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的微服务应用。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-订阅、消费组以及消息分区这三个核心概念。简单的说，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Spring Cloud Stream，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。由于Spring Cloud Stream基于Spring Boot实现，所以它秉承了Spring Boot的优点，实现了自动化配置的功能帮忙我们可以快速的上手使用，但是目前为止Spring Cloud Stream只支持下面两个著名的消息中间件的自动化配置：

- RabbitMQ
- Kafka

实战练习

下面我们通过构建一个简单的示例来对Spring Cloud Stream有一个初步认识。该示例主要目标是构建一个基于Spring Boot的微服务应用，这个微服务应用将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，在进行下面步骤之前请先确认已经在本地安装了RabbitMQ。

构建一个Spring Cloud Stream消费者

- 创建一个基础的Spring Boot工程，命名为：stream-hello
- 编辑pom.xml中的依赖关系，引入Spring Cloud Stream对RabbitMQ的支持：

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

- 创建用于接收来自RabbitMQ消息的消费者SinkReceiver，具体如下：

```
@EnableBinding(Sink.class)
public class SinkReceiver {
    private static Logger logger = LoggerFactory.getLogger(SinkReceiver.class);

    @StreamListener(Sink.INPUT)
    public void receive(Object payload) {
        logger.info("接收到信息: " + payload);
    }
}
```

- 创建应用主类：

```
@SpringBootApplication
public class StreamHelloApplication {
```

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

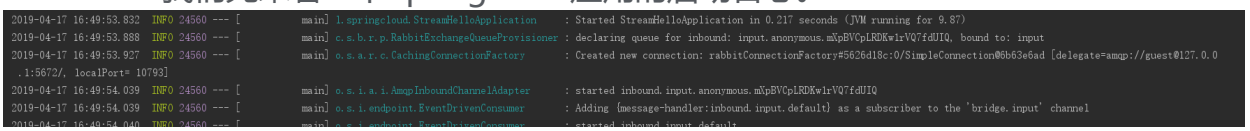
- 创建配置文件:

```
server.port=2401
```

到这里，我们快速入门示例的编码任务就已经完成了。下面我们分别启动RabbitMQ以及该Spring Boot应用，然后做下面的试验，看看它们是如何运作的。

手工测试验证

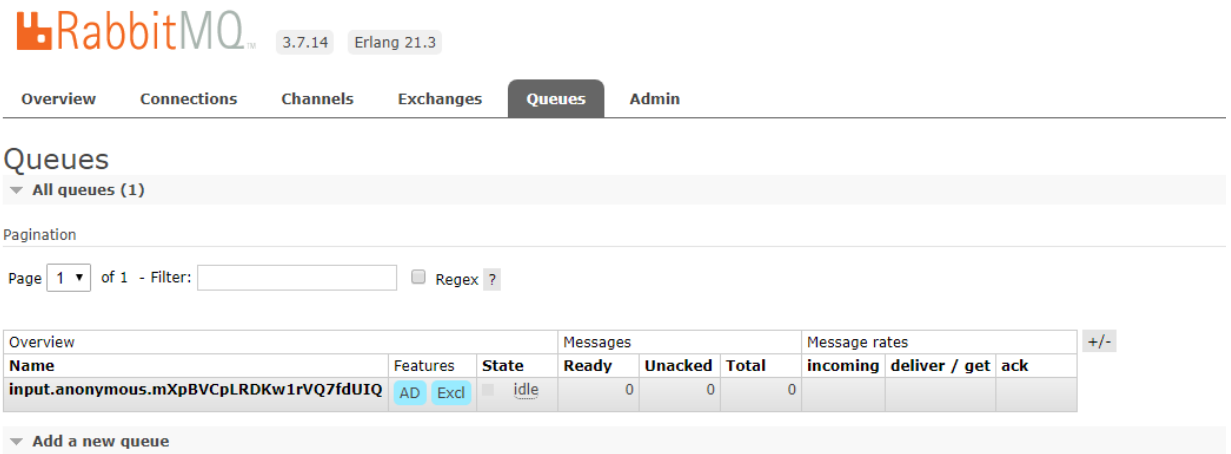
- 我们先来看一下Spring Boot应用的启动日志。



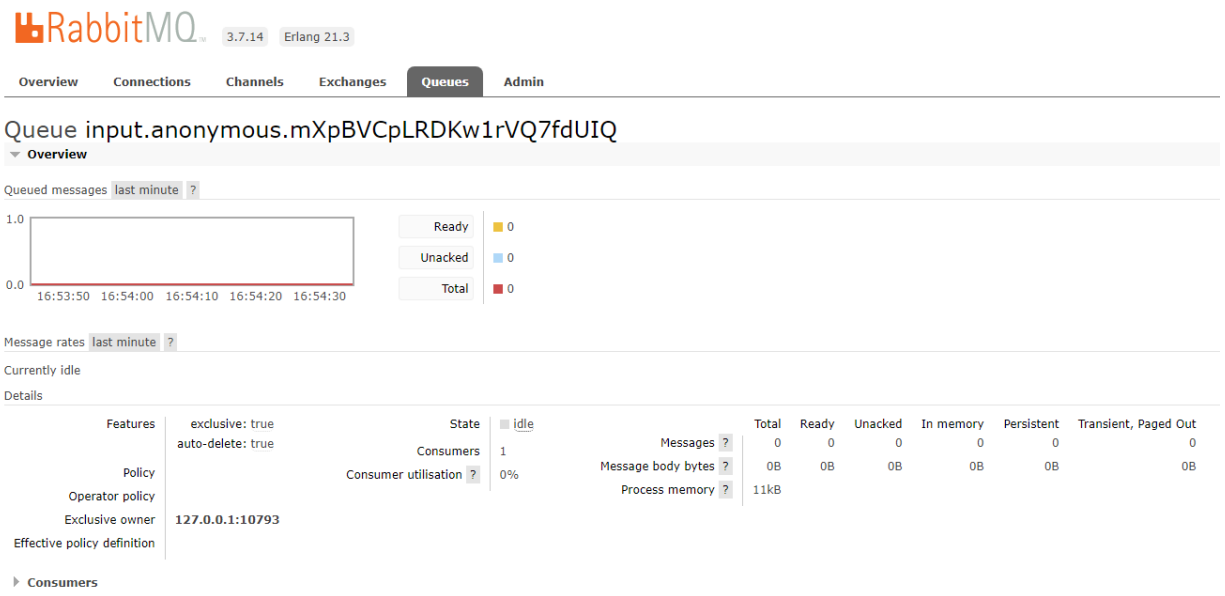

```
2019-04-17 16:49:53.927  INFO 24560 --- [           main]
o.s.a.r.c.CachingConnectionFactory       : Created new connection:
rabbitConnectionFactory#5626d18c:0/SimpleConnection@6b63e6ad
[delegate=amqp://guest@127.0.0.1:5672/, localPort= 10793]
2019-04-17 16:49:54.039  INFO 24560 --- [           main]
o.s.i.a.i.AmqpInboundChannelAdapter      : started
inbound.input.anonymous.mXpBVCpLRDKw1rVQ7fdUIQ
2019-04-17 16:49:54.039  INFO 24560 --- [           main]
o.s.i.endpoint.EventDrivenConsumer       : Adding {message-
handler:inbound.input.default} as a subscriber to the 'bridge.input' channel
2019-04-17 16:49:54.040  INFO 24560 --- [           main]
o.s.i.endpoint.EventDrivenConsumer       : started inbound.input.default
```

从上面的日志内容中，我们可以获得以下信息：

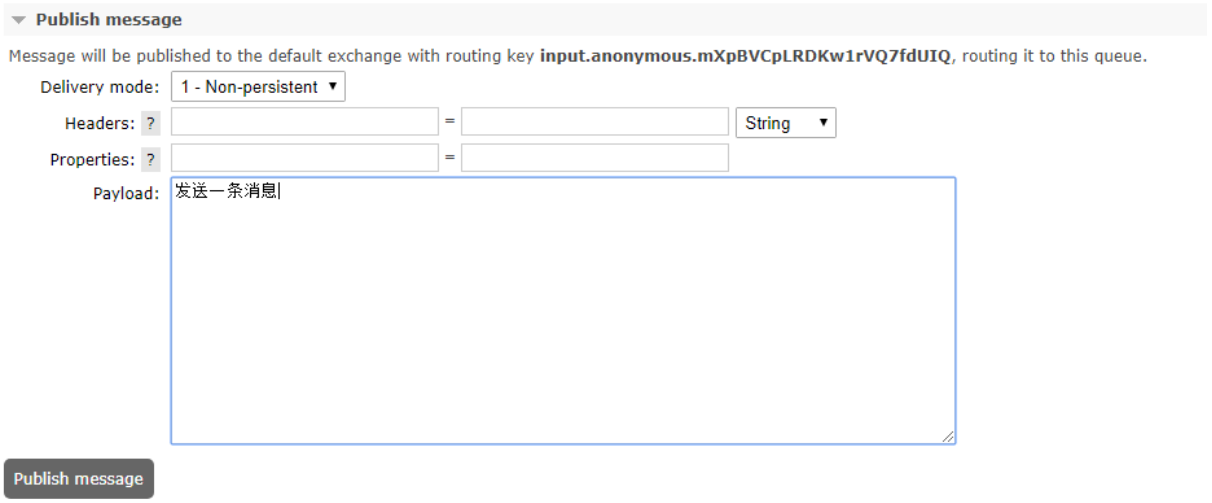
- 使用guest用户创建了一个指向127.0.0.1:15672位置的RabbitMQ连接，在RabbitMQ的控制台中我们也可以发现它。



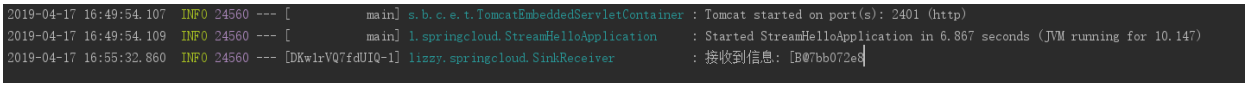

- 声明了一个名为input.anonymous.mXpBVCpLRDKw1rVQ7fdUIQ的队列，并通过RabbitMessageChannelBinder将自己绑定为它的消费者。这些信息我们也能在RabbitMQ的控制台中发现它们。




下面我们可以在RabbitMQ的控制台中进入input.anonymous.mXpBVCpLRDKw1rVQ7fdUIQ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。




此时，我们可以在当前启动的Spring Boot应用程序的控制台中看到下面的内容：




我们可以发现在应用控制台中输出的内容就是SinkReceiver中receive方法定义的，而输出的具体内容则是来自消息队列中获取的对象。这里由于我们没有对消息进行序列化，所以输出的只是该对象的引用，在后面的小节中我们会详细介绍接收消息后的处理。

在顺利完成上面快速入门的示例后，我们简单解释一下上面的步骤是如何将我们的Spring Boot应用连接上RabbitMQ来消费消息以实现消息驱动业务逻辑的。

首先，我们对Spring Boot应用做的就是引入spring-cloud-starter-stream-rabbit依赖，该依赖包是Spring Cloud Stream对RabbitMQ支持的封装，其中包含了对RabbitMQ的自动化配置等内容。从下面它定义的依赖关系中，我们还可以知道它等价于spring-cloud-stream-binder-rabbit依赖。

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
```

```
        <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
    </dependency>
</dependencies>
```

接着，我们再来看看这里用到的几个Spring Cloud Stream的核心注解，它们都被定义在SinkReceiver中：

- @EnableBinding，该注解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。在上面的例子中，我们通过@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，它的源码如下：

```
public interface Sink {
    String INPUT = "input";
    @Input(Sink.INPUT)
    SubscribableChannel input();
}
```

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，实际使用时我们也可以自己通过@Input和@Output注解来定义绑定消息通道的接口。当我们需要为@EnableBinding指定多个接口来绑定消息通道的时候，可以这样定义：@EnableBinding(value = {Sink.class, Source.class})。

- @StreamListener：该注解主要定义在方法上，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。在上面的例子中，我们通过@StreamListener(Sink.INPUT)注解将receive方法注册为对input消息通道的监听处理器，所以当我们在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。

### 编写消费消息的单元测试用例

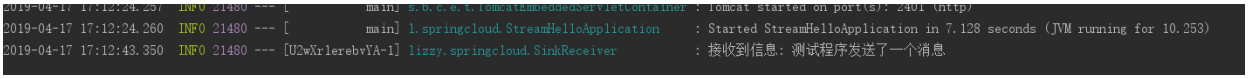
上面我们通过RabbitMQ的控制台完成了发送消息来验证了消息消费程序的功能，虽然这种方法比较low，但是通过上面的步骤，相信大家对RabbitMQ和Spring Cloud Stream的消息消费已经有了一些基础的认识。下面我们通过编写生产消息的单元测试用例来完善我们的入门内容。

- 在上面创建的工程中创建单元测试类：

```
@RunWith(SpringRunner.class)
@EnableBinding(value = {SinkSender.class})
public class StreamHelloApplicationTest {

    @Autowired
    private SinkSender sinkSender;

    @Test
    public void sinkSenderTester() {
```

```
        sinkSender.output().send(MessageBuilder.withPayload("测试程序发送了一个消息").build());
    }
}

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

- 在应用了上面的消息消费者程序之后，运行这里定义的单元测试程序，我们马上就能在消息消费者的控制台中收到下面的内容：




在上面的单元测试中，我们通过`@Output(SinkSender.OUTPUT)`定义了一个输出通过，而该输出通道的名称为`input`，与前文中的Sink中定义的消费通道同名，所以这里的单元测试与前文的消费者程序组成了一对生产者与消费者。