## 一. 同步与异步

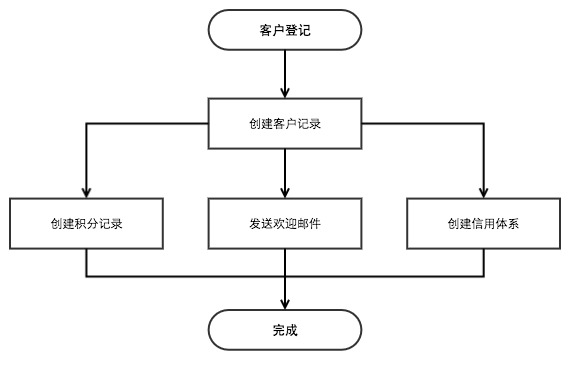
使用消息中间件不难，如何用的恰当却是门学问。

同步与异步这个基础性的选择会不可避免的引导我们使用不同的实现。

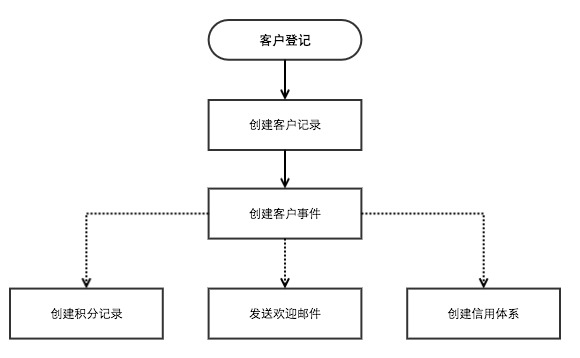
如果使用同步通信，发起一个远程服务调用后，调用方会阻塞自己并等待整个操作的完成。如果使用异步通信，调用方不需要等待操作完成就可以返回，甚至可能不需要关心这个操作是否完成与否。两种方式都有自己适用的场景，我们不扩展讨论，这里只讨论某些相比之下更适用于事件驱动的场景

这两种不同的通信模式有着各自的协作风格，既 请求/响应 和 基于事件 。

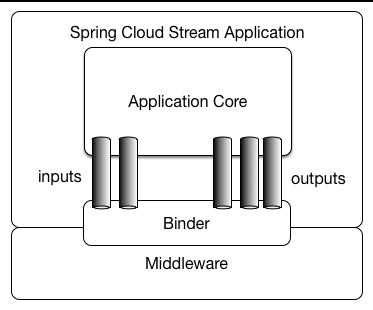
对于前者，通常是编排风格，我们会依赖某个中心大脑来指导并驱动整个流程，缺点是中心控制点承担了太多的职责，他会成为网状结构的中心枢纽及逻辑的起点，这个方法容易导致少量的“上帝”服务，而与其打交道的服务通常会沦为“贫血”的、基于CRUD的服务。



对于后者，通常是协同风格，客户端发起的不是一个请求，而是发布一个事件，然后其他协作者接收到该事件，并知道该怎么做。我们从来不会告知任何人去做任何事，基于事件的系统天生就是异步的。整个系统都很聪明，业务逻辑并非存在某个核心大脑，而是分布在不同的协作者中。基于事件的协作方式耦合性很低，这意味着你可以在不改变客户端代码的情况下，对该事件添加新的订阅者来完成新增的功能需求。



## 二. Stream应用模型



Middleware:一些消息中间件，本文用例使用kafka

Binder：粘合剂，将Middleware和Stream应用粘合起来，不同Middleware对应不同的Binder。

Channel：通道，应用程序通过一个明确的Binder与外界（中间件）通信。

ApplicationCore：Stream自己实现的消息机制封装，包括分区、分组、发布订阅的语义，与具体中间件无关，这会让开发人员很容易地以相同的代码使用不同类型的中间件。

Stream能自动发现并使用类路径中的binder,你也可以引入多个binders并选择使用哪一个，甚至可以在运行时根据不同的channels选择不同的binder实现。

## 三. 消费者分组

发布-订阅模型可以很容易地通过共享topics连接应用程序，但创建一个应用多实例的的水平扩展能力同等重要。当这样做时，应用程序的不同实例被放置在一个竞争的消费者关系中，其中只有一个实例将处理一个给定的消息，这种分组类似于Kafka consumer groups，灵感也来源于此。每个消费者通过spring.cloud.stream.bindings.<channelName>.group指定一个组名称，channelName是代码中定义好的通道名称，下文会有介绍。

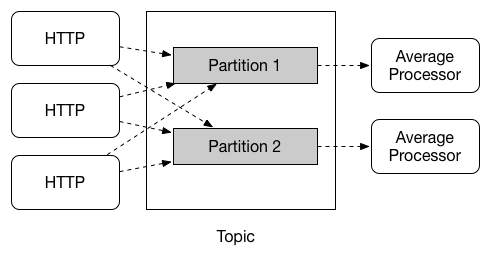
消费者组订阅是持久的，如果你的应用指定了group，那即便你这个组下的所有应用实例都挂掉了，你的应用也会在重新启动后从未读取过的位置继续读取。但如果不指定groupStream将分配给一个匿名的、独立的只有一个成员的消费组，该组与所有其他组都处于一个发布－订阅关系中，还要注意的是匿名订阅不是持久的，意味着如果你的应用挂掉，那么在修复重启之前topics中错过的数据是不能被重新读取到的。所以为了水平扩展和持久订阅，建议最好指定一个消费者组。

## 四. 分区

首先，你要放空你之前kafka分区的相关知识，从零开始去领会Stream分区，以免造成理解上的困扰。

Stream提供了一个通用的抽象，用于统一方式进行分区处理，和具体使用的中间件无关，因此分区可以用于自带分区的代理（如kafka）或者不带分区的代理（如rabbiemq），这句话要反复读几遍。

Stream支持在一个应用程序的多个实例之间数据分区，N个生产者的数据会发送给M个消费者，并保证共同的特性的数据由相同的消费者实例处理，这会提升你处理能力。



Stream使用多实例进行分区数据处理是一个复杂设置，分区功能需要在生产者与消费者两端配置，SpringCloudDataFlow可以显著的简化过程，而且当你没有用SpringCloudDataFlow时，会给你的配置带来一些不便，需要你提前规划好，而不能再应用启动后动态追加。

下面是生产者有效的和典型的配置（Output Bindings）

spring.cloud.stream.bindings.<channelName>.producer.partitionKeyExpression=payload.id

spring.cloud.stream.bindings.<channelName>.producer.partitionCount=5

分区key的值是基于partitionKeyExpression计算得出的，用于每个消息被发送至对应分区的输出channel，partitionKeyExpression是spirng EL表达式用以提取分区键

下面是消费者有效的和典型的配置（Input Bindings）

spring.cloud.stream.bindings.input.consumer.partitioned=true

spring.cloud.stream.instanceIndex=3

spring.cloud.stream.instanceCount=5

instanceCount表示应用实例的总数，instanceIndex在多个实例中必须唯一，并介于0~（instanceCount-1）之间。实例的索引可以帮助每个实例确定唯一的接收数据的分区，正确的设置这两个值十分重要，用来确保所有的数据被消费，以及应用实例接收相互排斥不重复消费。

## 五. 编程模型

**引入pom依赖**

配置binder参数

定义通道

配置通道绑定参数

通过@EnableBinding触发绑定

消费者通过@StreamListener监听

配置分区、分组信息

引入pom依赖

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-stream-kafka</artifactId>

</dependency>

也可以引入spring-cloud-stream-binder-kafka，这个少依赖了web和actuater的功能，这两个功能根据项目实际情况定制更合理，不需要的情况下没必要依赖。同理你可以引入spring-cloud-stream-binder-redis/rabbit

**配置binder参数**

SpringBoot项目启动会扫描到classpath中的kafka binder，并会用默认参数去连接本地的kafka服务和zookeeper服务，如果本地没有默认配置启动的这两个服务，一定会启动失败。所以我们要指定配置。

spring.cloud.stream.kafka.binder.brokers=10.79.96.52:9092

spring.cloud.stream.kafka.binder.zk-nodes=10.79.96.52:2182

spring.cloud.stream.kafka.binder.minPartitionCount=1

spring.cloud.stream.kafka.binder.autoCreateTopics=true

spring.cloud.stream.kafka.binder.autoAddPartitions=false

本例中配置的后三项配置值和默认值一致，当然可根据自己的需求定义。

这种配置有些讨巧，这个是kafka binder提供的Binder-Specific Configuration，这种方式让配置更看上去更清爽一些，但如果按照Stream的配置语义，应该如下配置

spring.cloud.stream.bindings.<channelName>.binder=<binderName>

spring.cloud.stream.binders.<binderName>.type=kafka

spring.cloud.stream.binders.<binderName>.environment.spring.cloud.stream.kafka.binder.brokers=10.79.96.52:9092

spring.cloud.stream.binders.<binderName>.environment.spring.cloud.stream.kafka.binder.zk-nodes=10.79.96.52:2182

先为channel对应的binder设置一个<binderName>，再根据这个<binderName>设置binder的type和environment。如果我们的应用只连接一个kafka，那我们完全可以用上面的配置方法，看起来更简洁。如果我们的应用要连接多个kafka服务，那我们必须用下面的配置方案，通过<binderName>来完成不同kafka服务的识别与隔离。

**定义通道**

Stream应用可以有任意数目的input和output通道，可通过@Input和@Output注解在接口中定义。注解默认通道名字为方法名 ，当然也可以自定义channel名字，@Input("myinputchannel")，下面的例子就完成了通道的定义，Stream在运行时会自动生成这个接口的实现类。

public interface Barista {

@Input

SubscribableChannel orders();

@Output

MessageChannel hotDrinks();

@Output

MessageChannel coldDrinks();

}

Stream为了方便开发者，内置了三个接口，在简单业务背景下，我们不用如上所述的去定义通道，直接利用预置通道会更便捷。这三个接口分别是Source，Sink，Processor。

Source用于有单个输出（outbound）通道的应用，通道名称为output

public interface Source {

String OUTPUT = "output";

@Output(Source.OUTPUT)

MessageChannel output();

}

Sink用于有单个输入（inbound）通道的应用，通道名称为input

public interface Sink {

String INPUT = "input";

@Input(Sink.INPUT)

SubscribableChannel input();

}

Processor用于单个应用同时包含输入和输出通道的情况，通道名称分别为output和input。

public interface Processor extends Source, Sink {

}

**配置通道绑定参数**

输入通道的绑定，本例中使用Sink定义输入通道，根据上面所述<channelName>=input

spring.cloud.stream.bindings.input.destination=wsh-topic-01

spring.cloud.stream.bindings.input.group=s3

spring.cloud.stream.bindings.input.consumer.concurrency=1

spring.cloud.stream.bindings.input.consumer.partitioned=false

输出通道的绑定，本例中使用Source定义输出通道，根据上面所述<channelName>=output

spring.cloud.stream.bindings.output.destination=wsh-topic-01

spring.cloud.stream.bindings.output.content-type=text/plain

spring.cloud.stream.bindings.output.producer.partitionCount=1

#spring.cloud.stream.bindings.output.producer.partitionKeyExpression=payload.id

通过@EnableBinding触发绑定

现在binder配置好了，channel也配置好了，需要做的就是将binder和channel在代码中绑定起来。

**生产者端**

@EnableBinding(Source.class)

public class SendService {

@Autowired

private Source source;

public void sendMessage(String msg) {

try {

source.output().send(MessageBuilder.withPayload(msg).build());

} catch (Exception e) {

e.printStackTrace();

}

}

}

消费者通过@StreamListener监听

**消费者端**

@EnableBinding(Sink.class)

public class MsgSink {

@StreamListener(Sink.INPUT)

public void messageSink(Object payload) {

System.out.println("Received: " + payload);

}

}

**配置分区、分组信息**

具体配置上文有提到，不重复描述，额外提一下spring.cloud.stream.kafka.binder.autoAddPartitions这个配置默认是false，通常情况下会产生无法启动的问题，强烈建议配置成true。

这里面的原理大致描述如下，比如你启动了一个生产者并配置producer.partitionCount=5，那么Stream底层是需要kafka提供5个kafka分区（注意Stream的5个分区 和 kafka的5个分区此时相等是巧合，请分开理解），如果kafka中如果没有目标topics，Stream会在启动的时候在kafka中创建5个分区，并成功启动，但是如果kafka中已经有了目标topics，并且目标topics不足5个分区，那么生产者启动失败。所以必须设置autoAddPartitions=true，生产者才能在启动的时候自动将kafka中的目标topics分区扩展成5个，方能启动成功。

如果此刻生产者启动成功，你会启动消费者，如果消费者你规划了5个实例，每个实例支持2个并发（concurrency=2），那么每个Stream底层需要5\*2=10个kafka分区（而此时kafka的目标topics只有5个分区），消费者也会启动失败，这种情况下需要将消费者的autoAddPartitions=true。

autoAddPartitions=true 有得也有失。得到的上文已经描述，这里再提一下失去的。 还拿上文举例，生产者启动了5个kafka分区，所以生产者实例只会往这5个分区中输出，这样就导致消费者扩展出来的另外5个分区收不到数据，所以要重启生产者，用以重新计算生产者与底层kafka分区的关系。 官方文档提到使用SpringCloudDataFlow可以显著的简化过程，我还没有尝试。

## 六. Content Type

@StreamListener是Stream提供的注解，Spring Integration也有一个类似功能的注解@ServiceActivator，两者都有监听通道功能，区别是@StreamListener可以根据contentType去解析数据，比如一个json格式的数据，@StreamListener可以自动解析成对象Vote

@EnableBinding(Processor.class)

public class TransformProcessor {

@Autowired

VotingService votingService;

@StreamListener(Processor.INPUT)//读取input通道的数据

@SendTo(Processor.OUTPUT)//经过方法处理后输出到output通道

public VoteResult handle(Vote vote) {

return votingService.record(vote);

}

}