第一章　Springboot 快速集成 RocketMQ

资料来源：

rocketmq-spring 的 Github 地址

CSDN 上的一篇优质介绍文章

1.1　添加 Maven 依赖

作者使用的版本是 2.2.2，也可以直接看rocketmq-spring-boot-starter 的 Maven 仓库来获得最新版本。

Listing 1.1: Maven 依赖

```
<dependency>
  <groupId>org.apache.rocketmq</groupId>
  <artifactId>rocketmq-spring-boot-starter</artifactId>
  <version>2.2.2</version>
</dependency>
```

1.2　添加配置文件属性

Listing 1.2: 配置文件 application.yml

```
rocketmq:
  name-server: 192.168.146.132:9876 # 名称服务访问地址
  producer:
    group: TEST_GROUP # 必须指定group
    send-message-timeout: 3000 # 消息发送超时时长，默认3s
```

```
6     retry-times-when-send-failed: 3 # 同步发送消息失败重试次数，默认2
7     retry-times-when-send-async-failed: 3 # 异步发送消息失败重试次数，默认2
```

1.3 生产者

1.3.1 普通消息

需要先注入 RocketMQTemplate 的 Bean，用于进行消息的发送; topic 用于指定发送消息到的主题，sendMessage 为发送的具体消息; 可使用 topic:tag 的格式附带消息的 Tag; 可使用 setHeader 方法来设置消息的 key。

同步发送

阻塞当前线程，等待 broker 响应发送结果。

Listing 1.3: 普通消息同步发送方式

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
3 public SendResult syncSend(Object sendMessage,String topic,String tag,String
  key) {
4   return rocketMQTemplate.syncSend(topic+":"+tag, MessageBuilder.withPayload(
    sendMessage).setHeader(RocketMQHeaders.KEYS, key).build());
5 }
```

Listing 1.4: 同步发送方式简化版

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
3 public SendResult syncSend(Object sendMessage,String topic) {
4   return rocketMQTemplate.syncSend(topic, MessageBuilder.withPayload(
    sendMessage).build());
5 }
```

异步发送

通过线程池执行发送到 broker 的消息任务，执行完后回调：在 SendCallback 中可处理相关成功失败时的逻辑。

Listing 1.5: 普通消息异步发送方式

```
@Resource
private RocketMQTemplate rocketMQTemplate;
public void asyncSend(Object sendMessage, String topic, String tag, String key) {
  rocketMQTemplate.asyncSend(topic + ":" + tag, MessageBuilder.withPayload(sendMessage).setHeader(RocketMQHeaders.KEYS, key).build(), new SendCallback() {
    @Override
    public void onSuccess(SendResult sendResult) {
      //发送异步消息成功后的处理...
    }

    @Override
    public void onException(Throwable throwable) {
      //发送异步消息失败后的处理...
    }
  });
}
```

单向发送

只负责发送消息，不等待应答，不关心发送结果。

Listing 1.6: 普通消息单向发送方式

```
@Resource
private RocketMQTemplate rocketMQTemplate;
```

```
3 public void oneWaySend(Object sendMessage,String topic,String tag,String key)
   {
4   rocketMQTemplate.sendOneWay(topic+":"+tag, MessageBuilder.withPayload(
    sendMessage).setHeader(RocketMQHeaders.KEYS, key).build());
5 }
```

1.3.2 顺序消息

topic 必须为顺序类型的主题，不允许将消息放到不同类型的主题。hashkey 用于确定发送到同一个主题中的哪个队列。要求顺序消费的多个消息必须使用同一个 hashkey 以保证进入同一个队列。

同步发送

Listing 1.7: 顺序消息同步发送方式

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
3 public SendResult syncOrderlySend(Object sendMessage,String topic,String tag,
  String key,String hashkey) {
4   return rocketMQTemplate.syncSendOrderly(topic+":"+tag,MessageBuilder.
    withPayload(sendMessage).setHeader(RocketMQHeaders.KEYS, key).build(),
    hashkey);
5 }
```

异步发送

Listing 1.8: 顺序消息异步发送方式

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
3 public void asyncOrderlySend(Object sendMessage,String topic,String tag,
  String key,String hashkey) {
```

```
4   rocketMQTemplate.asyncSendOrderly(topic+":"+tag,MessageBuilder.withPayload(
    sendMessage).setHeader(RocketMQHeaders.KEYS, key).build(),hashkey,new
    SendCallback() {
5     @Override
6     public void onSuccess(SendResult sendResult) {
7       System.out.println("发送异步消息成功" + JSON.toJSONString(sendResult));
8       //发送异步消息成功后的处理...
9     }
10
11    @Override
12    public void onException(Throwable throwable) {
13      System.out.println("发送异步消息失败" + JSON.toJSONString(throwable));
14      //发送异步消息失败后的处理...
15    }
16  });
17 }
```

1.3.3 延迟消息

延迟消息就是普通消息发送的后面加上一个 timeout 属性和 delayLevel 属性。timeout 是消息发送超时时长，为默认 3s。delayLevel 属性分为 18 个等级，分别为：1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h。

同步发送

Listing 1.9: 延迟消息同步发送方式

```
1 @Resource
2 private RocketMQTemplate rocketMQTemplate;
3 public SendResult syncDelaySend(Object sendMessage,String topic,String tag,
  String key,long timeout, int delayLevel) {
4   return rocketMQTemplate.syncSend(topic+":"+tag,MessageBuilder.withPayload(
    sendMessage).setHeader(RocketMQHeaders.KEYS, key).build(),timeout,delayLevel
```

```
  );
}
```

异步发送

Listing 1.10: 延迟消息异步发送方式

```
@Resource
private RocketMQTemplate rocketMQTemplate;
public void asyncDelaySend(Object sendMessage, String topic, String tag,
String key, long timeout, int delayLevel) {
  rocketMQTemplate.asyncSend(topic + ":" + tag, MessageBuilder.withPayload(
  sendMessage).setHeader(RocketMQHeaders.KEYS, key).build(), new SendCallback
  () {
    @Override
    public void onSuccess(SendResult sendResult) {
      System.out.println("发送异步消息成功" + JSON.toJSONString(sendResult));
      //发送异步消息成功后的处理...
    }

    @Override
    public void onException(Throwable throwable) {
      System.out.println("发送异步消息失败" + JSON.toJSONString(throwable));
      //发送异步消息失败后的处理...
    }
  }, timeout, delayLevel);
}
```

1.3.4 批量消息

批量消息的发送方式即将原先普通消息中的 sendMessage 换成 Collection<Message> 的对象即可。

同步发送

Listing 1.11: 批量消息同步发送方式（简化版）

```
@Resource
private RocketMQTemplate rocketMQTemplate;
public void syncBatchSend(List<Object> sendMessage, String topic) {
  ArrayList<Message<Object>> list = new ArrayList<>();
  for (Object message : sendMessage) {
    list.add(MessageBuilder.withPayload(message).build());
  }
  rocketMQTemplate.syncSend(topic, list);
}
```

复杂版本和异步版本省略。

1.3.5　事务消息

```
@SpringBootApplication
public class ProducerApplication implements CommandLineRunner{
    @Resource
    private RocketMQTemplate rocketMQTemplate;

    public static void main(String[] args){
        SpringApplication.run(ProducerApplication.class, args);
    }

    public void run(String... args) throws Exception {
        try {
            // Build a SpringMessage for sending in transaction
            Message msg = MessageBuilder.withPayload(..)...;
            // In sendMessageInTransaction(), the first parameter transaction
            name ("test")
```

```
15            // must be same with the @RocketMQTransactionListener's member
              field 'transName'
16            rocketMQTemplate.sendMessageInTransaction("test-topic", msg, null);
17        } catch (MQClientException e) {
18            e.printStackTrace(System.out);
19        }
20    }
21
22    // Define transaction listener with the annotation
      @RocketMQTransactionListener
23    @RocketMQTransactionListener
24    class TransactionListenerImpl implements RocketMQLocalTransactionListener
      {
25        @Override
26        public RocketMQLocalTransactionState executeLocalTransaction(Message
          msg, Object arg) {
27          // ... local transaction process, return bollback, commit or
            unknown
28          return RocketMQLocalTransactionState.UNKNOWN;
29        }
30
31        @Override
32        public RocketMQLocalTransactionState checkLocalTransaction(Message
          msg) {
33          // ... check transaction status and return bollback, commit or
            unknown
34          return RocketMQLocalTransactionState.COMMIT;
35        }
36    }
37}
```

1.4 消费者

在 RocketMQMessageListener 注解中，topic 用于指定接收的主题，selectorType 用于指定过滤的方式（默认为 Tag，可更改为 SQL92），selectorExpression 用于控制消息的过滤规则（Tag 模式下，* 代表全部 Tag），consumeMode 用于指定是即时接收消息还是顺序接收消息。（其他注解属性请自行了解）

1.4.1 Push 模式

消费消息仅通过消费监听器处理业务并返回消费结果。消息的获取、消费状态提交以及消费重试都通过 Apache RocketMQ 的客户端 SDK 完成。

Listing 1.12: Push 模式消费者

```
@Service
@RocketMQMessageListener(topic = "TEST_TOPIC", selectorExpression = "test",
consumerGroup = "Group_One")
public class ConsumerSend implements RocketMQListener<User> {
  // 监听到消息就会执行此方法
  @Override
  public void onMessage(Object message) {
    //此处为处理消息的方法
  }
}
```

1.4.2 Pull 模式

如果使用 Pull 模型，需要补充配置文件属性。（注意：如果不使用，就不要添加该属性。）

Listing 1.13: lite pull consumer 所需配置属性

```
rocketmq.pull-consumer.group=Group_One
rocketmq.pull-consumer.topic=TEST_TOPIC
```

Pull 模型通过调用 RocketMQTemplate 的 receive 方法实现。

Listing 1.14: Pull 模式消费者

```
1 public void Pull() throws Exception {
2   List<Object> messages = rocketMQTemplate.receive(Object.class);
3   //此处处理接收到的message集合
4 }
```