采用spring integration开发一个健壮的消息传送框架

先是总览了Spring integration，它是一种便捷的事件驱动消息框架；你可以用它协同消息、通道、适配器、网关。接着介绍了如何利用Spring Integration实现ActvieMQ和JMS，随后简短地介绍了针对轻量级负载和重量级负载的多个应用工作流程。

Spring Integration作为一种企业级集成框架，遵从现代经典书籍[《企业集成模式》](http://vdisk.weibo.com/s/zaPQt9WypmpAY)，为开发者提供了一种便捷的实现模式。Spring Integration构建在Spring[控制反转](http://www.javaworld.com/article/2071914/excellent-explanation-of-dependency-%20injection--inversion-of-control-.html)[设计模式](http://www.amazon.cn/gp/product/B001130JN8/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=importnew-23&linkCode=as2&camp=536&creative=3200&creativeASIN=B001130JN8)之上，抽象了消息源和目标，利用消息传送和消息操作来集成应用环境下的各种组件。采用Spring Integration构建的应用可以在组件之间发送消息，可以穿过一个消息总线，将该消息发送到应用环境中的另一个服务器，甚至是同一台虚拟机的其它类中。

我会在[Spring开源Java项目](http://www.javaworld.com/blog/open-source-java-projects/)的 第二部分向大家介绍Spring Integration。首先总览基于Spring Integration的事件驱动框架的各个组件，接着做一个简单的开发，了解Spring Integration是如何工作的。最后向大家展示一个更加复杂的应用场景，即在此场景下借助JMS，集成组件并贯穿整个ActiveMQ消息总线。

**事件驱动框架**

事件驱动框架是企业级集成领域最重要最成功的模式之一，也是本文关注的重点。在事件驱动框架中，系统发布事件，接着系统中相应的组件就会监听这些特定的事件、或者某种类型的事件。一旦某个感兴趣的事件发生了，组件就会告警，并做出必要的响应。

事件驱动框架的优势是耦合度很低、系统扩展性好，而且生产者无需关心消费者。这就使得在一个已存在或者旧系统中集成一个新的组件变得相对容易：该系统发布 事件，配置新组件用来监听这些事件。所有事件驱动框架交互都是异步的，因此组件可以适时地处理这些消息。试想如果负载增加很大，一个组件处理某个消息可能 需要耗费更多的时间，但这是避免不了的事情。  
（译者注，为了阅读的顺畅性， 本文中的生产者和消费者均指的是message producers、 message consumers）

某个应用响应可能变慢，但本不应该如此。

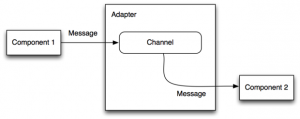
Spring Integration所支持的事件驱动框架基于三个核心组件：

* **消息**作为对象从一个组件传递给另一个组件；
* **通道**用来传递消息，它们可以是同步或者异步的；
* **适配器**调度一个通道输出进入另一个通道的输入中；

图1展示了Spring Integration中消息、通道、适配器之间的关系。

Figure 1. Messages, channels, and adapters

图1.消息、通道、适配器

[](http://jbcdn2.b0.upaiyun.com/2015/07/6bafe2bb51838bf57d6bc856ebcb0821.png)  
注意，一旦组件1发送一个消息到指定通道，适配器会调度通道输出到组件2中。最重要的是适配器指示，任何发送到此通道中的消息都应该定向到组件2中.

你好，Spring Integration！

没有“Hello World”，Java 技术介绍都不算完整。本例中，我利用Spring Integration集成一个小程序，调度一个文本消息，从一个组件传递到另一个组件。通过这个练习，希望大家会对Spring Integration的消息、通道、适配器如何工作有一个更加清晰的认识。（也可以通过最新的[Spring Integration Java文档](http://docs.spring.io/spring-integration/docs/latest-ga/api/)查看每个组件更多详细信息。）

首先，列表1展示applicationContext.xml文件，它像胶水一样将这三个应用组件整合在了一起。

列表1**、**applicationContext.xml

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <beans:beans xmlns:beans="http://www.springframework.org/schema/beans"               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"               xmlns:context="http://www.springframework.org/schema/context"               xmlns="http://www.springframework.org/schema/integration"               xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd                                   http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd                                   http://www.springframework.org/schema/integration http://www.springframework.org/schema/integration/spring-integration.xsd">        <!-- Component scan to find all Spring components -->      <context:component-scan base-package="com.geekcap.springintegrationexample" />        <!-- A Spring Integration channel -->      <channel id="helloWorldChannel" />        <!-- A Spring Integration adapter that routes messages sent to the helloWorldChannel to the bean named "helloServiceImpl"'s hello() method -->      <service-activator input-channel="helloWorldChannel" ref="helloServiceImpl" method="hello" />    </beans:beans> |

注意， <beans>节点定义了约束(schema)和XML命名空间，默认的上下文(xmlns)，定义在 http://www.springframework.org/schema/integration中，无需为通道或者服务激活器节点指定前缀。（这 么做主要是为了XML更易读，稍后我们切换到beans默认上下文，此时就需要为Spring Integration节点指定前缀。)

列表1定义了三个组件：

1. 利用 component-scan，我们就可以在代码中采用诸如@Service或者@Component等注解beans。一旦beans被注解，运行某个 组件扫描，Spring就能找到这些beans。component-scan节点会去扫描这个基础包，进而扫描此包路径下的所有子包。本例中，我们定 义，注释了两种bean：HelloService和GreeterService；
2. HelloService bean打印“Hello，name”到标准输出。GreeterService bean发送一个name到HelloService；
3. helloWorldChannel是一个通道，代码可以给它发送消息；
4. service-activator是一个适配器，指示所有发送到helloWorldChannel的消息，都应转发到helloServiceImpl的hello()方法中。注意，Spring默认bean名是类名，打头的字母需要小写。

列表2展示HelloService接口，接口并不是必需的，比如直接发送消息到某个bean就不需要接口。不过我们习惯了利用Spring定义接口，这样到后面实现改动起来会比较容易。（采用接口，单元测试也变得更加容易。)

列表2 、HelloService.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | package com.geekcap.springintegrationexample.service;    public interface HelloService  {      public void hello( String name );  } |

HelloService接口定义了一个hello()方法，接受一个字符串参数。Spring可以非常智能地找到该方法和其参数签名，并将该消息转换成一个字符串值。

列表3展示了HelloServiceImpl类，该类实现了HelloService接口。

列表3、HelloServiceImpl.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | package com.geekcap.springintegrationexample.service;    import org.springframework.stereotype.Service;    @Service  public class HelloServiceImpl implements HelloService  {      @Override      public void hello(String name)      {          System.out.println( "Hello, " + name );      }  } |

HelloServiceImpl实现了hello()方法，打印“Hello，name”到标准输出。用@Service注解该类，这样定义在 applicationContext.xml文件中的component-scan就可以找到它。注意，服务看起来很标准，没有涉及某个Spring Integration action。

列表4展示了GreeterService，这是greeters需要实现的接口。

列表4、GreeterService.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | package com.geekcap.springintegrationexample.service;    public interface GreeterService  {      public void greet( String name );  } |

列表5展示了GreeterService的接口实现。

列表5、GreeterServiceImpl.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package com.geekcap.springintegrationexample.service;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.integration.MessageChannel;  import org.springframework.integration.support.MessageBuilder;  import org.springframework.stereotype.Service;    @Service  public class GreeterServiceImpl implements GreeterService  {      @Autowired      private MessageChannel helloWorldChannel;        @Override      public void greet(String name)      {          helloWorldChannel.send( MessageBuilder.withPayload( name ).build() );      }  } |

更多的代码信息

GreeterServiceImpl类采用@Service注解，这样Spring可以认定其为一个服务（service）。该类自动注入名为 helloWorldChannel的MessageChannel；只要此处的通道名和applicationContext.xml文件中定义的相 同，Spring就会找到此通道。如果你想重载此MessageChannel，你可以给MessageChannel注解@Qualifier，给此 MessageChannel一个channel bean名字。一旦GreeterServiceImpl’s greet()方法被调用，它会创建并发送一个消息给helloWorldChannel。

MessageChannel是一个接口，定义了两种send()方法：一种用来接收超时，另一种不接受(取决于其实现是否为永久阻塞)。MessageBuilder类为[创建者设计模式（Builder design pattern）](http://www.javaworld.com/article/2077708/core-java/the-java-enclosing-class-as-an-implementation-of-the-builder-pattern.html)一 种实现形式，帮助创建消息。本例中我们将一个字符串传给MessageBuilder，字符串可以用来指定消息头、过期日期时间、优先级、关联ID、回复 和错误通道等等。一旦完成MessageBuilder配置，调用build()方法就会返回一个消息，此消息可以发送给任何一个通道。  
Listing 6 shows the source code for a command-line application that pulls all of our code together.  
列表6展示了某命令行应用源码，自此该应用将之前所有的代码串起来了。

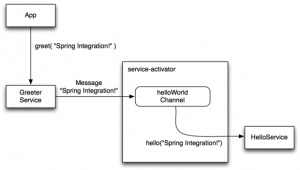
列表6、App.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package com.geekcap.springintegrationexample.main;    import com.geekcap.springintegrationexample.service.GreeterService;  import org.springframework.context.support.ClassPathXmlApplicationContext;    /\*\*   \* Main entry-point into the test application   \*/  public class App  {      public static void main( String[] args )      {          ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext( "applicationContext.xml" );            GreeterService greeterService = applicationContext.getBean( "greeterServiceImpl", GreeterService.class );            greeterService.greet( "Spring Integration!" );      }  } |

App类从classpath路径下加载applicationContext.xml文件，此文件位于src/main/resources路径下，通 过Maven自动地将其嵌入到JAR文件中。接下来，App从application context获取greeterServiceImpl bean，最后调用GreeterService的greet()方法。

A Spring Integration

图2展示了图1的Spring Integration示例图在本例中的具体实现。

图2、你好，Spring Integration!  


下面是集成应用流程：

1. App类调用了GreeterService的greet()方法，把字符串传给”Spring Integration!”给greet()；
2. GreeterService 用wired注入一个名为helloWorldChannel的MessageChannel。此通道利用一个MessageBuilder创建一个消 息，此消息包含”Spring Integration!”字符串，然后将此消息发送给MessageChannel；
3. 配置的service-activator负责将任何发送到helloWorldChannel的消息调度到HelloService的hello()；
4. HelloServiceImpl类的hello()方法调用后，”Hello, Spring Integration!”就会打印到屏幕上。

列表7展示Maven pom.xml文件构建这个应用示例：

列表7、你好，Spring Integration的Maven POM文件

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97 | <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">      <modelVersion>4.0.0</modelVersion>        <groupId>com.geekcap.javaworld</groupId>      <artifactId>HelloSpringIntegration</artifactId>      <version>1.0-SNAPSHOT</version>      <packaging>jar</packaging>        <name>HelloSpringIntegration</name>      <url>http://maven.apache.org</url>        <properties>          <spring.version>3.2.1.RELEASE</spring.version>          <spring.integration.version>2.2.5.RELEASE</spring.integration.version>          <java.version>1.6</java.version>          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>      </properties>        <dependencies>          <!-- Spring Dependencies -->          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-context</artifactId>              <version>${spring.version}</version>          </dependency>          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-core</artifactId>              <version>${spring.version}</version>          </dependency>          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-beans</artifactId>              <version>${spring.version}</version>          </dependency>            <!-- Spring Integration -->          <dependency>              <groupId>org.springframework.integration</groupId>              <artifactId>spring-integration-core</artifactId>              <version>${spring.integration.version}</version>          </dependency>            <!-- Testing -->          <dependency>              <groupId>junit</groupId>              <artifactId>junit</artifactId>              <version>4.11</version>              <scope>test</scope>          </dependency>      </dependencies>        <build>          <plugins>              <plugin>                  <groupId>org.apache.maven.plugins</groupId>                  <artifactId>maven-compiler-plugin</artifactId>                  <configuration>                      <source>${java.version}</source>                      <target>${java.version}</target>                  </configuration>              </plugin>              <plugin>                  <groupId>org.apache.maven.plugins</groupId>                  <artifactId>maven-jar-plugin</artifactId>                  <configuration>                      <archive>                          <manifest>                              <addClasspath>true</addClasspath>                              <classpathPrefix>lib/</classpathPrefix>                              <mainClass>com.geekcap.springintegrationexample.main.App</mainClass>                          </manifest>                      </archive>                  </configuration>              </plugin>              <plugin>                  <groupId>org.apache.maven.plugins</groupId>                  <artifactId>maven-dependency-plugin</artifactId>                  <executions>                      <execution>                          <id>copy</id>                          <phase>install</phase>                          <goals>                              <goal>copy-dependencies</goal>                          </goals>                          <configuration>                              <outputDirectory>${project.build.directory}/lib</outputDirectory>                          </configuration>                      </execution>                  </executions>              </plugin>          </plugins>            <finalName>hello-spring-integration</finalName>      </build>  </project> |

POM文件引入Spring核心、上下文、bean依赖，同时引入了特定的Spring Integration依赖，并且定义了三个插件：

* maven-compiler-plugin指示Spring采用Java1.6进行构建。
* maven- jar-plugin指示Spring包含JAR文件classpath路径下lib目录中的所有文件。同样它会指导Spring为此JAR文件生成名为 com.geekcap.springintegrationexample.main.App的main-class。只要对此文件执行Java -jar命令，该类就会执行。
* maven-dependency-plugin指示Maven将所有的项目(project)依赖拷贝到target/lib目录中。

可以通过下面的命令构建此项目：

|  |  |
| --- | --- |
| 1 | mvn clean install |

正确配置以后，target目录就会执行源代码：

|  |  |
| --- | --- |
| 1 | java -jar hello-spring-integration.jar |

Spring logger 会输出几行日志，紧随其后的是本程序的输出：

|  |  |
| --- | --- |
| 1 | Hello, Spring Integration! |

注意，采用log4j.properties文件配置的Spring logger日志相对简洁，此properties文件改变了Spring组件中的日志级别。

Spring Integration集成网关代理

除了发送消息，有时还需要做出回应。倘若你想执行某个服务的某个方法并做出回应，那么你可以使用网关代理。目前先介绍这些，稍后我会做出解释。

首先，创建一个通道和一个服务激活器(service-activator),就像我们前面在列表1中做的那样，不过这次还需要添加一个网关节点。

列表8 、New channel和service-activator

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <!-- A Spring Integration channel for use by our gateway -->     <channel id="helloWorldWithReplyChannel" />       <!-- A Spring Integration adapter that routes messages sent to the helloWorldChannel to the bean named "helloServiceImpl"'s getHelloMessage() method -->     <service-activator input-channel="helloWorldWithReplyChannel" ref="helloServiceImpl" method="getHelloMessage" />       <!-- Define a gateway that we can use to capture a return value -->     <gateway id="helloWorldGateway" service-interface="com.geekcap.springintegrationexample.service.HelloService" default-request-channel="helloWorldWithReplyChannel" /> |

网关是由一个接口定义的，在本例中该接口是由HelloService实现的。它定义了一个可用的默认请求/输入通道。服务激活器(service-activator)调用一个新方法getHelloMessage()，而非hello()。新方法返回”Hello, NAME”。

定义网关之后，就可以像列表5那样，将其自动注入到GreeterService。这次没有注入通道，而是注入网关，如列表9其类型为HelloService。

列表9 、Updated GreeterServiceImpl.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | package com.geekcap.springintegrationexample.service;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.integration.MessageChannel;  import org.springframework.integration.support.MessageBuilder;  import org.springframework.stereotype.Service;    @Service  public class GreeterServiceImpl implements GreeterService  {      @Autowired      private MessageChannel helloWorldChannel;        @Autowired      private HelloService helloWorldGateway;        @Override      public void greet(String name)      {          helloWorldChannel.send(MessageBuilder.withPayload(name).build());      }        @Override      public void greet2(String name)      {          System.out.println( helloWorldGateway.getHelloMessage( name ) );      }  } |

更新后的GreeterServiceImpl类自动注入了helloWorldGateway，后者会在applicationContext.xml 文件中被解析，它的类型为HelloService；新写入的方法greet2()会调用此helloWorldGateway，就像在调用 HelloService。从方法的角度看，greet2()仅仅调用了一个HelloService，此接口就如同一个服务bean：方法无需知晓 Spring Integration所涉及的具体事务。

列表10是更新后的App类，此类会调用新写入的greet2()方法.

列表10、更改后的 App.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | package com.geekcap.springintegrationexample.main;    import com.geekcap.springintegrationexample.service.GreeterService;  import org.springframework.context.support.ClassPathXmlApplicationContext;    /\*\*   \* Main entry-point into the test application   \*/  public class App  {      public static void main( String[] args )      {          ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext( "applicationContext.xml" );            GreeterService greeterService = applicationContext.getBean( "greeterServiceImpl", GreeterService.class );            greeterService.greet( "Spring Integration!" );            greeterService.greet2( "Spring Integration (with response)!");      }  } |

略去Spring的日志部分，此段代码的输出结果如下：

|  |  |
| --- | --- |
| 1  2 | Hello, Spring Integration!  Hello, Spring Integration (with response)! |

[重构](http://www.amazon.cn/gp/product/B003BY6PLK/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=importnew-23&linkCode=as2&camp=536&creative=3200&creativeASIN=B003BY6PLK)后的新轮子

这背后发生了什么？这里我们对重构应用做一个总结：

1. App类调用GreeterService的greet2()方法；
2. greet2()调用getHelloMessage()方法，此方法被认为是一个HelloService方法(确切的说它是一个网关)；
3. 网关实现了HelloService接口中的方法，该方法通过helloWorldWithReplyChannel通道对请求进行调度；
4. 通过配置服务激活器(service-activator)，任何发给helloWorldWithReplyChannel的消息都会传给helloServiceImpl的getHelloMessage()方法；
5. helloServiceImpl构造了本次响应并将其返回；
6. 服务激活器(service-activator)通过下面任意一种方式处理响应：一种是定义在服务激活器中的输出通道(output-channel),另一种是定义在消息头重的回复通道。网关自动创建一个临时的匿名、点对点回复通道；监听此通道，并将其添加到消息的replyHeader中；
7. 通道接受这个响应，将其作为消息，通过回复通道，将其转换成服务中定义的返回值；
8. 最后网关返回响应给调用者，如本例的GreeterServiceImpl。

Spring Integration集成JMS和ActiveMQ

到目前为止，通过传递消息我们抽象了一个服务调用。同样，我们在一个网关和通道后面隐藏了一个服务调用。以上两部分给了我们充足的信息，从概念上去理解 Spring Integration是如何工作的。接下来我们利用这些知识来拓展一个更加真实的企业级应用场景，创建通道，让其与某个运行在[ActiveMQ](http://activemq.apache.org/)上的JMS(Java Message Service)主题进行通信。ActiveMQ是一个支持JMS API的开源消息代理，它由Java语言写成，遵循apache协议，可免费使用。

JMS定义了两种消息传递方式：主题和队列。主题采用发布订阅方式进行运作，而队列采用点对点方式运作。发布订阅范例指的是一个消息生成者发布一个消息后，零个或者多个消费者接受这些消息。点对点范例指的是某个消息生产者发布一个消息，就有一个对应的消费者。用队列做两个组件之间异步通信是很棒的，而主题特别适合事件驱动框架。

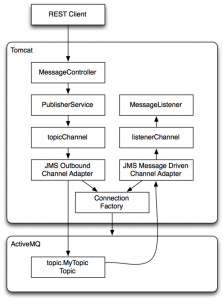
基于事件驱动框的应用要点是将消息生产者与其消费者解耦，方式和发布订阅范例一样；利用主题范例，一个组件通知变化或者更新，但仅有订阅了主题的消费者会接收这些消息。生产者并不知道消费者是谁，也无需关心。这是一个很好的松耦合例子。

为了展示企业级系统中发布订阅消息方式，我们会构建两个组件：

* **PublisherService：**一个发布消息给主题的组件；
* **MessageListener：**一个可以订阅主题，并接收消息的组件。

为了将消息传给一个RESTful服务-（作为一种发布消息到主题的方式，）我们需要围绕PublisherService构建基础组件。图3显示了各个组件之间是如何交互的，接下来的是具体细节。

图3、 JMS Spring Integration示例



下面是图3具体流程：

* REST客户端发布消息到MessageController，此Controller为Spring MVC Controller
* MessageController调用PublishService的 send() 方法发布一个消息到topicChannel
* 配置一个JMS 输出 channel适配器将topicChannel中的消息调度到topic.myTopic
* ConnectionFactory中定义了JMS配置文件
* 配置一个JMS消息驱动通道适配器用来监听topic.myTopic主题，并发送消息到listenerChannel
* 配置一个service-activator调度listenerChannel中的消息到messageListenerImpl的processMessage()方法中
* messageListenerImpl类接收这些消息，并对其进行处理，显示在屏幕上

本应用的配置在列表11中展示。

列表11、springintegrationexample-servlet.xml (Application Context)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"         xmlns:context="http://www.springframework.org/schema/context"         xmlns:int="http://www.springframework.org/schema/integration"         xmlns:int-jms="http://www.springframework.org/schema/integration/jms"         xmlns:oxm="http://www.springframework.org/schema/oxm"         xmlns:int-jme="http://www.springframework.org/schema/integration"         xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd                  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd                  http://www.springframework.org/schema/integration http://www.springframework.org/schema/integration/spring-integration.xsd                  http://www.springframework.org/schema/integration/jms http://www.springframework.org/schema/integration/jms/spring-integration-jms.xsd                  http://www.springframework.org/schema/oxm http://www.springframework.org/schema/oxm/spring-oxm-3.0.xsd">        <!-- Component scan to find all Spring components -->      <context:component-scan base-package="com.geekcap.springintegrationexample" />        <bean>          <property name="order" value="1" />          <property name="messageConverters">              <list>                  <!-- Default converters -->                  <bean/>                  <bean/>                  <bean />                  <bean/>                  <bean/>                  <bean />              </list>          </property>      </bean>        <!-- Define a channel to communicate out to a JMS Destination -->      <int:channel id="topicChannel"/>        <!-- Define the ActiveMQ connection factory -->      <bean id="connectionFactory">          <property name="brokerURL" value="tcp://localhost:61616"/>      </bean>        <!--          Define an adaptor that route topicChannel messages to the myTopic topic; the outbound-channel-adapter          automagically finds the configured connectionFactory bean (by naming convention)        -->      <int-jms:outbound-channel-adapter channel="topicChannel"                                        destination-name="topic.myTopic"                                        pub-sub-domain="true" />        <!-- Create a channel for a listener that will consume messages-->      <int:channel id="listenerChannel" />        <int-jms:message-driven-channel-adapter id="messageDrivenAdapter"                                              channel="listenerChannel"                                              destination-name="topic.myTopic"                                              pub-sub-domain="true" />        <int:service-activator input-channel="listenerChannel" ref="messageListenerImpl" method="processMessage" />    </beans> |

列表11的最开始为Spring MVC建立了AnnotationMethodHandlerAdapter，这个超越了本文的范畴。你需要掌握的是，消息转换器展示的内容由Spring Controllers返回。本例中最重要的部分如下：

* connectionFactory定义了ActiveMQ连接参数，ActiveMQ安装在本机上，默认配置，端口号为61616。
* topicChannel定义了一个用来发布消息的通道(如下)。
* outbound-channel-adapter定 义在Spring Integration JMS命名空间中，将topicChannel中的消息发布到topic.myTopic，并将其设置为一个主题范例。pub-sub-domain设置 为true意味着此为主题范例，设置为false则为队列范例；需要注意的是，outbound-channel-adapter通过connectionFactory的bean名找到ActiveMQ配置。
* listenerChannel定义了一个用来处理消息的channel。
* message-driven-channel-adapter定义了一个适配器，用来监听topic.myTopic主题中的消息，并将其派发给listenerChannel。
* service-activator再将listenerChannel中的消息派发给messageListenerImpl的processMessage()方法。

发布：MessageController.java

你可以[下载源码](http://images.techhive.com/assets/2014/04/04/jw-osjp-spring-integration-example-src.zip)获取所有文件。我仅仅回顾一下重点。首先，列表12展示了MessageController类，用来消息发布。

**列表12、 MessageController.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | package com.geekcap.springintegrationexample.web;    import com.geekcap.springintegrationexample.service.PublishService;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.http.HttpStatus;  import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.RequestBody;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.ResponseBody;    import javax.servlet.http.HttpServletResponse;    @Controller  public class MessageController  {      @Autowired      private PublishService publishService;        @RequestMapping( value = "/message", method = RequestMethod.POST )      @ResponseBody      public void postMessage( @RequestBody com.geekcap.springintegrationexample.model.Message message, HttpServletResponse response )      {          // Publish the message          publishService.send( message );            // Set the status to 201 because we created a new message          response.setStatus( HttpStatus.CREATED.value() );      }    } |

MessageController是一个Spring MVC controller，实现了RESTful web service处理POST到 /message 资源。要求一 个JSON对象，后面我们会提到，Spring MVC最自动将其转换为一个com.geekcap.springintegrationexample.model.Message。 MessageController将此message传给PublishService，后者调用send()方法，将Message传递给调用者。

列表13展示了PublishServiceImpl类：

列表13、PublishServiceImpl.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package com.geekcap.springintegrationexample.service;    import com.geekcap.springintegrationexample.model.Message;  import org.apache.log4j.Logger;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.beans.factory.annotation.Qualifier;  import org.springframework.integration.MessageChannel;  import org.springframework.integration.message.GenericMessage;  import org.springframework.integration.support.MessageBuilder;  import org.springframework.stereotype.Service;    @Service  public class PublishServiceImpl implements PublishService  {      private static final Logger logger = Logger.getLogger( PublishServiceImpl.class );        @Autowired      private MessageChannel topicChannel;        @Override      public void send( Message message )      {          logger.info( "Sending message to message channel: " + message );          topicChannel.send( MessageBuilder.withPayload( message.toString() ).build() );      }  } |

PublishServiceImpl类注入了topicChannel，topicChannel由application context创建；借助MessageBuilder类发送一个字符串类型的Message。这些都是发布一个消息到ActiveMQ主题所必需的。重 点是，Spring Integration利用outbound-channel-adapter查找connectionFactory，最终找到主题目标 topic.myTopic，并且将消息发送给它。

订阅：MessageListenerImpl

在操作的另一端，列表14展示了MessageListenerImpl类的源码：

列表14 、MessageListenerImpl.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package com.geekcap.springintegrationexample.listener;    import org.apache.log4j.Logger;  import org.springframework.integration.annotation.ServiceActivator;  import org.springframework.stereotype.Service;    @Service  public class MessageListenerImpl  {      private static final Logger logger = Logger.getLogger( MessageListenerImpl.class );        public void processMessage( String message )      {          logger.info( "Received message: " + message );          System.out.println( "MessageListener::::::Received message: " + message );      }  } |

MessageListenerImpl类作为一个服务，定义了一个方法：processMessage(String)。定义在application context中的message-driven-channel-adapter，负责监听发布在topic.myTopic主题中的消息，并将其派发 给listenerChannel。service-activator再将listenerChannel中的消息派发给 messageListenerImpl类的processMessage(String)方法。正如你所看到 的，MessageListenerImpl对于如何回应JMS 消息一无所知，所有的细节都交由application context配置文件处理。

构建和运行应用

此项目的POM文件如列表15所示

列表15 、pom.xml

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152 | <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">      <modelVersion>4.0.0</modelVersion>      <groupId>com.geekcap</groupId>      <artifactId>spring-integration-example</artifactId>      <packaging>war</packaging>      <version>1.0-SNAPSHOT</version>      <name>spring-integration-example Maven Webapp</name>      <url>http://maven.apache.org</url>        <properties>          <spring.version>3.2.1.RELEASE</spring.version>          <spring.integration.version>2.2.5.RELEASE</spring.integration.version>          <servlet-api.version>2.5</servlet-api.version>          <java.version>1.6</java.version>          <jackson.version>1.9.12</jackson.version>          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>      </properties>        <dependencies>          <!-- Spring Dependencies -->          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-context</artifactId>              <version>${spring.version}</version>          </dependency>          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-core</artifactId>              <version>${spring.version}</version>          </dependency>          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-web</artifactId>              <version>${spring.version}</version>          </dependency>          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-webmvc</artifactId>              <version>${spring.version}</version>          </dependency>          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-beans</artifactId>              <version>${spring.version}</version>          </dependency>          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-oxm</artifactId>              <version>${spring.version}</version>          </dependency>          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-orm</artifactId>              <version>${spring.version}</version>          </dependency>          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-jms</artifactId>              <version>${spring.version}</version>          </dependency>          <dependency>              <groupId>javax.servlet</groupId>              <artifactId>servlet-api</artifactId>              <version>${servlet-api.version}</version>              <scope>provided</scope>          </dependency>            <!-- Spring Integration-->          <dependency>              <groupId>org.springframework.integration</groupId>              <artifactId>spring-integration-core</artifactId>              <version>${spring.integration.version}</version>          </dependency>          <dependency>              <groupId>org.springframework.integration</groupId>              <artifactId>spring-integration-jms</artifactId>              <version>${spring.integration.version}</version>          </dependency>            <!-- Logging: Log4J -->          <dependency>              <groupId>commons-logging</groupId>              <artifactId>commons-logging</artifactId>              <version>1.1.1</version>          </dependency>          <dependency>              <groupId>org.slf4j</groupId>              <artifactId>slf4j-log4j12</artifactId>              <version>1.5.8</version>          </dependency>          <dependency>              <groupId>log4j</groupId>              <artifactId>log4j</artifactId>              <version>1.2.15</version>              <exclusions>                  <exclusion>                      <groupId>javax.mail</groupId>                      <artifactId>mail</artifactId>                  </exclusion>                  <exclusion>                      <groupId>javax.jms</groupId>                      <artifactId>jms</artifactId>                  </exclusion>                  <exclusion>                      <groupId>com.sun.jdmk</groupId>                      <artifactId>jmxtools</artifactId>                  </exclusion>                  <exclusion>                      <groupId>com.sun.jmx</groupId>                      <artifactId>jmxri</artifactId>                  </exclusion>              </exclusions>          </dependency>            <!-- Include Jackson so that we can render JSON responses -->          <dependency>              <groupId>org.codehaus.jackson</groupId>              <artifactId>jackson-jaxrs</artifactId>              <version>1.6.1</version>          </dependency>            <!-- Include ActiveMQ -->          <dependency>              <groupId>org.apache.activemq</groupId>              <artifactId>activemq-core</artifactId>              <version>5.7.0</version>          </dependency>            <!-- JUnit -->          <dependency>              <groupId>junit</groupId>              <artifactId>junit</artifactId>              <version>4.11</version>              <scope>test</scope>          </dependency>      </dependencies>        <build>          <plugins>              <plugin>                  <groupId>org.apache.maven.plugins</groupId>                  <artifactId>maven-compiler-plugin</artifactId>                  <configuration>                      <source>${java.version}</source>                      <target>${java.version}</target>                  </configuration>              </plugin>          </plugins>          <finalName>spring-integration-example</finalName>      </build>  </project> |

应用依赖

本例包括以下依赖：

* Core Spring
* Spring MVC
* Spring JMS
* Spring Integration
* Spring Integration JMS

你需要给集成后的组件添加额外的依赖，其中的一些配置可以有多个选项，但它们都可以整合到通道和适配器中。

部署Tomcat和ActiveMQ

接下来，利用下面的命令构建此应用：

|  |  |
| --- | --- |
| 1 | mvn clean install |

可以从[Apache Tomcat网站](http://tomcat.apache.org/)下载Tomcat，在本地解压，执行Tomcat bin目录下的startup.sh或者startup.bat文件，启动tomcat。将最终的WAR文件拷贝到Tomcat webapps目录下方便部署到tomcat中。也可以从[Apache ActiveMQ网站](http://activemq.apache.org/)下载ActiveMQ，在本地解压，执行bin目录下的命令:

|  |  |
| --- | --- |
| 1 | ./activemq start |

或者window操作系统：

|  |  |
| --- | --- |
| 1 | activemq start |

你可以执行activemq stop停止ActiveMQ，执行shutdown.sh或者shutdown.bat关闭Tomcat。建议先启动ActiveMQ，这样本例中的应用就能够连接到topic.myTopic主题，并开始消息监听。

整合应用和Spring Integration

大家已经了解消息、通道、适配器、网关如何一起工作，从消息生产者到消息消费者去抽象消息；了解ActiveMQ 和 JMS如何集成到Spring Integration中。Spring Integration最强大的地方，当开始思考构建模块和复用服务，你就可以用Spring Integration整合application工作流。比如，一个事件驱动应用既可以发布一个轻量级负载消息(鉴别某个系统，该系统产生受影响资源的事件和ID),也可以发布一个重量级负载消息(包含所有修改资源的内容)。

使用轻量级负载的好处是，倘若资源经常改变，无需担心重量级负载内容：回到系统记录–其实是一种资源，寻找最新的改动资源版本。缺点是每一次系统产生一个事件，所有监听器都会调用系统回复，这样很可能会增加负载。因此，折衷的办法是系统产生一个重量级负载，禁止监听器回掉。倘若系统同一资源产生多个事件，其中一些负载可能过期，调节性事件负载落到监听器上。

我们可以配置Spring Integration结合入站的、出站通道和模块组件去处理这两种场景。一个具体的例子，前面例子中定义的监听器，这里采用Spring Integration配置此监听器：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <!-- Create a channel for a listener that will consume messages-->      <int:channel id="listenerChannel" />        <int-jms:message-driven-channel-adapter id="messageDrivenAdapter"                                              channel="listenerChannel"                                              destination-name="topic.myTopic"                                              pub-sub-domain="true" />        <int:service-activator input-channel="listenerChannel" ref="messageListenerImpl" method="processMessage" /> |

从理论上讲，负载已经存在，并传递给了messageListenerImpl的processMessage()方法。倘若没有传递过去会怎样？倘若发送给topic.myTopic的消息仅仅是改动资源的ID会怎样？这取决于系统回掉以及获取负载？重写逻辑？幸运的是，这些都不需要。我们可以定义另外一个bean来获取这些负载，派发消息到这个bean，并将此bean的响应传给messageListenerImpl bean。

列表16展示了newRetrievePayloadImpl类的源码。

列表16、 RetrievePayloadImpl.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | package com.geekcap.springintegrationexample.service;    import org.springframework.stereotype.Service;    @Service  public class RetrievePayloadServiceImpl implements RetrievePayloadService  {      @Override      public String getPayload(String id)      {          // Go back to the SOR and retrieve the payload for the specified id ...          return "Payload for " + id;      }  } |

这个类理论上会返回到系统记录(SOR)，通过特定的ID为此组件获取负载并返回。本例中，仅仅返回字符串：”Payload for …”

现在我们在message-driven-channel和messageListenerImpl之间注入这个新的服务，如列表17。

列表17  、Updated applicationContext.xml file with the RetrievePayloadService

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <!-- Create a channel for a listener that will consume messages-->      <int:channel id="listenerChannel" />        <int-jms:message-driven-channel-adapter id="messageDrivenAdapter"                                              channel="getPayloadChannel"                                              destination-name="topic.myTopic"                                              pub-sub-domain="true" />        <int:service-activator input-channel="listenerChannel" ref="messageListenerImpl" method="processMessage" />        <int:channel id="getPayloadChannel" />        <int:service-activator input-channel="getPayloadChannel" output-channel="listenerChannel" ref="retrievePayloadServiceImpl" method="getPayload" /> |

不再直接将message-driven-channel-adapter传给listenerChannel，相反地将其传给getPayloadChannel。接着getPayloadChannel调用retrievePayloadServiceImpl的getPayload()方法，将其输出传给listenerChannel，后者将增加后的负载传给messageListenerImpl的processMessage()方法。通过配置，我们可以让MessageListenerImpl类独自完成所有这些工作，通过其它服务传递这些消息。

需要注意的是，一旦服务器暴露了某一个RESTful接口，那么RetrievePayloadService会被HTTP出站网关完整地取代。比如，列表18展示了某个项目配置，从一个Message对象中获取”ResourceLink” ，并将其作为HTTP请求的一部分。

列表18、利用一个HTTP输出网关回调一个服务

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <int:channel id="createEntityChannel" />      <int-http:outbound-gateway request-channel="createGuestChannel"                                 url="http://localhost:8080{link}"                                 http-method="GET"                                 expected-response-type="com.mycompany.model.Entity"                                 reply-channel="transformEntityChannel" >          <int-http:uri-variable name="link" expression="payload.getResourceLink()" />      </int-http:outbound-gateway> |

列表18仅仅是applicationContext.xml文件很小的一部分，但足以说明一个消息如何被发送到 createEntityChannel，接着被传递给某HTTP出站网关，后者进而从http://localhost:8080/link获取 com.mycompany.model.Entity，接着将其传递给transformEntityChannel。Spring Integration提供了一组丰富的适配器和网关，这样你就可以专注于业务，不用关心如何写代码去调用这些服务、发布消息、读取主题等等。

结语

Spring Integration有助于解决企业级整个问题，它实现了*Enterprise Integration Patterns*书中定义的设计模式。包括将消息消费者从消息生成者那里剥离的异步消息范例。不是方法直接调用，而是由Spring Integration发送一个消息到某个通道。一个适配器或者网关管理这个通道，并将此消息发送到恰当的目的地，不管目的地是运行在相同虚拟机上的其它服务，还是在某个企业级服务总线上的其它数据中心运行的服务。

在[Java开源项目](http://www.javaworld.com/blog/open-source-java-projects/)中，我定义了消息、通道、适配器、网关。接着展示了如何利用Spring Integration从一个组件到另一个组件传递消息，如何处理响应，如何利用JMS和ActiveMQ作为消息总线集成这些组件。最后，展示了如何通过写模块组件，定义通道控制消息流来整合应用。

本文仅仅是对Spring Integration做了一些初级的介绍。JMS只是适配器的一种，Spring Integration同样支持其它类型的适配器。比如利用Spring Integration借助邮件、文件系统、web service调用、推特等传递消息。关键是只有确保消息发送给正确的组件，才能得到你想要的结果。

原文链接： [javaworld](http://www.javaworld.com/article/2142107/spring-framework/open-source-java-projects-spring-integration.html) 翻译： [ImportNew.com](http://www.importnew.com/)- [乔永琪](http://www.importnew.com/author/qiaoyongqi)  
译文链接： <http://www.importnew.com/16538.html>