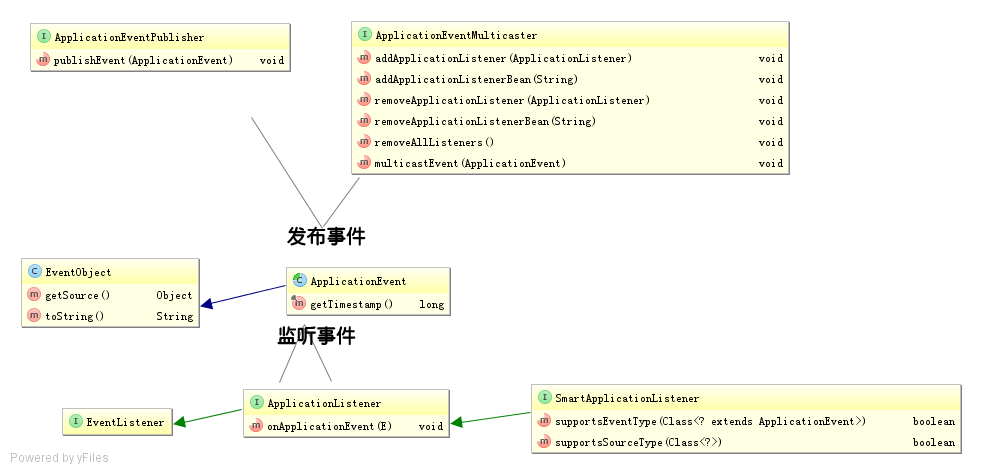
# spring 消息相关知识

# Spring的事件驱动模型（知识补充）

Spring的事件驱动模型中包含了三个基本概念：事件、事件监听者和事件发布者



**事件**：

Spring中定义了事件的抽象类ApplicationEvent，它继承自JDK的EventObject类。从图中我们可以看到，事件包含了两个成员变量：timestamp，该字段用于存储事件发生的时间戳，以及父类中source，该字段表示源事件对象。

当我们需要自定义事件的时候，只需要继承ApplicatinEvent，比如RemoteApplicationEvent、RefershRemoteApplicationEvent等，可以在自定义的Event中增加一些事件的属性来给事件监听者处理。

**事件监听者**：

Spring中定义了事件监听者的接口ApplicationListener，它继承自JDK的EventListener接口，同时ApplicationListener接口限定了ApplicationEvent子类作为该接口中的onApplicationEvent(E event)；函数的参数。

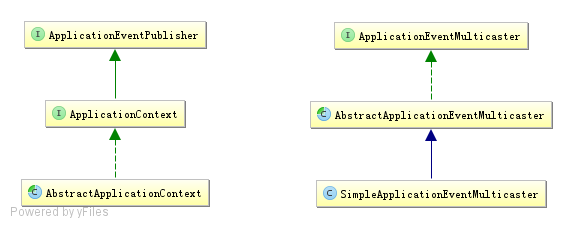
所以，每一个ApplicationListener都是针对某个ApplicationEvent子类的监听和处理者

**事件发布者**：

Spring中定义了ApplicationEventPublisher和ApplicationEventMulticater两个接口用来发布事件。其中ApplicationEventPublisher接口定义了发布事件。

ApplicationEventPublisher接口定义了发布事件的函数publishEvent(ApplicationEvent event)和publishEvent(Object event)；

而ApplicationEventMulticaster接口中定义了对ApplicationListener的维护操作（比如新增、移除等）以及将ApplicationEvent多路广播给可用ApplicationListener的操作。



1.ApplicationContext接口继承了ApplicationEventPublisher，并在AbstractApplicationContext实现了具体代码，实际执行是委托给ApplicationEventMulticaster（可以认为是多播）：

**s**

public void publishEvent(ApplicationEvent event) {

//省略部分代码

}

getApplicationEventMulticaster().multicastEvent(event);

if (this.parent != null) {

this.parent.publishEvent(event);

}

}

我们常用的ApplicationContext都继承自AbstractApplicationContext，如ClassPathXmlApplicationContext、XmlWebApplicationContext等。所以自动拥有这个功能。

2.ApplicationContext自动到本地容器里找一个名字为”“的ApplicationEventMulticaster实现，如果没有自己new一个SimpleApplicationEventMulticaster。其中SimpleApplicationEventMulticaster发布事件的代码如下：

**d**

public void multicastEvent(final ApplicationEvent event) {

for (final ApplicationListener listener : getApplicationListeners(event)) {

Executor executor = getTaskExecutor();

if (executor != null) {

executor.execute(new Runnable() {

public void run() {

listener.onApplicationEvent(event);

}

});

}

else {

listener.onApplicationEvent(event);

}

}

}

可以看到如果给它一个executor（java.util.concurrent.Executor），它就可以异步支持发布事件了。佛则就是通过发送。

所以我们发送事件只需要通过ApplicationContext.publishEvent即可，没必要再创建自己的实现了。除非有必要。

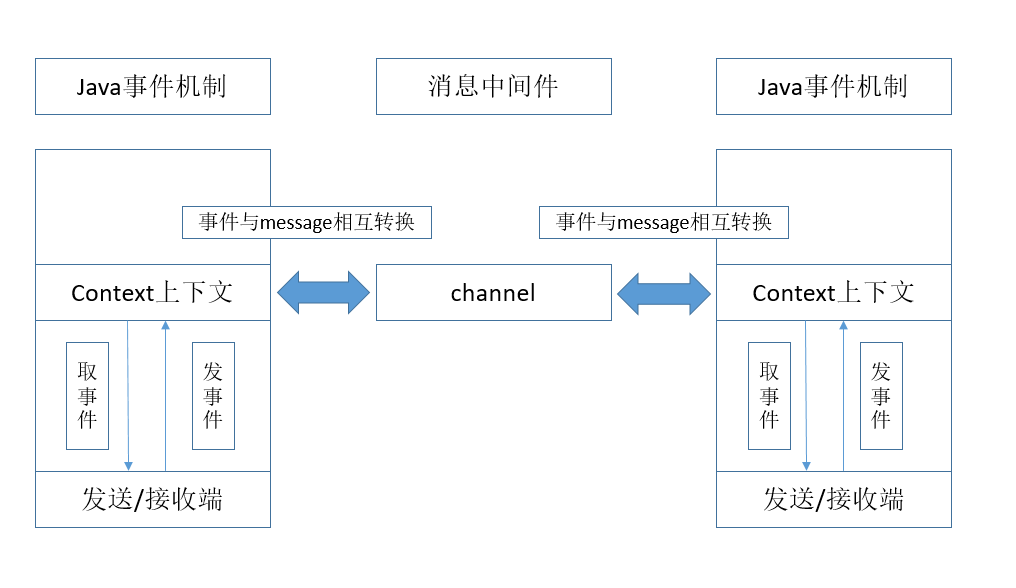
参考文章： <https://jinnianshilongnian.iteye.com/blog/1902886>（详解Spring事件驱动模型）

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

# Spring Cloud Bus（消息总线）

Spring Cloud Bus用轻量级的消息代理将分布式系统的节点连接起来。这可以用来广播状态的改变（比如配置的改变）或者其他关联的指令。一个关键的想法是，总线就像是一个分布式Actuator，

用于Spring Boot应用程序的扩展，但它也可以用作应用程序之间的通信通道。Spring Cloud提供了AMQP 传输的代理和Kafka启动Starters。



完整流程：

发送端（endpoint）构造事件event，将其publish到context上下文中（spring cloud bus有一个父上下文，bootstrap），然后将事件发送到channel中（json串message），接收端从channel中获取到message，将message转为事件event（转换过程这一块没有深究），

然后将event事件publish到context上下文中，最后接收端（Listener）收到event，调用服务进行处理。整个流程中，只有发送/接收端从context上下文中取事件和发送事件是需要我们在代码中明确写出来的，其它部分都由框架封装完成。

Spring Cloud Bus是在Spring Cloud Stream的基础上进行的封装，对于指定主题的消息的发布与订阅是通过Spring Cloud Stream的具体binder实现。

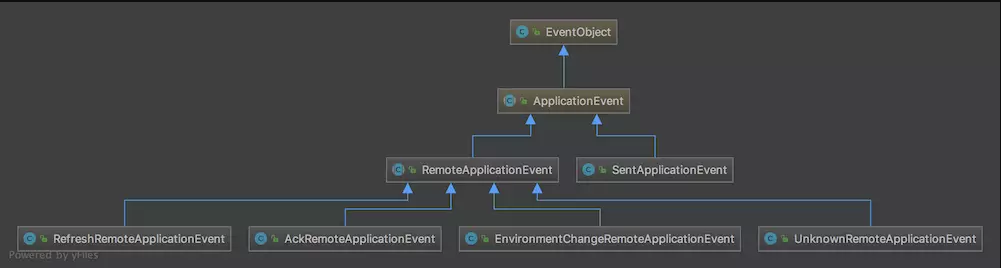
因此引入的依赖可以是spring-cloud-starter-bus-amqp和spring-cloud-starter-bus-kafka其中的一种，分别对应于binder的两种实现。

（我们可以查看spring-cloud-starter-bus-amqp和spring-cloud-starter-bus-kafka的依赖，可以看到它们分别依赖了spring-cloud-starter-stream-rabbit和spring-cloud-starter-stream-kafka，真正实现与这些消息代理进行交互操作的是Spring Cloud Stream。）

* 对指定主题消息的订阅与发布。
* 事件监听，包括刷新事件、环境变更事件、远端应用的ack事件以及本地服务端发送事件等。

源码分析：

Spring的事件驱动模型的事件定义均继承自ApplicationEvent，Spring Cloud Bus中有多个事件类，这些事件类都继承了一个重要的抽象类RemoteApplicationEvent，我们看一下事件类的类图：



涉及的事件类有：

代表了对特定事件确认的事件AckRemoteApplicationEvent、

环境变更的事件EnvironmentChangeRemoteApplicationEvent、

刷新事件RefreshRemoteApplicationEvent、

发送事件 SentApplicationEvent、

以及未知事件UnknownRemoteApplicationEvent

#### 抽象基类：RemoteApplicationEvent

通过上面的类图，我们知道RemoteApplicationEvent是其他事件类的基类，定义了事件对象的公共属性。

**基类**

@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "type") //序列化时使用子类的名称作为type

@JsonIgnoreProperties("source") //序列化时，忽略 source

public abstract class RemoteApplicationEvent extends ApplicationEvent {

private static final Object TRANSIENT\_SOURCE = new Object();

private final String originService;

private final String destinationService;

private final String id;

protected RemoteApplicationEvent(Object source, String originService,

String destinationService) {

super(source);

this.originService = originService;

if (destinationService == null) {

destinationService = "\*\*";

}

if (!"\*\*".equals(destinationService)) {

if (StringUtils.countOccurrencesOf(destinationService, ":") <= 1

&& !StringUtils.endsWithIgnoreCase(destinationService, ":\*\*")) {

//destination的所有实例

destinationService = destinationService + ":\*\*";

}

}

this.destinationService = destinationService;

this.id = UUID.randomUUID().toString();

}

...

}

在RemoteApplicationEvent中定义了主要的三个通用属性事件的来源originService、事件的目的服务destinationService和随机生成的全局id。

通过其构造方法可知，destinationService可以使用通配符的形式{serviceId}:{appContextId}，两个变量都省略的话，则通知到所有服务的所有实例。只省略appContextId时，则对应的destinationService为相应serviceId的所有实例。

另外，注解@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "type")对应于序列化时，使用子类的名称作为type；而@JsonIgnoreProperties("source")表示序列化时，忽略source属性，source定义在JDK中的EventObject。

#### EnvironmentChangeRemoteApplicationEvent

用于动态更新服务实例的环境属性，我们在基础应用中更新cloud.version属性时，关联到该事件。

**更新服务**

public class EnvironmentChangeRemoteApplicationEvent extends RemoteApplicationEvent {

private final Map<String, String> values;

public EnvironmentChangeRemoteApplicationEvent(Object source, String originService,

String destinationService, Map<String, String> values) {

super(source, originService, destinationService);

this.values = values;

}

...

}

可以看到，EnvironmentChangeRemoteApplicationEvent事件类的实现很简单。定义了Map类型的成员变量，key对应于环境变量名，而value对应更新后的值。

#### RefreshRemoteApplicationEvent

刷新远端应用配置的事件，用于接收远端刷新的请求。

**刷新**

public class RefreshRemoteApplicationEvent extends RemoteApplicationEvent {

public RefreshRemoteApplicationEvent(Object source, String originService,

String destinationService) {

super(source, originService, destinationService);

}

}

继承自抽象事件类RemoteApplicationEvent，没有特别的成员属性。

#### AckRemoteApplicationEvent

确认远端应用事件，该事件表示一个特定的RemoteApplicationEvent事件被确认。

**回调**

public class AckRemoteApplicationEvent extends RemoteApplicationEvent {

private final String ackId;

private final String ackDestinationService;

private Class<? extends RemoteApplicationEvent> event;

public AckRemoteApplicationEvent(Object source, String originService,

String destinationService, String ackDestinationService, String ackId,

Class<? extends RemoteApplicationEvent> type) {

super(source, originService, destinationService);

this.ackDestinationService = ackDestinationService;

this.ackId = ackId;

this.event = type;

}

...

public void setEventName(String eventName) {

try {

event = (Class<? extends RemoteApplicationEvent>) Class.forName(eventName);

} catch (ClassNotFoundException e) {

event = UnknownRemoteApplicationEvent.class;

}

}

}

该事件类在RemoteApplicationEvent基础上，定义了成员属性ackId、ackDestinationService和event。  
ackId和ackDestinationService，分别表示确认的时间的id和对应的目标服务。event对应事件类型，确认事件能够确认的必然是RemoteApplicationEvent的子类，因此event属性设值时需要进行检查，如果转换出现异常，则定义为未知的事件类型。

这些事件可以被任何需要统计总线事件响应的应用程序来监听。 它们的行为与普通的远程应用程序事件相似，即如果目标服务与本地服务ID匹配，则应用程序会在其上下文中触发该事件。

#### SentApplicationEvent

发送应用事件，表示系统中的某个地方发送了一个远端事件。

**发送**

@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "type")

@JsonIgnoreProperties("source")

public class SentApplicationEvent extends ApplicationEvent {

private static final Object TRANSIENT\_SOURCE = new Object();

private final String originService;

private final String destinationService;

private final String id;

private Class<? extends RemoteApplicationEvent> type;

protected SentApplicationEvent() {

// for serialization libs like jackson

this(TRANSIENT\_SOURCE, null, null, null, RemoteApplicationEvent.class);

}

public SentApplicationEvent(Object source, String originService,

String destinationService, String id,

Class<? extends RemoteApplicationEvent> type) {

super(source);

this.originService = originService;

this.type = type;

if (destinationService == null) {

destinationService = "\*";

}

if (!destinationService.contains(":")) {

// All instances of the destination unless specifically requested

destinationService = destinationService + ":\*\*";

}

this.destinationService = destinationService;

this.id = id;

}

...

}

可以看到该事件类继承自ApplicationEvent，它本身并不是一个RemoteApplicationEvent事件，所以不会通过总线发送，而是在本地生成（多为响应远端事件）。想要审计远端事件的应用可以监听该事件，

并且所有的AckRemoteApplicationEvent事件中的id来源于相应的SentApplicationEvent中定义的id。在其定义的成员属性中，相比于远端应用事件多了一个事件类型type，该类型限定于RemoteApplicationEvent的子类。

#### UnknownRemoteApplicationEvent

未知的远端应用事件，也是RemoteApplicationEvent事件类的子类。该事件类与之前的SentApplicationEvent、AckRemoteApplicationEvent有关，当序列化时遇到事件的类型转换异常，则自动构造成一个未知的远端应用事件。

spring cloud bus ： <https://www.jianshu.com/p/c5a3f1b988c9> （该文章介绍的比较详细，并且有简单的demo）

参考文章： <https://www.jianshu.com/p/d62612947e9e>

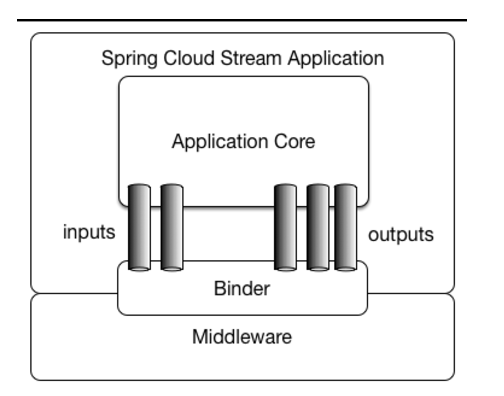
 消息总线 spring cloud bus ：<https://www.jianshu.com/p/a5012222d01e?utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation>

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

# Spring Cloud Stream（消息驱动）

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。他通过使用Spring Integration来连接消息代理中间件以实现消息事件驱动。

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，引用了发布-订阅、消费组、分区的三个核心概念。目前仅支持RabbitMQ、Kafka。



应用程序通过 inputs 或者 outputs 来与 Spring Cloud Stream 中binder 交互，通过我们配置来 binding ，而 Spring Cloud Stream 的 binder 负责与中间件交互。所以，我们只需要搞清楚如何与 Spring Cloud Stream 交互就可以方便使用消息驱动的方式。

## Binder（绑定器）

Binder 是 Spring Cloud Stream 的一个抽象概念，是应用与消息中间件之间的粘合剂。目前 Spring Cloud Stream 实现了 Kafka 和 Rabbit MQ 的binder。

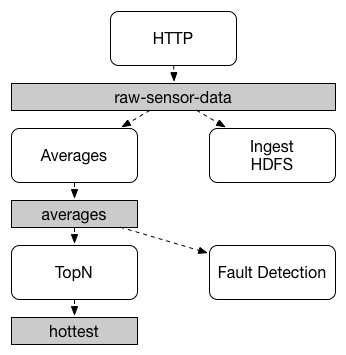
通过 binder ，可以很方便的连接中间件，可以动态的改变消息的 destinations（对应于 Kafka 的topic，Rabbit MQ 的 exchanges），这些都可以通过外部配置项来做到。

甚至可以任意的改变中间件的类型而不需要修改一行代码。

## Publish-Subscribe（发布-订阅）

消息的发布（Publish）和订阅（Subscribe）是事件驱动的经典模式。Spring Cloud Stream 的数据交互也是基于这个思想。生产者把消息通过某个 topic 广播出去（Spring Cloud Stream 中的 destinations）。

其他的微服务，通过订阅特定 topic 来获取广播出来的消息来触发业务的进行。这种模式，极大的降低了生产者与消费者之间的耦合。即使有新的应用的引入，也不需要破坏当前系统的整体结构。



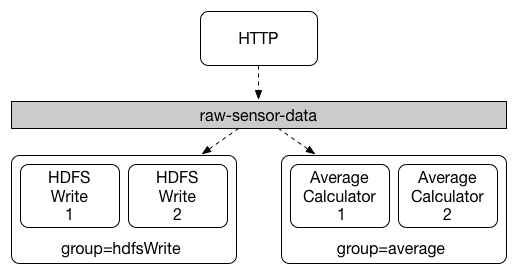
未处理的传感数据发布到raw-sensor-data的Topic进行广播，Averages 和IngestHDFS同时订阅了此消息，收到消息后触发自身的处理逻辑。

Topic可能对应不同的概念，在RabbitMQ表示的是Exchange，Kafka中对应Topic

## Consumer Groups（消费组）

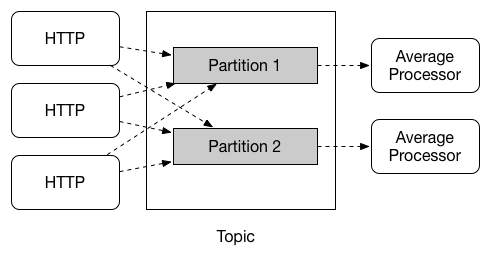
“Group”，如果使用过 Kafka 的童鞋并不会陌生。Spring Cloud Stream 的这个分组概念的意思基本和 Kafka 一致。

微服务中动态的缩放同一个应用的数量以此来达到更高的处理能力是非常必须的。对于这种情况，同一个事件防止被重复消费，只要把这些应用放置于同一个 “group” 中，就能够保证消息只会被其中一个应用消费一次。



## 消息分区

Spring Cloud Stream对给定应用的多个实例之间分隔数据予以支持。在分隔方案中，物理交流媒介（如：代理主题）被视为分隔成了多个片（partitions）。一个或者多个生产者应用实例给多个消费者应用实例发送消息并确保相同特征的数据被同一消费者实例处理。   
Spring Cloud Stream对分割的进程实例实现进行了抽象。使得Spring Cloud Stream 为不具备分区功能的消息中间件（RabbitMQ）也增加了分区功能扩展。



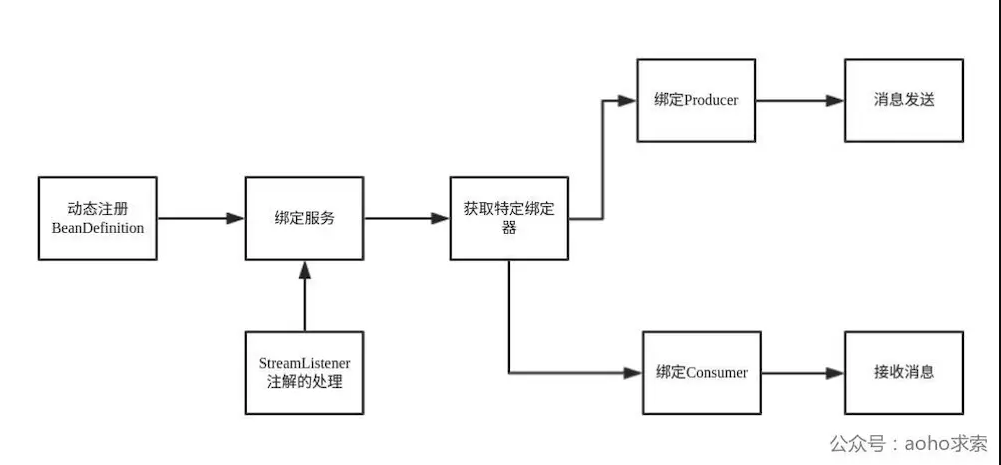
## Durability

消息事件的持久化是必不可少的。Spring Cloud Stream 可以动态的选择一个消息队列是持久化，还是 present。

## Bindings

bindings 是我们通过配置把应用和spring cloud stream 的 binder 绑定在一起，之后我们只需要修改 binding 的配置来达到动态修改topic、exchange、type等一系列信息而不需要修改一行代码。

## 原理简介（参考<https://www.jianshu.com/p/a4aca841cbcd>）



* Stream首先会动态注册相关BeanDefinition，并且处理@StreamListener注解；
* 然后在Bean实例初始化之后，会调用BindingService进行服务绑定；
* BindingService在绑定服务时会首先获取特定的Binder绑定器，然后绑定Producer和Consumer；
* 最后Stream的相关实例就会进行发送和接受消息的处理。

Spring Cloud Stream的demo：

Spring Cloud Stream的使用非常简单，在应用的main启动类上加上 @EnableBinding 注解就可以连接消息中间件，创建一个流式微服务。通过在方法上添加 @StreamListener注解就可以异步处理所接收到的消息。

文章参考： <https://blog.csdn.net/qq_32734365/article/details/81413218>

<https://blog.csdn.net/jack281706/article/details/73743148/>

<https://www.jianshu.com/p/fb7d11c7f798>

<https://www.jianshu.com/p/1621becd3c4a>

<https://www.cnblogs.com/hellxz/p/9396282.html>

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

# spring-intergration

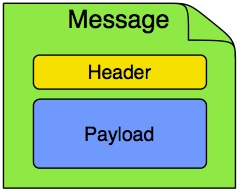
spring Ingegration 提供了基于Spring的EIP(Enterprise Integration Patterns,企业集成模式)的实现。Spring Integration 主要解决的问题是不同系统之间交互的问题，提供一个简单的模型来实现复杂的企业集成解决方案，

为基于spring的应用添加异步的、消息驱动的行为，通过异步消息驱动来达到系统交互时系统之间的松耦合。

Spring Integration 主要有**Message、Channel、Message EndPoint**组成。

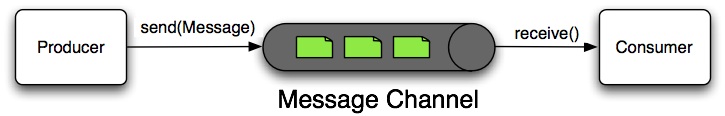
### Message：

Message是它的基础构件和核心，所有的流程都围绕着Message运转。Message分为两部分，header和payload。header是头部信息，用来存储传输的一些特性属性参数。payload是用来装载数据的，他可以携带的任何Object对象。



### MessageChannel

消息管道，生产者生产一个消息到channel，消费者从channel消费一个消息，所以channel可以对消息组件解耦，并且提供一个方便的拦截功能和监控功能。



消息管道分两种，一种是point-to-point点对点的，一种是publish-subscribe发布订阅形式的。

如果是点对点的channel，至少会有一个消费者consumer能收到发送的message，另一种订阅发布的channel，spring integration试图去用广播的形式发布message给那些订阅者subscriber。

MessageChannel 有两大子接口，分别是PollableChannel (可轮询)和SubscribableChannel(可订阅)。我们所有的消息通道类都是现实这两个接口。

主要实现如下：

* PublishSubscribeChannel：将消息广播给所有的订阅者。
* QueueChannel：可以缓存消息，在缓存消息没有达到上限时，消息发送者将消息发送到该通道后立即返回。如果缓存消息数量达到设定的容量，则消息发送者发送消息后会被阻塞，直到消息队列中有空间为止或者超时。对于消息接收者正好相反，尝试获取消息时，如果队列中有消息会立即返回，如果队列中没有消息则会一直阻塞直到超时（需要设定超时时间，不设定的话一直阻塞）。
* PriorityChannel：一个排序队列。默认是根据消息头中的“priority”属性值进行排序，也可以通过实现Comparator<Message<?>>接口的逻辑来进行排序。
* RendezvousChannel：类似于QueueChannel，只是容量为0。也就是发送者发送消息时，接受者必须将其接收才会返回，否则一直被阻塞。接受者也是一样。
* DirectChannel：用于点对点场景，但实现的是PublishSubscribeChannel接口，因此该通道会将消息直接分发给接收者，和PublishSubscribeChannel的区别就是，接收者只有一个。除此之外，DirectChannel还有一个最重要的特点就是发送和接收双方处于一个线程当中，如下：发送者发送消息->接收者接收消息并触发处理操作->返回。Spring Integration中的缺省的通道就是DirectChannel。例如<int:channel id="xxxx"/>。
* ExecutorChannel：用于点对点场景，和DirectChannel的配置相同，但主要的区别是ExecutorChannel将消息分发的操作委派给一个TaskExecutor的实例来进行，因此发送消息时不会进行阻塞。
* Scoped Channel：暂不理解。

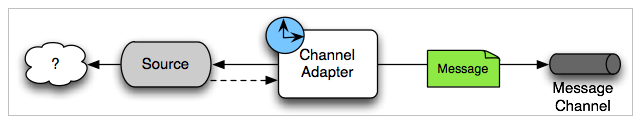
### Message Endpoint

消息的端点，或者我称他为消息节点，在channel你不能操作消息，只能在endpoint操作。

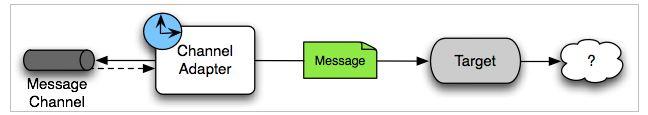
常用的消息节点：

* Transformer 转换器，作用是可以将消息转为你想要的类型。可以将xml形式转换成string类型。
* Filter  过滤器  类似于路由(Router)，不同的是过滤器不决定消息路由到哪里，而是决定消息是否可以传递给消息通道。
* Gateway 消息网关   类似于Adapter，但是提供了双向的请求/返回集成方式，也分为入站(inbound)和出站(outbound)。 Spring Integration 对响应的Adapter都提供了Gateway。
* Channel Adapter 通道适配器     连接一个消息通道和其他实体之间的对象。channel adapter也分inbound内绑定和outbound外绑定。通道适配器是单向的，入站通道适配器只支持接收消息，出站通道适配器只支持输出消息。

Inbound通道适配：通常的作用是将一个外部系统的资源进行转换，通过消息通道输送到系统中，用于进行后续的处理。

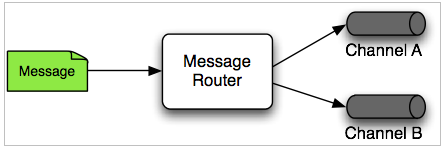


Outbound通道适配：将系统中的资源通过消息通道发送给Outbound通道适配，然后该“Outbound通道适配”将其转换为外部的资源。



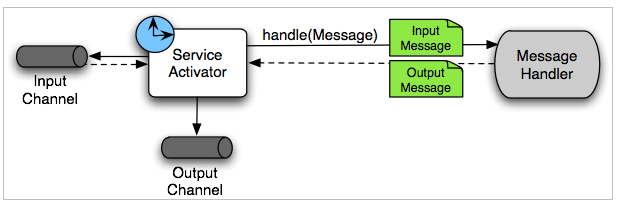
通常情况下，通道适配器将在消息与从其他系统（文件，HTTP请求，JMS消息等）接收或发送的任何对象或资源之间进行映射。

* Router  路由器    通常是一个输入通道，多个输出通道。可根据消息体内容(Payload Type Router)、消息头的值(Header Value Router) 以及定义好的接收表(Recipient List Router) 作为条件，来决定消息传递到的通道。



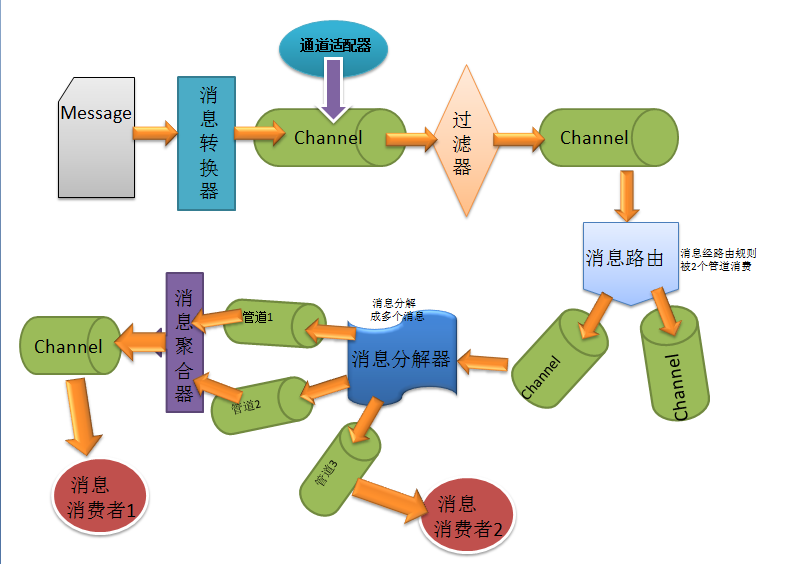
* Service Activator 服务激活器    用来连接应用的接口和message framework消息框架的组件 ，可调用Spring的Bean来处理消息，并将处理后的结果输出到指定的消息通道。

一个输入的频道input message channel必须被设定，一个service activator的方法被执行并且返回了一个值，那么可以提供一个输出频道output message channel（如果消息提供自己的返回地址，那么这是可选的）。



Channel Bridge 连接桥   使用连接桥(Bridge)可以简单的将两个消息通道连接起来

* Splitter  分解器  对应一个输入通道，多个输出通道。Splitter把消息从输入通道上分割发送到它的输出通道上。比如用于把一个复合型的payload负载分割成很多子负载payloads，发送到多个输出通道上。拆分器处理的返回值是一个集合或者数组。
* Aggregator  聚合器   和splitter对应。用于把多种message组合成一个单一的message。Aggregator在聚合时，会将消息头中消息ID相同的消息进行聚合。它接收一个java.util.List作为参数，将多个消息合并为一个消息。
* Enricher   当我们从外部获得消息后，需要增加额外的消息到已有的消息中，这时就需要使用消息增强器(Enricher)。消息增强器主要有消息体 增强器(Payload Enricher)和消息头增强器(Header Enricher)两种。



**通道拦截器**

Spring Integration给消息通道提供了通道拦截器(ChannelInterceptor)，用来拦截发送和接收消息的操作.

ChannelInterceptor接口定义如下，我们只需要实现这个接口即可：

**拦截器**

public interface ChannelInterceptor {

Message<?> preSend(Message<?> message, MessageChannel channel);

void postSend(Message<?> message, MessageChannel channel, boolean sent);

void afterSendCompletion(Message<?> message, MessageChannel channel, boolean sent, Exception ex);

boolean preReceive(MessageChannel channel);

Message<?> postReceive(Message<?> message, MessageChannel channel);

void afterReceiveCompletion(Message<?> message, MessageChannel channel, Exception ex);

}

通过如下代码给所有的channel增加拦截器

channel.addInterceptor(someInterceptor);

spring-integration初探 ： https://blog.csdn.net/qq\_27808011/article/details/80108622

Spring Integration 4.0: 一个完全无需 XML 的示例  ： <https://blog.csdn.net/FighterYang/article/details/80523732>

spring-intergration  demo：（小卖部订餐例子）<http://www.blogjava.net/xmatthew/archive/2008/11/29/243455.html>

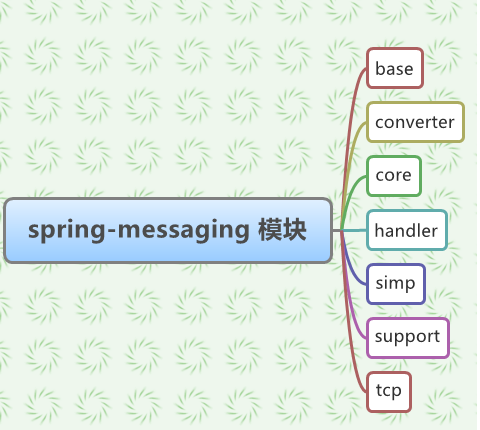
参考文章：  java开源项目：Spring Integration    （<http://www.importnew.com/16538.html>）

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

# spring-messaging

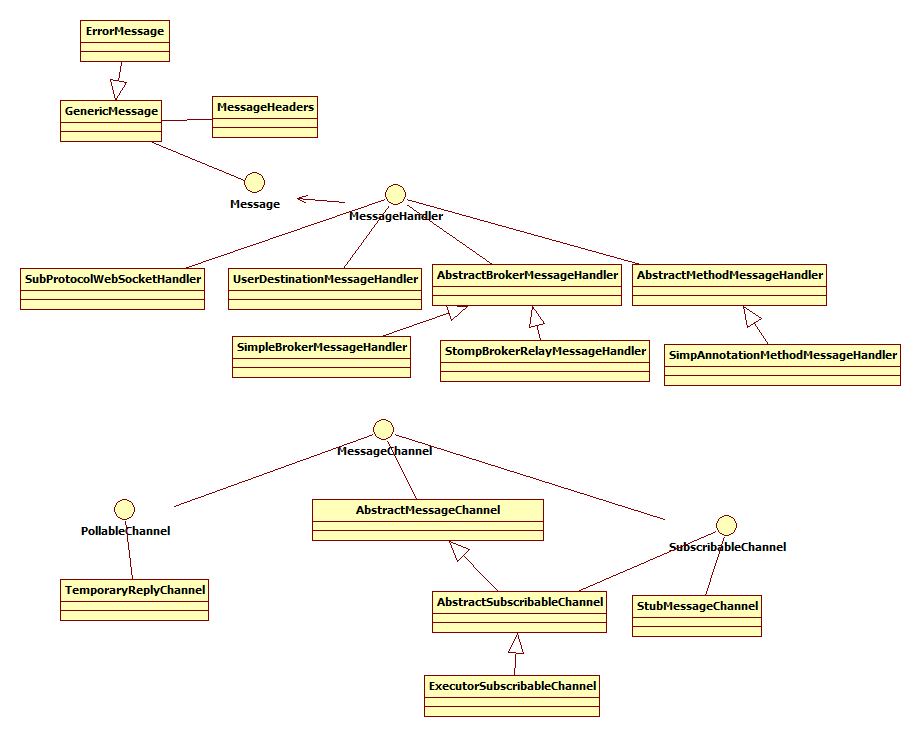
spring-messaging模块为集成messaging api和消息协议提供支持。

其代码结构为：

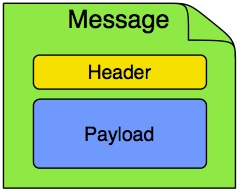


其中base定义了消息Message(MessageHeader和body)、消息处理MessageHandler、发送消息MessageChannel。

**1. base模块**

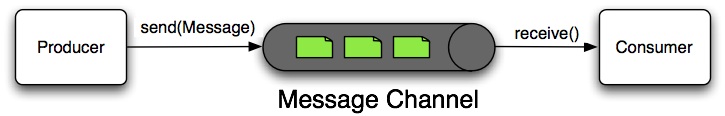


其中：message由两部分组成



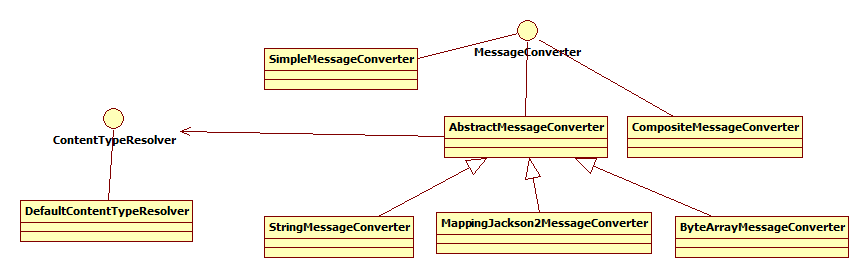
MessageHandler是一个处理消息的约定，spring messaging提供了丰富的消息处理方式。

MessageChannel表现为pipes-and-filters架构的管道。



**2. 转换器converter模块**

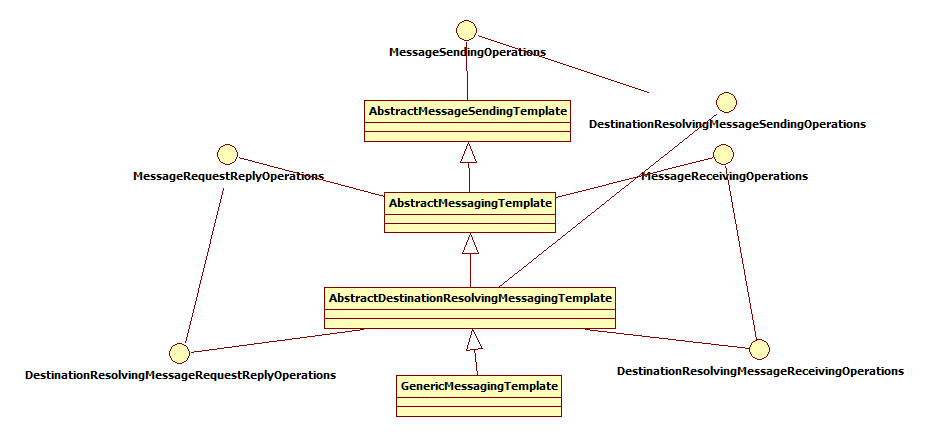
对消息转换提供支持。其结构如下：



从上图可以看出，有消息到string、json、byte数组之间的相互转换。

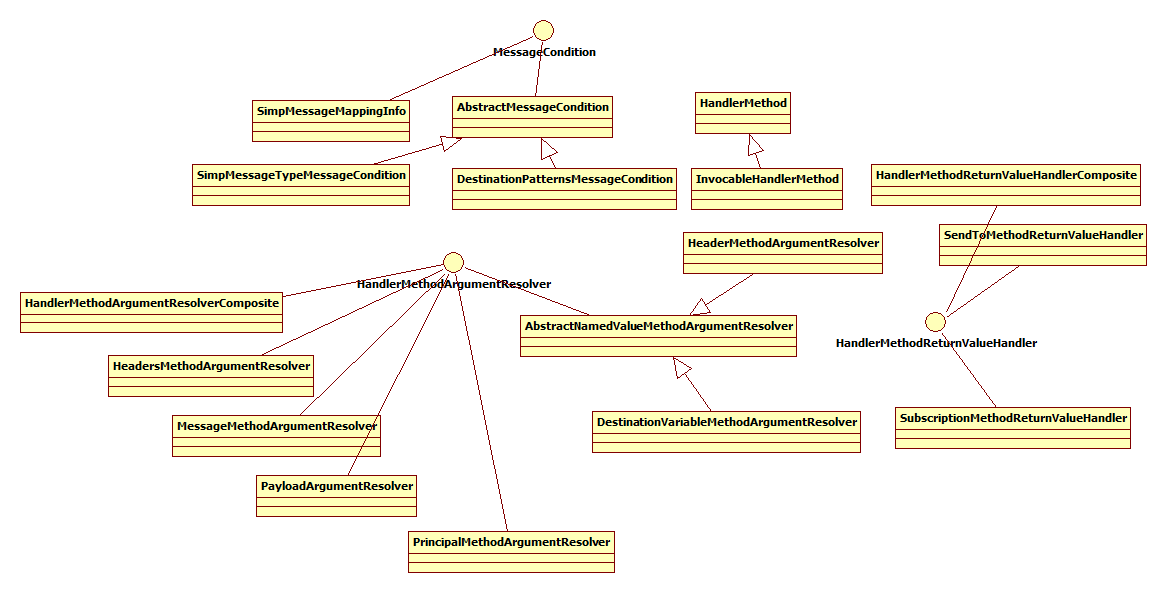
**3. 核心core模块**

核心模块提供消息的模板方法，其结构如下：



**4. 处理handler模块**

大致架构如下：



其中：

HandlerMethod封装了一个bean的方法相关信息(getMethod()和getBean（）方法)，提供了访问方法参数的便利工具。HandlerMethod可以在bean factory中使用createWithResolvedBean获取bean实例时获取该实例。

MessageCondition是一个将conditions映射到message的约定。

HandlerMethodArgumentResolver 是一个解析方法参数到Context中指定Message的参数值的策略接口。

HandlerMethodReturnValueHandler是一个处理从触发一个Message的method Handling返回值的策略接口。

另外，也提供了部分注解：

@interface Header:Annotation which indicates that a method parameter should be bound to a message header.

@interface Headers:Annotation which indicates that a method parameter should be bound to the headers of a message. The annotated parameter must be 　　assignable to {@link java.util.Map} with String keys and Object values.

@interface MessageExceptionHandler: Annotation for handling exceptions thrown from message-handling methods within a  specific handler class.

@interface MessageMapping:Annotation for mapping a {@link Message} onto message-handling methods by matching to the message destination.

@interface Payload:Annotation that binds a method parameter to the payload of a message. The payload may be passed through a {@link 　　MessageConverter} to convert it from serialized form with specific MIME type to an Object matching the target method parameter.

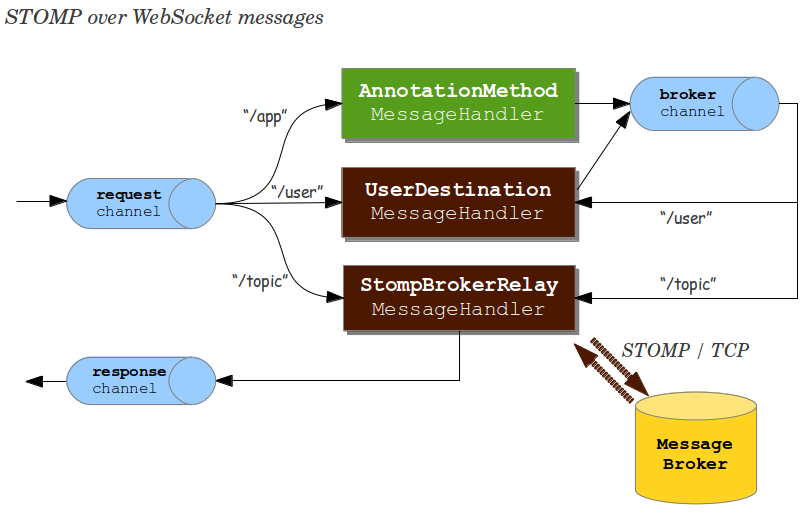
@interface SendTo:Annotation that indicates a method's return value should be converted to a {@link Message} and sent to the specified destination.

**5.Simp模块**

包含诸如STOMP协议的简单消息协议的通用支持。

STOMP，Streaming Text Orientated Message Protocol，是流文本定向消息协议，是一种为MOM(Message Oriented Middleware，面向消息的中间件)设计的简单文本协议。它提供了一个可互操作的连接格式，允许STOMP客户端与任意STOMP消息代理(Broker)进行交互，

类似于OpenWire(一种二进制协议)。由于其设计简单，很容易开发客户端，因此在多种语言和多种平台上得到广泛应用。其中最流行的STOMP消息代理是Apache ActiveMQ。  
[详细协议内容中文版本](http://www.cnblogs.com/davidwang456/p/4449428.html)参见。另stomp架构如下：



**6. support模块**

提供了Message的实现，及创建消息的MessageBuilder和获取消息头的MessageHeaderAccessor，还有各种不同的MessageChannel实现和channel interceptor支持。

**7. tcp模块**

   一方面提供了通过TcpOperations建立tcp connection、通过TcpConnectionHandler处理消息和通过TcpConnectionf发送消息的抽象及实现；另一方面包含了对基于Reactor的tcp 消息支持。

参考文章： <https://www.cnblogs.com/davidwang456/p/4446796.html>