# Mule ESB 3使用要点

## Mule ESB 3使用要点

Mule是一个灵活的消息处理和集成框架。你使用Mule的方式取决于你要尝试解决的问题。Mule3提供了多种配置构建方法，这些方法可以根据需要被混合和装配，来实现你的方案。

* 理解Mule配置
* 在流、模式或服务之间进行选择
* 消息源和消息处理器
* 配置组件
* 使用传输器做连接
* 配置端点
* 使用过滤器
* 使用转换器
* 使用Mule云连接来连接SaaS，社交媒体和电子商务
* Mule查询语言
* 使用表达式
* 消息属性域
* 事务管理
* 配置安全
* 错误处理
* 使用Web Services

## 理解Mule配置

### 理解Mule配置

[XML配置相关] [Schema参考] [默认值] [枚举值] [强类型语言的优点] [Spring配置相关] [Spring Beans] [Spring属性] [属性占位符] [Mule配置相关] [全局元素] [流] [配置模式] [服务] [自定义元素] [系统级配置] [管理器] [代理]

#### XML配置相关

Mule使用XML配置来定义每个应用，它完整的描述了运行应用需要的组成部分。一个基本的Mule应用可以使用非常简单的配置，例如：



下面我们将仔细查看这一配置的所有片段的细节。目前，尽管它很简单，却是一个完整的应用，并且非常易读：甚至对Mule只有短时间了解的人也能够解释清楚，它是在从标准输入向标准输出拷贝消息。

##### Schema参考

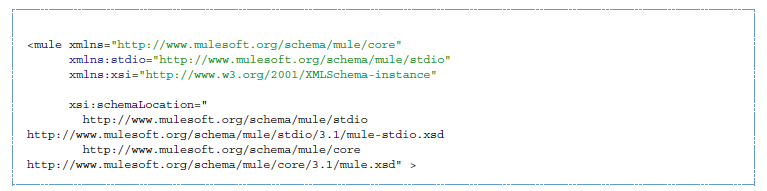
Mule配置的语法是由一组XML schema文件定义的。每一个配置列出了它用到的schema，并给出了它们的URL地址。它们中的大部分是你正在使用的Mule版本所对应的Mule schema， 但是另外也可能有一些第三方的schema，例如：

* Spring schema，定义了用到的所有Spring标签元素（比如Spring beans）的语法
* CXF schema，用于Mule CXF模块所处理的Web Services的配置

一个配置文件中引用到的每个schema都由两部分数据定义：

* 名字空间，这是一个URI
* 位置，这是一个URL

配置文件既定义了schema的名字空间URI作为XML名字空间，同时又关联了schema的名字空间和位置。像我们从上面的配置文件中看到的一样，这是由顶级元素mule标签来完成的：



这里将mule core的schema名字空间被定义为配置文件的默认名字空间。它是这里默认名字空间的最佳选择，因为有这么多配置的元素是属于核心名字空间的。

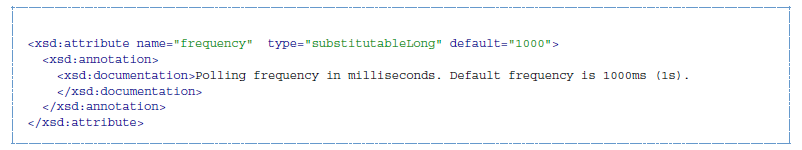
Mule stdio传输器允许使用标准I/O进行通信的，它的名字空间使用“stdio”作为前缀。Mule模块或者传输器里的约定是使用它们的名字作为名字空间的前缀。

Xsi:schema属性关联名字空间到它们的位置，这里它提供了stdio schema和mule core schema的位置。

Mule配置中包含这些部分是必须的，因为它们保证了schema能够被查找到，这样配置文件才能使用它们进行检验。

##### 默认值

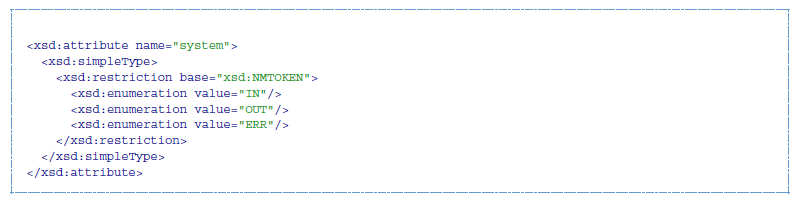
除定义了元素和属性的语法外，schema还可以给它们定义默认值。众所周知，这是对于让你的配置可读是极其有用的，因为它们不会因为不必要的信息变得杂乱。默认值可以通过schema本身来查找，或者在Mule的模块和传输器的该当中也定义了这些值。例如，用于重复地向端口轮询的<poll>元素，它的定义包含了以下的属性定义：



只有当要覆盖默认的1s的值时，才需要指定这一属性。

##### 枚举值

许多Mule的属性只可以接受一个限定的集合内的值。它们在schema中被定义为枚举值，来保证只有这些值才能被指定。这里是一个stdio传输器的例子：



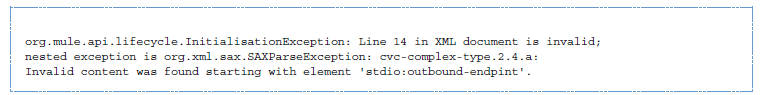
这强制指定了只接受有IN、OUT和ERR，它们分别代表标准输入、输出和错误。

##### 强类型语言的优点

要引用到所有schema的需求好像非常的笨重，但是它有两个远超出所付出的精力的优势。

第一，它帮助你一次就能创建出有效的配置。主流的集成开发环境都提供了schema感知的XML编辑器。因此，当你创建和编辑你的配置时，IDE会提示你在每个点上可以接受的元素和属性，在你只输入了几个字符后实例它们的名字，以及高亮指示出需要修改的输入错误。同样的，它提供了对枚举值的填充的相同的帮助功能。

第二，它允许Mule在应用启动时验证你的配置。不像一些其他的基于配置的系统那样，只是无声的忽略它们不认识的元素和属性，Mule会捕获这些错误，这样你就可以修正它们。例如，假设在上面的配置文件中，我们拼写错误了“outbound-endpoint”，一旦应用尝试启动，结果立即就会出现错误：



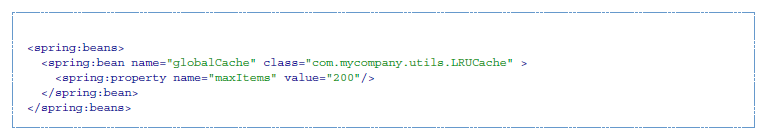
它直接指出了需要修正的那一行。这远比只是简单的忽略问题，让你困扰为什么没有任何输出要有用得多。

#### Spring配置相关

Mule解析配置的方便是嵌入于Spring中的，因此除了定义Mule相关的构造方法，Mule配置中可以做到Spring配置能做的所有事情：创建Sprign Bean，配置list和map，定义属性占位符等等。我们将在下面的章节中进行详细的研究。注意，一如往常，引用合适的schema是必不可少的。

##### Spring Beans

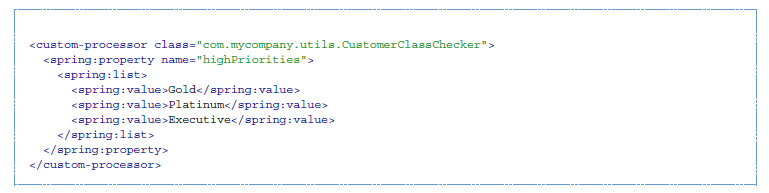
Mule配置中对Spring最简单的使用就是配置Spring bean。这些bean会和Mule相关的对象一起被 放到Mule注册中心里，你可以通过名字查找到任意自定义的Java对象，例如，自定义组件，你可以使用所有Spring的功能来创建它们，例如：



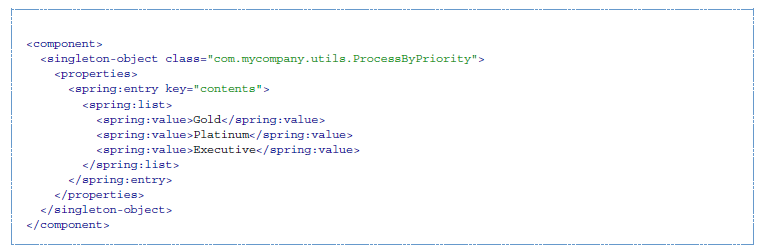
##### Spring配置

在Mule配置中，有很多地方都可以使用自定义的Java对象：自定义的转换器、过滤器、消息处理器等等。所有情况下，都可能要指定一个要实例化的类，和一组Spring属性用于配置这个实例。再说一次，你可以在属性中使用Spring语法的全集，包括list、map等等。

这里是一个例子：



为了在创建Java对象上允许更多的控制，创建自定义组件的语法稍有不同。例如，创建一个单例：



##### 属性占位符

Mule配置可以包含对属性占位符的引用，来保证能够引用到配置文件之外的值。这特性的一个重要的使用场景就是用户名和密码，它们应该使用一种更安全的方式来指定。属性占位符的语法就是简单的：${name}，这里的name就是一个Java属性文件里的一个属性。

这里是一个使用了属性占位符，以及它引用到的属性的例子：

配置文件：



属性文件：



注意这里给出的文件路径是基于classpath路径，另一种选择是使用URL，例如file:///etc/mule/conf/my-mule-app-override.properties。如上所示，也可以指定一组属性文件，它们中间使用逗号分隔。

#### Mule配置相关

##### 全局配置

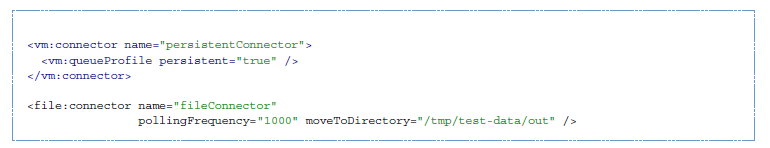
许多Mule元素可以在顶级进行指定，也就是说，作为最外层的mule元素的直接子元素。这些全局元素都有名称，这样才能被使用的地方被引用。注意对全局元素，Mule配置中使用了单一的，扁平的名字空间。任意两个全局元素不能共享同样的名称，即使它们是全然不同类型的东西，比如端点和过滤器。

我们来看一下最常见的全局元素：

###### 连接器

连接器是Mule传输器的具体实例，它的属性描述了传输器是怎样使用的。所有Mule端点使用的连接器都一个继承了连接器属性的相同的传输器。

这里是连接器的例子：



Vm连接器指定了它所有的端点都使用持久队列。文件连接器指定了它的每个端点一秒轮询一次，同时还指定了一旦执行处理，这些文件会被移动到的目录。

注意属性可能被属性指定，也可能被子元素指定。你可以通过查看连接器对应的传输器的参考文档来确定怎么来配置连接器的属性。

端点和它的连接器的关系其实是相当灵活的：

* 如果端点使用名字指定了连接器，它就会使用这个连接器。当然，如果端点和连接器使用了不同传输器，就会出错。
* 如果端点上没有指定连接器，而它的传输器有一个连接器，那端点就使用这个连接器。
* 如果端点上没有指定连接器，并且它的传输器也没有连接器，Mule会为这个传输器的所有端点创建一个默认的连接器供它们使用。
* 如果端点上没有指定连接器，同时它的传输器上有多于一个的连接器，就会出错。

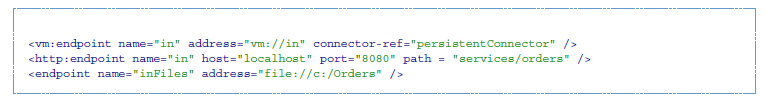
关于连接器和端点的传输器相关的信息，可以参考MULE3USER:Available Transports.

###### 端点

Mule端点是一个消息读出（入站）和写入（出站）的对象，它上面指定了一些属性，定义了这个过程是怎么完成的。端点可以使用两种方式指定：

* 作为全局元素配置的端点称为全局端点。流或者服务中配置的入站或者出站端点，可以使用ref属性引用一个全局端点。
* 流或者服务中配置的入站或者出站端点，可以不引用全局端点，直接进行配置。

全局端点指定了一组属性，包括它的位置。引用了这个全局端点的入站和出站端点都会继承它的属性。这里是几个全局端点的例子：

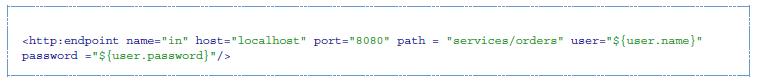


Vm端点指定了它的位置，并引用了上面的连接器。它使用了一个通用的address属性指定它的位置。Http端点使用了默认的http连接器。因为它被明确的配置为http端点，因此它可以使用http相关的属性host、port和path来指定它的位置。File端点指定了它读出（或写入）的目录，它会使用默认的文件连接器。因为它被配置为通用的端点，因此它必须使用address来指定位置。

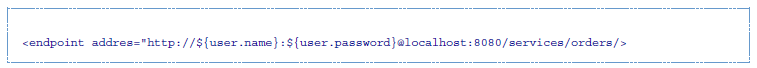
注意每个端点都会使用一种特定的传输器，但它可以通过两种不同的方式指定：

* 如果元素有前缀，它可以使用前缀对应的传输器，
* 如果没有前缀，前缀就会通过address属性来确定。

前缀风格的更值得推荐，尤其是位置配置会比较复杂时，比较一下



和

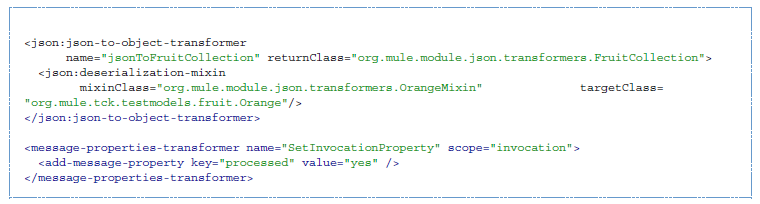


端点上面一个最重要的属性是它的消息交换方式（简称MEP），也就是说，消息是不是单向的或者请求返回响应的。这个可以在多个层次上指定：

* 一些传输器只支持一种MEP。例如，imap是单向的，因为当它读了一封email时，没有任何响应会被发送出去。另一种，servlet，总是请求-响应类型的。
* 每种传输器有一种默认的MEP。JMS默认是单向的，因为JMS消息通常不会和响应关联起来。http默认是请求-响应的，因为HTTP协议天生的对每个请求都有一个响应。
* 端点上可以定义MEP，尽管，只有MRP对它们的传输器合法，才能被接受。

###### 转换器

转换器是一个传输当前Mule消息的对象。Mule核心定义了转换器的基本集合，许多模块和传输器中定义了更多的转换器，例如JSON模块定义了的能够在对象和JSON间相互转换的转换器，而Email传输器定义了在二进行数组和MIME消息间转换的转换器。每种类型的转换器都通过定义XML配置定义了它们的属性。这里是一个转换器的例子：



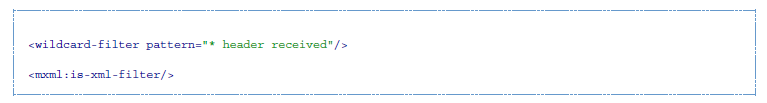
上面的转换器转换消息到JSON，指定了特定的处理过程来处理对org.mule.tck.testmodels. fruit.Orange类的转换。下面的转换器负责添加一个invocation-scoped属性到当前的消息上。

像端点一样，转换器可以被配置成全局元素，被使用到的地方引用，或者也可以在使用点上进行直接配置。

更多Mule转换器的内容，参考Using Transformers。

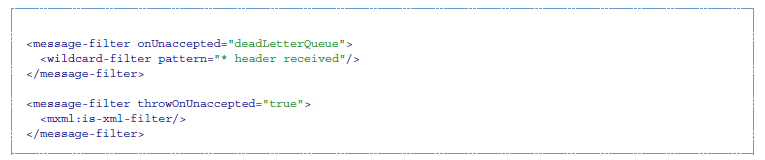
###### 过滤器

过滤器是确定消息是不是应该被处理的对象。和转换器一样，Mule核心定义了过滤器的基本集合，许多模块和传输器定义了更丰富的过滤器。这里是过滤器的一些例子：



上面的过滤器保证了只有当前消息匹配指定的模式时，才能被继续处理。下面的的过滤器只有当消息是XML文档时，才会继续处理。

还有几个搞定了其他的过滤器功能的特别的过滤器，第一个是message-filter：



如上所示，只有匹配指定的模式时，当前消息才会被继续处理。但是现在所有没有匹配的消息，不是被丢弃，而是被发送到dead letter队列来继续处理。在这里，只有当消息是XML文档，才会被处理，否则抛出异常。

其他特殊的过滤器是and-filter、or-filter和not-filter，它们可以让你将过滤器组聚合逻辑表达式：



当消息是优先级1，或者消息是优先级2，并且来自加拿大以外的国家时，才会被处理。

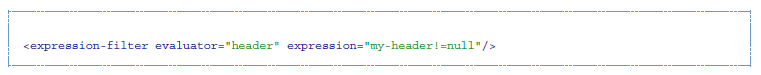
过滤器可以被配置成全局元素，被使用到的地方引用，或者也可以在使用点上进行直接配置。更多Mule过滤器的内容，参考Using Filters。

###### 表达式

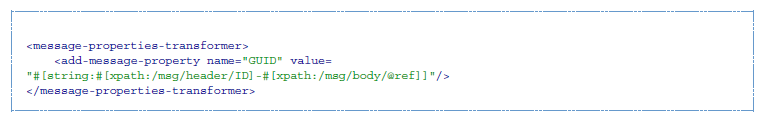
Mule有强大的表达式功能，允许基于系统不同部分的信息来影响消息的处理。因为Mule的许多不同部分可以判定表达式，指定一个表达式需要两个部分：

* 计算器，用来计算表达式。Mule提供了很多的计算器，同时你也可以添加你自己的。
* 表达式，用于被计算。表达式的语法是和计算器相关的。

根据表达式使用的位置不同，有两种方式指定表达式。常用的，基于表达式的元素，例如表达式转换器、表达式过滤器，基于表达式的路由器，例如表达式消息切分器，它们都有特定的表达式属性，计算器和自定义计算器。例如：

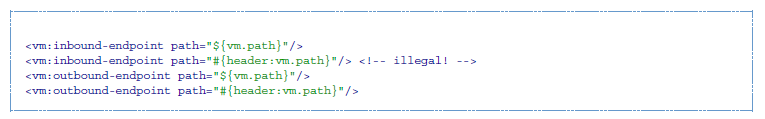


当使用string值替代表达式时，你要使用语法#[evaluator.expression]，例如：



这一表达式调用：首先xpath表达式被使用了两次从当前消息中抽取数据，然后调用string计算器，通过这些数据和连字符来构建出一个string。

表达式和属性占位符可能看起来很像，但是它们是完全不同的。属性占位符会在配置期会替换，用来构建静态的信息。表达式在运行时被替换，因此所有使用到它们的都是动态的。看一下下面的例子：



第一个是好的——它在配置期确定端点的位置。（当然，vm.path属性必须要设置）。

第二个是不合法的。入站端点的地址在配置期必须要设置，在配置被构建前，表达式是没有办法计算的。

第三个和第一个是很像的。

第四个是一个新的配置——动态端点。端点要发消息到的地点是在消息的处理过程中才确定的，可能每次都会不同。当然，没有配置vm.path属性的消息会导致出错。更多Mule表达式的内容，参考Using Expressions。

###### 名字和引用

正如我们看到的，许多Mule对象可以被全局定义。这样的好处是通过在都要好好的 地方引用它们，可以做到在整个应用中复用。对于这种使用，有一个通用的模式：

* 全局对象使用name属性来标识它的名字
* 它可能使用ref属性被引用

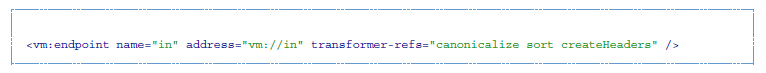
对每种类型的对象，都有一种泛型的元素用来引用它。

* 所有全局转换器都用transformer元素引用
* 所有全局的消息处理器都用processor元素进行引用
* 所有全局端点都用inbound-endpoint和outbound-endpoint进行引用
* 所有全局过滤器都用filter元素进行引用

例如：



另外，也有一些情况下，全局对象的名字是作为属性值来使用的，例如：



##### 流

流是Mule处理的基本单元。一个流从一个入站端点开始，从其中读取消息后，继续经常一系列的消息处理器，然后可选的终止于出站端点，从这里整个被处理后的消息被发送出去。我们已经见过一些消息处理器了：转换器和过滤器。其他类型的包括使用像Java或Groovy语言处理消息的组件，调用云服务的云连接器，和按需要选择消息流的路由器。下面是简单的流，我们将参考它来了解所有部分。



正如其名，这一个流是用来接收和处理订单的。当我们浏览时，记一下流的配置是怎么映射到它描述的逻辑上的：

消息从HTTP端口中被读取出来。

消息被转换成String。

将String用作Key来从数据库中查找订单列表。

订单被转换为XML格式。

如果订单没有准备好要被处理，跳过它。

列表可以选择被记录日志，以用于调试。

列表中的每一个订单被切分成单独的消息。

消息放大器在消息上添加一些信息

调用Authorize.net来授权订单。

订单上的邮箱地址保存备用。

调用Java组件处理订单。

调用另一个叫做processOrder的流处理订单。

processOrder的确认邮件发送订单的地址。

如果处理订单中导致了异常，异常处理策略就会被调用：

所有链上的消息处理器都被调用处理异常。

首先，消息被转换成字符器

最后，字符串被放入到错误队列中，等待人工处理。

这一个流中的任一步都在下面会被详细描述，通过构造进行组织。

###### 端点

###### 转换器

###### 消息放大器

###### 日志记录器

###### 过滤器

###### 路由器

###### 组件

## 在流、模式或服务之间进行选择

### 使用Mule服务

#### 使用消息路由器

##### 消息分解和聚合

###### 消息分解

列表消息分解器

消息分解器可用于将输出消息分解成多个部分和分派这些部分在路由器不同的端点上。这个列表消息分解器接受一个存放消息对象的列表,这些对象将被路由到不同的端点。实际的端点是否用于路由这个列表里的每个对象，是由这个端点本身的一个过滤器配置决定的。如果端点的过滤器接受指定对象,端点将用于路由此对象。

默认情况下，AbstractMessageSplitter在出站消息上设置一个相关ID和相关序列号，以便入站路由器例如Collection Aggregator 或者 Correlation Resequencer能够重新排序或组合那些分解的消息。

路由器配置低于预期消息有效负载时，该消息将被装载到一个java.util.List中，并将路由列表中类型为com.foo.Order, com.foo.Item和com.foo.Customer的对象。路由器将允许组合任意数量的对象。

配置路由器如下：

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

<outbound>

<list-message-splitter-router">

<jms:outbound-endpoint queue="order.queue">

<payload-type-filter expectedType="com.foo.Order"/>

</jms:outbound-endpoint>

<jms:outbound-endpoint queue="item.queue">

<payload-type-filter expectedType="com.foo.Item"/>

</jms:outbound-endpoint>

<jms:outbound-endpoint queue="customer.queue">

<payload-type-filter expectedType="com.foo.Customer"/>

</jms:outbound-endpoint>

<payload-type-filter expectedType="java.util.List"/>

</list-message-splitter-router>

</outbound>

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

注意，在路由器上还有一个过滤器,是确保消息有效负载时接收的类型是java.util.List。即如果列表中有对象不匹配该端点的任何过滤器,一个警告将被写入日志并且继续处理。为了路由那个没有匹配的对象类型到另一个端点,将在没有任何过滤器的列表的末尾加上该端点。

表达式分解路由器

这个路由器类似于列表消息分解路由器,但它将消息基于一个表达式分解。这个表达式必须返回一个或多个有效的消息部件。

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

<outbound>

<expression-splitter-router evaluator="xpath" expression="/mule:mule/mule:model/mule:service"

disableRoundRobin="true" failIfNoMatch="false">

<outbound-endpoint ref="service1">

<expression-filter evaluator="xpath" expression="/mule:service/@name = 'service splitter'"/>

</outbound-endpoint>

<outbound-endpoint ref="service2">

<expression-filter evaluator="xpath" expression="/mule:service/@name = 'round robin

deterministic'"/>

</outbound-endpoint>

</expression-splitter-router>

</outbound>

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

过滤XML消息分解器

* XML消息分解器已被弃用。使用表达式消息分解器和一个xpath表达式来替换。

这个路由器类似于列表消息分解器但只能操作XML文档。支持有效负载类型：

* org.dom4j.Document objects
* byte[]
* java.lang.String

如果没有发现匹配,那么它将被忽略并且记录警告级别的日志。

路由器分解将有效负载送入基于splitExpression属性的节点。实际的端点是否用于列表中的每个对象是由端点本身的一个过滤器配置决定。如果端点的过滤器接受指定对象，端点将用于路由该对象。

返回的每个部分实际上返回的是为一个新的DOM4J文档。

路由器可以选择性地针对一个外部XML模式文档执行验证。要想执行验证，必须到您类路径下的XSD文件中设置属性externalSchemaLocation。设置这个属性重写声明在XML头的任何模式文档。

默认情况下，如果没有任何一个端点过滤器匹配有效负载，路由器将匹配失败。为了防止路由器不会发生这种情况，您可以设置failIfNoMatch属性为false。

配置路由器如下：

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

<outbound>

<mulexml:filter-based-splitter splitExpression="root/nodes" validateSchema="true"

externalSchemaLocation="/com/example/TheSchema.xsd">

<vm:outbound-endpoint path="order">

<payload-type-filter expectedType="com.foo.Order"/>

</vm:outbound-endpoint>

<vm:outbound-endpoint path="item">

<payload-type-filter expectedType="com.foo.Item"/>

</vm:outbound-endpoint>

<vm:outbound-endpoint path="customer">

<payload-type-filter expectedType="com.foo.Customer"/>

</vm:outbound-endpoint>

<payload-type-filter expectedType="org.dom4j.Document"/>

</mulexml:filter-based-splitter>

</outbound>

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

上面的分解器只是将一个消息分解为多个部分，每一部分都是传递到下一个消息处理器或端点。也有消息路由器在路由之前分解消息，例如Round-Robin Message Splitter消息分解器。请查看关于Message Routers的更多信息。

###### 消息聚合

集合聚合器

集合聚合器在转发他们之前对传入的有匹配组id的消息进行分组,。组ID可以来自另一个和消息链接在一起的相关ID或属性。

你可以指定超时属性来确定路由器等待多长时间（以毫秒为单位）来完成接收该组消息。默认情况下，如果预期的消息在指定的超时时间内没有收到，就抛出一个异常，同时消息将不被转发。作为Mule 2.2,您可以设置failOnTimeout属性为false来阻止异常被抛出，同时只是转发到目前为止收到的任何消息。

该聚合器是基于选择性消费者（Selective Consumer），所以你也可以给消息应用过滤器。

这个路由器的配置如下：

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

<inbound>

<collection-aggregator-router timeout="6000" failOnTimeout="false">

<payload-type-filter expectedType="org.foo.some.Object"/>

</collection-aggregator-router>

</inbound>

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

消息组块聚合器

在出站路由器如列表消息分解器（List Message Splitter）将一个消息分解成多个部分之后，消息组块聚合器路由器将这些部分还原成一个单一的消息。该聚合器使用由出站路由器设置的相关ID，以确定哪些部分属于同一个消息。这种聚合器是基于选择性消费者（Selective Consumer）,所以过滤器还可以应用于消息。

这个路由器的配置如下：

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

<inbound>

<message-chunking-aggregator-router>

<expression-message-info-mapping correlationIdExpression="#[header:correlation]"/>

<payload-type-filter expectedType="org.foo.some.Object"/>

</message-chunking-aggregator-router>

</inbound>

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

这个可选的expression-message-info-mapping标签元素允许您使用一个表达式识别具有相关ID的消息。如果这个标签元素没有被指定，MuleMessage.getCorrelationId()方法将被使用。和上面描述的集合中聚合器（Collection Aggregator）一样，消息组块聚合器也支持超时时间及（Mule 2.2）failOnTimeout属性。

定制关联聚合器

这个路由器用于配置自定义消息聚合器。Mule提供了一个抽象实现，同时该抽象实现通过一个模板方法来执行消息聚合。一个普遍的聚合器路由器是结合多个请求的结果，像“问供应商X的最好的价格”一样。

这种聚合器是基于选择性消费者（Selective Consumer）,所以过滤器还可以应用于消息。

和上面描述的集合中聚合器（Collection Aggregator）一样，消息组块聚合器也支持超时时间及（Mule 2.2）failOnTimeout属性。

这个路由器的配置如下：

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

<inbound>

<custom-correlation-aggregator-router class="org.mule.CustomAgregator">

<payload-type-filter expectedType="org.foo.some.Object"/>

</custom-correlation-aggregator-router>

</inbound>

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

有一个AbstractEventAggregator，提供了一个线程安全的自定义聚合器实现，你可以用来编写一个自定义聚合器路由器。例如，贷款经纪人（Loan Broker）的例子在Mule分布中就使用了一个自定义BankQuotesInboundAggregator路由器聚合银行报价消息。

###### 异步响应路由器

[ [概览](#_概览) ] [ [单异步响应](#_单异步响应) ] [ [集合异步响应](#_集合异步响应) ] [ [自定义异步响应](#_自定义异步响应) ]

概览

异步响应路由器常用于请求/响应场景下，通过一个请求触发消息传输，并且该传输消息必须在一个响应消息发送之前被合并完成。经典的例子就是一个请求和任务并行执行。必须在返回响应之前，执行完每个任务并且处理完结果。

异步响应路由器仅仅用于同步调用的服务，因为当以异步方式调度一个消息时，没有响应。Mule提供聚合器路由器，可以结合使用消息分解器或收件人列表路由器在返回响应之前来合成消息。

关于这些路由器的更多信息,请参阅出站路由器（Outbound Routers）。

**例子**

请参考入站配置和异步响应路由器在贷款经纪人例子（Loan Broker）中的配置：

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

<service name="LoanBroker">

<inbound>

<vm:inbound-endpoint path="Loan.Requests"/>

</inbound>

<component class="org.mule.samples.loanbroker.SyncLoanBroker">

<outbound>

<static-recipient-list-router>

<reply-to address="jms://Loan.Quotes"/>

<message-property-filter expression="recipients!=null"/>

</static-recipient-list-router>

</outbound-router>

<async-reply>

<jms:inbound-endpoint queue="Loan.Quotes"/>

<custom-async-reply-router class=

"org.mule.samples.loanbroker.routers.BankQuotesResponseAggregator"/>

</async-reply>

</service>

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

这个配置指定Loan Broker将从vm://Loan.Requests接收请求，并将调度多个请求给不同的银行通过出站路由器。银行方端点被定义在一个存储出站消息的列表中，是出站消息的一个属性，称为“接受者（recipients）”。在出站路由器上重要的设置是<reply-to>端点，它告诉Mule路由所有的响应到jms://Loan.Quotes端点，它是正处于监听状态的异步响应路由器上的端点。当所有的响应都收到了，BankQuotesResponseAggregator选择最低的报价并返回它。Mule然后返回给请求者处理它。

< reply-to>端点应用到下一个服务调用。例如，如果service1分派给service2，service1有一个出站路由器与一个应答端点，service2将发送其调用的结果给应答端点。有关更多信息，请参见ReplyTo。

**响应转换器**

如果你想转换一个响应消息而没有对该响应做任何其他工作，那么您只需在响应路由器上设置那个转换器的属性，而不必配置任何其他路由信息。

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

<response-router transformers="OrderConfirmationToXml XmlToWebPage"/>

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

**超时时间**

超时设置确定Mule在返回结果之前的最大响应时间。默认值是由defaultSynchronousEventTimeout属性的值决定的，该属性已经被配置在Mule实例中。（有关更多信息，请参阅全局设置配置引用（Global Settings Configuration Reference））。您还可以为异步响应指定一个单独的超时值，在async-reply标签元素上，为给定的服务使用可选的超时属性。

可选的failOnTimeout属性决定在所有预期消息收到之前，如果路由器超时，是否抛出异常。如果设置为false(默认)，当前的消息会被返回同时继续处理。

例如：

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

<async-reply failOnTimeout="false" timeout="2000">

<inbound-endpoint ref="replyEndpoint"/>

<collection-async-reply-router/>

</async-reply>

这个页面的其余部分描述了你可以使用的异步响应路由器。关于配置异步响应路由器的详细信息，请参考异步响应路由器配置说明（Asynchronous Reply Router Configuration Reference）。

单异步响应

单一异步响应路由器配置单个应答路由器。这个路由器返回在一个响应端点上收到的第一个消息并且丢弃剩余部分。

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

<single-async-reply-router/>

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

集合异步响应

集合异步响应路由器配置一个集合应答路由器。这个路由器返回一个MuleMessageCollection类型消息，其中包含所有收到的当前相关消息。

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

<collection-async-reply-router/

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

自定义异步响应

自定义异步回复路由器允许您配置一个自定义异步响应路由器。设置类属性定制路由器类来配置自定义路由器。-----------------------------------------------------------------------------------------------------------------------------------------------------------

<custom-async-reply-router class="org.mule.CustomAsyncReplyRouter"/>

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

###### 全方位策略

[[转发](#_Forwarding)][[定制转发](#_定制转发)][[日志](#_日志)][[定制](#_定制)]

对于当前的消息如果没有发现路由路径，您可以配置调用一个全方位策略，入站或出站端点可以关联到这个全方位策略，以便捕获到任何孤立的消息并且路由到一个共同的位置。关于全方位策略在元素的详细信息配置方面，请查看全方位策略配置参考（Catch-all Strategy Configuration Reference）。

例如：

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

<service name="dataService">

<inbound>

<inbound-endpoint address="vm://in2" connector-ref="vmQueue">

<string-to-byte-array-transformer/>

</inbound-endpoint>

<selective-consumer-router>

<payload-type-filter expectedType="java.lang.Integer"/>

</selective-consumer-router>

<custom-forwarding-catch-all-strategy class=

"org.mule.test.usecases.routing.InboundTransformingForwardingCatchAllStrategy">

<outbound-endpoint address="vm://catchall" connector-ref="vmQueue">

<string-to-byte-array-transformer/>

</outbound-endpoint>

</custom-forwarding-catch-all-strategy>

</inbound>

...

<outbound>

<filtering-router transformer-refs="TestCompressionTransformer">

<outbound-endpoint address="test://appleQ2" name="TestApple-Out" />

<payload-type-filter expectedType="java.lang.String" />

</filtering-router>

<custom-catch-all-strategy class="org.mule.tck.testmodels.mule.TestCatchAllStrategy" />

</outbound>

...

</service>

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

以下是说明你可以使用的不同的全方位策略。

转发

这个全方位策略是用于将没有出站路由器匹配的消息转发到该配置端点。

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

<forwarding-catch-all-strategy>

<jms:outbound-endpoint queue="error.queue"/>

</forwarding-catch-all-strategy>

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

定制转发

这个全方位策略是同样的作为默认转发全方位策略,但它允许您指定一个自定义的实现，同时通过配置类属性来得到该实现。您还可以配置附加的可选属性。

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

<custom-forwarding-catch-all-strategy class="org.my.CustomForwardingCatchAllStrategy">

<jms:outbound-endpoint queue="error.queue"/>

<spring:property key="myProperty" value="myValue"/>

</forwarding-catch-all-strategy>

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

日志

这个全方位策略并没有对消息进行处理，而只是简单的记录了一个警告，表示消息没有被派发，原因是没有路由路径定义。

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

<logging-catch-all-strategy/>

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

定制

这个全方位策略允许您使用一个自定义的类来执行任何您需要的行为。为了实现一个自定义配置的全方位策略，转发消息到一个端点，您应该使用自定义转发代替全方位策略。

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

<custom-catch-all-strategy/>

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

###### 使用Mule配置模式

简介

配置Mule涉及XML,尽管使用一个像样的XML编辑器可以帮助很多(多亏了上下文帮助它提供从Mule的模式),仍有足够的尖括号来保证一个咖啡作为项目变得更加复杂。随着服务数量的增加在骡子项目,那么数量的噪音在它的配置文件,使它越来越难理解和维护。在Mule 3,我们决定解决这一问题,引入基于模式的配置。

Configuring Mule involves XML, and though using a decent XML editor can help a lot (thanks to the contextual help it provides from Mule's schemas), there are still a enough angle brackets to warrant a coffee break as projects get more complicated. As the number of services in a Mule project increases, so does the amount of noise in its configuration files, making it harder and harder to understand and maintain them. In Mule 3, we've decided to tackle this problem with the introduction of pattern-based configuration.

XML命名空间

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

xmlns:file=<http://www.mulesoft.org/schema/mule/pattern>

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

XML 模式定位

http://www.mulesoft.org/schema/mule/pattern

http://www.mulesoft.org/schema/mule/pattern/3.1/mule-pattern.xsd

何时使用某个模式

**一个配置模式提供了一个非常特定的集成特性。**

下面的表将帮助你捡起适合你需要的配置模式。

|  |  |
| --- | --- |
| 模式名称 | 用法 |
| 简单服务 | 公开了jax - ws注释组件作为SOAP web服务。公开了jax - rs带注解的bean作为RESTful组件。　　还可以处理JAXB,XML和原始内容与简单的POJO组件。  Exposes JAX-WS annotated components as SOAP web services. Exposes JAX-RS annotated beans as RESTful components.  Can also handle JAXB, XML and raw content with simple POJO components. |
| Web服务代理 | 代理远程web服务。可以执行转换的SOAP信封。可以修改或重定向到当地的远程wsdl。  Proxies remote web services. Can perform transformations on the SOAP envelope. Can rewrite or redirect remote WSDLs to local ones. |
| 桥接 | 建立一个入站端点之间直接管道和一个出站端点。支持请求-响应和单向桥接。可以执行转换。支持事务桥接的入站到出站。  Establishes a direct conduit between an inbound endpoint and an outbound endpoint. Supports request-response and one-way bridging. Can perform transformations. Supports transactional bridging of inbound to outbound. |
| 验证器 | 对一个已定义的入站消息验证验收过滤器。返回一个ACK或NACK响应同步和异步分派有效的消息。  Validates inbound messages against a defined acceptance filter. Returns an ACK or NACK response synchronously and dispatches valid messages asynchronously. |

应该没有现有模式满足你的需要,转向使用流服务编排为高级配置机制。

Should none of the existing patterns satisfy your need, turn to Using Flows for Service Orchestration for advanced configuration mechanisms.

相关主题

基于模式的配置

提供反馈或建议的新模式

###### 基于模式的配置

[[模式的作用](#_展示承诺模式Demonstrating_a_Commitment)][[共享属性](#_共享属性Sharing_Properties)][[未来方向](#_未来方向Future_Directions)]

涉及XML的配置Mule，尽管使用一个像样的XML编辑器可以帮助很多(多亏了提供Mule模式的上下文帮助)，尽管项目变得更加复杂，但仍有足够的尖括号来保证可以完全应对这些配置。在Mule项目中随着服务数量的增加，它的配置文件繁杂程度的增加，使它越来越难理解和维护。在Mule 3，我们决定引入基于模式的配置来解决这一问题。

模式的作用

基于模式的设计是一门软件科学，它鼓励识别周期性的实践来创建消耗品工件，鼓励后续和高效重用。我们相信这些原则可以应用于配置Mule。这就是为什么我们开始努力收集和实现配置模式。具体地说,这些模式将会把自己看成是对Mule配置优良语法。他们将是新的XML元素，将使您用最少数量的XML配置，执行常见的配置任务。

共享属性

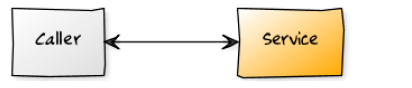
最重要的是，配置模式也将提供某种形式的继承，允许在他们共享属性。最后，他们将在很大程度上依赖于重新架构的Mule的核心，它为创造这样的专门的工件提供了辅助方式。任何的传输方式或组件将有可能贡献配置模式，那么这些模式将是高度专业化的。当然，当模式并不完全满足你的需要时，我们总是可以使用更一般的Mule元素配置的情况。

未来方向

这些文档页面将被更新到一个正在进行创建中的平台，作为一个新配置模式，出现在Mule社区。

###### 简单服务模式

简单服务模式的目标是和它的名字一样简单：提供一种简单的方式来公开请求-响应服务。以其他的术语来说，它是用来以同步的方式来暴露一些业务逻辑来应对你选择的传输方式。



核心特征

简单服务允许在Mule中以一种简洁高效的方式部署SOA风格的服务。闲话少说，让我们深入到示例,来阐明这些新配置元素的扩展能力。

**暴露一个POJO成为一个简单服务**

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

<pattern:simple-service name="maths-service"

address="vm://maths.in"

component-class="org.mule.tck.services.SimpleMathsComponent" />

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

这就配置完成了！SimpleMathsComponent 的一个实例现在可以通过一个已经自动配置为同步(即请求-响应)的VM端点。照例，Mule将使用一个分解策略去找到最好的方法来接收已经被发送给服务的负载。

当然,全局端点和组件定义也是可能的：

**使用说明**

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

<pattern:simple-service name="maths-service"

endpoint-ref="maths-service-endpoint"

component-ref="math-component" />

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

同样,转换器服务也可配置：

**转换器**

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

<pattern:simple-service name="byte-array-massager"

address="vm://bam.in"

transformer-refs="byte-array-to-string append-bar"

responseTransformer-refs="append-baz string-to-byte-array"

component-class="org.mule.component.simple.EchoComponent" />

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

简单服务也可以在任何受支持的模式中作为组件来配置元素。下面显示了一个非常方便的方式来配置在功能测试阶段的服务器存根：

**功能测试服务Functional Test Service**

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

<pattern:simple-service name="functional-test-component"

address="vm://ftc.in">

<test:component />

</pattern:simple-service>

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

同样,入站端点和异常策略都可以配置为子元素：

**子元素**

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

<pattern:simple-service name="maths-service"

component-class="org.mule.tck.services.SimpleMathsComponent">

<vm:inbound-endpoint path="maths.in"

exchange-pattern="request-response" />

<custom-exception-strategy class="com.acme.AcmeExceptionStrategy" />

</simple-service>

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

简单的服务,和所有的配置模式元素一样, 同样以Spring bean方式作为标准来支持继承。继承可以用于共享常见配置元素之间的设置。参加下面说明:

**继承**

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

<pattern:simple-service name="global-exception-strategy"

abstract="true">

<custom-exception-strategy class="com.acme.AcmeExceptionStrategy" />

</pattern:simple-service>

<pattern:simple-service name="inherited-exception-strategy"

parent="global-exception-strategy"

address="vm://maths.in"

component-class="org.mule.tck.services.SimpleMathsComponent" />

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

注意:继承一个嵌套组件的元素是不受支持的,因此不能被定义在一个抽象的简单服务上。

SOAP 服务

简单服务允许您暴露JAX-WS注释组件作为SOAP服务。所有您需要做的就是添加一个类型属性和你的能被SOAP调用的组件。这里有一个例子：

**JAX-WS 服务**

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

<pattern:simple-service name="weather-forecaster-ws"

address="http://localhost:6099/weather-forecast"

component-class="org.mule.test.integration.tck.WeatherForecaster"

type="jax-ws" />

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

注意,如果使用的传输方式是HTTP,简单服务将会很乐意为路径结尾是“? WSDL”请求的WSDL文档服务。

**RESTful 服务**

简单服务也能暴露JAX-RS注释组件作为RESTful服务。此外,使用相关的类型是唯一的工作量来开启RESTful资源服务,如下所示:

**JAX-RS 服务**

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

<pattern:simple-service name="weather-report-rsc"

address="http://localhost:6099/rest"

component-class="org.mule.test.integration.tck.WeatherReportResource"

type="jax-rs" />

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

如果组件在上面的配置是注释@ path(" / weather-report”),那么它的资源在这个URI下将是可获得的:

<http://localhost:6099/rest/weather-report>.

JAX-RS服务使用HTTP和Servlet方式传输。

**JAXB 支持**

简单服务可以处理JAXB序列化的有效载荷。不需要在XML上配置什么特别的东西：

**JAXB序列化服务**

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

<pattern:simple-service name="weather-jaxb-consumer"

address="vm://weather-consumer.in"

component-class="org.mule.test.integration.tck.WeatherReportConsumer" />

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

但是,对于这个工作,配置@Payload是必需的,它是Mule用于组件的特定注释：

**带注释的服务组件为JAXB提供有效载荷**

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

package org.mule.test.integration.tck;

import org.mule.api.annotations.param.Payload;

public class WeatherReportConsumer

{

public String consume(@Payload WeatherReportType weatherReport)

{

return weatherReport.report;

}

}

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

XPath 支持

最后,简单服务也可以处理XML有效负载，直接提取XPath表达式的值。与JAXB一样, 不需要在XML上配置什么特别的东西:

**XPath有效负载服务**

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

<pattern:simple-service name="weather-xpath-consumer"

address="vm://weather-xpath-consumer.in"

component-class="org.mule.test.integration.tck.WeatherReportXpathConsumer" />

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

此外,这个Mule的注释@xpath,在这种情况下,需要像下面这样工作:

**基于注释的XPath有效负载服务组件**

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

package org.mule.test.integration.tck;

import org.mule.api.annotations.expression.XPath;

public class WeatherReportXpathConsumer

{

public String consume(@XPath(value = "/weatherReport/report") String report)

{

return report;

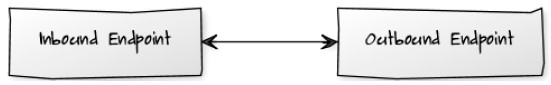
}

}

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

###### 桥接模式

将相互关联的系统连接在一起是最基本的集成任务。Mule ESB提供了必要的构建块来实现这样的连接。这些构建块可使消息来源和目的地之间建立桥梁，被称为消息桥接模式,这个构建块作为新引入的桥接配置元素在Mule 3中是有效的。



核心特征

一个消息传递桥扮演着一个入站端点和出站端点之间的直接管道的角色。这是一个中立组件,因为它不在流穿过消息上应用任何业务逻辑。正如我们下面将要看到的一样：一座桥可以执行协议适配和数据转换。默认情况下,桥接配置元素在其端点之间建立了一个请求-响应管道:任何来自出站端点的响应将被路由回入站端点。下面的代码展示了这样一个同步桥:

**同步-aka请求-响应-桥**

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

<pattern:bridge name="request-response-bridge"

inboundAddress="vm://synchronous-bridge.in"

outboundAddress="vm://maths-service.in" />

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

使用同步桥是有道理的,这是因为目标的出站端点是一个请求-响应的数学计算服务,他们的回应是预期的呼叫者的桥。在其他场景中,没有响应需要路由反回：该桥成为一个单向管道。考虑下面的例子,消息被发送到发出后就丢弃的日志系统：

**单向桥**

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

<pattern:bridge name="one-way-bridge"

exchange-pattern="one-way"

inboundAddress="vm://asynchronous-bridge.in"

outboundAddress="vm://log-service.in" />

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

该桥的元素,就像其他基于模式的配置元素一样,支持全局端点定义:

**端点说明**

<pattern:bridge name="endpoint-ref-bridge"

inboundEndpoint-ref="endpoint-ref-bridge-channel"

outboundEndpoint-ref="maths-service-channel" />

它还支持子元素来启用高级配置的端点和定义的异常策略(以防有特殊情况发生时，消息可以被处理的桥):

**子元素**

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

<pattern:bridge name="child-endpoint-bridge">

<vm:inbound-endpoint path="child-endpoint-bridge.in"

exchange-pattern="request-response" />

<vm:outbound-endpoint path="maths-service.in"

exchange-pattern="request-response" />

<custom-exception-strategy class="com.acme.AcmeExceptionStrategy" />

</bridge>

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

最后,继承也支持,使其可以在几个桥接模式之间共享属性:

**继承**

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

<pattern:bridge name="abstract-parent-bridge"

abstract="true"

outboundAddress="vm://maths-service.in" />

<pattern:bridge name="concrete-child-bridge"

parent="abstract-parent-bridge"

inboundAddress="vm://concrete-child-bridge.in" />

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

适配和转换

到目前为止,我们的例子都展示的是同种桥接模式之间的传输:相同的传输方式(VM)被用于入站和出站端点。因为它依赖于Mule的适配和消息抽象协议,有一种桥可以很好地处理使用异构协议的场景。考虑下面的例子:

**异构传输桥**

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

<pattern:bridge name="dlqDumper"

exchange-pattern="one-way"

inboundAddress="jms://myDlq"

outboundAddress="file://./test?outputPattern=#[header:INBOUND:JMSMessageID].dl" />

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

正如您看到的,我们已经配置了一种桥来处理这些在死信队列(DLQ)上接收的消息。这种桥清除了所有的死信消息发送到指定的JMS队列,并将其写入到文件系统。当然,这是一个单向桥:我们没有兴趣发送任何回应给DLQ !处理异构协议之外,这种桥可以处理不同的数据格式。事实上,正如上面提到的, 这种桥可以执行转换, 当集成完全不同的系统，这是时常需要的。下面显示了一个示例,数据根据一个标准的规范进行转换:

**转换桥**

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

<pattern:bridge name="transforming-bridge"

inboundAddress="vm://transforming-bridge.in"

transformer-refs="to-canonical-form"

responseTransformer-refs="from-canonical-form"

outboundAddress="vm://target-service.in" />

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

事务支持

有时,重要信息会中转到一个桥:在这种情况下,一个消息必须可靠地从入站端点被接收并且发送到出站端点。如果这个发送失败,消息应该被退回以便发送可以稍后再尝试。为实现这一目标声明了事务型的桥。在这种情况下,它将清除其入站消息和调度其出站事务内的消息。看看下面的例子,了解其工作原理:

**事务型桥**

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

<pattern:bridge name="queue-to-topic"

transacted="true"

inboundAddress="jms://myQueue"

outboundAddress="jms://topic:myTopic" />

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

在这个例子中,我们给一个主题创建一个JMS队列。我们使用事务属性的原因是我们想确保所有入站的消息能被成功地接收和重新发布。显然,对于这个工作,用于传输的桥应该支持事务。如果入站和出站端点使用异构协议,这种桥将会寻找一个XA事务管理器(它必须是已经配置)。

###### 验证器模式

当处理消息时,一个特定的格式总是受人推崇的,这样可以检索所需的数据。它是可能的,并且通常传入的消息需要非常自由的数据表示:只要你能找到所需的信息,其余的并不重要。但有时候,一个传入消息进行严格的预先验证是必要的。这是目的地配置了验证器模式的那种场景。



核心特征

公开一个SOAP API的服务受益于其固有主机web服务框架的验证,它根据严格的WSDL规范定义，执行传入消息的合法性验证。对于所有其他类型的服务,Mule的用于过滤和路由的基础组件中提供了所有必要的构建块,并将其放置于合适的地方来进行严格的验证。

验证器模式为你提供了一个框架来执行这种预先验证。当正在调度的有效请求被异步执行时，它以同步的方式执行验证。这提供了一种解耦的验证和处理阶段,这是一种常见的模式当客户信息的速度生产他们实际上是加工。验证器模式利用了Mule的广泛的表达式的框架来构建和拒绝确认消息。它使用Mule的过滤器来表达条件的消息是有效的。让我们看看一个验证器,只接受整数:

**整数验证器**

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

<pattern:validator name="integer-validator"

inboundAddress="vm://validator.in"

ackExpression="#[string:GOOD:#[message:payload]@#[context:serviceName]]"

nackExpression="#[string:BAD:#[message:payload]@#[context:serviceName]]"

outboundAddress="vm://test-service.in">

<payload-type-filter expectedType="java.lang.Integer"/>

</pattern:validator>

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

假设我们发送123到这个验证器:它将接受该消息并回复“GOOD:123@integer-validator”告知收到。相反的,如果我们把“abc”它将拒绝消息并回复“BAD:123@integer-validator”。它可以被用于全局端点和过滤器并且引用它们,显示以下:

**使用说明**

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

<pattern:validator name="validator-with-refs"

inboundEndpoint-ref="validator-with-refs-channel"

ackExpression="#[string:GOOD:#[message:payload]@#[context:serviceName]]"

nackExpression="#[string:BAD:#[message:payload]@#[context:serviceName]]"

validationFilter-ref="int-payload-filter"

outboundEndpoint-ref="test-service-channel" />

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

它也可以定义入站和出站端点的同时，添加一个自定义异常策略作为子元素 ,当您使用的端点需要复杂的配置时，这是很方便的:

**子元素**

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

<pattern:validator name="validator-with-child-elements">

<vm:inbound-endpoint path="validator-with-child-endpoints.in"

exchange-pattern="request-response" />

<payload-type-filter expectedType="java.lang.Integer"/>

<outbound-endpoint ref="test-service-channel" />

<custom-exception-strategy

class="com.acme.ValidationExceptionStrategy" />

</pattern:validator>

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

最后,像所有的配置模式一样,验证器元素支持继承。如果你想共享ack/nack表达式或验证跨多个验证器的规则,这是非常有用的:

**继承**

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

<pattern:validator name="abstract-parent-validator"

abstract="true"

ackExpression="#[string:GOOD:#[message:payload]@#[context:serviceName]]"

nackExpression="#[string:BAD:#[message:payload]@#[context:serviceName]]" />

<pattern:validator name="concrete-validator"

parent="abstract-parent-validator"

inboundAddress="vm://concrete-validator.in"

outboundAddress="vm://test-service.in">

<payload-type-filter expectedType="java.lang.Integer"/>

</pattern:validator>

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

出站错误

在默认情况下,如果在出站端点派遣一个有效消息期间发生错误,调用者将不知道。在这种情况下,可以使用Mule的异常处理机制来记录错误,存储有效的发送消息,甚至可以尝试重新提交。据说,这是可能的,在某些场景中调用者必须被告知其发送的有效消息已经失败了。对于这个问题,第三个选项的表达式可以被使用:

**继承**

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

<pattern:validator name="dispatch-error"

inboundAddress="vm://dispatch-error.in"

ackExpression="#[string:GOOD:#[message:payload]@#[context:serviceName]]"

nackExpression="#[string:BAD:#[message:payload]@#[context:serviceName]]"

errorExpression="#[string:ERROR:#[message:payload]@#[context:serviceName]]"

outboundAddress="http://acme.com/services/fragile"

validationFilter-ref="int-payload-filter" />

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

在这个配置中, errorExpression将用于出站调度失败时创建响应给调用者。

注意,当这个表达式被用时,出站端点将使用请求-响应交换模式。