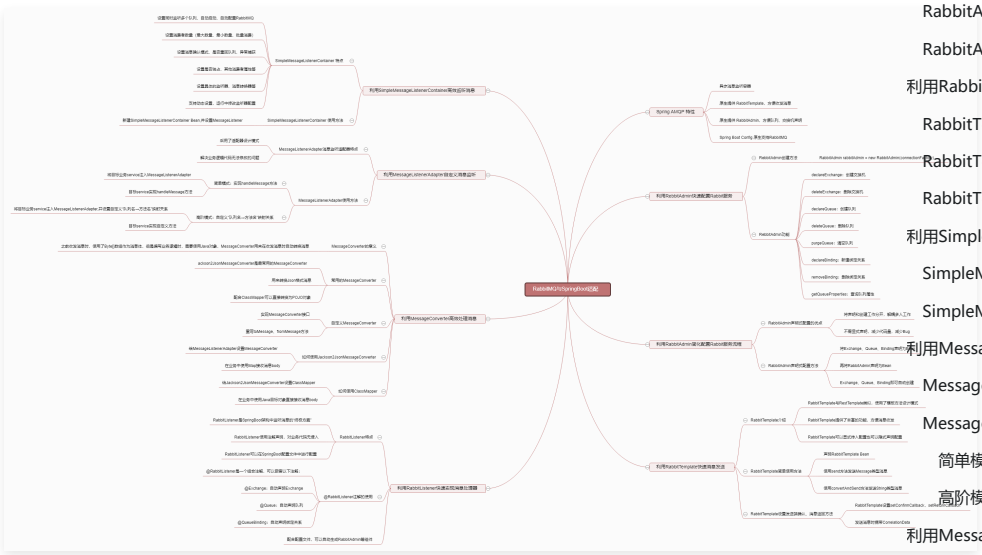
Moody • 更新于 2020-10-10

上一节 第四章RabbitM… 第六章RabbitM… 下一节



引入SpringBoot适配的重要性

Spring AMQP 特性

- 异步消息监听容器
- 原生提供 RabbitTemplate，方便收发消息
- 原生提供 RabbitAdmin，方便队列、交换机声明
- Spring Boot Config 原生支持RabbitMQ

利用RebbitAdmin快速配置Rabbit服务

RebbitAdmin创建方法

代码块

```
1    ConnectionFactory connectionFactory = new CachingConnectionFactory();
2    RabbitAdmin rabbitAdmin = new RabbitAdmin(connectionFactory);
```

RebbitAdmin功能

- declareExchange：创建交换机
- deleteExchange：删除交换机
- declareQueue：创建队列
- deleteQueue：删除队列
- purgeQueue：清空队列
- declareBinding：新建绑定关系
- removeBinding：删除绑定关系

代码块

```
connectionFactory.setHost("127.0.0.1");
connectionFactory.setPort(5672);
connectionFactory.setPassword("guest");
connectionFactory.setUsername("guest");

RabbitAdmin rabbitAdmin = new RabbitAdmin(connectionFactory);

/*--------------------restaurant--------------------*/
Exchange exchange = new DirectExchange("exchange.order.restaurant");
rabbitAdmin.declareExchange(exchange);

Queue queue = new Queue("queue.order");
rabbitAdmin.declareQueue(queue);

Binding binding = new Binding(
        "queue.order",
        Binding.DestinationType.QUEUE,
        "exchange.order.restaurant",
        "key.order",
        null);

rabbitAdmin.declareBinding(binding);

/*--------------------deliveryman--------------------*/
exchange = new DirectExchange("exchange.order.deliveryman");
rabbitAdmin.declareExchange(exchange);
binding = new Binding(
        "queue.order",
        Binding.DestinationType.QUEUE,
        "exchange.order.deliveryman",
        "key.order",
        null);
rabbitAdmin.declareBinding(binding);


/*---------settlement---------*/
exchange = new FanoutExchange("exchange.order.settlement");
rabbitAdmin.declareExchange(exchange);
exchange = new FanoutExchange("exchange.settlement.order");
rabbitAdmin.declareExchange(exchange);
binding = new Binding(
        "queue.order",
        Binding.DestinationType.QUEUE,
        "exchange.order.settlement",
        "key.order",
        null);
rabbitAdmin.declareBinding(binding);


/*--------------reward----------------*/
exchange = new TopicExchange("exchange.order.reward");
rabbitAdmin.declareExchange(exchange);
binding = new Binding(
        "queue.order",
        Binding.DestinationType.QUEUE,
        "exchange.order.reward",
        "key.order",
        null);
rabbitAdmin.declareBinding(binding);
```

利用RabbitAdmin简化配置Rabbit服务流程

RabbitAdmin声明式配置的优点

- 将声明和创建工作分开，解耦多人工作
- 不需显式声明，减少代码量，减少Bug

RabbitAdmin声明式配置方法

将Exchange、Queue、Binding声明为Bean

代码块

```
/*----------------------restaurant----------------------*/
@Bean
public Exchange exchange1(){
    return new DirectExchange("exchange.order.restaurant");
}
@Bean
public Queue queue1(){
    return new Queue("queue.order");
}
@Bean
public Binding binding1(){
    return new Binding(
            "queue.order",
            Binding.DestinationType.QUEUE,
            "exchange.order.deliveryman",
            "key.order",
            null);
}

/*----------------------deliveryman----------------------*/
@Bean
public Exchange exchange2(){
    return new DirectExchange("exchange.order.deliveryman");
}
@Bean
public Binding binding2(){
    return new Binding(
            "queue.order",
            Binding.DestinationType.QUEUE,
            "exchange.order.deliveryman",
            "key.order",
            null);
}

/*---------settlement---------*/
@Bean
public Exchange exchange3(){
    return new FanoutExchange("exchange.order.settlement");
}
@Bean
public Exchange exchange4(){
    return new FanoutExchange("exchange.settlement.order");
}
@Bean
public Binding binding3(){
    return new Binding(
            "queue.order",
            Binding.DestinationType.QUEUE,
            "exchange.settlement.order",
            "key.order"
```

```
52  }
53
54  /*--------------reward----------------*/
55  @Bean
56  public Exchange exchange5(){
57      return new TopicExchange("exchange.order.reward");
58  }
59  @Bean
60  public Binding binding4(){
61      return new Binding(
62              "queue.order",
63              Binding.DestinationType.QUEUE,
64              "exchange.order.reward",
65              "key.order",
66              null);
67  }
```

再将RabbitAdmin声明为Bean

代码块

```
1   @Bean
2   public ConnectionFactory connectionFactory(){
3       CachingConnectionFactory connectionFactory = new CachingConnectionFactory();
4       connectionFactory.setHost("127.0.0.1");
5       connectionFactory.setPort(5672);
6       connectionFactory.setPassword("guest");
7       connectionFactory.setUsername("guest");
8       connectionFactory.createConnection();
9       return connectionFactory;
10  }
11
12  @Bean
13  public RabbitAdmin rabbitAdmin(ConnectionFactory connectionFactory){
14      RabbitAdmin rabbitAdmin = new RabbitAdmin(connectionFactory);
15      rabbitAdmin.setAutoStartup(true);
16      return rabbitAdmin;
17  }
```

Exchange、Queue、Binding即可自动创建

利用RabbitTemplate快速消息发送

RabbitTemplate介绍

RabbitTemplate与RestTemplate类似，使用了模板方法设计模式
RabbitTemplate提供了丰富的功能，方便消息收发
RabbitTemplate可以显式传入配置也可以隐式声明配置

RabbitTemplate简单使用方法

声明RabbitTemplate Bean

代码块

```
1   @Bean
2   RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
3       RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
4       return rabbitTemplate;
5   }
```

代码块

```
rabbitTemplate.send(
        "exchange.order.restaurant",
        "key.restaurant",
        message,correlationData
);
```

使用convertAndSend方法发送String类型消息

代码块

```
rabbitTemplate.send(
        "exchange.order.restaurant",
        "key.restaurant",
        message
);
```

RabbitTemplate设置发送端确认、消息返回方法

RabbitTemplate设置setConfirmCallback，setReturnCallback

代码块

```
@Bean
RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
    RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
    rabbitTemplate.setMandatory(true);

    rabbitTemplate.setConfirmCallback((correlationData, ack, cause) ->
            log.info("correlationData:{}, ack:{}, cause{}",
                    correlationData,
                    ack,
                    cause));
    rabbitTemplate.setReturnCallback((message, replyCode, replyText, exchange, routingKe
            log.info(
                    "message:{}, replyCode:{}, replyText:{}, exchange:{}, routingKey{}",
                    message,
                    replyCode,
                    replyText,
                    exchange,
                    routingKey));
    return rabbitTemplate;
}
```

发送消息时携带CorrelationData

代码块

```
CorrelationData correlationData = new CorrelationData();
correlationData.setId(orderPO.getId().toString());
rabbitTemplate.send(
        "exchange.order.restaurant",
        "key.restaurant",
        message,correlationData
);
```

利用SimpleMessageListenerContainer高效监听消息

SimpleMessageListenerContainer 特点

设置同时监听多个队列、自动启动、自动配置RabbitMQ
设置消费者数量（最大数量、最小数量、批量消费）
设置消息确认模式、是否重回队列、异常捕获
设置是否独占、其他消费者属性等
设置具体的监听器、消息转换器等
支持动态设置，运行中修改监听器配置

SimpleMessageListenerContainer 使用方法

新建SimpleMessageListenerContainer Bean,并设置MessageListener

代码块

```
@Bean
public SimpleMessageListenerContainer messageListenerContainer(ConnectionFactory connect
    SimpleMessageListenerContainer messageListenerContainer = new SimpleMessageListenerC
    messageListenerContainer.setQueueNames("queue.order");
    messageListenerContainer.setConcurrentConsumers(1);
    messageListenerContainer.setMaxConcurrentConsumers(3);
    messageListenerContainer.setAcknowledgeMode(AcknowledgeMode.AUTO);
    messageListenerContainer.setMessageListener(new MessageListener() {
        @Override
        public void onMessage(Message message) {
            log.info("message:{}", message);
        }
    });
    messageListenerContainer.setPrefetchCount(2);
    messageListenerContainer.setAcknowledgeMode(AcknowledgeMode.MANUAL);
    messageListenerContainer.setMessageListener(new ChannelAwareMessageListener() {
        @Override
        public void onMessage(Message message, Channel channel) throws Exception {
            channel.basicAck(message.getMessageProperties().getDeliveryTag(),false);
        }
    });
    return messageListenerContainer;
}
```

利用MessageListenerAdapter自定义消息监听

MessageListenerAdapter消息监听适配器特点

采用了适配器设计模式
解决业务逻辑代码无法修改的问题

MessageListenerAdapter使用方法

简单模式：实现handleMessage方法

将目标业务service注入MessageListenerAdapter

代码块

```
@Bean
public SimpleMessageListenerContainer messageListenerContainer(
        @Autowired ConnectionFactory connectionFactory
) {
    SimpleMessageListenerContainer messageListenerContainer =
            new SimpleMessageListenerContainer(connectionFactory);
    messageListenerContainer.setQueueNames("queue.order");
```

```
9       messageListenerContainer.setMaxConcurrentConsumers(5);
10      messageListenerContainer.setAcknowledgeMode(AcknowledgeMode.AUTO);
11
12      MessageListenerAdapter listenerAdapter = new MessageListenerAdapter();
13      listenerAdapter.setDelegate(orderMessageService);
14
15      Map<String, String> methodMap = new HashMap<>(8);
16      messageListenerContainer.setMessageListener(listenerAdapter);
17      return messageListenerContainer;
18  }
```

目标service实现handleMessage方法

代码块

```
1   @Service
2   public class OrderMessageService {
3
4
5       public void handleMessage1(byte[] messageBody) throws IOException {
6           //业务逻辑
7
8       }
```

高阶模式：自定义“队列名→方法名”映射关系

将目标业务service注入MessageListenerAdapter,并设置自定义“队列名→方法名”映射关系

代码块

```
1   @Bean
2   public SimpleMessageListenerContainer messageListenerContainer(
3           @Autowired ConnectionFactory connectionFactory
4   ) {
5       SimpleMessageListenerContainer messageListenerContainer =
6               new SimpleMessageListenerContainer(connectionFactory);
7       messageListenerContainer.setQueueNames("queue.order");
8       messageListenerContainer.setConcurrentConsumers(3);
9       messageListenerContainer.setMaxConcurrentConsumers(5);
10      messageListenerContainer.setAcknowledgeMode(AcknowledgeMode.AUTO);
11
12      MessageListenerAdapter listenerAdapter = new MessageListenerAdapter();
13      listenerAdapter.setDelegate(orderMessageService);
14
15      Map<String, String> methodMap = new HashMap<>(8);
16      methodMap.put("queue.order", "handleMessage1");
17      listenerAdapter.setQueueOrTagToMethodName(methodMap);
18      messageListenerContainer.setMessageListener(listenerAdapter);
19      return messageListenerContainer;
20  }
```

目标service实现自定义方法

代码块

```
1   @Service
2   public class OrderMessageService {
3
4
5       public void handleMessage(byte[] messageBody) throws IOException {
6           //业务逻辑
7
8       }
```

利用MessageConverter高效处理消息

MessageConverter的意义

之前收发消息时，使用了Byte[]数组作为消息体，但是编写业务逻辑时，需要使用Java对象，MessageConverter用来在收发消息时自动转换消息

常用的MessageConverter

Jackson2JsonMessageConverter是最常用的MessageConverter
用来转换Json格式消息
配合ClassMapper可以直接转换为POJO对象

自定义MessageConverter

实现MessageConverter接口
重写toMessage、fromMessage方法

如何使用Jackson2JsonMessageConverter

给MessageListenerAdapter设置MessageConverter

代码块

```
@Bean
public SimpleMessageListenerContainer messageListenerContainer(
        @Autowired ConnectionFactory connectionFactory
) {
    SimpleMessageListenerContainer messageListenerContainer =
            new SimpleMessageListenerContainer(connectionFactory);
    messageListenerContainer.setQueueNames("queue.order");
    messageListenerContainer.setConcurrentConsumers(3);
    messageListenerContainer.setMaxConcurrentConsumers(5);
    messageListenerContainer.setAcknowledgeMode(AcknowledgeMode.AUTO);
    messageListenerContainer.setPrefetchCount(1);

    MessageListenerAdapter messageListenerAdapter = new MessageListenerAdapter(orderMess
    Jackson2JsonMessageConverter messageConverter = new Jackson2JsonMessageConverter();
    messageListenerAdapter.setMessageConverter(messageConverter);
    messageListenerContainer.setMessageListener(messageListenerAdapter);
    return messageListenerContainer;
}
```

在业务中使用Map接收消息body

代码块

```
public class OrderMessageService {
    public void handleMessage(Map<String, Object> orderMessageBody){
        //业务逻辑
    }
}
```

如何使用ClassMapper

给Jackson2JsonMessageConverter设置ClassMapper

代码块

```
@Bean
public SimpleMessageListenerContainer messageListenerContainer(
        @Autowired ConnectionFactory connectionFactory
) {
    SimpleMessageListenerContainer messageListenerContainer =
            new SimpleMessageListenerContainer(connectionFactory);
    messageListenerContainer.setQueueNames("queue.order");
    messageListenerContainer.setConcurrentConsumers(3);
    messageListenerContainer.setMaxConcurrentConsumers(5);
    messageListenerContainer.setAcknowledgeMode(AcknowledgeMode.AUTO);

    messageListenerContainer.setPrefetchCount(1);

    MessageListenerAdapter messageListenerAdapter = new MessageListenerAdapter(orderMessage
    Jackson2JsonMessageConverter messageConverter = new Jackson2JsonMessageConverter();
    messageConverter.setClassMapper(new ClassMapper() {
        @Override
        public void fromClass(Class<?> clazz, MessageProperties properties) {

        }

        @Override
        public Class<?> toClass(MessageProperties properties) {
            return OrderMessageDTO.class;
        }
    });
    messageListenerAdapter.setMessageConverter(messageConverter);
    messageListenerContainer.setMessageListener(messageListenerAdapter);
    return messageListenerContainer;
}
```

在业务中使用Java目标对象直接接收消息body

代码块

```
@Service
public class OrderMessageService {
    public void handleMessage(OrderMessageDTO orderMessageDTO) {
         //业务逻辑
    }
 }
```

利用RabbitListener快速实现消息处理器

RabbitListener特点

RabbitListener是SpringBoot架构中监听消息的“终极方案”
RabbitListener使用注解声明，对业务代码无侵入
RabbitListener可以在SpringBoot配置文件中进行配置

@RabbitListener注解的使用

@RabbitListener是一个组合注解，可以嵌套以下注解：
@Exchange：自动声明Exchange
@Queue：自动声明队列
@QueueBinding：自动声明绑定关系

代码块

```
    @RabbitListener(
//            containerFactory = "rabbitListenerContainerFactory",
//            queues = "queue.order",
//            admin = "rabbitAdmin",
            bindings = {
                    @QueueBinding(
                            value = @Queue(name = "${imooc.order-queue}",
                                    arguments = {
                                            //  @Argument(name =
                                            //  "x-message-ttl", value =
                                            //  "1000", type = "java.lang
                                            //  .Integer"),
                                            //  @Argument(name =
                                            //  "x-dead-letter-exchange",
                                            //  value = "aaaaa"),
                                            //  @Argument(name =
                                            //  "x-dead-letter-routing-key", value = "bb
                                    }),
                            exchange = @Exchange(name = "exchange.order.restaurant", typ
                            key = "key.order"
                    ),
                    @QueueBinding(
                            value = @Queue(name = "queue.order"),
                            exchange = @Exchange(name = "exchange.order.deliveryman", ty
                            key = "key.order"
                    ),
                    @QueueBinding(
                            value = @Queue(name = "queue.order"),
                            exchange = @Exchange(name = "exchange.settlement.order", typ
                            key = "key.order"
                    ),
                    @QueueBinding(
                            value = @Queue(name = "queue.order"),
                            exchange = @Exchange(name = "exchange.order.reward", type =
                            key = "key.order"
                    )
            }
    )
    public void handleMessage(@Payload Message message) throws IOException {
        //业务逻辑
    }

```

配合配置文件，可以自动生成RabbitAdmin等组件

代码块

```
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
spring.rabbitmq.addresses=127.0.0.1
spring.rabbitmq.port=6379
```

第四章RabbitMQ高级特性总结 上一节 下一节 第六章RabbitMQ集群入门

我要提出意见反馈