
3. def idMap: Dataset[String] = {
5. // 读表
6. val tbSource: String = HIVE\_TABLE\_ODS\_EVENT\_LOG
8. **import** spark.implicits.\_
9. val sql\_select: String = s"select line from $tbSource where dt='$currentDate'";
11. // 查询今天日志并转换为DS
12. val appLog: Dataset[String] = spark.sql(sql\_select).as[String]
13. //        val appLog = spark.read.textFile("file:///D:/data/mockData/doit.mall.access\_2020-07-01.log")
14. integration(appLog)
15. // 返回读取的DataSet和创建的idMap
16. appLog
17. }
19. def integration(appLog: Dataset[String]): Unit = {
21. // 读取路径
22. val root: String = ConfigurationManager.config.getString(Constants.PATH\_ID\_MAP)
23. val preDayPath: String = root + currentDateBefore + "/\*.parquet"
24. val todayPath: String = root + currentDate

27. // 二、提取每一类数据中每一行的标识字段
28. val app\_ids: RDD[Array[String]] = extractIds(appLog)
30. val ids: RDD[Array[String]] = app\_ids
32. // 三、构造图计算中的vertex集合
33. val vertices: RDD[(Long, String)] = ids.flatMap(arr => {
34. **for** (biaoshi <- arr) yield (biaoshi.hashCode.toLong, biaoshi)
35. })
37. // 四、构造图计算中的Edge集合
38. val edges: RDD[Edge[String]] = ids.flatMap(arr => {
39. // 用双层for循环，来对一个数组中所有的标识进行两两组合成边
40. // [a,b,c,d] ==>   a-b  a-c  a-d  b-c  b-d  c-d
41. **for** (i <- 0 to arr.length - 2; j <- i + 1 until arr.length) yield Edge(arr(i).hashCode.toLong, arr(j).hashCode.toLong, "")
42. })
43. // 将边变成 （边,1)来计算一个边出现的次数
44. .map(edge => (edge, 1))
45. .reduceByKey(\_ + \_)
46. // 过滤掉出现次数小于经验阈值的边
47. .filter(tp => tp.\_2 > 2)
48. .map(tp => tp.\_1)
50. **if** (!FileUtils.pathIsExist(spark, preDayPath)) {
51. val graph = Graph(vertices, edges)
53. // VertexRDD[VertexId] ==>  RDD[(点id-Long,组中的最小值)]
54. val res\_tuples: VertexRDD[VertexId] = graph.connectedComponents().vertices
56. // 可以直接用图计算所产生的结果中的组最小值，作为这一组的guid（当然，也可以自己另外生成一个UUID来作为GUID）
57. **import** spark.implicits.\_
58. // 保存结果
59. res\_tuples.toDF("biaoshi\_hashcode", "guid").write.parquet(todayPath)
61. **return**
62. }
64. // 五、将上一日的idmp映射字典，解析成点、边集合
65. val schema = **new** StructType()
66. .add("biaoshi\_hashcode", DataTypes.LongType)
67. .add("guid", DataTypes.LongType)
69. val preDayIdmp = spark.read.schema(schema).parquet(preDayPath)
71. // 构造点集合
72. val preDayIdmpVertices = preDayIdmp.rdd.map({
73. **case** Row(idFlag: VertexId, guid: VertexId) =>
74. (idFlag, "")
75. })
77. // 构造边集合
78. val preDayEdges = preDayIdmp.rdd.map(row => {
79. val idFlag = row.getAs[VertexId]("biaoshi\_hashcode")
80. val guid = row.getAs[VertexId]("guid")
81. Edge(idFlag, guid, "")
82. })
84. // 将当日的点集合union上日的点集合，当日的边集合union上日的边集合，构造图，并调用连通子图算法
85. val graph = Graph(vertices.union(preDayIdmpVertices), edges.union(preDayEdges))
87. // VertexRDD[VertexId] ==>  RDD[(点id-Long,组中的最小值)]
88. val res\_tuples: VertexRDD[VertexId] = graph.connectedComponents().vertices
90. // 八、将结果跟上日的映射字典做对比，调整guid
91. // 1.将上日的idmp映射结果字典收集到driver端，并广播
92. val preIdMap = preDayIdmp.rdd.map(row => {
93. val idFlag = row.getAs[VertexId]("biaoshi\_hashcode")
94. val guid = row.getAs[VertexId]("guid")
95. (idFlag, guid)
96. }).collectAsMap()
97. val bc = sc.broadcast(preIdMap)
99. // 2.将今日的图计算结果按照guid分组,然后去跟上日的映射字典进行对比
100. val todayIdmpResult: RDD[(VertexId, VertexId)] = res\_tuples.map(tp => (tp.\_2, tp.\_1))
101. .groupByKey()
102. .mapPartitions(iter => {
103. // 从广播变量中取出上日的idmp映射字典
104. val idmpMap = bc.value
105. iter.map(tp => {
106. // 当日的guid计算结果
107. var todayGuid = tp.\_1
108. // 这一组中的所有id标识
109. val ids = tp.\_2
111. // 遍历这一组id，挨个去上日的idmp映射字典中查找
112. var find = **false**
113. **for** (elem <- ids **if** !find) {
114. val maybeGuid: Option[Long] = idmpMap.get(elem)
115. // 如果这个id在昨天的映射字典中找到了，那么就用昨天的guid替换掉今天这一组的guid
116. **if** (maybeGuid.isDefined) {
117. todayGuid = maybeGuid.get
118. find = **true**
119. }
120. }
121. (todayGuid, ids)
122. })
123. })
124. .flatMap(tp => {
125. val ids = tp.\_2
126. val guid = tp.\_1
127. **for** (elem <- ids) yield (elem, guid)
128. })
130. // 可以直接用图计算所产生的结果中的组最小值，作为这一组的guid（当然，也可以自己另外生成一个UUID来作为GUID）
131. **import** spark.implicits.\_
132. // 保存结果
133. todayIdmpResult.coalesce(1).toDF("biaoshi\_hashcode", "guid").write.parquet(todayPath)
135. }
137. /\*\*
138. \* 从一个日志ds中提取各类标识id
139. \*
140. \* @param logDs
141. \* @return
142. \*/
143. def extractIds(logDs: Dataset[String]): RDD[Array[String]] = {
145. logDs.rdd.map(line => {
147. // 将一行数据解析成json对象
148. val jsonObj = JsonUtils.getJSON(line)
150. // 从json对象中取user对象
151. val userObj = jsonObj.getJSONObject("user")
152. val uid = userObj.getString("uid")
154. // 从user对象中取phone对象
155. val phoneObj = userObj.getJSONObject("phone")
156. val imei = phoneObj.getString("imei")
157. val mac = phoneObj.getString("mac")
158. val imsi = phoneObj.getString("imsi")
159. val androidId = phoneObj.getString("androidId")
160. val uuid = phoneObj.getString("uuid")
162. Array(uid, imei, mac, imsi, androidId, uuid).filter(StringUtils.isNotBlank(\_))
163. })
164. }
165. }