# 摘要

随着大数据时代的到来，海量数据的处理得到了越来越多的人们的重视。XML作为网络数据存储和交换的主要格式，XML数据流的处理也逐渐受到关注。

在XML数据流处理时，由于处理节点的流量限制，很可能出现处理的效率与到来数据流数据量不匹配的矛盾。因此，本课题设计并实现一种在数据流环境下具有缓冲机制并支持高效访问的流式XML DOM。

本课题用大型XML文档模拟XML数据流环境，默认所有XML数据流信息均在XML文档的根节点下扩展。通过使用Xerces-C++中的SAX解析器部分将XML数据流转化为内存中顺序存储的缓冲式XML DOM。该缓冲式XML DOM内部维持一个具有多个子DOM的DOM池，同时每一个子DOM拥有专属自己的读写锁。在子DOM内部存储的是XML数据流各个标签所对应XML DOM节点的索引。根据这些索引就可以提取相关信息封装成XML DOM节点。同时，采用Xpath可以遍历XML DOM节点进行查询。不过，由于XML数据流信息量巨大，因此只有在Xpath进行查询时才将需要的XML DOM节点封装起来进行处理。

本课题旨在设计和实现数据流环境下的缓冲式XML DOM，通过Xpath查询来验证实现的正确性。但由于缓冲式XML DOM对于XML数据流的处理是持续不断的，而Xpath查询对于缓冲式XML DOM的访问也是随时进行的。因此本课题通过对于缓冲式XML DOM内的每一个子DOM设置读写锁来控制XML数据流存储和Xpath查询的同步。

关键词:流数据；XML DOM；Xpath

# ABSTRUCT

With the arrival of the era of big data, more and more people pay attention to the processing of massive data. XML is the main format of network data storage and exchange, and the processing of XML data stream is gradually being concerned.

Due to the node's traffic constraints in XML data stream processing, it is possible to deal with the contradiction between the efficiency of the data flow and the amount of data flow does not match. Therefore, this paper designs and realizes a kind of streaming XML DOM which has the buffer mechanism and supports the efficient access in the data stream environment.

This topic uses a large XML document simulating XML data stream environment, all XML data flow information are the default extended under the root node of the XML document. By using the SAX parser in Xerces-C++, the XML data stream is converted to a buffered XML DOM that is sequentially stored in the memory.

The XML DOM maintains a DOM pool with a plurality of sub DOM, and each sub DOM has an exclusive read-write lock. The internal storage of the sub DOM is the index of the XML DOM node corresponding to each label of the XML data stream. According to these indexes, the relevant information can be extracted and encapsulated into XML DOM nodes. At the same time, the Xpath can be used to traverse the XML DOM node for query. However, due to the huge amount of XML data flow information, so only when the Xpath query needed to be packaged XML DOM nodes to deal with.

The purpose of this paper is to design and implement a buffer XML DOM in the data stream environment, and verify the correctness of the implementation by using Xpath query. However, due to the processing of XML data stream to XML DOM is continuous, and the Xpath query for XML DOM access is also at any time. Therefore through setting each sub DOM within XML DOM to have a read-write lock to control the synchronization of XML data stream storage and Xpath query.

目录

[摘要 1](#_Toc450930588)

[ABSTRUCT 2](#_Toc450930589)

[1.绪论 5](#_Toc450930590)

[1.1课题背景 5](#_Toc450930591)

[1.2任务以及主要工作 5](#_Toc450930592)

[1.3目的及意义 5](#_Toc450930593)

[1.4 本文结构 6](#_Toc450930594)

[2.相关技术介绍 6](#_Toc450930595)

[1.1 XML文档对象模型 6](#_Toc450930596)

[1.1.1 XML定义 6](#_Toc450930597)

[1.1.2 XML DOM定义 7](#_Toc450930598)

[1.1.3 XML DOM节点 7](#_Toc450930599)

[2.1.4 XML DOM节点树 7](#_Toc450930600)

[2.2 SAX解析器 9](#_Toc450930601)

[2.2.1 SAX的定义 9](#_Toc450930602)

[2.2.2 SAX2的定义 9](#_Toc450930603)

[2.2.3 SAX2的处理机制 9](#_Toc450930604)

[2.2.4 Xerces-C++ 11](#_Toc450930605)

[2.3数据流 12](#_Toc450930606)

[2.4 Xpath 13](#_Toc450930607)

[2.4.1 Xpath简介 13](#_Toc450930608)

[2.4.2 Xpath节点 13](#_Toc450930609)

[2.4.3语法 14](#_Toc450930610)

[3.缓冲式XML DOM与Xpath查询的设计 16](#_Toc450930611)

[3.1缓冲式XML DOM的设计 16](#_Toc450930612)

[3.1.1缓冲式XML DOM数据结构的设计 16](#_Toc450930613)

[3.1.2缓冲式XML DOM节点的设计 22](#_Toc450930614)

[3.2 Xpath的设计 23](#_Toc450930615)

[3.2.1 Xpath模块功能 23](#_Toc450930616)

[3.2.2 Xpath模块接口设计 23](#_Toc450930617)

[3.3系统同步机制设计 24](#_Toc450930618)

[4.系统实现 25](#_Toc450930619)

[4.1系统结构 25](#_Toc450930620)

[4.2缓冲式XML DOM的实现 27](#_Toc450930621)

[4.2.1 缓冲式XML DOM生成的实现 27](#_Toc450930622)

[4.2.2 缓冲式XML DOM节点的实现 33](#_Toc450930623)

[4.3 Xpath查询的实现 36](#_Toc450930624)

[4.3.1 Xpath的处理 36](#_Toc450930625)

[4.3.2 Xpath的查询 36](#_Toc450930626)

[5.测试实验 37](#_Toc450930627)

[5.1缓冲式XML DOM生成部分性能测试 37](#_Toc450930628)

[5.2系统功能测试 38](#_Toc450930629)

[结论 42](#_Toc450930630)

[参考文献 43](#_Toc450930631)

[致谢 44](#_Toc450930632)

# 绪论

## 课题背景

在大数据时代，来自实时监控、实时交易和社交网络等各种实时系统的、不断增长的海量数据处理需求不断增加，使得数据无法完整的读到内存中进行处理，因此流式计算(Stream Computing)的研究得到重视。XML作为World Wide Web Consortium（W3C）推荐标准，大量数据以XML格式数据流的形式在网络应用中存储和传输。

在使用XML数据时，需要获取文档中的数据内容，一种常用的方式就是将XML文档解析为内存中的XML DOM (Document Object Model)，并提供相关接口以支持数据查询使用。而主要的解析方式有两种：DOM解析和SAX解析。DOM解析是将所有的XML文件读取到内存中形成DOM树，再进行访问，这种方式不适合文件量过大时使用。而SAX解析则是顺序读取文件内容，进行处理，不受文件大小的控制。因此，使用SAX解析数据流环境下的XML文档较为合理。在XML数据流处理时，特别是分布式数据流处理过程中，每个节点分配的数据流的流量很多情况下并不受节点本身控制，并且节点处理能力也受一定限制，这就存在数据流处理与到来数据流数据量不匹配的矛盾。因此，需要设计一种在数据流环境下具有缓冲机制并支持高效访问的流式XML DOM，以支持对XML数据流进行高效的查询处理，保证在数据流流量超过节点处理能力的时候进行缓冲并形成DOM以支持节点的高效查询。

## 任务以及主要工作

基于顺序存储设计和实现一种缓冲式XML DOM，该设计可以实现XML数据流的解析和存储，并且随着数据流变化而进行更新。该缓冲式XML DOM遵循W3C标准。同时支持Xpath语言在数据流存储的同时查询流式XML DOM内的实时数据。

利用Apache组织所推动的一项XML文档解析开源项目Xerces进行XML数据流的解析；设计基于顺序表存储的缓冲DOM池以存储大量的流式数据；设计外部访问DOM内实时数据的接口；设计Xpath语言查询接口。

## 目的及意义

本课题的目的在于正确的存储流式XML数据，形成缓冲式XML DOM；同时，由于数据流是实时的、连续的、有序项的序列，因此数据流上的查询在周期性的时间中连续运行，当新数据到达之后，Xpath查询能够增量的返回结果。

现如今大数据的时代已经到来，人们对于海量数据的挖掘和运用已势不可挡。XML作为万维网传输媒介蕴含大量的有价值信息，因此本课题研究的在流数据环境下应用的缓冲XML DOM可以适应时代背景，同时也为后续的相关研究做下铺垫。此外，本课题基于顺序表存储形成DOM，与多数人采用的链式结构存储相比，顺序表存储方式极大地节省了内存空间，适合对XML数据流进行存储。

## 本文结构

本论文的结构如下：

* 第2章相关知识介绍中主要说明的是本课题中所有涉及到的技术名称、研究内容、以及相关术语的解释；
* 第3章缓冲式XML DOM与Xpath语言查询的设计中说明了对于缓冲式DOM的结构分析和整体设计，以及Xpath查询的设计思路；
* 第4章系统实现中主要说明的是缓冲式XML DOM系统的协作过程和实现方法；
* 第5章实验及测试中主要包括程序系统测试结果和实验分析。
* 最后是总结和致谢部分。

# 相关技术介绍

## XML相关知识介绍

### XML定义

可扩展标记语言（eXtensible Markup Language），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，简称XML。

可扩展标记语言具有以下特点：

* 其被设计用来传输和存储数据
* 其标签需自行定义
* 具有自我描述性
* 是W3C的推荐标准

XML 是各种应用程序之间进行数据传输的最常用的工具。因此，XML文档在日常工作和生活中非常普遍。XML文档是一种半结构化的、以数据为中心的、树形结构的文档。简而言之，它就是一种带有标签的文档。

表2-1 XML文档实例

表 2‑1

|  |  |
| --- | --- |
|  |  |
|  |  |

表 2‑2 XML文档实例

|  |
| --- |
| <?xml version="1.0" encoding="ISO-8859-1"?>  <!--This is a note-->  <note time=”Monday”>  <to>George</to>  <from>John</from>  <heading>Reminder</heading>  <body>Don't forget the meeting!</body>  </note> |

如表 2‑2所示，这是一个简单的XML文档实例。第一行是 XML文档声明。标签<note>内部嵌套四个标签，分别为<to><from><heading><body>。从表中可以明显看出此XML文档包含了 John 给 George 的一张便签，内容为“Don't forget the meeting!”。这就充分体现了XML出色的自我描述性。

### XML DOM定义

XML 文档对象模型（XML Document Object Model），即XML DOM。

XML DOM 定义了所有 XML 元素的对象和属性，以及访问它们的方法（接口）。简而言之，XML DOM定义了访问和操作XML文档的标准方法。

### 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 这样规定：

* 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

如上面表2-1所示的XML文档实例中所看到的，整个XML文档是一个文档节点，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 note-->”便是一个注释节点。之后的标签<note>以及其中包含的<to><from><heading><body>这些标签都是元素节点。包含在元素节点中的文本，例如<heading>标签中包含的文本Reminder构成了一个文本节点。此外，标签<note time=”Monday”>中的“time”构成属性节点，一个标签中可以以存在多个属性节点。

### XML DOM节点树

XML DOM 把 XML 文档视为一种树结构。这种树结构被称为节点树。这棵节点树可以展示出XML DOM节点的集合，以及它们之间的联系。



图 2‑1XML文档以及其对应的树形结构XML DOM

图2-1 XML文档以及其对应的树形结构XML DOM

如图 2‑1所示，图2-1(b)为图2-1(a)中的XML文档对应的树形XML DOM。XML DOM的树形结构源于XML文档本身的树形原型。从图2-1(b)中可以看出，XML DOM节点树从根节点开始，向下延伸至文本节点结束。当元素节点中包含文本节点时，文本节点一定是叶子节点。

此外，节点树中的节点彼此之间都有等级关系。这种等级关系包括父子关系和同级关系。如图2-1(b)中所展示的，节点<bookstore>和节点<book>之间存在父子关系。节点< bookstore >为父节点，而节点<book>则为子节点。位于相同层级上的子节点间存在同级关系，如图中的<title><author><year>和<price>节点，它们都互为同级节点，彼此之间存在同级关系。不过，值得注意的是：图中最低层的四个文本节点，它们之间是不存在同级关系的。同级节点是拥有相同父节点的节点，而最低层的四个文本节点不属于同一个父亲。

特别地，图2-1(b)中的属性节点“lang”和“category”显得有些特殊。但是仔细观察不难发现属性节点位于元素节点标签的内部，如图2-1(a)中的<book category="children">，因此，属性节点不作为元素节点的子节点处理，只是将它理解为元素节点的内部节点。

## SAX解析器

### SAX的定义

SAX，全称Simple API for XML。SAX既是一种接口，也是一种软件包。它是一种XML解析的替代方法。SAX进行解析时采用逐行扫描文档的方法，一边扫描一边解析。由于应用程序只是在读取数据时检查数据，因此不需要将数据存储在内存中。这就使得SAX解析更加适用于大型文档的解析工作。

SAX的工作原理简单地说就是对文档进行顺序扫描，当扫描到文档开始、元素开始与结束等地方时通知事件处理函数，由事件处理函数做出相应动作，然后继续同样的扫描，直至文档结束。

### SAX2的定义

用于XML的简单应用程序编程接口(Simple API for XML,version 2)，简称SAX2。

SAX2主要区别于SAX的其他版本在于：

* 支持命名空间（Namespace），这个特性是很多XML的高级功能所必须的。
* 提出了Filter（过滤器）的概念，在某些情况下，需要对流中运行的信息进行修改，例如剔除某些不需要的信息等，这时在流中的信息到达Handler之前进行过滤就成为理所当然的选择。

### SAX2的处理机制

SAX2处理XML文档时允许在读取文档的时候，立即对文档进行处理，解析完毕也就处理完成了，不必等到整个文档被分析储存之后才进行操作，因此非常适合处理较大的XML文档。利用 SAX 2解析 XML 文档，涉及两个部分：解析器和事件处理器。解析器负责读取 XML 文档，并向事件处理器发送事件；事件处理器负责对事件做出相应动作，对传递的 XML 数据进行处理。

SAX2解析器功能主要通过XML Reader接口实现。XML Reader 接口是 SAX2解析器必须实现的接口。该接口允许应用程序设置和查询解析器的功能和特性，注册处理文档的事件处理器，以及启动文档的解析。

SAX2事件处理器功能主要通过ContentHander 接口实现。

SAX2处理过程中有一些常用的接口，如图2-2所示。由于本课题中并没有涉及到所有的接口的使用，因此对于未使用到的接口只进行一些简单的介绍，不过多赘述。

* ContentHandler接口——内容处理

该接口封装了一些对事件处理的方法，当XML解析器开始解析XML输入文档时，它会遇到某些特殊的事件，比如文档的开始和结束、元素的开始和结束等事件。当遇到这些事件时，XML解析器会调用ContentHandler接口中的相应方法来响应该事件。

* DTDHandler接口——文档类型定义处理

用于接收基本的DTD相关事件的通知。此接口仅包括DTD事件的注释和未解析的实体声明部分。

* EntityResolver接口——

用于解析实体的基本接口。解析器将在打开任何外部实体前调用此方法。如果SAX应用程序需要实现自定义处理外部实体，则必须实现此接口。

* ErrorHandler接口——错误处理

是SAX错误处理程序的基本接口。如果SAX应用程序需要实现自定义的错误处理，则它必须实现此接口，然后解析器将通过此接口报告所有的错误和警告。

以上接口都通过DefaultHandler类实现。该类是SAX2事件处理程序的默认基类。

图2-2 SAX2的事件处理器接口

对于SAX2的解析器和处理器进行了简单的介绍。下面对于SAX2的事件处理机制进行简要说明。



图2-3 SAX2处理XML过程

如图2-3(a)所示，SAX2解析器通过SAX2XMLReader读取XML文件，调用SAX2XMLReader中的setContentHandler()方法创建一个ContentHandler()实例，同时DefaultHandler()类实现ContentHandler()接口，触发与文档相关的基本事件，调用ContentHandler()内部的相应方法，处理完成后返回，继续对XML文档进行扫描。直至文档结束，触发endDocument()。扫描到XML文件的各个标签所对应触发的基本事件如2-3(b)所示。

### Xerces-C++

现在存在许多的开源XML文档解析器，他们各有特点，经过比较权衡，本课题决定采用Xerces-C++来进行XML文档的解析。

Xerces是由Apache组织所推动的一项XML文档解析开源项目,它目前有多种语言版本包括JAVA、C++、PERL、COM等。在本课题的中采用的是c++版本的Xerces，即Xerces-C++。

Xerces-C++是用可移植的C++子集编写的XML分析器。允许对XML数据进行读写操作，能记录最近的DOM和SAX规范，可编程产生和验证XML。

Xerces-C++同时提供DOM解析和SAX解析两种解析方法，章节2.2.1节中也介绍了关于SAX解析的特点，在本课题中利用Xerces-C++中提供的SAX2解析部分来将XML文档解析存储并生成XML DOM。

由于Xerces-C++程序不依赖于所使用的API，所以不管是进行何种解析都需要初始化Xerces系统，解析结束时也需要终止系统。在章节2.2.3节中介绍了SAX2处理机制中所必须的SAX2XMLReader()以及DefaultHandler()，在这里需要分别对它们进行实例化，解析过程则是通过ContentHandler接口去触发各个事件函数的过程。流程如图2-4所示。



图2-4 Xerces-C++系统SAX2解析XML文档流程图

## 数据流

数据流的概念最初在1998年由Henzinger在文献87中提出，他将数据流定义为“只能以事先规定好的顺序被读取一次的数据的一个序列”。

数据流主要有以下几个特征：

* + 数据的到达—快速

短时间内处理大量数据。

* 数据的范围—广域

数据属性的取值范围大。

* 数据到达的时间—持续

意味着数据量可能是无限大的。

## Xpath

### Xpath简介

XPath即为XML路径语言(XML Path Language)，它被用来在 XML 文档中查找信息。

Path有以下特点：

* + XPath 是一个 W3C 标准；
* XPath 包含一个标准函数库；
* XPath 使用路径表达式在 XML文档中进行导航。

### Xpath节点

1 - <!-- W3School.com.cn bookstore example -->

2 - <bookstore>

3 - <book category="children">

4 - <title lang="en">Harry Potter</title>

5 - <year>2005</year>

6 - </book>

7 - <book category="cooking">

8 - <title lang="cn">Everyday Italian</title>

9 - <book>2012</book>

10-</book>

11-</bookstore>

图2-5 XML文档

Xpath是XML的路径查询语言，因此Xpath节点同XML以及XML DOM节点一样，存在7种节点：文档、元素、属性、文本、命名空间、处理指令以及注释节点。命名空间以及处理指令节点在本课题中不涉及，其他类型节点均在XML DOM节点部分进行了解释和说明，这里不重复介绍。

下面重点说明Xpath查询语言中的节点关系。在Xpath节点中主要存在以下几种关系：

* + 父(Parent)

上一级节点。每个元素以及属性都有一个父。如图2-4所示，bookstore元素是第3行与第7行book元素的父。

* 子(Children)

下一级节点。元素节点可有零个、一个或多个子。如图2-4所示，第3行和第7行的book元素都是bookstore元素的子。

* 同胞(Sibling)

拥有相同的父的节点。如图2-4所示，第3行与第7行的两个book元素是同胞。

* + 先辈(Ancestor)

某节点的父、父的父，等等。如图2-4所示，year元素的先辈是首个book 元素和 bookstore 元素。

* + 后代(Descendant)

某个节点的子，子的子，等等。如图2-4所示，bookstore 的后代是所有的book、title和year元素。

### XPath语法

XPath是使用路径表达式在 XML 文档中选取节点的。如表2-2所示，表中列出了Xpath最常用的表达式以及其描述。

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，不考虑位置。 |
| @ | 选取属性。 |

了解了Xpath查询语言的一些基本表示式，表述得相对抽象，下面通过实例来说明Xpath表达式的用法和查询过程。如表2-3所示，通过选取一些具体的Xpath路径表示式，显示其查询结果来帮助理解Xpath语言的查询方式。

表2-3 Xpath的路径表达式

|  |  |
| --- | --- |
| **路径表达式** | **结果** |
| bookstore | 选取bookstore元素的所有子节点。 |
| /bookstore | 选取根元素bookstore。 |
| bookstore/book | 选取属于bookstore的子元素的所有book元素。 |
| bookstore//book | 选择属于 bookstore 元素的后代的所有 book 元素，不管它们位于 bookstore 之下的什么位置。 |
| //@lang | 选取名为 lang 的所有属性。 |

为了能够更加清晰和详细地展示Xpath的查询过程，根据图2-4所给出的XML文档以及表2-3所给出的路径表达式以及对应的查询结果，可以绘制出相应的Xpath查询结果节点树。如图2-5所示。



图2-6 Xpath查询结果节点树

根据图2-5的(a)(b)(c)(d)可以清楚地看出，Xpath查询语言可以根据需要写出不同的Xpath路径表达式，根据路径表达式可以准确地锁定相应的节点，查询出节点所包含的信息。特别地，在路径表达式的表示上，需要注意“/”和“//”的差别。前者范围仅限子元素，而后者范围则为所有后代。

# 缓冲式XML DOM与Xpath查询的设计

## 缓冲式XML DOM的设计

### 缓冲式XML DOM数据结构的设计

XML具有标准的树形结构，用链式存储方式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开销较大。本课题研究在流数据环境下设计缓冲式XML DOM，由于实验环境条件有限，通过大型XML文档的解析存储来模拟流数据环境下的XML数据的解析存储。因此为了节省内存空间，避免大型XML文档进行链式存储导致内存溢出等问题，采用顺序存储方式。因此，采用数组来存储DOM信息。通过创建文本数组来保存XML文档中各个节点的文本信息。

此外，在流数据环境下，XML数据流是源源不断到达的，而缓冲式XML DOM应能够实时地对数据流进行解析和存储。由于数据流的数据量极大，通过设计DOM池来存储实时的XML数据流信息。池中保存最新的XML数据，随着数据流的流入，实时地对DOM池中的数据进行更新和替换。

在对XML数据流的信息进行解析和存储的同时，提供XPath语言的查询。查询实时的DOM池中对应的节点信息。由于DOM池中的数据面临存储和查询两个进程的同时调用，因此利用读写锁来控制存储和查询进程对于DOM池中某个DOM的访问。

3.1.1.1 DOM池的设计

之所以设计DOM池是因为在流数据环境下，需要存储的XML数据流信息量巨大。即使仅保存一段有效时间的XML数据流信息，由于数据流不断地流入也需要开辟较大的空间进行存储。在这样的情况下，设计二维数组pool来维持DOM池。数组pool的第一维空间保存的该子DOM在DOM池中的序号，第二维保存的则是XML数据流信息在该子DOM中的存储位置。

由于DOM池中的数据要随着XML数据流信息的流入而进行更新和替换，为了保持DOM池中各个子DOM所保存的XML数据流信息的时序性，设置标志位来保持数据的正确序列。

此外，由于在存储进程保存XML数据流信息的同时，还存在Xpath查询进程访问DOM池获取节点位置信息的可能，因此为每一个子DOM设计单独的读写锁。在DOM池中的某个子DOM写入XML数据流信息时，占有该子DOM，获取对应的写锁，直至写入完毕才释放该写锁。

3.1.1.2节点特征码的设计

由于存储过程采用顺序存储方式，为了记录每一个节点的节点信息，而又不至于占用过多的空间，设计了节点特征码。节点特征码主要记录各个节点的重要属性，以便查询时获取该节点的相关信息。详细设计如表3-1所示。

表3-1 节点特征码格式

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **命名空间数量** | **属性数量** | **节点名称位置** | **有无孩子** | **节点类型** |
| **nsNum** | **attrNum** | **namePos** | **child** | **kind** |
| 0000000 | 000000000 | 000000000000 | 0 | 000 |
| 7bits | 9bits | 12bits | 1bits | 3bits |

通过保存一个32位的整型来保存节点信息，包括命名空间数量，属性数量，节点名称在数组中的存储位置，有无孩子信息和节点类型信息。其中，命名空间在本课题中并未涉及；属性个数和节点名称位置可以通过特定函数获取；有孩子表示为1，无孩子表示为0；节点类型主要分为5种，分别为：元素节点ELEMENT=1，属性节点ATTRIBUTE=2，文本节点TEXT=3，文档节点（根节点）ROOT=4，注释节点COMMENT=5。*另外，需要说明的是，此处的根节点定义与Xpath中的根节点定义相同。*

3.1.1.3节点存储结构设计

在章节2.1.3 XML DOM节点中介绍了XML DOM的主要节点类型，同时3.1.1.2小节中也有涉及到不同节点类型的设计。但是不同的节点类型不仅在节点特征码上有差别，在节点的存储结构上也存在着巨大的不同。不同类型的节点因为它们各自的特征而需要存储不同的信息。通过总结，各个类型的节点特征如表3-2所示。

表3-2 不同类型节点特征

|  |  |  |  |
| --- | --- | --- | --- |
| **节点类型/包含节点** | **父节点** | **子节点** | **属性节点** |
| **元素节点** | ✓ | ✓ | ✓ |
| **属性节点** | 🗶 | 🗶 |  |
| **文本节点（注释节点）** | ✓ | 🗶 | 🗶 |
| **根节点** | 🗶 | ✓ | 🗶 |

根据不同类型节点的特征，为了保证保存信息的完整性同时又不浪费多余的空间去存储不必要的信息，对于不同类型的节点设置了不用的存储结构，如图3-1所示。

|  |  |
| --- | --- |
| 节点存储位置 | 内容 |
| 0 | 根节点特征码 |
| 1 | 根节点第一个孩子的nodePos |

1. 根节点的存储结构

|  |  |
| --- | --- |
| 节点存储位置 | 内容 |
| nodePos | 元素节点特征码 |
| nodePos+1 | 父节点的nodePos |
| nodePos+2 | 后继兄弟节点的nodePos |
| nodePos+3 | 第一个孩子节点的nodePos |

1. 元素节点的存储结构

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

1. 属性节点的存储结构

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

1. 文本节点、注释节点的存储结构

图3-1 不同类型节点的存储结构

通过对不同类型节点按照其对应的存储方式保存节点信息可以有效地记录下节点之间的关系，方便在后续的查询过程中根据存储的数据构建出XML数据流所对应的节点树。但是想要构建节点树，仅仅记录节点的位置信息是不够的，还要记录节点的文本信息。为此，对于不同类型节点的文本信息的存储进行了设计，如表3-3所示。

表3-3 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[] | 文本节点内容已经属性节点的属性值 |

3.1.1.4存储过程举例



图3-2 XML文档存储实例

如图3-2所示，图3-2(a)是一个简单的XML文档。图3-2(b)以及图3-2(c)分别展示了该XML文档的文本信息以及位置信息的存储情况。在进行XML文档解析与存储时，首先扫描根节点。由于根节点具有唯一性，因此在XML数据流不断流入，DOM池数据不断更新替换的过程中，根节点的位置信息始终保持不变，但是由于数据更替的缘故，根节点的第一个孩子位置记录是随时发生变化的。在根节点存入之后，陆续扫描XML数据流中的各个标签，根据不同标签判定对应的节点类型，按照不同类型节点的存储结构顺序存储在DOM池中，如图3-2(c)所示。

经过了图3-2的实例，可以更加清晰地了解XML数据流的存储过程，但是很难看到细致的存储信息，例如，节点特征码是以什么形式存储的；在XML数据流流入过程中DOM池中的相关状态等等。下面就详细地阐述一下存储的细节信息。如图3-3展示了图3-2(a)中的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节点的命名空间个数，属性个数以及节点名称位置，判断节点类型，将相应位置1；通过后续节点的解析和存储，确定有无孩子。获取各种信息之后，将二进制转化为十进制的整型存储在DOM池中的相应位置上。



图3-3 <book>节点特征码实例

了解过以上内容之后，整个存储过程就非常清晰了。但是DOM池的状态依旧是一个非常抽象的问题，随着XML数据流的不断流入，DOM池中的数据不断进行更替，很难能够说明DOM池中每一个时刻的状态。为了能够说明这个问题，假设XML数据流解析到</title>的位置，图3-4展示了此时DOM池的状态。



图3-4 某时刻DOM池状态

这里，需要特别说明的是，位置4和12处标有下划线的两个值-1只是暂时的，由于此时刻未遍历后续的节点，因此无法判断此时该节点的兄弟节点的位置，因此暂定为无兄弟。

另外，仔细观察图3-4，可以看出标志位的变化。这里由于XML文档实例内容较少，因此假设DOM池中仅存在5个子DOM，每个子DOM占有8个数组空间。存储从第一个子DOM可是进行，标志位初始化为0。当存储空间不足时，开始存储第二个子DOM，相应的，标志位也累加变为1，直至当前DOM池中所有的子DOM都存储满XML数据，便再次从第一个子DOM开始循环存储，同时标志位依旧不断累加。详细示意如图3-5所示。

图3-5 DOM池存储示意图

### 缓冲式XML DOM节点的设计

在上一章节3.1.1中，介绍了缓冲式XML DOM的数据结构。详细地说明了XML数据流是如何在DOM池中存储的。在章节2.1.3 XML DOM节点中也介绍了DOM主要以节点的形式存在，也通过对节点对象提供访问方法来获取节点的相关信息。因此，在使用时，需要对不同类型的节点对象进行封装，下面对于封装的节点对象进行简要的介绍。

在章节3.1.1.2中提到，常用的节点类型主要有5种，包括：文档节点（根节点），元素节点，属性节点，文本节点和注释节点。由于在实际的查询中，几乎不会出现对于注释节点的查询，而在XML数据流的环境下，建立节点不仅占用内存空间，同时也会耗费查询进程的时间，因此这里不进行注释节点的封装。

考虑文档节点（根节点）（RootNode），元素节点（ElementNode），属性节点（attributeNode）以及文本节点（TextNode）的特点，在整体上他们都属于XML DOM的节点，虽然类型不同，但是根据W3C的标准，他们都应该继承同一个父类DOMNode。通过对于父类方法的继承和多态来实现各自内部的方法。图3-6给出了XML DOM各节点的关系。



图3-6 XML DOM节点关系

这里DOMNode通过指针直接访问DOM池内的数据。外界若想获取DOM池中相关XML数据流的实时信息以及查询某些节点都需要通过DOMNode提供的特定接口才能获取这些信息。根据W3C标准的XML DOM所实现的方法，本课题中提供了一些Xpath语言查询时可能用到的方法，如表3-4所示。在查询时需要通过访问DOMNode中的相关方法获得XML数据流信息。

表3-4 DOMNode方法

|  |  |  |
| --- | --- | --- |
| 方法 | 返回类型 | 描述 |
| getNodeName() | String | 获取节点名称 |
| getNodeType() | nodeType | 获取节点类型 |
| getNameValue() | String | 获取节点值 |
| getFirstChild() | DOMNode\* | 获取第一个孩子 |
| getNextSibling() | DOMNode\* | 获取下一个兄弟 |
| getAllChild() | GetChildNode\* | 获取所有孩子 |

## Xpath的设计

### Xpath模块功能

在上一章2.4节中介绍了Xpath的基本知识，Xpath主要通过路径表达式对XML文档的特定节点进行查找。此处实现Xpath查询模块是为了验证缓冲式XML DOM存储的结果是否正确。根据Xpath路径表达式的表达方式，首先要进行分词，将路径表达式中的表达式以及路径词汇分离。例如：“bookstore/book”，需要提取出的部分是“bookstore”“/”“book”。本课题中主要进行缓冲式XML DOM的设计，Xpath查询仅仅是验证效果的作用，因为本课题中不研究如何进行解词，如有需要外接Xpath解词工具，连接Xpath接口即可。

### Xpath模块接口设计

由于Xpath查询不作为主要设计部分，因此，只编写Xpath的接口。即：获取解词之后的结果，将解词之后的结果分为两部分：

* + 操作

所谓“操作”就是Xpath路径表达式中的“/”和“//”。我们可能仅仅需要遍历该节点的子元素，也可能需要遍历该节点的所有后代。因此需要进行不同的操作。

* 操作对象

所谓的“操作对象”则是路径表达式中的“bookstore”或“book”这些名称。在查询时必须确定是在哪一个节点下进行查找。因此，操作对象的接口也是非常关键的。

在本课题中，利用大型的XML文档来模拟XML数据流，假设所有的XML数据流信息均在根节点内部，因此需要进行的Xpath查询都需要根节点的路径之下。因此进行如图3-7这样的设计。



图3-7 Xpath路径设计实例

图3-7基于上面章节3.1.1.4中的图3-2(a)的XML文档，在这里默认根节点为顶层节点“bookstore”，所有的查找都在“bookstore”路径下，这里实际进行的操作如图3-8所示。首先查找“bookstore”节点下所有的子元素，找到名为“book”的子元素，加入结果列表，遍历结果列表中的所有元素的子元素，找到名为“title”的子元素，最终加入结果列表。



图3-8 Xpath查询流程图

## 系统同步机制设计

本课题中，通过多线程来实现缓冲式XML DOM的存储和Xpath查询同时进行，而存储线程是将XML数据流信息写入DOM的过程，查询线程则是从DOM中读取XML数据流信息的过程，想要同时访问XML DOM这个临界资源，则需要读写锁来进行同步控制。

读写锁的特点是将对共享资源的访问者划分为读者和写者。一次只有一个线程可以占有写模式的读写锁，但可以有多个线程占有读模式的读写锁，但是一个读写锁不能既有读者又有写者，即读者和写者是互斥的。

在本程序中，存储线程作为写者，而查询线程作为读者。为DOM池中的每一个子DOM设计专属于自己的读写锁。写者需要向该子DOM块中写入XML数据流信息时需要提前获取该子DOM块的写锁，在该子DOM块写入完毕时释放写锁。同样的，读者需要遍历该子DOM块中的节点位置信息时，需要获取该子DOM块的读锁，在查询完毕之后释放读锁。

如图3-9所示，假设现在的场景是正在向第2块子DOM写入XML数据流信息，而当前根节点的第一个孩子在第4块子DOM中，首先要进行的操作便是对第2块子DOM加写锁，对第4块子DOM加读锁。在第2块子DOM和第4块子DOM中分别进行写入XML数据和遍历各个节点查询结果这两个操作。当第2块子DOM空间不足时，释放该DOM块上的写锁，为下一块子DOM加写锁。由于查询线程需要建立节点，遍历节点名称进行比较，因此存储线程速度较快，在存储线程检测到第3块子DOM空间不足时，需要释放该DOM的写锁，并且等待查询完毕后第4块子DOM的读锁释放，才能够获取该块的写锁，继续向该块中写入XML数据流信息。



图3-9 同步机制读写锁示意图

# 系统实现

## 系统结构

整个系统主要分为两个部分：

* + 流数据环境下的缓冲式XML DOM
* 流数据环境下对于XML DOM的Xpath查询

不难发现，所谓“流数据环境下”是一个大前提，而这就是难点。根据上一章3.1.1中对于缓冲式XML DOM数据结构的设计就可以看出，流数据环境下的存储需要进行数据的更新和替换，但同时又要保持数据的次序不被改变。而流数据环境下的查询则与普通的查询无异，只是困难在于流数据的存储实时更新，查询也需要根据每一时刻的存储而进行变化，这就出现了下一个难点：存储部分和查询部分两进程的同步问题。上述难点的实现会在下面的章节中进行详细地解释和说明。

根据系统结构的设计，分别需要对缓冲式XML DOM部分和Xpath语言查询实时XML数据流信息部分的工作进行描述。如图4-1所示，左侧描述的是缓冲式XML DOM的生成过程，右侧描述的是Xpath查询过程。缓冲式XML DOM的生成与Xpath的查询是同时进行的。图中主要包含三个模块：XML DOM生成器；Xpath解析器和查询引擎。其中，XML DOM生成器主要通过Xerces-C++解析器实现；Xpath解析器采用指定的Xpath解析路径实现；而查询引擎通过根据已生成的XML DOM建立节点，用Xpath\_Query类中的方法遍历DOM节点进行查询的方式实现。



图4-1 系统结构图

介绍过系统结构之后，分别来对系统的两个部分进行详细的说明。首先，图4-2展示了XML DOM生成的主要流程。



图4-2 XML DOM生成流程图

图4-3展示了Xpath查询的主要流程。



图4-3 Xpath查询流程图

图4-2和图4-3主要体现了存储和查询两个线程中的主要步骤以及读写锁加锁和解锁的条件。通过存储线程和查询线程的流程图可以更加深入的了解章节3.3中介绍的同步机制的设计。

## 缓冲式XML DOM的实现

### 缓冲式XML DOM生成的实现

上面章节2.2.4中提到本课题中采用开源解析器Xerces-C++来进行XML文档的解析。Xerces-C++已经给出了SAX2解析的实现，只需将其直接引入到项目中，重写SAX2事件触发函数便能够实现从XML文档到XML DOM的转化了。具体的解析过程如2.2.4节中的图2-4所示。下面对于事件触发函数的实现进行详细说明。

4.2.1.1 DOM

在SAX2读取XML文档的各个标签时，触发对应的函数，将标签中的信息存储进DOM中。DOM作为XML数据信息存储的载体起到相当重要的作用，所以根据章节3.1.1中的数据结构的设计，创建了DOM类，图4-4为DOM类的类图。



图4-4 DOM类类图

表4-1 DOM类方法

|  |  |  |
| --- | --- | --- |
| 函数名称 | 类型 | 描述 |
| encode | int | 获取节点信息，返回节点特征码 |
| expandDOM | void | 初始化各个子DOM，每个子DOM存储空间不足时根据不同情况进行处理 |
| findName | bool | 文本信息查询，不重复存储相同的文本 |
| setBlocks | void | 设置DOM中某一位置的值 |
| getBlocks | int | 获取DOM中某一位置的值 |
| CreateNode | void | 将DOM池中的信息传递给创建XML DOM节点函数CreateDOMNode |

这里，出现update数据类型的变量，update类型包含两个整型的value和flag：value保存位置信息，flag保存标志位信息。相关设计在章节3.1.1.4中进行了说明。

其实，DOM类中的方法只是为DefaultHandler触发相应事件函数，将XML文档转化为XML DOM时访问DOM提供的接口，这些方法实现起来都非常的通俗易懂，因此不多做解释。唯一需要简单说明的就是expandDOM函数，图4-5展示了该函数的流程。每当存储节点信息时都需要判定正在存储的该子DOM块是否还有足够的存储空间，如果存储空间不足，就要进行expandDOM操作。由于DOM池空间大小固定，由第一块子DOM向后存储，当最后一块子DOM存储空间不足时，则继续存入第一个子DOM中，以此类推，循环往复。



图4-5 expandDOM函数流程图

4.2.1.2 Handler

介绍了DOM类的构建，下面对于DefaultHandler是如何触发事件函数的过程进行说明。章节2.2.3中已经提到，想要实现事件触发函数需要通过ContentHandler接口，而ContentHandler接口需要通过DefaultHandler类实现。因此，为了实现事件触发函数并且生成顺序存储的XML DOM，需要创建MySAX2Handler类来重写事件触发函数，通过该类继承DefaultHandler中的方法，关系如图4-6所示。



图4-6 DefaultHandler与MySAX2Handler类图与关系图

可以看出MySAX2Handler中的各个事件触发函数与XML文档中的各种标签是一一对应的。例如：当扫描到XML文档中的<book>标签时，便会触发startElement事件，调用MySAX2Handler中的startElement函数进行处理。因此，想要保存各个类型的节点信息，只需在扫描该节点标签时获取该节点的各种信息，例如：属性个数，节点类型，节点名称等，将这些信息保存在相应位置。存储规则在章节3.1.1中已详细介绍。但是节点的父亲，后继兄弟以及第一个孩子的位置是无法立刻得知的。很容易理解，扫描该节点时只能获取该节点的相关信息，无法得知它与其前后节点的关系。此时，需要将其保存父节点位置，后继兄弟节点位置以及第一个孩子位置信息的空间留出，等待扫描到后续节点时便可回填这些信息了。



图4-7 处理算法举例和主要栈的栈中状态图

如图4-7所示，给出在触发不同的标签时MySAX2Handler类中主要操作的栈和队列的状态。表4-2给出了它们所存储的数据类型以及作用。

表4-2 主要栈的数据类型及作用

|  |  |  |
| --- | --- | --- |
| 栈/队列 | 数据类型 | 作用 |
| 栈labelMatch | string | 开闭标签匹配 |
| 队列rootChild | update | 保存根节点的孩子 |
| 栈positionStack | upadate | 保存节点特征码位置 |
| 栈judgeStack | bool | 判断节点是否出栈，帮助判断父兄关系 |

通过labelMatch栈进行标签匹配可以帮助rootChild队列保存根节点的每一个孩子位置。另外，positionStack和judgeStack两个栈是同时进行入栈出栈的。它们的协作规则是：每当遇到元素开始标签时，将该元素节点的位置压入位置栈，判断栈压入0即未访问状态；每当遇到文本标签时，将该文本节点的位置压入位置栈，判断栈压入1即已访问状态；每当遇到元素结束标签时，位置栈和判断栈同时出栈至判断栈栈顶为0，出栈元素即为栈顶元素的孩子，且出栈元素之间互为兄弟，出栈同时设置出栈元素的兄弟关系和栈顶元素的第一个孩子位置，出栈完毕后，设置判断栈栈顶为1。

综上所述，通过触发不同的XML文档标签，调用不同的事件触发函数，进行不同的栈操作来实现XML DOM的生成。下面对于各个触发事件函数的实现进行详细说明。

* startDocument文档开始事件

表4-3 文档开始事件函数

|  |  |  |
| --- | --- | --- |
| 函数 | 参数 | 返回值 |
| startDocument | void | void |

文档开始事件处理函数实现功能简单，仅需要对第一块子DOM上写锁，为后续的写入信息做准备。

* endDocument文档结束事件

表4-4 文档结束事件函数

|  |  |  |
| --- | --- | --- |
| 函数 | 参数 | 返回值 |
| endDocument | void | void |

同样，文档结束事件处理函数仅需要对正在储存的子DOM块释放写锁，方便后续查询。

* startElement元素开始事件

表4-5 元素开始事件函数

|  |  |  |
| --- | --- | --- |
| 函数 | 参数 | 返回值 |
| startElement | const XMLCh\* const uri, //命名空间uri  const XMLCh\* const localname, //元素名  const XMLCh\* const qname, //限定名  const Attributes& attrs //属性元素集合 | void |



图4-8 元素开始事件函数流程图

图4-8详细展示了元素开始事件函数的主要流程，需要简单说明的是：在确定某一元素节点为根节点ROOT的孩子时，需要暂定其没有后继兄弟即兄弟节点位置为-1。这样做的原因是，如果该节点有后继兄弟，此时还没有扫描到其后继兄弟节点的标签，若此时需要进行查询，又没有将该节点兄弟节点位置设置为-1，则会导致错误的建立该节点的后继兄弟节点，导致查询出现错误。因此，无论扫描到哪个位置，都需要保持XML DOM内数据的正确性。

* endElement元素结束事件

表4-6 元素开始事件函数

|  |  |  |
| --- | --- | --- |
| 函数 | 参数 | 返回值 |
| endElement | const XMLCh\* const uri, //命名空间uri  const XMLCh\* const localname, //元素名  const XMLCh\* const qname, //限定名 | void |



图4-9元素结束事件函数流程图

上图4-9展示的元素结束事件函数的主要流程，在元素结束事件函数部分不需要存入新的节点内容，只需要对该元素内部节点之间的关系进行回填。暂存栈是用来保存该元素的孩子元素的信息的。

* characters文本事件

表4-6 元素开始事件函数

|  |  |  |
| --- | --- | --- |
| 函数 | 参数 | 返回值 |
| characters | const XMLCh\* const chars, //文本内容  const XMLSize\_t length //文本长度 | void |



图4-10 文本事件函数流程图

图4-10详细说明了文本事件触发函数的流程。这里的position为当前正在处理的DOM池中的位置，position+2是为了预留出文本节点的父节点位置和兄弟节点位置，方便在后面扫描XML文档时进行回填。

### 缓冲式XML DOM节点的实现

在章节3.1.2中已经介绍了对缓冲式XML DOM节点的设计，下面来说明一下这些节点是如何实现的。

在创建缓冲式XML DOM节点时，不可置疑的需要用到已经生成的XML DOM中存储信息。因此，通过创建一个CreateDOMNode类，来进行XML DOM节点的封装。在章节4.2.1.1 DOM的设计中可以看到，DOM类中的CreateNode方法便是通过创建一个CreateDOMNode的实例来将XML DOM的信息传递给CreateDOMNode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CreateDOMNode类仅需要创建一个实例。因为节点的创建不仅耗时，而且占用内存空间，所以CreateDOMNode类使用单例模式实现。单例的构造函数是私有的，外部程序如果想要访问这个单例类的话，必须通过GetInstance()来请求得到这个单例类的实例。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。因此，在DOM类中创建了CreateDOMNode类的实例之后，其他类想要调用CreateDOMNode类中的方法都需要通过GetInstance()获取该实例进行操作。

图4-11给出了CreateDOMNode类的类图。其中createNode()方法和createChildNode()方法负责创建DOMNode节点。属性部分都是通过DOM类传递过来的指针。类中各个方法的含义和作用如表4-7所示。



图4-11 CreateDOMNode类类图

表4-7 CreateDOMNode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 |

不同的类型的节点对应不同的节点类，而DOMNode类作为基类派生这些不同的节点类。DOMNode类实现所有节点所需的方法，而不同类型的节点根据其自己本身的特性，选择是继承父类的方法还是通过多态来实现自己类中的方法。图4-12所示为DOMNode类类图。其中GetChildNode和GetAttrNode分别是为查找某一节点所有孩子和为查找某一元素节点所有属性所创建的类。



图4-12 DOMNode类类图

综上所述，CreateDOMNode类根据需要将XML DOM内存储的XML数据流信息转化为XML DOM节点信息，创建出相应的节点。而转化过程如图4-13所示。



图4-13 XML DOM存储信息转化为节点信息举例

如上图，假设该节点为结构最为复杂的元素节点，该节点已经获取到自己的存储信息。可以根据节点特征码所在位置去获取该节点的节点特征码。便可以根据其节点特征码获取该节点的属性个数，节点名称位置，有无孩子，节点类型等信息（如章节3.1.1中介绍节点特征码结构）。在这之后，便可以根据章节3.1.1中提到的该节点的存储结构去获取该节点的父节点，后继兄弟节点以及第一个孩子节点的位置信息，根据它们的位置来进一步获取它们的节点特征码。例如，如果想知道图4-13中元素节点的名称，只需要获取该元素节点的特征码，通过获取其特征码的相应位来得知其节点名称存储的位置，到elementNames数组中的相应位置便可以获取到该节点的名称了。在程序中，通过DOMNode类的getNodeName()方法来实现这一过程。

## Xpath查询的实现

### Xpath的处理

章节2.4.1中介绍，Xpath通过路径表达式查询XML文档中的信息。章节3.2.2中也介绍了Xpath模块接口的设计。因此，很容易根据Xpath查询的特点以及接口的设计来进行实现。首先，通过Xpath对象来接收Xpath路径表达式解析之后的结果。



图4-14 Xpath类的类图

如图4-14所示，通过创建一个名为Query的结构体来实现。Query结构体内部有两个成员变量：operation和nodename。它们分别表示要进行的操作以及操作对象的名称。由于只是简单查询，因此只提供两种查询操作。ONE\_CHILD操作指的是路径表达式中的“/”，而ALL\_CHILD操作指的则是路径表达式中的“//”。通过创建一个Query类型的一维数组来接收各种Xpath路径表达式的解析结果，在本程序中进行简单的指定查询。查询实例如图4-15所示。



图4-15 Xpath查询处理实例

### Xpath的查询

Xpath的查询通过Xpath\_Query类实现，需要获取根节点以及Xpath对象的指针。图4-16给出了Xpath\_Query类的类图。startQuery方法作为接口接收Xpath对象指针。根据要进行的操作以及操作对象名称调用getOneChild()或是getAllChild()方法从根节点开始遍历各节点进行查询，将中间的查询结果保存在handleList列表中，并将最后的查询结果保存在resultList列表中。图4-16给出了Xpath\_query类的类图。



图4-16 Xpath\_Query类的类图

注意观察不难发现，Xpath\_Query类中还存在读写锁变量。这是因为在查询时，需要获取根节点第一个孩子所在子DOM块以及该DOM块的读写锁，并且对其加读锁，待查询完毕后释放该DOM块的读锁，这样才能保证系统同步的正确性。下面简要说明一下getOneChild()函数的主要流程，getAllChild()函数流程与其相似，如图4-17所示。



图4-17 getOneChild()函数流程图

# 测试实验

## 缓冲式XML DOM生成部分性能测试

缓冲式XML DOM的生成是本课题的核心部分，为了了解这部分功能的性能，设计了不同量级的XML文档进行性能测试。经过对不同文档的多次测试，得到了表5-1中的结果。

表5-1 缓冲式XML DOM生成部分性能

|  |  |
| --- | --- |
| XML文档量级 | 缓冲式XML DOM完全生成所用时间(s) |
| 10MB | 2.69 |
| 100MB | 25.195 |
| 200MB | 50.537 |
| 300MB | 74.865 |
| 400MB | 99.814 |
| 500MB | 122.916 |
| 700MB | 178.978 |
| 1GB | 249.785 |

经过测试发现，整个缓冲式XML DOM的生成时间较长，为了了解程序的瓶颈，利用Visual studio内部的性能测试工具对该项目做了采样。结果如图5-1所示。根据热路径的函数名可以看出，程序很大一部分的时间花费在调用Xerces-C++中的扫描标签内容的函数上，因此，若是之后想要对缓冲式XML DOM生成部分的性能做改进，可以考虑是否可以重写标签扫描函数或是测试其他开源的XML文档解析工具。

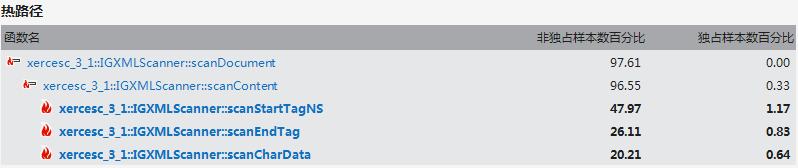


图5-1 CPU采样结果

## 系统功能测试

由于缓冲式XML DOM中的数据随着读入的XML数据流变化而变化，因此在不同的时刻进行查询，最后查询到的结果是不同的。为了检验缓冲式XML DOM生成的正确性，分别设置了三次查询，这三次查询的查询时间如表5-2所示。

表5-2 查询时间列表

|  |  |
| --- | --- |
| 查询次序 | 查询时间 |
| 1 | 程序开始后1800ms |
| 2 | 程序开始后2100ms |
| 3 | 程序开始后5000ms |

之所以这样设计是因为经过多次测试发现，第一次查询时是程序刚刚开始运行，此时DOM池中的子DOM中没有存储任何数据，缓冲式XML DOM中不进行任何数据的更新和替换。而第二次查询时，缓冲式XML DOM中已经出现了数据的更新和替换，最初被存储的XML数据流的数据已经被新的数据所覆盖。到了第三次查询，缓冲式XML DOM中的数据已经经过了多次的替换，最初存储的数据已经完全查询不到了。下面的图5-2，图5-3以及图5-4分别显示了第一次，第二次以及第三次查询的结果，通过这些结果能够明显的看出缓冲式XML DOM中数据更新替换的情况。



图5-2 第一次查询结果



图5-3 第二次查询结果



图5-4 第三次查询结果

# 结论

本文实现了一种在流数据环境下基于顺序存储的缓冲式XML DOM，同时支持Xpath语言对该缓冲式XML DOM进行实时查询。

本文设计的缓冲式XML DOM实质上由一个拥有多个子DOM的DOM池以及存储文本信息的容器组成。随着XML数据流的读入，将数据流信息转化为节点索引信息保存在缓冲式XML DOM中，当DOM池空间不足时，新的XML数据流信息覆盖旧的继续进行存储。由于XML数据流具有标准的树形结构，因此也需要设计记录这种树形结构的数据结构来保存XML数据流信息。在完成了XML DOM的存储方式以及数据结构设计之后，就可以通过SAX解析器解析XML数据流，将XML数据流中的各个不同标签所对应的节点根据它们独特的数据结构存储生成缓冲式XML DOM。为了能够对该缓冲式XML DOM内部的数据进行访问，将它们封装成为XML DOM节点。并提供访问这些节点信息的接口，例如：获取该节点的节点名称，获取该节点的第一个孩子等等。

在完成了XML数据流信息的存储，生成了缓冲式XML DOM之后，需要验证缓冲式XML DOM中的存储的数据的正确性。因此采用Xpath语言进行查询，但是在查询的同时又要保证不影响XML数据流信息持续存储进缓冲式XML DOM内。为此，为DOM池中的每一块子DOM设计读写锁。该子DOM解析存储XML数据流时获取写锁，在对该子DOM进行查询时获取读锁，用读写锁的方式来控制对缓冲式XML DOM进行存储操作和查询操作的同步。

在毕业设计规定的时间内完成了本课题要求的功能，但是有一些地方依旧存在问题，需要进行研究和改进。如在性能测试时发现Xerces-C++中SAX解析器部分处理效率并不高，可以尝试其他解析器。此外，本程序中的错误处理机制还不够完善，出现错误时并不能够明确指出错误原因和错误的位置。

# 参考文献

1. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (Fifth Edition)

https://www.w3.org/TR/2008/REC-xml-20081126/

1. Jonathan Robie, Don Chamberlin, Michael Dyck. XML Path Language (XPath) 3.0

https://www.w3.org/TR/2014/REC-xpath-30-20140408/

1. Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs. Document Object Model (DOM) Level 1 Specification

https://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/

1. A Dissertation by Dipl.-Inf. Christian Mathis. Storing, Indexing, and Querying XML Documents in Native XML Database Management Systems. April 2009
2. C++语言程序设计（第4版），郑莉，董渊，何江舟，清华大学出版社，2010年7月
3. C++设计模式，ALANEZUST，清华大学出版社，2007年8月
4. Effective C++中文版，Scott Meyers，华中科技大学出版社，2001年9月
5. C++面向对象多线程编程， 周良忠， 人民邮电出版社，2003年4月

# 致谢

首先，我要感谢我的毕业设计指导老师苏航老师。从毕业设计开始阶段，我对于很多基础的知识都还不了解，苏老师就不厌其烦地为我讲解。每当我有问题，苏老师都会放下手边的工作为我解答。正是因为这样，我才能对本课题有一个全面的认识和把握。在设计之初，我在很多问题的认识上都不够透彻，设计的程序也是漏洞百出，苏老师会再次跟我明确要实现的功能并且提供给我一个可以改进的方向。在实现通过读写锁进行存储查询线程同步时，我的程序调试总是出现问题，苏老师花费很多的时间和精力和我一起修改调试程序，思考解决方案，直到最终程序能够成功运行。在整个毕业设计时期，苏老师带给了我很多的帮助和鼓励。对于他严谨认真的治学态度我也感到深深地尊重和敬佩。

其次，我还要感谢实验室刘倚天学姐。在我对于这个课题完全陌生的时候，是她细心为我讲解一些基础知识，才让我对于这个课题有了一个很具体的认识。

另外，我还要感谢我的父母，在我毕业设计期间对我学习上、生活上的支持，使我能够专心学习。以及我的室友，在我撰写毕业论文期间在论文规范上对我的提示和帮助。