大数据项目之用户画像

（即席分群）

版本：V3.0

# 第九章 开发任务五之用户即席分群

## 9.1 功能需求

**1）点击【创建分群】**



**2）填写分群信息**



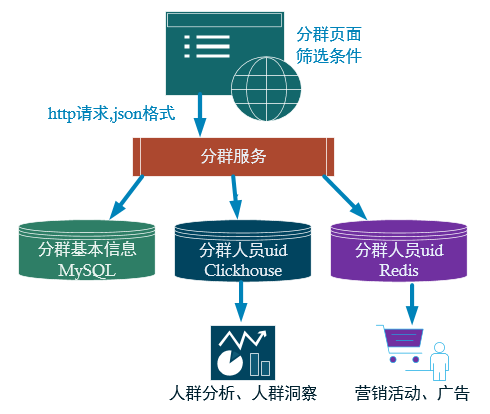
需要填写的内容说明

|  |  |
| --- | --- |
| **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**



## 9.2 流程分析



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。

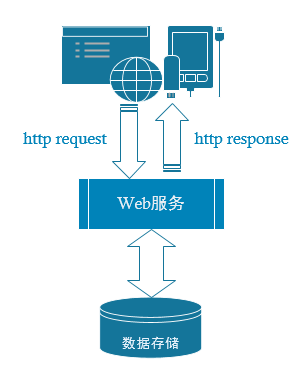
（3）利用clickhouse计算出分群的人群结果(Bitmap)保存起来，用于后续对该人群进行统计分析。

（4）将人群uid写入redis，用于营销、广告应用的高频访问数据。

## 9.3 Web服务搭建

### 9.3.1 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。



### 9.3.2 Springboot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。



### 9.3.3 Springboot优势

（1）不再需要那些千篇一律，繁琐的xml文件。

（2）内嵌Tomcat,不再需要外部的Tomcat

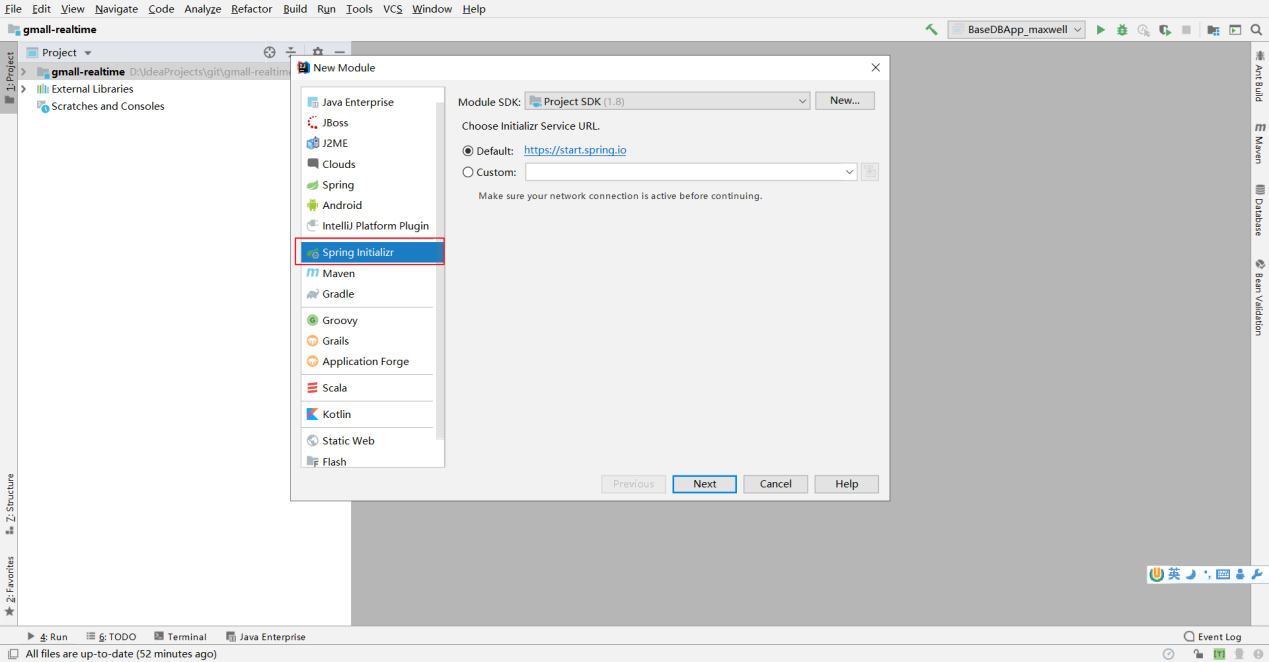
（3）更方便的和各个第三方工具（mysql,redis,elasticsearch,dubbo,kafka等等整合），而只要维护一个配置文件即可。

### 9.3.4 Springboot和ssm的关系

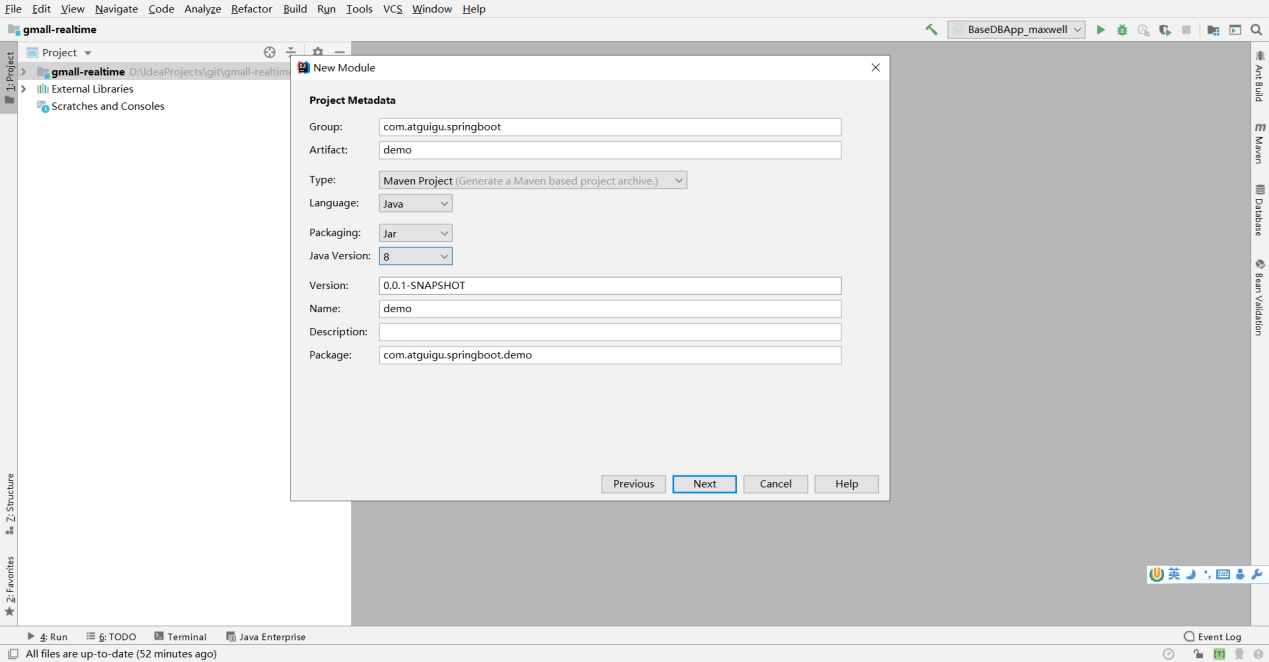
springboot整合了springmvc ,spring,mybatis 等核心功能。也就是说本质上实现功能的还是原有的spring ,springmvc的包，但是springboot单独包装了一层，这样用户就不必直接对springmvc， spring等，在xml中配置。

### 创建springboot 模块

1. **增加新模块，选择springboot初始模板**



**2）填写模块路径**

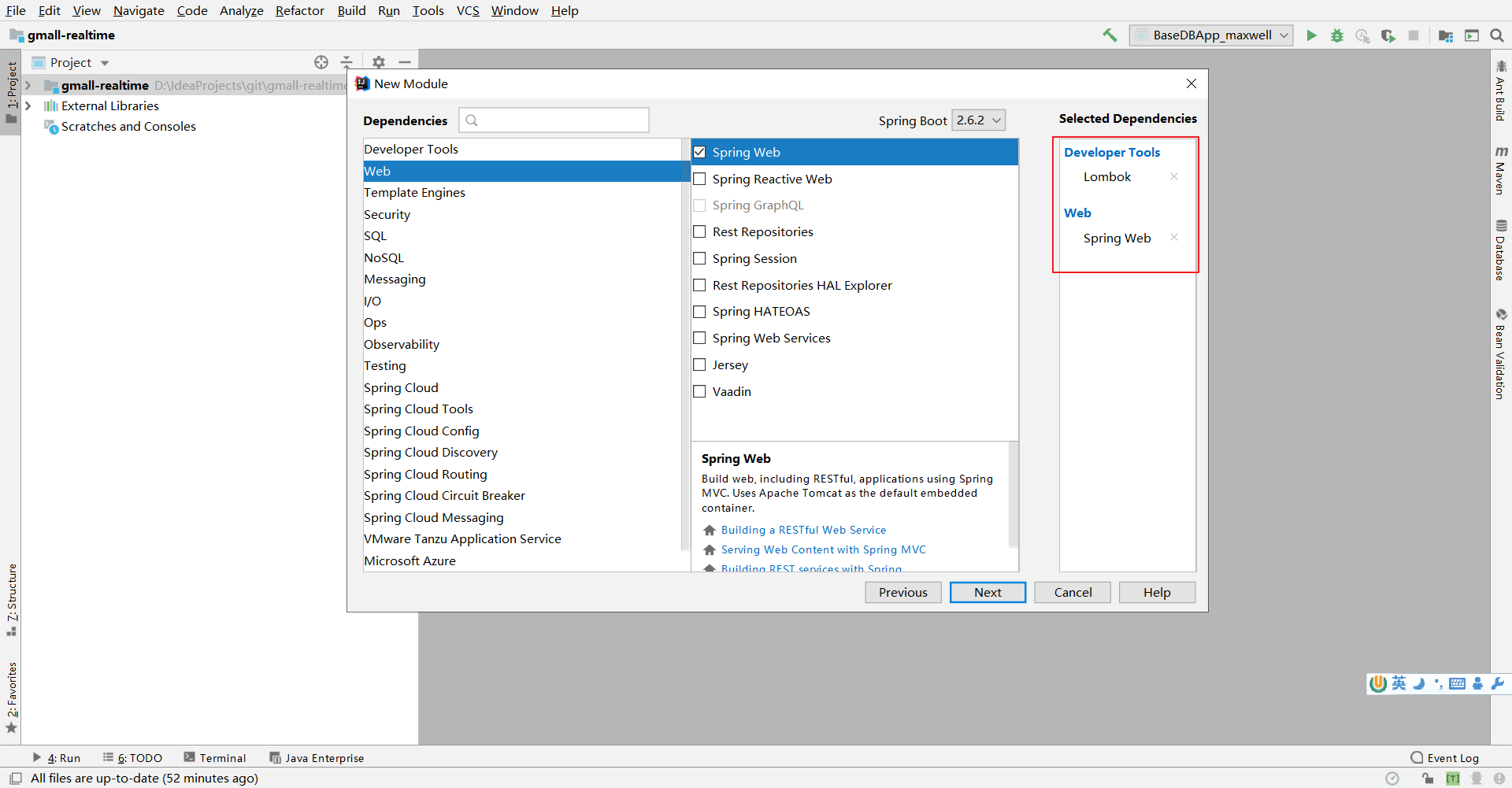


**3）选择依赖**

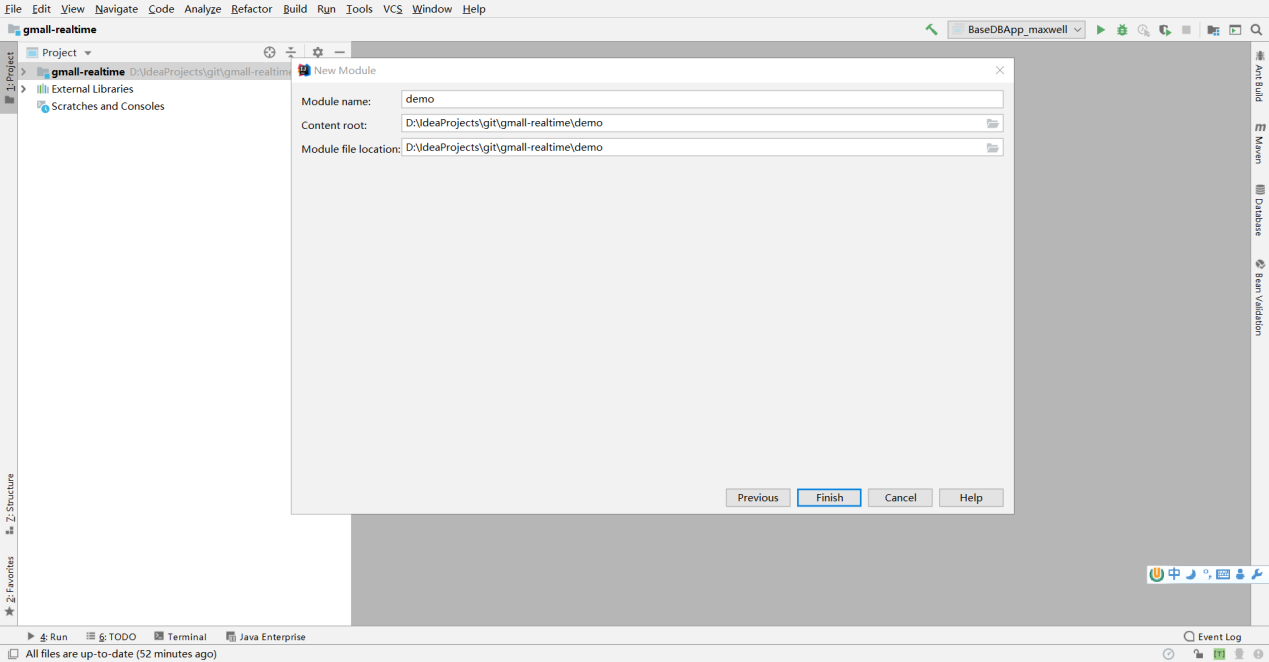
选择spring web 包括web应用的常用注解，以及web容器（默认tomcat）。

选择lombok，修饰实体类的注解插件。

这里不选也可以在之后去pom.xml里添加依赖。



**4）选择路径**

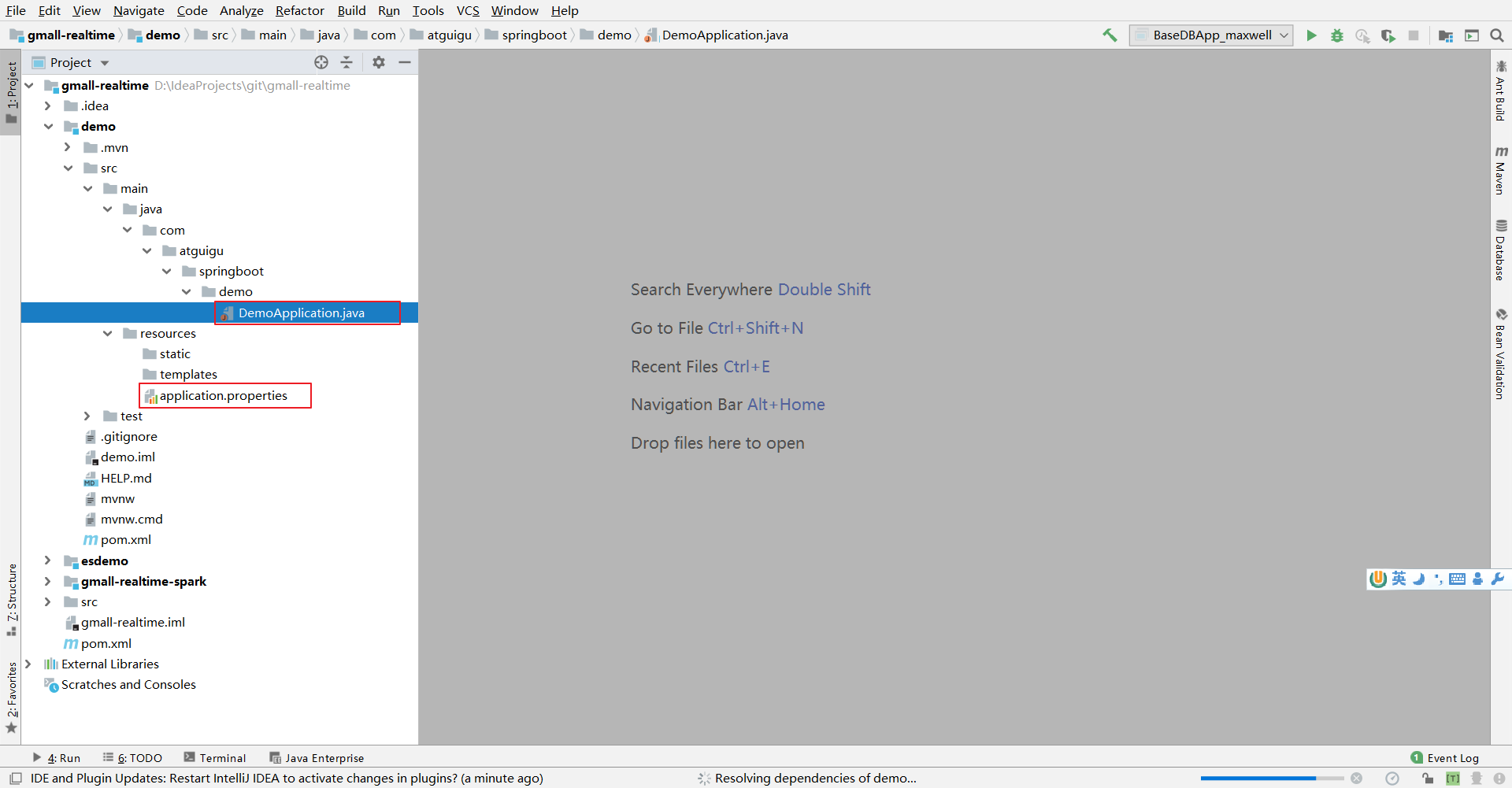


**5）自动产生的文件**

创建好后的，springboot的工程比maven的工程多了两个文件。其中：

DemoCustomerApplication 是整个程序的启动类。

application.properties 是整个工程的配置文件。



### 9.3.6 应用开发的分层

web的应用开发，经历了数十年发展至今已经有一套比较成熟的分层结构，目的就是让程序更加的清晰，各层之间各司其职。任何一个web应用开发程序员接手一个新项目都可以很快上手，就是因为大家都在履行这种分层约定。

|  |  |  |  |
| --- | --- | --- | --- |
| **分层** | **名称** | **职责** | **涉及的注解** |
| **controller** | 控制层 | 1接收请求和参数  2调用服务层  3 返回响应和结果 | @RestController  @RequestMapping  @RequestParam  @RequestBody  @PathVariable  @GetMapping  @PostMapping |
| **service** | 服务层（业务层） | 1接收控制层调用  2 处理业务  3 调用数据层  4 返回结果 | @Service  @Autowired |
| **mapper** | 数据层 | 1接收服务层调用  2数据库的操作 （SQL）  3 返回结果 | @Select  @Insert |

### 9.3.7 controller层

9.3.7.1 职责

（1）接收请求和参数

（2）调用服务层

（3）返回响应和结果

9.3.7.2 相关注解

|  |  |  |
| --- | --- | --- |
| **注解** | **位置** |  |
| **@RestController** | 类 | 标识入口类 |
| **@RequestMapping** | 方法 | 标识入口方法 |
| **@GetMapping** | 方法 | 标识入口方法GET请求专用 |
| **@PostMapping** | 方法 | 标识入口方法POST请求专用 |
| **@RequestParam** | 参数 | 接收路径上的键值对参数  http://xxxxx/xx?name=xxx&age=xxx |
| **@PathVariable** | 参数 | 接收路径上的值  http://xxxxx/customer/123 |
| **@RequestBody** | 参数 | 接收请求体(payload) 中的参数 |

9.3.7.3 HelloWorld

**1）请求打通**

package com.atguigu.springboot.controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

/\*\*

\* @RestController将类标识为Controller ， 作为请求的入口类。

\*/

@RestController

public class CustomerController {

/\*\*

\* 请求处理方法

\* 通过@RequestMapping将请求与方法进行映射

\*/

@RequestMapping("/hello")

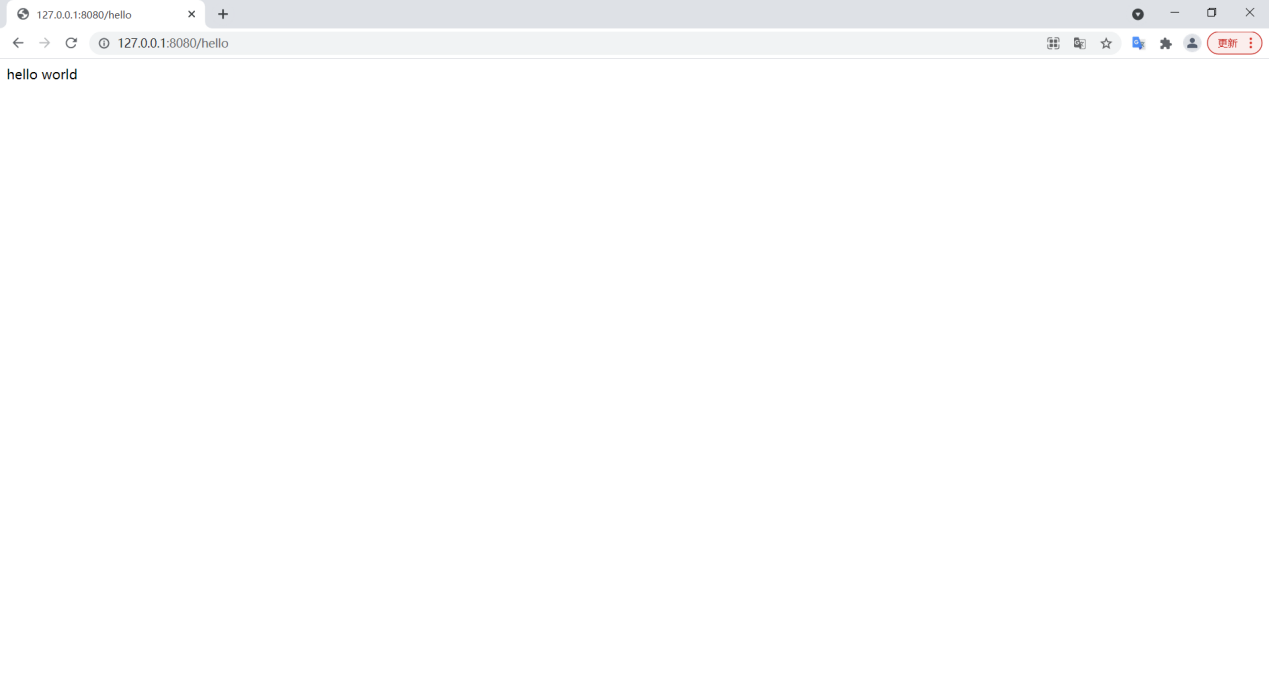
public String helloController(){

return "hello world" ;

}

}

**2）浏览器地址栏输入 127.0.0.1:8080/hello 进行访问**



9.3.7.4 请求参数

**1）请求路径中的键值对参数**

例如: <http://127.0.0.1:8080/hello?name=zs&age=22>

/\*\*

\* 请求处理方法

\*

\* @RequestParam 用来将请求路径中的键值对参数映射到方法的形参上

\*/

@RequestMapping("/hello")

public String helloController(@RequestParam("name") String name , @RequestParam("age") Integer age ){

return "hello world , name = " + name + " , age = "+ age ;

}

**2）请求路径中嵌入的参数**

例如: <http://127.0.0.1:8080/hello/lisi/33>

/\*\*

\* 请求处理方法

\*

\* @PathVariable 用来将请求路径中嵌入的参数映射到方法的形参上

\*/

@RequestMapping("/hello/{name}/{age}")

public String helloController(@PathVariable("name") String name , @PathVariable("age") Integer age ){

return "hello world , name = " + name + " , age = "+ age ;

}

**3）请求体中的参数， 单个参数映射**

/\*\*

\* 请求处理方法

\*

\* method 用来指定接收的请求类型

\* GET : 一般是用来查

\* POST: 一般是用来写

\*

\* @GetMapping = @RequestMapping + RequestMethod.GET

\* @PostMapping = @RequestMapping + RequestMethod.POST

\*

\* 如果请求参数名和方法形参名一致，可以不通过@RequestMapping指定

\*

\*/

//@RequestMapping(value = "/hello" ,method = RequestMethod.POST)

@PostMapping("/hello")

public String helloController(String name , Integer age ){

return "hello world , name = " + name + " , age = "+ age ;

}

**4）参数直接映射到bean**

/\*\*

\* 请求处理方法

\* @RequestBody 用来将请求体中的请求参数映射到Bean类中对应的属性上

\*/

@PostMapping("/hello")

public String helloController(@RequestBody Customer customer){

return "hello world , name = " + customer.getName() + " , age = "+ customer.getAge() ;

}

### 9.3.8 实体类

**1）Lombok注解**

|  |  |  |
| --- | --- | --- |
| **注解** | **位置** | **说明** |
| **@NoArgsConstructor** | 类 | 增加无参构造函数 |
| **@AllArgsConstructor** | 类 | 增加全参构造函数 |
| **@Data** | 类 | 增加getter setter方法 |

**2）实体类**

package com.atguigu.springboot.demo.bean;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Customer {

private Integer id ;

private String name ;

private Integer age ;

}

### 9.3.9 服务层

9.3.9.1 职责

（1）接收控制层调用

（2）接收其他服务层的调用/调用其他服务层

（3）处理业务

（4）调用数据层

（5）返回结果

9.3.9.2 相关注解

|  |  |  |
| --- | --- | --- |
| **注解** | **位置** | **说明** |
| **@Service** | 类 | 标注为服务层组件 |
| **@Autowired** | 类 | 自动装配 |

9.3.9.3 HelloWorld

**1）Service层由两个类组成，一个接口和一个实现类组成。**

（1）服务层接口

package com.atguigu.springboot.demo.service;

/\*\*

\* 服务层接口

\*/

public interface CustomerService {

}

（2）服务层实现类

package com.atguigu.springboot.demo.service;

/\*\*

\* 服务层实现类

\*/

public class CustomerServiceImpl implements CustomerService{

}

**2）在CustomerService中增加方法， 例如通过id查询Customer**

public interface CustomerService {

public Customer getCustomerById(Integer id ) ;

}

**3）在CunstomerServiceImpl中实现方法**

@Service

public class CustomerServiceImpl implements CustomerService{

@Override

public Customer getCustomerById(Integer id) {

return new Customer(id,"zhangsan",22);

}

}

**4）在Controller层调用服务层**

@RestController

public class CustomerController {

@Autowired

private CustomerService customerService ;

@GetMapping("/get/{id}")

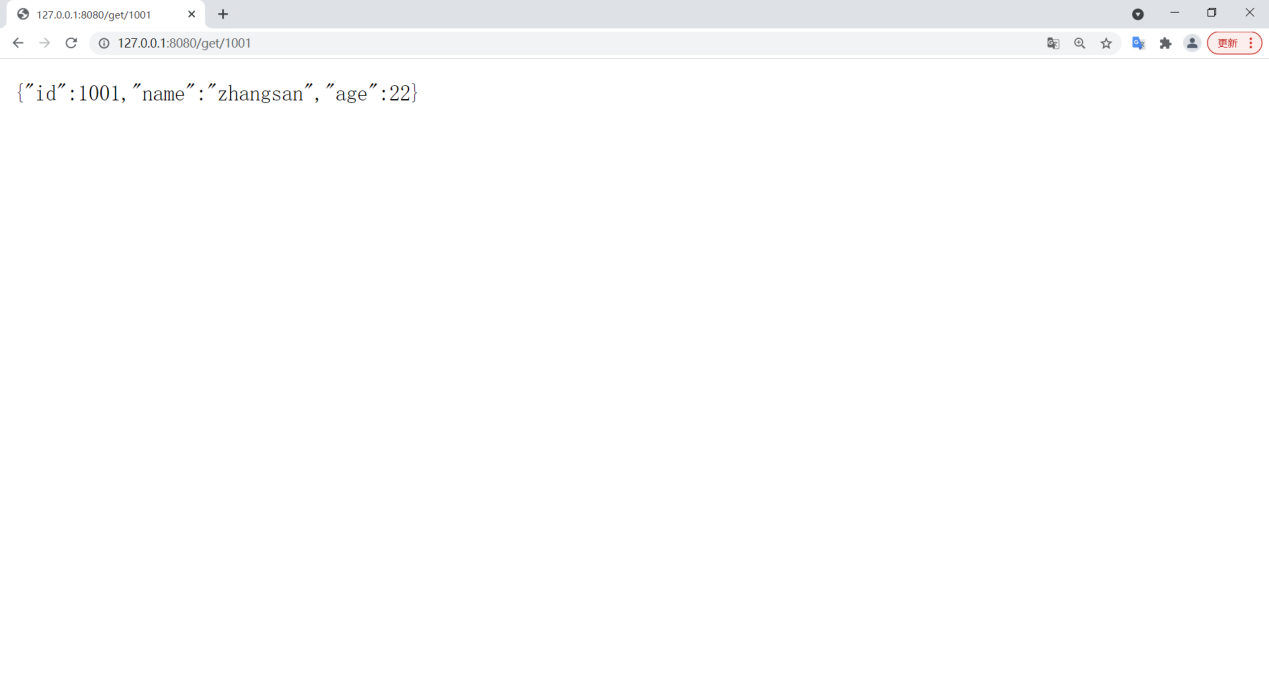
public Customer getCustomer(@PathVariable("id") Integer id ){

return customerService.getCustomerById(id) ;

}

}

**5）在浏览器地址栏中输入 <http://127.0.0.1/get/1001> 进行访问**



### 9.3.10 数据层

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。

9.3.10.1 职责

（1）接收服务层调用

（2）处理jdbc数据的操作

（3）返回结果

9.3.10.2 相关注解

|  |  |  |
| --- | --- | --- |
| **注解** | **位置** | **说明** |
| **@Mapper** | 类 | 标识Mapper接口 |
| **@Select** | 方法 | 查询语句 |
| **@Update** | **方法** | **修改语句** |
| **@Delete** | 方法 | 删除语句 |
| **@Insert** | **方法** | **插入语句** |
| **@Param** | 参数 | 标识SQL参数 |

关于注解中的参数

|  |  |
| --- | --- |
| **注解** | **说明** |
| **#{}** | 会识别参数的类型会自动补全单引  本身字符串值中的特殊符号会被替换、转义 |
| **${}** | 参数原值（一般程序中动态组合sql） |

9.3.10.3 HelloWorld

**1）创建测试表**

CREATE TABLE `customer` (

`id` bigint(20) NOT NULL AUTO\_INCREMENT,

`name` varchar(200) DEFAULT NULL,

`age` bigint(20) DEFAULT NULL,

PRIMARY KEY (`id`)

)

ENGINE=InnoDB DEFAULT CHARSET=utf8

**2）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>

**3）application.properties**

spring.datasource.url=jdbc:mysql://hadoop102:3306/user\_profile\_manager?characterEncoding=utf-8&useSSL=false

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.username=root

spring.datasource.password=123456

**4）创建Mapper接口**

@Mapper

public interface CustomerMapper {

@Select("select \* from customer ")

public List<Customer> getCustomerList();

@Select("select \* from customer where id=#{id}")

public Customer getCustomerById(Long id );

@ Select("select \* from customer")

public List<Customer> selectCustomerList();

@Insert("insert into customer(name,age) values (#{customer.name}, #{customer.age} )")

public void insertCustomer(@Param("customer") Customer customer);

@Update("update customer set age= #{customer.age} where id=#{ customer.id }")

public void updateCustomer (@Param("customer")Customer customer );

@Delete("delete customer\_info where id=#{id}")

public void deleteCustomerById (Long id );

}

**5）Service层中加入调用方法**

@Service

public class CustomerServiceImpl implements CustomerService {

@Autowired

CustomerMapper customerMapper;

@Override

public Customer getCustomerById(Integer id) {

return customerMapper.getCustomerById(id);

}

}

### 9.3.11 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。

9.3.11.1 HelloWorld

**1）引入mybatis依赖**

<!--mybatis-plus-->

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatis-plus-boot-starter</artifactId>

<version>3.4.1</version>

</dependency>

**2）增加Mybatis-plus规范的服务层和数据层**

Mapper接口

import com.baomidou.mybatisplus.core.mapper.BaseMapper;

@Mapper

public interface CustomerMapper extends BaseMapper<Customer> {

}

Service接口

public interface CustomerService extends IService<Customer> {

}

**3）重写CustomerService实现类**

@Service

public class CustomerImpl extends ServiceImpl<CustomerMapper, Customer> implements CustomerService {

}

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

**4）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 name;

int age;

}

**5）Controller直接调用封装方法**

@GetMapping("/customer/list")

public List<Customer> getCustomerList(){

List<Customer> list = customerService.list();

return list ;

}

@GetMapping("/customer")

public Customer getCustomer(@RequestParam("id") String id){

return customerService.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

customerService.saveOrUpdate(customer);

return "success";

}

@DeleteMapping("/customer")

public String deleteCustomer(@RequestParam("id") String id){

customerService.removeById(id);

return "success";

}

### 9.3.12 动态数据源

9.3.12.1 介绍

动态数据源可以实现在同一个项目中支持多种数据源，并能灵活切换不同的数据源。

9.3.12.2 相关注解

|  |  |  |
| --- | --- | --- |
| **注解** | **位置** | **说明** |
| **@DS** | 类或者方法 | @DS 可以注解在方法上或类上，同时存在就近原则 方法上注解 优先于 类上注解。 |

9.3.12.3 HelloWorld

**1）添加依赖**

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>dynamic-datasource-spring-boot-starter</artifactId>

<version>3.3.2</version>

</dependency>

**2）定义多个数据源**

spring.datasource.dynamic.datasource.mysql0111.url=jdbc:mysql://hadoop102:3306/user\_profile\_manager\_0111?characterEncoding=utf-8&useSSL=false

spring.datasource.dynamic.datasource.mysql0111.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.dynamic.datasource.mysql0111.username=root

spring.datasource.dynamic.datasource.mysql0111.password=123456

spring.datasource.dynamic.datasource.mysql0222.url=jdbc:mysql://hadoop102:3306/user\_profile\_manager\_0222?characterEncoding=utf-8&useSSL=false

spring.datasource.dynamic.datasource.mysql0222.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.dynamic.datasource.mysql0222.username=root

spring.datasource.dynamic.datasource.mysql0222.password=123456

**3）在service层和 mapper的类上增加默认数据源**

@Service

@DS("mysql0111")

public class CustomerServiceImpl extends ServiceImpl<CustomerMapper, Customer> implements CustomerService {

@Mapper

@DS("mysql0111")

public interface CustomerMapper extends BaseMapper<Customer> {

**4）在特定的方法上增加特定数据源**

@Insert("insert into customer(name,age) values (#{customer.name}, #{customer.age} )")

@DS("mysql0222")

public void insertCustomer0111(@Param("customer") Customer customer);

## 9.4 分群模块初始搭建

### 9.4.1 相关依赖配置

**1）pom.xml**

**2）application.properties**

### 9.4.2 应用开发的分层

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **控制层** | UserGroupController | 增加方法  1添加分群  2预估分群人数  3更新分群数据 |
| **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。 |

### 9.4.3 clickhouse建表

CREATE TABLE user\_group(

`user\_group\_id` String,

`us` AggregateFunction(groupBitmap, UInt64),

dt String

)

ENGINE = AggregatingMergeTree

PARTITION BY user\_group\_id

ORDER BY user\_group\_id

### 9.4.4 实体Bean-UserGroup

## 9.5 功能一之生成分群基本信息

### 9.5.1 controller层

9.5.1.1 需要添加的功能

完成保存分群信息请求的处理

9.5.1.1 开发

@RestController

public class UserGroupController {

@Autowired

UserGroupService userGroupService;

//添加分群

@PostMapping("/user-group")

@CrossOrigin

public String saveUserGroup(@RequestBody UserGroup userGroup){

/1. 保存分群信息

userGroupService.saveUserGroupInfo(userGroup);

//2. 计算出改分群的用户群体(分群包) , uid的集合, 由clickhouse负责，通过bitmap间的计算完成

userGroupService.genUserGroupUids( userGroup);

return "success";

}

}

### 9.5.2 service层

9.5.2.1 需要添加的功能

处理分群基本信息，然后进行保存

9.5.2.2 开发

**1）在UserGroupService中增加保存分群信息的方法**

//保存基本信息

public void saveUserGroup(UserGroup userGroup);

**2）在UserGroupServiceImpl中实现保存分群信息的方法**

@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){

//补充人群基本信息

String conditionJsonStr = JSON.toJSONString(userGroup.getTagConditions());

userGroup.setConditionJsonStr(conditionJsonStr);

String conditionComment = userGroup.conditionJsonToComment();

userGroup.setConditionComment(conditionComment);

userGroup.setCreateTime(new Date());

this.saveOrUpdate(userGroup);

}

}

9.5.2.3 说明

**UserGroupServiceImpl**实现类的父类是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。

### 9.5.3 mapper层

9.5.3.1 需要添加的功能

完成对分群信息的保存工作

9.5.3.2 开发

@DS("mysql")

public interface UserGroupMapper extends BaseMapper<UserGroup> {

}

9.5.3.3 说明

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。

## 9.6 功能二之生成人群包

生成分群的人群id集合，写入clickhouse及redis

### 9.6.1 controller层

9.6.1.1 需要添加的功能

生成分群信息的同时要生成对应的人群包，因此将生成人群包的代码直接写到saveUserGroup方法即可。

9.6.1.2 开发

@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";

}

}

### 9.6.2 service层

9.6.2.1 需要添加的功能

在clickhouse中生成人群包

9.6.2.2 Clickhouse中建表

CREATE TABLE user\_group

(

`user\_group\_id` String,

`us` AggregateFunction(groupBitmap, UInt64),

`dt` String

)

ENGINE = AggregatingMergeTree

PARTITION BY user\_group\_id

ORDER BY user\_group\_id

9.6.2.3 开发

**1）在UserGroupService中增加生成人群包的方法**

/\*\*

\* 生成人群包

\* @param userGroup

\*/

public void genUserGroupUids(UserGroup userGroup) ;

**2）在UserGroupServiceImpl中实现生成人群包的方法**

// 生成人群包

public void genUserGroupUids(UserGroup userGroup ){

// 单条子查询 => bitmapAnd组合多条子查询 => select = insert

String insertBitmapSql = getInsertBitmapSql(userGroup);

System.out.println(insertBitmapSql);

this.baseMapper.insertBitmap(insertBitmapSql);

}

public String getInsertBitmapSql(UserGroup userGroup){

String selectSql = getSelectBitmapSql(userGroup.getId().toString(), userGroup.getTagConditions(), userGroup.getBusiDate());

String insertSql = " insert into user\_group " + selectSql ;

return insertSql ;

}

private String getSelectBitmapSql(String userGroupId , List<TagCondition> tagConditionList , String businessDate ){

String bitmapSql = getBitmapSql(tagConditionList , businessDate);

String selectSql = "" ;

if(tagConditionList.size() == 1 ) {

selectSql = " select '" + userGroupId + "', bts.us , '" + businessDate + "' from (" + bitmapSql + ") bts" ;

}else{

selectSql = " select '" + userGroupId + "', " + bitmapSql + ",'" + businessDate+ "'" ;

}

System.out.println("selectSQL : " + selectSql);

return selectSql ;

}

// bitmapAnd( (subquery1) , (subquery2))

// 通过bitmapAnd函数， 拼接多个子查询

private String getBitmapSql( List<TagCondition> tagConditionList , String businessDate ){

Map<String, TagInfo> tagInfoMap = tagInfoService.getTagInfoMapWithCode();

StringBuilder sbs = new StringBuilder();

for (int i = 0; i < tagConditionList.size(); i++) {

TagCondition tagCondition = tagConditionList.get(i);

String subQuerySql = getSubQuerySql(tagCondition, businessDate, tagInfoMap);

if(i == 0 ){

sbs.append(subQuerySql);

}else{

sbs.insert(0 , "bitmapAnd( (" ).append("),(").append(subQuerySql).append("))");

}

}

System.out.println("bitmapSQL : " + sbs.toString());

return sbs.toString();

}

// 一个条件生成一个子查询

// select groupBitmapMergeState(us) from user\_tag\_value\_string where

// tag\_code = 'tg\_person\_base\_gender' and tag\_value = '男' and dt = '2020-06-14')

// user\_tag\_value\_string => 根据标签查询类型 => 根据标签类型确定不同的表

// tg\_person\_base\_gender => tag\_code

// tag\_value => 符号 => 翻译 => 把operator翻译成sql符号

// '男' ==> tag\_values => 要不要加括号、 单引， 如果是in 或者notin , 加括号, 如果是数字就不加单引 ，字符串加单引

// 日期

private String getSubQuerySql(TagCondition tagCondition , String businessDate ,Map<String,TagInfo> tagInfoMap ){

//1. 确定表

String tagCode = tagCondition.getTagCode();

TagInfo tagInfo = tagInfoMap.get(tagCode);

String tagValueType = tagInfo.getTagValueType() ;

//根据标签值类型选择表

String tableName = null ;

boolean singleQuotationMarks = false ;

if(ConstCodes.TAG\_VALUE\_TYPE\_LONG.equals(tagValueType)){

tableName = "user\_tag\_value\_long";

}else if(ConstCodes.TAG\_VALUE\_TYPE\_DECIMAL.equals(tagValueType)){

tableName = "user\_tag\_value\_decimal";

}else if(ConstCodes.TAG\_VALUE\_TYPE\_STRING.equals(tagValueType)){

tableName = "user\_tag\_value\_string";

singleQuotationMarks = true ;

}else if(ConstCodes.TAG\_VALUE\_TYPE\_DATE.equals(tagValueType)){

tableName = "user\_tag\_value\_date";

singleQuotationMarks = true ;

}

//2. 表中查询的tag\_code

String tableTagCode = tagCode.toLowerCase();

//3. 操作转换

String operator = getConditionOperator(tagCondition.getOperator());

//4. 处理tag\_value

String tagValueSQL = "" ;

List<String> tagValues = tagCondition.getTagValues();

//如果需要加单引

if(singleQuotationMarks){

tagValueSQL = "'" + StringUtils.join(tagValues, "','") + "'";

}else{

//不需要加单引号

tagValueSQL = StringUtils.join(tagValues, ",");

}

//是否加括号

if(tagCondition.getOperator().equals("in") || tagCondition.getOperator().equals("nin")){

tagValueSQL = "(" + tagValueSQL + ")" ;

}

String sql = " select groupBitmapMergeState(us) as us from " + tableName +

" where tag\_code = '" + tableTagCode + "'" +

" and tag\_value " + operator + " " + tagValueSQL +

" and dt = '" + businessDate + "'" ;

System.out.println("subQuerySQL : " + sql);

return sql ;

}

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("操作符号不正确!");

}

### 9.6.3 mapper层

9.6.3.1 需要添加的功能

在clickhouse中生成人群包

9.6.3.2 开发

**1）在UserGroupMapper中添加生成人群包方法**

@DS("mysql")

public interface UserGroupMapper extends BaseMapper<UserGroup> {

//写入数据的insert语句

@Insert("${insertSql}")

@DS("clickhouse")

public void insertBitmap(String insertSql);

}

## 9.7 功能三之将人群包写入Redis

### 9.7.1 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

### 9.7.2 工具类

工具类中增加RedisUtil的连接工具。

**1） 在application.properties中添加redis的配置**

**2） Redis工具类**

@Configuration

public class RedisUtil {

public static void main(String[] args) {

}

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 getJedis() {

if(jedisPool==null){

JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();

jedisPoolConfig.setMaxTotal(200); // 最大连接数

jedisPoolConfig.setMaxIdle(30);// 最多维持30

jedisPoolConfig.setMinIdle(10);// 至少维持10

jedisPoolConfig.setBlockWhenExhausted(true);

jedisPoolConfig.setMaxWaitMillis(5000);

jedisPoolConfig.setTestOnBorrow(true); //借走连接时测试

jedisPool = new JedisPool(jedisPoolConfig,redisHost,redisPort,60000);

}

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。

### 9.7.3 controller层

9.7.3.1 需要添加的功能

调用service层的保存分群信息到redis的方法

9.7.3.2 开发

@PostMapping("/user-group")

public String saveUserGroup(@RequestBody UserGroup userGroup){

//1. 保存分群信息

userGroupService.saveUserGroupInfo(userGroup);

//2. 计算出改分群的用户群体(分群包) , uid的集合, 由clickhouse负责，通过bitmap间的计算完成

userGroupService.genUserGroupUids( userGroup);

//3. 把计算的结果保存在可以处理高QPS的数据容器中， redis 、 mysql等。

userGroupService.saveUserGroupIdsToRedis(userGroup);

return "success";

}

### 9.7.4 service层

9.7.4.1 需要添加的功能

添加将人群包信息保存到Redis的方法

9.7.4.2 开发

**1）在UserGroupService中增加保存人群包信息到redis的方法**

public interface UserGroupService extends IService<UserGroup> {

//转储到redis中

public void saveUserGroupIdsToRedis(UserGroup userGroup);

}

**2）在UserGroupServiceImpl中实现保存人群包信息到redis的方法**

//转储人群包到redis中

@Override

public void saveUserGroupIdsToRedis(UserGroup userGroup) {

//redis:

//type: set

//key : user\_group

//value:

//写入API: sadd

//读取API: smembers sismembers

//过期时间: 不过期

//1. 根据主键查询clickhouse的人群包结果

Long userGroupId = userGroup.getId();

List<String> uidList = this.baseMapper.selectBitmapArrayById(userGroupId.toString());

//System.out.println("uidList : " + uidList);

//2. 写入 Redis

Jedis jedis = RedisUtil.getJedis();

String userGroupKey = "user:group:" + userGroupId ;

jedis.sadd(userGroupKey , uidList.toArray(new String[]{}));

jedis.close();

}

### 9.7.5 mapper

9.7.5.1 需要添加的功能

添加从clickhouse中查询人群包信息的方法

9.7.5.2 开发

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> {

@Select("select arrayJoin(bitmapToArray(us)) from user\_group where user\_group\_id = #{userGroupId}")

@DS("clickhouse")

public List<String> selectBitmapArrayById(String userGroupId);

}

## 9.8 功能四之预估人群数

根据分群条件查询clickhouse，返回人群数。

### 9.8.1 controller层

9.8.1.1 需要添加的功能

（1）添加处理预估人群数请求的方法

（2）补充创建分群对应的人群数

9.8.1.2 开发

@PostMapping("/user-group")

public String saveUserGroup(@RequestBody UserGroup userGroup){

//1. 保存分群信息

userGroupService.saveUserGroupInfo(userGroup);

//2. 计算出改分群的用户群体(分群包) , uid的集合, 由clickhouse负责，通过bitmap间的计算完成

userGroupService.genUserGroupUids( userGroup);

//3. 把计算的结果保存在可以处理高QPS的数据容器中， redis 、 mysql等。

userGroupService.saveUserGroupIdsToRedis(userGroup);

//更新人数

Long evaluateNum = userGroupService.evaluateNum(userGroup);

userGroup.setUserGroupNum(evaluateNum);

userGroupService.saveOrUpdate(userGroup);

return "success";

}

//预估人数

@PostMapping("/user-group-evaluate")

public Long userGroupEualuate(@RequestBody UserGroup userGroup){

return userGroupService.evaluateNum(userGroup) ;

}

### 9.8.2 service层

9.8.2.1 需要添加的功能

增加预估人数方法

9.8.2.2 开发

**1）在UserGroupService中增加预估人群数的方法**

//预估人数

public Long evaluateNum(UserGroup userGroup);

**2）在UserGroupServiceImpl中实现预估人群数的方法**

//预估人数

//1. 利用原有的bitmapsql

//2. 组合bitmapCardinality

//3. 执行

public Long evaluateNum(UserGroup userGroup){

String bitmapSql = getBitmapSql(userGroup.getTagConditions() , userGroup.getBusiDate());

String selectSql = "" ;

if(userGroup.getTagConditions().size() == 1 ) {

selectSql = " select bitmapCardinality(bts.us) from (" + bitmapSql + ") bts" ;

}else{

selectSql = " select bitmapCardinality(" + bitmapSql + ")" ;

}

System.out.println("selectSQL : " + selectSql);

return this.baseMapper.selectCardinality(selectSql) ;

}

### 9.8.3 mapper层

9.8.3.1 需要添加的功能

从clickhouse中查询分群人数

9.8.3.2 开发

@DS("mysql")

public interface UserGroupMapper extends BaseMapper<UserGroup> {

@Select("${bitmapCardiSQL}")

@DS("clickhouse")

public Long selectCardinality( String bitmapCardiSQL) ;

}

## 9.9 功能五之手动更新分群

### 9.9.1 controller层

9.9.1.1 需要添加的功能

添加处理更新分群请求的方法

9.9.1.2 开发

// 更新分群

@PostMapping("/user-group-refresh/{userGroupId}")

public String userGroupRefresh(@PathVariable("userGroupId") String userGroupId ,@RequestParam("busiDate") String busiDate ){

//1. 删除原有的人群包 (ck ,redis)

userGroupService.clearUserGroupUidCk(userGroupId);

userGroupService.clearUserGroupUidRedis(userGroupId);

//2. 重新生成人群包

UserGroup userGroup = userGroupService.getById(userGroupId);

//补充userGroup信息

userGroup.setTagConditions( JSON.parseArray(userGroup.getConditionJsonStr() , TagCondition.class));

userGroup.setBusiDate(busiDate);

userGroupService.genUserGroupUids(userGroup);

userGroupService.saveUserGroupIdsToRedis(userGroup);

//3. 修改基本信息

Long evaluateNum = userGroupService.evaluateNum(userGroup);

userGroup.setUserGroupNum(evaluateNum);

userGroup.setUpdateTime(new Date());

userGroupService.saveOrUpdate(userGroup);

return "success";

}

### 9.9.2 service层

9.9.2.1 需要添加的功能

增加清除clickhouse和redis中人群包的方法

9.9.2.2 开发

**1）在UserGroupService中增加清除人群包的方法**

//清除人群包 ck

public void clearUserGroupUidCk(String userGroupId );

//清除人群包 redis

public void clearUserGroupUidRedis(String userGroupId) ;

**2）在UserGroupServiceImpl中实现清除人群包的方法**

public void clearUserGroupUidCk(String userGroupId ){

this.baseMapper.deleteUserGroup(userGroupId);

}

public void clearUserGroupUidRedis(String userGroupId){

Jedis jedis = RedisUtil.getJedis();

String userGroupKey = "user:group:" + userGroupId ;

jedis.del(userGroupKey);

jedis.close();

}

### 9.9.3 mapper层

9.9.3.1 需要添加的功能

清除clickhouse和redis中的人群包

9.9.3.2 开发

@Delete("alter table user\_group delete where user\_group\_id = #{userGroupId}")

@DS("clickhouse")

public void deleteUserGroup(String userGroupId);

### 9.9.4 一个问题

更新人群包的时候，将人群包转储到redis的时候，如何防止从clickhouse中查出旧数据?

9.9.4.1 方案

（1）等待几秒， 负面效果是影响用户体验， 无法确定等待几秒

（2） optimize table xxx final (老版本可以(同步)， 新版本不行(异步)) ， 会影响clickhouse性能

（3）解耦，redis的执行结果不依赖clickhouse的计算结果 ，单独通过sql查询一次人群包，不再使用clickhouse计算的结果。

9.9.4.2 开发

**1）重构UserGroupServiceImpl中将数据转储到redis的方法**

//转储人群包到redis中

@Override

public void saveUserGroupIdsToRedis(UserGroup userGroup) {

//redis:

//type: set

//key : user\_group

//value:

//写入API: sadd

//读取API: smembers sismembers

//过期时间: 不过期

//1. 根据主键查询clickhouse的人群包结果

Long userGroupId = userGroup.getId();

//List<String> uidList = this.baseMapper.selectBitmapArrayById(userGroupId.toString());

//System.out.println("uidList : " + uidList);

//TODO: 更新人群包的时候，如何防止查询出旧数据

// 1. 等待几秒， 负面效果是影响用户体验， 无法确定等待几秒

// 2. optimize table xxx final (老版本可以(同步)， 新版本不行(异步)) ， 会影响clickhouse性能

// 3. 解耦 redis的执行结果不依赖clickhouse的计算结果

String bitmapSql = getBitmapSql(userGroup.getTagConditions() , userGroup.getBusiDate());

String selectSql = "" ;

if(userGroup.getTagConditions().size() == 1 ) {

selectSql = " select arrayJoin( bitmapToArray(bts.us) ) from (" + bitmapSql + ") bts" ;

}else{

selectSql = " select arrayJoin( bitmapToArray(" + bitmapSql + "))" ;

}

System.out.println("selectSQL : " + selectSql);

List<String> uidList = this.baseMapper.selectBitmapArraySQLById(selectSql);

//2. 写入 Redis

Jedis jedis = RedisUtil.getJedis();

String userGroupKey = "user:group:" + userGroupId ;

jedis.sadd(userGroupKey , uidList.toArray(new String[]{}));

jedis.close();

}

**2）在UserGroupMapper中添加新的方法从clickhouse中查询人群包**

@Select("${bitmapArraySQL}")

@DS("clickhouse")

public List<String> selectBitmapArraySQLById(String bitmapArraySQL);