1. 消息队列的选取
2. 单机吞吐量

万级 ： ActiveMQ, RabbitMQ

十万级：RocketMQ, Kafka

1. 时效性

ms级： ActiveMQ, RocketMQ, Kafka

微秒级： RabbitMQ

1. 消息可靠性

ActiveMQ ; 有较低的概率丢失数据。

RabbitMQ: 无丢失数据的概率。

RocketMQ: 经过参数优化配置，可以做到0丢失。

Kafka: 经过参数优化配置，可以做到0丢失。

1. 可用性

ActiveMQ, RabbitMQ: 高，基于主从架构实现高可用性。

RocketMQ, kafka: 分布式架构

1. 消息通知系统改进
2. 获取新消息通知
3. 改进

之前的新消息提醒使用定时轮询接口的方式过于低效，我们需要作出改进：

1. 将新消息提醒数据从数据表转移到rabbitmq中，减轻数据库访问压力。
2. 将轮询接口的伪推送改为真正采用全双工长连接的推送

消息通知微服务中加入netty框架， 为页面中的websocket连接提供接入服务。

netty框架与rabbitmq对接，接受并下发新消息提醒数据，

将页面中的定时轮询接口替换为websocket连接和事件处理。

1. 用户获取新消息的两种方式
2. 上线登录后向系统主动索取

接受者向服务端netty请求websocket连接，netty服务把连接放到自己的连接池中

netty根据接受者信息向rabbitmq查询消息，如果有新消息，则返回新消息通知

使用websocket连接向接受者返回新消息数量

1. 在线时系统向接受者主动推送新消息

rabbitmq将新消息推送给netty

netty从连接池中取出接受者的websocket连接

netty通过接受者的websocket连接返回新消息的数量

1. 文章订阅群发消息改进
2. 改进步骤
3. 准备rabbitmq消息中间件。
4. 改进文章订阅功能，创建Rabbitmq队列存放新消息通知。
5. 改进发布文章后群发消息通知功能，
6. 整合netty和websocket实现双向通信。
7. 虚拟机中启动rabbitmq

docker run -id --name=tensquare\_rabbit -p 5671:5671 -p 5672:5672

-p 4369:4369 -p 15672:15672 -p 25672:25672 rabbitmq:management

然后连接虚拟机IP的15672端口，使用guest账户，guest密码登录。

如果可以看到rabbitmq界面，说明成功。

1. 修改文章微服务配置文件

由于文章订阅功能需要增加rabbitmq的交换机和队列的绑定，解绑等操作，因此需要让rabbitmq\_article具有操作rabbitmq的能力，让每个用户都有一个队列来处理新消息。

在application.yml中添加配置：

spring:

rabbitmq:

host: 192.168.

在pom.xml中添加amqp依赖：

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

1. 修改文章订阅功能

之前我们是将用户订阅文章的订阅关系保存在了Redis中，每次从Redis中查询订阅关系。这里的改进是在订阅文章作者时，创建Rabbitmq的一个direct类型的交换机，然后创建该用户的队列，将该队列通过路由键绑定或解绑到交换机上。这样，当有新消息时，直接通过Rabbitmq的交换机发送到对应用户的队列中，实现新消息提醒功能。

1. 在ArticleService类中的subscribe方法添加操作rabbitmq的代码
2. 创建Rabbitmq的管理器，用于管理rabbitmq的各种操作，还需要注入RabbitmqTemplate模板类：

//创建Rabbit管理器  
RabbitAdmin rabbitAdmin = new RabbitAdmin(rabbitTemplate.getConnectionFactory());

1. 声明交换机

DirectExchange exchange = new DirectExchange("article\_subscribe");  
rabbitAdmin.declareExchange(exchange);

1. 创建用户队列

//创建用户的队列，用userId来区分  
Queue queue = new Queue("article\_subscribe" + userId, true);

（5） 绑定交换机和队列，并且通过路由键来绑定作者，确保订阅用户只收到对应作者的消息

Binding binding = BindingBuilder.*bind*(queue).to(exchange).with(authorId);

（6）当取消订阅时，删除交换机和队列的绑定关系

rabbitAdmin.removeBinding(binding);

（7） 订阅时，添加绑定关系

rabbitAdmin.declareQueue(queue);  
rabbitAdmin.declareBinding(binding);

（8） 测试文章订阅功能

同样的，使用POST请求发送文章订阅功能，当订阅成功时，rabbitmq中就会多了一个交换机和队列，并且有绑定关系；当取消订阅时，绑定关系删除。

1. 发布文章提醒消息

原有的处理中，将新消息通知保存到mysql中，这里我们使用rabbitmq将消息保存mq中。一有新消息产生，就发送至交换机队列中保存着。

1. 修改ArticleService中的save方法

原有的save方法中将消息和待推送消息都保存到mysql中，这里将待推送消息发送到rabbitmq中。因此，需要在save方法后添加发送语句：

//入库成功后，发送mq消息  
//交换机名称，路由键，内容  
rabbitTemplate.convertAndSend("article\_subscribe", userId, id);

1. 在NoticeService的save方法中删除新消息通知入库逻辑

由于文章微服务调用了通知微服务的save方法，用于保存消息和待推送消息到mysql中，因此，我们需要将待推送新消息入库的代码删除，因为这些待推送消息都会发送到rabbitmq中。

1. 测试效果

使用文章微服务的新增文章，查看rabbitmq中是否多了对应的交换机和绑定的队列，队列中是否有对应的待推送消息。

1. IO编程
2. 传统IO编程

传统的阻塞式IO通信模型中，每个客户端连接过来后，服务端都会启动一个线程来处理该客户端的请求。这种通信方式对于客户端连接巨大的情况是无法处理的，因为并不是所有的线程都会被使用，很大一部分的线程都会处于阻塞的情况，造成资源的极大浪费。且阻塞式通信，线程爆炸后操作系统频繁地进行线程切换，应用性能急剧下降。且IO编程中数据读写是以字节流为单位的，效率不高。

1. NIO编程

new-IO或者 non-blocking-IO，可理解为非阻塞IO。在该模型中，新的客户端连接不会在服务器中创建一个新的线程，而是将这条连接直接绑定到某个固定的线程，这条连接的所有的读写都由这个线程来负责。其实是将这条连接注册到选择器上，通过检查这个选择器，可以批量地检测出是否有数据可读的连接，进而读取数据。

1. NIO三大组件
2. 通道channel

传统IO中使用Stream流，是单向的，读写分离的。而Channel是双向的，既可以进行读操作，也可以进行写操作。

1. 缓存Buffer

Buffer是一块内存区域，可以写入数据也可以读取。

1. 选择器Selector

选择器可以实现一个单独的线程监控多个注册的信道，通过一定的选择机制，实现多路复用的效果。

1. NIO相对于IO的优势
2. IO是面向流的，一个一个字节地读取数据，并且数据只能从一端读取到另一端，不能前后移动流中的数据，NIO是面向缓冲区的，可以从缓冲区读取一块数据，也可以在缓冲区中前后移动。
3. IO是阻塞的，NIO是非阻塞的，可以异步操作。
4. NIO引入了IO多路复用器Selector，降低线程数量，线程切换效率大大提高。
5. Netty
6. 为什么使用Netty
7. Netty是一个异步事件驱动的网络应用框架，用于快速开发可维护的高性能服务器和客户端。
8. Netty底层IO模型随意切换，可以很快从NIO模型切换为IO模型。
9. Netty自带的拆包捷豹，异常检测等机制，可以从NIO的繁重细节中脱离出来，只需要关心业务逻辑。
10. 自带各种协议栈，在处理任何一种通用协议时都不用亲自动手。
11. 在RPC框架，消息中间件，分布式通信中间件线上的广泛验证，十分健壮。
12. 使用Netty

创建netty\_demo子项目。

1. 添加依赖

<dependency>  
 <groupId>io.netty</groupId>  
 <artifactId>netty-all</artifactId>  
 <version>4.1.50.Final</version>  
</dependency>

1. 编写Netty服务器端代码

直接复制，以后慢慢学。

package com.netty;  
  
import io.netty.bootstrap.ServerBootstrap;  
import io.netty.channel.ChannelHandlerContext;  
import io.netty.channel.ChannelInitializer;  
import io.netty.channel.SimpleChannelInboundHandler;  
import io.netty.channel.nio.NioEventLoopGroup;  
import io.netty.channel.socket.nio.NioServerSocketChannel;  
import io.netty.channel.socket.nio.NioSocketChannel;  
import io.netty.handler.codec.string.StringDecoder;  
  
public class NettyServer {  
 public static void main(String[] args) {  
 ServerBootstrap serverBootstrap = new ServerBootstrap();  
 NioEventLoopGroup boos = new NioEventLoopGroup();  
 NioEventLoopGroup worker = new NioEventLoopGroup();  
 serverBootstrap  
 .group(boos, worker)  
 .channel(NioServerSocketChannel.class)  
 .childHandler(new ChannelInitializer<NioSocketChannel>() {  
 protected void initChannel(NioSocketChannel ch){  
 ch.pipeline().addLast(new StringDecoder());  
 ch.pipeline().addLast(new SimpleChannelInboundHandler<String>() {  
 @Override  
 protected void channelRead0(ChannelHandlerContext ctx, String msg){  
 System.*out*.println(msg);  
 }  
 });  
 }  
 })  
 .bind(8000);  
 }  
}

1. 编写Netty客户端代码

package com.netty;  
  
import io.netty.bootstrap.Bootstrap;  
import io.netty.channel.Channel;  
import io.netty.channel.ChannelInitializer;  
import io.netty.channel.nio.NioEventLoopGroup;  
import io.netty.channel.socket.nio.NioSocketChannel;  
import io.netty.handler.codec.string.StringEncoder;  
  
public class NettyClient {  
 public static void main(String[] args) throws InterruptedException {  
 Bootstrap bootstrap = new Bootstrap();  
 NioEventLoopGroup group = new NioEventLoopGroup();  
 bootstrap.group(group)  
 .channel(NioSocketChannel.class)  
 .handler(new ChannelInitializer<Channel>() {  
 @Override  
 protected void initChannel(Channel ch){ ch.pipeline().addLast(new StringEncoder());}  
 });  
 Channel channel = bootstrap.connect("127.0.0.1", 8000).channel();  
 while(true){  
 channel.writeAndFlush("测试数据");  
 Thread.*sleep*(2000);  
 }  
 }  
}

1. 测试Netty

打开Netty服务器端和客户端，可以看到服务器端可以接收到客户端的数据。

1. Netty的事件驱动
2. 事件驱动模型思路

事件队列负责存储发生的事件，事件泵不断从队列中取出时间，根据不同的事件调用不同的函数。

1. 为什么使用事件驱动

程序中的任务可以并行执行，任务之间高度独立，不需要相互等待。

1. Netty事件驱动架构

事件队列： 接受事件的入口

分发器： 将不同的事件分发到不同的业务逻辑单元

事件通道： 分发器与处理器之间的联系通道

事件处理器： 实现业务逻辑，处理完成后会发出时间，触发下一步操作

1. Netty架构
2. Netty支持传统的阻塞BIO Socket传输，也支持非阻塞NIO Socket传输。BIO适合连接数小且固定的长连接；NIO适合连接数多的短连接。
3. Netty支持多种容器，如OSGI, JBossMC, Spring, Guice容器。
4. Netty支持多种协议，HTTP, Protobuf, 文本，WebSocket等
5. Netty核心组件
6. ServerBootstrap

用于接收客户端的连接以及为已接收的连接创建子通道，一般用于服务器。

1. Bootstrap

不接受新的连接，在父通道类完成操作，一般用于客户端。

1. Channel

对网络套接字的IO操作，例如读，写，连接，绑定等操作进行适配和封装。

1. EventLoop

处理所有注册其上的channel的IO操作，同样一个EventLoop可以为多个channel提供服务。

1. EventLoopGroup

包含有多个EventLoop实例，用来管理EventLoop。

1. ChannelHandler 和 ChannelPipeline

ChannelPipeline是Channel处理的流水线，ChannelHandler是流水线上的处理器，处理的都是事件。

1. ChannelInitializer

对刚刚创建的channel初始化，将ChannelHandler添加到channel的channelPipeline处理链路中。

1. ChannelFuture

实现并行处理，在操作执行成功或失败时自动触发监听器中的事件处理方法。

1. 整合Netty和WebSocket

使用Netty对接WebSocket连接，实现双向通信，这一步需要服务端的Netty程序，用来处理客户端的WebSocket连接操作，如建立连接，断开连接，收发数据。

（一） 添加配置

由于新消息通知是通过消息通知微服务实现的，因此需要在消息通知微服务中修改pom.xml文件，添加rabbitmq依赖，以及netty依赖，webscoket已经整合在了netty中。

<!-- rabbitmq-->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-amqp</artifactId>  
 </dependency>  
<!-- netty-->  
 <dependency>  
 <groupId>io.netty</groupId>  
 <artifactId>netty-all</artifactId>  
 <version>4.1.50.Final</version>  
 </dependency>

在application.yml中添加rabbitmq的地址，与rabbitmq实现交互：

spring:

rabbitmq:

host: 192.168

（二） 实现Netty的整合

1. 整合分析

前台页面中使用websocket连接，Netty管理这些连接，同时还需要获取Rabbitmq中的消息。WebSocket相当于客户端，NettyServer作为服务端，因此整合步骤如下：

1. 编写NettyServer，启动Netty服务。
2. 使用配置Bean创建Netty服务，编写NettyConfig，用于启动Netty服务。
3. 编写和WebSocket进行通信的类MyWebSocketHandler，作为流水线上的处理器，进行MQ和WebSocket的消息处理。
4. 使用配置Bean创建Rabbit监听器容器，使用监听器，编写RabbitConfig。
5. 编写Rabbit监听器SysNoticeListener，用来获取MQ消息并进行处理。

2.创建目录结构

创建上面需要的类。

3.实现整合

1. 复制工具类

复制资料中的ApplicationContextProvider.java到 config包下，这个类是一个工具类，能够获取Spring容器中的实例，如通过Bean的名称，类型来获取Bean等等。

1. 编写NettyServer

与之前创建Netty 服务端步骤相似，虽然是固定写法，但是很多定义需要学习。同时还需要为MyWebSocketHandler继承 SimpleChannelInboundHandler。

package com.tensquare.notice.netty;  
  
import io.netty.bootstrap.ServerBootstrap;  
import io.netty.channel.Channel;  
import io.netty.channel.ChannelInitializer;  
import io.netty.channel.EventLoopGroup;  
import io.netty.channel.nio.NioEventLoopGroup;  
import io.netty.channel.socket.nio.NioServerSocketChannel;  
import io.netty.handler.codec.http.HttpObjectAggregator;  
import io.netty.handler.codec.http.HttpServerCodec;  
import io.netty.handler.codec.http.websocketx.WebSocketServerProtocolHandler;  
  
public class NettyServer {  
  
 public void start(int port){  
 System.*out*.println("准备启动Netty......");  
 ServerBootstrap serverBootstrap = new ServerBootstrap();  
  
 //用来处理新连接  
 EventLoopGroup boos = new NioEventLoopGroup();  
 //用来处理业务逻辑  
 EventLoopGroup worker = new NioEventLoopGroup();  
  
 serverBootstrap.group(boos, worker)  
 .channel(NioServerSocketChannel.class)  
 .childHandler(new ChannelInitializer() {  
 @Override  
 protected void initChannel(Channel ch) throws Exception {  
 //请求消息解码器  
 ch.pipeline().addLast(new HttpServerCodec());  
 //将多个消息转换为单一的request或者response对象  
 ch.pipeline().addLast(new HttpObjectAggregator(65536));  
 //处理WebSocket的消息事件  
 ch.pipeline().addLast(new WebSocketServerProtocolHandler("/ws"));  
  
 //创建自己的WebSocket处理器，用来编写业务逻辑  
 MyWebSocketHandler myWebSocketHandler = new MyWebSocketHandler();  
 ch.pipeline().addLast(myWebSocketHandler);  
 }  
 }).bind(port);  
 }  
}

1. 编写NettyConfig

配置NettyServer，启动Netty服务，与普通的配置类一样，创建一个方法createNettyServer，返回创建的NettyServer对象，并且启动一个新的线程，在该线程中调用NettyServer类的start方法，最后返回这个NettyServer作为Spring容器中的Bean对象。

package com.tensquare.notice.config;  
  
import com.tensquare.notice.netty.NettyServer;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class NettyConfig {  
 @Bean  
 public NettyServer createNettyServer(){  
 NettyServer nettyServer = new NettyServer();  
  
 //启动Netty服务，使用新的线程  
 new Thread(){  
 @Override  
 public void run(){  
 nettyServer.start(1234);  
 }  
 }.start();  
 return nettyServer;  
 }  
}

测试当前的Netty服务是否能够启动，由于NettyServer已经是Spring 容器中的一个Bean了，因此只要启动通知微服务后，查看控制台中是否输出了Netty服务正在启动即可，即表示NettyServer已经启动。

1. 编写MyWebSocketHandler

cahnnelRead0是用户请求WebSocket服务端时执行的方法，因为WebSocket是全双工的。在用户第一次请求时，需要建立WebSocket连接，约定用户第一次请求携带的数据为Json格式的{“userId”:”1”}，来获取用户的ID。ObjectMapper是一个工具类，用于解析数据。

第一次请求时，需要建立WebSocket连接，并且需要存放这个连接，通过建立一个map，保存该用户ID和连接的对应关系。使用线程安全的ConcurrentHashMap存放，判断当前的用户是否已经有连接了，如果没有，则通过上下文获取WebSocket连接，保存到Map中。

获取RabbitMQ的消息内容，并发送给用户，这里需要使用RabbitAdmin，传入RabbitTemplate，之前我们的Spring容器中已经有了一个RabbitTemplate可以直接使用了，不过需要使用ApplicationContextProvider工具类从Spring容器中获取该Bean对象。

使用RabbitAdmin获取Rabbitmq中该用户的队列，并获取该用户的新消息数量，封装返回的数据，通过WebSocket发送给用户。

上面实现的功能都是在用户请求服务端时触发的动作，当用户在线时，服务端有新消息时可以主动推送给用户，不需要用户请求服务端。这就需要一个监听器监听着用户的消息通知队列了，同样的通过工具类ApplicationContextProvider从Spring容器中获取。然后将该用户通知消息队列添加到监听器容器中。

package com.tensquare.notice.netty;  
  
import com.fasterxml.jackson.databind.ObjectMapper;  
import com.tensquare.notice.config.ApplicationContextProvider;  
import entity.Result;  
import entity.StatusCode;  
import io.netty.channel.Channel;  
import io.netty.channel.ChannelHandlerContext;  
import io.netty.channel.SimpleChannelInboundHandler;  
import io.netty.handler.codec.http.websocketx.TextWebSocketFrame;  
import org.springframework.amqp.rabbit.core.RabbitAdmin;  
import org.springframework.amqp.rabbit.core.RabbitTemplate;  
import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;  
  
import java.util.HashMap;  
import java.util.Properties;  
import java.util.concurrent.ConcurrentHashMap;  
  
public class MyWebSocketHandler extends SimpleChannelInboundHandler<TextWebSocketFrame> {  
 //数据解析工具类  
 private static ObjectMapper *MAPPER* = new ObjectMapper();  
 //存放WebSocket连接的Map，根据用户ID存放  
 public static ConcurrentHashMap<String, Channel> *userChannelMap* = new ConcurrentHashMap<>();  
 //从Spring容器中获取RabbitTemplate  
 RabbitTemplate rabbitTemplate = ApplicationContextProvider.*getApplicationContext*()  
 .getBean(RabbitTemplate.class);  
 //从Spring容器中获取消息监听器容器，处理订阅消息sysNotice  
 SimpleMessageListenerContainer sysNoticeContainer = (SimpleMessageListenerContainer) ApplicationContextProvider.*getApplicationContext*()  
 .getBean("sysNoticeContainer");  
 //用户请求WebSocket服务端时执行的方法  
 @Override  
 protected void channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg) throws Exception {  
 //约定用户第一次请求携带的数据{"userId":"1"}  
 //获取用户数据并解析  
 String json = msg.text();  
 //解析json数据，获取用户ID  
 String userId = *MAPPER*.readTree(json).get("userId").asText();  
  
 //第一次请求时，建立WebSocket连接  
 Channel channel = *userChannelMap*.get(userId);  
 if(channel == null){  
 //获取WebSocket的连接  
 channel = ctx.channel();  
 //将连接放到Map中  
 *userChannelMap*.put(userId, channel);  
 }  
  
 //获取Rabbitmq的消息内容，并发送给用户  
 //只完成新消息的题型，获取消息的数量  
 RabbitAdmin rabbitAdmin = new RabbitAdmin(rabbitTemplate);  
 //拼接获取队列名称  
 String queueName = "article\_subscribe\_" + userId;  
 //获取Rabbit的Properties容器  
 Properties queueProperties = rabbitAdmin.getQueueProperties(queueName);  
 //获取消息数量  
 int noticeCount = 0;  
 if(queueProperties != null){  
 noticeCount = (int)queueProperties.get("QUEUE\_MESSAGE\_COUNT");  
 }  
  
 //封装返回的数据  
 HashMap countMap = new HashMap();  
 countMap.put("sysNoticeCount", noticeCount);  
 Result result = new Result(true, StatusCode.*OK*, "查询成功", countMap);  
  
 //将数据发送给用户  
 channel.writeAndFlush(new TextWebSocketFrame(*MAPPER*.writeValueAsString(result)));  
  
 //为用户的消息通知队列注册监听器，当用户在线时，一旦有消息，主动推送个用户，不需要用户请求  
 sysNoticeContainer.addQueueNames(queueName);  
 }  
}

1. 编写RabbitConfig

配置Rabbitmq监听器容器，在该配置类中创建监听器容器，使用channel监听，并将容器注册到Spring容器中。该容器中有多个监听器，每个监听器都需要传入ConnectionFactory，之前已经整合过Rabbitmq了，因此它会自动从Spring容器中获取ConnectionFactory。

package com.tensquare.notice.config;  
  
import com.tensquare.notice.listener.SysNoticeListener;  
import org.springframework.amqp.rabbit.connection.ConnectionFactory;  
import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class RabbitConfig {  
 @Bean("sysNoticeContainer")  
 public SimpleMessageListenerContainer create(ConnectionFactory connectionFactory){  
 SimpleMessageListenerContainer container = new SimpleMessageListenerContainer(connectionFactory);  
 //使用channel监听  
 container.setExposeListenerChannel(true);  
 //设置自己编写的监听器  
 container.setMessageListener(new SysNoticeListener());  
 return container;  
 }  
  
}

1. 编写SysNoticeListener监听器

实现ChannelAwareMessageListener接口，该接口有onMessage方法需要重写，当监听到一个消息时，就会回调该方法，在该方法中，我们可以通过Message获取消息的内容。

首先，需要从消费者队列名称中获取用户的ID，拿到用户的ID后可以通过该用户的WebSocket连接是否存在来判断是否在线。从MyWebSocketHandler类中获取到静态的存储WebSocket的Map对象，查看是否有该用户的WebSocket。如果该用户在线，就封装返回数据，数据使用Map来封装，转换成JSON格式也方便， 将数据通过WebSocket连接主动推送给用户。

这里也需要用到格式转换工具类，将封装的对象转换为Json格式数据。

package com.tensquare.notice.listener;  
  
import com.fasterxml.jackson.databind.ObjectMapper;  
import com.rabbitmq.client.Channel;  
import com.tensquare.notice.netty.MyWebSocketHandler;  
import entity.Result;  
import entity.StatusCode;  
import io.netty.handler.codec.http.websocketx.TextWebSocketFrame;  
import org.springframework.amqp.core.Message;  
import org.springframework.amqp.rabbit.listener.api.ChannelAwareMessageListener;  
  
import java.util.HashMap;  
  
public class SysNoticeListener implements ChannelAwareMessageListener {  
 private static ObjectMapper *MAPPER* = new ObjectMapper();  
  
 @Override  
 public void onMessage(Message message, Channel channel) throws Exception {  
 //获取用户的ID，通过队列名称获取  
 String queueName = message.getMessageProperties().getConsumerQueue();  
 String userId = queueName.substring(queueName.lastIndexOf("\_"+1));  
  
 io.netty.channel.Channel wsChannel = MyWebSocketHandler.*userChannelMap*.get(userId);  
 //判断用户是否在线  
 if(wsChannel != null){  
 //如果连接不为空，表示用户在线  
 //封装返回数据  
 HashMap countMap = new HashMap();  
 countMap.put("sysNoticeCount", 1);  
 Result result = new Result(true, StatusCode.*OK*, "查询成功", countMap);  
  
 //将数据通过WebSocket连接主动推送个用户  
 wsChannel.writeAndFlush(new TextWebSocketFrame(*MAPPER*.writeValueAsString(result)));  
 }  
 }  
}

1. 测试新的消息通知系统

将资料中的静态页面index.html加入到通知微服务的资源目录下，使用它进行测试。

发现在用户向服务端请求连接时，消息会被消费两次，因此需要在每次建立连接后将队列清空。

//用户建立连接后，将该消息队列清空，否则会被MQ监听到再消费一次  
if(noticeCount > 0){  
 rabbitAdmin.purgeQueue(queueName, true);  
}

同时，还要在文章微服务的订阅功能函数中添加发送消息到队列的代码，不然每次订阅后，RabbitMQ中并没有显示消息发送过来。只有队列中有消息时，前端页面才会显示出队列中的消息并处理。

1. 文章点赞点对点消息改进
2. 点赞消息发送至队列中

修改文章微服务中的ArticleService的thumbup方法，添加向用户的点对点消息队列中发送消息的功能。

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);  
  
 //创建Rabbit管理器  
 RabbitAdmin rabbitAdmin = new RabbitAdmin(rabbitTemplate.getConnectionFactory());  
 //创建队列，每个用户都有自己的队列，用用户ID区分  
 Queue queue = new Queue("article\_thumbup\_" + article.getUserid(), true);  
 rabbitAdmin.declareQueue(queue);  
 //发送消息  
 rabbitTemplate.convertAndSend("article\_thumbup\_" + article.getUserid(), articleId);  
}

1. 创建监听点赞消息的监听器UserNoticeListener

在消息微服务中，添加一个点赞消息的监听器，直接复制之前的新消息监听器SysNoticeListener，修改保存消息个数的map的键名即可。

package com.tensquare.notice.listener;  
  
import com.fasterxml.jackson.databind.ObjectMapper;  
import com.rabbitmq.client.Channel;  
import com.tensquare.notice.netty.MyWebSocketHandler;  
import entity.Result;  
import entity.StatusCode;  
import io.netty.handler.codec.http.websocketx.TextWebSocketFrame;  
import org.springframework.amqp.core.Message;  
import org.springframework.amqp.rabbit.listener.api.ChannelAwareMessageListener;  
  
import java.util.HashMap;  
  
public class UserNoticeListener implements ChannelAwareMessageListener {  
 private static ObjectMapper *MAPPER* = new ObjectMapper();  
  
 @Override  
 public void onMessage(Message message, Channel channel) throws Exception {  
 //获取用户的ID，通过队列名称获取  
 String queueName = message.getMessageProperties().getConsumerQueue();  
 String userId = queueName.substring(queueName.lastIndexOf("\_")+1);  
  
 io.netty.channel.Channel wsChannel = MyWebSocketHandler.*userChannelMap*.get(userId);  
 //判断用户是否在线  
 if(wsChannel != null){  
 //如果连接不为空，表示用户在线  
 //封装返回数据  
 HashMap countMap = new HashMap();  
 countMap.put("userNoticeCount", 1);  
 Result result = new Result(true, StatusCode.*OK*, "监听查询成功", countMap);  
  
 //将数据通过WebSocket连接主动推送个用户  
 wsChannel.writeAndFlush(new TextWebSocketFrame(*MAPPER*.writeValueAsString(result)));  
 }  
 }  
}

1. 修改通知微服务的RabbitConfig监听器容器配置

添加点赞监听器到容器中，即注册Bean到Spring容器中。

package com.tensquare.notice.config;  
  
import com.tensquare.notice.listener.SysNoticeListener;  
import com.tensquare.notice.listener.UserNoticeListener;  
import org.springframework.amqp.rabbit.connection.ConnectionFactory;  
import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class RabbitConfig {  
 @Bean("sysNoticeContainer")  
 public SimpleMessageListenerContainer createSys(ConnectionFactory connectionFactory){  
 SimpleMessageListenerContainer container = new SimpleMessageListenerContainer(connectionFactory);  
 //使用channel监听  
 container.setExposeListenerChannel(true);  
 //设置自己编写的监听器  
 container.setMessageListener(new SysNoticeListener());  
 return container;  
 }  
  
 @Bean("userNoticeContainer")  
 public SimpleMessageListenerContainer createUser(ConnectionFactory connectionFactory){  
 SimpleMessageListenerContainer container = new SimpleMessageListenerContainer(connectionFactory);  
 //使用channel监听  
 container.setExposeListenerChannel(true);  
 //设置自己编写的监听器  
 container.setMessageListener(new UserNoticeListener());  
 return container;  
 }  
}

1. 修改MyWebSocketHandler类

修改WebSocket业务逻辑，首先从Spring容器中获取一个点赞消息监听器容器，处理点赞消息。然后与订阅类消息一样，获取点赞类消息数量。同样的，封装消息数量到map中，清空该消息队列，最后将该点赞类消息队列添加到点赞消息监听器容器中，实时监控。

package com.tensquare.notice.netty;  
  
import com.fasterxml.jackson.databind.ObjectMapper;  
import com.tensquare.notice.config.ApplicationContextProvider;  
import entity.Result;  
import entity.StatusCode;  
import io.netty.channel.Channel;  
import io.netty.channel.ChannelHandlerContext;  
import io.netty.channel.SimpleChannelInboundHandler;  
import io.netty.handler.codec.http.websocketx.TextWebSocketFrame;  
import org.springframework.amqp.rabbit.core.RabbitAdmin;  
import org.springframework.amqp.rabbit.core.RabbitTemplate;  
import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;  
  
import java.util.HashMap;  
import java.util.Properties;  
import java.util.concurrent.ConcurrentHashMap;  
  
public class MyWebSocketHandler extends SimpleChannelInboundHandler<TextWebSocketFrame> {  
 //数据解析工具类  
 private static ObjectMapper *MAPPER* = new ObjectMapper();  
 //存放WebSocket连接的Map，根据用户ID存放  
 public static ConcurrentHashMap<String, Channel> *userChannelMap* = new ConcurrentHashMap<>();  
 //从Spring容器中获取RabbitTemplate  
 RabbitTemplate rabbitTemplate = ApplicationContextProvider.*getApplicationContext*()  
 .getBean(RabbitTemplate.class);  
 //从Spring容器中获取消息监听器容器，处理订阅消息sysNotice  
 SimpleMessageListenerContainer sysNoticeContainer = (SimpleMessageListenerContainer) ApplicationContextProvider.*getApplicationContext*()  
 .getBean("sysNoticeContainer");  
 //从Spring容器中获取消息监听器容器，处理点赞消息userNotice  
 SimpleMessageListenerContainer userNoticeContainer = (SimpleMessageListenerContainer) ApplicationContextProvider.*getApplicationContext*()  
 .getBean("userNoticeContainer");  
 //用户请求WebSocket服务端时执行的方法  
 @Override  
 protected void channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg) throws Exception {  
 //约定用户第一次请求携带的数据{"userId":"1"}  
 //获取用户数据并解析  
 String json = msg.text();  
 //解析json数据，获取用户ID  
 String userId = *MAPPER*.readTree(json).get("userId").asText();  
  
 //第一次请求时，建立WebSocket连接  
 Channel channel = *userChannelMap*.get(userId);  
 if(channel == null){  
 //获取WebSocket的连接  
 channel = ctx.channel();  
 //将连接放到Map中  
 *userChannelMap*.put(userId, channel);  
 }  
  
 //获取Rabbitmq的消息内容，并发送给用户  
 //只完成新消息的题型，获取消息的数量  
 RabbitAdmin rabbitAdmin = new RabbitAdmin(rabbitTemplate);  
 //拼接获取队列名称  
 String queueName = "article\_subscribe\_" + userId;  
 //获取Rabbit的Properties容器  
 Properties queueProperties = rabbitAdmin.getQueueProperties(queueName);  
 //获取消息数量  
 int noticeCount = 0;  
 if(queueProperties != null){  
 noticeCount = (int)queueProperties.get("QUEUE\_MESSAGE\_COUNT");  
 }  
  
 //--------------以上是订阅类消息，以下是点赞类消息-------------------------------  
 //拼接获取队列名称  
 String userQueueName = "article\_thumbup\_" + userId;  
 //获取Rabbit的Properties容器  
 Properties userQueueProperties = rabbitAdmin.getQueueProperties(userQueueName);  
 //获取消息数量  
 int userNoticeCount = 0;  
 if(userQueueProperties != null){  
 userNoticeCount = (int)userQueueProperties.get("QUEUE\_MESSAGE\_COUNT");  
 }  
  
 //封装返回的数据  
 HashMap countMap = new HashMap();  
 //订阅类消息数量  
 countMap.put("sysNoticeCount", noticeCount);  
 //点赞类消息数量  
 countMap.put("userNoticeCount", userNoticeCount);  
 Result result = new Result(true, StatusCode.*OK*, "查询成功了吗", countMap);  
  
 //将数据发送给用户  
 channel.writeAndFlush(new TextWebSocketFrame(*MAPPER*.writeValueAsString(result)));  
  
 //用户建立连接后，将该消息队列清空，否则会被MQ监听到再消费一次  
 if(noticeCount > 0){  
 rabbitAdmin.purgeQueue(queueName, true);  
 }  
 if(userNoticeCount > 0){  
 rabbitAdmin.purgeQueue(userQueueName, true);  
 }  
  
 //为用户的消息通知队列注册监听器，当用户在线时，一旦有消息，主动推送个用户，不需要用户请求  
 sysNoticeContainer.addQueueNames(queueName);  
 userNoticeContainer.addQueueNames(userQueueName);  
 }  
}

1. 测试点赞功能

首先，添加新的文章，测试新文章消息通知是否成功，然后在mysql中查找该新增文章的ID号，通过该文章ID号测试点赞消息功能，如果点赞后，Rabbitmq中出现了该文章作者ID的thumbup点赞队列，且其中的消息增加了一个，说明点赞消息测试成功。