

Broadview
www.broadview.com.cn

Netty

权威指南

李林锋 / 著

资深一线专家诚意之作

由浅入深助力Netty进阶

Netty: The Definitive Guide

10.3 Netty HTTP+XML 协议栈开发

由于 HTTP 协议的通用性，很多异构系统间的通信交互采用 HTTP 协议，通过 HTTP 协议承载业务数据进行消息交互，例如非常流行的 HTTP+XML 或者 RESTful+JSON。

在 Java 领域，最常用的 HTTP 协议栈就是基于 Servlet 规范的 Tomcat 等 Web 容器，由于谷歌等业界大佬的强力推荐，Jetty 等轻量级的 Web 容器也得到了广泛的应用。但是，很多基于 HTTP 的应用都是后台应用，HTTP 仅仅是承载数据交换的一个通道，是一个载体而不是 Web 容器，在这种场景下，一般不需要类似 Tomcat 这样的重量级 Web 容器。

在网络安全日益严峻的今天，重量级的 Web 容器由于功能繁杂，会存在很多安全漏洞，典型的如 Tomcat。如果你的客户是安全敏感型的，这意味着你需要为 Web 容器做很多安全加固工作去修补这些漏洞，然而你并没有使用到这些功能，这会带来开发和维护成本的增加。在这种场景下，一个更加轻量级的 HTTP 协议栈是个更好的选择。

本章节将介绍如何利用 Netty 提供的基础 HTTP 协议栈功能，扩展开发 HTTP+XML 协议栈。

10.3.1 开发场景介绍

作为一个示例程序，我们先模拟一个简单的用户订购系统。客户端填写订单，通过 HTTP 客户端向服务端发送订购请求，请求消息放在 HTTP 消息体中，以 XML 承载，即采用 HTTP+XML 的方式进行通信。HTTP 服务端接收到订购请求后，对订单请求进行修改，然后通过 HTTP+XML 的方式返回应答消息。双方采用 HTTP1.1 协议，连接类型为 CLOSE 方式，即双方交互完成，由 HTTP 服务端主动关闭链路，随后客户端也关闭链路并退出。

订购请求消息定义如表 10-3 所示。

表 10-3 订购请求消息定义 (Order)

字段名称	类 型	备 注
订购数量	Int64	订购的商品数量
客户信息	Customer	客户信息，负责 POJO 对象

第 10 章 HTTP 协议开发应用

账单地址	Address	账单的地址
寄送方式	Shipping	枚举类型如下： 普通邮递 宅急送 国际邮递 国内快递 国际快递
送货地址	Address	送货地址
总价	float	商品总价

客户信息定义如表 10-4 所示。

表 10-4 客户信息定义 (Customer)

字段名称	类 型	备 注
客户 ID	Int64	客户 ID, 长整型
姓	String	客户姓氏, 字符串
名	String	客户名字, 字符串
全名	List<String>	客户全称, 字符列表

地址信息定义如表 10-5 所示。

表 10-5 地址信息定义 (Address)

字段名称	类 型	备 注
街道 1	String	
街道 2	String	
城市	String	
省份	String	
邮政编码	String	
国家	String	

邮递方式定义如表 10-6 所示。

表 10-6 邮递方式定义 (Shipping)

字段名称	类 型	备 注
普通邮递	枚举类型	
宅急送	枚举类型	

Netty 权威指南

国际邮递	枚举类型	
国内快递	枚举类型	
国际快递	枚举类型	

数据定义完成之后，接着看订购请求消息的 XML Schema 定义。

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://
phei.com/netty/protocol/http/xml/pojo" elementFormDefault="qualified"
targetNamespace="http://phei.com/netty/protocol/http/xml/pojo">
  <xs:element type="tns:order" name="order"/>
  <xs:complexType name="address">
    <xs:sequence>
      <xs:element type="xs:string" name="street1" minOccurs="0"/>
      <xs:element type="xs:string" name="street2" minOccurs="0"/>
      <xs:element type="xs:string" name="city" minOccurs="0"/>
      <xs:element type="xs:string" name="state" minOccurs="0"/>
      <xs:element type="xs:string" name="postCode" minOccurs="0"/>
      <xs:element type="xs:string" name="country" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="order">
    <xs:sequence>
      <xs:element name="customer" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element type="xs:string" name="firstName" minOccurs="0"/>
            <xs:element type="xs:string" name="lastName" minOccurs="0"/>
            <xs:element type="xs:string" name="middleName" minOccurs="0"
maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute type="xs:long" use="required" name="customerNumber"/>
        </xs:complexType>
      </xs:element>
      <xs:element type="tns:address" name="billTo" minOccurs="0"/>
      <xs:element name="shipping" minOccurs="0">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="STANDARD_MAIL"/>
            <xs:enumeration value="PRIORITY_MAIL"/>
            <xs:enumeration value="INTERNATIONAL_MAIL"/>
            <xs:enumeration value="DOMESTIC_EXPRESS"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

```

```
        <xs:enumeration value="INTERNATIONAL_EXPRESS"/>
    </xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element type="tns:address" name="shipTo" minOccurs="0"/>
</xs:sequence>
<xs:attribute type="xs:long" use="required" name="orderNumber"/>
<xs:attribute type="xs:float" name="total"/>
</xs:complexType></xs:schema>
```

熟悉 XML 和 Schema 的读者理解上面的 Schema 定义没有什么困难，如果你对 XML 的相关知识还不太了解，建议先找一本 XML 开发入门方面的书籍进行学习，或者登录 W3C 的网站学习 XML 的相关知识。

开发背景介绍完毕，下面我们进入设计环节，看看如何设计和开发 HTTP+XML 协议栈。

10.3.2 HTTP+XML 协议栈设计

通过商品订购的流程图看下订购的关键步骤和主要技术点，找出当前 Netty HTTP 协议栈的功能不足之后，通过扩展的方式完成 HTTP+XML 协议栈的开发。如图 10-10 所示。

Netty 权威指南

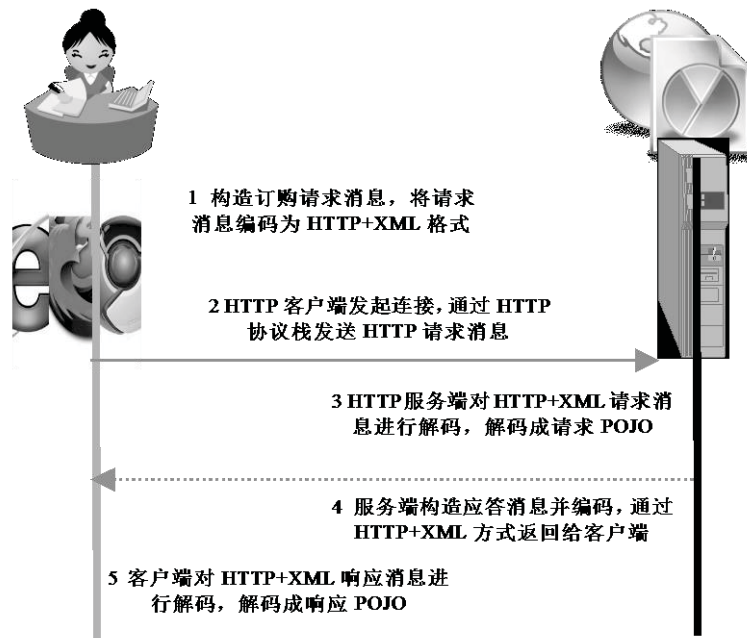


图 10-10 HTTP+XML 订购流程图

首先对订购流程图进行分析，先看步骤 1，构造订购请求消息并将其编码为 HTTP+XML 形式。Netty 的 HTTP 协议栈提供了构造 HTTP 请求消息的相关接口，但是无法将普通的 POJO 对象转换为 HTTP+XML 的 HTTP 请求消息，需要自定义 HTTP+XML 格式的请求消息编码器。

再看步骤 2，利用 Netty 的 HTTP 协议栈，可以支持 HTTP 链路的建立和请求消息的发送，所以不需要额外开发，直接重用 Netty 的能力即可。

步骤 3，HTTP 服务端需要将 HTTP+XML 格式的订购请求消息解码为订购请求 POJO 对象，同时获取 HTTP 请求消息头信息。利用 Netty 的 HTTP 协议栈服务端，可以完成 HTTP 请求消息的解码，但是，如果消息体为 XML 格式，Netty 无法支持将其解码为 POJO 对象，需要在 Netty 协议栈的基础上扩展实现。

步骤 4，服务端对订购请求消息处理完成后，重新将其封装成 XML，通过 HTTP 应答消息体携带给客户端，Netty 的 HTTP 协议栈不支持直接将 POJO 对象的应答消息以 XML 方式发送，需要定制。

步骤 5，HTTP 客户端需要将 HTTP+XML 格式的应答消息解码为订购 POJO 对象，同

时能够获取应答消息的 HTTP 头信息，Netty 的协议栈不支持自动的消息解码。

通过分析，我们可以了解到哪些能力是 Netty 支持的，哪些需要扩展开发实现。下面给出设计思路。

(1) 需要一套通用、高性能的 XML 序列化框架，它能够灵活地实现 POJO-XML 的互相转换，最好能够通过工具自动生成绑定关系，或者通过 XML 的方式配置双方的映射关系；

(2) 作为通用的 HTTP+XML 协议栈，XML-POJO 对象的映射关系应该非常灵活，支持命名空间和自定义标签；

(3) 提供 HTTP+XML 请求消息编码器，供 HTTP 客户端发送请求消息自动编码使用；

(4) 提供 HTTP+XML 请求消息解码器，供 HTTP 服务端对请求消息自动解码使用；

(5) 提供 HTTP+XML 响应消息编码器，供 HTTP 服务端发送响应消息自动编码使用；

(6) 提供 HTTP+XML 响应消息解码器，供 HTTP 客户端对应答消息进行自动解码使用；

(7) 协议栈使用者不需要关心 HTTP+XML 的编解码，对上层业务零侵入，业务只需要对上层的业务 POJO 对象进行编排。

下个小节我们将讲述 XML 框架的选型和开发，它是 HTTP+XML 协议栈的关键技术点。

10.3.3 高效的 XML 绑定框架 JiBx

JiBx 是一款非常优秀的 XML (Extensible Markup Language) 数据绑定框架。它提供灵活的绑定映射文件，实现数据对象与 XML 文件之间的转换，并不需要修改既有的 Java 类。另外，它的转换效率是目前很多其他开源项目都无法比拟的。

1. JiBx 入门

XML 已经成为目前程序开发配置的重要组成部分了，可以用来操作 XML 文件的开源项目也已经成熟，比如说流行的 Digester、XStream、Castor、JDOM、dom4j 和 Xalan 等，当然也少不了专门为 Java 语言设计的 XML 数据绑定框架 JiBx。它的主要优点包括：转换效率高、配置绑定文件简单、不需要操作 xpath 文件、不需要写属性的 get/set 方法、对象属性名与 XML 文件 element 名可以不同，等等。

使用 JiBx 绑定 XML 文档与 Java 对象需要分两步走：第一步是绑定 XML 文件，也就是映射 XML 文件与 Java 对象之间的对应关系；第二步是在运行时，实现 XML 文件与 Java

Netty 权威指南

实例之间的互相转换。这时，它已经与绑定文件无关了，可以说是完全脱耦了。

在运行程序之前，需要先配置绑定文件并进行绑定，在绑定过程中它将会动态地修改程序中相应的 class 文件，主要是生成对应对象实例的方法和添加被绑定标记的属性 JiBX_bindingList 等。它使用的技术是 BCEL (Byte Code Engineering Library)，BCEL 是 Apache Software Foundation 的 Jakarta 项目的一部分，也是目前 Java classworking 最广泛使用的一种框架，它可以让你深入 JVM 汇编语言进行类操作。在 JiBX 运行时，它使用了目前比较流行的一个技术 XPP (Xml Pull Parsing)，这也正是 JiBX 如此高效的原因。

JiBx 有两个比较重要的概念：Unmarshal (数据分解) 和 Marshal (数据编排)。从字面意思也很容易理解，Unmarshal 是将 XML 文件转换成 Java 对象，而 Marshal 则是将 Java 对象编排成规范的 XML 文件。JiBx 在 Unmarshal/Marshal 上如此高效，这要归功于使用了 XPP 技术，而不是使用基于树型 (tree-based) 方式，将整个文档写入内存，然后进行操作的 DOM (Document Object Model)，也不是使用基于事件流 (event stream) 的 SAX (Simple API for Xml)。XPP 使用的是不断增加的数据流处理方式，同时允许在解析 XML 文件时中断。

介绍完了 JiBx 的基础概念，下面我们就结合订购例程，来学习下如何使用 JiBx 进行 XML 的开发。

2 . POJO 对象定义

通过 JiBx 提供的工具 jar 包，可以根据 Schema 自动生成 POJO 对象，也可以根据普通的 POJO 对象生成 JiBx 绑定文件和 Schema 定义 XSD。

考虑到大多数人的编码习惯，我们采用先定义 POJO 对象，再生成 XML 和对象的绑定文件的方式。JiBx 对 POJO 对象没有特殊要求，只要符合 Java Bean 的规则即可，下面我们以订购例程为例，看下主要类的定义。

代码清单 10-3 HTTP+XML POJO 类定义 Order

```
2. public class Order {
3.     private long orderNumber;
4.     private Customer customer;
5.
6.     /** Billing address information. */
7.     private Address billTo;
8.
9.     private Shipping shipping;
```


第 10 章 HTTP 协议开发应用

```
10.  
11.     /**  
12.      * Shipping address information. If missing, the billing address is also  
13.      * used as the shipping address.  
14.      */  
15.     private Address shipTo;  
16.  
17.     private Float total;  
.....//定义 set 和 get 方法  
53. }
```

代码清单 10-4 HTTP+XML POJO 类定义 Customer

```
2. import java.util.List;  
3. public class Customer {  
4.     private long customerNumber;  
5.     /** Personal name. */  
6.     private String firstName;  
7.  
8.     /** Family name. */  
9.     private String lastName;  
10.    /** Middle name(s), if any. */  
11.    private List<String> middleNames;  
.....//定义 set 和 get 方法  
35. }
```

代码清单 10-5 HTTP+XML POJO 类定义 Address

```
2. public class Address {  
3.     /** First line of street information (required). */  
4.     private String street1;  
5.     /** Second line of street information (optional). */  
6.     private String street2;  
7.     private String city;  
8.  
9.     /**  
10.    * State abbreviation (required for the U.S. and Canada, optional  
11.    * otherwise).  
12.    */  
13.     private String state;  
14.  
15.     /** Postal code (required for the U.S. and Canada, optional otherwise). */  
16.     private String postCode;  
17.     /** Country name (optional, U.S. assumed if not supplied). */
```

Netty 权威指南

```

18.     private String country;
.....//定义 set 和 get 方法
54.     }

```

代码清单 10-6 HTTP+XML POJO 类定义 Shipping

```

1. package com.phei.netty.protocol.http.xml.pojo;
2.
3. public enum Shipping {
4.     STANDARD_MAIL, PRIORITY_MAIL, INTERNATIONAL_MAIL, DOMESTIC_EXPRESS,
INTERNATIONAL_EXPRESS
5. }

```

POJO 对象定义完成之后，通过 Ant 脚本来生成 XML 和 POJO 对象的绑定关系文件，同时也附加生成 XML 的 Schema 定义文件。下个小节开始介绍 Ant 脚本的编写。

3. 通过 Ant 脚步生成 XML 和对象的绑定关系

首先确认你使用的 Eclipse 安装了 Ant，通过【window】主菜单选择【Preferences】，在弹出的窗口中可以查看是否支持 Ant。如图 10-11 所示。

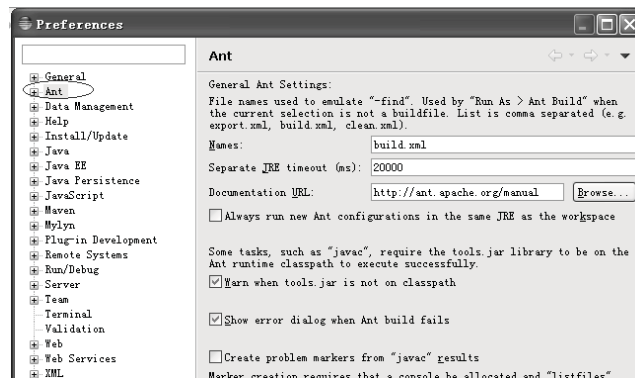


图 10-11 Eclipse 的 Ant 配置

目前主流的 Eclipse 版本默认都支持 Ant，如果你使用其他的开发工具，可以安装对应的 Ant 插件。

由于本书的重点并非讲解 Ant 的使用，所以，如果你对 Ant 感兴趣或者作为初学者学习 Ant，可以选择相关的 Ant 教材或者直接通过 Ant 的官方文档进行学习。

JiBx 的绑定和编译，重点关注两个 task。如图 10-12 所示。

第 10 章 HTTP 协议开发应用

```

<!-- generate default binding and schema -->
<target name="bindgen">
  <echo message="Running BindGen tool"/>
  <java classpathref="classpath" fork="true" failonerror="true"
        classname="org.jibx.binding.generator.BindGen">
    <arg value="-s"/>
    <arg value="\${basedir}/src/com/phei/netty/protocol/http/xml/pojo"/>
    <arg value="com.phei.netty.protocol.http.xml.pojo.Order"/>
  </java>
</target>

```

图 10-12 使用 BindGen 命令生成绑定文件

通过 JiBx 的 org.jibx.binding.generator.BindGen 工具类可以将指定的 POJO 对象 Order 类生成绑定文件和 Schema 定义文件，执行结果如图 10-13 所示。

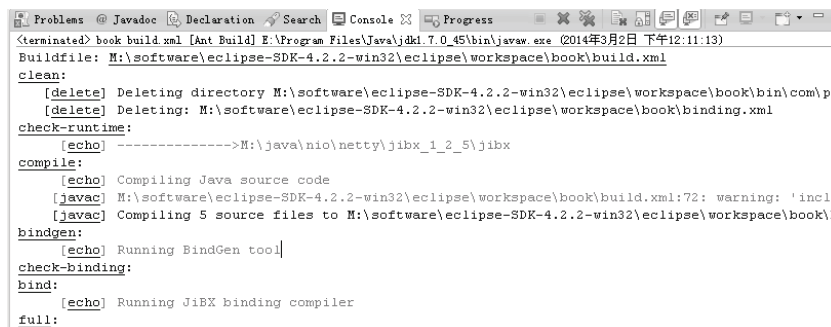


图 10-13 执行 Ant 脚本，生成 XML 绑定关系

执行成功后，在当前的工程目录下生成 binding.xml 和 pojo.xsd 文件，结果如图 10-14 所示。

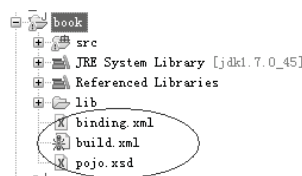


图 10-14 生成的 XML 绑定文件

打开绑定文件，我们可以发现绑定文件实际就是 XML 的元素和 POJO 对象字段之间的映射关系，生成之后的文件可以手工调整，例如修改 XML 中的元素名称，必选和可选标识等。如果你熟悉绑定规则，绑定文件也可以自己手工配置开发。XML 和 Order 对象的映射关系如图 10-15 所示。

Netty 权威指南

```

<binding xmlns:ns1="http://phei.com/netty/protocol/http/xml/pojo" name="binding" package="com.phei.netty
<namespace uri="http://phei.com/netty/protocol/http/xml/pojo" default="elements"/>
<mapping abstract="true" type-name="ns1:order" class="com.phei.netty.protocol.http.xml.pojo.Order">
  <value style="attribute" name="orderNumber" field="orderNumber"/>
  <structure field="customer" usage="optional" name="customer">
    <value style="attribute" name="customerNumber" field="customerNumber"/>
    <value style="element" name="firstName" field="firstName" usage="optional"/>
    <value style="element" name="lastName" field="lastName" usage="optional"/>
    <collection field="middleNames" usage="optional" create-type="java.util.ArrayList">
      <value name="middleName" type="java.lang.String"/>
    </collection>
  </structure>
  <structure map-as="ns1:address" field="billTo" usage="optional" name="billTo"/>
  <value style="element" name="shipping" field="shipping" usage="optional"/>
  <structure map-as="ns1:address" field="shipTo" usage="optional" name="shipTo"/>
  <value style="attribute" name="total" field="total" usage="optional"/>
</mapping>
<mapping class="com.phei.netty.protocol.http.xml.pojo.Order" name="order">
  <structure map-as="ns1:order"/>
</mapping>
<mapping abstract="true" type-name="ns1:address" class="com.phei.netty.protocol.http.xml.pojo.Address">
  <value style="element" name="street1" field="street1" usage="optional"/>
  <value style="element" name="street2" field="street2" usage="optional"/>
  <value style="element" name="city" field="city" usage="optional"/>
  <value style="element" name="state" field="state" usage="optional"/>
  <value style="element" name="postCode" field="postCode" usage="optional"/>
  <value style="element" name="country" field="country" usage="optional"/>
</mapping>

```

图 10-15 XML 和 Order 对象的映射关系

再看一下 JiBx 的编译命令，它的作用是根据绑定文件和 POJO 对象的映射关系和规则动态修改 POJO 类，Ant 脚本如图 10-16 所示。

```

<!-- bind as a separate step -->
<target name="bind" depends="check-binding">
  <echo message="Running JiBX binding compiler"/>
  <taskdef name="bind" classname="org.jibx.binding.ant.CompileTask">
    <classpath>
      <fileset dir="${jibx-home}/lib" includes="*.jar"/>
    </classpath>
  </taskdef>
  <bind binding="${basedir}/binding.xml">
    <classpath refid="classpath"/>
  </bind>
</target>

```

图 10-16 编译绑定脚本和动态修改 Class 的 Ant 脚本

编译结果如图 10-17 所示。

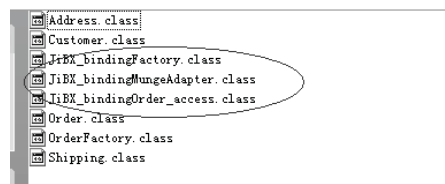


图 10-17 动态修改 Class 文件

到此为止，JiBx 相关的准备工作已经完成，下个小节通过一个简单的测试程序来学习 JiBx 类库的使用。

4 . JiBx 的类库使用

JiBx 的类库使用非常简单，下面直接看代码。

代码清单 10-7 HTTP+XML POJO 测试类定义 TestOrder

```
18. public class TestOrder {
19.     private IBindingFactory factory = null;
20.     private StringWriter writer = null;
21.     private StringReader reader = null;
22.     private final static String CHARSET_NAME = "UTF-8";
23.     private String encode2Xml(Order order) throws JiBxException,
IOException {
24.         factory = BindingDirectory.getFactory(Order.class);
25.         writer = new StringWriter();
26.         IMarshallingContext mctx = factory.createMarshallingContext();
27.         mctx.setIndent(2);
28.         mctx.marshalDocument(order, CHARSET_NAME, null, writer);
29.         String xmlStr = writer.toString();
30.         writer.close();
31.         System.out.println(xmlStr.toString());
32.         return xmlStr;
33.     }
34.
35.     private Order decode2Order(String xmlBody) throws JiBxException {
36.         reader = new StringReader(xmlBody);
37.         IUnmarshallingContext uctx = factory.createUnmarshallingContext();
38.         Order order = (Order) uctx.unmarshalDocument(reader);
39.         return order;
40.     }
41.
42.     public static void main(String[] args) throws JiBxException,
IOException {
43.         TestOrder test = new TestOrder();
44.         Order order = OrderFactory.create(123);
45.         String body = test.encode2Xml(order);
46.         Order order2 = test.decode2Order(body);
47.         System.out.println(order2);
48.     }
49. }
```

首先看第 24 行，根据 Order 的 Class 实例构造 IBindingFactory 对象。第 25 行创建新的

Netty 权威指南

StringWriter 对象,通过 IBindingFactory 构造 Marshalling 上下文,最后通过 marshalDocument 将 Order 序列化为 StringWriter, 通过 StringWriter 的 toString()方法可以返回 String 类型的 XML 对象。

解码与编码类似,不同的是它使用 StringReader 来读取 String 类型的 XML 对象, 然后通过 unmarshalDocument 方法将其反序列化为 Order 对象。

执行结果如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<order xmlns="http://phei.com/netty/protocol/http/xml/pojo" orderNumber=
"123" total="9999.999">
  <customer customerNumber="123">
    <firstName>李</firstName>
    <lastName>林峰</lastName>
  </customer>
  <billTo>
    <street1>龙眠大道</street1>
    <city>南京市</city>
    <state>江苏省</state>
    <postCode>123321</postCode>
    <country>中国</country>
  </billTo>
  <shipping>INTERNATIONAL_MAIL</shipping>
  <shipTo>
    <street1>龙眠大道</street1>
    <city>南京市</city>
    <state>江苏省</state>
    <postCode>123321</postCode>
    <country>中国</country>
  </shipTo>
</order>
```

Order [orderNumber=123, customer=Customer [customerNumber=123, firstName=李, lastName= 林峰, middleNames=null], billTo=Address [street1= 龙眠大道, street2=null, city= 南京市, state= 江苏省, postCode=123321, country= 中国], shipping=INTERNATIONAL_MAIL, shipTo=Address [street1=龙眠大道, street2=null, city=南京市, state=江苏省, postCode=123321, country=中国], total=9999.999]

通过上面的执行结果可以发现, XML 序列化和反序列化后的结果与预期一致, 我们开发的 JiBx 应用可以正常工作。

XML 绑定框架选型和开发完成之后，下面继续 Netty HTTP +XM 编解码框架的开发。

10.3.4 HTTP+XML 编解码框架开发

本节共有 6 个小节来讲解如何基于 Netty 开发 HTTP+XML 协议栈，在 Netty 提供的 HTTP 基础协议栈上进行扩展和封装，以实现对上层业务的零侵入。下面我们一起学习如何进行开发。

1. HTTP+XML 请求消息编码类

对于上层业务侧，构造订购请求消息后，以 HTTP+XML 协议将消息发送给服务端，如果要对业务零侵入或者尽可能少的侵入，协议层和应用层应该解耦。

考虑到 HTTP+XML 协议栈需要一定的定制扩展能力，例如通过 HTTP 消息头携带业务自定义字段，所以，应该允许业务利用 Netty 的 HTTP 协议栈接口自行构造私有的 HTTP 消息头。

HTTP+XML 的协议编码仍然采用 ChannelPipeline 中增加对应的编码 handler 类实现。

下面我们来一起看下 HTTP+XML 请求消息编码类的源码实现。

代码清单 10-8 HTTP+XML HTTP 请求消息编码类

```
11. public class HttpXmlRequestEncoder extends
12.     AbstractHttpXmlEncoder<HttpXmlRequest> {
13.
14.     @Override
15.     protected void encode(ChannelHandlerContext ctx, HttpXmlRequest msg,
16.         List<Object> out) throws Exception {
17.         ByteBuf body = encode0(ctx, msg.getBody());
18.         FullHttpRequest request = msg.getRequest();
19.         if (request == null) {
20.             request = new DefaultFullHttpRequest(HttpVersion.HTTP_1_1,
21.                 HttpMethod.GET, "/do", body);
22.             HttpHeaders headers = request.headers();
23.             headers.set(HttpHeaders.Names.HOST,
InetAddress.getLocalHost()
24.                 .getHostAddress());
25.             headers.set(HttpHeaders.Names.CONNECTION,
HttpHeaders.Values.CLOSE);
```

Netty 权威指南

```

26.         headers.set(HttpHeaders.Names.ACCEPT_ENCODING,
27.             HttpHeaders.Values.GZIP.toString() + ', '
28.             + HttpHeaders.Values.DEFLATE.toString());
29.         headers.set(HttpHeaders.Names.ACCEPT_CHARSET,
30.             "ISO-8859-1,utf-8;q=0.7,*;q=0.7");
31.         headers.set(HttpHeaders.Names.ACCEPT_LANGUAGE, "zh");
32.         headers.set(HttpHeaders.Names.USER_AGENT,
33.             "Netty xml Http Client side");
34.         headers.set(HttpHeaders.Names.ACCEPT,
"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8");
35.     }
36.     HttpHeaders.setContentLength(request, body.readableBytes());
37.     out.add(request);
38. }
39. }

```

第 17 行首先调用父类的 `encode0`，将业务需要发送的 POJO 对象 `Order` 实例通过 `JiBx` 序列化为 XML 字符串，随后将它封装成 Netty 的 `ByteBuf`。第 18 行对消息头进行判断，如果业务侧自定义和定制了消息头，则使用业务侧设置的 HTTP 消息头，如果业务侧没有设置，则构造新的 HTTP 消息头。

第 20~35 行用来构造和设置默认的 HTTP 消息头，由于通常情况下 HTTP 通信双方更关注消息体本身，所以这里采用了硬编码的方式，如果要产品化，可以做成 XML 配置文件，允许业务自定义配置，以提升定制的灵活性。

第 36 行很重要，由于请求消息消息体不为空，也没有使用 `Chunk` 方式，所以在 HTTP 消息头中设置消息体的长度 `Content-Length`，完成消息体的 XML 序列化后将重新构造的 HTTP 请求消息加入到 `out` 中，由后续 Netty 的 HTTP 请求编码器继续对 HTTP 请求消息进行编码。

下面我们来看父类 `AbstractHttpXmlEncoder` 的实现。

代码清单 10-9 HTTP+XML HTTP 请求消息编码基类 `AbstractHttpXmlEncoder`

```

14. public abstract class AbstractHttpXmlEncoder<T> extends
15.     MessageToMessageEncoder<T> {
16.     IBindingFactory factory = null;
17.     StringWriter writer = null;
18.     final static String CHARSET_NAME = "UTF-8";
19.     final static Charset UTF_8 = Charset.forName(CHARSET_NAME);
20.

```


第 10 章 HTTP 协议开发应用

```
21.     protected ByteBuf encode0(ChannelHandlerContext ctx, Object body)
22.         throws Exception {
23.         factory = BindingDirectory.getFactory(body.getClass());
24.         writer = new StringWriter();
25.         IMarshallingContext mctx = factory.createMarshallingContext();
26.         mctx.setIndent(2);
27.         mctx.marshalDocument(body, CHARSET_NAME, null, writer);
28.         String xmlStr = writer.toString();
29.         writer.close();
30.         writer = null;
31.         ByteBuf encodeBuf = Unpooled.copiedBuffer(xmlStr, UTF_8);
32.         return encodeBuf;
33.     }
34.
35.     @Override
36.     public void exceptionCaught(ChannelHandlerContext ctx, Throwable
cause)
37.         throws Exception {
38.         // 释放资源
39.         if (writer != null) {
40.             writer.close();
41.             writer = null;
42.         }
43.     }
44. }
```

首先看下 23~30 行，代码很熟悉，在 JiBx 章节已经介绍了 XML 序列化和反序列化的相关类库使用，在此将业务的 Order 实例序列化为 XML 字符串。第 31 行将 XML 字符串包装成 Netty 的 ByteBuf 并返回，实现了 HTTP 请求消息的 XML 编码。

下面继续看下 HttpXmlRequest 是如何实现的。

代码清单 10-10 HTTP+XML 请求消息 HttpXmlRequest

```
9.     public class HttpXmlRequest {
10.         private FullHttpRequest request;
11.         private Object body;
12.
13.         public HttpXmlRequest(FullHttpRequest request, Object body) {
14.             this.request = request;
15.             this.body = body;
16.         }
```

Netty 权威指南

```
17.  
18.  /**  
19.   * @return the request  
20.   */  
21.  public final FullHttpRequest getRequest() {  
22.    return request;  
23.  }  
24.  
25.  /**  
26.   * @param request  
27.   *           the request to set  
28.   */  
29.  public final void setRequest(FullHttpRequest request) {  
30.    this.request = request;  
31.  }  
32.  
33.  /**  
34.   * @return the object  
35.   */  
36.  public final Object getBody() {  
37.    return body;  
38.  }  
39.  
40.  /**  
41.   * @param object  
42.   *           the object to set  
43.   */  
44.  public final void setBody(Object body) {  
45.    this.body = body;  
46.  }  
47. }
```

它包含两个成员变量 `FullHttpRequest` 和编码对象 `Object`, 用于实现和协议栈之间的解耦。

2 . HTTP+XML 请求消息解码类

HTTP 服务端接收到 HTTP+XML 请求消息后, 需要从 HTTP 消息体中获取请求码流, 通过 `JiBx` 框架对它进行反序列化, 得到请求 `POJO` 对象, 然后对结果进行封装, 回调到业务 `handler` 对象, 业务得到的就是解码后的 `POJO` 对象和 HTTP 消息头。

下面就看下具体实现。

代码清单 10-11 HTTP+XML HTTP 请求消息解码类 HttpXmlRequestDecoder

```
20. public class HttpXmlRequestDecoder extends
21.     AbstractHttpXmlDecoder<FullHttpRequest> {
22.
23.     public HttpXmlRequestDecoder(Class<?> clazz) {
24.         this(clazz, false);
25.     }
26.
27.     public HttpXmlRequestDecoder(Class<?> clazz, boolean isPrint) {
28.         super(clazz, isPrint);
29.     }
30.
31.     @Override
32.     protected void decode(ChannelHandlerContext arg0, FullHttpRequest
arg1,
33.         List<Object> arg2) throws Exception {
34.         if (!arg1.getDecoderResult().isSuccess()) {
35.             sendError(arg0, BAD_REQUEST);
36.             return;
37.         }
38.         HttpXmlRequest request = new HttpXmlRequest(arg1, decode0(arg0,
39.             arg1.content()));
40.         arg2.add(request);
41.     }
42.
43.     private static void sendError(ChannelHandlerContext ctx,
44.         HttpResponseStatus status) {
45.         FullHttpResponse response = new DefaultFullHttpResponse(HTTP_1_1,
46.             status, Unpooled.copiedBuffer("Failure: " + status.toString()
47.                 + "\r\n", CharsetUtil.UTF_8));
48.         response.headers().set(CONTENT_TYPE, "text/plain; charset=UTF-8");
49.         ctx.writeAndFlush(response).addListener(ChannelFutureListener.
CLOSE);
50.     }
51. }
```

`HttpXmlRequestDecoder` 有两个参数，分别为需要解码的对象的类型信息和是否打印 HTTP 消息体码流的码流开关，码流开关默认关闭。第 34~37 行首先对 HTTP 请求消息本身的解码结果进行判断，如果已经解码失败，再对消息体进行二次解码已经没有意义。

第 43~50 行，如果 HTTP 消息本身解码失败，则构造处理结果异常的 HTTP 应答消

Netty 权威指南

息返回给客户端。作为演示程序，本例程没有考虑 XML 消息解码失败后的异常封装和处理，在商用项目中需要统一的异常处理机制，提升协议栈的健壮性和可靠性。

第 38 行通过 `HttpXmlRequest` 和反序列化后的 `Order` 对象构造 `HttpXmlRequest` 实例，最后将它添加到解码结果 `List` 列表中。

继续看下它的父类 `AbstractHttpXmlDecoder` 的实现。

代码清单 10-12 HTTP+XML HTTP 请求消息解码类 `AbstractHttpXmlDecoder`

```
18. public abstract class AbstractHttpXmlDecoder<T> extends
19.     MessageToMessageDecoder<T> {
20.     private IBindingFactory factory;
21.     private StringReader reader;
22.     private Class<?> clazz;
23.     private boolean isPrint;
24.     private final static String CHARSET_NAME = "UTF-8";
25.     private final static Charset UTF_8 = Charset.forName(CHARSET_NAME);
26.
27.     protected AbstractHttpXmlDecoder(Class<?> clazz) {
28.         this(clazz, false);
29.     }
30.
31.     protected AbstractHttpXmlDecoder(Class<?> clazz, boolean isPrint) {
32.         this.clazz = clazz;
33.         this.isPrint = isPrint;
34.     }
35.
36.     protected Object decode0(ChannelHandlerContext arg0, ByteBuf body)
37.         throws Exception {
38.         factory = BindingDirectory.getFactory(clazz);
39.         String content = body.toString(UTF_8);
40.         if (isPrint)
41.             System.out.println("The body is : " + content);
42.         reader = new StringReader(content);
43.         IUnmarshallingContext uctx = factory.createUnmarshallingContext();
44.         Object result = uctx.unmarshalDocument(reader);
45.         reader.close();
46.         reader = null;
47.         return result;
48.     }
49.     @Override
```

```
50.     public void exceptionCaught(ChannelHandlerContext ctx, Throwable
cause)
51.         throws Exception {
52.         // 释放资源
53.         if (reader != null) {
54.             reader.close();
55.             reader = null;
56.         }
57.     }
58. }
```

第 38~47 行从 HTTP 的消息体中获取请求码流，然后通过 JiBx 类库将 XML 转换成 POJO 对象。最后，根据码流开关决定是否打印消息体码流。增加码流开关往往是为了方便问题定位，在实际项目中，需要打印到日志中。

第 53~55 行，如果解码发生异常，要判断 `StringReader` 是否已经关闭，如果没有关闭，则关闭输入流并通知 JVM 对其进行垃圾回收。

3 . HTTP+XML 响应消息编码类

对于响应消息，用户可能并不关心 HTTP 消息头之类的，它将业务处理后的 POJO 对象丢给 HTTP+XML 协议栈，由基础协议栈进行后续的处理。为了降低业务的定制开发难度，我们首先封装一个全新的 HTTP XML 应答对象，它的实现如下。

代码清单 10-13 HTTP+XML HTTP XML 应答消息 `HttpXmlResponse`

```
9.     public class HttpXmlResponse {
10.         private FullHttpResponse httpResponse;
11.         private Object result;
12.
13.         public HttpXmlResponse(FullHttpResponse httpResponse, Object result) {
14.             this.httpResponse = httpResponse;
15.             this.result = result;
16.         }
17.
18.         /**
19.          * @return the httpResponse
20.          */
21.         public final FullHttpResponse getHttpResponse() {
22.             return httpResponse;
23.         }
}
```

Netty 权威指南

```
24.
25.     /**
26.      * @param httpResponse
27.      *         the httpResponse to set
28.      */
29.     public final void setHttpResponse(FullHttpResponse httpResponse) {
30.         this.httpResponse = httpResponse;
31.     }
32.
33.     /**
34.      * @return the body
35.      */
36.     public final Object getResult() {
37.         return result;
38.     }
39.
40.     /**
41.      * @param body
42.      *         the body to set
43.      */
44.     public final void setResult(Object result) {
45.         this.result = result;
46.     }
47. }
```

它包含两个成员变量：`FullHttpResponse` 和 `Object`，`Object` 就是业务需要发送的应答 POJO 对象。

下面继续看应答消息的 XML 编码类实现。

代码清单 10-14 HTTP+XML 应答消息编码类 `HttpXmlResponseEncoder`

```
17. public class HttpXmlResponseEncoder extends
18.     AbstractHttpXmlEncoder<HttpXmlResponse> {
19.
20.     /**
21.      * (non-Javadoc)
22.      *
23.      * @see
24.      *     io.netty.handler.codec.MessageToMessageEncoder#encode(io.netty.
channel
25.      *     .ChannelHandlerContext, java.lang.Object, java.util.List)
```

第 10 章 HTTP 协议开发应用

```

26.     */
27.     protected void encode(ChannelHandlerContext ctx, HttpXmlResponse msg,
28.         List<Object> out) throws Exception {
29.         ByteBuf body = encode0(ctx, msg.getResult());
30.         FullHttpResponse response = msg.getHttpResponse();
31.         if (response == null) {
32.             response = new DefaultFullHttpResponse(HTTP_1_1, OK, body);
33.         } else {
34.             response = new DefaultFullHttpResponse(msg.getHttpResponse()
35.                 .getProtocolVersion(), msg.getHttpResponse().getStatus(),
36.                 body);
37.         }
38.         response.headers().set(CONTENT_TYPE, "text/xml");
39.         setContentLength(response, body.readableBytes());
40.         out.add(response);
41.     }
42. }

```

它的实现非常简单，第 30 行对应答消息进行判断，如果业务侧已经构造了 HTTP 应答消息，则利用业务已有应答消息重新复制一个新的 HTTP 应答消息。无法重用业务侧自定义 HTTP 应答消息的主要原因，是 Netty 的 `DefaultFullHttpResponse` 没有提供动态设置消息体 `content` 的接口，只能在第一次构造的时候设置内容。由于这个局限，导致我们的实现有点麻烦。

作为示例程序并没有提供更多的 API 供业务侧灵活设置 HTTP 应答消息头，在实际商用时，可以基于本书提供的基础协议栈进行扩展。

第 38 行设置消息体内容格式为“text/xml”，然后在消息头中设置消息体的长度。第 40 行把编码后的 `DefaultFullHttpResponse` 对象添加到编码结果列表中，由后续 Netty 的 HTTP 编码类进行二次编码。

4. HTTP+XML 应答消息解码

客户端接收到 HTTP+XML 应答消息后，对消息进行解码，获取 `HttpXmlResponse` 对象，源码如下。

代码清单 10-15 HTTP+XML 应答消息解码类 `HttpXmlResponseDecoder`

```

11. public class HttpXmlResponseDecoder extends
12.     AbstractHttpXmlDecoder<DefaultFullHttpResponse> {

```

Netty 权威指南

```
13.
14.     public HttpXmlResponseDecoder(Class<?> clazz) {
15.         this(clazz, false);
16.     }
17.
18.     public HttpXmlResponseDecoder(Class<?> clazz, boolean isPrintlog) {
19.         super(clazz, isPrintlog);
20.     }
21.
22.     @Override
23.     protected void decode(ChannelHandlerContext ctx,
24.         DefaultFullHttpResponse msg, List<Object> out) throws Exception {
25.         HttpXmlResponse resHttpXmlResponse = new HttpXmlResponse(msg, decode0(
26.             ctx, msg.content()));
27.         out.add(resHttpXmlResponse);
28.     }
29. }
```

第 25 行通过 `DefaultFullHttpResponse` 和 HTTP 应答消息反序列化后的 POJO 对象构造 `HttpXmlResponse`，并将其添加到解码结果列表中。

5. HTTP+XML 客户端开发

客户端的功能如下。

- (1) 发起 HTTP 连接请求；
- (2) 构造订购请求消息，将其编码成 XML，通过 HTTP 协议发送给服务端；
- (3) 接收 HTTP 服务端的应答消息，将 XML 应答消息反序列化为订购消息 POJO 对象；
- (4) 关闭 HTTP 连接。

基于它的功能定位，我们首先开始主程序的开发。

代码清单 10-16 HTTP+XML 客户端启动类 `HttpXmlClient`

```
23. public class HttpXmlClient {
24.
25.     public void connect(int port) throws Exception {
26.         // 配置客户端 NIO 线程组
27.         EventLoopGroup group = new NioEventLoopGroup();
28.         try {
```


第 10 章 HTTP 协议开发应用

```
29.     Bootstrap b = new Bootstrap();
30.     b.group(group).channel(NioSocketChannel.class)
31.         .option(ChannelOption.TCP_NODELAY, true)
32.         .handler(new ChannelInitializer<SocketChannel>() {
33.             @Override
34.             public void initChannel(SocketChannel ch)
35.                 throws Exception {
36.                 ch.pipeline().addLast("http-decoder",
37.                     new HttpResponseDecoder());
38.                 ch.pipeline().addLast("http-aggregator",
39.                     new HttpObjectAggregator(65536));
40.                 // XML 解码器
41.                 ch.pipeline().addLast(
42.                     "xml-decoder",
43.                     new HttpXmlResponseDecoder(Order.class,
44.                         true));
45.                 ch.pipeline().addLast("http-encoder",
46.                     new HttpRequestEncoder());
47.                 ch.pipeline().addLast("xml-encoder",
48.                     new HttpXmlRequestEncoder());
49.                 ch.pipeline().addLast("xmlClientHandler",
50.                     new HttpXmlClientHandle());
51.             }
52.         });
53.
54.     // 发起异步连接操作
55.     ChannelFuture f=b.connect(new InetSocketAddress(port)).sync();
56.
57.     // 等待客户端链路关闭
58.     f.channel().closeFuture().sync();
59. } finally {
60.     // 优雅退出, 释放 NIO 线程组
61.     group.shutdownGracefully();
62. }
63. }
64.
65. /**
66.  * @param args
67.  * @throws Exception
68.  */
69. public static void main(String[] args) throws Exception {
```

Netty 权威指南

```
70.     int port = 8080;
71.     if (args != null && args.length > 0) {
72.         try {
73.             port = Integer.valueOf(args[0]);
74.         } catch (NumberFormatException e) {
75.             // 采用默认值
76.         }
77.     }
78.     new HttpXmlClient().connect(port);
79. }
80. }
```

在 ChannelPipeline 中新增了 HttpResponseDecoder，它负责将二进制码流解码成为 HTTP 的应答消息；随后第 38 行新增了 HttpObjectAggregator，它负责将 1 个 HTTP 请求消息的多个部分合并成一条完整的 HTTP 消息；第 41~44 行将前面开发的 XML 解码器 HttpXmlResponseDecoder 添加到 ChannelPipeline 中，它有两个参数，分别是解码对象的类型信息和码流开关，这样就实现了 HTTP+XML 应答消息的自动解码。

第 45~46 行将 HttpRequestEncoder 编码器添加到 ChannelPipeline 中时，需要注意顺序，编码的时候是按照从尾到头的顺序调度执行的，它后面放的是我们自定义开发的 HTTP+XML 请求消息编码器 HttpXmlRequestEncoder。

最后是业务的逻辑编排类 HttpXmlClientHandle，我们继续分析它的实现。

代码清单 10-17 HTTP+XML 客户端业务逻辑编排类 HttpXmlClientHandle

```
7.  public class HttpXmlClientHandle extends
8.      SimpleChannelInboundHandler<HttpXmlResponse> {
9.
10.     @Override
11.     public void channelActive(ChannelHandlerContext ctx) {
12.         HttpXmlRequest request = new HttpXmlRequest(null,
13.             OrderFactory.create(123));
14.         ctx.writeAndFlush(request);
15.     }
16.
17.     @Override
18.     public void exceptionCaught(ChannelHandlerContext ctx, Throwable
cause) {
19.         cause.printStackTrace();
20.         ctx.close();

```

第 10 章 HTTP 协议开发应用

```
21.     }
22.
23.     @Override
24.     protected void messageReceived(ChannelHandlerContext ctx,
25.         HttpXmlResponse msg) throws Exception {
26.         System.out.println("The client receive response of http header is : "
27.             + msg.getHttpResponse().headers().names());
28.         System.out.println("The client receive response of http body is : "
29.             + msg.getResult());
30.     }
31. }
```

客户端的实现非常简单,第 12 行构造 `HttpXmlRequest` 对象,调用 `ChannelHandlerContext` 的 `writeAndFlush` 发送 `HttpXmlRequest`。

第 24~30 行用于接收服务端的应答消息,从接口看,它接收到的已经是自动解码后的 `HttpXmlResponse` 对象了;第 28~29 将应答 POJO 消息打印出来,可以与服务端发送的原始对象进行对比,两者的内容将完全一致。

最后,看下订购对象工厂类的实现。

代码清单 10-18 HTTP+XML 订购对象工厂类 `OrderFactory`

```
2. public class OrderFactory {
3.     public static Order create(long orderID) {
4.         Order order = new Order();
5.         order.setOrderNumber(orderID);
6.         order.setTotal(9999.999f);
7.         Address address = new Address();
8.         address.setCity("南京市");
9.         address.setCountry("中国");
10.        address.setPostCode("123321");
11.        address.setState("江苏省");
12.        address.setStreet1("龙眠大道");
13.        order.setBillTo(address);
14.        Customer customer = new Customer();
15.        customer.setCustomerNumber(orderID);
16.        customer.setFirstName("李");
17.        customer.setLastName("林峰");
18.        order.setCustomer(customer);
19.        order.setShipping(Shipping.INTERNATIONAL_MAIL);
20.        order.setShipTo(address);
}
```

Netty 权威指南

```
21.     return order;
22.     }
23. }
```

6 . HTTP+XML 服务端开发

HTTP 服务端的功能如下。

- (1) 接收 HTTP 客户端的连接；
- (2) 接收 HTTP 客户端的 XML 请求消息，并将其解码为 POJO 对象；
- (3) 对 POJO 对象进行业务处理，构造应答消息返回；
- (4) 通过 HTTP+XML 的格式返回应答消息；
- (5) 主动关闭 HTTP 连接。

下面我们首先看下服务端监听主程序的实现。

代码清单 10-19 HTTP+XML 服务端主程序 HttpXmlServer

```
22. public class HttpXmlServer {
23.     public void run(final int port) throws Exception {
24.         EventLoopGroup bossGroup = new NioEventLoopGroup();
25.         EventLoopGroup workerGroup = new NioEventLoopGroup();
26.         try {
27.             ServerBootstrap b = new ServerBootstrap();
28.             b.group(bossGroup, workerGroup)
29.                 .channel(NioServerSocketChannel.class)
30.                 .childHandler(new ChannelInitializer<SocketChannel>() {
31.                     @Override
32.                     protected void initChannel(SocketChannel ch)
33.                         throws Exception {
34.                         ch.pipeline().addLast("http-decoder",
35.                             new HttpRequestDecoder());
36.                         ch.pipeline().addLast("http-aggregator",
37.                             new HttpObjectAggregator(65536));
38.                         ch.pipeline()
39.                             .addLast(
40.                                 "xml-decoder",
41.                                 new HttpXmlRequestDecoder(
42.                                     Order.class, true));
```

第 10 章 HTTP 协议开发应用

```
43.         ch.pipeline().addLast("http-encoder",
44.             new HttpResponseEncoder());
45.         ch.pipeline().addLast("xml-encoder",
46.             new HttpXmlResponseEncoder());
47.         ch.pipeline().addLast("xmlServerHandler",
48.             new HttpXmlServerHandler());
49.     }
50. });
51.     ChannelFuture future = b.bind(new InetSocketAddress(port)).
sync();
52.     System.out.println("HTTP 订购服务器启动，网址是：" +
"http://localhost:"
53.         + port);
54.     future.channel().closeFuture().sync();
55. } finally {
56.     bossGroup.shutdownGracefully();
57.     workerGroup.shutdownGracefully();
58. }
59. }
60.
61. public static void main(String[] args) throws Exception {
62.     int port = 8080;
63.     if (args.length > 0) {
64.         try {
65.             port = Integer.parseInt(args[0]);
66.         } catch (NumberFormatException e) {
67.             e.printStackTrace();
68.         }
69.     }
70.     new HttpXmlServer().run(port);
71. }
72. }
```

HTTP 服务端的启动与之前一样，在此不再详述，我们具体看下编解码 handler 是如何设置的。

第 34~37 行用于绑定 HTTP 请求消息解码器；第 38~42 行将我们自定义的 `HttpXmlRequestDecoder` 添加到 HTTP 解码器之后；第 45~47 行添加自定义的 `HttpXmlResponseEncoder` 编码器用于响应消息的编码。

下面我们继续看 `HttpXmlServerHandler` 的实现。

Netty 权威指南

代码清单 10-20 HTTP+XML 服务端处理类 HttpXmlServerHandler

```
30. public class HttpXmlServerHandler extends
31.     SimpleChannelInboundHandler<HttpXmlRequest> {
32.
33.     @Override
34.     public void messageReceived(final ChannelHandlerContext ctx,
35.         HttpXmlRequest xmlRequest) throws Exception {
36.         HttpRequest request = xmlRequest.getRequest();
37.         Order order = (Order) xmlRequest.getBody();
38.         System.out.println("Http server receive request : " + order);
39.         dobusiness(order);
40.         ChannelFuture future = ctx.writeAndFlush(new HttpXmlResponse(null,
41.             order));
42.         if (!isKeepAlive(request)) {
43.             future.addListener(new GenericFutureListener<Future<? super
Void>>() {
44.                 public void operationComplete(Future future) throws Exception {
45.                     ctx.close();
46.                 }
47.             });
48.         }
49.     }
50.
51.     private void dobusiness(Order order) {
52.         order.getCustomer().setFirstName("狄");
53.         order.getCustomer().setLastName("仁杰");
54.         List<String> midNames = new ArrayList<String>();
55.         midNames.add("李元芳");
56.         order.getCustomer().setMiddleNames(midNames);
57.         Address address = order.getBillTo();
58.         address.setCity("洛阳");
59.         address.setCountry("大唐");
60.         address.setState("河南道");
61.         address.setPostCode("123456");
62.         order.setBillTo(address);
63.         order.setShipTo(address);
64.     }
65.
66.     @Override
67.     public void exceptionCaught(ChannelHandlerContext ctx, Throwable
cause)
```

```
68.         throws Exception {
69.     cause.printStackTrace();
70.     if (ctx.channel().isActive()) {
71.         sendError(ctx, INTERNAL_SERVER_ERROR);
72.     }
73. }
74.
75. private static void sendError(ChannelHandlerContext ctx,
76.     HttpResponseStatus status) {
77.     FullHttpResponse response = new DefaultFullHttpResponse(HTTP_1_1,
78.         status, Unpooled.copiedBuffer("失败: " + status.toString()
79.             + "\r\n", CharsetUtil.UTF_8));
80.     response.headers().set(CONTENT_TYPE, "text/plain; charset=UTF-8");
81.     ctx.writeAndFlush(response).addListener(ChannelFutureListener.
CLOSE);
82. }
83. }
```

通过 `messageReceived` 的方法入参 `HttpXmlRequest`，可以看出服务端业务处理类接收到的已经是解码后的业务消息了。第 37 行用于获取请求消息对象；随后将它打印出来，可以与客户端发送的原始消息进行对比；第 39 行对订购请求消息进行业务逻辑编排；第 40~47 行用于发送应答消息，并且在发送成功之后主动关闭 HTTP 连接。

第 70~71 行，在发生异常并且链路没有关闭的情况下，构造内部异常消息发送给客户端，发送完成之后关闭 HTTP 链路。

到此，HTTP+XML 的协议栈开发工作全部完成，下一个小节我们看下运行结果。

10.3.5 HTTP+XML 协议栈测试

本小节对前面几节开发的 HTTP+XML 协议栈进行测试。

首先对工程进行编译，然后执行 JiBx 的 Ant 脚本，对涉及的 POJO 对象进行二次编译，执行完成之后，首先运行 HTTP 服务端，然后再运行客户端，执行结果如下。

1. 服务端

服务端接收到的请求消息码流打印如下。

```
The body is : <?xml version="1.0" encoding="UTF-8"?>
```

Netty 权威指南

```
<order xmlns="http://phei.com/netty/protocol/http/xml/pojo" orderNumber=
"123" total="9999.999">
  <customer customerNumber="123">
    <firstName>李</firstName>
    <lastName>林峰</lastName>
  </customer>
  <billTo>
    <street1>龙眠大道</street1>
    <city>南京市</city>
    <state>江苏省</state>
    <postCode>123321</postCode>
    <country>中国</country>
  </billTo>
  <shipping>INTERNATIONAL_MAIL</shipping>
  <shipTo>
    <street1>龙眠大道</street1>
    <city>南京市</city>
    <state>江苏省</state>
    <postCode>123321</postCode>
    <country>中国</country>
  </shipTo>
</order>
```

服务端解码后的业务对象如下。

```
Http server receive request : Order [orderNumber=123, customer=Customer
[customerNumber=123, firstName=李, lastName=林峰, middleNames=null], billTo=
Address [street1=龙眠大道, street2=null, city=南京市, state=江苏省, postCode=
123321, country=中国], shipping=INTERNATIONAL_MAIL, shipTo=Address [street1=
龙眠大道, street2=null, city=南京市, state=江苏省, postCode=123321, country=中国],
total=9999.999]
```

2. 客户端

客户端接收到的响应消息体码流如下。

```
The body is : <?xml version="1.0" encoding="UTF-8"?>
<order xmlns="http://phei.com/netty/protocol/http/xml/pojo" orderNumber=
"123" total="9999.999">
  <customer customerNumber="123">
    <firstName>狄</firstName>
```



```

    <lastName>仁杰</lastName>
    <middleName>李元芳</middleName>
</customer>
<billTo>
  <street1>龙眼大道</street1>
  <city>洛阳</city>
  <state>河南道</state>
  <postCode>123456</postCode>
  <country>大唐</country>
</billTo>
<shipping>INTERNATIONAL_MAIL</shipping>
<shipTo>
  <street1>龙眼大道</street1>
  <city>洛阳</city>
  <state>河南道</state>
  <postCode>123456</postCode>
  <country>大唐</country>
</shipTo>
</order>

```

解码后的响应消息如下。

```

The client receive response of http body is : Order [orderNumber=123,
customer=Customer [customerNumber=123, firstName= 狄 , lastName= 仁 杰 ,
middleNames=[ 李元芳]], billTo=Address [street1=龙眼大道, street2=null, city=洛
阳, state=河南道, postCode=123456, country=大唐], shipping=INTERNATIONAL_MAIL,
shipTo=Address [street1= 龙眼大道 , street2=null, city= 洛阳 , state= 河南道 ,
postCode=123456, country=大唐], total=9999.999]

```

测试结果表明，HTTP+XML 协议栈功能正常，达到了设计预期。

10.3.6 小结

需要指出的是，尽管本章节开发的 HTTP+XML 协议栈是个高性能、通用的协议栈，但是，作为例程我们忽略了一些异常场景的处理、可扩展性的 API 和一些配置能力。所以，如果你要打算在商用项目中使用本章节开发的 HTTP+XML 协议栈，仍需要做一些产品化的完善工作。

Netty 权威指南

10.4 总结

本章节重点介绍了 HTTP 协议以及如何使用 Netty 的 HTTP 协议栈开发基于 HTTP 的应用程序,最后通过 HTTP+XML 协议栈的开发向读者展示了如何基于 Netty 提供的 HTTP 协议栈做二次定制开发。

本章节的 HTTP+XML 协议栈在实际项目中非常有用,如果读者打算以它为基础进行商业应用,需要补齐一些产品化的能力,例如配置能力、容错能力、更丰富的 API。