Spring Cloud Stream：消息驱动的微服务

- Spring Cloud Stream 是一个用来为微服务应用构建消息驱动能力的框架
- 它可以基于Spring Boot 来创建独立的、 可用于生产的 Spring 应用程序
- 它通过使用 Spring Integration来连接消息代理中间件以实现消息事件驱动
- Spring Cloud Stream为一些供应商的消息中间件产品提供了个性化的自动化配置实现，并且引入了发布斗丁阅、 消费组以及分区这三个核心概念
- 简单地说， Spring Cloud Stream 本质上就是整合了 Spring Boot 和 Spring Integration, 实现了一套轻量级的消息驱动的微服务框架
- 通过使用SpringCloudStream,可以有效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
- 由于SpringCloudStream基于SpringBoot实现，所以它秉承了Spring Boot 的优点，自动化配置的功能可帮助快速上手使用，目前只支持RabbitMQ和Kafka消息中间件的自动化配置

使用

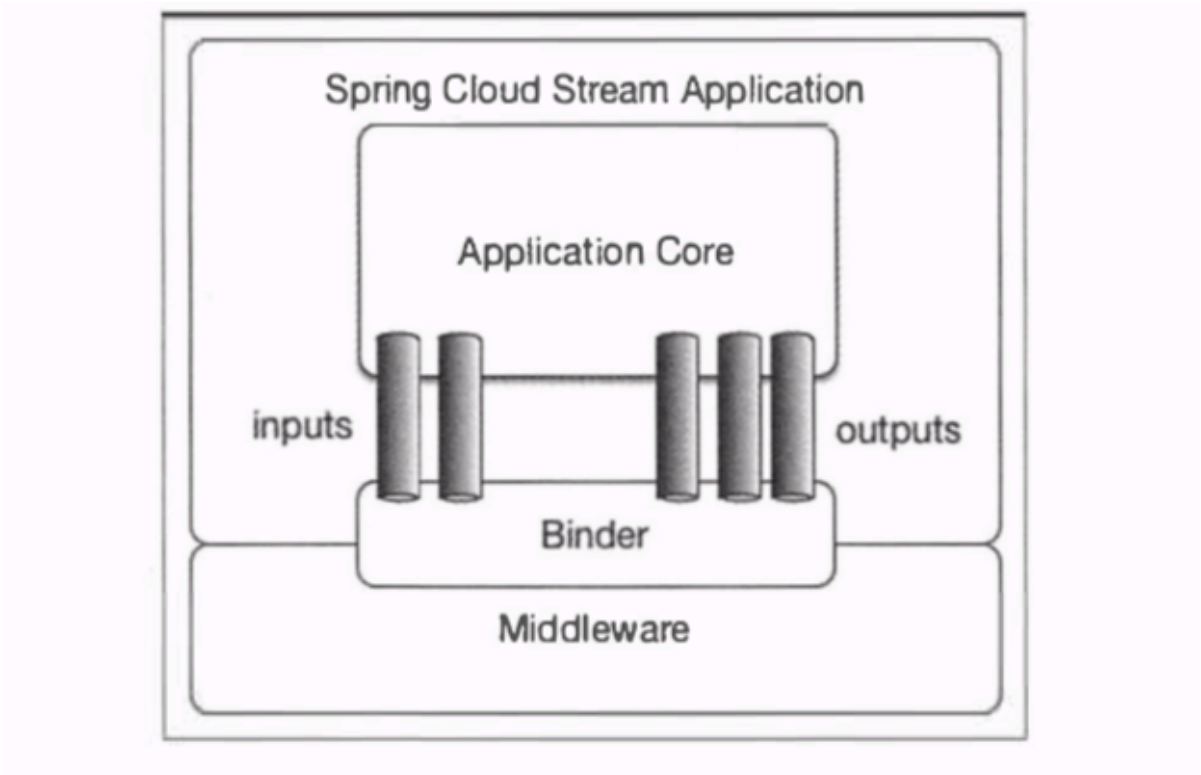
- 通过使用消息中间件 RabbitMQ 来接收消息并将消息打印到日志中
- `stream-hello`工程，引入`web`， `stream-rabbit`依赖（Spring Cloud Stream对RabbitMQ支待的封装， 其中包含了对 RabbitMQ的 自动化配置等内容，等价于spring-cloud-stream-binder-rabbit 依赖 ）
- 主类不需要加额外注解

```
//用于接收来自 RabbitMQ 消息的消费者
//EnableBinding用来指定一个或多个定义了@Input或@Output注解的接口，以此实现对消息通道（Channel）的绑定
//通过@EnableBinding(Sink.class)绑定了Sink接口，该接口是Spring Cloud Stream中默认实现的对输入消息通道绑定的定义，它通过@Input注解绑定了一个名为input的通道
//除了Sink之外，Spring Cloud Stream还默认实现了绑定output通道的Source接口，还有结合了Sink和 Source的Processor接口，实际使用时也可以自已通过@Input和@Output注解来定义绑定消息通道的接口
@EnableBinding({Sink.class, SinkSender.class})    //指定多个接口来绑定消息通道
public class SinkReceiver {
    private static Logger logger =
LoggerFactory.getLogger(SinkReceiver.class);
    // @StreamListener主要定义在方法上，作用是将被修饰的方法注册为消息中间件上数据流的事件监听器， 注解中的属性值对应了监听的消息通道名
    //通过@StreamListener(Sink.INPUT)注解将receive方法注册为input消息通道的监听处理器，所以当在RabbitMQ的控制页面中发布消息的时候， receive方法会做出对应的响应动作
    @StreamListener(Sink.INPUT)
    public void receive(User user) {
        logger.info("Received: " + user);
    }
}
```

- 分别启动 RabbitMQ 以及该Spring Boot 应用
- RabbitMQ 的控制台页面中Connection中声明了一个队列，并通过RabbitMessageChannelBinder将自己绑定为它的消费者
- RabbitMQ的控制台中进入该队列的管理页面，通过Publish message功能来发送一条消息到该队列中
- 应用控制台中输出的内容就是SinkReceiver中的receive方法定义的， 而输出的具体内容则来自消息队列中获取的对象

核心概念

- 在Spring CloudStream中是如何通过定义一些基础概念来对各种不同的消息中间件做抽象的



- Spring CloudStream应用模型的结构图
    - SpringCloudStream构建的应用程序与消息中间件之间是通过绑定器 Binder相关联的， 绑定器 对于应用程序而言起到了隔离作用， 它使得不同消息中间件的实现细节对应用程序来说是透明的
    - 所以对于每一 个Spring CloudStream的应用程序来说， 它不需要知晓消息中间件的通信细节， 它只需知道Binder 对应程序提供的消息通道（Channel）来使用消息中间件来实现业务逻辑即可， 而这个抽象概念
    - 如图所示， 在应用程序和Binder之间定义 了两条输入通道和三条输出通道来传递消息， 而绑定器则是作为这些通道和消息中间件之间的桥梁进行通信
- 绑定器
    - 通过定义绑定器作为中间层， 完美地实现了应用程序与消息中间件细节之间的隔离。 通过向应用程序暴露统一的Cahnnel通道， 使得应用程序不需要再考虑各种不同的消息中间 件的实现。 当需要升级消息中间件， 或是更换其他消息中间件产品时， 我们要做的就是更换它们对应的binder绑定器而不需要修改任何SpringBoot的应用逻辑
    - 目前版本的Spring CloudStream为主流的消息中间件产品RabbitMQ和Kafka提供了默认的

Binder 实现
    - Spring CloudStream还实现了一 个专门用于测试的TestSupportBinder, 开发者可以直接使用它来对通道的接收内容进行可靠的测试断言
    - 快速入门示例中， 并没有使用application.properties或是application.yml来做任何属性设置。 那是因为它也秉承了Spring Boot的设计理念，提供了对RabbitMQ默认的自动化配置。当然，我们也可以通过Spring Boot 应用支待的任何方式来修改这些配置， 比如， 通过应用程序参数、 环境变量、 application.properties或是application.yml配置文件等
- 发布-订阅模式
    - SpringCloudStream中的消息通信方式遵循了发布-订阅模式， 当一条消息被投递到消息中间件之后，它会通过共享的Topic主题进行广播，消息消费者在订阅的主题中收到它并触发自身的业务逻辑处理
    - 这里所 提到的Topic 主题是SpringCloudStream中的一个抽象概念， 用来代表发布共享消息给消费者的地方。 在不同的消息中间件中，Topic可能对 应不同的概念， 比如，在RabbitMQ中，它对应Exchange, 而在Kakfa中则对应Kafka中的Topic
    - 通过RabbitMQ的Channel发布消息给我们编写的应用程序消费， 而实际上SpringCloudStream应用启动的时候， 在RabbitMQ的Exchange中也创建了 一 个名为 input的Exchange交换器， 由于binder的隔离作用， 应用程序并无法感知它的存在，应用程序只知道自己指向Binder的输入或是输出通道
    - 启动的两个应用程序分别是 “ 订阅者-1"和 ” 订阅者-2", 它们都建立了一条输入通道绑定到同一个Topic(RabbitMQ的Exchange)上。 当该Topic中有消息发布进来后， 连接到该Topic上的所有订阅者可以收到该消息并根据自身的需求进行消费操作
    - 相对于点对点队列实现的消息通信来说，Spring CloudStream采用的发布-订阅模式可 以有效降低消息生产者与消费者之间的耦合。 当需要对同一类消息增加一种处理方式时，只需要增加 一 个应用程序并将输入通道绑定到既有的Topic中就可以实现功能的扩展，而不需要改变原来已经实现的任何内容
- 消费组
    - 通过引入消费组的概念， 能够在多实例的清况下， 保障每个消息只被组内的一个实例消费。
    - 在现实的微服务架构中， 我们的每一个微服务应用为了实现高可用和负载均衡， 实际上都会部署多个实例。在很多情况下， 消息生产者发送消息给某个具体微服务时， 只希望被消费一次。启动两个应用的例子， 虽然它们同属一个应用， 但是这个消息出现了被重复消费两次的情况
    - 如果在同一个主题上的应用需要启动多个实例的时候，我们可以通过 spring.cloud.stream.bindings.input.group属性为应用指定一 个组名，这样这个应用的多个实例在接收到消息的时候， 只会有一个成员真正收到消息并进行处理

```
//先实现一个消费者应用SinkReceiver,实现greetings主题上的输入通道绑定
@EnableBinding(value = {Sink.class})
public class SinkReceiver {
  private static Logger logger = LoggerFac七
ory.getLogger(SinkReceiver.class);
  @StreamListener(Sink.INPUT)
  public void receive(User user) {
    logger.info("Received: " + user) ;
  }
}
```

```
# 为了将SinkReceiver的输入通道目标设置为greetings主题，以及将该服务的实例设置为同一个消费组，可做如下设置：
# 指定了该应用 实例都属 于Service-A消组费
spring.cloud.stream.bindings.input.group=Service-A
# 指定了输入通道对应的主题名
spring.cloud.stream.bindings.input.destination= greetings
```

```
// 消息生产者应用SinkSender
@EnableBinding(value = {Source.class})
public class SinkSender {
  private static Logger logger =
LoggerFactory.getLogger(SinkSender.class);
  @Bean
  @InboundChannelAdapter(value = Source.OUTPUT, poller =
@Poller(fixedDelay = "2000"))
  public MessageSource<String> timerMessageSource() {
    return ()-> new GenericMessage<>("{\"name\":\"didi\",
\"age\":30)");
  }
}
```

```
# 为消息生产者SinkSender做一些设置，让它的输出通道绑定目标也指向greetings主题
spring.cloud.stream.bindings.output.destination= greetings
```

        - 消费者启动多个实例。通过控制台，可以发现，每个生产者发出的消息会被启动的消费者以轮询的方式进行接收和输出
    - 默认情况下，当没有为应用指定消费组的时候， Spring Cloud Stream会为其分配一个独立的匿名消费组。 所以， 如果同一 主题下的所有应用都没有被指定消费组的时候， 当有消息发布之后， 所有的应用都会对其进行消费， 因为它们各自都属于一个独立的组
    - 大部分情况下， 我们在创建Spring Cloud Stream应用的时候， 建议最好为其指定一个消费组，以防止对消息的重复处理， 除非该行为需要这样做(比如刷新所有实例的配置等)
- 消息分区
    - 消费组无法控制消息具体被哪个实例消费。 也就是说，对于同一条消息，它多次到达之后可能是由不同的实例进行消费的
    - 分区：当生产者将消息数据发送给多个消费者实例时， 保证拥有共同特征的消息数据始终是由同一 个消费者实例接收和处理
    - Spring Cloud Stream 为分区提供了通用的抽象实现， 用来在消息中间件的上层实现分区处理所以它对于消息中间件自身是否实现了消息分区并不关心， 这使得 Spring CloudStream 为不具备分区功能的消息中间件也增加了分区功能扩展
    - 在Spring CloudStream中实现消息分区非常简单，对消费组示例做一些配置修改就能实现

```
# 在消费者应用SinkReceiver中， 对配置文件做一些修改
# 要为消费者指定不同的实例索引号，这样当同一个消息被发送给消费组时， 可以发现只有一个消费实例在接收和处理这些相同的消息
spring.cloud.stream.bindings.input.group=Service-A
spring.cloud.stream.bindings.input.destination=greetings
# 开启消费者分区功能
spring.cloud.stream.bindings.input.consumer.partitioned=true
# 指定了当前消费者的总实例数量
spring.cloud.stream.instanceCount=2
# 设置当前实例的索引号，从0开始，最大值为spring.cloud.stream.instanceCount参数-1。试验的时候需要启动多个实例，可以通过运行参数来为不同实例设置不同的索引值
spring.cloud.stream.instanceindex=0
```

```
# 在生产者应用SinkSender中， 对配置文件也做一些修改
spring.cloud.stream.bindings.output.destination=greetings
# 指定了分区键的表达式规则，可以根据实际的输出消息规则配置SpEL来生成合适的分区键
spring.cloud.stream.bindings.output.producer.partitionKeyExpression=payload
# 指定了消息分区的数量
spring.cloud.stream.bindings.output.producer.partitionCount=2
```

使用详解

- 开启绑定功能
  - 在 Spring Cloud Stream 中，通过 @EnableBinding 注解来为应用启动消息驱动的功能
  - 它自身包含了@Configuration 注解，所以用它注解的类也会成为 Spring 的基本配置类。 另外该注解还通过@Impot加载了 Spring CloudStream 运行需要的几个基础配置类
  - @EnableBinding注解只有一个唯一的属性：value。 由于该注解@Import了BindingBeansRegistrar实现，所以在加载了基础配置内容之后，它会回调来读取value中的类，以创建消息通道的绑定。 另外， 由于value是 一个Class类型的数组， 所以我们可以通过value属性一 次性指定多个关于消息通道的配置
- 绑定消息通道
  - 在Spring CloudSteam中，可以在接口中通过input和@Output注解来定义消息通道， 而用于定义绑定消息通道的接口则可以被@EnableBinding注解的value参数来指定， 从而在应用启动的时候实现对定义消息通道的绑定
  - 可以使用Sink接口绑定的消息通道。Sink接口是SpringCloudSteam提供的 一个默认实现，除此之外还有Source和Processor
    - Sink和Source中分别通过@Input和@Output 注解定义了输入通道和输出通道，而Processor通过继承Source和Sink的方式同时定义了一个输入通道和一个输出通道
  - 另外，@Input和@Output 注解都还有一个value属性，该属性可以用来设置消息通道的名称，这里Sink和Source中指定的消息通道名称分别为input和output。如果我们直接使用这两个注解而没有指定具体的 value值，将默认使用方法名作为消息通道的名称

    - 最后，需要注意一点，当我们定义输出通道的时候，需要返回MessageChannel接口对象，该接口定义了向消息通道发送消息的方法; 而定义输入通道时，需要返回SubscribableChannel接口对象，该接口继承自MessageChannel接口，它定义了维护消息通道订阅者的方法

- 注入接口绑定
    - 在完成了消息通道绑定的定义之后，Spring CloudStream会为其创建具体的实例，而开发者只需要通过注入的方式来获取这些实例并直接使用即可
    - 举例：通过注入的方式实现一个消息生成者，向 input消息通道发送数据

```
//创建一个将Input消息通道作为 输出通道的接口
public interface SinkSender {
  @Output(Sink.INPUT) MessageChannel output();
}
//在@EnableBinding注解中增加对SinkSender接口的指定，使SpringCloudStream能创建出对应的实例
@EnableBinding{value = {Sink.class, SinkSender.class})
public class SinkReceiver {
  private static Logger logger =
LoggerFactory.getLogger(SinkReceiver.class);
  @StreamListener(Sink.INPUT)
  public void receive(Object payload) {
    logger.info("Received: " + payload);
  }
}
//创建一个单元测试类，通过@Autowired注解注入SinkSender的实例， 并在测试用例中调用它的发送消息方法
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = HelloApplication.class)
@WebAppConfiguration
public class HelloApplicationTests {
  @Autowired
  private SinkSender sinkSender;
  @Test
  public void contextLoads() {
    sinkSender.output().send(MessageBuilder.wi七hPayload("From
SinkSender").build());
  }
}
```

- 注入消息通道
    - 由于 Spring Cloud Stream 会根据绑定接口中的@input 和@Output 注解来创建消息通道实例， 所以我们也可以通过直接注入的方式来使用消息通道对象

```
//注入上面例子中 SinkSender 接口中定义的名为 input 的消息输入通道
@Autowired
private MessageChannel input;
@Test
public void contextLoads() {
        input.send(MessageBuilder.withPayload("From
MessageChannel").build());
}
//完成了与之前通过注入绑定接口 SinkSender 方式实现的测试用例相同的操作。 因为在通过注入绑定接口实现时， sinkSender.output()方法实际获得的就是SinkSender接口中定义的MessageChannel实例，只是在这里我们直接通过注入的方式来实现了而已
```

- 在一个微服务应用中可能会创建多个不同名的 MessageChannel 实例， 这样通过 @Autowired 注入 时， 要注意参数命名需要与通道同名才能被正确注入， 或者也可以使用 @Qualifier 注解来特别指定具体实例的名称， 该名称需要与定义 MessageChannel 的 @Output 中的value 参数一致， 这样才能被正确注入
  - 在一个接口中定义了两个输出通道， 分别命名为 Output-1和Output-2, 当要使用 Output-1 的时候， 可以通过@Qualifier("Output-1") 来指定这个具体的实例来注入使用
  - @Autowired @Qualifier("Output-1") private MessageChannel output;

- 消息生产与消费
  - 由于 Spring Cloud Stream 是基于 Spring Integration 构建起来的， 所以在使用 SpringCloud Stream 构建消息驱动服务的时候，完全可以使用 Spring Integration 的原生注解来实现各种业务需求。 同时， 为了简化面向消息的编程模型， Spring Cloud Stream 还提供了@StreamListener 注解对输入通道的处理做了进一步优化
  - Spring Integration 原生支持
    - 通过 Spring Integration 原生的@ServiceActivator和@InboundChannelAdapter 注解来尝试实现向名为input 的消息通道发送信息

```
// 应用1
// SinkReceiver 类属于消息消费者实现
// 使用原生的 @ServiceActivator 注解替换了@StreamListener, 实现对Sink.INPUT通道的监听处理， 而该通道绑定了名为input的主题
@EnableBinding(value = {Sink.class})
public class SinkReceiver {
        private static Logger logger =
LoggerFactory.getLogger(SinkReceiver.class);
        @ServiceActivator(inputChannel=Sink.INPUT)
  public void receive(Object payload) {
                logger.info("Received: " + payload);
  }
}
// 应用2
//SinkSender类属于消息生产者实现，它在内部定义了SinkOutput接口来将输出通道绑定到名为input的主题中。由于SinkSender和SinkReceiver共用一个主题，所以它们构成了一组生产者与消费者
@EnableBinding(value = {SinkSender.SinkOutput.class})
```

```
public class SinkSender {
        private static Logger logger =
LoggerFactory.getLogger(SinkSender.class);
  @Bean
        @InboundChannelAdapter(value = SinkOutput.OUTPUT, poller =
@Poller(fixedDelay = "2000"))
  //在SinkSender中还创建了用于生产消息的timerMessageSource方法，该方法会将当前时间作为消息返回。而InboundChannelAdapter 注解定义了该方法是对SinkOutput.OUTPUT通道的输出绑定，同时使用poller参数将该方法设置为轮询执行，这里定义为2000毫秒，所以它会以2秒的频率向 SinkOutput.OUTPUT 通道输出当前时间
        public MessageSource<Date> timerMessageSource() {
                return () -> new GenericMessage<>(new Date());
  }

  //还可以通过@Transformer 注解对指定通道的消息进行转换
  @Transformer(inputChannel = Sink.INPUT, outputChannel =
Sink.INPUT)
  public Object transform(Date message) {
                return new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").format(message);
  }

  public interface SinkOutput {
                String OUTPUT = "input";
                @Output(SinkOutput.OUTPUT)
    MessageChannel output();
  }
}
```

- @StrearnListener 详解
  - 通过该注解修饰的方法， Spring Cloud Steam 会将其注册为输入消息通道的监听器 。当输入 消息通道中有消息到达的时候， 会立即触发该注解修饰方法的处理逻辑对消息进行消费
- 消息转换
  - 都实现了对输入消息通道的监听，但是@StearnListener 相比@ServiceActivator 更为强大， 因为它还内置了一 系列的消息转换功能， 这使得基于@StearnListener 注解实现的消息处理模型更为简单
  - 当消息到达的时候，输入通道的监听器需要对该字符串做一定的转化， 将JSON或XML转换成具体的对象， 然后再做后续的处理
  - 由于@ServiceActivator本身不具备对消息的转换能力，所以当代表User对象的JSON字符串到达后，它自身无法将其转换成User对象。所以，这里需要通过@Transformer 注解帮助将字符串类型的消息转换成 User 对象， 并将转换结果传递给@ServiceActivator的处理方法做后续的消费。
  - 使用 @SteamListener 注解的话
    - @StreamListener 注解能够通过配置属性实现 JSON 字符串到对象的转换， 这是因为在 Spring Cloud Steam 中实现了一套可扩展的消息转换机制。

        - 在消息消费逻辑执行之前，消息转换机制会根据消息头信息中声明的消息类型(即上面对 input 通道配置的contentType 参数)， 找到对应的消息转换器并实现对消息的自动转换

```
//只需在配置文件中增加
spring.cloud.stream.bindings.input.contentTtype=application/json 属性设置
@EnableBinding(value = {Sink.class})
public class SinkReceiver {
        private static Logger logger = LoggerFactory.getLogger(SinkReceiver.class);
        @StrearnListener(Sink.INPUT)
        public void receive(User user) {
    logger.info("Received: "+ user);
  }
}
```

- 消息反馈
    - 处理完输入消息之后， 需要反馈一个消息给对方， 这时候可以通过@SendTo 注解来指定返回内容的输出通道

```
//Appl中实现了对input输入通道的监听，并且在接收到消息之后，对消息做了一些简单加工，然后通过@SendTo 把处理方法返回的内容以消息的方式发送到output通道中
@EnableBinding(value = {Processor.class})
public class Appl {
        private static Logger logger = LoggerFactory.getLogger(Appl.class);
        @StreamListener(Processor.INPUT)
  @SendTo(Processor.OUTPUT)
        public Object receiveFrominput(Object payload) {
                logger.info("Received: " +payload);
                return "From Input Channel Return - " + payload;
  }
}
```

```
# application.properties
# App2是Appl应用中input通道的生产者以及output通道的消费者
# 为了继续使用Processor绑定接口的定义，我们可以在配置文件中将该应用的input和output绑定反向地做一些配置
# 因为对于App2来说，它的input绑定通道实际上是对output主题的消费者，而output绑定通道实际上是对 input主题的生产者
# 配置：指定通道的具体主题来实现与Appl应用的消息交互
spring.cloud.stream.bindings.input.destination=output
spring.cloud.stream.bindings.output.destination=input
```

```
//App2
//通过 timerMessageSource 方法，向 output 绑定的输出通道中发送内容， 也就是
```

```
向名为 input 的主题中传递消息， 这样 Appl 中的 input 输入通道就可以收到这个消息。 同时， 这里还创建了对 input 输入消息通道的绑定。 通过上面的配置， 它会监听来自 OU七put 主题的消息， 通过receiveFromOutput方法， 会将消息内容输出
@EnableBinding(value = {Processor.class})
public class App2 {
        private static Logger logger =
LoggerFactory.getLogger(App2.class);
        @Bean
        @InboundChannelAdapter(value = Processor.OUTPUT, poller =
@Poller(fixedDelay = "2000"))
        public MessageSource<Date> timerMessageSource() {
        return () -> new GenericMessage<>(new Date());
  }
  @StreamListener(Processor.INPUT)
  public void receiveFromOutput(Object payload) {
    logger.info("Received: " +payload);
  }
}
```

- 响应式编程
  - 在 Spring Cloud Stream 中还支持使用基于 RxJava 的响应式编程来处理消息的输入和输出
- 消息类型
  - Spring CloudStream为了让开发者能够在消息中声明它的内容类型， 在输出消息中定义了一个默认的头信息: contentType。
  - 对于那些不直接支持头信息的消息中间件，Spring CloudStream提供了自己的 实现机制， 它会在消息发出前自动将消息 包装进它自定义的消息封装格式中，并加入头信息。
  - 而对于那些 自身就支待头信息 的消息中间件， SpringCloud Stream构建的服务可以接收并处理来自非Spring CloudStream构建但 包含符合规范头信息 的应用程序发出的消息 。
  - pringClodu Stream允许使用 spring.cloud.strearn.bindngs.. content-type属性以声明式的配置方式为绑定 的输入和输出通道设置消息 内容的类型
  - 目前， Spring Cloud Stream中自带支持 了以下几种常用的消息类型转换
    - JSON与POJO的互相转换
    - Object（必须可序列化）与 byte[]的互相转换
    - String与byte[]的互相转换
    - Object向纯文本的 转换: Object需要实现toString()方法
    - JSON与org.springfrarnework.tuple.Tuple的互相转换
    - 上面所指的 JSON类型可以 表现为 一 个 byte类型的数组， 也可以是 一 个包含有效 JSON内容的字符串 。 另外， Object对象可以由JSON、 byte数组或者字符串转换而来，但是在转换为JSON的时候总是 以字符串的形式返回
  - MIME类型
    - 在SpringCloudStream中定义的 content-type属性采用了 Media Type, 即Internet MediaType （互联网媒体类型)， 也被称为MIME类型

        - 常见的有 application/json、 text/plain;charset=UTF-8
        - MIME类型对于标示如何转换为String或byte[]非常有用。并且，还可以使用MIME类型格式来表示Java类型，只需要使用带有类型参数的 一 般类型 : application/x-java-object
        - 比如，可以使用application/x-java object;type= java.util.Map来表示传输的是一个java.unit.Map对象 ，或是使用application/x-java-object;type= core.didispace.User来表示传输的是一个 core.didispace.User对象。它还提供了自定义的MIME类型，比如通过 application/x-spring-tuple来指定Spring的 Tuple类型
    - 在Spring CloudStream中 默认提供了一 些可以开箱即用的类型转换器
        - POJO——JSON String——application/json
        - POJO——String(toString())——text/plain,java.lang.String
        - POJO——byte[] (java.io.serialized)——application/x-java-serialized-object
        - JSON byte[]或String——POJO——application/x-java- object
        - byte[]或String——Senalizable——apphcation/x-Java- object
        - byte[]——String——text/plain,java.lang.String
        - String——byte[]——application/octet-stream
    - 消息类型的转换行为只会在需要进行转换时才被执行，比如，当服务模块产生了一个头信息为application/json的XML字符串消息，Spring Cloud Stream是不会将该XML 字符串转换为JSON的， 这是因为该模块的输出内容已经是一个字符串类型了，所以它并不会将其做进一步的转换
    - 另外需要注意的是， Spring Cloud Stream虽然同时支待输入通道和输出通道的消息类型转换， 但还是推荐开发者尽量在输出通道中做消息转换。 因为对千输入通道的消费者来说， 当目标是一个POJO的时候， 使用@StreamListener注解是能够支待自动对其进行转换的
    - 在应用启用的时候， Spring Cloud Stream 会将所有org.springframework.messaging.converter.MessageConverter接口实现的自定义转换器以及默认实现的那些转换器都加载到消息转换工厂中， 以提供给消息处理时使用

绑定器详解

- 基本概念和作用：它是定义在应用程序与消息中间件之间的抽象层， 用来屏蔽消息中间件对应用的复杂性， 并提供简单而统一的操作接口给应用程序使用
- 绑定器SPI：
    - 绑定器 SPI 涵盖了一套可插拔的用于连接外部中间件的实现机制，其中包含了许多接口、 开箱即用的实现类以及发现策略等内容。 其中，最为关键的就是Binder接口，它是用来将输入和输出连接到外部中间件的抽象

```
public interface Binder<T, C extends ConsumerProperties, P extends
ProducerProperties> {
        Binding<T> bindConsumer(String name, String group, T
inboundBindTarget, C consumerProperties);
        Binding<T> bindProducer(String name, T outboundBindTarget, P
producerProperties);
}
```

    - 当应用程序对输入和输出通道进行绑定的时候， 实际上就是通过该接口的实现来完成的。
        - 向消息通道发送数据的生产者调用 bindProducer 方法来绑定输出通道时
            - 第一个参数代表了发往消息中间件的目标名称
            - 第二个参数代表了发送消息的本地通道实例
            - 第三个参数是用来创建通道时使用的属性配置(比如分区键的表达式等)。
        - 从消息通道接收数据的消费者调用 bindConsumer 方法来绑定输入通道时
            - 第一个参数代表了接收消息中间件的目标名称
            - 第二个参数代表了消费组的名称(如果多个消费者实例使用相同的组名， 则消息将对这些消费者实例实现负载均衡， 每个生产者发出的消息只会被组内一个消费者实例接收和处理)
            - 第三个参数代表了接收消息的本地通道实例
            - 第四个参数是用来创建通道时使用的属性配置
    - 一个典型的Binder 绑定器实现一般包含以下内容。
        - 一 个实现binder接口的类
        - 一个Spring配置加载类，用来创建连接消息中间件的基础结构使用的实例
        - 一个或多个能够在classpath下的META-INF/spring.binders 路径找到的绑定器定义文件。比如我们可以在 spring-cloud-starter-strearn-rabbit中找到该文件，该文件中存储了当前绑定器要使用的自动化配置类的路径:

            ```
            rabbit:\
            org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfigur ation
            ```

- 自动化配置
    - Spring Cloud Stream通过绑定器SPI的实现将应用程序逻辑上的输入输出通道连接到物理上的消息中间件。
    - 消息中间件之间通常都会有或多或少的差异性， 所以 为了适配不同的消息中间件， 需要为它们实现各自独有的绑定器
    - 目前， Sprni g Cloud Stream中默认实现了对Rabbti MQ、Kafka的绑定器
    - 默认情况下， Spring Cloud Stream也遵循Spring Boot自动化配置的特性。
        - 如果在 classpath中能够找到单个绑定器的实现，那么SpringCloudStream会自动加载它。而我们在classpath中引入绑定器的方法也非常简单，只需要在pom.xml 中增加对应消息中间件的绑定器依赖即可 `spring-cloud-stream-binder-rabbit`(spring-cloud- starter stream-rabbit 依赖中就包含)或`spring-cloud-stream-binder-kafka`
- 多绑定器配置
    - 当应用程序的classpath下存在多个绑定器时，SpringCloudStream在为消息通道做绑定操作时， 无法判断应该使用哪个具体的绑定器， 所以需要为每个输入或输出通道指定具体的绑定器

- 在一个应用程序中使用多个绑定器时， 往往其中一个绑定器会是主要使用的， 而第二个可能是为了适应一些特殊要求(比如性能等原因)。可以先通过设置默认绑定器来为大部分的通道设置绑定器。 比如， 使用 RabbitMQ 设置默认绑定器`spring.cloud.stream.defaultBinder=rabbit`，在设置了默认绑定器之后， 再为其他一些少数的消息通道单独设置绑定器`spring.cloud.stream.bindings.input.binder=kafka`
- 注意，设置参数时用来指定具体绑定器的值并不是消息中间件的名称， 而是在每个绑定器实现的 META-INF/spring.binders 文件中定义的标识(一个绑定器实现的标识可以定义多个， 以逗号分隔)， 所以上面配置的rabb江和kafka分别来自于各自的配置定义， 它们的具体内容如下所示:

```
rabbit:\
org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoC
onfiguration
kafka:\
org.springframework.cloud.stream.binder.kafka.config.KafkaBinderConfigur
ation
```

- 当需要在一个应用程序中使用同一 类型不同环境的绑定器时， 也可以通过配置轻松实现通道绑定。 比如， 当需要连接两个不同的 RabbitMQ 实例的时候， 可以参照如下配置:

```
spring.cloud.stream.bindings.input.binder=rabbitl
spring.cloud.stream.bindings.output.binder=rabbit2

spring.cloud.stream.binders.rabbitl.type=rabbit
spring.cloud.stream.binders.rabbitl.environment.spring.rabbitmq.host=192
.168.0.101
spring.cloud.stream.binders.rabbitl.envirorunent.spring.rabbitmq.port=56
72
spring.cloud.stream.binders.rabbitl.envirorunent.spring.rabbitmq.usernam
e=springcloud
spring.cloud.stream.binders.rabbitl.envirorunent.spring.rabbitmq.passwor
d=123456

spring.cloud.stream.binders.rabbit2.type=rabbit
spring.cloud.stream.binders.rabbit2.environment.spring.rabbitmq.host=192
.168.0.102
spring.cloud.stream.binders.rabbit2.environment.spring.rabbitmq.port=567
2
spring.cloud.stream.binders.rabbit2.environment.spring.rabbitmq.username
=springcloud
spring.cloud.stream.binders.rabbit2.environment.spring.rabbitmq.password
=123456
```

  - 当采用显式配置方式时会自动禁用默认的绑定器配置， 所以当定义了显式配置别名后，对于这些绑定器的配置需要通过 spring.cloud.stream. binders.属性来进行设置
    - spring.cloud.stream.binders..type 指定了绑定器的类型， 可以是rabbit、 kafak

或者其他自定义绑定器的标识名， 绑定器标识名的定义位于绑定器的META-INF/spring.binders 文件中
    - spring.cloud.stream.binders..environment参数可以直接用来设置各绑定器的属性， 默认为空
    - spring.cloud.stream.binders..inheritEnvironment 参数用来配置当前绑定器是否继承应用程序自身的环境配置， 默认为 true
    - spring.cloud.stream.binders..defaultCandidate参数用来设置当前绑定器配置是否被视为默认绑定器的候选项，默认为true。 当需要让当前配置不影响默认配置时， 可以将该属性设置为false。

Rabbit和Kafka绑定器

- 在RabbitMQ中，通过Exchange交换器来实现 Spring Cloud Stream 的主题概念，所以消息通道的输入输出目标映射了一个具体的Exchange交换器。而对于每个消费组，则会为对应的Exchange交换器绑定一个Queue队列进行消息收发
- 由于Kafka自身就有Topic概念， 所以 Spring Cloud Stream的主题 直接采用了 Kafka的Topic 主题概念， 每个消费组的通道目标都会直接连接Kafka 的主题 进行消息收发
- 在Spring Cloud Stream中对绑定通道和绑定器提供了通用的属性配置项， 一些绑定器 还允许使用附加属性来对消息中间件的一些独有特性进行配置。 这些属性的配置可以通过 Spring Boot支持的任何配置方式来进行， 包括使用环境变量、 YAML或者properties配置文件等
- 基础配置
    - Spring Cloud Stream应用级别的通用基础属性，这些属性都以spring.cloud.Stream.为前缀
        - instanceCount=1，应用程序部署的实例数屈。 当使用Kafka的时候需要设置分区
        - instanceIndex，应用程序实例的索引，该值从0开始，最大值设置为-1。当使用分区和Kafka的时候使用
        - dynamicDestinations，动态绑定的目标列表， 该列表默认为空， 当设置了具体列表之后， 只有列表中的目标才能被发现
        - defaultBinder，默认绑定器配置， 在应用程序中有多个绑定器时使用
        - overrideCloudConnectors，该属性只适用于激活cloud配置并且提供了SpringCloudConnectors的应用
- 绑定通道配置
    - 通 过 spring.cloud.stream.bindings..=格式的参数来进行设置
    - 代表在绑定接口中定义的通道名称，比如，Sink中的input、 Source 中的output
    - 由于绑定通道分为输入通道和输出通道， 所以在绑定通道的配置中包含了三类面向不同通道类型的配置：
        - 通用配置（通过spring.cloud.stream.bindings.. 前缀来进行设置）
            - destination，配置消息通道绑定在消息中间件中的目标名称，比如RabbitMQ的Exchange或Kafka的Topic。 如果配置的绑定通道是一个消费者(输入通道)， 那么它可以绑定多个目标，这些目标名称通过逗号来分隔。 如果没有设置该属性， 将使用通道名
            - group，设置绑定通道的消费组， 该参数主要作用于输入通道， 以保证同一消息组中的消息只会有一个消费示例接收和处理
            - contentType，设置绑定通道的消息类型

        - binder，当存在多个绑定器时使用该参数来指定当前通道使用哪个具体的绑定器
    - 消费者配置（以spring.cloud.stream. bindings..consumer.格式作为前缀 ）
        - 仅对输入通道的绑定有效
        - concurrency=1，输入通道消费者的并发数
        - partitioned=false，来自消息生产者的数据是否采用了分区
        - headerMode=embeddedHeaders，当设置为 raw的时候将禁用对消息头的解析，该属性只有在使用不支持消息头功能的中间件时有效， 因为Spring CloudStream默认会解析嵌入的头部信息
        - maxAttempts=3，对输入通道消息处理的最大重试次数
        - backOffinitialinterval=1000，重试消息处理的初始间隔时间
        - backOffMaxinterval=10000，重试消息处理的最大间隔时间
        - backOffMultiplier=2.0，重试消息处理时间间隔的递增乘数
    - 生产者配置（spring.cloud.stream. bindings..producer.格式作为前缀 ）
        - 仅对输出通道的绑定有效
        - partitionKeyExpression=null，该参数用来配置输出通道数据分区键的SpEL表达式。 当设置该属性之后， 将对当前绑定通道的输出数据进行分区处理。 同时，partionCount参数必须大于1才能生效。 该参数与partitionKeyExtractorClass 参数互斥， 不能同时设置
        - parttitionKeyExtractorClass=null，配置分区键提取策略接口PartitionKeyExractorStrategy的实现
        - partitionSelectorClass=null，指定分区选择器接口PartitionSelectorStrategy的实现
        - partitionSelectorExpression=null，配置自定义分区选择器的SpEl表达式
        - partitionCount=1，当分区功能开启时，配置消息数据的分区数，如果消息生产者已经配置了分区键的生成策略，那么它的值必须大于1
- 绑定器配置
    - RabbitMQ配置
        - 通用配置
            - RabbitMQ绑定器默认使用了Spring Boot的 ConnectionFactory
            - RabbitMQ绑定器支持在Spir ngBoot中的配置选项以spring.rabbitmq. 为前缀
            - 在SpringCloud Stream对RabbitMQ实现的绑定器中都以spring.cloud.stream. rabbit.binder.为前缀
            - 属性可以在org.springframework. cloud.stream.binder.rabbit.config.RabbitBinderConfigurationProperties中找到
            - adminAddresses，配置RabbitMQ管理插件的URL
            - nodes，配置RabbitMQ的节点名称
            - compressionLevel，绑定通道的压缩级别
        - 消费者配置
            - 仅对RabbitMQ输入通道的绑定有效，以spring.cloud.stream.rabbit.bindings..consumer. 格式作为前缀
            - acknowledgeMode=AUTO，用来设置消息的确认模式
            - autoBindDlq=false，是否自动声明DLQ (Dead-Letter-Queue), 并绑定到

DLX (Dead-Letter-Exchange) 上

- durableSubscription=true，订阅是否被持久化，该参数仅在group被设置的时候有效
- maxConcurrency=1，消费者的最大并发数
- prefetch=1，预取数量，它表示在 一次会话中从消息中间件中获取的消息数量，该值越大消 息处理越快，但是会导致非顺序处理的风险
- prefix，用来设置统 一 的目标和队列名称前缀
- recoveryinterval=5000，用来设置恢复连接的尝试时间间隔， 以毫秒为单位
- requeueRejected=true，消息传递失败时重传
- requestHeaderPatterns，需要被传递的请求头信息
- replyHeaderPatterns，需要被传递的响应头信息
- republishToDlq，默认情况下，消息在重试也失败之后会被拒绝
- transacted=false，是否启用channel- transacted, 即是否在消息中使用事务
- txSize=1，设置transaction-size的数量， 当acknowledge-Mode被设置为AUTO时， 容器会在处理txSize数目消息之后才开始应答

- 生产者配置
  - 仅对 RabbitMQ 输出通道的绑定有效，以 spring.cloud.stream.rabbit.endings..producer. 格式作为前缀
  - autoBindDlq=false，是否自动声明DLQ CDead-Letter-Queue), 并绑定到DLX (Dead-Letter-Exchange)上
  - batchingEnabled=false，是否启用 消息批处理
  - batchSize=100，当批处理开启时， 用来设置缓存的批处理消息数量
  - batchBufferLimit=10000，批处理缓存限制
  - batchTimeout=5000，批处理超时时间
  - compress=false，消息发送时是否启用压缩
  - deliveryMode=PERSISTENT，消息发送模式
  - prefix，用来设置统一 的目标前缀
  - requestHeaderPatterns，需要被传递的请求头信息
  - replyHeaderPatterns，需要被传递的响应头信息

- Kafka配置
  - 通用配置
    - 都以spring.cloud.stream.kafka.binder. 为前缀
    - brokers=localhost，Kafka绑定器连接的消息中间件列表
    - defaultBrokerPort=9092，消息中间件端口号
    - zkNodes=localhost，Kafka绑定器使用的ZooKeeper节点列表
    - defaultZkPort=2181，用来设置默认的ZooKeeper端口号
    - headers，用来设置会被传输的自定义头信息
    - offsetUpdateTimeWindow=10000，用来设置offset 的更新频率
    - offsetUpdateCount=0，用来设置offse七以次数表示的更新频率
    - requiredAcks=1，确认消息的数量
    - minPartitionCount=1，仅在设置了autoCreateTopics 和autoAddParitions时生效用来设置该绑定器所使用主题的全局分区最小数量。

        - replicationFactor=1，当autoCreateTopics参数为true时候， 用来配置自动创建主题的副本数品
        - autoCreateTopics=true，绑定器会自动地创建新主题
        - autoAddPartitions=false，绑定器会根据已经配置的主题分区来实现， 如果目标主题的分区数小于预期值， 那么绑定器 会启动失败。 如果该参数设置为true, 绑定器将在需要的时候自动创建新的分区
        - socketBufferSize=2097152，设置Kafka的Socket缓存大小
    - 消费者配置
        - 仅对Kafka输入通道的绑定有效，以spring.cloud.stream.kafka.bindings..consumer.格式作为前缀
        - autoComrnitOffset=true，否在处理消息 时自动提交offset。如果设置为false, 在消息头中会加入AKC 头信息以实现延迟确认
        - autoCommitOnError，只有在autoComrn止Offset设置为true 时才有效。当设置为false的时候， 引起错误的消息不会自动提交offset, 仅提交成功消息的offset。 如果设置为true, 不论消息是否成功，都会自动提交。当不设置该值 时，它实际上具有与enableDlq相同的配置值
        - recoveryInterval=5000，尝试恢复连接的时间间隔， 以毫秒为单位
        - resetOffsets=false，是否使用提供的startOffset值 来重置 消费者的offset值
        - startOffset=null，用来设置新建组的起始 offset, 该值也会在 resetOffsets开始时被使用
        - enableDlq=false，设置为true 时，将为消费者启用DLQ行为，引起错误的消息将被 发送到名为 error..的主题中去
    - 生产者配置
        - 仅对Kafka输出通道的绑定有效，以spring.cloud.stream.kafka.bindings..producer.格式作为前缀
        - bufferSize=16384，Kafka批址发送前的缓存数据上限， 以字节为单位
        - sync=false，设置 Kafka 消息生产者的发送模式， 默认为 false, 即采用async配置，允许批量发送数据。当设权为true 时，将采用sync配置， 消息将不会被批献发送， 而是一条一条地发送
        - batchTimeout=0，消息生产者批量发送时， 为了积累更多发送数据而设置的等待时间。 通常 情况下，生产者基本不会等待， 而是直接发送所有在前一批次发送时积累 的消息数据。 当我们设置一个非o值时，可以以延迟为代价来增加系统的 吞吐量