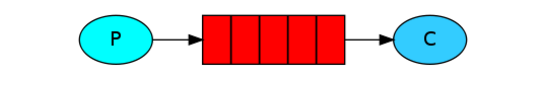
# Java操作队列

## 简单队列

功能：一个生产者P发送消息到队列Q,一个消费者C接收



P表示为生产者 、C表示为消费者 红色表示队列。

### Maven

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.rabbitmq</groupId>  <artifactId>amqp-client</artifactId>  <version>3.6.5</version>  </dependency>  </dependencies> |

### 封装Connection

|  |
| --- |
| **public** **class** MQConnectionUtils {  **public** **static** Connection newConnection() **throws** IOException, TimeoutException {  // 1.定义连接工厂  ConnectionFactory factory = **new** ConnectionFactory();  // 2.设置服务器地址  factory.setHost("127.0.0.1");  // 3.设置AMQP协议端口号  factory.setPort(5672);  // 4.设置vhost  factory.setVirtualHost("/test001\_host");  // 5.设置用户名称  factory.setUsername("test001");  // 6.设置用户密码  factory.setPassword("123456");  // 7.创建新的连接  Connection newConnection = factory.newConnection();  **return** newConnection;  }  } |

### 生产者

|  |
| --- |
| **public** **class** Producer {  **private** **static** **final** String ***QUEUE\_NAME*** = "test\_queue";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  // 1.获取连接  Connection newConnection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = newConnection.createChannel();  // 3.创建队列声明  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  String msg = "test\_yushengjun110";  System.***out***.println("生产者发送消息:" + msg);  // 4.发送消息  channel.basicPublish("", ***QUEUE\_NAME***, **null**, msg.getBytes());  channel.close();  newConnection.close();  }  } |

### 消费者

|  |
| --- |
| **public** **class** Customer {  **private** **static** **final** String ***QUEUE\_NAME*** = "test\_queue";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  System.***out***.println("002");  // 1.获取连接  Connection newConnection = MQConnectionUtils.*newConnection*();  // 2.获取通道  Channel channel = newConnection.createChannel();  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  DefaultConsumer defaultConsumer = **new** DefaultConsumer(channel) {  @Override  **public** **void** handleDelivery(String consumerTag, Envelope envelope, BasicProperties properties, **byte**[] body)  **throws** IOException {  String msgString = **new** String(body, "UTF-8");  System.***out***.println("消费者获取消息:" + msgString);  }  };  // 3.监听队列  channel.basicConsume(***QUEUE\_NAME***, **true**, defaultConsumer);  }  } |

## 工作队列

### 生产者

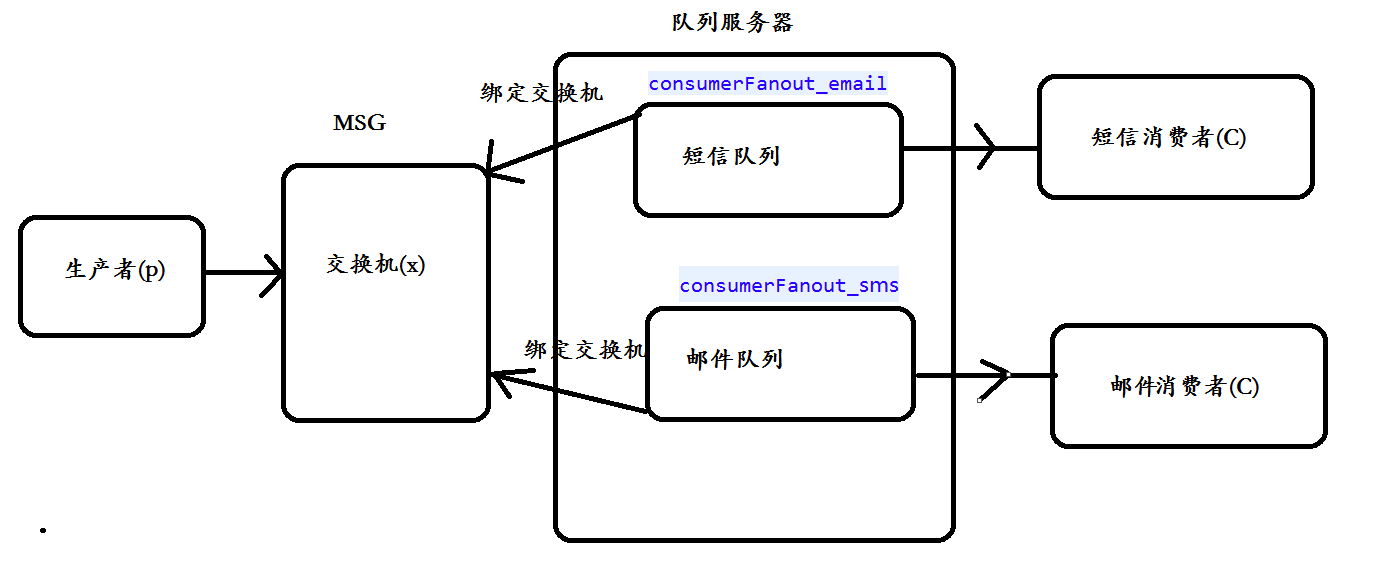
|  |
| --- |
| **public** **class** Producer {  **private** **static** **final** String ***QUEUE\_NAME*** = "test\_queue";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  // 1.获取连接  Connection newConnection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = newConnection.createChannel();  // 3.创建队列声明  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  channel.basicQos(1);// 保证一次只分发一次 限制发送给同一个消费者 不得超过一条消息  **for** (**int** i = 1; i <= 50; i++) {  String msg = "test\_yushengjun" + i;  System.***out***.println("生产者发送消息:" + msg);  // 4.发送消息  channel.basicPublish("", ***QUEUE\_NAME***, **null**, msg.getBytes());  }  channel.close();  newConnection.close();  }  } |

### 消费者

|  |
| --- |
| **public** **class** Customer1 {  **private** **static** **final** String ***QUEUE\_NAME*** = "test\_queue";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  System.***out***.println("001");  // 1.获取连接  Connection newConnection = MQConnectionUtils.*newConnection*();  // 2.获取通道  **final** Channel channel = newConnection.createChannel();  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  channel.basicQos(1);// 保证一次只分发一次 限制发送给同一个消费者 不得超过一条消息  DefaultConsumer defaultConsumer = **new** DefaultConsumer(channel) {  @Override  **public** **void** handleDelivery(String consumerTag, Envelope envelope, BasicProperties properties, **byte**[] body)  **throws** IOException {  String msgString = **new** String(body, "UTF-8");  System.***out***.println("消费者获取消息:" + msgString);  **try** {  Thread.*sleep*(1000);  } **catch** (Exception e) {  } **finally** {  // 手动回执消息  channel.basicAck(envelope.getDeliveryTag(), **false**);  }  }  };  // 3.监听队列  channel.basicConsume(***QUEUE\_NAME***, **false**, defaultConsumer);  }  } |

## 发布订阅

https://github.com/xuexionghui/RabbitMq.git



### 生产者

|  |
| --- |
| **public** **class** ProducerFanout {  **private** **static** **final** String ***EXCHANGE\_NAME*** = "fanout\_exchange";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  // 1.创建新的连接  Connection connection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = connection.createChannel();  // 3.绑定的交换机 参数1交互机名称 参数2 exchange类型  channel.exchangeDeclare(***EXCHANGE\_NAME***, "fanout");  String msg = "fanout\_exchange\_msg";  // 4.发送消息  channel.basicPublish(***EXCHANGE\_NAME***, "", **null**, msg.getBytes());  // System.out.println("生产者发送msg：" + msg);  // // 5.关闭通道、连接  // channel.close();  // connection.close();  // 注意：如果消费没有绑定交换机和队列，则消息会丢失  }  } |

### 邮件消费者

|  |
| --- |
| **public** **class** ConsumerEmailFanout {  **private** **static** **final** String ***QUEUE\_NAME*** = "consumerFanout\_email";  **private** **static** **final** String ***EXCHANGE\_NAME*** = "fanout\_exchange";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  System.***out***.println("邮件消费者启动");  // 1.创建新的连接  Connection connection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = connection.createChannel();  // 3.消费者关联队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  // 4.消费者绑定交换机 参数1 队列 参数2交换机 参数3 routingKey  channel.queueBind(***QUEUE\_NAME***, ***EXCHANGE\_NAME***, "");  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public** **void** handleDelivery(String consumerTag, Envelope envelope, BasicProperties properties, **byte**[] body)  **throws** IOException {  String msg = **new** String(body, "UTF-8");  System.***out***.println("消费者获取生产者消息:" + msg);  }  };  // 5.消费者监听队列消息  channel.basicConsume(***QUEUE\_NAME***, **true**, consumer);  }  } |

### 短信消费者

|  |
| --- |
| **public** **class** ConsumerSMSFanout {  **private** **static** **final** String ***QUEUE\_NAME*** = "ConsumerFanout\_sms";  **private** **static** **final** String ***EXCHANGE\_NAME*** = "fanout\_exchange";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  System.***out***.println("短信消费者启动");  // 1.创建新的连接  Connection connection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = connection.createChannel();  // 3.消费者关联队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  // 4.消费者绑定交换机 参数1 队列 参数2交换机 参数3 routingKey  channel.queueBind(***QUEUE\_NAME***, ***EXCHANGE\_NAME***, "");  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public** **void** handleDelivery(String consumerTag, Envelope envelope, BasicProperties properties, **byte**[] body)  **throws** IOException {  String msg = **new** String(body, "UTF-8");  System.***out***.println("消费者获取生产者消息:" + msg);  }  };  // 5.消费者监听队列消息  channel.basicConsume(***QUEUE\_NAME***, **true**, consumer);  }  } |

## 路由模式

### 生产者

|  |
| --- |
| **public** **class** ProducerDirect {  **private** **static** **final** String ***EXCHANGE\_NAME*** = "direct\_exchange";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  // 1.创建新的连接  Connection connection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = connection.createChannel();  // 3.绑定的交换机 参数1交互机名称 参数2 exchange类型  channel.exchangeDeclare(***EXCHANGE\_NAME***, "direct");  String routingKey = "info";  String msg = "direct\_exchange\_msg" + routingKey;  // 4.发送消息  channel.basicPublish(***EXCHANGE\_NAME***, routingKey, **null**, msg.getBytes());  System.***out***.println("生产者发送msg：" + msg);  // // 5.关闭通道、连接  // channel.close();  // connection.close();  // 注意：如果消费没有绑定交换机和队列，则消息会丢失  }  } |

### 邮件消费者

|  |
| --- |
| **public** **class** ConsumerEmailDirect {  **private** **static** **final** String ***QUEUE\_NAME*** = "consumer\_direct\_email";  **private** **static** **final** String ***EXCHANGE\_NAME*** = "direct\_exchange";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  System.***out***.println("邮件消费者启动");  // 1.创建新的连接  Connection connection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = connection.createChannel();  // 3.消费者关联队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  // 4.消费者绑定交换机 参数1 队列 参数2交换机 参数3 routingKey  channel.queueBind(***QUEUE\_NAME***, ***EXCHANGE\_NAME***, "error");  channel.queueBind(***QUEUE\_NAME***, ***EXCHANGE\_NAME***, "info");  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public** **void** handleDelivery(String consumerTag, Envelope envelope, BasicProperties properties, **byte**[] body)  **throws** IOException {  String msg = **new** String(body, "UTF-8");  System.***out***.println("消费者获取生产者消息:" + msg);  }  };  // 5.消费者监听队列消息  channel.basicConsume(***QUEUE\_NAME***, **true**, consumer);  }  } |

### 短信消费者

|  |
| --- |
| **public** **class** ConsumerSMSDirect {  **private** **static** **final** String ***QUEUE\_NAME*** = "consumer\_direct\_sms";  **private** **static** **final** String ***EXCHANGE\_NAME*** = "direct\_exchange";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  System.***out***.println("短信消费者启动");  // 1.创建新的连接  Connection connection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = connection.createChannel();  // 3.消费者关联队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  // 4.消费者绑定交换机 参数1 队列 参数2交换机 参数3 routingKey  channel.queueBind(***QUEUE\_NAME***, ***EXCHANGE\_NAME***, "error");  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public** **void** handleDelivery(String consumerTag, Envelope envelope, BasicProperties properties, **byte**[] body)  **throws** IOException {  String msg = **new** String(body, "UTF-8");  System.***out***.println("消费者获取生产者消息:" + msg);  }  };  // 5.消费者监听队列消息  channel.basicConsume(***QUEUE\_NAME***, **true**, consumer);  }  } |

## 主题模式

### 生产者

|  |
| --- |
| **public** **class** ProducerDirect {  **private** **static** **final** String ***EXCHANGE\_NAME*** = "topic\_exchange";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  // 1.创建新的连接  Connection connection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = connection.createChannel();  // 3.绑定的交换机 参数1交互机名称 参数2 exchange类型  channel.exchangeDeclare(***EXCHANGE\_NAME***, "topic");  String routingKey = "log.info.error";  String msg = "topic\_exchange\_msg" + routingKey;  // 4.发送消息  channel.basicPublish(***EXCHANGE\_NAME***, routingKey, **null**, msg.getBytes());  System.***out***.println("生产者发送msg：" + msg);  // // 5.关闭通道、连接  channel.close();  connection.close();  // 注意：如果消费没有绑定交换机和队列，则消息会丢失  }  } |

### 邮件消费者

|  |
| --- |
| **public** **class** ConsumerEmailDirect {  **private** **static** **final** String ***QUEUE\_NAME*** = "consumer\_topic\_email";  **private** **static** **final** String ***EXCHANGE\_NAME*** = "topic\_exchange";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  System.***out***.println("邮件消费者启动");  // 1.创建新的连接  Connection connection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = connection.createChannel();  // 3.消费者关联队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  channel.queueBind(***QUEUE\_NAME***, ***EXCHANGE\_NAME***, "log.#");  // 4.消费者绑定交换机 参数1 队列 参数2交换机 参数3 routingKey  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public** **void** handleDelivery(String consumerTag, Envelope envelope, BasicProperties properties, **byte**[] body)  **throws** IOException {  String msg = **new** String(body, "UTF-8");  System.***out***.println("消费者获取生产者消息:" + msg);  }  };  // 5.消费者监听队列消息  channel.basicConsume(***QUEUE\_NAME***, **true**, consumer);  }  } |

### 短信服务器

|  |
| --- |
| **public** **class** ConsumerSMSDirect {  **private** **static** **final** String ***QUEUE\_NAME*** = "consumer\_topic\_sms";  **private** **static** **final** String ***EXCHANGE\_NAME*** = "topic\_exchange";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  System.***out***.println("短信消费者启动");  // 1.创建新的连接  Connection connection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = connection.createChannel();  // 3.消费者关联队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  // 4.消费者绑定交换机 参数1 队列 参数2交换机 参数3 routingKey  channel.queueBind(***QUEUE\_NAME***, ***EXCHANGE\_NAME***, "log.\*");  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public** **void** handleDelivery(String consumerTag, Envelope envelope, BasicProperties properties, **byte**[] body)  **throws** IOException {  String msg = **new** String(body, "UTF-8");  System.***out***.println("消费者获取生产者消息:" + msg);  }  };  // 5.消费者监听队列消息  channel.basicConsume(***QUEUE\_NAME***, **true**, consumer);  }  } |

## RabbitMQ消息确认机制https://blog.csdn.net/anumbrella/article/details/81321701

生产者发送消息出去之后，不知道到底有没有发送到RabbitMQ服务器， 默认是不知道的。而且有的时候我们在发送消息之后，后面的逻辑出问题了，我们不想要发送之前的消息了，需要撤回该怎么做。 解决方案:都是在生产者进行配置改动 1.AMQP 事务机制 2.Confirm 模式事务模式: txSelect 将当前channel设置为transaction模式 txCommit 提交当前事务 txRollback 事务回滚

### AMQP 事务机制

#### 生产者

|  |
| --- |
| **public** **class** Producer {  **private** **static** **final** String ***QUEUE\_NAME*** = "test\_queue";  **public** **static** **void** main(String[] args) **throws** IOException, TimeoutException {  // 1.获取连接  Connection newConnection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = newConnection.createChannel();  // 3.创建队列声明  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  // 将当前管道设置为 txSelect 将当前channel设置为transaction模式 开启事务  channel.txSelect();  String msg = "test\_yushengjun110";  **try** {  // 4.发送消息  channel.basicPublish("", ***QUEUE\_NAME***, **null**, msg.getBytes());  // int i = 1 / 0;  channel.txCommit();// 提交事务  System.***out***.println("生产者发送消息:" + msg);  } **catch** (Exception e) {  System.***out***.println("消息进行回滚操作");  channel.txRollback();// 回滚事务  } **finally** {  channel.close();  newConnection.close();  }  }  } |

### Confirm 模式

|  |
| --- |
| // 1.获取连接  Connection newConnection = MQConnectionUtils.*newConnection*();  // 2.创建通道  Channel channel = newConnection.createChannel();  // 3.创建队列声明  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  // confirm机制  channel.confirmSelect();  String msg = "test\_yushengjun110";  // 4.发送消息  channel.basicPublish("", ***QUEUE\_NAME***, **null**, msg.getBytes());  System.***out***.println("生产者发送消息:" + msg);  **if** (!channel.waitForConfirms()) {  System.***out***.println("消息发送失败!");  } **else** {  System.***out***.println("消息发送成功!");  }  channel.close();  newConnection.close(); |

## SpringBoot整合RabbitMQ

https://github.com/xuexionghui/RabbitMq\_springboot.git

### 生产者

#### Maven环境依赖

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.0.RELEASE</version>  </parent>  <dependencies>  <!-- springboot-web组件 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!-- 添加springboot对amqp的支持 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-amqp</artifactId>  </dependency>  <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-lang3</artifactId>  </dependency>  <!--fastjson -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.2.49</version>  </dependency>  </dependencies> |

#### application.yml

|  |
| --- |
| spring:  rabbitmq:  ####连接地址  host: 127.0.0.1  ####端口号  port: 5672  ####账号  username: guest  ####密码  password: guest  ### 地址  virtual-host: / |

#### 交换机绑定队列

|  |
| --- |
| @Component  **public** **class** FanoutConfig {  // 邮件队列  **private** String FANOUT\_EMAIL\_QUEUE = "fanout\_eamil\_queue";  // 短信队列  **private** String FANOUT\_SMS\_QUEUE = "fanout\_sms\_queue";  // 短信队列  **private** String EXCHANGE\_NAME = "fanoutExchange";  // 1.定义队列邮件  @Bean  **public** Queue fanOutEamilQueue() {  **return** **new** Queue(FANOUT\_EMAIL\_QUEUE);  }  @Bean  **public** Queue fanOutSmsQueue() {  **return** **new** Queue(FANOUT\_SMS\_QUEUE);  }  // 2.定义交换机  @Bean  FanoutExchange fanoutExchange() {  **return** **new** FanoutExchange(EXCHANGE\_NAME);  }  // 3.队列与交换机绑定邮件队列  @Bean  Binding bindingExchangeEamil(Queue fanOutEamilQueue, FanoutExchange fanoutExchange) {  **return** BindingBuilder.*bind*(fanOutEamilQueue).to(fanoutExchange);  }  // 4.队列与交换机绑定短信队列  @Bean  Binding bindingExchangeSms(Queue fanOutSmsQueue, FanoutExchange fanoutExchange) {  **return** BindingBuilder.*bind*(fanOutSmsQueue).to(fanoutExchange);  }  } |

#### 生产者投递消息

|  |
| --- |
| @Component  **public** **class** FanoutProducer {  @Autowired  **private** AmqpTemplate amqpTemplate;  **public** **void** send(String queueName) {  String msg = "my\_fanout\_msg:" + **new** Date();  System.***out***.println(msg + ":" + msg);  amqpTemplate.convertAndSend(queueName, msg);  }  } |

#### 控制层调用代码

|  |
| --- |
| @RestController  **public** **class** ProducerController {  @Autowired  **private** FanoutProducer fanoutProducer;  @RequestMapping("/sendFanout")  **public** String sendFanout(String queueName) {  fanoutProducer.send(queueName);  **return** "success";  }  } |

### 消费者

#### Maven环境依赖

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.0.RELEASE</version>  </parent>  <dependencies>  <!-- springboot-web组件 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!-- 添加springboot对amqp的支持 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-amqp</artifactId>  </dependency>  <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-lang3</artifactId>  </dependency>  <!--fastjson -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.2.49</version>  </dependency>  </dependencies> |

#### 邮件消费者

|  |
| --- |
| @Component  @RabbitListener(queues = "fanout\_eamil\_queue")  **public** **class** FanoutEamilConsumer {  @RabbitHandler  **public** **void** process(String msg) **throws** Exception {  System.***out***.println("邮件消费者获取生产者消息msg:" + msg);  }  } |

#### 短信消费者

|  |
| --- |
| @Component  @RabbitListener(queues = "fanout\_sms\_queue")  **public** **class** FanoutSmsConsumer {  @RabbitHandler  **public** **void** process(String msg) {  System.***out***.println("短信消费者获取生产者消息msg:" + msg);  }  } |

#### application.yml

|  |
| --- |
| spring:  rabbitmq:  ####连接地址  host: 127.0.0.1  ####端口号  port: 5672  ####账号  username: guest  ####密码  password: guest  ### 地址  virtual-host: / |

# SpringBoot消息重试机制

### 消息重试机制幂等性

#### 如何合适选择重试机制

情况1:  消费者获取到消息后，调用第三方接口，但接口暂时无法访问，是否需要重试?      需要重试

情况2:  消费者获取到消息后，抛出数据转换异常，是否需要重试?                                      不需要重试

总结：对于情况2，如果消费者代码抛出异常是需要发布新版本才能解决的问题，那么不需要重试，重试也无济于事。应该采用日志记录+定时任务job健康检查+人工进行补偿

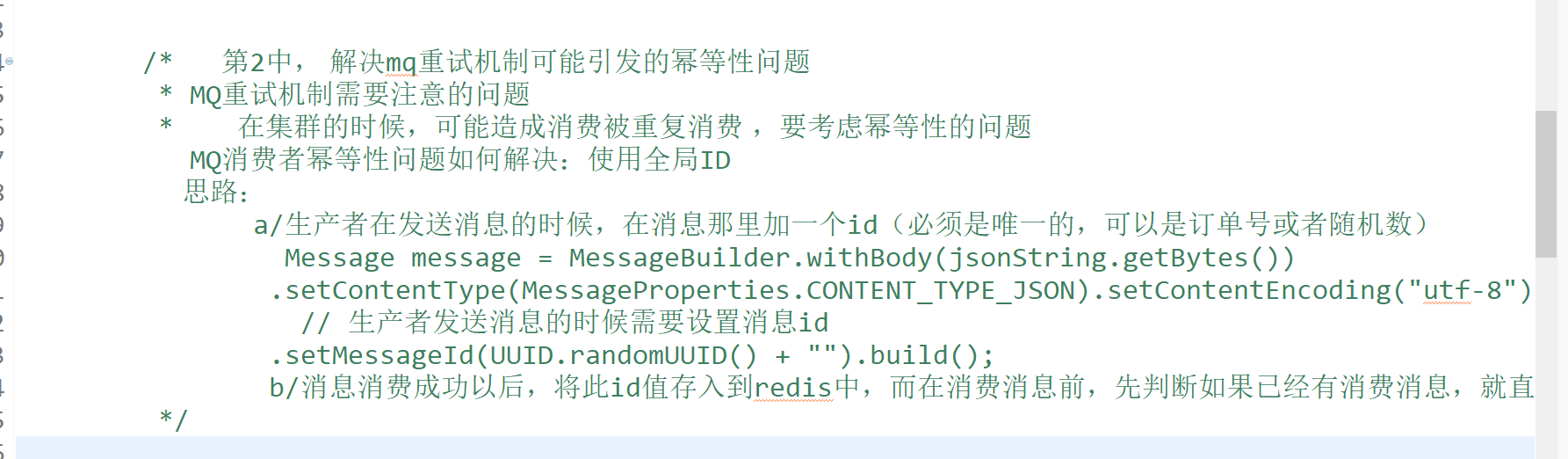
#### 消费者如果保证消息幂等性，不被重复消费

产生原因:网络延迟传输中，会造成进行MQ重试中，在重试过程中，可能会造成重复消费。

解决办法:

使用全局MessageID判断消费方使用同一个，解决幂等性。

#### 基于全局消息id区分消息，解决幂等性



##### 生产者:

请求头设置消息id（messageId）

|  |
| --- |
| @Component  **public** **class** FanoutProducer {  @Autowired  **private** AmqpTemplate amqpTemplate;  **public** **void** send(String queueName) {  String msg = "my\_fanout\_msg:" + System.*currentTimeMillis*();  Message message = MessageBuilder.*withBody*(msg.getBytes()).setContentType(MessageProperties.***CONTENT\_TYPE\_JSON***)  .setContentEncoding("utf-8").setMessageId(UUID.*randomUUID*() + "").build();  System.***out***.println(msg + ":" + msg);  amqpTemplate.convertAndSend(queueName, message);  }  } |

##### 消费者:

###### 核心代码

|  |
| --- |
| @Component  **public** **class** FanoutEamilConsumer {  @RabbitListener(queues = "fanout\_email\_queue")  **public** **void** process(Message message) **throws** Exception {  System.***out***  .println(Thread.*currentThread*().getName() + ",邮件消费者获取生产者消息msg:" + **new** String(message.getBody(), "UTF-8")  + ",messageId:" + message.getMessageProperties().getMessageId());  // int i = 1 / 0;  }  } |

###### application配置

|  |
| --- |
| spring:  rabbitmq:  ####连接地址  host: 127.0.0.1  ####端口号  port: 5672  ####账号  username: guest  ####密码  password: guest  ### 地址  virtual-host: /admin\_host  listener:  simple:  retry:  ####开启消费者重试  enabled: **true**  ####最大重试次数  max-attempts: 5  ####重试间隔次数  initial-interval: 3000    server:  port: 8081 |

#### RabbitMQ消费者重试调用接口

|  |
| --- |
| //邮件队列  @Component  **public** **class** FanoutEamilConsumer {  @RabbitListener(queues = "fanout\_email\_queue")  **public** **void** process(String msg) **throws** Exception {  System.***out***.println("邮件消费者获取生产者消息msg:" + msg);  JSONObject jsonObject = JSONObject.*parseObject*(msg);  // 获取email参数  String email = jsonObject.getString("email");  // 请求地址  String emailUrl = "http://127.0.0.1:8083/sendEmail?email=" + email;  JSONObject result = HttpClientUtils.*httpGet*(emailUrl);  **if** (result == **null**) {  // 因为网络原因,造成无法访问,继续重试  **throw** **new** Exception("调用接口失败!");  }  System.***out***.println("执行结束....");  }  } |

|  |
| --- |
| @RabbitListener(queues = "fanout\_email\_queue")  **public** **void** process(Message message, @Headers Map<String, Object> headers, Channel channel) **throws** Exception {  String messageId = message.getMessageProperties().getMessageId();  String msg = **new** String(message.getBody(), "UTF-8");  System.***out***.println("邮件消费者获取生产者消息msg:" + msg + ",消息id:" + messageId);  // 解决重试机制可能造成的幂等性问题，造成消息的重复读取  String string = redisTemplate.execute(**new** RedisCallback<String>() {  @Override  **public** String doInRedis(RedisConnection connection) **throws** DataAccessException {  **byte**[] codeBytes = connection  .get(redisTemplate.getStringSerializer().serialize("messageId:" + msgId));  **return** redisTemplate.getStringSerializer().deserialize(codeBytes);  }  });  **if**(string!=**null**&&string.equals(messageId)) {  **return** ;  }  JSONObject jsonObject = JSONObject.*parseObject*(msg);  String email = jsonObject.getString("email");  String emailUrl = "http://127.0.0.1:8083/sendEmail?email=" + email;  System.***out***.println("邮件消费者开始调用第三方邮件服务器,emailUrl:" + emailUrl);  JSONObject result = HttpClientUtils.*httpGet*(emailUrl);  // 如果调用第三方邮件接口无法访问，如何实现自动重试.    **if** (result == **null**) {  **throw** **new** Exception("调用第三方邮件服务器接口失败!");  }  System.***out***.println("邮件消费者结束调用第三方邮件服务器成功,result:" + result + "程序执行结束");  //消息消费成功，将messageId存入redis中  redisTemplate.execute(**new** RedisCallback<String>() {  @Override  **public** String doInRedis(RedisConnection connection) **throws** DataAccessException {  connection.setEx(redisTemplate.getStringSerializer().serialize("messageId:" + msgId),  5000, redisTemplate.getStringSerializer().serialize(messageId+ ""));  **return** **null**;  }  });  System.***out***.println("messageId是："+messageId);    // 手动ack  /\*Long deliveryTag = (Long) headers.get(AmqpHeaders.DELIVERY\_TAG);  // 手动签收  channel.basicAck(deliveryTag, false);\*/  }  // 默认是自动应答模式 |

# RabbitMQ签收模式

|  |
| --- |
| //邮件队列  @Component  **public** **class** FanoutEamilConsumer {  @RabbitListener(queues = "fanout\_email\_queue")  **public** **void** process(Message message, @Headers Map<String, Object> headers, Channel channel) **throws** Exception {  System.***out***  .println(Thread.*currentThread*().getName() + ",邮件消费者获取生产者消息msg:" + **new** String(message.getBody(), "UTF-8")  + ",messageId:" + message.getMessageProperties().getMessageId());  // 手动ack  Long deliveryTag = (Long) headers.get(AmqpHeaders.***DELIVERY\_TAG***);  // 手动签收  channel.basicAck(deliveryTag, **false**);  }  } |

开启手动应答

|  |
| --- |
| spring:  rabbitmq:  ####连接地址  host: 127.0.0.1  ####端口号  port: 5672  ####账号  username: guest  ####密码  password: guest  ### 地址  virtual-host: /admin\_host  listener:  simple:  retry:  ####开启消费者异常重试  enabled: **true**  ####最大重试次数  max-attempts: 5  ####重试间隔次数  initial-interval: 2000  ####开启手动ack  acknowledge-mode: manual |

# RabbitMQ解决分布式事务问题

RabbitMQ解决分布式事务原理： 采用最终一致性原理。需要保证以下三要素1、确认生产者一定要将数据投递到MQ服务器中（采用MQ消息确认机制）2、MQ消费者消息能够正确消费消息，采用手动ACK模式（注意重试幂等性问题）3、如何保证第一个事务先执行，采用补偿机制，在创建一个补单消费者进行监听，如果订单没有创建成功，进行补单。

## 订单项目

### 生产者

|  |
| --- |
| @Service  **public** **class** OrderService **extends** BaseApiService **implements** RabbitTemplate.ConfirmCallback {  @Autowired  **private** OrderMapper orderMapper;  @Autowired  **private** RabbitTemplate rabbitTemplate;  **public** ResponseBase addOrderAndDispatch() {  OrderEntity orderEntity = **new** OrderEntity();  orderEntity.setName("蚂蚁课堂永久会员充值");  orderEntity.setOrderCreatetime(**new** Date());  // 价格是300元  orderEntity.setOrderMoney(300d);  // 状态为 未支付  orderEntity.setOrderState(0);  Long commodityId = 30l;  // 商品id  orderEntity.setCommodityId(commodityId);  String orderId = UUID.*randomUUID*().toString();  orderEntity.setOrderId(orderId);  // ##################################################  // 1.先下单，创建订单 (往订单数据库中插入一条数据)  **int** orderResult = orderMapper.addOrder(orderEntity);  System.***out***.println("orderResult:" + orderResult);  **if** (orderResult <= 0) {  **return** setResultError("下单失败!");  }  // 2.使用消息中间件将参数存在派单队列中  send(orderId);  **return** setResultSuccess();  }  **private** **void** send(String orderId) {  JSONObject jsonObect = **new** JSONObject();  jsonObect.put("orderId", orderId);  String msg = jsonObect.toJSONString();  System.***out***.println("msg:" + msg);  // 封装消息  Message message = MessageBuilder.*withBody*(msg.getBytes()).setContentType(MessageProperties.***CONTENT\_TYPE\_JSON***)  .setContentEncoding("utf-8").setMessageId(orderId).build();  // 构建回调返回的数据  CorrelationData correlationData = **new** CorrelationData(orderId);  // 发送消息  **this**.rabbitTemplate.setMandatory(**true**);  **this**.rabbitTemplate.setConfirmCallback(**this**);  rabbitTemplate.convertAndSend("order\_exchange\_name", "orderRoutingKey", message, correlationData);  }  // 生产消息确认机制  @Override  **public** **void** confirm(CorrelationData correlationData, **boolean** ack, String cause) {  String orderId = correlationData.getId();  System.***out***.println("消息id:" + correlationData.getId());  **if** (ack) {  System.***out***.println("消息发送确认成功");  } **else** {  send(orderId);  System.***out***.println("消息发送确认失败:" + cause);  }  }  } |

### 补单消费者

|  |
| --- |
| @Component  **public** **class** CreateOrderConsumer {  @Autowired  **private** OrderMapper orderMapper;  @RabbitListener(queues = "order\_create\_queue")  **public** **void** process(Message message, @Headers Map<String, Object> headers, Channel channel) **throws** Exception {  String messageId = message.getMessageProperties().getMessageId();  String msg = **new** String(message.getBody(), "UTF-8");  System.***out***.println("补单消费者" + msg + ",消息id:" + messageId);  JSONObject jsonObject = JSONObject.*parseObject*(msg);  String orderId = jsonObject.getString("orderId");  // 判断订单是否存在，如果不存在 实现自动补单机制  OrderEntity orderEntityResult = orderMapper.findOrderId(orderId);  **if** (orderEntityResult != **null**) {  System.***out***.println("订单已经存在 无需补单 orderId:" + orderId);  **return**;  }  // 订单不存在 ，则需要进行补单  OrderEntity orderEntity = **new** OrderEntity();  orderEntity.setName("蚂蚁课堂永久会员充值");  orderEntity.setOrderCreatetime(**new** Date());  // 价格是300元  orderEntity.setOrderMoney(300d);  // 状态为 未支付  orderEntity.setOrderState(0);  Long commodityId = 30l;  // 商品id  orderEntity.setCommodityId(commodityId);  orderEntity.setOrderId(orderId);  // ##################################################  // 1.先下单，创建订单 (往订单数据库中插入一条数据)  **try** {  **int** orderResult = orderMapper.addOrder(orderEntity);  System.***out***.println("orderResult:" + orderResult);  **if** (orderResult >= 0) {  // 手动签收消息,通知mq服务器端删除该消息  channel.basicAck(message.getMessageProperties().getDeliveryTag(), **false**);  }  } **catch** (Exception e) {  // 丢弃该消息  channel.basicNack(message.getMessageProperties().getDeliveryTag(), **false**, **false**);  }  }  } |

### RabbitmqConfig

|  |
| --- |
| @Component  **public** **class** RabbitmqConfig {  // 下单并且派单存队列  **public** **static** **final** String ***ORDER\_DIC\_QUEUE*** = "order\_dic\_queue";  // 补单队列，判断订单是否已经被创建  **public** **static** **final** String ***ORDER\_CREATE\_QUEUE*** = "order\_create\_queue";  // 下单并且派单交换机  **private** **static** **final** String ***ORDER\_EXCHANGE\_NAME*** = "order\_exchange\_name";  // 1.定义订单队列  @Bean  **public** Queue directOrderDicQueue() {  **return** **new** Queue(***ORDER\_DIC\_QUEUE***);  }  // 2.定义补订单队列  @Bean  **public** Queue directCreateOrderQueue() {  **return** **new** Queue(***ORDER\_CREATE\_QUEUE***);  }  // 2.定义交换机  @Bean  DirectExchange directOrderExchange() {  **return** **new** DirectExchange(***ORDER\_EXCHANGE\_NAME***);  }  // 3.订单队列与交换机绑定  @Bean  Binding bindingExchangeOrderDicQueue() {  **return** BindingBuilder.*bind*(directOrderDicQueue()).to(directOrderExchange()).with("orderRoutingKey");  }  // 3.补单队列与交换机绑定  @Bean  Binding bindingExchangeCreateOrder() {  **return** BindingBuilder.*bind*(directCreateOrderQueue()).to(directOrderExchange()).with("orderRoutingKey");  }  } |

## 派单服务

### 消费者

|  |
| --- |
| @Component  **public** **class** DispatchConsumer {  @Autowired  **private** DispatchMapper dispatchMapper;  @RabbitListener(queues = "order\_dic\_queue")  **public** **void** process(Message message, @Headers Map<String, Object> headers, Channel channel) **throws** Exception {  String messageId = message.getMessageProperties().getMessageId();  String msg = **new** String(message.getBody(), "UTF-8");  System.***out***.println("派单服务平台" + msg + ",消息id:" + messageId);  JSONObject jsonObject = JSONObject.*parseObject*(msg);  String orderId = jsonObject.getString("orderId");  **if** (StringUtils.*isEmpty*(orderId)) {  // 日志记录  **return**;  }  DispatchEntity dispatchEntity = **new** DispatchEntity();  // 订单id  dispatchEntity.setOrderId(orderId);  // 外卖员id  dispatchEntity.setTakeoutUserId(12l);  // 外卖路线  dispatchEntity.setDispatchRoute("40,40");  **try** {  **int** insertDistribute = dispatchMapper.insertDistribute(dispatchEntity);  **if** (insertDistribute > 0) {  // 手动签收消息,通知mq服务器端删除该消息  channel.basicAck(message.getMessageProperties().getDeliveryTag(), **false**);  }  } **catch** (Exception e) {  e.printStackTrace();  // // 丢弃该消息  channel.basicNack(message.getMessageProperties().getDeliveryTag(), **false**, **false**);  }  }  } |