第 15 章　XML 解析

这一章将说明解析 XML 数据时的三种不同方法：两个来自 Qt 的 XML 模块，还有一个是新的，改进的来自 Qt 的 core 模块。这些例子会分别用来说明 SAX（Simple API for XML，XML 简单应用程序编程接口）的事件驱动解析（parse-event-driven parsing）方法、DOM（Document Object Model，文档对象模型）的解析树（tree-style parsing）方法和 `QXmlStreamReader` 的解析流（stream-style parsing）方法。

XML 是 Extensible Markup Language（可扩展标记语言）首字母的缩写词。它是这样一种标记语言，类似于 HTML（Hypertext Markup Language，超文本标记语言）但拥有更加严格的语法且没有定义任何语义（也就是说，其标记并没有关联任何含义）。

XML 严格的语法相较于 HTML 的语法而言，可构成强烈的对比。例如：

- 每个 XML `<tag>`都必须有一个对应的结束`</tag>`，或者是自我封闭，类似于这样的`<br/>`。
- XML 标记区分大小写：`<tag>`与`<Tag>`是不一样的。
- 为了避免解析器产生混淆，在 XML 文档中类似于>和<等实际上不是标记一部分的字符，必须替换成被动等价的字符，例如`&gt;`和`&lt;`。
- 此外，和符号字符（&）必须由它的等价被动字符`&amp;`进行替换，这也是出于同样的原因。

示例 15.1 是一个 HTML 文档，需要注意的是，这个文档并不符合 XML 的规则。

示例 15.1　src/xml/html/testhtml.html

```
<html>
<head> <title> This is a title </title>

 <!--Unterminated <link> and <input> tags are commonplace
        in HTML code:    -->
 <link rel="Saved Searches" title="oopdocbook"
   href="buglist.cgi?cmdtype=runnamed&amp;namedcmd=oopdocbook">
 <link rel="Saved Searches" title="queryj"
   href="buglist.cgi?cmdtype=runnamed&amp;namedcmd=queryj">
</head>
<body>
<p> This is a paragraph. What do you think of that?

HTML makes use of unterminated line breaks: <br>
And those do not make XML parsers happy. <br>

<ul>
<li> HTML is not very strict.
<li> An unclosed tag doesn't bother HTML parsers one bit.
</ul>
```

```
</body>
</html>
```

如果把 XML 的语法与 HTML 的元素语义结合起来，那么就可以得到称为 XHTML 的语言。示例 15.2 给出了示例 15.1 的 XHTML 版本。

示例 15.2　src/xml/html/testxhtml.html

```
<!DOCTYPE xhtml >
<html>
<head>
<title> This is a title </title>
<!-- These links are now self-terminated: -->
<link rel="Saved Searches" title="oopdocbook"
  href="buglist.cgi?cmdtype=runnamed" />
<link rel="Saved Searches" title="queryj"
  href="buglist.cgi?namedcmd=queryj" />
</head>
<body>

<p> This is a paragraph. What do you think of that? </p>
<p>
Html self-terminating line breaks are ok: <br/>
They don't confuse the XML parser. <br/>
</p>

<ul>
<li> This is proper list item </li>
<li> This is another list item </li>
</ul>

</body>
</html>
```

XML 是人类和应用程序都可以理解的一整类文件格式。目前，XML 已经发展成为 Web 应用程序存储和交换数据的流行格式，同时它也是一种擅长于表达层次(树状)信息的语言，而树状信息囊括了大部分的文档。

许多应用程序(例如，Qt 设计师、Umbrello 和 Dia)都使用某种特殊的 XML 文件格式来存储数据。Qt 设计师的 .ui 文件就是使用 XML 来描述一个 GUI 中 Qt 窗件的布局。本书英文版也是采用一种称为 Slacker 的 DocBook[①]的 XML 格式写成的，它类似于专门用来编写图书的 XML 语言 DocBook[②]，但是增加了一些来自 XHTML 的简写标记和自定义标记来描述课件。

XML 文档由节点(node)组成。基本元素(element)是形如<tag>文本或者元素</tag>的节点。一个开放标记(opening tag)可以包含属性，所有属性都有如下的形式：name="value"。相互交织的元素就组成了一种具有父-子关系的树状结构。

示例 15.3　src/xml/saxl/samplefile.xml

```
<section id="xmlintro">
    <title> Intro to XML </title>
    <para> This is a paragraph </para>
```

① 参见 http://slackerdoc.tigris.org。

② 参见 http://www.docbook.org。

```
    <ul>
        <li> This is an unordered list item. </li>
        <li c="textbook"> This only shows up in the textbook </li>
    </ul>
    <p> Look at this example code below: </p>
    <include src="xmlsamplecode.cpp" mode="cpp"/>
</section>
```

示例 15.3 中，<ul>有两个<li>子节点，而它的父亲是<section>。没有孩子的元素可以使用一个/>实现自我终止，例如，<include/>。一些元素拥有属性，例如<section>和<include>。尽管大部分解析器都会忽略多余的空白符，但把嵌套的元素进行缩进有助于提高可读性。

问题

<section>有多少个直接儿子?

XML 编辑器

有几种可用的开源 XML 编辑器。在转向寻求商业化软件方案之前，建议先试用一下这些软件。

1. jEdit[①]有一个使用起来非常不错的 XML 插件(要确保启用了 Sidekick)。
2. 对于 KDE 用户，可以使用 quanta[②]，就像 Kdevelop 一样，quanta 也基于 KDE 的高级文本编辑器 Kate。如果熟悉使用 emacs 的快捷键，那么可以下载并安装 emacs 版本的 Kate 插件：ktexteditor-emacsextensions[③]。

提示

自由工具 xmllint 对于检查 XML 文件中的错误非常拿手，它能够报告描述性的错误(不匹配的开始/结束标记、丢失的字符等)并可指出错误的位置。它可以用来将文档自动缩进成“pretty print”这种格式非常好的 XML 文档。同时，它还可以用来将使用 XInclude 或者外部实体引用(external entity reference)的多个部分的文档合并在一起——这两个特性都是 Qt XML 解析器尚不支持的。

15.1 Qt XML 解析器

XML 被广泛用作程序之间的信息交流媒介，也用作存储和传输层次化数据的一种方便的方式。这就意味着，对于每个可以生成 XML 输出的程序来说，就必须要有一个相应的读取程序来读取它并从中加工出有意义的东西。例如：把由 Umbrello 生成的.xmi 文件转换成图形化的图表，把由设计师生成的.ui 文件转换成屏幕上显示的窗件，把由 DocBook 生成的.xml 文件转换成用于印刷的.pdf 文件或者是可以在网络上阅读的.tml 文件等。要完成这样的转

① 参见http://www.jedit.org。

② 参见http://quanta.kdewebdev.org/。

③ 参见http://www.kde-apps.org/content/show.php?content=21706。

换，就需要能够对 XML 文件进行解析。以上所提到的每个应用程序都有一种解析 XML 的方法，它们使用 SAX, DOM 或者其他的 XML 解析应用程序编程接口(Application Programming Interface, API)。一些 API，如 DOM 和 `QXmlStreamWriter`，还会有一种更高级的生成 XML 文件的方法。

QXmlStreamReader 和 QXmlStreamWriter

从 Qt 4.3 开始，出现了另外一种 XML API：`QXmlStreamReader` 和 `QXmlStreamWriter`。这些基于流(stream)的类位于 QtCore 中，并建议用于新的应用程序，因为它们比 Qt 的 XML 模块提供了更多的灵活性和更好的性能。在 15.4 节，很快就可以看到一个用它建立元素树的例子。

Qt 的 XML 模块包括两个标准(跨语言)的 XML API Qt 实现，分别如下所示。

SAX[①]详细说明了一种用于一系列事件驱动 API 的类，它们可以低层次、顺序访问地解析 XML 文档。它最初是用 Java 开发的，也是打算用于 Java 型的解析器中。要利用 SAX API，就很有必要定义一些回调函数(callback function)，可对各种 SAX 事件进行被动响应，例如：

- 开始标记，结束标记。
- 字符数据(内容)。
- XML 处理指令(用<?和?>封闭的内容)[②]。
- XML 注释(用<!--和-->封闭的内容)。

SAX 可以处理几乎任何大小的 XML 文件。SAX 解析只可以向前处理，SAX 应用程序希望在解析文档的过程中就同时顺序完成它们的处理工作。

DOM[③]是一个可以把 XML 元素表示成导航树结构中对象(即节点)的标准 API。由于 DOM 会把整个 XML 文件加载到内存中，应用程序可以处理的最大文件大小要受限于可用的内存量。DOM 特别适用于需要对 XML 文档各个部分进行随机存取(而不是顺序访问)的应用程序。它不提供解析错误处理或者元素修改方法，但它提供了一个用于创建文档的 API。

要使用 Qt 的 XML 模块，需要在你的项目文件中添加下面一行代码：

```
QT += xml
```

15.2　SAX 解析

在使用 SAX 风格的 XML 解析器时，处理流程就完全取决于从文件或者流中顺序读取的数据。这种控制翻转(inversion of control)就意味着执行中的线程追踪需要一个栈，以跟踪对回调函数的被动调用。而且，Qt 库中的解析代码将会调用你的代码(重载虚函数)。

为了激活解析器，首先要创建一个 reader 和一个 handler，然后将它们连接起来，最后再调用 `parse()` 函数，如示例 15.4 所示。

① 参见http://www.saxproject.org。

② 所谓处理指令，是指一种 XML 节点类型，可以在文档的任何地方出现。原打算是包含应用程序的说明。

③ 参见http://www.w3c.org/DOM/。

示例 15.4 src/xml/sax1/tagreader.cpp

```
#include "myhandler.h"
#include <QFile>
#include <QXmlInputSource>
#include <QXmlSimpleReader>
#include <QDebug>

int main( int argc, char **argv ) {
    if ( argc < 2 ) {
        qDebug() << QString("Usage: %1 <xmlfile>").arg(argv[0]);
        return 1;
    }
    MyHandler handler;
    QXmlSimpleReader reader;                                    1
    reader.setContentHandler( &handler );                       2
    for ( int i=1; i < argc; ++i ) {
        QFile xmlFile( argv[i] );
        QXmlInputSource source( &xmlFile );
        reader.parse( source );                                 3
    }
    return 0;
}
```

1 通用解析器。

2 将各个对象连接在一起。

3 开始解析。

解析 XML 的接口在抽象基类 QXmlContentHandler 中进行描述，可以将它称为被动接口，因为调用 MyHandler 函数的不是自己的代码。QXmlSimpleReader 对象会读取 XML 文件并生成解析事件，然后通过调用 MyHandler 函数对其响应。图 15.1 给出了这一过程中所涉及的主要的类。

为了使 XML reader 能够提供有用信息，需要有一个对象能够接受解析事件，这个对象就是解析事件处理器，它必须实现一个由抽象基类指定的接口，这样才能够“插入”解析器中，如图 15.2 所示。

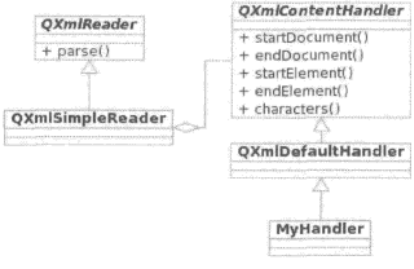

图 15.1 抽象 SAX 类和具体 SAX 类

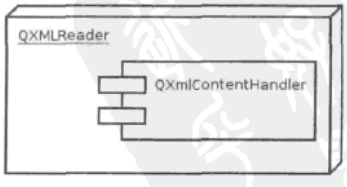

图 15.2 插件组件的结构

处理程序(直接或者间接地)继承自 QXmlContentHandler。在解析过程中遇到 XML 文件的各种元素时，解析器就会调用这些虚方法。无须直接调用这些函数。示例 15.5 给出了一个类，它扩展了默认的处理程序，以便根据应用程序所需的特定方式响应解析事件。

示例 15.5　src/xml/sax1/myhandler.h

```
[ . . . . ]
#include <QXmlDefaultHandler>
class QString;
class MyHandler : public QXmlDefaultHandler {
  public:
    bool startDocument();
    bool startElement( const QString & namespaceURI,
                       const QString & localName,
                       const QString & qName,
                       const QXmlAttributes & atts);
    bool characters(const QString& text);
    bool endElement( const QString & namespaceURI,
                     const QString & localName,
                     const QString & qName );
  private:
    QString indent;
};
[ . . . . ]
```

这些被动调用的函数通常称为回调函数，它们负责响应解析器生成的各种事件。例如，MyHandler 的客户代码是 Qt XML 模块中的 QXmlSimpleReader 类。

ContentHandler 还是 DefaultHandler

QXmlContentHandler 是一个包含许多纯虚函数的抽象类，任何派生实体类都必须重载这些虚函数。Qt 提供了一个 QXmlDefaultHandler 实体类，它在实现抽象基类的纯虚函数时将它们设置为不进行任何操作的空函数。可以把这个类作为实体基类。从这个类派生处理程序时，没有必要重载所有方法，但是为了完成工作必须重载其中的一些方法。

如果没有适当地重载应用程序中用到的每一个方法，那么程序将会调用对应的 QXmlDefaultHandler 方法，即使这些方法不做任何工作。在一个处理程序的函数体中，可以执行下列动作。

- 将解析结果存储在一个数据结构中。
- 根据特定的规则创建对象。
- 按照不同的格式打印或者转换数据。
- 做一些其他有意义的工作。

示例 15.6 包含了一个具体事件处理器的定义。

示例 15.6　src/xml/sax1/myhandler.cpp

```
[ . . . . ]
QTextStream cout(stdout);

bool MyHandler::startDocument() {
    indent = "";
    return TRUE;
}
```

```
bool MyHandler::characters(const QString& text) {
    QString t = text;
    cout << t.remove('\n');
    return TRUE;
}

bool MyHandler::startElement( const QString&,                    1
                              const QString&, const QString& qName,
                              const QXmlAttributes& atts) {
    QString str = QString("\n%1\\%2").arg(indent).arg(qName);
    cout << str;
    if (atts.length()>0) {
        QString fieldName = atts.qName(0);
        QString fieldValue = atts.value(0);
        cout << QString("(%2=%3)").arg(fieldName).arg(fieldValue);
    }
    cout << "{";
    indent += "    ";
    return TRUE;
}
bool MyHandler::endElement( const QString&,
    const QString& , const QString& ) {
    indent.remove( 0, 4 );
    cout << "}";
    return TRUE;
}
[ . . . . ]
```

1 忽略了那些没有使用的参数名，这就避免了编译器总是发出“unused parameter”警告的问题。

传递给 `startElement()` 函数的 `QXmlAttributes` 对象是一个映射，用来存放包含在 XML 元素中 *name* = *value* 的属性值对。

处理文件的过程中，`parse()` 函数在文件中某种“事件”出现时就会相应地调用 `characters()`, `startElement()` 和 `endElement()` 等函数。无论何时，只要在标记的开始处和结尾之间遇到一个普通字符组成的字符串，处理过程都会将它传递给 `characters()` 函数。

将该程序运行于示例 15.3 中的内容，它会将这个文档转换成示例 15.7 中所示的内容，看起来像 LaTeX 的另一种文档格式。

示例 15.7 src/xml/sax1/tagreader-output.txt

```
\section(id=xmlintro){
    \title{ Intro to XML }
    \para{ This is a paragraph }
    \ul{
        \li{ This is an unordered list item. }
        \li(c=textbook){ This only shows up in the textbook }    }
    \p{ Look at this example code below: }
    \include(src=xmlsamplecode.cpp){}}
```

15.3　XML，树结构和 DOM

文档对象模型(Document Object Model，DOM)是操作 XML 的一个高级接口。可以非常直观地处理和浏览 DOM 文档(如果对 QObject 子对象比较熟悉，就更容易懂些)。

图 15.3 给出了 DOM 涉及的主要类。setContent() 函数用于解析文件，在此之后的 QDomDocument 就会包含由 QDomNode 对象组成的一颗结构化树。而每个 QDomNode 可能是 QDomElement，QDomAttr 或者 QDomText 的一个实例。除了作为根的 QDomDocument 之外，每个 QDomNode 都有一个父亲。每个节点都可以从父亲那里找到自己。DOM 可以看成是组合模式(Composite Pattern)的另外一个应用。

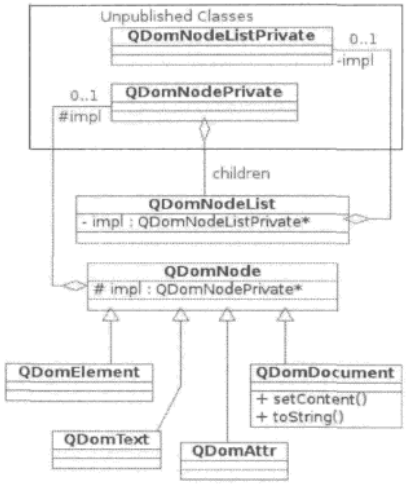

图 15.3　QDom UML 模型

QDom 类是对一些私有实现类的封装。除了指针之外，它们不包含任何其他数据。这使得可以通过值将 QDomNode 传递给其他可能会改变其指向对象(通过添加属性或者儿子或者修改属性)的函数。这使得 QDom 接口就更加类似于 Java 风格的接口了。

15.3.1　DOM 树遍历

Qt 提供对 XML 数据树的完全读/写访问。可以通过一个与 QObject 的接口相类似但稍有不同的接口来遍历节点。在接口的下层，SAX 会直接进行解析操作，DOM 则定义了一个在内存中创建对象树的内容处理器。客户代码所需要做的全部工作就是调用 setContent() 方法，这将会对输入的 XML 和生成的树进行解析。

示例 15.8 在合适的位置将一个 XML 文档进行了转换。在对树进行操作之后，将其序列化为一个 QTextStream 流，这样就可以将其保存或者再次进行解析。图 15.4 给出了这个例子中用到的主要类。

示例 15.8　src/xml/domwalker/main.cpp

```
[ . . . . ]
int main(int argc, char **argv) {
    QApplication app(argc, argv);
    QString filename;
    if (argc < 2) {
        cout << "Usage: " << argv[0] << " filename.xml" << endl;
        filename = "samplefile.xml";
    }
    else {
        filename = argv[1];
    }
```

```
    QFile f(filename);
    QString errorMsg;
    int errorLine, errorColumn;
    QDomDocument doc("SlackerDoc");
    bool result = doc.setContent(&f, &errorMsg,
        &errorLine, &errorColumn);                          1
    QDomNode before = doc.cloneNode(true);                  2
    Slacker slack(doc);                                     3
    QDomNode after = slack.transform();                     4
    cout << QString("Before: ") << before << endl;
    cout << QString("After: ") << after << endl;
    QWidget * view = twinview(before, after);               5
    view->show();
    app.exec();
    delete view;
}
[ . . . . ]
```

1 把文件解析成一棵 DOM 树，然后将树存储在一个空文档中。

2 深层复制。

3 将树发送给 `Slacker`。

4 开始访问。

5 创建一对 `QTreeView` 对象，通过滑动条将其分隔开来，并使用 `QDomDocument` 作为模型。

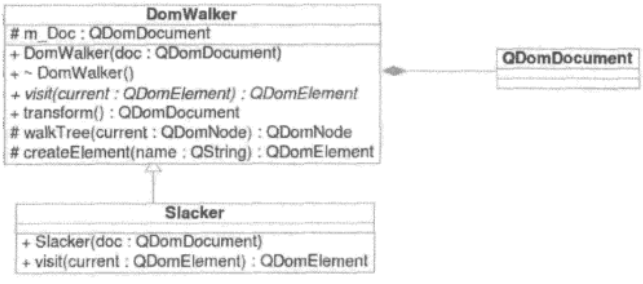

图 15.4 Domwalker 和 Slacker

`Slacker` 派生自 `DomWalker`，这是一个专门用来遍历 DOM 树的应用。

需要注意的是，示例 15.9 中定义的 `walkTree()` 函数没有使用任何指针或者类型转换。这里的 `QDom(Node|Element|Document|Attribute)` 类型是智能指针。我们使用 `QDomNode::toElement()` 或者 `QDomNode::toXXX()` 转换函数来把 `QDomNode` “向下”转换为一个 `QDomElement` 或者 `QDomXXX`。

示例 15.9 src/xml/domwalker/domwalker.cpp

```
[ . . . . ]
QDomDocument DomWalker::transform() {
    walkTree(m_Doc);
    return m_Doc;
}

QDomNode DomWalker::walkTree(QDomNode current) {
```

```
    QDomNodeList dnl = current.childNodes();                    1
    for (int i=dnl.count()-1; i >=0; --i)
        walkTree(dnl.item(i));
    if (current.nodeType() == QDomNode::ElementNode) {          2
       QDomElement ce = current.toElement();                    3
       return visit(ce);
    }
    return current;
}
[ . . . . ]
```

1 首先递归地处理所有儿子。

2 仅处理元素，不会改变其他节点。

3 取代类型转换。

提示

在对一棵树进行遍历时，可以只使用 QDomNode 接口。但是，当操作一个实际的 XML 元素时，“向下转换”为 QDomElement 就可以增加一些将元素及其属性(其本身也是 QDomNode 的子对象)作为整体进行处理的函数，这是非常方便的。

虽然这个例子中 QDomNode/QDomElement 对象是按值传递的，也是按值返回的，但是仍然可以通过临时复制改变底层的 DOM 对象。通过接口欺骗(interface trickery)，QDOM 看起来非常类似于 Java 风格的引用，它们在内部保存了指针而不是实际的数据。

Slacker 定义了将文档从一种 XML 格式转换成另外一种 XML 格式的方法。这是 DomWalker 的一个扩展，其中仅仅重载了一个 visit() 方法。该方法对于每种元素都有一个特定的规则，示例 15.10 中给出了它们的定义。

示例 15.10　src/xml/domwalker/slacker.cpp

```
[ . . . . ]
QDomElement Slacker::visit(QDomElement element) {
    QString name = element.tagName();


    QString cvalue = element.attribute("c", QString()) ;
    if (cvalue != QString()) {                                  1
        element.setAttribute("condition", cvalue);
        element.setAttribute("c", QString());
    }
    [ . . . . ]
    if (name == "b") {
        element.setAttribute("role", "strong");
        element.setTagName("emphasis");
        return element;
}
if (name == "li") {                                             2
    QDomNode parent = element.parentNode();
    QDomElement listitem = createElement("listitem");
    parent.replaceChild(listitem, element);                     3
    element.setTagName("para");                                 4
```

```
    listitem.appendChild(element);
    return listitem;
}
[ . . . . ]
```

1 修改属性——把所有的 c=变成 condition=。

2 这个转换更有意思，因为会用<listitem><para> text </para></listitem>来替换 text 。

3 移除 li 标记，在 li 的位置放一个 listitem。

4 就地修改标记的名称。

运行这个例子时，首先会弹出一个如图 15.5 所示的窗口，这个窗口中带有两个并排放置的树视图，应仔细分析一下转换之前和转换之后的 XML 文档。

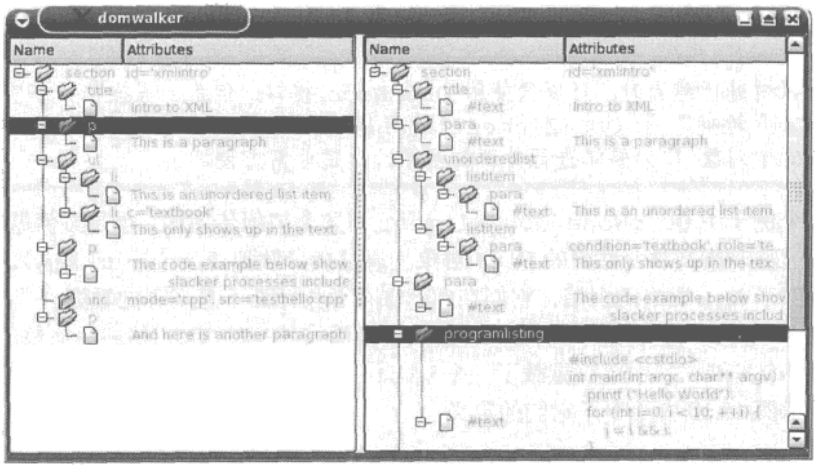

图 15.5 XML 的树视图

15.3.2 用 DOM 生成 XML

DOM 文档通常是由解析器从一个输入流中创建以表示 XML，但是 DOM 也可以用生成的 XML 结构作为输出。一般更倾向于使用 API 而不是通过打印格式化字符串来生成 XML，这是因为 DOM 生成的结果可确保它能被再次进行解析。

图 15.6 中，DocbookDoc 是 QDomElements 的一个工厂类，它从 QDomDocument 派生而来，专门用来创建 Docbook/XML 文档。

在这个类的头文件中，正如示例 15.11 中节选的部分代码一样，我们添加了一个 typedef 语句来提高代码的可读性。在 DOM 标准中，所有的 DOM 类都会分别命名为 Node，Element，Document 和 Attribute。

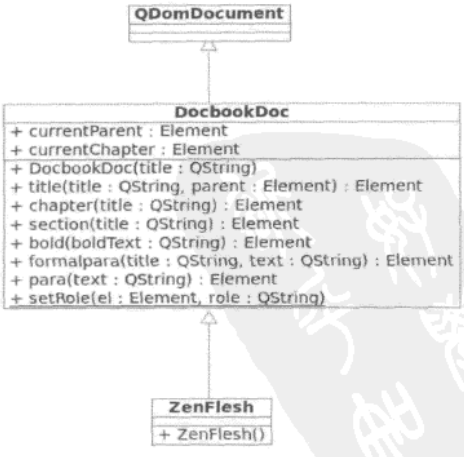

图 15.6 DocbookDoc

示例 15.11　src/libs/docbook/docbookdoc.h

```
[ . . . . ]

typedef QDomElement Element;  1
```

1 节省输入且与 Java DOM 一致。

正如示例 15.12 给出的那样，可以通过创建章、节、段落来构建一个文档。

示例 15.12　src/xml/dombuilder/zenfl esh.cpp

```
#include <QTextStream>
#include <docbookdoc.h>

class ZenFlesh : public DocbookDoc {
    public: ZenFlesh();
};

ZenFlesh::ZenFlesh() :
    DocbookDoc("Zen Flesh, Zen Bones") {

    chapter("101 Zen Stories");
    section("A cup of tea");
    para("Nan-in served a cup of tea.");
    section("Great Waves");
    QDomElement p = para("o-nami the wrestler sat in meditation and "
            "tried to imagine himself as a bunch of great waves.");
    setRole(p, "remark");
    chapter("The Gateless Gate");
    formalpara("The Gateless Gate",
      "In order to enter the gateless gate, you must have a ");
    bold(" mindless mind.");

    section("Joshu's dog");
    para("Has a dog buddha nature or not?");

    section("Haykujo's Fox");
    QDomElement fp = formalpara("This is a special topic",
       "Which should have a role= remark attribute");
    setRole(fp, "remark");
}

int main() {
   QTextStream cout(stdout);
   ZenFlesh book;
   cout << book << endl;
}
```

构造函数使用 XML 生成了一本小书，在经过优美的印刷之后，其结果会类似于示例 15.13。

示例 15.13　src/xml/zen.xml

```
<book>
    <title>Zen Flesh, Zen Bones</title>
    <chapter>
```

```
        <title>101 Zen Stories</title>
        <section>
            <title>A cup of tea</title>
            <para>Nan-in served a cup of tea.</para>
        </section>
        <section>
            <title>Great Waves</title>
            <para>
            o-nami the wrestler sat in meditation and tried
            to imagine himself as a bunch of great waves.
            </para>
        </section>
    </chapter>
    <chapter>
        <title>The Gateless Gate</title>
        <formalpara>
            <title>The Gateless Gate</title>
            In order to enter the gateless gate,
            you must have a <emphasis role="strong">
            mindless mind</emphasis>
        </formalpara>
        <section>
            <title>Joshu's dog</title>
            <para>Has a dog buddha nature or not?</para>
        </section>
        <section>
            <title>Haykujo's Fox</title>
            <formalpara role="remark">
                <title>This is a special topic</title>
                Which should have a role="remark" attribute
            </formalpara>
        </section>
    </chapter>
</book>
```

这种格式的优势是可以使用 xsltproc 和 Docbook/XSL 样式表[docbookxsl]等工具轻松地将它转换成 HTML[①](也可以转换成 PDF 或者 LaTeX)。示例 15.14 给出了生成 HTML 版本的做法。

示例 15.14 src/xml/zen2html

```
#!/bin/sh
# Translates zen.xml into index.html.
# Requires gnu xsltproc and docbook-xsl.
# DOCBOOK=/usr/share/docbook-xsl
xsltproc $DOCBOOK/html/onechunk.xsl zen.xml
```

现在，可以看一看创建元素的示例 15.15。Docbook 语言的每一个主要元素在 DocbookDoc 中都有一个对应的工厂方法。

示例 15.15 src/libs/docbook/docbookdoc.cpp

```
[ . . . . ]

Element DocbookDoc::bridgehead(QString titleStr) {
```

① 参见../docs/src/xml/zen.html。

```
    Element retval = createElement("bridgehead");
    Element titleEl = title(titleStr);
    currentParent.appendChild(retval);
    return retval;
}
Element DocbookDoc::title(QString name, Element parent) {
    Element retval = createElement("title");
    QDomText tn = createTextNode(name);
    retval.appendChild(tn);
    if (parent != Element())
        parent.appendChild(retval);
    return retval;
}

Element DocbookDoc::chapter(QString titleString) {
    Element chapter = createElement("chapter");
    title(titleString, chapter);
    documentElement().appendChild(chapter);
    currentParent = chapter;
    currentChapter = chapter;
    return chapter;
}

Element DocbookDoc::para(QString textstr) {
    QDomText tn = createTextNode(textstr);
    Element para = createElement("para");
    para.appendChild(tn);
    currentParent.appendChild(para);
    currentPara = para;
    return para;
}
```

另外，还有一些仅仅修改文本的字符级别的元素，示例 15.16 中给出了它们的具体示例。

示例 15.16　src/libs/docbook/docbookdoc.cpp

```
[ . . . . ]

Element DocbookDoc::bold(QString text) {
    QDomText tn = createTextNode(text);
    Element emphasis = createElement("emphasis");
    setRole(emphasis, "strong");
    emphasis.appendChild(tn);
    currentPara.appendChild(emphasis);
    return emphasis;
}

void  DocbookDoc::setRole(Element el, QString role) {
    el.setAttribute("role", role);
}
```

因为每个 QDomNode 都必须由 QDomDocument 创建，所以扩展 QDomDocument 来编写自己的 DOM 工厂是很有意义的。

取决于创建的是哪一种元素，DocbookDoc 会将新创建的元素作为孩子添加到之前所创建的元素中。

15.4 XML 流

QXmlStreamReader 和 QXmlStreamWriter 为读取和写入 XML 提供了一个更为快速、更为强大的 API。它们不是 Qt 的 XML 模块的一部分，自 Qt 4.3 以来，它们一直在 QtCore 模块中。

为什么不使用“标准”API

DOM 虽然方便和标准，但需要较大的内存和较高的处理器，而且理应可以进一步简化一些。SAX 虽然速度快且有着相对高效的内存，但对解析过程中创建特定标准结构的需求理应更快些，而且 SAX 也不是很简单，因为要恰当地使用它需要能够理解继承关系并恰当地使用一些纯虚函数。

Qt XML 模块中的接口基于 Java 标准中的相应部分，且其他语言中也有这些实现，因此如果之前使用过它们，就很容易上手。但如果已经对它们有所了解，那么上手 XML 解析的其他 API 也很容易，比如这个。

QXmlStreamReader API 并不会生成树结构。示例 15.17 给出了一个类，展示了如何用该 API 来生成一个树结构。该类本身是 QStandardItemModel，会为每个树节点都创建一个 QStandardItem 实例。

示例 15.17 src/xml/streambuilder/xmltreemodel.h

```
[ . . . . ]
class XmlTreeModel : public QStandardItemModel {
    Q_OBJECT
public:
    enum Roles {LineStartRole = Qt::UserRole + 1,
                LineEndRole};                                    1
    explicit XmlTreeModel(QObject *parent = 0);
public slots:
    void open(QString fileName);
private:
    QXmlStreamReader m_streamReader;
    QStandardItem* m_currentItem;
};
[ . . . . ]
```

1 用于 data() 的自定义角色。

示例 15.18 给出了实际用于解析操作的代码。

例 15.18 src/xml/streambuilder/xmltreemodel.cpp

```
[ . . . . ]
void XmlTreeModel::open(QString fileName) {
    QFile file (fileName);                                       1
    if (!file.open(QIODevice::ReadOnly)) {
```

```
        qDebug() << "Can't open file: " << fileName;
        abort();
    }
    m_streamReader.setDevice(&file);                                    2
    while (!m_streamReader.atEnd()) {
        QXmlStreamReader::TokenType tt = m_streamReader.readNext();
        switch (tt) {
            case QXmlStreamReader::StartElement: {
                QString name = m_streamReader.name().toString();
                QStandardItem* item = new QStandardItem(name);
                item->setData(m_streamReader.lineNumber(),
                              LineStartRole);                           3
                QXmlStreamAttributes attrs = m_streamReader.attributes();
                QStringList sl;
                sl << tr("Line# %1").arg(m_streamReader.lineNumber());
                foreach (QXmlStreamAttribute attr, attrs) {
                    QString line = QString("%1='%2'").arg(attr.name().toString())
                                    .arg(attr.value().toString());
                    sl.append(line);
                }
                item->setToolTip(sl.join("\n"));
                if (m_currentItem == 0)
                   setItem(0, 0, item);                                 4
                else
                   m_currentItem->appendRow(item);                      5
                m_currentItem = item;
                break; }
            case QXmlStreamReader::Characters: {
                QString tt = m_currentItem->toolTip();
                tt.append("\n");
                tt.append(m_streamReader.text().toString());
                m_currentItem->setToolTip(tt);
                break; }
            case QXmlStreamReader::EndElement:
                m_currentItem->setData(m_streamReader.lineNumber(), LineEndRole);
                m_currentItem = m_currentItem->parent();                6
                break;
            case QXmlStreamReader::EndDocument:
            default:
                break;
        }
    }
}
```

1 不在作用域时，自动关闭。

2 开始解析——可以使用任何 QIODevice。

3 自定义 data()。

4 在模型中设置 root 项。

5 作为子节点添加到当前项中。

6 遍历树。

如果运行这个例子，可以看到左侧的 QTreeView 会与右侧文本编辑器中 XML 文件结构的各个元素相互匹配，与图 15.7 类似。鼠标放在树中的元素上，就会显示其中的文字，单击该元素，文本编辑器中的光标会移动到文档中相应的位置。

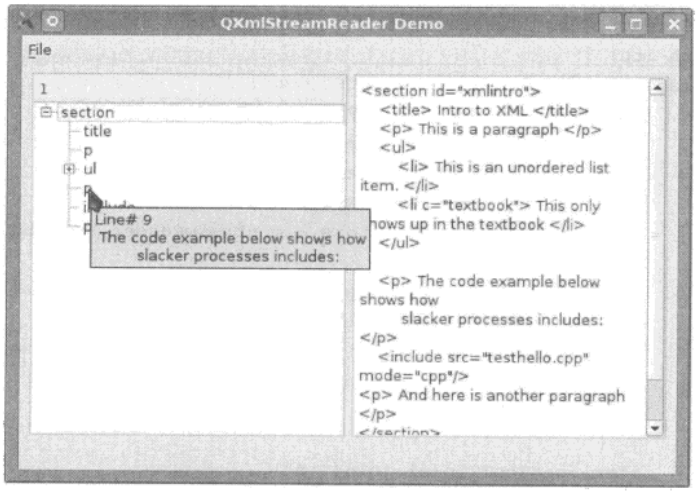

图 15.7 TreeBuilder 示例

15.5 复习题

1. 如果在 XML 文件中存在一个语法错误，那么应该如何判断错误的起因和位置？
2. SAX 是一个事件驱动解析器。它可以响应哪些种类的事件？
3. 对 SAX 和 DOM 做比较。为什么要选择一个而不选择另一个？
4. 如果有一个 QDomNode 且它实际上“指向”QDomElement，那么应该如何引用这个 QDomElement？
5. 使用 QXmlStreamReader 而不是 SAX 的好处是什么？
6. 使用 QXmlStreamReader 而不是 DOM 的好处是什么？