第 4-6 课：Spring Boot RabbitMQ 详解

RabbitMQ 介绍

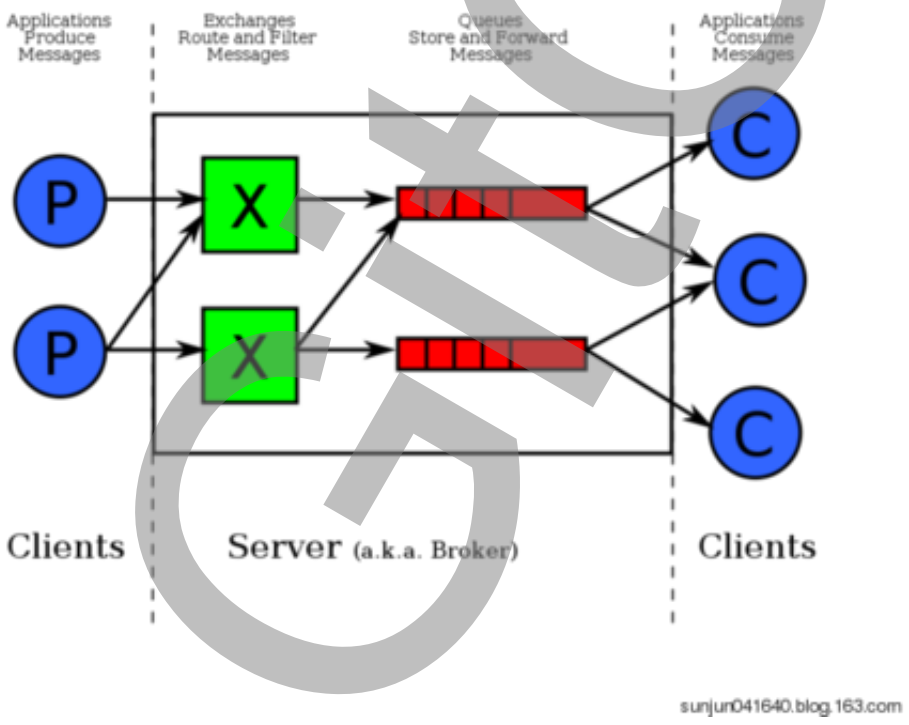
AMQP（Advanced Message Queuing Protocol，高级消息队列协议）是应用层协议的一个开放标准，为面向消息的中间件设计。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。

AMQP 的主要特征是面向消息、队列、路由（包括点对点和发布/订阅）、可靠性、安全。

RabbitMQ 是一个开源的 AMQP 实现，服务器端用 Erlang 语言编写，支持多种客户端，如 Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP 等，支持 AJAX。用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。

相关概念

通常我们谈到队列服务，会有三个概念：发消息者、队列、收消息者。RabbitMQ 在这个基本概念之上，多做了一层抽象，在发消息者和队列之间加入了交换器（Exchange）。这样发消息者和队列就没有直接联系，转而变成发消息者把消息给交换器，交换器根据调度策略再把消息再给队列。



- 左侧 P 代表生产者，也就是往 RabbitMQ 发消息的程序。
- 中间即是 RabbitMQ，其中包括了交换机和队列。
- 右侧 C 代表消费者，也就是往 RabbitMQ 拿消息的程序。

那么，其中比较重要的概念有 4 个，分别为：虚拟主机、交换机、队列和绑定。

- 虚拟主机：一个虚拟主机持有一组交换机、队列和绑定，为什么需要多个虚拟主机呢？很简单，RabbitMQ 当中，**用户只能在虚拟主机的粒度进行权限控制**。因此，如果需要禁止 A 组访问 B 组的交换机/队列/绑定，必须为 A 和 B 分别创建一个虚拟主机，每一个 RabbitMQ 服务器都有一个默认的虚拟主机“/”。
- 交换机：Exchange 用于转发消息，但是它不会做存储，如果没有 Queue bind 到 Exchange 的话，它会直接丢弃掉 Producer 发送过来的消息。

> 这里有一个比较重要的概念：**路由键** 。消息到交换机的时候，交互机会转发到对应的队列中，那么究竟转发到哪个队列，就要根据该路由键。

- 绑定：也就是交换机需要和队列相绑定，这其中如上图所示，是多对多的关系。

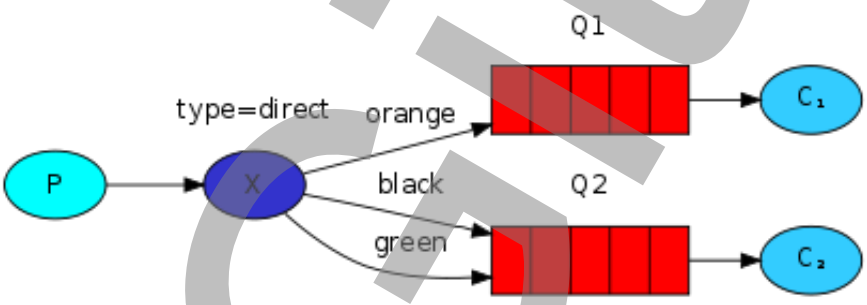
交换机（Exchange）

交换机的功能主要是接收消息并且转发到绑定的队列，交换机不存储消息，在启用 ack 模式后，交换机找不到队列，会返回错误。交换机有四种类型：Direct、topic、Headers and Fanout。

- Direct：其类型的行为是“先匹配、再投送”，即在绑定时设定一个 **routing_key**，消息的 **routing_key** 匹配时，才会被交换器投送到绑定的队列中去。
- Topic：按规则转发消息（最灵活）。
- Headers：设置 header attribute 参数类型的交换机。
- Fanout：转发消息到所有绑定队列。

Direct Exchange

Direct Exchange 是 RabbitMQ 默认的交换机模式，也是最简单的模式，根据 key 全文匹配去寻找队列。



第一个 X - Q1 就有一个 binding key，名字为 orange；X - Q2 就有 2 个 binding key，名字为 black 和 green。当消息中的**路由键**和这个 binding key 对应上的时候，那么就知道了该消息去到哪一个队列中。

> 注意：为什么 X 到 Q2 要有 black、green，2 个 binding key 呢，一个不就行了吗？这个主要是因为可能又有 Q3，而 Q3 只接收 black 的信息，而 Q2 不仅接收 black 的信息，还接收 green 的信息。

Topic Exchange

Topic Exchange 转发消息主要是根据通配符。在这种交换机下，队列和交换机的绑定会定义一种路由模式，那么，通配符就要在这种路由模式和路由键之间匹配后交换机才能转发消息。

在这种交换机模式下：

- 路由键必须是一串字符，用句号（.）隔开，比如 agreements.us，或者 agreements.eu.stockholm 等；
- 路由模式必须包含一个 星号（* ），主要用于匹配路由键指定位置的一个单词，比如，一个路由模式是这样子，agreements..b.*，那么就只能匹配路由键是这样子的，第一个单词是 agreements，第四个单词是 b；井号（#）就表示相当于一个或者多个单词，例如一个匹配模式是 agreements.eu.berlin.#，那么，以 agreements.eu.berlin 开头的路由键都是可以的。

具体代码发送的时候还是一样，第一个参数表示交换机，第二个参数表示 routing key，第三个参数即消息。如下：

```
rabbitTemplate.convertAndSend("testTopicExchange","key1.a.c.key2", " this is  RabbitMQ!");
```

Topic 和 Direct 类似, 只是匹配上支持了“模式”，在“点分”的 routing_key 形式中, 可以使用两个通配符:

- * 表示一个词；
- # 表示零个或多个词。

Headers Exchange

Headers 也是根据规则匹配，相较于 Direct 和 Topic 固定地使用 routing_key，headers 则是一个自定义匹配规则的类型。

在队列与交换器绑定时，会设定一组键值对规则，消息中也包括一组键值对（headers 属性），当这些键值对有一对或全部匹配时，消息被投送到对应队列。

Fanout Exchange

Fanout Exchange 消息广播的模式，不管路由键或者是路由模式，**会把消息发给绑定给它的全部队列**，如果配置了 routing_key 会被忽略。

Spring Boot 集成 RabbitMQ

Spring Boot 集成 RabbitMQ 非常简单，仅需非常少的配置就可使用，Spring Boot 提供了 spring-boot-starter-amqp 组件对 MQ 消息支持。

简单使用

（1）配置 pom 包，主要是添加 spring-boot-starter-amqp 的支持

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

（2）配置文件

配置 rabbitmq 的安装地址、端口以及账户信息：

```
spring.application.name=Spring-boot-rabbitmq

spring.rabbitmq.host=192.168.0.1
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=123456
```

（3）定义队列

```
@Configuration
public class RabbitConfig {

    @Bean
    public Queue Queue() {
        return new Queue("hello");
    }

}
```

（4）发送者

AmqpTemplate 是 Spring Boot 提供的默认实现：

```
public class HelloSender {

    @Autowired
    private AmqpTemplate rabbitTemplate;

    public void send() {
        String context = "hello " + new Date();
        System.out.println("Sender : " + context);
        this.rabbitTemplate.convertAndSend("hello", context);
    }

}
```

（5）接收者

注意使用注解 @RabbitListener，使用 queues 指明队列名称，@RabbitHandler 为具体接收的方法。

```
@Component
@RabbitListener(queues = "hello")
public class HelloReceiver {

    @RabbitHandler
    public void process(String hello) {
        System.out.println("Receiver  : " + hello);
    }

}
```

（6）测试

```
@RunWith(SpringRunner.class)
@Spring BootTest
public class RabbitMqHelloTest {

    @Autowired
    private HelloSender helloSender;

    @Test
    public void hello() throws Exception {
        helloSender.send();
        Thread.sleep(1000l);
    }

}
```

> 注意，发送者和接收者的 queue name 必须一致，不然不能接收。

让测试方法等待一秒，让接收者接收到消息，不然应用退出时可能还没有接收到消息。

以上一个最常用简单的示例就完成了，下面我们介绍更复杂的使用场景。

多方测试

一个发送者和 N 个接收者或者 N 个发送者和 N 个接收者会出现什么情况呢?

一对多发送

对上面的代码进行了小改造，接收端注册了两个 Receiver，Receiver1 和 Receiver2，发送端加入参数计数，接收端打印接收到的参数，下面是测试代码，发送一百条消息，来观察两个接收端的执行效果。

发送者示例：

```
@Component
public class NeoSender {

    @Autowired
    private AmqpTemplate rabbitTemplate;

    public void send(int i) {
        String context = "Spring boot neo queue"+" ****** "+i;
        System.out.println("Sender1 : " + context);
        this.rabbitTemplate.convertAndSend("neo", context);
    }

}
```

接收者 1 示例，接收者 2 和 1基本一致。

```
@Component
@RabbitListener(queues = "neo")
public class NeoReceiver1 {

    @RabbitHandler
    public void process(String neo) {
        System.out.println("Receiver 1: " + neo);
    }

}
```

我们设计了一个发送者，两个接收者，发送一百个消息看看效果。

```
@Test
public void oneToMany() throws Exception {
    for (int i=0;i<100;i++){
        neoSender.send(i);
    }
    Thread.sleep(10000l);
}
```

结果如下：

```
Receiver 2: Spring boot neo queue ****** 1
Receiver 1: Spring boot neo queue ****** 0
Receiver 1: Spring boot neo queue ****** 2
Receiver 2: Spring boot neo queue ****** 3
Receiver 1: Spring boot neo queue ****** 4
Receiver 2: Spring boot neo queue ****** 5
Receiver 2: Spring boot neo queue ****** 7
Receiver 1: Spring boot neo queue ****** 6
Receiver 2: Spring boot neo queue ****** 8
Receiver 1: Spring boot neo queue ****** 9
Receiver 1: Spring boot neo queue ****** 11
Receiver 2: Spring boot neo queue ****** 10
...
```

根据返回结果得到以下结论：

> 一个发送者，N 个接收者，经过测试接收端均匀接收到消息，也说明接收端自动进行了均衡负载，我们也可以利用这个特性做流量分发。

多对多发送

复用以上的发送者和接收者，再增加一个发送者 2 加入标记，在一百个循环中相互交替发送。

发送者方法如下：

```
public void send(int i) {
    String context = "Spring boot neo queue"+" ****** "+i;
    System.out.println("Sender2 : " + context);
    this.rabbitTemplate.convertAndSend("neo", context);
}
```

两个发送者两个接收者的测试用例如下：

```
@Test
public void manyToMany() throws Exception {
    for (int i=0;i<100;i++){
        neoSender.send(i);
        neoSender2.send(i);
    }
    Thread.sleep(10000l);
}
```

结果如下：

```
Sender1 : Spring boot neo queue ****** 0
Sender2 : Spring boot neo queue ****** 0
Sender1 : Spring boot neo queue ****** 1
Sender2 : Spring boot neo queue ****** 1
Sender1 : Spring boot neo queue ****** 2
Sender2 : Spring boot neo queue ****** 2
Sender1 : Spring boot neo queue ****** 3
Sender2 : Spring boot neo queue ****** 3
Sender1 : Spring boot neo queue ****** 4
Sender2 : Spring boot neo queue ****** 4
Sender1 : Spring boot neo queue ****** 5
Sender2 : Spring boot neo queue ****** 5
...

Receiver 2: Spring boot neo queue ****** 0
Receiver 1: Spring boot neo queue ****** 0
Receiver 2: Spring boot neo queue ****** 1
Receiver 1: Spring boot neo queue ****** 1
Receiver 2: Spring boot neo queue ****** 2
Receiver 1: Spring boot neo queue ****** 2
Receiver 2: Spring boot neo queue ****** 3
Receiver 1: Spring boot neo queue ****** 3
Receiver 2: Spring boot neo queue ****** 4
Receiver 1: Spring boot neo queue ****** 4
Receiver 2: Spring boot neo queue ****** 5
Receiver 1: Spring boot neo queue ****** 5
...
```

> 结论：发送端交替发送消息，接收端仍然会均匀接收到消息。

高级使用

对象的支持

Spring Boot 已经完美的支持对象的发送和接收，不需要格外的配置。

```
//发送者
public void send(User user) {
    System.out.println("Sender object: " + user.toString());
    this.rabbitTemplate.convertAndSend("object", user);
}

...

//接收者
@RabbitHandler
public void process(User user) {
    System.out.println("Receiver object : " + user);
}
```

测试用例如下：

```
@Test
public void sendOject() throws Exception {
    User user=new User();
    user.setName("neo");
    user.setPass("123456");
    sender.send(user);
    Thread.sleep(10000l);
}
```

结果如下：

```
Sender object: com.neo.model.User@66971f6b[name=neo,pass=123456]
Receiver object : com.neo.model.User@14e38d67[name=neo,pass=123456]
```

Topic Exchange

Topic 是 RabbitMQ 中最灵活的一种方式，可以根据 routing_key 自由的绑定不同的队列。

首先对 Topic 规则配置，这里使用两个队列来测试：

```
@Configuration
public class TopicRabbitConfig {

    final static String message = "topic.message";
    final static String messages = "topic.messages";

    //定义队列
    @Bean
    public Queue queueMessage() {
        return new Queue(TopicRabbitConfig.message);
    }

    @Bean
    public Queue queueMessages() {
        return new Queue(TopicRabbitConfig.messages);
    }

    //交换机
    @Bean
    TopicExchange exchange() {
        return new TopicExchange("exchange");
    }

    //将队列和交换机绑定
    @Bean
    Binding bindingExchangeMessage(Queue queueMessage, TopicExchange exchange) {
        return BindingBuilder.bind(queueMessage).to(exchange).with("topic.message"
);
    }

    @Bean
    Binding bindingExchangeMessages(Queue queueMessages, TopicExchange exchange) {
        return BindingBuilder.bind(queueMessages).to(exchange).with("topic.#");
    }
}
```

设计 queueMessages 同时匹配两个队列，queueMessage 只匹配“topic.message”队列。

发送者代码如下：

```
public void send1() {
    String context = "hi, i am message 1";
    System.out.println("Sender : " + context);
    this.rabbitTemplate.convertAndSend("exchange", "topic.message", context);
}

public void send2() {
    String context = "hi, i am messages 2";
    System.out.println("Sender : " + context);
    this.rabbitTemplate.convertAndSend("exchange", "topic.messages", context);
}
```

接收者1代码如下：

```
@Component
@RabbitListener(queues = "topic.message")
public class TopicReceiver {

    @RabbitHandler
    public void process(String message) {
        System.out.println("Topic Receiver1  : " + message);
    }

}
```

接收者 2 和 1 类似。

发送 send1 会匹配到 topic.# 和 topic.message 两个 Receiver 都可以收到消息；发送 send2 只有 topic.# 可以匹配，Receiver2 监听到了消息。

首先测试发送 send1()：

```
@Test
public void topic1() throws Exception {
    sender.send1();
    Thread.sleep(10000);
}
```

返回结果如下：

```
Sender : hi, i am message 1
Topic Receiver1  : hi, i am message 1
Topic Receiver2  : hi, i am message 1
```

说明两个接收者都接收到了消息。

再测试发送send2()：

```
@Test
public void topic2() throws Exception {
    sender.send2();
    Thread.sleep(10001);
}
```

返回结果如下：

```
Sender : hi, i am messages 2
Topic Receiver2  : hi, i am messages 2
```

只有接收者2接收到了消息。两个方法都满足我们的预期，验证了我们的设想。

Fanout Exchange

Fanout 就是我们熟悉的广播模式或者订阅模式，给 Fanout 交换机发送消息，绑定了这个交换机的所有队列都收到这个消息。

Fanout 相关配置：

```
@Configuration
public class FanoutRabbitConfig {

    //定义队列
    @Bean
    public Queue AMessage() {
        return new Queue("fanout.A");
    }

    @Bean
    public Queue BMessage() {
        return new Queue("fanout.B");
    }

    @Bean
    public Queue CMessage() {
        return new Queue("fanout.C");
    }

    //定义交换机
    @Bean
    FanoutExchange fanoutExchange() {
        return new FanoutExchange("fanoutExchange");
    }

    //分部进行绑定
    @Bean
    Binding bindingExchangeA(Queue AMessage,FanoutExchange fanoutExchange) {
        return BindingBuilder.bind(AMessage).to(fanoutExchange);
    }

    @Bean
    Binding bindingExchangeB(Queue BMessage, FanoutExchange fanoutExchange) {
        return BindingBuilder.bind(BMessage).to(fanoutExchange);
    }

    @Bean
    Binding bindingExchangeC(Queue CMessage, FanoutExchange fanoutExchange) {
        return BindingBuilder.bind(CMessage).to(fanoutExchange);
    }

}
```

这里使用了 A、B、C 三个队列绑定到 Fanout 交换机上面，发送端的 routing_key 写任何字符都会被忽略：

发生者如下：

```
public void send() {
    String context = "hi, fanout msg ";
    System.out.println("Sender : " + context);
    this.rabbitTemplate.convertAndSend("fanoutExchange","", context);
}
```

接收者 A 代码如下：

```
@Component
@RabbitListener(queues = "fanout.A")
public class FanoutReceiverA {

    @RabbitHandler
    public void process(String message) {
        System.out.println("fanout Receiver A: " + message);
    }

}
```

接收者 B、C 和接收者 A 类似，相见示例代码。

写测试用例调用 send() 进行测试：

```
@Test
public void fanoutSender() throws Exception {
    sender.send();
    Thread.sleep(10001);
}
```

结果如下：

```
Sender : hi, fanout msg
fanout Receiver B: hi, fanout msg
fanout Receiver C: hi, fanout msg
fanout Receiver A: hi, fanout msg
```

结果说明，绑定到 fanout 交换机上面的队列都收到了消息。

总结

RabbitMQ 一个非常高效的消息队列组件，使用 RabbitMQ 可以方便的解耦项目之间的依赖，同时利用 RabbitMQ 的特性可以做很多的解决方案。Spring Boot 为 RabbitMQ 提供了支持组件 spring-boot-starter-amqp，加载的时候会自动进行配置，并且预置了 RabbitTemplate，可以让我们在项目中方便的调用。在测试使用的过程中发现，RabbitMQ 非常的灵活，可以使用各种策略将不同的发送者和接收者绑定在一起，这些特性在实际项目使用中非常的高效便利。

> 点击这里下载源码。