第6章 消息中间件RabbitMQ

学习目标：

- 理解RabbitMQ的主要概念，完成RabbitMQ的安装
- 理解直接模式和分列模式，说出直接模式和分列模式在电商系统的应用场景
- 完成用户注册，能够将消息发送给RabbitMQ
- 完成短信服务，能够接收消息并调用阿里云通信完成短信发送

1. 走进RabbitMQ

1.1消息中间件简介

消息中间件（消息队列）是分布式系统中重要的组件，主要解决应用耦合，异步消息，流量削锋等问题实现高性能，高可用，可伸缩和最终一致性[架构] 使用较多的消息队列有ActiveMQ，RabbitMQ，ZeroMQ，Kafka，MetaMQ，RocketMQ

以下介绍消息队列在实际应用中常用的使用场景：异步处理，应用解耦，流量削锋和消息通讯四个场景

1.2什么是RabbitMQ

RabbitMQ 是一个由 Erlang 语言开发的 AMQP 的开源实现。

AMQP ：Advanced Message Queue，高级消息队列协议。它是应用层协议的一个开放标准，为面向消息的中间件设计，基于此协议的客户端与消息中间件可传递消息，并不受产品、开发语言等条件的限制。

RabbitMQ 最初起源于金融系统，用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。具体特点包括：

1.可靠性（Reliability）

RabbitMQ 使用一些机制来保证可靠性，如持久化、传输确认、发布确认。

2.灵活的路由（Flexible Routing）

在消息进入队列之前，通过 Exchange 来路由消息的。对于典型的路由功能，RabbitMQ 已经提供了一些内置的 Exchange 来实现。针对更复杂的路由功能，可以将多个 Exchange 绑定在一起，也通过插件机制实现自己的 Exchange 。

3.消息集群（Clustering）

多个 RabbitMQ 服务器可以组成一个集群，形成一个逻辑 Broker 。

4.高可用（Highly Available Queues）

队列可以在集群中的机器上进行镜像，使得在部分节点出问题的情况下队列仍然可用。

5.多种协议（Multi-protocol）

RabbitMQ 支持多种消息队列协议，比如 STOMP、MQTT 等等。

6.多语言客户端（Many Clients）

RabbitMQ 几乎支持所有常用语言，比如 Java、.NET、Ruby 等等。

7.管理界面（Management UI）

RabbitMQ 提供了一个易用的用户界面，使得用户可以监控和管理消息 Broker 的许多方面。

8.跟踪机制（Tracing）

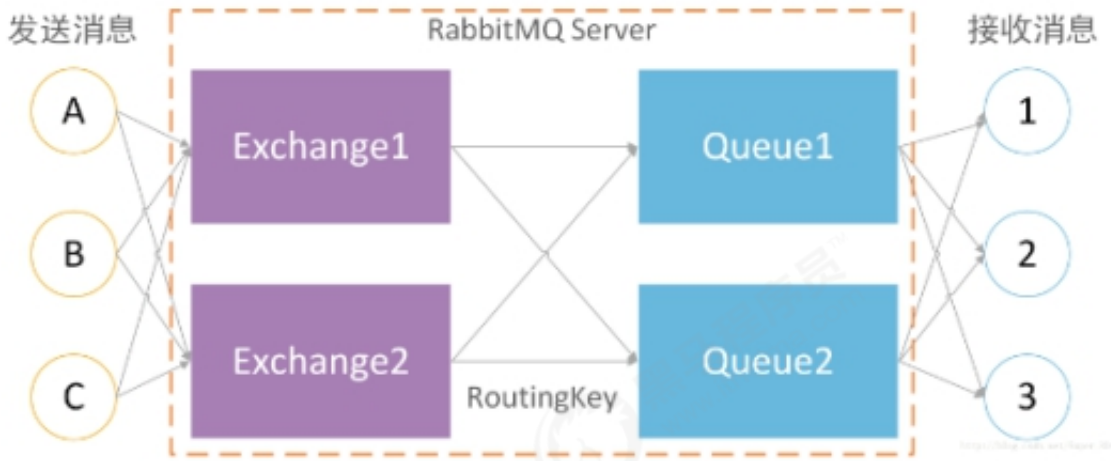
如果消息异常，RabbitMQ 提供了消息跟踪机制，使用者可以找出发生了什么。

9.插件机制（Plugin System）

RabbitMQ 提供了许多插件，来从多方面进行扩展，也可以编写自己的插件。

1.3架构图与主要概念

1.3.1架构图



1.3.2主要概念

RabbitMQ Server： 也叫broker server，它是一种传输服务。 他的角色就是维护一条从Producer到Consumer的路线，保证数据能够按照指定的方式进行传输。

Producer： 消息生产者，如图A、B、C，数据的发送方。消息生产者连接RabbitMQ服务器然后将消息投递到Exchange。

Consumer：消息消费者，如图1、2、3，数据的接收方。消息消费者订阅队列，RabbitMQ将Queue中的消息发送到消息消费者。

Exchange：生产者将消息发送到Exchange（交换器），由Exchange将消息路由到一个或多个Queue中（或者丢弃）。Exchange并不存储消息。RabbitMQ中的Exchange有direct、fanout、topic、headers四种类型，每种类型对应不同的路由规则。

Queue：（队列）是RabbitMQ的内部对象，用于存储消息。消息消费者就是通过订阅队列来获取消息的，RabbitMQ中的消息都只能存储在Queue中，生产者生产消息并最终投递到Queue中，消费者可以从Queue中获取消息并消费。多个消费者可以订阅同一个Queue，这时Queue中的消息会被平均分摊给多个消费者进行处理，而不是每个消费者都收到所有的消息并处理。

RoutingKey：生产者在将消息发送给Exchange的时候，一般会指定一个routing key，来指定这个消息的路由规则，而这个routing key需要与Exchange Type及binding key联合使用才能最终生效。在Exchange Type与binding key固定的情况下（在正常使用时一般这些内容都是固定配置好的），我们的生产者就可以在发送消息给Exchange时，通过指定routing key来决定消息流向哪里。RabbitMQ为routing key设定的长度限制为255 bytes。

Connection：（连接）：Producer和Consumer都是通过TCP连接到RabbitMQ Server的。以后我们可以看到，程序的起始处就是建立这个TCP连接。

Channels：（信道）：它建立在上述的TCP连接中。数据流动都是在Channel中进行的。也就是说，一般情况是程序起始建立TCP连接，第二步就是建立这个Channel。

VirtualHost：权限控制的基本单位，一个VirtualHost里面有若干Exchange和MessageQueue，以及指定被哪些user使用

1.4 RabbitMQ安装与启动

（1）下载并安装 Eralng

配套软件中已提供otp_win64_20.2.exe

（2）下载并安装rabbitmq

配套软件中已提供rabbitmq-server-3.7.4.exe。双击安装，注意不要安装在包含中文和空格的目录下！安装后window服务中就存在rabbitMQ了，并且是启动状态。

（3）安装管理界面（插件）

进入rabbitMQ安装目录的sbin目录，输入命令

```
rabbitmq-plugins enable rabbitmq_management
```

（4）重新启动服务

（5）打开浏览器，地址栏输入http://127.0.0.1:15672 ,即可看到管理界面的登陆页

输入用户名和密码，都为guest 进入主界面：

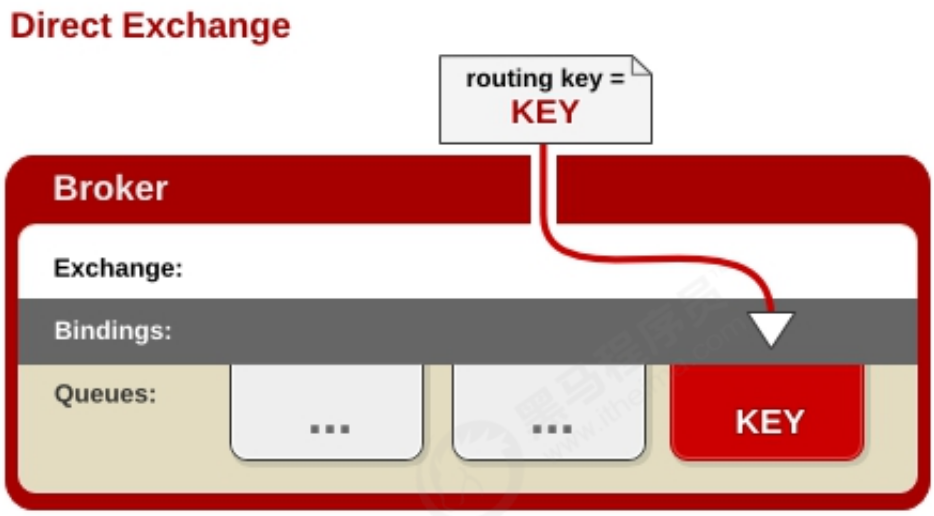
最上侧的导航以此是：概览、连接、信道、交换器、队列、用户管理

2. RabbitMQ发送与接收消息

2.1 直接模式（Direct）

2.1.1 什么是Direct模式

我们需要将消息发给唯一一个节点时使用这种模式，这是最简单的一种形式。



任何发送到Direct Exchange的消息都会被转发到RouteKey中指定的Queue。

1.一般情况可以使用rabbitMQ自带的Exchange：""(该Exchange的名字为空字符串，下文称其为default Exchange)。

2.这种模式下可以不需要将Exchange进行任何绑定(binding)操作

3.消息传递时需要一个“RouteKey”，可以简单的理解为要发送到的队列名字。

4.如果vhost中不存在RouteKey中指定的队列名，则该消息会被抛弃。

2.1.2 创建队列

创建队列，名为queue.test

2.1.3 代码实现-消息生产者

（1）创建工程rabbitmq_demo，引入依赖，pom.xml如下：

```
<dependency>
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit</artifactId>
    <version>2.1.4.RELEASE</version>
</dependency>
```

（2）编写配置文件applicationContext-rabbitmq-producer.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:rabbit="http://www.springframework.org/schema/rabbit"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/rabbit
http://www.springframework.org/schema/rabbit/spring-rabbit.xsd">
    <!--连接工厂-->
    <rabbit:connection-factory id="connectionFactory" host="127.0.0.1"
port="5672" username="guest" password="guest" />
    <rabbit:template id="rabbitTemplate" connection-
factory="connectionFactory" />
</beans>
```

（3）编写测试代码

```
ApplicationContext context=new
ClassPathXmlApplicationContext("applicationContext-rabbitmq-
producer.xml");
RabbitTemplate rabbitTemplate=
(RabbitTemplate)context.getBean("rabbitTemplate");
rabbitTemplate.convertAndSend("exchange.direct_test ","test","直接模式");
((ClassPathXmlApplicationContext) context).close();
```

执行后观察控制台。

2.1.4 代码实现-消息消费者

（1）编写消息监听类

```
public class MessageConsumer implements MessageListener {
    public void onMessage(Message message) {
        System.out.println("接收到消息： " +new String(message.getBody())
);
    }
}
```

（2）创建配置文件applicationContext-rabbitmq-consumer.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:rabbit="http://www.springframework.org/schema/rabbit"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/rabbit
http://www.springframework.org/schema/rabbit/spring-rabbit.xsd">
    <!--连接工厂-->
    <rabbit:connection-factory id="connectionFactory" host="127.0.0.1"
port="5672" username="guest" password="guest"/>
    <!--队列-->
    <rabbit:queue name="queue.test" />
    <!--消费者监听类-->
    <bean id="messageConsumer" class="cn.itcast.demo.MessageConsumer">
</bean>
    <!--设置监听容器-->
    <rabbit:listener-container connection-factory="connectionFactory"
acknowledge="auto" >
        <rabbit:listener queue-names="queue.test" ref="messageConsumer"/>
    </rabbit:listener-container>
</beans>
```

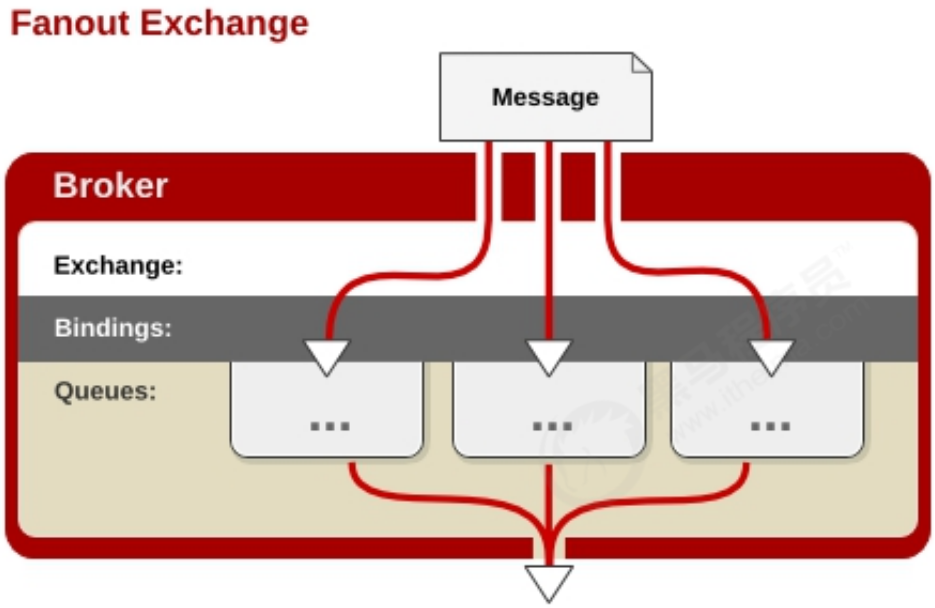
（3）编写测试代码Test2

```
ApplicationContext context=new
ClassPathXmlApplicationContext("applicationContext-rabbitmq-
consumer.xml");
```

2.2 分列模式（Fanout）

2.2.1 什么是分列（Fanout）模式

当我们需要将消息一次发给多个队列时，需要使用这种模式。如下图：



任何发送到Fanout Exchange的消息都会被转发到与该Exchange绑定(Binding)的所有Queue上。

1.这种模式需要提前将Exchange与Queue进行绑定，一个Exchange可以绑定多个Queue，一个Queue可以同多个Exchange进行绑定。

2.这种模式不需要RouteKey

3.如果接受到消息的Exchange没有与任何Queue绑定，则消息会被抛弃。

2.2.2 交换器绑定队列

创建队列queue.test1 和queue.test2

创建交换器exchange.fanout_test，并绑定queue.test1和queue.test2

2.2.3 代码实现-消息生产者

编写代码

```
ApplicationContext context=new
ClassPathXmlApplicationContext("applicationContext-rabbitmq-
producer.xml");
RabbitTemplate rabbitTemplate=
(RabbitTemplate)context.getBean("rabbitTemplate");
rabbitTemplate.convertAndSend("exchange.fanout_test","","分列模式走起");

((ClassPathXmlApplicationContext) context).close();
```

测试代码，发现队列queue.test1 和queue.test2都收到消息

2.2.4 代码实现-消息消费者

（1）将MessageConsumer复制为MessageConsumer1和MessageConsumer2。

（2）修改applicationContext-rabbitmq-consumer.xml，增加配置

```
<rabbit:queue name="queue.test1" />
<rabbit:queue name="queue.test2" />
<bean id="messageConsumer1" class="cn.itcast.demo.MessageConsumer1">
</bean>
<bean id="messageConsumer2" class="cn.itcast.demo.MessageConsumer2">
</bean>
```

（3）修改applicationContext-rabbitmq-consumer.xml的

```
<!--设置监听容器-->
<rabbit:listener-container connection-factory="connectionFactory" >
    <rabbit:listener queue-names="queue.test" ref="messageConsumer"/>
    <rabbit:listener queue-names="queue.test1" ref="messageConsumer1"/>
    <rabbit:listener queue-names="queue.test2" ref="messageConsumer2"/>
</rabbit:listener-container>
```

测试：启动Test2测试

2.1.5 创建队列与交换器（配置方式）

修改配置文件applicationContext-rabbitmq-producer.xml

```
<!--rabbitAdmin 封装管理操作-->
<rabbit:admin connection-factory="connectionFactory"></rabbit:admin>
<!--创建队列-->
<rabbit:queue name="queue.test1"  />
<!--创建队列-->
<rabbit:queue name="queue.test2"  />
<!--创建分发交换器 -->
<rabbit:fanout-exchange name="exchange.fanout_test" >
    <rabbit:bindings>
        <rabbit:binding  queue="queue.test1"></rabbit:binding>
        <rabbit:binding  queue="queue.test2"></rabbit:binding>
    </rabbit:bindings>
</rabbit:fanout-exchange>
```

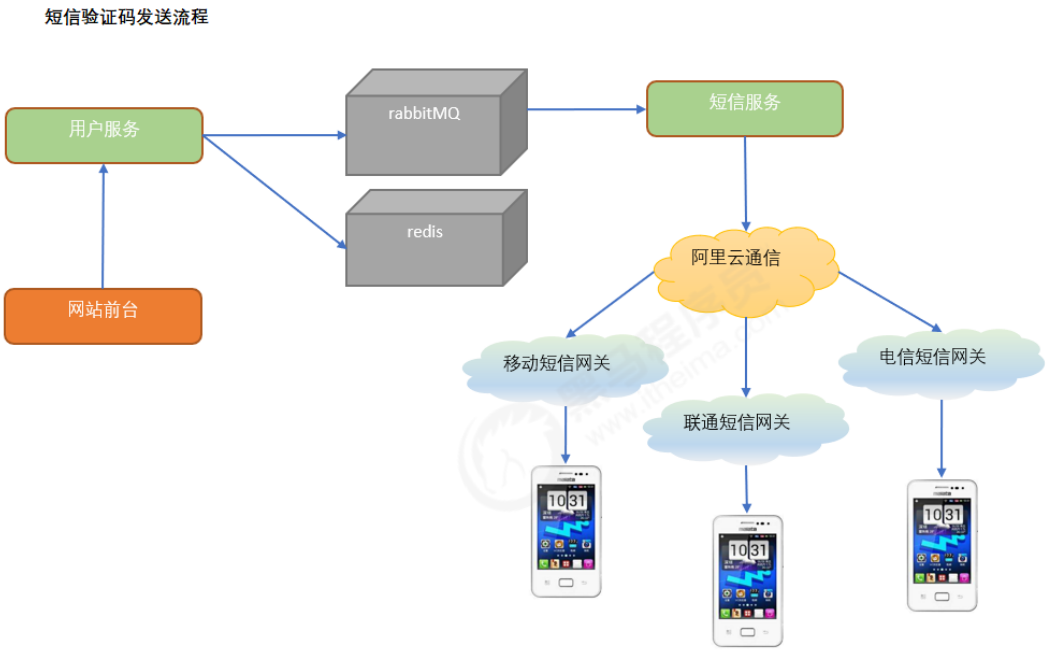
3. 用户注册

3.1 需求分析

注册账号，用手机号注册，填写后发送短信验证码，填写短信验证码正确方可注册成功。

3.2 实现思路

（1）发送短信验证码逻辑：用户服务将要发送的短信验证码发送给rabbitmq和redis，短信服务将消息从rabbitmq中取出并调用阿里云通信发送短信。阿里云通信整合了三大运营商的短信网关，最终把验证码发送到用户的手机上。rabbitmq采用直接模式，用户服务为消息生产者，短信服务为消息消费者。



（2）注册逻辑：注册时从redis中提取短信验证码与用户填写的验证码进行比对，如果一致则可以注册，否则拦截请求。

3.3 后端代码

首先生成用户服务，并添加到当前工程。

3.3.1 发送短信验证码到MQ

实现思路： 在用户服务编写API ,生成手机验证码，存入Redis并发送到RabbitMQ

（1）因为要用到消息队列，所以在用户服务引入rabbit与spring的整合依赖。

```
<dependency>
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit</artifactId>
    <version>2.1.4.RELEASE</version>
</dependency>
```

（2）添加配置文件applicationContext-rabbitmq-producer.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:rabbit="http://www.springframework.org/schema/rabbit"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/rabbit
http://www.springframework.org/schema/rabbit/spring-rabbit.xsd">
    <!--连接工厂-->
    <rabbit:connection-factory id="connectionFactory" host="127.0.0.1"
port="5672" username="guest" password="guest" />
    <rabbit:admin connection-factory="connectionFactory"></rabbit:admin>
    <!--创建队列-->
    <rabbit:queue name="queue.sms" />
    <rabbit:template id="rabbitTemplate" connection-
factory="connectionFactory" />
</beans>
```

（3）在UserService中新增方法定义

```
    /**
     * 发送短信验证码
     * @param mobile
     */
    public void sendSms(String phone);
```

（4）UserServiceImpl方法实现

```
    @Autowired
    private RedisTemplate redisTemplate;

    @Autowired
    private RabbitTemplate rabbitTemplate;

    /**
     * 发送短信验证码
     * @param phone
     */
    public void sendSms(String phone){
        //1.得到六位短信验证码
        int max=999999;
        int min=100000;
        Random random = new Random();
        int code = random.nextInt(max);
        if(code<min){
            code=code+min;
        }
        System.out.println("短信验证码: "+code);
        //2.保存到redis里
        redisTemplate.boundValueOps("code_"+phone).set(code+"");
        redisTemplate.boundValueOps("code_"+phone).expire(5,
TimeUnit.MINUTES);//5分钟失效
        //3.发送给RabbitMQ
        Map<String,String> map=new HashMap();
        map.put("phone", phone);
        map.put("code", code+"");
        rabbitTemplate.convertAndSend("","queue.sms",
JSON.toJSONString(map));
    }
```

（5）qingcheng_web_portal新增UserController

```
@RestController
@RequestMapping("/user")
public class UserController {

    @Reference
    private UserService userService;

    /**
     * 发送短信验证码
     * @param phone
     */
    @GetMapping(value="/sendSms")
    public Result sendSms(String phone){
        userService.sendSms(phone);
        return new Result();
    }
}
```

3.3.2 短信服务接收消息

短信发送是由单独的短信服务提供的功能，所有的短信都是先发送到消息队列，短信服务从消息队列中提取手机号和验证码，调用短信发送接口进行发送短信。

我们这个环节实现的是将手机号和验证码从消息队列中提取出来，打印到控制台上。

（1）创建qingcheng_service_sms工程，pom文件引入依赖

```
<dependency>
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit</artifactId>
    <version>2.1.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>com.qingcheng</groupId>
    <artifactId>qingcheng_common</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

（2）添加web.xml，参见其它服务工程

（3）添加监听实现类SmsMessageConsumer

```
public class SmsMessageConsumer implements MessageListener {

    public void onMessage(Message message) {
        String jsonString = new String(message.getBody());
        Map<String,String> map = JSON.parseObject(jsonString, Map.class);
        String phone = map.get("phone");
        String code=map.get("code");
        System.out.println("手机号："+phone+"验证码："+code);
    }
}
```

（4）添加配置文件applicationContext-rabbitmq-consumer.xml

```
<!--连接工厂-->
<rabbit:connection-factory id="connectionFactory" host="127.0.0.1"
port="5672" username="guest" password="guest" />
<!--创建队列-->
<rabbit:queue name="queue.sms" />
<!--消费者监听类-->
<bean id="messageConsumer"
class="com.qingcheng.consumer.SmsMessageConsumer"></bean>
<!--设置监听容器-->
<rabbit:listener-container connection-factory="connectionFactory" >
    <rabbit:listener queue-names="queue.sms" ref="messageConsumer"/>
</rabbit:listener-container>
```

3.3.3 用户注册

（1）UserService增加方法定义

```
    /**
     * 增加
     * @param user
     * @param smsCode
     */
    public void add(User user,String smsCode);
```

（2）UserServiceImpl实现方法

```
    /**
     * 增加
     * @param user
     * @param smsCode
     */
    public void add(User user,String smsCode) {
        //比较短信验证码
        //获取系统短信验证码
        String sysCode= (String)
redisTemplate.boundValueOps("code_"+user.getPhone()).get();
        if(sysCode==null){
            throw new RuntimeException("验证码未发送或已过期");
        }
        if(!smsCode.equals(sysCode)){
            throw new RuntimeException("验证码不正确");
        }
        if(user.getUsername()==null){
            user.setUsername(user.getPhone());
        }
        User searchUser=new User();
        searchUser.setUsername(user.getUsername());
        if(userMapper.selectCount(searchUser)>0) {  //查询是否存在相同记录
            throw new RuntimeException("该手机号已注册");
        }
        user.setCreated(new Date());
        user.setUpdated(new Date());
        user.setPoints(0);//积分初始值为0
        user.setStatus("1");//状态1
        user.setIsEmailCheck("0");//邮箱认证
        user.setIsMobileCheck("1");//手机认证
        userMapper.insert(user);
    }
```

（3）UserController增加方法

```
    @PostMapping("/save")
    public Result save(@RequestBody User user , String smsCode ){
        //密码加密
        BCryptPasswordEncoder encoder=new BCryptPasswordEncoder();
        String newpassword = encoder.encode(user.getPassword());
        user.setPassword(newpassword);

        userService.add(user,smsCode);
        return new Result();
    }
```

3.4 前端代码

（1）将register.html拷贝到qingcheng_web_portal的webapp下，在body中放置div

```
<div id="app">
...
</div>
```

（2）表单部分代码绑定

```
<div class="control-group">
    <label class="control-label">手机号：</label>
    <div class="controls">
        <input type="text" placeholder="请输入你的手机号" v-
model="pojo.phone" class="input-xfat input-xlarge">
    </div>
</div>
<div class="control-group">
    <label  class="control-label">验证码：</label>
    <div class="controls">
        <input type="text" placeholder="验证码" v-model="smsCode"
class="input-xfat input-xlarge">
        <a  @click="sendSms()" >发送短信验证码</a>
    </div>
</div>
<div class="control-group">
    <label  class="control-label">登录密码：</label>
    <div class="controls">
        <input type="password" placeholder="设置登录密码" v-
model="pojo.password" class="input-xfat input-xlarge">
    </div>
</div>
<div class="control-group">
    <label  class="control-label">确认密码：</label>
    <div class="controls">
        <input type="password" placeholder="再次确认密码" v-
model="password"  class="input-xfat input-xlarge">
    </div>
</div>
<div class="control-group">
    <label  class="control-label">      
</label>
    <div class="controls">
        <input name="m1" type="checkbox" value="2" checked=""><span>同意协
议并注册《品优购用户协议》</span>
    </div>
</div>
<div class="control-group">
    <label class="control-label"></label>

    <div class="controls btn-reg">
```

```
        <a class="sui-btn btn-block btn-xlarge btn-danger"
@click="save()" >完成注册</a>
    </div>
</div>
```

（3）编写js部分代码

```
<script src="/js/vue.js"></script>
<script src="/js/axios.js"></script>
<script>
    var vue=new Vue({
        el: '#app',
        data(){
            return {
                pojo: {},
                smsCode:"",
                password:""
            }
        },
        methods:{
            sendSms(){
                axios.get(`/user/sendSms.do?
phone=${this.pojo.phone}`).then(response => {
                    alert(response.data.message);
                });
            },
            save (){
                if(this.password!=this.pojo.password){
                    alert("两次输入密码不一致");
                    return ;
                }
                axios.post(`/user/save.do?
smsCode=${this.smsCode}`,this.pojo).then(response => {
                    alert(response.data.message);
                    this.pojo={};
                    this.smsCode="";
                    this.password="";
                });
            }
        }
    })
</script>
```

4. 阿里云通信（了解）

4.1 阿里云通信简介

阿里云通信（原名阿里大于），是阿里云旗下产品，融合了三大运营商的通信能力，通过将传统通信业务和能力与互联网相结合，创新融合阿里巴巴生态内容，全力为中小企业和开发者提供优质服务阿里大于提供包括短信、语音、流量直充、私密专线、店铺手机号等个性化服务。通过阿里大于打通三大运营商通信能力，全面融合阿里巴巴生态，以开放API及SDK的方式向开发者提供通信和数据服务，更好地支撑企业业务发展和创新服务。

4.2 发短信前你要准备什么

（1）在阿里云官网 https://www.aliyun.com/ 注册账号

（2）手机下载”阿里云“APP，完成实名认证

（3）登陆阿里云，产品中选择”短信服务“

（4）申请签名（选择验证码类型）

（5）申请模板

（6）创建 accessKey （注意保密！）

（7）充值 （没必要充太多，1至2元足矣，土豪请随意~）

4.3 快速入门

（1）创建工程引入依赖

```
<dependency>
  <groupId>com.aliyun</groupId>
  <artifactId>aliyun-java-sdk-core</artifactId>
  <version>4.0.3</version>
</dependency>
```

（2）创建测试类，以下代码从官网获取

```
import com.aliyuncs.CommonRequest;
import com.aliyuncs.CommonResponse;
import com.aliyuncs.DefaultAcsClient;
import com.aliyuncs.IAcsClient;
import com.aliyuncs.exceptions.ClientException;
import com.aliyuncs.exceptions.ServerException;
import com.aliyuncs.http.MethodType;
import com.aliyuncs.profile.DefaultProfile;
/*
pom.xml
<dependency>
  <groupId>com.aliyun</groupId>
  <artifactId>aliyun-java-sdk-core</artifactId>
  <version>4.0.3</version>
</dependency>
*/
public class CommonRpc {
    public static void main(String[] args) {
        DefaultProfile profile = DefaultProfile.getProfile("cn-hangzhou",
"*******", "********");
        IAcsClient client = new DefaultAcsClient(profile);
        CommonRequest request = new CommonRequest();
        //request.setProtocol(ProtocolType.HTTPS);
        request.setMethod(MethodType.POST);
        request.setDomain("dysmsapi.aliyuncs.com");
        request.setVersion("2017-05-25");
        request.setAction("SendSms");
        request.putQueryParameter("RegionId", "cn-hangzhou");
        request.putQueryParameter("PhoneNumbers", "177********");
        request.putQueryParameter("SignName", "青橙");
        request.putQueryParameter("TemplateCode", "SMS_165116876");
        request.putQueryParameter("TemplateParam", "
{\"code\":\"123123\"}");
        try {
            CommonResponse response = client.getCommonResponse(request);
            System.out.println(response.getData());
        } catch (ServerException e) {
            e.printStackTrace();
        } catch (ClientException e) {

            e.printStackTrace();
```

```
        }
    }
}
```

4.4 短信服务发送短信

（1）短信服务pom.xml引入阿里云sdk依赖

```
<dependency>
  <groupId>com.aliyun</groupId>
  <artifactId>aliyun-java-sdk-core</artifactId>
  <version>4.0.3</version>
</dependency>
```

（2）添加配置文件sms.properties

```
accessKeyId=******
accessKeySecret=*******
smsCode=SMS_165116876
param={"code":"[value]"}
```

（3）添加配置文件applicationContext-sms.xml

```
<context:component-scan base-package="com.qingcheng"></context:component-scan>
```

（4）创建短信工具类SmsUtil

```java
@Component
public class SmsUtil {

    @Value("${accessKeyId}")
    private String accessKeyId;

    @Value("${accessKeySecret}")
    private String accessKeySecret;

    public CommonResponse sendSms(String phone,String smsCode,String
param){
        DefaultProfile profile = DefaultProfile.getProfile("cn-hangzhou",
accessKeyId, accessKeySecret);
        IAcsClient client = new DefaultAcsClient(profile);

        CommonRequest request = new CommonRequest();
        //request.setProtocol(ProtocolType.HTTPS);
        request.setMethod(MethodType.POST);
        request.setDomain("dysmsapi.aliyuncs.com");
        request.setVersion("2017-05-25");
        request.setAction("SendSms");
        request.putQueryParameter("RegionId", "cn-hangzhou");
        request.putQueryParameter("PhoneNumbers", phone);
        request.putQueryParameter("SignName", "青橙");
        request.putQueryParameter("TemplateCode", smsCode);
        request.putQueryParameter("TemplateParam", param);
        try {
            CommonResponse response = client.getCommonResponse(request);
            System.out.println(response.getData());
            return response;
        } catch (ServerException e) {
            e.printStackTrace();
            return null;
        } catch (ClientException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

（5）修改消息监听类，完成短信发送

```
public class SmsMessageConsumer implements MessageListener {

    @Autowired
    private SmsUtil smsUtil;

    @Value("${smsCode}")
    private String smsCode;//短信模板编号

    @Value("${param}")
    private String param;//短信参数

    public void onMessage(Message message) {
        String jsonString = new String(message.getBody());
        Map<String,String> map = JSON.parseObject(jsonString, Map.class);
        String phone = map.get("phone");
        String code=map.get("code");
        System.out.println("手机号："+phone+"验证码："+code);
        String param=  templateParam_smscode.replace("[value]",code);
        try {
            SendSmsResponse smsResponse = smsUtil.sendSms(phone, smsCode,
param);
        } catch (ClientException e) {
            e.printStackTrace();
        }
    }
}
```

5. 商品上下架消息处理（作业）

5.1 需求分析

在商品上架后，生成商品详细页和新增elasticsearch索引数据。

在商品下架后，删除商品详细页和删除elasticsearch数据.

以上操作采用消息中间件rabbitmq解耦调用。

5.2 实现思路

因为有多种业务逻辑需要处理，所以我们需要使用分列模式来处理请求。

（1）创建两个服务工程

一是商品详细页生成服务（qingcheng_service_page），对上架的商品重新生成上新品详细页。

二是索引数据更新服务（qingcheng_service_index），对上架的商品重新导入索引库。

（2）通过配置文件实现：

rabbitmq 新增两个交换器，分别是商品上架交换器和商品下架交换器。

rabbitmq 新增商品详细页生成队列、商品详细页删除队列、elasticsearch数据新增队列、elasticsearch数据删除队列。

商品上架交换器绑定新增商品详细页生成队列和elasticsearch数据新增队列

商品下架交换器绑定商品详细页删除队列和elasticsearch数据删除队列

（3）在商品服务的上架发送消息到rabbitmq的商品上架交换器，在商品服务的下架发送消息到rabbitmq的商品下架交换器

（4）商品详细页生成服务从 新增商品详细页生成队列和商品详细页删除队列提取消息进行逻辑处理

（5）索引数据更新服务从elasticsearch数据新增队列和elasticsearch数据删除队列提取消息进行逻辑处理。