## 一站式微服务解决方案Spring Cloud Alibaba

主讲：Cat 老师

北京动力节点教育科技有限公司  
2020 • 北京

动力节点•版权所有•禁止传播

## 快速回顾

1、Spring家族开源项目梳理

2、Spring Cloud下的开源项目梳理

3、Spring Cloud Alibaba下的开源及商业项目梳理

4、微服务的基础模型：服务消费者-注册中心-服务提供者

5、What is Nacos？

6、Nacos的运行环境部署（Java写的，springboot项目）

7、Nacos的后台web管控台

8、Nacos作为注册中心注册服务

9、Nacos作为注册中心发现/订阅服务

10、服务消费者负载均衡调用服务提供者（ribbon）restTemple、feign（openfeign）

11、Nacos宕机时服务消费者缓存注册中心信息

12、Nacos作为配置中心存储项目各种配置

13、Nacos作为配置中心支持自动配置刷新（不需要重启应用）

14、Nacos配置中心DataId+Group+Properties/yaml+配置内容（比较灵活）

15、Nacos配置中心多环境配置（profile）

即${spring.application.name}-${profile}.${file-extension:properties}

16、Nacos服务配置数据模型（命名空间、Group、Data Id）

17、Nacos数据持久化（mysql）

18、Ncaos集群部署（nginx）

19、主要调用方式：restTemplate、feign、ribbon（spring cloud）

20、流量控制Sentinel（流控、降级、热点、系统、授权 规则）

21、Sentinel Darshboard通信原理（与微服务通信）

22、Sentinel 对应用保护的三种方式

23、Sentinel整合RestTemplate流控熔断

24、Sentinel整合Feign流控熔断

25、Sentinel规则持久化（默认、pull模式、push模式）

26、Spring Cloud Gateway网关（核心概念、如何工作、路由、谓词11个、过滤器31个）

27、Spring Cloud Gateway自定义谓词

28、Spring Cloud Gateway谓词不匹配404处理

29、Spring Cloud Gateway自定义路由过滤器

30、Spring Cloud Gateway全局过滤器（默认自动配置，无需单独配置）

31、Spring Cloud Gateway集成ribbon负载均衡

32、Spring Cloud Gateway集成Sentinel

33、Spring Cloud Gateway集成Sentinel规则持久化（文件、nacos）

34、Spring Cloud Gateway内部流程源码分析

35、Spring cloud gateway跨域CORS

36、Skywalking分布式链路跟踪

37、Skywalking主要功能特性和整体架构

38、SkyWalking 环境搭建部署

39、SkyWalking Agent跟踪微服务

40、IDEA中使用Skywalking Agent跟踪运行的程序

41、Skywalking告警和回调通知

42、Skywalking持久化到elasticsearch

43、Skywalking跨多个微服务跟踪

44、自定义SkyWalking链路追踪

45、Skywalking集成日志框架logback

46、Skywalking ui页面功能

47、Skywalking集群

48、什么是分布式事务

49、What is Seata？

50、Seata TC Server运行环境部署

51、AT事务模式-单体应用多数据源应用

52、AT事务模式-微服务应用

53、AT事务模式工作机制

54、Seata TC Server集群部署

55、TCC事务模式执行机制

56、基于SpringBoot单体应用的TCC事务

57、基于Spring Cloud Alibaba的TCC分布式事务

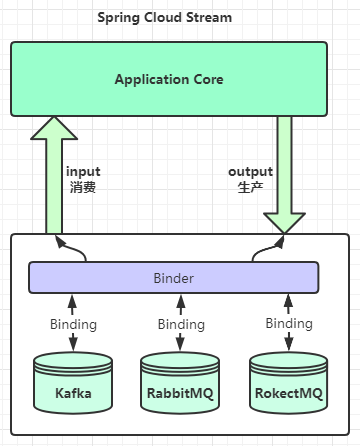
## What is Spring Cloud Stream？

在微服务的开发过程中，可能会经常用到消息中间件，通过消息中间件在服务与服务之间传递消息，不管你使用的是哪款消息中间件，比如RabbitMQ、Kafka和RocketMQ，那么消息中间件和服务之间都有一点耦合性，这个耦合性就是指如果我原来使用的RabbitMQ，现在要替换为RocketMQ，那么我们的微服务都需要修改，变动会比较大，因为这两款消息中间件有一些区别，如果我们使用Spring Cloud Stream来整合我们的消息中间件，那么这样就可以降低微服务和消息中间件的耦合性，做到轻松在不同消息中间件间切换，当然Spring Cloud Stream官方只支持rabbitmq 和 kafka，spring cloud alibaba新写了一个starter可以支持RocketMQ；

按照官方的定义，Spring Cloud Stream 是一个构建消息驱动微服务的框架；

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Spring Cloud Stream对消息中间件的进一步封装，可以做到代码层面对消息中间件的无感知，甚至于动态的切换中间件(rabbitmq切换为rocketmq或者kafka)，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；

## Spring Cloud Stream 重要概念



Spring Cloud Stream 内部有几个概念：Binder 、Binding、input、output；

1、Binder: 跟外部消息中间件集成的组件，用来创建Binding，各消息中间件都有自己的 Binder 实现；

比如 Kafka 的实现 KafkaMessageChannelBinder，RabbitMQ 的实现 RabbitMessageChannelBinder 以及 RocketMQ 的实现 RocketMQMessageChannelBinder；

2、Binding: 包括 Input Binding 和 Output Binding；

Binding 在消息中间件与应用程序提供的 Provider 和 Consumer 之间提供了一个桥梁，实现了开发者只需使用应用程序的 Provider 或 Consumer 生产或消费数据即可，屏蔽了开发者与底层消息中间件的接触；

3、input

应用程序通过input（相当于消费者consumer）与Spring Cloud Stream中Binder交互，而Binder负责与消息中间件交互，因此，我们只需关注如何与Binder交互即可，而无需关注与具体消息中间件的交互。

4、Output

output（相当于生产者producer）与Spring Cloud Stream中Binder交互；

|  |  |
| --- | --- |
| **组成** | **说明** |
| Binder | Binder是应用与消息中间件之间的封装，目前实现了Kafka和RabbitMQ的Binder，通过Binder可以很方便的连接中间件，可以动态的改变消息类型(对应于Kafka的topic，RabbitMQ的exchange)，这些都可以通过配置文件来实现； |
| @Input | 该注解标识输入通道，通过该输入通道接收消息进入应用程序 |
| @Output | 该注解标识输出通道，发布的消息将通过该通道离开应用程序 |
| @StreamListener | 监听队列，用于消费者的队列的消息接收 |
| @EnableBinding | 将信道channel和exchange、topic绑定在一起 |

下面我们看一个具体案例，来理解上面的几个概念；

## Spring Cloud Stream应用

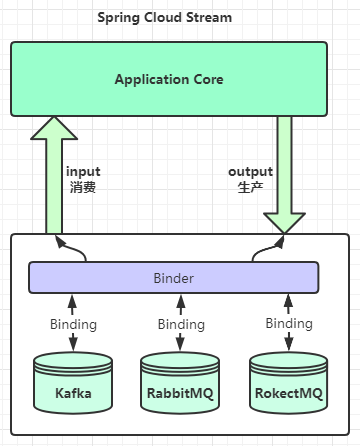
消息生产者

1、创建SpringBoot应用31-rocket-spring-cloud-stream；

2、添加依赖：

<dependency>  
 <groupId>com.alibaba.cloud</groupId>  
 <artifactId>spring-cloud-starter-stream-rocketmq</artifactId>  
</dependency>

配置文件



*########## RocketMQ 通用配置  
# 客户端接入点，必填*spring.cloud.stream.rocketmq.binder.name-server=192.168.172.128:9876  
  
*# 日志级别*logging.level.com.alibaba.cloud.stream.binder.rocketmq=INFO

*########## Consumer Config  
# input 的配置：*spring.cloud.stream.bindings.input.destination=test-topic  
spring.cloud.stream.bindings.input.content-type=text/plain  
spring.cloud.stream.bindings.input.group=test-group  
  
*########## Produce Config  
# output 的配置如下：*spring.cloud.stream.bindings.output.destination=test-topic  
spring.cloud.stream.bindings.output.content-type=text/plain

兼容性问题：

注意版本需要使用springboot2.2.5

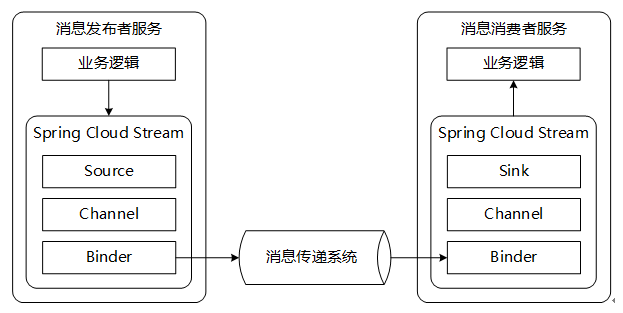
<spring-boot.version>2.2.5.RELEASE</spring-boot.version>  
<spring-cloud-alibaba.version>2.2.1.RELEASE</spring-cloud-alibaba.version>

消息发送：

@EnableBinding(Source.class)  
@Service  
public class SenderService {  
  
 @Autowired  
 private Source source;  
  
 public void send(String msg) throws Exception {  
 boolean flag = source.output().send(MessageBuilder.*withPayload*(msg).build());  
 System.*out*.println("消息发送：" + flag);  
 }  
}

消息接收：

@EnableBinding(Sink.class)  
public class ReceiveService {  
  
 @StreamListener("input")  
 public void receiveInput1(String receiveMsg) {  
 System.*out*.println("input 接收到的消息: " + receiveMsg);  
 }  
}



## Spring Cloud Stream自定义信道

在前面的案例中，我们已经实现了一个基础的 Spring Cloud Stream 消息传递处理操作，但在操作之中使用的是系统提供的 Source （output）、Sink（input），接下来我们来看一下自定义信道名称；

public interface MySource {

String *OUTPUT1* = "output1";

@Output(MySource.*OUTPUT1*)

MessageChannel output1();  
}

public interface MySink {

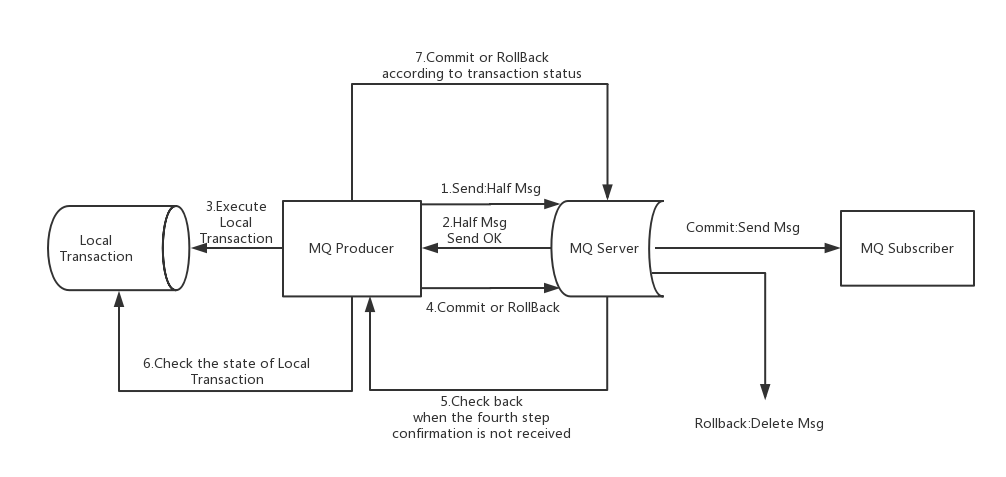
String *INPUT1* = "input1";  
  
 @Input(MySink.*INPUT1*)

SubscribableChannel input1();  
  
}

*########## 自定义  
# input 的配置：*spring.cloud.stream.bindings.input1.destination=test-topic  
spring.cloud.stream.bindings.input1.content-type=text/plain  
spring.cloud.stream.bindings.input1.group=test-group  
  
*# output 的配置：*spring.cloud.stream.bindings.output1.destination=test-topic  
spring.cloud.stream.bindings.output1.content-type=text/plain

## Spring Cloud Stream RocketMQ事务消息

Apache RocketMQ在4.3.0版中已经支持分布式事务消息，这里RocketMQ采用了2PC的思想来实现了提交事务消息，同时增加一个补偿逻辑来处理二阶段超时或者失败的消息，如下图所示：



上图说明了事务消息的大致方案，其中分为两个流程：正常事务消息的发送及提交、事务消息的补偿流程；

1.事务消息发送及提交：

(1) 发送消息（half消息）；

(2) 服务端响应消息写入结果；

(3) 根据发送结果执行本地事务（如果写入失败，此时half消息对业务不可见，本地逻辑不执行）；

(4) 根据本地事务状态执行Commit或者Rollback（Commit操作生成消息索引，消息对消费者可见）

2.补偿流程：

(1) 对没有Commit/Rollback的事务消息（pending状态的消息），从服务端发起一次“回查”；

(2) Producer收到回查消息，检查回查消息对应的本地事务的状态；

(3) 根据本地事务状态，重新Commit或者Rollback；

其中，补偿阶段用于解决消息Commit或者Rollback发生超时或者失败的情况；

事务消息一共有三种状态：提交状态、回滚状态、中间状态；

TransactionStatus.CommitTransaction: 提交事务，代表消费者可以消费此消息；

TransactionStatus.RollbackTransaction: 回滚事务，代表消息将被删除，不能被消费；

TransactionStatus.Unknown: 中间状态，代表需要检查消息队列来确定状态；

具体代码操作：

*#--------------------------事务消息--------------------------------  
#生产的配置*spring.cloud.stream.bindings.outputTX.destination=TransactionTopic  
spring.cloud.stream.bindings.outputTX.content-type=application/json  
spring.cloud.stream.rocketmq.bindings.outputTX.producer.group=myTxProducerGroup  
*#是否为事务消息，默认为false表示不是事务消息，true表示是事务消息*spring.cloud.stream.rocketmq.bindings.outputTX.producer.transactional=true  
  
*#消费的配置：*spring.cloud.stream.bindings.inputTX.destination=TransactionTopic  
spring.cloud.stream.bindings.inputTX.content-type=text/plain  
spring.cloud.stream.bindings.inputTX.group=transaction-group  
spring.cloud.stream.rocketmq.bindings.inputTX.consumer.broadcasting=false

消息发送

public <T> void sendTransactionalMsg(T msg, int num) throws Exception {  
 MessageBuilder builder = MessageBuilder.*withPayload*(msg)  
 .setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*);  
 builder.setHeader("test", String.*valueOf*(num));  
 Message message = builder.build();  
 boolean flag = mySource.output4().send(message);  
 System.*out*.println("inputTX 事务消息发送：" + flag);  
}

执行本地事务和本地事务检查：

@RocketMQTransactionListener(txProducerGroup = "myTxProducerGroup", corePoolSize = 5, maximumPoolSize = 10)  
public class TransactionListenerImpl implements RocketMQLocalTransactionListener {  
  
 @Override  
 public RocketMQLocalTransactionState executeLocalTransaction(Message msg, Object arg) {  
 Object num = msg.getHeaders().get("test");  
 if ("1".equals(num)) {  
 System.*out*.println("executer: " + new String((byte[]) msg.getPayload()) + " unknown");  
 return RocketMQLocalTransactionState.*UNKNOWN*;  
 } else if ("2".equals(num)) {  
 System.*out*.println("executer: " + new String((byte[]) msg.getPayload()) + " rollback");  
 return RocketMQLocalTransactionState.*ROLLBACK*;  
 }  
 System.*out*.println("executer: " + new String((byte[]) msg.getPayload()) + " commit");  
 return RocketMQLocalTransactionState.*COMMIT*;  
 }  
  
 @Override  
 public RocketMQLocalTransactionState checkLocalTransaction(Message msg) {  
 System.*out*.println("check: " + new String((byte[]) msg.getPayload()));  
 return RocketMQLocalTransactionState.*COMMIT*;  
 }  
}

接收：

@StreamListener("inputTX")  
public void receiveTransactionalMsg(String transactionMsg) {  
 System.*out*.println("inputTX 接收到的transaction消息: " + transactionMsg);  
}

## Spring Cloud Stream RocketMQ配置选项

### RocketMQ Binder Properties

***spring.cloud.stream.rocketmq.binder.name-server***

RocketMQ NameServer 地址(老版本使用 namesrv-addr 配置项)；

Default: 127.0.0.1:9876.

***spring.cloud.stream.rocketmq.binder.access-key***

阿里云账号 AccessKey。

Default: null.

***spring.cloud.stream.rocketmq.binder.secret-key***

阿里云账号 SecretKey。

Default: null.

***spring.cloud.stream.rocketmq.binder.enable-msg-trace***

是否为 Producer 和 Consumer 开启消息轨迹功能

Default: true.

***spring.cloud.stream.rocketmq.binder.customized-trace-topic***

消息轨迹开启后存储的 topic 名称。

Default: RMQ\_SYS\_TRACE\_TOPIC.

### RocketMQ Consumer Properties

下面的这些配置是以 spring.cloud.stream.rocketmq.bindings.<channelName>.consumer. 为前缀的 RocketMQ Consumer 相关的配置。

***enable***

是否启用 Consumer；

默认值: true.

***tags***

Consumer 基于 TAGS 订阅，多个 tag 以 || 分割；

默认值: empty.

***sql***

Consumer 基于 SQL 订阅；

默认值: empty.

***broadcasting***

Consumer 是否是广播消费模式。如果想让所有的订阅者都能接收到消息，可以使用广播模式；

默认值: false.

***orderly***

Consumer 是否同步消费消息模式；

默认值: false.

***delayLevelWhenNextConsume***

异步消费消息模式下消费失败重试策略：

-1,不重复，直接放入死信队列

0,broker 控制重试策略

>0,client 控制重试策略

默认值: 0.

***suspendCurrentQueueTimeMillis***

同步消费消息模式下消费失败后再次消费的时间间隔；

默认值: 1000.

### RocketMQ Provider Properties

下面的这些配置是以 spring.cloud.stream.rocketmq.bindings.<channelName>.producer. 为前缀的 RocketMQ Producer 相关的配置；

***enable***

是否启用 Producer；

默认值: true.

***group***

Producer group name；

默认值: empty.

***maxMessageSize***

消息发送的最大字节数；

默认值: 8249344.

***transactional***

是否发送事务消息；

默认值: false.

***sync***

是否使用同步得方式发送消息；

默认值: false.

***vipChannelEnabled***

是否在 Vip Channel 上发送消息；

默认值: true.

***sendMessageTimeout***

发送消息的超时时间(毫秒)；

默认值: 3000.

***compressMessageBodyThreshold***

消息体压缩阀值(当消息体超过 4k 的时候会被压缩)；

默认值: 4096.

***retryTimesWhenSendFailed***

在同步发送消息的模式下，消息发送失败的重试次数；

默认值: 2.

***retryTimesWhenSendAsyncFailed***

在异步发送消息的模式下，消息发送失败的重试次数；

默认值: 2.

***retryNextServer***

消息发送失败的情况下是否重试其它的 broker；

默认值: false.

## 开发的时候有两种选择

一种就是 直接SpringBoot + RocketMQ整合实现消息传送；

一种就是 使用Spring Cloud Stream对消息中间件的包装，来实现消息传送；