第 19 章　Spring OXM

本章从 XML 各种解析技术发展历程讲起，并介绍另外一些主流 O/X Mapping 组件的使用方法。通过这些实例讲解，读者可以快速了解目前各种 O/X Mapping 技术。本章筛选了目前一些流行的 O/X Mapping 组件如 XStream、Castor、JiBX、JAXB 等，从独立使用到与 Spring 整合逐步演进，逐步揭开各 O/X Mapping 组件的神秘面纱。通过本章的学习，读者可以根据需要，选用合适的 O/X Mapping 组件来处理对象 XML 之间转换，为开发 Web Services 应用打下良好的基础。

本章主要内容：

- 认识 XML 解析技术
- XML 处理利器：XStream
- 基于 Castor 对象 XML 映射
- 基于 JiBX 对象 XML 映射
- 基于 JAXB 对象 XML 映射
- 与 Spring OXM 整合

本章亮点：

- 各种 XML 解析技术特点分析
- XML 处理利器：XStream 及其他 O/X Mapping 组件讲解
- 与 Spring OXM 整合

19.1 认识 XML 解析技术

17.1.1 什么是 XML

XML（Extensible Markup Language）即可扩展标记语言，从 1996 年开始有其雏形，并向 W3C（全球信息网联盟）提案，并在 1998 年 2 月发布为 W3C 的标准（XML 1.0）。XML 的前身是 SGML（The Standard Generalized Markup Language），是 IBM 从 20 世纪 60 年代就开始发展的 GML（Generalized Markup Language）标准化后的名称。其目的是为了促进 Internet 上结构化文档的交换。简单地说，XML 是一组规则和准则的集合，用来描述结构化数据。

XML 为描述结构良好的文档提供了一整套灵活的语法。正因为它的这种灵活性，我们需要一些方法来验证 XML 文档是否与我们预计的格式保持一致。于是人们就逐步提出 DTD 和 XML Schema。

DTD 是一套关于标记符的语法规则，它是 XML 1.0 规范的一部分，是 XML 文件的验证机制，属于 XML 文件组成的一部分。DTD 是一种保证 XML 文档格式正确的有效方法，可以通过比较 XML 文档和 DTD 文件来看文档是否符合规范，以及元素和标签使用是否正确。

XML Schema 指定 XML Schema 定义语言，该语言提供了描述 XML 1.0 文档结构和限制其内容的工具，其中包括那些利用 XML Namespace 的工具。模式语言自身用 XML 1.0 表示并使用名称空间，它在很大程度上重构了 XML 1.0 DTD 的能力，并在一定程度上解决了 DTD 许多局限性，如：

- 不支持名称空间；
- 对模块化和重用的支持非常有限；
- 不支持对声明的扩展或继承；
- 编写、维护和读取大型 DTD 以及定义系列相关模式都很困难；
- 没有嵌入式、结构化自我文档编制；
- 内容和属性声明不能依靠属性或元素上下文。

XML 不仅仅是一种标记语言，而是一系列的技术，如 DTD、XSD、DOM、SAX、XSL、XLink、XPointer、SMIL 等的集合体。这一技术家族为我们开发可扩展性和互操作性的软件提供了一种解决方案。目前，广泛应用于系统配置、数据存储以及数据交换的格式等，它是 Web 服务基础，也是现代面向服务的架构（Service-Oriented Architecture，SOA）设计模式的基础。

19.1.2 XML 的处理技术

为了有效使用 XML，需要通过一个 XML 处理器或是 XML API 来访问其数据。目前 JAXP1.6(JSR 206)的两种处理 XML 文档的方法已经得到了广泛应用，它们是 DOM（Document Object Model）和 SAX（Simple API for XML）。

DOM 文档对象模型是一种通过编程方式对 XML 文档中数据及结构进行访问的标准，基于 XML 文档在内存中的树状结构。当一个 XML 文件被装入处理器时，内存中建立起一棵相应的树。DOM 还定义了用来遍历一棵 XML 树以及管理各个元素、值和属性的编程接口（包括方法和属性的名字）。

DOM 标准的一个主要不足在于将整个 XML 文档装入内存所引起的巨大内存开销。当文件的数据量非常大时，这会带来很大的性能瓶颈。于是人们就开始创立一种新的标准，这就是 SAX。

SAX 是一种非常简单的 XML API，它允许开发者使用事件驱动的 XML 解析。与 DOM 不同，SAX 并不要求将整个 XML 文件一起装入内存。它的想法十分简单，一旦 XML 处理器完成对 XML 元素的操作，它就立刻调用一个自定义事件处理器及时处理这个元素及相关数据。

虽然 SAX 解决了 DOM 速度慢、内存占用大的问题，但在灵活性方面受到很大制约，如无法随机访问文档。于是又一种新的基于流的 Stream API for XML（简称 StAX）逐步出现在人们的视野中，它不仅提高了 XML 的处理速度，而且又较好兼顾了灵活性。StAX 是 JSR 173 标准，目前已经加入到 Java 6.0 的 JAXP 1.4 里面。

StAX 如其名称所暗示的那样，把重点放在流上。实际上，StAX 与其他方法的区别就在于应用程序能够把 XML 作为一个事件流来处理。将 XML 作为一组事件来处理的想法并不新颖（事实上 SAX 已经提出来了），但 StAX 并不使用 SAX 的“推”模型，而是使用“拉”模型进行事件处理。此外，StAX 解析器也不使用回调机制，而是根据应用程序的要求返回事件。StAX 还提供了用户友好的 API，用于读入和写出。

StAX 实际上包括两套处理 XML 的 API，分别提供了不同程度的抽象。基于指针的 API 允许应用程序把 XML 作为一个标记（或事件）流来处理。应用程序可以检查解析器的状态，获得解析的上一个标记信息，然后再处理下一个标记，依此类推。这是一种低层 API，尽管效率高，但是没有提供底层 XML 结构的抽象。较为高级的基于迭代器的 API 允许应用程序把 XML 作为一系列事件对象来处理，每个对象和应用程序交换 XML 结构的一部分。应用程序只需要确定解析事件的类型，将其转换成对应的具体类型，然后利用其方法获得属于该事件的信息。

DOM、SAX、StAX 技术都是从 XML 的角度来处理文档和建立模型，这对于只关注文档 XML 结构的应用程序来说是适用的，但是有很多应用程序仅仅将 XML 作为数据交换的媒介，更多关注的是文档数据本身，为此人们又提出了一种 XML 数据绑定技术，可以使应用程序在很大程度上忽略 XML 文档的实际结构，而直接使用文档的数据内容。

数据绑定是指将数据从一些存储媒介（如 XML 文档和数据库）中取出，并通过程序表示这些数据的过程，即把数据绑定到虚拟机能够理解并且可以操作的某种内存结构中。数据绑定并不是一个新鲜的概念，它在关系数据库上早已得到了广泛的应用，如 Hibernate 就是针对数据库的轻量级数据绑定框架。而针对 XML 数据绑定的 Castor 框架在 2000 年就已经出现，目前已经涌现出了许多类似的框架，如 JAXB、JiBX、Quick 和 Zeus 等。

19.2 XML 处理利器：XStream

19.2.1 XStream 概述

XStream 是一套简洁易用的开源类库，用于将 Java 对象序列化为 XML 或者将 XML 反序列化为 Java 对象，是 Java 对象和 XML 之间一个双向转换器。2004 年 1 月份发布了第一个版本 XStream 0.3，目前最新版本是 2016 年 3 月份发布的 Xstream 1.4.9。

XStream 主要特点

- 灵活易用：提供了简单、灵活、易用的统一接口，用户无须了解底层细节。
- 无须映射：大多数对象都可以在无须映射的情况下进行序列化与反序列化的操作。
- 高速稳定：解析速度快，占用内存少。
- 灵活转换：转换策略是可以定制的，允许自定义类型存储为指定 XML 格式。
- 易于集成：通过实现特定的接口，可以直接与其他任何树型结构进行序列化与反序列化操作。

XStream 架构组成

- **Converters（转换器）**

当 XStream 遇到需要转换的对象时，它会委派给合适的转换器实现，XStream 为通用类型提供了多种转换器实现，包括基本数据类型、String、Collections、Arrays、null、Date 等。

- **IO（输入/输出）**

XStream 是通过接口 HierarchicalStramWriter 和 HierarchialStreamReader 从底层 XML 数据中抽象而来的，分别用于序列化和反序列化操作。

- **Context（上下文引用）**

在 XStream 序列化或反序列化对象时，它会创建两个类 MarshallingContext 和 UnmarshallingContext，由它们来处理数据并委派合适的转换器。XStream 提供了三种上下文的默认实现，它们之间存在细微的差别。默认值可以通过方法 XStream.setMode()调整，可选值为：

1）XStream.XPATH_REFERENCES：（默认的）通过 XPath 引用来标识重复的引用

2）XStream.ID_REFERENCES：使用 ID 引用来标识重复的引用

3）XStream.NO_REFERENCES：对象作为树型结构，重复的引用被视为两个不同的对象，循环引用会导致异常产生，这种模式速度快，占用内存少。

- **Facade（统一入口）**

作为 XStream 统一入口点，它将上面所提及的重要组件集成在一起，以统一的接口开放出来。

19.2.2 快速入门

本节通过一个实例开始 Xstream 的应用之旅，采用 User 用户领域对象及 LoginLog 登

录日志领域对象作为 XStream 实例转换对象，如代码清单 19-1 及 19-2 所示。

代码清单 19-1　User.java

```
package com.smart.oxm.domain;
…
public class XStreamSample {
        private int userId;
        private String userName;
        private String password;
        private int credits;
        private String lastIp;
        private Date lastVisit;
        private List logs;

        //set、get方法省略
}
```

代码清单 19-2　LoginLog.java

```
package com.smart.oxm.domain;
…
public class XStreamSample {
        private int loginLogId;
        private int userId;
        private String ip;
        private Date loginDate;

        //set、get方法省略
}
```

在开始实例的开发之前，要先将 XStream 的依赖加入到 pom.xml 文件中，本实例采用的是目前最新版本 xstream 1.4.9。如下所示：

```
<dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.4.9</version>
</dependency>
```

接下来我们就开始使用 XStream 进行对象与 XML 之间的互换，如代码清单 19-3 所示。

代码清单 19-3　XStreamSample.java

```
package com.smart.oxm.xstream;
…
public class XStreamSample {
        private static XStream xstream;
        static{
            //① 创建XStream实例并指定一个XML解析器
            xstream = new XStream(new DomDriver());
        }
```

```
    //初始化转换对象
    public static User getUser(){
        LoginLog log1 = new LoginLog();
        log1.setIp("192.168.1.91");
        log1.setLoginDate(new Date());
        User user = new User();
        user.setUserId(1);
        user.setUserName("xstream");
        user.addLoginLog(log1);
        return user;
    }

    //② Java对象转化为XML
    public static void objectToXML()throws Exception{

        //②-1 获取转换的User对象实例
        User user = getUser()

        //②-2 实例化一个文件输出流
        FileOutputStream outputStream = new FileOutputStream("out/XStreamSample.xml");
        //②-3 将User对象转换为XML，并保存到指定文件
        xstream.toXML(user, outputStream);
    }

    //③ XML转化为Java对象
    public static void XMLToObject()throws Exception{

        //③-1 实例化一个文件输入流
        FileInputStream inputStream= new FileInputStream("out/XStreamSample.xml ");

        //③-2 将XML文件输入流转化为对象
        User user = (User) xstream.fromXML(inputStream);

        for(LoginLog log : user.getLogs()){
          if(log!=null){
                System.out.println("访问IP: " + log.getIp());
                System.out.println("访问时间: " + log.getLoginDate());
            }
        }
    }
  ...
}
```

在①处，创建一个 XStream 实例，并指定一个 DOM XML 解析器。如果不指定一个 XML 解析器，则 XStream 会采用默认 XPP（XML Pull Parser，一种高速解析 XML 文件的方式，速度要比传统方式快很多）解析器来解析 XML 文件。

在②-1 处，首先实例化需要转换为 XML 的类，如实例中的 User、LoginLog。在②-2 处，创建一个用于输出 XML 文件的文件输出流。在②-3 处，调用 XStream#toXML()方法

将 User 实例转换为 XML 格式，并转出到指定的文件流中。同理，如果将 XML 文件转换为对象，需要先读取要转换的 XML 文件到文件流中，如③-1 所示，然后调用 XStream #fromXML()方法将输入文件流转换为对象。在 IDE 工具中，执行当前实例，在类路径下生一个 XML 文件，如下所示：

```
<com.smart.oxm.domain.User>
  <userId>1</userId>
  <userName>xstream</userName>
  <credits>0</credits>
  <logs>
    <com.smart.oxm.domain.LoginLog>
      <loginLogId>0</loginLogId>
      <userId>0</userId>
      <ip>192.168.1.91</ip>
      <loginDate>2016-03-06 21:05:37.879 UTC</loginDate>
    </com.smart.oxm.domain.LoginLog>
    <com.smart.oxm.domain.LoginLog>
      <loginLogId>0</loginLogId>
      <userId>0</userId>
      <ip>192.168.1.92</ip>
      <loginDate>2016-03-06 21:05:37.879 UTC</loginDate>
    </com.smart.oxm.domain.LoginLog>
  </logs>
</com.smart.oxm.domain.User>。
```

19.2.3　使用 XStream 别名

在默认情况下，Java 对象到 XML 的映射，是 Java 对象属性名对应 XML 的元素名，Java 类的全名对应 XML 根元素的名字。在实际的应用中，如果 XML 和 Java 类都已经存在相应的名字，在进行转换时，需要设置别名进行映射。

XStream 别名配置包含三种情况：

- 类别名，用 alias(String name, Class type)；
- 类成员别名，用 aliasField(String alias, Class definedIn, String fieldName)；
- 类成员作为属性别名，用 aliasAttribute(Class definedIn, String attributeName, String alias)，单独命名没有意义，还要通过 useAttributeFor(Class definedIn, String fieldName)应用到某个类上。

从上一个实例生成的 XML 文件来看，生成 XML 元素结构不是很友好。接下来我们就使用 XStream 提供的别名机制，来修饰生成 XML 元素的结构，如代码清单 19-4 所示。

代码清单 19-4　XStreamAliasSample.java

```
package com.smart.oxm.xstream.alias;
…
public class XStreamAliasSample{
…

    static{
```

```
        ...

        //① 设置类别名,默认为当前类名称加上包名
        xstream.alias("loginLog", LoginLog.class);
        xstream.alias("user", User.class);

        //② 设置类成员别名
        xstream.aliasField("id",User.class,"userId");

        //③ 把LoginLog的userId属性视为 XML属性,默认为XML的元素
        xstream.aliasAttribute(LoginLog.class, "userId","id");
        xstream.useAttributeFor(LoginLog.class, "userId");

        //④ 去掉集合类型生成XML的父节点,即忽略 XML中的 <logs></logs>标记
        xstream.addImplicitCollection(User.class, "logs");
    }
    ...
}
```

在①处，通过 XStream# alias ()方法来设置类别名，默认为当前类全限定名。在②处，通过 XStream#aliasField ()方法将 User 类的 "userId" 属性设置为"id"。在③处，通过 XStream#aliasAttribute ()和 XStream#useAttributeFor()方法将 LoginLog 类的 "userId" 属性设置为"id"，并设置为 LoginLog 元素的属性，默认为 LoginLog 元素的子元素。在④处，通过 XStream#addImplicitCollection()方法删除集合节点 logs，即忽略 XML 中的<logs></logs>标记。

在 IDE 工具中，执行当前实例，在类路径下生一个 XML 文件，如下所示：

```
<user>
  <id>1</id>
  <userName>xstream</userName>
  <credits>0</credits>
  <loginLog id="0">
    <loginLogId>0</loginLogId>
    <ip>192.168.1.91</ip>
    <loginDate>2016-03-17 05:12:39.826 UTC</loginDate>
  </loginLog>
   …
</user>
```

19.2.4 XStream 转换器

在开发过程中，有时可能需要转换一些自定义类型，此时默认的映射方式可能无法满足需要。不用担心，XStream 已经提供了丰富的扩展点，你可以实现自己的转换器，然后调用 registerConverter()方法注册自定义的转换器。实现自定义的转换器很简单，只要实现 XStream 提供的 Converter 接口并实现其方法即可。

上一个实例已成功应用 XStream 进行对象与 XML 的相互转换，并应用 XStream 提供的别名机制设置生成 XML 的结构。下面我们使用 XStream 提供的转换器扩展接口，对生

成的 XML 文件继续优化。

首先我们需要实现 Converter 接口来编写一个日期转换器，具体的实现如代码清单 19-5 所示。

代码清单 19-5　DateConverter.java

```
package com.smart.oxm.xstream.converters;
…
public class DateConverter implements Converter {
    private Locale locale;
    public DateConverter(Locale locale) {
        super();
        this.locale = locale;
    }
    //①实现该方法，判断要转换的类型
    public boolean canConvert(Class clazz) {
        return Date.class.isAssignableFrom(clazz);
    }

    //②实现该方法，编写Java对象到XML转换逻辑
    public void marshal(Object value, HierarchicalStreamWriter writer,
                        MarshallingContext context){
        DateFormat formatter = DateFormat.getDateInstance(DateFormat.DATE_FIELD,
                                                          this.locale);
        writer.setValue(formatter.format(value));
    }

    //③实现该方法，编写XML到Java对象转换逻辑
    public Object unmarshal(HierarchicalStreamReader reader,
    UnmarshallingContext context) {
        GregorianCalendar calendar = new GregorianCalendar();
        DateFormat formatter = DateFormat.getDateInstance(DateFormat.DATE_FIELD,
                                                          this.locale);
        try {
            calendar.setTime(formatter.parse(reader.getValue()));
        } catch (ParseException e) {
            throw new ConversionException(e.getMessage(), e);
        }
        return calendar.getGregorianChange();
    }
}
```

在①处，通过实现 ConverterMatcher#canConvert ()方法来判断转换逻辑。在②处，通过实现 Converter#marshal()方法，完成对象编组（对象转换为 XML）操作时处理逻辑。在③处，通过实现 Converter#unmarshal ()方法，完成对象反编组（XML 转换为对象） 操作时处理逻辑。编写好转换器之后，剩下的工作就是调用 XStream#registerConverter()方法注册自定义的日期转换器即可。

在 IDE 工具中，执行当前实例，将会在类路径下生成一个 XML 文件，如下所示：

```
<user>
    <userId>1</userId>
    <userName>xstream</userName>
    <credits>0</credits>
    <logs>
        <loginLog>
        <loginLogId>0</loginLogId>
        <userId>0</userId>
        <ip>192.168.1.91</ip>
        <loginDate> 2016年3月6日 星期六 </loginDate>
        </loginLog>
        …
    </logs>
</user>
```

19.2.5 XStream 注解

XStream 不但可以通过编码的方式对 XML 进行转换，而且还支持基于注解的方式进行转换。下面我们就使用 XStream 提供的注解（如表 19-1 所示），来改造上面实例。首先需要对 User、LoginLog 添加相应的注解，如代码清单 19-6 所示。

代码清单 19-6 User.java

```
package com.smart.oxm.xstream.annotations;
…
@XStreamAlias("user")
public class User {
    @XStreamAsAttribute
    @XStreamAlias("id")
     private int userId;

     @XStreamAlias("username")
     private String userName;

     @XStreamAlias("password")
     private String password;

     @XStreamAlias("credits")
     private int credits;

     @XStreamAlias("lastIp")
     private String lastIp;

    @XStreamConverter(DateConverter.class)
     private Date lastVisit;

     @XStreamImplicit
     private List logs;
```

```
    …
}
```

表 19-1　XStream 常用注解

注　　解	说　　明	作用目标
@XStreamAlias	别名注解	类，字段
@XStreamAsAttribute	转换成属性	字段
@XStreamOmitField	忽略字段	字段
@XStreamConverter	注入转换器	对象
@XStreamImplicit	隐式集合	集合字段

其中@XStreamAlias 为别名注解，一般作用于目标类或字段，如实例中的@XStreamAlias("user")、@XStreamAlias("id")。@XStreamConverter 注解用于注入自定义的转换器，如实例中的@XStreamConverter(DateConverter.class)，注入一个日期转换器。对于集合类型，可以使用@XStreamImplicit 注解来隐藏，其作用与 XStream#addImplicitCollection()方法一样。@XStreamAsAttribute 注解将 Java 对象属性映射为 XML 元素的一个属性。@XStreamOmitField 注解标注 Java 对象的属性将不出现在 XML 中。LoginLog 对象添加注解方式与 User 一样，这里不再阐述，下面来看一下，如何应用 XStream 提供的注解进行 Java 对象与 XML 的相互转换，如代码清单 19-7 所示。

代码清单 19-7　XStreamAnnotationSample.java

```
package com.smart.oxm.xstream.annotations;
package com.smart.oxm.xstream.annotations;
import com.thoughtworks.xstream.XStream;
import com.thoughtworks.xstream.io.xml.DomDriver;
…
public class XStreamAnnotationSample {
    private static XStream xstream;
    …
    static{
        xstream = new XStream(new DomDriver());

        //① 实现该方法，判断要转换的类型
        xstream.processAnnotations(User.class);
        xstream.processAnnotations(LoginLog.class);
        //② 自动加载注解Bean
        //xstream.autodetectAnnotations(true);
    }

   //Java对象转化为XML
   public static void objectToXml()throws Exception{
        User user = getUser();
        FileOutputStream fs = new FileOutputStream("out/XStreamAnnotationSample.xml");
        OutputStreamWriter writer = new OutputStreamWriter(fs, Charset.forName("UTF-8"));
        xstream.toXML(user, writer);
   }
```

```
    ...
}
```

要启用 XStream 提供的注解功能，需要在执行对象与 XML 转换之前，先注册标注了 XStream 注解的 Java 对象，如实例中在①处，通过 XStream#processAnnotations ()方法手工注册 User、LoginLog 对象。XStream 除了手工注册，还提供一个自动检测标注了 XStream 注解的 Java 对象的方法 XStream#autodetectAnnotations()，如实例中②所示。自动检测方法不仅使用方便，而且还提供了缓存机制来缓存所有标注了 XStream 注解的 Java 对象。

19.2.6 流化对象

XStream 为 java.io.ObjectInputStream 和 java.io.ObjectOutputStream 提供替代的实现，允许以对象流方式进行 XML 序列化或反序列化操作。这对于处理集合对象非常有用（List<User> users），在内存中只保留一个 User 对象流。很显然，我们应该使用基于流而非 DOM 的 XML 解析器读取 XML，以提高性能。

创建一个输出流，至于怎么输出可以使用多种方法，其原理是一样的。在这里就不得不提到 HierarchicalStreamWriter，HierarchicalStreamWriter 是一个接口，从字面意思上来说它是有层级关系的输出流。XStream 默认提供几个常用的实现类用于输出，如 CompactWriter、PrettyPrintWriter。下面我们应用 XStream 流化对象来处理 XML 序列化及反序列化，如代码清单 19-8 所示。

代码清单 19-8 ObjectStreamSample.java

```
package com.smart.oxm.xstream;
...
public class ObjectStreamSample {
    private static   XStream xstream;
    static{
        xstream = new XStream();
    }

    //Java对象转化为XML
    public void objectToXml() throws Exception{
        //① 实例化序列化对象
        User user = getUser();

        //② 创建一个PrintWriter对象，用于输出
        PrintWriter pw=new PrintWriter("out/ObjectStreamSample.xml");

        //③ 选用一个HierarchicalStreamWriter的实现类来创建输出
        PrettyPrintWriter ppw=new PrettyPrintWriter(pw);
        //CompactWriter cw=new CompactWriter(pw);

        //④ 创建对象输出流
        ObjectOutputStream out = xstream.createObjectOutputStream(ppw);
        out.writeObject(user);
        out.close();
```

```
    }

    //XML转化为Java对象
    public User xmlToObject()throws Exception{

        //⑤ 通过对象流进行输入操作
        FileReader reader=new FileReader("out/ObjectStreamSample.xml");
        BufferedReader bufferedReader =new BufferedReader(reader);

        //⑥ 创建对象输入流
        ObjectInputStream input = xstream.createObjectInputStream(bufferedReader);

        //⑦ 从XML文件中读取对象
        User user=(User)input.readObject();
        return user;
    }
  ...
}
```

通过对象流进行输出操作，首先需要实例化转换的对象，如实例中的①处所示。然后，在②处，创建一个 PrintWriter 对象，将对象序列化到指定的 XML 文件中。在③处，选用一个 HierarchicalStreamWriter 的实现类 PrettyPrintWriter 来创建输出，也可以采用 CompactWriter 实现类。CompactWriter 与 PrettyPrintWriter 的区别在于，用 CompactWriter 方法输出的为连续的没有分隔的 XML 文件，而用 PrettyPrintWriter 方法输出的为有分隔有一定格式的 XML 文件。在④处，调用 XStream#createObjectOutputStream()方法创建对象输出流。

通过对象流进行输入操作，需要使用 FileReader 和 BufferedReader 读取指定的 XML 文件，如实例中的⑤处所示。在⑥处，调用 XStream#createObjectInputStream()方法创建对象输入流，最后从该输入流读取对象。

19.2.7　持久化 API

如果需要将一个集合中所有对象持久化到文件系统中，通常我们会使用 java.io 的 API 将集合的对象逐个输入到文件中，虽然这个方法很简单，但过程比较复杂。XStream 提供相关集合类接口实现类，如 XmlArrayList、XmlSet、XmlMap，用一个简单的方法就可以将集合中的每个对象持久化到文件中。下面我们应用 XmlArrayList 来持久化一个集合中的所有对象，具体实现如代码清单 19-9 所示。

代码清单 19-9　PersistenceSample.java

```
package com.smart.oxm.xstream. persistence;
...
public class PersistenceSample {
    ...
    public void persist() {

        //① 实例化需要持久化对象
```

```
            users = new ArrayList<User>();
            User user = new User();
            user.setUserId(1);
            user.setCredits(10);
            user.setUserName("tom");
            user.setPassword("123456");
            users.add(user);

            //② 创建持久化策略
            File file = new File("out");
            PersistenceStrategy strategy = new FilePersistenceStrategy(file);

            //③ 持久化集合对象
            List list = new XmlArrayList(strategy);
            list.addAll(users);
        }
        ...
}
```

XStream 为 XmlArrayList、XmlSet、XmlMap 实现类提供统一创建接口，在创建 XmlArrayList（及相关的集合）时，需要指定一个持久化策略 PersistenceStrategy。XStream 提供一个持久化到文件的策略 FilePersistenceStrategy。这个策略将集合中每个对象持久化到指定目录不同的文件中，如 int@0.xml、int@1.xml。

19.2.8 额外功能：处理 JSON

目前，在 Web 开发领域，主要的数据交换格式有 XML 和 JSON，相信每个 Web 开发者对两者都不会感到陌生。

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，易于阅读和编写，同时也易于机器解析和生成。它是基于 1999 年颁布的 ECMA-262 标准的 JavaScript 语言的一个子集。 JSON 采用完全独立于语言的文本格式，但是也使用了类似于高级语言的一些习惯（如 Java、C#等），这些特性使 JSON 成为理想的数据交换语言。虽然目前 XML 是业界数据交换标准，在可扩展性、数据类型的描述方面有着明显的优势，但相比于 JSON 这种轻量级的数据交换格式，XML 显得有些“笨重”。

对于大多数 Web 应用来说，他们根本不需要复杂的 XML 来传输数据，XML 宣称的扩展性在此没有多大优势。大多数 AJAX 应用直接使用 JSON 发送和接收数据，以构建动态页面的内容，非常灵活易用。如果使用 XML，程序代码将会复杂不少。当然，在 Web Service 领域中 XML 目前仍有不可动摇的地位。

在 Java 的世界里有不少处理 Java 对象与 JSON、XML 相互转换的组件，如 JSON-lib、XMLBeans。但同时支持两种数据格式转换的轻量级组件并不多，XStream 组件就是少数中的佼佼者。

XStream 的 JettisonMappedXmlDriver 和 JsonHierarchicalStreamDriver 都可以很好地完成 Java 对象和 JSON 的相互转换工作。值得注意的是，使用 JettisonMappedXmlDriver 时，必须事先将 jettison 依赖包添加到工程 pom 中。如下所示：

```
<dependency>
        <groupId>org.codehaus.jettison</groupId>
        <artifactId>jettison</artifactId>
        <version>1.3.2</version>
 </dependency>
```

下面我们应用这两个驱动来将对象转换为 JSON，如代码清单 19-10 所示。

代码清单 19-10　XStreamJSONSample.java

```
package com.smart.oxm.xstream.json;
…
public class XStreamJSONSample {
    private XStream xstream;
    …
    //①连续的没有分隔JSON串
    public static void toJsonByJettisonMappedXmlDriver()throws Exception {
        User user = getUser();
          FileOutputStream outputSteam = new
                  FileOutputStream("out/JettisonMappedSample.json");
          OutputStreamWriter writer = new
                              OutputStreamWriter(outputSteam,Charset.forName("UTF-8"));
          xstream = new XStream(new JettisonMappedXmlDriver());
          xstream.setMode(XStream.NO_REFERENCES);
          xstream.alias("user", User.class);
          xstream.toXML(user,writer);
    }

    //② 格式化良好的JSON串
    public void toJsonByJsonHierarchicalStreamDriver()throws Exception {
        User user = getUser();
        FileOutputStream outputSteam = new
                              FileOutputStream("out/JsonByJsonHierarchicalSample.json");
         OutputStreamWriter writer = new
                             OutputStreamWriter(outputSteam, Charset.forName("UTF-8"));
         xstream = new XStream(new JsonHierarchicalStreamDriver());
         xstream.alias("user", User.class);
         xstream.toXML(user,writer);
    }
    …
}
```

JSON 的转换和 XML 的转换用法一样，只要创建 XStream 实例时，传递一个 XML 到 JSON 映射转换的驱动器如 JettisonMappedXmlDriver、JsonHierarchicalStreamDriver 即可。使用 JettisonMappedXmlDriver 驱动生成的为连续的没有分隔 JSON 串，而使用 JsonHierarchical StreamDriver 驱动生成一个格式化后的 JSON 串。如果将 JSON 转换为对象，只能用 JettisonMappedXmlDriver 驱动。

在 IDE 工具中，执行当前实例，在类路径下生成两个 JSON 文件，使用 JettisonMapped

XmlDriver 输出的 JSON 如下所示：

{"user":{"userId":1,"userName":"xstream","credits":0,"logs":[{"com.smart.oxm.domain.LoginLog":[{"loginLogId":0,"userId":0,"ip":"192.168.1.91","loginDate":"2016-03-13 15:56:30.517 UTC"},{"loginLogId":0,"userId":0,"ip":"192.168.1.92","loginDate":"2016-03-13 15:56:30.517 UTC"}]}]}}

而使用 JsonHierarch icalStreamDriver 输出的 JSON 如下所示：


```
{"user": {
   "userId": 1,
   "userName": "xstream",
   "credits": 0,
   "logs": [
      {
         "loginLogId": 0,
         "userId": 0,
         "ip": "192.168.1.91",
         "loginDate": "2016-03-13 15:41:27 "
      },

      {
         "loginLogId": 0,
         "userId": 0,
         "ip": "192.168.1.92",
         "loginDate": "2016-03-13 15:41:27 "
      }
   ]
}}
```


19.3 其他常见 O/X Mapping 开源项目

19.3.1 JAXB

JAXB（Java Architecture for XML Binding）是一个业界的标准，是一项可以根据 XML Schema 产生 Java 类的技术。JAXB 也提供了将 XML 文件反向生成 Java 对象树的方法，并能将 Java 对象树的内容重新写到 XML 文件。从另一方面来讲，JAXB 提供了快速而简便的方法将 XML 模式绑定到 Java 对象，从而使得开发者在 Java 应用程序中能方便地结合 XML 数据和处理函数。

利用 JAXB 技术，我们不需要深入 XML 编程细节，就能在 Java 应用程序中灵活操作 XML 数据，而且可以充分使用 XML 的优势而不用依赖于复杂的 XML 处理模型，如 SAX 或 DOM 等。JAXB 隐藏了底层操作细节，并且取消了 SAX 和 DOM 中没用的关系，生成的 JAXB 类仅描述原始模型中定义的关系。其结果是整合高度可移植 Java 代码和高度可移植的 XML 文档。通过这些代码可创建灵活、轻便的应用程序和 Web 服务。

下面我们使用 JAXB 组件处理 Java 对象与 XML 之间的相互转换，采用 User、LoginLog 两实体来创建 Schema 文档，通过 Schema 文档创建相应的 Java 源代码。在进行 Java 对象编组与反编组操作之前，需要编写 XML Schema 文档，并通过模式文档生成相应的 Java

代码。

编写 XML Schema

XML Schema 文件是一个 XML 的约束文件，它定义了 XML 文件的结构和元素，以及对元素和结构的相关约束，JAXB 使用这个 Schema 文件生成相应的 Java 代码，User 模式结构图如图 19-1 所示，文档结构代码如代码清单 19-11 所示。

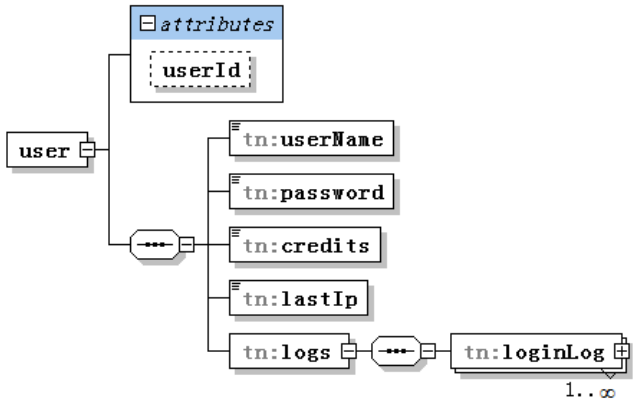

图 19-1　User 模式结构图

代码清单 19-11　user.xsd

```
<?XML version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" version="1.0"
 XMLns:xs="http://www.w3.org/2001/XMLSchema"
 XMLns:JAXB="http://Java.sun.com/xml/ns/JAXB"
      XMLns:tn="http://www.smart.oxm.com/domain/JAXB"
      targetNamespace="http://www.smart.oxm.com/domain/JAXB"
      JAXB:version="2.0" >
      <xs:annotation>
            <xs:appinfo>
                  <JAXB:globalBindings>
                        <JAXB:JavaType name="Java.util.Calendar" XMLType="xs:date"
                              parseMethod="javax.xml.bind.DatatypeConverter.parseDate"
                              printMethod="javax.xml.bind.DatatypeConverter.printDate" />
                  </JAXB:globalBindings>
            </xs:appinfo>
      </xs:annotation>
      <xs:element name="user">
            <xs:complexType>
                  <xs:sequence>
                        <xs:element name="userName" type="xs:string" />
                        <xs:element name="password" type="xs:string" />
                        <xs:element name="credits" type="xs:int" />
                        <xs:element name="lastIp" type="xs:string" />
                        <xs:element name="logs">
```

```
                        <xs:complexType>
                            <xs:sequence>
                                <xs:element maxOccurs="unbounded" name="loginLog"
                                    type="tn:LoginLog" />
                            </xs:sequence>
                        </xs:complexType>
                    </xs:element>
                </xs:sequence>
                <xs:attribute name="userId" type="xs:int" />
            </xs:complexType>
        </xs:element>
        <xs:complexType name="LoginLog">
            <xs:sequence>
                <xs:element name="userId" type="xs:string" />
                <xs:element name="ip" type="xs:string" />
                <xs:element name="loginDate" type="xs:date" />
            </xs:sequence>
            <xs:attribute name="loginLogId" type="xs:int" />
        </xs:complexType>
    </xs:schema>
```

JAXB 在生成代码时，如果元素类型是日期类型，也就是 type="xs:date"时，默认情况生成 Java 对象属性类型为“javax.xml.datatype.XMLGregorianCalendar”，这个明显不是我们想要的类型，不过无须担心，JAXB 已经为我们提供了一个全局的类型转换机制，如实例在 Schema 文档中设置一个全局解析日期类型：

```
        <xs:annotation>
            <xs:appinfo>
                <JAXB:globalBindings>
                    <JAXB:javaType name="java.util.Calendar"   xmlType="xs:date"
                        parseMethod="javax.xml.bind.DatatypeConverter.parseDate"
                        printMethod="javax.xml.bind.DatatypeConverter.printDate" />
                </JAXB:globalBindings>
            </xs:appinfo>
        </xs:annotation>
```

此时 JAXB 生成对应实例类属性类型为就变成“java.util.Calendar”，细心的读者可能会有疑问，如果想要的类型是“java.util.Date”该如何设置，由于 JAXB 提供日期解析器只能转换成 Calendar 类型，所以我们需要自己编写一个 Date 类型适配解析器来替换“javax.xml.bind.DatatypeConverter.parseDate”，接下来我们来编写一个 Date 类型适配器，如代码清单 19-12 所示。

代码清单 19-12　DateAdapter.java

```
package com.smart.oxm.JAXB;
...
public class DateAdapter {
    public static Date parseDate(String s) {
```

```
        return DatatypeConverter.parseDate(s).getTime();
    }
    public static String printDate(Date dt) {
        Calendar cal = new GregorianCalendar();
        cal.setTime(dt);
        return DatatypeConverter.printDate(cal);
    }
```

生成 Java 类

xjc 工具基于 XML Schema 文档定义来绑定一个模式到 Java 类，也就是根据 XML Schema 文档定义生成相应的 Java 实体类。针对当前 XML 的模式来进行绑定的命令是：

xjc user.xsd

xjc 命令行一些常用参数选项说明如下：

-nv 对于输入的模式不执行严格的 XML 验证
-b <file> 指定外部的绑定文件
-d <dir> 指定生成文件的存放路径
-p <pkg> 指定目标包
-classpath <arg> 指定 classpath
-XMLschema 输入的模式是一个 W3C XML 模式（默认）

xjc 命令提供了包参数选项-p，来设置生成 Java 实体类包结构，如果该参数没有输入，xjc 将利用 Schema 文档中定义的 namespace 自动产生包名（package name）。如实例中设置的 targetNamespace="http://www.smart.oxm.com/domain/jaxb"，生成的包结构为“com.smart.oxm.domain.jaxb”。

使用用参数-p 指定包名，生成源代码结构如下所示：

```
D:\masterspring\code\chapter19\bin>
                                    xjc -p com.smart.oxm.domain.jaxb  user.xsd -d src
parsing a schema...
compiling a schema...
com\smart\oxm\domain\ jaxb\Adapter1. java
com\smart\oxm\domain\ jaxb\LoginLog. java
com\smart\oxm\domain\ jaxb\ObjectFactory. java
com\smart\oxm\domain\ jaxb\User. java
com\smart\oxm\domain\ jaxb\package-info. java
```

使用 Schema 中定义的 namespace 作为包名，生成源代码结构如下所示：

```
D:\masterspring\code\chapter19\bin>xjc user.xsd -d src
parsing a schema...
compiling a schema...
com\smart\oxm\domain\ jaxb\LoginLog. java
com\smart\oxm\domain\ jaxb\ObjectFactory. java
com\smart\oxm\domain\ jaxb\User. java
com\smart\oxm\domain\ jaxb\package-info. java
org\w3\_2001\XMLschema\Adapter1. java
```

使用 JAXB 进行编组与反编组操作之前，需要创建一个 JAXBContext 对象，即 JAXB 的上下文，如实例中 JAXBContext context = JAXBContext.newInstance(User.class)，通过其

createMarshaller()方法创建一个编组器，编组时调用 Marshaller#marshal()方法就完成转换操作，如果希望生成 XML 文件格式良好，需要调用 Marshaller#setProperty()方法，设置输出 XML 文件的格式，反编组时调用其 createUnmarshaller()方法创建一个反编组器，然后调用 Unmarshaller#unmarshal()方法就可完成 XML 到 Java 对象的转换，如代码清单 19-13 所示。

代码清单 19-13　JAXBSample.xml

```
package com.smart.oxm.JAXB;
...
public class JAXBSample {

    // Java转化为XML对象
    public static void objectToXML() throws Exception {
        User user = getUser();
        JAXBContext context = JAXBContext.newInstance(User.class);
        Marshaller m = context.createMarshaller();
         m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
        FileWriter writer = new FileWriter("out/JaxbSample.xml");
        m.marshal(user, writer);
    }

    // XML转化为Java对象
    public static void XMLToObject() throws Exception {
        JAXBContext context = JAXBContext.newInstance(User.class);
        FileReader reader = new FileReader("out/JaxbSample.xml");
        Unmarshaller um = context.createUnmarshaller();
        User u = (User) um.unmarshal(reader);
        ...
    }
  ...
}
```

实战经验

目前 XML 数据绑定组件基本都支持两种方式，一是映射绑定，二是代码生成，如果采用代码生成，即根据 XML 样本数据文法生成 Java 语言代码，则必须编写 XML Schema 文件。对于熟悉 XML 的开发人员，可以自己来写这个 Schema 文件，对于不熟悉 XML 的开发人员，可以通过一些工具来完成，比较有名的如 Trang、XMLSpy、Stylus Studio 都可以通过 XML 样本数据来生成相应的 Schema 文件。

19.3.2　Castor

Castor 是 ExoLab Group 下面的一个开放源代码的项目，其主要目标是在 XML 数据、Java 对象和数据库关系数据之间提供一种直接的映射，使得这三种对象数据可以相互转换。Castor 项目在 2000 年 3 月发布第一个 0.8 版本，项目几经重构设计，目前最新版本为

1.4。Castor 项目主要包括 XML 与 Java 对象的映射（Castor XML）、Java 对象与关系数据库表映射（Castor JDO）两个关键功能。在 Castor 中，我们通过定义数据映射（Mapping）文件，在 XML 文档元素（节点、属性、文本等）、Java 类（类、属性等）和数据库表（关联表、表、字段）之间建立一一映射。通过 XML Schema 定义可以自动生成实体 Java 类定义，通过实体 Java 类定义可以自动生成 XML 元素与 Java 类之间的映射文件。

Castor 主要特性

- 通过定义映射文件在 XML 文档与 Java 对象实体间建立映射。
- Castor XML 在 Java 对象模型与 XML InfoSet 之间建立相互转换和数据绑定功能。
- 提供从 XML 文档定义自动生成 Java 类定义文件的功能（Code Generator）。
- 通过自省方式在 XML 文档与 Java 对象间建立映射。
- 提供基于 Java 类定义生成映射文件的功能。
- 提供基于 XML 输入文档生成 Schema 定义文件的功能。
- Castor JDO 为 Java 对象与数据库关系数据之间提供一种序列化与反序列化的框架。
- 基于 XML 映射文件的方式定义三种不同数据对象模型之间的映射。
- 提供内存缓存和提交写入方式，减少数据库 JDBC 操作。
- 支持两阶段事务、对象操作回滚和数据库锁定侦测。
- 支持 OQL 与标准 SQL 之间的映射，Castor JDO 对象查询采用 OQL。

Castor 组件提供了几种进行 Java 对象与 XML 相互转换的方式：一是采用 XML Schema 生成代码方式；二是采用自省方式；三是采用映射文件方式。下面，我们就通过这几种不同方式体验 Castor 组件的灵活与强大。

在开发实例之前，我们需要在 pom.xml 中引入 Castor 的依赖包，包含 castor-xml 和，castor-core。本实例选用 Castor 1.3 版本，如下代码所示：

```
<dependency>
        <groupId>org.codehaus.castor</groupId>
        <artifactId>castor-xml</artifactId>
        <version>1.3</version>
</dependency>
<dependency>
         <groupId>org.codehaus.castor</groupId>
          <artifactId>castor-core</artifactId>
          <version>1.3</version>
 </dependency>
```

XML Schema 生成代码

创建 XML Schema 文件，Castor 使用这个 Schema 文件生成一系列相关的 Java 实体类及相应的类描述符，本例 User 模式结构图如图 19-3 所示。

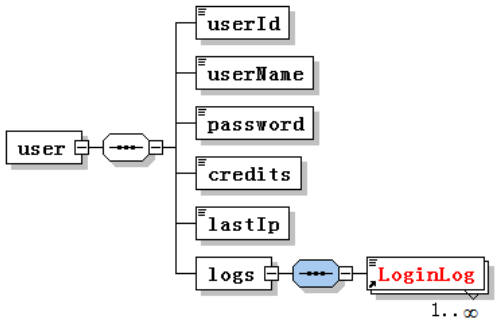

图 19-3　User 模式结构图

为了应用 Castor 生成 Java 实体类，需要把以下包 castor-1.3.jar、castor-1.3-codegen.jar、castor-1.3-XML-schema.jar 放到类路径下，还需要编写一个批处理文件或者 Ant 构建文件，首先编写一个批处理文件 user.bat，如代码清单 19-14 所示。

代码清单 19-14 user.bat

```
@echo off
REM Change the following line to set your JDK path
set JAVA_HOME=%JAVA_HOME%
set JAVA=%JAVA_HOME%\bin\Java
set JAVAC=%JAVA_HOME%\bin\Javac
@echo Create the classpath
set CP=.
for %%i in (lib\*.jar) do call cp.bat %%i
set CP=%CP%;%JDK_BIN%\lib\tools.jar
@echo.
@echo Using classpath: %CP%
@echo Castor Test Cases
@echo.
@echo Generating classes...
@rem Java 2 style collection types
@rem %JAVA% org.exolab.castor.builder.SourceGeneratorMain -i invoice.xsd -f -types j2 -binding-file
bindingInvoice.xml
@rem Java 1.1 collection types
%JAVA% -cp %CP% org.exolab.castor.builder.SourceGeneratorMain -i user.xsd -types j2 -package
com.smart.oxm.domain.castor
@echo.
@pause
```

①生成代码入口

其中 org.exolab.castor.builder.SourceGeneratorMain 生成代码入口，参数-i 指定 schema 文件，参数-package 指定生成实例类所在的包名。

运行 user.bat 文件，生成一系列 Java 实体类以及实体类相应的类描述符，如本实例将生成三个实例类（User.java、LoginLog.java、Logs.java）及三个实体类对应的类描述符（UserDescriptor. java、LoginLogDescriptor. java、LogsDescriptor. Java）。

与 XMLBeans 一样，Castor 也可通过生成 Java 代码的方式完成对象与 XML 相互转换，但 Castor 生成实体类及类描述符之后，转换工作比 XMLBeans 更加简洁，把一个实体对象转换为 XML，只要创建当前对象的实例，调用实体对象 marshal(Writer out)方法，就完成了对象编组工作，也就是把当前对象转换成一个 XML 文件。反编组也一样简洁，只需要调用实例类反编组静态方法 unmarshal(Reader reader)就可完成转换工作，如代码 19-15 所示。

代码清单 19-15 CastorGeneratorSampe.java

```
package com.smart.oxm.castor;
…
public class CastorGeneratorSampe {

    // Java对象转化为XML
    public static void objectToXML()throws Exception {
        User user = getUser();
        FileWriter writer = new FileWriter("out/CastorSample.xml");
        user.marshal(writer);
    }
```

```
    // XML转化为Java对象
    public static void XMLToObject()throws Exception {
            FileReader reader = new FileReader("out/CastorSample.xml");
            User u = User.unmarshal(reader);
            …
    }
 …
}
```

自省方式的编组与反编组

Castor 提供自省的方式在 Java 对象与 XML 文档之间实现转换，也就是说，我们不需要编写 Java 对象与 XML 文档之间的映射文件，Castor 在编组的时候，根据实体对象属性生成相应的 XML 元素。

Castor 提供了 Marshaller、Unmarshaller 两个类处理编组与反编组操作，编组的时候，调用 Marshaller#marshal()，就完成对象转换为 XML 的工作。反编组的时候，调用 Unmarshaller 静态反编组方法 Unmarshaller#unmarshal()即可，如实例中 Unmarshaller#unmarshal()，通过 FileReader 把 XML 文件内容读入并映射到 User 对象，如代码清单 19-16 所示。

代码清单 19-16　CastorSample.java

```
package com.smart.oxm.castor;
…
public class CastorSample {

    // Java对象转化为XML
    public static void objectToXML()throws Exception {
        User user = getUser();
        FileWriter writer = new FileWriter("out/CastorSample.xml");
        Marshaller marshaller = new Marshaller(writer);
        marshaller.setEncoding("GBK");
        marshaller.marshal(user);
    }

    // XML转化为Java对象
    public static void XMLToObject()throws Exception {
         FileReader reader = new FileReader("out/CastorSample.xml");
         User u = (User) Unmarshaller.unmarshal(User.class, reader);
         …
    }
    …
}
```

映射文件方式的编组与解编

Castor 虽然提供自省的方式在 Java 对象与 XML 文档之间进行转换，但当 Java 对象属性与 XML 文档元素名称不一致时，就无法采用自省方式，这时就需要编写它们之间的映射文件，告诉 Castor 它们之间的对应关系，本实例仍然使用 User. java 及 LoginLog.java 作

为实例的操作对象。

我们首先要编写一个映射文件 mapping.xml，如代码清单 19-17 所示。

代码清单 19-17 mapping.xml

```
<?XML version="1.0"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"
                         "http://castor.org/mapping.dtd">
<mapping>
    <class name="com.smart.oxm.domain.User">
        <map-to xml="user" /><!—①设置XML文件的根据元素 -->
        <field name="userId" type="int">
            <bind- xml name="id" node="attribute" />
        </field>
          …
        <field name="userName" type="Java.lang.String">
            <bind- xml   name="userName"   />
        </field>
        <field name="logs" type="com.smart.oxm.domain.LoginLog" collection="arraylist">
            <bind-xml name="log" />
        </field>
    </class>

    <!-- ②通过在该元素中加入 auto-complete 属性并把值设为 true ，您可以告诉 Castor 对于该类的任何属性，只要没有在这个元素中专门列出，就使用默认映射 -->
    <class name="com.smart.oxm.domain.LoginLog" identity="userId"   auto-complete="true">
        <map-to xml="logs" />
        <field name="loginLogId" type="int">
            <bind-xml name="id" node="attribute" />
        </field>
        <field name="userId" type="int">
            <bind-xml name="userId" />
        </field>
    </class>
</mapping>
```

与 Castor 自省方式编组及反编组操作方式一样，编组的时候，也是调用 Marshaller#marshal()方法，完成 Java 对象转换为 XML 工作，不同的是在编组之前，需要先把映射文件加载到 Mapping 中，然后调用 Marshaller#setMapping()方法设置到编组器中。反编组的时候，与自省方式不同，需要先创建 Unmarshaller 实例，并把映射文件加载到其实例中，调用 Unmarshaller 对象方法 unmarshal(FileReader reader)进行反编组操作，而不是调用静态方法。

Castor 使用 Marshaller 和 Unmarshaller 类的静态方法以自省的方式实现 XML 文档与 Java 对象之间的编组和反编组，以及使用 Mapping 映射文件通过调用 Marshaller 和 Unmarshaller 的实例方法实现 XML 文档与 Java 对象之间的编组和反编组两种方式。前一种方式不需要构造 Marshaller 和 Unmarshaller 的实例，后一种方式需要构造 Marshaller 和 Unmarshaller 的实例，调用非静态方法完成上述功能。调用 Marshaller 和 Unmarshaller 的

实例方法时，必须提供 XML 文档结构和 Java 类模型间的映射定义文件。

Castor 通过类描述符和字段描述符几乎可以编组/解编组所有的 Java 对象。当某个 Java 对象类描述符不存在时，编组框架通过使用反射（Reflection）方式获取对象的结构信息，并在内存中为此对象构造类描述符和字段描述符。当实体类的描述符存在时，Castor 使用这些描述符提供的 Java 对象模型信息处理编组和解编组。在编组与解编组过程中，Castor 要求实体类定义必须提供 public 类型的默认构造函数（不含参数）声明以及必要的 getter 和 setter 方法。

19.3.3　JiBX

JiBX 是一款非常优秀的 XML 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；并不需要你修改既有的 Java 类。不管在灵活性还是在效率上是目前很多开源项目都无法比拟的，如它具有转换效率高、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 文件元素名不需要相同等优点。

JiBX 使用 Java 字节码的增强技术，即 BCEL（Byte Code Engineering Library），在编译期，根据绑定配置文件，生成对应对象实例的方法和添加被绑定标记的属性，在运行期，它已经与绑定配置文件无关了，不需要再做任何的配置，因为其已经嵌入到相应的 Java 类中。2005 年 12 月份发布 JiBX 1.0 版本，目前最新版本是 2015 年 1 月份发布 JiBX 1.2.6。

JiBX 主要特性

- JiBX 使用 BCEL 技术在编译时对相应的类字节码进行增强。
- JiBX 使用 XPP（XML Pull Parse）技术，它提供了一个更自然的接口来处理文档中的元素序列，能够使用相对简单的代码来处理数据。
- JiBX 可以自由灵活地定义 XML 中的每个字段和 Java 类的对应关系。

在开发实例之前，我们需要在 pom.xml 中引入 JiBX 的依赖包，本实例采用 JiBX 1.1.5 版本，如下代码所示：

```
<dependency>
    <groupId>net.sourceforge.jibx</groupId>
    <artifactId>com.springsource.org.jibx.runtime</artifactId>
    <version>1.1.5</version>
</dependency>
```

使用 JiBX 进行编组及反编组操作时，需要编写 Java 对象与 XML 之间的绑定映射文件，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。它有如此大的魔法，得益于 BCEL 技术，BCEL 是 Apache Software Foundation 的 Jakarta 项目的一部分，BCEL 在单独的 JVM 指令级别上进行操作，可以深入 Java 类的字节码，用它转换现有的类或者构建新的类，如代码清单 19-18 所示。

代码清单 19-18　binding.xml

```
<binding>
```

```
  <mapping name="user" class="com.smart.oxm.domain.User">
    <value name="userName" field="userName"/>
    〈!—name 设置XML元素名称，field设置object 对象属性名称 --〉
    <value name="password" field="password" usage="optional" />
    <value name="credits" field="credits" usage="optional" />
    <collection field="logs" factory="com.smart.oxm.jibx.JiBXInterfaceFactory.getArrayListInstance">
     <structure name="log" type="com.smart.oxm.domain.LoginLog">
            <value name="ip" field="ip"/>
            <value name="loginDate" field="loginDate"/>
          </structure>
    </collection>
  </mapping>
</binding>
```

在 JiBX 绑定 Java 对象时，如果你使用了基本类型，又设置字段为可选值时，如<value name="credits" field="credits" usage="optional" />中 usage="optional"，JiBX 根据 credits 类型的默认值来决定是否编组当前元素，如果编组的时候，credits 的值刚好是 0，则 credits 无法显示在 XML 文件中，也就是说基本类型为可选项时，是无法输出默认值的。为了避免出现这个问题，我们在使用基本类型时，需要把字段设置为不可选。

使用接口处理从 XML 到 Java 对象的转换时，先要创建该类的实例，如果转换的是一个实体类，创建实例就不会有什么问题，但是如果我们要使用接口编程，转换的对象就必须是一个接口，否则会出现错误，因为 JiBX 并不清楚你需要创建这个接口的哪个实例，所以我们需要在绑定文件中指明该接口的创建工厂方法。例如，本例 User 中声明一个 List 类型的属性 logs，想要它指向一个 ArrayList 的实例，我们需要写一个返回 ArrayList 实例的工厂类 JiBXInterfaceFactory，如代码清单 19-19 所示。

代码清单 19-19　JiBXInterfaceFactory.java

```
package com.smart.oxm.jibx;
…
public class JiBXInterfaceFactory {
    //获取List实现类实例
    public static List getArrayListInstance(){
        return new ArrayList();
    }
}
```

编写完绑定映射文件后，我们还要编写一个 ant 构建文件，完成 Java 类编译及字节码增强的工作，以增强实体对象编组及反编组的能力，如代码清单 19-20 所示。

代码清单 19-20　build.xml

```
<?XML version="1.0" encoding="gb2312"?>
<project name="oxm" default="run">
    <property name="project" value="D:/masterspring/code/chapter19"/>
    <property name="src" value="D:/masterspring/code /chapter19/src/main/java "/>
    <property name="lib" value="D:/masterspring/code/libs"/>
     <property name="classes" value="D:/masterspring/code/chapter19/target/classes"/>
```

```
    <target name="clean">
        <delete includeEmptyDirs="true">
            <fileset dir="${classes}" includes="**/*" defaultexcludes="no"/>
        </delete>
    </target>
     …
    <taskdef name="bind" classname="org.jibx.binding.ant.CompileTask"
classpath="${lib}/jibx-bind.jar"/>
    <target name="jibx-binding" depends="compile">
            <bind verbose="false" load="true"
            binding="${classes}/com/smart/oxm/jibx/binding.xml">
            <classpath>
                <path location="${classes}">
                </path>
                <pathelement location="${lib}/jibx-bind.jar"/>
            </classpath>
        </bind>
    </target>
    <target name="run" depends="jibx-binding" description="run">
        <Java classname="com.smart.oxm.jibx.JibxSample">
            <classpath>
                <path location="${classes}">
                </path>
                …
            </classpath>
        </Java>
    </target>
</project>
```

用 Ant 运行当前构建文件，在 User.class 同目录下，我们可以看到 JiBX 自动编译生成 JiBX_MungeAdapter.class 等三个类，用反编译工具查看 User.class 就会发现，User 已经自动实现 IMarshallable、IUnmarshallable 编组与反编组两个接口。

使用 JiBX 进行编组及反编组操作时，需要通过 BindingDirectory 获取一个相应实体类的绑定工厂实例，如实例中 IBindingFactory bfact = BindingDirectory.getFactory(User.class)，编组的时候，调用其 createMarshallingContext()方法创建一个编组上下文，调用编组上下文实例的 marshalDocument()方法进行编组，反编组的时候，调用其 createUnmarshallingContext()方法创建一个反编组上下文，并通过实例的 unmarshalDocument()方法来进行反编组操作，如代码清单 19-21 所示。

代码清单 19-21　JibxSample.xml

```
package com.smart.oxm.jibx;
…
public class JibxSample {
    // Java对象转化为XML
    public static void objectToXML() throws Exception {
        User user = getUser();
        IBindingFactory bfact = BindingDirectory.getFactory(User.class);
```

```
            IMarshallingContext ctx = bfact.createMarshallingContext();
            FileOutputStream os = new FileOutputStream("out/JibxSample.xml");
            ctx.marshalDocument(user, "UTF-8", null, os);
        }

        // XML转化为Java对象
        public static void XMLToObject() throws Exception {
            IBindingFactory bfact = BindingDirectory.getFactory(User.class);
            IUnmarshallingContext uctx = bfact.createUnmarshallingContext();
            File dataFile = new File("out/JibxSample.xml");
            InputStream in = new FileInputStream(dataFile);
            User user = (User) uctx.unmarshalDocument(in, null);
            ...
        }
    ...
}
```

JiBX 是一款高性能的数据绑定框架。如果 XML 文件格式比较固定，同时数据转换比较频繁时，可以考虑使用它来助你一臂之力，需要注意的是 JiBX 对中文的支持不是很好，目前 JiBX 只支持 Java 标准的字符集，如果使用的是 UTF-8 来处理中文，则没有任何问题，如果使用 GB2312 或 GBK 编码的话，则需要你自己实现 GB2312 和 GBK 的 Escaper 类，重写 writeAttribute(String, Writer)、writeCData(String, Writer)和 writeContent(String, Writer)方法。

19.3.4 总结比较

上文已经详细介绍了目前一些主流 O/X Mapping 组件的特点及使用方法，不同 O/X Mapping 组件都有各自的应用场景，如在构建基于 Spring 轻量级 Restful 服务过程中，可以使用 XStream 组件处理服务请求响应的报文（XML 或 JSON 数据格式），在构建基于 Spring 契约优先的 Web Services 时，可以使用 XMLBeans、JAXB 等组件来处理服务请求响应的报文。表 19-2 列举了目前各 O/X Mapping 组件的优势、劣势及其应用场景，读者可以从中根据实际需求选择合适的 O/X Mapping 组件。

表 19-2 各 O/X Mapping 组件对比

XML 编组框架	优　　势	劣　　势	应用场景
XStream	简单易用； 同时支持 XML 和 JSON 数据格式	非标准； 默认 Driver 不支持 CDATA； 映射集合时对 XML 的格式要求很严格； 修改映射就需要修改代码	应用对象各种序列化及反序列化操作； 基于 Spring 轻量级 Restful 服务中的对象映射处理
Castor	提供丰富工具，如根据 XML 文档生成 Java 代码、根据 Java 类定义生成映射文件等； 支持自省方式在 XML 与	不支持使用注解	应用各种数据绑定需求； 应用 JPA 相关实现； 使用 XML 交换数据的应用程序

	Java 对象间建立映射； 支持绑定任意类； 完全支持 JDO		
JiBX	支持 CDATA； 对集合的兼容性很好； 允许定义空标签； 在配置文件映射，而不是在代码映射； 基于字节码增强技术，运行速度非常快	需要对 POJO 进行增强，每次重新编译后都必须重新进行增强。不过可以预编译，也可以运行时编译； 配置文件相对复杂些； 不支持灵活的验证机制	广泛应用各种数据绑定需求； Web Services 常用数据绑定框架
JAXB	业界的标准	XML 模式（XML schema）发生变化，就必须重新编译； XML 模式，以便重新生成； JAXB，重新编译工作可能会引起使用由 XML 模式生成的 JAXB 代码发生变化	广泛应用各种数据绑定需求； Web Services 常用数据绑定框架； 使用 XML 交换数据的应用程序

19.4　与 Spring OXM 整合

19.4.1　Spring OXM 概述

Spring OXM 是 Spring 3.0 的一个新特性，为主流 O/X Mapping 组件提供了统一层抽象和封装，而在 Spring 4.0 中 OXM 没有增加更多新的特性。O/X 映射器这个概念并不新鲜，O 代表 Object，X 代表 XML，目的是在 Java 对象和 XML 之间进行转换操作。Spring-OXM 不仅屏蔽了各 O/X Mapping 组件实现的差异性，而且还提供了统一、高效的编程模型。Spring OXM 的结构框架如图 19-4 所示。

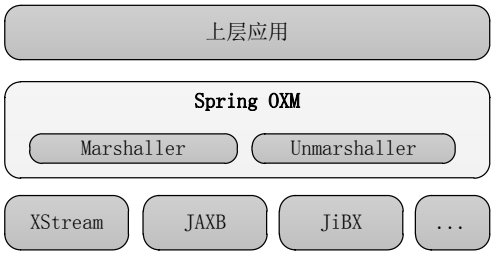

图 19-4　Spring OXM 结构框架

Marshaller 和 Unmarshaller 是 Spring OXM 两个核心接口。实现 Marshaller 接口可以实现从 Java 对象到 XML 的映射转换，实现 Unmarshaller 接口可实现从 XML 到 Java 对象的映射转换。下面从 Spring OXM 源代码中摘录部分 Marshaller 接口的定义，如代码清单 19-22 所示。

代码清单 19-22　Marshaller.java

```
package org.springframework.oxm;
import java.io.IOException;
import javax.xml.transform.Result;
public interface Marshaller {

    //① 判断支持的编组类类型
    boolean supports(Class<?> clazz);

    //② 对传入的对象进行编组操作
    void marshal(Object graph, Result result) throws IOException, XmlMappingException;

}
```

Marshaller 接口中提供两个方法，其中 Marshaller# supports()方法用于类判断支持编组的类类型。Marshaller# marshal()方法用于对象进行编组操作，其中 graph 参数为目标转换对象，result 参数为 JDK 提供的 XML 转换接口，实现此接口的对象包含构建转换结果树所需的信息。Spring OXM 支持的 Result 转换实现类有 DOMResult、SAXResult、StreamResult、StaxResult，其中 DOMResult、SAXResult、StreamResult 为 JDK 提供的实现类，StaxResult 为 Spring OXM 提供的扩展类。

下面从 Spring OXM 源代码中摘录部分 Unmarshaller 接口的定义。

代码清单 19-23 Unmarshaller.java

```
package org.springframework.oxm;
import java.io.IOException;
import javax.xml.transform. Source;
public interface Unmarshaller{

    //① 判断支持的反编组类类型
    boolean supports(Class<?> clazz);

    //② 对输入源对象进行反编组操作
    Object unmarshal(Source source) throws IOException, XmlMappingException;

}
```

Unmarshaller 接口中提供两个方法，其中 Marshaller# supports()方法用于类判断支持反编组的类类型。Marshaller#unmarshal()方法用于将输入源对象进行反编组操作，其中 source 参数为目标输入源对象，Source 参数为 JDK 提供的输入源接口，实现此接口的对象包含充当源输入（XML 源或转换指令）所需的信息。Spring OXM 支持的 Source 输入源实现类有 DOMSource、SAXSource、StreamSource、StaxSource，其中 DOMSource、SAXSource、StreamSource 为 JDK 提供的实现类，StaxSource 为 Spring OXM 提供的扩展类。

Spring OXM 统一封装底层的 O/X Mapping 组件异常，将各 O/X Mapping 组件原始异常对象包装到 Spring 自身专为 OXM 建立的运行时异常 XmlMappingException 中，并通过 MarshallingFailureException 和 UnmarshallingFailureException 处理编组和反编组操作之间的区别。Spring OXM 异常层次结构如图 19-5 所示。

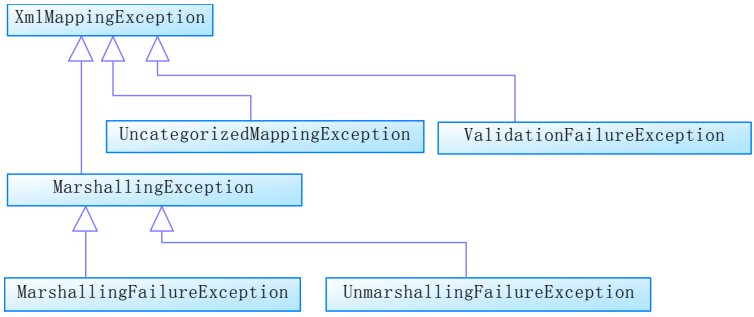

图 19-5　Spring OXM 异常层次结构

19.4.2　整合 OXM 实现者

Spring OXM 默认提供目前几个主流 O/X Mapping 组件实现，包括 XStream、XMLBeans、Castor、JiBX、JAXB。这些 O/X Mapping 组件统一实现 Spring OXM 两个核心接口 Marshaller 和 Unmarshaller，具体的实现体系类图如图 19-6 所示。

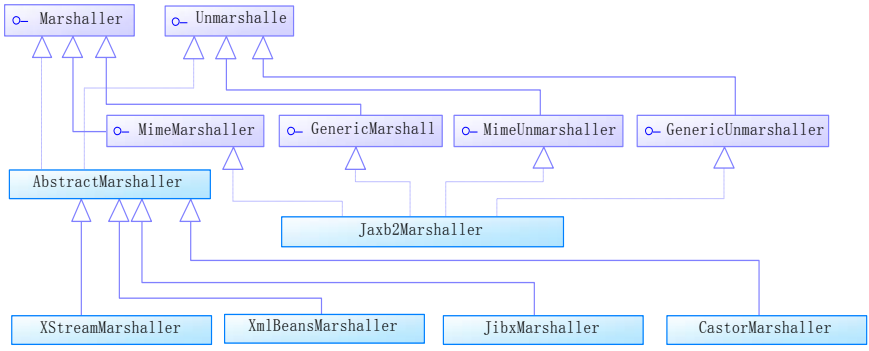

图 19-6　Spring OXM 核心接口实现

要使用 Spring OXM 的 O/X 功能，首先需要一个在 Java 对象和 XML 之间来回转换的组件，并将上文提到 XStream、JAXB 等相应的类库添加到工程中，各 O/X Mapping 组件对应的实现类如表 19-3 所示。

表 19-3　各 O/X Mapping 组件包装器

O/X Mapping 组件	Spring OXM 实现类
XStream	org.springframework.oxm.xstream.XStreamMarshaller
Castor	org.springframework.oxm.castor.CastorMarshaller
JiBX	org.springframework.oxm.jibx.JibxMarshaller
JAXB2.0	org.springframework.oxm.jaxb.JAXB2Marshaller

19.4.3 如何在 Spring 中进行配置

到目前为止，已经介绍了各 O/X Mapping 组件的使用方法以及 Spring OXM 总体框架，接下来介绍如何在 Spring 中整合这些 O/X Mapping 组件，不同的 O/X Mapping 组件在 Spring 中的配置略有不同，下面分别对各主流 O/X Mapping 组件的配置进行讲解。

OXM 命名空间

为了简化 Spring OXM 各 O/X Mapping 组件的配置，Spring OXM 提供了一个 OXM 命名空间，目前支持 OXM 命名空间的 O/X Mapping 组件有 3 个，分别是 Jaxb2Marshaller、JibxMarshaller、XmlBeansMarshaller。要启用 OXM 命名空间，首先需要在 Spring 配置文件中引用 OXM 模式定义文件，如代码清单 19-24 中的粗体所示。

代码清单 19-24 OXM 命名空间

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:oxm="http://www.springframework.org/schema/oxm"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/oxm
http://www.springframework.org/schema/oxm/spring-oxm.xsd">
    …
</beans>
```

XStreamMarshaller 配置

要启用 Spring OXM 的 XStreamMarshaller 功能，首先需要将 XStream O/X Mapping 组件相应类库添加到工程 pom.xml 的依赖中，本章中使用 xstream-1.4.9。

```
<dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.4.9</version>
</dependency>
```

默认情况下，不需要对 XStreamMarshaller 做任何配置，直接在 Spring 应用程序上下文中配置即可。如果需要自定义的 XML 格式，可以设置类别名映射及属性别名映射，如代码清单 19-25 所示。

代码清单 19-25 XStreamMarshaller 配置实例

```
<bean id="xstreamMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller"
        p:autodetectAnnotations="true">
        〈!-- 设置类名别名 --〉
        <property name="aliases">
            <map>
                〈!-- User这个类的别名就变成了user --〉
                <entry key="user" value="com.smart.oxm.domain.User" />
```

```
            <entry key="LoginLog" value="com.smart.oxm.domain.LoginLog" />
        </map>
    </property>
    〈!-- 设置属性别名 --〉
    <property name="fieldAliases">
        <map>
            〈!-- User中的lastVisit这个属性 --〉
            <entry key="com.smart.oxm.domain.User.lastVisit" value="lastVisitDate" />
        </map>
    </property>
</bean>
```

在上面实例配置中，通过 aliases 属性设置类名别名，通过 fieldAliases 设置类属性别名，通过 autodetectAnnotations 属性设置自动加载 XStream 注解 Bean。

Jaxb2Marshaller 配置

要启用 Spring OXM 的 Jaxb2Marshaller 功能，首先需要将 Jaxb O/X Mapping 组件相应类库增加到 pom.xml 的依赖中，如本章中使用 jaxb-api-2.1.4、jaxb-impl-2.1.8。Jaxb2Marshaller 可以使用与 Jaxb1Marshaller 相同配置属性的 contextPath 来指定编组对象。Jaxb2Marshaller 提供了一个 classesToBeBound 的属性，用来设置编组对象数组。Schema 属性指定一个或多个模式资源，如代码清单 19-26 所示。

代码清单 19-26　Jaxb2Marshaller 配置实例

```
〈!--① 传统配置方法 --〉
<bean id="jaxb2Marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <!--<property name="contextPath" value="com.smart.oxm.domain.jaxb"></property>-->
    <property name="classesToBeBound">
        <array>
            <value>com.smart.oxm.domain.jaxb.User</value>
            <value>com.smart.oxm.domain.jaxb.LoginLog</value>
        </array>
    </property>
    <property name="schema" value="classpath:com/smart/oxm/jaxb/user.xsd" />
</bean>
〈!--② 基于OXM命名空间配置方法 --〉
<oxm:jaxb2-marshaller id="jaxb2Marshaller2">
    <oxm:class-to-be-bound name="com.smart.oxm.domain.jaxb.User" />
    <oxm:class-to-be-bound name="com.smart.oxm.domain.jaxb.LoginLog" />
</oxm:jaxb2-marshaller>
```

Jaxb1Marshaller 在 Spring OXM 3.0 之后不被支持，如果没有特殊需求，推荐使用 Jaxb2Marshaller 作为 O/X Mapping 编组与反编组包装器。

CastorMarshaller 配置

要启用 Spring OXM 的 CastorMarshaller 功能，首先需要将 Castor O/X Mapping 组件相应依赖包添加到工程的 pom.xml 中，如本章中使用 castor-1.3-core、castor-1.3-xml、

castor-1.3-commons。不需要对 CastorMarshaller 做任何配置，直接在 Spring 应用程序上下文中配置即可。如果应用自定义的映射配置文件，需要配置 mappingLocation 属性，如代码清单 19-27 所示。

代码清单 19-27 CastorMarshaller 配置实例

```
<bean id="castorMarshaller"
      class="org.springframework.oxm.castor.CastorMarshaller"
      p:mappingLocation="classpath:/config/mapping.xml" />
```

虽然 Castor 支持自省方式在 XML 与 Java 对象间建立映射，但有时可能需要自定义设置它们之间的映射关系，这时就需要使用 Castor 的映射文件，可以通过 mappingLocation 属性来指定自定义的配置文件。

JibxMarshaller 配置

要启用 Spring OXM 的 JibxMarshaller 功能，首先需要将 JiBX O/X Mapping 组件相应依赖包添加到工程 pom.xm 的依赖中，如本章中使用 jibx-run、jibx-bind、jibx-extras。配置 JibxMarshaller 很简单，只需在 Spring 应用程序上下文中配置一个 JibxMarshaller，并通过 targetClass 属性配置目标编组类即可。如果需要设置绑定名称，可以通过 bindingName 属性指定一个绑定配置文件，如代码清单 19-28 所示。

代码清单 19-28 JibxMarshaller 配置实例

```
<!--① 传统配置方法 -->
<bean id="jibxMarshaller" class="org.springframework.oxm.jibx.JibxMarshaller">
    <property name="targetClass" value="com.smart.oxm.domain.User" />
    <!--<property name="bindingName"
                value="classpath:com/smart/oxm/jibx/binding.xml" />-->
</bean>
<!--② 基于OXM命名空间配置方法 -->
<oxm:jibx-marshaller id="jibxMarshaller2"   target-class="com.smart.oxm.domain.User" />
```

一个 JibxMarshaller 只能配置一个目标编组类。如果要配置多个编组类，只能为不同编组类各配置一个 JibxMarshaller。

19.4.4 Spring OXM 简单实例

到目前为止，已经介绍了各 O/X Mapping 组件的使用方法及 Spring OXM 框架，接下来通过一个简单的实例来讲解如何应用 Spring OXM 并整合第三方 O/X Mapping 组件。实例使用 Castor 这个 O/X Mapping 组件处理 Java 对象与 XML 之间的相互转换映射，首先我们在 Spring 上下文（applicationContext.xml）中配置一个 Castor 编组器并注入到实例 SpringOxmSample 中，如代码清单 19-29 所示。

代码清单 19-29 applicationContext.xml

```
<beans XMLns="http://www.springframework.org/schema/beans"
       XMLns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        XMLns:p="http://www.springframework.org/schema/p"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean id="springOxm" class="com.smart.oxm.SpringOxmSample"
            p:marshaller-ref="castorMarshaller"
            p:unmarshaller-ref="castorMarshaller" />
    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"
        p:mappingLocation="classpath:/config/mapping.xml" />
</beans>
```

Spring OXM 仍然遵守不重造轮子的原则，只是为各个绑定映射组件提供一致的访问接口及异常处理机制，与单独使用 Castor 组件一样，我们需要编写一个映射文件，如代码清单 19-30 所示。

代码清单 19-30　mapping.xml

```
<?XML version="1.0"?>
<mapping>
    <class name="com.smart.oxm.domain.User">
        <map-to XML="user"/>
        <field name="userId" type="integer">
            <bind-XML name="userId" node="element"/>
        </field>
         …
        <field name="logs" type="com.smart.oxm.domain.User"    collection="arraylist">
            <bind-XML name="logs" />
        </field>
    </class>
</mapping>
```

Spring OXM 屏蔽第三方映射组件中编组器及反编组器实现细节，也就是说我们映射转换程序与具体映射组件解耦，如果有一天需求突然发生变化，当前的映射组件不能满足需求，我们只需要修改 Spring 配置文件，换成另一个合适的映射组件即可，而不需要去修改我们的映射转换程序。

代码清单 19-31　SpringOxmSample.java

```
package com.smart.oxm;
 …
public class SpringOxmSample {
    private Marshaller marshaller;
    private Unmarshaller unmarshaller;
     //Java转化为XML对象
    public   void objectToXML()throws Exception{
        User user = getUser();
        FileOutputStream os = null;
        try {
            os = new FileOutputStream("out/SpringOxmSample.xml");
             this.marshaller.marshal(user, new StreamResult(os));
        } finally {
```

```
            ...
        }
    }
    //XML转化为Java对象
    public   void XMLToObject()throws Exception{
        FileInputStream is = null;
        User user = null;
        try {
            is = new FileInputStream("out/SpringOxmSample.xml");
             user = (User) this.unmarshaller.unmarshal(new StreamSource(is));
        } finally {
            ...
        }
    }
}
```

Spring OXM 的一个最直接的好处是可以通过使用 Spring 框架的其他特性简化配置。Spring 的 Bean 支持将实例化的 O/X 编组器注入（即前面提到过的“依赖项注入”）使用那些编组器的对象。遵循坚实的面向对象的设计实践，Spring OXM 框架只定义两个接口：Marshaller 和 Unmarshaller，它们用于执行 O/X 功能。另一个重大好处是，这些接口的实现完全对开发人员开放，开发人员可以轻松切换它们而无须修改代码。例如，如果你一开始使用 Castor 进行 O/X 转换，但后来发现它缺乏你需要的某个功能，这时你可以切换到 XMLBeans 而无须任何代码更改。唯一需要做的就是更改 Spring 配置文件以使用新的 O/X Mapping 组件。

使用 Spring OXM 的另一个好处是统一的异常层次结构。Spring OXM 遵循使用它的数据访问模块建立的模式，将原始异常对象包装到 Spring 自身专为 OXM 建立的运行时异常中。由于第三方 O/X Mapping 组件的原始异常被包装到 Spring 运行时异常中，这也是我们能够查明出现异常的根本原因。不必费心修改代码以捕获异常，因为异常已经包装到一个运行时异常中。

19.5 小结

在本章中，我们分析 XML 及其解析技术的发展过程，详细介绍 XML 处理利器 XSteam O/X Mapping 组件的使用，并介绍目前流行的另外几个 O/X Mapping 组件的使用方法，并进行总结比较。接着结合 Spring 提供的 OXM 特性，分析 Spring OXM 框架及两个重要的接口 Marshaller 和 Unmarshaller，并详细介绍 O/X Mapping 组件的配置方法。最后通过一个简单的实例，介绍如何应用 Spring OXM 处理 Java 对象与 XML 之间的相互映射。不管采用何种方式处理，Java 应用程序的 XML 数据绑定我们可以归纳为两种方式：一是根据 XML 文档文法生成 Java 语言代码（如 JAXB、Castor 等）。二是使用某种形式的映射绑定方法，也就是设定 Java 类如何与 XML 进行关联（如 XStream、Castor、JiBX）。

在实际应用过程中，这两种方式都有各自的长处，如果使用由 Schema 或 DTD 定义的

稳定文档结构，并且该结构适合应用程序的需要，则代码生成方法可能是最佳的选择。如果使用现有的 Java 实体类，或者希望使用类的结构，该结构反映应用程序对数据的用法，而不是 XML 结构，则映射方法是最佳的选择。