JaxWs基於消息編程

博客分類： cxf

DispatchProviderCXF異步調用消息編程

JaxWs 基於消息編程

1 兩種消息模式

2 三種數據類型

2.1 Source

2.2 SOAPMessage

2.3 DataSource

3 服務端訪問底層信息

4 客戶端訪問底層消息

4.1 Dispatch的三種請求方式

5 統一入口

6 注意

6.1 MESSAGE和PAYLOAD的區別

通過SEI（Service Endpoint Interface）在服務端和客戶端進行操作時，我們是直接使用的對應實現類或代理類對象。表面上看我們是使用的對象在服務端和客戶端進行通訊，而實際上底層還是通過發送消息和解析消息進行的。有時候我們可能會希望或者需要直接訪問這些消息，這個時候我們就可以通過Provider和Dispatch來實現了。Provider是應用在服務端的，而Dispatch是應用在客戶端的。

1 兩種消息模式

在使用Provider和Dispatch時，我們可以使用兩種消息模式，MESSAGE和PAYLOAD。

當使用MESSAGE模式時我們可以訪問整個的消息，包括綁定的任何header和wrapper。而使用PAYLOAD模式時我們僅僅可以訪問payload的消息。如當我們的Dispatch在以PAYLOAD模式工作時，它只能訪問到返回的SOAPMessage的body部分，而binding層將處理任何綁定的header和wrapper。

2 三種數據類型

Provider和Dispatch在進行資訊傳遞時只能使用三種數據類型：

l javax.xml.transform.Source

l javax.xml.soap.SOAPMessage

l javax.activation.DataSource

2.1 Source

Source是一個介面，它持有一個XML文檔對象。每一個Source介面的實現類都提供了一系列的方法來訪問和操縱其持有的XML文檔的內容。Source介面的實現類有DOMSource、SAXSource和StreamSource等。

2.2 SOAPMessage

SOAPMessage是一個抽象類，使用SOAPMessage的時候需要滿足兩個條件：

第一：Provider實現類使用的是SOAP綁定，即SOAPBinding；

第二：Provider實現類使用的是MESSAGE Mode。

SOAPMessage持有一個SOAP消息。

2.3 DataSource

DataSource是一個介面，使用時需要滿足以下兩個條件：

第一：Provider實現類使用的是Http綁定，即HttpBinding；

第二：Provider實現使用的是MESSAGE Mode。

DataSource是對數據集合的抽象，在適當的時候可以通過InputStream和OutputStream的形式提供對該數據的訪問。其實現類有FileDataSource和URLDataSource。

3 服務端訪問底層信息

服務端訪問底層資訊是通過Provider介面進行的。通過實現Provider介面並且把實現類發佈為一個WebService，我們就可以在客戶端發起請求時訪問到其發送過來的底層消息對象，Source、SOAPMessage或者DataSource。Provider介面只定義了一個invoke方法，該方法接收一個消息對象，並返回一個同類型的消息對象，而且消息對象的類型只能是上面介紹的三種類型之一。

在使用Provider的時候我們需要在其實現類上使用@WebServiceProvider進行標記（使用@WebService標記好像也行），並且Provider<T>指定的消息對象類型必須是上面提到的三種數據類型之一。WebService使用的消息模式默認為PAYLOAD，我們可以在Provider實現類上使用@ServiceMode來指定其它值，如@ServiceMode(Service.Mode.MESSAGE)。

下面我們來看一個使用Provider的例子：

Java代碼 收藏代碼

import javax.xml.soap.SOAPMessage;

import javax.xml.ws.Provider;

import javax.xml.ws.Service.Mode;

import javax.xml.ws.ServiceMode;

import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(serviceName = "SOAPMessageService", portName = "SOAPMessagePort", targetNamespace = "http://provider.jaxws.sample.cxftest.tiantian.com/")

@ServiceMode(Mode.MESSAGE)

public class SOAPMessageModeProvider implements Provider<SOAPMessage> {

public SOAPMessage invoke(SOAPMessage request) {

SOAPMessage response = null;

try {

System.out.println("客戶端以SOAPMessage通過MESSAGE Mode請求如下： ");

request.writeTo(System.out);

response = MessageUtil.getInstance().create(null, "/provider/SOAPMessageResp.xml");

} catch (Exception ex) {

ex.printStackTrace();

}

return response;

}

}

在上面代碼中，我們的SOAPMessageModeProvider：

l 實現了Provider介面；

l 通過Provider介面定義的泛型指定使用的消息對象數據類型為SOAPMessage；

l 通過@WebServiceProvider標註其為一個WebService，並指定了serviceName等屬性；

l 通過@ServiceMode指定其使用的消息模式為MESSAGE；

l 在invoke方法中接收了一個SOAPMessage，並返回了一個SOAPMessage。

其中MessageUtil類的代碼為：

Java代碼 收藏代碼

import java.io.IOException;

import java.io.InputStream;

import javax.xml.soap.MessageFactory;

import javax.xml.soap.MimeHeaders;

import javax.xml.soap.SOAPException;

import javax.xml.soap.SOAPMessage;

public class MessageUtil {

private static MessageUtil instance = new MessageUtil();

private MessageFactory factory;

private MessageUtil() {

try {

factory = MessageFactory.newInstance();

} catch (SOAPException e) {

e.printStackTrace();

thrownew RuntimeException(e);

}

}

public static MessageUtil getInstance() {

returninstance;

}

/\*\*

\* 創建一個默認的SOAPMessage

\* @return

\* @throws SOAPException

\*/

public SOAPMessage create() throws SOAPException {

returnfactory.createMessage();

}

/\*\*

\* 根據MimeHeaders和soap格式檔路徑創建一個SOAPMessage

\* @param headers

\* @param filePath

\* @return

\* @throws IOException

\* @throws SOAPException

\*/

public SOAPMessage create(MimeHeaders headers, String filePath) throws IOException, SOAPException {

InputStream is = MessageUtil.class.getResourceAsStream(filePath);

SOAPMessage message = factory.createMessage(headers, is);

is.close();

return message;

}

/\*\*

\* 獲取MessageFactory

\* @return

\*/

public MessageFactory getMessageFactory() {

returnfactory;

}

}

文件SOAPMessageResp.xml的內容為：

Xml代碼 收藏代碼

<?xml version="1.0" encoding="utf-8" ?>

<SOAP-ENV:Envelope

xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"

xmlns:xs="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<SOAP-ENV:Body>

<ns4:sayHiResponse xmlns:ns4="http://provider.jaxws.sample.cxftest.tiantian.com/">

<ns4:responseType>SOAPMessage Response</ns4:responseType>

</ns4:sayHiResponse>

</SOAP-ENV:Body>

</SOAP-ENV:Envelope>

有了Provider實現類之後我們就可以把它發佈為一個WebService了，如：

Java代碼 收藏代碼

Object service = new SOAPMessageModeProvider();

Endpoint.publish("http://localhost:8080/test/jaxws/services/SOAPMessage", service);

4 客戶端訪問底層消息

客戶端訪問底層消息是通過Dispatch介面進行的，跟服務端的Provider介面一樣，Dispatch介面中同樣定義了一個invoke方法，該方法負責向服務端發送一種數據類型的消息，並返回一個對應類型的消息。不同的是Dispatch介面的實現類可以不需要我們自己定義和實現。我們可以通過創建代表服務端對應WebService對象的Service對象來創建一個Dispatch對象。Service類中定義了一系列的createDispatch重載方法，但比較常用的還是如下方法：

Java代碼 收藏代碼

public <T> Dispatch<T> createDispatch(QName portName, Class<T> type, Mode mode)

該方法接收三個參數：

l 第一個參數QName類型的portName代表目標Service中對應的portName；

l 第二個參數表示底層發送和接收消息時使用的數據類型，根據配置的不同可以是前面提到的三種數據類型中的一種；

l 第三個參數表示使用的消息模式。

下面我們來看一個創建Dispatch，並使用它來與服務端進行交互的例子：

Java代碼 收藏代碼

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

//定義serviceName對應的QName，第一個參數是對應的namespace

QName serviceName = new QName("http://provider.jaxws.sample.cxftest.tiantian.com/", "SOAPMessageService");

//定義portName對應的QName

QName portName = new QName("http://provider.jaxws.sample.cxftest.tiantian.com/", "SOAPMessagePort");

//使用serviceName創建一個Service對象，該對象還不能直接跟WebService對象進行交互

Service service = Service.create(serviceName);

//創建一個port，並指定WebService的位址，指定位址後我們就可以創建Dispatch了。

service.addPort(portName, SOAPBinding.SOAP11HTTP\_BINDING, "http://localhost:8080/test/jaxws/services/SOAPMessage");

//創建一個Dispatch對象

Dispatch<SOAPMessage> dispatch = service.createDispatch(portName, SOAPMessage.class, Mode.MESSAGE);

//創建一個SOAPMessage

SOAPMessage request = MessageUtil.getInstance().create(null, "/dispatch/SOAPMessageReq.xml");

//調用Dispatch的invoke方法，發送一個SOAPMessage請求，並返回一個SOAPMessage響應。

SOAPMessage response = dispatch.invoke(request);

System.out.println("服務端返回如下： ");

response.writeTo(System.out);

}

在上面的代碼中，我們先通過serviceName創建了一個Service對象，然後再通過addPort方法指定其對應的WebService位址。其實，我們也可以像下面這樣，通過WebService對應的wsdl文件和serviceName創建對應的Service對象。

Java代碼 收藏代碼

//指定wsdl檔的位置

URL wsdl = new URL("http://localhost:8080/test/jaxws/services/SOAPMessage?wsdl");

Service service = Service.create(wsdl, serviceName);

上述例子中對應的SOAPMessageReq.xml文件的內容如下：

Xml代碼 收藏代碼

<?xml version="1.0" encoding="utf-8" ?>

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"

xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<SOAP-ENV:Body>

<ns4:sayHi xmlns:ns4="http://provider.jaxws.sample.cxftest.tiantian.com/">

<ns4:requestType>SOAPMessage Request</ns4:requestType>

</ns4:sayHi>

</SOAP-ENV:Body>

</SOAP-ENV:Envelope>

4.1 Dispatch的三種請求方式

同步請求

上述Dispatch的invoke方法請求是同步的，也就是阻塞式的，程式在調用了invoke方法之後會一直等待服務端的返回。

異步請求

異步請求是通過Dispatch的invokeAsync方法進行的。Dispatch的異步請求有兩種方式，一種是通過定期輪詢Dispatch調用invokeAsync方法返回的Response對象是否已經可以返回，另一種是通過回調函數的形式。所以，針對於這兩種方式，Dispatch的invokeAsync有兩個重載方法：

Java代碼 收藏代碼

public Response<T> invokeAsync(T msg);

public Future<?> invokeAsync(T msg, AsyncHandler<T> handler);

使用定期輪詢的方式時，我們在執行完invokeAsync之後會返回一個Response對象，該對象會定期輪詢判斷invokeAsync方法是否已經完成。當invokeAsync方法調用完成之後，Response對象的isDone()方法會返回true，但是這種調用的完成並不一定是成功的完成，有可能是出異常了，或者其他什麼問題。在調用完成，也就是isDone()方法的結果為true之後，我們就可以通過Response對象的get()方法嘗試獲取對應的返回對象了，之所以說是嘗試獲取，是因為我們的invokeAsync方法不一定是正常的完成了，如果沒有正常完成，調用get()方法將拋出異常。上面Dispatch調用的例子如果我們把它改為定期輪詢的異步請求的話，其調用過程的代碼可以是這樣子：

Java代碼 收藏代碼

SOAPMessage request = MessageUtil.getInstance().create(null, "/dispatch/SOAPMessageReq.xml");

Response<SOAPMessage> response = dispatch.invokeAsync(request);

System.out.println("開始判斷調用是否已完成");

while (!response.isDone()) {

Thread.sleep(200l);

}

SOAPMessage responseMsg = null;

try {

responseMsg = response.get();

} catch (Exception e) {

System.out.println("調用失敗");

}

if (responseMsg != null) {

System.out.println("服務端返回如下： ");

responseMsg.writeTo(System.out);

}

使用回調函數的方式時，我們需要給invokeAsync方法傳遞一個AsyncHandler介面的實現類作為回調對象。AsyncHandler介面中定義了一個handleResponse方法可以處理服務端返回的結果。當請求完成以後，Dispatch後端的線程會調用AsyncHandler對象的handleResponse方法。前面Dispatch調用的例子如果我們把它改為使用回調函數異步調用的話，其核心代碼可以是如下這個樣子：

Java代碼 收藏代碼

SOAPMessage request = MessageUtil.getInstance().create(null, "/dispatch/SOAPMessageReq.xml");

uture<?> future = dispatch.invokeAsync(request, new AsyncHandler<SOAPMessage>() {

@Override

public void handleResponse(Response<SOAPMessage> res) {

try {

System.out.println("回調函數被調用了……");

SOAPMessage responseMsg = res.get();

responseMsg.writeTo(System.out);

} catch (Exception e) {

e.printStackTrace();

}

}

});

System.out.println("可以開始做其他事情了……");

while (!future.isDone()) {

System.out.println("在請求完成之前，整個程式不能結束，否則回調函數不會被調用");

Thread.sleep(200l);

}

注意，在使用回調函數方式使用Dispatch的異步請求時，請求結果未返回前整個程式不能停止。如果在請求結果返回以前，整個程式結束了，回調函數不會被調用。當然這種情況只會出現在如上單次執行的測試環境下，我們經常使用的Web環境是不會出現此問題的。由此看來，回調函數應該是被Dispatch內部的守護線程調用的。

一次請求

一次請求是通過invokeOneWay方法來進行的。它表示我們的客戶端只需要發送請求，而不需要等待服務端的返回。

Java代碼 收藏代碼

SOAPMessage request = MessageUtil.getInstance().create(null, "/dispatch/SOAPMessageReq.xml");

dispatch.invokeOneWay(request);

5 統一入口

JaxWs基於消息編程的一個好處是我們可以在服務端使用一個Provider來接收和處理所有的WebService請求，使用一個Dispatch或多個Dispatch來發送請求，從而達到對WebService的統一管理；另一個好處是客戶端可以不定義或者說是不需要使用SEI介面及其相關的類。下面我們來看一個客戶端和服務端之間直接通過消息編程的簡單示例。

在服務端定義一個Provider<DOMSource>的實現類UniteServiceProvider。

Java代碼 收藏代碼

@WebServiceProvider(serviceName="UniteService", portName="UniteServicePort", targetNamespace="http://provider.jaxws.sample.cxftest.tiantian.com/")

@ServiceMode(Service.Mode.MESSAGE)

@BindingType(HTTPBinding.HTTP\_BINDING)

public class UniteServiceProvider implements Provider<DOMSource> {

@Override

public DOMSource invoke(DOMSource request) {

DOMSource response = null;

MessageUtil.getInstance().printSource(request);

Document requestDoc = (Document)request.getNode();

Element commandEle = (Element)requestDoc.getElementsByTagName("command").item(0);

Element paramEle = (Element)requestDoc.getElementsByTagName("param").item(0);

String command = commandEle.getTextContent();

try {

response = this.getResponse(command, paramEle);

} catch (Exception e) {

e.printStackTrace();

}

MessageUtil.getInstance().printSource(response);

return response;

}

/\*\*

\* 根據指令和對應的參數進行相關操作並返回對應的操作結果

\* @param command

\* @param paramEle

\* @return

\* @throws Exception

\*/

private DOMSource getResponse(String command, Element paramEle) throws Exception {

String responseContent = "<response><product><id>1</id><name>Apple</name></product></response>";

InputStream is = new ByteArrayInputStream(responseContent.getBytes("UTF-8"));

Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(is);

DOMSource response = new DOMSource(doc);

return response;

}

}

在上述代碼中我們的服務端接收到一個DOMSource的request請求之後，通過DOMSource的getNode方法取到存放在其中的Document對象。之後我們對該Document的內容作了一個解析，然後把對應的返回結果封裝成一個DOMSource進行返回。

上述使用到的MessageUtil類裡面的printSource方法的代碼為：

Java代碼 收藏代碼

/\*\*

\* 輸出Source的內容

\* @param source

\*/

public void printSource(Source source) {

StreamResult result = new StreamResult(System.out);

try {

TransformerFactory.newInstance().newTransformer().transform(source, result);

System.out.println();

} catch (Exception e) {

e.printStackTrace();

}

}

發佈上述Provider的過程這裡就不再贅述了。接著來看一下客戶端調用的代碼：

Java代碼 收藏代碼

public class UniteServiceClient {

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

String requestContent = "<request><command>10001</command><parameter><id>1</id></parameter></request>";

InputStream is = new ByteArrayInputStream(requestContent.getBytes("UTF-8"));

Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(is);

DOMSource requestMsg = new DOMSource(doc);

MessageUtil.getInstance().printSource(requestMsg);

QName serviceName = new QName("http://provider.jaxws.sample.cxftest.tiantian.com/", "UniteService");

QName portName = new QName("http://provider.jaxws.sample.cxftest.tiantian.com/", "UniteServicePort");

Service service = Service.create(serviceName);

//指定綁定方式為HTTPBinding

service.addPort(portName, HTTPBinding.HTTP\_BINDING, "http://localhost:8080/test/jaxws/services/UniteService");

Dispatch<DOMSource> dispatch = service.createDispatch(portName, DOMSource.class, Mode.MESSAGE);

DOMSource responseMsg = dispatch.invoke(requestMsg);

System.out.println("服務端返回來的資訊是：");

MessageUtil.getInstance().printSource(responseMsg);

}

}

6 注意

6.1 MESSAGE和PAYLOAD的區別

在文章的開始部分介紹了兩種消息模式，以及它們之間的區別。MESSAGE模式是訪問的整個消息，而PAYLOAD模式訪問的只是消息的部分內容。我們也知道，當我們使用DOMSource作為消息對象時我們可以使用MESSAGE和PAYLOAD這兩種模式。所以接下來我們就來說說使用DOMSource作為消息對象時使用MESSAGE模式和PAYLOAD模式的區別。

為了具有可比性，我們指定使用的BindingType為SOAPBinding，（未指定BindingType時默認也為SOAPBinding），下面來看看使用MESSAGE模式和PAYLOAD模式的區別。

MESSAGE模式

使用MESSAGE模式，我們在發送DOMSource消息對象時，如果我們的DOMSource消息對象裡面持有的Document不是一個SOAPPart（SOAPPart是一個實現了Document介面的抽象類），那麼系統會先生成一個SOAPPart，然後把我們的DOMSource裡面持有的Document作為SOAPPart關聯的SOAPEnvelope對象的SOAPBody部分。然後再把該SOAPPart作為DOMSource持有的Document對象。這個時候如果我們只想獲取到最原始的document，也就是SOAPBody包裹的那一段文檔，我們得這樣來取：

Java代碼 收藏代碼

public DOMSource invoke(DOMSource request) {

SOAPPart soapPart = (SOAPPart) request.getNode();

try {

SOAPEnvelope soapEnvelop= soapPart.getEnvelope();

SOAPBody soapBody = soapEnvelop.getBody();

Document preDoc = soapBody.extractContentAsDocument();

} catch (SOAPException e1) {

e1.printStackTrace();

}

returnnull;

}

如果DOMSource本身持有的Document對象就是一個SOAPPart的話就可以直接發送了，不需要再做轉換了。當我們的DOMSource持有的不是一個SOAPPart時，系統在生成SOAPPart時很可能會拋出異常資訊：HIERARCHY\_REQUEST\_ERR: 嘗試在不允許的位置插入節點。所以當我們配合使用SOAPBinding、DOMSource消息對象和MESSAGE模式時，我們最好給DOMSource傳入一個SOAPPart對象或者是SOAPPart格式的Document對象。

PAYLOAD模式

使用PAYLOAD模式時，我們發送的DOMSource消息會直接發送過去。對方接收到的內容和發送時的內容是一樣的，注意只是內容是一樣的，其持有的Document對象還是會當做一個普通的Document對象處理，如DocumentImpl。比如發送的時候DOMSource持有的是一個SOAPPart，那麼接收的時候接收到的DOMSource裡面的Document的內容還是發送時SOAPPart的內容，但是對象卻是一個普通的Document對象，而不是發送時的SOAPPart對象；而如果發送的時候發送的是一個普通的Document對象，那麼接收到的內容也只是一個普通Document的內容，不會像MESSAGE模式那樣會有多餘的SOAPHeader等資訊。