1. 消息通知系统
2. 消息通知的业务场景

消息通知微服务的定位是平台内的消息功能，分为全员消息，订阅类消息，点对点消息。

1. 全员消息

系统通知，活动通知等所有用户都会收到的消息。

1. 订阅类消息

关注某一类数据的用户，当该数据更新时就会通知该用户。

1. 点对点消息

某位用户对另外一位用户进行操作后，系统向被操作的用户发送的消息。

1. 搭建消息通知微服务
2. 业务分析

总体的业务功能是用户对文章作者进行订阅，当被订阅的用户发布新的文章时，可以通过消息通知系统发送消息给订阅者。业务流程分析如下：

1. 用户在页面中订阅某篇文章的作者。
2. 文章微服务能够接收到普通用户和文章作者的订阅关系，并保存起两者的订阅关系。
3. 当文章作者发布新的文章时，会调用消息通知系统，首先查询已经订阅的用户的信息，然后向这些用户显示通知。

接口分析：

1. 文章微服务需要添加订阅文章作者的接口，由于一个文章作者在发布文章时会通知所有的订阅用户，订阅关系会被频繁地调用，因此将订阅关系放在Redis中，并且Redis要实现持久化。
2. 消息通知系统需要有创建消息，保存消息，消息通知的接口，将所有消息放在Mysql中。
3. 用户微服务添加查询用户信息的接口
4. 表结构分析

创建tensquare\_notice通知数据库，添加两张数据表，tb\_notice和tb\_notice\_fresh。将资料中的sql脚本导入到数据库中，创建数据库和表。

1. tb\_notice

该数据表用于保存用户的消息通知数据，即每个消息的具体信息记录。具体的字段名如下所示：

1. ID

每个消息有特定的ID。

2）receiverId

该消息最终会有一个接受者，receiverId为接受者的ID。

3） operatorId

该消息由哪个用户发送的，存储的是该用户的ID。

4） action

消息可以由很多操作发布，如给文章评论，点赞等等，都可以发送消息。action字段存储着该消息的操作类型。

5） targetType

消息由某一个操作触发，该操作的对象是哪种类型的，是对一个文章还是对用户进行评论，targetType字段存储的是被操作的对象类型。

6） targetId

被操作对象的ID。

7） createtime

消息点的发表日期。

8） state

消息是否被阅读。

1. tb\_notice\_fresh

该数据表存储的是要发送给接受者的消息，即还没有发送的消息，适用场景为：当用户登录，我们检查是否有需要推送给用户的消息。字段名如下所示：

1. userId

接受者的ID。

1. noticeId

要发送的消息ID。

1. 为何使用mysql而不是Redis存储消息

Redis通过内存存储消息，可能出现消息丢失的情况，同时Redis存储在内存中，当用户不在线时，Redis会在内存中越积越多，造成资源的消耗。

同时由于该表只有在用户登录时才会进行调用，推测并发量并不大，因此可以放在Mysql中。

1. 创建tensquare\_notice子工程
2. 创建tensquare\_notice子工程

继承自tensquare\_parent父工程。

1. 添加依赖

通知子工程需要使用Mysql ，因此需要添加mybatis-plus依赖：

<dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatisplus-spring-boot-starter</artifactId>  
 <version>${mybatisplus-spring-boot-starter.version}</version>  
</dependency>  
<dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatis-plus</artifactId>  
 <version>${mybatisplus.version}</version>  
</dependency>

链接数据库时需要设置连接的资源池，因此添加Druid数据库连接池依赖：

<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid-spring-boot-starter</artifactId>  
 <version>1.1.9</version>  
</dependency>

mysql驱动：

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

tensquare\_common公共模块：

<dependency>  
 <groupId>com.tensquare</groupId>  
 <artifactId>tensquare\_common</artifactId>  
 <version>1.0-SNAPSHOT</version>  
</dependency>

通知有些数据需要保存到redis中，添加data-redis依赖：

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
</dependency>

通知微服务需要注册到注册中心中，因此添加eureka客户端：

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
</dependency>

通知微服务需要与其他微服务进行交互，因此添加微服务之间调用服务feign:

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-openfeign</artifactId>  
</dependency>

1. 添加配置文件application.yml

spring:  
 application:  
 name: tensquare-notice  
 datasource:  
 driver-class-name: com.mysql.jdbc.Driver  
 url: jdbc:mysql://192.168.112.128:3306/tensquare\_notice?characterEncoding=utf-8  
 username: root  
 password: root  
 redis:  
 host: 192.168.112.128  
  
server:  
 port: 9014  
  
*# Mybatis-Plus 配置*mybatis-plus:  
 *# mapper-locations: classpath:/mapper/\*Mapper.xml  
 #实体扫描，多个package用逗号或者分号分隔* typeAliasesPackage: com.tensquare.notice.pojo  
 global-config:  
 id-type: 1 *#0:数据库ID自增 1:用户输入id* db-column-underline: false  
 refresh-mapper: true  
 configuration:  
 map-underscore-to-camel-case: true  
 cache-enabled: true *#配置的缓存的全局开关* lazyLoadingEnabled: true *#延时加载的开关* multipleResultSetsEnabled: true *#开启延时加载，否则按需加载属性* log-impl: org.apache.ibatis.logging.stdout.StdOutImpl *#打印sql语句,调试用*eureka:  
 client:  
 service-url:  
 defaultZone: http://127.0.0.1:6868/eureka/  
 instance:  
 prefer-ip-address: true

1. 创建启动类

@SpringBootApplication  
@EnableEurekaClient  
@EnableFeignClients  
public class NoticeApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(NoticeApplication.class, args);  
 }  
  
 @Bean  
 public IdWorker createIdWorker(){  
 return new IdWorker(1, 1);  
 }  
}

1. 添加POJO类

根据创建的通知数据表，创建对应的POJO类。注意，Notice.java对应的是tb\_notice数据表，添加了一个进行操作的用户昵称字段operatorName，因为该字段可能要对用户进行展示，所以需要一个用户看得懂的字段。还添加了一个字段targetName，表示对象的名称或简介。这些数据表中没有的字段都需要添加注解@TableField(exist=false)，表示数据库中没有这些字段。

package com.tensquare.notice.pojo;  
  
import com.baomidou.mybatisplus.annotations.TableField;  
import com.baomidou.mybatisplus.annotations.TableId;  
import com.baomidou.mybatisplus.annotations.TableName;  
import com.baomidou.mybatisplus.enums.IdType;  
  
import java.io.Serializable;  
import java.util.Date;  
  
@TableName("tb\_notice")  
public class Notice implements Serializable {  
  
 @TableId(type = IdType.*INPUT*)  
 private String id;//ID  
  
 private String receiverId;//接收消息的用户ID  
 private String operatorId;//进行操作的用户ID  
  
 @TableField(exist = false)  
 private String operatorName;//进行操作的用户昵称  
 private String action;//操作类型（评论，点赞等）  
 private String targetType;//对象类型（评论，点赞等）  
  
 @TableField(exist = false)  
 private String targetName;//对象名称或简介  
 private String targetId;//对象id  
 private Date createtime;//创建日期  
 private String type; //消息类型 sys系统消息 user用户消息  
 private String state; //消息状态（0 未读，1 已读）  
  
 public String getId() {  
 return id;  
 }  
  
 public void setId(String id) {  
 this.id = id;  
 }  
  
 public String getReceiverId() {  
 return receiverId;  
 }  
  
 public void setReceiverId(String receiverId) {  
 this.receiverId = receiverId;  
 }  
  
 public String getOperatorId() {  
 return operatorId;  
 }  
  
 public void setOperatorId(String operatorId) {  
 this.operatorId = operatorId;  
 }  
  
 public String getOperatorName() {  
 return operatorName;  
 }  
  
 public void setOperatorName(String operatorName) {  
 this.operatorName = operatorName;  
 }  
  
 public String getAction() {  
 return action;  
 }  
  
 public void setAction(String action) {  
 this.action = action;  
 }  
  
 public String getTargetType() {  
 return targetType;  
 }  
  
 public void setTargetType(String targetType) {  
 this.targetType = targetType;  
 }  
  
 public String getTargetName() {  
 return targetName;  
 }  
  
 public void setTargetName(String targetName) {  
 this.targetName = targetName;  
 }  
  
 public String getTargetId() {  
 return targetId;  
 }  
  
 public void setTargetId(String targetId) {  
 this.targetId = targetId;  
 }  
  
 public Date getCreatetime() {  
 return createtime;  
 }  
  
 public void setCreatetime(Date createtime) {  
 this.createtime = createtime;  
 }  
  
 public String getType() {  
 return type;  
 }  
  
 public void setType(String type) {  
 this.type = type;  
 }  
  
 public String getState() {  
 return state;  
 }  
  
 public void setState(String state) {  
 this.state = state;  
 }  
}

同理，根据数据表tb\_notice-fresh创建对应的POJO类NoticeFresh类：

package com.tensquare.notice.pojo;  
  
import com.baomidou.mybatisplus.annotations.TableName;  
  
@TableName("tb\_notice\_fresh")  
public class NoticeFresh {  
  
 private String userId;  
 private String noticeId;  
  
 public String getUserId() {  
 return userId;  
 }  
  
 public void setUserId(String userId) {  
 this.userId = userId;  
 }  
  
 public String getNoticeId() {  
 return noticeId;  
 }  
  
 public void setNoticeId(String noticeId) {  
 this.noticeId = noticeId;  
 }  
}

（6） 创建DAO层接口

DAO层接口使用Mybatis-plus，继承BaseMapper接口：

一个NoticeFreshDao接口操作NoticeFresh类

package com.tensquare.notice.dao;  
  
import com.baomidou.mybatisplus.mapper.BaseMapper;  
import com.tensquare.notice.pojo.NoticeFresh;  
  
public interface NoticeFreshDao extends BaseMapper<NoticeFresh> {  
}

一个Notice接口操作Notoce类：

package com.tensquare.notice.dao;  
  
import com.baomidou.mybatisplus.mapper.BaseMapper;  
import com.tensquare.notice.pojo.Notice;  
  
public interface NoticeDao extends BaseMapper<Notice> {  
}

（7） 配置mybatis-plus config配置类

mybatis-plus需要配置扫描包，且需要一个分页拦截器，因此需要配置类：

package com.tensquare.notice.config;  
  
import com.baomidou.mybatisplus.plugins.PaginationInterceptor;  
import org.mybatis.spring.annotation.MapperScan;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
@MapperScan("com.tensquare.notice.dao")  
public class MyBatisPlusConfig {  
   
 @Bean  
 public PaginationInterceptor createPaginationInterceptor(){  
 return new PaginationInterceptor();  
 }  
}

（8） 创建Service层

Service层需要注入Dao层接口，调用DAO层的方法：

package com.tensquare.notice.service;  
  
import com.netflix.discovery.converters.Auto;  
import com.tensquare.notice.dao.NoticeDao;  
import com.tensquare.notice.dao.NoticeFreshDao;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
@Service  
public class NoticeService {  
  
 @Autowired  
 private NoticeDao noticeDao;  
  
 @Autowired  
 private NoticeFreshDao noticeFreshDao;  
}

（9） 创建Controller层

注入Service层的类对象：

package com.tensquare.notice.controller;  
  
import com.netflix.discovery.converters.Auto;  
import com.tensquare.notice.service.NoticeService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.CrossOrigin;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
@RequestMapping("notice")  
@CrossOrigin  
public class NoticeController {  
 @Autowired  
 private NoticeService noticeService;  
}

1. 实现基本增删改查功能
2. 根据ID查询消息通知

添加Controller层方法selectById:

//根据ID查询消息通知  
//http://127.0.0.1:9014/notice/{id} GET  
@RequestMapping(value="{id}", method=RequestMethod.*GET*)  
public Result selectById(@PathVariable String id){  
 Notice notice = noticeService.selectById(id);  
 return new Result(true, StatusCode.*OK*, "查询成功", notice);  
}

添加Service层方法selectById

public Notice selectById(String id) {  
 return noticeDao.selectById(id);  
}

1. 根据条件分页查询消息通知

添加Controller层方法selectByList

//根据条件分页查询消息通知  
// http://127.0.0.1:9014/notice/search/{page}/{size} POST  
@RequestMapping(value="search/{page}/{size}", method=RequestMethod.*POST*)  
public Result selectByList(@RequestBody Notice notice,  
 @PathVariable Integer page,  
 @PathVariable Integer size){  
 Page<Notice> pageData = noticeService.selectByPage(notice, page, size);  
 PageResult<Notice> pageResult = new PageResult<>(pageData.getTotal(), pageData.getRecords());  
 return new Result(true, StatusCode.*OK*, "查询成功", pageResult);  
}

添加Service层的selectByList方法：

public Page<Notice> selectByPage(Notice notice, Integer page, Integer size) {  
 //封装分页对象  
 Page<Notice> pageData = new Page<>(page, size);  
  
 //执行分页查询  
 List<Notice> noticeList = noticeDao.selectPage(pageData, new EntityWrapper<>(notice));  
  
 //设置结果集到分页对象中  
 pageData.setRecords(noticeList);  
  
 //返回  
 return pageData;  
}

测试分页查询消息效果，通过POST请求，输入Body内容作为Notice对象的查询条件，可以查询出对应的消息数据，并分页显示。

1. 新增消息通知

当作者发布了一篇文章时，需要新增一条消息。

添加Controller层的save方法：

//新增消息通知  
// http://127.0.0.1:9014/notice POST  
@RequestMapping(method=RequestMethod.*POST*)  
public Result save(@RequestBody Notice notice){  
 noticeService.save(notice);  
 return new Result(true, StatusCode.*OK*, "新增成功");  
}

添加Servic层的save方法：

注入 分布式ID生成器，生成ID， 添加消息到消息数据表以及 带推送消息数据表：”

public void save(Notice notice) {  
 //添加消息到 消息数据表以及带推送消息数据表中  
 //设置初始值  
 notice.setState("0");  
 notice.setCreatetime(new Date());  
  
 //使用分布式ID生成器， 生成ID  
 String id = idWorker.nextId() + "";  
 notice.setId(id);  
 //保存消息到消息数据表中  
 noticeDao.insert(notice);  
  
 //保存到带推送数据表中  
 NoticeFresh noticeFresh = new NoticeFresh();  
 noticeFresh.setNoticeId(id);  
 noticeFresh.setUserId(notice.getReceiverId());  
 noticeFreshDao.insert(noticeFresh);  
}

通过POST请求发送一个Notice对象的Json格式数据，可以看到数据表新增消息成功，添加了一条新的消息。

1. 修改通知 方法

我们需要对消息的状态进行修改，因此需要修改通知方法。

添加 Controller层方法 updateById

//修改通知  
// http://127.0.0.1:9014/notice PUT  
@RequestMapping(method=RequestMethod.*PUT*)  
public Result updateById(@RequestBody Notice notice){  
 noticeService.updateById(notice);  
 return new Result(true, StatusCode.*OK*, "修改成功");  
}

添加Service层方法 updateById

public void updateById(Notice notice) {  
 noticeDao.updateById(notice);  
}

通过PUT请求，发送一个notice消息对象的Json格式数据，根据消息id修改对应的消息。

1. 根据用户ID条件分页查询用户的待推送新消息

同样是分页查询，这里查询的是带推送的新消息

添加Controller层方法 freshPage:

//根据用户ID分页查询改用护的带推送新消息  
// http://127.0.0.1:9014/notice/fresh/{userId}/{page}/{size} GET  
@RequestMapping(value="fresh/{userId}/{page}/{size}", method = RequestMethod.*GET*)  
public Result freshPage(@PathVariable String userId,  
 @PathVariable Integer page,  
 @PathVariable Integer size){  
 Page<NoticeFresh> pageData = noticeService.freshPage(userId,page, size );  
 PageResult<NoticeFresh> pageResult = new PageResult<>(pageData.getTotal(), pageData.getRecords());  
 return new Result(true, StatusCode.*OK*, "查询成功", pageResult);  
}

添加Service层方法freshPage:

public Page<NoticeFresh> freshPage(String userId, Integer page, Integer size) {  
 //封装查询条件  
 NoticeFresh noticeFresh = new NoticeFresh();  
 noticeFresh.setUserId(userId);  
  
 //创建分页对象  
 Page<NoticeFresh> pageData = new Page<>(page, size);  
  
 //执行查询  
 List<NoticeFresh> list = noticeFreshDao.selectPage(pageData, new EntityWrapper<>(noticeFresh));  
  
 //设置查询结果集到分页对象中  
 pageData.setRecords(list);  
  
 //返回结果  
 return pageData;  
}

通过发送GET请求 访问 127.0.0.1:9014/notice/fresh/2/1/2， 可以查询出消息ID为2的所有数据并分页展示。

1. 删除待推送新消息

当新消息已经被用户读取到时，就需要将该消息从待推送数据表中删除。

添加Controller方法 freshDelete：

//删除待推送新消息  
// http://127.0.0.1:9014/notice/fresh DELETE  
@RequestMapping(value = "fresh", method=RequestMethod.*DELETE*)  
public Result freshDelete(@RequestBody NoticeFresh noticeFresh){  
 noticeService.freshDelete(noticeFresh);  
 return new Result(true, StatusCode.*OK*, "删除成功");  
}

添加Service方法

public void freshDelete(NoticeFresh noticeFresh) {  
 noticeFreshDao.delete(new EntityWrapper<>(noticeFresh));  
}

发送DELETE请求，删除待推送消息数据表的数据。

1. 完善返回的消息内容

在数据表中为了性能，我们并没有存储用户昵称，文章标题等信息，仅仅存储了用户的ID和文章的ID。但是我们在消息对象中设置了用户昵称和文章标题字段，以方便用户阅读消息的体验，因此，我们需要从用户微服务中查询用户昵称，从文章微服务中查询文章标题，并且在返回消息之前设置到消息对象中。

1. 通知微服务调用文章微服务

由于要在通知微服务中通过Feign远程调用文章的微服务，首先要在通知微服务的引导类中添加注解@EnableFeignClient，在tensquare\_notice子工程中创建client包，用于放置远程调用的类和接口。

client包中只需要添加接口，调用其他微服务的函数即可。因此添加ArticleClient接口用于调用文章微服务的功能，添加注解@FeignCient(“tensquare\_article”)，表示这是一个Feign调用的客户端，调用的是tensquare\_artilce服务，这是文章微服务在注册中心中的名称。然后，根据文章微服务中Controller层中已有的功能函数findBtId，在ArticleClient接口中添加该函数，注意，注解@RequestMapping中的value值要填写文章微服务中的完整后缀，如文章微服务中为/{articleId}，就要修改为article/{articleId}，同时，要在注解@PathVariable中添加articleId。如下所示：

package com.tensquare.notice.client;  
  
import entity.Result;  
import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
  
@FeignClient("tensquare-article")  
public interface ArticleClient {  
   
 //根据文章ID查询文章数据  
 // GET /article/{articleId}   
 @RequestMapping(value="article/{articleId}", method= RequestMethod.*GET*)  
 public Result findById(@PathVariable("articleId") String articleId);  
}

1. 通过通知微服务调用用户微服务

同理，创建UserClient接口调用用户微服务的查询用户的功能。

package com.tensquare.notice.client;  
  
import entity.Result;  
import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
  
@FeignClient("tensquare-article")  
public interface UserClient {  
  
 //根据用户id查询用户  
 // http://127.0.0.1:9008/user/userId GET  
 @RequestMapping(value="user/{userId}", method= RequestMethod.*GET*)  
 public Result selectById(@PathVariable("userId") String userId);  
}

1. 通知微服务中实现查询文章标题和用户昵称

由于通知微服务中通过Controller层的selectById方法查询消息数据表中的消息信息，该方法调用Service层中的selectById方法，返回消息对象。因此，我们直接在Service层中创建新的私有方法getInfo，用于调用上面定义好的ArticleClient接口和UserClient接口获取文章标题和用户昵称。

由于传递的都是 Json格式数据，可以将数据转换为HashMap格式的数据，方便取数据。因此，我们可以将从文章微服务中返回的Result数据强转为HashMap类型数据，获取到其中的Data数据，再从中获取到文章标题数据，存储到消息对象中。

@Autowired  
private UserClient userClient;  
  
@Autowired  
private ArticleClient articleClient;  
  
private void getInfo(Notice notice){  
 //查询用户昵称  
 Result userResult = userClient.selectById(notice.getOperatorId());  
 HashMap userMap = (HashMap)userResult.getData();  
 //设置操作用户昵称到消息对象中  
 notice.setOperatorId(userMap.get("nickname").toString());  
   
 //查询文章标题  
 Result articleResult = articleClient.findById(notice.getTargetId());  
 HashMap articleMap = (HashMap) articleResult.getData();  
 //设置文章标题到消息对象中  
 notice.setTargetName(articleMap.get("title").toString());  
}

1. 在Service层的查询用户信息和查询文章信息的方法中调用getInfo方法

只有调用了getInfo方法，文章标题和用户昵称才会写入到消息对象中，因此在Service层中的两个查询消息的方法中调用getInfo方法：

public Notice selectById(String id) {  
 Notice notice = noticeDao.selectById(id);  
 //完善信息  
 getInfo(notice);  
 return notice;  
}

public Page<Notice> selectByPage(Notice notice, Integer page, Integer size) {  
 //封装分页对象  
 Page<Notice> pageData = new Page<>(page, size);  
  
 //执行分页查询  
 List<Notice> noticeList = noticeDao.selectPage(pageData, new EntityWrapper<>(notice));  
  
 //完善消息  
 for(Notice n : noticeList){  
 getInfo(n);  
 }  
   
 //设置结果集到分页对象中  
 pageData.setRecords(noticeList);  
  
 //返回  
 return pageData;  
}

启动4个微服务，article ,eureka, user , notice ，通过访问notice获取消息，可以看到notice对象的用户昵称和文章标题已经被填上了对应的字符串。

注意，这里需要校验是否存在用户昵称和文章标题，否则，再调用getInfo时会获取不到对应的数据，导致异常！！！

四． 文章订阅-实现群发消息功能

（一） 订阅文章作者

1. 需求分析

用户在登录后，可以通过文章订阅文章作者，之后可以收到这个作者发布的新文章消息，需要根据文章ID订阅文章作者。

2. 功能分析

（1） 用户与作者的订阅关系存储在Redis中。

（2） 用户维护一个订阅作者集合（set集合），作者维护一个订阅用户集合（set集合），set集合保证了数据的单一性。

1. Contorller层subscribe方法

用户通过访问文章微服务，订阅文章作者，因此我们需要在文章微服务tensquare\_article的ArticleController中添加一个订阅方法subscribe，传入articleId和userId，调用Service层的subscribe方法，根据返回值，判断是订阅作者还是取消订阅。

//根据文章ID和用户ID，建立订阅关系，保存作者ID和用户ID的关系  
// http://127.0.0.1:9004/article/subscribe POST  
@RequestMapping(value="subscribe", method=RequestMethod.*POST*)  
public Result subscribe(@RequestBody Map map){  
 //返回True订阅，false取消订阅  
 Boolean flag = articleService.subscribe(map.get("articleId").toString(),  
 map.get("userId").toString());  
 if(flag == true){  
 return new Result(true, StatusCode.*OK*, "订阅成功");  
 }else{  
 return new Result(true, StatusCode.*OK*, "取消订阅成功");  
 }  
}

1. Service层subscribe方法

Service层中需要根据文章ID查询作者ID，然后查询Redis中存储的订阅关系，该用户是否已经订阅过该作者了，如果订阅了取消订阅，否则订阅。这里需要使用到Redis，因此需要在虚拟机中启动Redis的docker容器，使用RDM可视化工具查看Redis中的内容。同时注入reids的模板类RedisTemplate，方便操作。

public Boolean subscribe(String articleId, String userId) {  
 //根据文章ID查询文章作者ID  
 String authorId = articleDao.selectById(articleId).getUserid();  
 //存放用户订阅信息的集合key，存放着作者ID  
 String userKey = "article\_subscribe\_" + userId;  
 //存放作者订阅者信息的集合Key，存放订阅者ID  
 String authorKey = "article\_author\_" + authorId;  
  
 //查询用户的订阅关系，是否有订阅该作者  
 Boolean flag = redisTemplate.boundSetOps(userKey).isMember(authorId);  
  
 if(flag == true){  
 //如果用户已经订阅了该作者，则取消  
 //在用户订阅信息的集合中删除作者  
 redisTemplate.boundSetOps(userKey).remove(authorId);  
 //在作者订阅者消息结合中删除订阅者  
 redisTemplate.boundSetOps(authorKey).remove(userId);  
 return false;  
 }else{  
 redisTemplate.boundSetOps(userKey).add(authorId);  
 redisTemplate.boundSetOps(authorKey).add(userId);  
 return true;  
 }  
}

1. 测试订阅关系存储

通过POST请求访问 <http://127.0.0.1:9004/article/subscribe>，并且添加RequestBody内容为

{

"articleId":"1",

"userId":"3"

}

可以看到，Redis中添加了订阅者集合和用户集合，测试成功。

（二）新增文章群发消息

1．需求分析

在作者新发布了一篇文章后，需要通知所有订阅的用户，因此文章微服务需要调用消息通知微服务，创建消息。

用户登录以后，访问前端页面， 页面需要定时轮询通知接口，获取消息。

2. 添加依赖

由于文章微服务需要调用其他的微服务，因此需要在文章微服务中添加feign依赖。

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-openfeign</artifactId>  
</dependency>

3 [文章微服务启动类添加注解@EnableFeignClients](mailto:3.文章微服务添加注解@EnableFeignClients)开启feign客户端

4． 在文章微服务中编写调用消息微服务的接口

注意，这里需要使用到消息通知微服务中的一个POJO，需要将它从消息通知微服务中复制到文章微服务中。

package com.tensquare.article.client;  
  
import com.tensquare.article.pojo.Notice;  
import entity.Result;  
import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.RequestBody;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
  
@FeignClient(value="tensquare-notice")  
public interface NoticeClient {  
 //新增通知  
 // http:127.0.0.1:9014/notice POST  
 @RequestMapping(value="notice", method= RequestMethod.*POST*)  
 public Result save(@RequestBody Notice notice);  
}

5 修改文章微服务的Service层save新增文章方法

当新增一篇文章时，同时需要调用消息通知微服务通知所有订阅用户，首先我们要获取当前作者的信息，通过作者的ID号在Redis中获取到所有的订阅用户号，为每个用户创建新的消息通知，然后为这个消息添加各种参数，如操作盒，接受用户……最后调用消息通知微服务的新增方法，保存到数据库中。

public void save(Article article) {  
 //使用Id生成器  
 String id = idWorker.nextId()+"";  
 article.setId(id);  
  
 //初始化非设置的数据，而是自增的数据  
 article.setVisits(0);//浏览量  
 article.setThumbup(0);//点赞数  
 article.setComment(0);//评论数  
  
 //新增  
 articleDao.insert(article);  
  
 //新增文章后，创建消息，通知给订阅者  
 //通过微服务的鉴权获取当前用户的信息，用户ID，即文章作者ID  
 String userId = "3";  
 article.setUserid(userId);  
 //存放作者订阅者消息的集合key,里面存放着订阅者ID  
 String authorKey = "article\_author\_" + userId;  
 Set<String> set = redisTemplate.boundSetOps(authorKey).members();  
 if(null != set && set.size() > 0){  
 //给订阅者创建消息通知  
 Notice notice = new Notice();  
 for(String uid : set){  
  
 //接受消息用户的ID  
 notice.setReceiverId(uid);  
 //进行操作用户的ID  
 notice.setOperatorId(userId);  
 //操作类型  
 notice.setAction("publish");  
 //被操作的对象类型  
 notice.setTargetType("article");  
 //被操作对象的ID  
 notice.setTargetId(id);  
 //通知类型  
 notice.setType("sys");  
 //调用消息通知微服务的save方法  
 noticeClient.save(notice);  
 }  
 }  
  
}

1. 测试消息群发效果

启动eureka, user, article, notice 4个微服务，我们之前已经为作者ID为3的作者订阅了2个用户，用户ID分别为1和2.那么，当我们通过POST请求访问 <http://127.0.0.1:9004/article>时文章微服务就会查阅Redis中的订阅关系，找出订阅者Set集合中的用户ID，新增两个通知消息并保存消息到Mysql数据库中。

1. 前端页面显示

前端页面通过轮询，不断查询是否有新的消息出现，通过消息微服务中的根据ID查询接口查询。

五． 文章点赞，实现点对点消息功能

（一） 实现文章点赞

在文章微服务的Controller层中添加thumbup点赞方法，首先需要获取到当前用户的ID，然后查询Redis中当前用户对当前文章的点赞信息，如果没有点赞，就进行点赞，并且保存点赞信息。

//根据文章ID点赞文章  
// http://127.0.0.1:9004/article/thumbup/{articleId} PUT  
@RequestMapping(value="thumbup/{articleId}", method=RequestMethod.*PUT*)  
public Result thumbup(@PathVariable String articleId){  
 // 通过JWT鉴权获取当前用户的ID  
 String userId = "4";  
 //查询用户对文章的点赞信息  
 String key = "thumbup\_article\_" + userId + " " + articleId;  
 Object flag = redisTemplate.opsForValue().get(key);  
  
 //判断查询的结果  
 if(flag == null){  
 //用户未点过赞  
 articleService.thumbup(articleId, userId);  
 //保存点赞信息  
 redisTemplate.opsForValue().set(key,1);  
 return new Result(true, StatusCode.*OK*, "点赞成功");  
 }else{  
 return new Result(false, StatusCode.*REPERROR*, "不能重复点赞");  
 }  
}

编写Service层的thumbup方法，同样的，首先点赞，修改点赞信息，然后创建消息点对点地发送给文章作者，设置消息的参数，最后保存消息到数据库中。

public void thumbup(String articleId, String userId) {  
  
 //文章点赞  
 Article article = articleDao.selectById(articleId);  
 article.setThumbup(article.getThumbup() + 1);  
 articleDao.updateById(article);  
  
 //发送消息给文章作者  
 Notice notice = new Notice();  
 notice.setReceiverId(article.getUserid());  
 notice.setOperatorId(userId);  
 notice.setAction("publish");  
 notice.setTargetType("article");  
 notice.setTargetId(articleId);  
 notice.setType("user");  
  
 //保存消息到数据库中  
 noticeClient.save(notice);  
}

六． 基于数据库实现的消息通知系统存在的问题

（一） 消息通知系统的构成

一个消息通知系统，主要由消息发送者，消息存储，消息接受者，新消息提醒机制组成。

1. 消息发送者

系统的开发者设置了某种消息发送的规则，规则中包含一些条件，当条件都满足后，触发系统生成消息数据，系统将消息数据保存并推送给接受者。

1. 消息存储

消息通知的存储包含消息通知实体数据的存储和新消息数据的存储。

1. 消息接受者

消息最终的目的地。

1. 新消息提醒机制

系统产生新的消息后，需要告知接受者有新的消息，如提醒新消息的数量，置顶新消息并标记。

（二）现有消息系统存在的问题

1. 数据库访问压力大

消息实体和新消息提醒都在数据库中，读写操作频繁。

2. 服务器性能压力大

现有系统采用页面轮询接口的方式实现服务器向用户推送消息通知，服务器的接口访问压力大。可以改进使用rabbitmq实现新消息提醒数据的缓存功能，代替新消息数据表；使用全双工长连接的方式实现服务器向用户推送最新的消息通知，替换轮询，页面采用websocket，微服务端使用异步高性能框架netty。