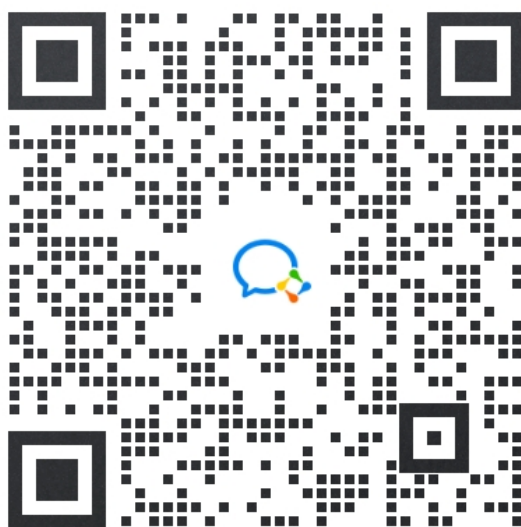


3. RabbitMQ应用

版权说明

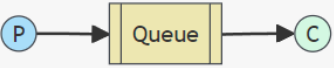
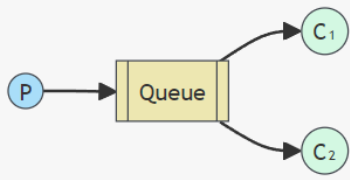
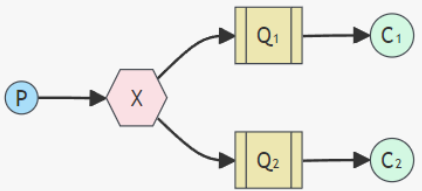
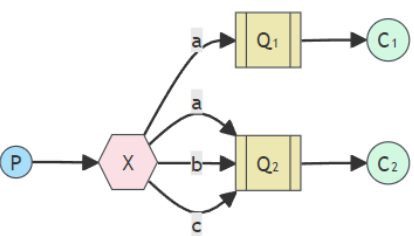
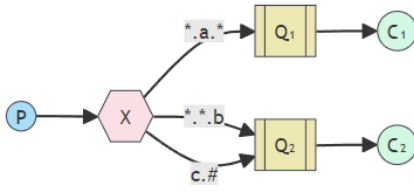
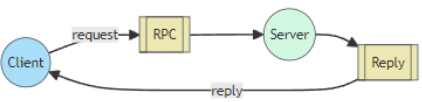
本“比特就业课”课程（以下简称“本课程”）的所有内容，包括但不限于文字、图片、音频、视频、软件、程序、数据库、设计、布局、界面等，均由本课程的开发者或授权方拥有版权。我们鼓励个人学习者使用本课程进行学习和研究。在遵守相关法律法规的前提下，个人学习者可以下载、浏览、学习本课程的内容，并为了个人学习、研究或教学目的而使用其中的材料。但请注意，未经我们明确授权，个人学习者不得将本课程的内容用于任何商业目的，包括但不限于销售、转让、许可或以其他方式从中获利。此外，个人学习者也不得擅自修改、复制、传播、展示、表演或制作本课程内容的衍生作品。任何未经授权的使用均属侵权行为，我们将依法追究法律责任。如果您希望以其他方式使用本课程的内容，包括但不限于引用、转载、摘录、改编等，请事先与我们联系，获取书面授权。感谢您对“比特就业课”课程的关注与支持，我们将持续努力，为您提供更好的学习体验。特此说明。比特就业课版权所有方。

对比特课程感兴趣，可以联系这个微信。



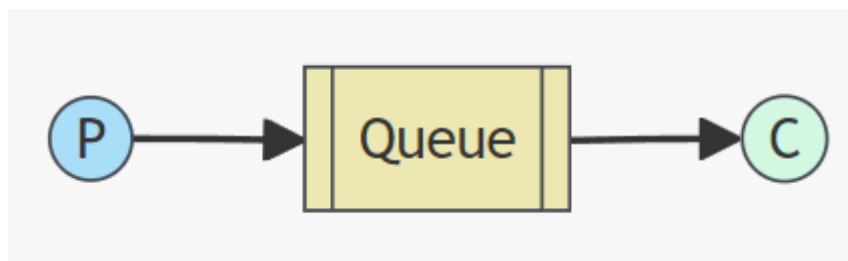
RabbitMQ 共提供了7种工作模式, 进行消息传递, 我们入门程序的案例, 其实就是一个简单模式.

官方文档 [RabbitMQ Tutorials](#) | [RabbitMQ](#)

<p>1. "Hello World!"</p> <p>The simplest thing that does <i>something</i></p>  <pre> graph LR P((P)) --> Queue[Queue] Queue --> C((C)) </pre>	<p>2. Work Queues</p> <p>Distributing tasks among workers (the <i>competing consumers pattern</i>)</p>  <pre> graph LR P((P)) --> Queue[Queue] Queue --> C1((C1)) Queue --> C2((C2)) </pre>	<p>3. Publish/Subscribe</p> <p>Sending messages to many consumers at once</p>  <pre> graph LR P((P)) --> X{X} X --> Q1[Q1] X --> Q2[Q2] Q1 --> C1((C1)) Q2 --> C2((C2)) </pre>
<p>4. Routing</p> <p>Receiving messages selectively</p>  <pre> graph LR P((P)) --> X{X} X -- a --> Q1[Q1] X -- b --> Q2[Q2] X -- c --> Q2[Q2] Q1 --> C1((C1)) Q2 --> C2((C2)) </pre>	<p>5. Topics</p> <p>Receiving messages based on a pattern (topics)</p>  <pre> graph LR P((P)) --> X{X} X -- "a.*" --> Q1[Q1] X -- ".*b" --> Q2[Q2] X -- "c.#" --> Q2[Q2] Q1 --> C1((C1)) Q2 --> C2((C2)) </pre>	<p>6. RPC</p> <p><i>Request/reply pattern</i> example</p>  <pre> graph LR Client((Client)) -- request --> RPC[RPC] RPC --> Server((Server)) Server -- reply --> Client </pre>
<p>7. Publisher Confirms</p> <p>Reliable publishing with publisher confirms</p>		

1. 7种工作模式介绍

1.1 Simple(简单模式)



P: 生产者, 也就是要发送消息的程序

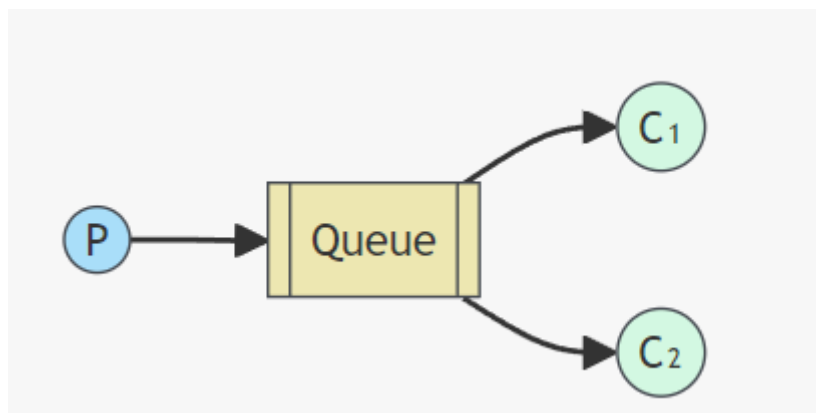
C: 消费者, 消息的接收者

Queue: 消息队列, 图中黄色背景部分. 类似一个邮箱, 可以缓存消息; 生产者向其中投递消息, 消费者从其中取出消息.

特点: 一个生产者P, 一个消费者C, 消息只能被消费一次. 也称为点对点(Point-to-Point)模式.

适用场景: 消息只能被单个消费者处理

1.2 Work Queue(工作队列)

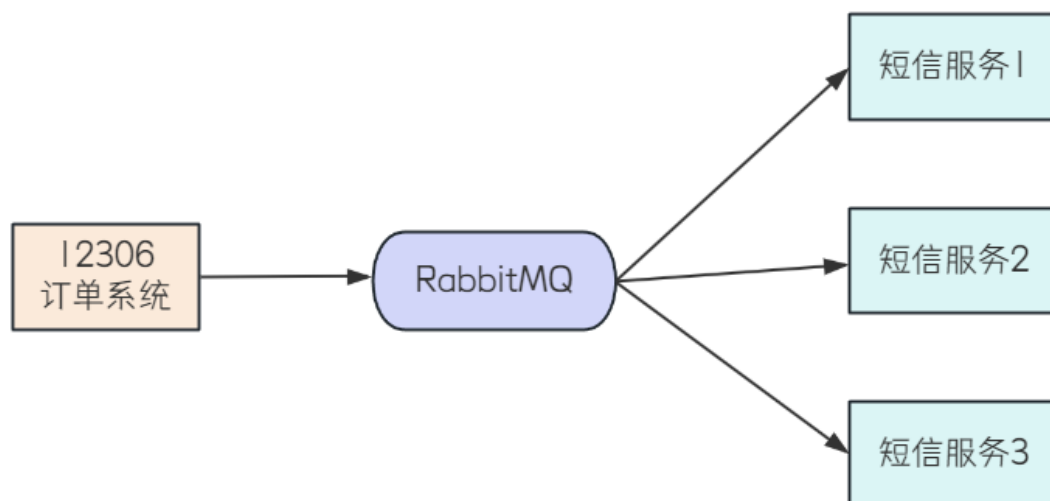


一个生产者P，多个消费者C1,C2. 在多个消息的情况下, Work Queue 会将消息分派给不同的消费者, 每个消费者都会接收到不同的消息.

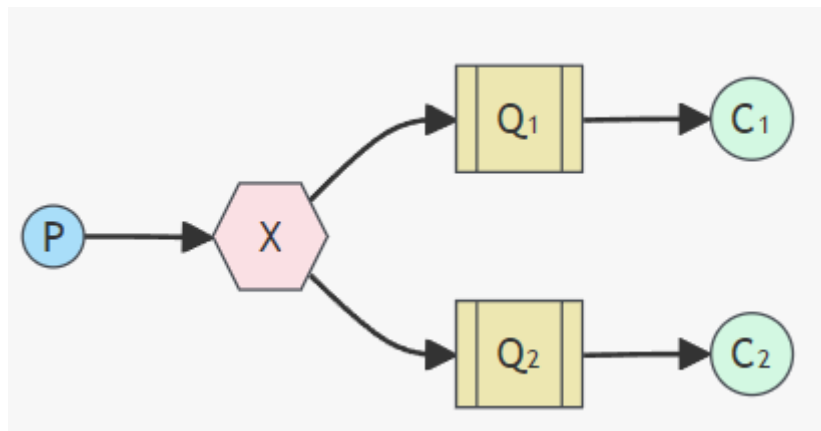
特点: 消息不会重复, 分配给不同的消费者.

适用场景: 集群环境中做异步处理

比如12306 短信通知服务, 订票成功后, 订单消息会发送到RabbitMQ, 短信服务从RabbitMQ中获取订单信息, 并发送通知信息(在短信服务之间进行任务分配)



1.3 Publish/Subscribe(发布/订阅)



图中X表示交换机, 在订阅模型中, 多了一个Exchange角色, 过程略有变化

概念介绍

Exchange: 交换机 (X).

作用: 生产者将消息发送到Exchange, 由交换机将消息按一定规则路由到一个或多个队列中(上图中生产者将消息投递到队列中, 实际上这个在RabbitMQ中不会发生.)

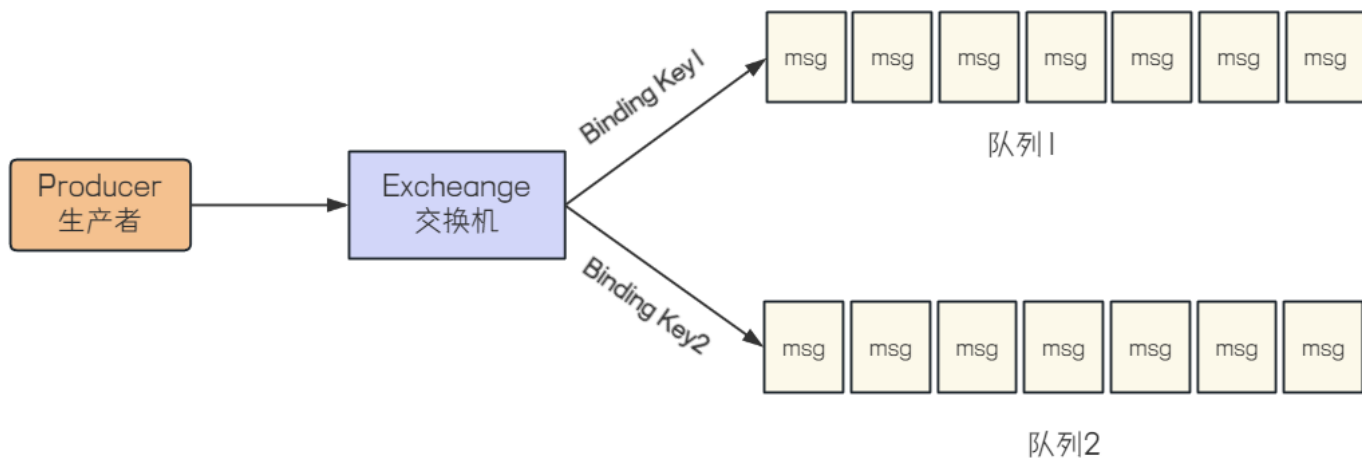
RabbitMQ交换机有四种类型: fanout,direct, topic, headers, 不同类型有着不同的路由策略. AMQP协议里还有另外两种类型, System和自定义, 此处不再描述.

1. Fanout:广播, 将消息交给所有绑定到交换机的队列(Publish/Subscribe模式)
2. Direct:定向, 把消息交给符合指定routing key的队列(Routing模式)
3. Topic:通配符, 把消息交给符合routing pattern(路由模式)的队列(Topics模式)
4. headers类型的交换器不依赖于路由键的匹配规则来路由消息, 而是根据发送的消息内容中的headers属性进行匹配. headers类型的交换器性能会很差,而且也不实用,基本上不会看到它的存在.

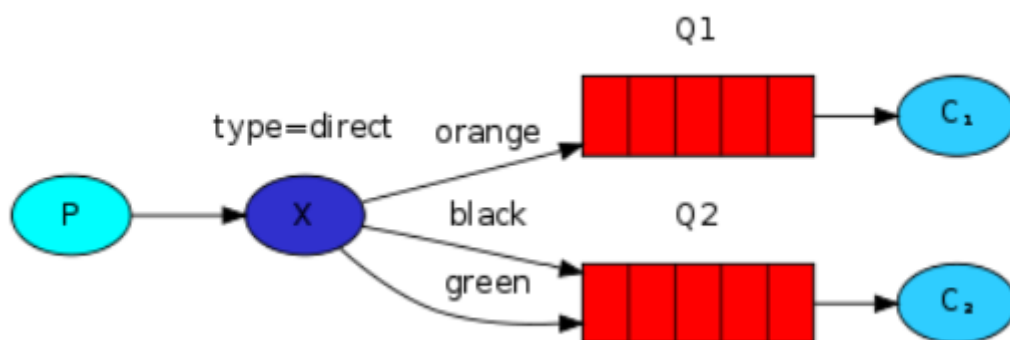
Exchange(交换机) 只负责转发消息, 不具备存储消息的能力, 因此如果没有任何队列与Exchange绑定, 或者没有符合路由规则的队列, 那么消息就会丢失

RoutingKey: 路由键.生产者将消息发给交换器时, 指定的一个字符串, 用来告诉交换机应该如何处理这个消息.

Binding Key:绑定. RabbitMQ中通过Binding(绑定)将交换器与队列关联起来, 在绑定的时候一般会指定一个Binding Key, 这样RabbitMQ就知道如何正确地将消息路由到队列了.



比如下图: 如果在发送消息时, 设置了RoutingKey 为orange, 消息就会路由到Q1



当消息的Routing key与队列绑定的Bindingkey相匹配时, 消息才会被路由到这个队列.

BindingKey其实也属于路由键中的一种, 官方解释为:the routingkey to use for the binding.

可以翻译为:在绑定的时候使用的路由键. 大多数时候,包括官方文档和RabbitMQJava API 中都把BindingKey和RoutingKey看作RoutingKey, 为了避免混淆,可以这么理解:

1. 在使用绑定的时候,需要的路由键是BindingKey.
2. 在发送消息的时候,需要的路由键是RoutingKey.

本课程后续也可能把两者合称为Routing Key, 大家根据使用场景来区分

Publish/Subscribe模式

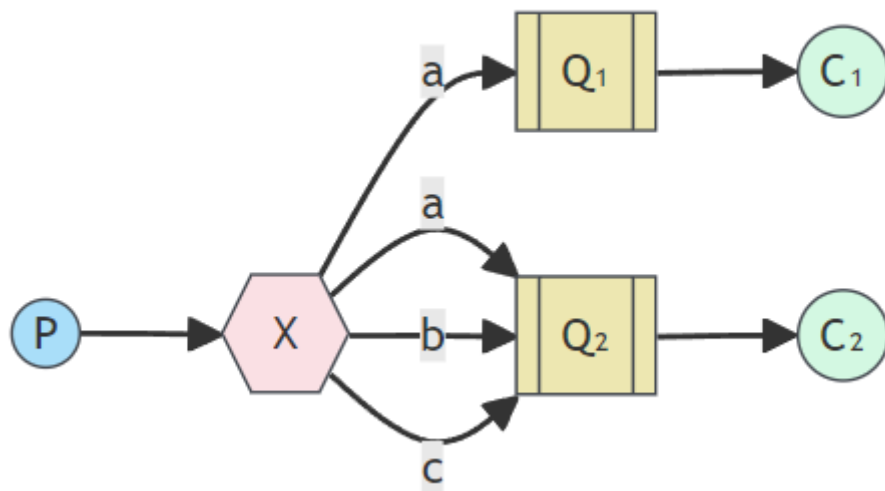
一个生产者P, 多个消费者C1, C2, X代表交换机消息复制多份, 每个消费者接收相同的消息

生产者发送一条消息, 经过交换机转发到多个不同的队列, 多个不同的队列就有多个不同的消费者

适合场景: 消息需要被多个消费者同时接收的场景. 如: 实时通知或者广播消息

比如中国气象局发布"天气预报"的消息送入交换机, 新浪, 百度, 搜狐, 网易等门户网站接入消息, 通过队列绑定到该交换机, 自动获取气象局推送的气象数据

1.4 Routing(路由模式)



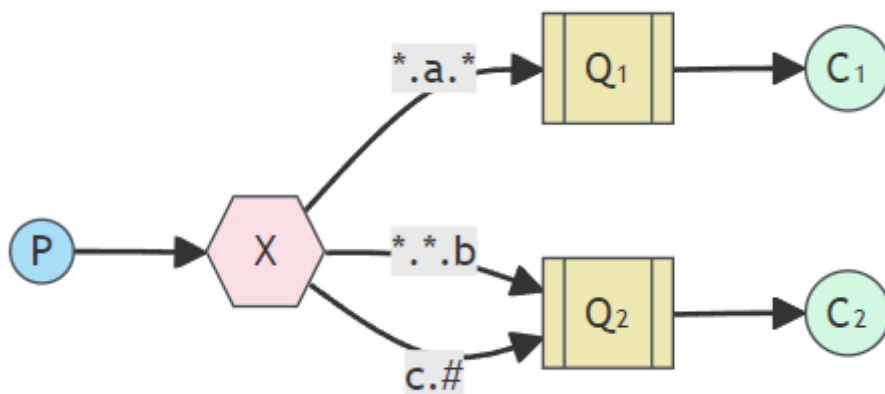
路由模式是发布订阅模式的变种, 在发布订阅基础上, 增加路由key

发布订阅模式是无条件的将所有消息分发给所有消费者, 路由模式是Exchange根据RoutingKey的规则, 将数据筛选后发给对应的消费者队列

适合场景: 需要根据特定规则分发消息的场景.

比如系统打印日志, 日志等级分为error, warning, info, debug, 就可以通过这种模式, 把不同的日志发送到不同的队列, 最终输出到不同的文件

1.5 Topics(通配符模式)



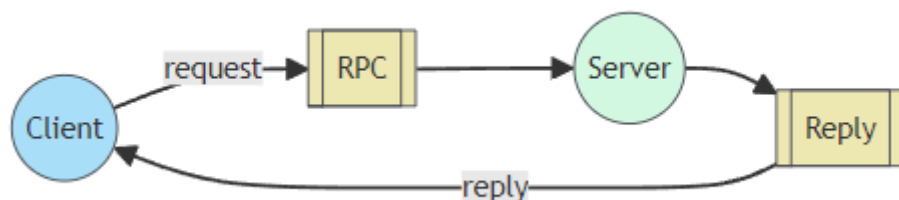
路由模式的升级版, 在routingKey的基础上, 增加了通配符的功能, 使之更加灵活.

Topics和Routing的基本原理相同, 即: 生产者将消息发给交换机, 交换机根据RoutingKey将消息转发给与RoutingKey匹配的队列. 类似于正则表达式的方式来定义Routingkey的模式.

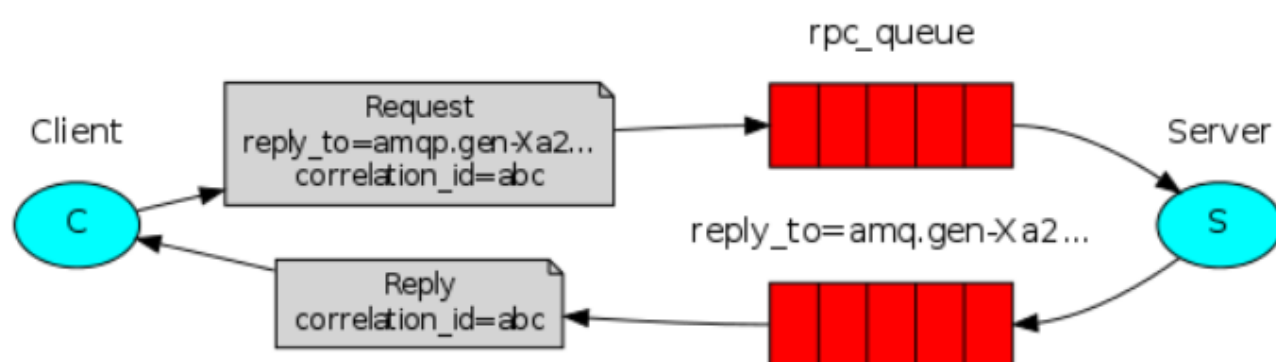
不同之处是：routingKey的匹配方式不同，Routing模式是相等匹配，topics模式是通配符匹配。

适合场景: 需要灵活匹配和过滤消息的场景

1.6 RPC(RPC通信)



在RPC通信的过程中, 没有生产者和消费者, 比较像咱们RPC远程调用, 大概就是通过两个队列实现了一个可回调的过程.



1. 客户端发送消息到一个指定的队列, 并在消息属性中设置replyTo字段, 这个字段指定了一个回调队列, 用于接收服务端的响应.
2. 服务端接收到请求后, 处理请求并发送响应消息到replyTo指定的回调队列
3. 客户端在回调队列上等待响应消息. 一旦收到响应, 客户端会检查消息的correlationId属性, 以确保它是所期望的响应.

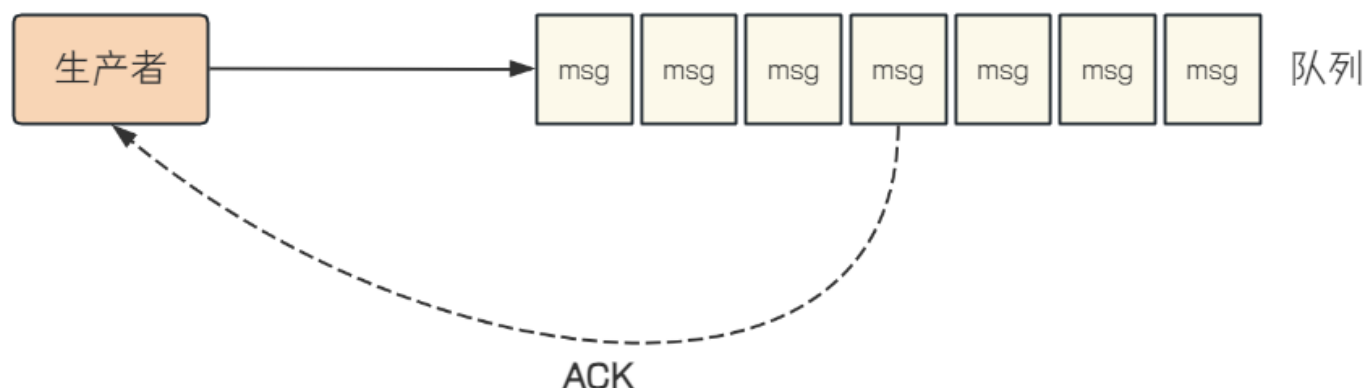
1.7 Publisher Confirms(发布确认)

Publisher Confirms模式是RabbitMQ提供的一种确保消息可靠发送到RabbitMQ服务器的机制。在这种模式下, 生产者可以等待RabbitMQ服务器的确认, 以确保消息已经被服务器接收并处理。

1. 生产者将Channel设置为confirm模式(通过调用channel.confirmSelect()完成后), 发布的每一条消息都会获得一个唯一的ID, 生产者可以将这些序列号与消息关联起来, 以便跟踪消息的状态.
2. 当消息被RabbitMQ服务器接收并处理后, 服务器会异步地向生产者发送一个确认(ACK)给生产者 (包含消息的唯一ID), 表明消息已经送达.

通过Publisher Confirms模式，生产者可以确保消息被RabbitMQ服务器成功接收, 从而避免消息丢失的问题.

适用场景: 对数据安全性要求较高的场景. 比如金融交易, 订单处理.

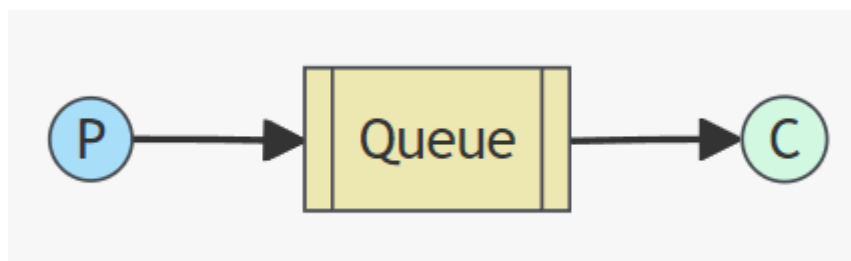


2. 工作模式的使用案例

咱们在前面学习了简单模式的写法, 接下来学习另外几种工作模式的写法

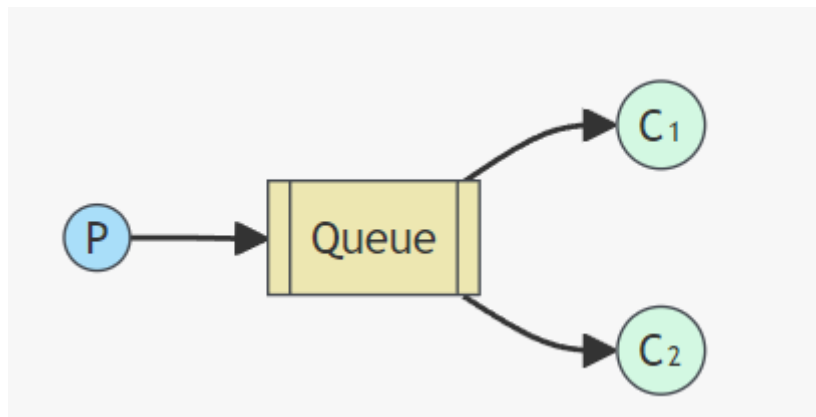
2.1 简单模式

快速入门程序就是简单模式. 此处省略.



2.2 Work Queues(工作队列)

简单模式的增强版, 和简单模式的区别就是: 简单模式有一个消费者, 工作队列模式支持多个消费者接收消息, 消费者之间是竞争关系, 每个消息只能被一个消费者接收



步骤:

1. 引入依赖
2. 编写生产者代码
3. 编写消费者代码

引入依赖

```
1 <dependency>
2   <groupId>com.rabbitmq</groupId>
3   <artifactId>amqp-client</artifactId>
4   <version>5.7.3</version>
5 </dependency>
```

编写生产者代码

工作队列模式和简单模式区别是有多个消费者,所以生产者消费者代码差异不大

相比简单模式,生产者的代码基本一样,为了能看到多个消费者竞争的关系,我们一次发送10条消息

我们把发送消息的地方,改为一次发送10条消息

```
1 for (int i = 0; i < 10; i++) {
2     String msg = "Hello World" + i;
3     channel.basicPublish("", QUEUE_NAME, null, msg.getBytes());
4 }
```

整体代码:

```
1 public class Constants {
2     public static final String HOST = "110.41.51.65";
```

```

3     public static final Integer PORT = 15673;
4     public static final String VIRTUAL_HOST = "bite";
5     public static final String USER_NAME = "study";
6     public static final String PASSWORD = "study";
7
8     public static final String WORK_QUEUE_NAME = "work_queues";
9 }

```

```

1 import com.rabbitmq.client.Channel;
2 import com.rabbitmq.client.Connection;
3 import com.rabbitmq.client.ConnectionFactory;
4 import constant.Constants;
5
6 public class WorkRabbitProducer {
7
8     public static void main(String[] args) throws Exception {
9         //1. 创建channel通道
10        ConnectionFactory factory = new ConnectionFactory();
11        factory.setHost(Constants.HOST); //ip 默认值localhost
12        factory.setPort(Constants.PORT); //默认值5672
13        factory.setVirtualHost(Constants.VIRTUAL_HOST); //虚拟机名称, 默认 /
14
15        factory.setUsername(Constants.USER_NAME); //用户名, 默认guest
16        factory.setPassword(Constants.PASSWORD); //密码, 默认guest
17        Connection connection = factory.newConnection();
18        Channel channel = connection.createChannel();
19        //2. 声明队列
20        //如果没有一个这样的队列, 会自动创建, 如果有, 则不创建
21        channel.queueDeclare(Constants.WORK_QUEUE_NAME, true, false, false,
22            null);
23        //3. 发送消息
24        for (int i = 0; i < 10; i++) {
25            String msg = "Hello World" + i;
26
27            channel.basicPublish("", Constants.WORK_QUEUE_NAME, null, msg.getBytes());
28        }
29        //4. 释放资源
30        channel.close();
31        connection.close();
32    }
33 }

```

编写消费者代码

消费者代码和简单模式一样, 只是复制两份. 两个消费者代码可以是一样的

```
1 import com.rabbitmq.client.*;
2 import constant.Constants;
3
4 import java.io.IOException;
5
6 public class WorkRabbitmqConsumer1 {
7
8     public static void main(String[] args) throws Exception {
9         //1. 创建channel通道
10        ConnectionFactory factory = new ConnectionFactory();
11        factory.setHost(Constants.HOST); //ip 默认值localhost
12        factory.setPort(Constants.PORT); //默认值5672
13        factory.setVirtualHost(Constants.VIRTUAL_HOST); //虚拟机名称, 默认 /
14
15        factory.setUsername(Constants.USER_NAME); //用户名, 默认guest
16        factory.setPassword(Constants.PASSWORD); //密码, 默认guest
17        Connection connection = factory.newConnection();
18        Channel channel = connection.createChannel();
19        //2. 声明队列
20        //如果没有一个这样的队列, 会自动创建, 如果有, 则不创建
21        channel.queueDeclare(Constants.WORK_QUEUE_NAME, true, false, false,
22            null);
23
24        //3. 接收消息, 并消费
25        DefaultConsumer consumer = new DefaultConsumer(channel) {
26            @Override
27            public void handleDelivery(String consumerTag, Envelope envelope,
28                AMQP.BasicProperties properties, byte[] body) throws IOException {
29                System.out.println("接收到消息: " + new String(body));
30            }
31        };
32        channel.basicConsume(Constants.WORK_QUEUE_NAME, true, consumer);
33    }
```

运行程序, 观察结果

先启动两个消费者运行, 再启动生产者

如果先启动生产者, 在启动消费者, 由于消息较少, 处理较快, 那么第一个启动的消费者就会瞬间把10条消息消费掉, 所以我们先启动两个消费者, 再启动生产者

1. 启动2个消费者

2. 启动生产者

可以看到两个消费者都打印了消费信息

WorkQueuesConsumer1 打印

```
1 body:Hello World0
2 body:Hello World2
3 body:Hello World4
4 body:Hello World6
5 body:Hello World8
```

WorkQueuesConsumer2 打印

```
1 body:Hello World1
2 body:Hello World3
3 body:Hello World5
4 body:Hello World7
5 body:Hello World9
```

可以看到管理界面上显示两个消费者

OverviewConnectionsChannelsExchangesQueuesAdmin

Queues

▼ All queues (2)

Page 1 of 1 - Filter: ☐ Regex ?

Overview							Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer capacity	State	Ready	Unacked	Total	incoming	deliver / get	ack	
bite	hello_world	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	work_queues	classic	D	2	100%	idle	0	0	0	0.00/s	0.00/s	0.00/s	

► Add a new queue

2.3 Publish/Subscribe(发布/订阅)

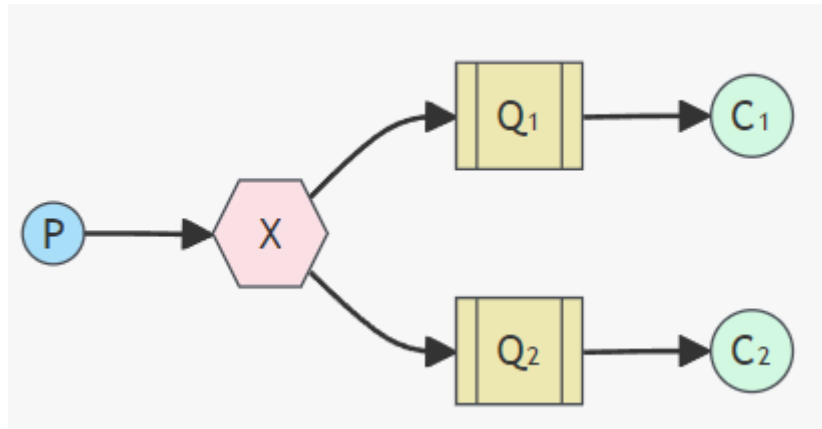
在发布/订阅模型中，多了一个Exchange角色.

Exchange 常见有三种类型, 分别代表不同的路由规则

- a) Fanout:广播，将消息交给所有绑定到交换机的队列(Publish/Subscribe模式)
- b) Direct:定向，把消息交给符合指定routing key的队列(Routing模式)
- c) Topic:通配符，把消息交给符合routing pattern(路由模式)的队列(Topics模式)

也就分别对应不同的工作模式

我们来看看Publish/Subscribe 模式



步骤:

1. 引入依赖
2. 编写生产者代码
3. 编写消费者代码

引入依赖

```
1 <dependency>
2   <groupId>com.rabbitmq</groupId>
3   <artifactId>amqp-client</artifactId>
4   <version>5.7.3</version>
5 </dependency>
```

编写生产者代码

和前面两个的区别是:

需要创建交换机, 并且绑定队列和交换机

创建交换机

```
1 /*
2  exchangeDeclare(String exchange, BuiltinExchangeType type, boolean durable,
3    boolean autoDelete, boolean internal, Map<String, Object> arguments)
4  参数:
5  1. exchange:交换机名称
6  2. type:交换机类型
7    * DIRECT("direct"), 定向,直连, routing
8    * FANOUT("fanout"), 扇形(广播), 每个队列都能收到消息
9    * TOPIC("topic"), 通配符
10   * HEADERS("headers") 参数匹配(工作用的较少)
```

```

10 3. durable: 是否持久化.
11     true-持久化, false非持久化.
12     持久化可以将交换器存盘, 在服务器重启的时候不会丢失相关信息
13 4. autoDelete: 自动删除.
14     自动删除的前提是至少有一个队列或者交换器与这个交换器绑定, 之后所有与这个交换器绑定的
    队列或者交换器都与此解绑.
15     而不是这种理解: 当与此交换器连接的客户端都断开时, RabbitMQ会自动删除本交换器.
16 5. internal: 内部使用, 一般false.
17     如果设置为true, 表示内部使用.
18     客户端程序无法直接发送消息到这个交换器中, 只能通过交换器路由到交换器这种方式
19 6. arguments: 参数
20 */
21
22 channel.exchangeDeclare(Constants.FANOUT_EXCHANGE_NAME,
    BuiltinExchangeType.FANOUT, true, false, false, null);

```

声明两个队列

后面验证是否两个队列都能收到消息

```

1 //如果没有一个这样的队列, 会自动创建, 如果有, 则不创建
2 channel.queueDeclare(Constants.FANOUT_QUEUE_NAME1, true, false, false, null);
3 channel.queueDeclare(Constants.FANOUT_QUEUE_NAME2, true, false, false, null);

```

绑定队列和交换机

```

1 /*
2 queueBind(String queue, String exchange, String routingKey)
3 参数:
4 1. queue: 队列名称
5 2. exchange: 交换机名称
6 3. routingKey: 路由key, 路由规则
7     如果交换机类型为fanout, routingkey设置为"", 表示每个消费者都可以收到全部信息
8 */
9 channel.queueBind(Constants.FANOUT_QUEUE_NAME1, Constants.FANOUT_EXCHANGE_NAME,
    "");
10 channel.queueBind(Constants.FANOUT_QUEUE_NAME2, Constants.FANOUT_EXCHANGE_NAME,
    "");

```

发送消息

```

1 /**

```

```

2  * basicPublish(String exchange, String routingKey, AMQP.BasicProperties
    props, byte[] body)
3  * 参数说明:
4  * Exchange: 交换机名称
5  * routingKey: 如果交换机类型为fanout, routingKey设置为"", 表示每个消费者都可以收到全部
    信息
6  */
7  String msg = "hello fanout";
8  channel.basicPublish(Constants.FANOUT_EXCHANGE_NAME, "", null, msg.getBytes());

```

完整代码

```

1  public static String FANOUT_EXCHANGE_NAME = "test_fanout";
2  public static String FANOUT_QUEUE_NAME1 = "fanout_queue1";
3  public static String FANOUT_QUEUE_NAME2 = "fanout_queue2";

```

```

1  import com.rabbitmq.client.BuiltinExchangeType;
2  import com.rabbitmq.client.Channel;
3  import com.rabbitmq.client.Connection;
4  import com.rabbitmq.client.ConnectionFactory;
5  import constant.Constants;
6
7  public class FanoutRabbitProducer {
8
9      public static void main(String[] args) throws Exception {
10         //1. 创建channel通道
11         ConnectionFactory factory = new ConnectionFactory();
12         factory.setHost(Constants.HOST); //ip 默认值localhost
13         factory.setPort(Constants.PORT); //默认值5672
14         factory.setVirtualHost(Constants.VIRTUAL_HOST); //虚拟机名称, 默认 /
15
16         factory.setUsername(Constants.USER_NAME); //用户名, 默认guest
17         factory.setPassword(Constants.PASSWORD); //密码, 默认guest
18         Connection connection = factory.newConnection();
19         Channel channel = connection.createChannel();
20         //2. 创建交换机
21         /*
22         exchangeDeclare(String exchange, BuiltinExchangeType type, boolean
            durable, boolean autoDelete, boolean internal, Map<String, Object> arguments)
23         参数:
24         1. exchange: 交换机名称
25         2. type: 交换机类型
26         * DIRECT("direct"), 定向, 直连, routing

```

```

27         * FANOUT("fanout"),扇形(广播), 每个队列都能收到消息
28         * TOPIC("topic"),通配符
29         * HEADERS("headers") 参数匹配(工作用的较少)
30     3. durable: 是否持久化
31     4. autoDelete: 自动删除
32     5. internal: 内部使用, 一般false
33     6. arguments: 参数
34     */
35
36     channel.exchangeDeclare(Constants.FANOUT_EXCHANGE_NAME,
BuiltinExchangeType.FANOUT, true, false, false, null);
37     //3. 声明队列
38     //如果没有一个这样的队列, 会自动创建, 如果有, 则不创建
39     channel.queueDeclare(Constants.FANOUT_QUEUE_NAME1, true, false, false,
null);
40     channel.queueDeclare(Constants.FANOUT_QUEUE_NAME2, true, false, false,
null);
41     //4. 绑定队列和交换机
42     /*
43     queueBind(String queue, String exchange, String routingKey,
Map<String, Object> arguments)
44     参数:
45     1. queue: 队列名称
46     2. exchange: 交换机名称
47     3. routingKey: 路由key, 路由规则
48         如果交换机类型为fanout, routingkey设置为"", 表示每个消费者都可以收到全部信息
49     */
50
51     channel.queueBind(Constants.FANOUT_QUEUE_NAME1, Constants.FANOUT_EXCHANGE_NAME,
"");
52
53     channel.queueBind(Constants.FANOUT_QUEUE_NAME2, Constants.FANOUT_EXCHANGE_NAME,
"");
54
55     //5. 发送消息
56     /**
57     * basicPublish(String exchange, String routingKey,
AMQP.BasicProperties props, byte[] body)
58     * 参数说明:
59     * Exchange: 交换机名称
60     * routingKey: 如果交换机类型为fanout, routingkey设置为"", 表示每个消费者都可以
收到全部信息
61     */
62     String msg = "hello fanout";
63
64     channel.basicPublish(Constants.FANOUT_EXCHANGE_NAME, "", null, msg.getBytes());
65
66     //6. 释放资源

```



```
63         channel.close();
64         connection.close();
65     }
66 }
```

编写消费者代码

交换机和队列的绑定关系及声明已经在生产方写完, 所以消费者不需要再写了

去掉声明队列的代码就可以了

1. 创建Channel
2. 接收消息, 并处理

完整代码

消费者1

```
1  import com.rabbitmq.client.*;
2  import constant.Constants;
3
4  import java.io.IOException;
5
6  public class FanoutRabbitmqConsumer1 {
7
8      public static void main(String[] args) throws Exception {
9          //1. 创建channel通道
10         ConnectionFactory factory = new ConnectionFactory();
11         factory.setHost(Constants.HOST); //ip 默认值localhost
12         factory.setPort(Constants.PORT); //默认值5672
13         factory.setVirtualHost(Constants.VIRTUAL_HOST); //虚拟机名称, 默认 /
14
15         factory.setUsername(Constants.USER_NAME); //用户名, 默认guest
16         factory.setPassword(Constants.PASSWORD); //密码, 默认guest
17         Connection connection = factory.newConnection();
18         Channel channel = connection.createChannel();
19
20         //2. 接收消息, 并消费
21         DefaultConsumer consumer = new DefaultConsumer(channel) {
22             @Override
23             public void handleDelivery(String consumerTag, Envelope envelope,
24                 AMQP.BasicProperties properties, byte[] body) throws IOException {
25                 System.out.println("接收到消息: " + new String(body));
26             }
27         };
28         channel.basicConsume(Constants.FANOUT_QUEUE_NAME1, true, consumer);
29     }
30 }
```

消费者2 把队列名称改一下就可以了. 此处省略

运行程序, 观察结果

1. 运行生产者

a) 可以看到两个队列分别有了一条消息

Overview

Connections

Channels

Exchanges

Queues

Admin

Queues

▼ All queues (4)

Pagination

Page 1 of 1

- Filter:

☐ Regexp ?

Overview							Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack	
bite	fanout_queue1	classic	D	0	0%	idle	1	0	1	0.20/s	0.00/s	0.00/s	
bite	fanout_queue2	classic	D	0	0%	idle	1	0	1	0.20/s	0.00/s	0.00/s	
bite	hello	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	work_queues	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	

► Add a new queue

b) Exchange多了队列绑定关系

Overview	Connections	Channels	Exchanges	Queues	Admin
----------	-------------	----------	-----------	--------	-------

Details

Type

fanout

Features

durable: true

Policy

▼ Bindings

This exchange

⇓

To	Routing key	Arguments	
fanout_queue1			Unbind
fanout_queue2			Unbind

Add binding from this exchange

To queue ▼

:

*

Routing key:

Arguments:

=

String ▼

Bind

2. 运行消费者

消费者1

```
1 接收到消息: hello fanout
```

消费者2

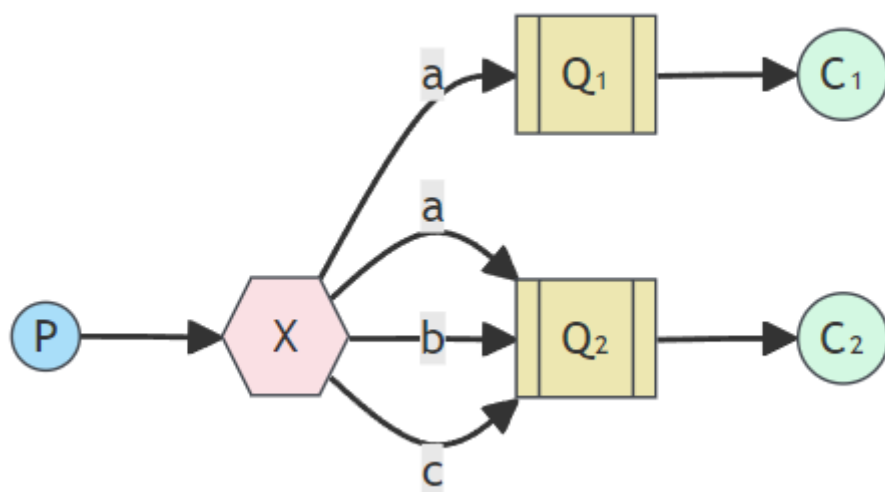
```
1 接收到消息: hello fanout
```

2.4 Routing(路由模式)

队列和交换机的绑定, 不能是任意的绑定了, 而是要指定一个BindingKey(RoutingKey的一种)

消息的发送方在向Exchange发送消息时, 也需要指定消息的RoutingKey

Exchange也不再把消息交给每一个绑定的key, 而是根据消息的RoutingKey进行判断, 只有队列绑定时的BindingKey和发送消息的RoutingKey 完全一致, 才会接收到消息



接下来我们看看Routing模式的实现

步骤:

1. 引入依赖
2. 编写生产者代码
3. 编写消费者代码

引入依赖

```
1 <dependency>
2     <groupId>com.rabbitmq</groupId>
3     <artifactId>amqp-client</artifactId>
4     <version>5.7.3</version>
5 </dependency>
```

编写生产者代码

和发布订阅模式的区别是: 交换机类型不同, 绑定队列的BindingKey不同

创建交换机, 定义交换机类型为BuiltinExchangeType.DIRECT

```
1 channel.exchangeDeclare(Constants.DIRECT_EXCHANGE_NAME,
    BuiltinExchangeType.DIRECT, true, false, false, null);
```

声明队列

```
1 channel.queueDeclare(Constants.DIRECT_QUEUE_NAME1, true, false, false, null);
2 channel.queueDeclare(Constants.DIRECT_QUEUE_NAME2, true, false, false, null);
```

绑定交换机和队列

```
1 //队列1绑定orange
2 channel.queueBind(Constants.DIRECT_QUEUE_NAME1, Constants.DIRECT_EXCHANGE_NAME,
    "orange");
3 //队列2绑定black, green
4 channel.queueBind(Constants.DIRECT_QUEUE_NAME2, Constants.DIRECT_EXCHANGE_NAME,
    "black");
5 channel.queueBind(Constants.DIRECT_QUEUE_NAME2, Constants.DIRECT_EXCHANGE_NAME,
    "green");;
```

发送消息

```
1 //发送消息时, 指定RoutingKey
2 String msg = "hello direct, I am orange";
3 channel.basicPublish(Constants.DIRECT_EXCHANGE_NAME, "orange", null, msg.getBytes(
    ));
4
5 String msg_black = "hello direct, I am black";
```

```

6 channel.basicPublish(Constants.DIRECT_EXCHANGE_NAME,"black",null,msg_black.getBytes());
7
8 String msg_green= "hello direct, I am green";
9 channel.basicPublish(Constants.DIRECT_EXCHANGE_NAME,"green",null,msg_green.getBytes());

```

完整代码:

```

1 public static String DIRECT_EXCHANGE_NAME = "test_direct";
2 public static String DIRECT_QUEUE_NAME1 = "direct_queue1";
3 public static String DIRECT_QUEUE_NAME2 = "direct_queue2";

```

```

1
2 import com.rabbitmq.client.BuiltinExchangeType;
3 import com.rabbitmq.client.Channel;
4 import com.rabbitmq.client.Connection;
5 import com.rabbitmq.client.ConnectionFactory;
6 import constant.Constants;
7
8 public class DirectRabbitProducer {
9
10     public static void main(String[] args) throws Exception {
11         //1. 创建channel通道
12         ConnectionFactory factory = new ConnectionFactory();
13         factory.setHost(Constants.HOST); //ip 默认值localhost
14         factory.setPort(Constants.PORT); //默认值5672
15         factory.setVirtualHost(Constants.VIRTUAL_HOST); //虚拟机名称, 默认 /
16
17         factory.setUsername(Constants.USER_NAME); //用户名, 默认guest
18         factory.setPassword(Constants.PASSWORD); //密码, 默认guest
19         Connection connection = factory.newConnection();
20         Channel channel = connection.createChannel();
21         //2. 创建交换机
22         channel.exchangeDeclare(Constants.DIRECT_EXCHANGE_NAME,
            BuiltinExchangeType.FANOUT, true, false, false, null);
23         //3. 声明队列
24         //如果没有一个这样的队列, 会自动创建, 如果有, 则不创建
25         channel.queueDeclare(Constants.DIRECT_QUEUE_NAME1, true, false, false,
            null);
26         channel.queueDeclare(Constants.DIRECT_QUEUE_NAME2, true, false, false,
            null);

```

```

27         //4. 绑定队列和交换机
28         //队列1绑定orange
29
30         channel.queueBind(Constants.DIRECT_QUEUE_NAME1, Constants.DIRECT_EXCHANGE_NAME,
31             "orange");
32         //队列2绑定black, green
33
34         channel.queueBind(Constants.DIRECT_QUEUE_NAME2, Constants.DIRECT_EXCHANGE_NAME,
35             "black");
36
37         channel.queueBind(Constants.DIRECT_QUEUE_NAME2, Constants.DIRECT_EXCHANGE_NAME,
38             "green");
39
40         //5. 发送消息
41         String msg = "hello direct, I am orange";
42
43         channel.basicPublish(Constants.DIRECT_EXCHANGE_NAME, "orange", null, msg.getBytes(
44             ));
45
46         String msg_black = "hello direct,I am black";
47
48         channel.basicPublish(Constants.DIRECT_EXCHANGE_NAME, "black", null, msg_black.getB
49             ytes());
50
51         String msg_green= "hello direct, I am green";
52
53         channel.basicPublish(Constants.DIRECT_EXCHANGE_NAME, "green", null, msg_green.getB
54             ytes());
55
56         //6. 释放资源
57         channel.close();
58         connection.close();
59     }
60 }

```

编写消费者代码

Routing模式的消费者代码和Publish/Subscribe 代码一样, 同样复制出来两份

消费者1: DirectRabbitmqConsumer1

消费者2: DirectRabbitmqConsumer2

修改消费的队列名称就可以

完整代码:

```

1
2 import com.rabbitmq.client.*;
3 import constant.Constants;
4
5 import java.io.IOException;
6
7 public class DirectRabbitmqConsumer1 {
8
9     public static void main(String[] args) throws Exception {
10         //1. 创建channel通道
11         ConnectionFactory factory = new ConnectionFactory();
12         factory.setHost(Constants.HOST); //ip 默认值localhost
13         factory.setPort(Constants.PORT); //默认值5672
14         factory.setVirtualHost(Constants.VIRTUAL_HOST); //虚拟机名称, 默认 /
15
16         factory.setUsername(Constants.USER_NAME); //用户名, 默认guest
17         factory.setPassword(Constants.PASSWORD); //密码, 默认guest
18         Connection connection = factory.newConnection();
19         Channel channel = connection.createChannel();
20
21         //2. 接收消息, 并消费
22         DefaultConsumer consumer = new DefaultConsumer(channel) {
23             @Override
24             public void handleDelivery(String consumerTag, Envelope envelope,
25             AMQP.BasicProperties properties, byte[] body) throws IOException {
26                 System.out.println("接收到消息: " + new String(body));
27             }
28         };
29         channel.basicConsume(Constants.DIRECT_QUEUE_NAME1, true, consumer);
30     }
31 }

```

运行程序, 观察结果

1. 运行生产者

a) 可以看到direct_queue1队列中, 路由了一条消息. direct_queue2队列中, 路由了一条消息

Queues

▼ All queues (6)

Pagination

Page 1 ▼ of 1 - Filter: ☐ Regex ?

Overview							Messages			Message rates			+
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	Incoming	deliver / get	ack	
bite	direct_queue1	classic	D	0	0%	idle	1	0	1	0.00/s			
bite	direct_queue2	classic	D	0	0%	idle	2	0	2	0.00/s			
bite	fanout_queue1	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	fanout_queue2	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	hello	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	work_queues	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	

b) exchange下队列和Routing Key的绑定关系

OverviewConnectionsChannelsExchangesQueuesAdmin

Policy

▼ Bindings

This exchange

⇓

To	Routing key	Arguments	
direct_queue1	orange		Unbind
direct_queue2	black		Unbind
direct_queue2	green		Unbind

2. 运行消费者

DirectRabbitmqConsumer1:

1 接收到消息: hello direct, I am orange

DirectRabbitmqConsumer2:

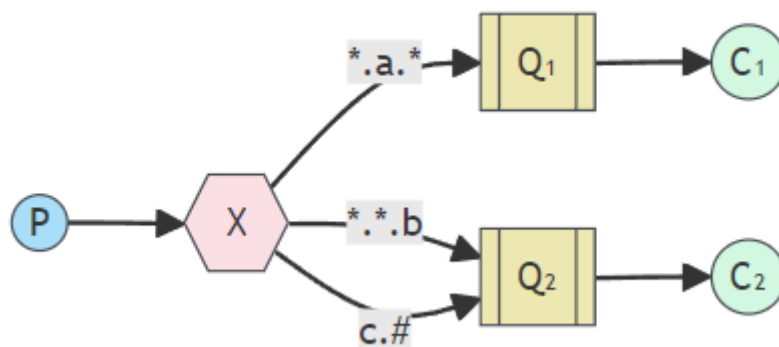
1 接收到消息: hello direct,I am black
2 接收到消息: hello direct, I am green

2.5 Topics(通配符模式)

Topics 和Routing模式的区别是:

- 1. topics 模式使用的交换机类型为topic(Routing模式使用的交换机类型为direct)

2. topic 类型的交换机在匹配规则上进行了扩展, Binding Key支持通配符匹配(direct类型的交换机路由规则是BindingKey和RoutingKey完全匹配).



在topic类型的交换机在匹配规则上, 有些要求:

1. RoutingKey 是一系列由点(`.`)分隔的单词, 比如 `"stock.usd.nyse"`, `"nyse.vmw"`, `"quick.orange.rabbit"`
2. BindingKey 和RoutingKey一样, 也是点(`.`)分割的字符串.
3. Binding Key中可以存在两种特殊字符串, 用于模糊匹配
 - `*` 表示一个单词
 - `#` 表示多个单词(0-N个)

比如:

- Binding Key 为 `"d.a.b"` 会同时路由到 `Q1` 和 `Q2`
- Binding Key 为 `"d.a.f"` 会路由到 `Q1`
- Binding Key 为 `"c.e.f"` 会路由到 `Q2`
- Binding Key 为 `"d.b.f"` 会被丢弃, 或者返回给生产者(需要设置 `mandatory` 参数)

接下来我们看看Routing模式的实现

步骤:

1. 引入依赖
2. 编写生产者代码
3. 编写消费者代码

引入依赖

```
1 <dependency>
2     <groupId>com.rabbitmq</groupId>
3     <artifactId>amqp-client</artifactId>
4     <version>5.7.3</version>
5 </dependency>
```

编写生产者代码

和路由模式, 发布订阅模式的区别是: 交换机类型不同, 绑定队列的RoutingKey不同

创建交换机

定义交换机类型为BuiltinExchangeType.TOPIC

```
1 channel.exchangeDeclare(Constants.TOPIC_EXCHANGE_NAME,
    BuiltinExchangeType.TOPIC, true, false, false, null);
```

声明队列

```
1 channel.queueDeclare(Constants.TOPIC_QUEUE_NAME1, true, false, false, null);
2 channel.queueDeclare(Constants.TOPIC_QUEUE_NAME2, true, false, false, null);
```

绑定交换机和队列

```
1 //队列1绑定error, 仅接收error信息
2 channel.queueBind(Constants.TOPIC_QUEUE_NAME1, Constants.TOPIC_EXCHANGE_NAME,
    "*.error");
3 //队列2绑定info, error: error, info信息都接收
4 channel.queueBind(Constants.TOPIC_QUEUE_NAME2, Constants.TOPIC_EXCHANGE_NAME,
    "#.info");
5 channel.queueBind(Constants.TOPIC_QUEUE_NAME2, Constants.TOPIC_EXCHANGE_NAME,
    "*.error");
```

发送消息

```
1 String msg = "hello topic, I'm order.error";
2 channel.basicPublish(Constants.TOPIC_EXCHANGE_NAME, "order.error", null, msg.getBytes());
3
4 String msg_black = "hello topic, I'm order.pay.info";
```

```

5 channel.basicPublish(Constants.TOPIC_EXCHANGE_NAME,"order.pay.info",null,msg_bla
  ack.getBytes());
6
7 String msg_green= "hello topic, I'm pay.error";
8 channel.basicPublish(Constants.TOPIC_EXCHANGE_NAME,"pay.error",null,msg_green.g
  etBytes());

```

完整代码:

```

1 public static String TOPIC_EXCHANGE_NAME = "test_topic";
2 public static String TOPIC_QUEUE_NAME1 = "topic_queue1";
3 public static String TOPIC_QUEUE_NAME2 = "topic_queue2";

```

```

1 import com.rabbitmq.client.BuiltinExchangeType;
2 import com.rabbitmq.client.Channel;
3 import com.rabbitmq.client.Connection;
4 import com.rabbitmq.client.ConnectionFactory;
5 import constant.Constants;
6
7 public class TopicRabbitProducer {
8
9     public static void main(String[] args) throws Exception {
10         //1. 创建channel通道
11         ConnectionFactory factory = new ConnectionFactory();
12         factory.setHost(Constants.HOST);//ip 默认值localhost
13         factory.setPort(Constants.PORT); //默认值5672
14         factory.setVirtualHost(Constants.VIRTUAL_HOST);//虚拟机名称, 默认 /
15
16         factory.setUsername(Constants.USER_NAME);//用户名,默认guest
17         factory.setPassword(Constants.PASSWORD);//密码, 默认guest
18         Connection connection = factory.newConnection();
19         Channel channel = connection.createChannel();
20         //2. 创建交换机
21         channel.exchangeDeclare(Constants.TOPIC_EXCHANGE_NAME,
BuiltinExchangeType.TOPIC, true, false, false, null);
22         //3. 声明队列
23         //如果没有一个这样的队列, 会自动创建, 如果有, 则不创建
24         channel.queueDeclare(Constants.TOPIC_QUEUE_NAME1, true, false, false,
null);
25         channel.queueDeclare(Constants.TOPIC_QUEUE_NAME2, true, false, false,
null);
26         //4. 绑定队列和交换机
27         //队列1绑定error, 仅接收error信息

```

```

28     channel.queueBind(Constants.TOPIC_QUEUE_NAME1, Constants.TOPIC_EXCHANGE_NAME,
29         "*.error");
30         //队列2绑定info, error: error, info信息都接收
31
32     channel.queueBind(Constants.TOPIC_QUEUE_NAME2, Constants.TOPIC_EXCHANGE_NAME,
33         "#.info");
34
35     channel.queueBind(Constants.TOPIC_QUEUE_NAME2, Constants.TOPIC_EXCHANGE_NAME,
36         "*.error");
37
38         //5. 发送消息
39         String msg = "hello topic, I'm order.error";
40
41     channel.basicPublish(Constants.TOPIC_EXCHANGE_NAME, "order.error", null, msg.getBytes());
42
43         String msg_black = "hello topic, I'm order.pay.info";
44
45     channel.basicPublish(Constants.TOPIC_EXCHANGE_NAME, "order.pay.info", null, msg_black.getBytes());
46
47         String msg_green = "hello topic, I'm pay.error";
48
49     channel.basicPublish(Constants.TOPIC_EXCHANGE_NAME, "pay.error", null, msg_green.getBytes());
50
51         //6. 释放资源
52     channel.close();
53     connection.close();
54 }
55 }

```

编写消费者代码

Routing模式的消费者代码和Routing模式代码一样, 修改消费的队列名称即可.

同样复制出来两份

消费者1: TopicRabbitmqConsumer1

消费者2: TopicRabbitmqConsumer2

完整代码:

```

1 import com.rabbitmq.client.*;
2 import constant.Constants;
3
4 import java.io.IOException;
5
6 public class TopicRabbitmqConsumer1 {
7
8     public static void main(String[] args) throws Exception {
9         //1. 创建channel通道
10        ConnectionFactory factory = new ConnectionFactory();
11        factory.setHost(Constants.HOST); //ip 默认值localhost
12        factory.setPort(Constants.PORT); //默认值5672
13        factory.setVirtualHost(Constants.VIRTUAL_HOST); //虚拟机名称, 默认 /
14
15        factory.setUsername(Constants.USER_NAME); //用户名, 默认guest
16        factory.setPassword(Constants.PASSWORD); //密码, 默认guest
17        Connection connection = factory.newConnection();
18        Channel channel = connection.createChannel();
19
20        //2. 接收消息, 并消费
21        DefaultConsumer consumer = new DefaultConsumer(channel) {
22            @Override
23            public void handleDelivery(String consumerTag, Envelope envelope,
24                AMQP.BasicProperties properties, byte[] body) throws IOException {
25                System.out.println("接收到消息: " + new String(body));
26            }
27        };
28        channel.basicConsume(Constants.TOPIC_QUEUE_NAME1, true, consumer);
29    }

```

运行程序, 观察结果

1. 运行生产者, 可以看到队列的消息数

Overview							Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack	
bite	direct_queue1	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	direct_queue2	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	fanout_queue1	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	fanout_queue2	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	hello	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	topic_queue1	classic	D	0	0%	idle	2	0	2	0.00/s			
bite	topic_queue2	classic	D	0	0%	idle	3	0	3	0.00/s			
bite	work_queues	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	

2. 运行消费者:

TopicRabbitmqConsumer1:

- 1 接收到消息: hello topic, I'm order.error
- 2 接收到消息: hello topic, I'm pay.error

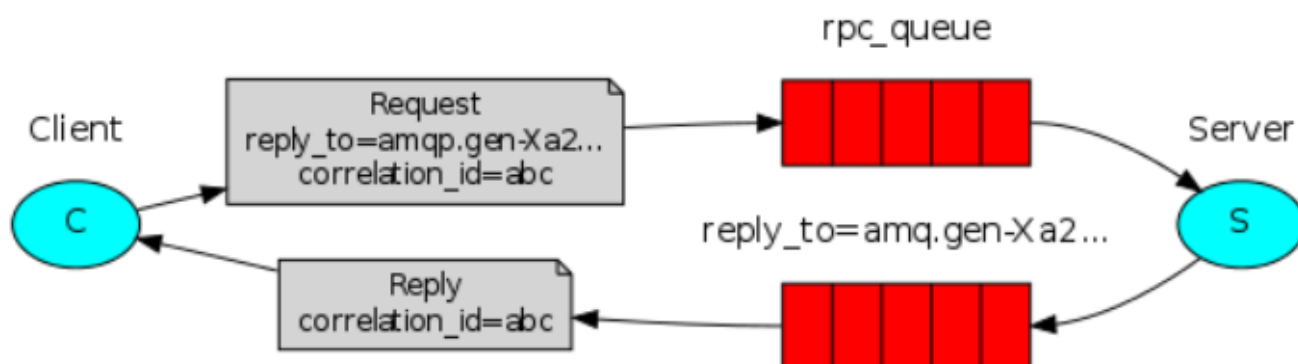
TopicRabbitmqConsumer2:

- 1 接收到消息: hello topic, I'm order.error
- 2 接收到消息: hello topic, I'm order.pay.info
- 3 接收到消息: hello topic, I'm pay.error

2.6 RPC(RPC通信)

RPC(Remote Procedure Call), 即远程过程调用. 它是一种通过网络从远程计算机上请求服务, 而不需要了解底层网络的技术. 类似于Http远程调用.

RabbitMQ实现RPC通信的过程, 大概是通过两个队列实现一个可回调的过程.



大概流程如下:

1. 客户端发送消息到一个指定的队列, 并在消息属性中设置replyTo字段, 这个字段指定了一个回调队列, 服务端处理后, 会把响应结果发送到这个队列.
2. 服务端接收到请求后, 处理请求并发送响应消息到replyTo指定的回调队列
3. 客户端在回调队列上等待响应消息. 一旦收到响应, 客户端会检查消息的correlationId属性, 以确保它是所期望的响应.

接下来我们看看RPC模式的实现

步骤:

1. 引入依赖

2. 编写客户端

3. 编写服务端

引入依赖

```
1 <dependency>
2     <groupId>com.rabbitmq</groupId>
3     <artifactId>amqp-client</artifactId>
4     <version>5.7.3</version>
5 </dependency>
```

编写客户端代码

客户端代码主要流程如下:

1. 声明两个队列, 包含回调队列replyQueueName, 声明本次请求的唯一标志corrId
2. 将replyQueueName和corrId配置到要发送的消息队列中
3. 使用阻塞队列来阻塞当前进程, 监听回调队列中的消息, 把请求放到阻塞队列中
4. 阻塞队列有消息后, 主线程被唤醒, 打印返回内容

声明队列

```
1 //2. 声明队列, 发送消息
2 channel.queueDeclare(Constants.RPC_REQUEST_QUEUE_NAME, true, false, false,
3     null);
```

定义回调队列

```
1 // 定义临时队列, 并返回生成的队列名称
2 String replyQueueName = channel.queueDeclare().getQueue();
```

使用内置交换机发送消息

```
1 // 本次请求唯一标志
2 String corrId = UUID.randomUUID().toString();
3 // 生成发送消息的属性
4 AMQP.BasicProperties props = new AMQP.BasicProperties
5     .Builder()
6     .correlationId(corrId) // 唯一标志本次请求
```

```

7         .replyTo(replyQueueName) // 设置回调队列
8         .build();
9 // 通过内置交换机，发送消息
10 String message = "hello rpc...";
11 channel.basicPublish("", Constants.RPC_REQUEST_QUEUE_NAME, props,
    message.getBytes());

```

使用阻塞队列, 来存储回调结果

```

1 // 阻塞队列, 用于存储回调结果
2 final BlockingQueue<String> response = new ArrayBlockingQueue<>(1);
3 //接收服务端的响应
4 DefaultConsumer consumer = new DefaultConsumer(channel) {
5     @Override
6     public void handleDelivery(String consumerTag, Envelope envelope,
    AMQP.BasicProperties properties, byte[] body) throws IOException {
7         System.out.println("接收到回调消息:" + new String(body));
8         //如果唯一标识正确, 放到阻塞队列中
9         if (properties.getCorrelationId().equals(correlationId)) {
10             response.offer(new String(body, "UTF-8"));
11         }
12     }
13 };
14 channel.basicConsume(replyQueueName, true, consumer);

```

获取回调结果

```

1 // 获取回调的结果
2 String result = response.take();
3 System.out.println(" [RPCClient] Result:" + result);

```

完整代码

```

1 public static String RPC_REQUEST_QUEUE_NAME = "rpc_request_queue";

```

```

1 import com.rabbitmq.client.*;
2 import constant.Constants;
3
4 import java.io.IOException;

```



```

5 import java.util.UUID;
6 import java.util.concurrent.ArrayBlockingQueue;
7 import java.util.concurrent.BlockingQueue;
8
9 public class RPCClient {
10
11     public static void main(String[] args) throws Exception {
12         //1. 创建Channel通道
13         ConnectionFactory factory = new ConnectionFactory();
14         factory.setHost(Constants.HOST); //ip 默认值localhost
15         factory.setPort(Constants.PORT); //默认值5672
16         factory.setVirtualHost(Constants.VIRTUAL_HOST); //虚拟机名称, 默认 /
17
18         factory.setUsername(Constants.USER_NAME); //用户名, 默认guest
19         factory.setPassword(Constants.PASSWORD); //密码, 默认guest
20         Connection connection = factory.newConnection();
21         Channel channel = connection.createChannel();
22         //2. 声明队列
23         channel.queueDeclare(Constants.RPC_REQUEST_QUEUE_NAME, true, false,
false, null);
24
25         // 唯一标志本次请求
26         String corrId = UUID.randomUUID().toString();
27         // 定义临时队列, 并返回生成的队列名称
28         String replyQueueName = channel.queueDeclare().getQueue();
29         // 生成发送消息的属性
30         AMQP.BasicProperties props = new AMQP.BasicProperties
31             .Builder()
32             .correlationId(corrId) // 唯一标志本次请求
33             .replyTo(replyQueueName) // 设置回调队列
34             .build();
35         // 通过内置交换机, 发送消息
36         String message = "hello rpc...";
37         channel.basicPublish("", Constants.RPC_REQUEST_QUEUE_NAME, props,
message.getBytes());
38
39         // 阻塞队列, 用于存储回调结果
40         final BlockingQueue<String> response = new ArrayBlockingQueue<>(1);
41         //接收服务端的响应
42         DefaultConsumer consumer = new DefaultConsumer(channel) {
43             @Override
44             public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
45                 System.out.println("接收到回调消息:" + new String(body));
46                 if (properties.getCorrelationId().equals(corrId)) {
47                     response.offer(new String(body, "UTF-8"));
48                 }

```

```

49         }
50     };
51     channel.basicConsume(replyQueueName, true, consumer);
52     // 获取回调的结果
53     String result = response.take();
54     System.out.println(" [RPCClient] Result:" + result);
55
56     //释放资源
57     channel.close();
58     connection.close();
59 }
60 }

```

编写服务端代码

服务端代码主要流程如下:

1. 接收消息
2. 根据消息内容进行响应处理, 把应答结果返回到回调队列中

声明队列

```

1 //2. 声明队列, 发送消息
2 channel.queueDeclare(Constants.RPC_REQUEST_QUEUE_NAME, true, false, false,
    null);

```

设置同时最多只能获取一个消息

如果不设置basicQos, RabbitMQ 会使用默认的QoS设置, 其prefetchCount默认值为0. 当prefetchCount为0时, RabbitMQ 会根据内部实现和当前的网络状况等因素, 可能会同时发送多条消息给消费者. 这意味着在默认情况下, 消费者可能会同时接收到多条消息, 但具体数量不是严格保证的, 可能会有所波动(后面会讲)

在RPC模式下, 通常期望的是一对一的消息处理, 即一个请求对应一个响应. 消费者在处理完一个消息并确认之后, 才会接收到下一条消息.

```

1 // 设置同时最多只能获取一个消息
2 channel.basicQos(1);
3 System.out.println("Awaiting RPC request");

```

接收消息, 并做出相应处理

```

1 Consumer consumer = new DefaultConsumer(channel){
2     @Override
3     public void handleDelivery(String consumerTag, Envelope envelope,
4         AMQP.BasicProperties properties, byte[] body) throws IOException {
5         AMQP.BasicProperties replyProps = new AMQP.BasicProperties
6             .Builder()
7             .correlationId(properties.getCorrelationId())
8             .build();
9         // 生成返回
10        String message = new String(body);
11        String response = "request:" + message + ", response: 处理成功";
12        // 回复消息, 通知已经收到请求
13        channel.basicPublish( "", properties.getReplyTo(), replyProps,
14            response.getBytes());
15        // 对消息进行应答
16        channel.basicAck(envelope.getDeliveryTag(), false);
17    }
18 };
19 channel.basicConsume(Constants.RPC_REQUEST_QUEUE_NAME, false, consumer);

```

RabbitMQ 消息确定机制

在RabbitMQ中, basicConsume方法的autoAck参数用于指定消费者是否应该自动向消息队列确认消息

自动确认 (autoAck=true) : 消息队列在将消息发送给消费者后, 会立即从内存中删除该消息. 这意味着, 如果消费者处理消息失败, 消息将丢失, 因为消息队列认为消息已经被成功消费

手动确认 (autoAck=false) : 消息队列在将消息发送给消费者后, 需要消费者显式地调用basicAck方法来确认消息. 手动确认提供了更高的可靠性, 确保消息不会被意外丢失, 适用于消息处理重要且需要确保每个消息都被正确处理的场景.

完整代码

```

1 import com.rabbitmq.client.*;
2 import constant.Constants;
3
4 import java.io.IOException;
5 import java.util.concurrent.TimeoutException;
6
7 public class RPCServer {
8     public static void main(String[] args) throws IOException,
9         TimeoutException {
10        //1. 创建Channel通道
11        ConnectionFactory factory = new ConnectionFactory();

```

```

11      factory.setHost(Constants.HOST); //ip 默认值localhost
12      factory.setPort(Constants.PORT); //默认值5672
13      factory.setVirtualHost(Constants.VIRTUAL_HOST); //虚拟机名称, 默认 /
14
15      factory.setUsername(Constants.USER_NAME); //用户名, 默认guest
16      factory.setPassword(Constants.PASSWORD); //密码, 默认guest
17      Connection connection = factory.newConnection();
18      Channel channel = connection.createChannel();
19      //2. 声明队列
20      channel.queueDeclare(Constants.RPC_REQUEST_QUEUE_NAME, true, false,
false, null);
21      //3. 接收消息, 并消费
22      // 设置同时最多只能获取一个消息
23      channel.basicQos(1);
24      System.out.println("Awaiting RPC request");
25      Consumer consumer = new DefaultConsumer(channel){
26          @Override
27          public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
28          AMQP.BasicProperties replyProps = new AMQP.BasicProperties
29              .Builder()
30              .correlationId(properties.getCorrelationId())
31              .build();
32          // 生成返回
33          String message = new String(body);
34          String response = "request:" + message + ", response: 处理成功";
35          // 回复消息, 通知已经收到请求
36          channel.basicPublish( "", properties.getReplyTo(), replyProps,
response.getBytes());
37          // 对消息进行应答
38          channel.basicAck(envelope.getDeliveryTag(), false);
39
40      }
41  };
42      channel.basicConsume(Constants.RPC_REQUEST_QUEUE_NAME, false,
consumer);
43  }

```

运行程序, 观察结果

运行客户端:

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	amq.gen-iiijHJbGabr7E2aK1KQV3Nw	classic	AD Excl	1	0%	idle	0	0	0					
bite	direct_queue1	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	direct_queue2	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	fanout_queue1	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	fanout_queue2	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	hello	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	rpc_request_queue	classic	D	0	0%	idle	1	0	1	0.20/s	0.00/s	0.00/s		
bite	topic_queue1	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	topic_queue2	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	work_queues	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		

amq.gen-iiijHJbGabr7E2aK1KQV3Nw 就是回调队列

运行服务端, 观察客户端日志

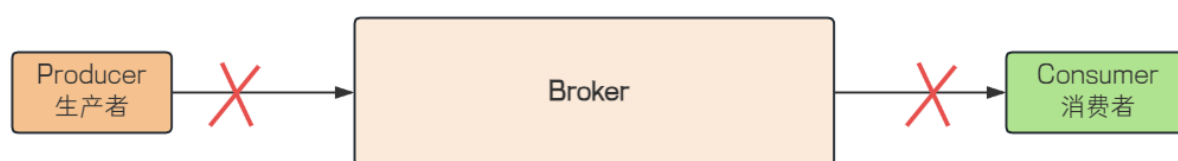
- 1 接收到回调消息:request:hello rpc..., response: 处理成功
- 2 [RPCClient] Result:request:hello rpc..., response: 处理成功

2.7 Publisher Confirms(发布确认)

作为消息中间件, 都会面临消息丢失的问题.

消息丢失大概分为三种情况:

1. 生产者问题. 因为应用程序故障, 网络抖动等各种原因, 生产者没有成功向broker发送消息.
2. 消息中间件自身问题. 生产者成功发送给了Broker, 但是Broker没有把消息保存好, 导致消息丢失.
3. 消费者问题. Broker 发送消息到消费者, 消费者在消费消息时, 因为没有处理好, 导致broker将消费失败的消息从队列中删除了.



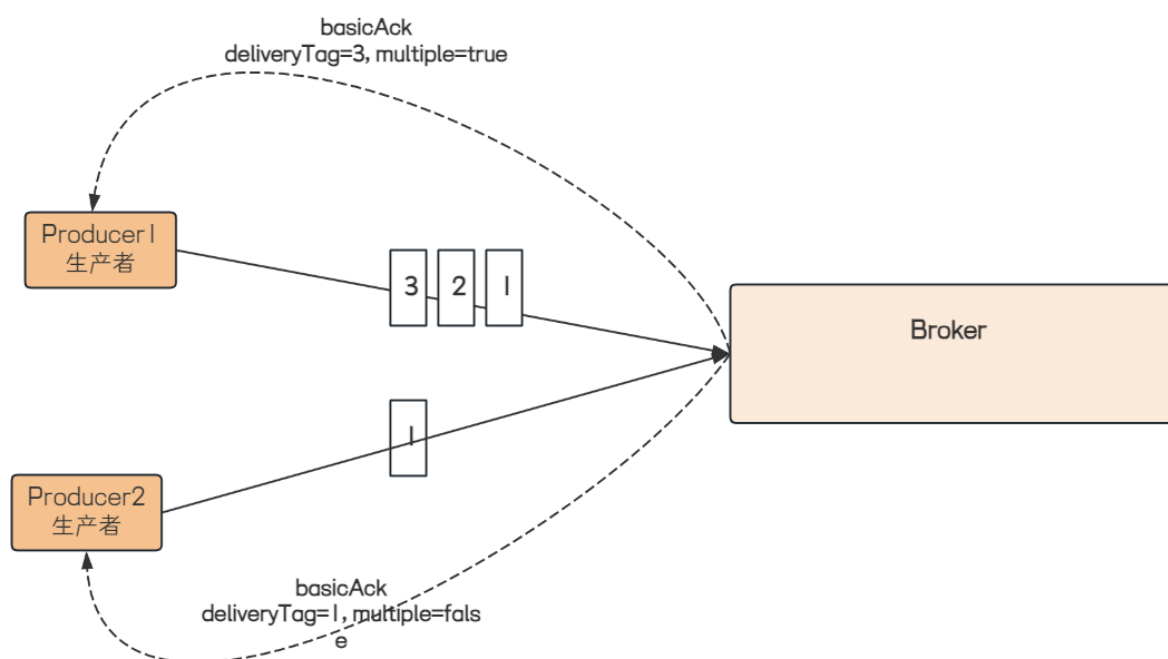
RabbitMQ也对上述问题给出了相应的解决方案. 问题2可以通过持久化机制. 问题3可以采用消息应答机制. (后面详细讲)

针对问题1, 可以采用发布确认(Publisher Confirms)机制实现.

发布确认 属于RabbitMQ的七大工作模式之一.

生产者将信道设置成confirm(确认)模式, 一旦信道进入confirm模式, 所有在该信道上发布消息都会被指派一个唯一的ID(从1开始), 一旦消息被投递到所有匹配的队列之后, RabbitMQ就会发送一个确认给生产者(包含消息的唯一ID), 这就使得生产者知道消息已经正确到达目的队列了, 如果消息和队列是可持久化的, 那么确认消息会在将消息写入磁盘之后发出. broker回传给生产者的确认消息中

deliveryTag 包含了确认消息的序号, 此外 broker 也可以设置channel.basicAck方法中的multiple参数, 表示到这个序号之前的所有消息都已经得到了处理.



发送方确认机制最大的好处在于它是异步的, 生产者可以同时发布消息和等待信道返回确认消息.

1. 当消息最终得到确认之后, 生产者可以通过回调方法来处理该确认消息.
2. 如果RabbitMQ因为自身内部错误导致消息丢失, 就会发送一条nack(Basic.Nack)命令, 生产者同样可以在回调方法中处理该nack命令.

使用发送确认机制, 必须要信道设置成confirm(确认)模式.

发布确认是 AMQP 0.9.1 协议的扩展, 默认情况下它不会被启用. 生产者通过channel.confirmSelect()将信道设置为confirm模式.

```
1 Channel channel = connection.createChannel();
2 channel.confirmSelect();
```

发布确认有3种策略, 接下来我们来学习这三种策略.

Publishing Messages Individually(单独确认)

代码示例:

```
1 static void publishMessagesIndividually() throws Exception {
2     try (Connection connection = createConnection()) {
3         //创建channel
```

```

4      Channel ch = connection.createChannel();
5      //开启信道确认模式
6      ch.confirmSelect();
7      //声明队列
8      ch.queueDeclare(PUBLISHER_CONFIRMS_QUEUE_NAME1, true, false, true,
null);
9
10     long start = System.currentTimeMillis();
11     //循环发送消息
12     for (int i = 0; i < MESSAGE_COUNT; i++) {
13         String body = "消息"+ i;
14         //发布消息
15         ch.basicPublish("", PUBLISHER_CONFIRMS_QUEUE_NAME1, null,
body.getBytes());
16         //等待确认消息.只要消息被确认,这个方法就会被返回
17         //如果超时过期,则抛出TimeoutException。如果任何消息被nack(丢失),
waitForConfirmsOrDie将抛出IOException。
18         ch.waitForConfirmsOrDie(5_000);
19     }
20     long end = System.currentTimeMillis();
21     System.out.format("Published %d messages individually in %d ms",
MESSAGE_COUNT, end - start);
22 }
23 }

```

观察运行结果:

```
1 Published 200 messages individually in 6265 ms
```

可以发现, 发送200条消息, 耗时很长.

观察上面代码, 会发现这种策略是每发送一条消息后就调用channel.waitForConfirmsOrDie方法, 之后等待服务端的确认, 这实际上是一种串行同步等待的方式. 尤其对于持久化的消息来说, 需要等待消息确认存储在磁盘之后才会返回(调用Linux内核的fsync方法).

但是发布确认机制是支持异步的. 可以一边发送消息, 一边等待消息确认.

由此进行了改进, 接下来看另外两种策略:

- Publishing Messages in Batches(批量确认): 每发送一批消息后, 调用channel.waitForConfirms方法, 等待服务器的确认返回.
- Handling Publisher Confirms Asynchronously(异步确认): 提供一个回调方法, 服务端确认了一条或者多条消息后客户端会回这个方法进行处理

Publishing Messages in Batches(批量确认)

代码示例:

```
1 static void publishMessagesInBatch() throws Exception {
2     try (Connection connection = createConnection()) {
3         //创建信道
4         Channel ch = connection.createChannel();
5         //信道设置为confirm模式
6         ch.confirmSelect();
7         //声明队列
8         ch.queueDeclare(PUBLISHER_CONFIRMS_QUEUE_NAME2, true, false, true,
9             null);
10
11         int batchSize = 100;
12         int outstandingMessageCount = 0;
13
14         long start = System.currentTimeMillis();
15         for (int i = 0; i < MESSAGE_COUNT; i++) {
16             String body = "消息" + i;
17             //发送消息
18             ch.basicPublish("", PUBLISHER_CONFIRMS_QUEUE_NAME2, null,
19                 body.getBytes());
20             outstandingMessageCount++;
21             //批量确认消息
22             if (outstandingMessageCount == batchSize) {
23                 ch.waitForConfirmsOrDie(5_000);
24                 outstandingMessageCount = 0;
25             }
26         }
27         //消息发送完, 还有未确认的消息, 进行确认
28         if (outstandingMessageCount > 0) {
29             ch.waitForConfirmsOrDie(5_000);
30         }
31         long end = System.currentTimeMillis();
32         System.out.format("Published %d messages in batch in %d ms",
33             MESSAGE_COUNT, end - start);
34     }
35 }
```

观察运行结果:

```
1 Published 200 messages in batch in 128 ms
```


可以观察到,性能提高了很多.

相比于单独确认策略,批量确认极大地提升了confirm的效率,缺点是出现Basic.Nack或者超时时,我们不清楚具体哪条消息出了问题.客户端需要将这一批次的消息全部重发,这会带来明显的重复消息数量.当消息经常丢失时,批量确认的性能应该是不升反降的.

Handling Publisher Confirms Asynchronously(异步确认)

异步confirm方法的编程实现最为复杂. Channel 接口提供了一个方法addConfirmListener. 这个方法可以添加ConfirmListener 回调接口.

ConfirmListener 接口中包含两个方法: `handleAck(long deliveryTag, boolean multiple)` 和 `handleNack(long deliveryTag, boolean multiple)`, 分别对应处理 RabbitMQ 发送给生产者的ack和nack.

`deliveryTag` 表示发送消息的序号. `multiple` 表示是否批量确认.

我们需要为每一个Channel 维护一个已发送消息的序号集合. 当收到RabbitMQ的confirm 回调时, 从集合中删除对应的消息. 当Channel开启confirm模式后, channel上发送消息都会附带一个从1开始递增的deliveryTag序号. 我们可以使用SortedSet 的有序性来维护这个已发消息的集合.

1. 当收到ack时, 从序列中删除该消息的序号. 如果为批量确认消息, 表示小于等于当前序号deliveryTag的消息都收到了, 则清除对应集合
2. 当收到nack时, 处理逻辑类似, 不过**需要结合具体的业务情况, 进行消息重发等操作.**

代码示例:

```
1 static void handlePublishConfirmsAsynchronously() throws Exception {
2     try (Connection connection = createConnection()) {
3         Channel ch = connection.createChannel();
4
5         ch.queueDeclare(PUBLISHER_CONFIRMS_QUEUE_NAME3, false, false, true,
6             null);
7
8         ch.confirmSelect();
9
10        //有序集合,元素按照自然顺序进行排序,存储未confirm消息序号
11        SortedSet<Long> confirmSet = Collections.synchronizedSortedSet(new
12            TreeSet<>());
13
14        ch.addConfirmListener(new ConfirmListener() {
15            @Override
16            public void handleAck(long deliveryTag, boolean multiple) throws
17                IOException {
18                //System.out.println("ack, SeqNo: " + deliveryTag +
19                    ",multiple:" + multiple);
```

```

16         //multiple 批量
17         //confirmSet.headSet(n)方法返回当前集合中小于n的集合
18         if (multiple) {
19             //批量确认:将集合中小于等于当前序号deliveryTag元素的集合清除, 表示
    这批序号的消息都已经被ack了
20             confirmSet.headSet(deliveryTag+1).clear();
21         } else {
22             //单条确认:将当前的deliveryTag从集合中移除
23             confirmSet.remove(deliveryTag);
24         }
25     }
26
27     @Override
28     public void handleNack(long deliveryTag, boolean multiple) throws
    IOException {
29         System.err.format("deliveryTag: %d, multiple: %b%n",
    deliveryTag, multiple);
30         if (multiple) {
31             //批量确认:将集合中小于等于当前序号deliveryTag元素的集合清除, 表示
    这批序号的消息都已经被ack了
32             confirmSet.headSet(deliveryTag+1).clear();
33         } else {
34             //单条确认:将当前的deliveryTag从集合中移除
35             confirmSet.remove(deliveryTag);
36         }
37         //如果处理失败, 这里需要添加处理消息重发的场景。此处代码省略
38
39     }
40 });
41
42 //循环发送消息
43 long start = System.currentTimeMillis();
44 for (int i = 0; i < MESSAGE_COUNT; i++) {
45     String message = "消息" + i;
46     //得到下次发送消息的序号, 从1开始
47     long nextPublishSeqNo = ch.getNextPublishSeqNo();
48     //System.out.println("消息序号:" + nextPublishSeqNo);
49     ch.basicPublish("", PUBLISHER_CONFIRMS_QUEUE_NAME3, null,
    message.getBytes());
50     //将序号存入集合中
51     confirmSet.add(nextPublishSeqNo);
52 }
53
54 //消息确认完毕
55 while (!confirmSet.isEmpty()){
56     Thread.sleep(10);
57 }

```

```

58
59         long end = System.currentTimeMillis();
60         System.out.format("Published %d messages and handled confirms
        asynchronously in %d ms%n", MESSAGE_COUNT, end - start);
61     }
62
63 }

```

观察运行结果:

```

1 Published 200 messages and handled confirms asynchronously in 92 ms

```

三种策略对比

完整代码:

```

1
2 import com.rabbitmq.client.Channel;
3 import com.rabbitmq.client.ConfirmListener;
4 import com.rabbitmq.client.Connection;
5 import com.rabbitmq.client.ConnectionFactory;
6 import constant.Constants;
7
8 import java.io.IOException;
9 import java.util.Collections;
10 import java.util.SortedSet;
11 import java.util.TreeSet;
12
13 public class PublisherConfirms {
14
15     private static final int MESSAGE_COUNT = 500;
16     private static final String PUBLISHER_CONFIRMS_QUEUE_NAME1 =
17     "publisher_confirms_queue1";
18     private static final String PUBLISHER_CONFIRMS_QUEUE_NAME2 =
19     "publisher_confirms_queue2";
20     private static final String PUBLISHER_CONFIRMS_QUEUE_NAME3 =
21     "publisher_confirms_queue3";
22
23     static Connection createConnection() throws Exception {
24         ConnectionFactory factory = new ConnectionFactory();
25         factory.setHost(Constants.HOST); //ip 默认值localhost
26         factory.setPort(Constants.PORT); //默认值5672
27         factory.setVirtualHost(Constants.VIRTUAL_HOST); //虚拟机名称, 默认 /

```

```

25
26     factory.setUsername(Constants.USER_NAME); //用户名,默认guest
27     factory.setPassword(Constants.PASSWORD); //密码,默认guest
28     return factory.newConnection();
29 }
30
31 public static void main(String[] args) throws Exception {
32     publishMessagesIndividually();
33     publishMessagesInBatch();
34     handlePublishConfirmsAsynchronously();
35 }
36
37 static void publishMessagesIndividually() throws Exception {
38     try (Connection connection = createConnection()) {
39         //创建channel
40         Channel ch = connection.createChannel();
41         //开启信道确认模式
42         ch.confirmSelect();
43         //声明队列
44         ch.queueDeclare(PUBLISHER_CONFIRMS_QUEUE_NAME1, true, false, true,
45 null);
46
47         long start = System.currentTimeMillis();
48         //循环发送消息
49         for (int i = 0; i < MESSAGE_COUNT; i++) {
50             String body = "消息" + i;
51             //发布消息
52             ch.basicPublish("", PUBLISHER_CONFIRMS_QUEUE_NAME1, null,
53 body.getBytes());
54             //等待确认消息.只要消息被确认,这个方法就会被返回
55             //如果超时过期,则抛出TimeoutException. 如果任何消息被nack(丢失),
56             waitForConfirmsOrDie将抛出IOException。
57             ch.waitForConfirmsOrDie(5_000);
58         }
59         long end = System.currentTimeMillis();
60         System.out.format("Published %d messages individually in %d ms%n",
61 MESSAGE_COUNT, end - start);
62     }
63 }
64
65 static void publishMessagesInBatch() throws Exception {
66     try (Connection connection = createConnection()) {
67         //创建信道
68         Channel ch = connection.createChannel();
69         //信道设置为confirm模式
70         ch.confirmSelect();
71         //声明队列

```

```

68         ch.queueDeclare(PUBLISHER_CONFIRMS_QUEUE_NAME2, true, false, true,
        null);
69
70         int batchSize = 100;
71         int outstandingMessageCount = 0;
72
73         long start = System.currentTimeMillis();
74         for (int i = 0; i < MESSAGE_COUNT; i++) {
75             String body = "消息" + i;
76             //发送消息
77             ch.basicPublish("", PUBLISHER_CONFIRMS_QUEUE_NAME2, null,
        body.getBytes());
78             outstandingMessageCount++;
79             //批量确认消息
80             if (outstandingMessageCount == batchSize) {
81                 ch.waitForConfirmsOrDie(5_000);
82                 outstandingMessageCount = 0;
83             }
84         }
85         //消息发送完, 还有未确认的消息, 进行确认
86         if (outstandingMessageCount > 0) {
87             ch.waitForConfirmsOrDie(5_000);
88         }
89         long end = System.currentTimeMillis();
90         System.out.format("Published %d messages in batch in %d ms%n",
        MESSAGE_COUNT, end - start);
91     }
92 }
93
94 static void handlePublishConfirmsAsynchronously() throws Exception {
95     try (Connection connection = createConnection()) {
96         Channel ch = connection.createChannel();
97
98         ch.queueDeclare(PUBLISHER_CONFIRMS_QUEUE_NAME3, false, false,
        true, null);
99
100        ch.confirmSelect();
101
102        //有序集合, 元素按照自然顺序进行排序, 存储未confirm消息序号
103        SortedSet<Long> confirmSet = Collections.synchronizedSortedSet(new
        TreeSet<>());
104
105        ch.addConfirmListener(new ConfirmListener() {
106            @Override
107            public void handleAck(long deliveryTag, boolean multiple)
        throws IOException {

```

```

108 //          System.out.println("ack, SeqNo: " + deliveryTag +
    ",multiple:" + multiple);
109          //multiple 批量
110          //confirmSet.headSet(n)方法返回当前集合中小于n的集合
111          if (multiple) {
112              //批量确认:将集合中小于等于当前序号deliveryTag元素的集合清
    除, 表示这批序号的消息都已经被ack了
113              confirmSet.headSet(deliveryTag+1).clear();
114          } else {
115              //单条确认:将当前的deliveryTag从集合中移除
116              confirmSet.remove(deliveryTag);
117          }
118      }
119
120      @Override
121      public void handleAck(long deliveryTag, boolean multiple)
    throws IOException {
122          System.err.format("deliveryTag: %d, multiple: %b\n",
    deliveryTag, multiple);
123          if (multiple) {
124              //批量确认:将集合中小于等于当前序号deliveryTag元素的集合清
    除, 表示这批序号的消息都已经被ack了
125              confirmSet.headSet(deliveryTag+1).clear();
126          } else {
127              //单条确认:将当前的deliveryTag从集合中移除
128              confirmSet.remove(deliveryTag);
129          }
130          //如果处理失败, 这里需要添加处理消息重发的场景。此处代码省略
131
132      }
133  });
134
135  //循环发送消息
136  long start = System.currentTimeMillis();
137  for (int i = 0; i < MESSAGE_COUNT; i++) {
138      String message = "消息" + i;
139      //得到下次发送消息的序号, 从1开始
140      long nextPublishSeqNo = ch.getNextPublishSeqNo();
141      //          System.out.println("消息序号:" + nextPublishSeqNo);
142      ch.basicPublish("", PUBLISHER_CONFIRMS_QUEUE_NAME3, null,
    message.getBytes());
143      //将序号存入集合中
144      confirmSet.add(nextPublishSeqNo);
145  }
146
147  //消息确认完毕
148  while (!confirmSet.isEmpty()){

```

```
149         Thread.sleep(10);
150     }
151
152     long end = System.currentTimeMillis();
153     System.out.format("Published %d messages and handled confirms
    asynchronously in %d ms%n", MESSAGE_COUNT, end - start);
154 }
155
156 }
157
158 }
```

消息数越多, 异步确认的优势越明显

200条消息的结果对比:

- 1 Published 200 messages individually in 6931 ms
- 2 Published 200 messages in batch in 137 ms
- 3 Published 200 messages and handled confirms asynchronously in 73 ms

500条消息结果对比

- 1 Published 500 messages individually in 15805 ms
- 2 Published 500 messages in batch in 246 ms
- 3 Published 500 messages and handled confirms asynchronously in 107 ms

3. Spring Boot整合RabbitMQ

对于RabbitMQ开发, Spring 也提供了一些便利. Spring 和RabbitMQ的官方文档对此均有介绍

Spring官方: [Spring AMQP](#)

RabbitMQ 官方: [RabbitMQ tutorial - "Hello World!"](#) | [RabbitMQ](#)

下面来看如何基于SpringBoot 进行RabbitMQ的开发.

只演示部分常用的工作模式

3.1 工作队列模式

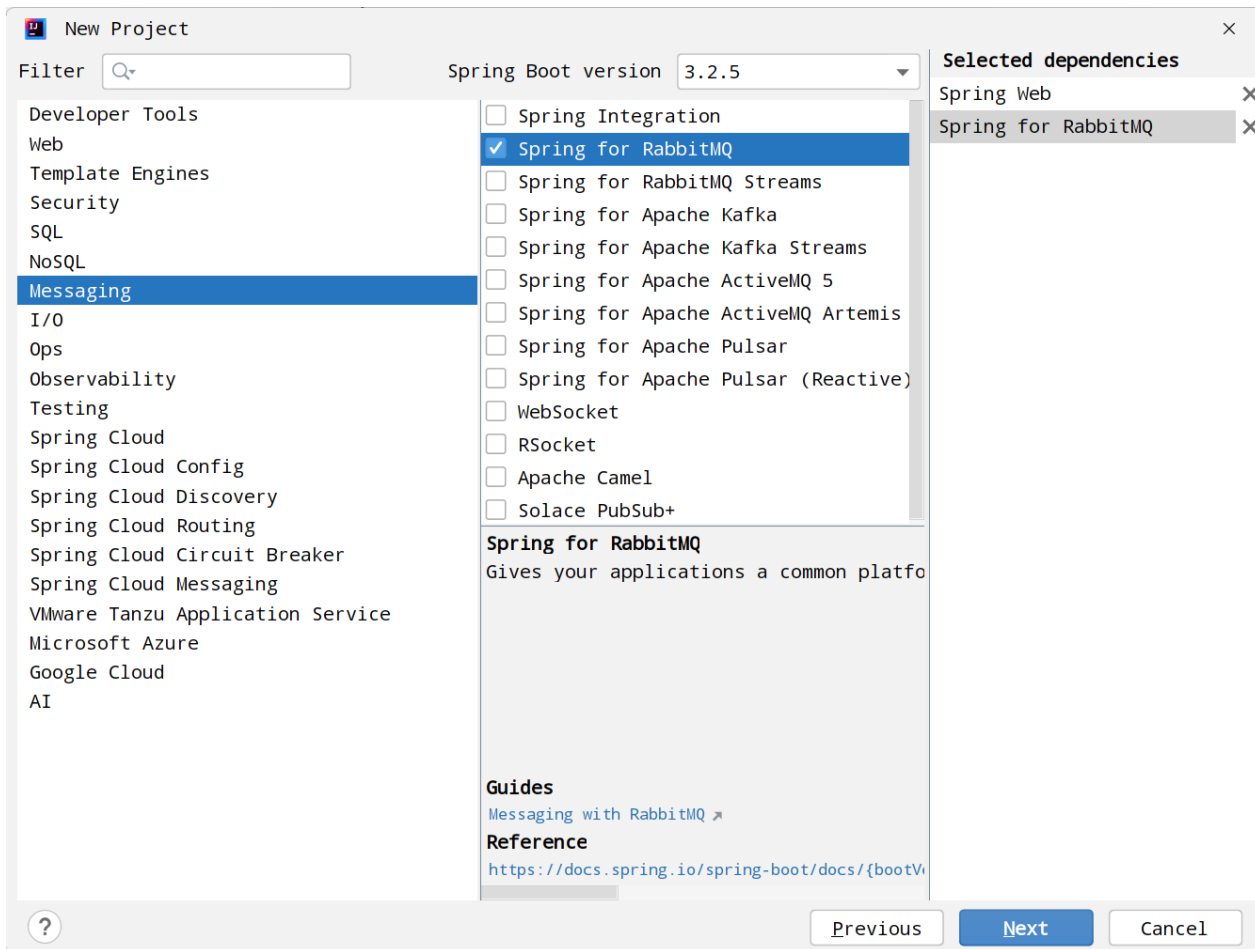
步骤:

1. 引入依赖
2. 编写yml配置，基本信息配置
3. 编写生产者代码
4. 编写消费者代码
 - a. 定义监听类, 使用@RabbitListener注解完成队列监听
5. 运行观察结果

引入依赖

```
1
2 <!--Spring MVC相关依赖-->
3 <dependency>
4     <groupId>org.springframework.boot</groupId>
5     <artifactId>spring-boot-starter-web</artifactId>
6 </dependency>
7 <dependency>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-test</artifactId>
10    <scope>test</scope>
11 </dependency>
12 <!--RabbitMQ相关依赖-->
13 <dependency>
14     <groupId>org.springframework.boot</groupId>
15     <artifactId>spring-boot-starter-amqp</artifactId>
16 </dependency>
```

也可以通过创建项目时, 加入依赖



添加配置

```
1 #配置RabbitMQ的基本信息
2 spring:
3   rabbitmq:
4     host: 110.41.51.65
5     port: 15673 #默认为5672
6     username: study
7     password: study
8     virtual-host: bite #默认值为 /
```

或以下配置

```
1 #amqp://username:password@Ip:port/virtual-host
2 spring:
3   rabbitmq:
4     addresses: amqp://study:study@110.41.51.65:15673/bite
```

编写生产者代码

为方便测试, 我们通过接口来发送消息

```
1 //work模式队列名称
2 public static final String WORK_QUEUE = "work_queue";
```

声明队列

```
1 import com.bite.rabbitmq.constant.Constants;
2 import org.springframework.amqp.core.*;
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5
6
7 @Configuration
8 public class RabbitMQConfig {
9     //1. 工作模式队列
10    @Bean("workQueue")
11    public Queue workQueue() {
12        return QueueBuilder.durable(Constants.WORK_QUEUE).build();
13    }
14 }
```

```
1 @RequestMapping("/producer")
2 @RestController
3 public class ProducerController {
4     @Autowired
5     private RabbitTemplate rabbitTemplate;
6
7     @RequestMapping("/work")
8     public String work(){
9         for (int i = 0; i < 10; i++) {
10             //使用内置交换机发送消息，routingKey和队列名称保持一致
11             rabbitTemplate.convertAndSend("", Constants.WORK_QUEUE, "hello
spring amqp: work...");
12         }
13         return "发送成功";
14     }
15 }
```

编写消费者代码

定义监听类

```

1 import com.bite.rabbitmq.constant.Constants;
2 import org.springframework.amqp.core.Message;
3 import org.springframework.amqp.rabbit.annotation.RabbitListener;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class WorkListener {
8     @RabbitListener(queues = Constants.WORK_QUEUE)
9     public void listenerQueue(Message message){
10         System.out.println("listener 1["+Constants.WORK_QUEUE+"]收到消息:" +
11             message);
12     }
13     @RabbitListener(queues = Constants.WORK_QUEUE)
14     public void listenerQueue2(Message message){
15         System.out.println("listener 2["+Constants.WORK_QUEUE+"]收到消息:" +
16             message);
17     }
18 }

```

`@RabbitListener` 是Spring框架中用于监听RabbitMQ队列的注解, 通过使用这个注解, 可以定义一个方法, 以便从RabbitMQ队列中接收消息. 该注解支持多种参数类型, 这些参数类型代表了从RabbitMQ接收到的消息和相关信息.

以下是一些常用的参数类型:

1. `String`: 返回消息的内容
2. `Message` (`org.springframework.amqp.core.Message`): Spring AMQP的 `Message` 类, 返回原始的消息体以及消息的属性, 如消息ID, 内容, 队列信息等.
3. `Channel` (`com.rabbitmq.client.Channel`): RabbitMQ的通道对象, 可以用于进行更高级的操作, 如手动确认消息.

运行程序, 观察结果

生产者测试(测试时需要把监听先注掉, 不然会立马被消费掉)

Overview							Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer capacity	State	Ready	Unacked	Total	incoming	deliver / get	ack	
bite	work_queue	classic	D	0	0%	idle	10	0	10	0.00/s			

点进去可以看到消息的内容

▼ Get messages

Warning: getting messages from a queue is a destructive action. ?

Ack Mode:

Encoding: ?

Messages:

Get Message(s)

Message 1

The server reported 9 messages remaining.

Exchange	(AMQP default)
Routing Key	work_queue
Redelivered	o
Properties	priority: 0 delivery_mode: 2 headers: content_encoding: UTF-8 content_type: text/plain
Payload 26 bytes Encoding: string	hello spring amqp: work...

消费者测试, 打印消息内容

```
1 listener 1[work_queue]接收到消息:(Body:'hello spring amqp: work...'
  MessageProperties [headers={}, contentType=text/plain, contentEncoding=UTF-8,
  contentLength=0, receivedDeliveryMode=PERSISTENT, priority=0,
  redelivered=true, receivedExchange=, receivedRoutingKey=work_queue,
  deliveryTag=1, consumerTag=amq.ctag-pNYV0TLX7zeFmD7sL9NXw,
  consumerQueue=work_queue])
2 listener 2[work_queue]接收到消息:(Body:'hello spring amqp: work...'
  MessageProperties [headers={}, contentType=text/plain, contentEncoding=UTF-8,
  contentLength=0, receivedDeliveryMode=PERSISTENT, priority=0,
  redelivered=true, receivedExchange=, receivedRoutingKey=work_queue,
  deliveryTag=1, consumerTag=amq.ctag-ghlFZQoXLpDBD_BXedIBdQ,
  consumerQueue=work_queue])
3 ...
```

3.2 Publish/Subscribe(发布订阅模式)

在发布/订阅模型中, 多了一个Exchange角色.

Exchange 常见有三种类型, 分别代表不同的路由规则

- a) Fanout:广播, 将消息交给所有绑定到交换机的队列(Publish/Subscribe模式)
- b) Direct:定向, 把消息交给符合指定routing key的队列(Routing模式)
- c) Topic:通配符, 把消息交给符合routing pattern(路由模式)的队列(Topics模式)

步骤:

1. 引入依赖
2. 编写生产者代码
3. 编写消费者代码

引入依赖

```
1 <!--Spring MVC相关依赖-->
2 <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-web</artifactId>
5 </dependency>
6 <dependency>
7     <groupId>org.springframework.boot</groupId>
8     <artifactId>spring-boot-starter-test</artifactId>
9     <scope>test</scope>
10 </dependency>
11 <!--RabbitMQ相关依赖-->
12 <dependency>
13     <groupId>org.springframework.boot</groupId>
14     <artifactId>spring-boot-starter-amqp</artifactId>
15 </dependency>
```

编写生产者代码

和简单模式的区别是: 需要创建交换机, 并且绑定队列和交换机

声明队列, 交换机, 绑定队列和交换机

```
1 //发布/订阅模式
2 public static final String FANOUT_QUEUE1 = "fanout_queue1";
3 public static final String FANOUT_QUEUE2 = "fanout_queue2";
4 public static final String FANOUT_EXCHANGE_NAME = "fanout_exchange";
```

```
1 //2. 发布订阅模式
```

```

2 //声明2个队列, 观察是否两个队列都收到了消息
3 @Bean("fanoutQueue1")
4 public Queue fanoutQueue1() {
5     return QueueBuilder.durable(Constants.FANOUT_QUEUE1).build();
6 }
7 @Bean("fanoutQueue2")
8 public Queue fanoutQueue2() {
9     return QueueBuilder.durable(Constants.FANOUT_QUEUE2).build();
10 }
11 //声明交换机
12 @Bean("fanoutExchange")
13 public FanoutExchange fanoutExchange() {
14     return
15         ExchangeBuilder.fanoutExchange(Constants.FANOUT_EXCHANGE_NAME).durable(true).build();
16 }
17 //队列和交换机绑定
18 @Bean
19 public Binding fanoutBinding(@Qualifier("fanoutExchange") FanoutExchange
20     exchange, @Qualifier("fanoutQueue1") Queue queue) {
21     return BindingBuilder.bind(queue).to(exchange);
22 }
23 @Bean
24 public Binding fanoutBinding2(@Qualifier("fanoutExchange") FanoutExchange
25     exchange, @Qualifier("fanoutQueue2") Queue queue) {
26     return BindingBuilder.bind(queue).to(exchange);
27 }

```

使用接口发送消息

```

1 @RequestMapping("/fanout")
2 public String fanoutProduct(){
3     //routingKey为空, 表示所有队列都可以收到消息
4     rabbitTemplate.convertAndSend(Constants.FANOUT_EXCHANGE_NAME, "", "hello
5     spring boot: fanout");
6     return "发送成功";
7 }

```

编写消费者代码

交换机和队列的绑定关系及声明已经在生产方写完, 所以消费者不需要再写了
定义监听类, 处理接收到的消息即可.

```

1 import com.bite.rabbitmq.constant.Constants;
2 import org.springframework.amqp.core.Message;
3 import org.springframework.amqp.rabbit.annotation.RabbitListener;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class FanoutListener {
8     //指定监听队列的名称
9     @RabbitListener(queues = Constants.FANOUT_QUEUE1)
10    public void ListenerQueue(String message){
11        System.out.println("[ "+Constants.FANOUT_QUEUE1+ " ]接收到消息:"+ message);
12    }
13
14    @RabbitListener(queues = Constants.FANOUT_QUEUE2)
15    public void ListenerQueue2(String message){
16        System.out.println("[ "+Constants.FANOUT_QUEUE2+ " ]接收到消息:"+ message);
17    }
18
19 }

```

运行程序, 观察结果

1. 运行项目, 调用接口发送消息

<http://127.0.0.1:8080/producer/fanout>

← → ↺ 🏠 🔍 <http://127.0.0.1:8080/producer/fanout>

发送成功

也可以把监听类注释掉, 观察两个队列的信息

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	fanout_queue1	classic	D	0	0%	idle	1	0	1	0.00/s	0.00/s	0.00/s		
bite	fanout_queue2	classic	D	0	0%	idle	1	0	1	0.00/s	0.00/s	0.00/s		
bite	work_queue	classic	D	1	0%	idle	0	0	0					

2. 监听类收到消息, 并打印

```

1 [fanout_queue1]接收到消息:hello spring boot: fanout
2 [fanout_queue2]接收到消息:hello spring boot: fanout

```

3.3 Routing(路由模式)

交换机类型为**Direct**时, 会把消息交给符合指定routing key的队列.

队列和交换机的绑定, 不是任意的绑定了, 而是要指定一个RoutingKey(路由key)

消息的发送方在向Exchange发送消息时, 也需要指定消息的RoutingKey

Exchange也不再把消息交给每一个绑定的key, 而是根据消息的RoutingKey进行判断, 只有队列的RoutingKey 和消息的RoutingKey 完全一致, 才会接收到消息

步骤:

1. 引入依赖(同上)
2. 编写生产者代码
3. 编写消费者代码

编写生产者代码

和发布订阅模式的区别就是: 交换机类型不同, 绑定队列的RoutingKey不同

声明队列, 交换机, 绑定队列和交换机

```
1 //routing模式
2 public static final String DIRECT_QUEUE1 = "direct_queue1";
3 public static final String DIRECT_QUEUE2 = "direct_queue2";
4 public static final String DIRECT_EXCHANGE_NAME = "direct_exchange";
```

```
1 //Routing模式
2 @Bean("directQueue1")
3 public Queue routingQueue1() {
4     return QueueBuilder.durable(Constants.DIRECT_QUEUE1).build();
5 }
6 @Bean("directQueue2")
7 public Queue routingQueue2() {
8     return QueueBuilder.durable(Constants.DIRECT_QUEUE2).build();
9 }
10 //声明交换机
11 @Bean("directExchange")
12 public DirectExchange directExchange() {
13     return
14         ExchangeBuilder.directExchange(Constants.DIRECT_EXCHANGE_NAME).durable(true).build();
15 }
```



```

15 //队列和交换机绑定
16 //队列1绑定orange
17 @Bean
18 public Binding directBinding(@Qualifier("directExchange") DirectExchange
    exchange, @Qualifier("directQueue1") Queue queue) {
19     return BindingBuilder.bind(queue).to(exchange).with("orange");
20 }
21 //队列2绑定black, green
22 @Bean
23 public Binding directBinding2(@Qualifier("directExchange") DirectExchange
    exchange, @Qualifier("directQueue2") Queue queue) {
24     return BindingBuilder.bind(queue).to(exchange).with("black");
25 }
26 @Bean
27 public Binding directBinding3(@Qualifier("directExchange") DirectExchange
    exchange, @Qualifier("directQueue2") Queue queue) {
28     return BindingBuilder.bind(queue).to(exchange).with("green");
29 }

```

使用接口发送消息

```

1 @RequestMapping("/direct")
2 public String directProduct(String routingKey){
3     //routingKey作为参数传递
4     rabbitTemplate.convertAndSend(Constants.DIRECT_EXCHANGE_NAME,
    routingKey,"hello spring boot: direct "+routingKey);
5     return "发送成功";
6 }

```

编写消费者代码

交换机和队列的绑定关系及声明已经在生产方写完, 所以消费者不需要再写了

定义监听类, 处理接收到的消息即可.

```

1 import com.bite.rabbitmq.constant.Constants;
2 import org.springframework.amqp.rabbit.annotation.RabbitListener;
3 import org.springframework.stereotype.Component;
4
5 @Component
6 public class DirectListener {
7     //指定监听队列的名称
8     @RabbitListener(queues = Constants.DIRECT_QUEUE1)
9     public void ListenerQueue(String message){

```

```
10         System.out.println "["+Constants.DIRECT_QUEUE1+ "]接收到消息:"+ message);
11     }
12
13     @RabbitListener(queues = Constants.DIRECT_QUEUE2)
14     public void ListenerQueue2(String message){
15         System.out.println "["+Constants.DIRECT_QUEUE2+ "]接收到消息:"+ message);
16     }
17
18 }
```

运行程序, 观察结果

1. 运行项目
2. 调用接口发送routingkey为orange的消息

<http://127.0.0.1:8080/producer/direct?routingKey=orange>

观察后端日志, 队列1收到消息

```
1 [direct_queue1]接收到消息:hello spring boot: direct orange
```

2. 调用接口发送routingkey为black的消息

<http://127.0.0.1:8080/producer/direct?routingKey=black>

观察后端日志, 队列2收到消息

```
1 [direct_queue2]接收到消息:hello spring boot: direct black
```

3. 调用接口发送routingkey为green的消息

<http://127.0.0.1:8080/producer/direct?routingKey=green>

观察后端日志, 队列2收到消息

```
1 [direct_queue2]接收到消息:hello spring boot: direct green
```

3.4 Topics(通配符模式)

Topics 和Routing模式的区别是:

1. topics 模式使用的交换机类型为topic(Routing模式使用的交换机类型为direct)

2. topic 类型的交换机在匹配规则上进行了扩展, Binding Key支持通配符匹配

步骤:

1. 引入依赖(同上)
2. 编写生产者代码
3. 编写消费者代码

编写生产者代码

和发布订阅模式的区别就是: 交换机类型不同, 绑定队列的RoutingKey不同

声明队列, 交换机, 绑定队列和交换机

```
1 //topics模式
2 public static final String TOPICS_QUEUE1 = "topics_queue1";
3 public static final String TOPICS_QUEUE2 = "topics_queue2";
4 public static final String TOPICS_EXCHANGE_NAME = "topics_exchange";
```

```
1 //topic模式
2 @Bean("topicsQueue1")
3 public Queue topicsQueue1() {
4     return QueueBuilder.durable(Constants.TOPICS_QUEUE1).build();
5 }
6 @Bean("topicsQueue2")
7 public Queue topicsQueue2() {
8     return QueueBuilder.durable(Constants.TOPICS_QUEUE2).build();
9 }
10 //声明交换机
11 @Bean("topicExchange")
12 public TopicExchange topicExchange() {
13     return
14         ExchangeBuilder.topicExchange(Constants.TOPICS_EXCHANGE_NAME).durable(true).build();
15 }
16 //队列和交换机绑定
17 //队列1绑定error, 仅接收error信息
18 @Bean
19 public Binding topicBinding(@Qualifier("topicExchange") TopicExchange
20     exchange, @Qualifier("topicsQueue1") Queue queue) {
21     return BindingBuilder.bind(queue).to(exchange).with("*.error");
22 }
23 //队列2绑定info, error: error, info信息都接收
```

```

22 @Bean
23 public Binding topicBinding2(@Qualifier("topicExchange") TopicExchange
    exchange, @Qualifier("topicsQueue2") Queue queue) {
24     return BindingBuilder.bind(queue).to(exchange).with("#.info");
25 }
26 @Bean
27 public Binding topicBinding3(@Qualifier("topicExchange") TopicExchange
    exchange, @Qualifier("topicsQueue2") Queue queue) {
28     return BindingBuilder.bind(queue).to(exchange).with("*.error");
29 }

```

使用接口发送消息

```

1 @RequestMapping("/topics")
2 public String topicProduct(String routingKey){
3     //routingKey为空, 表示所有队列都可以收到消息
4     rabbitTemplate.convertAndSend(Constants.TOPICS_EXCHANGE_NAME,
    routingKey,"hello spring boot: topics "+routingKey);
5     return "发送成功";
6 }

```

编写消费者代码

定义监听类, 处理接收到的消息.

```

1 import com.bite.rabbitmq.constant.Constants;
2 import org.springframework.amqp.rabbit.annotation.RabbitListener;
3 import org.springframework.stereotype.Component;
4
5 @Component
6 public class TopicListener {
7     //指定监听队列的名称
8     @RabbitListener(queues = Constants.TOPICS_QUEUE1)
9     public void ListenerQueue(String message){
10         System.out.println("[ "+Constants.TOPICS_QUEUE1+ " ]接收到消息:"+ message);
11     }
12
13     @RabbitListener(queues = Constants.TOPICS_QUEUE2)
14     public void ListenerQueue2(String message){
15         System.out.println("[ "+Constants.TOPICS_QUEUE2+ " ]接收到消息:"+ message);
16     }
17
18 }

```

运行程序, 观察结果

1. 运行项目
2. 调用接口发送routingkey为order.error的消息

<http://127.0.0.1:8080/producer/topics?routingKey=order.error>

观察后端日志, 队列1和队列2均收到消息

```
1 [topics_queue2]接收到消息:hello spring boot: topics order.error
2 [topics_queue1]接收到消息:hello spring boot: topics order.error
```

2. 调用接口发送routingkey为order.pay.info的消息

<http://127.0.0.1:8080/producer/topics?routingKey=order.pay.info>

观察后端日志, 队列2收到消息

```
1 [topics_queue2]接收到消息:hello spring boot: topics order.pay.info
```

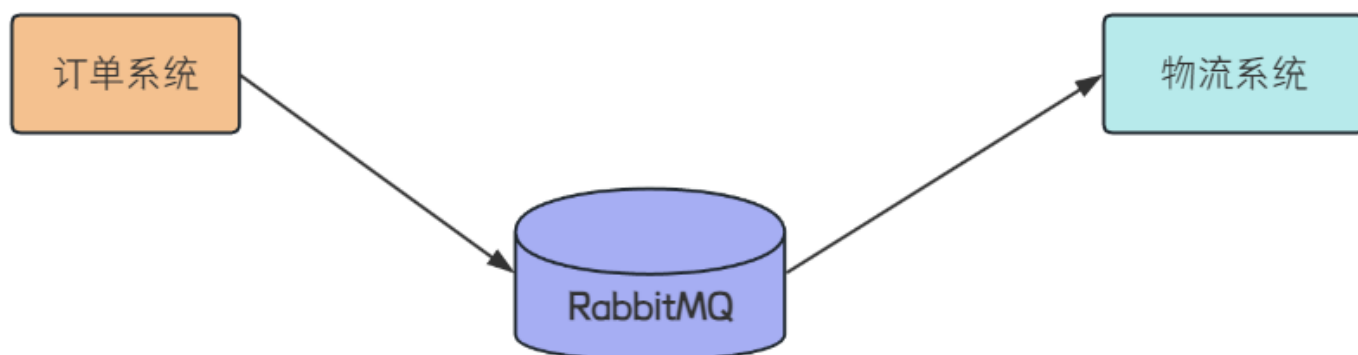
4. 基于SpringBoot+RabbitMQ完成应用通信

作为一个消息队列, RabbitMQ也可以用作应用程序之间的通信. 上述代码生产者和消费者代码放在不同的应用中即可完成不同应用程序的通信.

接下来我们来看, 基于SpringBoot+RabbitMQ完成应用间的通信.

需求描述:

用户下单成功之后, 通知物流系统, 进行发货. (只讲应用通信, 不做具体功能实现)

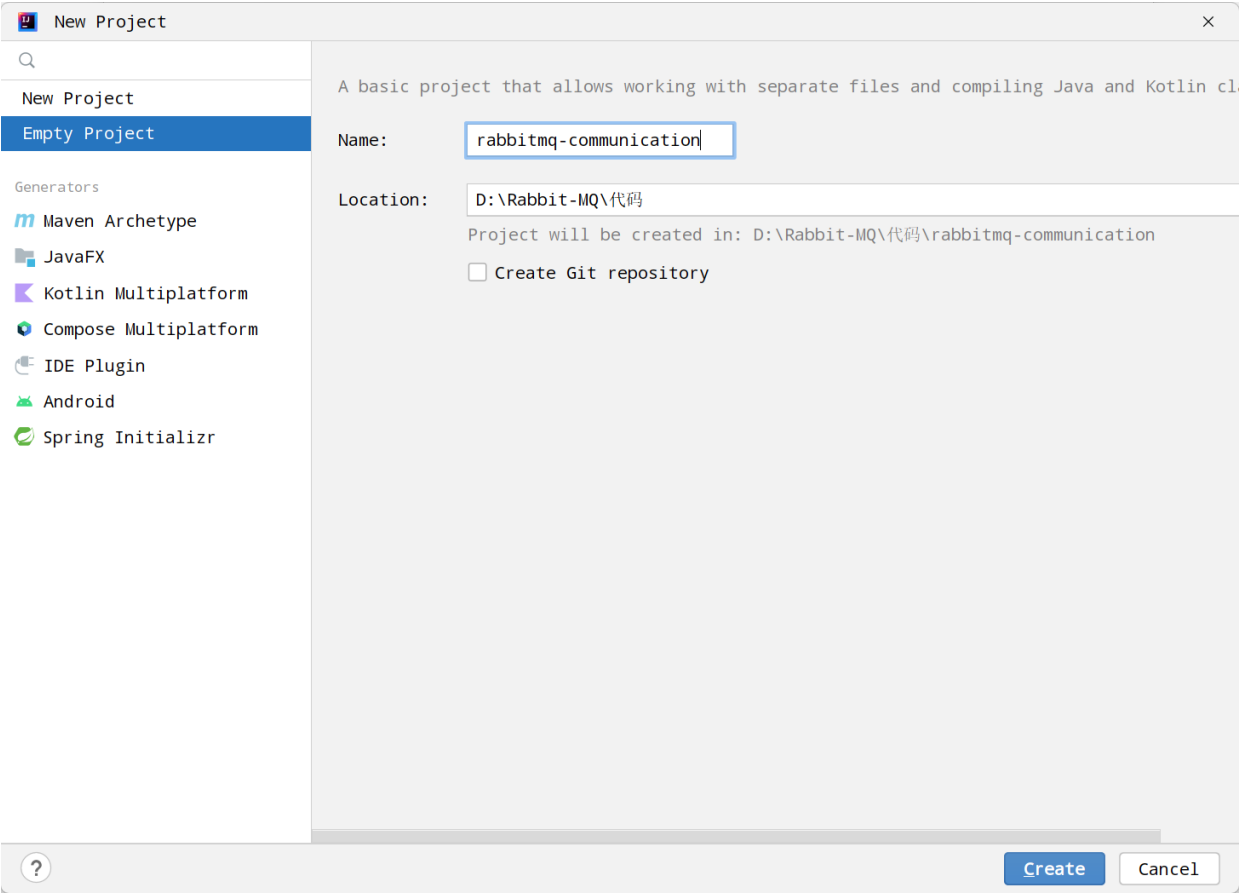


订单系统作为一个生产者, 物流系统作为一个消费者

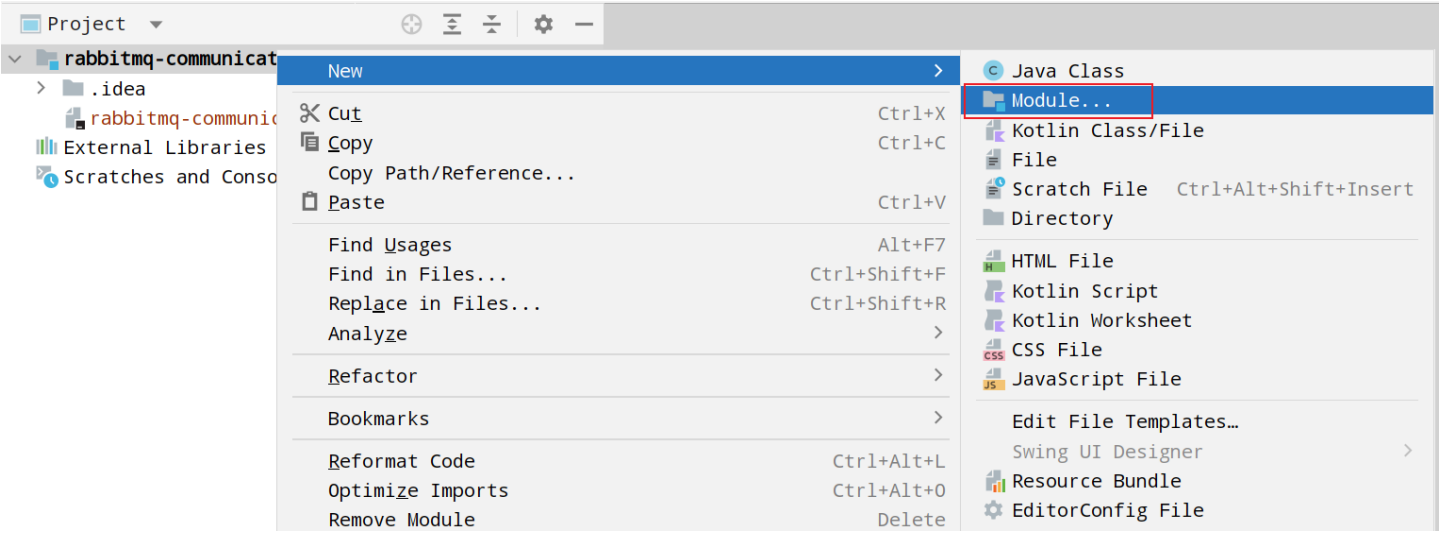
4.1 创建项目

为方便讲解, 把两个项目放在一个项目中(也可独立创建)

1. 创建一个空的项目 rabbitmq-communication(其实就是一个空的文件夹)



2. 在这个项目里, 创建Module



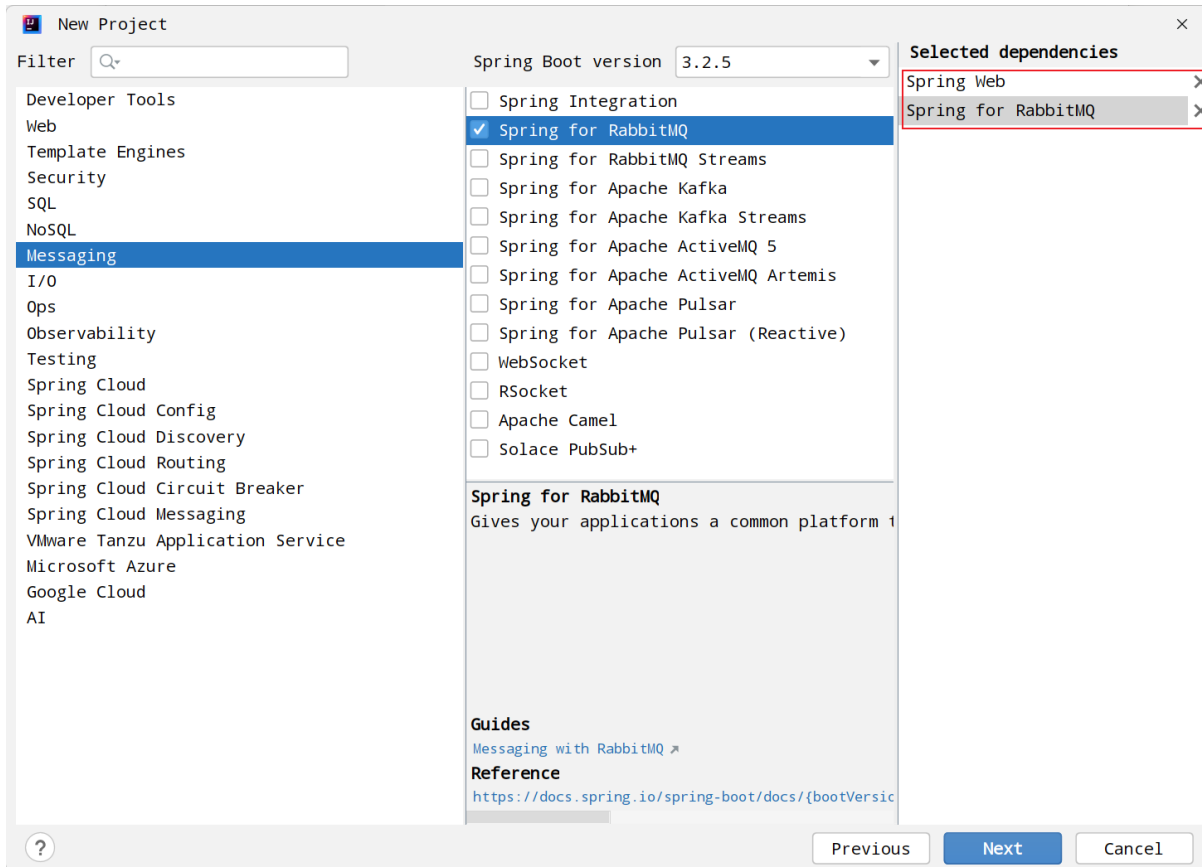
3. 后续流程和创建SpringBoot项目一样

添加对应依赖

创建两个项目

1) logistics-service

2) order-service



4. 最终结构如下



4.2 订单系统(生产者)

1. 完善配置信息

```
1 server.port=8080
2 #amqp://username:password@Ip:port/virtual-host
3 spring.rabbitmq.addresses=amqp://study:study@110.41.51.65:15673/bite
```

2. 声明队列

```
1 import org.springframework.amqp.core.Queue;
2 import org.springframework.amqp.core.QueueBuilder;
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5
6
7 @Configuration
8 public class RabbitConfig {
9     //1. 工作模式队列
10    @Bean("workQueue")
11    public Queue workQueue() {
12        return QueueBuilder.durable("order.create").build();
13    }
14 }
```

3. 编写下单接口, 下单成功之后, 发送订单消息

```
1 import org.springframework.amqp.rabbit.core.RabbitTemplate;
2 import org.springframework.beans.factory.annotation.Autowired;
3 import org.springframework.web.bind.annotation.RequestMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 import java.util.UUID;
7
8 @RequestMapping("/order")
9 @RestController
10 public class OrderController {
11     @Autowired
12     private RabbitTemplate rabbitTemplate;
13
14     @RequestMapping("/createOrder")
15     public String createOrder(){
16         //下单相关操作, 比如参数校验, 操作数据库等, 代码省略
17         //发送消息通知
18         String orderId = UUID.randomUUID().toString();
19         rabbitTemplate.convertAndSend("", "order.create", "下单成功, 订单
ID:"+orderId);
20         return "下单成功";
21     }
22 }
```


4. 启动服务, 观察结果

1) 访问接口, 模拟下单请求: <http://127.0.0.1:8080/order/createOrder>

可以观察到消息发送成功

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	Incoming	deliver / get	ack		
bite	direct_queue1	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	direct_queue2	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	fanout_queue1	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	fanout_queue2	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	order.create	classic	D	0	0%	idle	1	0	1	0.00/s	0.00/s	0.00/s		
bite	topics_queue1	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	topics_queue2	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	work_queue	classic	D	0	0%	idle	0	0	0					

查看消息

Get Message(s)

Message 1

The server reported 0 messages remaining.

Exchange	(AMQP default)
Routing Key	order.create
Redelivered	0
Properties	priority: 0 delivery_mode: 2 headers: content_encoding: UTF-8 content_type: text/plain
Payload 59 bytes Encoding: string	下单成功, 订单ID: 77d822e4-eb75-4721-8757-eb8b383e67b0

4.3 物流系统(消费者)

1. 完善配置信息

8080端口已经被订单系统占用了, 修改物流系统的端口号为9090

```
1 server.port=9090
2 #amqp://username:password@Ip:port/virtual-host
3 spring.rabbitmq.addresses=amqp://study:study@110.41.51.65:15673/bite
```

2. 监听队列

```
1 import org.springframework.amqp.rabbit.annotation.RabbitListener;
```

```

2 import org.springframework.stereotype.Component;
3
4 @Component
5 public class OrderCreateListener {
6
7     //指定监听队列的名称
8     @RabbitListener(queues = "order.create")
9     public void ListenerQueue(String message){
10         System.out.println("接收到消息:" + message);
11         //收到消息后的处理, 代码省略
12     }
13 }

```

4.4 启动服务, 观察结果

访问订单系统的接口, 模拟下单请求: <http://127.0.0.1:8080/order/createOrder>

在物流系统的日志中, 可以观察到, 通过RabbitMQ, 成功把下单信息传递给了物流系统

```

1 接收到消息:下单成功, 订单ID:c0c25851-6fe0-49dd-bf93-074667732432

```

4.5 发送消息格式为对象

如果通过 `RabbitTemplate` 发送一个对象作为消息, 我们需要对该对象进行序列化. Spring AMQP推荐使用JSON序列化, Spring AMQP提供了 `Jackson2JsonMessageConverter` 和 `MappingJackson2MessageConverter` 等转换器, 我们需要把一个 `MessageConverter` 设置到 `RabbitTemplate` 中.

```

1 @Bean
2 public Jackson2JsonMessageConverter jackson2JsonMessageConverter() {
3     return new Jackson2JsonMessageConverter();
4 }
5
6 @Bean
7 public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
8     RabbitTemplate template = new RabbitTemplate(connectionFactory);
9     template.setMessageConverter(jackson2JsonMessageConverter()); // 设置消息转换器
10    return template;
11 }

```

定义一个对象

```

1 @AllArgsConstructor
2 @NoArgsConstructor
3 @Data
4 public class OrderInfo {
5     private String orderId;
6     private String name;
7     private long price;
8 }

```

生产者代码:

```

1 @RequestMapping("/createOrder")
2 public String createOrder(){
3     //下单相关操作, 比如参数校验, 操作数据库等, 代码省略
4     //发送消息通知
5     String orderId = UUID.randomUUID().toString();
6     OrderInfo orderInfo = new OrderInfo(orderId, "商品", 536);
7     rabbitTemplate.convertAndSend("", "order.create", orderInfo);
8     return "下单成功";
9 }

```

消费者代码:

```

1 @Component
2 public class OrderCreateListener {
3     @RabbitHandler
4     @RabbitListener(queues = "order.create")
5     public void ListenerQueue(OrderInfo message){
6         System.out.println("接收到消息:" + message);
7         //收到消息后的处理, 代码省略
8     }
9 }

```

`@RabbitListener(queues = "order.create")` 可以加在类上, 也可以加在方法上, 用于定于一个类或者方法作为消息的监听器。

`@RabbitHandler` 是一个方法级别的注解, 当使用 `@RabbitHandler` 注解时, 这个方法将被调用处理特定的消息。