导航处理 XQuery 的概要设计

负责人：陆世潮
编写人：陆世潮
系统版本号：OrientX Version 2.0
完成时间：2004/9/9
开发单位：中国人民大学 IDKE 实验室 XML 工作组

引言

OrientX 系统是一个 Native XML 数据库系统，专门用于管理和维护 XML 数据。众所周知，XML 数据是一种树状结构，具有自我描述的特性，一个结点下面可有 0 个或多个结点。在存储时，Native XML 数据库把 XML 数据划分成多个较小的子树，每一棵 XML 子树就是一个记录，保留了 XML 数据的树状结构。存储模块向上层模块提供了类似 DOM 的数据读取方法，以遍历 XML 数据。

如同关系数据库需要用 SQL 语言来检索数据一样，XML 数据库也需要一种简单易用的查询语言来从海量的 XML 数据中抽取、重构出符合特定条件的数据。关于 XML 查询语言，曾出现过很多种，例如 Quilt，XQL，XPath，XQuery 等；W3C 推荐的 XQuery 正日臻完善，逐渐得到大家的公认，最有可能成为 XML 查询语言的标准。

OrientX 系统在 1.5 版本已经支持 XPath 查询语言;但是由于其 XPath 的表达能力有限，OrientX 在 2.0 版本中增加支持 XQuery。本文就是概要设计在 OrientX 系统如何导航式处理 XQuery。

本文正是概要设计 Xquery 在 OrientX 系统中的处理方法。后面文章的组织方式如下：第 2 部分介绍 OrientX 系统导航式处理 XQuery 的背景。第 3 部分介绍 OrientX 系统的总体框架，XQuery 查询引擎在整个系统的地位，以及 XQuery 处理的流程；第 4 部分则是 OrientX 系统 Xquery 处理模块与其他模块的接口；第 5 部分是实现 Xquery 的主要类的设计。第 6 部分介绍处理 Xquery 的重要算法。第 7 部分总结 XQuery 处理的经验，以及尚未解决的问题。

2．Xquery 的处理策略

Xquery 的处理策略，按照读取数据的方法可以分为基于导航的和基于集合的。

所谓基于导航的处理，就是根据 XQuery 查询的导航性语义，采用类似 DOM 接口的方式遍历 XML 文档，抽取出符合特定条件的数据片断。例如对于查询 doc(“books.xml”)//book/author，它将从文档的根节点遍历整个文档 books.xml，对于找到的每一个 book 结点，再往下遍历其子结点，每遇到一个 author 结点，就向上返回。导航式处理的好处在于，它通过导航来遍历 XML 数据树，与 Xquery 导航性语义相吻合，使得这种基于语法的导航式处理更加符合人们的思维。目前实现了 XQuery 导航式处理的系统有 IPSI-XQ ,Galax，Xindice 等。

受到 SQL 的处理策略的启发，人们认为 XQuery 设计一套基于集合的代数，采用每次一集合地处理 XML 数据，其效率比每次一结点的导航处理方式会更高。集合式处理 XQuery 的典型代表是 Timber 系统，它从与关系数据库进行类比出发，定义了一套称为 TAX 的 XML 代数。它极力模仿关系代数，以利用关系代数系统的成熟理论。TAX 的每一个操作符以一个或多个有序的子树集合作为输入，产生出一个有序的子树集合。

在没有得到确切的实验数据之前，我们暂且不讨论孰优孰劣。但是，可以肯定的是，导航式处理的思想，更加符合 XQuery 语言的过程化特征。

在 OrientX 系统 2.0 版本，我们实现了 XQuery 的导航式处理。本文概要地设计 XQuery 导航式处理的思想以及程序设计过程中的一些重要技术问题。

3. 总体设计

Xquery 查询引擎在整个系统的地位如图 1 所示，它的**功能主要是**为用户提供快速高效的数据查询、检索功能。为此，我们处理 Xquery 时除了要确保结果的正确性之外，还要尽可能提高查询的效率。

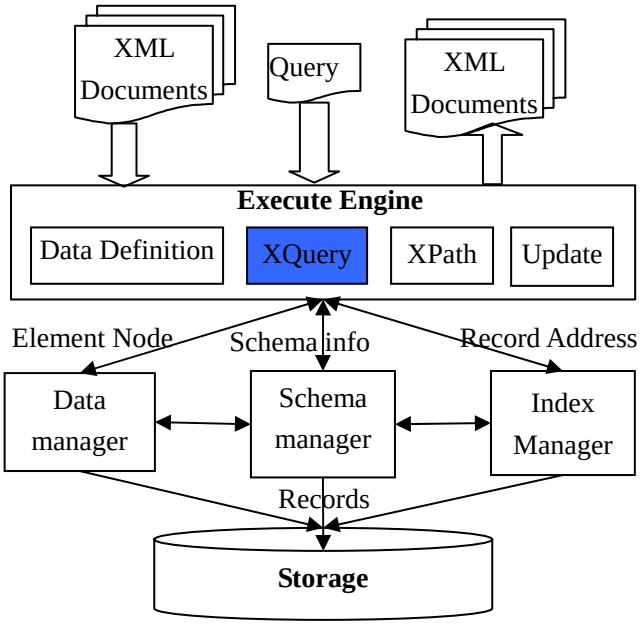

图 1　OrientX 系统框架

XQuery 查询内部的结构框架如图 2 所示。处理 Xquery 主要包括三个步骤：首先是语法分析、语义检查、生成执行计划；然后是优化，选择一个代价最优的执行计划；最后是执行，返回结果。

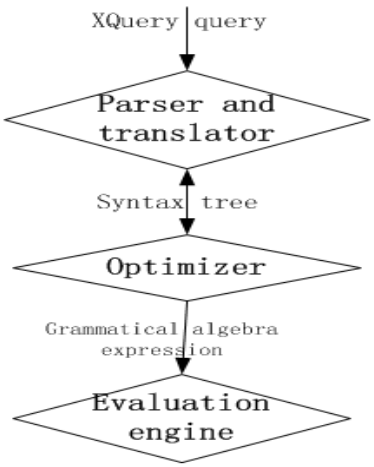

图 2　XQuery 查询引擎

步骤 1：对用户输入的 XQuery，先用 W3C 推荐的语法规则（请参照文档“XQuery 文法规则的改写”）进行语法分析，判断它是否一个符合 XQuery 语法要求的查询。如果符合的话，那么就可以构建出相应的语法树（请看文档“XQuery Syntax Parse 概要设计”）。**但是**，我们并没有在物理上构建出 XQuery 的语法树，而是在构建逻辑上的语法树的同时，就直接生成相应的查询计划。

这里的执行计划是用 XAlgebra 代数的 13 个操作符表示的。关于 XAlgebra 代数以及生成查询计划的算法请看第 3 节介绍。

步骤 2：优化从语法树生成的原始执行计划，选择其中执行代价最优的一个。由于种种原因，这部分优化工作，还没有开展起来。

步骤 3：根据选定的执行计划树，执行查询。由于我们采用导航式处理策略，执行计划树上的每一个操作以其子结点的输出作为它的输入。而每个操作都有一个上下文环境（根节点例外），这个环境是由其父结点操作动态提供的。因此导航式处理，就是根据执行计划树，自上而下地执行计划树上的每一个操作，当需要输入时，调用其子操作即可。

我们可以说，XQuery 的导航式处理，就是一系列的函数递归调用。

4. XAlgebra 代数

4.1 XAlgebra 代数的提出

XQuery 的语法形式千变万化，满足了用户书写复杂查询的要求，却给系统实现带了很大的麻烦。其中最大挑战在于设计一套适合 XQuery 处理的代数（例如关系代数之于 SQL），用于明确地定义 XQuery 的语义（XQuery 查询具有很大的语义隐含性，实现时需要将其明确地表示出来），表示查询在查询引擎内部的执行计划，操纵 XML 数据。基于这样的代数系统，查询才能够得以执行，才能进行下一步优化以提高查询效率。

我们提出了一套基于语法的代数，命名为 ***XAlgebra***。之所以称之为“基于语法的”，是因为 *XAlgebra* 里的操作符是与 XQuery 的语法单位相对应的，也就是说，*XAlgebra* 的每一个操作符对应于 XQuery 的有限确定个语法产生式，XQuery 的每一个语法产生式可以用 *XAlgebra* 的有限确定个操作符来表示。因此， XAlgebra 是具有**完备性**的，即它可以表示所有情形的 XQuery 查询。

但是 OrientX 只是一个原型系统，只是为了提供一个试验平台，并没有必要支持所有情形的 XQuery。因此，XAlgebra 中只设有 13 个操作符，但是可以表示绝大多数形式的 XQuery 查询。量的操作符不仅有利于简化查询执行计划，而且有利于查询执行引擎的实现。此外，*XAlgebra* 的操作导航式地操纵 XML 数据；这与 XQuery 语义的过程化特性相符。XQuery 的导航式处理，遇到一个数据结点时，如果该结点下没有子结点，那么就不再往下遍历而是返回上一层的结点；如果该结点下有多个子结点，则依次递归遍历所有子结点。这样的处理方式很好地解决了 XML 数据的不确定性问题。

关于 XAlgebra 代数的背景、思想等，可以参考论文“OrientX 中 XQuery 的导航式实现”。

后面将介绍 XAlgebra 的设计，然后再介绍如何根据语法树生成执行计划树，并举例说明。

查询 1：查找所有书中相同书名的价格最低的书。

```
<results>{
    let $doc := doc("docs/prices.xml")
    for $t in distinct-values($doc//book/title)
    let $p := $doc//book[title = $t]/price
    return
      <minprice title="{ $t }">
        <price>{ min($p) }</price>
      </minprice>
  }</results>
```

4.2 XAlgebra 的设计

根据 OrientX 现已支持的 XQuery 的功能，现有 XAlgebra 共有 13 操作符：Step，CondTreeNode，Path，ForVarBind，LetVarBind，FLWR，EleConstructor，AttrConstructor，BuiltInFun，IfThenElse，Qualify，SetOpt，SortBy。

这里还没有支持类型表达式（例如 cast as，instance of，typeswitch 等）、用户自定义环境（例如函数自定义，数据自定义，模式自定义等）。

1. Step

Step 操作，对应于 XQuery 语法中能够规约到 StepExpr 表达式的表达式，包括常量（Literal），变量（Variable），函数（PrimayExpr，PrimaryExpr 中目前只支持函数），普通的 Step（例如 Element，Attribute，WildCard，Text 等）。每一个 Step 操作，还必须指明从上下文结点与本结点的关系：父子关系“/”还是祖先后代关系“//”。还可以附带一个谓词操作 CondTreeNode。

对每一个 Step 操作进行求值时，总会有一个上下文环境，该环境表明是从文档树上哪个数据结点往下遍历。当然，如果 Step 操作是常量，变量，函数等类型，该上下文环境并不起作用。

2. CondTreeNode

操作 CondTreeNode，本身只是一个条件过滤操作，对应于 StepExpr 中的谓词，或者 FLWR 语句中的 Where 条件，或者 IfThenElse 语句中的条件子句，Quantify 语句中的 Satisfy 子句。

CondTreeNode 操作符根据它的操作结果类型可以分成三种类型：

- 布尔条件，返回真或假。常见的情况是路径表达式与常量，或者路径表达式和变量的比较操作。
- 范围条件，返回的是一个整数区域或者一个整数
- 存在谓词。可以返回任意类型的数据，例如 ElementNode，常量等。

3. Path

操作符 Path 对应于 Xquery 语法中的 PathExpr。Path 操作由一系列有序的 Step 操作组成。我们通过示例来看看路径操作符的语义：对于操作 path(doc("bib.xml"), //bib, /book)，其语义是，对文档 bib.xml 中的根结点，以该根结点为上下文结点，找到其所有的名称为 bib 的子孙结点，然后对每一个 bib 节点，以其为上下文结点，返回其所有的名称为 book 的孩子结点。

Path 的第 1 个 Step 操作通常是变量或函数。这样第 1 个 Step 操作就无需上下文结点。

Path 的执行，就是从左至右依次执行 Step 操作：对第 i 个 Step 操作的结果集的每一个结点，以该结点为上下文结点，执行第 i+1 的 Step 操作。最后的 Step 操作的结果就是此 Path 的结果。

4. ForVarBind

对应于 XQuery 语法中的 ForClause 语句，将一个变量绑定到一个操作上（最常见的是 Path 操作）；像关系数据库中的游标一样，迭代地指向该操作的结果集中的每一个结点。

5. LetVarBind

对应于 Let 语句，跟变量的 For 绑定类似，将一个变量绑定到一个操作上；但是变量的 Let 绑定，是一次性的绑定到一个集合上，而不是迭代地指向该操作结果集的结点。进行 Let 绑定的变量，通常要进行聚簇操作。

6. FLWR

操作 FLWR 对应于 Xquery 语法中的 FLWRExpr。

输入：有序的 For 和 Let 绑定变量列表；CondTreeNode 操作；构造结果的操作（对应于 FLWR 的 return 子句）

输出：Element Node List

7. EleConstructor

用于构建新的 Element 结点。对应于 XQuery 语法中的 ElementConstructor 表达式和 ComputedElementConstructor 表达式。

输入：新构建结点的名字；产生 Sub-Element 的操作列表；构造属性节点的操作列表。

输出：新构建的 Element Node 列表

例如查询 1 中构造出了两个新的元素结点：minprice 和 price。构造 Minprice 的操作的输入，包括新 Element 的名称“minprice”，一个子操作 EleConstruct（用于产生子结点 price），和一个 AttrConstruct（用于构造属性 title）。

8. AttrConstructor

用于构建新的属性结点。

输入：新构建属性的名字；用于产生属性值的操作，或者属性常量值。

输出：新构建的属性节点

例如查询 1 中 AttrConstruct 操作，构造了 title 属性。输入：操作 Path ($t)，输出：新构建的 title 属性结点

9. BuiltInFun

对应于 Xquery 语法规约到 FunctionCall 表达式的所有的函数，包括聚集操作(count，Max，Min，Avg，Sum 等)，数据源函数（Document，Collection，Input 等），取消重复（DistinctValue，DistinctNode），定位函数（Last，First 等），字符串匹配 Contain 等函数。由于 OrientX 系统目前没有支持用户自定义函数，因此这里的函数调用只是 XQuery 内嵌函数。

输入：函数参数。

输出：函数操作的结果可能是原子类型的值，也可能是结点序列

10. IfThenElse

条件表达式 if <Expr1> then <Expr2> else <Expr3>的语义也很直观：如果表达式 expr1 为真，则返回子操作 expr2 的结果，否则返回子操作 expr3 的结果。

之所以把 IfThenElse 单独分出来当作一个操作，是因为它根据条件满足与否而有不用的操作。这个功能是前面所提到的操作都做不到的。

输入：条件判定操作 CondTreeNode，条件满足与否所要做的操作

输出：Element Node List

11. Quanlify

形式如：(Some | Every) var in Xpath satisfy condition 的量词表达式中的量词，包括存在量词 some 和全称量词 every；以 some <Var> in <Expr1> satisfied <Expr2>为例，其语义是，对表达式 expr1 的结果集中的每一个结点，绑定到变量 var 上，然后进行 expr2 的条件判断。只要 expr1 中有一个结点使得 expr2 表达式为真，则整个量词表达式的结果为真。Quanlify 操作对应着 XQuery 表面语法中的量词表达式。

输入：量词类型；变量名；产生数据集的操作；CondTreeNode 操作

输出：真假值。

12. SetOpt

SetOpt 操作，主要是对两个结点集合进行并（Union），交（Intersect），或者差（Except）操作，对应于 XQuery 语法中的 UnionExpr，IntersectExceptExpr 等操作。

输入：两个子操作数

输出：Element Node List

13. SortBy

SortBy 操作，本来应该是属于 BuiltInFun 的。但是 XQuery 语法中，SortBy 却单独作为一个产生式。

这个操作虽然根据需要已经设计出来了，但是当前版本的 OrientX 系统还没有支持。

4.3 执行计划树的生成

定义了 *XAlgebra* 代数，接下来的主要问题就是如何用它来表示 XQuery，也就是从 XQuery 表面语法生成 *XAlgebra* 所表示的查询执行计划的算法问题。

从前面的介绍可以知道，*XAlgebra* 中的每一个操作符都对应着 XQuery 表面语法中的一个或多个产生式；XQuery 表面语法中的每一个产生式可以用 *XAlgebra* 中的一个或多个操作符迭代表示出来。这些对应关系是**有限确定**的(可以通过*穷尽法*证明)。

因此，在我们的实现中，从 XQuery 表面语法生成 *XAlgebra* 所表示的查询执行计划的算法，是基于 XQuery 语法产生式的。也就是，在对 XQuery 查询进行语法分析时，每当按照某一个语法产生式进行规约，如果该语法产生式对应于某个操作符，就构建新的操作，并把它挂载在相应的语法树结点下；而产生右边的子表达式的相应的子操作，作为新的操作结点的子操作。当查询的语法分析完毕，语法树的根节点所对应的操作树，就是查询的 *XAlgebra* 形式的执行计划。

下面的程序是根据 XQuery 中 if-then-else 表达式的表面语法产生式来生成相应的操作。

```
IfExpr :   If Lpar Expr Rpar Then Expr Else IfExpr    /*  语法产生式  */
{//规约时所做的动作
 //…  其他操作，例如构建语法树等
CxqeIfExpr* p = new CxqeIfExpr( );   //构建条件判断操作符
//读取产生式右边第 3 个语法树结点对应的子操作，并置为新构建的操作的条件
p->SetCondition($3->GetQueryPlan());
//读取产生式右边第 6 个语法树结点对应的子操作，并置为新构建的操作的 then 操作
p->SetThenOpt($6->GetQueryPlan());
//读取产生式右边第 8 个语法树结点对应的子操作，并置为新构建的操作的 else 操作
p->SetElseOpt($8->GetQueryPlan());
$$->SetQueryPlan(p);//把新构建的操作挂载相应的语法树结点($$)下。
//…  其他操作
}
```

4.4 示例

以查询 1 中的 Xquery 查询为例，其用 XAlgebra 来描述的执行计划如图 3 所示。

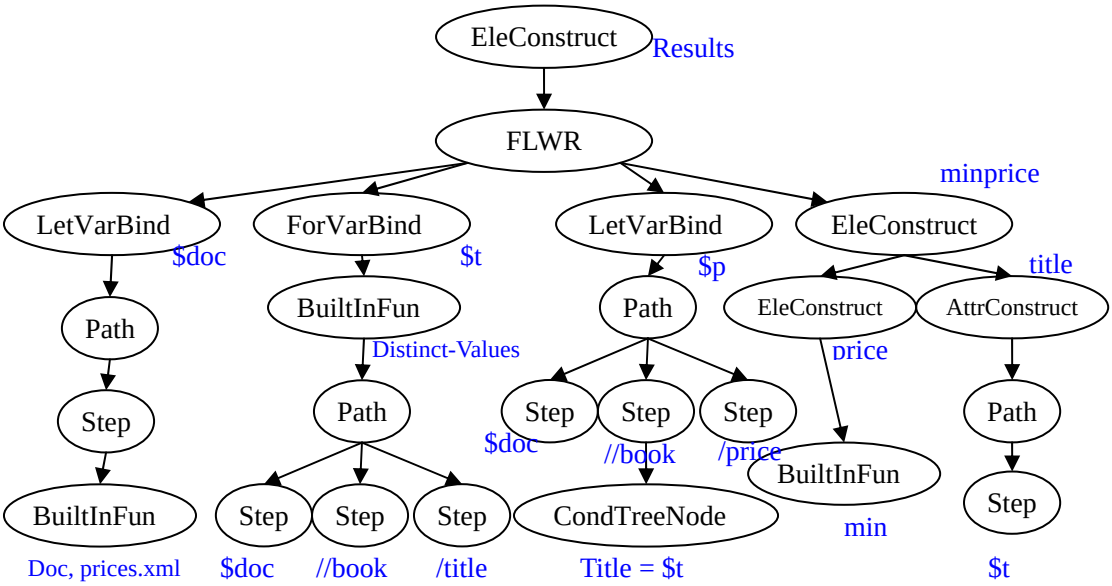

图 3　查询 1 的执行计划树

4. 接口设计

XQuery 处理模块的功能是，接收用户提交的 XQuery 查询，并返回数据检索的结果。因此，XQuery 处理模块的服务，通过如下的接口提供。

CxqeXQueryExecute 类：

1) 初始化查询执行环境。

```
int             initialize();
```

2) 提交并执行查询

XQuery查询用文本形式通过文件inputFile提交。参数outputFile指明输出文件。

```
int             run(const char* inputFile, const char* outputFile);
```

3) 清除查询执行环境。

```
int             clear();
```

5. 主要类的设计

XQuery 导航式处理模块，主要涉及 20 个类，这些类的继承关系如图 4 所示。

- **OXObject** 类是 OrientX 系统中所有类的公共基类；
- **CxqeXAlgebraNode** 类是 XAlgebra 代数 13 个操作类的公共基类；关于 XAlgebra 代数的操作类的前缀，C 表示 Class，xqe 是 XQuery Execute 的缩写，表示该类属于 XQuery 处理模块。
- **CxqaEnvironment** 类用于保存执行引擎的环境，例如变量的当前绑定值等；
- **CComputEleNode** 和 **CComputeAttrNode** 类则是新构建的 Element 和 Attribute 结点的类型。
- **CxqeOperData** 类是对 XQuery 查询引擎所涉及到的数据类型（包括常量 CConstValue，元素 ElementNode，新构建元素结点 CComputEleNode，结点序列 NxdbList）的简单封装，提供统一的接口；
- **NxdbList** 是一个双向链表，其元素为 NxdbListItem 类型。NxdbListItem 通过成员变量 OXObject*指向任意类型的数据对象。一般而言，NxdbList 里所放的数据都是同一类型的。

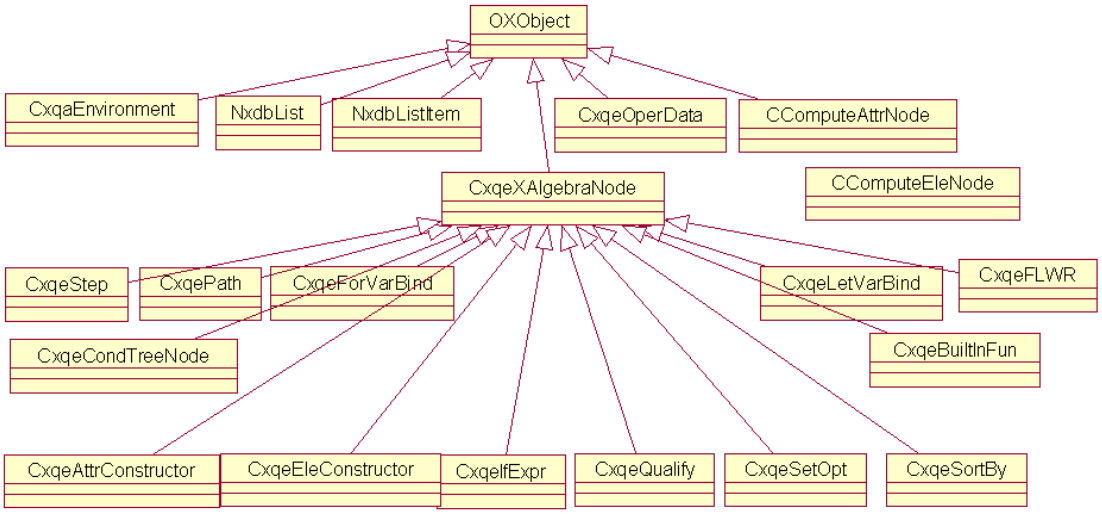

图 4　类关系图

主要的类大概有 10 多个，很难在实现之前详细地设计出所有的方法，数据和接口。

1. CxqeXAlgebraNode 类

CxqeXAlgebraNode 类是 XAlgebra 操作类的公共基类。XAlgebra 操作中有部分可以实现流水线操作（例如 Step，Path 等），有些则不可以，通过函数 toBePipeLine 判断。

进行**流水线操作**时，先打开流水线操作 beginPipeline，然后就可以不断地取值 getNextValue，取完所有值后，调用函数 closePipeline 关闭流水线操作，释放资源。

进行**物化操作**时，先调用函数 initialize 初始化执行环境，然后开始执行物化操作 run，物化操作的结果被存放在成员变量 m_ValueBuffer 中，通过函数 getCurrentData 来依次取值，通过 cursorMoveForward 等函数来设置 m_ValueBuffer 正在活动的数据项。数据被取空之后，就可以调用 clear 函数来释放该操作所占用的资源。

所有继承 CxqeXAlgebraNode 类的操作类，都是提供流水线操作和物化操作两套接口，它们的调用方法如上所示。后面的操作类中，我们将不再重复。

- **主要成员变量**

(1) 用于保存本操作进行物化实现时的结果

```
NxdbList                m_ValueBuffer;
```

(2) 外部要读取成员变量m_ValueBuffer的内容，只能从左至右逐个地取。m_CurItem用于指示m_ValueBuffer中正在活动的数据项。没有活动的数据时，为空（NULL）。

NxdbListItem* m_CurItem;

(3) 流水线操作时，用于保存上下文结点。

CxqeOperData m_Context;

(4) 流水线操作时，用于保存本操作正访问的数据结点。

CxqeOperData m_CurrentData;

(5) 流水线操作时，用于记录已经返回的数据项的个数

int m_ReturnedNum;

(6) 用于记录本操作是什么操作，例如FLWR，或EleConstruct等。

enum XAlgebra m_XAlgebraType;

- **主要成员函数：**

(1) 判断是否能够进行流水线操作。可以则返回true，否则返回false

virtual bool toBePipeLine() = 0 ;

(2) 物化操作时，用于初始化本操作的环境。

virtual int initialize();

(3) 开始物化操作。操作的结果存放在成员变量m_ValueBuffer中

virtual int run() = 0;

(4) 重置m_ValueBuffer的活动结点，指向第1个数据项

virtual int resetCursor();

(5) 把m_ValueBuffer的活动结点指向下一个数据项

virtual int cursorMoveForward();

(6) 删除m_ValueBuffer中正在活动的数据项，并置下一个数据项为活动结点。

virtual int cursorDelAndMoveForward();

(7) 读取m_ValueBuffer中活动结点的**数据值**

virtual int getCurrentData(CxqeOperData& data);

(8) 物化操作结束，清楚操作环境，释放空间。

virtual int clear() ;

(9) 以context为环境结点，打开流水线操作。

virtual int beginPipeline(const CxqeOperData& context);

(10) 以流水线的方式取下一个数据，成功则返回0，否则返回-1

virtual int getNextValue(CxqeOperData& data) = 0;

(11) 关闭流水线操作。

virtual int closePipeline();

(12) 把操作oper的结果合并到本操作中来，即操作oper的成员变量m_ValueBuffer的成员移动到本操作的m_ValueBuffer中来，放在链表的尾部。

virtual int mergeValue(CxqeXAlgebraNode* oper);

(13) 引用成员变量m_ValueBuffer。

NxdbList* getValueBuffer();

(14) 读取本结点的类型。

enum XAlgebra getNodeType();

2. CxqeStep 类

- **主要成员变量**

(1) 谓词列表

```
vector<CxqeCondTreeNode*>      m_PredicatesList;
```

(2) 用于记录从上一个step到达本step的连接类型：/或者//

```
enum StepLinkType m_FromLinkType;
```

(3) 轴的类型。

```
enum AxisType m_Axis;
```

(4) 用于指示本step的类型，例如FunctionCall，CConstValue，Element，Text等

```
enum   xqeStepType     m_StepType;
```

(5) 数据值

```
union{
    CNameTest* NameTest;//名称
    int        VarID;//变量ID
    CConstValue* ConstValue;//指向常量的指针
    CxqeXAlgebraNode* opt;//指向操作结点的指针
}m_Value;
```

(6) Eid列表，根据上下文结点和连接类型， step的名称可以转换成相应的Eid

```
vector<DTDNodeCode>        m_EidList;
```

(7) 记录当前正在使用哪一个Eid进行求值

```
int        m_ActiveEidIndex;
```

3. CxqePath 类

- **主要成员变量**

(1) 路径片段列表，其中路径片段主要是CxqeStep类型。

```
vector<CxqeXAlgebraNode*> m_FraPathList;
```

(2) 流水线操作时，用于记录从左到右初始化了几个路径片段。

```
int        m_BeginPipeLineFraPath;
```

4. CxqeForVarBind 类

(1) For绑定的变量名称

```
char*      m_VarName;
```

(2) 变量ID，系统为查询中的所有变量进行编号。

```
int        m_VarID;
```

(3) 指向被绑定的操作

```
CxqeXAlgebraNode*     m_BoundOperator;
```

5. CxqeLetVarBind 类

(1) Let绑定的变量名称

```
char*   m_VarName;
```

(2) 变量ID，系统为查询中的所有变量进行编号。

```
int        m_VarID;
```

(3) 指向被绑定的操作

CxqeXAlgebraNode* m_BoundOperator;

6. CxqeFLWR 类

(1) For和Let绑定操作列表，从左至右的顺序对应于FLWR语句中从外到内的顺序。

vector<CxqeXAlgebraNode*> m_BoundVarList;

(2) 过滤条件，对应于Where子句

CxqeCondTreeNode* m_CondTree;

(3) 结果构造操作，对应于return子句

CxqeXAlgebraNode* m_ReturnOperator;

(4) 用于指示本操作是否更新的操作。OrientX系统扩展了XQuery使之支持更新功能，更新的操作替代了return操作。

FLWRTYPE m_flwrType;

(5) 指示更新操作的入口

CxqeXAlgebraNode* m_UpdateEntry;

7. CxqeCondTreeNode 类

条件表达式，根据其操作的优先级，可以组成一棵谓词表达式树。例如 $price > $p + 50 or $year < 2000 and $publisher = “Renmin Youdian”，则可以表示成 ($price >($p + 50)) or (($year < 2000)and ($publisher = “Renmin Youdian”))。这条件树上的每一个操作都是 CxqeCondTreeNode 类型。

- **主要成员变量**

(1) 指示本结点的操作类型，例如and，or，Plus，Equal等。需要注意的是，对于条件树上的叶结点，它的值为optAtomic，表示是原子类型结点。

enum OperatorType m_OperatorType;

(2) 当m_OperatorType指示为原子类型结点时，LeafOperator记录叶结点的操作，用于产生操作数。

CxqeXAlgebraNode* m_LeafOperator;

(3) 当m_OperatorType指示为非原子类型结点时，指向左操作结点。

CxqeCondTreeNode* m_LeftChild;

(4) 当m_OperatorType指示为非原子类型结点时，指向右操作结点。如果没有右操作则为空。

CxqeCondTreeNode* m_RightChild;

- **主要成员函数**

(1) 判断是否范围谓词，也就是本结点操作结果为一个整数区域。是则返回true，否则返回false

bool isRangeCond()const;

(2) 判断本操作是否是叶结点且是常量。是则返回true，否则返回false

bool isConstValue()const;

对以本结点为根的子树进行求值。返回布尔值

bool evaluate();

(3) 对以本结点为根的子树进行求值。返回整数区域[from，to]

int evaluate(int& from, int& to);

8. CxqeEleConstructor 类

- **主要成员变量**

(1) 新构建的元素结点名称。

char*　　m_TagName;

(2) 构建属性结点的操作列表，其成员为CxqeAttrConstructor*类型

NxdbList m_ComputeAttrOperList;

(3) 构建元素结点内容的操作列表，其成员为CxqeEleConstructor*类型

NxdbList m_EleContentList;

- **主要成员函数**

9. CxqeAttrConstructor 类

(1) 新构建属性结点的名称

char*　　m_TagName;

(2) 产生属性值的操作

CxqeXAlgebraNode*　　m_ComputeContent;

(3) 属性值，用于保存操作的结果。

CComputeAttrNode*　　m_Value;

10. CComputeEleNode 类

说明：新构建元素结点。继承于 NxdbNode。XQuery 查询中新构建的 Element 结点不同于数据库中现有的 Element 结点，后者用 Eid 来标识，而后者没有 Eid，只有 TagName。而且它们的属性结点的类型也各不相同。

CComputeEleNode 类型结点的左右兄弟，父子结点由基类 NxdbNode 相应的变量来指示。

- **主要成员变量**

(1) 元素标识名

char*　　m_TagName;

(2) 属性节点列表

vector<CComputeAttrNode*> m_AttrList;

11. CComputeAttrNode 类

说明：新构建属性结点的类型。

- **主要成员变量**

(1) 属性名称

char* m_TagName;

(2) 属性值

TextNode m_Value;

12. CxqeOperData 类

说明：XQuery 查询处理模块涉及的数据主要是 4 类：CConstValue，ElementNode，CComputEleNode，NxdbList。为了统一处理，我们把这 4 类数据封装到 CxqeOperData 中，用 void*类型的 m_pValue 指向真正的数据，用枚举类型 m_Type 指明数据到底是何种类型。

由于历史原因，ElementNode 类型的数据，由底层缓冲区 RecordCache 统一管理，负责释放。而，新构建的元素结点，则选择了在查询结束后统一释放。因此 CxqeOperData 的析构函数，只负责释放 CConstValue 类型和 NxdbList*类型的数据对象。

6. 重要的算法

7. 总结和问题

XQuery

7.1 中间结果

XQuery 处理过程中所涉及的来自数据库的 ElementNode 类型数据，由底层缓冲区 RecordCache 来负责管理。上层模块对 ElementNode 类型的数据的访问，都是通过指针指向。而 RecordCache 并不知道有哪些对象正被访问，因此那些数据无法释放！当查询访问过的数据量很大时，RecordCache 就放不下这么多数据，导致内存溢出。这就是中间结果保存的问题！！