User portrait of big data project

(Ad hoc grouping)

Version : V3.0​

# Chapter 9 Development Task 5: Ad hoc user grouping

## 9.1 Functional Requirements

**1) Click [Create Group]**



**2) Fill in the grouping information**



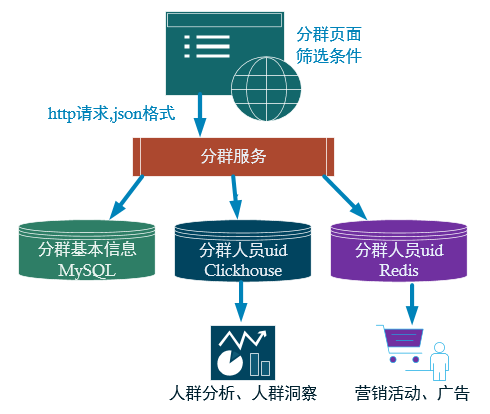
Description of the content that needs to be filled in

|  |  |
| --- | --- |
| **Fields** |  |
| **Group name** | User group information |
| **Business Date (Commissioning)** | The actual production environment should be the day before the current date |
| **Filters** | Filter by three-level tags. |
| **Number of people in the group** | It is used for users to adjust the filter conditions and understand the number of people |
| **Update Type** | Automatic updates are scheduled daily, and manual updates are manually triggered through the interface. |

**3) View the group list**



## 9.2 Process Analysis



The grouping service provides several main functions:

(1) Receive grouping information from the page

(2) Save the basic information of the cluster in MySQL as management data.

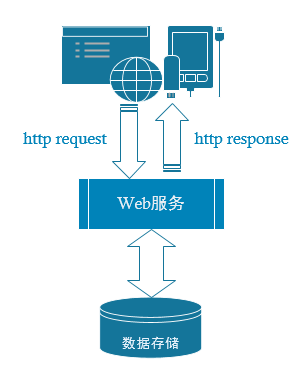
(3) Use Clickhouse to calculate the grouped population results ( Bitmap) and save them for subsequent statistical analysis of the population.

(4) Write the crowd UI into Redis for high-frequency access data in marketing and advertising applications.

## 9.3 Web Service Construction

### 9.3.1 Web Service Overview

Web service (http service ) receives http requests initiated by web pages or App applications , and then processes, calculates, queries, stores, etc. through programs, and then returns the results to the page or App application .



### 9.3.2 Introduction to Springboot

Spring Boot is a new framework provided by the Pivotal team. It is designed to simplify the initial setup and development process of new Spring applications. The framework uses a specific approach to configuration, so that developers no longer need to define boilerplate configurations.



### 9.3.3 Springboot Advantages​

(1) No more need for those cookie-cutter, cumbersome XML files.

(2) Embedded Tomcat, no need for external Tomcat

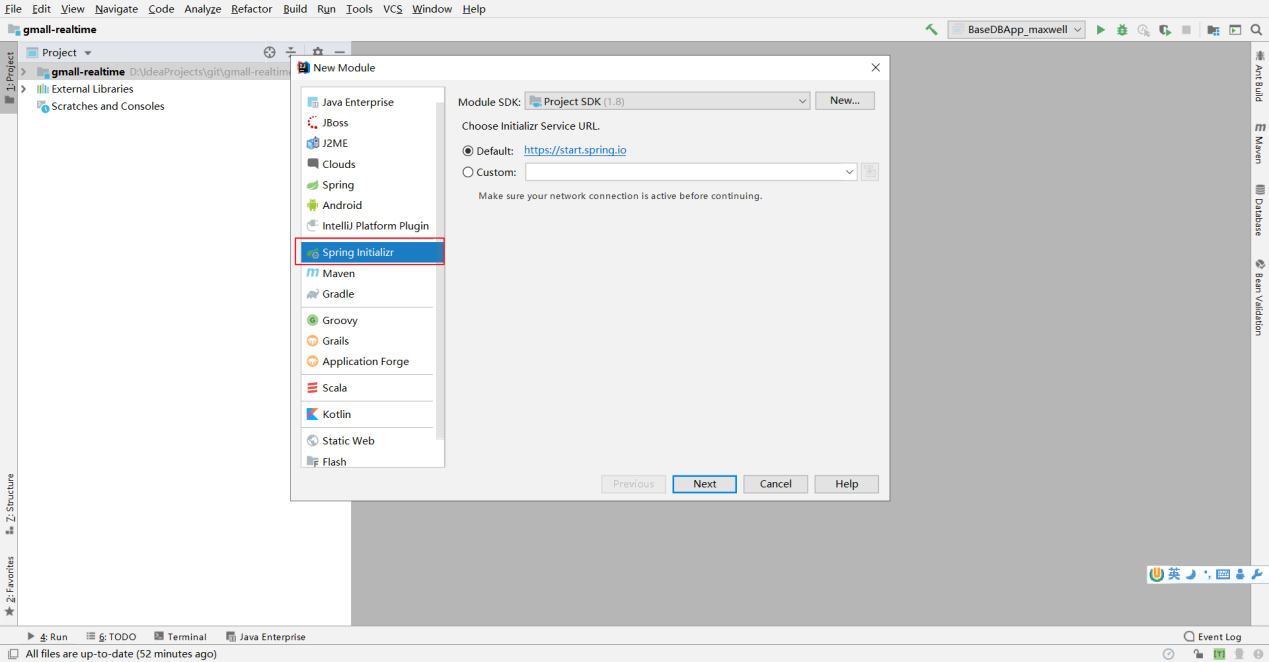
(3) It is more convenient to integrate with various third-party tools ( mysql, redis, elasticsearch, dubbo, kafka , etc.), and only one configuration file needs to be maintained.

### 9.3.4 Relationship between Springboot and SSM

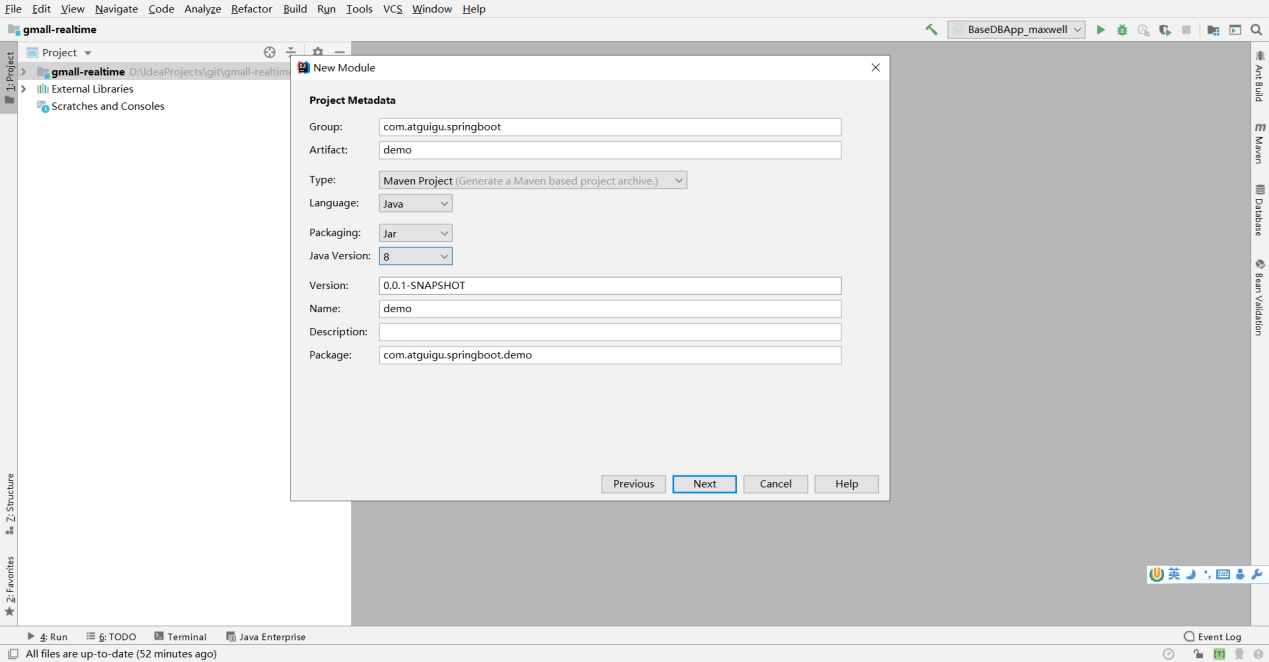
Springboot integrates core functions such as springmvc, spring, mybatis , etc. In other words, the functions are essentially implemented by the original spring and springmvc packages, but springboot has a separate package, so users do not have to directly configure springmvc , spring , etc. in xml .

### Create springboot module

1. **Add a new module and select the springboot initial template**



**2) Fill in the module path**

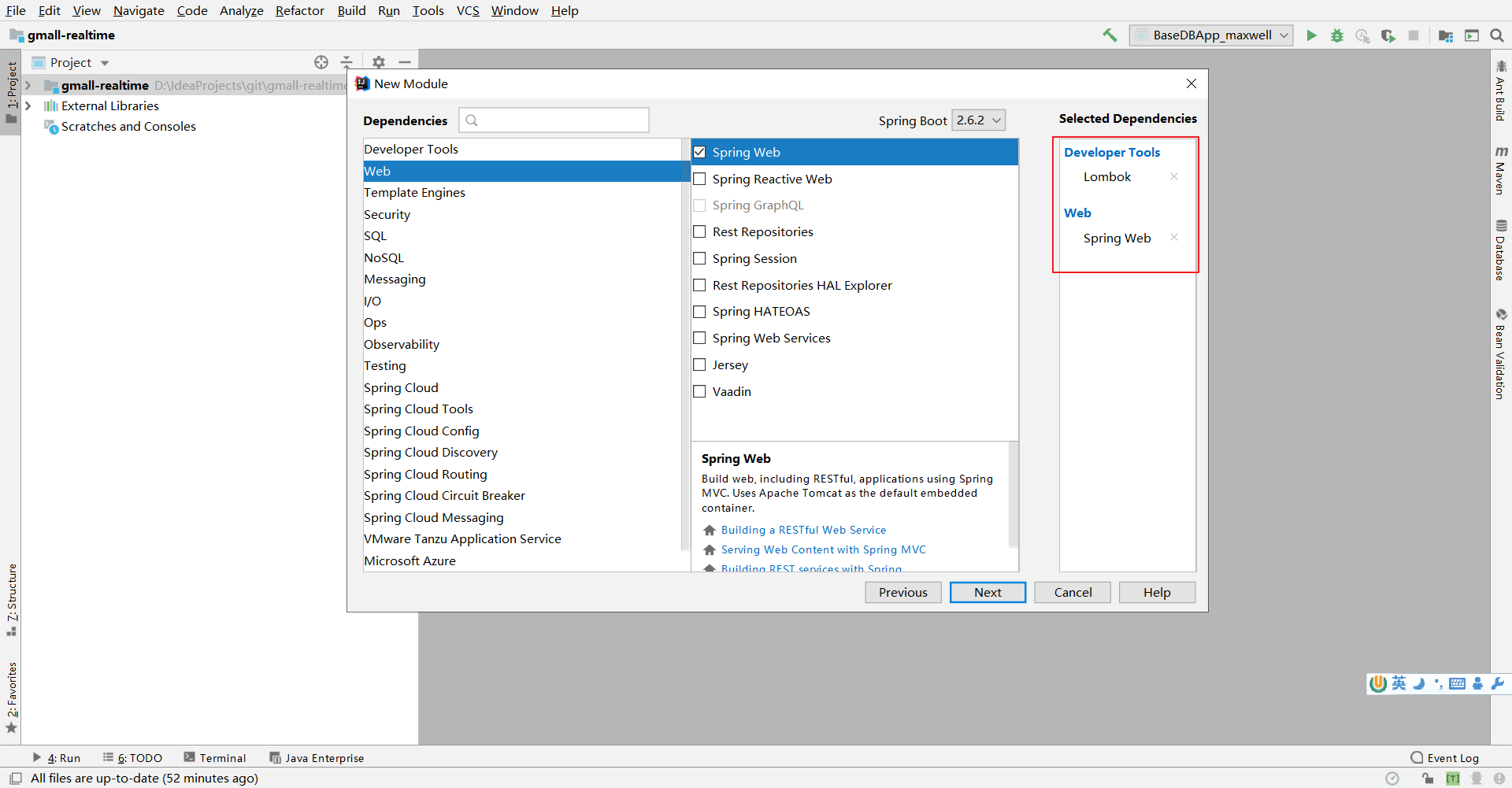


**3) Select Dependency**

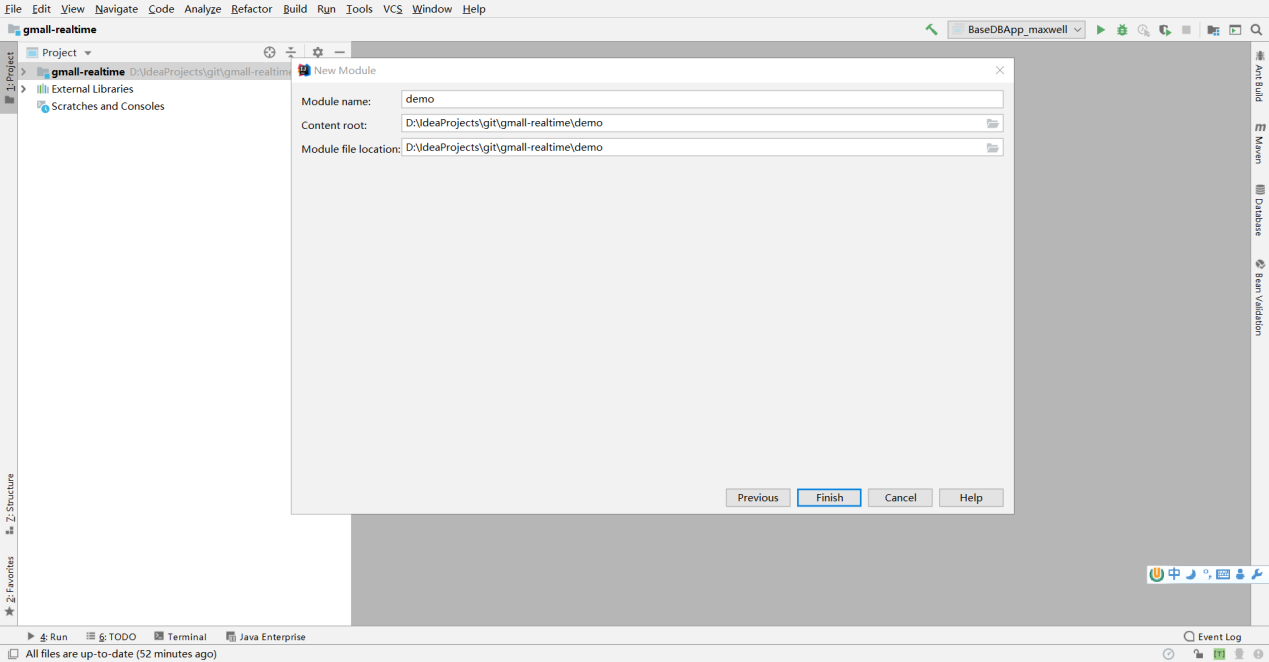
Select spring web to include common annotations for web applications, as well as the web container ( tomcat by default ).

Select Lombok , the annotation plug-in that modifies the entity class.

add dependencies in p om.xml later .



**4) Select the path**

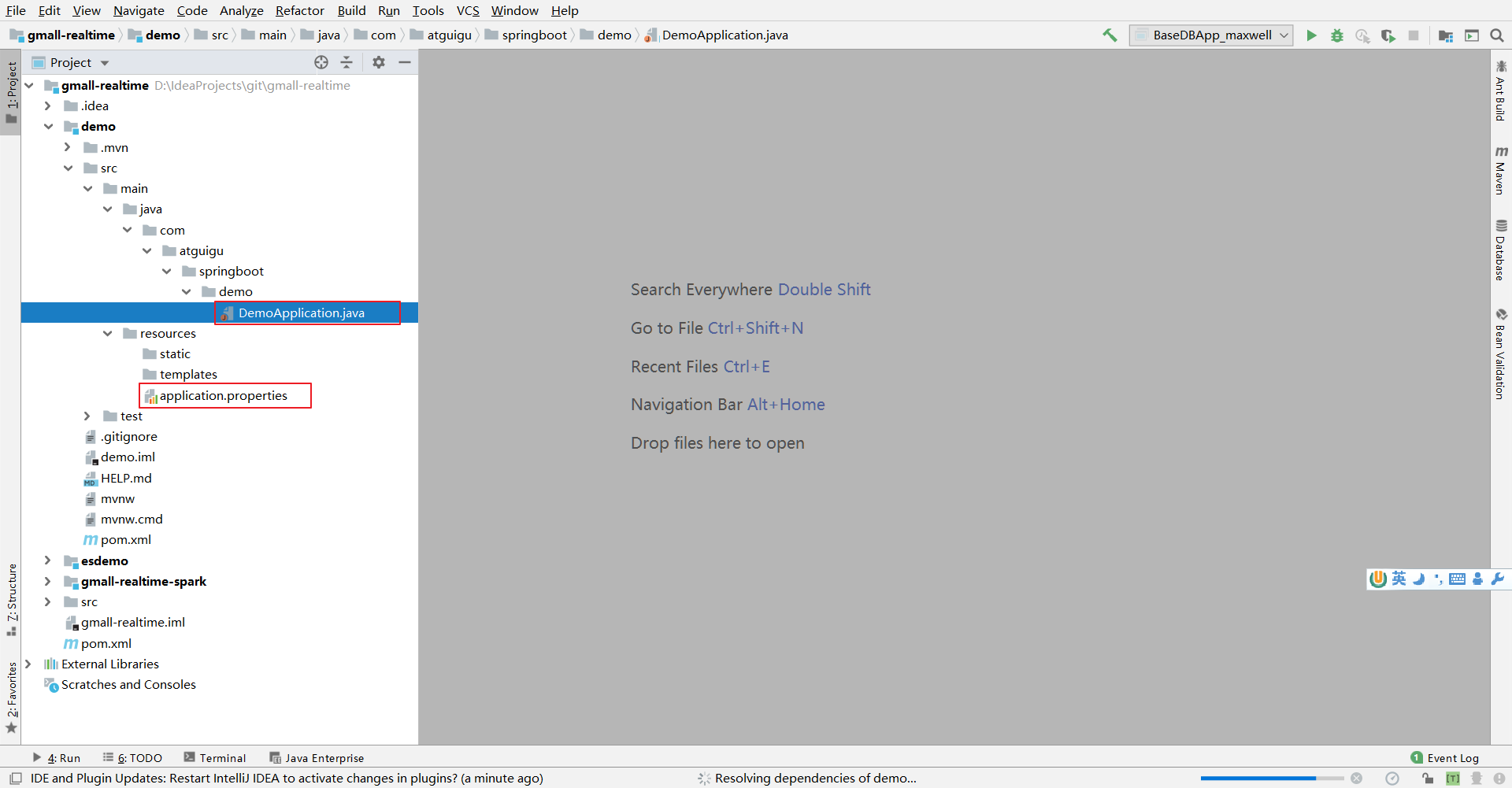


**5) Automatically generated files**

After creation, the springboot project has two more files than the maven project. Among them:

DemoCustomerApplication is the startup class of the entire program.

application.properties is the configuration file for the entire project.



### 9.3.6 Layering of application development

After decades of development, web application development has developed a relatively mature layered structure, which aims to make the program clearer and each layer has its own responsibilities. Any web application development programmer can quickly get started when taking over a new project because everyone is fulfilling this layered agreement.

|  |  |  |  |
| --- | --- | --- | --- |
| **Layering** | **name** | **Responsibilities** | **Notes related to** |
| **Content Roller** | Control Layer | 1 Receive request and parameters  2 Calling the service layer  3 Return response and results | @RestController​  @RequestMapping​  @RequestParam​  @RequestBody​  @PathVariable  @GetMapping  @PostMapping |
| **service​** | Service layer (business layer) | 1Receive control layer call  2. Handle business  3 Calling the data layer  4Return results | @Service  @Autowired​ |
| **mapper​** | Data Layer | 1 Receive service layer call  2Database Operations (SQL)  3 Return results | @ Select  @Insert​ |

### 9.3.7 Controller Layer​

9.3.7.1 Responsibilities​

(1) Receive request and parameters

(2) Calling the service layer

(3) Return response and results

9.3.7.2 Related Notes

|  |  |  |
| --- | --- | --- |
| **annotation** | **Location** |  |
| **@RestController** | kind | Identify the entry class |
| **@RequestMapping** | method | Identify the entry method |
| **@GetMapping** | method | Identifies the entry method for GET requests only |
| **@PostMapping** | method | Identifies the entry method for POST requests only |
| **@RequestParam** | parameter | Receive key-value pair parameters on the path  h ttp://xxxxx/xx?name=xxx&age=xxx |
| **@PathVariable** | parameter | Receive the value on the path  http ://xxxxx/customer/123 |
| **@RequestBody** | parameter | Receive parameters in the request body ( payload) |

9.3.7.3 HelloWorld

**1) Request to connect**

package com.atguigu.springboot.controller;

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

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

/\*\*

\* @RestController identifies the class as a Controller, which serves as the entry class for the request.

\*/

@RestController

public class CustomerController {

/\*\*

\* Request processing method

\* Map requests to methods through @RequestMapping

\*/

@RequestMapping("/hello")

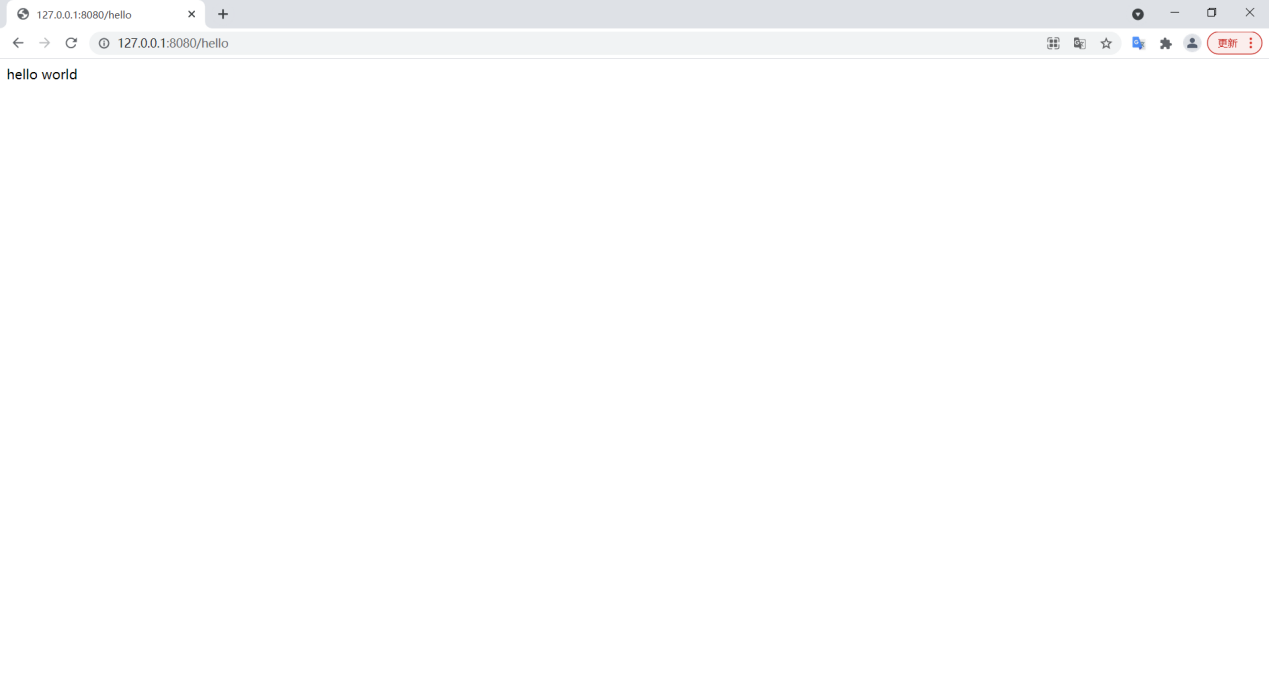
public String helloController(){

return "hello world" ;

}

}

**127.0.0.1:8080/hello in the browser address bar to access**



9.3.7.4 Request Parameters

**1) Key-value pair parameters in the request path**

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

/\*\*

\* Request processing method

\*

\* @RequestParam is used to map the key-value pair parameters in the request path to the formal parameters of the method

\*/

@RequestMapping("/hello")

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

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

}

**2 ) Parameters embedded in the request path**

For example: <http://127.0.0.1:8080/hello/lisi/33>

/\*\*

\* Request processing method

\*

\* @PathVariable is used to map the parameters embedded in the request path to the formal parameters of the method

\*/

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

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

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

}

**3) Parameters in the request body, single parameter mapping**

/\*\*

\* Request processing method

\*

\* method is used to specify the type of request received

\* GET: Generally used to query

\* POST: Generally used to write

\*

\* @GetMapping = @RequestMapping + RequestMethod.GET

\* @PostMapping = @RequestMapping + RequestMethod.POST

\*

\* If the request parameter name and method parameter name are the same, you do not need to specify it through @RequestMapping

\*

\*/

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

@PostMapping("/hello")

public String helloController(String name , Integer age ){

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

}

**4) Parameters are mapped directly to beans**

/\*\*

\* Request processing method

\* @RequestBody is used to map the request parameters in the request body to the corresponding properties in the Bean class

\*/

@PostMapping("/hello")

public String helloController(@RequestBody Customer customer){

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

}

### 9.3.8 Entity Class

**1 ) Lombok Annotations**

|  |  |  |
| --- | --- | --- |
| **annotation** | **Location** | **illustrate** |
| **@NoArgsConstructor** | kind | Add a no-argument constructor |
| **@AllArgsConstructor** | kind | Add full parameter constructor |
| **@Data** | kind | Add getter setter method |

**2) Entity Class**

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 Service Layer

9.3.9.1 Responsibilities​

(1) Receive control layer calls

(2) Receive calls from other service layers/call other service layers

( 3 ) Business processing

(4) Calling the data layer

( 5 ) Return results

9.3.9.2 Related Notes

|  |  |  |
| --- | --- | --- |
| **annotation** | **Location** | **illustrate** |
| **@Service** | kind | Annotated as a service layer component |
| **@A utowired** | kind | Automatic Wiring |

9.3.9.3 HelloWorld​

**1 ) The Service layer consists of two classes, an interface and an implementation class.**

(1) Service layer interface

package com.atguigu.springboot.demo.service;

/\*\*

\* Service layer interface

\*/

public interface CustomerService {

}

(2) Service layer implementation class

package com.atguigu.springboot.demo.service;

/\*\*

\* Service layer implementation class

\*/

public class CustomerServiceImpl implements CustomerService{

}

**2) Add methods to CustomerService, such as querying Customer by id**

public interface CustomerService {

public Customer getCustomerById(Integer id ) ;

}

**3) Implement the method in CunstomerServiceImpl**

@Service

public class CustomerServiceImpl implements CustomerService{

@Override

public Customer getCustomerById(Integer id) {

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

}

}

**4) Call the service layer in the Controller layer**

@RestController

public class CustomerController {

@Autowired

private CustomerService customerService ;

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

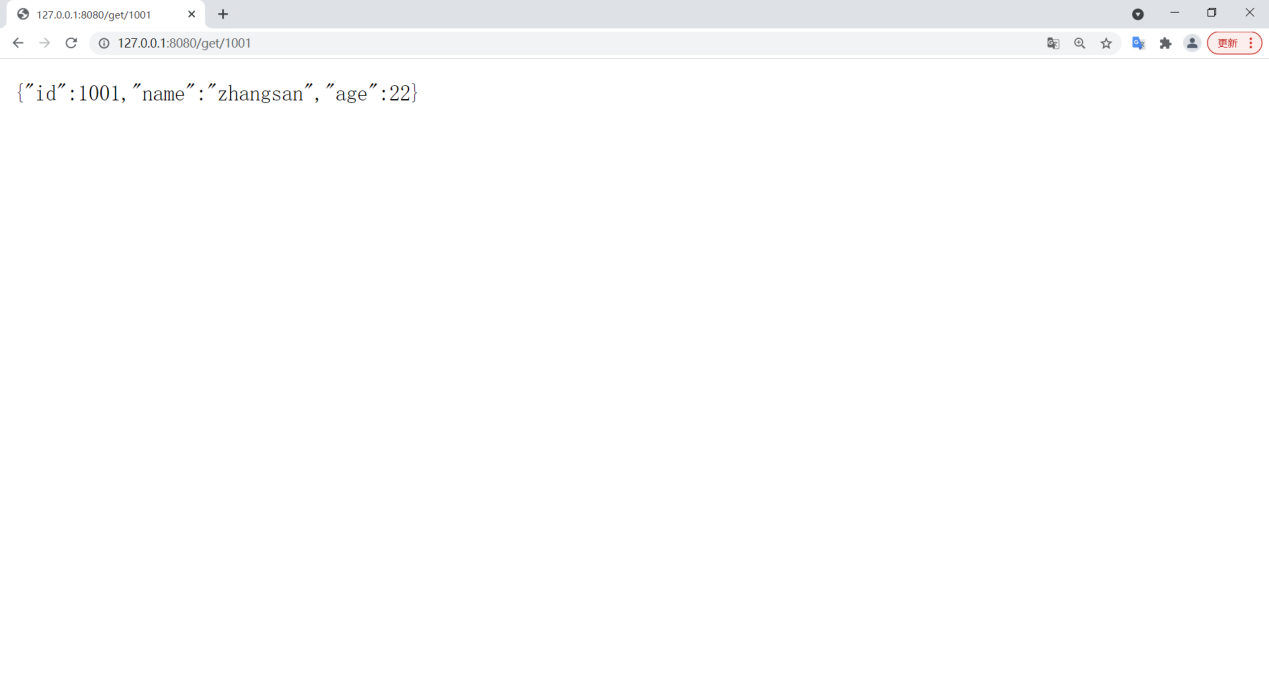
public Customer getCustomer(@PathVariable("id") Integer id ){

return customerService.getCustomerById(id);

}

}

**) Enter http://127.0.0.1/get/1001 in the browser address bar [to](http://127.0.0.1/get/1001) access**



### 9.3.10 Data Layer

MyBatis is an excellent persistence framework that supports custom SQL . MyBatis eliminates almost all JDBC code and the work of setting parameters and obtaining result sets. MyBatis can be configured and mapped through simple XML or annotations.

9.3.10.1 Responsibilities​

(1) Receive service layer calls

(2) Operations for processing jdbc data

(3) Return results

9.3.10.2 Relevant notes

|  |  |  |
| --- | --- | --- |
| **annotation** | **Location** | **illustrate** |
| **@Mapper** | kind | Identify the Mapper interface |
| **@Select** | method | Check for phrases |
| **@Update** | **method** | **Modify statement** |
| **@Delete** | method | Delete Statement |
| **@Insert** | **method** | **Insert Statement** |
| **@Param​** | parameter | Identify SQL parameters |

About parameters in annotations

|  |  |
| --- | --- |
| **annotation** | **illustrate** |
| **#{}** | It will recognize the type of the parameter and automatically complete the single quote  Special symbols in the string value itself will be replaced and escaped |
| **$ { }** | sql in general program ) |

9.3.10.3 HelloWorld​

**1) Create a test table**

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) Introducing dependencies**

<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 ) Create a Mapper interface**

@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 ) Add calling methods to the Service layer**

@Service

public class CustomerServiceImpl implements CustomerService {

@Autowired

CustomerMapper customerMapper;

@Override

public Customer getCustomerById(Integer id) {

return customerMapper . getCustomerById(id);

}

}

### 9.3.11 Myb atis-plus

(MP for short) is a Mybatis enhancement tool. It only enhances MyBatis without making any changes . It is completely SQL - free and encapsulates a large number of CURD operations . Even the CRUD operations are encapsulated in the Service layer, so you can directly call the ready-made CRUD service layer in the controller , which is extremely comfortable and worry-free.

Limitations: Only simple CRUD operations are supported . Multi-table operations (join , union, subquery), GroupBy and various functions are not supported.

9.3.11.1 HelloWorld

**1) Introduce MYBATIS dependency**

<!--mybatis-plus-->

<dependency>

<groupId>com.baomidou</groupId>

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

<version>3.4.1</version>

</dependency>

**2 ) Add the service layer and data layer of the M ybatis - plus specification**

Mapper Interface​

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

@Mapper

public interface CustomerMapper extends BaseMapper<Customer> {

}

Service interface​

public interface CustomerService extends IService<Customer> {

}

**3 ) Rewrite the CustomerService implementation class**

@Service

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

}

As long as these three inheritances are added, the C RUD command can be used directly in the controller .

**4) Add the primary key identifier to the Bean**

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Customer {

@TableId(value = "id",type = IdType.AUTO) //Identifies the primary key generated by the database auto-increment

String id;

String name;

int age;

}

**5 ) Controller directly calls the encapsulation method**

@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)); //Direct return will be automatically converted to json (jackson)

}

@PostMapping("/customer")

public String saveCustomer(@RequestBody Customer customer){

//Whether to insert or modify will be determined based on whether the primary key is empty

customerService.saveOrUpdate(customer);

return "success";

}

@DeleteMapping("/customer")

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

customerService.removeById(id);

return "success";

}

### 9.3.12 Dynamic Data Source

9.3.12.1 Introduction​​​

Dynamic data sources can support multiple data sources in the same project and can flexibly switch between different data sources.

9.3.12.2 Related Notes​​

|  |  |  |
| --- | --- | --- |
| **annotation** | **Location** | **illustrate** |
| **@DS** | Class or method | @DS can be annotated on a method or a class. At the same time, according to the proximity principle, annotations on methods take precedence over annotations on classes. |

9.3.12.3 HelloWorld​​​​

**1 ) Add dependencies**

<dependency>

<groupId>com.baomidou</groupId>

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

<version>3.3.2</version>

</dependency>

**2 ) Define multiple data sources**

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) Add default data sources to the service layer and mapper classes**

@Service

@DS("mysql0111")

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

@Mapper

@DS("mysql0111")

public interface CustomerMapper extends BaseMapper<Customer> {

**4 ) Add specific data sources to specific methods**

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

@DS("mysql0222")

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

## 9.4 Initial construction of the clustering module

### 9.4.1 Related dependency configuration​

**1 ) POm.xml**

**2 ) application.properties**

### 9.4.2 Layering of application development

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Control Layer** | UserGroupController | Add method  1Add a group  2Estimated number of people in each group  3. Update clustering data |
| **Service Layer** | UserGroupService | Add method declaration   1. Save basic information of grouping 2. Generate a set of lickhouse grouped users 3. Generate redis group user collection 4. Returns the estimated number of people according to the query conditions 5. Update group user collection |
| UserGroup ServiceImpl | 1. Save basic information of the group (provided by MySQL Plus ) 2. Generate a set of lickhouse grouped users 3. Generate redis group user collection 4. Returns the estimated number of people according to the query conditions   5 Update group user collection |
| **Data Layer** | **UserGroup Mapper** | 1. of UserGroup in MySQL ( provided by MySQL ) 2. Insert the grouped user set into Clickhouse 3. Update the grouped user collection to Clickhouse   4 Query Clickhouse according to the filter conditions to get the number of people in each group. |

### 9.4.3 c lickhouse table creation

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 Entity Bean -UserGroup​

## 9.5 Function 1 : Generate basic information of grouping

### 9.5.1 Controller Layer​

9.5.1.1 Features to be added

Complete the processing of the request to save the group information

9.5.1.1 Development​​

@RestController

public class UserGroupController {

@Autowired

UserGroupService userGroupService;

//Add group

@PostMapping("/user-group")

@CrossOrigin

public String saveUserGroup(@RequestBody UserGroup userGroup){

/1. Save grouping information

userGroupService.saveUserGroupInfo(userGroup);

//2. Calculate the user group (grouping package) and the set of uids for the group change, which is done by clickhouse through calculations between bitmaps

userGroupService.genUserGroupUids( userGroup);

return "success";

}

}

### 9.5.2 Service layer​​​

9.5.2.1 Features to be added

Process the basic information of the group and then save it

9.5.2.2 Development​​

**1 ) Add a method to save group information in User GroupService**

//Save basic information

public void saveUserGroup(UserGroup userGroup);

**2 ) Implement the method of saving group information in UserGroupServiceImp l**

@Service

@Slf4j

public class UserGroupServiceImpl extends ServiceImpl<UserGroupMapper, UserGroup> implements UserGroupService {

@Autowired

UserGroupMapper userGroupMapper;

/\*\*

\* Save basic information

\* @param userGroup

\*/

public void saveUserGroup(UserGroup userGroup){

//Supplement basic information of the crowd

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 Description​​

The parent class of the UserGroupServiceImpl implementation class is **ServiceImpl < UserGroupMapper** , UserGroup>.

This is a class provided by the dependency of MySQL - plus , which is dedicated to providing general methods for basic insertion, deletion, modification and query of the encapsulated service layer.

Among them, saveOrUpdate() is responsible for both adding and modifying. Whether to add or modify depends on whether the primary key (@TableId(value = "id")) identified in the entity Bean is empty .

### 9.5.3 Mapper Layer​

9.5.3.1 Functions to be added

Complete the storage of grouping information

9.5.3.2 Development​

@DS("mysql")

public interface UserGroupMapper extends BaseMapper<UserGroup> {

}

9.5.3.3 Description​

Although there is no method declared here, the data source is actually marked for the parent class BaseMapper<UserGroup> , and the most basic insertion, deletion, modification and query of UserGroup all use the methods of BaseMapper<UserGroup> .

## 9.6 Function 2 : Generate Crowd Pack

Generate a group of people ID sets and write them to lickhouse and redis

### 9.6.1 Controller Layer​

9.6.1.1 Features to be added

When generating grouping information, the corresponding crowd package must be generated, so the code for generating the crowd package can be directly written into the save UserGroup method.

9.6.1.2 Development​​​

@RestController

public class UserGroupController {

@Autowired

UserGroupService userGroupService;

//Add group

@PostMapping("/user-group")

@CrossOrigin

public String saveUserGroup(@RequestBody UserGroup userGroup){

System.out.println(userGroup);

//Save basic information (mysql)

userGroupService.saveUserGroup(userGroup);

//Generate group population package (clickhouse, Redis)

userGroupService.genUserGroupUids(userGroup);

return "success";

}

}

### 9.6.2 Service Layer​

9.6.2.1 Features to be added

Generate crowd package in clickhouse

9.6.2.2 Creating tables in ClickHouse

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 Development​​​

**1 ) Add a method to generate a crowd package in User GroupService**

/\*\*

\* Generate crowd pack

\* @param userGroup

\*/

public void genUserGroupUids(UserGroup userGroup);

**2 ) Implement the method of generating crowd packages in User GroupService Impl**

// Generate crowd package

public void genUserGroupUids(UserGroup userGroup ){

// Single subquery => bitmapAnd combines multiple subqueries => 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))

// Use the bitmapAnd function to concatenate multiple subqueries

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();

}

// A condition generates a subquery

// 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 => Based on tag query type => Determine different tables based on tag type

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

// tag\_value => symbol => translation => translate operator into sql symbol

// '男' ==> tag\_values => whether to add brackets or single quotes, if it is in or notin, add brackets, if it is a number, do not add single quotes, and add single quotes for strings

// date

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

//1. Confirm the table

String tagCode = tagCondition.getTagCode();

TagInfo tagInfo = tagInfoMap.get(tagCode);

String tagValueType = tagInfo.getTagValueType();

//Select table based on tag value type

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. Tag\_code queried in the table

String tableTagCode = tagCode.toLowerCase();

//3. Operation conversion

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

//4. Process tag\_value

String tagValueSQL = "";

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

//If you need to add a single quote

if (singleQuotationMarks) {

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

}else{

// No single quotes needed

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

}

//Whether to add brackets

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("The operation symbol is incorrect!");

}

### 9.6.3 Mapper layer​

9.6.3.1 Features to be added

Generate crowd package in clickhouse

9.6.3.2 Development​​​

**1 ) Add a method to generate a crowd package in UserGroupMapper**

@DS("mysql")

public interface UserGroupMapper extends BaseMapper<UserGroup> {

//insert statement to write data

@Insert("${insertSql}")

@DS("clickhouse")

public void insertBitmap(String insertSql);

}

## 9.7 Function 3 : Write crowd packets into Redis

### 9.7.1 Analysis​

In order to make the upstream business system use the crowd package more conveniently and with better performance, the crowd package of Clickhouse needs to be written to Redis.

There are two main operations:

(1) Read the Bitmap crowd package from clickhouse and convert it into a user ID list .

(2) Write data to Redis

### 9.7.2 Tools​

Util connection tool to the tool category .

**1) Add redis configuration in application.properties**

**2 ) Redis tools**

@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); // Maximum number of connections

jedisPoolConfig.setMaxIdle(30); // Maintain up to 30

jedisPoolConfig.setMinIdle(10); // Maintain at least 10

jedisPoolConfig.setBlockWhenExhausted(true);

jedisPoolConfig.setMaxWaitMillis(5000);

jedisPoolConfig.setTestOnBorrow(true); //Test when borrowing a connection

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

}

return jedisPool.getResource();

}

}

Where @Configuration The function is to use the @ Value tag to extract the redis configuration in the configuration file into a variable.

### 9.7.3 Controller layer

9.7.3.1 Features to be added

Call the service layer to save the grouping information to redis

9.7.3.2 Development​​​

@PostMapping("/user-group")

public String saveUserGroup(@RequestBody UserGroup userGroup){

//1. Save grouping information

userGroupService.saveUserGroupInfo(userGroup);

//2. Calculate the user group (grouping package) and the set of uids for the group change, which is done by clickhouse through calculations between bitmaps

userGroupService.genUserGroupUids( userGroup);

//3. Save the calculation results in a data container that can handle high QPS, such as redis, mysql, etc.

userGroupService.saveUserGroupIdsToRedis(userGroup);

return "success";

}

### 9.7.4 Service Layer

9.7.4.1 Features to be added

Add method to save crowdpack information to Redis

9.7.4.2 Development​​​

**1 ) Add a method to save crowd package information to redis in User GroupService**

public interface UserGroupService extends IService<UserGroup> {

//Dump to redis

public void saveUserGroupIdsToRedis(UserGroup userGroup);

}

**2) Implement the method of saving crowd package information to redis in User GroupService Impl**

//Dump the crowd package to redis

@Override

public void saveUserGroupIdsToRedis(UserGroup userGroup) {

//redis:

//type: set

//key : user\_group

//value:

//Write API: sadd

//Read API: smembers sismembers

//Expiration time: Not expired

//1. Query the clickhouse crowd package results based on the primary key

Long userGroupId = userGroup.getId();

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

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

//2. Write to 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 Features to be added

Added a method to query crowdpack information from clickhouse

9.7.5.2 Development​​​

Reading the crowd package of clickhouse mainly relies on the query method in mapper .

@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 Function 4 : Estimated number of people

Query ClickHouse based on the grouping criteria and return the number of people.

### 9.8.1 Controller Layer​

9.8.1.1 Features to be added

(1) Add a method to handle requests for estimating crowd size

(2) Add the number of people corresponding to the group

9.8.1.2 Development​​​

@PostMapping("/user-group")

public String saveUserGroup(@RequestBody UserGroup userGroup){

//1. Save grouping information

userGroupService.saveUserGroupInfo(userGroup);

//2. Calculate the user group (grouping package) and the set of uids for the group change, which is done by clickhouse through calculations between bitmaps

userGroupService.genUserGroupUids( userGroup);

//3. Save the calculation results in a data container that can handle high QPS, such as redis, mysql, etc.

userGroupService.saveUserGroupIdsToRedis(userGroup);

//Update the number of people

Long evaluateNum = userGroupService.evaluateNum(userGroup);

userGroup.setUserGroupNum(evaluateNum);

userGroupService.saveOrUpdate(userGroup);

return "success";

}

//Estimated number of people

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

public Long userGroupEualuate(@RequestBody UserGroup userGroup){

return userGroupService.evaluateNum(userGroup);

}

### 9.8.2 Service Layer​

9.8.2.1 Features to be added

How to increase the estimated number of people

9.8.2.2 Development​​​

**1 ) How to add the estimated number of people in User GroupService**

//Estimated number of people

public Long evaluateNum(UserGroup userGroup);

**2 ) Implement the method of estimating the number of people in User GroupServiceImpl**

//Estimated number of people

//1. Use the original bitmapsql

//2. Combine bitmapCardinality

//3. Execution

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 Layer

9.8.3.1 Features to be added

Query the number of people in a group from ClickHouse

9.8.3.2 Development​​​

@DS("mysql")

public interface UserGroupMapper extends BaseMapper<UserGroup> {

@Select("${bitmapCardiSQL}")

@DS("clickhouse")

public Long selectCardinality( String bitmapCardiSQL) ;

}

## 9.9Function 5 : Manually update grouping

### 9.9.1 Controller Layer​

9.9.1.1 Features to be added

Add a method to handle update grouping requests

9.9.1.2 Development​​​

// Update grouping

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

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

//1. Delete the original crowd package (ck, redis)

userGroupService.clearUserGroupUidCk(userGroupId);

userGroupService.clearUserGroupUidRedis(userGroupId);

//2. Regenerate the crowd package

UserGroup userGroup = userGroupService.getById(userGroupId);

//Add userGroup information

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

userGroup.setBusiDate(busiDate);

userGroupService.genUserGroupUids(userGroup);

userGroupService.saveUserGroupIdsToRedis(userGroup);

//3. Modify basic information

Long evaluateNum = userGroupService.evaluateNum(userGroup);

userGroup.setUserGroupNum(evaluateNum);

userGroup.setUpdateTime(new Date());

userGroupService.saveOrUpdate(userGroup);

return "success";

}

### 9.9.2 Service Layer​

9.9.2.1 Features to be added

Added method to clear crowd packets in clickhouse and redis

9.9.2.2 Development​​​

**1 ) Add a method to clear the crowd package in User GroupService**

// Clear the crowd package ck

public void clearUserGroupUidCk(String userGroupId );

//Clear crowd package redis

public void clearUserGroupUidRedis(String userGroupId);

**2 ) Implement the method of clearing the crowd package in User GroupService Impl**

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 Layer

9.9.3.1 Features to be added

Clear crowd packages in clickhouse and redis

9.9.3.2 Development​​​

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

@DS("clickhouse")

public void deleteUserGroup(String userGroupId);

### 9.9.4 A question

When updating a crowd package, how can I prevent old data from being found in ClickHouse when dumping the crowd package to Redis?

9.9.4.1 Solution​​​

(1) Waiting for a few seconds. The negative effect is that it affects the user experience. It is impossible to determine how many seconds to wait.

(2) optimize table xxx final (the old version can do it (synchronous), but the new version cannot do it (asynchronous)), which will affect the performance of ClickHouse

(3) Decoupling: the execution results of Redis do not depend on the calculation results of ClickHouse. The crowd package is queried once through SQL alone, and the results calculated by ClickHouse are no longer used.

9.9.4.2 Development​​​

**1 ) Refactor the method in UserGroupServiceImpl that dumps data to redis**

//Dump the crowd package to redis

@Override

public void saveUserGroupIdsToRedis(UserGroup userGroup) {

//redis:

//type: set

//key : user\_group

//value:

//Write API: sadd

//Read API: smembers sismembers

//Expiration time: Not expired

//1. Query the clickhouse crowd package results based on the primary key

Long userGroupId = userGroup.getId();

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

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

//TODO: How to prevent old data from being queried when updating the crowd package

// 1. Wait for a few seconds. The negative effect is that it affects the user experience. It is impossible to determine how many seconds to wait.

// 2. optimize table xxx final (old version can (synchronous), new version cannot (asynchronous)), which will affect clickhouse performance

// 3. Decouple the execution results of redis from relying on the calculation results of clickhouse

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. Write to Redis

Jedis jedis = RedisUtil.getJedis();

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

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

jedis.close();

}

**2 ) Add a new method in UserGroupMapper to query the crowd package from clickhouse**

@Select("${bitmapArraySQL}")

@DS("clickhouse")

public List<String> selectBitmapArraySQLById(String bitmapArraySQL);