北京工业大学

毕业设计（论文）任务书

题目 XML 数据流节点索引与存储模块的设计和实现

专业 计算机科学与技术 学号 16071127 姓名 王言萱

主要内容：

设计一种在数据流环境下能将 XML 数据流中的各个节点进行编码（前缀码或区位码），并对应处理的有效节点提供索引和存储（以标签流形式存储）。

毕设主要内容包括：

1.学习使用现有的 SAX 解析器，并能够解析 XML 文档，形成模拟的 XML 节点流

2.实现针对 XML 节点流中各个节点的编码，编码方式博包括区位码和前缀码编码

3.在编码的同时，提供带索引的存储，存储用户所需的节点，并提供查询接口。即将用户所需节点存储在标签流中，并提供针对标签流中各个节点的索引访问功能。

4.进行测试与效率测试分析，并撰写毕设论文

基本要求：

熟练运用 C++或 Java 编程语言，利用 Apache 提供的 SAX 解析器，独立完成任务书中所需功能的支持，并进行功能和性能的测试。

主要参考资料：

[1].Apache Xerces C++，http://xerces.apache.org/
[2].A Dissertation by Dipl.-Inf. Christian Mathis. Storing, Indexing, and Querying XML Documents in Native XML Database Management Systems. April 2009
[3].郑莉. C++语言程序设计（第四版） 清华大学出版社 2010 年
[4].Xerces C++ SAX 解析 XML 文档，
http://xerces.apache.org/xerces-c/program-3.html

完成期限：2020-5-31

指导教师签章：

专业负责人签章：

2020 年 2 月 1 日

独 创 性 声 明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京工业大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签名：王言萱　　日期：2020.6

关于论文使用授权的说明

本人完全了解北京工业大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

（保密的论文在解密后应遵守此规定）

签名：王言萱　　导师签名：苏航　　日期：2020.6

摘要

随着网络技术的飞速发展，XML 为 Web 数据管理提供了新的数据模型，被广泛应用于各种领域。XML 已经成为数据表示和交换的新标准。目前人们已经针对高效的 XML 数据查询方面做了很多相关研究，从大量的 XML 数据中查找需要的信息成为 XML 数据库研究的热点问题。小枝模式，亦称作 Twig 模式，是一种效率较高的 XML 查询方法。为了能够应用现有高效查询算法对 XML 流数据进行 Twig 模式匹配，需要编码 XML 数据流中的各个节点，并对应处理的有效节点提供索引和存储。

本课题聚焦于设计并实现一个系统，能读取并解析 XML 文档，存储其数据内容并建立索引，并能读入查询语句来对数据内容进行高效查询。我们采用了顺序读取文件内容，进行处理，不受文件大小的控制的 SAX 解析器来解析 XML 文档，用以顺序表为存储结构的索引树和文本数组作为存储 XML 数据的 DOM。同时我们设计了 XPath 解析器，来读入简单的查询语句，最终让查询语句转化为的查询树，和 XML 文档转化为的文档树进行 Twig 模式匹配，以达到高效查询的目的。

关键字：SAX；XML DOM；小枝匹配

Abstract

With the rapid development of network technology, XML provides a new data model for Web data management and is widely used in various fields. XML has become the new standard for data presentation and exchange. At present, people have done a lot of research on efficient XML data query, and searching the required information from a large number of XML data has become a hot topic in XML database research. Twig schema is an efficient way to query XML. In order to be able to apply the existing efficient query algorithm to Twig pattern matching of XML stream data, each node in the XML data stream needs to be encoded, and provide indexes and storage corresponding to the effective nodes processed.

This topic focuses on the design and implementation of a system that can read and parse XML documents, store and index their data content, and read in query statements to efficiently query the data content. We used the SAX parser, which is not controlled by the size of the file, to parse the XML document. We used the sequential table to realize the index tree and the text array as the DOM to store the XML data. At the same time, we designed an XPath parser to read in simple query statements, and finally made the query statements into the query tree, and Twig pattern matching between the document tree and the XML document into the Twig pattern, so as to achieve the purpose of efficient query.

Keywords： SAX, XML DOM, Twig Pattern

目录

1. 绪论

1.1 课题简介

1.1.1 课题背景

随着网络技术的飞速发展，XML 为 Web 数据管理提供了新的数据模型，被广泛应用于各种领域。XML 作为网络数据存储和交换的主要格式已经成为 W3C 推荐标准，越来越多的网络应用采用 XML 语言进行数据交换和存储，大量的流式数据以 XML 作为数据的传输格式。目前人们已经针对高效的 XML 数据查询方面做了很多相关研究。在针对 XML 数据的查询请求中，经常出现多个结构连接组成的查询模式。以祖先后代关系（AD 关系）和双亲子女关系（PC 关系）组成的树型查询模式被认为是 XML 查询的核心操作，亦称作 Twig 模式[1]。其中 TwigList 算法相比之前的 TwigStack 等算法改进了中间结果数据结构，查询性能得到了显著提升。

为了能够应用现有高效查询算法对 XML 流数据进行高效的查询处理，需要将 XML 数据流中的各个节点进行编码，并对应处理的有效节点提供索引和存储。

要对 XML 数据进行存储，要先读取文档中的数据内容，也就需要对 XML 文档进行解析。因为 XML 是一种树形结构的文档，它有两种标准的解析 API：DOM[2]解析方式和 SAX[3]解析方式。DOM 解析是顺序读取 XML，并在内存中表现为树形结构。但 DOM 使用时内存占用太大，所以我们使用另一种基于流的 SAX 来解析文档，它可以边读取 XML 边解析，并以事件回调的方式让调用者获取数据。在 SAX 解析 XML 文档时，我们采用顺序表存储索引树和文本内容，同时生成标签流以便于进行数据查询。

1.1.2 课题研究任务

课题的主要工作内容包括以下三部分：

（1）使用现有 Apache 等开源的 SAX 解析器解析 XML 文档，形成模拟的 XML 节点流；

（2）实现 XML 节点标签流的生成，并且为每个标签添加相应的编码，以便支持 TwigList 的匹配处理；

（3）对添加编码的节点进行存储并建立索引，以支持最终完成匹配的查询结果的组织。

1.1.3 研究的目的和意义

目前人们已经针对高效的 XML 数据查询方面做了很多相关研究，从大量的 XML 数据中查找需要的信息成为 XML 数据库研究的热点问题。小枝模式整体匹配

是 XML 查询处理方法中效率较高的一种，亦称作 Twig 模式。为了能够应用现有高效查询算法对 XML 流数据进行 Twig 模式匹配，需要将 XML 数据流中的各个节点进行编码，并对应处理的有效节点提供索引和存储。

要对 XML 数据进行存储，要先读取文档中的数据内容，也就需要对 XML 文档进行解析，所以我们要寻找较优的 XML 解析方式，并实现能够存储 XML 数据的同时，能快速从中查询信息，这就要求我们设计易于查询的存储和索引方式，并能生成相应的标签流来应用于 Twig 模式匹配。

1.2 本文结构

第 2 章我们介绍了一些与本文相关的背景知识。第 3 章我们详细的介绍了存储和索引模块的设计。第 4 章我们介绍了 XPath 解析模块和 Twig 模式匹配模块的设计，第 5 章我们介绍了以上设计的实现。第 6 章我们对完成的各个模块和整个系统进行一些测试。

2. 相关知识介绍

2.1 XML 相关知识介绍

2.1.1 XML 定义

XML （eXtensible Markup Language）指可扩展标记语言，标准通用标记语言的子集，是一种用于标记电子文件使其具有结构性的标记语言。

可扩展标记语言有以下特点：

- XML 是一种标记语言，很类似 HTML
- XML 的设计宗旨是传输数据，而非显示数据
- XML 标签没有被预定义，需要自行定义标签。
- XML 具有自我描述性。
- XML 是 W3C 的推荐标准

在表 2.1 中，我们举例出一个具体的 XML 文档，其中，标签<class>嵌套两个<student>标签，每个<student>标签中又嵌套四个标签<name><age><major><id>。它们具有自我描述性描述一个 class 里的一位 student 而这位 student 的 name, age, major, id 分别是 Zhang San，20，computer science，16070001，且这些标签并不是 XML 语言所内置的，是可以根据需要自行定义的，如我们还可以在<student>下加入一个<gender>标签表示性别等等。

2.1.2 XML DOM 定义

XML 文档对象模型（Document Object Model，简称 DOM），定义了所有 XML 元素的对象和属性，以及使程序和脚本有能力动态地访问和更新文档的内容，结构的接口。也就是说，XML DOM 是用于获取、更改、增加或删除 XML 元素的标准。

XML DOM 具有以下特点：

- 用于 XML 的标准对象模型
- 于 XML 的标准编程接口
- XML DOM 立于 W3C 的推荐标准。
- 中立于平台和语言

2.1.3 XML DOM 节点

XML 文档中的每个成分都是 XML DOM 中的一个节点。

XML DOM 文档规定：

- 整个文档是一个文档节点
- 每个 XML 标签是一个元素节点
- 包含在 XML 元素中的文本是文本节点
- 每一个 XML 属性是一个属性节点
- 注释属于注释节点。

如表 2.1 所示，表中展现了一个 XML 文档，整个 XML 文档是一个文档节点，也就是说文档节点可以被看作一个虚节点，它不是一个标签，但是也可以被认为是整个文档的一个虚根。文档第二行的“<!--This is a class example-->”是一个注释节点。之后，标签<class>包含一个<student>标签，一个<student>标签又包含四个标签，分别是<name>，<age>，<major>和<id>，这些标签都是元素节点。标签< name >中包含的文本 Zhang San 构成一个文本节点。标签<class category=”university”>中的属性“category”为一个属性节点，并且，一个标签中可以包含多个属性节点，如可以定义一个形如<class category=”university” size=”mid”>的标签。

表 2.1 XML 文档示例

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- This is a class example -->
<class>
  <class category="university">
    <student>
    <name>Zhang San</ student >
    <age>20</age>
    < major >computer science</ major >
    <id>16070001</id>
    </student>
</class>
```

2.1.4 XML DOM 节点树

XML DOM 把 XML 文档视为一种树形结构，称为节点树，这棵节点树展示了 XML 文档中所有节点和节点间关系，可通过这棵树访问所有节点。

XML DOM 的树结构源于 XML 文档本身的树结构。这棵树从根节点开始，再向在树的最低层级向文本节点延伸。如图 2.1.1，展示了一棵 XML DOM 节点树，从图 2.1.2 中我们可以看出一个 XML 文档由一个文档节点开始展开，逐渐向下延伸到文本节点结束。

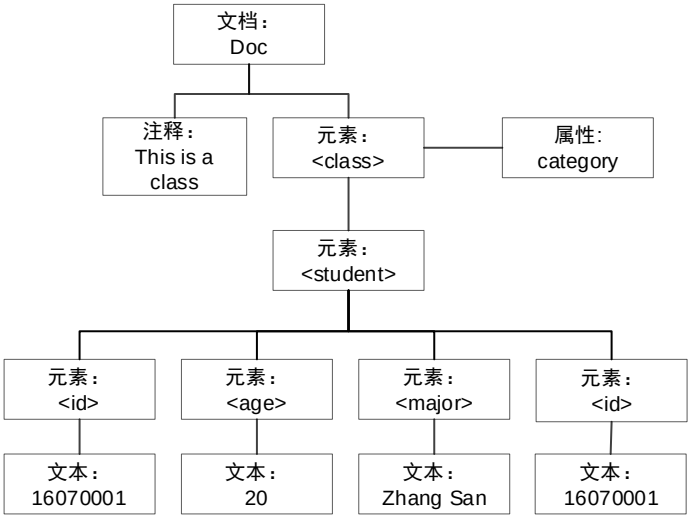


图 2.1.1 XML 树形结构

- **节点树中的节点间的关系**

  此外，节点树中的节点相互之间都有等级关系，包括父子关系和同级关系。节点间关系还有以下特点：

  - 拥有同一个父亲且位于相同层级上的子节点称为同级节点
  - 除了根节点，每个节点都有且只有一个父节点
  - 节点可以有任何数量的子节点
  - 没有子节点的节点是叶子节点

如图 2.1.2 所示，<class>节点和<student>节点之间是父子关系，<class>为<student>的父节点，且<student>为<class>的子节点。位于相同层级上的子节点间存在同级关系，如图中的<class>和注释节点<!—This is a class-->，以及图中的<name>，<age>，<major>和<id>。不过，值得注意的是图 2.1.2 中表示出的四个文本节点并不是同级节点，因为同级节点的前提是：父节点拥有子节点，图 2.1.2 中最下面一层的四个文本节点不具有同级关系，因为他们不属于同一个父亲。

特别地，图 2.1.2 中并未出现属性节点。在 XML 文档中，属性节点位于元素节点标签的内部,如表 2.1 中第三行<class category=”university”>。因此，属性节点不作为元素节点的子节点出现，而是理解为元素节点的内部节点。但是我们通常将该标签看作属性节点的父亲。

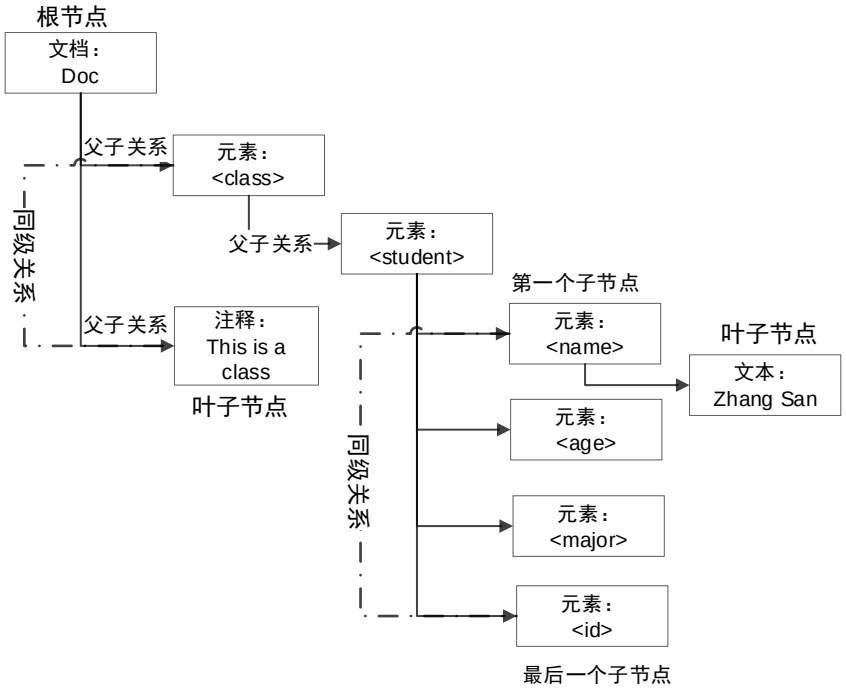


图 2.1.2 XML DOM 节点间关系

2.2 用于 XML 的简单应用程序编程接口

2.2.1 SAX 简介

SAX（simple API for XML）是一种基于流的 XML 解析方法，可以一边读取 XML 数据一边解析，并以事件回调的方式让调用者获取数据。

因为 XML 是一种树结构的文档，它有两种标准的解析 API：

- DOM：一次性读取 XML，并在内存中体现为树结构；
- SAX：以流的形式读取 XML，使用事件回调。

运用 DOM 解析 XML 的优点是用起来方便，但它的主要缺陷是内存占用太大。而 SAX 一边读一边解析，所以无论 XML 有多大，都只占用很小的内存。

2.2.2 SAX 工作机制

SAX 的工作是对文档进行顺序扫描，在读取 XML 文档的时候，同步即时对文档进行处理，例如当扫描到文档开始、文档结束、元素开始、元素结束等地方时触发事件处理函数，由事件处理函数做相应动作，然后继续同样的扫描，直至文档结尾，此时对 XML 文档的解析和处理都完成了，不必等到整个文档被分析储存之后才进行操作。

SAX 处理过程中有一些常用的接口，见图 2.2.1。以下介绍部分常用的接口：

- ContentHandler 接口——内容处理

该接口封装了一些对事件处理的方法，这是大多数 SAX 应用程序实现的主接

口:如果应用程序需要了解基本的解析事件，它将实现此接口并使用setContentHandler方法向SAX解析器注册一个实例。解析器使用实例来触发文档相关的基本事件，比如比如文档的开始与结束、元素的开始与结束等事件。

- DTDHandler接口——文档类型定义处理

如果SAX应用程序需要有关符号和未解析实体的信息,则该应用程序将实现此接口。解析器使用实例向应用程序报告符号和未解析的实体声明。

- EntityResolver接口——解析实体

如果SAX应用程序需要实现对外部实体的自定义处理,则它必须实现此接口并使用该setEntityResolver 方法向SAX驱动程序注册实例。

- ErrorHandler接口——错误处理

是SAX错误处理程序的基本接口。若SAX应用程序需实现自定义的错误处理操作，则它必须实现此接口。实现后，解析器将通过该界面报告错误和警告。

以上接口都通过SAX事件处理程序的默认基类，DefaultHandler类实现。

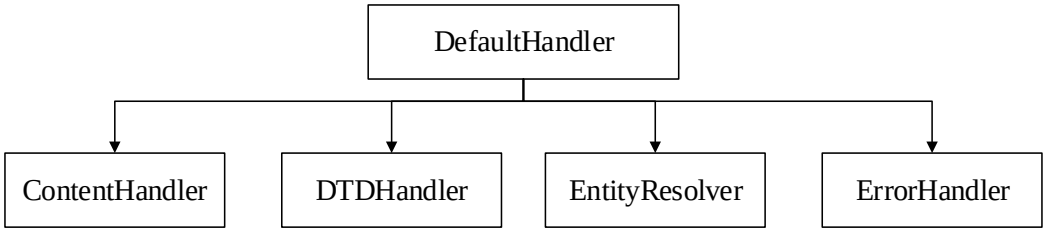


图 2.2.1 SAX的事件处理器接口

如图 2.2.2所示，对SAX的事件处理机制进行简要说明。SAX解析器通过SAX Reader读取XML文件，当触发与文档相关的基本事件，SAX解析器调用ContentHandler()内部的相应方法处理这些事件。处理完事件后返回，继续对XML文档进行扫描。直至文档结束。

图 2.2.3是一个SAX解析XML文档时，元素标签和触发事件的对应。可以看到出，文件开始标签时触发startDocument事件，之后解析到元素开始标签时触发startElement事件，解析到元素结束标签时触发endElement事件。直至文档结束，触发endDocumen事件。

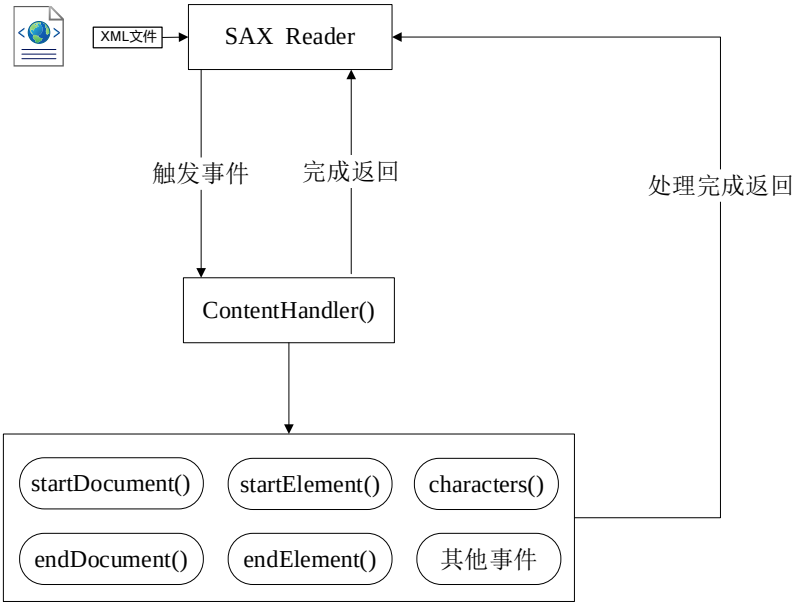


图 2.2.2 SAX 处理机制

	→	触发 startDocument 事件
<!—This is a event-driven example-->	→	触发 comment 事件
<student>	→	触发 startElement 事件
<name>	→	触发 startElement 事件
Zhang San	→	触发 character 事件
</name>	→	触发 endElement 事件
</student>	→	触发 endElement 事件
	→	触发 endDocument 事件

图 2.2.3 XML 文档触发事件过程

2.3 XPath

2.3.1 XPath 简介

XPath 是 W3C 组织制定的一种 XML 数据查询语言，其核心是利用路径表达式对 XML 数据进行检索和查询，获取 XML 文档中的元素集合，并支持在路径表达式中加入约束条件，如节点检测、限定谓语等。

2.3.2 XPath 语法

XPath 利用路径表达式在 XML 中选取节点。如表 2.2 所示，表中列出了 XPath 的基本表达式以及其相应的描述。

表 2.2 XPath 的常用表达式

表达式	描述
nodename	选取此节点的所有子节点。
/	从根节点选取。
//	从当前节点选择文档中的节点
@	选取属性。

表 2.3 举例了一些具体的 XPath 路径表示式和其查询结果。

表 2.3 XPath 的路径表达式

路径表达式	查询结果
a	选取 a 元素的所有子节点。
/a	选取根元素 a
a/b	选取属于 a 的子元素的所有 b 元素
a//c	选择属于 a 元素的后代的所有 c 元素
//@lang	选取名为 lang 的所有属性。

2.4 支持查询的文档编码方法

在 XML 查询中，特别是结构查询，快速判断出 XML 文档树中任意两个结点之间的结构关系对查询效率起到决定性的影响。最流行的方法是文档结点编码方法，通过编码能够惟一确定文档中的某个结点，并且直接比较编码就可以判断出两个结点之间的结构关系，从而加快了 XML 结构查询的计算，同时将查询计算转化为一种结构连接操作。

2.4.1 区位码

下面区位码方法的基本思想是：每一个结点对应一对整数值，这对整数之间的区间表示该结点包含的后代范围。给定任意两个结点 p1 和 p2，它们的区域分别为 r1 和 r2，如果 r1 包含 r2，则 p1 结点是 p2 结点的祖先。

在本文采用了简化的起止编码方式。结点的编码是二元组(start，end)，其中 start 和 end 是深度优先遍历 XML 文档树时产生的序号值，start 值是在开始访问该结点时产生的，end 值是在结束访问它时产生的。

结构关系的判断方法：给定两个结点 p 和 q，它们的区位码分别是 R(p)和 R(q)，则当 R(p).start<R(q).start<R(q).end<R(p).end 时，p 是 q 的祖先结点。

2.5 标签流

标签流[4]是文档树中元素的集合，且每一个这些元素含有相同的标签类型。

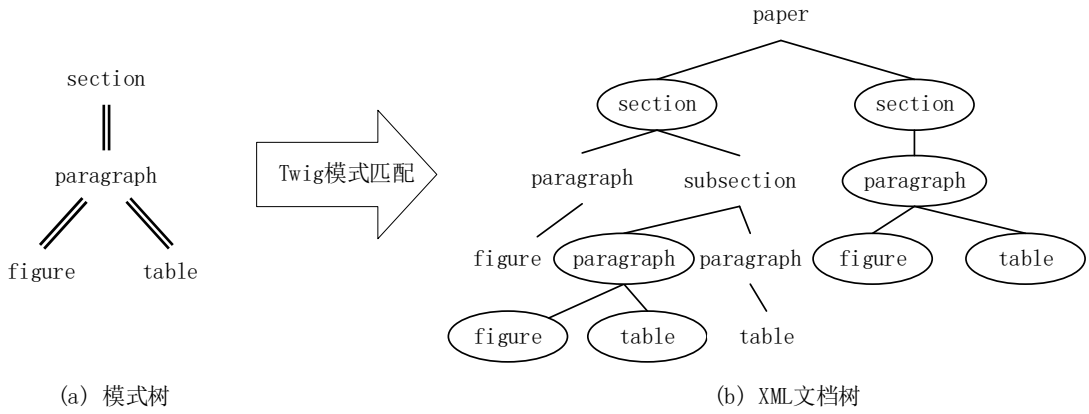
标签流中还包含一些其他的信息。例如，如图 2.5.1 所示的标签流中包含在 DOM 中元素位置的索引。

图 2.5.1 标签流图

2.6 Twig 模式

Twig 模式[6]，又称为小枝模式或模式树，是 XPath 路径表达式被表示成标签树的形式。树中的标签代表表达式中的结点，边代表表达式中结点之间的关系。小枝模式查询就是在 XML 文档树中找出和这个小枝模式相匹配的所有实例。

简单来说，Twig 查询就是一棵 Twig 模式树在文档树上匹配的过程[7]。如图 2.6.1 所示，用大写字母表示模式树的节点，对应 XML 文档中的元素。可以从图中看出找到了两个小枝模式的实例片段。



(a) 模式树　　(b) XML文档树

图 2.6.1 Twig 模式匹配示例

2.7 TwigList 算法

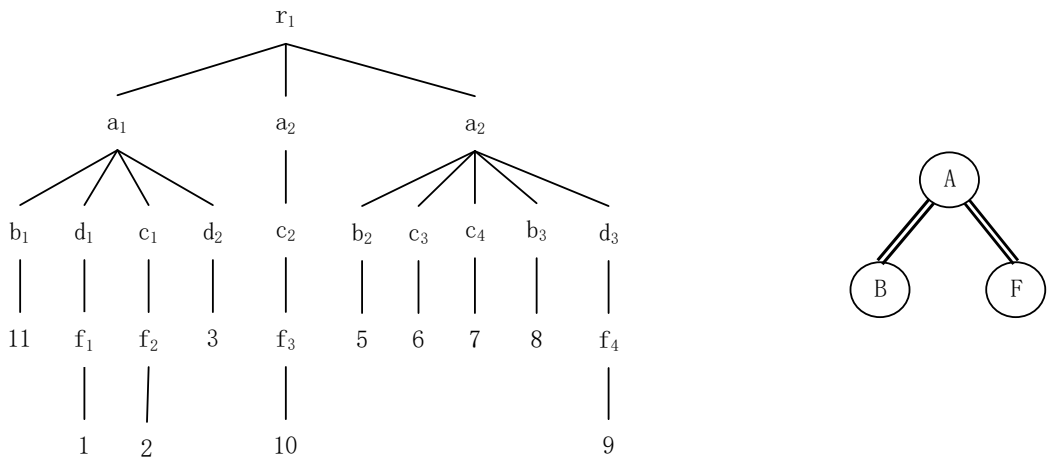
针对 XML 查询处理，研究者提出了许多整体 Twig 查询算法，其中，TwigList[5]算法是其中经典并有效的一种。TwigList 算法由 Qin 等人提出，它改进了之前算法的中间结果数据结构。

此算法采用的是线性表的数据模型，每一个树模式查询节点都对应一个标签流，并再为每个结点 q 分配一个列表 L_q，列表中的每个元素 e_q 都有一对区间指针<start,end>，指向 q 的孩子节点所对应的列表，以确定其中 e_q 的后代范围。

另外，TwigList 算法只为整个 Twig 模式分配一个栈 S，来存放可能匹配 Twig 模式的元素。

TwigList 算法的主要思想是：以区位码的 start 值递增的顺序扫描标签流中的元素，使用栈 S 暂时存储可能匹配 Twig 模式的元素。如果一个元素结点 e_q 是 S 的栈顶元素 em 的后代结点或栈 S 为空，则 e_q 进栈；否则弹出栈顶元素 e_m，直到栈顶元素是 e_q 的祖先，然后 e_q 进栈。被弹出的结点 m 若是叶子结点，或者 m 的孩子结点列表中都有相应的元素结点与 e_m 满足查询树中的关系，则元素结点 e_m 进入对应的列表，若在 $m_i \in children(m)$ 的列表中有其后代元素，那么 e_m 的 start 指针和 end 指针要分别指向 e_m 在 L_{mi} 列表中的第一个后代和最后一个后代。

下面以图 2.7.1 所示的 XML 文档和小枝模式为例来说明 TwigList 算法的执行过程。



(a) XML文档树　　(b) 模式树

图 2.7.1 TwigList 算法实例

查询结点 A、B、F 对应的标签流分别为：$T_A=\{a_1,a_2,a_3\}$，$T_B=\{b_1,b_2,b_3\}$，$T_F=\{f_1,f_2,f_3,f_4\}$。算法读取元素结点的顺序是 a_1，b_1，f_1，f_2，a_2，f_3，a_3，b_2，b_3，f_4。如图 2.7.2 (a)所示，初始状态下，栈和各个结点的列表均为空。算法执行过程中栈和各个列表的内容变化如图 2.7.2 (b)、(c)、(d)所示。

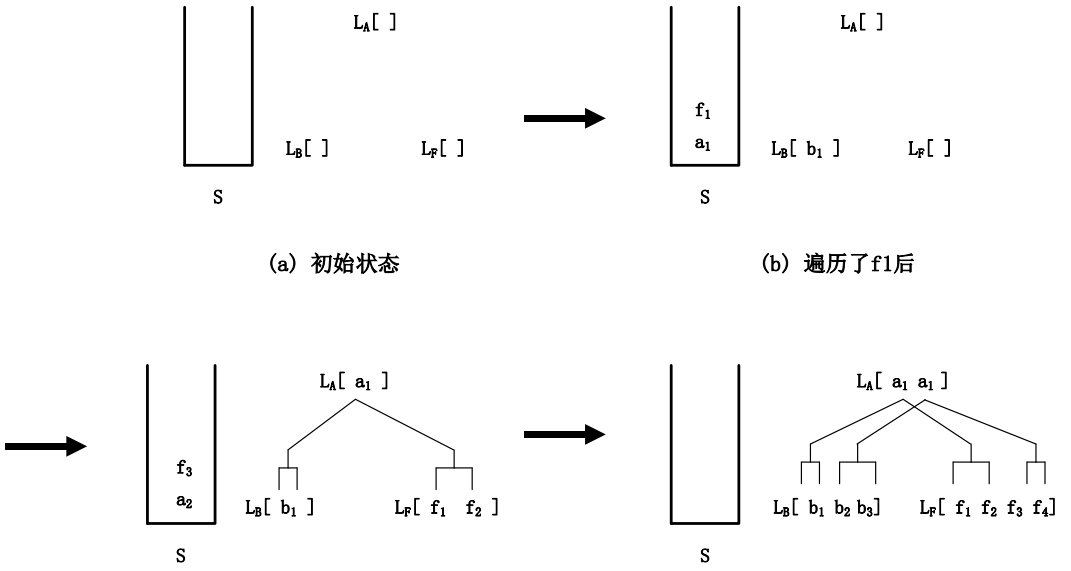


图 2.7.2 TwigList 算法运行时栈和列表示意图

算法运行过程如下：

(1) 读取元素结点 a_1，如图 2.7.2 (a)所示，初始状态栈 S 为空，a_1 进栈；

(2) 继续读取 b_1，b_1 是 a_1 的后代，进栈；

(3) 读取 f_1，f_1 不是栈顶元素 b_1 的后代，所以 b_1 出栈，因为 B 是叶子结点，因此将 b_1 加入列表 L_B 中，f_1 是 a_1 的后代，进栈。此时栈和列表的状态如图 2.7.2(b)所示。

(4) 读取 f_2，f_2 不是 f_1 的后 代，所以 f_1 出栈，因为 F 是叶子结点，所以将 f_1 追加到列表 L_F 中，f_2 是 a_1 的后代，进栈；

(5) 读取 a_2，a_2 既不是 f_2 的后代也不是 a_1 的后代，所以 f_2、a_1 相继出栈，f_2 加人列表 L_F，a_1 在列表 L_B 和 L_F 中都有与其满足祖先后代关系的后代结点，所以将其追加到列表 L_A 中，a_1 的 start 指针和 end 指针指向列表 L_B 和 L_F 中的后代，a_2 进栈；

(6) 读取 f_3，f_2 是 a_2 的后代，f_3 进栈，如图 2.7.2(c)所示；

(7) 读取 a_3，a_3 既不是 f_3 的后代也不是 a_2 的后代，所以 f_3、a_2 相继出栈，将 f_3 追加到列表 L_F 中，a_2 因为在列表 L_B 中没有后代，所以将其舍弃，a_3 进栈；

(8) 读取 b_2，它是 a_3 的后代，进栈；

(9) 读取 b_3，它不是 b_2 的后代，b_2 出栈并追加到列表 L_B 中，b_3 是 a_3 的后代，进栈；

(10) 读取 f_4，它不是 b_3 的后代 b_3 出栈 并追加到列表 L_B 中，f_4 进栈因为它是 a_3 的后代；.

(11) 这时元素已全部读取完，栈中的元素 全部出栈并追加到各自的列表中，如图 2.7.2(d)所示。最后从根结点列表中的元素开始顺着指针输出匹配小枝模式的所有实例树。

3. 索引和存储模块的设计

3.1 XML DOM 设计

3.1.1 XML DOM 数据结构的设计

XML 是标准的树形结构，一般的 DOM 使用链式存储方式，这样虽然能够简单方便的保持其原有结构进行存储，但是链式存储的内存占用较大。因此为了减小使用的内存空间，避免内存溢出等问题，我们采用顺序表存储，使用数组来存储 DOM，通过创建多个文本数组来保存 XML 文档中各个成分的文本信息。

为了提高索引的效率，减少索引树的冗余，我们分开保存文本与索引。创建一个数组来保存 DOM 节点索引树，记录了 XML 节点之间的关系，并创建多个文本数组来分别保存 XML 文档中各个成分的文本信息。索引树描述了 XML 的树结构，并保存节点值保存在文本数组中的位置。我们设计了区分各种节点并保存节点内文本存储的位置的节点特征码，将其保存在索引树中。

我们用一个一维整型数组 blocks[]保存节点索引树，用来保存节点特征码和 XML 的树形结构。表 3.1 中说明了节点特征码的格式，节点特征码是一个 32 位的整型，它保存的节点信息包括命名空间数，属性数，节点名称在数组中的存储位置，该节点有无孩子和节点类型。其中，本课题不涉及命名空间，因此特征码中表示命名空间个数的前 7 位一直置 0；属性数量和节点名称位置可以在通过 XML 文档解析器获取；有孩子置 1，无孩子置 0；节点类型主要分为 4 种，分别为：文档节点 DOCUMENT = 1，元素节点 ELEMENT = 2，属性节点 ATTRIBUTE = 3，文本节点 TEXT = 4。节点特征码格式如表 3.1 所示。

表 3.1 节点特征码格式

命名空间个数	属性个数	节点名称位置	有无孩子	节点类型
NSNum	AttrNum	TextPos	Child	Kind
XXXXXXX	XXXXXXXXX	XXXXXXXXXXXX	X	XXX
7bits	9bits	12bits	1bits	3bits

同时，索引树并不止保存各个节点的节点特征码，还要保存 XML 的树结构，表现各个节点之间的关系。因此一维数组 blocks[]中不是所有的位置都保存节点特征码。

如 2.1.3 章节所介绍的，XML DOM 主要有三种节点，分别是：元素节点、属性节点和文本节点，不同类型的节点不仅节点特征码不同，也在节点的存储结构上有所差异。不同类型的节点和其孩子节点之间有不同的关系，需要存储不同的信息。我们在下面列出：

- **元素节点：**有父节点、可能有子节点、可能包含属性节点

- **属性节点：**无父节点、无子节点
- **文本节点：**有父节点、无子节点、不包含属性节点

根据这些特性，在下列三个表中为不同的节点设计了不同的存储方式。

表 3.2 元素节点的存储结构

节点存储位置	内容
namePos	元素节点特征码
namePos +1	父节点的 namePos
namePos +2	后继兄弟节点的 namePos

表 3.3 属性节点的存储结构

节点存储位置	内容
namePos	属性节点特征码
namePos +1	属性值的位置

表 3.4 文本节点、注释节点的存储结构

节点存储位置	内容
namePos	文本节点、注释节点特征码
namePos +1	父节点的 namePos
namePos +2	后继兄弟节点的 namePos

通过以上的方式对不同类型节点依照其相应的存储方式保存节点信息，我们能够将节点之间的关系完整地记录在索引树中，但我们还需要存储节点的文本信息，这样大量的信息不能存储在索引树中。因此对于各种类型的节点的文本信息，我们设计了三个数组来储存它们，如表 3.5 所示。

表 3.5 文本信息存储结构

数组名称	存储内容
elementNames[]	元素节点名称
attributeNames[]	属性节点名称
textNames[]	文本节点内容以及属性节点的属性值

对应如图 3.1.1 所示的文档， 表 3.6 显示了文本在相应数组中位置具体的存储。

```
<bookstore>
  <book  category="children" lang="en">
    <title>1000 Questions</title>
    <year>2000</year>
    <price>18.8 </price>
  </book>
</bookstore>
```

图 3.1.1 示例 XML 文档

表 3.6 文本在相应数组中的存储

	elementNames[]	attributeNames[]	textNames[]
0	bookstore	category	children
1	book	lang	en
2	title		1000 Questions
3	year		2000
4	price		18.8

表 3.7 为索引树存储数据对应类型的表格，在表格第一列“位置”描述在索引树 blocks 中的位置，表格第二列“存储内容”描述在索引树当前位置存储的内容，表格第三列“值”为 blocks 实际存储的值。

表 3.7 索引存储内容

位置	存储内容	值	位置	存储内容	值
0	元素节点<bookstore>特征码	10	15	后继兄弟节点位置	-1
1	父节点位置	-1	16	元素节点<year>特征码	58
2	后继兄弟节点位置	-1	17	父节点位置	3
3	元素节点<book>特征码	131098	18	后继兄弟节点位置	22
4	父节点位置	0	19	文本节点 2000 特征码	52
5	后继兄弟节点位置	-1	20	父节点位置	16
6	属性节点 category 特征码	3	21	后继兄弟节点位置	-1
7	属性值位置	0	22	元素节点<price>特征码	74
8	属性节点 lang 特征码	19	23	父节点位置	3
9	属性值位置	1	24	后继兄弟节点位置	-1
10	元素节点<title>特征码	42	25	文本节点 18.8 特征码	68
11	父节点位置	3	26	父节点位置	22
12	后继兄弟节点位置	16	27	后继兄弟节点位置	-1
13	文本节点 1000 Questions 特征码	36	15	后继兄弟节点位置	-1
14	父节点位置	10	16	元素节点<year>特征码	58

下面就对于存储的细节信息进行详细阐述。如表 3.8 展示了图 3.1.1 的 XML 文档中的 book 节点的节点特征码。该节点特征码通过获取 book 节点的命名空间数，属性数以及节点名称位置，节点类型；并在解析和存储完后续节点之后，确定此节点有无孩子。获取节点特征码所需信息之后，将其编码并存储在索引树中。

表 3.8 <book>节点特征码实例

含义	命名空间个数	属性个数	节点名称位置	有无孩子	节点类型
二进制	0000 000	0 0000 0010	0000 0000 0001	1	010
实际情况	0	2	1	有	2
十进制	131098				

下面详述如何解析图 3.1.1 的示例 XML 文档，将相应的信息存储入索引树和 DOM 的三个文本数组中。结合表 3.7 和表 3.8 来看，首先读到元素节点<bookstore>，我们将文本”bookstore”存入 elementNames[]中，将元素节点<bookstore>的节点特征码存入索引树顺序表 blocks 中，此时因为是第一个节点（不将文档节点存入索引树中），所以存入的位置是 blocks[0]。因为<bookstore>是元素节点，所以索引树顺序表中的下一格，即 blocks[1]，存储<bookstore>的父节点，而<bookstore>无父节点，所以存入-1。接下来，再下一格，blocks[2]存储<bookstore>的后继兄弟节点，也不存在，所以也存入-1。按照这样的规律，程序读到第二个节点，元素节点<book>时，会将文本“book”存入 elementNames[]，而索引树 blocks[3], blocks[4], blocks[5]分别存入<book>的节点特征码(如表 3-8 所示为 131098)，父节点的位置 0，后继兄弟节点的位置-1，即不存在。特殊的是，<book>节点含有两个属性 category 和 lang，所以我们将文本”category” 和 ” lang” 存入 attributeNames[] 中，此时它们在 attributeNames[]中的位置按顺序排是 0 和 1，所以接下来将属性节点 category 的节点特征码，属性值位置 0 存入 blocks[]中，将属性节点 lang 的节点特征码，属性值位置 1 存入 blocks[]中。接下来的节点也按上述规律存储，即可得到表 3.7 和表 3.8。

3.1.2 XML DOM 节点的设计

在上一章节 3.1.1 中，介绍了 DOM 内部数据结构的存储，XML DOM 在使用时经常被封装成 Node 对象，并提供各种接口，下面将介绍节点的封装。

我们根据不同的节点类型和它们的数据结构封装了 3 种节点：元素节点（ElementNode），属性节点（AttributeNode）以及文本节点（TextNode），他们都继承于一个父类 DOMNode。图 3.1.2 中给出了节点间的继承关系，DOMNode 有两个成员变量，blocks 是存储索引树的一维整型数组，index 是该节点的节点特征码在 blocks 中的位置，即该节点在索引树中的位置。

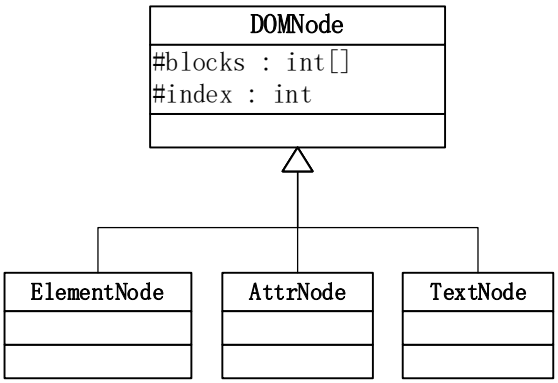


图 3.1.2 DOM 节点继承关系

3.1.3 XML DOM 接口的设计

所谓接口，外界使用和修改 DOM 中数据的通道，由于在从 DOM 中获取 XML 信息时通过将其封装为 DOMNode 的方式，DOMNode 中提供的方法就作为 DOM 对外的接口。根据一些已有的按照 W3C 标准实现的 DOM，我们设计了一些可以对 DOM 查询的方法，在表 3.9 中给出了 DOMNode 对外提供的方法。

表 3.9 DOMNode 对外提供的方法

方法	返回类型	描述
getNodeName()	String	获取节点的名字
getNodeType()	NodeType	获取节点的类型
getNodeValue()	String	获取节点的值
getParentNode()	DOMNode	获取父节点
getFirstChild()	DOMNode	获取第一个孩子
getChildNodes()	ArrayList<DOMNode>	获取所有的孩子
getNextSibling()	DOMNode	获取后一个兄弟

3.2 区位码设计

如 2.4 节的定义，区位码是每一个结点对应一对整数值，这对整数之间的区间表示该结点包含的后代范围。

我们设计一个节点的区间编码是一个二元组(start,end)，start 和 end 分别表示结点在文档树中前序遍历的起始和结束位置。图 3.2.1 中描述了一个 XML 文档树，以及文档树节点对应的区间编码.

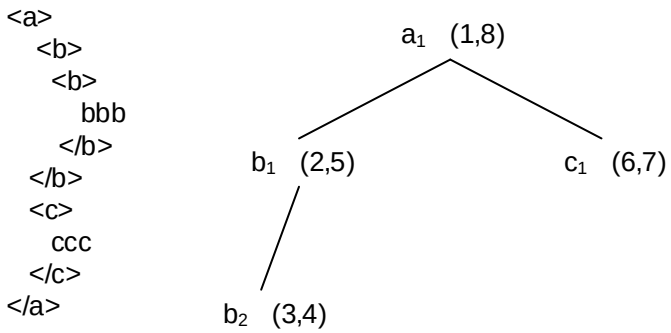


图 3.2.1 示例文档及其对应的文档树

3.3 标签流设计

如 2.7 节的定义，标签流是文档树中元素的集合，且每一个这些元素含有相同的标签类型。标签流中还包含一些其他的信息。首先我们设计标签流中每一个标签储存的内容。

为了支持 TwigList 的匹配处理，我们为每个标签添加区位码，并且储有元素（结点）对应在 DOM 中元素位置的索引。故每个标签储存的内容是

- 序列号
- 区位码
- DOM 中元素位置的索引

例如，对应图 3.2.1 所示的示例文档，解析出的标签流为

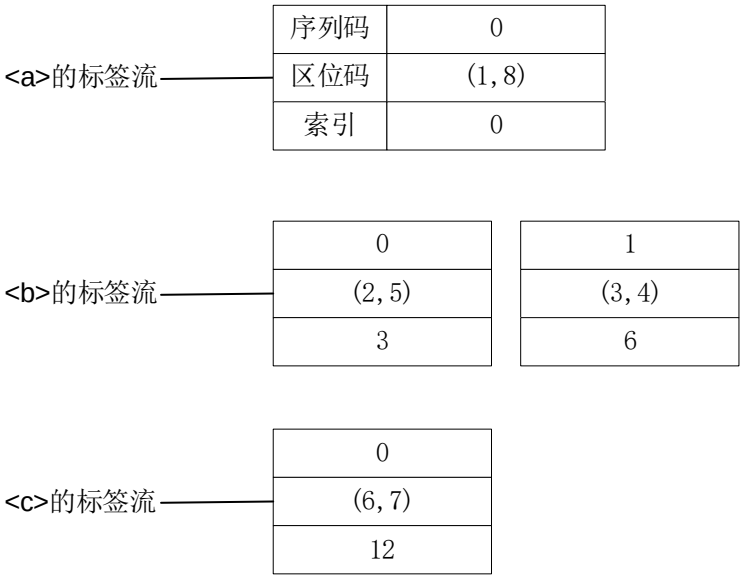


图 3.3.1 标签流示意图

对于每一个标签，我们设计了一个 Label 类，如图所示，Label 类含有 4 个成员变量，分别是序列码 num，区位码的 start 和 end，以及该标签所存储文本在 DOM 的索引，及在索引树中的位置。其类图如图 3.3.2。

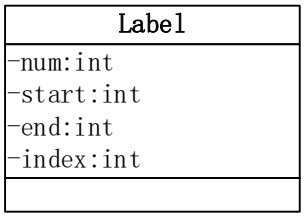


图 3.3.2 Label 类图

定义了标签,如何获取标签流呢？我们设计了用 Map<String,List<Label>> 来存储标签流，其中 Map 的 key 值是标签的文本，value 值是一个 Label 的数组，存储含有相同的标签类型的所有标签。这样我们就可以很容易地存储和获取每一个标签，以及它在 DOM 中的位置。

4. XPath 查询和 Twig 模式匹配的设计

4.1 XPath 的设计

4.1.1 XPath 模块功能

为了实现可以进行模式匹配的查询组织，我们除了 XML 索引和存储模块提供文档树之外，我们还需要读入简单的 XPath 语句，并生成查询树，以最终完成文档树和查询树的匹配。所以我们编写一个简化版的 XPath 的处理模块，仅支持生成只有一个根节点的无绝对路径的查询，即 AD 关系的查询，例如 XPath 语句“//a//b|//a//c//f”是可用的。此模块的输入为 XPath 语句，输出为一棵查询树，并被 Twig 模式匹配模块调用。

4.1.2 XPath 模块类设计

- NodePair 类

其类图如图 4.1.1 示，含有一对成员变量 ancestor 和 desendant，代表了查询树上两个相接节点的祖孙关系。在图 2.7.1(b)所示的查询树中，含有 2 对 NodePair，分别是<A,F>和<B,F>。

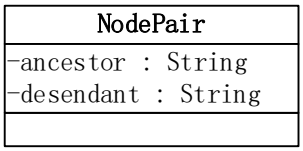


图 4.1.1 NodePair 类图

- XPathTree 类

该类代表了一棵查询树，其类图如图 4.1.2 所示。它的成员变量 nodePairs 是一个标签对的数组，用来保存查询树中节点之间的关系；qEleNames 则用来保存查询树中所有元素的名字，以便在匹配时于提取需要的标签流。它的两个方法分别用来解析 XPath 语句和判断查询树中的某个节点是否是叶子节点，这两个方法都将在 Twig 模式匹配模块中被调用。

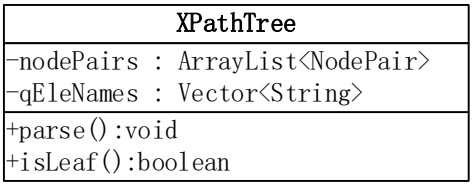


图 4.1.2 XPathTree 类图

4.2 Twig 模式匹配的设计

4.2.1 匹配模块功能

在章节 2.6 中的说明，Twig 模式匹配就是在 XML 文档树中找出和查询树相匹配的所有实例。 所以我们在匹配模块中调用索引和存储模块，读取其生成的文档树，并调用 XPath 模块，读取其生成的查询树，在此模块中用 TwigList 算法将它们匹配，生成查询结果。

4.2.2 匹配模块类设计

我们设计了 TwigList 类来保存匹配的结果，其类图如所图 4.2.1 示。它的成员变量 LMap 用来保存每一个树模式查询节点和每个结点 q 对应的列表 L_q。S 则是用来存放可能匹配 Twig 模式的元素。它的两个方法分别用来进行匹配并存储结果和打印结果。

TwigList
-LMap : Map<String,ArrayList<Element>> -S : Stack<Integer>
+match():void +printResult():void

图 4.2.1 TwigList 类图

- Element 类

其类图如图 4.2.2 所示。该类表示标签流中每种元素 q 对应的列表 L_q 中的每个元素 e_q。

Element
-label : Label -pointerMap : HashMap<String,Pointer>

图 4.2.2 Element 类图

- Pointer 类

其类图如图 4.2.3 所示。该类表示标签流中每种元素 q 对应的列表 L_q 中的每个元素 e_q 都有一对区间指针<start,end>，指向 q 的孩子节点所对应的列表，以确定其中 e_q 的后代范围。

Pointer
-start : Element -end : Element

图 4.2.3 Pointer 类图

5. 系统实现

5.1 系统总体结构

整个系统主要分为三个部分：

(1)索引和存储模块。其包括 SAX 解析器和 DOM 生成器。接收 XML 数据，经过 SAX 解析为 XML 信息，再经过 DOM 生产器形成 DOM，索引树和标签流。

(2)XPath 解析模块

(3)Twig 模式匹配模块

其中，索引和存储模块读取 XML 文档，生成文档树，XPath 解析模块读取 XPath 语句，生成查询树，而 Twig 模式匹配模块查询出符合查询树的所有文档树中的实例。图 5.1.1 表现了系统的总体结构。

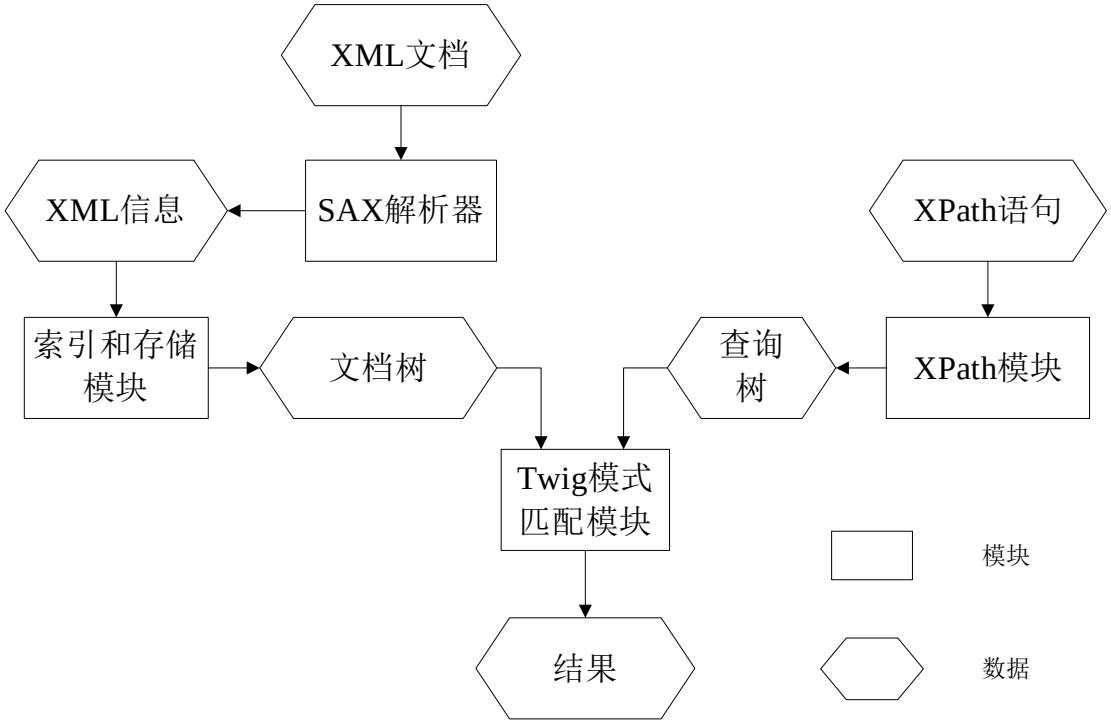


图 5.1.1 系统结构图

其中数据的相关关系可由图 5.1.2 体现。索引树中保持了各个标签内容在 DOM 中的位置以及标签之间的关系。标签流中提取了 Twig 模式查询树中各个查询节点的标签流，其中各个标签存储了代表节点关系的区位码和代表了标签内容的索引。

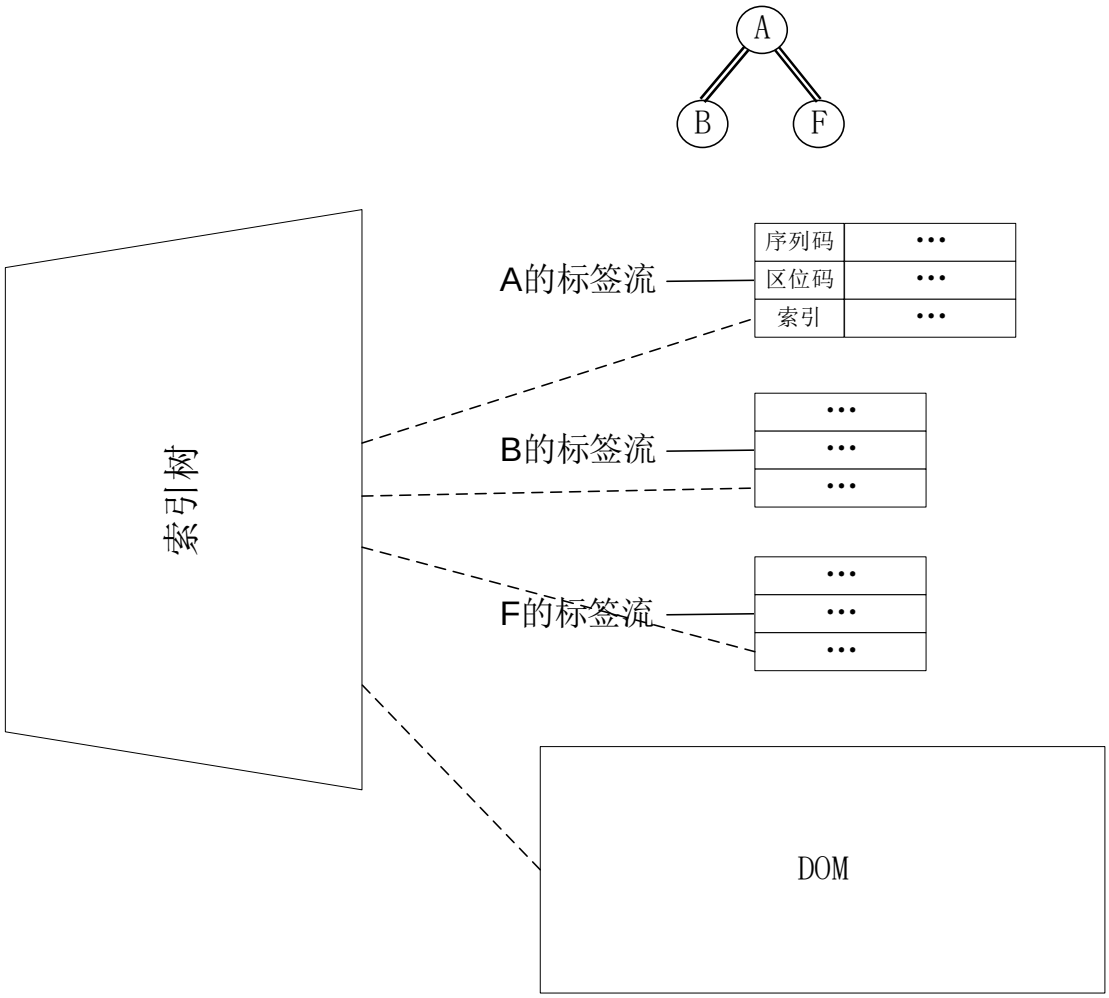


图 5.1.2 系统数据联系图

5.2 XML DOM 模块的实现

5.2.1 主要功能

XML DOM 模块读入 XML 文档并将其解析为文档树，顺序存储在内存中，在生成的 DOM 的基础上，我们将 DOM 转化为三种类型的 DOM 节点，并对外提供一些接口以支持 Twig 模式匹配。概要设计图见图 5.2.1。

实现 DOM 模块有几个关键的步骤：实现 SAX 解析→通过 SAX 解析将 XML 文档转换为 DOM→DOM 节点的实现，我们将在下面的章节一一说明。

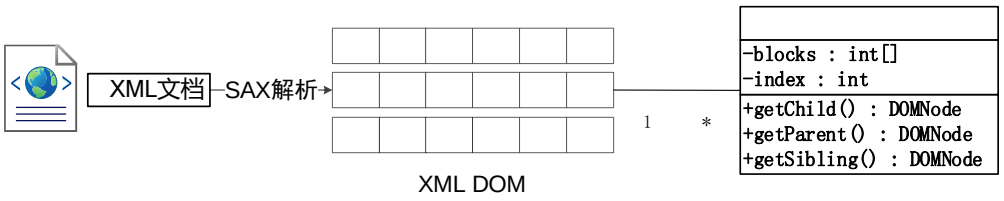


图 5.2.1 顺序表结构 XML DOM 部分概要图

5.2.2 实现 SAX 解析

SAX 在诞生之初，就是一个纯 Java 的 API，所以我们可以很容易地在 Java 程序中使用 SAX 来解析 XML 文档。

在前面章节 2.2.1 中，介绍了有关 SAX 这个 API，所以我们将其直接引入项目中。参照 SAX 官方给出的例子，很容易的就能将 SAX 解析器直接应用于的项目之中。在实现 SAX 解析后，就可以编写在 SAX 的事件触发函数，下面介绍如何通过 SAX 实现 XML 文档到 XML DOM 的转化。

5.2.3 通过 SAX 将 XML 文档转换为 DOM

- **DOM 节点及接口的实现**

在 SAX 扫描 XML 文档的每个标签时，触发相应的函数，将标签中的信息存储进 XML DOM 中。DOM 是重要的 XML 数据信息存储的载体，我们依据章节 3.1.1 中 DOM 数据结构的设计，创建了 DOM 类，下方的图 5.2.2 为 DOM 类的类图。

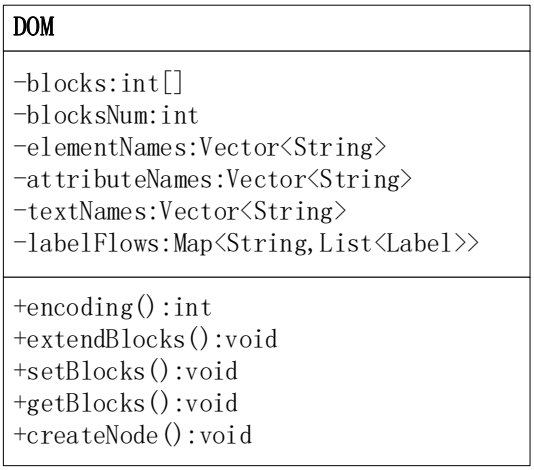


图 5.2.2 DOM 类类图

DOM 类中的各个方法的详细描述如下 5.1 所示。

5.1 DOM 类方法

函数名称	类型	描述
encoding	int	获取节点信息，返回节点特征码
expandBlocks	void	扩展存储空间
CreateNode	void	将 DOM 池中的信息传递给创建 XML DOM 节点函数 DOMNodeCreate

- **事件触发函数部分**

章节 2.2.2 中已经提到，事件触发函数由 ContentHandler 接口实现，而 ContentHandler 接口则需由 DefaultHandler 类实现。所以，为了实现事件触发函数并且将 XML 文档转化为 XML DOM，我们了创建了继承自 DefaultHandler 的 MyHandler 类来重写事件触发函数。MyHandler 类的类图如图 5.2.3 所示。DefaultHandler 提供的事件触发函数对应 XML 文档中各个标签，但其内部没有任何处理，所以 MyHandler 类继承 DefaultHandler 类，并重写事件触发函数来实现本课题的顺序存储的 XML DOM。

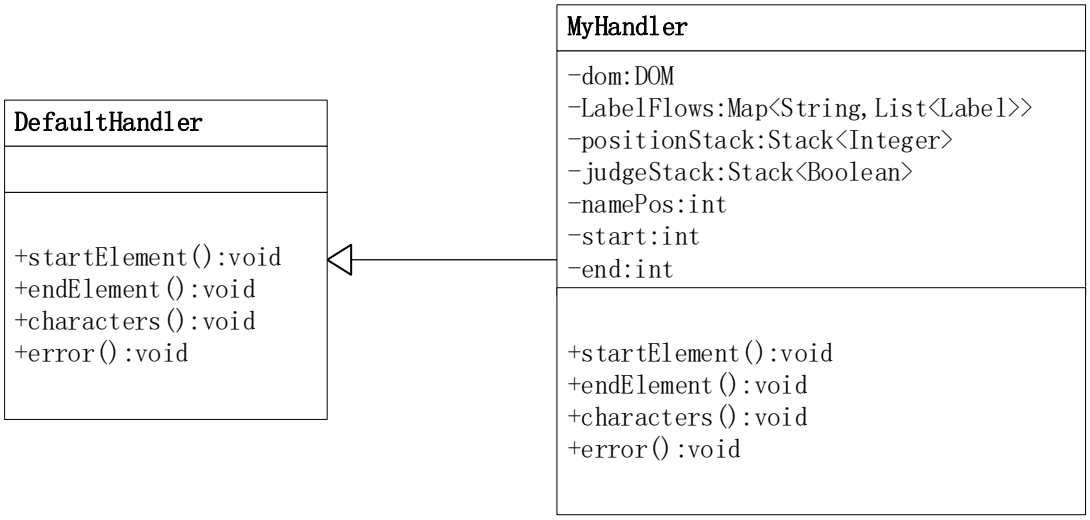


图 5.2.3 DefaultHandler 与 MyHandler 类图与关系图

如图 5.2.3 示，MyHandler 中的各个事件触发函数一一对应 XML 文档中的各种标签。例如：当扫描到 XML 文档中的<student>标签时，便会触发 startElement 事件，调用 MyHandler 中的 startElement 函数进行处理。此时 SAX 会传来此标签解析出的一系列参数，如属性个数，节点名称等。所以，我们可以在扫描该节点标签时保存各个类型的节点信息。存储规则在章节 3.1.1 中已详细介绍。然而，当我们顺序扫描时，只能获取单个节点的信息，无法获取节点之间的联系。我们需要存储在索引树中的内容常常有该节点的节点特征码，该节点的父节点位置和右兄弟节点位置。我们可以用 SAX 解析出的参数编码此节点的节点特征码，可是如何获取父节点位置和右兄弟节点位置呢？此时，我们使用了回填算法，即使用 SAX 顺序解析 XML 文档时，将已知的内容保存到索引树中并依索引树各节点存储方式保留出父节点、兄弟节点位置（先置为-1），同时依次将每个节点该及节点在索引树中的位置压入位置栈中，将节点访问状态压入判断栈中，在遇到元素结束标签时进行出栈并且回写节点父子兄弟位置信息。

观察 MyHandler 类的类图可以发现，我们设置了两个栈来记录节点信息以方便之后回填父节点和右兄弟节点的位置信息：位置栈 positionStack 和判断栈 judgeStack。由于 XML 文档是树结构的，因此我们可以很方便地用栈来保存这种结构。表 5.2 说明了这两种栈属性的作用。

表 5.2 主要栈的数据类型及作用

栈/队列	数据类型	作用
栈 positionStack	Integer	保存节点位置
栈 judgeStack	bool	判断节点是否出栈，帮助判断父兄关系

这两个栈的具体工作模式如下：

1. 遇到元素开始标签：

将节点特征码的位置/0（未被访问过）对应入栈

2. 遇到文本或注释节点：

将节点特征码的位置/1（被访问过）对应入栈

3. 遇到元素结束标签：

出栈至判断栈中栈顶为 0（所有出栈元素为当前栈顶元素的孩子，且他们互为兄弟），边出栈边设置右兄弟位置，出栈完成设置栈顶元素有孩子且所有孩子的父节点位置为栈顶元素节点特征码位置，设置判断栈栈顶为 1（元素被访问过）

注：位置/判断栈同时入栈出栈，元素一直保持对应

例如，假设 XML 文档信息如图 5.2.4 所示。

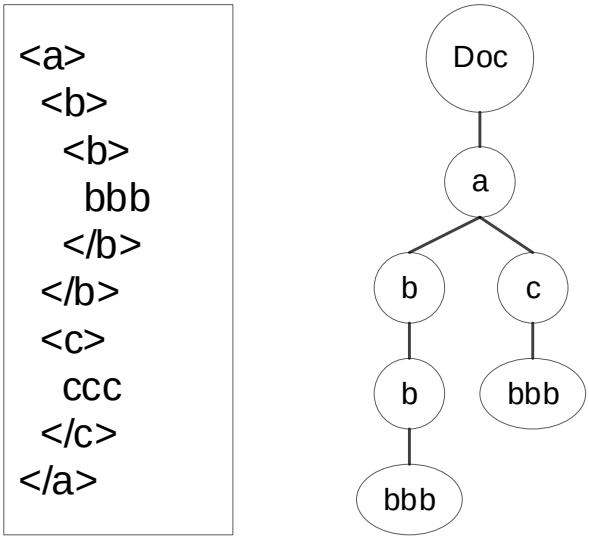


图 5.2.4 XML 文档及其节点树实例

图 5.2.5 说明了在解析图 5.2.4 所示的 XML 文档过程中某些时刻的栈中状态。通过这些状态图，可以清晰地了解在触发各个事件函数时是如何进行节点信息回填的。以下对图 5.2.5 中的重要状态进行解释说明。如图中第一行所示，文档开始，读入元素节点 a 的开标签，即把这个元素节点<a>的索引压入位置栈中，0 压入判断栈中。接下来读入两个元素节点的开标签<b>和<b>，注意这两个节点是不同的，它们互为父子关系，自然索引也不同，在这一段解释说明中，我们将区分为 b(1)和 b(2)。将它们的索引依次压入位置栈中，将 0 和 0 依次压入判断栈中，如图中第三行所示，现在位置栈中的状态为 a b(1) b(2)，判断栈中

的状态为 0 0 0。接下来读入文本节点 bbb，将 bbb 的索引，即其在索引树中的位置压入位置栈，而因为文本节点没有孩子，所以判断栈中直接压入 1。继续扫描文档，读到了第一个闭标签</b(2)>，此时文本节点 bbb 可以出栈，文本节点 bbb 的父节点和兄弟节点信息均可回填，同时在判断栈中节点 b(2)对应的位置也可置 1，即已访问。接下来，读到 b(1)的闭标签，由于元素节点 b(2)已被访问，因此可以出栈，并且回填它的父节点和右兄弟节点位置。按照以上规律继续扫描文档，记录信息，到最后一行读入元素节点 a 的闭标签，a 所有内部标签都已匹配。此时，节点 b(1)和 c 都为 a 的孩子，且节点 b(1)和 c 都已被访问过。所以，扫描到 a 的闭标签时，节点 b(1)和 c 均可以出栈。

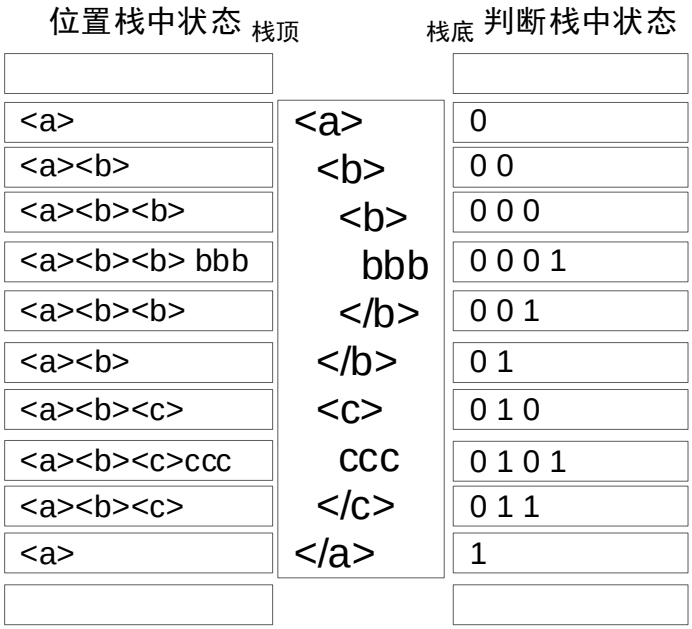


图 5.2.5 回填时两个栈的状态

综上所述，SAX 解析文档时触发不同的 XML 文档标签，调用各自的事件触发函数，并进行相应的栈操作来生成 XML DOM。

下面详细说明各个触发事件函数的实现。

- startElement 元素开始事件

表 5.3 为元素开始事件函数的参数和返回值。在该函数中需要获取所扫描的元素节点的名称以及该节点的属性元素集合。

表 5.3 元素开始事件函数

函数	参数	返回值
startElement	String uri, //命名空间 uri String localname, //元素名 Stringt qname, //限定名 Attributes attributes //属性元素集合	void
功能	将元素节点的信息写入 DOM 中	

下面是元素开始事件函数 startElement 的具体描述：

```
1. 检测存储空间是否足够，不足时进行扩展
2. 属性数量 = attrs.getLength();
3. //namePos 为目前要存入的位置
4. pos 为元素名字在 elementNames 中的位置
5. //元素存入索引树
6. if(元素名字存在于 elementNames 中){
7.       pos=元素名字在 elementNames 中的位置;
8. }
9. else{
10.      把元素名字存入 elementNames;
11.      pos=元素名字在 elementNames 中的位置;
12.    }
13.    将编码后的节点特征码存入 block[namePos];
14.    namePos++;
15.    位置栈入栈(当前 pos);
16.    判断栈入栈(false);
17.    将-1 存入 dom.blocks[namePos+1]  //父节点位置暂时设为空;
18.    将-1 存入 dom.blocks[namePos+2]   //右兄弟节点位置暂时设为
空;
19.    pos+=3;  //在索引树中保留出父节点和兄弟节点位置的空位，之
后回填
20.    /*属性存入索引树*/
21.    for(int i(0);i<属性数量;++i){
22.          attrPos 为元素名字在 attributeNames 中的位置
23.          if(属性名字存在于 attributeNames 中){
24.            attrPos=元素名字在 attributeNames 中的位置;
25.          }
26.          else{
27.            把元素名字存入 attributeNames;
28.            attrPos=元素名字在 attributeNames 中的位置;
29.          }
30.          将编码后的节点特征码存入 block[namePos++];
31.          将属性的文本内容存入 textNames;
32.          将属性的文本内容在 textNames 中的位置存入[namePos++];
33.      }
34.    }
```

- endElement 元素结束事件

表 5.4 为元素结束事件函数的参数和返回值。主要获取元素节点的节点名称。

表 5.4 元素结束事件函数

函数	参数	返回值
endElement	String uri, //命名空间 uri String localname, //元素名 String qname, //限定名	void
功能	与元素的开始标签匹配，回填与元素节点相关的父兄孩子信息。	

下面为元素结束事件函数的具体描述：

```
1. /*保存位置栈出栈元素，将最后栈顶元素设置为他们的父亲*/
2. 声明一个暂存栈 tempStack;
3. int flag = 0;
4. /*回填右兄弟位置且将所有孩子存入暂存栈*/
5. while(判断栈栈顶 == true){
6.       if(flag =0){  //设置第一个出栈元素没有右兄弟
7.            pos = 位置栈栈顶;
8.            设置 pos+2 为-1;  //pos 为该节点节点特征码存储位置
9.            flag ++;
10.      }
11.      位置栈栈顶元素入暂存栈;
12.      位置栈出栈;
13.      判断栈出栈;
14.      if(判断栈栈顶 == false) break;
15.      //设置当前元素的右兄弟位置为前一个出栈元素位置
16.      block[位置栈栈顶+2]设为暂存栈栈顶元素;
17.    }
18.    判断栈栈顶元素=ture;  //设置父节点已经被访问过
19.    if(暂存栈不为空){
20.      修改当前栈顶元素节点特征码，将第 4 位设置为 1 即有孩子;
21.    }
22.    //设置所有暂存栈元素的父亲为当前栈顶元素
23.    while(暂存栈不为空){
24.       //子节点元素的父节点位置设置为当前栈顶元素
25.       block[暂存栈栈顶+1]=位置栈栈顶元素;
26.    }
```

- characters 文本事件

表 5.5 所示为文本事件函数的参数和返回值。需要获取文本节点的文本内容。

表 5.5 文本事件函数

函数	参数	返回值
characters	char[] chars, //文本内容 int start, //起始位置 int length //文本长度	void
功能	处理文本节点，将文本节点的信息写入 DOM 池。	

下面为文本事件函数的具体描述：

```
1. 检测存储空间是否足够
2. 文本 = chars;
3. 去除文本前后的空格以及串内\n\t;
4. if(文本大小!=0){
5.       把文本存入 textNames;
6.       将文本在 textNames 中的位置存入 blocks[namePos];
7.       将位置栈栈顶元素存入 blocks[namePos+1]; //父节点位置
8.       将-1 存入 blocks[namePos+2]; //无右兄弟节点
9.       位置栈入栈(pos);
10.      判断栈入栈(true);
11.      namePos+=3; //保留出父子节点位置等待回填
12.   }
```

- error 错误事件

表 5.6 所示为错误事件函数的参数和返回值。需要获取 sax 解析过程错误对象。

表 5.6 错误事件函数

函数	参数	返回值
characters	SAXParseException e//SAX 解析过程错误对象	void
功能	打印错误。	

至此已经将 SAX 读入的 XML 文档全部按照数据结构保存到 DOM 中，之后我们创建 DOMNode（DOM 节点）类并且通过其提供对外接口。

- XML DOM 节点的实现

章节 3.1.2 介绍了 XML DOM 节点的设计，下面说明这些节点的实现。

在创建 XML DOM 节点时，需要用到已经生成的 XML DOM 中的存储信息。我们创建了一个 DOMNodeCreate 类来封装 XML DOM 节点。在章节 5.2.1 中可以看到，DOM 类中的 CreateNode 方法便是通过创建一个 DOMNodeCreate 的实例来将 XML DOM 的信息传递给 DOMNodeCreate 类，并通过该类中的方法来访问 XML DOM 中的

信息。

在本课题中，负责创建封装 XML DOM 节点的 DOMNodeCreate 类仅需要创建一个实例。因为我们每次只需要一个 DOM 被创建，所以 DOMNodeCreate 类使用单例模式。单例类负责创建自己的对象，同时确保只有单个对象被创建。它的构造函数是私有的，外部调用 getInstance()获取被创建的实例。如果该单例类没有创建过实例则创建第一个实例，如果创建过了则返回该实例。所以在 DOM 类中创建了 DOMNodeCreate 类的实例之后，其他类想要调用 DOMNodeCreate 类中的方法都需要通过 getInstance()获取该实例进行操作。

图 5.2.6 给出了 DOMNodeCreate 类的类图。

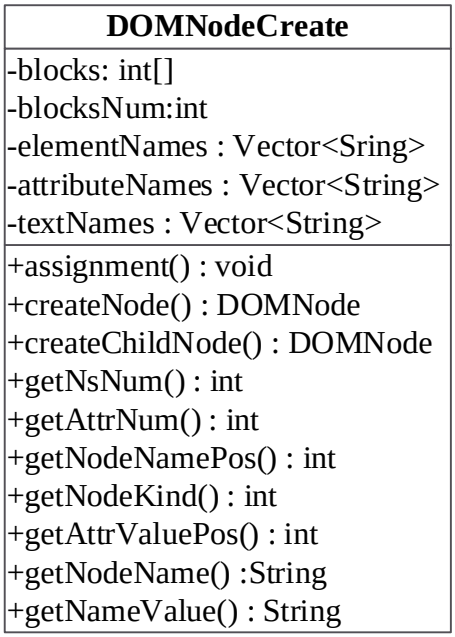


图 5.2.6 DOMNodeCreate 类的类图

类中各个方法的含义和作用如表 5.7 DOMNodeCreate 类中方法所示。

表 5.7 DOMNodeCreate 类中方法

函数名	返回值	含义及作用
assignment	void	将 XML DOM 中信息传入类中
getNsNum	int	获取命名空间个数
getAttrNum	int	获取属性个数
getNodeNamePos	int	获取节点名称位置
getNodeKind	int	获取节点类型
getAttrValuePos	int	获取属性值位置
getNodeName	String	获取节点名称
getNameValue	String	获取节点值

不同的节点类对应不同类型的节点，而 DOMNode 类是这节点类的基类。因此 DOMNode 类实现所有子类节点所需的方法，而不同类型的节点根据它们各自的特

性，有的直接继承 DOMNode 类的方法，有的在自己的类中给出新的实现。图 5.2.7 所示为 DOMNode 类图。

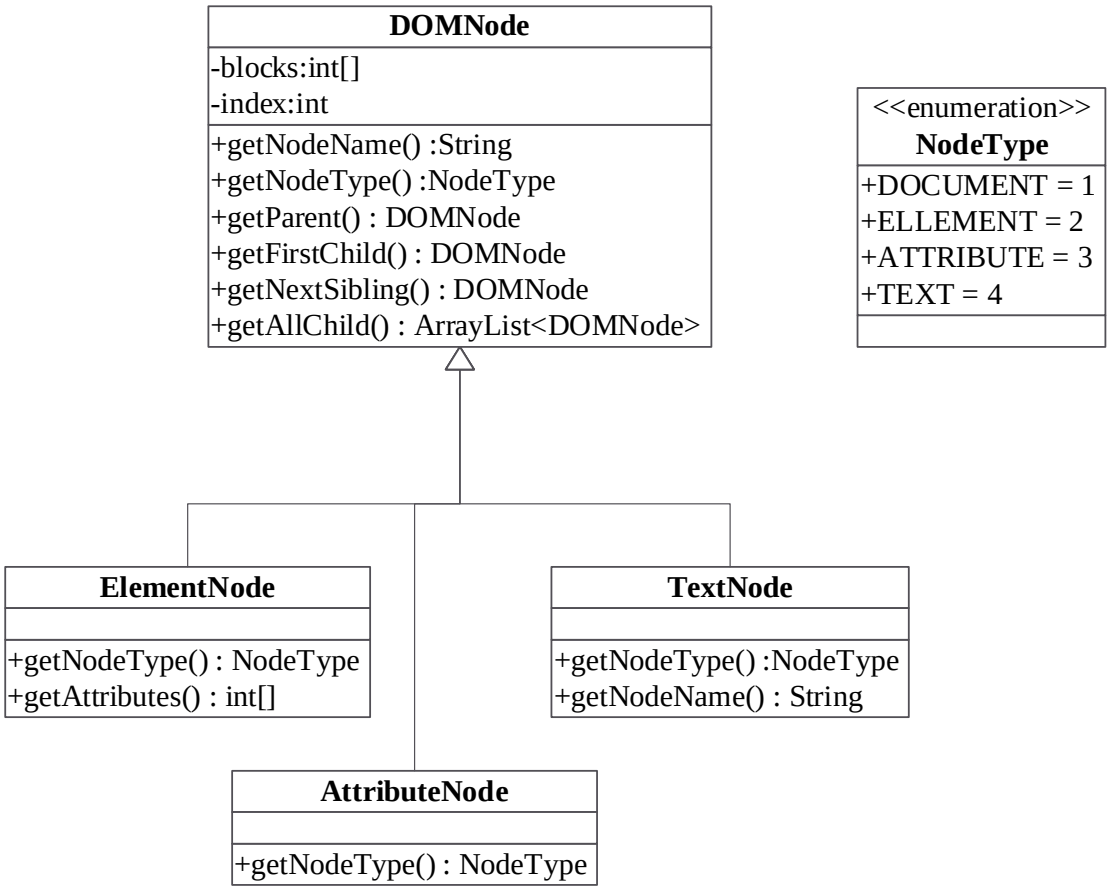


图 5.2.7 DOMNode 类的类图

- 生成标签流

在章节 2.3 中，我们设计了 Label 类，并用 Map<String,List<Label>>来存储标签流。如图 2.3 可知，Map<String,List<Label>>型的变量是 DOM 类的成员变量，它随着 DOM 的一个实例在文档开始事件中被初始化，在元素开始事件和元素结束事件中被存入信息。下面具体说明在这些事件函数中，标签流是如何创建的。

- startDocument 文档开始事件

图 5.2.8 显示了标签流的初始化

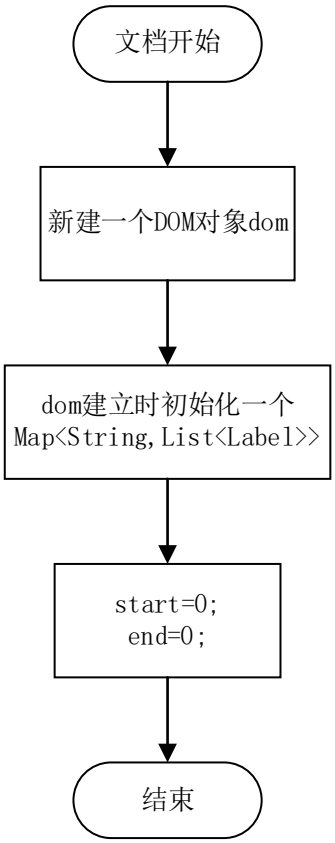


图 5.2.8 文档开始事件中标签流处理流程图

- startElement 元素开始事件

图 5.2.9 显示了遇到开标签时如何处理标签流

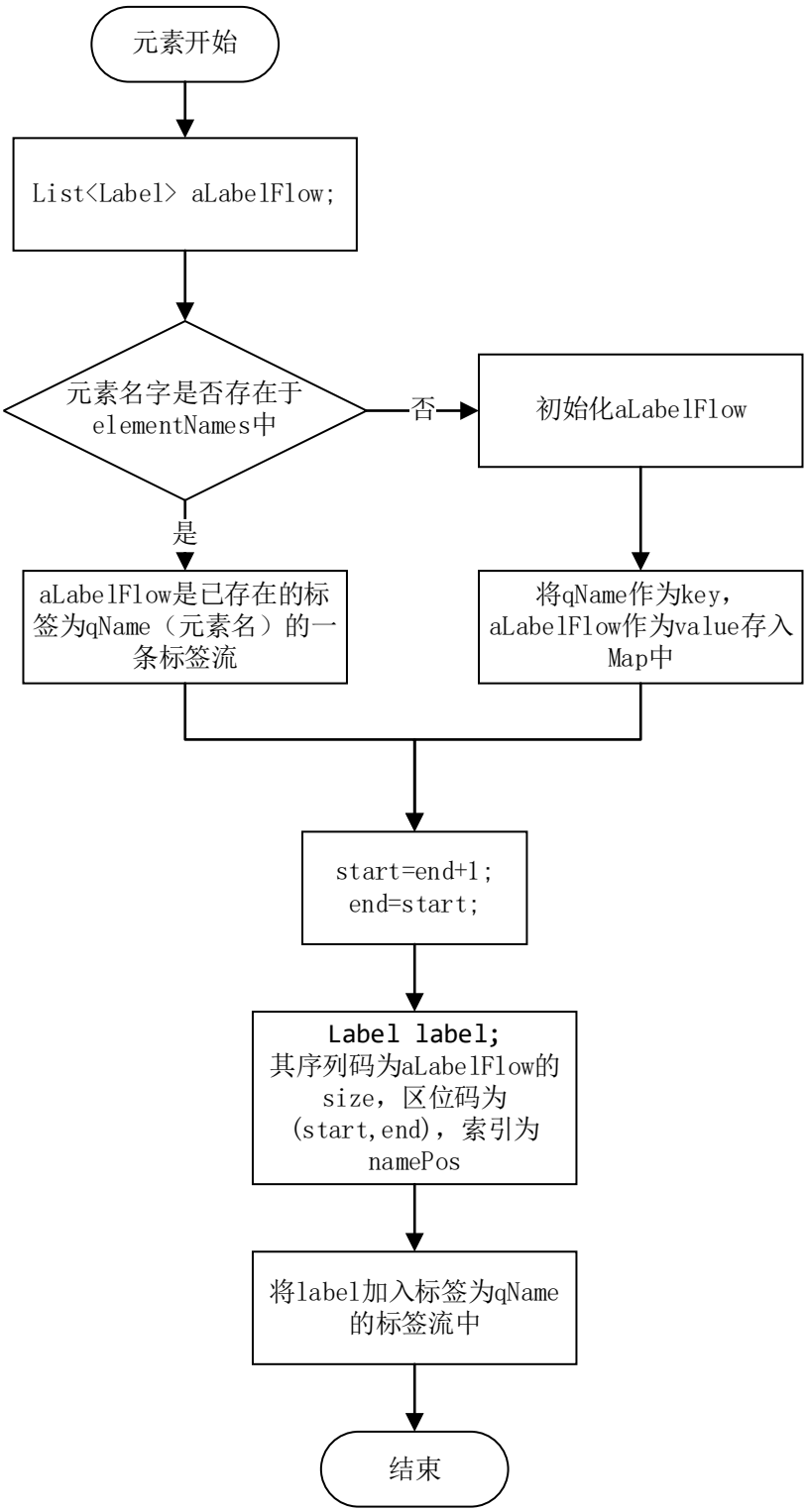


图 5.2.9 元素开始事件中标签流处理流程图

- **endElement 元素结束事件**

图 5.2.10 显示了遇到闭标签时如何处理标签流

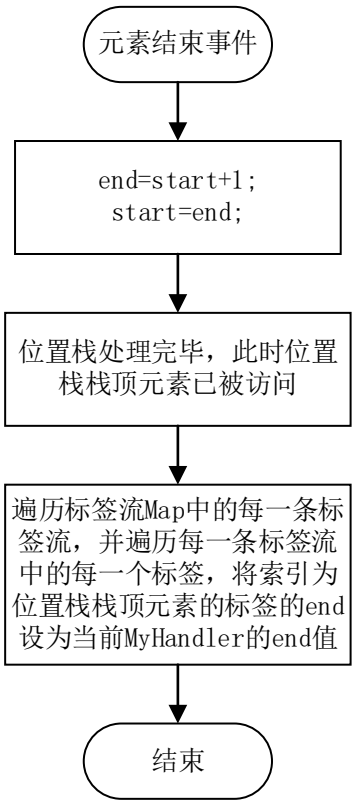


图 5.2.10 元素结束事件中标签流处理流程图

5.3 XPath 语句的读入与分析

在章节 4.14.1.1 中我们说明了 XPath 模块的功能和类设计，接下来我们将说明此模块的具体实现。

XPathTree 类中的各个方法的详细描述如下所示。

- XPath 语句解析函数

函数	**参数**	**返回值**
parse	String XPathString /XPath 语句	void
功能	解析 XPath 语句为查询树	

下面是元素开始事件函数 parse 的流程图。

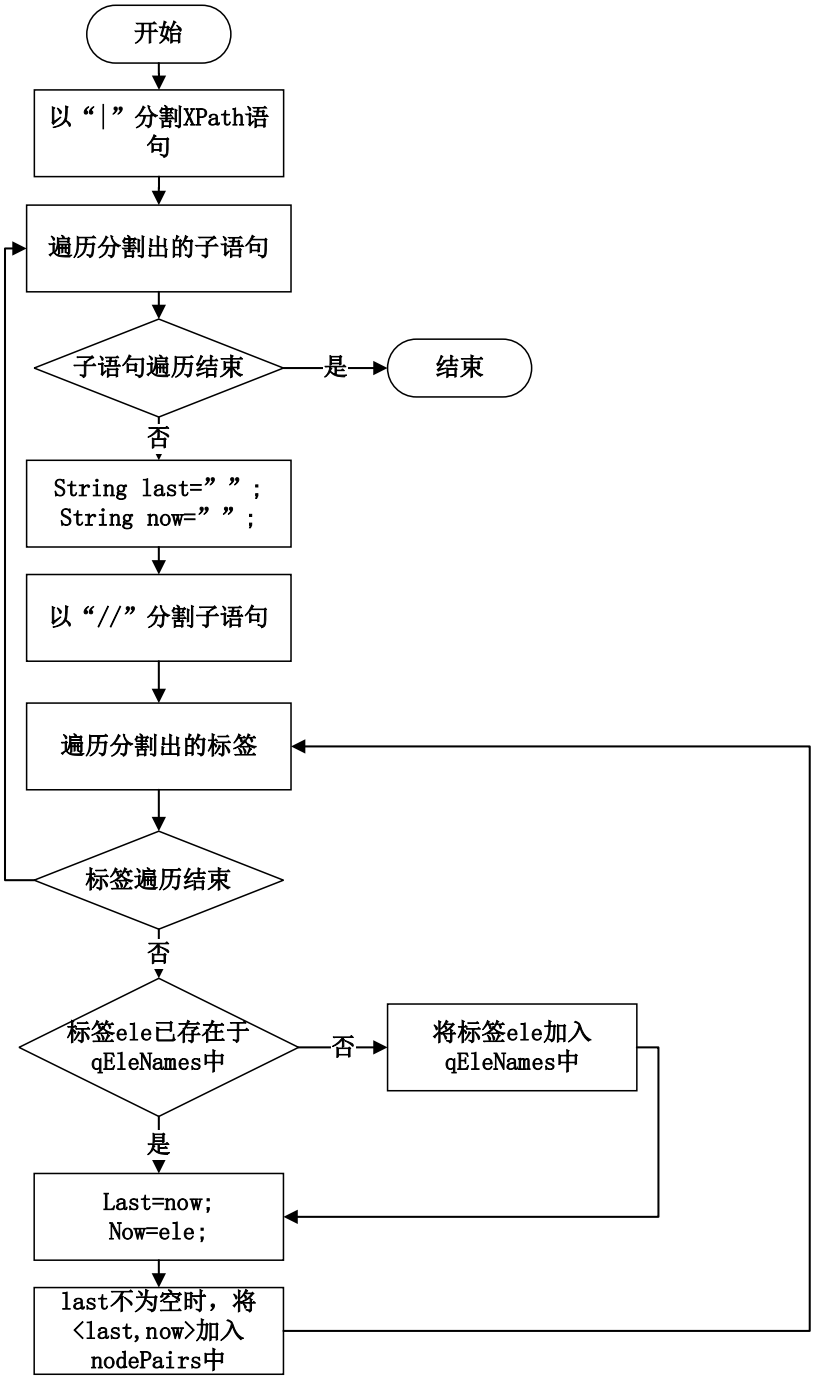


图 5.3.1 parse 的流程图

5.4 Twig 模式匹配的实现

在章节 4.14.1.1 中我们说明了 Twig 模式匹配模块的功能和类设计，接下来我们将说明此模块的具体实现。

TwigList 类中的各个方法的详细描述如下所示。

表 5.8 Twig 模式匹配函数

函数	参数	返回值
match	String XPathString, //XPath 语句 DOMNodeCreate DNC //文档树	void
功能	进行模式匹配	

下面是元素开始事件函数 match 的流程图。

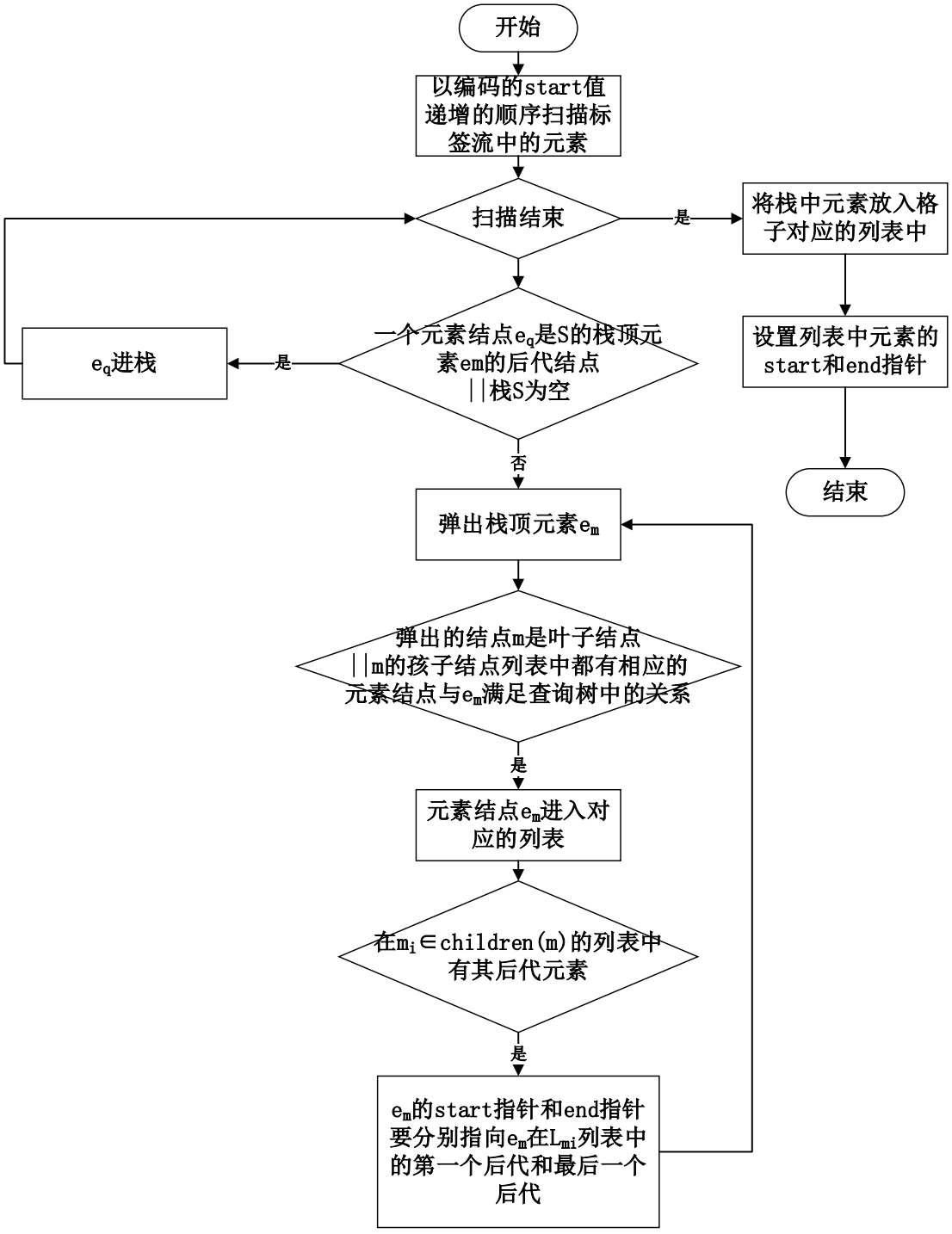


图 5.4.1 函数 match 的流程图

6. 系统测试

6.1 索引和存储模块测试

下面我们将对索引和存储模块进行测试，我们使用以下的文档进行测试，并打印结果显示相应的标签流内容和 DOM 存储内容，如所示。

表 6.1 测试文档

```
<!--This is a bookstore-->
<bookstore>
 <book category="novel">
  <title lang="en">The Island</title>
  <author>Victoria Hislop</author>
  <year>2009</year>
  <price>28.00</price>
</book>
<book category="web">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
 </book>
</bookstore>
```

- 测试 1：打印 DOM 存储内容和标签流

  打印出 DOM 中三个字符串向量存储的内容：

表 6.2 测试 1 输出文本数组存储内容

```
elementNames: [bookstore, book, title, author, year, price]
attributeNames: [category, lang]
textNames: [novel, en, The Island, Victoria Hislop, 2009, 28.00, web, en, Learning XML, Erik T. Ray, 2003, 39.95]
```

打印出标签流：

表 6.3 测试 1 输出标签流

```
[0<1,22>0,  1<2,11>3,  2<3,4>8,  3<5,6>16,  4<7,8>22,  5<9,10>28,  6<12,21>34, 7<13,14>39, 8<15,16>47, 9<17,18>53, 10<19,20>59]
```

比较文档和打印出的内容可知结果正确。

接下来测试 XML DOM 的接口，选取节点为第一个<book>标签。

测试 2：输出该节点所有孩子的标签名和内容

打印出结果如下：

表 6.4 测试 2 输出结果

title
ELEMENT
The Island
author
ELEMENT
Victoria Hislop
year
ELEMENT
2009
price
ELEMENT
28.00

比较文档和打印出的内容可知结果正确。

6.2 XPath 解析模块测试

下面我们将对 XPath 解析模块进行测试，输入一行 XPath 语句，输出所有 NodePair 和所有节点标签。

- **测试 1：简单两个节点的查询树**

输入：

表 6.5 测试 1 输入

//a//b

输出：

表 6.6 测试 1 输出

NodePairs:[<a,b>]
qEleNames[a, b]

- **测试 2：所有节点都最多有一个孩子的查询树**

输入：

表 6.7 测试 2 输入

//a//b//c//d

输出：

表 6.8 测试 2 输出

NodePairs:[<a,b>, <b,c>, <c,d>]
qEleNames[a, b, c, d]

- **测试 3：一个根节点有一个以上孩子的查询树**

输入：

表 6.9 测试 3 输入

//a//b|//a//d//f

输出：

表 6.10 测试 3 输出

NodePairs:[<a,b>, <a,d>, <d,f>] qEleNames[a, b, d, f]

测试结果皆与预期相符。

6.3 系统整体测试

下面我们将对整个系统进行测试，输入 XML 文档和一行 XPath 语句，输出生成的节点列表和列表中每个元素的指针。

我们使用以下和图 2.7.1(a)所对应的文档进行测试，不同的测试改变 XPath 语句，查看输出结果。

表 6.11 测试文档

```
<r>
    <a>
        <b>11</b>
        <d>
            <f>1</f>
        </d>
        <c>
            <f>2</f>
        </c>
        <d>3</d>
    </a>
    <a>
        <c>
            <f>10</f>
        </c>
    </a>
    <a>
        <b>5</b>
        <c>6</c>
        <c>7</c>
        <b>8</b>
        <d>
            <f>9</f>
        </d>
    </a>
</r>
```

- **测试 1：简单两个节点的查询树**

  输入 XPath 语句为：

表 6.12 测试 1 输入

//a//b

对比文档树，我们期望的匹配结果应如图 6.3.1 所示。

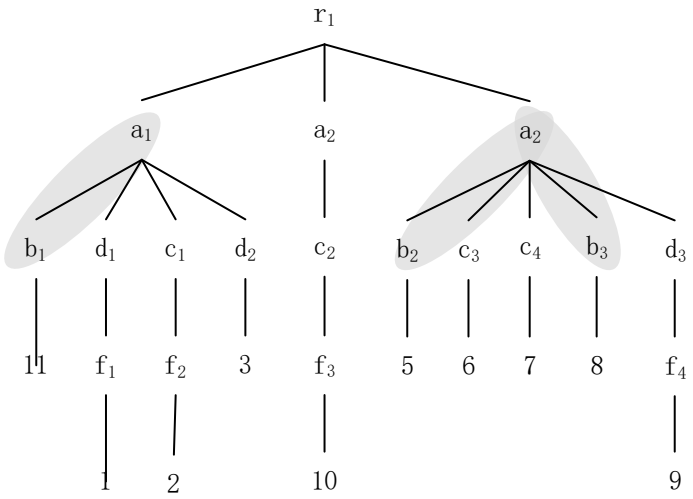


图 6.3.1 测试 1 期望结果

输出：

表 6.13 测试 1 输出

a的列表: 第0个元素a(3): b start:6,end:6 第1个元素a(48): b start:51,end:69 b的列表: 第0个元素b(6): 第1个元素b(51): 第 2 个元素 b(69):

其意为标签 A 的列表 L_A 中有两个元素，a(3)和 a(48)的意思是这两个元素在索引树中位置分别为 3 和 48，下面皆为此简写，不再解释。标签 B 的列表 L_B 中有三个元素，是 b(6)，b(51)，b(69)。另外，a(3)对列表 L_B 的区间指针为<6,6>，a(48) 对列表 L_B 的区间指针为<51,69>，所以匹配出三个结果：[a(3),b(6)]，[a(48),b(51)]，[a(48),b(69)]。

测试结果与预期相符。

- **测试 2：所有节点都最多有一个孩子的查询树**

输入 XPath 语句为：

表 6.14 测试 2 输入

//a//c//f

对比文档树，我们期望的匹配结果应如图 6.3.2 所示。

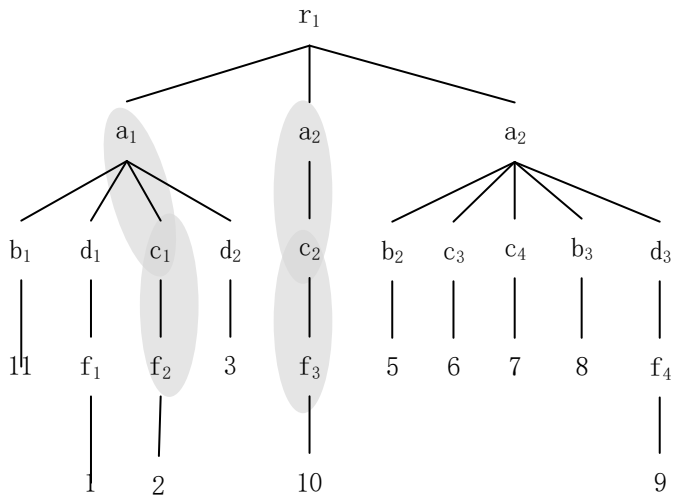


图 6.3.2 测试 2 期望结果

输出：

表 6.15 测试 2 输出

a的列表:
第0个元素a(3):
c start:21,end:21
第1个元素a(36):
c start:39,end:39
第2个元素a(48):
c的列表:
第0个元素c(21):
f start:24,end:24
第1个元素c(39):
f start:42,end:42
f的列表:
第0个元素f(15):
第1个元素f(24):
第2个元素f(42):
第 3 个元素 f(78):

其意为标签 A 的列表 L_A 中有 3 个元素，a(3)，a(36)，a(48)。标签 C 的列表 L_C 中有 2 个元素，是 c(21)，c(39)。标签 F 的列表 L_F 中有 4 个元素，是 f(15)，f(24)， f(42) ，f(78)。另外，a(3)对列表 L_C 的区间指针为<21,21>，a(36) 对列表 L_C 的区间指针为<39,39>。注意，虽然 a(48)在列表中，但它没有对其它元素的指针，所以它并不属于匹配结果。c(21)对列表 L_F 的区间指针为<24,24>，a(39) 对列表 L_F 的区间指针为<42,42>。所以匹配出 2 个结果：[a(3),c(21),f(24)]，[a(36),c(39), f(42)]。

测试结果与预期相符。

- **测试 3：一个根节点有一个以上孩子的查询树**

输入 XPath 语句为：

表 6.16 测试 3 输入

//a//b|//a//f

对比文档树，我们期望的匹配结果应如图 6.3.3 所示。

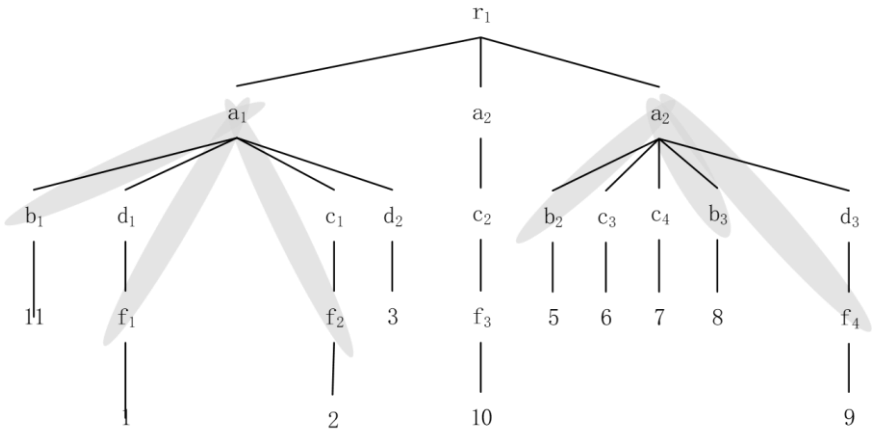


图 6.3.3 测试 3 期望结果

输出：

表 6.17 测试 3 输出

a的列表: 第0个元素a(3): b start:6,end:6 f start:15,end:24 第1个元素a(48): b start:51,end:69 f start:78,end:78 b的列表: 第0个元素b(6): 第1个元素b(51): 第2个元素b(69): f的列表: 第0个元素f(15): 第1个元素f(24): 第2个元素f(42): 第 3 个元素 f(78):

其意为标签 A 的列表 L_A 中有 2 个元素，a(3)，a(48)。标签 B 的列表 L_B 中有三个元素，是 b(6)，b(51)，b(69)。标签 F 的列表 L_F 中有 4 个元素，是 f(15)，f(24)， f(42) ，f(78)。另外，a(3)对列表 L_B 的区间指针为<6,6>，a(3)对列表 L_F 的区间指针为<15,24>，a(48) 对列表 L_B 的区间指针为<51,69>，a(48)对列表

L_F的区间指针为<78,78>。所以匹配出 4 个结果:[a(3),b(6), f(15)],[a(3),b(6), f(24)] , [a(48),b(51), f(78)] , [a(48),b(69), f(78)]。
测试结果皆与预期相符。

6.4 效率测试

下面我们对索引和存储模块进行效率测试，从以下几个方面进行比较。

- 文件大小对效率的影响

  下面分别测试 10kb，100kb，1000kb 的 xml 文件 10 次运行的平均时间：

表 6.18 不同大小文件运行时间表

文件大小	10kb	100kb	1000kb
运行时间	142ms	1434ms	94217ms

- 节点的复杂度对效率的影响

  将大小均为 100kb，节点内包含多个子结点，多个子结点又包含不同节点（分三层）与节点内包含多个子结点，多个子结点又包含不同节点（分六层）的 xml 文档进行比较，分别记录 5 次运行的时间：

  - 分三层

表 6.19 分三层文件运行时间表

运行次数	运行时间
1	753ms
2	767ms
3	825ms
4	790ms
5	783ms
平均	784ms

  - 分六层

表 6.20 分六层文件运行时间表

运行次数	运行时间
1	1019ms
2	1120ms
3	1134ms
4	1118ms
5	1082ms
平均	1095ms

■ 分九层

表 6.21 分九层文件运行时间表

运行次数	运行时间
1	2085ms
2	2079ms
3	2054ms
4	2114ms
5	2088ms
平均	2084ms

将大小均为 100kb，分三层，分六层，分九层的 xml 文档进行比较，分别记录 5 次运行的平均时间：

表 6.22 不同节点复杂度文件运行时间表

节点复杂度	分三层	分六层	分九层
运行时间	784ms	1095ms	2084ms

比较可得，在文件大小相同的情况下，系统效率随节点复杂度的升高而变低。

结论

本文实现了一个对 XML 文档数据流节点存储并建立索引，并支持 Twig 模式匹配的系统。

我们的整个系统由 XML 的存储和索引模块，XPath 解析模块以及 Twig 模式匹配模块三部分组成。其中我们为 XML 的索引和存储设计了 XML DOM，它包括一个用数组存储的索引树，和三个存储文本的数组。另外，存储和索引模块还会生成一个包含所有标签的标签流，其中每个标签流节点存储了能确定此节点与其他节点关系的区位码，以及它在 DOM 文本数组中位置的索引。同时，我们为 XPath 的解析设计了一个解析器，以便我们将 XPath 语句转化为一棵查询树。最后，我们实现了一个 Twig 模式匹配模块，它采用 TwigList 算法，将存储和索引模块生成的文档树，和 XPath 解析模块生成的查询树进行匹配，并最终得到符合查询语句的结果。

虽然我们实现了模式匹配的系统，但是还有一些地方值得我们研究与改进：第一，DOM 接口的实现不完全。本文中的 DOM 只对外提供了一些有关于查询的接口，而插入删除修改 DOM 的方法并没有实现。第二，此系统只能解析简单的 XPath 语句并进行匹配，若能引入已有的语法分析器，也许可以增加能解析的 XPath 语法。

参考文献

[1] Jagadish H, Lakshmanan L, Srivastava D. TAX: A Tree Algebra for XML: Lecture Notes in Computer Science[Z]. Ghelli G, Grahne G. Springer Berlin/ Heidelberg, 2002: 2397, 149-164.

[2] Wood L, Le Hors A, Apparao V, et al. Document object model (dom) level 1 specification[J]. W3C recommendation, 1998, 1.

[3] 王芳, 李正凡.用 SAX 解析 XML 文档的实现方法[J]. 华东交通大学学报, 2004(01):87-89.

[4] 徐超,张东站.sTwig——一种基于流的 XML 小枝匹配算法[J].计算机研究与发展,2010,47(z1):86-92.

[5] Qin Lu, Yu J X, Ding Bolin. TwigList: Make twig pattern matching fast[C].//Proc of DASFAA’07. Thailand: Springer, 2007:850-862.

[6] 朱新向.基于小枝模式匹配的 XML 数据查询处理算法研究[J]. 中国石油大学 (华东), 2012: 1-2.

[7] 高万辰.基于部分求值的 Twig 查询优化技术[D]. 北京工业大学, 2015.

[8] Mathis C. Storing, indexing, and querying XML documents in native XML database management systems[M]. Verlag Dr. Hut, 2009.

[9] CS• 霍斯特曼 (美), Gray• 柯内尔 (美), 程峰, 等. Java 2 核心技术: 基础知识. 卷 I[M]. 机械工业出版社, 2003.

[10] 刘立新, 王永平.基于有序对的不确定 XML 小枝模式查询算法[J]. 计算机与数字工程, 2017(3).

[11] 张青平.XML 查询中 Twig 模式匹配算法的研究[D]. 南京航空航天大学.

[12] Harold E R . Processing Xml with Java[M]. 科学出版社, 2002.

致　谢

本次毕业设计和本论文能顺利完成需要感谢我的导师苏航老师。从毕业设计的正式开始之前，一直到本论文的完成，苏老师提供了我非常多的帮助与激励。在本次设计开始之初，我对 XML 及其相关概念都不甚了解，苏老师写了好几大张纸为我讲解，从基础的概念，到整体的系统设计，都有苏老师为我理清思路，提供帮助。每次我将进度报告发给老师，老师都会发来长长的回复，为我解答疑惑。苏老师也指导了本篇论文的撰写，对于苏老师细致耐心的指导和严谨的治学态度，我表示深深的尊重和敬佩。

其次，我还要感谢刘倚天和张雨学姐。她们的前期研究给本课题提供了很多支持。

最后，感谢我的父母，同学和朋友们，感谢他们在精神上的支持与鼓励，以及在生活上的陪伴与照顾，是他们支持我完成了大学四年的课程和这次的毕业设计，非常感谢！