尚硅谷电商用户画像

版本：V 1.0

张晨

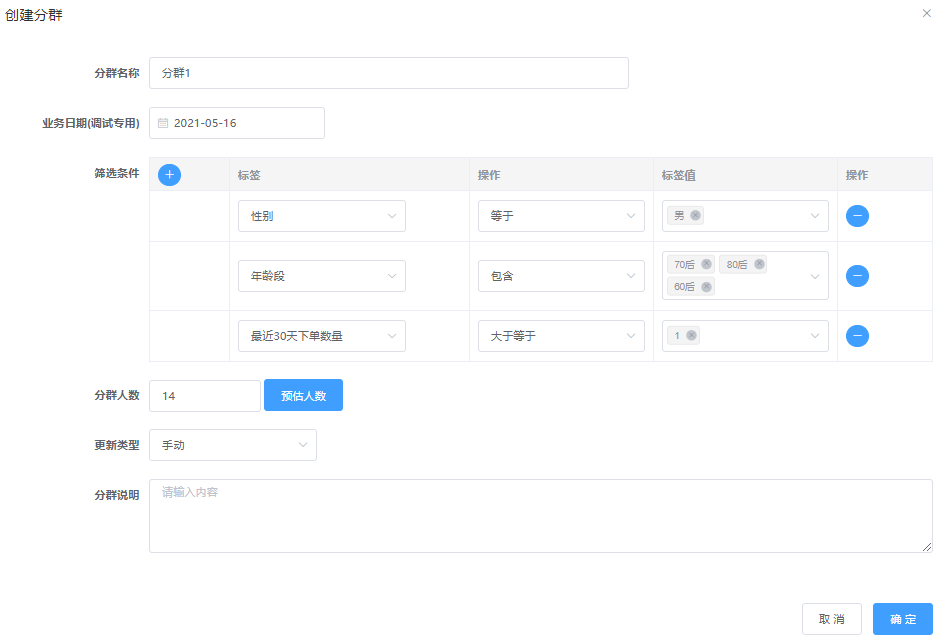
# 第九章 开发任务五 用户分群

## 9.1 功能需求：

1）点击分区【创建分群】



2）填写分群信息



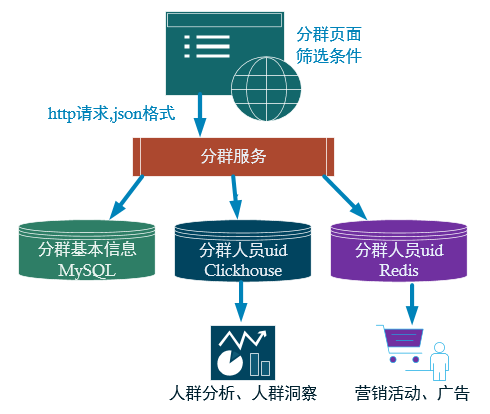
需要填写的内容

|  |  |
| --- | --- |
| 字段 |  |
| 分群名称 | 用户分群的信息 |
| 业务日期（调试） | 实际生产环境应该是当前日期的前一日 |
| 筛选条件 | 根据三级标签进行筛选， |
| 分群人数 | 用于圈选的用户一边调整筛选条件一边了解人群数 |
| 更新类型 | 自动就是每日定时更新，手动更新就是通过界面手动触发。 |

3）查看分群列表



### 结构分析：



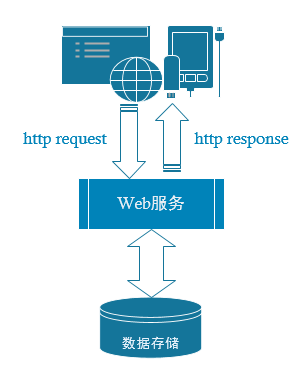
分群服务提供主要几个功能：

1. 接收页面传来的分群信息
2. 把分群基本信息保存在MySQL作为管理数据。
3. 利用clickhouse计算出分群的人群结果(Bitmap)保存起来，用于后续对该人群进行统计分析。
4. 将人群uid写入redis，用于营销、广告应用的高频访问数据。

## 9.2 Web服务搭建

### 9.2.1 Web服务概况：

Web服务（http服务），接收网页或者App应用发起的http请求，然后通过程序进行处理、计算、查询、存储等，再将结果返回给页面或App应用。



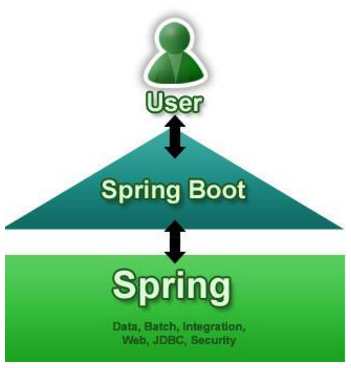
### graphic 9.2.2 Spring boot简介

Spring Boot 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。 该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。

### 9.2.3 Springboot和ssm的关系

springboot整合了springmvc ，spring等核心功能。也就是说本质上实现功能的还是原有的spring ,springmvc的包，但是springboot单独包装了一层，这样用户就不必直接对springmvc， spring等，在xml中配置。

### 9.2.4 没有xml，我们要去哪配置

* springboot实际上就是把以前需要用户手工配置的部分，全部作为默认项。除非用户需要额外更改不然不用配置。这就是所谓的：“约定大于配置”

如果需要特别配置的时候，去修改application.properties （application.yml）

### 9.2.5 Springboot 的优势

* 不再需要那些千篇一律，繁琐的xml文件。
* 内嵌Tomcat,不再需要外部的Tomcat
* 更方便的和各个第三方工具（mysql,redis,elasticsearch,dubbo,kafka等等整合），而只要维护一个配置文件即可。

### 9.2.6 数据层 mybatis

MyBatis 是一款优秀的持久层框架，它支持自定义 SQL。 MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis 可以通过简单的 XML 或注解来配置和映射。

#### 创建表

|  |
| --- |
| CREATE TABLE `customer` (   `id` bigint(20) NOT NULL AUTO\_INCREMENT,   `address` varchar(20) DEFAULT NULL,   `age` bigint(20) DEFAULT NULL,   `gender` varchar(20) DEFAULT NULL,   PRIMARY KEY (`id`) ) ENGINE=InnoDB DEFAULT CHARSET=utf8 |

#### 引入依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>2.1.4</**version**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-jdbc</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**scope**>runtime</**scope**> </**dependency**> |

#### 创建Mapper类

|  |
| --- |
| @Mapper **public interface** CustomerMapper {   @Select(**"select \* from customer "**)  **public** List<Customer> getCustomerList();   @Select(**"select \* from customer where id=#{id}"**)  **public** Customer getCustomerById(String id );  @Insert(**"insert into customer values(null, #{customer.address},#{customer.age},#{customer.gender}) "**)  **public void** insertCustomer (@Param(**"customer"**) Customer customer );  @Insert(**"update customer\_info set address= #{customer.address}, age=#{customer.age}, gender = #{customer.gender} where id=#{id}"**)  **public void** updateCustomer (@Param(**"customer"**)Customer customer );  @Insert(**"delete customer\_info where id=#{id}"**)  **public void** deleteCustomerById (String id ); } |

#### Service层中加入调用方法

|  |
| --- |
| @Service **public class** CustomerServiceImpl **implements** CustomerService {   @Autowired  CustomerMapper **customerMapper**;   @Override  **public void** saveCustomer(Customer customer) {  System.***out***.println(**"保存操作："**+customer);  **customerMapper**.insertCustomer(customer);   customer.setAge(100);  **customerMapper**.updateCustomer(customer);   **customerMapper**.deleteCustomerById(**"1"**);   Customer customerRs = **customerMapper**.getCustomerById(**"2"**);  System.***out***.println(customerRs);   List<Customer> customerList = **customerMapper**.getCustomerList();  System.***out***.println(customerList);  } } |

### 9.2.7 Mybatis-plus

（简称 MP）是一个Mybatis 的增强工具，在 MyBatis 的基础上只做增强不做改变，完全去SQL化，封装好了大量的CURD操作。 甚至把CRUD操作封装到了Service层，可以直接在controller调用现成的CRUD服务层，极度舒适省心。

**局限**： 只支持简单的CRUD 操作。不支持多表操作（join ,union,子查询） ，不支持GroupBy 和各种函数。

#### 引入mybatis依赖

|  |
| --- |
| *<!--mybatis-plus-->* <**dependency**>  <**groupId**>com.baomidou</**groupId**>  <**artifactId**>mybatis-plus-boot-starter</**artifactId**>  <**version**>3.4.1</**version**> </**dependency**> |

#### 增加Mybatis-plus规范的服务层和数据层

Mapper

|  |
| --- |
| **import** com.baomidou.mybatisplus.core.mapper.BaseMapper;  @Mapper **public interface** CustomerPlusMapper **extends** BaseMapper<Customer> { } |

Service接口

|  |
| --- |
| **public interface** CustomerPlusService **extends** IService<Customer> { } |

#### ServiceImpl实现类

|  |
| --- |
| @Service **public class** CustomerPlusImpl **extends** ServiceImpl<CustomerPlusMapper, Customer> **implements** CustomerPlusService { } |

只要加上这个三个继承。在controller 中直接就可以使用CRUD命令。

#### Controller直接调用封装方法

|  |
| --- |
| @Autowired CustomerPlusService **customerPlusService**;  @RequestMapping(**"/hello"**) *//把请求路径与方法做映射* **public** String getHelloWorld(@RequestParam(**"name"**) String name , @RequestBody Customer customer ){  System.***out***.println(**"Customer:"**+customer);  **customerService**.saveCustomer(customer);  **return "hello ,"**+name; }  @GetMapping(**"/customer/list"**) **public** List<Customer> getCustomerList(){  List<Customer> list = **customerPlusService**.list();  **return** list ; } @GetMapping(**"/customer"**) **public** Customer getCustomer(@RequestParam(**"id"**) String id){  **return customerPlusService**.getById(Integer.*valueOf*(id)); *//直接返回会自动转为json (jackson)* }  @PostMapping(**"/customer"**) **public** String saveCustomer(@RequestBody Customer customer){  *//会根据主键是否为空来决定 是插入还是修改* **customerPlusService**.saveOrUpdate(customer);  **return "success"**; }  @DeleteMapping(**"/customer"**) **public** String deleteCustomer(@RequestParam(**"id"**) String id){  *//会根据主键是否为空来决定 是插入还是修改* **customerPlusService**.removeById(id);  **return "success"**; } |

#### 在Bean上加入对主键的标识

|  |
| --- |
| @Data @NoArgsConstructor @AllArgsConstructor **public class** Customer {  @TableId(value = **"id"**,type = IdType.***AUTO***) //标识根据数据库自增生成主键  String **id** ;  String **address**;  Integer **age**;  String **gender** ; } |

### 9.2.8 应用开发的分层

|  |  |  |
| --- | --- | --- |
| 分层 | 职责 | 涉及的标签 |
| controller | 主要用于对接http请求，返回http响应。 | @RestController  @RequestMapping  @RequestParam  @RequestBody |
| service | 用于处理业务请求 | @Service  @Autowired |
| mapper | 用于处理数据库操作请求 | @Select  @Insert |

|  |  |  |
| --- | --- | --- |
| 标签 | 职责 | 工作层级 |
| @RestController | 标识请求的入口类 类上 | controller层 |
| @RequestMapping | 标识请求入口方法 | controller层 |
| @RequestParam | 接收请求URL路径中的参数 | controller层 |
| @RequestBody | 接收Post请求体中的数据 | controller层 |
| @Autowired | 自动将接口实例化 | controller层、service层 |
| @Service | 标识了业务实现类的组件 | service层 |
| @Select | 标识要执行的select语句 | mapper |
| @Insert | 标识要执行的insert语句 | mapper |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 控制层 | UserGroupController | 增加两个方法  1添加分群  2预估分群人数  3 更新分群数据  职责：  a 接收参数  b 调用业务方法  c 根据格式执行结果 |
| 服务层 | UserGroupService | 增加方法的声明   1. 保存分群基本信息 2. 生成clickhouse 分群用户集合 3. 生成 redis 分群用户集合 4. 根据查询条件返回预估人数 5. 更新分群用户集合 |
| UserGroupServiceImpl | 1. 保存分群基本信息 （由MybatisPlus提供） 2. 生成clickhouse 分群用户集合 3. 生成 redis 分群用户集合 4. 根据查询条件返回预估人数   5 更新分群用户集合 |
| 数据层 | **UserGroupMapper** | 1. 对UserGroup 在MySQL基本信息的保存修改 (由Mybatis提供) 2. 插入分群用户集合到Clickhouse中 3. 更新分群用户集合到Clickhouse中   4 根据筛选条件查询Clickhouse，得到分群人数。 |

## 9.3 分群模块初始搭建

### 9.3.1 涉及到的配置

pom.xml

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-jdbc</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**> </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.47</**version**> </**dependency**>  *<!-- https://mvnrepository.com/artifact/org.apache.commons/commons-lang3 -->* <**dependency**>  <**groupId**>org.apache.commons</**groupId**>  <**artifactId**>commons-lang3</**artifactId**>  <**version**>3.11</**version**> </**dependency**> <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>fastjson</**artifactId**>  <**version**>1.2.68</**version**> </**dependency**>  <**dependency**>  <**groupId**>org.projectlombok</**groupId**>  <**artifactId**>lombok</**artifactId**>  <**optional**>true</**optional**> </**dependency**>  *<!--mybatis-plus-->* <**dependency**>  <**groupId**>com.baomidou</**groupId**>  <**artifactId**>mybatis-plus-boot-starter</**artifactId**>  <**version**>3.4.1</**version**> </**dependency**>  <**dependency**>  <**groupId**>com.baomidou</**groupId**>  <**artifactId**>dynamic-datasource-spring-boot-starter</**artifactId**>  <**version**>3.3.2</**version**> </**dependency**> <**dependency**>  <**groupId**>ru.yandex.clickhouse</**groupId**>  <**artifactId**>clickhouse-jdbc</**artifactId**>  <**version**>0.1.55</**version**> </**dependency**> |

application.properties

|  |
| --- |
| ***#后台服务端口号* server.port**=**80  *#用于画像管理平台的数据* spring.datasource.dynamic.datasource.mysql.url**=**jdbc:mysql://bigdata01:3306/user\_profile\_manager?characterEncoding=utf-8&useSSL=false spring.datasource.dynamic.datasource.mysql.driver-class-name**=**com.mysql.jdbc.Driver spring.datasource.dynamic.datasource.mysql.username**=**root spring.datasource.dynamic.datasource.mysql.password**=**Flzx3qc!   *#用于连接clickhouse 保存标签计算结果 和 人群包 如果这里没装clickhouse 这里可以先注掉* spring.datasource.dynamic.datasource.clickhouse.url**=**jdbc:clickhouse://bigdata01:8123/user\_profile2077 spring.datasource.dynamic.datasource.clickhouse.driver-class-name**=**ru.yandex.clickhouse.ClickHouseDriver    *#mybaits-plus空值处理* mybatis-plus.global-config.db-config.field-strategy**=**not\_null   logging.level.com.atguigu.userprofile.mapper**=**debug** |

### 9.3.2 clickhouse 增加表

|  |
| --- |
| CREATE TABLE user\_group  (  `user\_group\_id` String,  `us` AggregateFunction(groupBitmap, UInt64)  )  ENGINE = AggregatingMergeTree  PARTITION BY user\_group\_id  ORDER BY user\_group\_id |

### 9.3.3 实体Bean

|  |
| --- |
| @Data public class UserGroup implements Serializable {   private static final long *serialVersionUID* = 1L;   @TableId(value = "id", type = IdType.*AUTO*)  private Long id;   private String userGroupName;   private String conditionJsonStr;   @TableField(exist = false)  private List<TagCondition> tagConditions;   private String conditionComment;   private Long userGroupNum;   private String updateType;   private String userGroupComment;   private Date updateTime;   private Date createTime;  @TableField(exist = false)  private String busiDate;   public String conditionJsonToComment(){  StringBuilder comment=new StringBuilder();  for (TagCondition tagCondition : tagConditions) {  comment.append(tagCondition.tagName+ " "+tagCondition.operatorName+" "+ StringUtils.*join*(tagCondition.getTagValues(),",")+" ;\n");  }  return comment.toString();  }  } |

## 9.4 功能一 ：生成分群基本信息

### 9.4.1 controller层

|  |
| --- |
| @RestController **public class** UserGroupController {   @Autowired  UserGroupService **userGroupService**;  *//添加分群* @PostMapping(**"/user-group"**) @CrossOrigin **public** String saveUserGroup(@RequestBody UserGroup userGroup){  System.***out***.println(userGroup);  *//保存基本信息(mysql)* **userGroupService**.saveUserGroup(userGroup);  *//生成分群人群包 （clickhouse、Redis)* **userGroupService**.genUserGroupUids(userGroup);  **return "success"**; }  } |

### 9.4.2 service层

该方法调用了 两个服务层方法

1. 把分群基本信息保存在Mysql
2. 生成分群的人群id集合，写入clickhouse及redis

业务服务层接口增加两个方法

UserGroupService

|  |
| --- |
| **public interface** UserGroupService **extends** IService<UserGroup> {  {  *//保存基本信息* **public void** saveUserGroup(UserGroup userGroup);    } |

UserGroupServiceImpl

|  |
| --- |
| @Service @Slf4j **public class** UserGroupServiceImpl **extends** ServiceImpl<UserGroupMapper, UserGroup> **implements** UserGroupService {  @Autowired UserGroupMapper **userGroupMapper**;  */\*\*  \* 保存基本信息  \** ***@param userGroup*** *\*/* **public void** saveUserGroup(UserGroup userGroup){  *//把筛选条件列表转换为文字描述，和json* userGroup.setConditionComment(userGroup.conditionJsonToComment());  userGroup.setConditionJsonStr(JSON.*toJSONString*( userGroup.getTagConditions()));  *//生成创建时间* userGroup.setCreateTime(**new** Date());  saveOrUpdate(userGroup); }  } |

其中实现的类的父类是ServiceImpl<UserGroupMapper, UserGroup>

这个是Mybatis-plus的依赖提供的类，专门提供封装好的服务层的基础插删改查的通用方法。

其中saveOrUpdate()即负责新增，也用于修改，新增还是修改取决于实体Bean中所标识的主键（@TableId(value = **"id"**)）是否为空。

Mybatis-plus的依赖

|  |
| --- |
| *<!--mybatis-plus-->* <**dependency**>  <**groupId**>com.baomidou</**groupId**>  <**artifactId**>mybatis-plus-boot-starter</**artifactId**>  <**version**>3.4.1</**version**> </**dependency**> |

### 9.4.3 mapper层

|  |
| --- |
| @DS(**"mysql"**) **public interface** UserGroupMapper **extends** BaseMapper<UserGroup> {  } |

此处虽然没有声明方法，但是实际上为父类BaseMapper<UserGroup>标记了数据源，而对UserGroup进行最基本的插删改查都是使用了BaseMapper<UserGroup>的方法。

#### 关于Jdbc数据源

这里使用了一个为了支持多个JDBC数据源的工具包

|  |
| --- |
| <**dependency**>  <**groupId**>com.baomidou</**groupId**>  <**artifactId**>dynamic-datasource-spring-boot-starter</**artifactId**>  <**version**>3.3.2</**version**> </**dependency**> |

该工具包，可以通过配置文件+注解来实现不同JDBC数据源的使用。

在配置文件application.properties中,通过dynamic来配置数据源

|  |
| --- |
| ***#用于画像管理平台的数据* spring.datasource.dynamic.datasource.mysql.url**=**jdbc:mysql://bigdata01:3306/user\_profile\_manager?characterEncoding=utf-8&useSSL=false spring.datasource.dynamic.datasource.mysql.driver-class-name**=**com.mysql.jdbc.Driver spring.datasource.dynamic.datasource.mysql.username**=**root spring.datasource.dynamic.datasource.mysql.password**=**Flzx3qc!   *#用于连接clickhouse 保存标签计算结果 和 人群包 如果这里没装clickhouse 这里可以先注掉* spring.datasource.dynamic.datasource.clickhouse.url**=**jdbc:clickhouse://bigdata01:8123/user\_profile2077 spring.datasource.dynamic.datasource.clickhouse.driver-class-name**=**ru.yandex.clickhouse.ClickHouseDriver** |

其中橙色部分就是数据源名称。

在代码中通过**@DS(“数据源名称”) ，**在方法或者类上标记即可。**类**上标记标识该类默认使用的数据源，**方法**上标记标识该方法使用特定的数据源。

如下，UserGroupMapper中的声明

|  |
| --- |
| @DS(**"mysql"**) **public interface** UserGroupMapper **extends** BaseMapper<UserGroup> {  } |

## 9.5 功能二 生成人群包

### 9.5.1 controller层

|  |
| --- |
| @RestController **public class** UserGroupController {   @Autowired  UserGroupService **userGroupService**;  *//添加分群* @PostMapping(**"/user-group"**) @CrossOrigin **public** String saveUserGroup(@RequestBody UserGroup userGroup){  System.***out***.println(userGroup);  *//保存基本信息(mysql)* **userGroupService**.saveUserGroup(userGroup);  *//生成分群人群包 （clickhouse、Redis)* **userGroupService**.genUserGroupUids(userGroup);  **return "success"**; }  } |

### 9.5.2 service层

UserGroupService

|  |
| --- |
| */\*\*  \* 生成人群包  \** ***@param userGroup*** *\*/* **public void** genUserGroupUids(UserGroup userGroup) ; |

|  |
| --- |
| */\*\*  \* 生成人群包  \** ***@param userGroup*** *\*/*  **public void** genUserGroupUids(UserGroup userGroup){  *//生成人群包写入clickhouse* Long userGroupCount = insertUserGroupOnCH(userGroup);  *//把clickhouse的人群包写入redis* insertUserGroupOnRedis(userGroup);  *// 更新MySQL 最后更新时间和人数* refreshUpdateTime(userGroupCount,userGroup.getId());  } |

|  |
| --- |
| **public** Long insertUserGroupOnCH(UserGroup userGroup ){  *//生成人群包先清除旧的人群包 -sql* String deleteSql = **"alter table user\_group delete where user\_group\_id= '"**+userGroup.getId()+**"'"**;   *//生成bitmap人群包的查询sql* String bitmapSql = genBitmapSql(userGroup);  *//组合成人群包的写入sql* String insertSql = **" insert into user\_group select '"**+userGroup.getId()+**"',("** + bitmapSql + **") as res"**;  *//执行sql* ***log***.info(deleteSql);  **this**.getBaseMapper().deleteBySQL(deleteSql);  ***log***.info(insertSql);  **this**.getBaseMapper().insertBySQL(insertSql);  *//查询该人群包的人数* Long userGroupCount = **this**.getBaseMapper().getUserGroupCount(userGroup.getId().toString());    **return** userGroupCount; } |

|  |
| --- |
| /\* bitmapAnd(  bitmapAnd(  (  SELECT groupBitmapMergeState(us) AS us  FROM user\_tag\_value\_string  WHERE (dt = '2021-05-16') AND (tag\_code = 'TG\_PERSON\_BASE\_GENDER') AND (tag\_value IN ('男', '女'))  ),  (  SELECT groupBitmapMergeState(us) AS us  FROM user\_tag\_value\_string  WHERE (dt = '2021-05-16') AND (tag\_code = 'TG\_PERSON\_BASE\_AGEGROUP') AND (tag\_value IN ('80后', '90后'))  )  ), (  SELECT groupBitmapMergeState(us) AS us  FROM user\_tag\_value\_long  WHERE (dt = '2021-05-16') AND (tag\_code = 'TG\_BEHAVIOR\_BUY\_ORDER\_CT\_30D') AND (tag\_value >= 1) ))  \*/ **private** String genBitmapSql(UserGroup userGroup ){  *//获得基于标签编码的标签信息的map* Map<String, TagInfo> tagInfoMapWithCode = **tagInfoService**.getTagInfoMapWithCode();  List<TagCondition> tagConditionList = userGroup.getTagConditions();   StringBuilder conditionSqlAllBuilder=**new** StringBuilder();  *//循环转换每个筛选条件 依次组合为sql语句* **for** (**int** i = 0; i < tagConditionList.size(); i++) {  TagCondition tagCondition = tagConditionList.get(i);  TagInfo tagInfo = tagInfoMapWithCode.get(tagCondition.getTagCode());  String tagValueType = tagInfo.getTagValueType();  String tableName=**""**;  String condition=**""**;  *// 把操作代码转换为 判断符号* String conditionOperator = getConditionOperator(tagCondition.getOperator());   **if** (tagValueType.equals(ConstCodes.***TAG\_VALUE\_TYPE\_LONG***)){  tableName=**"user\_tag\_value\_long"**;  *//根据操作类型 和 值类型 来 获得 值部分的SQL表达式（比如要不要加括号，或者单引)* String tagValue=getTagValue(tagCondition.getTagValues(),conditionOperator,**false**);  condition = conditionOperator+**" "**+tagValue;  }**else if** (tagValueType.equals(ConstCodes.***TAG\_VALUE\_TYPE\_DECIMAL***)){  tableName=**"user\_tag\_value\_decimal"**;  String tagValue=getTagValue(tagCondition.getTagValues(),conditionOperator,**false**);  condition = conditionOperator+**" "**+tagValue;  } **else if** (tagValueType.equals(ConstCodes.***TAG\_VALUE\_TYPE\_STRING***)){  tableName=**"user\_tag\_value\_string"**;  String tagValue=getTagValue(tagCondition.getTagValues(),conditionOperator,**true**);  condition = conditionOperator+**" "**+tagValue;  } **else if** (tagValueType.equals(ConstCodes.***TAG\_VALUE\_TYPE\_DATE***)){  tableName=**"user\_tag\_value\_date"**;  String tagValue=getTagValue(tagCondition.getTagValues(),conditionOperator,**true**);  condition = conditionOperator+**" "**+tagValue;  }  *//组合为单条条件语句* String conditionSql = **"(select groupBitmapMergeState(us) us from "**+ tableName +  **" where dt='"**+userGroup.getBusiDate()+**"' "** +  **"and tag\_code='"**+tagCondition.getTagCode() +**"'"** +  **" and tag\_value "**+condition+**")"**;   *//把单条条件 通过bitmapAnd 拼接* **if**(conditionSqlAllBuilder.length()>0) {  conditionSqlAllBuilder.insert(0,**"bitmapAnd ("**)  .append(**","**+conditionSql+**")"**);  }**else**{  conditionSqlAllBuilder.append(conditionSql);  }  }  **return** conditionSqlAllBuilder.toString(); } |

### 9.5.3 两个工具方法

|  |
| --- |
| *//根据操作类型 和 值类型 来 获得 值部分的SQL表达式（比如要不要加括号，或者单引)* **private** String getTagValue(List<String> tagValues,String operator,Boolean needQuote) {   String tagValueStr=**""**;  **if**(needQuote) {  tagValueStr = **"'"**+StringUtils.*join*(tagValues, **"','"**)+**"'"**;  }**else**{  tagValueStr=StringUtils.*join*(tagValues, **","**);  }  **if**(operator.indexOf(**"in"**) >=0) {  tagValueStr = **"("** + tagValueStr + **")"**;  }  **return** tagValueStr;  } |

|  |
| --- |
| *// 把中文的操作代码转换为 判断符号* **private** String getConditionOperator(String operator){  **switch** (operator){  **case "eq"**:  **return "="**;  **case "lte"**:  **return "<="**;  **case "gte"**:  **return ">="**;  **case "lt"**:  **return "<"**;  **case "gt"**:  **return ">"**;  **case "neq"**:  **return "<>"**;  **case "in"**:  **return "in"**;  **case "nin"**:  **return "not in"**;  }  **throw new** RuntimeException(**"操作符不正确"**); } |

### 9.5.4 mapper层

在代码中通过**@DS(“数据源名称”) ，**在方法或者类上标记即可。**类**上标记标识该类默认使用的数据源，**方法**上标记标识该方法使用特定的数据源。

如下，UserGroupMapper中的声明

|  |
| --- |
| @DS(**"mysql"**) **public interface** UserGroupMapper **extends** BaseMapper<UserGroup> {   *//查询预估分群人数* @Select(**"${sql}"**)  @DS(**"clickhouse"**)  **public** Long getCountBySQL(@Param(**"sql"**) String sql);  *//从clickhouse中查询分群uid集合*  @Select(**"select bitmapCardinality(us ) ct from user\_group where user\_group\_id=#{id}"**)  @DS(**"clickhouse"**) **public** Long getUserGroupCount(@Param(**"id"**) String id);  *//写入人群包到clickhouse* @Insert(**"${sql}"**)  @DS(**"clickhouse"**)  **public** Long insertBySQL(@Param(**"sql"**) String sql);   *//删除clickhouse人群包* @Delete(**"${sql}"**)  @DS(**"clickhouse"**)  **public** Long deleteBySQL(@Param(**"sql"**) String sql);  } |

这里其实采用了两种处理SQL的方法：

一种是动态的SQL，需要通过java程序比较复杂的组合SQL，然后将组合好的整条SQL语句作为，比如人群包的写入SQL。

另一种是静态的SQL，一般SQL没有结构上的变化，只有参数变化。可以把SQL写在注解中，并镂空参数，通过接口方法传入参数替换即可。

## 9.6 将人群包写入Redis

### 9.6.1 分析

为了上游的业务系统能够更方便，性能更好的使用人群包。需要把Clickhouse的人群包写入 Redis。

主要有两个操作：

1. 从clickhouse中读取Bitmap人群包，并转换为用户ID 列表。
2. 把数据写入Redis

### 9.6.2 mapper

读取clickhouse的人群包，主要靠mapper中的查询方法

|  |
| --- |
| @DS(**"mysql"**) **public interface** UserGroupMapper **extends** BaseMapper<UserGroup> {  *//从clickhouse中查询分群uid集合* @Select(**"select arrayJoin(bitmapToArray(us )) uids from user\_group where user\_group\_id=#{id}"**)  @DS(**"clickhouse"**)  **public** List<String> getListFromBitmap(@Param(**"id"**) String id); } |

Service层

实现类中增加Redis的写入方法

|  |
| --- |
| *//从clickhouse中读取人群包，写入redis* **public void** insertUserGroupOnRedis(UserGroup userGroup){  Jedis jedis = RedisUtil.*getJedisFromPool*();   List<String> userGroupUidList = getUserGroupUidList(userGroup);  jedis.del(**"user\_group:"**+userGroup.getId());  jedis.sadd(**"user\_group:"**+userGroup.getId(),userGroupUidList.toArray(**new** String[userGroupUidList.size()] ));  jedis.close(); } |

### 9.6.3 工具类和配置

工具类中增加RedisUtil的连接工具

配置类

|  |
| --- |
| @Configuration **public class** RedisUtil {    **public static** String *redisHost*;   **public static** Integer *redisPort*;   @Value(**"${spring.redis.host}"**)  **public void** setRedisHost(String redisHost){  RedisUtil.*redisHost*=redisHost;  }  @Value(**"${spring.redis.port}"**)  **public void** setRedisPort(String redisPort){  RedisUtil.*redisPort*=Integer.*valueOf*(redisPort);  }   **private static** JedisPool *jedisPool*=**null**;   **public static** Jedis getJedisFromPool(){  **if**(*jedisPool*==**null**){  JedisPoolConfig jedisPoolConfig =**new** JedisPoolConfig();  jedisPoolConfig.setMaxTotal(200); *//最大可用连接数* jedisPoolConfig.setMaxIdle(50); *//最大闲置连接数* jedisPoolConfig.setMinIdle(20); *//最小闲置连接数* jedisPoolConfig.setBlockWhenExhausted(**true**); *//连接耗尽是否等待* jedisPoolConfig.setMaxWaitMillis(2000); *//等待时间* jedisPoolConfig.setTestOnBorrow(**true**); *//取连接的时候进行一下测试 ping pong   jedisPool*=**new** JedisPool(jedisPoolConfig,*redisHost*, Integer.*valueOf*(*redisPort*) );  **return** *jedisPool*.getResource();  }**else**{  **return** *jedisPool*.getResource();  }  }   } |

其中 @Configuration 作用是能够使用@Value标签，把配置文件中的redis配置提取到变量中。

在application.properties加入

|  |
| --- |
| ***#redis* spring.redis.host**=**bigdata01 spring.redis.port**=**6379** |

## 9.7 预估人群数

### 9.7.1 controller层

|  |
| --- |
| *//预估分群人数* @PostMapping(**"/user-group-evaluate"**) @CrossOrigin **public** Long userGroupEvaluate(@RequestBody UserGroup userGroup){  System.***out***.println(userGroup);  userGroup.setBusiDate(userGroup.getBusiDate() );  Long userGroupCount = **userGroupService**.getUserGroupCount(userGroup);  **return** userGroupCount; } |

### 9.7.2 service层

增加预估人数方法

|  |
| --- |
| *//预估人群* **public** Long getUserGroupCount(UserGroup userGroup ); |

实现类方法

|  |
| --- |
| **public** Long getUserGroupCount(UserGroup userGroup ){  String bitmapSql = genBitmapSql(userGroup);  String countSql = **"select bitmapCardinality( "** + bitmapSql + **") as res"**;  **return this**.getBaseMapper().getCountBySQL(countSql); } |

## 9.8 练习 ： 补充手动更新方法