dom4j 源码阅读分析

肖睿 2017K8009929031

1. XML 和 dom4j 简介

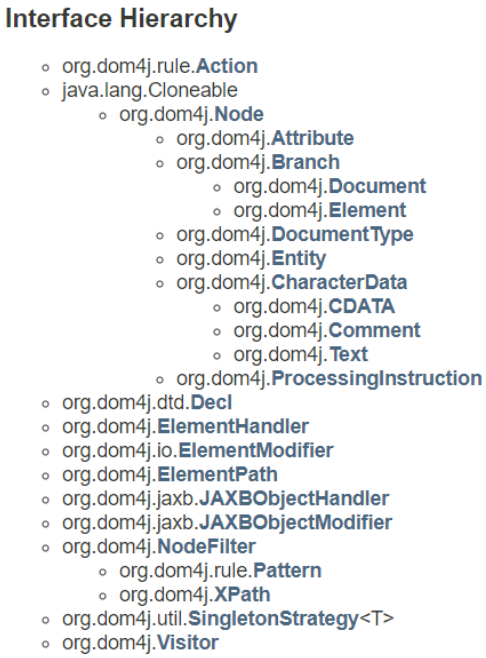
在介绍 dom4j 之前，先让我们了解一下什么是 XML。XML 是 Extensible Markup Language 的缩写，意为可扩展标记语言。它是一种简单而通用的标记语言，被广泛用于数据的存储和传输。而 dom4j 是一个开源的 Java 的 XML API，主要用于解析 XML 文件。

dom4j 名字中，dom 指的是文档对象模型（Document Object Model），它是一种基于树的 API 文档，其将 XML 文档表示为一个树结构，每个结点是一个对象，表示一个标签或标签中的文本项。解析 XML 文档时将其转化为一棵 DOM 树，通过对树的操作完成对 XML 文档的各种操作。而 4j 指 for Java，意指 dom4j 是用 Java 实现的 DOM XML API。

2. dom4j 主要功能

dom4j 可以用来解析 XML 文档，创建、修改、读取 XML 文档，对其中的结点进行新增、修改、删除、遍历等操作，功能十分全面。

dom4j 的这些功能主要建立在大量的接口之上，体现了面向接口，依赖倒转的思想。dom4j 中的接口有如图（来自官方文档）的继承关系：



图表 1 dom4j **接口继承关系图**

其中，主要的用于处理 dom4j 树的接口都继承或间接继承自 Node 结点接口，其中 Document 和 Element 接口继承自 Branch 接口，这意味着它们可以包含子结点，其他接口则

不行，这符合了 XML 树的特点。

```
/**
 * <code>Branch</code> interface defines the common behaviour for Nodes which
 * can contain child nodes (content) such as XML elements and documents. This
 * interface allows both elements and documents to be treated in a polymorphic
 * manner when changing or navigating child nodes (content).
 *
```

图表 2 Branch 接口注释

以创建 XML 文档这一主要功能为例，用户希望使用 dom4j 新建一个 XML 文档的 Java 对象并修改其内容。dom4j.github.io 上给出了一份用 dom4j 创建 XML 文档的示例。首先用 DocumentHelper 帮助类的 createDocument 方法创建一个文档对象，然后用 addElement 方法添加根结点，再在根结点上分别添加两个元素，并添加相应的属性和文本，就完成了一棵 XML 文档树的构建，并返回文档对象。

替代处理，除添加属性、文本外，还可进行用 addComment 方法添加注释等操作。

```
import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;

public class Foo {

    public Document createDocument() {
        Document document = DocumentHelper.createDocument();
        Element root = document.addElement("root");

        Element author1 = root.addElement("author")
            .addAttribute("name", "James")
            .addAttribute("location", "UK")
            .addText("James Strachan");

        Element author2 = root.addElement("author")
            .addAttribute("name", "Bob")
            .addAttribute("location", "US")
            .addText("Bob McWhirter");

        return document;
    }
}
```

图表 3 创建 XML 文档示例代码

3. dom4j 创建 XML 文档的详细过程

上面简单分析了 dom4j 创建 XML 文档的大致流程，下面我们一步一步分析 dom4j 创建 XML 文档的详细过程，包括创建 XML 文档时用到的各个类和接口的关联。

```
Document document = DocumentHelper.createDocument();
```

这一行，调用了 DocumentHelper 类的 createDocument 方法。DocumentHelper 类包含

了许多帮助方法，包括 createDocument、createElement、createAttribute 等。这里使用的是无参的 createDocument 方法，具体实现是使用 getDocumentFactory 方法生成一个 DocumentFactory 工厂类的实例，再用其中的 createDocument 方法创建文档对象并返回。

```
/**
 * <code>DocumentHelper</code> is a collection of helper methods for using
 * DOM4J.
 *
 * @author <a href="mailto:jstrachan@apache.org">James Strachan </a>
 * @version $Revision: 1.26 $
 */
@SuppressWarnings("unused")
public final class DocumentHelper {
    private DocumentHelper() {
    }

    private static DocumentFactory getDocumentFactory() {
        return DocumentFactory.getInstance();
    }

    // Static helper methods
    public static Document createDocument() {
        return getDocumentFactory().createDocument();
    }

    public static Document createDocument(Element rootElement) {
        return getDocumentFactory().createDocument(rootElement);
    }
```

图表 4 DocumentHelper

这里我们暂时跳过如何通过 DocumentFactory 类的 getInstance 方法获得其实例的部分，因为这涉及到设计模式中的单例模式，后面再叙。我们首先看一下 DocumentFactory 类是怎么创建一个文档对象的。

DocumentFactory 类是一个工厂类，集合了许多工厂方法，用于生成 Document、Element、Attribute 等接口的具体实现类的实例。其中无参的 createDocument 方法如图所示，它创建了一个 DefaultDocument 对象并返回。

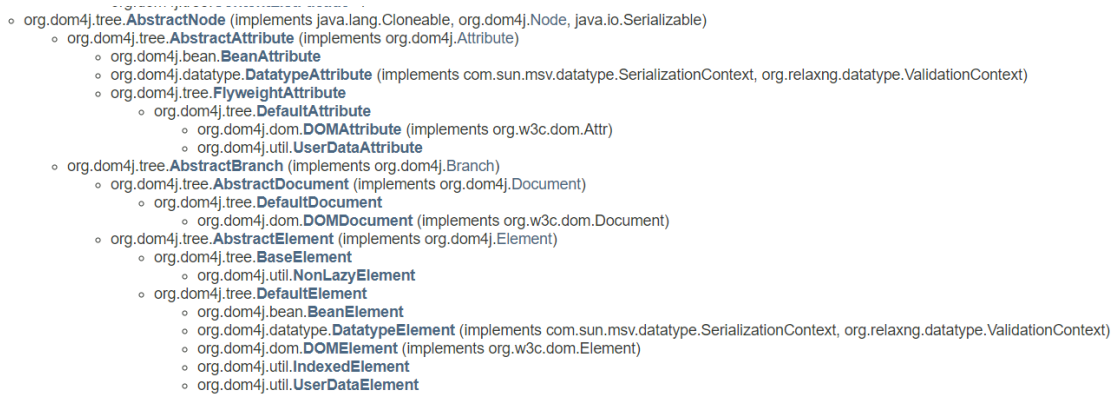
```
    // Factory methods
    public Document createDocument() {
        DefaultDocument answer = new DefaultDocument();
        answer.setDocumentFactory(this);

        return answer;
    }
```

图表 5

DefaultDocument 继承自抽象类 AbstractDocument，而抽象类 AbstractDocument 则实现了接口 Document，继承和实现的关系如图 6 所示。这里使用了 DefaultDocument 的无参构造方法，创建了接口 Document 的抽象实现类 AbstractDocument 的具体子类

DefaultDocument 的实例并返回。这样就获得了一个文档对象。



图表 6 dom4j 类继承关系图（不完整）

接下来，代码的第二行：`Element root = document.addElement("root");`调用了 addElement 方法来为文档对象创建一个根结点，并返回一个引用 root。直接在 DefaultDocument 中找不到这个 addElement 方法；这个方法真正的实现是在 DefaultDocument 的父类，抽象类 AbstractDocument 中。它复写了父类 AbstractBranch 的 addElement 方法，而 AbstractBranch 的 addElement 方法又是接口 Branch 中对应方法的实现。

```
public Element addElement(String name) {
    Element element = getDocumentFactory().createElement(name);
    add(element);

    return element;
}

public Element addElement(String qualifiedName, String namespaceURI) {
    Element element = getDocumentFactory().createElement(qualifiedName,
            namespaceURI);
    add(element);

    return element;
}

public Element addElement(QName qName) {
    Element element = getDocumentFactory().createElement(qName);
    add(element);

    return element;
}
```

图表 7 AbstractDocument 中的 addElement 方法

这里的 addElement 方法首先用与之前创建文档对象类似的方法创建一个元素对象，然后用 add 方法添加这个元素到文档对象中。这个 add 方法继承自 AbstractBranch 类，是接口 Branch 中 add 方法的实现。

接下来，`Element author1 = root.addElement("author")`

```
        .addAttribute("name", "James")
        .addAttribute("location", "UK")
```

```
            .addText("James Strachan");
```

首先调用 addElement 方法在根结点 root 上增加元素 author，由于此方法会返回新增的元素，所以后面直接再调用方法 addAttribute 为元素增加属性。addAttribute 方法不同于 addElement 方法，不是源自于 Branch 接口，而是 Element 接口和 AbstractElement 抽象类中新增的方法，在 Document 和 AbstractDocument 中不存在。这是因为文档对象本身不需要 Attribute，元素对象需要 Attribute。addAttribute 返回元素对象本身，因此可以继续在后面调用 addAttribute 方法。addText 方法和 addAttribute 方法类似。

```
public Element addAttribute(String name, String value) {
    // adding a null value is equivalent to removing the attribute
    Attribute attribute = attribute(name);

    if (value != null) {
        if (attribute == null) {
            add(getDocumentFactory().createAttribute(this, name, value));
        } else if (attribute.isReadOnly()) {
            remove(attribute);

            add(getDocumentFactory().createAttribute(this, name, value));
        } else {
            attribute.setValue(value);
        }
    } else if (attribute != null) {
        remove(attribute);
    }

    return this;
}
```

图表 8 addAttribute

通过上面的操作，一棵简单的 XML 文档树就构造完了。

在创建 XML 文档时，每个方法的返回值 Document、Element 都是接口，实际上返回的是对应接口实现类的对象，再次体现了面向接口编程，依赖倒转的思想。

4. dom4j 设计模式选析

让我们再次回到创建 XML 文档的流程中来。在图 4 中我们可以看到，getDocumentFactory 方法通过 DocumentFactory 类的 getInstance 方法获得其实例，而不是直接 new 一个出来。DocumentFactory 类的 getInstance 方法如图 9 所示。

```
/**
 * Access to singleton implementation of DocumentFactory which is used if no
 * DocumentFactory is specified when building using the standard builders.
 *
 * @return the default singleon instance
 */
public static synchronized DocumentFactory getInstance() {
    if (singleton == null) {
        singleton = createSingleton();
    }
    return singleton.instance();
}
```

图表 9 DocumentFactory 类中的单例模式

这种不直接调用构造方法，而是使用 getInstance 之类的方法获取一个类的实例的模式称为单例模式（Singleton）。单例模式是一种创建型设计模式，其设计意图是保证一个类仅有一个实例，并提供一个访问它的全局访问点。有些类只应该有一个实例或只需要有一个实例，这时候就可以使用单例模式。使用单例模式的类，需要自己创建并保存自己的唯一实例，并让客户通过 getInstance 方法来访问这个实例。

在 dom4j 此处的代码中，getInstance 方法采用了“懒汉式”的单例模式实现方法，调用此方法时，首先检查唯一实例 singleton 是否已经创建，如果没有则调用 createSingleton 方法创建一个并返回此实例，有则直接返回此实例。此外，加上 synchronized 关键字，保证多线程安全，不会在多线程情况下创建出多个实例。被称作“懒汉式”的原因是这种实现方式下，只有调用了 getInstance 之后单例才会被创建出来。与之相对的是“饿汉式”，即在单例类中用 static 关键字直接在类装载时初始化一个单例，getInstance 直接返回这个单例即可。除此之外还有一些实现方法，不再赘述。

那么，dom4j 这里为什么要在 DocumentFactory 类里使用单例模式呢？前面我们提到，DocumentFactory 类集合了许多工厂方法，用于生成 Document、Element、Attribute 等接口的具体实现类的实例，可以说内容非常多，因此它如果多次实例化就会占用很多资源。并且，DocumentFactory 类本身的特点也决定了它只需要一个实例就够用。因此对 DocumentFactory 类使用单例，可以有效节省资源。

```
public DocumentFactory() {
    init();
}
```

图表 10 DocumentFactory 的无参构造方法

但是，这里存在一个问题。我注意到，DocumentFactory 类的无参构造方法使用了 public 关键字。而一般的单例模式都要求单例类的构造方法需为私有 private，以保证不会被其他类调用而创造出多个实例。这是为什么呢？

- org.dom4j.**DocumentFactory** (implements java.io.Serializable)
  - org.dom4j.bean.**BeanDocumentFactory**
  - org.dom4j.datatype.**DatatypeDocumentFactory**
  - org.dom4j.datatype.**DatatypeElementFactory**
  - org.dom4j.dom.**DOMDocumentFactory** (implements org.w3c.dom.DOMImplementation)
  - org.dom4j.util.**IndexedDocumentFactory**
  - org.dom4j.util.**NonLazyDocumentFactory**
  - org.dom4j.util.**UserDataDocumentFactory**

图表 11 DocumentFactory 的子类

查看 DocumentFactory 类的继承关系，发现其下有数个子类。其中 BeanDocumentFactory 等几个子类并没有自己的构造方法，需要调用 DocumentFactory 的构造方法，如图 12 所示。

如果 DocumentFactory 的构造方法为 private，就不能被子类继承了。

```
public class BeanDocumentFactory extends DocumentFactory {
    /** The Singleton instance */
    private static BeanDocumentFactory singleton = new BeanDocumentFactory();
```

图表 12 子类对 DocumentFactory 构造方法的调用

（注：从图 12 可看出这个子类也用了“单例模式”，并且采用了“饿汉式”的实现方法）

因此，DocumentFactory 类为了让其子类可以使用它的构造方法而没有把构造方法设为 private，可以说并不是完全的“单例模式”。

当我们查看 DocumentFactory 类中用于创建单例的 createSingleton 方法时，我们会发现它的返回值是一个比较独特的接口 SingletonStrategy。

```
private static SingletonStrategy<DocumentFactory> createSingleton() {
    SingletonStrategy<DocumentFactory> result;

    String documentFactoryClassName;
    try {
        documentFactoryClassName = System.getProperty("org.dom4j.factory",
                "org.dom4j.DocumentFactory");
    } catch (Exception e) {
        documentFactoryClassName = "org.dom4j.DocumentFactory";
    }

    try {
        String singletonClass = System.getProperty(
                "org.dom4j.DocumentFactory.singleton.strategy",
                "org.dom4j.util.SimpleSingleton");
        Class<SingletonStrategy> clazz = (Class<SingletonStrategy>) Class.forName(singletonClass);
        result = clazz.newInstance();
    } catch (Exception e) {
        result = new SimpleSingleton<DocumentFactory>();
    }

    result.setSingletonClassName(documentFactoryClassName);

    return result;
}
```

图表 13 createSingleton 方法

查看 SingletonStrategy 接口的注释，我们会发现这里又运用了另一种设计模式：策略模式。策略模式是一种行为型模式，其意图是针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。体现了封装的思想。它可以避免根据输入进行 if-else 判断来决定选哪种算法，增强扩展性。但是需要为每个算法建一个类，可能导致类的数量过多，并且这时客户端需要知道自己该使用什么算法。

```
/**
 * <code>SingletonStrategy</code> is an interface used to provide common
 * factory access for the same object based on an implementation strategy for
 * singleton. Right now there are two that accompany this interface:
 * SimpleSingleton and PerThreadSingleton. This will replace previous usage of
 * ThreadLocal to allow for alternate strategies.
 *
```

图表 14 SingletonStrategy 注释

根据 SingletonStrategy 接口的注释我们得知，dom4j 中为单例模式提供了 SimpleSingleton 和 PerThreadSingleton 两种策略。SimpleSingleton 是普通的单例模式，而 PerThreadSingleton 则为每个线程创建一个“单例”。不过，查看图 13 中的方法我们不难发现，dom4j 创建 XML 的流程中只使用了 SimpleSingleton 的策略，我搜索整个 dom4j 代码，除了测试文件，也没有找到 PerThreadSingleton 策略使用的地方，应该是留作客户可以使用的一种策略工具，本身并没有在默认的代码逻辑中使用。

除了单例模式、策略模式以外，dom4j 还使用了享元模式 Flyweight、访问者模式 Visitor 等，但这里就不再一一介绍了。