

## 27 构建消息驱动的微服务

更新时间：2019-07-24 14:39:23



人生太短，要干的事太多，我要争分夺秒。

——爱迪生

上篇文章和大家聊了 Spring Cloud Bus 的基本使用，同时给大家演示了如何使用 Spring Cloud Bus 实现配置文件的动态刷新，今天我们在上文的基础上，继续来学习消息驱动的微服务 Spring Cloud Stream。

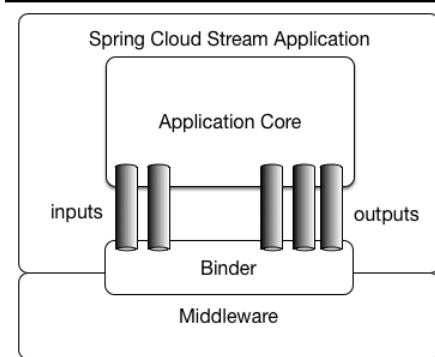
根据 Spring Cloud Stream 官方文档的介绍，Spring Cloud Stream 是一个用于构建消息驱动的微服务应用程序的框架。Spring Cloud Stream 构建于 Spring Boot 之上，用于创建独立的生产级 Spring 应用程序，并使用 Spring Integration 提供与消息代理的连接。它提供了来自多个供应商的中间件的固定配置，定义了发布、订阅、分组以及分区的概念。接下来我们就来向大家介绍下 Spring Cloud Stream 的具体用法。

### 核心概念

在 Spring Cloud Stream 中，我们的微服务通过 inputs 或者 outputs 来与 Spring Cloud Stream 中的 Binder 进行交互，而这里的 Binder 相当于微服务和消息中间件之间的一个粘合剂，这个 Binder 则可以负责与消息中间件如 RabbitMQ 或者 Kafka 进行交互，这个时候对于开发者而言，我们只需要关注微服务和 Spring Cloud Stream 之间的通信方式，即消息要怎么样发送、怎么样订阅，做好这些工作之后，剩下的事情就交给 Spring Cloud Stream 来做，它会帮助我们完成和消息中间件之间的交互。在整个过程中，有几个关键的概念，我们来了解下。

#### 通信模型

Spring Cloud Stream 应用程序由中间件驱动，应用程序通过 Spring Cloud Stream 提供的输入和输出通道与外界通信，整个过程通过中间件特定的 Binder 来实现，通道连接到外部代理，官网提供了一张比较形象的工作模型图，如下：



## Binder

Spring Cloud Stream 默认认为 Kafka 和 RabbitMQ 提供了 Binder 的实现。所谓的 Binder 实际上是一个抽象的概念，如上文所说，它是应用程序和消息中间件之间的一个粘合剂。使用 Binder，我们可以在程序运行时，动态修改消息的 destination，具体到 RabbitMQ 中就是 exchange，具体到 Kafka 中就是 topic，这些我们都可以通过外部属性或者其它 Spring Boot 支持的配置方式（如 application.properties 或 application.yaml）来实现，甚至不需要改变一行代码。

## Publish-Subscribe

消息驱动最经典的模式莫过于发布订阅了（Publish-Subscribe），在这个过程中数据通过共享主题广播，消息生产者通过某一个 topic 将消息广播出去，其它的微服务则通过订阅这个 topic 来消费消息，这种模式使得消息生产者和消费者进行了极大的解耦，未来如果有新的消息生产者加进来或者新的消息消费者加进来，都不必改变项目的原有架构！

## Consumer Groups

在真正的生产环境下，微服务通常都是集群化部署，而不会以单节点的方式运行在生产环境，当一个微服务进行集群化部署时，它的所有实例都会绑定到同一个消息通道的 Topic 上。那么在默认情况下，当消息生产者发出一条消息到绑定的 Topic 上时，这条消息会产生多个副本被每个消息消费者实例接收和处理，但是更多的情况下，我们可能只需要其中一个消息消费者来消费这个消息，此时我们通过 Consumer Groups 功能就能实现这个功能，分组之后，就能够确保消息只被某一个实例消费一次。

## Consumer Types

Spring Cloud Stream 支持两种类型的消费者：

- 消息驱动（有时称为异步）
- 轮询（有时称为同步）

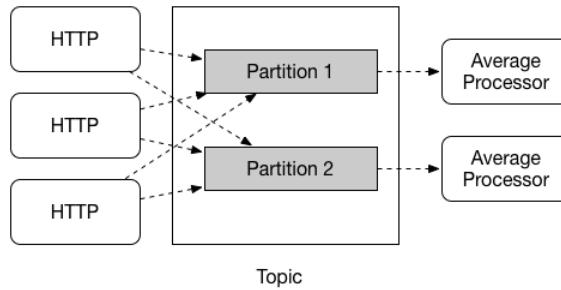
在2.0版之前，Spring Cloud Stream 仅支持异步使用者，即消息一旦可用就会传递，并且可以使用一个线程来处理它。如果需要控制处理消息的速率，需要使用同步使用者。

## Durability

Spring Cloud Stream 可以动态选择一个消息队列是持久化，还是 present。对于匿名订阅本质上是非持久的。对于某些绑定器实现（例如RabbitMQ），可以具有非持久的组订阅。

## Partitioning Support

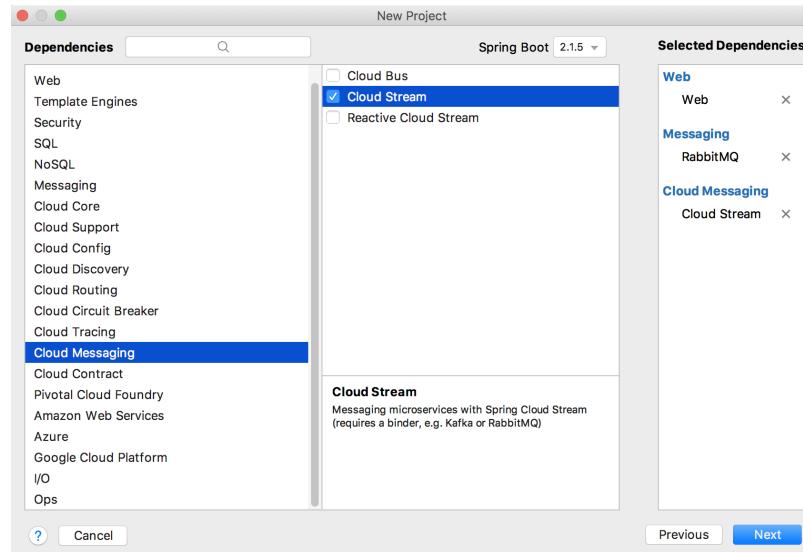
**Partitioning Support** 即分区支持，Spring Cloud Stream 支持在给定应用程序的多个实例之间对数据进行分区。例如在某些场景下，我们需要同一个特征的数据被同一个实例消费，这种情况下，单纯的消息分组就不能满足我的需求了，于是 Spring Cloud Stream 又引入了消息分区的概念，通过消息分区，可以确保同一特征的消息始终被同一个消息消费者处理。



## Spring Cloud Stream 基本用法

前面和大家大概理了一下 Spring Cloud Stream 的一些核心概念，接下来我们就来通过一个两个简单的案例向大家来演示一下 Spring Cloud Stream 中基本的消息收发机制。

那么在开始学习之前，我们需要先来创建一个工程，这里我直接创建一个名为 `helostream` 的工程，创建时候添加三个依赖 `Web`、`RabbitMQ` 和 `Cloud Stream`，如下：



Spring Boot 创建完成后，接下来我们还需要确保 Docker 容器中的 RabbitMQ 已经在运行了，然后我们就可以来看看 Spring Cloud Stream 的一个基本用法了。

### 基本用法

创建完成后，首先我们来在 `application.properties` 中配置一下 RabbitMQ 的基本信息：

```
spring.rabbitmq.password=guest
spring.rabbitmq.username=guest
spring.rabbitmq.host=127.0.0.1
spring.rabbitmq.port=5672
```

接下来我们来创建一个简单的消息接收器，如下：

```

@EnableBinding(Sink.class)
public class MsgReceiver {
    @StreamListener(Sink.INPUT)
    public void receive(Object payload) {
        System.out.println("Received:" + payload);
    }
}

```

在这里，我们首先使用了 `@EnableBinding` 注解实现对消息通道的绑定，哪个消息通道呢？就是参数 `Sink`，事实上，这里的参数可以有多个，`Sink` 是一个默认定义的消息通道。接下来我们在 `MsgReceiver` 类中定义了 `receive` 方法，并在该方法上添加了 `@StreamListener` 注解，该注解表示该方法为消息中间件上数据流的事件监听器，`Sink.INPUT` 参数表示这是 `input` 消息通道上的监听处理器（在 `Spring Cloud Stream` 中可以有多个通道）。

定义完成后，启动我们的 `Spring Boot` 项目，启动日志中，可以看到如下信息：

```

: Channel 'application.errorchannel' has 1 subscriber(s).
: started _org.springframework.integration.errorLogger
: declaring queue for inbounds: input.anonymous.qFU-SFBeQjizUhfH2nt1Q, bound to: input
: Attempting to connect to: [127.0.0.1:5672]
[...] [org.springframework.amqp.rabbit.RabbitConnectionFactory] - Connected to [tcp://127.0.0.1:5672/SimpleConnection@6f58d55c [delegate=amqp://guest@127.0.0.1:5672/, localPort= 59784]]
: Registering MessageChannel input.anonymous.qFU-SFBeQjizUhfH2nt1Q.errors
: Channel 'application.input.anonymous.qFU-SFBeQjizUhfH2nt1Q.errors' has 1 subscriber(s).
: Channel 'application.input.anonymous.qFU-SFBeQjizUhfH2nt1Q.errors' has 2 subscriber(s).
: started inbound.input.anonymous.qFU-SFBeQjizUhfH2nt1Q
: Tomcat started on port(s): 8080 [http] with context path ''

```

看到这条日志说明我们的 `Spring Boot` 工程已经连接上 `RabbitMQ` 了。接下来我们登录 `RabbitMQ` 管理页面，通过管理页面发送一条消息，看看我们的 `Spring Boot` 工程是否能够收到这条消息，登录 `RabbitMQ` 消息管理页面之后，在 `Queue` 选项卡中，找到我们刚刚的消息通道，如下：

Name	Features	State	Messages	Message rates				
			Ready	Unacked	Total	incoming	deliver / get	ack
age-queue	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
hello	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
hello-javaboy	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
huawei	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
input.anonymous.J5HBK8bT32_wlf7AYLmjQ	AD	Excl	0	0	0	0.00/s	0.00/s	0.00/s
name-queue	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
phone	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
queue-one	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
queue-two	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
xiaomi	D	idle	0	0	0	0.00/s	0.00/s	0.00/s

点进去，然后找到 `Publish Message`，发送一条消息，如下：

Delivery mode: 1 - Non-persistent

Headers:  =  String

Properties:  =

Payload: hello stream

**Publish message**

发送完成后，我们发现 `Spring Boot` 项目的控制台已经打印出日志了，表示消息已经成功发送并且消费了。

## 自定义消息通道

上面的案例是使用 **RabbitMQ** 的控制台进行消息的发送，在真正的生产环境中，这种需求一般不多，我们都是通过代码进行消息的发送。接下来我们就看看如何自定义消息通道。

首先我们创建一个类名为 **MyChannel**，如下：

```
public interface MyChannel {  
    String INPUT = "mychannel-input";  
    String OUTPUT = "mychannel-output";  
  
    @Output(OUTPUT)  
    MessageChannel output();  
  
    @Input(INPUT)  
    SubscribableChannel input();  
}
```

代码解释：

- 首先我们定义了两个消息通道的名字，两个名字是不一样的；
- 接下来定义了一个消息输出通道，什么是输出通道呢？其实就是消息发送通道；
- 最后定义了一个消息输入通道，所谓的消息输入通道就是消息接收通道；
- 一会我们在消息发送通道发送消息，在消息接收通道接收消息，有人可能会说这两个明明都不在同一个通道，能收到消息吗？不在同一个通道当然是收不到消息的，但是从 **Sprint Cloud Finchley** 版开始，默认使用通道名作为相应的实例名，因此这里我们不能使用相同的通道名，否则实例将创建失败，进而导致项目启动失败。一会我们将在 **application.properties** 中做一些额外配置，使消息接收通道能够收到消息。

接下来我们再定义一个消息消费者，消费我们自己通道上的消息，如下：

```
@EnableBinding(MyChannel.class)  
public class MsgReceiver2 {  
    @StreamListener(MyChannel.INPUT)  
    public void receive(Object payload) {  
        System.out.println("Received2:" + payload);  
    }  
}
```

这个消息消费者的定义和我们前面的定义差不多，主要是监控的通道变了，变成了我们自己的消息通道了。

然后再来定义一个消息生产者，如下：

```
@RestController  
public class HelloController {  
    @Autowired  
    MyChannel myChannel;  
  
    @GetMapping("/test1")  
    public void hello() {  
        myChannel.output().send(MessageBuilder.withPayload("hello stream!").build());  
    }  
}
```

调用 **MyChannel** 对象中的 **output** 方法，就可以成功发送一条消息出去，这条消息将在消息发送通道上发出！消息本身使用 **MessageBuilder** 来构建。

这样消息就能成功收发了吗？理论上来说是没错的，但是运行之后你会发现收不到消息，怎么回事呢？消息发送成功没？成功了！消息收到了吗？没有！什么原因呢？这就是我们刚刚开头说的，消息收发目前不在一个通道上，所以发出去的消息，没法收到，那么怎么办呢？很简单，只需要我们在配置文件 `application.properties` 中，再添加如下两行配置：

```
spring.cloud.stream.bindings.mychannel-input.destination=javaboy-topic  
spring.cloud.stream.bindings.mychannel-output.destination=javaboy-topic
```

这样就使得我们的消息收发在同一个通道上了，此时，启动 Spring Boot 项目，然后在浏览器中发送 `http://localhost:8080/test1` 请求，此时，我们可以看到 IntelliJ IDEA 控制台打印出来日志，表示消息已经被收到了：

```
2019-05-19 20:12:02.601 INFO 90125 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : 2019-05-19 20:12:02.622 INFO 90125 --- [nio-8080-exec-1] o.s.a.r.c.CacheManager : 2019-05-19 20:12:02.629 INFO 90125 --- [nio-8080-exec-1] o.s.a.r.c.CacheManager : 2019-05-19 20:12:02.631 INFO 90125 --- [nio-8080-exec-1] o.s.amqp.rabbitmq.listener.BlockingQueueConsumer : 2019-05-19 20:12:02.631 INFO 90125 --- [nio-8080-exec-1] o.s.amqp.rabbitmq.listener.BlockingQueueConsumer : Received2:hello stream!
```

这样一个简单的例子向大家展示了自定义消息通道。

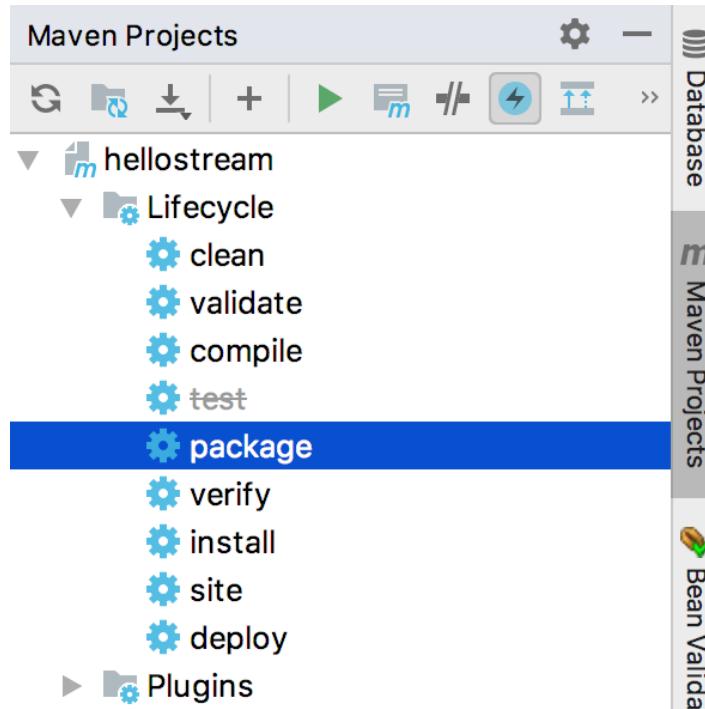
### 消息分组

消息分组我们在文章一开头的时候和大家说过，就是一条消息默认情况下被同一个微服务的所有实例消费（该微服务集群化部署），有的时候我们只需要被其中一个实例消费即可，此时我们需要进行消息分组配置。

配置方式很简单，在 `application.properties` 中添加如下配置：

```
spring.cloud.stream.bindings.mychannel-input.group=g1  
spring.cloud.stream.bindings.mychannel-output.group=g1
```

表示消息输入输出通道都是属于消费组 `g1` 的，配置完成后，我们将 Spring Boot 项目打成 `jar` 包，点击右边 Maven Project 中的 `package` 按钮，如下：



打包成功之后，分别在定位到 `jar` 包所在的目录，分别执行如下两行命令，启动两个端口不同的实例：

```
java -jar helostream-0.0.1-SNAPSHOT.jar --server.port=8080  
java -jar helostream-0.0.1-SNAPSHOT.jar --server.port=8081
```

启动成功之后，我们再次调用 <http://localhost:8080/test1> 接口，发现每次消息只被一个实例消费（每次只有一个实例的控制台有日志打印出来）。

## 消息分区

还有一个概念叫做消息分区，就是说具有相同特征的消息总是被同一个实例处理，单纯的消息分组是无法实现这个功能的。在前面的消息分组中，相同特征的消息也会被发送到不同的实例上去执行。如果想发送到一个实例上去执行，我们只需要添加如下配置即可（下面的配置是在消息分组的基础上配置的）：

```
spring.cloud.stream.bindings.mychannel-input.consumer.partitioned=true  
spring.cloud.stream.instance-count=2  
spring.cloud.stream.instance-index=0  
spring.cloud.stream.bindings.mychannel-output.producer.partition-key-expression=1  
spring.cloud.stream.bindings.mychannel-output.producer.partitionCount=2
```

代码解释：

- 首先第一行配置表示开启消息分区；
- 第二行配置消息消费者实例的个数；
- 第三行配置表示当前实例的下标；
- 第四行配置表示这个消息将被下标为 1 的消息消费者所消费；
- 第五行表示消费端的节点数量为 2；
- 由于消息消费者和消息生产者在同一个项目中，因此这里的配置我写在了一起；如果消息消费者和生产者是两个项目，那么前三行消息消费者相关的配置写在消息消费者中，后两行消息生产者相关的配置写在消息生产者中。

配置完成后，我们将项目重新打包，然后执行如下命令启动两个消息消费者实例：

```
java -jar helostream-0.0.1-SNAPSHOT.jar --server.port=8080 --spring.cloud.stream.instance-index=0  
java -jar helostream-0.0.1-SNAPSHOT.jar --server.port=8081 --spring.cloud.stream.instance-index=1
```

注意在启动命令中需要标记实例的 ID，即 `spring.cloud.stream.instance-index` 参数。启动成之后，我们再次调用 <http://localhost:8080/test1> 接口发送消息，消息就只会被 ID 为 1 的实例所接收。

## 小结

好了，本文主要向读者介绍了 Spring Cloud Stream 中的一些核心概念，使大家对于 Spring Cloud Stream 的用法有一个基本认识，然后向大家介绍了 Spring Cloud Stream 的一些基本用法，包括默认的消息发送、自定义消息通道、消息分组以及消息分区等，通过这几个案例，相信大家对于 Spring Cloud Stream 已经有了一个基本的认知。



目前暂无任何讨论