# 项目地址

<https://gitee.com/java-liupx/apache-camel>

# 为什么使用Apache Camel

## Spring Integration

Spring Integration只提供了非常基础的支持，如文件，FTP，JMS，TCP，HTTP或Web服务。集成是通过编写**大量的XML代码**（没有一个真正的DSL）实现的

## Mule ESB

Mule ESB不是仅仅一个集成框架，而是一个包括一些额外功能的完整ESB，比Spring集成它更像是一个DSL。因为是一个完整的ESB，所以**集成逻辑**会比较**复杂**

## Apache Camel

Apache Camel实现了你能想到的几乎每一个技术，提供很多组件，同时你可以很容易的自定义组件。而且**Camel和Spring的集成很完善**

# Apache Camel简介

Camel HTTP、Camel FTP、Camel File、Camel JMS、Camel CXF(WebService)、Camel Jetty、Camel Timer、Camel JDBC

Apache Camel 是一个非常强大的基于规则的路由引擎，采用URI来描述各种组件，Camel还可以嵌入到任何java应用程序中，其中包含的模块有 HTTP, ActiveMQ, JMS, JBI, SCA, MINA or CXF Bus API

Aapche Camel支持DSL(domain-specific languages)特定领域语言

# EndPoint端点

from表示从这个endpoing取消息，to表示将消息发往这个endpoint，endpoint是消息地址，包含协议类型以及url

示例：from("stomp:queue:test?tcp://localhost:61613&login=username")

说明：从test的MQ队列中接收消息，消息格式为stomp，用户名为username，监听本地端口61613

示例：to("stomp:queue:test?tcp://localhost:61613&login=username")

说明：将消息发送到名为test的MQ队列中，消息格式为stomp

示例：from("http://localhost:8080/xxxxxx/queryOrgDetailById")

说明：主动向http URI描述的路径发出请求

要注意"from"部分的说明！！

from("http://localhost:8080/dbk.manager.web/queryOrgDetailById")：这个是主动向目标http url发起访问

from("jetty:http://0.0.0.0:8282/doHelloWorld")：这个是向外部提供服务，使用jetty开头

# Exchange结构

1. ExchangeID：一个Exchange贯穿着整个编排的路由规则，ExchangeID就是它的唯一编号信息，同一个路由规则的不同实例(对路由规则分别独立的两次调用)，ExchangeID不相同
2. fromEndpoint：表示exchange实例初始来源的Endpoint控制端点(类的实例)，一般来说就是开发人员设置路由时由"from"关键字所表示的Endpoint
3. properties：贯穿整个路由的自定义参数配置，以key-value方式存储
4. Pattern：是一个枚举类型，描述了消息的传播方式，类路径：org.apache.camel.ExchangePattern
5. Exception：如果在处理器Processor处理过程中，开发人员需要抛出异常并终止整个消息路由的执行过程，可以通过设置Exchange中的exception属性实现
6. Message
   1. inMessage
      1. MessageID：在系统开发阶段，提供给开发人员使用的，标示信息对象唯一性的属性，这个属性可以没有值
      2. Header：属性中的信息采用K-V的方式进行存储，并可以随着Message对象的传递将信息带到下一个参与路由的元素中。org.apache.camel.impl.DefaultMessage中对headers属性的实现是一个名叫org.apache.camel.util.CaseInsensitiveMap的类，headers属性的特点是忽略大小写
      3. Body：Message的业务信息内容存放在这里
      4. Attachement：存储各种文件内容信息，以便这些文件内容在Camel路由的各个元素间进行流转，同样使用K-V键值对形式进行文件内容的存储，但不同的是，这里的V是一个javax.activation.DataHandler类型的对象
   2. outMessage
      1. 同inMessage属性

# 操作文件

1. noop=true: 表示此操作为复制，如果去掉此参数则为移动
2. move=.done：处理完的文件移动到.done目录中进行备份，不指定的话，会默认备份至.camel
3. delete=true：删除处理完成的文件
4. file:inbox?preMove=inprogress&move=.done：文件第一步从inbox移动到inprogress，处理完成之后，再移动到.done
5. 按日期备份处理完的文件：file:inbox?move=backup/${date:now:yyyyMMdd}/${file:name}
6. 记录处理失败的文件：moveFailed=/error/${file:name.noext}-${date:now:yyyyMMddHHmmssSSS}.${file:ext}

## 说明

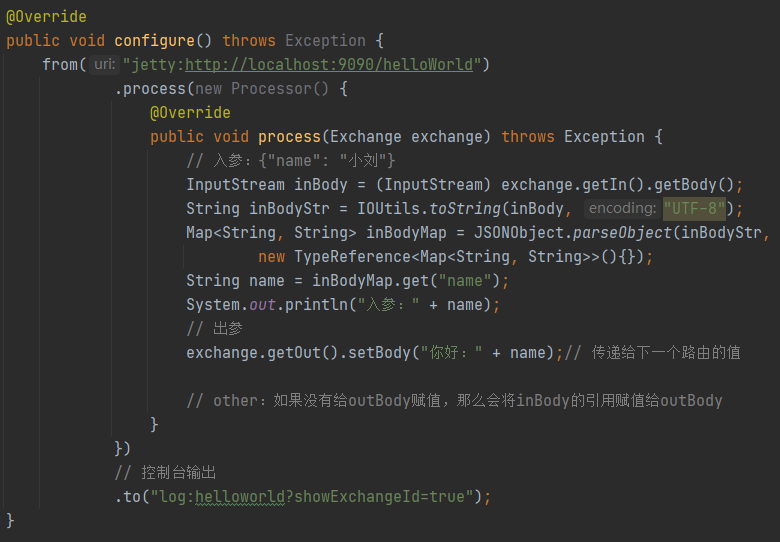
创建RouteBuilder的实例，并实现configure方法，在addRoutes方法中调用了this.setContext(context)，这样RouteBuilder对象就获得了与之对应的CamelContext，然后调用configure()，在configure方法中，可以调用例如：from()，filter()，choice()，when()，isEqualTo()，otherwise()，to()等方法

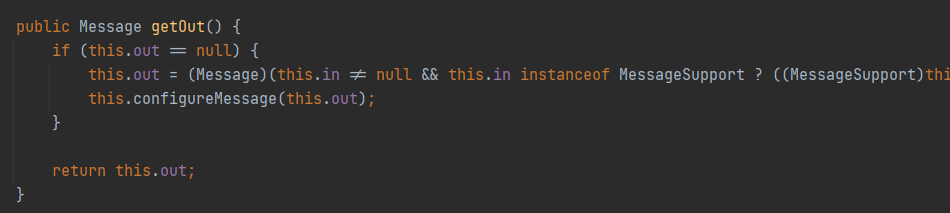
## 代码截图：CamelContext



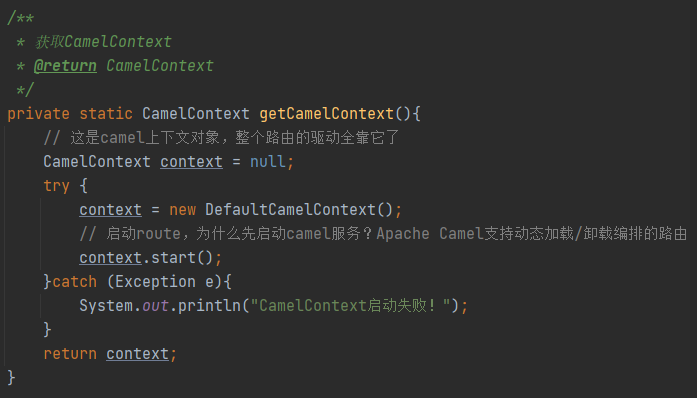
# Camel发布HTTP

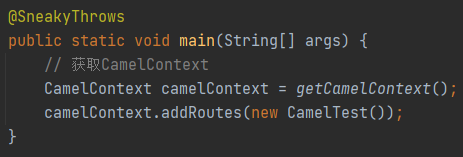
## 代码截图：继承RouteBuilder，重写configure()

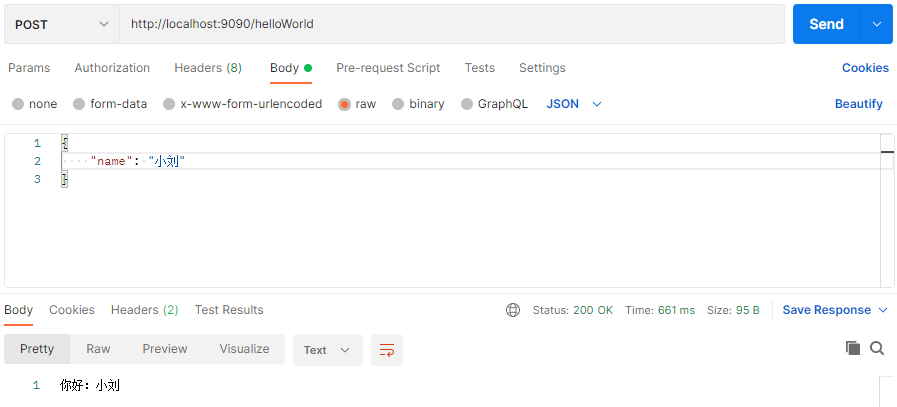




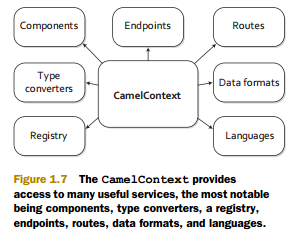
## 启动 与 执行结果







# Camel结构



1. CamelContext

CamelContext是一个容器，可以把它看作Camel的运行时系统，它把所有的东西放在一起。提供了很多服务，值得注意的有：类型转换器，注册表，端点，路由，数据格式和语言

1. routing engine(路由引擎)

Camel的路由引擎是实际上将消息转移到底层的东西。这个引擎不会暴露给开发者，并且它完成了所有繁重的工作，确保消息的路由正确

1. routes(路由)

是Camel的核心抽象。定义路由最简单的方法就是作为一个处理器链。 在消息应用程序中使用路由器有很多原因。 通过解耦客户端与服务器，以及消费者的生产者，路由可以

1. 动态决定客户端将调用什么服务器
2. 提供灵活的方式来添加额外的处理
3. 允许客户和服务器独立开发
4. 允许服务器的客户端被删除（使用mock）进行测试
5. 增强某些系统（如消息代理和ESB）的功能和特性
6. domain-specific language (DSL)(域特定语言)

为了将处理器和终端连接起来形成路由，Camel定义了一个DSL。 DSL这个术语在这里有点松散。 在Camel中，DSL意味着流畅的Java API

1. processor(处理器)

处理器是一个核心Camel概念，代表了一个能够使用，创建或修改传入交换的节点。 在路由期间，交换从一个处理器流向另一个; 因此，您可以将路由视为具有专用处理器作为节点的图形，以及将一个处理器的输出连接到另一个处理器的输入的线路

1. component(组件)

组件是Camel的主要扩展点。到目前为止，Camel生态系统中有80多个组件，其功能从数据传输到DSL，数据格式等等，甚至可以自定义组件

1. endpoint(端点)

端点是模拟系统可以通过其发送或接收消息的通道的末端的Camel抽象

1. producer(生产者)

生产者是Camel抽象，是指能够创建消息并将消息发送到端点的实体

1. consumer(消费者)

消费者是接收生产者产生的消息的服务，将其包装在交换中并发送给他们进行处理

1. event-driven consumer(事件驱动的消费者)

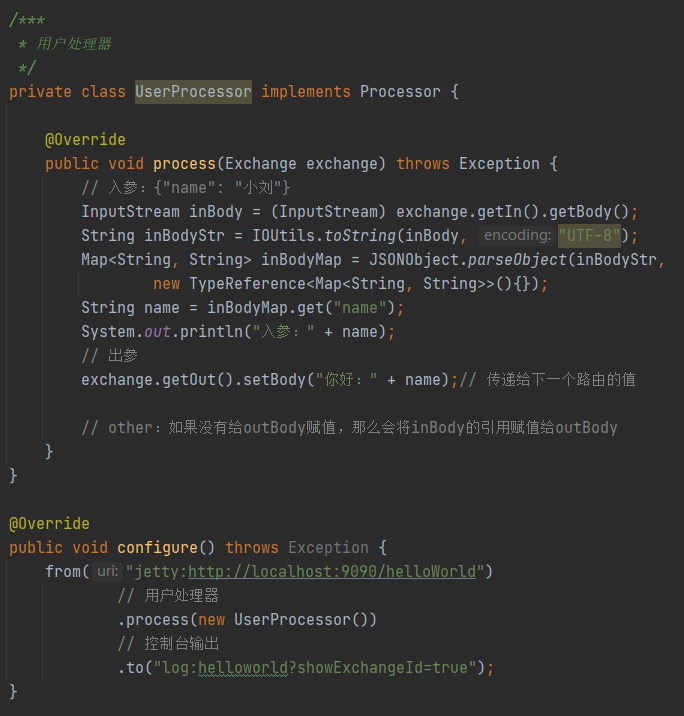
一个事件驱动的消费者等待空闲，直到消息到达，然后唤醒并消费该消息，被称为异步接收器

1. polling consumer(轮询消费者)

轮询消费者主动检查新消息

# Camel数据转换

## 使用process进行转换



## 使用bean进行转换

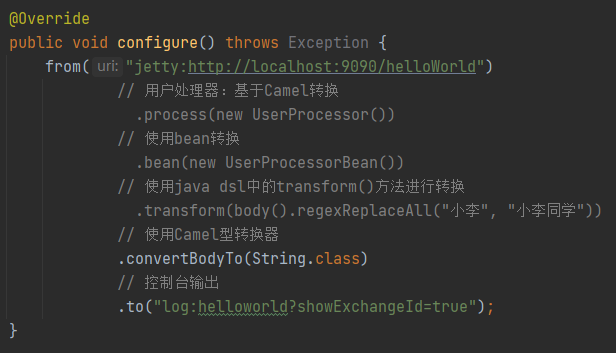


## 使用java dsl中的transform()方法进行转换

Transform() 是Java DSL中的一种方法，可用于Camel路由来转换消息。通过允许使用表达式，transform() 允许很大的灵活性，并且在DSL中直接使用表达式有时可以节省时间



## 使用Camel型转换器



# CommonContext

public class CommonContext {  
  
 // 这是camel上下文对象，整个路由的驱动全靠它了  
 private static CamelContext *context* = null;  
  
 */\*\*  
 \* 获取CamelContext  
 \** ***@return*** *CamelContext  
 \*/* public static void getCamelContext(){  
 try {  
 *context* = new DefaultCamelContext();  
 // 启动route，为什么先启动camel服务？Apache Camel支持动态加载/卸载编排的路由  
 *context*.start();  
 }catch (Exception e){  
 System.*out*.println("CamelContext启动失败！");  
 }  
 }  
  
 */\*\*  
 \* 添加路由  
 \** ***@param*** *routesBuilder 路由  
 \*/* public static void addRoutes(RoutesBuilder routesBuilder){  
 try {  
 *context*.addRoutes(routesBuilder);  
 } catch (Exception e) {  
 e.printStackTrace();  
 System.*out*.println("路由添加失败：" + routesBuilder);  
 }  
 }  
  
 */\*\*  
 \* 模拟业务执行缓慢  
 \** ***@param*** *time sleep时长  
 \*/* public static void sleep(Long time){  
 try {  
 Thread.*sleep*(time);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* 打印body信息  
 \** ***@param*** *exchange Exchange  
 \** ***@param*** *prefix 打印前缀  
 \** ***@return*** *String  
 \*/* public static String printBodyContent(Exchange exchange, String prefix){  
 InputStream inBodyStream = (InputStream) exchange.getIn().getBody();  
 String inBodyStr = "";  
 try {  
 inBodyStr = IOUtils.*toString*(inBodyStream, "UTF-8");  
 System.*out*.println(prefix + inBodyStr);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 return inBodyStr;  
 }  
}

# Camel特殊的Endpoint

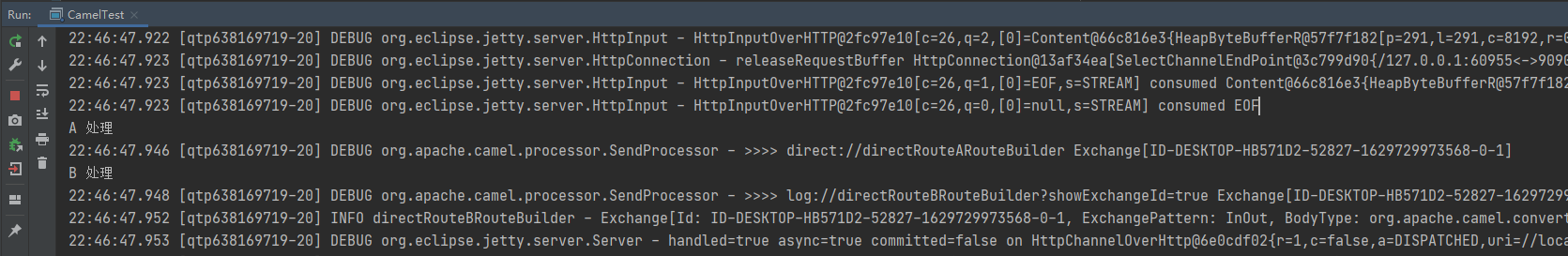
Endpoint Direct用于在两个编排好的路由之间实现Exchange消息的连接，被连接的两个路由使用的Exchange对象是同一个。

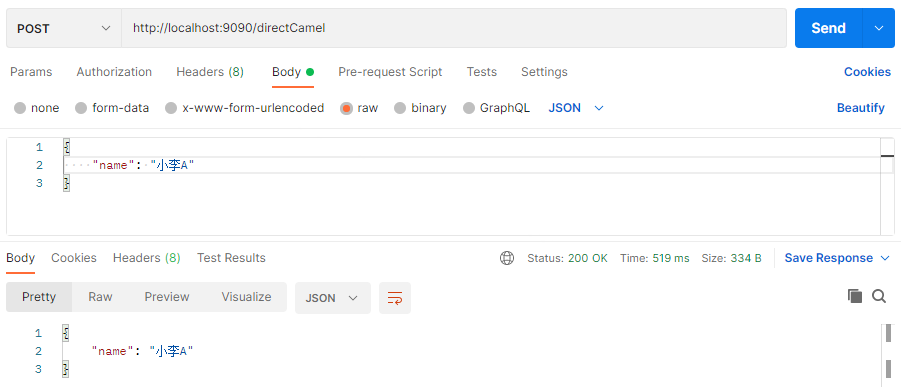
注意：两个被连接的路由一定要是可用的，并且存在于同一个Camel服务中

## 路由代码

*/\*\*\*  
 \* Camel特殊的Endpoint  
 \** ***@author*** *liupx  
 \** ***@date*** *2021年8月22日  
 \*/*public class DirectRouteTest extends CommonContext {  
  
 @SneakyThrows  
 public static void main(String[] args) {  
 *getCamelContext*();  
 *addRoutes*(new DirectRouteARouteBuilder());  
 *addRoutes*(new DirectRouteBRouteBuilder());  
 }  
  
 // A 路由  
 static class DirectRouteARouteBuilder extends RouteBuilder{  
 @Override  
 public void configure() throws Exception {  
 from("jetty:http://localhost:9090/directCamel")  
 .process(new Processor() {  
 @Override  
 public void process(Exchange exchange) throws Exception {  
 System.*out*.println("A 处理");  
 }  
 })  
 .to("direct:directRouteARouteBuilder");  
 }  
 }  
  
 // B 路由  
 static class DirectRouteBRouteBuilder extends RouteBuilder{  
 @Override  
 public void configure() throws Exception {  
 from("direct:directRouteARouteBuilder")  
 .process(new Processor() {  
 @Override  
 public void process(Exchange exchange) throws Exception {  
 System.*out*.println("B 处理");  
 }  
 })  
 .to("log:directRouteBRouteBuilder?showExchangeId=true");  
 }  
 }  
}

## 启动 与 执行结果

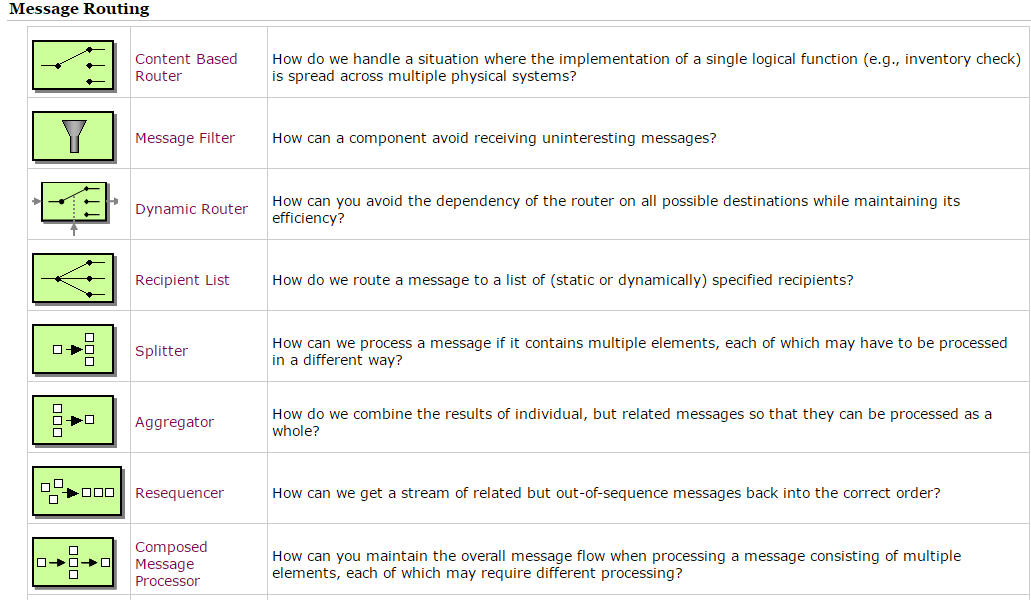




这里可以看到，代码中：路由A，路由B没有做任何处理，也没有向下传递值，结果却返回了输入的值

# Camel路由条件

## Routing路由条件



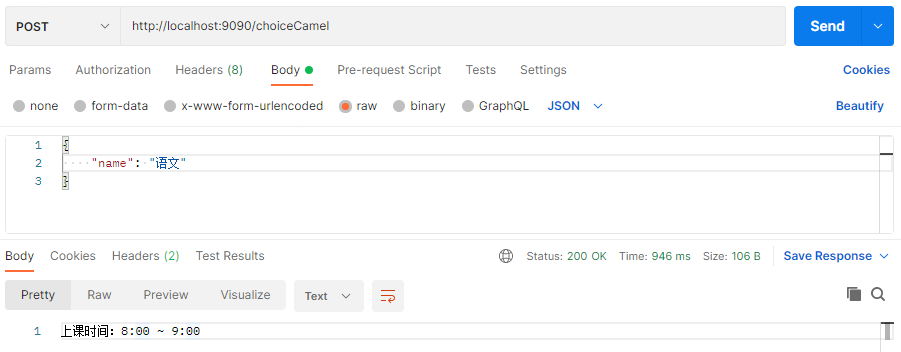
## Content Based Router 基于内容的路由

它并不是一种单一的路由方式，而是多种基于条件和判断表达式的路由方式

### 代码示例

*/\*\*\*  
 \* Content Based Router 基于内容的路由  
 \** ***@author*** *liupx  
 \*/*public class ChoiceCamel extends CommonContext {  
  
 public static void main(String[] args) {  
 *getCamelContext*();  
 *addRoutes*(new ChoiceCamelRouteBuilder());  
 }  
  
 static class ChoiceCamelRouteBuilder extends RouteBuilder {  
 @Override  
 public void configure() throws Exception {  
 from("jetty:http://localhost:9090/choiceCamel")  
 .process(exchange -> {  
 String body = exchange.getIn().getBody(String.class);  
 Map<String, Object> orderMap = JSONObject.*parseObject*(body, new TypeReference<Map<String, Object>>(){});  
 exchange.setProperty("name", orderMap.get("name").toString());  
 })  
 .choice()  
 .when(exchange -> exchange.getProperty("name").equals("语文"))  
 .process(exchange -> exchange.getOut().setBody("上课时间：8:00 ~ 9:00"))  
  
 .when(exchange -> exchange.getProperty("name").equals("数学"))  
 .process(exchange -> exchange.getOut().setBody("上课时间：9:30 ~ 10:30"))  
  
 .otherwise()  
 .process(exchange -> exchange.getOut().setBody("输入有误！"))  
  
 .endChoice();  
 }  
 }  
}

### 执行结果



## StaticRecipient List(静态路由)

使用multicast方式时，Camel将会把上一元素输出的Exchange复制多份发送给这个列表中的所有接收者，并且按照顺序逐一执行这些接收者(可设置为并行处理)

区别：加了multicast后,消息会复制多份,发送到后面所有的to节点,multicast可以指定线程池来并发,发送消息 没加的话,消息会依次流转到后面的节点

注意：Exchange在路由与路由之间，或者，处理器与处理器之间是**复制**关系，所以将一条消息复制多份并且让其执行互不干扰受到影响。所以**内存区域是不同的**，ExchangeId也是不同的。

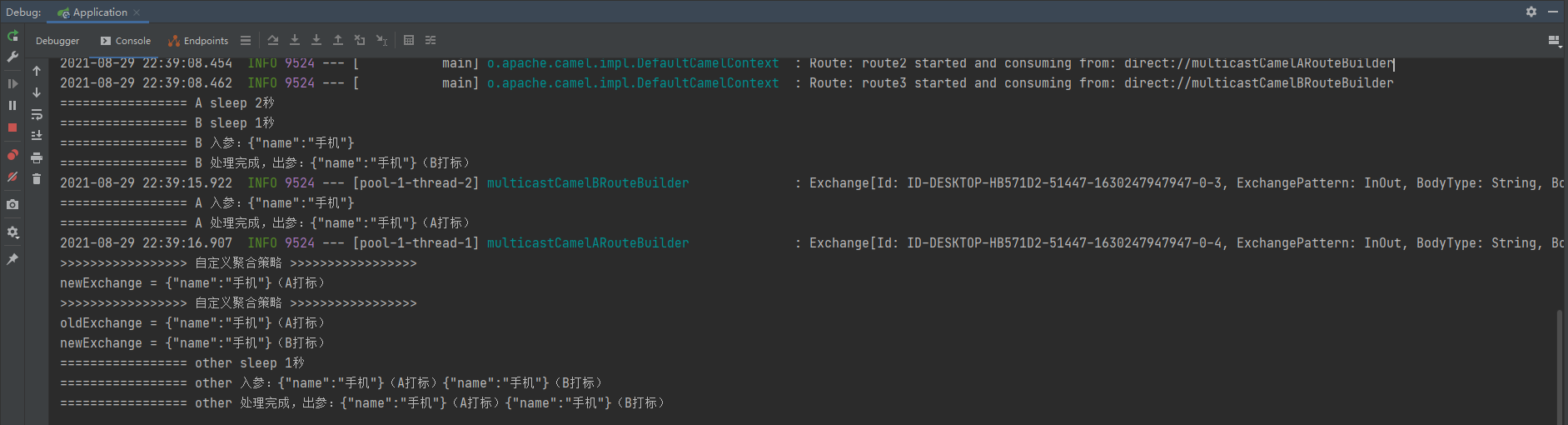
### 代码示例

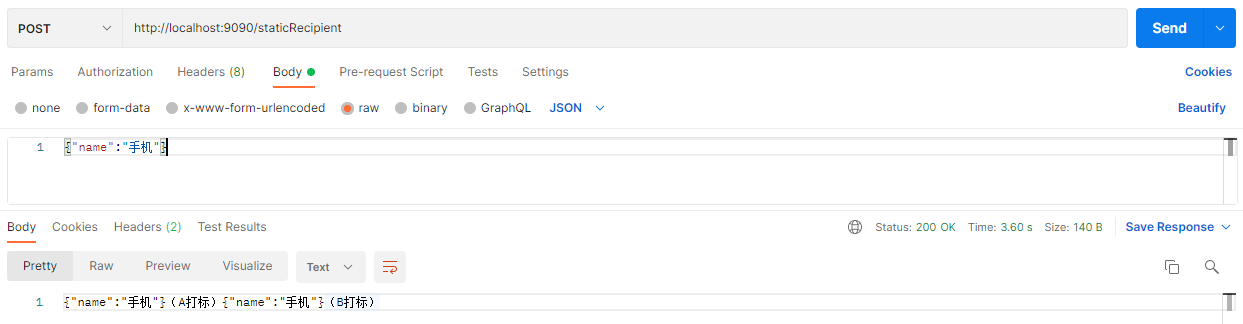
*/\*\*\*  
 \* 使用multicast 处理 StaticRecipient List(静态路由)  
 \** ***@author*** *liupx  
 \*/*@Component  
public class StaticRecipient extends CommonContext {  
  
 public void init(){  
 *addRoutes*(new MulticastCamelRouteBuilder());  
 *addRoutes*(new MulticastCamelARouteBuilder());  
 *addRoutes*(new MulticastCamelBRouteBuilder());  
 }  
  
 static class MulticastCamelRouteBuilder extends RouteBuilder {  
 @Override  
 public void configure() throws Exception {  
 // multicast需要并发执行，这里创建一个线程池  
 ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(3, 10, 5, TimeUnit.*SECONDS*, new ArrayBlockingQueue<>(100));  
  
 MulticastDefinition multicast = from("jetty:http://localhost:9090/staticRecipient").multicast();  
 // 并发执行  
 multicast.setParallelProcessing(true);  
 // 为并发执行设置一个独立的线程池  
 multicast.setExecutorService(threadPoolExecutor);  
 // 自定义聚合策略  
 multicast.setAggregationStrategy(new CustomAggregationStrategy());  
  
 // 注意，multicast中各路由路径的Excahnge都是基于上一路由元素的excahnge复制而来  
 // 无论前者Excahnge中的Pattern如何设置，其处理结果都不会反映在最初的Excahnge对象中  
 multicast.to("direct:multicastCamelARouteBuilder", "direct:multicastCamelBRouteBuilder")  
 // 一定要使用end，否则OtherProcessor会被做为multicast中的一个分支路由  
 .end()  
 // 所以您在OtherProcessor中看到的Excahnge中的Body、Header等属性内容  
 // 不会有“复制的Exchange”设置的任何值的痕迹  
 .process(new OtherProcessor());  
 }  
 }  
  
 // 路由A  
 static class MulticastCamelARouteBuilder extends RouteBuilder {  
 @Override  
 public void configure() throws Exception {  
 from("direct:multicastCamelARouteBuilder")  
 .process(new MulticastCamelAProcessor())  
 .to("log:multicastCamelARouteBuilder?showExchangeId=true");  
 }  
 }  
 // 路由A 处理器  
 static class MulticastCamelAProcessor extends CommonContext implements Processor {  
 @Override  
 public void process(Exchange exchange) throws Exception {  
 System.*out*.println("================= A sleep 2秒");  
 *sleep*(2000L);  
 String inBodyStr = *printBodyContent*(exchange, "================= A 入参：");  
  
 String outBody = inBodyStr + "（A打标）";  
 exchange.getOut().setBody(outBody);  
 System.*out*.println("================= A 处理完成，出参：" + outBody);  
 }  
 }  
  
 // 路由B  
 static class MulticastCamelBRouteBuilder extends RouteBuilder {  
 @Override  
 public void configure() throws Exception {  
 from("direct:multicastCamelBRouteBuilder")  
 .process(new MulticastCamelBProcessor())  
 .to("log:multicastCamelBRouteBuilder?showExchangeId=true");  
 }  
 }  
 // 路由B 处理器  
 static class MulticastCamelBProcessor extends CommonContext implements Processor {  
 @Override  
 public void process(Exchange exchange) throws Exception {  
 System.*out*.println("================= B sleep 1秒");  
 *sleep*(1000L);  
 String inBodyStr = *printBodyContent*(exchange, "================= B 入参：");  
  
 String outBody = inBodyStr + "（B打标）";  
 exchange.getOut().setBody(outBody);  
 System.*out*.println("================= B 处理完成，出参：" + outBody);  
 }  
 }  
  
 // 其他 处理器  
 static class OtherProcessor implements Processor {  
 @Override  
 public void process(Exchange exchange) throws Exception {  
 System.*out*.println("================= other sleep 1秒");  
 *sleep*(1000L);  
  
 String inBody = (String) exchange.getIn().getBody();  
 System.*out*.println("================= other 入参：" + inBody);  
  
 InputStream outBody = (InputStream) exchange.getOut().getBody();  
 String outStr = IOUtils.*toString*(outBody, "UTF-8");  
 if(StringUtils.*isEmpty*(outStr)){  
 exchange.getOut().setBody(inBody);  
 }  
 System.*out*.println("================= other 处理完成，出参：" + exchange.getOut().getBody());  
 }  
 }  
}

### 自定义聚合策略

*/\*\*\*  
 \* 自定义聚合策略  
 \** ***@author*** *liupx  
 \*/*public class CustomAggregationStrategy implements AggregationStrategy {  
  
 @Override  
 public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {  
 System.*out*.println(">>>>>>>>>>>>>>>>> 自定义聚合策略 >>>>>>>>>>>>>>>>>");  
  
 String currentBody = "";  
  
 // oldExchange  
 if(oldExchange != null){  
 Object body = oldExchange.getIn().getBody();  
 currentBody += body;  
 System.*out*.println("oldExchange = " + body);  
 }  
  
 // newExchange  
 if(newExchange != null){  
 Object body = newExchange.getIn().getBody();  
 currentBody += body;  
 System.*out*.println("newExchange = " + body);  
 }  
  
 newExchange.getOut().setBody(currentBody);  
  
 return newExchange;  
 }  
}

### 运行结果



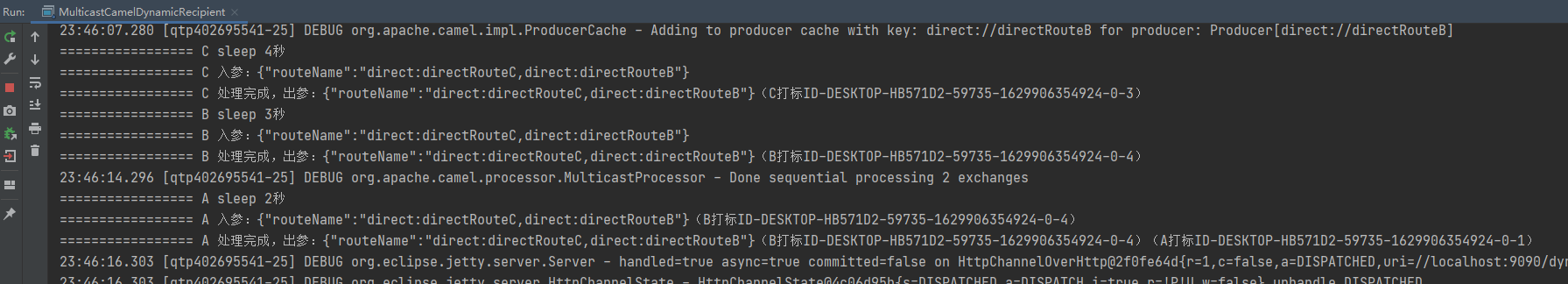


## Dynamic Recipient List(动态路由)

### 代码示例

*/\*\*\*  
 \* 使用multicast 处理Dynamic Recipient List(动态路由)  
 \** ***@author*** *liupx  
 \** ***@date*** *2021年8月25日  
 \*/*public class MulticastCamelDynamicRecipient extends CommonContext {  
  
 */\*\*  
 \* 访问时传参：{"routeName":"direct:directRouteB,direct:directRouteC"}  
 \** ***@param*** *args  
 \*/* public static void main(String[] args) {  
 *getCamelContext*();  
 *addRoutes*(new DirectRouteA());  
 *addRoutes*(new DirectRouteB());  
 *addRoutes*(new DirectRouteC());  
 }  
  
 // 路由A  
 static class DirectRouteA extends RouteBuilder {  
 @Override  
 public void configure() throws Exception {  
 from("jetty:http://localhost:9090/dynamicRecipient")  
 .setExchangePattern(ExchangePattern.*InOnly*)  
 // 该方法可以像multicast方法那样进行并发执行或者运行线程池的设置  
 .recipientList()  
 .jsonpath("$.routeName").delimiter(",")  
 .end()  
 .process(new Processor() {  
 @Override  
 public void process(Exchange exchange) throws Exception {  
 System.*out*.println("================= A sleep 2秒");  
 *sleep*(2000L);  
 String inBodyStr = exchange.getIn().getBody(String.class);  
 System.*out*.println("================= A 入参：" + inBodyStr);  
  
 String outBody = inBodyStr + "（A打标"+exchange.getExchangeId()+"）";  
 exchange.getOut().setBody(outBody);  
 System.*out*.println("================= A 处理完成，出参：" + outBody);  
 }  
 });  
 }  
 }  
  
 // 路由B  
 static class DirectRouteB extends RouteBuilder {  
 @Override  
 public void configure() throws Exception {  
 from("direct:directRouteB")  
 .process(new Processor() {  
 @Override  
 public void process(Exchange exchange) throws Exception {  
 System.*out*.println("================= B sleep 3秒");  
 *sleep*(3000L);  
 String inBodyStr = *printBodyContent*(exchange, "================= B 入参：");  
  
 String outBody = inBodyStr + "（B打标"+exchange.getExchangeId()+"）";  
 exchange.getOut().setBody(outBody);  
 System.*out*.println("================= B 处理完成，出参：" + outBody);  
 }  
 });  
 }  
 }  
  
 // 路由C  
 static class DirectRouteC extends RouteBuilder {  
 @Override  
 public void configure() throws Exception {  
 from("direct:directRouteC")  
 .process(new Processor() {  
 @Override  
 public void process(Exchange exchange) throws Exception {  
 System.*out*.println("================= C sleep 4秒");  
 *sleep*(4000L);  
 String inBodyStr = *printBodyContent*(exchange, "================= C 入参：");  
  
 String outBody = inBodyStr + "（C打标"+exchange.getExchangeId()+"）";  
 exchange.getOut().setBody(outBody);  
 System.*out*.println("================= C 处理完成，出参：" + outBody);  
 }  
 });  
 }  
 }  
}

### 运行结果





## 循环动态路由 Dynamic Router

未完待续。。。