

- 一、回顾RabbitMQ基础概念
- 二、RabbitMQ基础编程模型
 - 1、maven依赖
 - 2、基础编程模型
 - step1、首先创建连接，获取Channel
 - step2、声明Exchange
 - step3、声明queue
 - step4、声明Exchange与Queue的绑定关系
 - step5、Producer根据应用场景发送消息到queue
 - step6、Consumer消费消息
 - step7、完成以后关闭连接，释放资源
 - 3、关于消息监听与回溯
- 三、RabbitMQ常用的消息场景
 - 1: hello world体验
 - 2: Work queues 工作序列
 - 3: Publish/Subscribe 订阅 发布 机制
 - 4: Routing 基于内容的路由
 - 5: Topics 基于话题的路由
 - 6: Publisher Confirms 发送者消息确认
 - 7: Headers 头部路由机制
- 四、SpringBoot集成RabbitMQ
 - 1: 引入依赖
 - 2: 配置关键参数
 - 3: 声明Exchange, Queue和Binding
 - 4: 使用RabbitmqTemplate对象发送消息
 - 5: 使用@RabbitListener注解声明消费者
- 五、章节总结

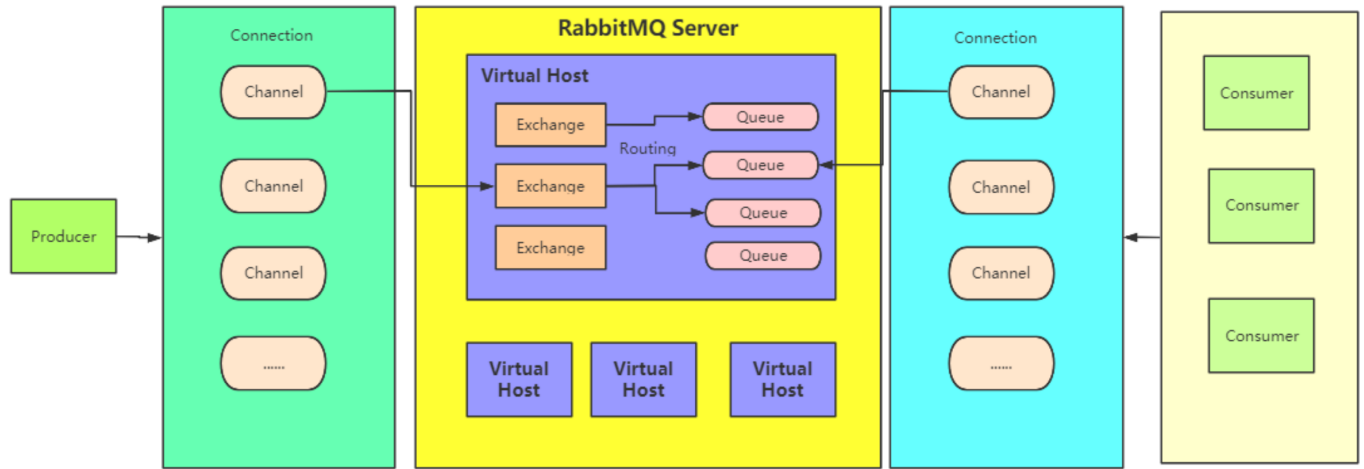
RabbitMQ客户端应用开发实战

-- 楼兰

这一章节我们将快速完成RabbitMQ客户端基础功能的开发实战。

一、回顾RabbitMQ基础概念

![] (file:///Users/roykingw/Desktop/a-work/RabbitMQ/%E7%AC%AC%E5%85%AD%E6%9C%9FVIP/img/RabbitMQ%E6%A6%82%E5%BF%B5.png?lastModify=1722150281)



这个RabbitMQ的核心组件， 是进行应用开发的基础。

二、RabbitMQ基础编程模型

RabbitMQ提供了很多种主流编程语言的客户端支持。这里我们只分析Java语言的客户端。

上一章节提供了一个简单的RabbitMQ客户端实现， 这里， 就以上一章节的客户端示例为基础， 了解下RabbitMQ客户端开发的基础流程， 为后续理解各种业务场景打下基础。

1、maven依赖

```
<dependency>
  <groupId>com.rabbitmq</groupId>
  <artifactId>amqp-client</artifactId>
  <version>5.21.0</version>
</dependency>
```

amqp是一种标准的消息驱动实现协议， RabbitMQ是对这一协议的具体实现。由于协议具有稳定性， 所以， 通常RabbitMQ的客户端不一定要求与服务端版本一致。

2、基础编程模型

结合上一章节的第一个案例， 详细拆解每个核心概念的具体实现方式以及功能扩展。

step1、首先创建连接， 获取Channel

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost(HOST_NAME);
factory.setPort(HOST_PORT);
factory.setUsername(USER_NAME);
factory.setPassword(PASSWORD);
factory.setVirtualHost(VIRTUAL_HOST);
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
```

通常情况下，我们在一个客户端里都只是创建一个Channel就可以了，因为一个Channel只要不关闭，是可以一直复用的。但是，如果你想要创建多个Channel，要注意一下Channel冲突的问题。

在创建channel时，可以在createChannel方法中传入一个分配的int参数channelNumber。这个ChannelNumber就会作为Channel的唯一标识。而RabbitMQ防止ChannelNumber重复的方式是：如果对应的Channel没有创建过，就会创建一个新的Channel。但是如果ChannelNumber已经创建过一个Channel了，这时就会返回一个null。

step2、声明Exchange

关键代码：

```
channel.exchangeDeclare(String exchange, String
type, boolean durable, boolean autoDelete, Map<String, Object> arguments) throws IOException;
```

api说明：

Declare an exchange.

Params:

exchange – the name of the exchange

type – the exchange type

durable – true if we are declaring a durable exchange (the exchange will survive a server restart)

autoDelete – true if the server should delete the exchange when it is no longer in use

arguments – other properties (construction arguments) for the exchange

Returns:

a declaration-confirm method to indicate the exchange was successfully declared

Throws:

IOException – if an error is encountered See Also: AMQP.Exchange.Declare, AMQP.Exchange.DeclareOk

在声明Exchange时需要注意，如果Broker上没有对应的Exchange，那么RabbitMQ会自动创建一个新的交换机。但是如果Broker上已经有了这个Exchange，那么你声明时的这些参数需要与Broker上的保持一致。如果不一致就会报错。

声明Exchange需要传入的参数，不用死记硬背。实际上这些参数，包括最后的arguments，都可以在管理控制台上看到。并且，不同版本的RabbitMQ，参数是有变动的。这些最好都以管理控制台上的为准。

▼ Add a new exchange

Virtual host:

Name: *

Type: ← 队列类型

Durability:

Auto delete: ?

Internal: ?

Arguments: = String ▾

Add Alternate exchange ?

重点需要关注的是Exchange的类型。Exchange共有四种不同类型，对应了四种不同的消息分发逻辑。这里暂时先不用管，下面介绍消息场景时会详细分析。

step3、声明queue

关键代码：

```
channel.queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete,
Map<String, Object> arguments);
```

api说明：

Declare a queue

Parameters:

queue the name of the queue

durable true if we are declaring a durable queue (the queue will survive a server restart)

exclusive true if we are declaring an exclusive queue (restricted to this connection)

autoDelete true if we are declaring an autodelete queue (server will delete it when no longer in use)

arguments other properties (construction arguments) for the queue

Returns:

a declaration-confirm method to indicate the queue was successfully declared

Throws:

java.io.IOException - if an error is encountered

See Also:

com.rabbitmq.client.AMQP.Queue.Declare

com.rabbitmq.client.AMQP.Queue.DeclareOk

与Exchange一样，如果你声明的Queue在Broker上不存在，RabbitMQ会创建一个新的队列。但是如果Broker上已经有了这个队列，那么声明的属性必须和Broker上的队列保持一致，否则也会报错。

声明Queue时，同样大部分的参数是可以从管理平台看到的。比如Durability，AutoDelete以及后面的arguments参数可以传哪些参数，都可以从页面上看到。

同样，queue的大部分参数，也以管理平台中的为准。例如在创建队列时，之前的版本会有一个AutoDelete选项，但是现在这个版本已经没有这个选项了。另外，下方的参数也与老版本有一些不同。

▼ Add a new queue

Virtual host:

Type:

Name: *

Durability:

Arguments: =

Add ? | ? | ?
 ? | ? | ?
 ? | ?
| ?

Durability表示是否持久化。Durable选项表示会将队列的消息写入硬盘，这样服务重启后这些消息就不会丢失。而另外一个选项Transient表示不持久化，消息只在内存中流转。这样服务重启后这些消息就会丢失。当然这也意味着消息读写的效率会比较高。

但是与Exchange不同的是，在管理控制台上，对列有个Type参数，这个参数并没有在API中体现。这里是有历史版本的原因的，也有不同类型的对列实现方式不同的原因。例如对于Quorum和Stream类型，根本没有Durability参数，因为他们的消息默认就是必须要持久化的。后面对列的详细参数也会有很大的区别。

客户端API中默认只能声明Classic类型对列。如果需要声明其他类型的对列，只能通过后面的arguments参数来区分。

如果要声明一个Quorum队列，则只需要在后面的arguments中传入一个参数，**x-queue-type**，参数值设定为**quorum**。

```
Map<String,Object> params = new HashMap<>();
params.put("x-queue-type","quorum");
//声明Quorum队列的方式就是添加一个x-queue-type参数，指定为quorum。默认是classic
channel.queueDeclare(QueueName, true, false, false, params);
```

注意：1、对于Quorum类型，durable参数就必须是true了，设置成false的话，会报错。同样，exclusive参数必须设置为false

如果要声明一个Stream队列，则 **x-queue-type**参数要设置为 **stream**。

```
Map<String,Object> params = new HashMap<>();
params.put("x-queue-type","stream");
params.put("x-max-length-bytes", 20_000_000_000L); // maximum stream size: 20 GB
params.put("x-stream-max-segment-size-bytes", 100_000_000); // size of segment files: 100
MB
channel.queueDeclare(QueueName, true, false, false, params);
```

注意：1、同样，durable参数必须是true，exclusive必须是false。对于这两种队列，这两个参数是不可以选择的。

2、x-max-length-bytes 表示日志文件的最大字节数。x-stream-max-segment-size-bytes 每一个日志文件的最大大小。这两个是可选参数，通常为了防止stream日志无限制累计，都会配合stream队列一起声明。

3、stream类型的对列，并不能像之前两种对列一样使用。例如前两种对列类型，声明消费者后，可以从控制台直接发消息，消费者端就能接受到。但是Stream类型是接收不到的。具体原因，在后面介绍几种对列类型的区别时再详细分析。

实际项目中用得最多的是RabbitMQ的Classic经典队列，但是从RabbitMQ官网就能看到，目前RabbitMQ更推荐的是使用Quorum队列。至于Stream队列目前企业用得还比较少。

step4、声明Exchange与Queue的绑定关系

关键代码：

```
channel.queueBind(String queue, String exchange, String routingKey) throws IOException;
```

api说明：

Bind a queue to an exchange, with no extra arguments.

Params:

queue – the name of the queue

exchange – the name of the exchange

routingKey – the routing key to use for the binding

Returns:

a binding-confirm method if the binding was successfully created

Throws:

IOException – if an error is encountered

See Also:

AMQP.Queue.Bind,

AMQP.Queue.BindOk

如果我们声明了Exchange和Queue，那么就还需要声明Exchange与Queue的绑定关系Binding。有了这些Binding，Exchange才可以知道Producer发送过来的消息将要分发到哪些Queue上。这些Binding涉及到消息的不同分发逻辑，与Exchange和Queue一样，如果Broker上没有建立绑定关系，那么RabbitMQ会按照客户端的声明，创建这些绑定关系。但是如果声明的Binding存在了，那么就需要与Broker上的保持一致。

另外，在声明Binding时，还可以传入两个参数，routingKey和props。这两个参数都是跟Exchange的消息分发逻辑有关。同样会留到后面进行详细分享。

step5、Producer根据应用场景发送消息到queue

关键代码：

```
channel.basicPublish(String exchange, String routingKey, BasicProperties props, message.getBytes("UTF-8"));
```

api说明：

Publish a message. Publishing to a non-existent exchange will result in a channel-level protocol exception, which closes the channel. Invocations of Channel#basicPublish will eventually block if a resource-driven alarm is in effect.

Parameters:

exchange the exchange to publish the message to

routingKey the routing key

props other properties for the message - routing headers etc

body the message body

这其中Exchange如果不需要，传个空字符串就行了。routingKey跟Exchange的消息分发逻辑有关。后面介绍业务场景时会详细说明。

然后关于props参数，可以传入一些消息相关的属性。这些属性你同样不用死记。管理控制台上明确的说明。

OverviewConnectionsChannelsExchangesQueues and StreamsAdmin

Queue streamQueue in virtual host /mirror

OverviewConsumers (0)Bindings (1)Publish message

Message will be published to exchange: Headers: ? Properties: ? Payload: stream

You can set other message properties here (delivery mode and headers are pulled out as the most common cases).

Invalid properties will be ignored. Valid properties are:

- content_type
- content_encoding
- priority
- correlation_id
- reply_to
- expiration
- message_id
- timestamp
- type
- user_id
- app_id
- cluster_id

可选的参数

Close

props的这些配置项，可以用RabbitMQ中提供的一个Builder对象来构建。

```
AMQP.BasicProperties.Builder builder = new AMQP.BasicProperties.Builder();
//对应页面上的Properties部分，传入一些预定的参数值。
builder.deliveryMode(MessageProperties.PERSISTENT_TEXT_PLAIN.getDeliveryMode());
builder.priority(MessageProperties.PERSISTENT_TEXT_PLAIN.getPriority());
//builder.headers(headers);对应页面上的Headers部分。传入自定义的参数值
builder.build()
AMQP.BasicProperties prop = builder.build();
```

在发送消息时要注意一下消息的持久化问题。MessageProperties.PERSISTENT_TEXT_PLAIN是RabbitMQ提供的持久化消息的默认配置。而RabbitMQ中消息是否持久化不光取决于消息，还取决于Queue。通常为了保证消息安全，会将Queue和消息同时声明为持久化。

step6、Consumer消费消息

定义消费者，消费消息进行处理，并向RabbitMQ进行消息确认。确认了之后就表明这个消息已经消费完了，否则RabbitMQ还会继续发起重试。

Consumer主要有两种消费方式

1、Push推模式

Consumer等待rabbitMQ 服务器将message推送过来再消费。一般是启一个一直挂起的线程来等待。

关键代码

```
channel.basicConsume(String queue, boolean autoAck, Consumer callback);
```

api说明:

Start a non-nolocal, non-exclusive consumer, with a server-generated consumerTag.

Parameters:

queue the name of the queue

autoAck true if the server should consider messages acknowledged once delivered; false if the server should expect explicit acknowledgements

callback an interface to the consumer object Returns: the consumerTag generated by the server

2、Pull拉模式

Comsumer主动到rabbitMQ服务器上去拉取messge进行消费。

关键代码

```
GetResponse response = channel.basicGet(QUEUE_NAME, boolean autoAck);
```

api说明:

Retrieve a message from a queue using com.rabbitmq.client.AMQP.Basic.Get

Parameters:

queue the name of the queue

autoAck true if the server should consider messages acknowledged once delivered; false if the server should expect explicit acknowledgements Returns: a GetResponse containing the retrieved message data

Throws:

java.io.IOException - if an error is encountered

See Also:

com.rabbitmq.client.AMQP.Basic.Get

com.rabbitmq.client.AMQP.Basic.GetOk

com.rabbitmq.client.AMQP.Basic.GetEmpty

其中需要注意点的是autoAck，自动应答。Consumer处理完一条消息后，需要给Broker一个Ack应答，Broker就不会重复投递消息。如果Broker没有收到应答，就会重复投递消息。

实际开发中，更建议使用Push推模式，这样消息处理能够比较及时，并且也不会给服务端带来重复查询的压力。

step7、完成以后关闭连接，释放资源

```
channel.close();
conection.clouse();
```

用完之后主动释放资源。如果不主动释放的话，大部分情况下，过一段时间RabbitMQ也会将这些资源释放掉，但是这就需要额外消耗系统资源。

3、关于消息监听与回溯

上一节只是列出了用得最多的几个常用的方法。但是，RabbitMQ的客户端还提供了很多重载方法和扩展方法，这些需要自行掌握。

例如，在消费消息是，channel还提供了重载的方法：

```
String basicConsume(String queue, DeliverCallback deliverCallback, CancelCallback cancelCallback,
ConsumerShutdownSignalCallback shutdownSignalCallback) throws IOException;
```

这些callback实际上就是RabbitMQ在Consumer中保留的业务扩展点。这些拓展的方法，学习的时候可能没有太大的作用。但是，如果你没有提前总结，等到真正开发的时候，肯定想不到这些扩展点。

另外，你也可以自己做一个小案例。

```

public class CallbackConsumer {
    private static final String EXCHANGE_NAME="callbackExchange";
    private static final String ALTER_EXCHANGE_NAME="alterExchange";
    private static final String QUEUE_NAME = "callbackQueue";

    public static void main(String[] args) throws Exception {
        Connection connection = RabbitMQUtil.getConnection();
        Channel channel = connection.createChannel();
        Map<String,Object> params = new HashMap<>();
        params.put("alternate-exchange",ALTER_EXCHANGE_NAME);
        channel.exchangeDeclare(EXCHANGE_NAME,BuiltinExchangeType.DIRECT,true,false,params);
        channel.exchangeDeclare(ALTER_EXCHANGE_NAME,BuiltinExchangeType.DIRECT,true,false,null);

        channel.queueDeclare(QUEUE_NAME, true, false, false, null);

        channel.queueBind(QUEUE_NAME,EXCHANGE_NAME,"key1");

        channel.basicConsume(QUEUE_NAME, new DeliverCallback() {
            @Override
            public void handle(String consumerTag, Delivery message) throws IOException {
                long deliveryTag = message.getEnvelope().getDeliveryTag();
                String correlationId = message.getProperties().getCorrelationId();
                System.out.println("received message consumerTag: " + consumerTag + ";
message: " + new String(message.getBody())+";deliveryTag: "+deliveryTag+";correlationId:
"+correlationId);

                channel.basicAck(deliveryTag,false);
            }
        }, new CancelCallback() {
            @Override
            public void handle(String consumerTag) throws IOException {
                System.out.println("canceled message consumerTag: " + consumerTag + ";
");
            }
        }, new ConsumerShutdownSignalCallback() {
            @Override
            public void handleShutdownSignal(String consumerTag, ShutdownSignalException
sig) {
                System.out.println("consumer shutdown message consumerTag: " +
consumerTag + "; Exception: " + sig);
            }
        });
    }
}

```

然后往队列里多次发送消息。你能看到这样的结果：

```

received message consumerTag: amq.ctag-6d_w_WZWpu66H61kzuUfTw; message: message;deliveryTag: 1
received message consumerTag: amq.ctag-6d_w_WZWpu66H61kzuUfTw; message: message;deliveryTag: 2
received message consumerTag: amq.ctag-6d_w_WZWpu66H61kzuUfTw; message: message;deliveryTag: 3
received message consumerTag: amq.ctag-6d_w_WZWpu66H61kzuUfTw; message: message;deliveryTag: 4

```

这些就是从DeliverCallback接口反应出来的结果。从这个结果你是否能理解之前没有详细分享的consumerTag和deliveryTag到底是什么东西？是的，**consumerTag**代表的是与客户端的一个会话。**deliveryTag**代表的是这个**Channel**处理的一条消息。这都是RabbitMQ服务端分配的一些内部参数。日后，如果你希望对Consumer处理的每一条消息增加溯源过功能时，把consumerTag+deliveryTag作为消息编号，保存下来，这就是一个不错的设计。

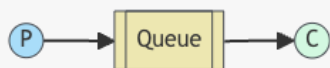
三、RabbitMQ常用的消息场景

这一部分是学习以及使用RabbitMQ的重中之重。RabbitMQ的客户端API使用是比较简单的，但是如果面临复杂业务时，要用好RabbitMQ还是需要一些功力的。对这一部分的学习，理解业务场景是最为重要的。

具体参见 <https://www.rabbitmq.com/tutorials>。其中可以看到，RabbitMQ官方提供了总共七种典型的使用场景，这其中，6 RPC部分是使用RabbitMQ来实现RPC远程调用，这个场景在实际开发中通常不会选择使用MQ来实现，所以后续不会分享。另外，接下来我会给你补充一种RabbitMQ比较小众的Header路由机制。

1. "Hello World!"

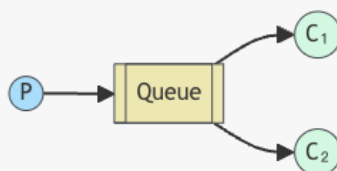
The simplest thing that does *something*



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

2. Work Queues

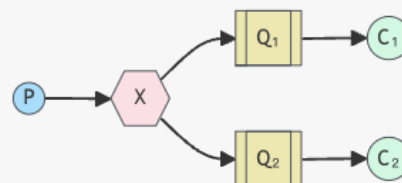
Distributing tasks among workers (the *competing consumers pattern*)



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

3. Publish/Subscribe

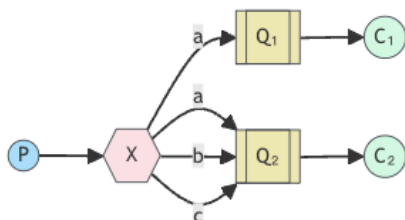
Sending messages to many consumers at once



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

4. Routing

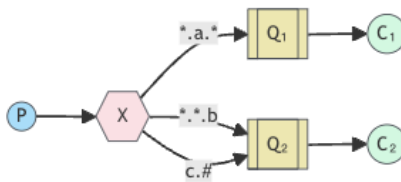
Receiving messages selectively



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

5. Topics

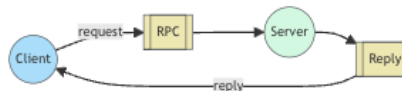
Receiving messages based on a pattern (topics)



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

6. RPC

Request/reply pattern example



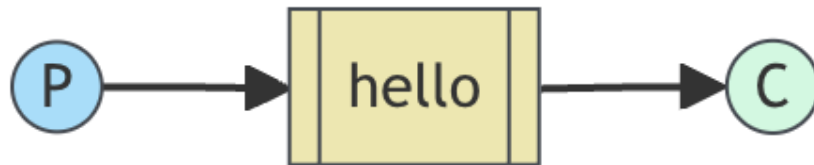
- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Spring AMQP

7. Publisher Confirms

Reliable publishing with publisher confirms

- Java
- C#

1: hello world体验



lastModify=1722150281)

最直接的方式，P端发送一个消息到一个指定的queue，中间不需要任何exchange规则。C端按queue方式进行消费。

关键代码：(其实关键的区别也就是几个声明上的不同。)

producer:

```
channel.queueDeclare(Queue_NAME, false, false, false, null);
channel.basicPublish("", Queue_NAME, null, message.getBytes("UTF-8"));
```

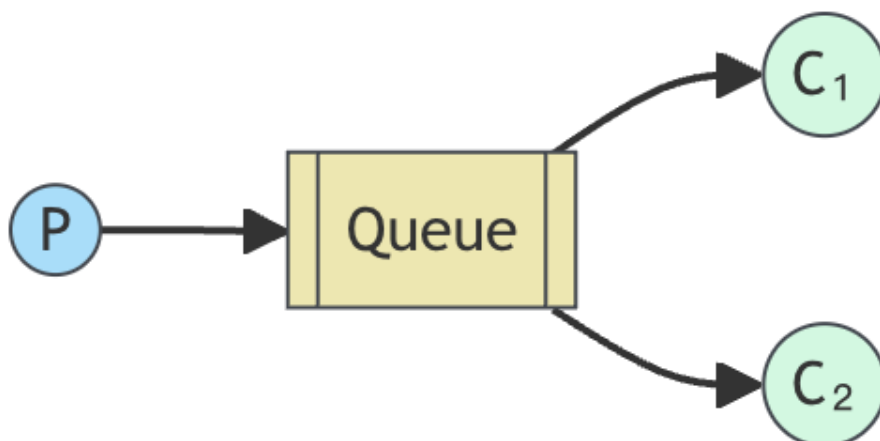
consumer:

```
channel.queueDeclare(Queue_NAME, false, false, false, null);
```

这个模式基本就是我们之前演示的Demo。最简单的一个入门案例。

2: Work queues 工作序列

这是RabbitMQ最基础也是最常用的一种工作机制。



lastModify=1722150281)

工作任务模式，领导部署一个任务，由下面的一个员工来处理。只关心任务被正确处理，不关心给谁处理。

Producer消息发送给queue，多个Consumer同时往队列上消费消息。

关键代码： ===》 producer: 将消息直接到Queue上。

```
channel.queueDeclare(TASK_QUEUE_NAME, true, false, false, null); //任务一般是不能因为消息中间件的服务而被耽误的，所以durable设置成了true，这样，即使rabbitMQ服务断了，这个消息也不会消失
channel.basicPublish("", TASK_QUEUE_NAME, MessageProperties.PERSISTENT_TEXT_PLAIN,
    message.getBytes("UTF-8"));
```

Consumer: 每次拉取一条消息。

```
channel.queueDeclare(TASK_QUEUE_NAME, true, false, false, null);
channel.basicQos(1);
channel.basicConsume(TASK_QUEUE_NAME, false, consumer);
```

这个模式应该是最常用的模式，也是官网讨论比较详细的一种模式。有几个问题需要注意：

- **1: Consumer对每个消息必须应答**

Consumer端每消费完一个消息需要给服务端一个ack应答，这个应答可以是手动应答，也可以是自动应答。如果Consumer一直没有给服务端应答，那么服务端会不断地将这条消息重复进行投递，这就会不断地消耗系统资源。这就是Poison Message(毒消息)。这是使用RabbitMQ时非常容易犯的错误。

- **2, RabbitMQ并不完全保证消息安全**

关键的message不能因为服务出现问题而被忽略。如果想要保证消息不丢失，在RabbitMQ中，需要同时将队列和消息的durable属性都设置成true。

但是，官方也明确说了，就算把队列和消息都设置为durable，RabbitMQ也并不能保证消息完全不丢失。因为RabbitMQ对于持久化的消息，会写入到文件当中。但是此时只是写入到PageCache缓存中，而不是磁盘。缓存中的数据断电就会丢失，只有磁盘中的数据断电才不会丢失。消息从PageCache到写入磁盘，需要进行一次刷盘。RabbitMQ并不会对每个消息都执行刷盘操作，而操作系统进行刷盘操作是有一定的间隔的，因此，如果出现服务器异常断电，RabbitMQ在这个层面是可能造成消息丢失的。

其实数据刷盘是所有应用程序都要面临的问题。而RocketMQ对这个问题做了单独的设计。在RabbitMQ中如何处理这种问题呢？就要用到后面会介绍的Publisher Confirms机制。

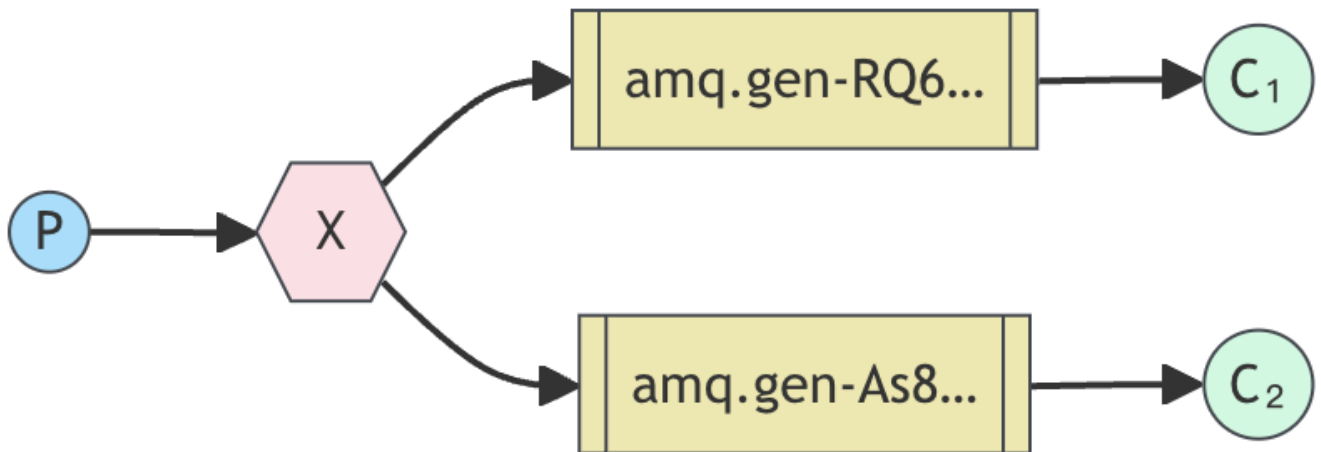
- **3, 消息如何在多个Consumer之间分发**

这里，RabbitMQ默认是采用的fair dispatch，也叫round-robin模式，就是把消息轮询，在所有consumer中轮流发送。这种方式，没有考虑消息处理的复杂度以及consumer的处理能力。而他们改进后的方案，是consumer可以向服务器声明一个**prefetchCount**，我把他叫做预处理能力值。**channel.basicQos(prefetchCount)**;表示当前这个consumer可以同时处理几个message。这样服务器在进行消息发送前，会检查这个consumer当前正在处理中的message(message已经发送，但是未收到consumer的basicAck)有几个，如果超过了这个consumer节点的能力值，就不再往这个consumer发布。这种模式，官方也指出还是有问题的，消息有可能全部阻塞，所有consumer节点都超过了能力值，那消息就阻塞在服务器上，这时需要自己及时发现这个问题，采取措施，比如增加consumer节点或者其他策略

Note about queue size If all the workers are busy, your queue can fill up. You will want to keep an eye on that, and maybe add more workers, or have some other strategy.

3: Publish/Subscribe 订阅发布机制

type为**fanout**的exchange：



这个机制是对上面的一种补充。也就是把producer与Consumer进行进一步的解耦。producer只负责发送消息，至于消息进入哪个queue，由exchange来分配。如上图，就是把producer发送的消息，交由exchange同时发送到两个queue里，然后由不同的Consumer去进行消费。

关键代码 ==》 producer: //只负责往exchange里发消息，后面的事情不管。

```
channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes("UTF-8"));
```

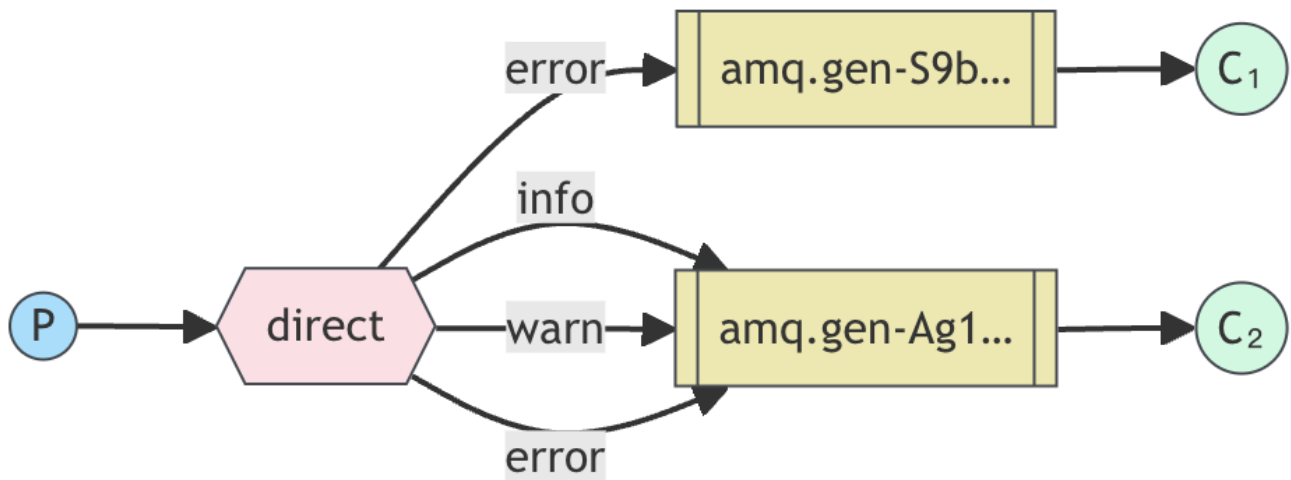
binding: //将消费的目标队列绑定到exchange上。

```
channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
String queueName = channel.queueDeclare().getQueue();
channel.queueBind(queueName, EXCHANGE_NAME, "");
```

关键处就是type为"fanout" 的exchange,这种类型的exchange只负责往所有已绑定的队列上发送消息。

4: Routing 基于内容的路由

type为"direct" 的exchange



这种模式一看图就清晰了。在上一章 exchange 往所有队列发送消息的基础上，增加一个路由配置，指定exchange如何将不同类别的消息分发到不同的queue上。

关键代码==> Producer 同样是往Exchange发送消息，但是需要指定一个routingKey

```
channel.exchangeDeclare(EXCHANGE_NAME, "direct");
channel.basicPublish(EXCHANGE_NAME, routingKey, null, message.getBytes("UTF-8"));
```

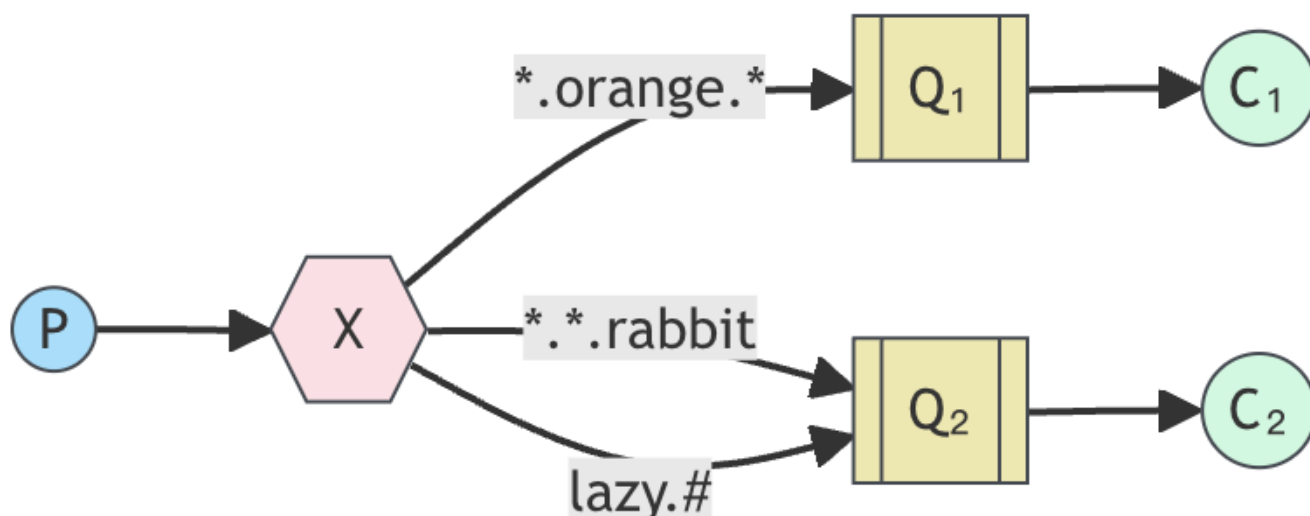
Bindings 在建立Exchange和Queue的绑定关系时，需要指定routingKey。

```
channel.exchangeDeclare(EXCHANGE_NAME, "direct");
channel.queueBind(queueName, EXCHANGE_NAME, routingKey1);
channel.queueBind(queueName, EXCHANGE_NAME, routingKey2);
channel.basicConsume(queueName, true, consumer);
```

消息就会根据routingkey转发到对应的Queue上，然后给消费者处理。

5: Topics 基于话题的路由

type为"topic" 的exchange



这个模式也就在Routing模式的基础上，对routingKey进行了模糊匹配。单词之间用,隔开，* 代表一个具体的单词。# 代表0个或多个单词。

关键代码==> Producer，依然是往Exchange发送消息，并且需要带上routingKey。

```
channel.exchangeDeclare(EXCHANGE_NAME, "topic");
channel.basicPublish(EXCHANGE_NAME, routingKey, null, message.getBytes("UTF-8"));
```

Bindings: 绑定routingKey

```
channel.exchangeDeclare(EXCHANGE_NAME, "topic");
channel.queueBind(queueName, EXCHANGE_NAME, routingKey1);
channel.queueBind(queueName, EXCHANGE_NAME, routingKey2);
channel.basicConsume(queueName, true, consumer);
```

消息在根据routingKey进行转发时，会进行模糊匹配。

6: Publisher Confirms 发送者消息确认

RabbitMQ的消息可靠性是非常高的，但是他以往的机制都是保证消息发送到了MQ之后，可以推送到消费者消费，不会丢失消息。但是发送者发送消息是否成功是没有保证的。我们可以回顾下，发送者发送消息的基础API：Producer.basicPublish方法是没有返回值的，也就是说，一次发送消息是否成功，应用是不知道的，这在业务上就容易造成消息丢失。而这个模块就是通过给发送者提供一些确认机制，来保证这个消息发送的过程是成功的。

发送者确认模式默认是不开启的，所以如果需要开启发送者确认模式，需要手动在channel中进行声明。

```
channel.confirmSelect();
```

在官网的示例中，重点解释了三种策略：

1、发布单条消息

即发布一条消息就确认一条消息。核心代码：

```
for (int i = 0; i < MESSAGE_COUNT; i++) {
    String body = String.valueOf(i);
    channel.basicPublish("", queue, null, body.getBytes());
    channel.waitForConfirmsOrDie(5_000);
}
```

channel.waitForConfirmsOrDie(5_000);这个方法就会在channel端等待RabbitMQ给出一个响应，用来表明这个消息已经正确发送到了RabbitMQ服务端。但是要注意，这个方法会同步阻塞channel，在等待确认期间，channel将不能再继续发送消息，也就是说会明显降低集群的发送速度即吞吐量。

官方说明了，其实channel底层是异步工作的，会将channel阻塞住，然后异步等待服务端发送一个确认消息，才解除阻塞。但是我们在使用时，可以把他当作一个同步工具来看待。

然后如果到了超时时间，还没有收到服务端的确认机制，那就会抛出异常。然后通常处理这个异常的方式是记录错误日志或者尝试重发消息，但是尝试重发时一定要注意不要使程序陷入死循环。

2、发送批量消息

之前单条确认的机制会对系统的吞吐量造成很大的影响，所以稍微中和一点的方式就是发送一批消息后，再一起确认。

核心代码：

```
int batchSize = 100;
int outstandingMessageCount = 0;

long start = System.nanoTime();
for (int i = 0; i < MESSAGE_COUNT; i++) {
    String body = String.valueOf(i);
    ch.basicPublish("", queue, null, body.getBytes());
    outstandingMessageCount++;

    if (outstandingMessageCount == batchSize) {
        ch.waitForConfirmsOrDie(5_000);
        outstandingMessageCount = 0;
    }
}

if (outstandingMessageCount > 0) {
    ch.waitForConfirmsOrDie(5_000);
}
```

这种方式可以稍微缓解下发送者确认模式对吞吐量的影响。但是也有个固有的问题就是，当确认出现异常时，发送者只能知道是这一批消息出问题了，而无法确认具体是哪一条消息出了问题。所以接下来就需要增加一个机制能够具体对每一条发送出错的消息进行处理。

3、异步确认消息

实现的方式也比较简单，Producer在channel中注册监听器来对消息进行确认。核心代码就是一个：

```
channel.addConfirmListener(ConfirmCallback var1, ConfirmCallback var2);
```

按说监听只要注册一个就可以了，那为什么这里要注册两个呢？成功一个，失败一个

然后关于这个ConfirmCallback，这是个监听器接口，里面只有一个方法：void handle(long sequenceNumber, boolean multiple) throws IOException; 这个方法中的两个参数，

- `sequenceNumber`: 这个是一个唯一的序列号, 代表一个唯一的消息。在RabbitMQ中, 他的消息体只是一个二进制数组, 默认消息是没有序列号的。那么在回调的时候, `Producer`怎么知道是哪一条消息成功或者失败呢? RabbitMQ提供了一个方法 `int sequenceNumber = channel.getNextPublishSeqNo();` 来生成一个全局递增的序列号, 这个序列号将会分配给新发送的那一条消息。然后应用程序需要自己来将这个序列号与消息对应起来。没错! 是的! 需要客户端自己去做对应!
- `multiple`: 这个是一个Boolean型的参数。如果是false, 就表示这一次只确认了当前一条消息。如果是true, 就表示RabbitMQ这一次确认了一批消息, 在`sequenceNumber`之前的所有消息都已经确认完成了。

4、三种确认机制的区别

这三种确认机制都能够提升Producer发送消息的安全性。通常情况下, 第三种异步确认机制的性能是最好的。

实际上, 在当前版本中, Publisher不光可以确认消息是否到了Exchange, 还可以确认消息是否从Exchange成功路由到了Queue。在Channel中可以添加一个ReturnListener。这个ReturnListener就会监控到这一部分发送成功了, 但是无法被Consumer消费到的消息。

那接下来, 更进一步, 这部分消息要如何处理呢? 还记得在Web控制台声明Exchange交换机的时候, 还可以添加一个属性吗? 是的。当前版本在Exchange交换机中可以添加一个属性`alternate-exchange`。这个属性可以指向另一个Exchange。其作用, 就是将这些无法正常路由的消息转到另外的Exchange进行兜底处理。

7、Headers 头部路由机制

在官网的体验示例中, 还有一种路由策略并没有提及, 那就是Headers路由。其实官网之所以没有过多介绍, 就是因为这种策略在实际中用得比较少, 但是在某些比较特殊的业务场景, 还是挺好用的。

官网示例中的集中路由策略, `direct`, `fanout`, `topic`等这些Exchange, 都是以`routingkey`为关键字来进行消息路由的, 但是这些Exchange有一个普遍的局限就是都是只支持一个字符串的形式, 而不支持其他形式。Headers类型的Exchange就是一种忽略`routingKey`的路由方式。他通过Headers来进行消息路由。这个headers是一个键值对, 发送者可以在发送的时候定义一些键值对, 接受者也可以在绑定时定义自己的键值对。当键值对匹配时, 对应的消费者就能接收到消息。匹配的方式有两种, 一种是`all`, 表示需要所有的键值对都满足才行。另一种是`any`, 表示只要满足其中一个键值就可以了。

关于Headers路由的示例, 首先在Web管理页面上, 可以看到默认创建了一个`amqp.headers`这样的Exchange交换机, 这个就是Headers类型的路由交换机。

在Consumer端, 声明Queue与Exchange绑定关系时, 可以增加声明headers, 表明自己对哪些信息感兴趣。

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("x-match", "any"); //x-match:特定的参数。all表示必须全部匹配才算成功。any表示只要匹配一个就算成功。
headers.put("loglevel", "info");
headers.put("buslevel", "product");
headers.put("syslevel", "admin");

Connection connection = RabbitMQUtil.getConnection();
Channel channel = connection.createChannel();

channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.HEADERS);
String queueName = channel.queueDeclare("ReceiverHeader", true, false, false, null).getQueue();

channel.queueBind(queueName, EXCHANGE_NAME, routingKey, headers);
```

在Producer端, 发送消息时, 带上消息的headers相关属性。

```

public class EmitLogHeader {

    private static final String EXCHANGE_NAME = "logs";
    /**
     * exchange有四种类型， fanout topic headers direct
     * headers用得比较少，他是根据头信息来判断转发路由规则。头信息可以理解为一个Map
     * @param args
     * @throws Exception
     */
    public static void main(String[] args) throws Exception{
        // header模式不需要routingKey来转发，他是根据header里的信息来转发的。比如消费者可以只订阅LogLevel=info的
        // 消息。
        // 然而，消息发送的API还是需要一个routingKey。
        // 如果使用header模式来转发消息，routingKey可以用来存放其他的业务消息，客户端接收时依然能接收到这个
        // routingKey消息。
        String routingKey = "ourTestRoutingKey";
        // The map for the headers.
        Map<String, Object> headers = new HashMap<>();
        headers.put("loglevel", "error");
        headers.put("buslevel", "product");
        headers.put("syslevel", "admin");

        String message = "LOG INFO asdfasdf";

        Connection connection = RabbitMQUtil.getConnection();
        Channel channel = connection.createChannel();
        //发送者只管往exchange里发消息，而不用关心具体发到哪些queue里。
        channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.HEADERS);

        AMQP.BasicProperties.Builder builder = new AMQP.BasicProperties.Builder();
        builder.deliveryMode(MessageProperties.PERSISTENT_TEXT_PLAIN.getDeliveryMode());
        builder.priority(MessageProperties.PERSISTENT_TEXT_PLAIN.getPriority());
        builder.headers(headers);

        channel.basicPublish(EXCHANGE_NAME, routingKey, builder.build(), message.getBytes("UTF-8"));

        channel.close();
        connection.close();
    }
}

```

Headers交换机的性能相对比较低，因此官方并不建议大规模使用这种交换机，也没有把他列入基础的示例当中。

四、SpringBoot集成RabbitMQ

SpringBoot官方就集成了RabbitMQ，所以RabbitMQ与SpringBoot的集成是非常简单的。不过，SpringBoot集成RabbitMQ的方式是按照Spring的一套统一的MQ模型创建的，因此SpringBoot集成插件中对于生产者、消息、消费者等重要的对象模型，与RabbitMQ原生的各个组件有对应关系，但是并不完全相同。这一点需要在后续试验过程中加深理解。

1: 引入依赖

SpringBoot官方集成了RabbitMQ，只需要快速引入依赖包即可使用。RabbitMQ与SpringBoot集成的核心maven依赖就下面一个。

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

要特别注意下版本。不同版本下的配置方式会有变化。

然后所有的基础运行环境都在application.properties中进行配置。所有配置以spring.rabbitmq开头。通常按照示例进行一些基础的必要配置就可以跑了。关于详细的配置信息，可以参见RabbitProperties，源码中有各个字段说明。

2：配置关键参数

基础的运行环境参数以及生产者的一些默认属性配置都集中到了application.properties配置文件中。所有配置项都以spring.rabbitmq开头。

关于详细的配置信息，可以参见RabbitProperties类的源码，源码中有各个字段的简单说明。

如果需要更详细的配置资料，那就需要去官方的github仓库上去查了。github地址：<https://github.com/spring-projects/spring-amqp>。

3：声明Exchange，Queue和Binding

所有的exchange, queue, binding的配置，都需要以对象的方式声明。默认情况下，这些业务对象一经声明，应用就会自动到RabbitMQ上常见对应的业务对象。但是也是可以配置成绑定已有业务对象的。

业务对象的声明方式，具体请参见示例工程。

详细的属性声明，同样参见github仓库。

4：使用RabbitmqTemplate对象发送消息

生产者的所有属性都已经在application.properties配置文件中配置。项目启动时，就会在Spring容器中初始化一个RabbitmqTemplate对象，然后所有的发送消息操作都通过这个对象来进行。

5：使用@RabbitListener注解声明消费者

消费者都是通过@RabbitListener注解来声明。在@RabbitMQListener注解中包含了非常多对Queue进行定制的属性，大部分的属性都是有默认值的。例如ackMode默认是null，就表示自动应答。在日常开发过程中，通常都会简化业务模型，让消费者只要绑定队列消费即可。

使用SpringBoot框架集成RabbitMQ后，开发过程可以得到很大的简化，所以使用过程并不难，对照一下示例就能很快上手。但是，需要理解一下的是，SpringBoot集成后的RabbitMQ中的很多概念，虽然都能跟原生API对应上，但是这些模型中间都是做了转换的，比如Message，就不是原生RabbitMQ中的消息了。使用SpringBoot框架，尤其需要加深对RabbitMQ原生API的理解，这样才能以不变应万变，深入理解各种看起来简单，但是其实坑很多的各种对象声明方式。

五、章节总结

这一章节是与开发人员联系最紧密的部分，因此也是必须要掌握的重点，尤其是基础的编程模型。各种不同的业务场景其实都是将这些步骤在不同业务场景中落地。

对于这么多看起来差不多的API，首先，你一定需要自己动手敲敲这些代码，熟悉起来。熟悉之后，你一定要开始代入到一些业务场景当中，开始逐步去理解RabbitMQ各种机制的特点。针对各种不同的实际问题，尝试去理解RabbitMQ逐步解决这些问题的思路。

例如面试中经常被问到的如何保证消息不丢失的问题，RabbitMQ是如何处理的，你有想法了吗？不妨尝试做一下梳理。其实这样的问题，并不是简简单单的面试八股，非常体现你对产品的整体理解。理解的程度不同，能回答出来的方案也不同。

【有道云笔记】二、应用开发篇.md <https://note.youdao.com/s/Uuz61dz8>