阿里云开发者社区 ALIBABA CLOUD DEVELOPER COMMUNITY | 阿里巴巴云原生 | RocketMQ

在Spring生态中玩转RocketMQ

动手实战！一次吃透三大 RocketMQ 接入方法

在Spring生态中使用RocketMQ到底有多少种方式？
他们各自适用于什么场景？各自有什么优劣势？
如何开始实战？

藏经阁

阿里云开发者电子书系列

RocketMQ 中国社区钉钉群

欢迎各位开发者进群交流、勘误

微信扫码关注

“阿里巴巴云原生公众号”

阿里云开发者“藏经阁”

海量电子书免费下载

目录

开篇：在 Spring 生态中玩转 RocketMQ

Apache RocketMQ 作为阿里开源的业务消息的首选，通过双 11 业务打磨，在消息和流处理领域被广泛应用。而微服务生态 Spring 框架也是业务开发中最受开发者欢迎的框架之一，两者的完美契合使得 RocketMQ 成为 Spring Messaing 实现中最受欢迎的消息实现。

在 Spring 生态中使用 RocketMQ 到底有多少种方式？他们各自适用于什么场景？各自有什么优劣势？

如何开始实战？本书将一一解答。

我们先会带领各位开发者：

- 回顾罗美琪（RocketMQ）和春波特（SpringBoot）故事开始的时候，rocketmq-spring 经过 6 个多月的孵化，作为 Apache RocketMQ 的子项目正式毕业。
- 回顾 rocketmq-spring 毕业后的两年，是如何成为 Spring 生态中最受欢迎的 messaging 实现的？

最后将通过图文和实操地方式带来给位开发者玩转在 Spring 生态中使用 RocketMQ 的三种主流方式。

所有动手实操的部分大家可以登录 start.aliyun.com 知行动手实验室免费体验。

开卷有益，希望各位开发者通过阅读本书、动手实操有所收获。

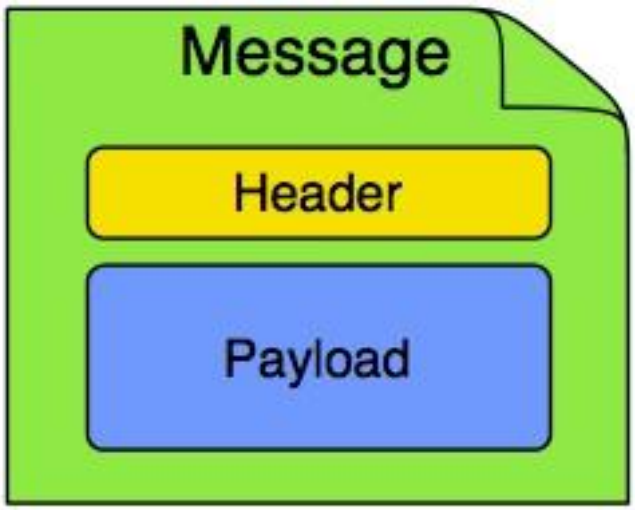
一、RocketMQ 与 Spring 的碰撞

在介绍 RocketMQ 与 Spring 故事之前，不得不提到 Spring 中的两个关于消息的框架，Spring Messaging 和 Spring Cloud Stream。它们都能够与 Spring Boot 整合并提供了一些参考的实现。和所有的实现框架一样，消息框架的目的是实现轻量级的消息驱动的微服务，可以有效地简化开发人员对消息中间件的使用复杂度，让系统开发人员可以有更多的精力关注于核心业务逻辑的处理。

二、Spring Messaging

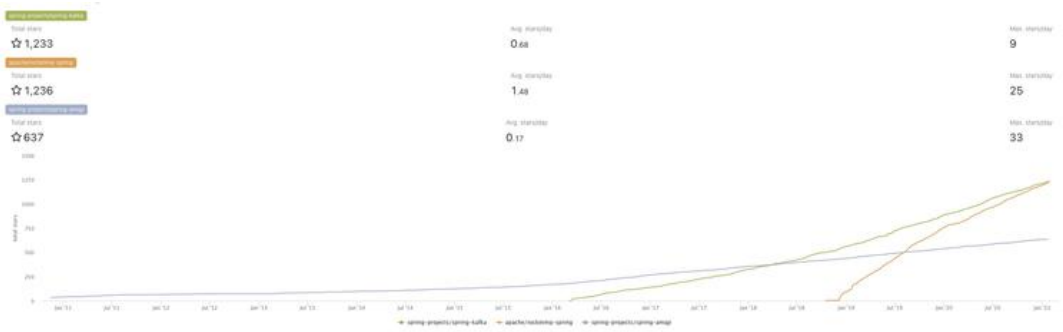
Spring Messaging 是 Spring Framework 4 中添加的模块，是 Spring 与消息系统集成的一个扩展性的支持。它实现了从基于 JmsTemplate 的简单的使用 JMS 接口到异步接收消息的一整套完整的基础架构，Spring AMQP 提供了该协议所要求的类似的功能集。 在与 Spring Boot 的集成后，它拥有了自动配置能力，能够在测试和运行时与相应的消息传递系统进行集成。

单纯对于客户端而言，Spring Messaging 提供了一套抽象的 API 或者说是约定的标准，对消息发送端和消息接收端的模式进行规定，比如消息 Messaging 对应的模型就包括一个消息体 Payload 和消息头 Header。不同的消息中间件提供商可以在这个模式下提供自己的 Spring 实现：在消息发送端需要实现的是一个 XXXTemplate 形式的 Java Bean，结合 Spring Boot 的自动化配置选项提供多个不同的发送消息方法；在消息的消费端是一个 XXXMessageListener 接口（实现方式通常会使用一个注解来声明一个消息驱动的 POJO），提供回调方法来监听和消费消息，这个接口同样可以使用 Spring Boot 的自动化选项和一些定制化的属性。



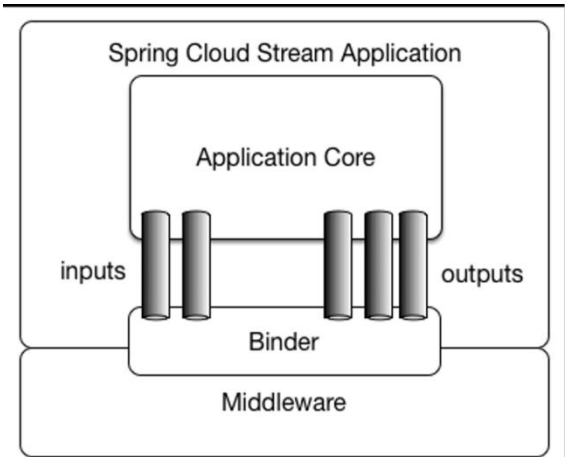
在 Apache RocketMQ 生态中，RocketMQ-Spring-Boot-Starter（下文简称 RocketMQ-Spring）就是一个支持 Spring Messaging API 标准的项目。该项目把 RocketMQ 的客户端使用 Spring Boot 的方式进行了封装，可以让用户通过简单的 annotation 和标准的 Spring Messaging API 编写代码来进行消息的发送和消费，也支持扩展出 RocketMQ 原生 API 来支持更加丰富的消息类型。在 RocketMQ-Spring 毕业初期，RocketMQ 社区同学请 Spring 社区的同学对 RocketMQ-Spring 代码进行 review，

引出一段罗美琪（RocketMQ）和春波特（Spring Boot）故事的佳话[1]，著名 Spring 布道师 Josh Long 向国外同学介绍如何使用 RocketMQ-Spring 收发消息[2]。RocketMQ-Spring 也在短短两年时间超越 Spring-Kafka 和 Spring-AMQP（注:两者均由 Spring 社区维护），成为 Spring Messaging 生态中最活跃的消息项目。



三、Spring Cloud Stream

Spring Cloud Stream 结合了 Spring Integration 的注解和功能，它的应用模型如下：



Spring Cloud Stream 框架中提供一个独立的应用内核，它通过输入(@Input)和输出(@Output)通道与外部世界进行通信，消息源端(Source)通过输入通道发送消息，消费目标端(Sink)通过监听输出通道来获取消费的消息。这些通道通过专用的 Binder 实现与外部代理连接。开发人员的代码只需要针对应用内核提供的固定的接口和注解方式进行编程，而不需要关心运行时具体的 Binder 绑定的消息中间件。

在运行时，Spring Cloud Stream 能够自动探测并使用在 classpath 下找到的 Binder。这样开发人员可以轻松地在相同的代码中使用不同类型的中间件：仅仅需要在构建时包含进不同的 Binder。在更加复杂的使用场景中，也可以在应用中打包多个 Binder 并让它自己选择 Binder，甚至在运行时为不同的通道使用不同的 Binder。

Binder 抽象使得 Spring Cloud Stream 应用可以灵活的连接到中间件，加之 Spring Cloud Stream 使用利用了 Spring Boot 的灵活配置配置能力，这样的配置可以通过外部配置的属性和 Spring Boot 支持的任何形式来提供（包括应用启动参数、环境变量和 application.yml 或者 application.properties 文件），部署人员可以在运行时动态选择通道连接 destination（例如，RocketMQ 的 topic 或者 RabbitMQ 的 exchange）。

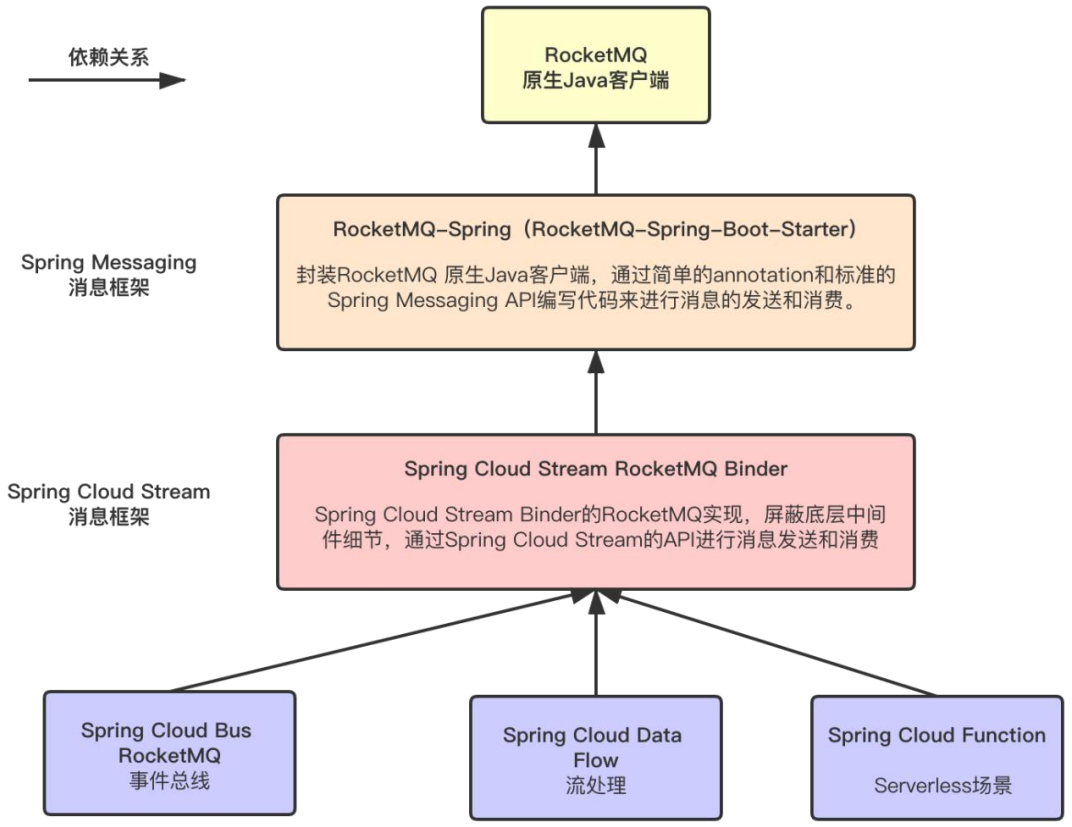
Spring Cloud Stream 屏蔽了底层消息中间件的实现细节，希望以统一的一套 API 来进行消息的发送/消费，底层消息中间件的实现细节由各消息中间件的 Binder 完成。Spring 官方实现了 Rabbit binder 和 Kafka Binder。Spring Cloud Alibaba 实现了 RocketMQ Binder[3]，其主要实现原理是把发送消息最终代理给了 RocketMQ-Spring 的 RocketMQTemplate，在消费端则内部会启动 RocketMQ-Spring Consumer Container 来接收消息。以此为基础，Spring Cloud Alibaba 还实现了 Spring Cloud Bus RocketMQ，用户可以使用 RocketMQ 作为 Spring Cloud 体系内的消息总线，来

连接分布式系统的所有节点。通过 Spring Cloud Stream RocketMQ Binder，RocketMQ 可以与 Spring Cloud 生态更好的结合。比如与 Spring Cloud Data Flow、Spring Cloud Funtion 结合，让 RocketMQ 可以在 Spring 流计算生态、Serverless(FaaS)项目中被使用。

如今 Spring Cloud Stream RocketMQ Binder 和 Spring Cloud Bus RocketMQ 做为 Spring Cloud Alibaba 的实现已登陆 Spring 的官网[4]，Spring Cloud Alibaba 也成为 Spring Cloud 最活跃的实现。

四、如何在 Spring 生态中选择 RocketMQ 实现?

通过介绍 Spring 中的消息框架，介绍了以 RocketMQ 为基础与 Spring 消息框架结合的几个项目，主要是 RocketMQ-Spring、Spring Cloud Stream RocketMQ Binder、Spring Cloud Bus RocketMQ、Spring Data Flow 和 Spring Cloud Function。它们之间的关系可以如下图表示。



如何在实际业务开发中选择相应项目进行使用？下表列出了每个项目的特点和使用场景。

项目	特点	使用场景
RocketMQ-Spring	1.作为起步依赖，简单引入一个包就能在 Spring 生态用到 RocketMQ 客户端的所有功能。 2.利用了大量自动配置和注解简化了编程模型，并且支持 Spring Messaging API 3.与 RocketMQ 原生 Java SDK 的功能完全对齐	适合在 Spring Boot 中使用 RocketMQ 的用户，希望能用到 RocketMQ 原生 java 客户端的所有功能，并通过 Spring 注解和自动配置简化编程模型。
Spring Cloud Stream RocketMQ Binder	1.屏蔽底层 MQ 实现细节，上层 Spring Cloud Stream 的 API 是统一的。如果想从 Kafka 切到 RocketMQ，直接改个配置即可。 2.与 Spring Cloud 生态整合更加方便。比如 Spring Cloud Data Flow，这上面的流计算都是基于 Spring Cloud Stream；Spring Cloud Bus 消息总线内部也是用的 Spring Cloud Stream。 3.Spring Cloud Stream 提供的注解，编程体验都是非常棒。	在代码层面能完全屏蔽底层消息中间件的用户，并且希望能项目能更好的接入 Spring Cloud 生态（Spring Cloud Data Flow、Spring Cloud Funtcion 等）。

项目	特点	使用场景
Spring Cloud Bus RocketMQ	将 RocketMQ 作为事件的“传输器”，通过发送事件（消息）到消息队列上，从而广播到订阅该事件（消息）的所有节点上。完成事件的分发和通知。	在 Spring 生态中希望用 RocketMQ 做消息总线的用户，可以用在应用间事件的通信，配置中心客户端刷新等场景
Spring Cloud Data Flow	以 Source/Processor/Sink 组件进行流式任务处理。RocketMQ 作为流处理过程中的中间存储组件	流处理，大数据处理场景
Spring Cloud Function	消息的消费/生产/处理都是一次函数调用，融合 Java 生态的 Function 模型	Serverless 场景

RocketMQ 作为业务消息的首选，在消息和流处理领域被广泛应用。而微服务生态 Spring 框架也是业务开发中最被，两者的完美契合使得 RocketMQ 成为 Spring Messaing 实现中最受欢迎的消息实现。书的后半部分讲给各位开发者详细讲述在 Spring 生态中使用 RocketMQ 的三种主流的方式。

RocketMQ Spring 最初的故事：罗美琪和春波特的故事...

作者：辽天

rocketmq-spring 经过 6 个多月的孵化，作为 Apache RocketMQ 的子项目正式毕业，发布了第一个 Release 版本 2.0.1。这个项目是把 RocketMQ 的客户端使用 Spring Boot 的方式进行了封装，可以让用户通过简单的 annotation 和标准的 Spring Messaging API 编写代码来进行消息的发送和消费。

在项目发布阶段我们很荣幸的邀请了 Spring 社区的原创人员对我们的代码进行了 Review，通过几轮 slack 上的深入交流感受到了 Spring 团队对开源代码质量的标准，对 SpringBoot 项目细节的要求。本文是对 Review 和代码改进过程中的经验和技巧的总结，希望从事 Spring Boot 开发的同学有帮助。我们把这个过程整理成 RocketMQ 社区的贡献者罗美琪和 Spring 社区的春波特（SpringBoot）的故事。

一、故事的开始

故事的开始是这样的，罗美琪美眉有一套 RocketMQ 的客户端代码，负责发送消息和消费消息。早早的听说春波特小哥哥的大名，通过 Spring Boot 可以把自己客户端调用变得非常简单，只使用一些简单的注解(annotation)和代码就可以使用独立应用的方式启动，省去了复杂的代码编写和参数配置。

聪明的她参考了业界已经实现的消息组件的 Spring 实现了一个 RocketMQ Spring 客户端:

1.需要一个消息的发送客户端，它是一个自动创建的 Spring Bean,并且相关属性要能够根据配置文件的配置自动设置， 命名它为: RocketMQTemplate，同时让它封装发送消息的各种同步和异步的方法。

```
@Resourceprivate RocketMQTemplate rocketMQTemplate;
...
SendResult sendResult = rocketMQTemplate.syncSend(xxxTopic, "Hello, World!");
```

2.需要消息的接收客户端，它是一个能够被应用回调的 Listener，来将消费消息回调给用户进行相关的处理。

```
@Service@RocketMQMessageListener(topic = "xxx", consumerGroup = "xxx_consumer")
public class StringConsumer implements RocketMQListener<String> {
 @Override public void onMessage(String message) {
    System.out.printf("------- StringConsumer received: %s \n", message);
 }
}
```

特别说明一下：这个消费客户端 Listener 需要通过一个自定义的注解。

@RocketMQMessageListener 来标注，这个注解的作用有两个：

- 定义消息消费的配置参数(如: 消费的 topic，是否顺序消费，消费组等)；

- 可以让 spring-boot 在启动过程中发现标注了这个注解的所有 Listener，并进行初始化，详见 ListenerContainerConfiguration 类及其实现 SmartInitializingSingleton 的接口方法 afterSingletonsInstantiated()。

通过研究发现，Spring-Boot 最核心的实现是自动化配置(auto configuration)，它需要分为三个部分：

- AutoConfiguration 类，它由@Configuration 标注，用来创建 RocketMQ 客户端所需要的 SpringBean，如上面所提到的 RocketMQTemplate 和能够处理消费回调 Listener 的容器，每个 Listener 对应一个容器 SpringBean 来启动 MQPushConsumer，并将来将监听到的消费消息并推送给 Listener 进行回调。可参考 RocketMQ AutoConfiguration.java (编者注：这个是最终发布的类，没有 review 的痕迹啦)
- 上面定义的 Configuration 类，它本身并不会“自动”配置，需要由 META-INF/spring.factories 来声明，可参考 spring.factories 使用这个 META 配置的好处是上层用户不需要关心自动配置类的细节和开关，只要 classpath 中有这个 META-INF 文件和 Configuration 类，即可自动配置。
- 另外，上面定义的 Configuration 类，还定义了@EnableConfiguraitonProperties 注解来引入 ConfigurationProperties 类，它的作用是定义自动配置的属性,可参考 RocketMQProperties.java 上层用户可以根据这个类里定义的属性来配置相关的属性文件(即 META-INF/application.properties 或 META-INF/application.yaml)

二、故事的发展

罗美琪美眉按照这个思路开发完成了 RocketMQ SpringBoot 封装并形成了 starter 交给社区的小伙伴们试用，nice，大家使用后反馈效果不错。但是还是想请教一下专业的春波特小哥哥，看看他的意见。

春波特小哥哥相当的负责地对罗美琪的代码进行了 Review，首先他抛出了两个链接:

https://github.com/spring-projects/spring-boot/wiki/Building-On-Spring-Boot

https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-developing-auto-configuration.html

然后解释道："在 Spring Boot 中包含两个概念: auto-configuration 和 starter-POMs，它们之间相互关联，但是不是简单绑定在一起的：

a. auto-configuration 负责响应应用程序的当前状态并配置适当的 Spring Bean。它放在用户的 CLASSPATH 中结合在 CLASSPATH 中的其它依赖就可以提供相关的功能；

b. Starter-POM 负责把 auto-configuration 和一些附加的依赖组织在一起，提供开箱即用的功能，它通常是一个 maven project，里面只是一个 POM 文件，不需要包含任何附加的 classes 或 resources.

换句话说，starter-POM 负责配置全量的 classpath，而 auto-configuration 负责具体的响应(实现)；前者是 total-solution，后者可以按需使用。

你现在的系统是单一的一个 module 把 auto-configuration 和 starter-POM 混在了一起，这个不利于以后的扩展和模块的单独使用。"

罗美琪了解到了区分确实对日后的项目维护很重要，于是将代码进行了模块化

|--- rocketmq-spring-boot-parent 父 POM
|--- rocketmq-spring-boot auto-configuraiton 模块
|--- rocketmq-spring-stater starter 模块 (实际上只包含一个 pom.xml 文件)
|--- rocketmq-spring-samples 调用 starter 的示例样本

"很好，这样的模块结构就清晰多了"，春波特小哥哥点头，"但是这个 AutoConfiguration 文件里的一些标签的用法并不正确，帮你注释一下，另外，考虑到 Spring 官方到明年 8 月 Spring Boot 1.X 将不再提供支持，所以建议实现直接支持 Spring Boot 2.X"

```
@Configuration

@EnableConfigurationProperties(RocketMQProperties.class)

@ConditionalOnClass(MQClientAPIImpl.class)

@Order ~~春波特: 这个类里使用 Order 很不合理呵，不建议使用，完全可以通过其他方式控制 runtime 是 Bean 的构建顺序
@Slf4j
```

```
public class RocketMQAutoConfiguration {

    @Bean

    @ConditionalOnClass(DefaultMQProducer.class) ~~春波特: 属性直接使用类是不科学的，需要用(name="类全名") 方式，这样在类不在 classpath 时，不会抛出 CNFE

    @ConditionalOnMissingBean(DefaultMQProducer.class)

    @ConditionalOnProperty(prefix = "spring.rocketmq", value = {"nameServer", "producer.group"}) ~~春波特: nameServer 属性名要写成 name-server [1]

    @Order(1) ~~春波特: 删掉呵    public DefaultMQProducer mqProducer(RocketMQProperties rocketMQProperties) {

        ...

    }

    @Bean

    @ConditionalOnClass(ObjectMapper.class)

    @ConditionalOnMissingBean(name = "rocketMQMessageObjectMapper") ~~春波特: 不建议与具体的实例名绑定，设计的意图是使用系统中已经存在的 ObjectMapper，如果没有，则在这里实例化一个，需要改成

     @ConditionalOnMissingBean(ObjectMapper.class)
```

```
public ObjectMapper rocketMQMessageObjectMapper() {
    return new ObjectMapper();
}
@Bean(destroyMethod = "destroy")
@ConditionalOnBean(DefaultMQProducer.class)
@ConditionalOnMissingBean(name = "rocketMQTemplate") ~~春波特: 与上面一样
@Order(2) ~~春波特: 删掉呵
 public RocketMQTemplate rocketMQTemplate(DefaultMQProducer mqProducer,
    @Autowired(required = false)                    ~~春波特: 删掉
    @Qualifier("rocketMQMessageObjectMapper") ~~春波特: 删掉，不要与具体实例绑定

        ObjectMapper objectMapper) {
    RocketMQTemplate rocketMQTemplate = new RocketMQTemplate();
    rocketMQTemplate.setProducer(mqProducer);
    if (Objects.nonNull(objectMapper)) {
```

```
                rocketMQTemplate.setObjectMapper(objectMapper);

            }

            return rocketMQTemplate;

        }

        @Bean(name = RocketMQConfigUtils.ROCKETMQ_TRANSACTION_ANNOTATION_PROCESSOR_BEAN_NAME)

        @ConditionalOnBean(TransactionHandlerRegistry.class)

        @Role(BeanDefinition.ROLE_INFRASTRUCTURE) ~~春波特: 这个 bean（RocketMQTransactionAnnotationProcessor）建议声明成 static 的，因为这个 RocketMQTransactionAnnotationProcessor 实现了 BeanPostProcessor 接口,接口里方法在调用的时候（创建 Transaction 相关的 Bean 的时候）可以直接使用这个 static 实例，而不要等到这个 Configuration 类的其他的 Bean 都构建好 [2]

        public RocketMQTransactionAnnotationProcessor transactionAnnotationProcessor(

        TransactionHandlerRegistry transactionHandlerRegistry) {

            return new RocketMQTransactionAnnotationProcessor(transactionHandlerRegistry);

        }

        @Configuration ~~春波特: 这个内嵌的 Configuration 类比较复杂，建议独立成一个顶级类，并且使用
```

```
@Import 在主 Configuration 类中引入

@ConditionalOnClass(DefaultMQPushConsumer.class)

@EnableConfigurationProperties(RocketMQProperties.class)

@ConditionalOnProperty(prefix = "spring.rocketmq", value = "nameServer") ~~春波特: name-server

public static class ListenerContainerConfiguration implements ApplicationContextAware, InitializingBean {

    ...

    @Resource ~~春波特: 删掉这个 annotation, 这个 field injection 的方式不推荐，建议使用 setter 或者构造参数的方式初始化成员变量

    private StandardEnvironment environment;

     @Autowired(required = false)  ~~春波特: 这个注解是不需要的

     public ListenerContainerConfiguration(

            @Qualifier("rocketMQMessageObjectMapper") ObjectMapper objectMapper) { ~~春波特: @Qualifier 不需要

            this.objectMapper = objectMapper;

     }
```

注[1]: 在声明属性的时候不要使用驼峰命名法，要使用-横线分隔，这样才能支持属性名的松散规则(relaxed rules)。

注[2]: BeanPostProcessor 接口作用是：如果需要在 Spring 容器完成 Bean 的实例化、配置和其他的初始化的前后添加一些自己的逻辑处理，就可以定义一个或者多个 BeanPostProcessor 接口的实现，然后注册到容器中。为什么建议声明成 static 的，春波特的英文原文:

If they don't we basically register the post-processor at the same "time" as all the other beans in that class and the contract of BPP is that it must be registered very early on. This may not make a difference for this particular class but flagging it as static as the side effect to make clear your BPP implementation is not supposed to drag other beans via dependency injection.

AutoConfiguration 里果真很有学问，罗美琪迅速的调整了代码，一下看起来清爽了许多。不过还是被春波特提出了两点建议:

```
@Configuration
public class ListenerContainerConfiguration implements ApplicationContextAware, SmartI
nitializingSingleton {
 private ObjectMapper objectMapper = new ObjectMapper(); ~~春波特: 性能上考虑，不要
初始化这个成员变量，既然这个成员是在构造/setter 方法里设置的，就不要在这里初始化，尤其是当
它的构造成本很高的时候。
```

```
private void registerContainer(String beanName, Object bean) { Class<?> clazz = AopUtils.getTargetClass(bean);
if(!RocketMQListener.class.isAssignableFrom(bean.getClass())){
  throw new IllegalStateException(clazz + " is not instance of " + RocketMQListener.class.getName());
}
RocketMQListener rocketMQListener = (RocketMQListener) bean; RocketMQMessageListener annotation = clazz.getAnnotation(RocketMQMessageListener.class);
validate(annotation); ~~春波特: 下面的这种手工注册 Bean 的方式是 Spring 4.x 里提供能，可以考虑使用 Spring5.0 里提供的 GenericApplicationContext.registerBean 的方法,通过 supplier 调用 new 来构造 Bean 实例 [3]
BeanDefinitionBuilder beanBuilder = BeanDefinitionBuilder.rootBeanDefinition(DefaultRocketMQListenerContainer.class);
beanBuilder.addPropertyValue(PROP_NAMESERVER, rocketMQProperties.getNameServer());
...
beanBuilder.setDestroyMethodName(METHOD_DESTROY);
String containerBeanName = String.format("%s_%s", DefaultRocketMQListenerContainer.class.getName(), counter.incrementAndGet());
DefaultListableBeanFactory beanFactory = (DefaultListableBeanFactory) applicationContext.getBeanFactory();
beanFactory.registerBeanDefinition(containerBeanName, beanBuilder.getBeanDefinition());
DefaultRocketMQListenerContainer container = beanFactory.getBean(containerBeanName, DefaultRocketMQListenerContainer.class); ~~春波特: 你这里的启动方法是通过 afterPropertiesSet() 调用的，这个是不建议的，应该实现 SmartLifecycle 来定义启停方法，这样在 ApplicationContext 刷新时能够自动启动；并且避免了 context 初始化时由于底层资源问题导致的挂住(stuck)的危险
if (!container.isStarted()) {
  try {
    container.start();
  } catch (Exception e) {
```

```
        log.error("started container failed. {}", container, e);       throw new RuntimeException
(e);
        }
      }
      ...
    }
    }
```

注[3]: 使用 GenericApplicationContext.registerBean 的方式

```
public final < T > void registerBean(
 Class< T > beanClass, Supplier< T > supplier, BeanDefinitionCustomizer…
ustomizers)
```

"**还有，还有"**，在罗美琪采纳了春波特的意见比较大地调整了代码之后，春波特哥哥有提出了 Spring Boot 特有的几个要求：

- 使用 Spring 的 Assert 在传统的 Java 代码中我们使用 assert 进行断言，Spring Boot 中断言需要使用它自有的 Assert 类，如下示例:

```
import org.springframework.util.Assert;
...
Assert.hasText(nameServer, "[rocketmq.name-server] must not be null");
```

- Auto Configuration 单元测试使用 Spring 2.0 提供的 ApplicationContextRunner

```
public class RocketMQAutoConfigurationTest {
 private ApplicationContextRunner runner = new ApplicationContextRunner() .withConfi
guration(AutoConfigurations.of(RocketMQAutoConfiguration.class));

 @Test(expected = NoSuchBeanDefinitionException.class) public voidtestRocketMQAutoCo
nfigurationNotCreatedByDefault() {
 runner.run(context -> context.getBean(RocketMQAutoConfiguration.class)); }
 @Test
 public void testDefaultMQProducerWithRelaxPropertyName() {
 runner.withPropertyValues("rocketmq.name-server=127.0.0.1:9876", "rocketmq.prod
ucer.group=spring_rocketmq").
 run((context) -> {
 assertThat(context).hasSingleBean(DefaultMQProducer.class); assertThat(co
ntext).hasSingleBean(RocketMQProperties.class); });
 }
```

- 在 auto-configuration 模块的 pom.xml 文件里，加入 spring-boot-configuration-processor 注解处理器，这样它能够生成辅助元数据文件，加快启动时间。详情见这里(https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-developing-auto-configuration.html#boot-features-custom-starter-module-autoconfigure)。

最后，春波特还相当专业地向罗美琪美眉提供了如下两方面的意见：

1. 通用的规范，好的代码要易读易于维护

注释与命名规范

我们常用的代码注释分为多行(/** … */)和单行(// ...)两种类型，对于需要说明的成员变量，方法或者代码逻辑应该提供多行注释；有些简单的代码逻辑注释也可以使用单行注释。在注释时通用的要求是首字母大写开头，并且使用句号结尾；对于单行注释，也要求首字母大写开头；并且不建议行尾单行注释。

在变量和方法命名时尽量用词准确，并且尽量不要使用缩写，如：sendMsgTimeout，建议写成 sendMessageTimeout；包名 supports，建议改成 support。

是否需要使用 Lombok

使用 Lombok 的好处是代码更加简洁，只需要使用一些注释就可省略 constructor，setter 和 getter 等诸多方法(bolierplate code)；但是也有一个坏处就是需要开发者在自己的 IDE 环境配置 Lombok 插件来支持这一功能，所以 Spring 社区的推荐方式是不使用 Lombok，以便新用户可以直接查看和维护代码，不依赖 IDE 的设置。

对于包名(package)的控制

如果一个包目录下没有任何 class，建议要去掉这个包目录。例如：

org.apache.rocketmq.spring.starter 在 spring 目录下没有具体的 class 定义，那么应该去掉这层目录（编者注：我们最终把 package 改为 org.apache.rocketmq.spring，将 starter 下的目录和 classes 上移一层）。

我们把所有 Enum 类放在包 org.apache.rocketmq.spring.enums 下，这个包命名并不规范，需要把 Enum 类调整到具体的包中，去掉 enums 包；

类的隐藏，对于有些类，它只被包中的其它类使用，而不需要把具体的使用细节暴漏给最终用户，建议使用 package private 约束，例如: TransactionHandler 类。

不建议使用 Static Import，虽然使用它的好处是更少的代码，坏处是破坏程序的可读性和易维护性。

2. 效率，深入代码的细节

static + final method

一个类的 static 方法不要结合 final，除非这个这个类本身是 final 并且声明 private 构造(ctor)，如果两者结合以为这子类不能再(hiding)定义该方法，给将来的扩展和子类调用带来麻烦。

在配置文件声明的 Bean 尽量使用构造函数或者 Setter 方法设置成员变量，而不要使用@Autowared，@Resource 等方式注入。[4]

不要额外初始化无用的成员变量。

如果一个方法没有任何地方调用，就应该删除；如果一个接口方法不需要，就不要实现这个接口类

注[4]: 下面的截图是有 FieldInjection 转变成构造函数设置的代码示例：

```
public class ListenerContainerConfiguration implements ApplicationContextAware, SmartInitializingSingleton {
    private final static Logger log = LoggerFactory.getLogger(ListenerContainerConfiguration.class);

    private ConfigurableApplicationContext applicationContext;

    private AtomicLong counter = new AtomicLong( initialValue: 0);

    @Resource
    private StandardEnvironment environment;

    @Autowired
    private RocketMQProperties rocketMQProperties;

    @Autowired
    private ObjectMapper objectMapper;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = (ConfigurableApplicationContext) applicationContext;
    }
```


转换成：

```
public class ListenerContainerConfiguration implements ApplicationContextAware, SmartInitializingSingleton {
    private final static Logger log = LoggerFactory.getLogger(ListenerContainerConfiguration.class);

    private ConfigurableApplicationContext applicationContext;

    private AtomicLong counter = new AtomicLong( initialValue: 0);

    private StandardEnvironment environment;

    private RocketMQProperties rocketMQProperties;

    private ObjectMapper objectMapper;

    public ListenerContainerConfiguration(ObjectMapper rocketMQMessageObjectMapper,
        StandardEnvironment environment,
        RocketMQProperties rocketMQProperties) {
        this.objectMapper = rocketMQMessageObjectMapper;
        this.environment = environment;
        this.rocketMQProperties = rocketMQProperties;
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = (ConfigurableApplicationContext) applicationContext;
    }
```


三、故事的结局

罗美琪根据上述的要求调整了代码，使代码质量有了很大的提高，并且总结了 Spring Boot 开发的要点：

- 编写前参考成熟的 spring boot 实现代码。
- 要注意模块的划分，区分 autoconfiguration 和 starter。
- 在编写 autoconfiguration Bean 的时候，注意@Conditional 注解的使用；尽量使用构造器或者 setter 方法来设置变量，避免使用 Field Injection 方式；多个 Configuration Bean 可以使用@Import 关联；使用 Spring 2.0 提供的 AutoConfigruation 测试类。
- 注意一些细节：static 与 BeanPostProcessor； Lifecycle 的使用；不必要的成员属性的初始化等。

通过本次的 Review 工作了解到了 spring-boot 及 auto-configuration 所需要的一些约束条件，信心满满地提交了最终的代码，又可以邀请 RocketMQ 社区的小伙伴们一起使用 rocketmq-spring 功能了，广大读者可以在参考代码库查看到最后修复代码，也希望有更多的宝贵意见反馈和加强，加油！

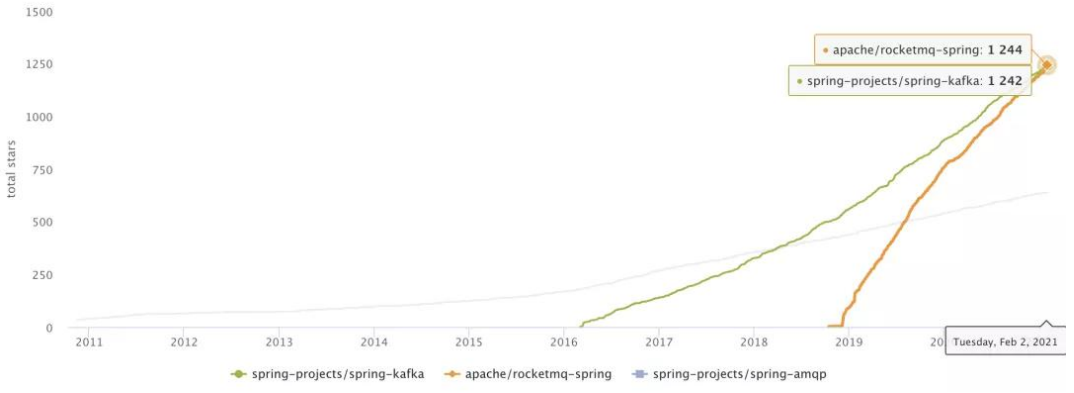
四、后记

开源软件不仅仅是提供一个好用的产品，代码质量和风格也会影响到广大的开发者，活跃的社区贡献者罗美琪还在与 RocketMQ 社区的小伙伴们不断完善 spring 的代码，并邀请春波特的 Spring 社区进行宣讲和介绍，下一步将 rocketmq-spring-starter 推进到 Spring Initializr，让用户可以直接在 https://start.aliyun.com/bootstrap.html 网站上像使用其它 starter（如：Tomcat starter）一样使用 rocketmq-spring.

RocketMQ-Spring 毕业两周年，为什么能成为 Spring 生态中最受欢迎的 messaging 实现

2019 年 1 月，孵化 6 个月的 RocketMQ-Spring 作为 Apache RocketMQ 的子项目正式毕业，发布了第一个 Release 版本 2.0.1。该项目是把 RocketMQ 的客户端使用 Spring Boot 的方式进行了封装，可以让用户通过简单的 annotation 和标准的 Spring Messaging API 编写代码来进行消息的发送和消费。当时 RocketMQ 社区同学请 Spring 社区的同学对 RocketMQ-Spring 代码进行 review，引出一段罗美琪（RocketMQ）和春波特（Spring Boot）的故事。

时隔两年，RocketMQ-Spring 正式发布 2.2.0。在这期间，RocketMQ-Spring 迭代了数个版本，以 RocketMQ-Spring 为基础实现的 Spring Cloud Stream RocketMQ Binder、Spring Cloud Bus RocketMQ 登上了 Spring 的官网，Spring 布道师 baeldung 向国外同学介绍如何使用 RocketMQ-Spring，越来越多国内外的同学开始使用 RocketMQ-Spring 收发消息，RocketMQ-Spring 仓库的 star 数也在短短两年时间内超越了 Spring-Kafka 和 Spring-AMQP（注：两者均由 Spring 社区维护），成为 Apache RocketMQ 最受欢迎的生态项目之一。



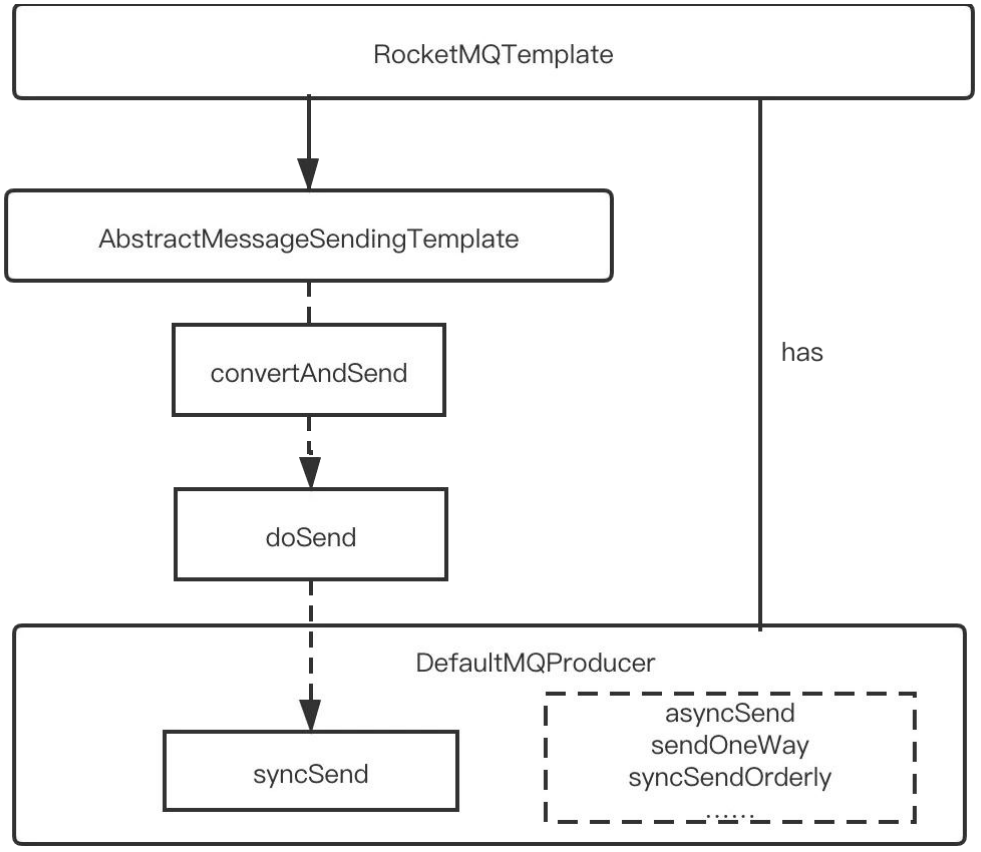
RocketMQ-Spring 的受欢迎一方面得益于支持丰富业务场景的 RocketMQ 与微服务生态 Spring 的完美契合，另一方面也与 RocketMQ-Spring 本身严格遵循 Spring Messaging API 规范，支持丰富的消息类型分不开。

一、遵循 Spring Messaging API 规范

Spring Messaging 提供了一套抽象的 API，对消息发送端和消息接收端的模式进行规定，不同的消息中间件提供商可以在这个模式下提供自己的 Spring 实现：在消息发送端需要实现的是一个 XXXTemplate 形式的 Java Bean，结合 Spring Boot 的自动化配置选项提供多个不同的发送消息方法；在消息的消费端是一个 XXXMessageListener 接口（实现方式通常会使用一个注解来声明一个消息驱动的 POJO），提供回调方法来监听和消费消息，这个接口同样可以使用 Spring Boot 的自动化选项和一些定制化的属性。

1. 发送端

RocketMQ-Spring 在遵循 Spring Messaging API 规范的基础上结合 RocketMQ 自身的功能特点提供了相应的 API。在消息的发送端，RocketMQ-Spring 通过实现 RocketMQTemplate 完成消息的发送。如下图所示，RocketMQTemplate 继承 AbstractMessageSendingTemplate 抽象类，来支持 Spring Messaging API 标准的消息转换和发送方法，这些方法最终会代理给 doSend 方法，doSend 方法会最终调用 syncSend，由 DefaultMQProducer 实现。



除 Spring Messaging API 规范中的方法，RocketMQTemplate 还实现了 RocketMQ 原生客户端的一些方法，来支持更加丰富的消息类型。值得注意的是，相比

于原生客户端需要自己去构建 RocketMQ Message（比如将对象序列化成 byte 数组放入 Message 对象），RocketMQTemplate 可以直接将对象、字符串或者 byte 数组作为参数发送出去（对象序列化操作由 RocketMQ-Spring 内置完成），在消费端约定好对应的 Schema 即可正常收发。

```
RocketMQTemplate Send API:
SendResult syncSend(String destination, Object payload)
SendResult syncSend(String destination, Message<?> message)
void asyncSend(String destination, Message<?> message, SendCallback sendCallback)
void asyncSend(String destination, Message<?> message, SendCallback sendCallback)
.....
```

2. 消费端

在消费端，需要实现一个包含 @RocketMQMessageListener 注解的类（需要实现 RocketMQListener 接口，并实现 onMessage 方法，在注解中进行 topic、consumerGroup 等属性配置），这个 Listener 会一对一的被放置到 DefaultRocketMQListenerContainer 容器对象中，容器对象会根据消费的方式(并发或顺序)，将 RocketMQListener 封装到具体的 RocketMQ 内部的并发或者顺序接口实现。在容器中创建 RocketMQ DefaultPushConsumer 对象，启动并监听定制的 Topic 消息，完成约定 Schema 对象的转换，回调到 Listener 的 onMessage 方法。

```
@Service
@RocketMQMessageListener(topic = "${demo.rocketmq.topic}", consumerGroup =
"string_consumer", selectorExpression = "${demo.rocketmq.tag}")public class StringConsu
mer implements RocketMQListener<String> {
```

```
@Override
public void onMessage(String message) {
System.out.printf("------- StringConsumer received: %s \n", message);
    }
      }
```

除此 Push 接口之外，**在最新的 2.2.0 版本中，RocketMQ-Spring 实现了 RocketMQ Lite Pull Consumer**。通过在配置文件中进行 consumer 的配置，利用 RocketMQTemplate 的 Recevie 方法即可主动 Pull 消息。

```
配置文件 resource/application.properties：
rocketmq.name-server=localhost:9876
rocketmq.consumer.group=my-group1
rocketmq.consumer.topic=test
Pull Consumer 代码：

while(!isStop) {
List<String> messages = rocketMQTemplate.receive(String.class);
 System.out.println(messages);
}
```

二、丰富的消息类型

RocketMQ Spring 消息类型支持方面与 RocketMQ 原生客户端完全对齐，包括同步/异步/one-way、顺序、延迟、批量、事务以及 Request-Reply 消息。在这里，主要介绍较为特殊的事务消息和 request-reply 消息。

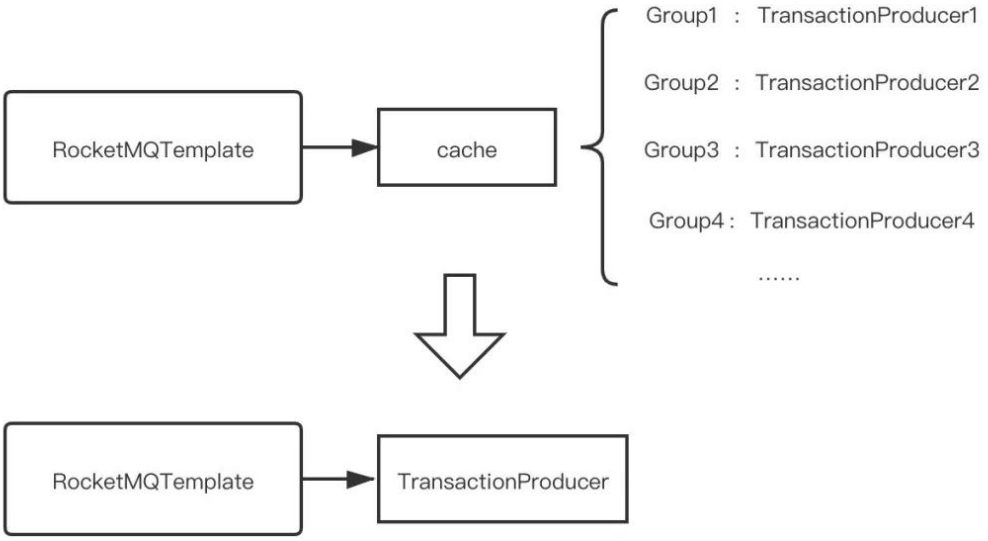
1. 事务消息

RocketMQ 的事务消息不同于 Spring Messaging 中的事务消息，依然采用 RocketMQ 原生事务消息的方案。如下所示，发送事务消息时需要实现一个包含 @RocketMQTransactionListener 注解的类，并实现 executeLocalTransaction 和 checkLocalTransaction 方法，从而来完成执行本地事务以及检查本地事务执行结果。

```
// Build a SpringMessage for sending in transactionMessage msg =
MessageBuilder.withPayload(..)...;
// In sendMessageInTransaction(), the first parameter transaction name ("test")
// must be same with the @RocketMQTransactionListener's member field 'transName'
rocketMQTemplate.sendMessageInTransaction("test-topic", msg, null);

// Define transaction listener with the annotation
@RocketMQTransactionListener@RocketMQTransactionListener
class TransactionListenerImpl implements RocketMQLocalTransactionListener {
@Override
public RocketMQLocalTransactionState executeLocalTransaction(Message msg, Object arg)
{
// ... local transaction process, return bollback, commit or unknown
return RocketMQLocalTransactionState.UNKNOWN;
    }
@Override
public RocketMQLocalTransactionState checkLocalTransaction(Message msg) {
 // ... check transaction status and return bollback, commit or unknown
return RocketMQLocalTransactionState.COMMIT;
    }
}
```

在 2.1.0 版本中，RocketMQ-Spring 重构了事务消息的实现，如下图所示，旧版本中每一个 group 对应一个 TransactionProducer，而在新版本中改为每一个 RocketMQTemplate 对应一个 TransationProducer，从而解决了并发使用多个事务消息的问题。当用户需要在单进程使用多个事务消息时，可以使用 ExtRocketMQTemplate 来完成（一般情况下，推荐一个进程使用一个 RocketMQTemplate，ExtRocketMQTemplate 可以使用在同进程中需要使用多个 Producer / LitePullConsumer 的场景，可以为 ExtRocketMQTemplate 指定与标准模版 RocketMQTemplate 不同的 nameserver、group 等配置），并在对应的 RocketMQTransactionListener 注解中指定 rocketMQTemplateBeanName 为 ExtRocketMQTemplate 的 BeanName。



2. Request-Reply 消息

在 2.1.0 版本中，RocketMQ-Spring 开始支持 Request-Reply 消息。Request-Reply 消息指的是上游服务投递消息后进入等待被通知的状态，直到消费端返回结果并

返回给发送端。在 RocketMQ-Spring 中，发送端通过 RocketMQTemplate 的 sendAndReceivce 方法进行发送，如下所示，主要有同步和异步两种方式。异步方式中通过实现 RocketMQLocalRequestCallback 进行回调。

```
// 同步发送 request 并且等待 String 类型的返回值
String replyString = rocketMQTemplate.sendAndReceive("stringRequestTopic", "request string", String.class);

// 异步发送 request 并且等待 User 类型的返回值
rocketMQTemplate.sendAndReceive("objectRequestTopic", new User("requestUserName",(byte) 9), new RocketMQLocalRequestCallback<User>() {
@Override public void onSuccess(User message) {
 ......
 }
@Override public void onException(Throwable e) {
    ......
 }
});
```

在消费端，仍然需要实现一个包含 @RocketMQMessageListener 注解的类，但需要实现的接口是 RocketMQReplyListener<T, R> 接口（普通消息为 RocketMQListener<T> 接口），其中 T 表示接收值的类型，R 表示返回值的类型，接口需要实现带返回值的 onMessage 方法，返回值的内容返回给对应的 Producer。

```
@Service
@RocketMQMessageListener(topic = "stringRequestTopic", consumerGroup = "stringRequestConsumer")
public class StringConsumerWithReplyString implements RocketMQReplyListener<String, String> {
@Override
public String onMessage(String message) {
    ......
  return "reply string";
  }
}
```

RocketMQ-Spring 遵循 Spring 约定大于配置（Convention over configuration）的理念，通过启动器（Spring Boot Starter）的方式，在 pom 文件引入依赖（groupId：org.apache.rocketmq，artifactId：rocketmq-spring-boot-starter）便可以在 Spring Boot 中集成所有 RocketMQ 客户端的所有功能，通过简单的注解使用即可完成消息的收发。在 RocketMQ-Spring Github Wiki 中有更加详细的用法和常见问题解答。

据统计，从 RocketMQ-Spring 发布第一个正式版本以来，RocketMQ-Spring 完成 16 个 bug 修复，37 个 imporvement，其中包括事务消息重构，消息过滤、消息序列化、多实例 RocketMQTemplate 优化等重要优化，欢迎更多的小伙伴能参与到 RocketMQ 社区的建设中来，罗美琪（RocketMQ）和春波特（Spring Boot）的故事还在继续...

方法一：使用 rocketmq-spring-boot-starter 来配置、发送和消费 RocketMQ 消息

作者：辽天

本文将对 rocktmq-spring-boot 的设计实现做一个简单的介绍，读者可以通过本文了解将 RocketMQ Client 端集成为 spring-boot-starter 框架的开发细节，然后通过一个简单的示例来一步一步的讲解如何使用这个 spring-boot-starter 工具包来配置，发送和消费 RocketMQ 消息。

本文配套可交互教程已登录阿里云知行动手实验室，PC 端登录 start.aliyun.com 在浏览器中立即体验。

作者简介：辽天，阿里巴巴技术专家，Apache RocketMQ 内核控，拥有多年分布式系统研发经验，对 Microservice、Messaging 和 Storage 等领域有深刻理解， 目前专注 RocketMQ 内核优化以及 Messaging 生态建设。

通过本文，您将了解到：

- Spring 的消息框架介绍
- rocketmq-spring-boot 具体实现
- 使用示例

一、前言

上世纪 90 年代末，随着 Java EE(Enterprise Edition)的出现，特别是 Enterprise Java Beans 的使用需要复杂的描述符配置和死板复杂的代码实现，增加了广大开发者的学习曲线和开发成本，由此基于简单的 XML 配置和普通 Java 对象(Plain Old Java Objects)的 Spring 技术应运而生，依赖注入(Dependency Injection)，控制反转(Inversion of Control)和面向切面编程(AOP)的技术更加敏捷地解决了传统 Java 企业及版本的不足。

随着 Spring 的持续演进，基于注解(Annotation)的配置逐渐取代了 XML 文件配置，2014 年 4 月 1 日，Spring Boot 1.0.0 正式发布，它基于“约定大于配置”（Convention over configuration)这一理念来快速地开发、测试、运行和部署 Spring 应用，并能通过简单地与各种启动器(如 spring-boot-web-starter)结合，让应用直接以命令行的方式运行，不需再部署到独立容器中。这种简便直接快速构建和开发应用的过程，可以使用约定的配置并且简化部署，受到越来越多的开发者的欢迎。

Apache RocketMQ 是业界知名的分布式消息和流处理中间件，简单地理解，它由 Broker 服务器和客户端两部分组成：

其中客户端一个是消息发布者客户端(Producer)，它负责向 Broker 服务器发送消息；另外一个是消息的消费者客户端(Consumer)，多个消费者可以组成一个消费组，来订阅和拉取消费 Broker 服务器上存储的消息。

为了利用 Spring Boot 的快速开发和让用户能够更灵活地使用 RocketMQ 消息客户端，Apache RocketMQ 社区推出了 spring-boot-starter 实现。随着分布式事务消息功能在 RocketMQ 4.3.0 版本的发布，近期升级了相关的 spring-boot 代码，通过注解方式支持分布式事务的回查和事务消息的发送。

本文将对当前的设计实现做一个简单的介绍，读者可以通过本文了解将 RocketMQ Client 端集成为 spring-boot-starter 框架的开发细节，然后通过一个简单的示例来一步一步的讲解如何使用这个 spring-boot-starter 工具包来配置，发送和消费 RocketMQ 消息。

二、Spring 中的消息框架

顺便在这里讨论一下在 Spring 中关于消息的两个主要的框架，即 Spring Messaging 和 Spring Cloud Stream。它们都能够与 Spring Boot 整合并提供了一些参考的实现。和所有的实现框架一样，消息框架的目的是实现轻量级的消息驱动的微服务，可以有效地简化开发人员对消息中间件的使用复杂度，让系统开发人员可以有更多的精力关注于核心业务逻辑的处理。

1. Spring Messaging

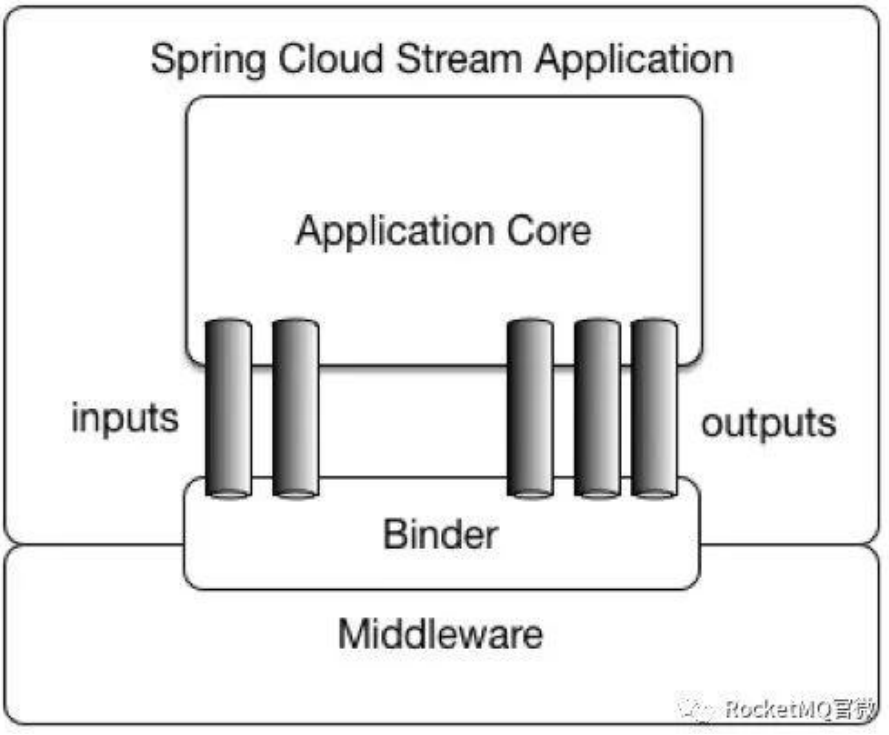
Spring Messaging 是 Spring Framework 4 中添加的模块，是 Spring 与消息系统集成的一个扩展性的支持。它实现了从基于 JmsTemplate 的简单的使用 JMS 接口到异步接收消息的一整套完整的基础架构，Spring AMQP 提供了该协议所要求的类似的功能集。 在与 Spring Boot 的集成后，它拥有了自动配置能力，能够在测试和运行时与相应的消息传递系统进行集成。

单纯对于客户端而言，Spring Messaging 提供了一套抽象的 API 或者说是约定的标准，对消息发送端和消息接收端的模式进行规定，不同的消息中间件提供商可以在这个模式下提供自己的 Spring 实现：在消息发送端需要实现的是一个 XXXTemplate 形式的 Java Bean，结合 Spring Boot 的自动化配置选项提供多个不同的发送消息方法；在消息的消费端是一个 XXXMessageListener 接口（实现方式通常会使用一个注解来声明一个消息驱动的 POJO），提供回调方法来监听和消费消息，这个接口同样可以使用 Spring Boot 的自动化选项和一些定制化的属性。

如果有兴趣深入的了解 Spring Messaging 及针对不同的消息产品的使用，推荐阅读这个文件。参考 Spring Messaging 的既有实现，RocketMQ 的 spring-boot-starter 中遵循了相关的设计模式并结合 RocketMQ 自身的功能特点提供了相应的 API(如，顺序，异步和事务半消息等)。

2. Spring Cloud Stream

Spring Cloud Stream 结合了 Spring Integration 的注解和功能，它的应用模型如下：



该图片引自 spring cloud stream

Spring Cloud Stream 框架中提供一个独立的应用内核，它通过输入(@Input)和输出(@Output)通道与外部世界进行通信，消息源端(Source)通过输入通道发送消息，消费目标端(Sink)通过监听输出通道来获取消费的消息。这些通道通过专用的 Binder 实现与外部代理连接。开发人员的代码只需要针对应用内核提供的固定的接口和注解方式进行编程，而不需要关心运行时具体的 Binder 绑定的消息中间件。在运行时，Spring Cloud Stream 能够自动探测并使用在 classpath 下找到的 Binder。

这样开发人员可以轻松地在相同的代码中使用不同类型的中间件：仅仅需要在构建时包含进不同的 Binder。在更加复杂的使用场景中，也可以在应用中打包多个 Binder 并让它自己选择 Binder，甚至在运行时为不同的通道使用不同的 Binder。

Binder 抽象使得 Spring Cloud Stream 应用可以灵活的连接到中间件，加之 Spring Cloud Stream 使用利用了 Spring Boot 的灵活配置配置能力，这样的配置可以通过外部配置的属性和 Spring Boo 支持的任何形式来提供（包括应用启动参数、环境变量和 application.yml 或者 application.properties 文件），部署人员可以在运行时动态选择通道连接 destination（例如，Kafka 的 topic 或者 RabbitMQ 的 exchange）。

Binder SPI 的方式来让消息中间件产品使用可扩展的 API 来编写相应的 Binder，并集成到 Spring Cloud Steam 环境，目前 RocketMQ 还没有提供相关的 Binder，我们计划在下一步将完善这一功能，也希望社区里有这方面经验的同学积极尝试，贡献 PR 或建议。

三、spring-boot-starter 的实现

在开始的时候我们已经知道，spring boot starter 构造的启动器对于使用者是非常方便的，使用者只要在 pom.xml 引入 starter 的依赖定义，相应的编译，运行和部署功能就全部自动引入。因此常用的开源组件都会为 Spring 的用户提供一个 spring-boot-starter 封装给开发者，让开发者非常方便集成和使用，这里我们详细的介绍一下 RocketMQ（客户端）的 starter 实现过程。

1. spring-boot-starter 的实现步骤

对于一个 spring-boot-starter 实现需要包含如下几个部分：

在 pom.xml 的定义

- 定义最终要生成的 starter 组件信息

```
<groupId>org.apache.rocketmq</groupId>
<artifactId>spring-boot-starter-rocketmq</artifactId>
<version>1.0.0-SNAPSHOT</version>
```

- 定义依赖包

它分为两个部分：A、Spring 自身的依赖包； B、RocketMQ 的依赖包

```
<dependencies>
    <!-- spring-boot-start internal depdencies -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>


    <!-- rocketmq dependencies -->
    <dependency>
        <groupId>org.apache.rocketmq</groupId>
        <artifactId>rocketmq-client</artifactId>
        <version>${rocketmq-version}</version>
    </dependency>
</dependencies>
    <dependencyManagement>
    <dependencies>
        <!-- spring-boot-start parent depdency definition -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>${spring.boot.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

配置文件类

定义应用属性配置文件类 RocketMQProperties,这个 Bean 定义一组默认的属性值。用户在使用最终的 starter 时，可以根据这个类定义的属性来修改取值，当然不是直接修改这个类的配置，而是 spring-boot 应用中对应的配置文件：src/main/resources/application.properties.

定义自动加载类

定义 src/resources/META-INF/spring.factories 文件中的自动加载类，其目的是让 spring boot 更具文中中所指定的自动化配置类来自动初始化相关的 Bean，Component 或 Service，它的内容如下：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.apache.rocketmq.spring.starter.RocketMQAutoConfiguration
```

在 RocketMQAutoConfiguration 类的具体实现中，定义开放给用户直接使用的 Bean 对象. 包括：

- RocketMQProperties 加载应用属性配置文件的处理类；
- RocketMQTemplate 发送端用户发送消息的发送模板类；
- ListenerContainerConfiguration 容器 Bean 负责发现和注册消费端消费实现接口类，这个类要求：由@RocketMQMessageListener 注解标注；实现 RocketMQListener 泛化接口。

最后具体的 RocketMQ 相关的封装。

在发送端（producer）和消费端(consumer)客户端分别进行封装，在当前的实现版本提供了对 Spring Messaging 接口的兼容方式。

2. 消息发送端实现

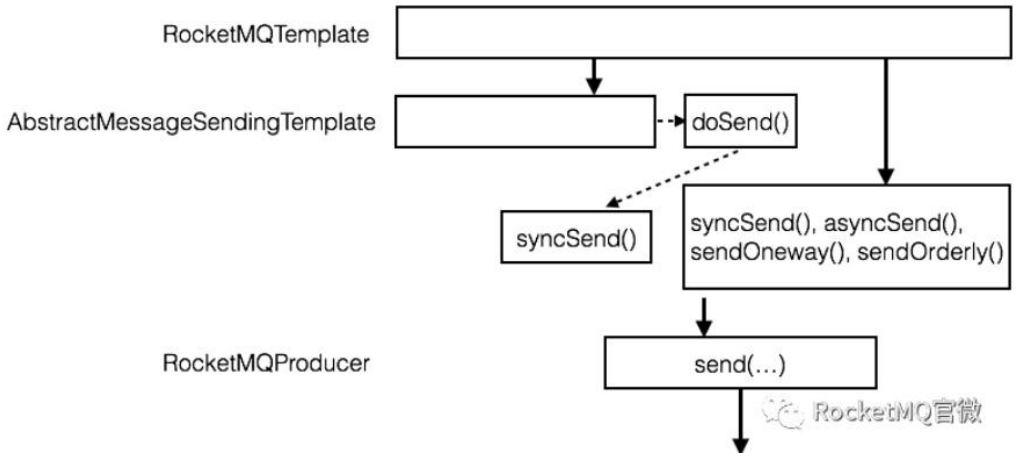
普通发送端

发送端的代码封装在 RocketMQTemplate POJO 中，下图是发送端的相关代码的调用关系图：

为了与 Spring Messaging 的发送模板兼容，在 RocketMQTemplate 集成了 AbstractMessageSendingTemplate 抽象类，来支持相关的消息转换和发送方法，这些方法最终会代理给 doSend()方法；doSend()以及 RocoketMQ 所特有的一些方法如异步，单向和顺序等方法直接添加到 RoketMQTempalte 中，这些方法直接代理调用到 RocketMQ 的 Producer API 来进行消息的发送。

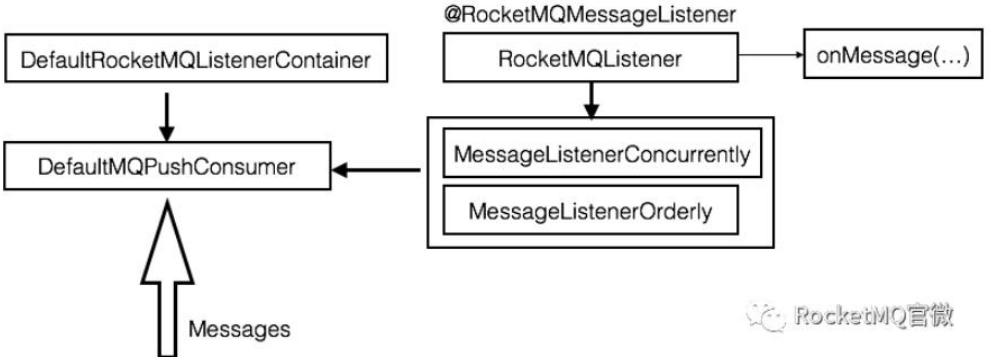
事务消息发送端

对于事务消息的处理，在消息发送端进行了部分的扩展，参考下图的调用关系类图：



RocketMQTemplate 里加入了一个发送事务消息的方法 sendMessageInTransaction()，并且最终这个方法会代理到 RocketMQ 的 TransactionProducer 进行调用，在这个 Producer 上会注册其关联的 TransactionListener 实现类，以便在发送消息后能够对 TransactionListener 里的方法实现进行调用。

3. 消息消费端实现



在消费端 Spring-Boot 应用启动后，会扫描所有包含@RocketMQMessageListener 注解的类(这些类需要集成 RocketMQListener 接口，并实现 onMessage()方法)，这个 Listener 会一对一的被放置到

DefaultRocketMQListenerContainer 容器对象中，容器对象会根据消费的方式(并发或顺序)，将 RocketMQListener 封装到具体的 RocketMQ 内部的并发或者顺序接口实现。在容器中创建 RocketMQ Consumer 对象，启动并监听定制的 Topic 消息，如果有消费消息，则回调到 Listener 的 onMessage()方法。

四、使用示例

上面的一章介绍了 RocketMQ 在 spring-boot-starter 方式的实现，这里通过一个最简单的消息发送和消费的例子来介绍如何使这个 rocketmq-spring-boot-starter。

1. RocketMQ 服务端的准备

启动 NameServer 和 Broker

要验证 RocketMQ 的 Spring-Boot 客户端，首先要确保 RocketMQ 服务正确的下载并启动。可以参考 RocketMQ 主站的快速开始来进行操作。确保启动 NameServer 和 Broker 已经正确启动。

创建实例中所需要的 Topics

在执行启动命令的目录下执行下面的命令行操作：

```
bash bin/mqadmin updateTopic -c DefaultCluster -t string-topic
```

2. 编译 rocketmq-spring-boot-starter

目前的 spring-boot-starter 依赖还没有提交的 Maven 的中心库，用户使用前需要自行下载 git 源码，然后执行 mvn clean install 安装到本地仓库。

git clone https://github.com/apache/rocketmq-externals.git

cd rocketmq-spring-boot-starter

mvn clean install

3. 编写客户端代码

用户如果使用它，需要在消息的发布和消费客户端的 maven 配置文件 pom.xml 中添加如下的依赖：

```
<properties>   <spring-boot-starter-rocketmq-version>1.0.0-SNAPSHOT</spring-boot-starter-rocketmq-version>
</properties>

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

属性 spring-boot-starter-rocketmq-version 的取值为：1.0.0-SNAPSHOT， 这与上一步骤中执行安装到本地仓库的版本一致。

消息发送端的代码

发送端的配置文件 application.properties：

```
# 定义name-server地址
spring.rocketmq.name-server=localhost:9876
# 定义发布者组名
spring.rocketmq.producer.group=my-group1
# 定义要发送的topic
spring.rocketmq.topic=string-topic
```

发送端的 Java 代码：

```
import org.apache.rocketmq.spring.starter.core.RocketMQTemplate;
...

@SpringBootApplication
public class ProducerApplication implements CommandLineRunner {
    // 声明并引用RocketMQTemplate
    @Resource
    private RocketMQTemplate rocketMQTemplate;

    // 使用application.properties里定义的topic属性
    @Value("${spring.rocketmq.springTopic}")
    private String springTopic;

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

    public void run(String... args) throws Exception {
        // 以同步的方式发送字符串消息给指定的topic
        SendResult sendResult = rocketMQTemplate.syncSend(springTopic, "Hello, World!");
        // 打印发送结果信息
        System.out.printf("string-topic syncSend1 sendResult=%s %n", sendResult);
    }
}
```

消息消费端代码

消费端的配置文件 application.properties：

```
# 定义name-server地址
spring.rocketmq.name-server=localhost:9876
# 定义发布者组名
spring.rocketmq.consumer.group=my-customer-group1
# 定义要发送的topic
spring.rocketmq.topic=string-topic
```

消费端的 Java 代码：

```
@SpringBootApplication
public class ConsumerApplication {

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


// 声明消费消息的类，并在注解中指定，相关的消费信息
@Service
@RocketMQMessageListener(topic = "${spring.rocketmq.topic}", consumerGroup = "${spring.rocketmq.consumer.group}")
class StringConsumer implements RocketMQListener<String> {
    @Override
    public void onMessage(String message) {
        System.out.printf("------- StringConsumer received: %s %f", message);
    }
}
```

这里只是简单的介绍了使用 spring-boot 来编写最基本的消息发送和接收的代码，如果需要了解更多的调用方式，如: 异步发送，对象消息体，指定 tag 标签以及指定事务消息，请参看 github 的说明文档和详细的代码。我们后续还会对这些高级功能进行陆续的介绍。

方法二：Spring Cloud Stream 体系及原理介绍：spring-cloud-stream-binder-rocektmq

Photo by Med Badr Chemmaoui on Unsplash

Spring Cloud Stream 在 Spring Cloud 体系内用于构建高度可扩展的基于事件驱动的微服务，其目的是为了简化消息在 Spring Cloud 应用程序中的开发。

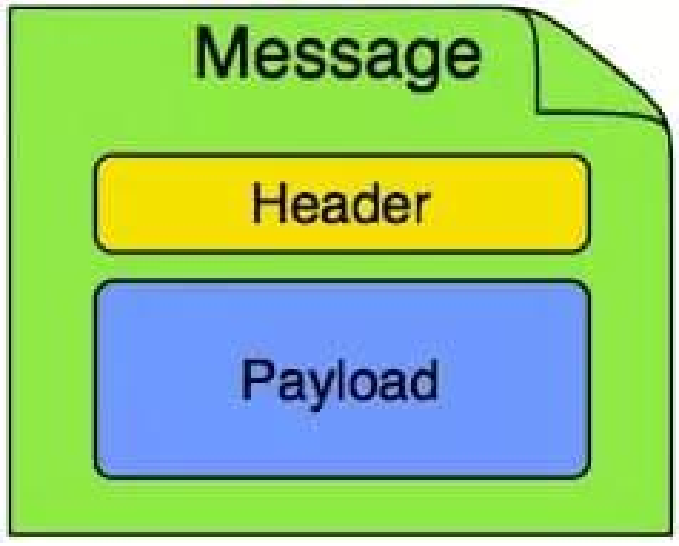
Spring Cloud Stream（后面以 SCS 代替 Spring Cloud Stream）本身内容很多，而且它还有很多外部的依赖，想要熟悉 SCS，必须要先了解 Spring Messaging 和 Spring Integration 这两个项目，接下来，文章将从围绕以下三点进行展开：

- 什么是 Spring Messaging?
- 什么是 Spring Integration?
- 什么是 SCS 体系及其原理?

一、Spring Messaging

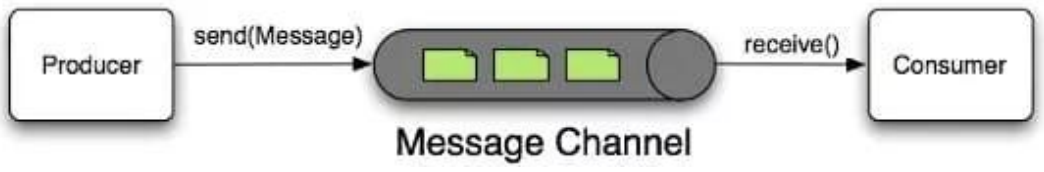
Spring Messaging 是 Spring Framework 中的一个模块，其作用就是统一消息的编程模型。

- 比如消息 `Messaging` 对应的模型就包括一个消息体 Payload 和消息头 Header:



```
package org.springframework.messaging;
public interface Message<T> {
   T getPayload();
MessageHeaders getHeaders();
}
```

- 消息通道 MessageChannel 用于接收消息，调用 send 方法可以将消息发送至该消息通道中 ：




```
@FunctionalInterface
public interface MessageChannel {
   long INDEFINITE_TIMEOUT = -1;
default boolean send(Message<?> message) {
    return send(message, INDEFINITE_TIMEOUT);
    }
boolean send(Message<?> message, long timeout);
}
```

消息通道里的消息如何被消费呢?

- 由消息通道的子接口可订阅的消息通道 SubscribableChannel 实现，被 MessageHandler 消息处理器所订阅:

```
public interface SubscribableChannel extends MessageChannel {
boolean subscribe(MessageHandler handler);
boolean unsubscribe(MessageHandler handler);
}
```

- 由 MessageHandler 真正地消费/处理消息:

```
@FunctionalInterface
public interface MessageHandler {
  void handleMessage(Message<?> message) throws MessagingException;
}
```

Spring Messaging 内部在消息模型的基础上衍生出了其它的一些功能，如：

1. 消息接收参数及返回值处理：消息接收参数处理器 HandlerMethodArgumentResolver 配合 @Header, @Payload 等注解使用；消息接收后的返回值处理器 HandlerMethodReturnValueHandler 配合 @SendTo 注解使用；
2. 消息体内容转换器 MessageConverter；
3. 统一抽象的消息发送模板 AbstractMessageSendingTemplate；
4. 消息通道拦截器 ChannelInterceptor；

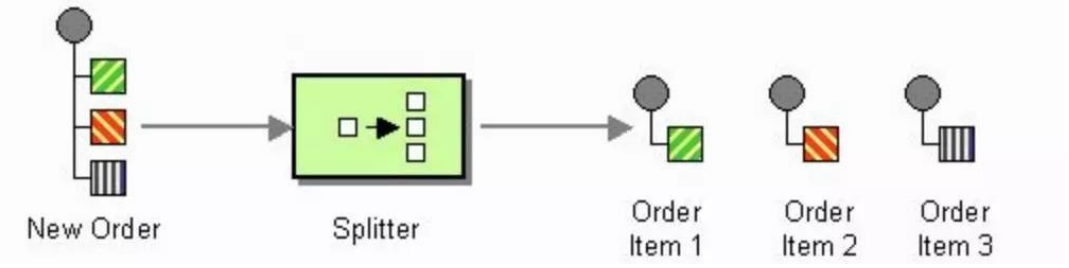
二、Spring Integration

Spring Integration 提供了 Spring 编程模型的扩展用来支持企业集成模式(Enterprise Integration Patterns)，是对 Spring Messaging 的扩展。

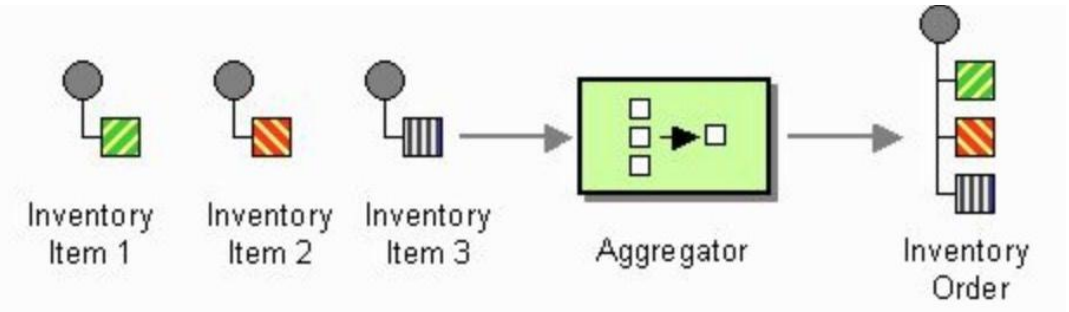
它提出了不少新的概念，包括消息路由 MessageRoute、消息分发 MessageDispatcher、消息过滤 Filter、消息转换 Transformer、消息聚合 Aggregator、消息分割 Splitter 等等。同时还提供了 MessageChannel 和 MessageHandler 的实现，分别包括 DirectChannel、ExecutorChannel、PublishSubscribeChannel 和 MessageFilter、ServiceActivatingHandler、MethodInvokingSplitter 等内容。

这里为大家介绍几种消息的处理方式：

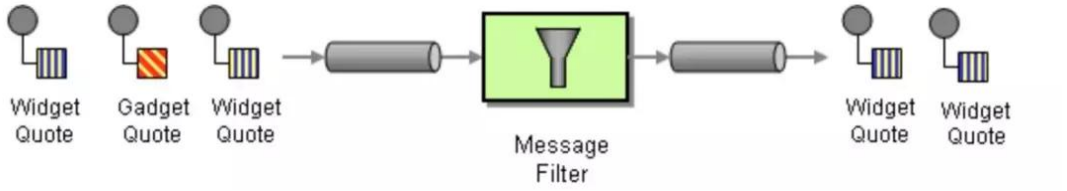
- 消息的分割：



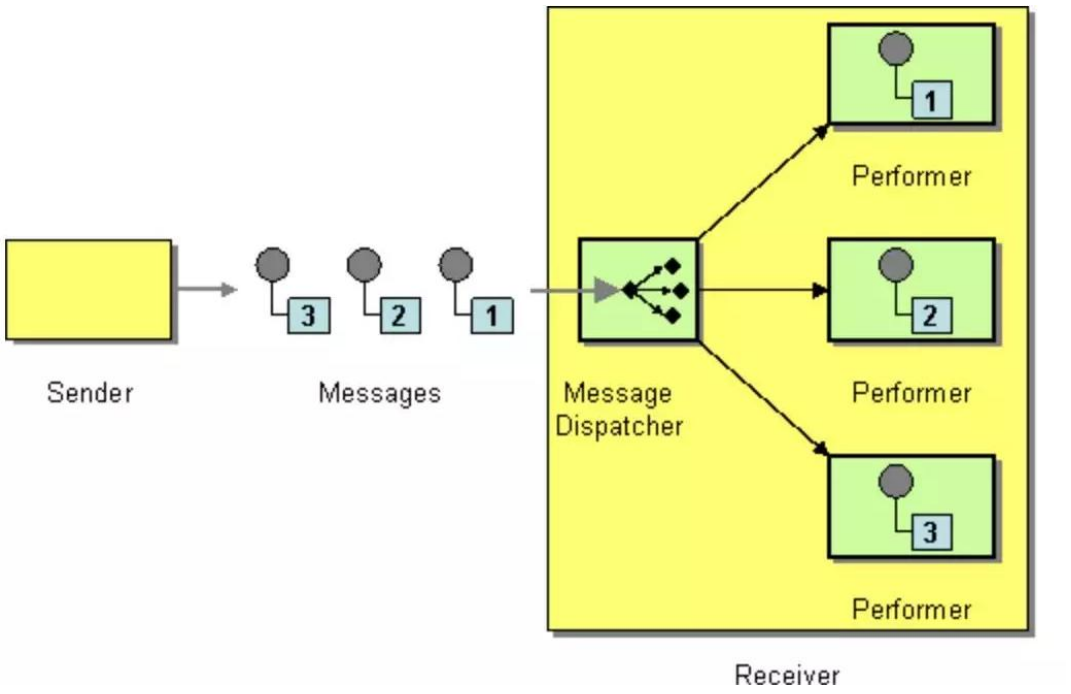
- 消息的聚合：



- 消息的过滤：



- 消息的分发：



接下来，我们以一个最简单的例子来尝试一下 Spring Integration:

这段代码解释为：

```
SubscribableChannel messageChannel =new DirectChannel(); // 1
messageChannel.subscribe(msg-> { // 2
System.out.println("receive: " +msg.getPayload());
});
messageChannel.send(MessageBuilder.withPayload("msgfrom alibaba").build()); // 3
```

1. 构造一个可订阅的消息通道 messageChannel；
2. 使用 MessageHandler 去消费这个消息通道里的消息；
3. 发送一条消息到这个消息通道，消息最终被消息通道里的 MessageHandler 所消费。

最后控制台打印出: receive: msg from alibaba；

DirectChannel 内部有个 UnicastingDispatcher 类型的消息分发器，会分发到对应的消息通道 MessageChannel 中，从名字也可以看出来，UnicastingDispatcher 是个单播的分发器，只能选择一个消息通道。那么如何选择呢？内部提供了 LoadBalancingStrategy 负载均衡策略，默认只有轮询的实现，可以进行扩展。

我们对上段代码做一点修改，使用多个 MessageHandler 去处理消息：

```
SubscribableChannel messageChannel = new DirectChannel();
messageChannel.subscribe(msg -> {
System.out.println("receive1: " + msg.getPayload());
});
messageChannel.subscribe(msg -> {
System.out.println("receive2: " + msg.getPayload());
```

```
});
messageChannel.send(MessageBuilder.withPayload("msg from alibaba").build());
messageChannel.send(MessageBuilder.withPayload("msg from alibaba").build());
```

由于 DirectChannel 内部的消息分发器是 UnicastingDispatcher 单播的方式，并且采用轮询的负载均衡策略，所以这里两次的消费分别对应这两个 MessageHandler。控制台打印出：

```
receive1: msg from alibaba
receive2: msg from alibaba
```

既然存在单播的消息分发器 UnicastingDispatcher，必然也会存在广播的消息分发器，那就是 BroadcastingDispatcher，它被 PublishSubscribeChannel 这个消息通道所使用。广播消息分发器会把消息分发给所有的 MessageHandler：

```
SubscribableChannel messageChannel = new PublishSubscribeChannel();
messageChannel.subscribe(msg -> {
   System.out.println("receive1: " + msg.getPayload());
});
messageChannel.subscribe(msg -> {
System.out.println("receive2: " + msg.getPayload());
});
messageChannel.send(MessageBuilder.withPayload("msg from alibaba").build());
messageChannel.send(MessageBuilder.withPayload("msg from alibaba").build());
```

发送两个消息，都被所有的 MessageHandler 所消费。控制台打印：

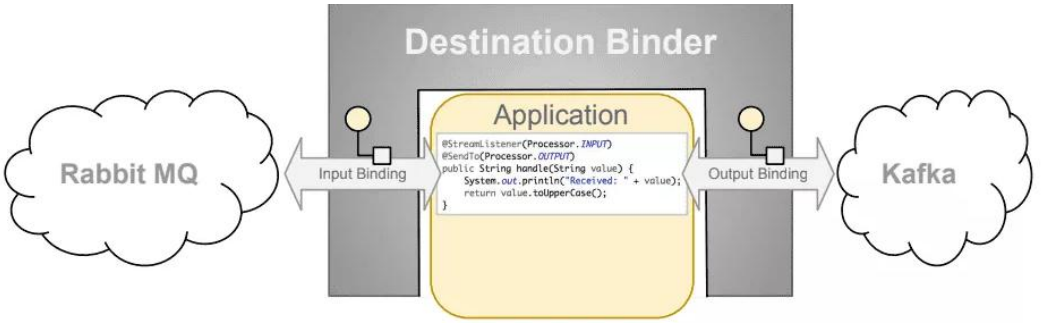
```
receive1: msg from alibaba
receive2: msg from alibaba
receive1: msg from alibaba
receive2: msg from alibaba
```

三、Spring Cloud Stream

SCS 与各模块之间的关系是：

- SCS 在 Spring Integration 的基础上进行了封装，提出了 Binder, Binding, @EnableBinding, @StreamListener 等概念;
- SCS 与 Spring Boot Actuator 整合，提供了 /bindings, /channels endpoint;
- SCS 与 Spring Boot Externalized Configuration 整合，提供了 BindingProperties, BinderProperties 等外部化配置类;
- SCS 增强了消息发送失败的和消费失败情况下的处理逻辑等功能;
- SCS 是 Spring Integration 的加强，同时与 Spring Boot 体系进行了融合，也是 Spring Cloud Bus 的基础。它屏蔽了底层消息中间件的实现细节，希望以统一的一套 API 来进行消息的发送/消费，底层消息中间件的实现细节由各消息中间件的 Binder 完成。

Binder 是提供与外部消息中间件集成的组件，为构造 Binding 提供了 2 个方法，分别是 bindConsumer 和 bindProducer ，它们分别用于构造生产者和消费者。目前官方的实现有 Rabbit Binder 和 Kafka Binder， Spring Cloud Alibaba 内部已经实现了 RocketMQ Binder。



从图中可以看出，Binding 是连接应用程序跟消息中间件的桥梁，用于消息的消费和生产。我们来看一个最简单的使用 RocketMQ Binder 的例子，然后分析一下它的底层处理原理：

- 启动类及消息的发送：

```
@SpringBootApplication
@EnableBinding({ Source.class, Sink.class }) // 1
public class SendAndReceiveApplication {

public static void main(String[] args) {
SpringApplication.run(SendAndReceiveApplication.class, args);
    }
@Bean // 2
 public CustomRunner customRunner() {
return new CustomRunner();
    }
public static class CustomRunner implements CommandLineRunner {
@Autowired
private Source source;
```

```
@Override
public void run(String... args) throws Exception {
 int count = 5;
for (int index = 1; index <= count; index++) {
    source.output().send(MessageBuilder.withPayload("msg-" + index).build()); // 3

      }
     }
  }
}
```

消息的接收：

```
@Service
public class StreamListenerReceiveService {
 @StreamListener(Sink.INPUT) // 4
public void receiveByStreamListener1(String receiveMsg) {
 System.out.println("receiveByStreamListener: " + receiveMsg);
  }
}
```

这段代码很简单，没有涉及到 RocketMQ 相关的代码，消息的发送和接收都是基于 SCS 体系完成的。如果想切换成 RabbitMQ 或 Kafka，只需修改配置文件即可，代码无需修改。

我们来分析下这段代码的原理:

1. @EnableBinding 对应的两个接口属性 Source 和 Sink 是 SCS 内部提供的。SCS 内部会基于 Source 和 Sink 构造 BindableProxyFactory，且对应的 output 和 input 方法返回的 MessageChannel 是 DirectChannel。output 和 input 方法修饰的注解对应的 value 是配置文件中 binding 的 name。

```
public interface Source {
String OUTPUT = "output";
@Output(Source.OUTPUT)
MessageChannel output();}public interface Sink {
String INPUT = "input";
@Input(Sink.INPUT)
SubscribableChannel input();
}
```

配置文件里 bindings 的 name 为 output 和 input，对应 Source 和 Sink 接口的方法上的注解里的 value：

```
spring.cloud.stream.bindings.output.destination=test-topic
spring.cloud.stream.bindings.output.content-type=text/plain
spring.cloud.stream.rocketmq.bindings.output.producer.group=demo-group
spring.cloud.stream.bindings.input.destination=test-topic
spring.cloud.stream.bindings.input.content-type=text/plain
spring.cloud.stream.bindings.input.group=test-group1
```

2. 构造 CommandLineRunner，程序启动的时候会执行 CustomRunner 的 run 方法。

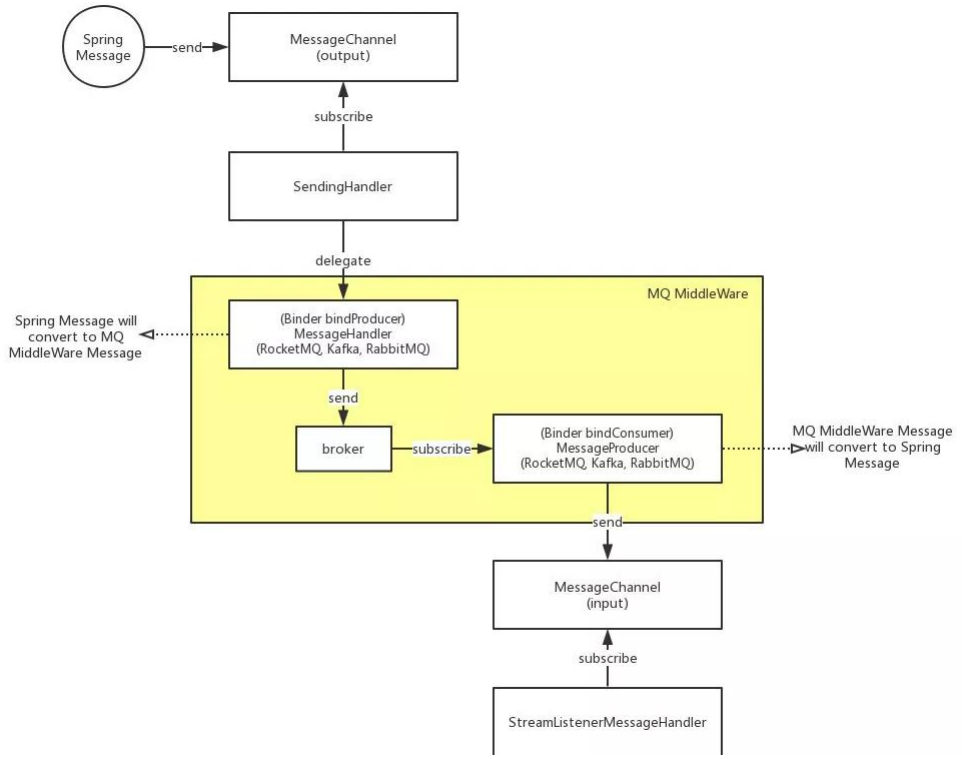
3. 调用 Source 接口里的 output 方法获取 DirectChannel，并发送消息到这个消息通道中。这里跟之前 Spring Integration 章节里的代码一致。

- Source 里的 output 发送消息到 DirectChannel 消息通道之后会被 AbstractMessageChannelBinder#SendingHandler 这个 MessageHandler 处理，然后它会委托给 AbstractMessageChannelBinder#createProducerMessageHandler 创建的 MessageHandler 处理(该方法由不同的消息中间件实现)；
- 不同的消息中间件对应的 AbstractMessageChannelBinder#createProducerMessageHandler 方法返回的 MessageHandler 内部会把 Spring Message 转换成对应中间件的 Message 模型并发送到对应中间件的 broker；

4. 使用 @StreamListener 进行消息的订阅。请注意，注解里的 Sink.input 对应的值是 "input"，会根据配置文件里 binding 对应的 name 为 input 的值进行配置：

- 不同的消息中间件对应的 AbstractMessageChannelBinder#createConsumerEndpoint 方法会使用 Consumer 订阅消息，订阅到消息后内部会把中间件对应的 Message 模型转换成 Spring Message；
- 消息转换之后会把 Spring Message 发送至 name 为 input 的消息通道中；
- @StreamListener 对应的 StreamListenerMessageHandler 订阅了 name 为 input 的消息通道，进行了消息的消费；

这个过程文字描述有点啰嗦，用一张图总结一下(黄色部分涉及到各消息中间件的 Binder 实现以及 MQ 基本的订阅发布功能)：



SCS 章节的最后，我们来看一段 SCS 关于消息的处理方式的一段代码：

```
@StreamListener(value = Sink.INPUT, condition = "headers['index']=='1'")
public void receiveByHeader(Message msg) {
 System.out.println("receive by headers['index']=='1': " + msg);
}
@StreamListener(value = Sink.INPUT, condition = "headers['index']=='9999'")
public void receivePerson(@Payload Person person) {
 System.out.println("receive Person: " + person);
}
@StreamListener(value = Sink.INPUT)
public void receiveAllMsg(String msg) {
```

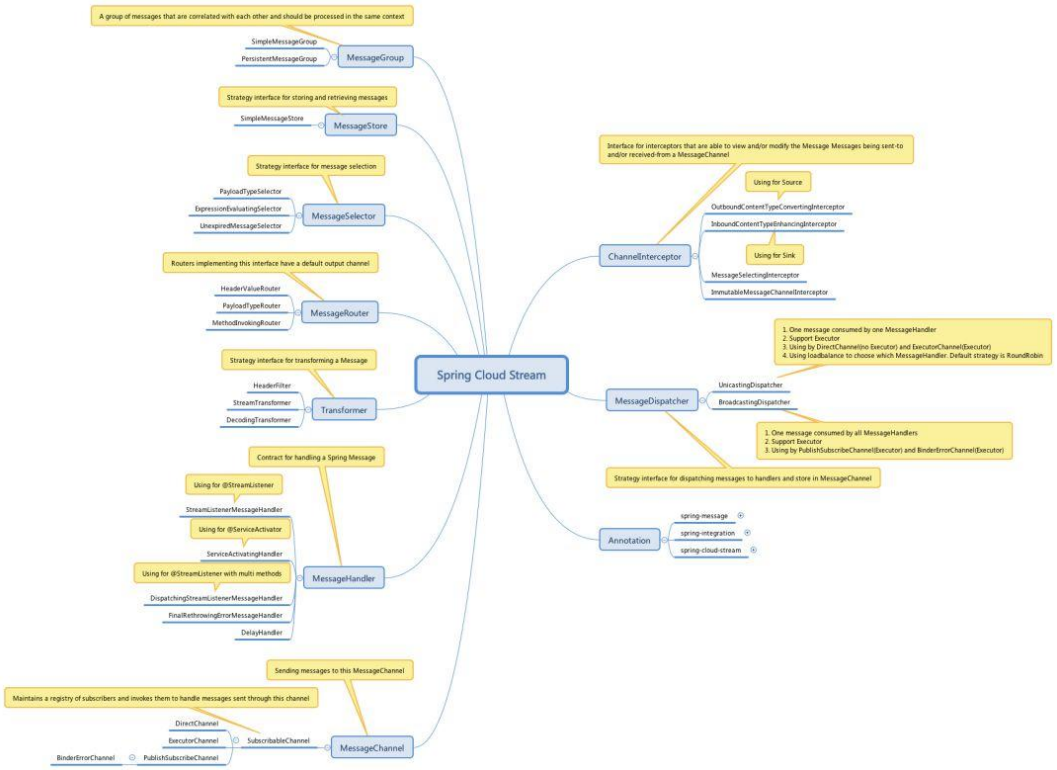
```
    System.out.println("receive allMsg by StreamListener. content: " + msg);
}
@StreamListener(value = Sink.INPUT)
public void receiveHeaderAndMsg(@Header("index") String index, Message msg) {
System.out.println("receive by HeaderAndMsg by StreamListener. content: " + msg);
}
```

有没有发现这段代码跟 Spring MVC Controller 中接收请求的代码很像？实际上他们的架构都是类似的，Spring MVC 对于 Controller 中参数和返回值的处理类分别是 org.springframework.web.method.support.HandlerMethodArgumentResolver、 org.springframework.web.method.support.HandlerMethodReturnValueHandler。

Spring Messaging 中对于参数和返回值的处理类之前也提到过，分别是 org.springframework.messaging.handler.invocation.HandlerMethodArgumentResolver、org.springframework.messaging.handler.invocation.HandlerMethodReturnValueHandler。

它们的类名一模一样，甚至内部的方法名也一样。

四、总结



上图是 SCS 体系相关类说明的总结，关于 SCS 以及 RocketMQ Binder 更多相关的示例，可以参考 RocketMQ Binder Demos（Demos 地址：点击“阅读原文”），包含了消息的聚合、分割、过滤；消息异常处理；消息标签、SQL 过滤；同步、异步消费等等。

下一篇文章，我们将分析消息总线(Spring Cloud Bus) 在 Spring Cloud 体系中的作用，并逐步展开，分析 Spring Cloud Alibaba 中的 RocketMQ Binder 是如何实现 Spring Cloud Stream 标准的。

欢迎大家使用钉钉扫描二维码加入 Spring Cloud Alibaba 开源讨论群：

扫码，加入我们！

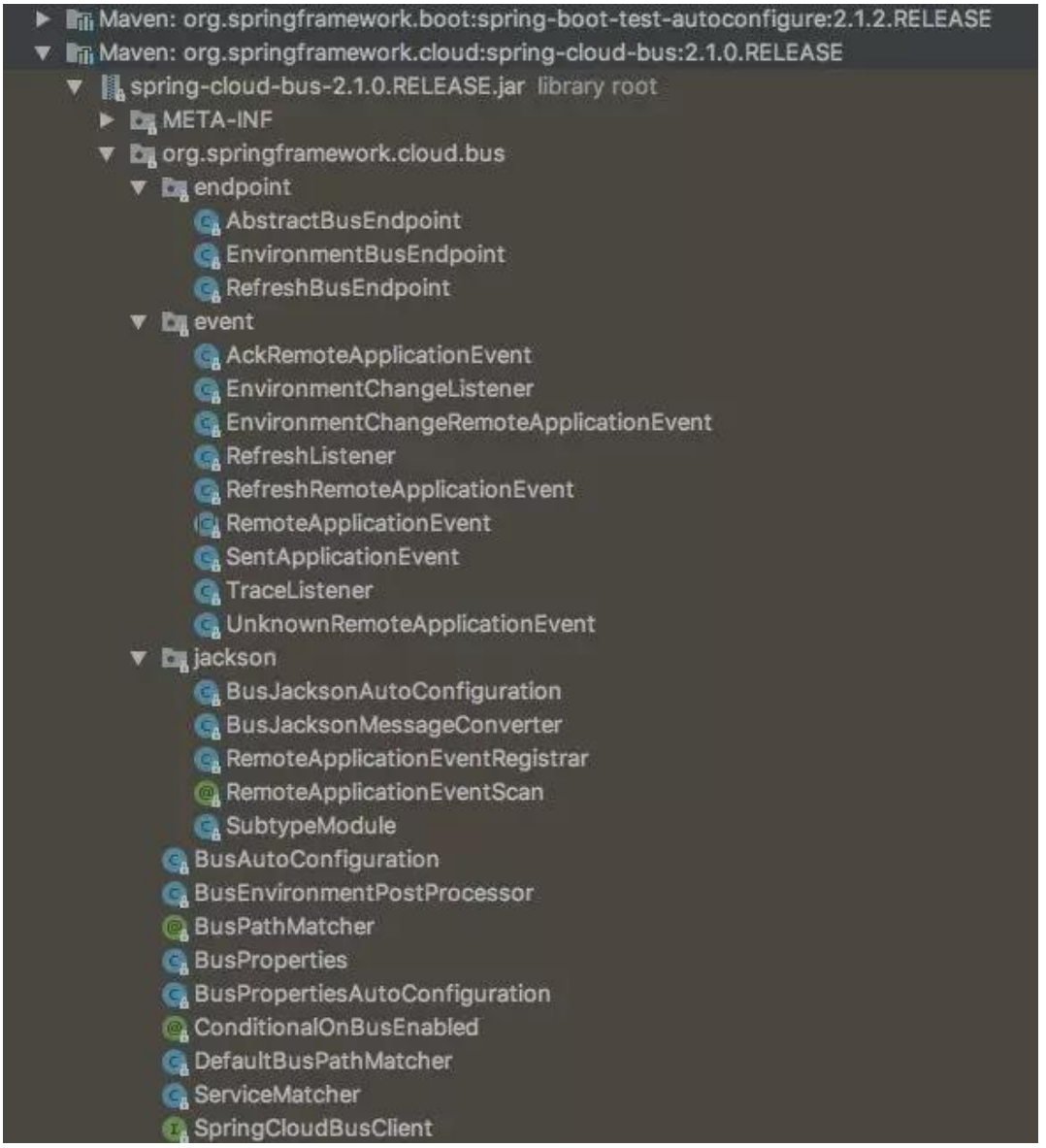
方法三：Spring Cloud Bus 消息总线介绍

作者：方剑（花名：洛夜）

本期我们来了解下 Spring Cloud 体系中的另外一个组件 Spring Cloud Bus (建议先熟悉 Spring Cloud Stream，不然无法理解 Spring Cloud Bus 内部的代码)。

Spring Cloud Bus 对自己的定位是 Spring Cloud 体系内的消息总线，使用 message broker 来连接分布式系统的所有节点。Bus 官方的 Reference 文档 比较简单，简单到连一张图都没有。

这是最新版的 Spring Cloud Bus 代码结构(代码量比较少)：



一、Bus 实例演示

在分析 Bus 的实现之前，我们先来看两个使用 Spring Cloud Bus 的简单例子。

1. 所有节点的配置新增

Bus 的例子比较简单，因为 Bus 的 AutoConfiguration 层都有了默认的配置，只需要引入消息中间件对应的 Spring Cloud Stream 以及 Spring Cloud Bus 依赖即可，之后所有启动的应用都会使用同一个 Topic 进行消息的接收和发送。

Bus 对应的 Demo 已经放到了 github 上（地址：https://github.com/fangjian0423/rocketmq-binder-demo/tree/master/rocketmq-bus-demo ），该 Demo 会模拟启动 5 个节点，只需要对其中任意的一个实例新增配置项，所有节点都会新增该配置项。

访问任意节点提供的 Controller 提供的获取配置的地址(key 为 `hangzhou`)：

```
curl -X GET 'http://localhost:10001/bus/env?key=hangzhou'
```

所有节点返回的结果都是 unknown，因为所有节点的配置中没有 `hangzhou` 这个 key。

Bus 内部提供了 `EnvironmentBusEndpoint` 这个 Endpoint 通过 message broker 用来新增/更新配置。

访问任意节点该 Endpoint 对应的 `url: /actuator/bus-env?name=hangzhou&value=alibaba` 进行配置项的新增(比如访问 node1 的 url):

```
curl -X POST 'http://localhost:10001/actuator/bus-env?name=hangzhou&value=alibaba' -H 'content-type: application/json'
```

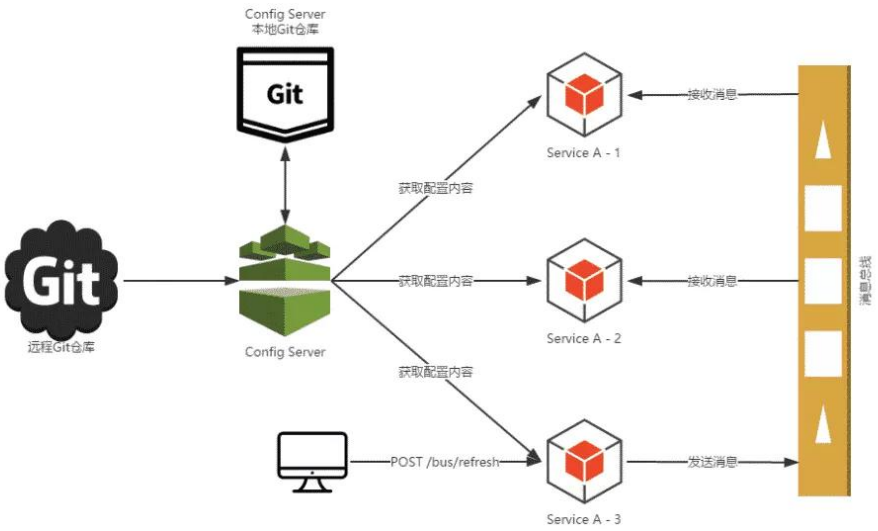
然后再次访问所有节点 `/bus/env` 获取配置：

```
$ curl -X GET 'http://localhost:10001/bus/env?key=hangzhou'
unknown%
~ ⌚
$ curl -X GET 'http://localhost:10002/bus/env?key=hangzhou'
unknown%
~ ⌚
$ curl -X GET 'http://localhost:10003/bus/env?key=hangzhou'
unknown%
~ ⌚
$ curl -X GET 'http://localhost:10004/bus/env?key=hangzhou'
unknown%
~ ⌚
$ curl -X GET 'http://localhost:10005/bus/env?key=hangzhou'
unknown%
~ ⌚
$ curl -X POST 'http://localhost:10001/actuator/bus-env?name=hangzhou&value=alibaba' -H 'content-type: application/json'
~ ⌚
$ curl -X GET 'http://localhost:10005/bus/env?key=hangzhou'
alibaba%
~ ⌚
$ curl -X GET 'http://localhost:10004/bus/env?key=hangzhou'
alibaba%
```

```
~ ⌚ $ curl -X GET 'http://localhost:10003/bus/env?key=hangzhou'
alibaba%
~ ⌚
$ curl -X GET 'http://localhost:10002/bus/env?key=hangzhou'
alibaba%
~ ⌚
$ curl -X GET 'http://localhost:10001/bus/env?key=hangzhou'
alibaba%
```

可以看到，所有节点都新增了一个 key 为 hangzhou 的配置，且对应的 value 是 alibaba。这个配置项是通过 Bus 提供的 EnvironmentBusEndpoint 完成的。

这里引用 程序猿 DD 画的一张图片，Spring Cloud Config 配合 Bus 完成所有节点配置的刷新来描述之前的实例(本文实例不是刷新，而是新增配置，但是流程是一样的)：



2. 部分节点的配置修改

比如在 node1 上指定 destination 为 rocketmq-bus-node2（node2 配置了 spring.cloud.bus.id 为 rocketmq-bus-node2:10002，可以匹配上）进行配置的修改：

```
curl -X POST 'http://localhost:10001/actuator/bus-env/rocketmq-bus-node2?name=hangzhou&value=xihu' -H 'content-type: application/json'
```

访问 /bus/env 获取配置(由于在 node1 上发送消息，Bus 也会对发送方的节点 node1 进行配置修改)：

```
~ ⌚
$ curl -X POST 'http://localhost:10001/actuator/bus-env/rocketmq-bus-node2?name=hangzhou&value=xihu' -H 'content-type: application/json'
~ ⌚
$ curl -X GET 'http://localhost:10005/bus/env?key=hangzhou'
alibaba%
~ ⌚
$ curl -X GET 'http://localhost:10004/bus/env?key=hangzhou'
alibaba%
~ ⌚
$ curl -X GET 'http://localhost:10003/bus/env?key=hangzhou'
alibaba%
~ ⌚
$ curl -X GET 'http://localhost:10002/bus/env?key=hangzhou'
xihu%
~ ⌚
$ curl -X GET 'http://localhost:10001/bus/env?key=hangzhou'
xihu%
```

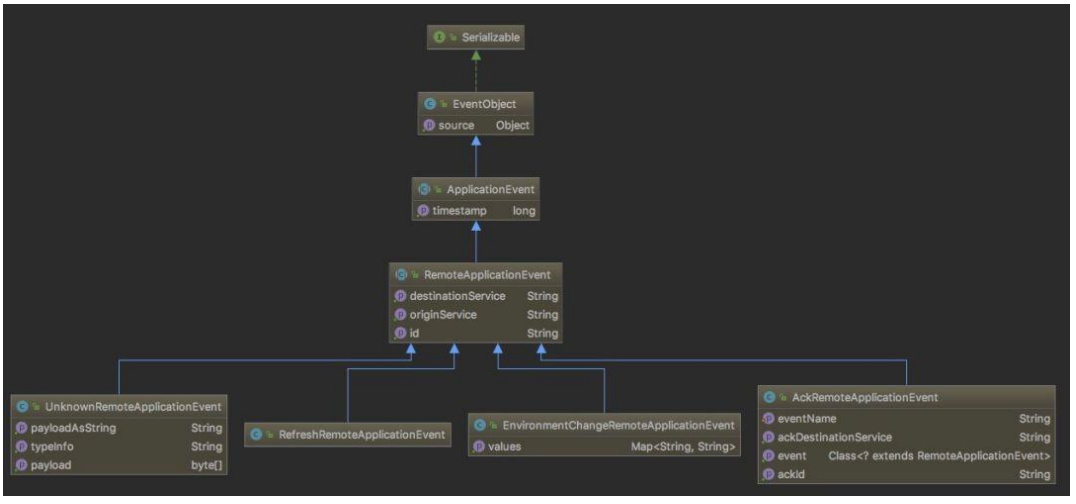
可以看到，只有 node1 和 node2 修改了配置，其余的 3 个节点配置未改变。

二、Bus 的实现

1. Bus 概念介绍

事件

Bus 中定义了远程事件 `RemoteApplicationEvent`，该事件继承了 Spring 的事件 `ApplicationEvent`，而且它目前有 4 个具体的实现：



- EnvironmentChangeRemoteApplicationEvent：远程环境变更事件。主要用于接收一个 Map 类型的数据并更新到 Spring 上下文中 `Environment` 中的事件。文中的实例就是使用这个事件并配合 `EnvironmentBusEndpoint` 和 `EnvironmentChangeListener` 完成的。

- AckRemoteApplicationEvent：远程确认事件。Bus 内部成功接收到远程事件后会发送回 AckRemoteApplicationEvent 确认事件进行确认。
- RefreshRemoteApplicationEvent：远程配置刷新事件。配合 @RefreshScope 以及所有的 @ConfigurationProperties 注解修饰的配置类的动态刷新。
- UnknownRemoteApplicationEvent：远程未知事件。Bus 内部消息体进行转换远程事件的时候如果发生异常会统一包装成该事件。

Bus 内部还存在一个非 RemoteApplicationEvent 事件 - SentApplicationEvent 消息发送事件，配合 Trace 进行远程消息发送的记录。

这些事件会配合 ApplicationListener 进行操作，比如 EnvironmentChangeRemoteApplicationEvent 配了 EnvironmentChangeListener 进行配置的新增/修改：

```
public class EnvironmentChangeListener
   implements ApplicationListener<EnvironmentChangeRemoteApplicationEvent> {
 private static Log log = LogFactory.getLog(EnvironmentChangeListener.class);
@Autowired
private EnvironmentManager env;
@Override
 public void onApplicationEvent(EnvironmentChangeRemoteApplicationEvent event) {
Map<String, String> values = event.getValues();
log.info("Received remote environment change request. Keys/values to update "
      + values);
for (Map.Entry<String, String> entry : values.entrySet()) {
env.setProperty(entry.getKey(), entry.getValue());
    }
 } }
```

收到其它节点发送来 EnvironmentChangeRemoteApplicationEvent 事件之后调用 EnvironmentManager#setProperty 进行配置的设置，该方法内部针对每一个配置项都会发送一个 EnvironmentChangeEvent 事件，然后被 ConfigurationPropertiesRebinder 所监听，进行 rebind 操作新增/更新配置。

Actuator Endpoint

Bus 内部暴露了 2 个 Endpoint，分别是 EnvironmentBusEndpoint 和 RefreshBusEndpoint，进行配置的新增/修改以及全局配置刷新。它们对应的 Endpoint id 即 url 是 bus-env 和 bus-refresh。

配置

Bus 对于消息的发送必定涉及到 Topic、Group 之类的信息，这些内容都被封装到了 BusProperties 中，其默认的配置前缀为 spring.cloud.bus，比如：

- spring.cloud.bus.refresh.enabled 用于开启/关闭全局刷新的 Listener。
- spring.cloud.bus.env.enabled 用于开启/关闭配置新增/修改的 Endpoint。
- spring.cloud.bus.ack.enabled 用于开启开启/关闭 AckRemoteApplicationEvent 事件的发送。
- spring.cloud.bus.trace.enabled 用于开启/关闭息记录 Trace 的 Listener。

消息发送涉及到的 Topic 默认用的是 springCloudBus，可以配置进行修改，Group 可以设置成广播模式或使用 UUID 配合 offset 为 lastest 的模式。

每个 Bus 应用都有一个对应的 Bus id，官方取值方式较复杂：

```
${vcap.application.name:${spring.application.name:application}}:${vcap.application.instance_index:${spring.application.index:${local.server.port:${server.port:0}}}}:${vcap.application.instance_id:${random.value}}
```

建议手动配置 Bus id，因为 Bus 远程事件中的 destination 会根据 Bus id 进行匹配：

```
spring.cloud.bus.id=${spring.application.name}-${server.port}
```

Bus 底层分析

Bus 的底层分析无非牵扯到这几个方面：

- 消息是如何发送的；
- 消息是如何接收的；
- destination 是如何匹配的；
- 远程事件收到后如何触发下一个 action；

`BusAutoConfiguration` 自动化配置类被 `@EnableBinding(SpringCloudBusClient.class)`所修饰。

`@EnableBinding` 的用法在上期文章 《干货｜Spring Cloud Stream 体系及原理介绍》 中已经说明，且它的 value 为 `SpringCloudBusClient.class`，会在

SpringCloudBusClient 中基于代理创建出 input 和 output 的 DirectChannel：

```
public interface SpringCloudBusClient {
String INPUT = "springCloudBusInput";
String OUTPUT = "springCloudBusOutput";
@Output(SpringCloudBusClient.OUTPUT)
MessageChannel springCloudBusOutput();
@Input(SpringCloudBusClient.INPUT)
SubscribableChannel springCloudBusInput();
}
```

springCloudBusInput 和 springCloudBusOutput 这两个 Binding 的属性可以通过配置文件进行修改(比如修改 topic)：

```
spring.cloud.stream.bindings:
springCloudBusInput:
destination: my-bus-topic
springCloudBusOutput:
destination: my-bus-topic
```

消息的接收的发送：

```
// BusAutoConfiguration
@EventListener(classes = RemoteApplicationEvent.class) // 1
public void acceptLocal(RemoteApplicationEvent event) {
if (this.serviceMatcher.isFromSelf(event)
&& !(event instanceof AckRemoteApplicationEvent)) { // 2
```

```
this.cloudBusOutboundChannel.send(MessageBuilder.withPayload(event).build()); // 3
 }
}
@StreamListener(SpringCloudBusClient.INPUT) // 4
public void acceptRemote(RemoteApplicationEvent event) {
 if (event instanceof AckRemoteApplicationEvent) {
 if (this.bus.getTrace().isEnabled()&& !this.serviceMatcher.isFromSelf(event)
 && this.applicationEventPublisher != null) { // 5
this.applicationEventPublisher.publishEvent(event);
      }
// If it's an ACK we are finished processing at this point
return;
}
 if (this.serviceMatcher.isForSelf(event)
&& this.applicationEventPublisher != null) { // 6
 if (!this.serviceMatcher.isFromSelf(event)) { // 7
this.applicationEventPublisher.publishEvent(event);
 }
 if (this.bus.getAck().isEnabled()) { // 8
 AckRemoteApplicationEvent ack = new AckRemoteApplicationEvent(this,
this.serviceMatcher.getServiceId(),
this.bus.getAck().getDestinationService(),
 event.getDestinationService(), event.getId(), event.getClass());
this.cloudBusOutboundChannel
 .send(MessageBuilder.withPayload(ack).build());
this.applicationEventPublisher.publishEvent(ack);
       }
    }
 if (this.bus.getTrace().isEnabled() && this.applicationEventPublisher != null) { // 9
```

```
    // We are set to register sent events so publish it for local consumption,
    // irrespective of the origin
     this.applicationEventPublisher.publishEvent(new SentApplicationEvent(this,
 event.getOriginService(), event.getDestinationService(),
       event.getId(),
       event.getClass()));
         }
       }
```

1. 利用 Spring 事件的监听机制监听本地所有的 RemoteApplicationEvent 远程事件(比如 bus-env 会在本地发送 EnvironmentChangeRemoteApplicationEvent 事件， bus-refresh 会在本地发送 RefreshRemoteApplicationEvent 事件，这些事件在这里都会被监听到)。

2. 判断本地接收到的事件不是 AckRemoteApplicationEvent 远程确认事件(不然会死循环，一直接收消息，发送消息...)以及该事件是应用自身发送出去的(事件发送方是应用自身)，如果都满足执行步骤 3。

2. 构造 Message 并将该远程事件作为 payload，然后使用 Spring Cloud Stream 构造的 Binding name 为 springCloudBusOutput 的 MessageChannel 将消息发送到 broker。

4. @StreamListener 注解消费 Spring Cloud Stream 构造的 Binding name 为 springCloudBusInput 的 MessageChannel，接收的消息为远程消息。

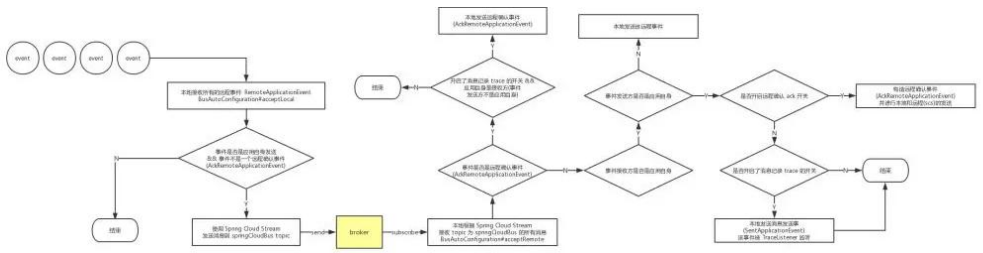
5. 如果该远程事件是 AckRemoteApplicationEvent 远程确认事件并且应用开启了消息追踪 trace 开关，同时该远程事件不是应用自身发送的(事件发送方不是应用自身，表示事件是其它应用发送过来的)，那么本地发送 AckRemoteApplicationEvent 远程确认事件表示应用确认收到了其它应用发送过来的远程事件，流程结束。

6. 如果该远程事件是其它应用发送给应用自身的(事件的接收方是应用自身)，那么进行步骤 7 和 8，否则执行步骤 9。

7. 该远程事件不是应用自身发送(事件发送方不是应用自身)的话，将该事件以本地的方式发送出去。应用自身一开始已经在本地被对应的消息接收方处理了，无需再次发送。

8. 如果开启了 AckRemoteApplicationEvent 远程确认事件的开关，构造 AckRemoteApplicationEvent 事件并在远程和本地都发送该事件(本地发送是因为步骤 5 没有进行本地 AckRemoteApplicationEvent 事件的发送，也就是自身应用对自身应用确认；远程发送是为了告诉其它应用，自身应用收到了消息)。

9. 如果开启了消息记录 Trace 的开关，本地构造并发送 SentApplicationEvent 事件。



bus-env 触发后所有节点的 EnvironmentChangeListener 监听到了配置的变化，控制台都会打印出以下信息：

```
o.s.c.b.event.EnvironmentChangeListener : Received remote environment change request. Keys/values to update {hangzhou=alibaba}
```

如果在本地监听远程确认事件 AckRemoteApplicationEvent，都会收到所有节点的信息，比如 node5 节点的控制台监听到的 AckRemoteApplicationEvent 事件如下：

```
ServiceId [rocketmq-bus-node5:10005] listeners on {"type":"AckRemoteApplicationEvent","timestamp":1554124670484,"originService":"rocketmq-bus-node5:10005","destinationService":"**","id":"375f0426-c24e-4904-bce1-5e09371fc9bc","ackId":"750d033f-356a-4aad-8cf0-3481ace8698c","ackDestinationService":"**","event":"org.springframework.cloud.bus.event.EnvironmentChangeRemoteApplicationEvent"}
ServiceId [rocketmq-bus-node5:10005] listeners on {"type":"AckRemoteApplicationEvent","timestamp":1554124670184,"originService":"rocketmq-bus-node1:10001","destinationService":"**","id":"91f06cf1-4bd9-4dd8-9526-9299a35bb7cc","ackId":"750d033f-356a-4aad-8cf0-3481ace8698c","ackDestinationService":"**","event":"org.springframework.cloud.bus.event.EnvironmentChangeRemoteApplicationEvent"}
ServiceId [rocketmq-bus-node5:10005] listeners on {"type":"AckRemoteApplicationEvent","timestamp":1554124670402,"originService":"rocketmq-bus-node2:10002","destinationService":"**","id":"7df3963c-7c3e-4549-9a22-a23fa90a6b85","ackId":"750d033f-356a-4aad-8cf0-3481ace8698c","ackDestinationService":"**","event":"org.springframework.cloud.bus.event.EnvironmentChangeRemoteApplicationEvent"}
ServiceId [rocketmq-bus-node5:10005] listeners on {"type":"AckRemoteApplicationEvent","timestamp":1554124670406,"originService":"rocketmq-bus-node3:10003","destinationService":"**","id":"728b45ee-5e26-46c2-af1a-e8d1571e5d3a","ackId":"750d033f-356a-4aad-8cf0-3481ace8698c",
```

"ackDestinationService":"**","event":"org.springframework.cloud.bus.event.EnvironmentChangeRemoteApplicationEvent"}

ServiceId [rocketmq-bus-node5:10005] listeners on {"type":"AckRemoteApplicationEvent","timestamp":1554124670427,"originService":"rocketmq-bus-node4:10004","destinationService":"**","id":"1812fd6d-6f98-4e5b-a38a-4b11aee08aeb","ackId":"750d033f-356a-4aad-8cf0-3481ace8698c","ackDestinationService":"**","event":"org.springframework.cloud.bus.event.EnvironmentChangeRemoteApplicationEvent"}

那么回到本章节开头提到的 4 个问题，我们分别做一下解答：

- 消息是如何发送的：在 `BusAutoConfiguration#acceptLocal` 方法中通过 Spring Cloud Stream 发送事件到 `springCloudBus` topic 中。
- 消息是如何接收的：在 `BusAutoConfiguration#acceptRemote` 方法中通过 Spring Cloud Stream 接收 `springCloudBus` topic 的消息。
- destination 是如何匹配的：在 `BusAutoConfiguration#acceptRemote` 方法中接收远程事件方法里对 destination 进行匹配。
- 远程事件收到后如何触发下一个 action：Bus 内部通过 Spring 的事件机制接收本地的 `RemoteApplicationEvent` 具体的实现事件再做下一步的动作(比如 `EnvironmentChangeListener` 接收了 `EnvironmentChangeRemoteApplicationEvent` 事件， `RefreshListener` 接收了 `RefreshRemoteApplicationEvent` 事件)。

三、总结

Spring Cloud Bus 自身内容还是比较少的，不过还是需要提前了解 Spring Cloud Stream 体系以及 Spring 自身的事件机制，在此基础上，才能更好地理解 Spring Cloud Bus 对本地事件和远程事件的处理逻辑。

目前 Bus 内置的远程事件较少，大多数为配置相关的事件，我们可以继承 `RemoteApplicationEvent` 并配合 `@RemoteApplicationEventScan` 注解构建自身的微服务消息体系。

欢迎加入 Spring Cloud Alibaba 开源社群，和我们一起交流与探索 Spring Cloud Alibaba 的实践以及发展路径：

Spring Cloud Alibaba 开源钉钉群：

RocketMQ 中国社区钉钉群

欢迎各位开发者进群交流、勘误

微信扫码关注

“阿里巴巴云原生公众号”

阿里云开发者“藏经阁”

海量电子书免费下载