

## 第 7 章 基于 JBI 的 ServiceMix 服务总线

ServiceMix 是一个完全实现了 JBI 标准的开源项目，通过理解 ServiceMix，也就可以更容易地理解 JBI 的概念。ServiceMix 是一个基于 SOA 架构和事件驱动的 ESB。

本章主要介绍 ServiceMix 所拥有的和 SOA 集成相关的以下功能。

- 如何在 ServiceMix 中创建 Web Service 服务。
- 如何创建代理，主要包括如何创建监听外部服务请求的绑定组件，以及调用外部服务的绑定组件。
- 如何实现和外部服务集成。
- 如何实现信息格式转换。
- 如何根据消息内容实现动态路由。
- 如何通过 BPEL 实现服务集成。

### 7.1 ServiceMix 所支持的主要功能

ServiceMix 是一个轻量级的 JBI 容器，已经集成了 Spring 支持，它可能成为一个服务总线的提供者，也可以给另外一个服务总线提供服务，也可以将 ServiceMix 用在另外一个 J2EE 的应用服务器里面。ServiceMix 集成了 BPEL、XPath 转换、Drools 规则引擎、对 RSS 的支持与 JCA 等。

#### (1) ServiceMix 所支持的服务引擎

- Quartz: 定时任务服务;
- Cache: 缓存服务;
- JCA: J2C Connector Architecture, 用于解决和企业信息系统 EIS 的连接;
- Groovy: 一种运行于 Java VM 的脚本语言, 用来创建服务;
- Scripting: 适用于满足 JSR223 的脚本创建服务;
- XSLT: 通过 XSLT 对 XML 格式的消息进行转换;
- XPath Routing: 实现基于 Xpath 的路由;
- Validation: 校验消息格式;
- PXE: BPEL 集成服务;
- servicemix-bpe: BPEL 集成服务;
- servicemix-sca: 支持 SCA 集成;

- **servicemixlwcontainer**: ServiceMix 提供的轻量级容器, 主要目的在于允许一些轻量级的组件 (POJO) 能够在运行时发布, 而不是通过静态的 `servicemix.xml` 来发布。

#### (2) ServiceMix 所支持的绑定组件

- **Email**: 支持 Java mail;
- **File**: 支持对文件的收发;
- **FTP**: 支持 FTP 协议;
- **HTTP**: 支持 HTTP 协议, 包括 Post/Get 请求;
- **Jabber**: Linux 即时通信器;
- **XSQL**: 一种 XML 和 SQL 的混合体, 支持数据库的查询, 然后将查询结果放在一个 XML 文件里面;
- **VFS**: 将不同类型的文件系统的访问封装成统一的接口;
- **WSIF**: Apache 的 Web 服务调用框架 (WSIF), 它使用统一的编程模型来调用 Web 服务;
- **JAX WS**: Java API for XML Web Services, 它是开发 SOAP 的基本技术, 计划用来代替 JAX-RPC;
- **JMS**: 基于 ActiveMQ 的 Java 消息服务; 因为 ServiceMix 本质上是一个消息传递系统, 所以该功能是非常重要的;
- **RSS**: 访问并处理 RSS 种子;
- **SAAJ**: 支持 Apache Axis;
- **servicemix-http**: ServiceMix 开发的 HTTP 绑定组件;
- **servicemix-jms**: JMS 绑定组件;
- **servicemix-jsr181**: 基于 JSR181 开发 Web Service。

## 7.2 ServiceMix架构体系

ServiceMix 的架构主要由三大部分组成。

- **服务引擎组件**: 负责格式转换、路由及其相关的服务 (如 BPEL 服务集成、XSLT 格式转换、规则引擎、脚本调用等);
- **绑定组件**: 主要用来接收外部的服务请求, 以及向外部服务提供者发出请求;
- **ServiceMix 的 JBI 容器**: 实现了 JBI 的功能的服务总线。

图 7-1 显示了 ServiceMix 的架构体系。

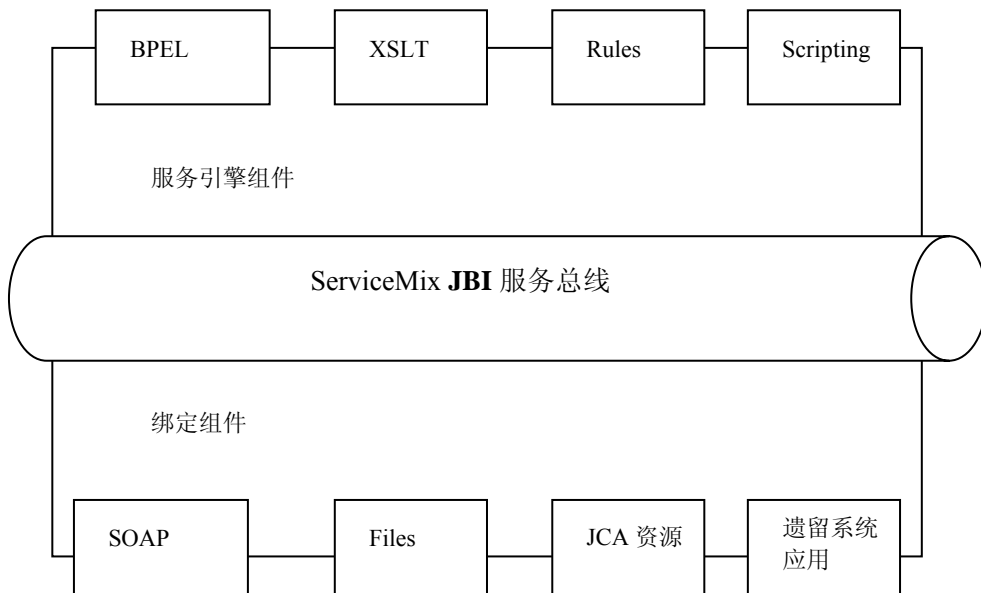


图 7-1 ServiceMix 的架构体系

## 7.3 安装ServiceMix3.1

ServiceMix 需要 JDK1.5 的支持，JDK1.5 已经在第 2 章做了介绍。本章的 ServiceMix 实例需要 Ant 脚本的编译支持，Ant 可到下面地址下载然后安装即可：

<http://ant.apache.org/bindownload.cgi>

ServiceMix 的主页地址：

<http://incubator.apache.org/servicemix/home.html>

ServiceMix3.1 的下载地址：

<http://incubator.apache.org/servicemix/servicemix-31.html>

本章的实例都是采用 `apache-servicemix-3.1-incubating.zip`，是目前相对来说一个比较稳定的版本，如果采用其他的版本，由于其采用的 Java 类库可能有所变化，读者对实际实例可能需要做相应的调整才能运行。

直接将所下载的 `apache-servicemix-3.1-incubating.zip` 解压缩就可以了。

本章的实例因为需要和第 2 章的 Web Service 实例进行集成，所以最好是在完成第 2 章的实例后，再来做本章的实例比较好。

图 7-2 显示了 ServiceMix3.1 安装的目录结构。

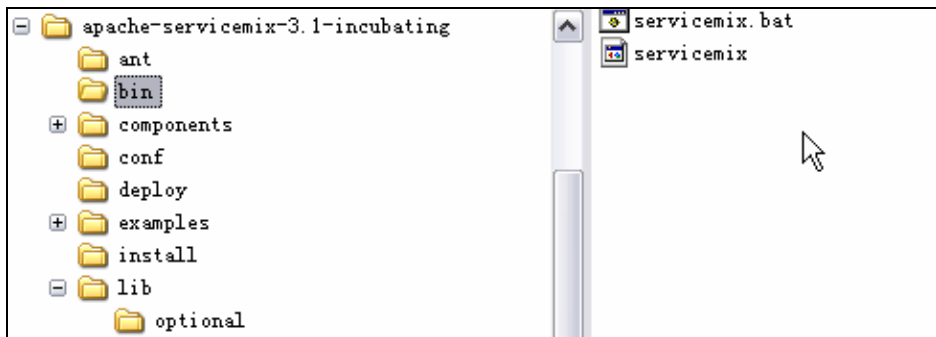


图 7-2 ServiceMix3.1 安装的目录结构

其中 bin 目录下面的 servicemix.bat 文件将负责启动 ServiceMix。

\\lib\\optional 目录下面可以另外放置相应的 jar 文件，它本身带有：

- commons-collections-3.1.jar
- commons-pool-1.2.jar

#### Tips

运行本章的配套光盘实例，需要将光盘中提供的 jar 文件复制到“\\lib\\optional”目录下面。

## 7.4 创建和发布Web服务实例

本小节的主要目的在于如何通过 ServiceMix 创建一个 Web Service，以及通过客户端如何调用这个 Web Service。

其基本过程是首先创建一个 Web Service 的实现类，然后通过 Servicemix.xml 将其配置成 WebService，ServiceMix 可以直接运行 Servicemix.xml，发布服务组件，包括 BC 绑定组件（接收 HTTP 的连接），SE 服务组件（创建 Web Service）。

创建 Web Service 主要用到了 ServiceMix 提供的 servicemix-jsr181 组件，它是一个 JBI 的 SE 服务引擎，它可以直接将 POJO 发布成 JBI 服务总线上的服务，它里面采用了 Xfire 实现服务和 XML 的序列化工作。

JSR181 是 J2SE5.0 所提供的新的注释功能，可以以声明的方式来定义 Web Service。JSR181 的主要目的在于不需要为每个不同的 Web Service 运行环境（如 Axis、Xfire、Axis2）编写不同的配置文件（第 2 章讲述了如何创建相应的配置文件）。只要运行环境支持 JSR181，就可以采用统一的配置文件。

例如对于 hello 的 Axis 的发布时的配置文件 server-config.wsdd 如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
<handler type="java:org.apache.axis.handlers.http.URLMapper" name="URLMapper"/>
  <service name="hello" provider="java:RPC">
    <parameter name="className" value="server.HelloService"/>
    <parameter name="allowedMethods" value="getHello"/>
  </service>
</deployment>
```

```
</service>
<transport name="http">
  <requestFlow>
    <handler type="URLMapper"/>
  </requestFlow>
</transport>
</deployment>
```

例如对于 hello 的 Xfire 的发布时的配置文件 services.xml 如下：

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">
  <service>
    <name>hello</name>
    <namespace>hello</namespace>
    <serviceClass>server.Hello</serviceClass>
    <implementationClass>server.HelloService</implementationClass>
  </service>
</beans>
```

可以看到 Axis 和 Xfire 的 Web Service 的配置文件的内容并不一致。如果采用 JSR181，Xfire 的配置文件 services.xml 则只需要 serviceClass 和 serviceFactory 即可，程序如下：

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">
  <service>
    <serviceClass> server.HelloService </serviceClass>
    <serviceFactory>jsr181</serviceFactory>
  </service>
</beans>
```

如果采用 JSR181，需要加入注释行的语句实例如下面黑体字所显示：

```
package server;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
@WebService(name="HelloService",serviceName=" HelloService ",targetNamespace="hello")
public class HelloService implements Hello
{
    @WebMethod
    @WebResult
    public String getHello(@WebParam String name)
    {
        return "Hello "+name+ "! This is AXIS Web Service Response";
    }
}
```

#### 7.4.1 业务流程

本节的业务流程如下：

- 客户端通过 HTTP 的方式送一个 SOAP 消息给 ServiceMix 的运行环境。
- ServiceMix 的 HTTP BC 组件收到一个客户端的 HTTP/SOAP 的请求消息后，转给 ServiceMix 的 Web Service JSR181 的组件。
- Web Service JSR181 的组件将执行结果返回给 HTTP BC 组件。
- HTTP BC 组件将结果返回给服务请求者。

图 7-3 显示了 ServiceMix 创建 Web 服务的流程图。

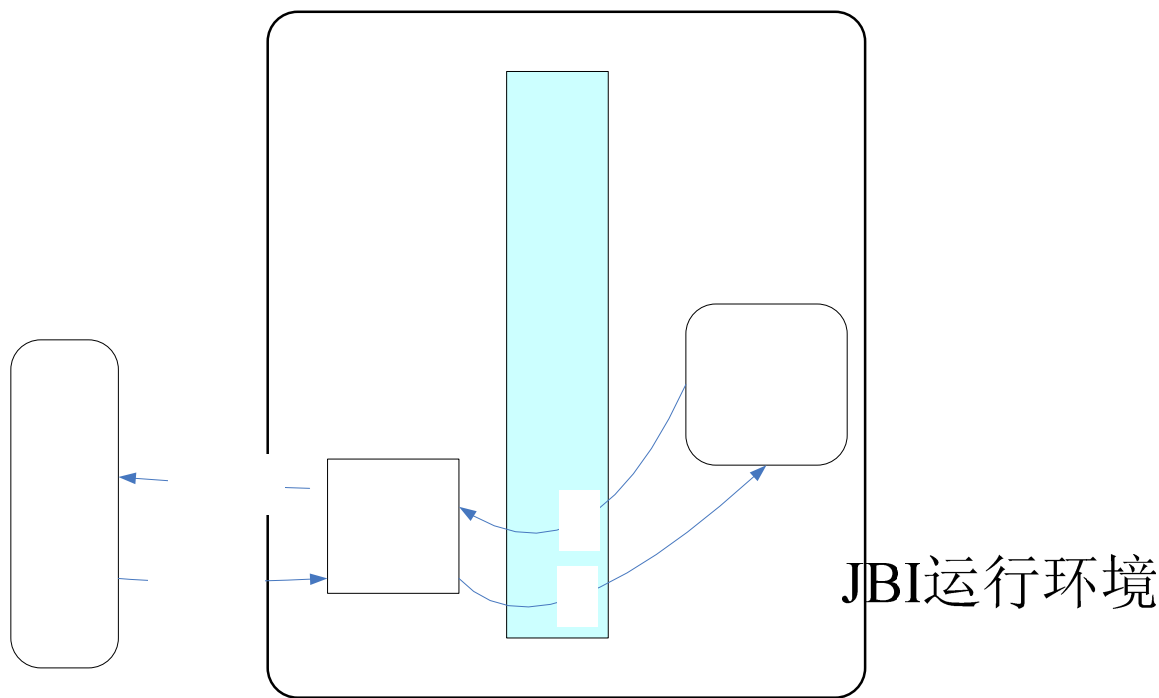


图 7-3 ServiceMix 创建 Web 服务实例流程图

在 ServiceMix 上创建一个 Web Service 包括下面的过程。

- 创建 Servicemix.xml 配置文件，它用来配置 ServiceMix 运行时的组件；
- 创建 WSDL 文件；
- 创建服务器端的 Java 接口类和 Java 实现类；
- 发布 ServiceMix 服务；
- 创建客户端调用程序。

下面将分别予以介绍。

#### 7.4.2 Servicemix.xml

ServiceMix.xml 是整个 ServiceMix 发布的核心，通过它，可以将内外服务完全整合起来。首先要定义和 JSR181 相关的命名空间 `xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0"`，ServiceMix 运行环境在看到了命名空间之后，会调用相应的 JSR181 的组件，这个组件由 ServiceMix 所提供的 `servicemix-jsr181-3.1-incubating-installer.zip` 来实现。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
  xmlns:spring="http://xbean.org/schemas/spring/1.0"
  xmlns:sm="http://servicemix.apache.org/config/1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0"
  xmlns:http="http://servicemix.apache.org/http/1.0"
  xmlns:hello="hello" <!-- 必须与 hello.wSDL 的目标空间保持一致 -->
  xsi:schemaLocation="http://xbean.org/schemas/spring/1.0 ../conf/spring-beans.xsd
    http://servicemix.org/config/1.0 ../conf/servicemix.xsd"
  xmlns:foo="http://servicemix.org/demo">
  <bean id="jndi" class="org.apache.xbean.spring.jndi.SpringInitialContextFactory"
    factory-method="makeInitialContext" singleton="true" />
  <sm:container id="jbi" useMBeanServer="true" createMBeanServer="true">
    <sm:activationSpec>
```

下面是配置“消费端口”监听外部服务请求的标准写法。

- Service: 服务名称，标准写法为“命名空间: 服务名称”，如“hello:hello”;
- Endpoint: 端口名称，必须与 WSDL 所定义的 Service 的 Port 一致;
- role: 必须是“consumer”，因为这里是配置消费端口;
- locationURI: 服务监听的 URL 地址，是外部服务请求者发送服务请求的地址;
- defaultMep: 默认的 Mep 的 URI，MEP (Message Exchange Pattern) 指消息交换模型，详细参考见 3.4 节。一般采用“http://www.w3.org/2004/08/wsdl/in-out”，表示它是一个请求和响应的模型，表示得到请求消息后，需要返回响应消息。
- soap: 是否为 SOAP，如果被设置为 true，那么组件将会解析 SOAP 消息，然后在送到 NMR;
- soapAction: 当调用 Web Service 时所送的 SOAPAction 的 header 消息部分。

```
<sm:activationSpec>
  <sm:component>
    <http:component>
      <http:endpoints>
        <http:endpoint service="hello:hello"
          endpoint="helloHttpPort" 必须与 WSDL 所定义的 service 的 port 一致
          role="consumer"
          locationURI="http://localhost:8192/hello/" 通过它可以验证
          defaultMep="http://www.w3.org/2004/08/wsdl/in-out"
          soap="true"
          soapAction="getHello" />
      </http:endpoints>
    </http:component>
  </sm:component>
</sm:activationSpec>
```

下面是通过 JSR181 创建 Web Service 服务引擎组件 (SE) 的标准写法。

- .pojoClass: 输入 Web Service 服务的 Java 实现类，包括 package 的名称;



- **wSDLResource**: 应用的 WSDL 文件, 包括路径名;
- **Style**: 使用的 SOAP 类型, 包括 document (文档类型)、RPC (远程调用类型)、wrapped 类型 (wrapped document 大量使用于 .NET 平台, 所以如果需要和 .NET 平台交互, 可以采用这种类型);
- **serviceInterface**: 对应于 Java 的 Interface 类;
- **annotations**: 注释声明, 可以下面 4 种情形, 如果没有声明, ServiceMix 运行环境将通过查看 Java 类来发现声明类型。
  - "none": 没有声明。
  - "java5": J2SE5.0 的声明。
  - "jsr181": JSR181 的声明。
  - "commons": 指 commons-attributes 的声明, 它是 Apache 的一个项目, 它使得 Java 程序中可以采用 C#/.NET 类型的属性。

```
<sm:activationSpec>
  <sm:component>
    <jsr181:component>
      <jsr181:endpoints>
        <jsr181:endpoint.pojoClass="server.HelloService"
          wSDLResource="classpath:hello_xfire.wSDL"
          style="document"
          serviceInterface="server.Hello"
          annotations="none" />
      </jsr181:endpoints>
    </jsr181:component>
  </sm:component>
</sm:activationSpec>

</sm:activationSpecs>
</sm:container>
</beans>
```

### 7.4.3 创建 hello\_xfire.wSDL

应用 ServiceMix 来创建 Web Service 时, ServiceMix 事实上是通过内部集成 Xfire 来创建 Web Service 的。一定要首先写好 WSDL 文件, ServiceMix 发布 Web Service 时用到了 WSDL 文件里面的各种参数 (如 targetNamespace、port name 等), 具体信息如下。

```
hello_xfire.wSDL
|——接口 (Interface): "helloPortType"
|——操作 (operation): "getHello"
|——输入 (input): "getHello"
|—— "in0"
|——输出 (output): "getHelloResponse"
|—— "out"
|——绑定 (Bindings): "helloHttpBinding"
```



|——操作: “getHello”  
|——端口 (Endpoints): “helloHttpPort”

WSDL 文件的内容如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="hello" xmlns:tns="hello"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:soap12=
    "http://www.w3.org/2003/05/soap-envelope" xmlns:xsd=
    "http://www.w3.org/2001/XMLSchema" xmlns:soapenc11=
    "http://schemas.xmlsoap.org/soap/encoding/" xmlns:soapenc12=
    "http://www.w3.org/2003/05/soap-encoding" xmlns:soap11=
    "http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" attributeFormDefault=
      "qualified" elementFormDefault="qualified" targetNamespace="hello">
      <xsd:element name="getHello">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element maxOccurs="1" minOccurs="1" name="in0" nillable="true" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getHelloResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element maxOccurs="1" minOccurs="1" name="out" nillable="true" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="getHelloResponse">
    <wsdl:part name="parameters" element="tns:getHelloResponse">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="getHelloRequest">
    <wsdl:part name="parameters" element="tns:getHello">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="helloPortType">
    <wsdl:operation name="getHello">
      <wsdl:input name="getHelloRequest" message="tns:getHelloRequest">
      </wsdl:input>
      <wsdl:output name="getHelloResponse" message="tns:getHelloResponse">
      </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
```

```
<wsdl:binding name="helloHttpBinding" type="tns:helloPortType">
  <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="getHello">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="getHelloRequest">
      <wsdlsoap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="getHelloResponse">
      <wsdlsoap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="hello">
  <wsdl:port name="helloHttpPort" binding="tns:helloHttpBinding">
    <wsdlsoap:address location="http://localhost:8192/hello"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

#### 7.4.4 创建 Java 服务类

servicemix.xml 的设置包括服务接口类和服务实现类，它通过<jsr181:endpoint>元素中的属性 `pojoClass="server.HelloService"`来配置 Web Service 实现类，通过 `serviceInterface="server.Hello"`来配置 Web Service 的接口类。例程 7-1 显示了 Web Service 的接口类，例程 7-2 显示了 Web Service 的实现类。Web Service 的配置如下。

```
<sm:activationSpec>
  <sm:component>
    <jsr181:component>
      <jsr181:endpoints>
        <jsr181:endpoint pojoClass="server.HelloService"
                        wsdlResource="classpath:hello_xfire.wsdl"
                        style="document"
                        serviceInterface="server.Hello"
                        annotations="none" />
      </jsr181:endpoints>
    </jsr181:component>
  </sm:component>
</sm:activationSpec>
```

例程 7-1 Hello.java

```
服务接口类
package server;
public interface Hello {
    public String getHello(String name);
}
```

例程 7-2 HelloService.java

服务实现类

```
package server;
public class HelloService implements Hello
{
    public String getHello(String name)
    {
        return "Hello "+name+ "! This is SERVICEMIX Web Service Response.";
    }
}
```

#### 7.4.5 在 ServiceMix 上发布和运行 Web Service

首先设置 `JAVA_HOME = C:\jdk1.5`。

机器的环境变量的 `path` 里面加入设置 `C:\apache-servicemix-3.1-incubating\bin`，以便运行 `servicemix.bat` 命令文件。

发布时将上面创建的 `HelloService.java`、`Hello.java` 编译后的 `class` 文件和 `hello_xfire.wSDL` 文件打成一个 `jar` 包（如 `helloServer.jar`）放在 `{ServiceMix_Home}\lib\optional` 目录下面，`{ServiceMix_Home}` 指 `ServiceMix3.1` 的安装目录，发布包实例结构如下。

```
helloServer.jar
|——HelloService.class
|——Hello.class
|——hello_xfire.wSDL
```

运行 `ServiceMix` 时进入 `{ServiceMix_Home}\soa_servicemix_sample\ws-create` 目录，这是本例在配套光盘中的目录。输入命令行：`servicemix servicemix.xml`，就启动了 `ServiceMix`。

可以通过浏览器直接单击下面的 URL 来验证服务是否发布成功：

`http://localhost:8192/hello/main.wSDL`

图 7-4 表示通过浏览器验证，服务发布成功。

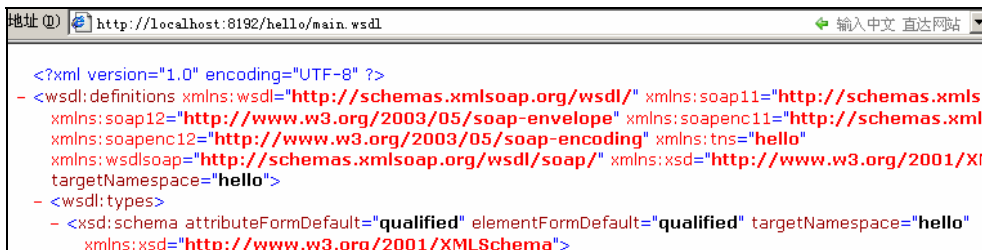


图 7-4 浏览器验证 ServiceMix 是否发布成功

#### 7.4.6 创建客户端调用程序

客户端通过 `URLConnection` 来调用 `Service` 提供的服务。客户端的运行程序主要包括 3

个步骤。

- 建立 URLConnection 连接;
- 发送 SOAP 请求消息;
- 接收返回 SOAP 消息。

运行客户端程序, 需要下面的 request.xml 作为输入参数, 它是一个 SOAP/HTTP 的消息格式, 具体内容如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/" >
  <env:Body>
    <getHello xmlns="hello">
      <in0 xmlns="hello">Aihu</in0>
    </getHello>
  </env:Body>
</env:Envelope>
```

例程 7-3 显示了客户端的调用程序。

例程 7-3 HttpClient.java

```
import java.io.*;
import java.NET.URL;
import java.NET.URLConnection;

public class HttpClient {
    public static void main(String[] args) throws Exception {

        //创建 URLConnection 的连接
        URLConnection connection = new URL("http://localhost:8192/hello/").openConnection();
        connection.setDoOutput(true);
        OutputStream os = connection.getOutputStream();

        //发送请求消息
        FileInputStream fis = new FileInputStream("request.xml");
        int c;
        while ((c = fis.read()) >= 0) {
            os.write(c);
        }
        os.close();
        fis.close();

        //读取返回消息
        BufferedReader in = new BufferedReader(new InputStreamReader
            (connection.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            System.out.println(inputLine);
        }
    }
}
```

```
    }  
    in.close();  
  
    }  
}
```

在 ServiceMix 的服务启动后, 直接运行 Ant 命令执行上面的客户端调用程序。将输出下面的结果:

```
<?xml version='1.0' encoding='UTF-8'?>  
  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">  
    <soapenv:Body>  
      <getHelloout xmlns="hello">Hello Aihu! This is SERVICEMIX Web Service Response.  
    </getHelloout>  
    </soapenv:Body>  
  </soapenv:Envelope>
```

## 7.5 创建SOAP绑定代理服务实例

上小节的实例是 ServiceMix 创建 Web Service 服务。事实上, 作为服务总线, 最关键的事情就是要集成外部的服务。所以这里首先介绍如何作为代理来集成外部的服务。

在 ServiceMix 上创建一个 SOAP 绑定代理包括下面的过程。

- 创建 Servicemix.xml 配置文件, 它用来配置 ServiceMix 运行时的组件;
- 发布 ServiceMix 服务;
- 创建客户端调用程序。

下面将分别予以介绍。

### 7.5.1 流程图

业务流程如下:

- 外部服务请求者 (此处为一个 Java 的客户端程序) 将送一个 HTTP 的请求到 JBI 的专门接收基于 HTTP 协议的请求消息的 httpReciever 组件;
- httpReciever 将请求消息转化成规格化消息 NM 送到 NMR (规格化消息路由器), NMR 再送到 helloSOAP 组件;
- helloSOA 将规格化消息转化成 SOAP 消息送到外部服务 Xfire Web Service 的 HelloService;
- Axis Web Service 将结果响应的 SOAP 消息返回给 helloSOAP;
- helloSOAP 将响应消息转化成规格化的消息送到 NMR, NMR 再将 NM 送到 httpReciever 绑定组件;
- httpReciever 将规格化消息转化成 HTTP 消息送到外部服务请求者。

图 7-5 显示了 ServiceMix 创建 SOAP 绑定代理服务流程图。

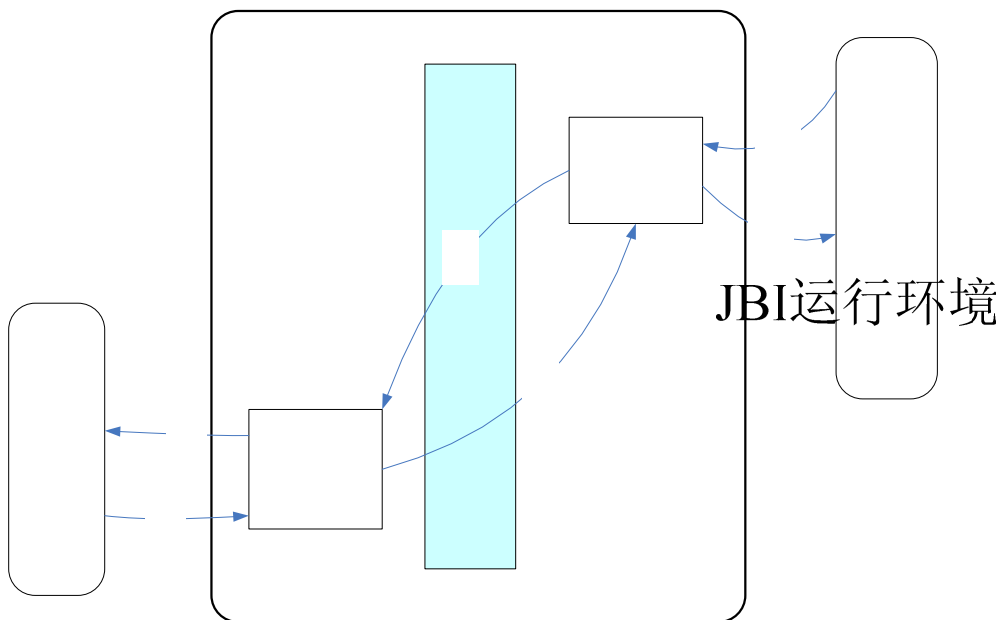


图 7-5 ServiceMix 创建 SOAP 绑定代理服务流程图

### 7.5.2 Servicemix.xml

这个实例主要用到了 ServiceMix 的两个功能。

- 一个是 `org.apache.servicemix.components.http.HttpConnector`，通过这个类在 ServiceMix 外面创建一个 HTTP 的端口“`httpReceiver`”来监听外面的请求。然后通过建立 `destinationService` 来指向所要代理的服务。
- 另外一个就是 `org.apache.servicemix.components.saaj.SaajBinding`，通过 SOAP 绑定将所收到的信息传递给外部服务。所谓 SOAP 绑定，也就是说，将所收到的信息加一个 SOAP 封装的信息传递到外面。

例如如果所收到的 SOAP 信息为：

```
<getHello xmlns="hello">
  <in0 xmlns="hello">Aihu</in0>
</getHello>
```

那么经过封装之后就会变为：

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    <getHello xmlns="hello">
      <in0 xmlns="hello">Aihu</in0>
    </getHello>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

下面是本小节的 servicemix.xml 的内容:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
  xmlns:spring="http://xbean.org/schemas/spring/1.0"
  xmlns:sm="http://servicemix.apache.org/config/1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0"
  xmlns:http="http://servicemix.apache.org/http/1.0"
  xmlns:hello="hello"
  xsi:schemaLocation="http://xbean.org/schemas/spring/1.0 ../../conf/spring-beans.xsd
    http://servicemix.org/config/1.0 ../../conf/servicemix.xsd"
  xmlns:foo="http://servicemix.org/demo/">

  <!-- 创建 JBI 容器-->
  <bean id="jndi" class="org.apache.xbean.spring.jndi.SpringInitialContextFactory"
    factory-method="makeInitialContext" singleton="true" />
  <sm:container id="jbi" useMBeanServer="true" createMBeanServer="true">
    <sm:activationSpecs>
```

创建一个接收外部 HTTP 服务请求的绑定组件 BC 的写法包括两步。

1) 在<sm:activationSpec>元素中定义组件

- **componentName**: 定义组件名称, 如此处为 “httpReceiver”;
- **service**: 定义服务名称, 写法为 “命名空间: 服务名称”, 如 “foo:httpBinding”;
- **endpoint**: 定义端口名称, “httpReceiver”;
- **destinationService**: 定义目标服务, 即绑定组件在收到服务请求消息后, 所要转发到的下一个服务, 写法为 “命名空间: 服务名称”, 如 hello:hello。

2) 在< sm:component >元素中定义组件实现类

- **class**: 定义实现类名, 本例采用 “org.apache.servicemix.components.http.HttpConnector” 接收外部请求消息;
- **Property**: 定义属性名称。属性 “host” 定义服务的 IP, 属性 “port” 定义服务的端口号。

```
<sm:activationSpec componentName="httpReceiver"
  service="foo:httpBinding"
  endpoint="httpReceiver"
  destinationService="hello:hello">

  <sm:component>
    <bean xmlns="http://xbean.org/schemas/spring/1.0"
      class="org.apache.servicemix.components.http.HttpConnector">
      <property name="host" value="localhost"/>
      <property name="port" value="8902"/>
    </bean>
  </sm:component>
</sm:activationSpec>
```

创建一个向外部发出 SOAP 服务请求的绑定组件 BC 的写法包括两步。



## 1) 在&lt;sm:activationSpec&gt;元素中定义组件

- **componentName**: 定义组件名称, 如此处为 “helloService”;
- **service**: 定义服务名称, 写法为 “命名空间: 服务名称”, “hello:hello”, 和上面所定义的目标服务的名称是一致的;
- **endpoint**: 定义端口名称, “helloHttpPort”;

## 2) 在&lt;sm:component&gt;元素中定义组件实现类

- **class**: 定义实现类名, 本例采用 “org.apache.servicemix.components.saaj.SaajBinding” 向服务提供者发送 SOAP 请求消息;
- **Property**: 属性为 “soapEndpoint”, 引用了 javax.xml.messaging.URLEndpoint, 其中构造方法的参数为引入所要调用的外部服务提供者的 URL;

```
<!-- 调用外部的 Xfire hello Web Service -->
<sm:activationSpec componentName="helloService"
                    service="hello:hello"
                    endpoint="helloHttpPort">

  <sm:component>
    <bean xmlns="http://xbean.org/schemas/spring/1.0"
          class="org.apache.servicemix.components.saaj.SaajBinding">
      <property name="soapEndpoint">
        <bean class="javax.xml.messaging.URLEndpoint">
          <constructor-arg value="http://localhost:8080/
                                xfireModule/services/hello"/>
        </bean>
      </property>
    </bean>
  </sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:container>
</beans>
```

## 7.5.3 发布服务和创建客户端的调用程序

因为本例用到了 `HttpConnector` 作为绑定组件来接收外部请求消息, 所以需要加入 `servicemix-components-3.0-incubating.jar` 来支持 `HttpConnector` 的功能; 此外, 本例中也用到了 SOAP 绑定组件来调用外部的服务, 所以需要 `saaj-api-20051026.jar` 和 `saaj-impl-20051026.jar` 来支持 SOAP 绑定的功能。将上面的 jar 包加入到下面的目录即可:

`C:\apache-servicemix-3.1-incubating\lib\optional`

本小节是代理外部的 Web Service 服务, 这里调用的是第 2 章的 Xfire 和 Axis 的 Web Service 服务, 所以需要启动第 2 章的 Web Service 来发动 Xfire 和 Axis 的 Web Service 服务。下面是本小节所要代理的外部服务:

`http://localhost:8080/xfireModule/services/hello`

`http://localhost:8080/axisModule/services/hello`

例程 7-4 显示了客户端的代码。

例程 7-4 HttpClient.java

```
import java.io.*;
import java.NET.URL;
import java.NET.URLConnection;
public class HttpClient {

    public static void main(String[] args) throws Exception {

        URLConnection connection = new URL("http://localhost:8902").openConnection();
        connection.setDoOutput(true);
        OutputStream os = connection.getOutputStream();

        // Post the request file.
        FileInputStream fis = new FileInputStream("request.xml");
        int c;
        while ((c = fis.read()) >= 0) {
            os.write(c);
        }
        os.close();
        fis.close();

        // Read the response.
        BufferedReader in = new BufferedReader(new InputStreamReader
            (connection.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            System.out.println(inputLine);
        }
        in.close();
    }
}
```

所需要的输入参数是 request.xml，内容如下：

```
<getHello xmlns="hello">
    <in0 xmlns="hello">Aihu</in0>
</getHello>
```

上面的请求消息没有采用完整的 SOAP 包信息，因为在本例的 servicemix.xml 的设置中采用了<property name="soapEndpoint">，它会将上面的请求信息通过 SOAP 封装，变成一个完整的 SOAP 请求信息发给服务提供者。

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <getHello xmlns="hello">
            <in0 xmlns="hello">Aihu</in0>
        </getHello>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
</getHello>
</SOAP-ENV:Body></SOAP-ENV:Envelope>
```

如果 request.xml 采用完整的 SOAP 包如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/" >
  <env:Body>
    <getHello xmlns="hello">
      <in0 xmlns="hello">Aihu</in0>
    </getHello>
  </env:Body>
</env:Envelope>
```

那么通过 servicemix.xml 的<property name="soapEndpoint">的封装, 请求信息将会变成:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    <env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
      <env:Body>
        <getHello xmlns="hello">
          <in0 xmlns="hello">Aihu</in0>
        </getHello>
      </env:Body>
    </env:Envelope>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

这样会出现重复封装的情形, 一旦传到服务提供者, 服务提供者将不会认识这样的重复封装 SOAP 消息, 从而出现异常。

本章所代理的是第 2 章的 Xfire 的实例, servicemix.xml 中的配置如下:

```
<constructor-arg value="http://localhost:8080/xfireModule/services/hello"/>
```

运行客户端实例时, 返回的响应消息为:

```
<?xml version="1.0" encoding="UTF-8"?>
<ns1:getHelloResponse xmlns:ns1="hello" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance">
  <ns1:out>Hello Aihu! This is XFIRE Web Service Response.</ns1:out></ns1:getH
elloResponse>
```

读者也可以试着连接第 2 章的 Axis 的实例, 将配置文件改为如下:

```
<constructor-arg value="http://localhost:8080/axisModule/services/hello"/>
```

看看能得到什么样的反应。

## 7.6 创建HTTP绑定代理服务实例

HTTP 绑定和 SOAP 绑定的差别在于 SOAP 绑定将所传来的消息加上一个 SOAP 封装，再传给外部服务，HTTP 绑定将不加上 SOAP 封装包，将所传过来的 HTTP 消息直接传给外部服务。

- 创建 Servicemix.xml 配置文件，它用来配置 ServiceMix 运行时的组件；
- 发布 ServiceMix 服务；
- 创建客户端调用程序。

### 7.6.1 流程图

本小节的业务流程如下。

(1) 外部服务请求者（此处为一个 Java 的客户端程序）将送一个 HTTP 的请求到 JBI 的专门接收基于 HTTP 协议的请求消息的 httpReciever 组件；

(2) httpReciever 将请求消息转化成规格化消息 NM 送到 NMR（规格化消息路由器），NMR 再送到 helloHTTP 组件；

(3) helloHTTP 将规格化消息转化成 HTTP 消息送到外部服务 Xfire Web Service 的 HelloService；

(4) Axis Web Service 将结果响应的 HTTP 消息返回给 helloHTTP；

(5) helloHTTP 将响应消息转化成规格化的消息送到 NMR，NMR 再将 NM 送到 httpReciever 绑定组件；

(6) httpReciever 将规格化消息转化成 HTTP 消息送到外部服务请求者。

图 7-6 显示了 ServiceMix 创建 HTTP 绑定代理服务流程图。

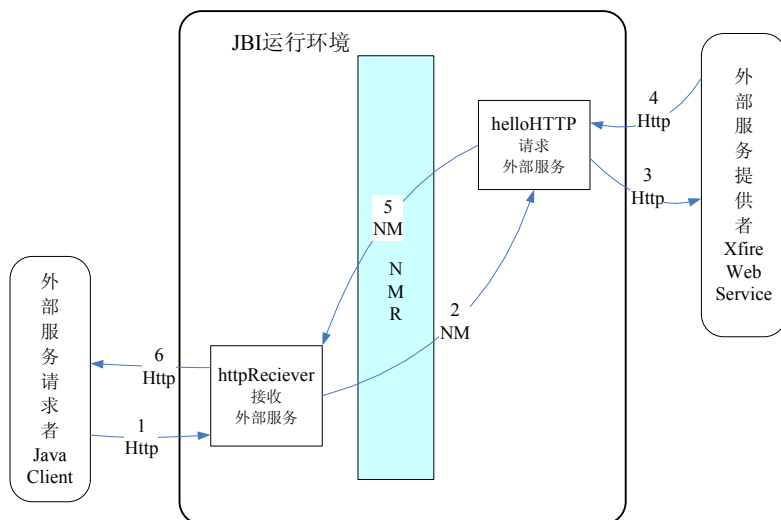


图 7-6 ServiceMix 创建 HTTP 绑定代理服务流程图

## 7.6.2 Servicemix.xml

本小节 HTTP 绑定和 SOAP 绑定的主要差别在于 HTTP 绑定将会采用 ServiceMix 所提供的 `org.apache.servicemix.components.http.HttpInvoker` 来实现代理，而不是采用上节的 `org.apache.servicemix.components.saaj.SaajBinding`，这样它将不会对消息增加一个 SOAP 的封装。

因为要调用外部的 Web Service 服务，所以传过来的消息需要带有 SOAP 格式。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
  xmlns:spring="http://xbean.org/schemas/spring/1.0"
  xmlns:sm="http://servicemix.apache.org/config/1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jsr181="http://servicemix.apache.org/jsr181/1.0"
  xmlns:http="http://servicemix.apache.org/http/1.0"
  xmlns:hello="hello"
  xsi:schemaLocation="http://xbean.org/schemas/spring/1.0 ../../conf/spring-beans.xsd
    http://servicemix.apache.org/config/1.0 ../../conf/servicemix.xsd"
  xmlns:foo="http://servicemix.org/demo/">

  <!-- JBI 容器 -->

  <bean id="jndi" class="org.apache.xbean.spring.jndi.SpringInitialContextFactory"
    factory-method="makeInitialContext" singleton="true" />
  <sm:container id="jbi" useMBeanServer="true" createMBeanServer="true">

    <sm:activationSpecs>
```

此处创建的接收服务请求的绑定组件 BC 完全等同于 7.5 节的实例。

```
    <sm:activationSpec componentName="httpReceiver"
      service="foo:httpBinding"
      endpoint="httpReceiver"
      destinationService="hello:hello">

      <sm:component>
        <bean xmlns="http://xbean.org/schemas/spring/1.0"
          class="org.apache.servicemix.components.http.HttpConnector">
          <property name="host" value="localhost"/>
          <property name="port" value="8902"/>
        </bean>
      </sm:component>
    </sm:activationSpec>
```

下面创建一个向外部发出 HTTP 服务请求的绑定组件 BC，包括两步。

1) 在 `<sm:activationSpec>` 元素中定义组件

- **componentName**: 定义组件名称，如此处为 “helloHTTP”;
- **service**: 定义服务名称，写法为 “命名空间: 服务名称”，“hello:hello”;

## 第 7 章 基于 JBI 的 ServiceMix 服务总线

- endpoint: 定义端口名称, “helloHttpPort”。
- 2) 在< sm:component >元素中定义组件实现类
- class: 定义实现类名, 本例采用“org.apache.servicemix.components.http.HttpInvoker”向服务提供者发送 HTTP 请求消息;
- Property: 属性为 “url”, 为外部服务提供者的 URL。

```
<!--调用外部的 Xfire hello Web Service -->
<sm:activationSpec componentName="helloHTTP"
                    service="hello:hello"
                    endpoint="helloHttpPort">

    <sm:component>
        <bean xmlns="http://xbean.org/schemas/spring/1.0"
            class="org.apache.servicemix.components.http.HttpInvoker">
            <property name="url" value="http://localhost:8080/
                xfireModule/services/hello"/>
        </bean>
    </sm:component>
</sm:activationSpec>
</sm:activationSpecs>
</sm:container>
</beans>
```

发布服务和客户端的调用程序同上小节, 只是所采用的 request.xml 不一样, 含有 SOAP 消息格式如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/" >
    <env:Body>
        <getHello xmlns="hello">
            <in0 xmlns="hello">Aihu</in0>
        </getHello>
    </env:Body>
</env:Envelope>
```

返回的结果如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/
XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soap:Body>
        <ns1:getHelloResponse xmlns:ns1="hello"><ns1:out>Hello Aihu! This is XFIRE Web Service
            Response.</ns1:out>
        </ns1:getHelloResponse>
    </soap:Body>
</soap:Envelope>
```

## 7.7 集成并路由到不同的外部服务实例

上面的实例只是通过服务总线将服务请求代理转给了外部的服务，没有对消息做相应的处理，最多只是通过 SOAP 绑定对消息加了一个 SOAP 包。

在实际的 SOA 集成项目中，主要有下面的需求。

路由的需求，也就是服务总线将会根据所传来的请求消息的内容不同，将请求转给不同的外部服务提供者。如本例中，如果传来的消息为“Aihu (axis)”，则传给下面的地址：

http://localhost:8080/axisModule/services/hello

如果传来的消息为“Aihu (xfire)”，则将请求消息传给下面的地址：

http://localhost:8080/xfireModule/services/hello

### 1) 消息内容的转化 (Transformer)

如本例中，将传来的请求消息格式 Aihu (axis) 转化为 Aihu 后再转给外部服务提供者。

### 2) 传输协议的转换

如本例中将传来的 JMS 的服务请求，转给外部的 HTTP/SOAP 的服务提供者。因为 ServiceMix 是依据 JBI 的架构，所以服务请求者完全不需要关心服务提供者是采用的什么样的传输协议。因为服务请求者并不直接调用服务提供者所提供的服务，它只是将服务请求传给 JBI 服务总线就可以了。

通俗一点来说，服务请求者只要关心是否能够把请求消息传递给服务总线，一旦能够传给服务总线，也就万事大吉了，剩下的事，完全交给服务总线了，是一种非常松散的耦合方式。一个服务组件完全不知道其所要调用的服务组件。

在这一关键点上，和 SCA 的架构是有区别的。作为 SCA 架构，其中一个服务组件，是直接调用另一个服务组件的。SCA 架构和传统的服务组件的进步在于，当完成了基于 SCA 组件的开发之后，可以输出 (Export) 不同的绑定方式让其他服务组件来调用 (如 IBM WebSphere Integration Developer 所提供的 SCA 绑定方式、HTTP/SOAP 的绑定方式和 XML 格式的绑定方式，包括可以提供 Java Reference 来直接让 Java 调用等)。此外，作为 SCA 架构的开发方式，当开发组件需要调用其他服务组件时，也可以选择输入 (import) 各种绑定方式 (主要是 WSDL 绑定方式和 SCA 绑定方式)。一旦实现集成之后，服务组件之间处于一种紧密耦合的状态。

本实例将介绍如何到不同的外部服务提供者，这里分别为 Xfire Web Service 和 Axis Web Service，根据名字“Aihu (axis)”或者名字“Aihu (xfire)”，返回的名字只含有 Aihu。

### 7.7.1 流程图

业务流程如下。

- 外部服务请求者 (此处为一个 Java 的客户端程序) 将送一个 HTTP 的请求到 JBI 的专门接收基于 HTTP 协议的请求消息的 httpReceiver 组件；
- httpReceiver 将请求消息转化成规格化消息 NM 送到 NMR (规格化消息路由器)，NMR 再送到 mytransformer 组件。



mytransformer 组件根据请求消息内容的不同，分别将服务请求消息转发给不同的服务请求绑定组件。如果请求消息内容中含有“(axis)”，则将请求消息转发给“axisHello”绑定组件，请求消息的实例如下：

```
<getHello xmlns="hello">  
  <in0 xmlns="hello">Aihu (axis)</in0>  
</getHello>
```

“axisHello”绑定组件将规格化消息转化成 SOAP 消息送到外部服务 Axis Web Service 的 HelloService。

如果请求消息内容中含有“(xfire)”，mytransformer 则将请求消息转发给“xfireHello”绑定组件，请求消息的实例如下：

```
<getHello xmlns="hello">  
  <in0 xmlns="hello">Aihu (xfire)</in0>  
</getHello>
```

“xfireHello”绑定组件将规格化消息转化成 SOAP 消息送到外部服务 Xfire Web Service 的 HelloService。

图 7-7 显示了 ServiceMix 创建 HTTP 绑定代理服务流程图。

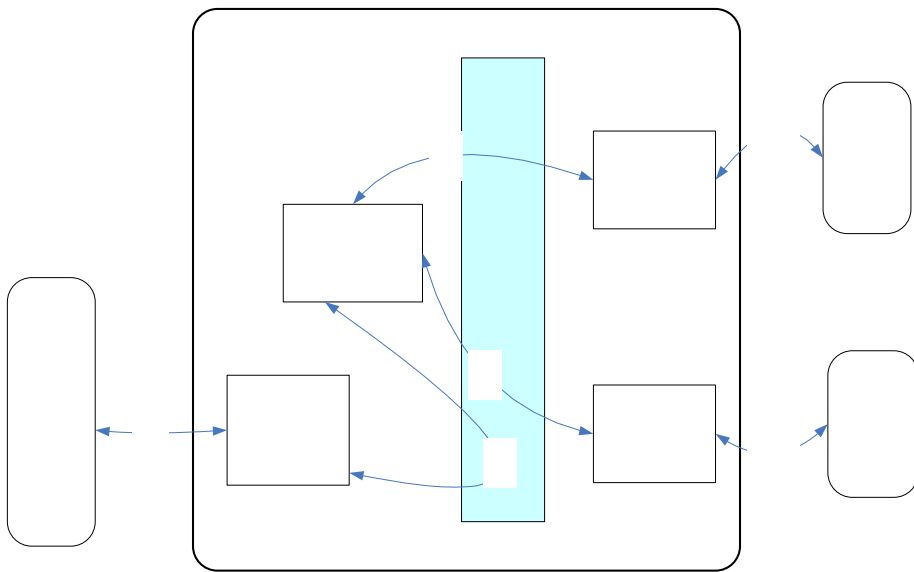


图 7-7 ServiceMix 创建 HTTP 绑定代理服务流程图

### 7.7.2 Servicemix.xml

本例将创建 3 个绑定组件，1 个服务引擎组件。

- httpReceiver 绑定组件：专门负责接收外部的服务请求消息；
- axisHello 绑定组件：向外部服务提供者 Axis Web Service 发出服务请求；
- xfireHello 绑定组件：向外部服务提供者 Xfire Web Service 发出服务请求；
- mytransformer 服务引擎组件：作为 httpReceiver 的目标服务，在收到传来的服务

请求消息后，解析服务请求消息，根据服务请求消息内容的不同，分别将请求消息转给 axisHello 绑定组件和 xfireHello 绑定组件。

以上绑定组件的创建方法在上面章节都做了详细的介绍，下面介绍如何在 servicemix.xml 中创建服务引擎组件。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xbean.org/schemas/spring/1.0"
  xmlns:spring="http://xbean.org/schemas/spring/1.0"
  xmlns:sm="http://servicemix.apache.org/config/1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xbean.org/schemas/spring/1.0 ../conf/spring-beans.xsd
    http://servicemix.org/config/1.0 ../conf/servicemix.xsd"
  xmlns:foo="http://servicemix.org/demo">

  <!-- JBI 容器 -->
  <bean id="jndi" class="org.apache.xbean.spring.jndi.SpringInitialContextFactory"
    factory-method="makeInitialContext" singleton="true" />
  <sm:container id="jbi" useMBeanServer="true" createMBeanServer="true">

    <sm:activationSpecs>

      <!-- 创建一个端口为 8912 的 HTTP 服务，其目的地指向 foo:mytransformer -->
      <sm:activationSpec componentName="httpReceiver"
        service="foo:httpBinding"
        endpoint="httpReceiver"
        destinationService="foo:mytransformer">

        <sm:component>
          <bean xmlns="http://xbean.org/schemas/spring/1.0"
            class="org.apache.servicemix.components.http.HttpConnector">
            <property name="host" value="localhost"/>
            <property name="port" value="8912"/>
          </bean>
        </sm:component>
      </sm:activationSpec>
```

在<sm:activationSpec>元素中定义的属性如下。

- **componentName**: 组件名称，此处为"mytransformer";
- **service**: 服务名称，写法为“命名空间: 服务名称”，此处为"foo:mytransformer"。

在<sm: component >元素服务引擎组件的实现类，用到了 Spring 的配置机制。

```
<bean xmlns="http://xbean.org/schemas/spring/1.0" class=具体的实现类/>
<sm:activationSpec componentName="mytransformer" service="foo:mytransformer">
  <sm:component>
    <bean xmlns="http://xbean.org/schemas/spring/1.0"
      class="tranformer.MyTransformer"/>
  </sm:component>
</sm:activationSpec>
```

```
<!-- 调用外部的 Axis Web Service -->
    <sm:activationSpec componentName="axisHello"
                        service="foo:axisHello"
                        endpoint="axisHello">

        <sm:component>
            <bean xmlns="http://xbean.org/schemas/spring/1.0"
                class="org.apache.servicemix.components.saaaj.SaaajBinding">
                <property name="soapEndpoint">
                    <bean class="javax.xml.messaging.URLEndpoint">
                        <constructor-arg value="http://localhost:8081/
                            axisModule/services/hello"/>
                    </bean>
                </property>
            </bean>
        </sm:component>
    </sm:activationSpec>

<!--调用外部的 Xfire Web Service -->
    <sm:activationSpec componentName="xfireHello"
                        service="foo:xfireHello"
                        endpoint="xfireHello">

        <sm:component>
            <bean xmlns="http://xbean.org/schemas/spring/1.0"
                class="org.apache.servicemix.components.saaaj.SaaajBinding">
                <property name="soapEndpoint">
                    <bean class="javax.xml.messaging.URLEndpoint">
                        <constructor-arg value="http://localhost:8081/
                            xfireModule/services/hello"/>
                    </bean>
                </property>
            </bean>
        </sm:component>
    </sm:activationSpec>
</sm:activationSpecs>
</sm:container>
</beans>
```

### 7.7.3 服务器端路由（Router）和格式转化（Transformer）程序

下面是服务引擎组件的实现类，它负责在收到请求消息后，解析请求消息，并根据消息内容的不同将消息转发给不同的绑定组件。

例程 7-5 显示了引擎组件 MyTransformer.java 的代码，它的 onMessageExchange 方法不停地监听和接收来自 NMR 的规格化消息。在 JBI 里面，所有的信息都是通过 MessageExchange 传过来的，每一个传过来的 MessageExchange 都有一个唯一的确认号 ExchangeId。一个具体的 MessageExchange 的内容如下：

```
InOut[
  id: ID:OEM-MICRO-3085-1191149544195-2:0
  status: Active
  role: provider
  service: {http://servicemix.org/demo}mytransformer
  endpoint: mytransformer
  in: <?xml version="1.0" encoding="UTF-8"?><getHello xmlns="hello">
    <in0 xmlns="hello">Aihu (xfire)</in0>
  </getHello>
]
```

其中，InOut 继承了 MessageExchange，表示它是一个请求和响应的消息模式，它所包含的属性如下。

- id: 对应于 ExchangeId，对于每一个 MessageExchange，都有一个唯一的确认号；
- status: 表示该消息交换（MessageExchange）的状态，有“active”、“done”、“error”3 种类型；
- role: 角色，此处为“provider”，表示为服务提供者；
- service: 服务名称，为“{http://servicemix.org/demo}mytransformer”；
- endpoint: 端口，为“mytransformer”；

#### 例程 7-5 MyTransformer.java

```
package transformer;
import javax.jbi.messaging.MessageExchange;
.....省略 import

public class MyTransformer extends ComponentSupport implements MessageExchangeListener {
    private static final Log log = LogFactory.getLog(MyTransformer.class);
    public MyTransformer() {
        //定义服务的名称，以便 JBI 通过此名字来找到此服务组件
        super(new QName("http://servicemix.org/demo", "receiver"), "input");
    }
    public void onMessageExchange(MessageExchange exchange) throws MessagingException
    {
        //下面的方法可以得到 exchangeId 号
        String exchangid = exchange.getExchangeId();
        //消息交换状态 ExchangeStatus 是 active 状态，
        if (exchange.getStatus() == ExchangeStatus.ACTIVE)
        {
            //下面的方法得到端口号的名称，通过下面的方法可以得到端口号的名称
            String endPoint = exchange.getEndpoint().getEndpointName();
            //端口为"mytransformer"时，将会解析消息，然后根据消息内容的不同，
            //转给不同的绑定组件
            if (endPoint.equalsIgnoreCase("mytransformer"))
            {
                //将 MessageExchange 转化为 InOut 的形式
            }
        }
    }
}
```

## 第 7 章 基于 JBI 的 ServiceMix 服务总线

```
InOut inout = (InOut) exchange;
// 取出 NormalizedMessage
NormalizedMessage inMessage = inout.getInMessage();
//对下面的请求消息进行解析, 取出()内部的消息, 如果是(axis), 表示将会
//调用 axis //Web service
//<getHello xmlns="hello"><in0 xmlns="hello">Aihu (axis)</in0></getHello>
String dest = getProcessedMessage(exchange);

//通过 QName 来定义目标服务, 为此服务将请求消息所要传到的下一个
//绑定服务组件
QName qName = null;
if(dest.equalsIgnoreCase("axis"))
    qName = new QName("http://servicemix.org/demo", "axisHello");
else
    qName = new QName("http://servicemix.org/demo", "xfireHello");
//创建 InOut 的消息交换
InOut inout1 = createInOutExchange(qName, null, null);

//因为是异步方式的调用, 所有的请求和响应都是通过 JBI 来控制的, 所以
//这里首先把 exchangeId 存到 MessageExchange 的属性 (Property) 里面,
//以便以后 JBI 能够取出来, 送回服务响应消息
inout1.setProperty("requestId", exchangeId);
//将收到的请求消息置于新创建的 InOut 消息交换中
inout1.setInMessage(inMessage);
send(inout1);
}
else //端口不是"mytransformer"时, 将响应消息返回给服务请求者
{
    InOut inout = (InOut) exchange;
    NormalizedMessage outMessage = inout.getOutMessage();

    //取出原始的发出请求的 MessageExchange 的 ExchangeId, 通过这个 ID 来
    //找到原始的 MessageExchange, 前面已经将它存入到 ConcurrentHashMap
    //里面, 因为每一个 MessageExchange 都有一个端口 Endpoint 来监听消息,
    //这样的话, 它就可以从 JBI 收到消息后送回给服务请求者
    String id = (String) getProperty(inout, "requestId");
    InOut inout33 = (InOut) ht.get(id);
    inout33.setOutMessage(outMessage);
    send(inout33);
}

//当角色为服务提供者时, 保存此消息交换的 ID
if (exchange.getRole() == Role.PROVIDER)
{
    ht.put(exchangid, exchange);
} else
{
}
```

```
        //改变消息交换的状态为“done”，表示此消息交换结束。  
        done(exchange);  
    }  
}
```

程序中的其他方法如下。

- **getProperty**: 得到消息交换中的属性值;
- **getInProperty**: 得到消息交换中输入消息的属性值;
- **getOutProperty**: 得到消息交换中输出消息的属性值;
- **getProcessedMessage**: 解析消息交换中的“()”中的信息;
- **processNode**: 通过解析规格消息中的 XML 节点, 得到 XML 节点值;
- **textValueOfXPath**: 通过 xpath 和节点, 得到节点值;
- **nodeOfXPath**: nodeOfXPath。

下面是具体的程序部分:

```
//通过“name”参数, 得到消息交换中的属性值  
protected Object getProperty(MessageExchange me, String name) {  
    return me.getProperty(name);  
}  
  
//通过“name”参数, 得到消息交换中输入消息的属性值  
protected Object getInProperty(MessageExchange me, String name) {  
    return me.getMessage("in").getProperty(name);  
}  
  
//通过“name”参数, 得到消息交换中输出消息的属性值  
protected Object getOutProperty(MessageExchange me, String name) {  
    return me.getMessage("out").getProperty(name);  
}  
  
private Map ht = new ConcurrentHashMap();  
  
//解析消息交换中的“()”中的信息  
protected String getProcessedMessage(MessageExchange me)  
{  
    String destination = null;  
    NormalizedMessage inMessage = me.getMessage("in");  
    if(inMessage != null)  
    {  
        try {  
            destination = (String) processNode(inMessage);  
        } catch (TransformerException e) {  
            e.printStackTrace();  
        }  
        .....//省略其他的 catch 部分
```

```

    }
    NormalizedMessage outMessage = me.getMessage("out");

    if(outMessage != null)
    {
        try {
            destination = (String) processNode(outMessage);
        } catch (TransformerException e) {
            e.printStackTrace();
        }
        .....//省略其他的 catch 部分
    }
    return destination;
}

//通过解析规格消息中的 XML 节点，得到 XML 节点值
public String processNode(NormalizedMessage nm) throws TransformerException,
    ParserConfigurationException, SAXException, IOException
{
    String destination = null;
    Document document = null;
    try {
        document= (Document) new SourceTransformer().toDOMNode(nm);
    } catch (MessagingException e) {
        e.printStackTrace();
    }
    .....//省略其他的 catch 部分

    String nodeValue = textValueOfXPath(document, "/*[local-name()='in0']");
    Node node = nodeOfXPath(document, "/*[local-name()='in0']");

    if(nodeValue.indexOf("(xfire)")>-1)
    {
        String modifiedValue = nodeValue.substring(0,nodeValue.indexOf("(xfire)")-1);
        node.getFirstChild().setNodeValue(modifiedValue);
        destination = "xfire";
    } else if(nodeValue.indexOf("(axis)")>-1)
    {
        String modifiedValue = nodeValue.substring(0,nodeValue.indexOf("(axis)")-1);
        node.getFirstChild().setNodeValue(modifiedValue);
        destination = "axis";
    }
    DOMSource domsource = new DOMSource( document );
    try {
        nm.setContent(domsource);
    } catch (MessagingException e) {

```



```

        e.printStackTrace();
    }
    return destination;
}

```

//通过 xpath 和节点，得到节点值

```

protected String textValueOfXPath(Node node, String xpath) throws TransformerException {
    CachedXPathAPI cachedXPathAPI = new CachedXPathAPI();
    NodeIterator iterator = cachedXPathAPI.selectNodeIterator(node, xpath);
    Node root = iterator.nextNode();
    if (root instanceof Element) {
        Element element = (Element) root;
        if (element == null) {
            return "";
        }
        String text = DOMUtil.getElementText(element);
        return text;
    }
    else if (root != null) {
        return root.getNodeValue();
    }
    else {
        return null;
    }
}

```

//通过 xpath 和节点，得到节点值

```

protected Node nodeOfXPath(Node node, String xpath) throws TransformerException {
    CachedXPathAPI cachedXPathAPI = new CachedXPathAPI();
    NodeIterator iterator = cachedXPathAPI.selectNodeIterator(node, xpath);
    Node root = iterator.nextNode();
    if (root instanceof Element) {
        Element element = (Element) root;
        if (element == null) {
            return null;
        }
        return element;
    }
    else if (root != null) {
        return root;
    }
    else {
        return null;
    }
}

```

#### 7.7.4 发布服务和创建客户端调用程序

发布服务时将上面的服务程序编译后打成 jar 包放到 {ServiceMix\_Home}\lib\optional 目录, 然后参照 7.4.5 节来发布和运行 ServiceMix 即可。

因为本小节代理外部的 Web Service 服务, 这里调用的是第 5 章的 Xfire 和 Axis 的 Web Service 服务, 所以需要启动第 5 章的 Web Service 来发动 Xfire 和 Axis 的 Web Service 服务。下面是本小节所要代理的外部服务。

- <http://localhost:8081/xfireModule/services/hello>
- <http://localhost:8081/axisModule/services/hello>

客户端通过 Java 客户端的 URLConnection 建立连接, 将请求消息发给 ServiceMix 的绑定组件, ServiceMix 会进一步调用服务提供者, 然后将响应消息返回给服务请求者。例程 7-6 显示了客户端的代码程序。

例程 7-6 HttpClient.java

```
import java.io.*;
import java.NET.URL;
import java.NET.URLConnection;

public class HttpClient {

    public static void main(String[] args) throws Exception {

        //建立连接
        URLConnection connection = new URL("http://localhost:8912").openConnection();
        connection.setDoOutput(true);
        OutputStream os = connection.getOutputStream();

        //从 request.xml 中取出请求消息, 送到 ServiceMix
        FileInputStream fis = new FileInputStream("request.xml");
        int c;
        while ((c = fis.read()) >= 0) {
            os.write(c);
        }
        os.close();
        fis.close();

        // 读取返回的响应消息
        BufferedReader in = new BufferedReader(new InputStreamReader
            (connection.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            System.out.println(inputLine);
        }
        in.close();
    }
}
```

```
}  
}
```

输入参数为下面的 request.xml 文件，内容为：

```
<getHello xmlns="hello">  
  <in0 xmlns="hello">Aihu (axis)</in0>  
</getHello>
```

运行 ANT，下面是得到的响应结果：

```
[echo] Running example client  
[java] <?xml version="1.0" encoding="UTF-8"?>  
<ns1:getHelloResponse xmlns:ns1="hello"  
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"  
xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
<getHelloReturn xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"  
xsi:type="soapenc:string">Hello Aihu! This is AXIS Web Service Response  
</getHelloReturn>  
</ns1:getHelloResponse>
```

可以看到所输入的“Aihu (axis)”，已经返回了“Aihu”，并且路由到了 Axis 所提供的 Web Service 服务。

如果将上面的 request.xml 中的 **Aihu (axis)** 改为 **Aihu(xfire)**，重新运行 Ant，将会看到下面的结果。

```
[echo] Running example client  
[java] <?xml version="1.0" encoding="UTF-8"?>  
<ns1:getHelloResponse xmlns:ns1="hello"  
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"  
xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
<getHelloReturn xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"  
xsi:type="soapenc:string">Hello Aihu! This is Xfire Web Service Response  
</getHelloReturn>  
</ns1:getHelloResponse>
```

可以看到，ServiceMix 会根据消息内容的不同，路由到不同的 Web Service 服务，同时可以将消息做一定的修改。

## 7.8 集成并路由到不同服务的房屋贷款实例

本小节的实例主要介绍如何通过 JBI 的 Java 编程来实现业务流程的控制和管理。只需要输入用户姓名，这个系统会首先自动查出客户的房屋数量，然后自动转到相应的银行进行房屋贷款服务，客户最后可以得到对应于他的目前已有房屋数量的贷款首付和贷款利率。

本小节实例的主要目的在于如何在 ServiceMix 里面创建各种复杂的服务组件，以及如何实现这些组件之间的调用。

本小节是通过 JMS 来接收服务请求者的服务请求的。

此外本小节也介绍了如何应用 SA (Service Assembly, 服务集成模块) 来发布 ServiceMix 的服务模块。

### 7.8.1 流程图

图 7-8 显示了 ServiceMix 的房屋贷款服务的实现流程图, 具体过程如下。

- ServiceMix 通过 JMS 接收客户的服务请求。
- ServiceMix 将首先以客户姓名 (name) 作为输入变量, 调用 HouseLoanAgency 的服务, 得到客户的目前拥有的房屋数量;
- HouseLoanAgency 服务进一步将客户的目前拥有的房屋数量结果发送到 LenderGateway 服务;
- LenderGateway 服务根据房屋数量的不同调用不同的 Bank 服务。如果客户房屋数量为 0, 将调用 Bank0 的服务; 如果客户房屋数量为 1, 将调用 Bank1 的服务; 如果客户房屋数量为 2, 将调用 Bank2 的服务; 如果客户房屋数量大于 2, 将调用 Bank3 的服务;
- HouseLoanAgency 将从 Bank 返回的首付比率和贷款利率返回给服务请求者。

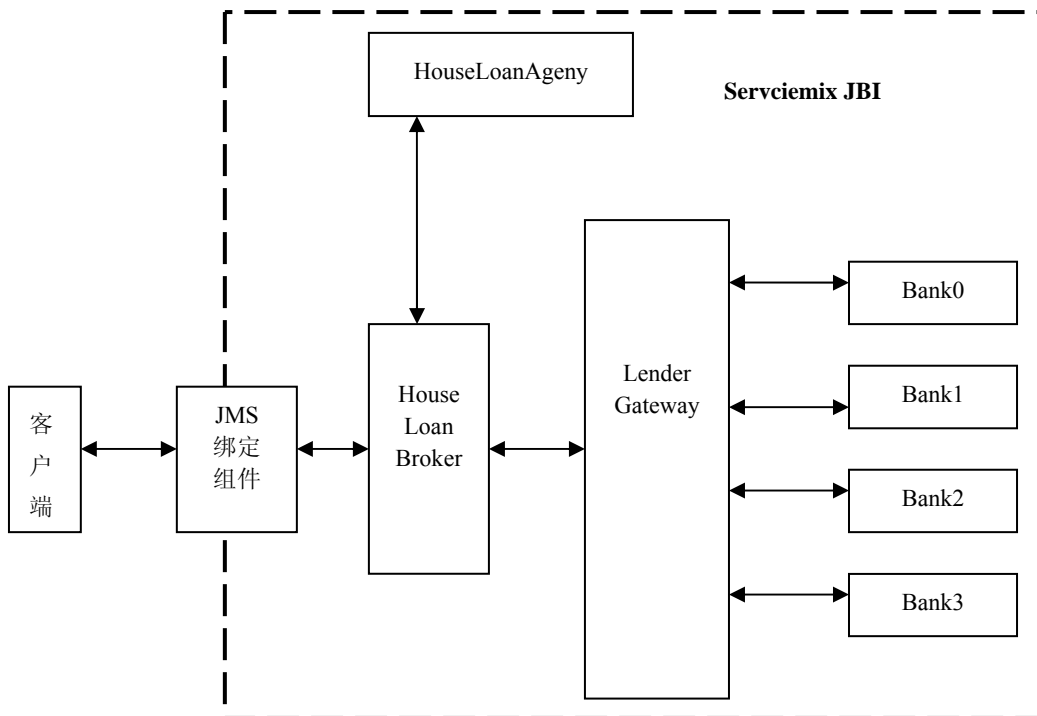


图 7-8 ServiceMix 创建房屋贷款服务流程图

在 ServiceMix 上创建房屋贷款应用包括下面的过程。

- 创建服务器端的 Java 程序;
- 创建 Servicemix.xml 配置文件, 包括配置 SA (Service Assembly) 和 SU (Service

Unit) “服务单元” 来配置服务模块;

- 发布 ServiceMix 服务, 创建客户端调用程序。

下面将分别予以介绍。

## 7.8.2 服务器端程序

房屋贷款的主要模块, 它接收外部服务请求者的请求消息后, 首先调用 HouseLoanAgency 服务, 通过客户的姓名得到客户的房屋数量; 再调用 HouseLoanGateway 服务, HouseLoanGateway 服务会根据房屋数量的不同调用不同的 Bank 服务。它采用了异步调用方式, 在 HouseLoanBroker 收到绑定组件通过 NMR 送来的消息交换 (MessageExchange) 后, 并不是马上返回响应消息, 而是将 MessageExchange 通过其唯一编号保存于 ConcurrentHashMap, 在最终得到了 Bank 的服务响应后, 才从 ConcurrentHashMap 取出 MessageExchange, 返回响应消息。

例程 7-7 显示了房屋贷款的主要 Java 实现类 HouseLoanBroker.java。下面介绍其方法。

- onMessageExchange: 不定期地监听来自 NMR 的消息, 如果消息交换的角色是服务提供者, 则处理服务的请求消息, 否则, 处理服务的响应消息;
- processLoanQuote: 收到贷款的响应消息后, 将响应消息返回给服务请求者。

例程 7-7 HouseLoanBroker.java

```
package houseloan;
//.....省略 impor

//监听来自 NMR 的消息, 处理请求消息
public class HouseLoanBroker extends ComponentSupport implements MessageExchangeListener {

    private static final Log log = LogFactory.getLog(HouseLoanBroker.class);

    public HouseLoanBroker() {
        super(new QName(Constants.LOANBROKER_NS, Constants.LOANBROKER_
            SERVICE), "input");
    }

    public void onMessageExchange(MessageExchange exchange) throws MessagingException {
        // Provider role, 如果是服务提供者的角色, 则开始处理请求消息
        if (exchange.getRole() == Role.PROVIDER) {
            processInputRequest(exchange);
            // Consumer role, 如果是服务消费者的角色, 则分别处理来自 Houseagency 的响
            //应和来自借款代理//的响应
        } else {
            ServiceEndpoint ep = exchange.getEndpoint();
            // House agency response
            if (ep.getServiceName().getLocalPart().equals(Constants.
                HOUSELOANAGENCY_SERVICE)) {
                processHouseLoanAgencyResponse(exchange);
            } else if (ep.getServiceName().getLocalPart().
                equals(Constants.LENDERGATEWAY_SERVICE)) {
```

```
        processLenderGatewayResponse(exchange);
    } else {
        processLoanQuote(exchange);
    }
}

//处理贷款响应业务
private void processLoanQuote(MessageExchange exchange) throws MessagingException {
    log.info("Receiving loan quote");
    // Get aggregation
    String id = (String) getProperty(exchange, Constants.PROPERTY_CORRELATIONID);
    //从聚集中得到原始的消息交换的 ID
    Aggregation ag = (Aggregation) aggregations.get(id);
    // 从贷款额度 LoanQuote 中得到银行、贷款利率、首付比率
    LoanQuote q = new LoanQuote();
    q.bank = exchange.getEndpoint().getServiceName().getLocalPart();
    q.rate = (Double) getOutProperty(exchange, Constants.PROPERTY_RATE);
    q.firstPaidRatio = (Double) getOutProperty(exchange, Constants.FIRST_PAID_RATIO);
    done(exchange);
    // 检查是否所有的贷款额度都收到了
    synchronized (ag) {
        ag.quotes.add(q);
        if (ag.quotes.size() == ag.numbers) {
            LoanQuote best = null;
            for (Iterator iter = ag.quotes.iterator(); iter.hasNext();) {
                q = (LoanQuote) iter.next();
                best = q;
            }
            NormalizedMessage response = ag.request.createMessage();
            response.setProperty(Constants.FIRST_PAID_RATIO, best.firstPaidRatio);
            response.setProperty(Constants.PROPERTY_RATE, best.rate);
            response.setProperty(Constants.PROPERTY_BANK, best.bank);
            ag.request.setMessage(response, "out");
            send(ag.request);
            aggregations.remove(id);
        }
    }
}
```

接下来介绍 HouseLoanBroker 其他的方法。

- **processLenderGatewayResponse**: 处理来自贷款服务的响应结果, 首先从消息交换中得到原始的消息交换 ID, 接下来从聚集中得到原始的消息交换及其相关消息的实例, 创建新的消息交换, 将响应消息返回给服务请求者。
- **processHouseLoanAgencyResponse**: 收到房屋数量的结果后, 将服务转给 LenderGateway, LenderGateway 会根据不同的房屋数量调用不同的 Bank 服务。
- **processInputRequest**: 处理输入请求后, 将客户姓名转给房屋贷款机构, 房屋贷款机构将会根据姓名来查询客户的房屋数量。

下面是具体的代码：

```
//集成银行的返回信息
private void processLenderGatewayResponse(MessageExchange exchange) throws
    MessagingException {
    log.info("Receiving lender gateway response");
    // Get aggregation
    //首先从消息交换中得到原始的消息交换 ID
    String id = (String) getProperty(exchange, Constants.PROPERTY_CORRELATIONID);
    //从聚集中得到原始的消息交换及其相关消息的实例
    Aggregation ag = (Aggregation) aggregations.get(id);
    //得到所有的响应银行，本例中只有一个
    QName[] recipients = (QName[]) getOutProperty(exchange,
        Constants.PROPERTY_RECIPIENTS);
    ag.numbers = recipients.length;
    for (int i = 0; i < recipients.length; i++) {
        //创建新的消息交换
        InOut inout = createInOutExchange(recipients[i], null, null);
        inout.setProperty(Constants.PROPERTY_CORRELATIONID, id);
        NormalizedMessage msg = inout.createMessage();

        //消息赋值到新的消息交换中
        inout.setInMessage(msg);
        //将响应消息返回给服务请求者
        send(inout);
    }
    done(exchange);
}

//收到房屋数量的结果后，将服务转给 LenderGateway
private void processHouseLoanAgencyResponse(MessageExchange exchange) throws
    MessagingException {
    log.info("Receiving credit agency response");
    String id = (String) getProperty(exchange, Constants.PROPERTY_CORRELATIONID);
    Aggregation ag = (Aggregation) aggregations.get(id);
    // 填写信息

    // 发送到借贷处
    ag.houseNumber=(Integer) getOutProperty(exchange, Constants.HOUSE_NUMBER);
    InOut inout = createInOutExchange(new QName(Constants.LOANBROKER_NS,
        Constants.LENDERGATEWAY_SERVICE), null, null);
    inout.setProperty(Constants.PROPERTY_CORRELATIONID, id);
    NormalizedMessage msg = inout.createMessage();

    msg.setProperty(Constants.HOUSE_NUMBER, ag.houseNumber);
    inout.setInMessage(msg);
    send(inout);
    done(exchange);
}
```

//处理输入请求后，将客户姓名转给房屋贷款机构，房屋贷款机构将会根据姓名来查询客户的



## 第7章 基于 JBI 的 ServiceMix 服务总线

```
//房屋数量
private void processInputRequest(MessageExchange exchange) throws MessagingException {
    if (exchange.getStatus() == ExchangeStatus.ACTIVE) {
        log.info("Receiving loan request");
        String id = exchange.getExchangeId();
        Aggregation ag = new Aggregation();
        ag.request = exchange;

        ag.name = (String) getInProperty(exchange, Constants.NAME);
        aggregations.put(id, ag);

        //将客户姓名转给房屋贷款机构，房屋贷款机构将会根据姓名来查询客户的
        //房屋数量
        InOut inout = createInOutExchange(new QName(Constants.LOANBROKER_NS,
                                                    Constants.HOUSELOANAGENCY_
                                                    SERVICE), null, null);
        inout.setProperty(Constants.PROPERTY_CORRELATIONID, id);
        NormalizedMessage msg = inout.createMessage();

        msg.setProperty(Constants.NAME, ag.name);
        inout.setInMessage(msg);
        send(inout);
    }
}

//得到消息的属性
protected Object getProperty(MessageExchange me, String name) {
    return me.getProperty(name);
}

//得到输入消息的属性
protected Object getInProperty(MessageExchange me, String name) {
    return me.getMessage("in").getProperty(name);
}

//得到输出消息的属性
protected Object getOutProperty(MessageExchange me, String name) {
    return me.getMessage("out").getProperty(name);
}

private Map aggregations = new ConcurrentHashMap();

//聚集原始的消息交换的信息
public static class Aggregation {
    public MessageExchange request;
    public int numbers;
    public String name;
    public Integer houseNumber;
    public List quotes = new ArrayList();
}
```

```
//银行返回的贷款额度，包括银行、贷款比率、贷款额度等
public static class LoanQuote {
    public String bank;
    public Double rate;
    public Double firstPaidRatio;
}

}
```

例程 7-8 显示了房屋贷款机构如何根据顾客姓名输出房屋数量。比较关键之处是 HouseLoanAgency.java 继承了 ServiceMix 的 TransformComponentSupport，这样可以通过 transform 来改变消息交换的内容。它首先从输入消息交换中得到客户姓名，通过客户姓名查询房屋数量记录得到客户的房屋数量，并将房屋数量赋值到输出消息交换的属性中发送出去。

例程 7-8 HouseLoanAgency.java

```
package houseloan;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.servicemix.components.util.TransformComponentSupport;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.MessagingException;
import javax.jbi.messaging.NormalizedMessage;

public class HouseLoanAgency extends TransformComponentSupport {

    private static final Log log = LogFactory.getLog(HouseLoanAgency.class);

    //每个客户所拥有的房屋记录
    private static String[][] loanStatusList =
        { {"Zhang San","1"},
          {"Li Si","2"},
          {"Wang Wu","3"},
          {"Zhao Liu","4"}
        };

    //得到每个客户的房屋数量
    private int getHouseNumber(String name) {
        int houseNumber = 0;
        for (int i=0; i<loanStatusList.length; i++){
            String[] item = loanStatusList[i];
            String itemName = item[0];
            int itemNum = Integer.parseInt(item[1]);

            if (name.equalsIgnoreCase(itemName))
            {
                houseNumber = itemNum;
            }
        }
    }
}
```

```
        break;
    }
}
return houseNumber;
}

//改变消息的内容
protected boolean transform(MessageExchange exchange, NormalizedMessage in,
    NormalizedMessage out)
throws MessagingException
{
    //从输入消息交换中得到客户姓名
    String name = (String) in.getProperty(Constants.NAME);
    log.info("Receiving loan request house agency name>>>>>"+name+"<<<<");
    int houseNumber = getHouseNumber(name);
    log.info("Receiving loan request house number >>>>>"+houseNumber+"<<<<");

    //将得到的房屋数量的结果赋值到输出消息交换的属性中
    out.setProperty(Constants.HOUSE_NUMBER, new Integer(houseNumber));
    log.info("Receiving credit agency request");
    return true;
}
}
```

例程 7-9 显示了 HouseLoanGateway.java 的代码，它通过不同的房屋数量调用不同的 Bank 服务：

- 房屋数量为 0 时，将会调用 Bank0 的服务；
- 房屋数量为 1 时，将会调用 Bank1 的服务；
- 房屋数量为 2 时，将会调用 Bank2 的服务；
- 房屋数量大于 2 时，将会调用 Bank3 的服务。

例程 7-9 HouseLoanGateway.java

```
package houseloan;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.servicemix.components.util.TransformComponentSupport;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.MessagingException;
import javax.jbi.messaging.NormalizedMessage;
import javax.xml.namespace.QName;

public class HouseLoanGateway extends TransformComponentSupport {

    private static final Log log = LogFactory.getLog(HouseLoanGateway.class);

    protected boolean transform(MessageExchange exchange, NormalizedMessage in,
        NormalizedMessage out)
```

```
throws MessagingException
{
    //通过消息交换的属性得到房屋数量
    int houseNumber = ((Integer) in.getProperty(Constants.HOUSE_NUMBER)).intValue();

    //根据房屋数量的不同调用不同的 Bank 服务
    QName[] recipients;
    if (houseNumber==0) {
        recipients = new QName[] { new QName(Constants.LOANBROKER_NS, "bank0") };
    } else
    if (houseNumber==1) {
        recipients = new QName[] { new QName(Constants.LOANBROKER_NS, "bank1") };
    } else
    if (houseNumber==2) {
        recipients = new QName[] { new QName(Constants.LOANBROKER_NS, "bank2") };
    } else {
        recipients = new QName[] { new QName(Constants.LOANBROKER_NS,
            "bank3") };
    }
    out.setProperty(Constants.PROPERTY_RECIPIENTS, recipients);
    return true;
}
}
```

例程 7-10 显示了 Bank 服务的代码，它将返回贷款利率和首付比率的服务请求。比较关键之处是 Bank.java 继承了 ServiceMix 的 TransformComponentSupport，这样可以通过 transform 方法得到配置在 serviemix.xml 中的每个 Bank 的名称、贷款利率、首付比率。并将贷款利率和首付比率赋值到消息交换的属性中返回给服务请求者。

例程 7-10 Bank.java

```
package houseloan;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.servicemix.components.util.TransformComponentSupport;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.MessagingException;
import javax.jbi.messaging.NormalizedMessage;

public class Bank extends TransformComponentSupport {

    private static final Log log = LogFactory.getLog(Bank.class);

    //贷款利率
    private double rate;
    //首付比率
    private double firstPaidRatio;

    public double getFirstPaidRatio() {
        return firstPaidRatio;
    }
}
```

```
public void setFirstPaidRatio(double firstPaidRatio) {
    this.firstPaidRatio = firstPaidRatio;
}

public void setRate(double rate1)
{
    rate = rate1;
}

public double getRate()
{
    return rate;
}

protected boolean transform(MessageExchange exchange, NormalizedMessage in,
    NormalizedMessage out)
    throws MessagingException
{
    //分别将贷款利率和首付比率赋值到消息交换的属性中
    out.setProperty(Constants.PROPERTY_RATE, new Double(getRate()));
    out.setProperty(Constants.FIRST_PAID_RATIO, new Double(getFirstPaidRatio()));

    try {
        Thread.sleep((int) (Math.random() * 10) * 10);
    } catch (InterruptedException e) {
    }
    return true;
}
}
```

例程 7-11 显示了程序中用到的一些常量。

例程 7-11 Constants.java

```
package houseloan;
public interface Constants {

    String LOANBROKER_NS = "http://servicemix.org/demos/loan-broker";

    String LOANBROKER_SERVICE = "loan-broker";
    String HOUSELOANAGENCY_SERVICE = "houseloan-agency";
    String LENDERGATEWAY_SERVICE = "lender-gateway";
    String PROPERTY_RECIPIENTS = "recipients";
    String PROPERTY_CORRELATIONID = "correlationId";
    String PROPERTY_RATE = "rate";
    String FIRST_PAID_RATIO = "firstPaidRatio";
    String NAME = "name";
    String HOUSE_NUMBER = "house_number";
    String PROPERTY_BANK = "bank";

}
```

### 7.8.3 配置 SU、SA、jbi.xml、servicemix.xml

本小节的实例不同于以前的实例，以前的实例只有一个简单的 servicemix.xml 的配置，相对来说比较简单。对于比较复杂的应用，往往需要将一个服务模块分成一些运行单元，有的单元运行在 JMS 环境中，有的单元运行在 lwcontainer 环境中等，这些小的运行单元称为 SU (Service Unit) “服务单元”。这些小的运行单元并不是分别发布运行的，而是集中在 SA (Service Assembly) “服务集成模块”下面一起运行，最后发布的是 SA。如图 7-9 所示。

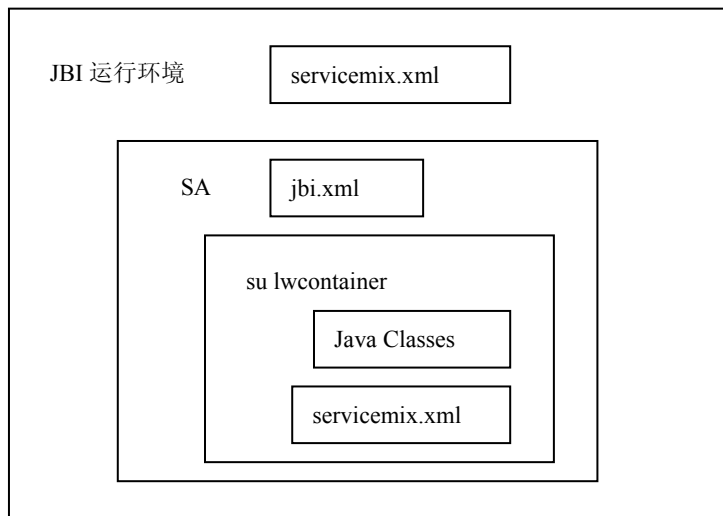


图 7-9 ServiceMix 的 SA 运行模块

ServiceMix 可以提供不同的运行环境的支持。本小节将会运行 lwcontainer，目录结构如下：

```
House-loan
|--build.xml、servicemix.xml
|--build   (包括编译之后的 Classes 文件)
|--data    (运行 ServiceMix 后所产生的一些运行文件，运行之前可以清理掉)
|--deploy  (运行 ServiceMix 所需要的 zip 包，发布文件夹)
|   |--loanbroker-sa.zip
|--install
|   |--servicemix-lwcontainer-3.1-incubating-installer (运行 lwcontainer 所需要的)
|   |--servicemix-shared-3.1-incubating-installer (运行 ServiceMix 的共享库)
|--src     (Java 源程序和配置文件)
|   |--client (客户端源程序)
|   |--components 服务器端源程序，将会被发布到服务总线
|   |--sa
|       |--META-INF
|       |--jbi.xml (JBI 配置文件)
|       |--su
|--servicemix.xml
```

本例将会创建 3 个配置文件。

- SA 目录下面的 jbi.xml;
- SU 目录下面的配置文件 servimix.xml;
- 根目录下面的 servicemix.xml。

#### 1. SA 目录下面的 jbi.xml

SA 目录下面的配置文件 jbi.xml 负责配置运行在 ServiceMix 运行环境中的 SA (Service Assembly) “服务集成模块”。

一个运行模块只能有一个 SA, 可以有多个 SU。每个 SU 将会打包成一个单独的 SU 包, 然后将多个 SU 的 zip 包打成一个 SA 包来运行。

本小节的 jbi.xml 内容如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <service-assembly>
    <identification>
      <name>loanbroker</name>
      <description>LoanBroker Service Assembly</description>
    </identification>
    <service-unit>
      <identification>
        <name>loanbroker</name>
        <description>LoanBroker Service Unit</description>
      </identification>
      <target>
        <artifacts-zip>loanbroker-su.zip</artifacts-zip>
        <component-name>servicemix-lwcontainer</component-name>
      </target>
    </service-unit>
  </service-assembly>
</jbi>
```

jbi.xml 表示将会通过 servicemix-lwcontainer 容器来运行 loanbroker-su.zip。

本小节有两个 servicemix.xml, 通过 SU 目录下面的配置文件 servimix.xml 配置服务单元。此外通过项目根目录下面的 servicemix.xml 来运行服务。

#### 2. SU 目录下面的配置文件 servimix.xml

SU 目录下面的配置文件 servimix.xml 配置服务单元。

本小节将会建立 1 个 JMS 绑定组件, 7 个服务引擎组件。

- JMS 绑定组件: 负责接收外部的服务请求, 并将服务请求通过 NMR 送给 HouseLoanBroker 服务引擎;
- HouseLoanBroker 服务引擎: 主要的业务流程控制模块, 它接收外部服务请求者的请求消息后, 首先调用 HouseLoanAgency 服务, 通过客户的姓名得到客户的房屋数量; 再调用 HouseLoanGateway 服务;
- HouseLoanAgency 服务引擎: 从 HouseLoanBroker 得到服务请求, 根据顾客的姓名, 得到不同的房屋数量;
- HouseLoanGateway 服务引擎: 根据客户房屋数量的不同分别调用 Bank0、Bank1、



Bank2、Bank3 的服务；

- Bank0 服务引擎：提供房屋数量为 0 的客户的房屋贷款，提供相应的首付比率和贷款利率；
- Bank1 服务引擎：提供房屋数量为 1 的客户的房屋贷款，提供相应的首付比率和贷款利率；
- Bank2 服务引擎：提供房屋数量为 2 的客户的房屋贷款，提供相应的首付比率和贷款利率；
- Bank3 服务引擎：提供房屋数量大于 2 的客户的房屋贷款，提供相应的首付比率和贷款利率。

下面是 servicemix.xml 开始部分，定义命名空间：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:lb="http://servicemix.org/demos/loan-broker">
  <classpath>
    <location>./</location>
  </classpath>
  <sm:serviceunit id="jbi">
    <sm:activationSpecs>
```

建立 JMS 绑定组件，接收外部的服务请求。

- componentName：组件名称，为“loanBrokerJmsBinding”；
- destinationService：目标服务，JMS 绑定组件在收到服务请求消息后，会将消息通过 NMR 转给目标服务，目标服务的写法为“命名空间：服务名称”，此处为“lb:loan-broker”；
- JMS 的实现类：“org.apache.servicemix.components.jms.JmsServiceComponent”；
- 实现类的属性名称：“template”，它所对应的 Bean 的类名为“org.springframework.jms.core.JmsTemplate”，其属性“connectionFactory”（连接工厂），引用了下面配置的“jmsFactory”；其属性“defaultDestinationName”（默认目标名称），对应于“demo.org.servicemix.source”；其属性“pubSubDomain”的值为“false”，表示没有订阅的工作区域。

```
<!-- In/out binding 配置 JMS 的监听端口，来监听服务请求 -->
<sm:activationSpec componentName="loanBrokerJmsBinding"
                  destinationService="lb:loan-broker">
  <sm:component>
    <bean class="org.apache.servicemix.components.jms.JmsServiceComponent">
      <property name="template">
        <bean class="org.springframework.jms.core.JmsTemplate">
          <property name="connectionFactory" ref="jmsFactory" />
          <property name="defaultDestinationName" value="
            demo.org.servicemix.source" />
          <property name="pubSubDomain" value="false" />
        </bean>
      </property>
```

## 第7章 基于 JBI 的 ServiceMix 服务总线

```
</bean>
</sm:component>
</sm:activationSpec>

<!-- Loan broker component
配置 loanBroker 服务，它专门接收各种服务请求，然后分发给不同的服务，最后将响应返回给服务请求者。它实现的是一个流程管理的功能，类似于 BPEL 的功能 -->
<sm:activationSpec componentName="loanBroker"
                    service="lb:loan-broker">

    <sm:component>
        <bean class="houseloan.HouseLoanBroker" />
    </sm:component>
</sm:activationSpec>

<!-- HouseLoan Agency 下面配置 houseloan agency 服务，它用来查询顾客的已有的房屋贷款数量 -->
<sm:activationSpec componentName="houseloanAgency"
                    service="lb:houseloan-agency">

    <sm:component>
        <bean class="houseloan.HouseLoanAgency" />
    </sm:component>
</sm:activationSpec>

<!-- Lender Gateway 下面服务贷款服务控制器，它将根据顾客已有的房屋数量，将贷款服务分发给不同的银行，因为不同的银行将给出不同的首付比率，贷款利率 -->
<sm:activationSpec componentName="lenderGateway"
                    service="lb:lender-gateway">

    <sm:component>
        <bean class="houseloan.HouseLoanGateway" />
    </sm:component>
</sm:activationSpec>

<!-- Banks 下面配置不同的银行服务，因为不同的银行将会给出不同的收费比率、贷款利率-->
<sm:activationSpec componentName="bank0" service="lb:bank0">
    <sm:component><bean class="houseloan.Bank" >
        <property name="rate" value="6.0"/>
        <property name="firstPaidRatio" value="20.0"/>
    </bean>
    </sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="bank1" service="lb:bank1">
    <sm:component><bean class="houseloan.Bank" >
        <property name="rate" value="6.5"/>
        <property name="firstPaidRatio" value="30.0"/>
    </bean>
    </sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="bank2" service="lb:bank2">
    <sm:component><bean class="houseloan.Bank" >
```

```

        <property name="rate" value="7.0"/>
        <property name="firstPaidRatio" value="40.0"/>
    </bean>
</sm:component>
</sm:activationSpec>
<sm:activationSpec componentName="bank3" service="lb:bank3">
    <sm:component><bean class="housetloan.Bank" >
        <property name="rate" value="7.5"/>
        <property name="firstPaidRatio" value="50.0"/>
    </bean>
    </sm:component>
</sm:activationSpec>

</sm:activationSpecs>
</sm:serviceunit>

<!-- 配置 JMS 的连接工厂 -->
<bean id="jmsFactory" class="org.apache.activemq.pool.PooledConnectionFactory">
    <property name="connectionFactory">
        <bean class="org.apache.activemq.ActiveMQConnectionFactory">
            <property name="brokerURL" value="tcp://localhost:61616" />
        </bean>
    </property>
</bean>
</beans>

```

### 3. 根目录下面的 servicemix.xml

根目录下面的 servicemix.xml，负责发布整个服务。

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0">

    <bean id="jndi" class="org.apache.xbean.spring.jndi.SpringInitialContextFactory"
        factory-method="makeInitialContext" singleton="true" />

    <!-- JMX 服务器 -->
    <sm:jmxServer id="jmxServer" locateExistingServerIfPossible="true" />

    <import resource="classpath:activemq.xml" />

    <!-- the JBI 容器 -->
    <sm:container id="jbi"
        rootDir="./data/smx"
        installationDirPath="./install"
        deploymentDirPath="./deploy"
        flowName="seda">

        <sm:activationSpecs>
        </sm:activationSpecs>
    </sm:container>
</beans>

```

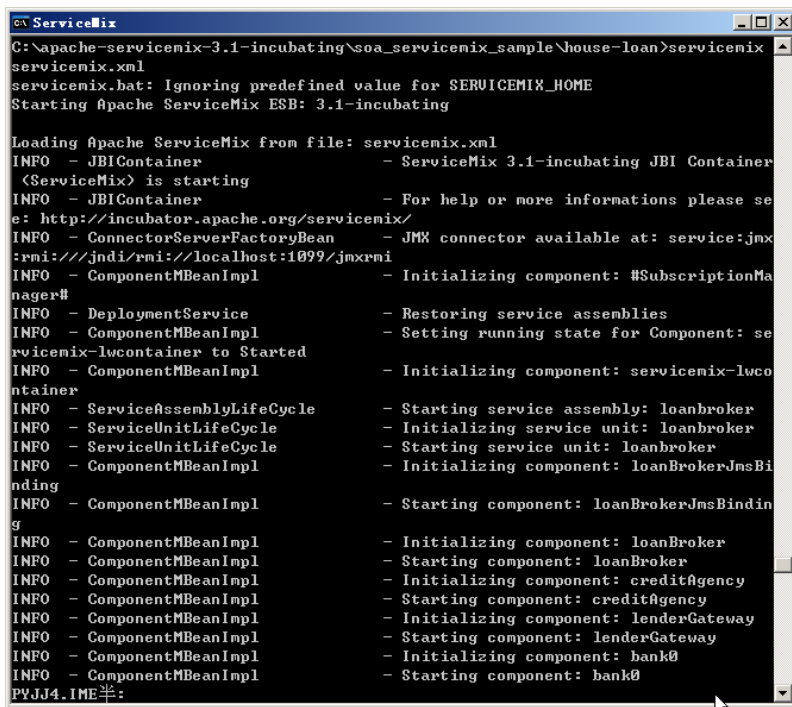
#### 7.8.4 创建和发布实例

运行本章配套光盘所提供的 Ant 脚本，将会生成一个 zip 包放在 deploy 目录下面。运行命令为“ant clean”，以及“ant setup”。

发布 zip 包的结构如下。

```
loanbroker-sa.zip
|---META-INF
|       |--- jbi.xml
|---loanbroker-su.zip
|       |---servicemix.xml
|       |---Java Classes
```

然后在项目根目录下直接运行命令“servicemix servicemix.xml”即可启动 ServiceMix 的服务。图 7-10 显示了 ServiceMix 运行房屋贷款服务的过程。



```
C:\apache-servicemix-3.1-incubating\soa_servicemix_sample\house-loan>servicemix
servicemix.xml
servicemix.bat: Ignoring predefined value for SERVICEMIX_HOME
Starting Apache ServiceMix ESB: 3.1-incubating

Loading Apache ServiceMix from file: servicemix.xml
INFO - JBIContainer - ServiceMix 3.1-incubating JBI Container
(ServiceMix) is starting
INFO - JBIContainer - For help or more informations please see: http://incubator.apache.org/servicemix/
INFO - ConnectorServerFactoryBean - JMX connector available at: service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
INFO - ComponentMBeanImpl - Initializing component: #SubscriptionManager#
INFO - DeploymentService - Restoring service assemblies
INFO - ComponentMBeanImpl - Setting running state for Component: servicemix-lwcontainer to Started
INFO - ComponentMBeanImpl - Initializing component: servicemix-lwcontainer
INFO - ServiceAssemblyLifeCycle - Starting service assembly: loanbroker
INFO - ServiceUnitLifeCycle - Initializing service unit: loanbroker
INFO - ServiceUnitLifeCycle - Starting service unit: loanbroker
INFO - ComponentMBeanImpl - Initializing component: loanBrokerJmsBinding
INFO - ComponentMBeanImpl - Starting component: loanBrokerJmsBinding
INFO - ComponentMBeanImpl - Initializing component: loanBroker
INFO - ComponentMBeanImpl - Starting component: loanBroker
INFO - ComponentMBeanImpl - Initializing component: creditAgency
INFO - ComponentMBeanImpl - Starting component: creditAgency
INFO - ComponentMBeanImpl - Initializing component: lenderGateway
INFO - ComponentMBeanImpl - Starting component: lenderGateway
INFO - ComponentMBeanImpl - Initializing component: bank0
INFO - ComponentMBeanImpl - Starting component: bank0
PYJJ4.IME半:
```

图 7-10 ServiceMix 运行房屋贷款服务

#### 7.8.5 创建客户端调用程序

客户端通过 JMS 来调用 Service 提供的服务，并发送一个 SOAP 的请求消息给 JMS 端口。例程 7-12 显示了客户端的调用程序，它包括如下的步骤。

- 应用 ActiveMQConnectionFactory 创建连接工厂；
- 应用连接工厂、进出队列创建 Requestor 的实例；
- 通过 Requestor 实例创建请求消息；
- 将客户信息置于请求消息的属性之中；

- 送出请求消息，得到响应消息；
- 通过响应消息的属性得到贷款利率、首付比例等。

例程 7-12 JMSClient.java

```
import edu.emory.mathcs.backport.java.util.concurrent.CountDownLatch;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQQueue;
import org.logicblaze.lingo.jms.Requestor;
import org.logicblaze.lingo.jms.JmsProducerConfig;
import org.logicblaze.lingo.jms.impl.MultiplexingRequestor;
import edu.emory.mathcs.backport.java.util.concurrent.ExecutorService;
import edu.emory.mathcs.backport.java.util.concurrent.Executors;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.Message;

public class JMSClient implements Runnable {

    private static ConnectionFactory factory;
    private static CountDownLatch latch;
    private static Requestor requestor;

    public static void main(String[] args) throws Exception {
        System.out.println("Connecting to JMS server.");

        //创建 JMS 连接
        factory = new ActiveMQConnectionFactory("tcp://localhost:61616");
        Destination inQueue = new ActiveMQQueue("demo.org.servicemix.source");
        Destination outQueue = new ActiveMQQueue("demo.org.servicemix.output" +
                                                    (int)(1000*Math.random()));
        requestor = MultiplexingRequestor.newInstance(factory, new JmsProducerConfig(),
                                                    inQueue, outQueue);

        if (args.length == 0) {
            new JMSClient().run();
        } else {
            int nb = Integer.parseInt(args[0]);
            int th = 30;
            if (args.length > 1) {
                th = Integer.parseInt(args[1]);
            }
            latch = new CountDownLatch(nb);
            ExecutorService threadPool = Executors.newFixedThreadPool(th);
            for (int i = 0; i < nb; i++) {
                threadPool.submit(new JMSClient());
            }
            latch.await();
        }
        System.out.println("Closing.");
    }
}
```

```
        requestor.close();
    }

    public void run() {
        try {
            System.out.println("Sending request.");
            //将请求消息参数值设置到请求消息的属性中，再传到服务器端
            Message out = requestor.getSession().createMapMessage();
            //将客户姓名赋值于请求消息的属性之中
            out.setStringProperty("name", "Li Si");
            Message in = requestor.request(null, out);
            if (in == null) {
                System.out.println("Response timed out.");
            }
            else {
                System.out.println("Response was: rate=" + in.getDoubleProperty("rate") +
                    "% and first paid ratio=" + in.getDoubleProperty("firstPaidRatio")+
                    " from " + in.getStringProperty("bank"));
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (latch != null) {
                latch.countDown();
            }
        }
    }
}
```

直接运行 Ant，因为本文代码为 `out.setStringProperty("name", "Li Si")`，所以有：

```
[echo] Running exsample client
[java] Connecting to JMS server.
[java] Sending request.
[java] Response was: rate=7.0% and first paid ratio=40.0 from bank2
[java] Closing.
```

因为服务器端是采用的 JMS 的监听端口，所以这里采用 JMS 的客户端来调用。

## 7.9 创建BPEL房屋贷款实例

从上面的实例，可以看到 ServiceMix 可以通过 Java 编码来处理业务流程。但是根据 SOA 的基本思想，应该将业务流程和基本服务分离，业务流程最好交给流程管理器 BPEL 来负责处理。

这里介绍如何通过编写目前主流的 BPEL 来实现与 7.8 节同样的功能。所有与流程控制有关的地方都由 BPEL 来实现，只保留银行贷款业务和查询顾客房屋数量的 Java 模块。

### 7.9.1 流程图

只需要输入用户姓名，这个系统会首先自动查出客户的房屋数量，然后自动转到相应的银行进行房屋贷款服务，客户最后可以得到对应于他的目前已有房屋数量的贷款首付和贷款利率。具体过程如下。

(1) BPEL 将首先以客户姓名 `name` 作为输入变量，调用 `HouseLoanAgency` 的 `Web Service`，得到客户的目前拥有的房屋数量；

(2) 如果客户房屋数量为 0，BPEL 将调用 `Bank0` 的服务；如果客户房屋数量为 1，BPEL 将调用 `Bank1` 的服务；如果客户房屋数量为 2，BPEL 将调用 `Bank2` 的服务；如果客户房屋数量大于 2，BPEL 将调用 `Bank3` 的服务；

(3) BPEL 将 `Bank` 返回的首付比率和贷款利率返回给服务请求者。

图 7-11 显示了 `ServiceMix` 的房屋贷款 BPEL 服务流程图。

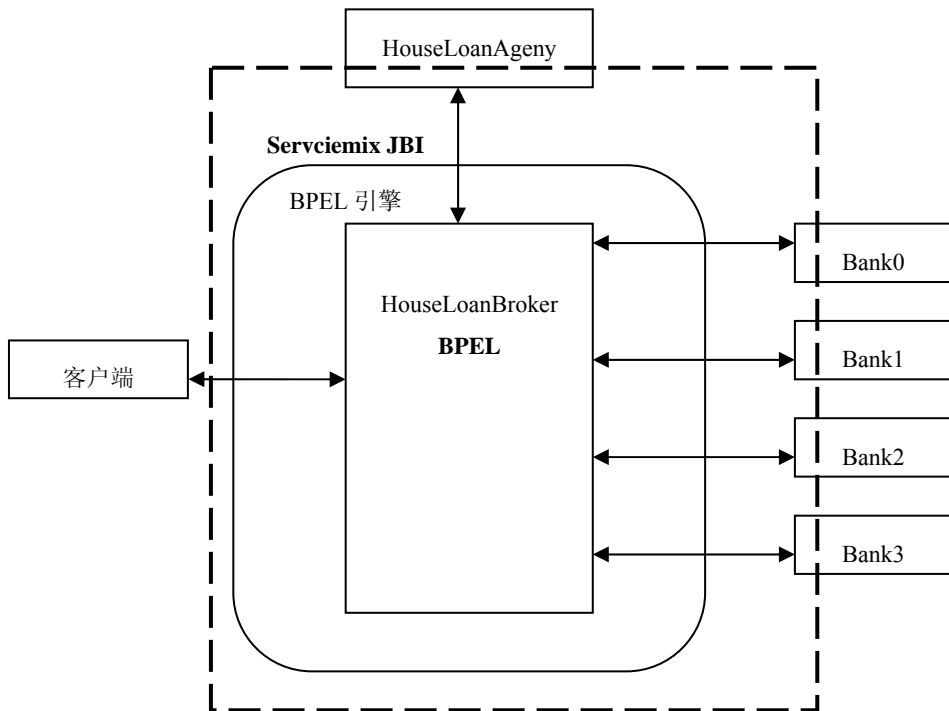


图 7-11 ServiceMix 创建房屋贷款 BPEL 服务流程图

在 `ServiceMix` 上创建房屋贷款应用 BPEL 服务包括下面的过程。

- 创建 BPEL 模块文件，包括 `housetloanbroker.bpel`、`Housetloanbroker.wsdl`、`Housetloanagency.wsdl`、`Bank.wsdl`。请详细参考 6.2 节。
- 创建服务器端的 Java 程序；
- 创建 `Servicemix.xml` 配置文件，包括配置 SA（Service Assembly）和 SU（Service Unit）来配置服务模块；
- 发布 `ServiceMix` 服务，创建客户端调用程序。

下面将分别予以介绍。



### 7.9.2 服务器端程序

因为本小节采用 BPEL 流程管理服务，所有与流程相关的模块都已经被 BPEL 取代，只剩下 2 个基本的服务模块。一个是 HouseLoanAgency.java，另一个是 Bank.java。

例程 7-13 显示了 HouseLoanAgency.java 的代码，其主要功能就在于通过请求消息得到客户姓名“name”后，通过“name”查询得到用户的房屋数量，并创建相应的返回消息；如果系统中没有该客户姓名，则返回一个错误信息，此错误信息将会被 BPEL 运行环境截获，并被返回给服务请求者。其中主要用到了 XML 技术对消息交换（Message Exchange）的请求消息进行解析。用户的房屋数量信息存在一个数组里面，实际项目中用户房屋数量信息应该存在数据库中。

例程 7-13 HouseLoanAgency.java

```
package houseloanbroker;
...省略 import
public class HouseLoanAgency extends ComponentSupport implements MessageExchangeListener {

    //创建 HouseLoanAgency 服务
    public HouseLoanAgency() {
        setService(new QName("urn:sample:soa:houseloanagency", "HouseLoanAgencyService"));
        setEndpoint("houseLoanAgency");
    }

    //创建客户房屋数量信息
    private static String[][] loanStatusList = {
        {"Zhang San", "0"},
        {"Li Si", "1"},
        {"Wang Wu", "2"},
        {"Zhao Liu", "3"}
    };

    //onMessageExchange 不停地监听请求消息
    public void onMessageExchange(MessageExchange exchange) throws MessagingException {
        //接收到消息交换（请求—响应类型）
        InOut inOut = (InOut) exchange;

        //如果消息交换的状态为“done”，则返回
        if (inOut.getStatus() == ExchangeStatus.DONE) {
            return;
        }
        //如果消息交换的状态为“error”，也返回
        } else if (inOut.getStatus() == ExchangeStatus.ERROR) {
            return;
        }
        try {
            //将消息交换的输入信息转换为 XML 的 document 对象
            Document doc = (Document) new SourceTransformer().toDOMNode(inOut.getInMessage());

            //得到节点为“name”的节点值
            String name = textValueOfXPath(doc, "/*[local-name()='name']");
```



```

String operation = null;
if (inOut.getOperation() != null) {
    operation = inOut.getOperation().getLocalPart();
} else {
    operation = doc.getDocumentElement().getLocalName();
}
String output;
//如果请求消息中的操作为“getHouseNumber”，那么创建返回带有
//客户房屋数量的返回消息“output”
If ("getHouseNumber".equals(operation)) {
    output = "<getHouseNumberResponse
        xmlns=\"urn:sample:soa:houseloanagency\"><housenumber>\" +
        getHouseNumber(name) + \"</housenumber>
        </getHouseNumberResponse>\";
} else {
    throw new UnsupportedOperationException(operation);
}
//如果查询的房屋数量为-1，表示系统中没有该客户，
//那么返回没有该客户的错误信息
If (getHouseNumber(name) == -1)
{
    Fault fault = inOut.createFault();
    fault.setContent(new StringSource("<InvalidNAME
        xmlns=\"urn:sample:soa:houseloanagency\">
        <name>\" + name + \"</name></InvalidNAME>\"));
    fail(inOut, fault);
} else
{
    //将正常的返回消息“output”返回给服务请求者
    NormalizedMessage answer = inOut.createMessage();
    answer.setContent(new StringSource(output));
    answer(inOut, answer);
}

} catch (Exception e) {
    throw new MessagingException(e);
}
}

//通过姓名“name”得到房屋数量
private int getHouseNumber(String name)    {
    int houseNumber = -1;
    for (int I=0; I<loanStatusList.length; I++){
        String[] item = loanStatusList[I];
        String itemName = item[0];
        int itemNum = Integer.parseInt(item[1]);

        if (name.equalsIgnoreCase(itemName))
    
```

```
        {
            houseNumber = itemNum;
            break;
        }
    }

    return houseNumber;
}

//通过 xpath 得到节点值
protected String textValueOfXPath(Node node, String xpath) throws TransformerException {
    CachedXPathAPI cachedXPathAPI = new CachedXPathAPI();
    NodeIterator iterator = cachedXPathAPI.selectNodeIterator(node, xpath);
    Node root = iterator.nextNode();
    if (root instanceof Element) {
        Element element = (Element) root;
        if (element == null) {
            return "";
        }
        String text = DOMUtil.getElementText(element);
        return text;
    }
    else if (root != null) {
        return root.getNodeValue();
    }
    else {
        return null;
    }
}
}
```

例程 7-14 显示了 Bank 服务的代码，它将返回贷款利率和首付比率的服务请求。比较关键之处是 Bank.java 继承了 ServiceMix 的 ComponentSupport 类，并实现了 ServiceMix 的 MessageExchangeListener 接口。通过 onMessageExchange 方法不停地监听来自 NMR 的消息，并将带有贷款利率和首付比率的响应消息返回给服务请求者，贷款利率和首付比率是在 servicemix.xml 文件中配置的。这里和 7.8 节的不同之处是 7.8 节是将贷款利率和首付比例置于消息的属性中送出，这里是直接创建带有贷款利率和首付比率的消息。

例程 7-14 Bank.java

```
package loanbroker;
import javax.jbi.messaging.ExchangeStatus;
import javax.jbi.messaging.InOut;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.MessagingException;
import javax.jbi.messaging.NormalizedMessage;
import javax.xml.namespace.Qname;
import org.apache.servicemix.MessageExchangeListener;
import org.apache.servicemix.components.util.ComponentSupport;
import org.apache.servicemix.jbi.jaxp.StringSource;
```

```

public class Bank extends ComponentSupport implements MessageExchangeListener {

    private double rate;
    private double firstPaidRatio;

    //创建 Bank 服务
    public Bank(int number) {
        setService(new QName("urn:logicblaze:soa:bank", "Bank" + number));
        setEndpoint("bank");
    }

    //监听请求消息，并将带有贷款利率和首付比率的响应消息返回给服务请求者，贷款利率
    //和首付比率是在 servicemix.xml 文件中配置的
    public void onMessageExchange(MessageExchange exchange) throws MessagingException {

        InOut inOut = (InOut) exchange;
        if (inOut.getStatus() == ExchangeStatus.DONE) {
            return;
        } else if (inOut.getStatus() == ExchangeStatus.ERROR) {
            return;
        }
        System.err.println(getService().getLocalPart() + " requested");
        try {
            String output = "<getLoanQuoteResponse xmlns='urn:logicblaze:soa:bank'><rate>"
                + getRate() + "</rate><firstpaidratio>" + getFirstPaidRatio() + "</firstpaidratio>"
                + "</getLoanQuoteResponse>";
            NormalizedMessage answer = inOut.createMessage();
            answer.setContent(new StringSource(output));
            answer(inOut, answer);
        } catch (Exception e) {
            throw new MessagingException(e);
        }
    }

    public double getFirstPaidRatio() {
        return firstPaidRatio;
    }

    public void setFirstPaidRatio(double firstPaidRatio) {
        this.firstPaidRatio = firstPaidRatio;
    }

    public double getRate() {
        return rate;
    }

    public void setRate(double rate) {
        this.rate = rate;
    }
}

```

### 7.9.3 配置 SU、SA、jbi.xml、servicemix.xml、xbean.xml

本小节的实例因为要运行 BPEL，所以配置更为复杂，包括配置 3 个 SU 单元，一个 SA。其中 3 个 SU 单元分别如下。

- su (jms): 通过 xbean.xml 创建 JMS 监听端口，接收外部服务请求消息；
- su (bpel): 创建房屋贷款 BPEL 流程管理服务，含有 houseloanbroker.bpel 文件，其中包括 Houseloanbroker.wsdl（为 BPEL 的接口文件），Houseloanagency.wsdl（房屋贷款中介的 WSDL），Bank.wsdl（银行业务的 WSDL）；
- su (lwcontainer): 创建基本服务 Houseloanagency 和 Bank。

图 7-12 显示了 ServiceMix 的房屋贷款 BPEL 运行模块的配置。

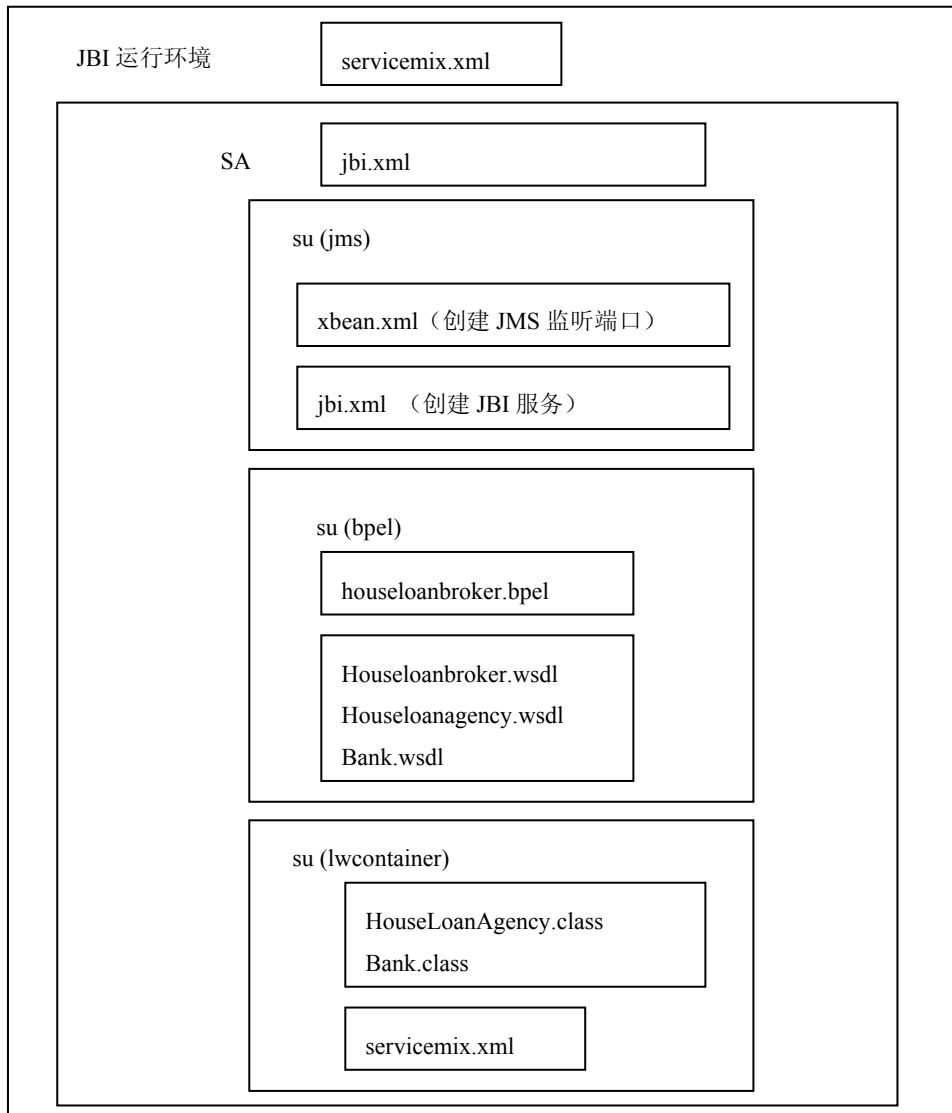


图 7-12 ServiceMix 的房屋贷款 BPEL 运行模块

开发的结构如下。

```
Hosue-loan-bpel
|--build.xml, servicemix.xml
|--build    (包括编译之后的 Classes 文件)
|--data     (运行 ServiceMix 后所产生的一些运行文件, 运行之前可以清理掉)
|--deploy   (运行 ServiceMix 所需要的 zip 包, 发布文件夹)
|   |--loan-broker-sa-3.1-incubating
|--install
|   |--servicemix-bpe-3.1-incubating-installer.zip 运行 BPEL 的库
|   |--servicemix-jms-3.1-incubating-installer.zip 运行 JMS 的库
|--servicemix-lwcontainer-3.1-incubating-installer (运行 lwcontainer 所需要的)
|   |--servicemix-shared-3.1-incubating-installer (运行 ServiceMix 的共享库)
|--src (Java 源程序和配置文件)
|   |--client (客户端 Java 源程序)
|   |--components 服务器端源程序, 将会被发布到服务总线
|   |--bpe-su
|       |--bank.wsdl、creditagency、loanbroker、loanbroker.bpel
|       |--META-INF
|           |--jbi.xml
|   |--jms-su
|       |--xbean.xml
|       |--META-INF
|           |--jbi.xml (JBI 配置文件)
|   |--lw-su
|       |--servicemix.xml
|       |--lib
|           |--servicemix-components-3.1-incubating.zip
|           |--META-INF
|           |--jbi.xml
|   |--sa
|       |--META-INF
|       |--jbi.xml
```

本例将会创建下面的配置文件。

- SA 目录下面的 jbi.xml, 配置整个服务的 SA 服务集成模块;
- lw-su 目录配置下的 servicemix.xml、jbi.xml, 主要用来运行 Java 服务;
- jms-su 目录下面的 xbean.xml、jbi.xml, 主要用来配置 JMS 的监听端口;
- bpe-su 目录下面的 jbi.xml, 主要用来运行 BPEL 服务;
- 根目录下面的 servicemix.xml。

#### 1. SA 目录下面的 jbi.xml

SA 目录下面的配置文件 jbi.xml 负责配置运行在 ServiceMix 运行环境中的 SA(Service Assembly)“服务集成模块”。

整个运行模块只能有一个 SA, 可以有多个 SU。每个 SU 将会打包成一个单独的 SU 包, 然后将多个 SU 的 zip 包打成一个 SA 包来运行, 本 SA 将运行 3 个 SU 的运行单元, 所创建的 jbi.xml 文件内容如下:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <service-assembly>
    <identification>
      <name>loan-broker-sa</name>
      <description>ServiceMix :: Samples :: Loan Broker :: SA</description>
    </identification>
    <service-unit>
      <identification>
        <name>loan-broker-bpe-su</name>
        <description>ServiceMix is an open source ESB based on the Java Business
        Integration framework - JSR-208</description>
      </identification>
      <target>
        <artifacts-zip>loan-broker-bpe-su-3.1-incubating.zip</artifacts-zip>
        <component-name>servicemix-bpe</component-name>
      </target>
    </service-unit>
    <service-unit>
      <identification>
        <name>loan-broker-lw-su</name>
        <description>ServiceMix is an open source ESB based on the Java Business
        Integration framework - JSR-208</description>
      </identification>
      <target>
        <artifacts-zip>loan-broker-lw-su-3.1-incubating.zip</artifacts-zip>
        <component-name>servicemix-lwcontainer</component-name>
      </target>
    </service-unit>
    <service-unit>
      <identification>
        <name>loan-broker-jms-su</name>
        <description>ServiceMix is an open source ESB based on the Java Business
        Integration framework - JSR-208</description>
      </identification>
      <target>
        <artifacts-zip>loan-broker-jms-su-3.1-incubating.zip</artifacts-zip>
        <component-name>servicemix-jms</component-name>
      </target>
    </service-unit>
  </service-assembly>
</jbi>
```

## 2. lw-su 目录下的 servimix.xml、jbi.xml

lw-su 目录下面的 servimix.xml 负责配置服务单元。jbi.xml 负责配置 JBI 运行环境。

lw 是 light weight 的缩写，表示服务将会运行在 Service 开发的轻量级运行环境 lwcontainer 中。

## 3. lw-su 目录下面的 servimix.xml

w-su 目录下面的配置 servimix.xml 负责创建基本服务 HouseLoanAgency 和 4 个 Bank

服务。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:lb="urn:logicblaze:soa:loanbroker"
       xmlns:ca="urn:logicblaze:soa:houseloanagency">

  <!-- 创建房屋贷款服务 -->
  <sm:serviceunit id="jbi">
    <sm:activationSpec>

      <sm:activationSpec id="houseLoanAgency"
                        interfaceName="ca:HouseLoanAgency">
        <sm:component>
          <bean class="loanbroker.HouseLoanAgency" />
        </sm:component>
      </sm:activationSpec>

    <!-- 创建 Bank 服务，配置银行贷款利率和贷款首付 -->
    <sm:activationSpec id="bank0">
      <sm:component>
        <bean class="loanbroker.Bank">
          <constructor-arg value="0" />
          <property name="rate" value="6.0"/>
          <property name="firstPaidRatio" value="20.0"/>
        </bean>
      </sm:component>
    </sm:activationSpec>

    <sm:activationSpec id="bank1">
      <sm:component>
        <bean class="loanbroker.Bank">
          <constructor-arg value="1" />
          <property name="rate" value="6.5"/>
          <property name="firstPaidRatio" value="30.0"/>
        </bean>
      </sm:component>
    </sm:activationSpec>

    <sm:activationSpec id="bank2">
      <sm:component>
        <bean class="loanbroker.Bank">
          <constructor-arg value="2" />
          <property name="rate" value="7.0"/>
          <property name="firstPaidRatio" value="40.0"/>
        </bean>
      </sm:component>
    </sm:activationSpec>

    <sm:activationSpec id="bank3">
      <sm:component>
```

```
<bean class="loanbroker.Bank">
  <constructor-arg value="3" />
  <property name="rate" value="7.5"/>
  <property name="firstPaidRatio" value="50.0"/>
</bean>
</sm:component>
</sm:activationSpec>

</sm:activationSpecs>
</sm:serviceunit>
</beans>
```

#### 4. lw-su 目录下面的配置 jbi.xml

jbi.xml 服务配置基本服务的 JBI 运行环境。

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <services binding-component="false"/>
</jbi>
```

#### 5. jms-su 目录下面的 xbean.xml、jbi.xml

jms-su 目录下面的 xbean.xml 配置 JMS 的运行环境，jbi.xml 负责配置 JBI 运行环境。

##### (1) jms-su 目录下面的 xbean.xml

jms-su 目录下面的 xbean.xml 主要用来配置 JMS 的端口来监听各种服务请求。定义服务器端口“endpoint”、服务名称“lb:LoanBrokerService”等。定义 defaultMep 默认的消息模式，这里采用的是 in-out 模式。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
  xmlns:lb="urn:logicblaze:soa:loanbroker">

  <jms:endpoint service="lb:LoanBrokerService"
    endpoint="endpoint"
    targetService="lb:LoanBrokerService"
    defaultOperation="lb:getLoanQuote"
    role="consumer"
    connectionFactory="#jmsFactory"
    destinationStyle="queue"
    jmsProviderDestinationName="demo.org.servicemix.source"
    defaultMep="http://www.w3.org/2004/08/wsdl/in-out" />

  <bean id="jmsFactory" class="org.apache.activemq.pool.PooledConnectionFactory">
    <property name="connectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="tcp://localhost:61616" />
      </bean>
    </property>
  </bean>
</beans>
```

##### (2) jms-su 目录下面的 jbi.xml



jbi.xml 配置 JMS 的 JBI 运行环境。

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <services binding-component="false" xmlns:lb="urn:logicblaze:soa:loanbroker">
    <consumes service-name="lb:LoanBrokerService"/>
  </services>
</jbi>
```

#### 6. bpe-su 目录下面的 jbi.xml

bpe-su 目录下面的 jbi.xml 负责配置 BPEL 的运行环境。BPEL 的 houseloanbroker.bpel、Houseloanbroker.wsdl、Houseloanagency.wsdl、Bank.wsdl 已经在第 6 章做了详细的介绍。下面的 jbi.xml 配置 JBI 的运行环境。

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <services binding-component="false"/>
</jbi>
```

#### 7. 根目录下面的 servicemix.xml

根目录下面的 servicemix.xml 用来运行整个房屋贷款服务 SA，事实上是一个通用的 servicemix.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0">

  <bean id="jndi" class="org.apache.xbean.spring.jndi.SpringInitialContextFactory"
        factory-method="makeInitialContext" singleton="true" />

  <!-- JMX 服务器-->
  <sm:jmxServer id="jmxServer" locateExistingServerIfPossible="true" />

  <import resource="classpath:activemq.xml" />

  <!-- JBI 容器 -->
  <sm:container id="jbi"
        rootDir="./data/smx"
        installationDirPath="./install"
        deploymentDirPath="./deploy"
        flowName="seda">
    <sm:activationSpecs>
    </sm:activationSpecs>
  </sm:container>
</beans>
```

### 7.9.4 创建和发布实例

运行本章所提供的 ANT 脚本，将会生成一个 zip 包放在 deploy 目录下面，运行命令为“ant clean”、“ant setup”，所生成的 jar 包的具体结构如下。

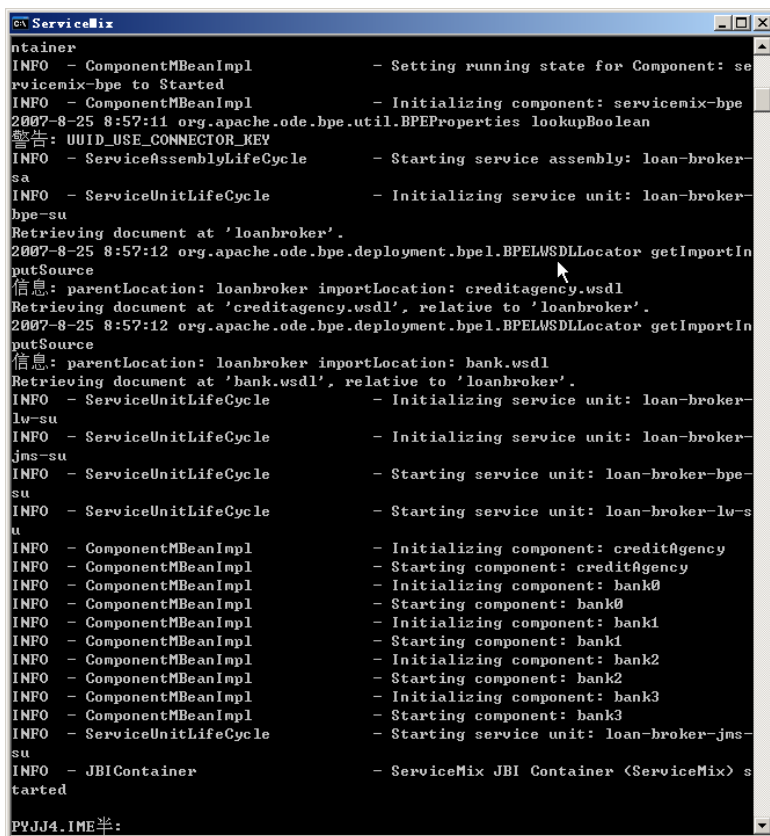
```
loan-broker-sa-3.1-incubating.zip
```

## 第 7 章 基于 JBI 的 ServiceMix 服务总线

```
|---META-INF
      |--- jbi.xml
|---loan-broker-bpe-su-3.1-incubating.zip    (创建 BPEL 流程管理服务)
      |---META-INF
      |--- jbi.xml
      |---bank.wsdl、creditagency、loanbroker、loanbroker.bpel
|---loan-broker-jms-su-3.1-incubating.zip
      |---xbean.xml    (发布 JMS 端口，监听请求)
      |---META-INF
      |--- jbi.xml
|---loan-broker-lw-su-3.1-incubating.zip
      |---servicemix.xml    (发布 Bank 服务、房屋数量服务)
      |---lib
      |---servicemix-components-3.1-incubating.zip
|---META-INF
      |--- jbi.xml
      |--- Java Classes
```

也就是说，只要形成了上面的 zip 包，里面的配置文件写对了，就可以运行了（不论是采用 ANT 脚本，或者采用 maven 脚本，或者手工打包成一个压缩文件），然后在本应用的根目录下面直接运行 `servicemix servicemix.xml` 即可。

图 7-13 显示了 ServiceMix 运行 BPEL 房屋贷款服务的过程。



```
ntainer
INFO - ComponentMBeanImpl - Setting running state for Component: se
rvicemix-hpe to Started
INFO - ComponentMBeanImpl - Initializing component: servicemix-hpe
2007-8-25 8:57:11 org.apache.ode.bpel.util.BPEProperties lookupBoolean
警告: UUID_USE_CONNECTOR_KEY
INFO - ServiceAssemblyLifeCycle - Starting service assembly: loan-broker-
sa
INFO - ServiceUnitLifeCycle - Initializing service unit: loan-broker-
hpe-su
Retrieving document at 'loanbroker'.
2007-8-25 8:57:12 org.apache.ode.bpel.deployment.bpel.BPELWSDLLocator getImportIn
putSource
信息: parentLocation: loanbroker importLocation: creditagency.wsdl
Retrieving document at 'creditagency.wsdl', relative to 'loanbroker'.
2007-8-25 8:57:12 org.apache.ode.bpel.deployment.bpel.BPELWSDLLocator getImportIn
putSource
信息: parentLocation: loanbroker importLocation: bank.wsdl
Retrieving document at 'bank.wsdl', relative to 'loanbroker'.
INFO - ServiceUnitLifeCycle - Initializing service unit: loan-broker-
lw-su
INFO - ServiceUnitLifeCycle - Initializing service unit: loan-broker-
jms-su
INFO - ServiceUnitLifeCycle - Starting service unit: loan-broker-hpe-
su
INFO - ServiceUnitLifeCycle - Starting service unit: loan-broker-lw-s
u
INFO - ComponentMBeanImpl - Initializing component: creditAgency
INFO - ComponentMBeanImpl - Starting component: creditAgency
INFO - ComponentMBeanImpl - Initializing component: bank0
INFO - ComponentMBeanImpl - Starting component: bank0
INFO - ComponentMBeanImpl - Initializing component: bank1
INFO - ComponentMBeanImpl - Starting component: bank1
INFO - ComponentMBeanImpl - Initializing component: bank2
INFO - ComponentMBeanImpl - Starting component: bank2
INFO - ComponentMBeanImpl - Initializing component: bank3
INFO - ComponentMBeanImpl - Starting component: bank3
INFO - ServiceUnitLifeCycle - Starting service unit: loan-broker-jms-
su
INFO - JBIContainer - ServiceMix JBI Container <ServiceMix> s
tarted
PVJJ4.IME半:
```

图 7-13 ServiceMix 运行 BPEL 房屋贷款服务

### 7.9.5 客户端调用程序

客户端通过 JMS 来调用 Service 提供的服务,和 7.8 节的不同之处在于本例是通过客户端发送 SOAP 请求消息来调用的,因为 BPEL 都是集成的 Web Service。例程 7-15 显示了客户端的调用程序,它包括如下的步骤。

- 应用 ActiveMQConnectionFactory 创建连接工厂;
- 应用连接工厂、进出队列创建 Requestor 的实例;
- 通过 Requestor 实例创建请求消息;
- 建立 JMS 连接;
- 创建 SOAP 请求消息;
- 发送并接收 SOAP 消息。

例程 7-15 JMSClient.java

```
import edu.emory.mathcs.backport.java.util.concurrent.CountDownLatch;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQQueue;
import org.logicblaze.lingo.jms.Requestor;
```

## 第 7 章 基于 JBI 的 ServiceMix 服务总线

```
import org.logicblaze.lingo.jms.JmsProducerConfig;
import org.logicblaze.lingo.jms.impl.MultiplexingRequestor;
import edu.emory.mathcs.backport.java.util.concurrent.ExecutorService;
import edu.emory.mathcs.backport.java.util.concurrent.Executors;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.Message;
import javax.jms.TextMessage;

public class JMSSClient implements Runnable {

    private static ConnectionFactory factory;
    private static CountDownLatch latch;
    private static Requestor requestor;

    public static void main(String[] args) throws Exception {
        System.out.println("Connecting to JMS server.");
        //创建 JMS 连接
        factory = new ActiveMQConnectionFactory("tcp://localhost:61616");
        Destination inQueue = new ActiveMQQueue("demo.org.servicemix.source");
        Destination outQueue = new ActiveMQQueue("demo.org.servicemix.output"
            + (int)(1000*Math.random()));
        requestor = MultiplexingRequestor.newInstance(factory, new JmsProducerConfig(),
            inQueue, outQueue);

        if (args.length == 0) {
            new JMSSClient().run();
        } else {
            int nb = Integer.parseInt(args[0]);
            int th = 30;
            if (args.length > 1) {
                th = Integer.parseInt(args[1]);
            }
            latch = new CountDownLatch(nb);
            ExecutorService threadPool = Executors.newFixedThreadPool(th);
            for (int i = 0; i < nb; i++) {
                threadPool.submit(new JMSSClient());
            }
            latch.await();
        }
        System.out.println("Closing.");
        requestor.close();
    }

    public void run() {
        try {
```

```
        System.out.println("Sending request.");

        //创建 SOAP 请求消息
        String request =
            "<getLoanQuoteRequest xmlns='urn:logicblaze:soa:loanbroker'>" +
            "  <name>Zhang San</name>" +
            "</getLoanQuoteRequest>";
        TextMessage out = requestor.getSession().createTextMessage(request);

        //发送 SOAP 请求消息
        TextMessage in = (TextMessage) requestor.request(null, out);
        if (in == null) {
            System.out.println("Response timed out.");
        }
        else {
            System.out.println("Response was: " + in.getText());
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (latch != null) {
            latch.countDown();
        }
    }
}
```

通过运行本例配套光盘提供的 ANT 脚本，执行上面的客户端程序，将得到下面结果：

```
[java] Response was: <?xml version='1.0' encoding='UTF-8'?>
    <tns:getLoanQuoteResponse xmlns:tns="urn:logicblaze:soa:loanbroker">
        <tns:rate>6</tns:rate>
        <tns:firstpaidratio>20</tns:firstpaidratio>
    </tns:getLoanQuoteResponse>
```

可以看到返回的贷款利率为“6”（%），首付比率为“20”（%）。这是因为请求的姓名为“zhang san”，查出有 0 套住房，Bank0 来处理贷款请求。

如果将前面的服务请求的名字改为“Zhang San1”，将会得到下面的反应。说明是一个无效的名字。这是 BPEL 的异常机制的处理结果。

```
[java] Response was: <?xml version='1.0' encoding='UTF-8'?>
<InvalidNAME xmlns="urn:logicblaze:soa:creditagency">
    <name>Zhang San1</name>
</InvalidNAME>
```

## 7.10 小结

本章介绍了开源的基于 JBI 规范的服务总线 ServiceMix 的基本架构和主要功能。主要目的在于通过实例来介绍如何应用 ServiceMix 实现服务的集成。

首先介绍了如何通过 ServiceMix 来创建和发布 Web 服务实例，主要是通过 `<http:endpoint>` 来配置监听端口，通过 `<jsr181:endpoint>` 来配置 Web Service 的实现，包括在 `<jsr181:endpoint>` 中配置服务接口类和服务实现类。

接下来介绍了如何创建 SOAP 绑定来实现服务代理实例，它通过 `HttpConnector` 配置接收消息的绑定组件，通过 `SaajBinding` 配置调用外部服务的绑定组件。

进而介绍了创建 HTTP 绑定代理服务实例，它主要通过 `HttpInvoker` 来调用外部服务。

又介绍了集成并路由到不同的外部服务实例，主要通过一个 SE 组件在收到消息后通过解析 XML 的内容来实现不同的消息路由。

接着介绍集成并路由到不同服务的房屋贷款实例，所有的服务都是在 ServiceMix 中实现的，通过保存原始消息交换的 ID 来实现异步调用。

最后介绍了 BPEL 实现房屋贷款的实例，它的基本思想是业务流程通过 BPEL 来实现，基本服务调用外部服务。