# 使用RabbitMQ+WebSocket实现消息推送过程

介绍：

HTML5定义了WebSocket，它能够实现浏览器与服务器之间全双工通信；其优点有两个：

一是服务器与客户端之间交换的标头信息很少；二是服务器可以主动传送数据给客户端；

选用RabbitMQ作为服务器消息队列，是因为其有丰富的第三方插件，用户可以在AMQP协议的基础上自己扩展应用；针对WebSocket通信RabbitMQ提供了Web STOMP插件，是一个实现了STOMP协议的插件；可以理解为WebSocket与STOMP协议间的桥接，目的是让浏览器能够使用RabbitMQ，当RabbitMQ启用了Web STOMP插件后，浏览器就可以使用WebSocket与之通信了；当有新消息需要发布时，系统后台将消息数据发送到RabbitMQ中，再通过WebSocket将数据推送给浏览器；

这里是属于一个生产者，多个消费者的模式；

|  |
| --- |
| 1、启动或关闭RabbitMQ服务命令  启动（默认端口15672）：net start RabbitMQ  关闭：net stop RabbitMQ  2、启动stomp服务命令  进入mq安装目录C:\Program Files\RabbitMQ Server\rabbitmq\_server-3.6.5\sbin\  启动（默认端口15674）：rabbitmq-plugins enable rabbitmq\_web\_stomp  关闭：rabbitmq-plugins disable rabbitmq\_web\_stomp |

以上2是启用Web STOMP插件

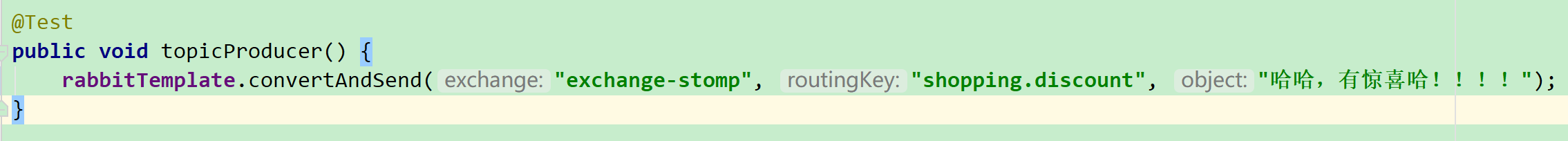
前端接收推送过来的消息代码：

|  |
| --- |
| <!DOCTYPE **html**> <**html**> <**head**>  <**meta charset="UTF-8"** />  <**title**>消息提醒示例</**title**>  <**link rel="icon" href="data:image/ico;base64,aWNv"**> </**head**> <**noscript**>  <**h2 style="color**:**#ff0000"**>貌似你的浏览器不支持websocket</**h2**> </**noscript**> <**link rel="stylesheet" type="text/css" href="../static/css/default.css"**> <**link rel="stylesheet" type="text/css" href="../static/css/jquery.notify.css"**>   <**script src="https://cdn.bootcss.com/stomp.js/2.3.3/stomp.min.js"**></**script**> <**script src="https://cdn.bootcss.com/jquery/3.3.1/jquery.min.js"**></**script**> <**script type="text/javascript" src="../static/js/jquery.notify.js"**></**script**> <**script type="text/javascript"**>   **$**.notifySetup({**sound**: **'../static/audio/notify.wav'**});   *// 初始化 ws 对象* **var *ws*** = **new *WebSocket***(**'ws://192.168.101.118:15674/ws'**);   *// 获得Stomp client对象* **var *client*** = Stomp.over(***ws***);   *// SockJS does not support heart-beat: disable heart-beats* ***client***.heartbeat.**outgoing** = 1000;   ***client***.heartbeat.**incoming** = 1000;    *// 定义连接成功回调函数* **var** *on\_connect* = **function**(x) {   *//data.body是接收到的数据* ***client***.subscribe(**"/exchange/exchange-stomp/shopping.discount"**, **function**(data) {   **var** msg = data.**body**;   **$**(**"<p>"**+data.**body**+**"</p>"**).notify({**stay**: 20000});  *// alert("收到数据：" + msg);* });   };   *// 定义错误时回调函数* **var** *on\_error* = **function**(msg) {   **$**(**"<p>服务器错误："**+msg+**"</p>"**).notify(**"error"**);   };   *// 连接RabbitMQ* ***client***.connect(**'admin'**, **'admin'**, *on\_connect*, *on\_error*, **'/'**);   ***console***.log(**">>>连接上http://192.168.101.118:15672"**);  </**script**> </**body**> </**html**> |

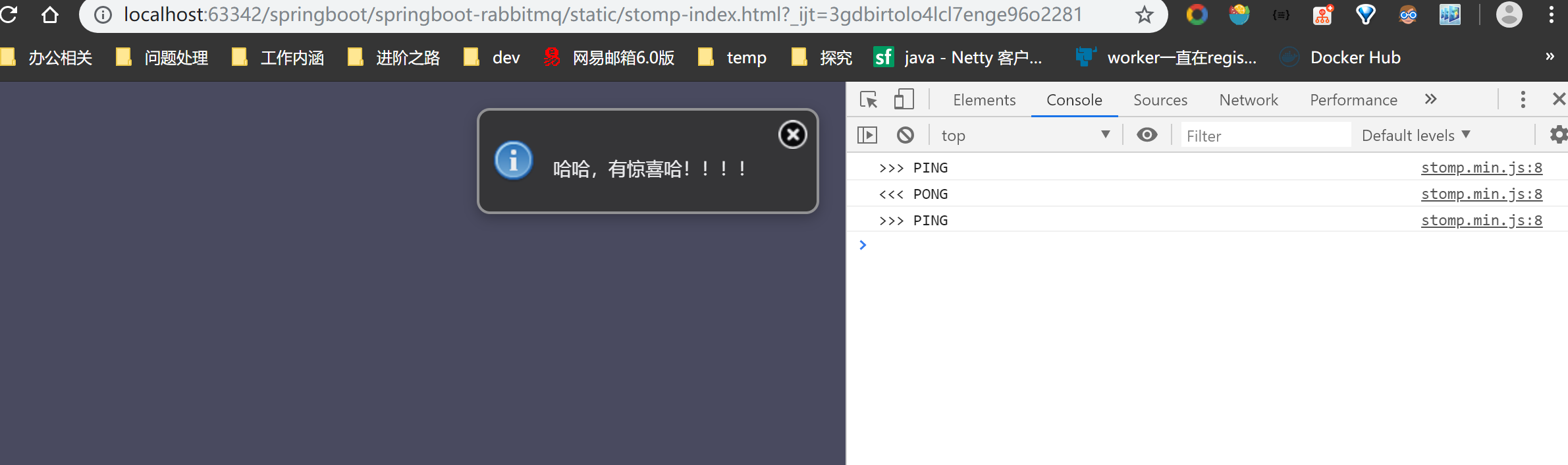
后端定义Exchange和路由键并绑定到队列：

|  |
| --- |
| @Configuration **public class** TopicConfig {   @Bean  **public** Queue topicQueueOne() {  **return** QueueBuilder.*durable*(**"topic.queue.one"**).build();  }   @Bean  **public** Queue topicQueueTwo() {  **return** QueueBuilder.*durable*(**"topic.queue.two"**).build();  }   @Bean  **public** TopicExchange topicExchange() {  **return** (TopicExchange) ExchangeBuilder.*topicExchange*(**"exchange-stomp"**).build();  }   @Bean  **public** Binding topicBindingOne() {  **return** BindingBuilder.*bind*(topicQueueOne()).to(topicExchange()).with(**"shopping.discount"**);  }   @Bean  **public** Binding topicBindingTwo() {  **return** BindingBuilder.*bind*(topicQueueTwo()).to(topicExchange()).with(**"#"**);  } } |

服务端生产消息：



前端会受到推送过来的消息如下：



# RabbitMQ实践建议

## 虚拟主机：

1.Vhost是AMQP协议里的一个基本概念；

2.虚拟主机就是一台缩小版的RabbitMQ服务器，其内部有自己的队列，交换器，绑定等；特别的是RabbitMQ中的权限控制是以vhost为单位的；也就是说，消息客户端在访问时不能把vhost A中的交换器绑定到vhost B的队列中；所以，实际场景中用一台RabbitMQ服务器服务多个不同的应用，应用间通过不同虚拟主机的划分提供访问消息时逻辑上的隔离，从而为应用程序提供安全访问；

3.RabbitMQ有一个默认的vhost，它的值时“/”；

#创建虚拟主机

rabbitmqctl add\_vhost test

#删除虚拟主机

rabbitmqctl delete\_vhost test

#查询当前RabbitMQ服务器中的所有虚拟机

rabbitmqctl list\_vhosts

## 消息保存：

对于Queue的保存方式有disk和RAM两种；

两种情况下会将消息写入磁盘，一是在发布消息时指明需要写入磁盘；二是当消息服务器内存紧张时会将部分内存中的消息转移到磁盘；采用disk方式，消息数据会被保存在以.rdq后缀命名的文件中；当文件达到一定大小（默认是16MB）时会生成一个新的文件，当文件中的已经被删除的消息比例大于阈值时会触发文件合并操作；而采用RAM方式，只是在RAM中保存内部数据库表数据，而不会保存消息、消息存储索引、队列索引和其他节点状态等数据，所以必须在启动时从集群中的其他节点同步原来的消息数据，这也就意味着集群中必须包含至少一个disk方式的节点；正因为这样，所以不能手工删除集群中的最后一个disk节点；

## 消息确认模式：

设置生产者确认模式有三种：

普通确认：每发送完一条消息后，就调用waitForConfirms方法等待Broker的确认消息，本质上这就是串行方式确认；

批量确认：每发送完一批消息后，再调用waitForConfirms方法等待Broker的确认；

异步确认：通过调用addConfirmListener方法注册回调，在Broker确认了一条或多条消息之后由客户端回调该方法；

## 消费者应答：

消息生产者有时候不仅关心消息有没有送到消息服务器，还会关心消息消费者的消费是成功的还是失败的，这里就是 消费者回执；

自动回执；

自动回执模式下，当Broker成功发送消息给消费者后就会立即把此消息从队列中删除，而不用等待消费者回送确认消息；

手动回执；

手动回执模式下，当Broker发送消息给消费者后并不会立即把此消息删除，而是要等收到消费者回送的确认消息后才会删除；

## 流控机制：