Spring Cloud 系列之 Stream 消息驱动

历史修订

本次修订日期: 2023-03-12	下次修订日期:

修订编号	修订日期	变更描述	说明
V0.1	2023-03-10	起草	李宗在
V0.2	2023-03-12	验证	李宗在
V0.3			

1. 技术介绍

- Spring Boot
- Spring Cloud
- Ubuntu
- RabbitMQ
- Stream

2. 学习目标

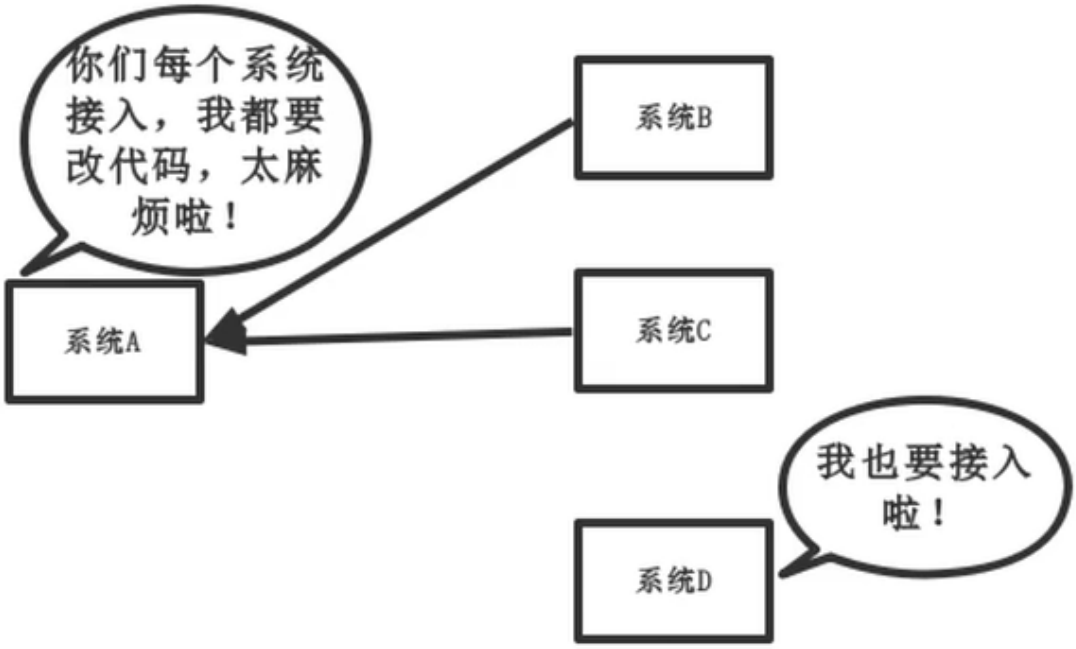
在实际开发过程中，服务与服务之间通信经常会使用到消息中间件，消息中间件解决了应用解耦、异步处理、流量削锋等问题，实现高性能，高可用，可伸缩和最终一致性架构。

不同中间件内部实现方式是不一样的，这些中间件的差异性导致我们实际项目开发给我们造成了一定的困扰，比如项目中间件为 Kafka，如果我们要替换为 RabbitMQ，这无疑就是一个灾难性的工作，一大堆东西都要重做，因为它跟我们系统的耦合性非常高。这时我们可以使用 Spring Cloud Stream 来整合我们的消息中间件，降低系统和中间件的耦合性。

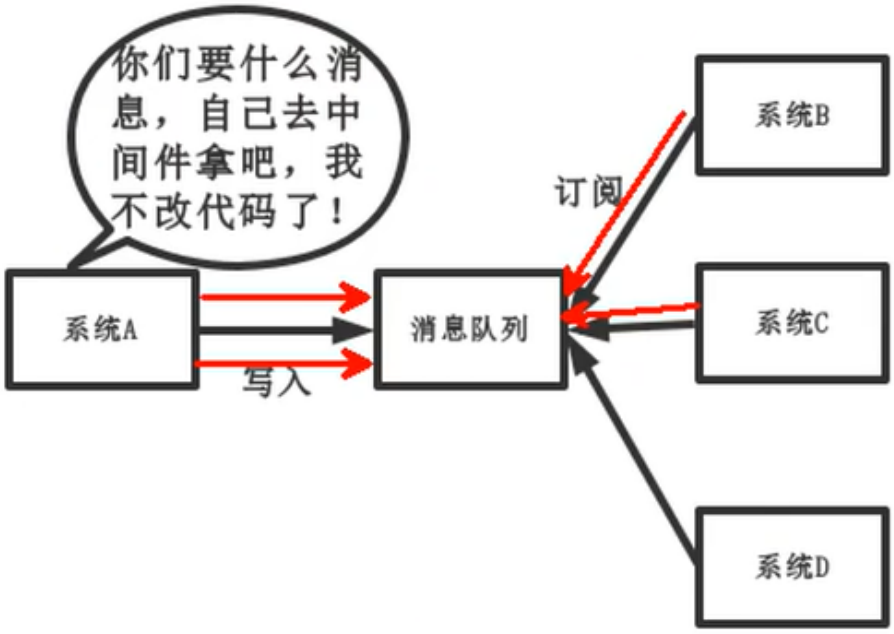
3. 消息中间件应用场景

3.1 应用解耦

3.1.1 传统模式



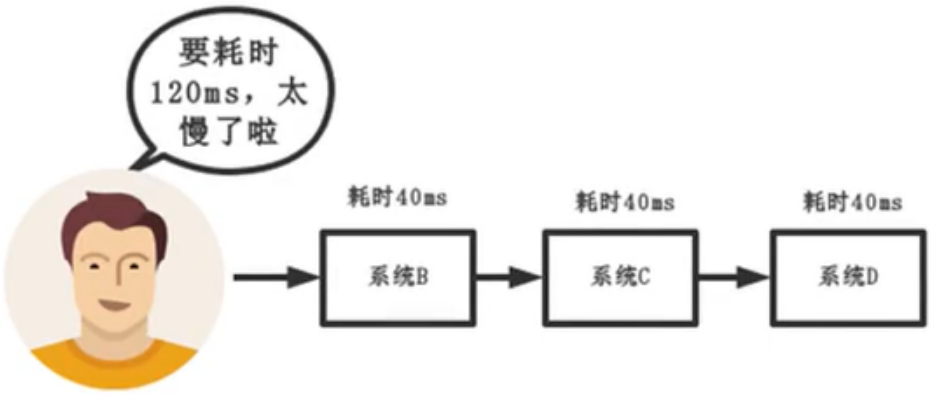
3.1.2 中间件模式



3.2 异步处理

比如用户在电商网站下单，下单完成后会给用户推送短信或邮件，发短信和邮件的过程就可以异步完成。因为下单付款才是核心业务，发邮件和短信并不属于核心功能，且可能耗时较长，所以针对这种业务场景可以选择先放到消息队列中，由其他服务来异步处理。

3.2.1 传统模式

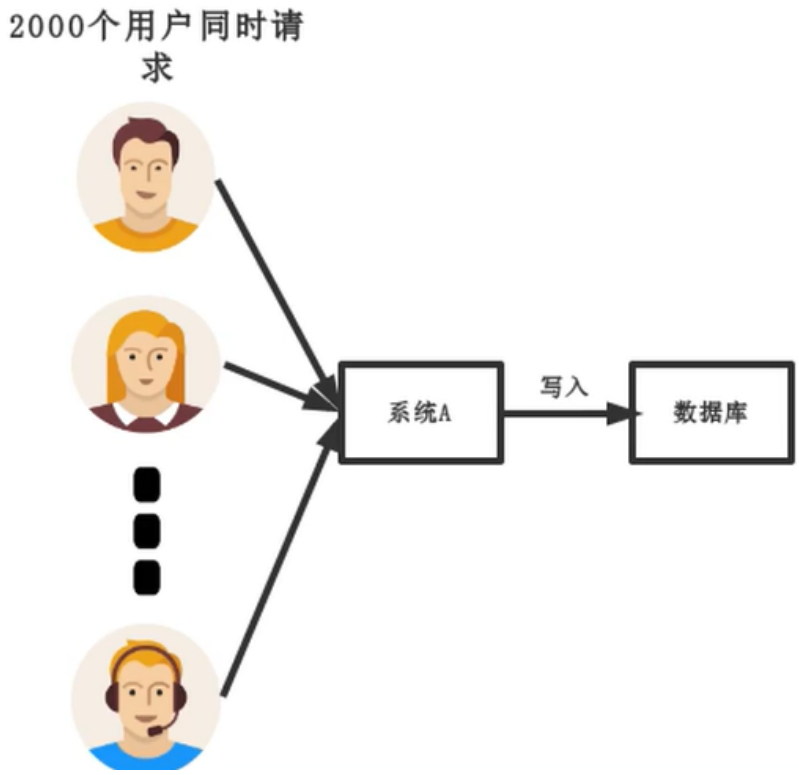


3.2.2 中间件模式

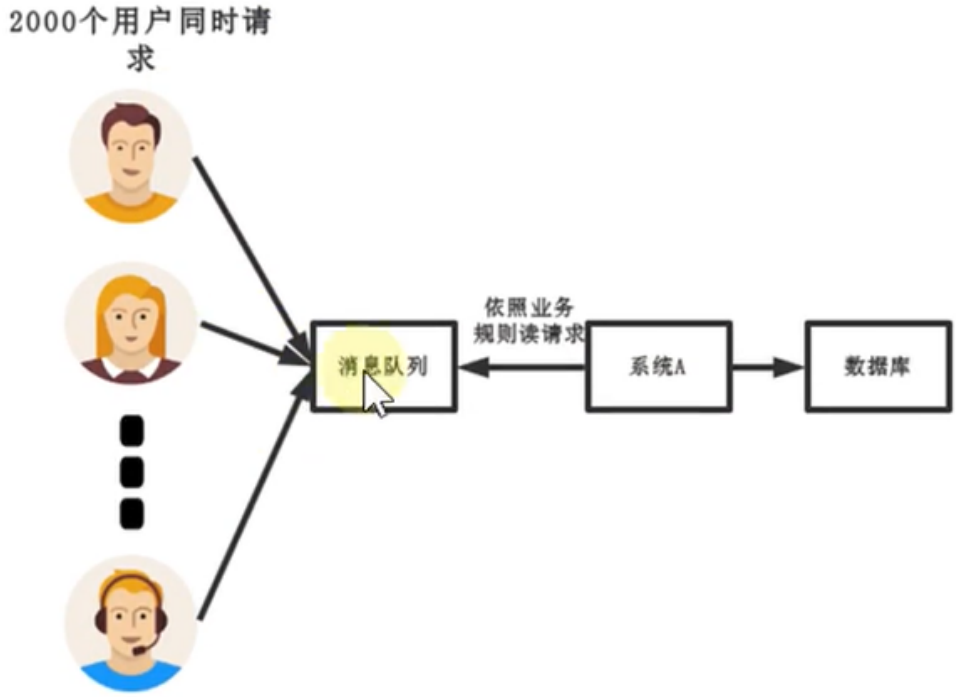
3.3 流量削峰

比如秒杀活动，一下子进来好多请求，有的服务可能承受不住瞬时高并发而崩溃，针对这种场景，在中间加一层消息队列，把请求先入队列，然后再把队列中的请求平滑的推送给服务，或者让服务去队列拉取。

3.3.1 传统模式



3.3.2 中间件模式



3.4 日志处理

对于小型项目来说，我们通常对日志的处理没有那么多的要求，但是当用户量，数据量达到一定的峰值之后，问题就会随之而来。比如：

- 用户日志怎么存放
- 用户日志存放后怎么利用
- 怎么在存储大量日志而不对系统造成影响

等很多其他的问题，这样我们就需要借助消息队列进行业务的上解耦，数据上更好的传输。

Kafka 最开始就是专门为了处理日志产生的。

3.5 总结

消息队列，是分布式系统中重要的组件，其通用的使用场景可以简单地描述为：**当不需要立即获得结果，但是并发量又需要进行控制的时候，差不多就是需要使用消息队列的时候。**在项目中，将一些无需即时返回且耗时的操作提取出来，进行了异步处理，而这种异步处理的方式大大的节省了服务器的请求响应时间，从而提高了系统的吞吐量。

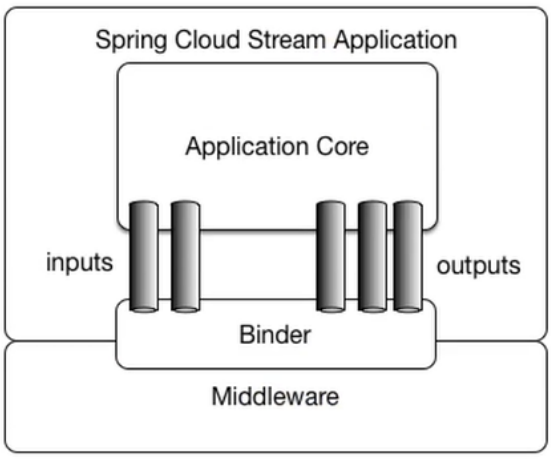
当遇到上面几种情况的时候，就要考虑用消息队列了。如果你碰巧使用的是 RabbitMQ 或者 Kafka，而且同样也在使用 Spring Cloud，那你可以考虑下用 Spring Cloud Stream。

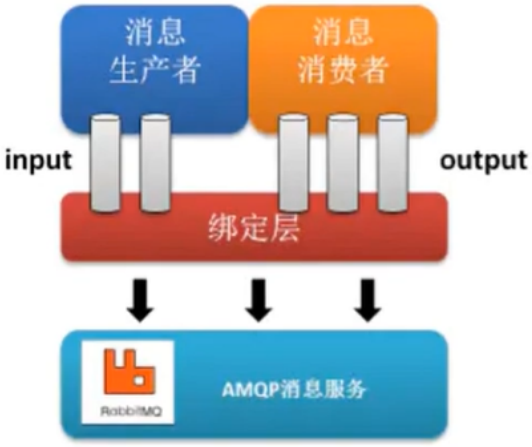
4. 什么是Spring Cloud Stream

Spring Cloud Stream 是用于构建消息驱动微服务应用程序的框架。该框架提供了一个灵活的编程模型，该模型建立在已经熟悉 Spring 习惯用法的基础上，它提供了来自多家供应商的中间件的合理配置，包括 publish-subscribe，消息分组和消息分区处理的支持。

Spring Cloud Stream 解决了开发人员无感知的使用消息中间件的问题，因为 Stream 对消息中间件的进一步封装，可以做到代码层面对中间件的无感知，甚至于动态的切换中间件，使得微服务开发的高度解耦，服务可以关注更多自己的业务流程。

5. 核心概念

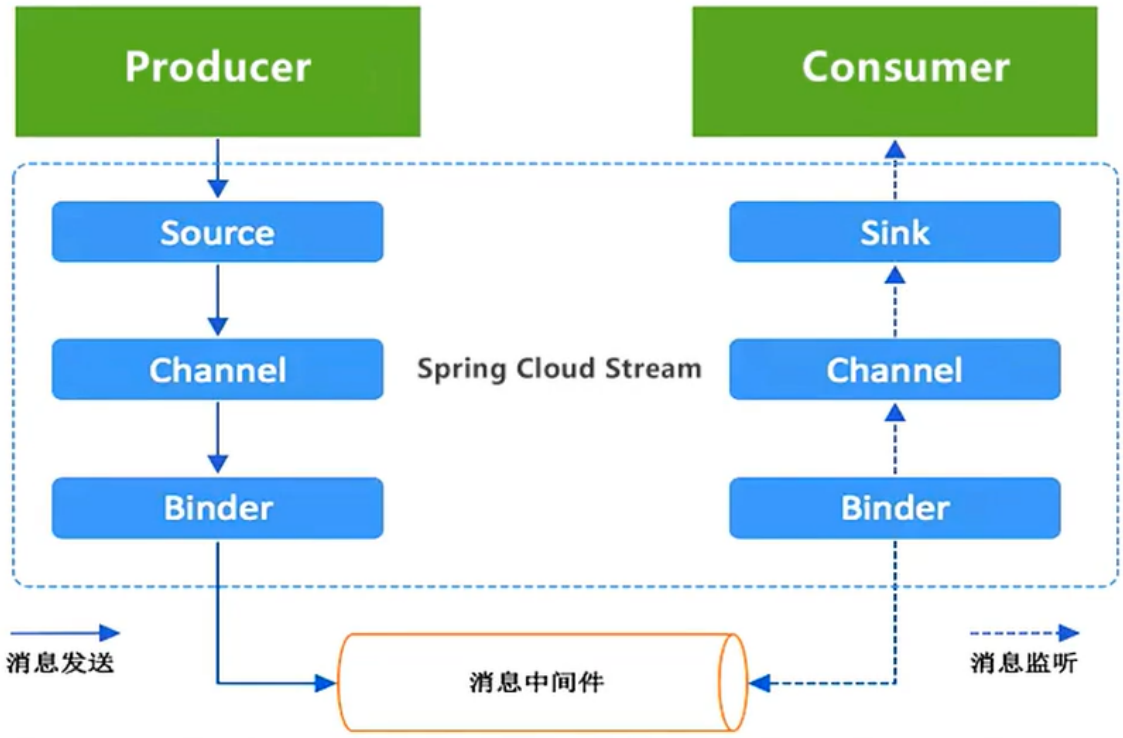




组成	说明
Middleware	中间件，支持 RabbitMQ 和 Kafka。
Binder	目标绑定器，目标指的是 Kafka 还是 RabbitMQ。绑定器就是封装了目标中间件的包。如果操作的是 Kafka 就使用 `spring-cloud-stream-binder-kafka` ，如果操作的是 RabbitMQ 就使用 `spring-cloud-stream-binder-rabbit` 。
@Input	注解标识输入通道，接收（消息消费者）的消息将通过该通道进入应用程序。
@Output	注解标识输出通道，发布（消息生产者）的消息将通过该通道离开应用程序。
@StreamListener	监听队列，消费者的队列的消息接收。
@EnableBinding	注解标识绑定，将信道 `channel` 和交换机 `exchange` 绑定在一起。

6.工作原理

通过定义**绑定器**作为中间层，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 Channel 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，或者是更换其他消息中间件产品时，我们需要做的就是更换对应的 `Binder` 绑定器而不需要修改任何应用逻辑。



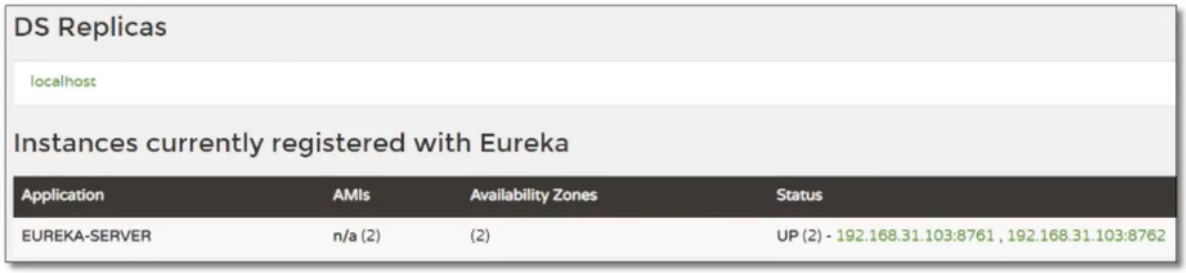
该模型图中有如下几个核心概念：

- `Source`：当需要发送消息时，我们就需要通过 `Source.java`，它会把我们所要发送的消息进行序列化（默认转换成 JSON 格式字符串），然后将这些数据发送到 Channel 中；
- `Sink`：当我们需要监听消息时就需要通过 `Sink.java`，它负责从消息通道中获取消息，并将消息反序列化成消息对象，然后交给具体的消息监听处理；
- `Channel`：通常我们向消息中间件发送消息或者监听消息时需要指定主题（Topic）和消息队列名称，一旦我们需要变更主题的时候就需要修改消息发送或消息监听的代码。通过 `Channel` 对象，我们的业务代码只需要对应 `Channel` 就可以了，具体这个 Channel 对应的是哪个主题，可以在配置文件中来指定，这样当主题变更的时候我们就不用对代码做任何修改，从而实现了与具体消息中间件的解耦；
- `Binder`：通过不同的 `Binder` 可以实现与不同的消息中间件整合，`Binder` 提供统一的消息收发接口，从而使得我们可以根据实际需要部署不同的消息中间件，或者根据实际生产中所部署的消息中间件来调整我们的配置。

7. 环境准备

`stream` 聚合工程。`SpringBoot 2.2.4.RELEASE`、`Spring Cloud Hoxton.SR1`。

- `RabbitMQ`：消息队列
- `eureka-server`：注册中心
- `eureka-server02`：注册中心



8. 入门案例

8.1 消息生产者

在 `stream` 项目下创建 `stream-producer` 子项目

8.1.1 添加依赖

要使用 `RabbitMQ` 绑定器，可以通过使用以下 `Maven` 坐标将其添加到 `Spring Cloud Stream` 应用程序中：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

或者使用 Spring Cloud Stream RabbitMQ Starter：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

完整依赖如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>stream-producer</artifactId>
    <version>1.0-SNAPSHOT</version>

    <!-- 继承父依赖 -->
    <parent>
        <groupId>com.example</groupId>
        <artifactId>stream-demo</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
```

```
    <!-- 项目依赖 -->
    <dependencies>
        <!-- netflix eureka client 依赖 -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
        <!-- spring cloud stream binder rabbit 绑定器依赖 -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
        </dependency>

        <!-- spring boot test 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>

</project>
```

8.1.2 配置文件

配置 `RabbitMQ` 消息队列和 `Stream` 消息发送与接收的通道。

```
server:
  port: 8001 # 端口

spring:
  application:
    name: stream-producer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: lizongzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
  cloud:
    stream:
      bindings:
        # 消息发送通道
        # 与 org.springframework.cloud.stream.messaging.Source 中的 @Output("output") 注
解的 value 相同
        output:
          destination: stream.message # 绑定的交换机名称

# 配置 Eureka Server 注册中心
eureka:
```

```yaml
  instance:
    prefer-ip-address: true       # 是否使用 ip 地址注册
    instance-id: ${spring.cloud.client.ip-address}:${server.port} # ip:port
  client:
    service-url:                  # 设置服务注册中心地址
      defaultZone: http://localhost:8761/eureka/,http://localhost:8762/eureka/
```

8.1.3 发送消息

`MessageProducer.java`

```java
package com.example.producer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;

/**
 * 消息生产者
 */
@Component
@EnableBinding(Source.class)
public class MessageProducer {

    @Autowired
    private Source source;

    /**
     * 发送消息
     *
     * @param message
     */
    public void send(String message) {
        source.output().send(MessageBuilder.withPayload(message).build());
    }

}
```

8.1.4 启动类

```java
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class StreamProducerApplication {

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

}
```

8.2 消息消费者

在 `stream` 项目下创建 `stream-consumer` 子项目。

8.2.1 添加依赖

要使用 `RabbitMQ` 绑定器，可以通过使用以下 `Maven` 坐标将其添加到 `Spring Cloud Stream` 应用程序中：

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

或者使用 Spring Cloud Stream RabbitMQ Starter：

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

完整依赖如下：

```xml
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>stream-consumer</artifactId>
    <version>1.0-SNAPSHOT</version>

    <!-- 继承父依赖 -->
    <parent>
        <groupId>com.example</groupId>
        <artifactId>stream-demo</artifactId>
        <version>1.0-SNAPSHOT</version>
```

```
    </parent>

    <!-- 项目依赖 -->
    <dependencies>
        <!-- netflix eureka client 依赖 -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
        <!-- spring cloud stream binder rabbit 绑定器依赖 -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
        </dependency>

        <!-- spring boot test 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>

</project>
```

8.2.1 配置文件

配置 `RabbitMQ` 消息队列和 `Stream` 消息发送与接收的通道。

```
server:
  port: 8002 # 端口

spring:
  application:
    name: stream-consumer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: lizongzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
  cloud:
    stream:
      bindings:
        # 消息接收通道
        # 与 org.springframework.cloud.stream.messaging.Sink 中的 @Input("input") 注解的
value 相同
        input:
          destination: stream.message # 绑定的交换机名称
```

```yaml
# 配置 Eureka Server 注册中心
eureka:
  instance:
    prefer-ip-address: true       # 是否使用 ip 地址注册
    instance-id: ${spring.cloud.client.ip-address}:${server.port} # ip:port
  client:
    service-url:                  # 设置服务注册中心地址
      defaultZone: http://localhost:8761/eureka/,http://localhost:8762/eureka/
```

8.2.3 接收消息

`MessageConsumer.java`

```java
package com.example.consumer;

import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.cloud.stream.messaging.Sink;
import org.springframework.stereotype.Component;

/**
 * 消息消费者
 */
@Component
@EnableBinding(Sink.class)
public class MessageConsumer {

    /**
     * 接收消息
     *
     * @param message
     */
    @StreamListener(Sink.INPUT)
    public void receive(String message) {
        System.out.println("message = " + message);
    }

}
```

8.2.4 启动类

`StreamConsumerApplication.java`

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class StreamConsumerApplication {

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

}
```

8.2.5 单元测试

`MessageProducerTest.java`

```
package com.example;

import com.example.producer.MessageProducer;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = {StreamProducerApplication.class})
public class MessageProducerTest {

    @Autowired
    private MessageProducer messageProducer;

    @Test
    public void testSend() {
        messageProducer.send("hello spring cloud stream");
    }

}
```

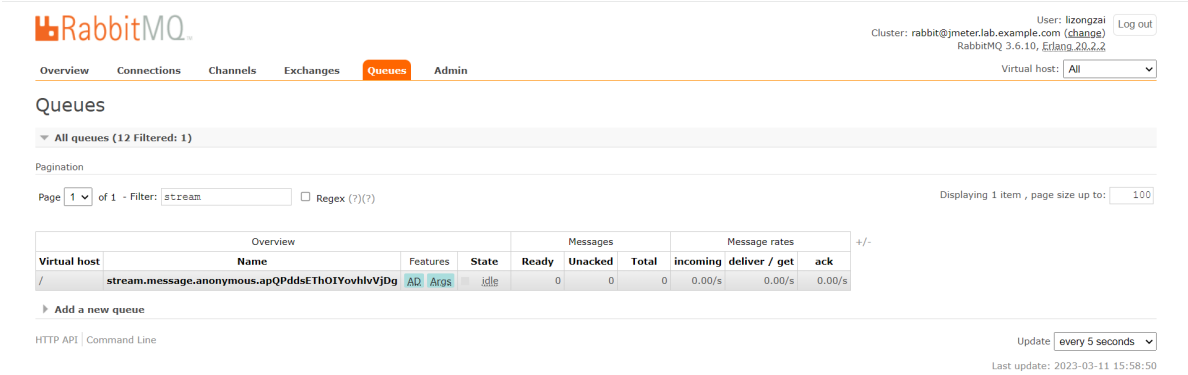
8.2.6 访问RabbitMQ

启动消息消费者，运行单元测试，消息消费者控制台打印结果如下：

```
message = hello spring cloud stream
```

RabbitMQ 界面如下：

http://192.168.126.64:15672/#/queues



8.3 自定义消息通道

8.3.1 创建消息通道

参考源码 `Source.java` 和 `Sink.java` 创建自定义消息通道。

自定义消息发送通道 `MySource.java`

```
package com.example.channel;

import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;

/**
 * 自定义消息发送通道
 */
public interface MySource {

    String MY_OUTPUT = "my_output";

    @Output(MY_OUTPUT)
    MessageChannel myOutput();

}
```

自定义消息接收通道 `MySink.java`

```
package com.example.channel;

import org.springframework.cloud.stream.annotation.Input;
import org.springframework.messaging.SubscribableChannel;

/**
 * 自定义消息接收通道
 */
public interface MySink {
```

```
    String MY_INPUT = "my_input";

    @Input(MY_INPUT)
    SubscribableChannel myInput();
```

8.3.2 配置文件

消息生产者

```
server:
  port: 8001 # 端口

spring:
  application:
    name: stream-producer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: liozngzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
  cloud:
    stream:
      bindings:
        # 消息发送通道
        # 与 org.springframework.cloud.stream.messaging.Source 中的 @Output("output") 注解的 value 相同
        output:
          destination: stream.message # 绑定的交换机名称
        my_output:
          destination: my.message # 绑定的交换机名称
```

消息消费者

```
server:
  port: 8002 # 端口

spring:
  application:
    name: stream-consumer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: lizongzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
  cloud:
    stream:
      bindings:
        # 消息接收通道
        # 与 org.springframework.cloud.stream.messaging.Sink 中的 @Input("input") 注解的 value 相同
        input:
```

```
          destination: stream.message # 绑定的交换机名称
        my_input:
          destination: my.message # 绑定的交换机名称
```

8.3.3 代码重构

消息生产者 `MyMessageProducer.java`。

```
package com.example.producer;

import com.example.channel.MySource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;

/**
 * 消息生产者
 */
@Component
@EnableBinding(MySource.class)
public class MyMessageProducer {

    @Autowired
    private MySource mySource;

    /**
     * 发送消息
     *
     * @param message
     */
    public void send(String message) {
        mySource.myOutput().send(MessageBuilder.withPayload(message).build());
    }

}
```

消息消费者 `MyMessageConsumer.java`。

```
package com.example.consumer;

import com.example.channel.MySink;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.stereotype.Component;

/**
 * 消息消费者
 */
@Component
@EnableBinding(MySink.class)
public class MyMessageConsumer {

    /**
```

```
     * 接收消息
     *
     * @param message
     */
    @StreamListener(MySink.MY_INPUT)
    public void receive(String message) {
        System.out.println("message = " + message);
    }

}
```

8.3.4 单元测试

`MessageProducerTest.java`

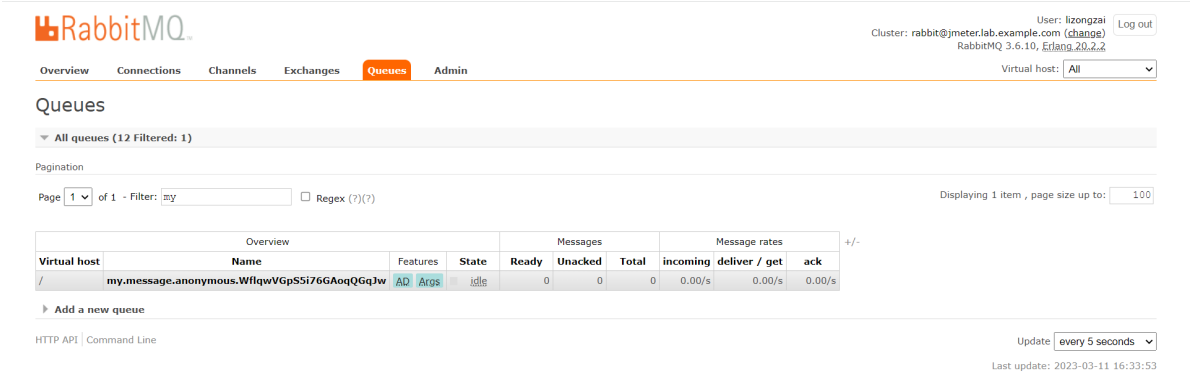
```
package com.example;

import com.example.producer.MyMessageProducer;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = {StreamProducerApplication.class})
public class MessageProducerTest {

    @Autowired
    private MyMessageProducer myMessageProducer;

    @Test
    public void testMySend() {
        myMessageProducer.send("hello spring cloud stream");
    }

}
```

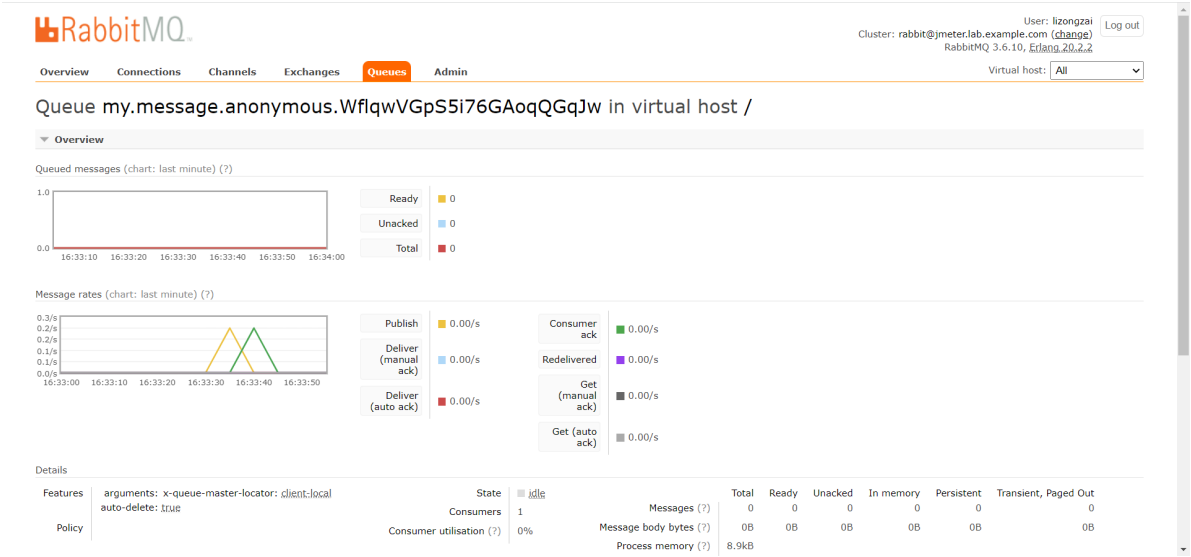
8.3.5 访问

启动消息消费者，运行单元测试，消息消费者控制台打印结果如下：

```
message = hello spring cloud stream
```

RabbitMQ 界面如下：





8.4 配置优化

Spring Cloud 微服务开发之所以简单，除了官方做了许多彻底的封装之外还有一个优点就是**约定大于配置**。开发人员仅需规定应用中不符约定的部分，在没有规定配置的地方采用默认配置，以力求最简配置为核心思想。

> 简单理解就是：Spring 遵循了推荐默认配置的思想，当存在特殊需求时候，自定义配置即可否则无需配置。

在 Spring Cloud Stream 中，`@Output("output")` 和 `@Input("input")` 注解的 `value` 默认即为绑定的交换机名称。所以自定义消息通道的案例我们就可以重构为以下方式。

8.4.1 创建消息通道

参考源码 `Source.java` 和 `Sink.java` 创建自定义消息通道。

自定义消息发送通道 `MySource02.java`

```
package com.example.channel;

import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;

/**
 * 自定义消息发送通道
 */
public interface MySource02 {

    String MY_OUTPUT = "default.message";

    @Output(MY_OUTPUT)
    MessageChannel myOutput();

}
```

自定义消息接收通道 `MySink02.java`

```
package com.example.channel;

import org.springframework.cloud.stream.annotation.Input;
import org.springframework.messaging.SubscribableChannel;

/**
 * 自定义消息接收通道
 */
public interface MySink02 {

    String MY_INPUT = "default.message";

    @Input(MY_INPUT)
    SubscribableChannel myInput();

}
```

8.4.2 配置文件

`消息生产者`

```yaml
server:
  port: 8001 # 端口

spring:
  application:
    name: stream-producer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: lizongzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
```

消息消费者

```yaml
server:
  port: 8002 # 端口

spring:
  application:
    name: stream-consumer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: lizongzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
```

8.4.3 代购重构

消息生产者 `MyMessageProducer02.java`

```java
package com.example.producer;

import com.example.channel.MySource02;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;

/**
 * 消息生产者
 */
@Component
@EnableBinding(MySource02.class)
public class MyMessageProducer02 {

    @Autowired
    private MySource02 mySource02;

    /**
     * 发送消息
     *
     * @param message
```

```
     */
    public void send(String message) {
        mySource02.myOutput().send(MessageBuilder.withPayload(message).build());
    }

}
```

消息消费者 `MyMessageConsumer02.java`

```
package com.example.consumer;

import com.example.channel.MySink02;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.stereotype.Component;

/**
 * 消息消费者
 */
@Component
@EnableBinding(MySink02.class)
public class MyMessageConsumer02 {

    /**
     * 接收消息
     *
     * @param message
     */
    @StreamListener(MySink02.MY_INPUT)
    public void receive(String message) {
        System.out.println("message = " + message);
    }

}
```

8.4.4 单元测试

`MessageProducerTest.java`

```
package com.example;

import com.example.producer.MyMessageProducer02;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = {StreamProducerApplication.class})
public class MessageProducerTest {

    @Autowired
    private MyMessageProducer02 myMessageProducer02;

    @Test
    public void testMySend02() {
```

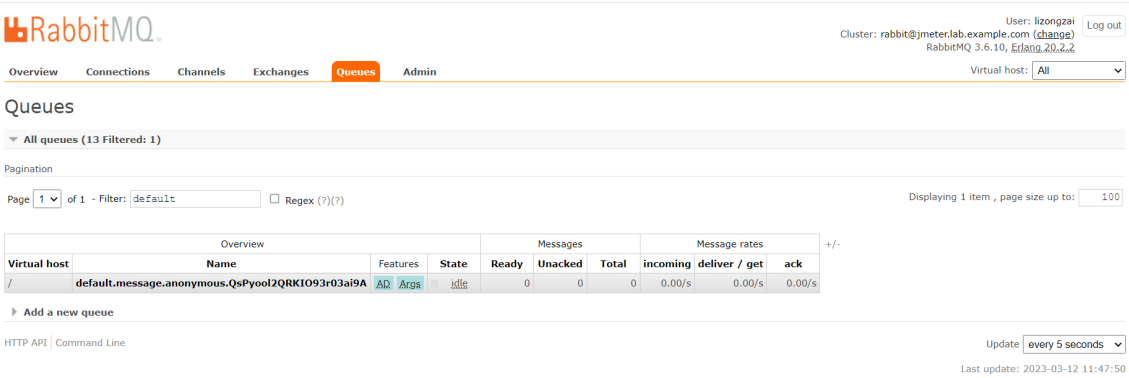
```
        myMessageProducer02.send("约定大于配置");
    }

}
```

8.4.5 访问接口

启动消息消费者，运行单元测试，消息消费者控制台打印结果如下：

```
message = 约定大于配置
```

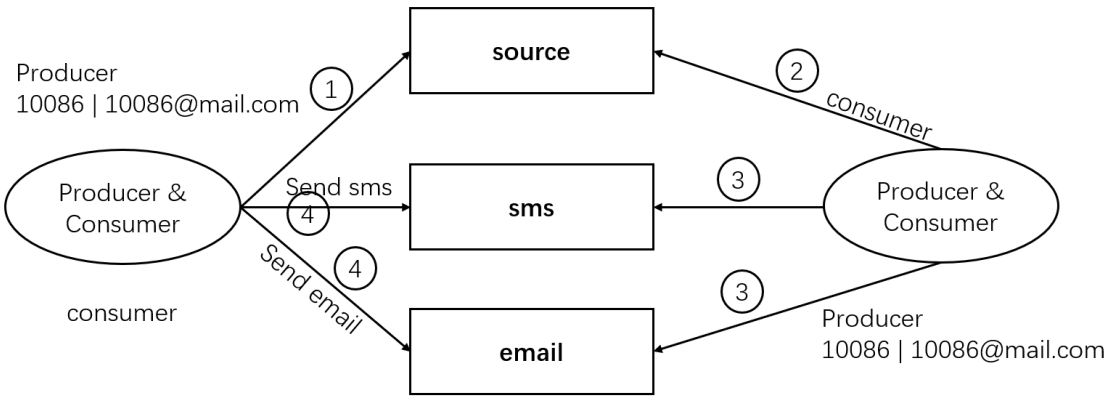
RabbitMQ 界面如下：



9. 短信邮件发送案例

一个消息驱动微服务应用可以既是消息生产者又是消息消费者。接下来模拟一个短信邮件发送的消息处理过程：

- 原始消息发送至 `source.message` 交换机；
- 消息驱动微服务应用通过 `source.message` 交换机接收原始消息，经过处理分别发送至 `sms.message` 和 `email.message` 交换机；
- 消息驱动微服务应用通过 `sms.message` 和 `email.message` 交换机接收处理后的消息并发送短信和邮件。



9.1 创建消息通道

发送原始消息，接收处理后的消息并发送短信和邮件的消息驱动微服务应用。

```
package com.example.channel;

import org.springframework.cloud.stream.annotation.Input;
import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.SubscribableChannel;

/**
 * 自定义消息通道
 */
public interface MyProcessor {

    String SOURCE_MESSAGE = "source.message";
    String SMS_MESSAGE = "sms.message";
    String EMAIL_MESSAGE = "email.message";

    @Output(SOURCE_MESSAGE)
    MessageChannel sourceOutput();

    @Input(SMS_MESSAGE)
    SubscribableChannel smsInput();

    @Input(EMAIL_MESSAGE)
    SubscribableChannel emailInput();

}
```

接收原始消息，经过处理分别发送短信和邮箱的消息驱动微服务应用。

```
package com.example.channel;

import org.springframework.cloud.stream.annotation.Input;
import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.SubscribableChannel;

/**
 * 自定义消息通道
 */
public interface MyProcessor {

    String SOURCE_MESSAGE = "source.message";
    String SMS_MESSAGE = "sms.message";
    String EMAIL_MESSAGE = "email.message";

    @Input(SOURCE_MESSAGE)
    MessageChannel sourceOutput();
```

```
    @Output(SMS_MESSAGE)
    SubscribableChannel smsOutput();

    @Output(EMAIL_MESSAGE)
    SubscribableChannel emailOutput();

}
```

9.2 配置文件

约定大于配置，配置文件只修改端口和应用名称即可，其他配置一致。

```
spring:
  application:
    name: stream-producer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: lizongzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
```

```
spring:
  application:
    name: stream-consumer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: lizongzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
```

9.3 消息驱动微服务A

9.3.1 发送消息

发送原始消息 `10086|10086@email.com` 至 `source.message` 交换机。

```
package com.example.producer;

import com.example.channel.MyProcessor;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;

/**
 * 消息生产者
```

```
 */
@Component
@EnableBinding(MyProcessor.class)
public class SourceMessageProducer {

    private Logger logger = LoggerFactory.getLogger(SourceMessageProducer.class);

    @Autowired
    private MyProcessor myProcessor;

    /**
     * 发送原始消息
     *
     * @param sourceMessage
     */
    public void send(String sourceMessage) {
        logger.info("原始消息发送成功，原始消息为：{}", sourceMessage);

 myProcessor.sourceOutput().send(MessageBuilder.withPayload(sourceMessage).build());
    }

}
```

9.3.2 消息接收

接收处理后的消息并发送短信和邮件。

```
package com.example.consumer;

import com.example.channel.MyProcessor;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.stereotype.Component;

/**
 * 消息消费者
 */
@Component
@EnableBinding(MyProcessor.class)
public class SmsAndEmailMessageConsumer {

    private Logger logger = LoggerFactory.getLogger(SmsAndEmailMessageConsumer.class);

    /**
     * 接收消息 电话号码
     *
     * @param phoneNum
     */
    @StreamListener(MyProcessor.SMS_MESSAGE)
    public void receiveSms(String phoneNum) {
        logger.info("电话号码为：{}，调用短信发送服务，发送短信...", phoneNum);
    }

    /**
```

```
     * 接收消息 邮箱地址
     *
     * @param emailAddress
     */
    @StreamListener(MyProcessor.EMAIL_MESSAGE)
    public void receiveEmail(String emailAddress) {
        logger.info("邮箱地址为：{}，调用邮件发送服务，发送邮件...", emailAddress);
    }

}
```

9.4 消息驱动微服务B

9.4.1 消息接收

接收原始消息 `10086|10086@email.com` 处理后并发送至 `sms.message` 和 `email.message` 交换机。

```
package com.example.consumer;

import com.example.channel.MyProcessor;
import com.example.producer.SmsAndEmailMessageProducer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.stereotype.Component;

/**
 * 消息消费者
 */
@Component
@EnableBinding(MyProcessor.class)
public class SourceMessageConsumer {

    private Logger logger = LoggerFactory.getLogger(SourceMessageConsumer.class);

    @Autowired
    private SmsAndEmailMessageProducer smsAndEmailMessageProducer;

    /**
     * 接收原始消息，处理后并发送
     *
     * @param sourceMessage
     */
    @StreamListener(MyProcessor.SOURCE_MESSAGE)
    public void receive(String sourceMessage) {
        logger.info("原始消息接收成功，原始消息为：{}", sourceMessage);
        // 发送消息 电话号码
        smsAndEmailMessageProducer.sendSms(sourceMessage.split("\\|")[0]);
        // 发送消息 邮箱地址
        smsAndEmailMessageProducer.sendEmail(sourceMessage.split("\\|")[1]);
    }
```

```
}
```

9.4.2 发送消息

发送电话号码 `10086` 和邮箱地址 `10086@email.com` 至 `sms.message` 和 `email.message` 交换机。

```
package com.example.producer;

import com.example.channel.MyProcessor;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;

/**
 * 消息生产者
 */
@Component
@EnableBinding(MyProcessor.class)
public class SmsAndEmailMessageProducer {

    private Logger logger = LoggerFactory.getLogger(SmsAndEmailMessageProducer.class);

    @Autowired
    private MyProcessor myProcessor;

    /**
     * 发送消息 电话号码
     *
     * @param smsMessage
     */
    public void sendSms(String smsMessage) {
        logger.info("电话号码消息发送成功，消息为：{}", smsMessage);
        myProcessor.smsOutput().send(MessageBuilder.withPayload(smsMessage).build());
    }

    /**
     * 发送消息 邮箱地址
     *
     * @param emailMessage
     */
    public void sendEmail(String emailMessage) {
        logger.info("邮箱地址消息发送成功，消息为：{}", emailMessage);

 myProcessor.emailOutput().send(MessageBuilder.withPayload(emailMessage).build());
    }

}
```

9.5 测试

9.5.1 单元测试

`MessageProducerTest.java`

```
package com.example;

import com.example.producer.SourceMessageProducer;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = {StreamProducerApplication.class})
public class MessageProducerTest {

    @Autowired
    private SourceMessageProducer sourceMessageProducer;

    @Test
    public void testSendSource() {
        sourceMessageProducer.send("10086|10086@email.com");
    }

}
```

9.5.2 访问

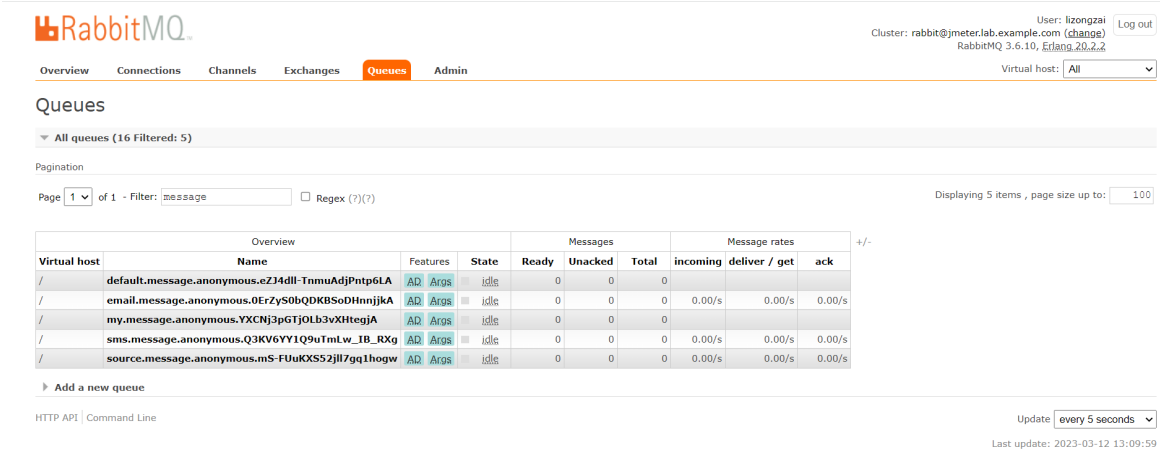
消息生产者驱动微服务 A 控制台打印结果如下：

```
电话号码为：10086，调用短信发送服务，发送短信...
邮箱地址为：10086@email.com，调用邮件发送服务，发送邮件...
```

消息消费者驱动微服务 B 控制台打印结果如下：

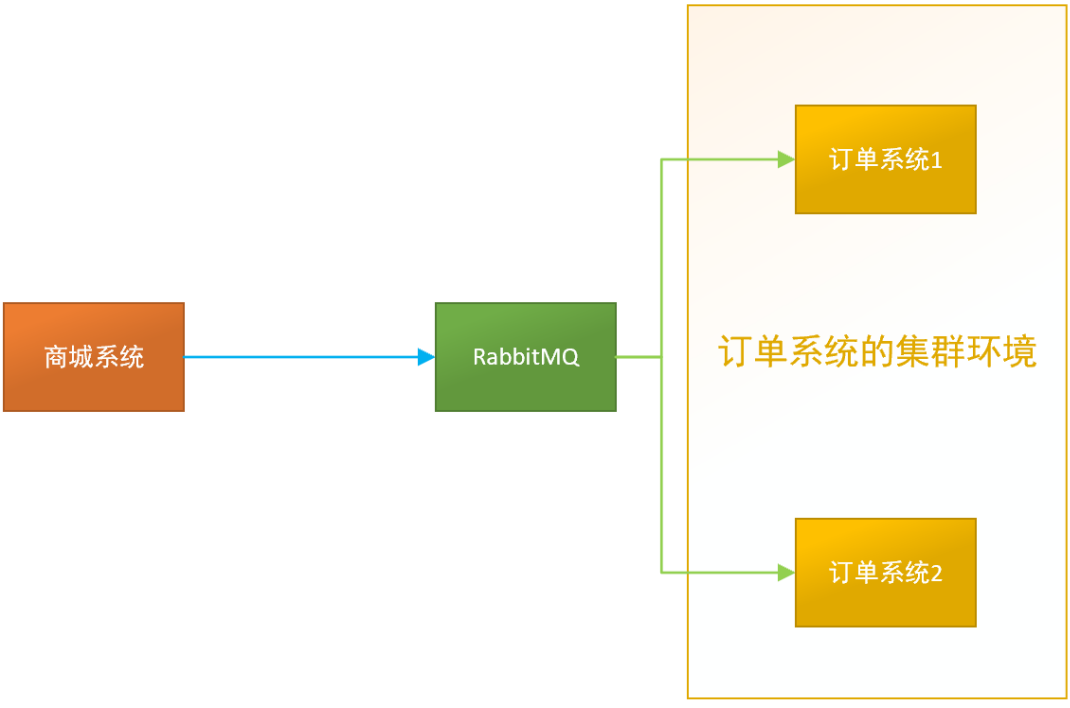
```
原始消息接收成功，原始消息为：10086|10086@email.com
电话号码消息发送成功，消息为：10086
邮箱地址消息发送成功，消息为：10086@email.com
```

RabbitMQ 界面如下：

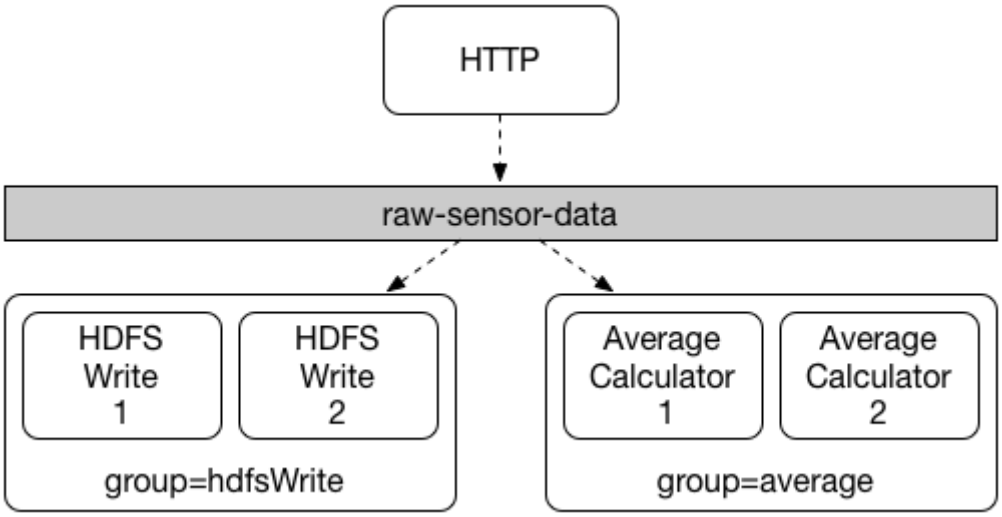


10. 消息分组

如果有多个消息消费者，那么消息生产者发送的消息会被多个消费者都接收到，这种情况在某些实际场景下是有很大问题的，比如在如下场景中，订单系统做集群部署，都会从 RabbitMQ 中获取订单信息，如果一个订单消息同时被两个服务消费，系统肯定会出现问题。为了避免这种情况，Stream 提供了消息分组来解决该问题。



在 Stream 中处于同一个 `group` 中的多个消费者是竞争关系，能够保证消息只会被其中一个应用消费。不同的组是可以消费的，同一个组会发生竞争关系，只有其中一个可以消费。通过 `spring.cloud.stream.bindings.<bindingName>.group` 属性指定组名。



10.1 问题演示

在 `stream-demo` 项目下创建 `stream-consumer02` 子项目。

项目代码使用入门案例中消息消费者的代码。

单元测试代码如下：

```
package com.example;

import com.example.producer.MessageProducer;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = {StreamProducerApplication.class})
public class MessageProducerTest {

    @Autowired
    private MessageProducer messageProducer;

    @Test
    public void testSend() {
        messageProducer.send("hello spring cloud stream");
    }

}
```

10.2 测试

运行单元测试发送消息，两个消息消费者控制台打印结果如下：

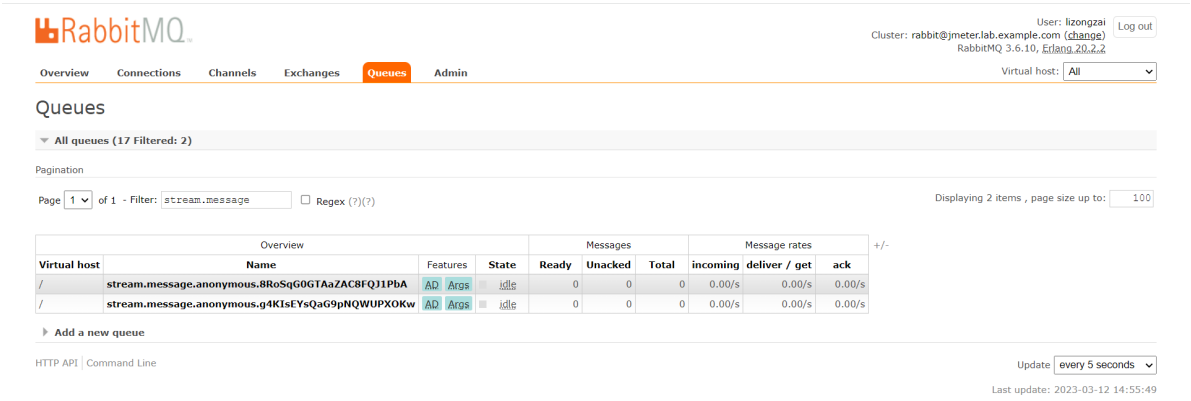
stream-consumer 的控制台：

```
message = hello spring cloud stream
```

stream-consumer02 的控制台：

```
message = hello spring cloud stream
```

通过结果可以看到消息被两个消费者同时消费了，原因是因为它们属于不同的分组，默认情况下分组名称是随机生成的，通过 RabbitMQ 也可以得知：



10.3 配置分组

stream-consumer 的分组配置为：`group-devops`

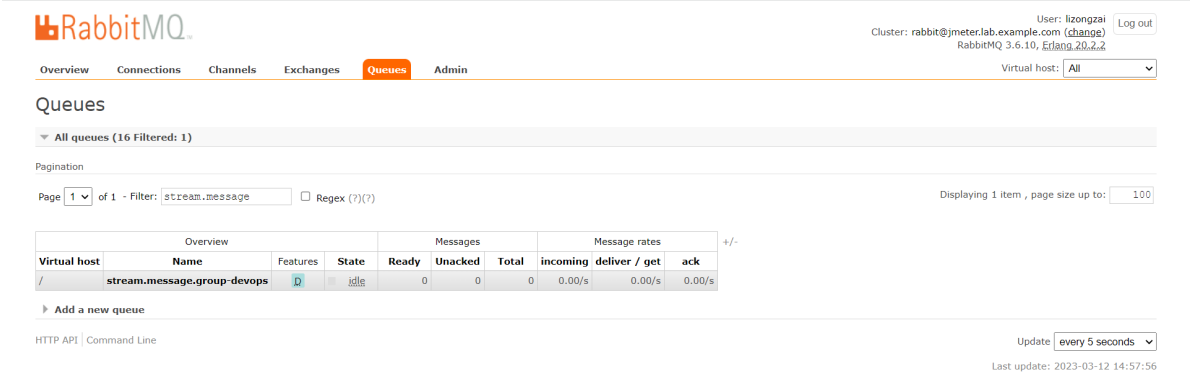
```yaml
server:
  port: 8002 # 端口

spring:
  application:
    name: stream-consumer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: lizongzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
  cloud:
    stream:
      bindings:
        # 消息接收通道
        # 与 org.springframework.cloud.stream.messaging.Sink 中的 @Input("input") 注解的 value 相同
        input:
          destination: stream.message # 绑定的交换机名称
          group: group-devops
```

stream-consumer02 的分组配置为：`group-devops`

```yaml
server:
  port: 8003 # 端口

spring:
  application:
    name: stream-consumer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: lizongzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
  cloud:
    stream:
      bindings:
        # 消息接收通道
        # 与 org.springframework.cloud.stream.messaging.Sink 中的 @Input("input") 注解的 value 相同
        input:
          destination: stream.message # 绑定的交换机名称
          group: group-devops
```

10.4 测试

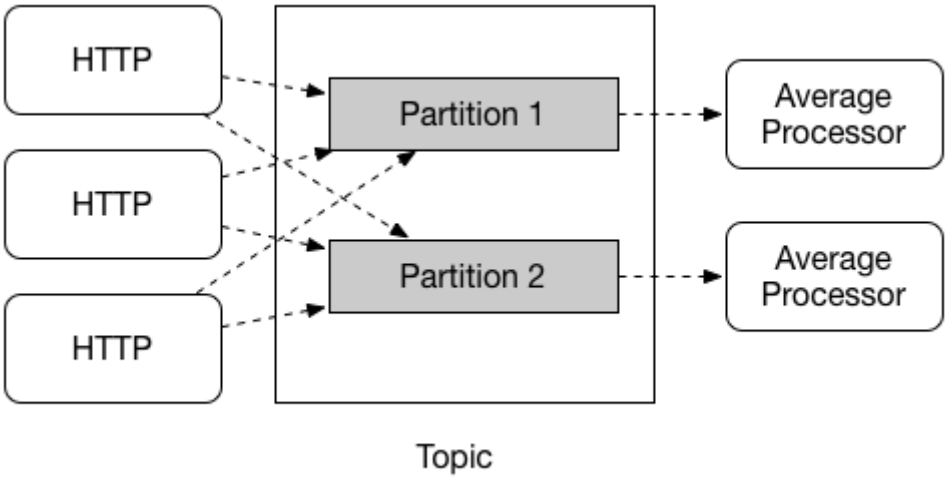
运行单元测试发送消息，此时多个消息消费者只有其中一个可以消费。RabbitMQ 结果如下：



11. 消息分区

通过消息分组可以解决消息被重复消费的问题，但在某些场景下分组还不能满足我们的需求。比如，同时有多条同一个用户的数据发送过来，我们需要根据用户统计，但是消息被分散到了不同的集群节点上了，这时我们就可以考虑使用**消息分区**了。

当生产者将消息发送给多个消费者时，保证同一消息始终由同一个消费者实例接收和处理。消息分区是对消息分组的一种补充。



11.1 问题演示

先给大家演示一下消息未分区的效果，单元测试代码如下：

```
package com.example;

import com.example.producer.MessageProducer;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = {StreamProducerApplication.class})
public class MessageProducerTest {

    @Autowired
    private MessageProducer messageProducer;

    @Test
    public void testSend() {
        for (int i = 1; i <= 10; i++) {
            messageProducer.send("hello spring cloud stream");
        }
    }

}
```

11.2 测试

运行单元测试发送消息，两个消息消费者控制台打印结果如下：

stream-consumer 的控制台：

```
message = hello spring cloud stream
message = hello spring cloud stream
message = hello spring cloud stream
message = hello spring cloud stream
message = hello spring cloud stream
```

stream-consumer02 的控制台：

```
message = hello spring cloud stream
message = hello spring cloud stream
message = hello spring cloud stream
message = hello spring cloud stream
message = hello spring cloud stream
```

假设这 10 条消息都来自同一个用户，正确的方式应该都由一个消费者消费所有消息，否则系统肯定会出现问题。为了避免这种情况，Stream 提供了消息分区来解决该问题。

11.3 配置分区

消息生产者配置**分区键的表达式规则**和**消息分区的数量**。

```yaml
server:
  port: 8001 # 端口

spring:
  application:
    name: stream-producer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: lizongzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
  cloud:
    stream:
      bindings:
        # 消息发送通道
        # 与 org.springframework.cloud.stream.messaging.Source 中的 @Output("output") 注解的 value 相同
        output:
          destination: stream.message # 绑定的交换机名称
          producer:
            partition-key-expression: payload # 配置分区键的表达式规则
            partition-count: 2 # 配置消息分区的数量
```

通过 `partition-key-expression` 参数指定分区键的表达式规则，用于区分每个消息被发送至对应分区的输出 `channel`。

该表达式作用于传递给 `MessageChannel` 的 `send` 方法的参数，该参数实现 `org.springframework.messaging.Message` 接口的 `GenericMessage` 类。

源码 `MessageChannel.java`

```
package org.springframework.messaging;

@FunctionalInterface
public interface MessageChannel {
    long INDEFINITE_TIMEOUT = -1L;

    default boolean send(Message<?> message) {
        return this.send(message, -1L);
    }

    boolean send(Message<?> var1, long var2);
}
```

源码 `GenericMessage.java`

```
package org.springframework.messaging.support;

import java.io.Serializable;
import java.util.Map;
import org.springframework.lang.Nullable;
import org.springframework.messaging.Message;
import org.springframework.messaging.MessageHeaders;
import org.springframework.util.Assert;
import org.springframework.util.ObjectUtils;

public class GenericMessage<T> implements Message<T>, Serializable {
    private static final long serialVersionUID = 4268801052358035098L;
    private final T payload;
    private final MessageHeaders headers;

    ...

}
```

如果 `partition-key-expression` 的值是 `payload`，将会使用所有放在 `GenericMessage` 中的数据作为分区数据。`payload` 是消息的实体类型，可以为自定义类型比如 `User`，`Role` 等等。

如果 `partition-key-expression` 的值是 `headers["xxx"]`，将由 `MessageBuilder` 类的 `setHeader()` 方法完成赋值，比如：

```
package com.example.producer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;

/**
 * 消息生产者
 */
```

```
@Component
@EnableBinding(Source.class)
public class MessageProducer {

    @Autowired
    private Source source;

    /**
     * 发送消息
     *
     * @param message
     */
    public void send(String message) {
        source.output().send(MessageBuilder.withPayload(message).setHeader("xxx",
0).build());
    }

}
```

消息消费者配置**消费者总数**和**当前消费者的索引**并**开启分区支持**。

`stream-consumer` 的 `application.yml`

```
server:
  port: 8002 # 端口

spring:
  application:
    name: stream-consumer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: lizongzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
  cloud:
    stream:
      instance-count: 2 # 消费者总数
      instance-index: 0 # 当前消费者的索引
      bindings:
        # 消息接收通道
        # 与 org.springframework.cloud.stream.messaging.Sink 中的 @Input("input") 注解的
value 相同
        input:
          destination: stream.message # 绑定的交换机名称
          group: group-devops
          consumer:
            partitioned: true # 开启分区支持
```

`stream-consumer02` 的 `application.yml`

```
server:
  port: 8003 # 端口

spring:
```

```yaml
  application:
    name: stream-consumer # 应用名称
  rabbitmq:
    host: 192.168.126.64  # 服务器 IP
    port: 5672            # 服务器端口
    username: lizongzai   # 用户名
    password: password    # 密码
    virtual-host: /       # 虚拟主机地址
  cloud:
    stream:
      instance-count: 2 # 消费者总数
      instance-index: 1 # 当前消费者的索引
      bindings:
        # 消息接收通道
        # 与 org.springframework.cloud.stream.messaging.Sink 中的 @Input("input") 注解的
value 相同
        input:
          destination: stream.message # 绑定的交换机名称
          group: group-devops
          consumer:
            partitioned: true # 开启分区支持
```

11.4 测试

运行单元测试发送消息，此时多个消息消费者只有其中一个可以消费所有消息。RabbitMQ 结果如下：