摘 要

XQuery是一种查询XML数据源的语言，和SQL类似，但它具有一些高级语言的特性，诸如自定义函数，声明变量等。以XML为实现基础的集成化的数据库，可以使用XQuery作为统一的查询接口，来达到查询数据的目的。

目前，北京工业大学计算机软件与理论实验室正在开发一款XQuery查询引擎GeoQuery。为了便于处理查询和优化查询，课题组设计了一种函数式XML查询语言FXQL，作为中间语言用于描述XQuery查询计划。目前，课题组希望在GeoQuery系统中为FXQL加入静态类型检查，以便在执行FXQL表达式之前，及时发现程序中存在的类型错误，减少因类型错误所带来的开销。同时，静态类型检查为FXQL表达式所标注的类型信息，可以为后续的优化提供依据。

另外，由于XQuery在处理来自XML文档中的节点时，节点的类型信息可以从XML Schema中获取。那么，相应地在对FXQL进行静态类型检查前，就需要引入XML Schema以获得XML节点的类型，增强类型检查的精度。

本文主要研究FXQL静态类型检查方法，以及用于增强类型检查的XML Schema导入方法，详细讨论了设计与实现这两个方法所需要的数据结构和算法，以及为提高实现这两个方法的功能模块的软件质量属性，在实现过程中所采取的相应的技术和方法。

关键字：静态类型检查；XML；XQuery；XML Schema；Schema导入

**Abstract**

XQuery is an XML query language, which is similar with SQL on the part of the usage. However, XQuery possesses some characters of high-level language.For example, it supports user-defined functions and it permits variable declaration. XQuery can be used as a uniform query interface for the XML-based integrated database, thus meeting the needs of querying XML data.

At the moment, the Computer Software and Theory Laboratory in Beijing University of Technology are developing an XQuery engine called GeoQuery. For the convenience of dealing with the query and optimizing the query, our research group designed a simple functional XML query language (we called it FXQL for short.) to describe the XQuery query plan.At present, in order to decrease the expense which is brought by type mistakes, the research group hope to add static type checking to the GeoQuery system, so that the GeoQuery can find type mistakes in the program as early as possible before executing the FXQL expression. At the same time, the type information, annotated by static type checking, can provide the basis for the following optimization operations.

On the other hand, when XQuery is dealing with the XML nodes, the type information of the nodes can be obtained from XML Schema. As a result, importing XML Schema before static type checking for FXQL expression, which gets XML nodes type information to enhance the precision of type checking, is necessary.

This paper mainly discusses how to implement the static type checking for FXQL expression and how to import XML Schema for enhancing the precision of type checking. The paper gives the data structures and algorithms in implementing the Static Type Checking and Schema Importing, and the methods and technologies adopted in enhancing the software quality attributes of the implementation.

**Keywords:** Static Type Checking；XML；XQuery；XML Schema；Schema Importing

目 录

[1 绪 论 1](#_Toc294859217)

[1.1 课题背景 1](#_Toc294859218)

[1.2 研究现状 1](#_Toc294859219)

[1.3 课题研究任务 2](#_Toc294859220)

[1.3.1 问题的提出 2](#_Toc294859221)

[1.3.2 研究任务 3](#_Toc294859222)

[1.4 课题主要工作 4](#_Toc294859223)

[1.5 研究意义 4](#_Toc294859224)

[1.6 本文结构 4](#_Toc294859225)

[2 XML、XQuery和FXQL基础 5](#_Toc294859226)

[2.1 XML 5](#_Toc294859227)

[2.2 XML Schema 6](#_Toc294859228)

[2.3 XQuery 7](#_Toc294859229)

[2.3.1 XQuery类型系统 8](#_Toc294859230)

[2.3.2 XQuery的类型运算 10](#_Toc294859231)

[2.4 FXQL 12](#_Toc294859232)

[3 算法设计 14](#_Toc294859233)

[3.1 静态类型检查的依据 14](#_Toc294859234)

[3.2 静态类型检查算法设计 15](#_Toc294859235)

[3.2.1 输入与输出说明 15](#_Toc294859236)

[3.2.2 基本算法 16](#_Toc294859237)

[3.2.3 算法复杂度 28](#_Toc294859238)

[3.2.4 其他需要说明的问题 29](#_Toc294859239)

[3.3 XML Schema导入算法 30](#_Toc294859240)

[3.3.1 基本算法 30](#_Toc294859241)

[3.3.2 算法复杂度 32](#_Toc294859242)

[3.3.3 递归的处理 33](#_Toc294859243)

[4 静态类型检查模块的设计与实现 35](#_Toc294859244)

[4.1 静态类型检查模块设计 35](#_Toc294859245)

[4.2 数据结构设计 37](#_Toc294859246)

[4.2.1 FXQL类型系统 37](#_Toc294859247)

[4.2.2 Schema导入的数据结构 41](#_Toc294859248)

[4.2.3 表达式相关的数据结构 42](#_Toc294859249)

[4.3 错误处理机制 43](#_Toc294859250)

[4.3.1 静态类型检查的错误处理 44](#_Toc294859251)

[4.3.2 Schema导入的错误处理 45](#_Toc294859252)

[5 测 试 46](#_Toc294859253)

[5.1 功能测试 46](#_Toc294859254)

[5.2 性能测试 46](#_Toc294859255)

[6 结 论 48](#_Toc294859256)

[7 致 谢 49](#_Toc294859257)

[8 主要参考文献 50](#_Toc294859258)

[附录I 测试用例 51](#_Toc294859259)

[附录II Schema导入设计方案 53](#_Toc294859260)

[附录III FXQL静态类型错误整理 60](#_Toc294859261)

[附录IV 原子类型的类设计 61](#_Toc294859262)

[附录V 路径表达式分析处理过程说明 62](#_Toc294859263)

# 绪 论

## 课题背景

随着Internet不断发展，越来越多的人开始在工作、生活中使用Internet，对Internet的需求变得越来越多样，人们迫切寻求一种一致的通信接口，以便能够在网络上更方便地进行数据交换。XML（“eXtensible Markup Language”的缩写），即“可扩展标记语言”的出现，解决了这一问题。

XML具有清晰简单的语法和无歧义的结构，计算机应用程序可以很轻松地阅读、解析XML文件。这使得XML得到了广泛的应用——不仅仅在Internet中经常使用XML来传输信息，文件系统、数据库中的很多数据也是以XML格式保存的。

这又产生了另一个需求：如何在XML文件中，快速地找到我们感兴趣的数据，并将其转换为我们所需要的格式？为解决这一问题，W3C（World Wide Web Consortium，万维网联盟）设计了一种查询语言——XQuery。XQuery允许我们选择感兴趣的XML数据元素，重新组织它们，并按选定的某种结构输出结果。而如何实现XQuery执行引擎，直接影响到XQuery查询的处理速度。因此，如何设计XQuery执行引擎，以便高效地实现XQuery语言已成为业界研究的重点。

## 研究现状

目前，北京工业大学计算机软件与理论实验室正在开发一款XQuery执行引擎GeoQuery系统。GeoQuery系统对于XQuery查询的处理分为两个阶段：静态分析阶段和动态执行阶段。静态分析阶段完成了查询语句静态语义的分析，在这个过程中不断地访问静态上下文来进行查询语句静态信息的存储和获取。其中，静态上下文包含用户自定义函数、当前查询中使用的名称空间、缺省的名称空间等信息。动态执行阶段即查询的执行阶段，使用动态上下文来保存执行的状态信息，供执行引擎访问。

GeoQuery系统主要由语法分析、翻译、程序优化、执行引擎四个功能模块组成，各模块具体的功能如下：

* 语法分析模块：对于输入的XQuery查询语言进行词法、语法分析，生成语法树，并将XQuery查询序言中的内容导入到静态上下文中。
* 翻译模块：将XQuery语法树翻译为相应的FXQL表达式，FXQL语言是查询计划的描述手段，有关FXQL的详细说明参见“ ”。
* 程序优化模块：对FXQL表达式进行优化，以提高处理速度。
* 执行引擎模块：执行FXQL查询语句，查询XML文档，计算出查询结果，并输出。

具体的处理流程如“图 1‑1”所示。



图 1‑1 GeoQuery处理XQuery查询语句的流程

## 课题研究任务

问题的提出

由于XQuery是一种书写格式较为自由的语言，它对程序的书写格式没有严格的位置上的要求，所以，从用户输入的原始查询中分析出用户的查询意图是研究XQuery语言实现的难点所在。为了便于处理查询和优化查询，软件与理论实验室设计了一种函数式语言——FXQL，用于表示XQuery查询计划。用户输入的XQuery查询语句首先将会被翻译成相应的FXQL表达式，再经过程序优化模块处理后，提交给执行引擎执行，输出查询结果。

但目前通过翻译XQuery查询语句所获得的FXQL表达式，还不具有类型信息，而XQuery是一种强类型语言，XQuery语句中的变量、查询所得的结果是具有类型的。这就意味着，只有到了FXQL表达式执行阶段，才能发现原始XQuery查询中所存在的类型错误。因此，软件与理论实验室希望在FXQL表达式提交给执行引擎执行前，对FXQL表达式进行静态类型检查，以提前发现类型错误，减少由类型错误所引起的开销，从而提高GeoQuery系统对XQuery查询的处理效率。

研究任务

本课题主要研究任务是为GeoQuery系统添加上“静态类型检查”模块，以便对FXQL表达式进行类型检查，如“图 1‑2”所示。



图 1‑2 静态类型检查在查询处理流程中的位置

需要说明的是，由于XML节点的类型信息一般保存在XML Schema中，因此，对FXQL表达式进行类型检查的过程中，如果涉及到了XML节点数据，就需要将相应Schema导入到系统中，使用Schema中的类型信息继续类型检查，以增强类型检查的精度。

综上，静态类型检查模块所应具有的功能如下：

1. 基于FXQL类型系统，推导出FXQL语句中每一子表达式的类型，并将其标注到表达式上；
2. 对于无法事先预知其具体类型的XML节点类型（如元素节点、属性节点等），如果提供了节点对应的XML Schema，则将Schema导入，从而获得节点的具体类型，提高类型检查的精度。对于没有提供XML Schema的节点类型，则将其视为缺省类型“anyType”（表示任意类型），继续进行类型检查，并且这时如果碰到必须获得XML节点结构和内容的约束信息才能进行类型检查的情况，则跳过对该节点的检查，继续对FXQL表达式中的其他部分做类型检查。
3. 对于期望类型与实际类型不符合（没有语法错误，但存在类型错误）的FXQL表达式，给出具体的错误。

## 课题主要工作

笔者主要工作内容是：

1. 参考XQuery的类型系统，建立了FXQL类型系统；
2. 在GeoQuery静态分析阶段，对FXQL表达式进行类型检查；
3. 为了提高处理的精度，引入XML Schema，修改了系统中原有的XML Schema导入算法。

## 研究意义

在GeoQuery系统中，在静态分析阶段对FXQL表达式进行类型检查的意义主要有两点：

1. 在静态分析阶段提前发现程序中的类型错误，不用等到执行时再报错，提高系统的处理效率。
2. 为FXQL表达式标注上类型信息，为之后对FXQL表达式的优化提供依据。

## 本文结构

第2章介绍了XML、XQuery、XML Schema的基础知识，并简要介绍了FXQL的相关知识；第3章比较详细地介绍了静态类型检查算法，以及XML Schema导入算法；第4章介绍了静态类型检查和XML Schema导入的数据结构设计和实现方式；第5章介绍系统测试的内容以及测试结果。

# XML、XQuery和FXQL基础

## XML

XML是“eXtensible Markup Language”的缩写，即“可扩展标记语言”，它具有如下特征：

* + XML可以用一种符合用户需求的格式来存储和组织几乎任何信息。
  + XML使用Unicode作为标准字符集，支持大量的书写系统和符号。
  + XML提供了语法规则、内部链接检查、数据类型等多种方法来检验文档的质量。
  + XML具有清晰简单的语法和无歧义的结构，人和程序都能很轻松地阅读、解析。
  + XML易于和样式表组合，以便以任何用户想要的样式来创建格式化文档。

使用XML编写的文档，主要由标记（markup）和内容（content）两部分组成，标记和内容合在一起包含了文档的所有信息——标记是一些添加到文档中的信息，用以标识文档的各个部分以及各部分之间的关系；内容是读者关注的具体信息。“”是一个典型的XML文档：

表 2‑1 一个XML文档store.xml

|  |
| --- |
| <store>  <book year="2000">  <title>Data on the Web</title>  <publisher>Morgan Kaufmann Publishers</publisher>  <price>39.95</price>  </book>  <book year="1999">  <title>  The Economics of Technology and Content for Digital TV  </title>  <publisher>Kluwer Academic Publishers</publisher>  <price>129.95</price>  </book>  </store> |

就“<title>Data on the Web</title>”而言，标记是“<title>”和“</title>”，内容是“Data on the Web”。

之所以需要使用标记，是因为XML文档是要由计算机应用程序来处理的，如果上面提到的XML文档中没有标记，那么，将得到结果如“表 2‑2”所示。

表 2‑2 去掉标记后的store.xml

|  |
| --- |
| 2000  Data on the Web  Abiteboul Serge  39.95  1999  The Economics of Technology and Content for Digital TV  Kluwer Academic Publishers  129.95 |

显然，去掉标记后，计算机应用程序在处理该文档时，根本无法将“Data on the Web”与其他文本区分开。在这里，标记“<title>”的作用，就是为了要告诉XML处理器，该标注所包含的内容“Data on the Web”是一本书的书名。

## XML Schema

XML最强大的特征之一就是允许创建用户自己的标记语言，定义最适合用户想要封装的信息的元素和属性，而不是将用户限制在一种不适合的通用语言之中，XML Schema就是这种定义元素和属性的方法之一。XML Schema是一种特殊的XML文档，与一般XML文档的不同之处在于，它所包含的信息是某一类针对特定应用领域的XML文档在格式方面的要求。开发人员通过使用XML Schema，可以描述一个文档的结构，以及定义元素与属性内容的数据类型。

仍以“表 2‑1”为例，该XML文件对应的XML Schema如“表 2‑3”所示。

XML Schema的主要组成部分如“图 2‑1”所示。其中，开发人员经常使用的是类型定义，元素声明和属性声明。类型定义允许开发人员根据需要，自己定义数据类型；元素声明和属性声明使开发人员能够对元素、属性的名称、所允许包含的内容进行约束。

而XML Schema的主要作用，就是作为判断XML文档是否合法的依据。XML处理器通过参考正在处理的XML文档所对应的Schema，就可以对该XML文档进行语义上的分析和验证。如果一个XML文档不符合Schema的要求，XML处理器将会报错。就“”而言，它的Schema“”要求“book”元素下只能有“title”、“publisher”、“price”三种元素，并且规定了每种元素具体的类型，例如要求“price”元素必须是decimal类型，如果“”中，第一个“book”元素的“price”是一个字符串，那么XML处理器在处理时将会报错。

表 2‑3 store.xml的XML Schema

|  |
| --- |
| <?xml version="1.0"?>  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">  <xsd:element name="book">  <xsd:complexType>  <xsd:sequence>  <xsd:element name="title" type="xsd:string"/>  <xsd:element name="publisher" type="xsd:string"/>  <xsd:element name="price" type="xsd:decimal"/>  </xsd:sequence>  <xsd:attribute name="year" type="xsd:integer"></xsd:attribute>  </xsd:complexType>  </xsd:element>  <xsd:element name="store">  <xsd:complexType>  <xsd:sequence>  <xsd:element ref="book"></xsd:element>  </xsd:sequence>  </xsd:complexType>  </xsd:element>  </xsd:schema> |



图 2‑1 XML Schema的层次结构

## XQuery

XQuery和SQL类似，是一种查询语言，不同之处在于，SQL的查询对象是关系数据，返回的查询结果是关系数据；而XQuery的查询对象是XML数据，返回的查询结果也是XML数据。因此，XQuery也被人们称为“XML的SQL”。并且，XQuery还具有一些高级语言的特性，比如说它支持用户自定义函数。

XQuery查询语句主要由两部分组成：序言和主体。序言包含有各种不同的声明，如命名空间声明、XML Schema的导入、变量声明、函数声明等；查询主体则包含了具体的查询语句。

### XQuery类型系统

XQuery是一种强类型语言，每个变量、表达式都有具体的类型。这使得开发人员可以在XQuery查询语句执行前，对其进行静态类型检查，在查询中较早地标识出类型错误。XQuery采用了XML Schema的类型系统，包括表示常见数据类型的内置类型，例如整数类型，浮点数类型，日期类型以及字符串类型等等。此外，由于XML Schema允许开发人员基于内置类型，定义适合自己的数据类型，因此XQuery的类型系统中还包括用户自定义类型。XQuery类型系统的层次结构如“”所示，从该图可以看出，XQuery类型大致可以分为以下4大类：原子类型，节点类型，序列类型和用户自定义类型。以下将逐一介绍。

**（1）原子类型**

XQuery中内置了40多种原子类型，它们是XQuery类型计算的基础。一个原子类型既可以单独使用，也可以与其他类型组合，生成更为复杂的类型。例如，若一个变量var的值为“32”，那么它的类型就是“integer”，这是原子类型单独使用的情况；“（integer, double）”则表示一个由“integer”和“double”类型组成的序列类型（有关“序列类型”的说明将在之后给出）。

**（2）节点类型**

XQuery中有7种节点类型：文档节点，元素节点，属性节点，文本节点，注释节点，处理指令节点，命名空间节点。这与XML数据模型中规定的节点类型是一一对应的。其中，开发人员经常使用到的是元素节点和属性节点，任何一个元素节点由名称、属性（可以有多个）和内容（可以是简单类型，也可以是多个子节点）三部分组成，而一个属性节点则由名称、类型两部分构成。就以“”为例，其中名称为“book”的元素节点就包含有一个“year”属性，以及名为“title”、“publisher”的两个元素节点。

可以看出，节点的结构是较为复杂的，要想对节点类型进行精确的类型检查，就必须知道该节点具体的结构是什么、对数据有什么要求，而这些信息一般保存在XML Schema中——XML Schema规定了XML文档的组织结构，规定了对节点的结构以及内容的约束信息。这也就意味着，要想对XQuery查询语句中出现的XML节点进行较为精确的类型检查，就必须找到相应的XML Schema，并将其导入，作为类型检查的依据。

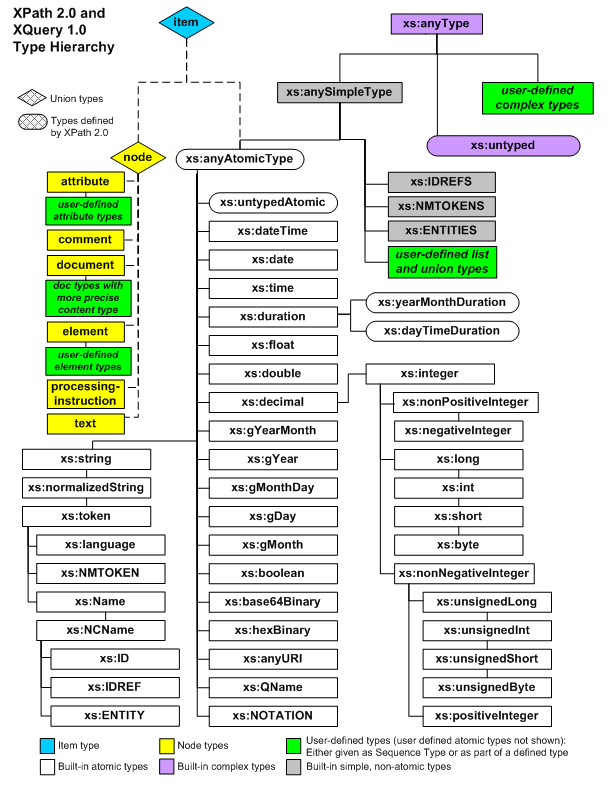


图 2‑2 XQuery类型系统结构图5

**（3）序列类型**

序列类型由两部分组成，一部分是它具体的类型，另一部分是具体类型出现的次数，它的语法规则如“表 2‑4”所示6，这表明序列类型是由项类型（ItemType，XQuery数据模型中对原子类型、节点类型的统称）通过逗号运算符和括号表达式的组合构成的，并且通过出现描述符“？”、“+”和“\*”表示该序列类型的出现次数。

表 2‑4 序列类型语法规则

|  |
| --- |
| SequenceType := ItemType |  (SequenceType（，SequenceType）?)·OccurrenceIndicator?  OccurrenceIndicator := “？”|“+”|“\*” |

**（4）用户自定义类型**

用户自定义类型是使用XML Schema来定义的，它所保存的信息实际上就是对节点类型结构、内容的约束信息。例如“”中，就包含了两个用户自定义类型：“store”元素节点，“book”元素节点。

正如之前所提到的那样，只有进行了Schema导入操作，才能进行高精度的静态类型检查。如果XQuery程序所涉及的XML Schema没有导入，那么XQuery处理器将该节点当做缺省类型处理，之后，继续对其他查询语句进行类型检查。

### XQuery的类型运算

**（1）类型运算“|”**

“|”(“并”运算)是XQuery类型计算中，经常被用到一种运算，在XQuery形式语义中，其具体定义如“式 2‑1”所示。

式 2‑1

其中prime定义如“表 2‑5”所示，quantifier的定义如“表 2‑6”所示。

表 2‑5 prime运算定义

|  |
| --- |
| prime(ItemType) = ItemType  prime(empty) = empty  prime(Type1 | Type2) = (Type1 | Type2)  prime(Type1 | empty) = Type1  prime(Type1 | anyType) = anyType  prime(Type ?) = prime(Type)  prime(Type \*) = prime(Type)  prime(Type +) = prime(Type) |

表 2‑6 quantifier运算定义

|  |
| --- |
| quantifier(empty) = 1  quantifier(Type1 | Type2) =  quantifier(Type1) | quantifier(Type2)  quantifier(Type ?) = quantifier(Type)·?  quantifier(Type \*) = quantifier(Type)·\*  quantifier(Type +) = quantifier(Type)·+ |

其中，“empty”表示“空”类型，“anyType”表示任意类型（本文的其他部分还将使用这种表示，没有给出特殊说明的情况下，“empty”、“anyType”的含义与此处相同）。“prime”是关于其具体类型的运算，而“quantifier”是关于其出现次数的运算，“quantifier”定义中出现的“1”、“？”、“|”、“\*”被称为“OccurenceIndicator”，即“出现描述符”，这些“出现描述符”的“|”运算规则如“”所示。

表 2‑7 OccurenceIndicator的“|”运算规则

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| “|”运算 | 1 | ? | + | \* |
| 1 | 1 | ? | + | \* |
| ? | ? | ? | \* | \* |
| + | + | \* | + | \* |
| \* | \* | \* | \* | \* |

**“|”运算下的零元和幺元：**

根据“|”运算的定义，可以推出“empty”是“|”运算下的幺元，而“anyType”是“|”运算下的零元。

“|”是类型检查中最常用的运算，这个运算下的零元和幺元将在静态类型检查的一些特殊情形中扮演重要角色。

**OccurrenceIndicator的偏序关系：**

OccurrenceIndicator有四种数值可取，分别是“1”，“？”，“+”，“\*”。分别表明出现“一次”，“零次或一次”，“一次或多次”，“零次或多次”。

它们之间的偏序关系如下：

式 2‑2

式 2‑3

式 2‑4

式 2‑5

它们是其他一些常用类型计算的重要理论依据。

**（2）其他常用的类型运算**

在XQuery中，其他常用的类型运算如“表 2‑8”所示。

表 2‑8 XQuery常用类型运算

|  |
| --- |
| boolean isSameType(Type t1, Type t2)： 判断两类型是否为同一类型  boolean isSubType(Type t1, Type t2)： 判断t1是否为t2的子类型  boolean promoteTo(Type t1, Type t2)： 判断t1能否提升为t2  boolean castAllowed(Type t1, Type t2)： 判断t1能否转化为t2 |

## FXQL

在“ ”中已经提到过，GeoQuery系统内部使用课题组自行设计的FXQL语言作为XQuery查询的抽象表示。该语言能够精确地描述执行动作，同时为基于查询重写和代价的优化留有空间。

FXQL语言的特色主要表现在以下几个方面：

* 是一种函数式语言，无副作用；
* 支持函数和递归函数的定义和调用；
* 支持函数作为参数；
* 是一种XML查询语言，提供了丰富的原语函数，来完成各类数据处理和程序逻辑控制的任务。

FXQL的抽象语法域如“”所示，而FXQL的语法规则如“”所示。

表 2‑9 FXQL抽象语法域

|  |  |
| --- | --- |
| 标识符 | 表示 |
| *Prog* | FXQL程序 |
| *Exp* | 表达式 |
| *Func* | 函数定义 |
| *Arg* | 实参表达式 |
| *Step* | XPath查询步 |
| *Id* | 变量 |
| *Const* | 常数 |
| *Axis* | 轴操作 |
| *NodeTest* | 节点测试（名称测试或类型测试） |

从FXQL语法规则不难看出，FXQL表达式的种类远远少于XQuery表达式的种类，在这种情况下，为了使FXQL能够实现XQuery语言提供的诸多查询功能，在FXQL中以内置函数的形式，针对各种不同的功能设计了相应的原语，这些内置函数就被称之为“原语函数”。在FXQL中，原语函数扮演着十分重要的角色，FXQL中大部分的操作都是使用原语函数来完成的，比如说XQuery中的算术加法操作“1 + 2”，在FXQL中将使用“plus(1, 2)”来表示，其中“plus()”就是FXQL中用于表示算术加法的原语函数。对于XQuery的内置函数，FXQL中也是以原语函数的形式实现的，比如说XQuery中用于返回参数绝对值的“abs()”函数，FXQL中将使用原语函数“abs()”来表示。

由于XQuery查询语句可以使用FXQL来表示，通过翻译得到的FXQL表达式与原本的XQuery查询语句的语义是一致的，那么相应地，W3C制定的用于说明XQuery语义的形式语义规范，可以直接应用于FXQL。比如说XQuery中的算术加法运算符“+”，XQuery形式语义规范要求“+”的操作数必须有两个，并且两操作数必须是数值类型，根据操作数类型，按照该规范中给出的类型计算规则进行推导，所得到的结果类型也是数值类型。那么，相应地，FXQL中的“plus()”就要求其参数个数也只能是两个，并且其参数只能是数值类型，并且直接采用XQuery形式语义规范中“+”的计算规则来推导结果的类型，函数返回的结果也必须是数值类型。（注：下面还将多次使用“plus()”这一原语函数举例，与此处相同，都表示用于算术加法运算的FXQL原语函数。）

因此，在对FXQL进行类型检查时，可以使用XQuery形式语义作为类型推导和类型检查的参考依据。

对于FXQL更加详细的介绍请参考文献[,]。

表 2‑10 FXQL语法规则

|  |  |
| --- | --- |
| 语法规则 | 语义说明 |
| *<Prog>* **→** *<Exp>* | FXQL程序 |
| *<Prog>* **→** *<Exp>*  where *Id := fun(Id\*)<Exp>* | FXQL程序 |
| *<Exp>* **→** *Id* | 变量 |
| *<Exp>* **→** *Const* | 常数 |
| *<Exp>* **→** *Axis(<Exp>, NodeTest)* | 节点检测和轴操作 |
| *<Exp>* **→**if *<Exp>*  then *<Exp>* else *<Exp>* | 条件表达式 |
| *<Exp>* **→** *Id( <Arg>…<Arg> )* | 函数调用表达式 |
| *<Exp>* **→** *<Exp>* where *Id := <Exp>* | 具有局部定义的表达式 |
| *<Exp>* **→** *Id <Step>+* | 路径表达式（多个查询步） |
| *<Step>* **→** *Axis <NodeTest> [<Arg>]\** | 查询步（具有多个谓词） |
| *<Arg>* **→** *<Exp>* | 实参表达式 |
| *<Arg>* **→** *fun(Id\*)<Exp>* | 匿名函数（实参） |

# 算法设计

## 静态类型检查的依据

在“ ”中已经提到过，由于FXQL是用于描述XQuery的一种中间语言，一个FXQL表达式所表达的语义与它所对应的XQuery查询语句是一致的，因此可以直接使用XQuery形式语义规范来描述FXQL表达式的语义。而该规范中给出了对XQuery查询语句进行静态类型检查的形式化描述，如何推导一个表达式的类型、对每种表达式的类型有什么要求、什么情况下报错以及应该给出何种错误信息，在该规范中都有详细的描述。因此，要对FXQL表达式进行静态类型检查，只需要根据其对应的XQuery查询语句，在XQuery形式语义规范中找到相应的静态类型检查规则，根据这些规则对FXQL表达式进行检查即可。

下面以条件表达式为例，说明如何根据XQuery形式语义规范，对FXQL表达式进行类型检查。

表 3‑1 条件表达式的静态类型检查形式化描述

|  |
| --- |
| statEnv |- Expr1 : xs:Boolean  statEnv |- Expr2 : Type2  statEnv |- Expr3 : Type3  -------------------------------------------------------------------------------  statEnv |- if (Expr1) then Expr2 else Expr3 : (Type2 | Type3) |

XQuery形式语义规范中，条件表达式“if then else ”的静态类型检查规则的形式化描述如“”所示，其中的“statEnv”、“|-”等符号的具体含义在XQuery形式语义规范中有详细描述，这里不做过多说明，而“（Type2 | Type3）”中的“|”，即为“ ”中所提到的“|”类型运算。而“”所描述的检查规则如“”所述。

表 3‑2 XQuery条件表达式的类型检查规则

|  |
| --- |
| 1. 计算Expr1的类型，并且Expr1的类型只能是boolean类型，如果不是则报错； 2. 计算出Expr2的类型Type2； 3. 计算出Expr3的类型Type3； 4. 整个条件表达式的类型就是Type2和Type3进行“|”运算的结果。 |

那么，参考“”的检查规则，对FXQL表达式“if then else ”进行类型检查的算法很容易得出，具体算法参见“”。

表 3‑3 FXQL条件表达式的类型检查算法

|  |
| --- |
| 1. 对进行类型检查，获得该表达式的类型type1； 2. 判断type1的类型是否为“Boolean”类型（FXQL类型系统中“boolean”类型），如果不是则报错； 3. 对进行类型检查，获得该表达式的类型type2； 4. 对进行类型检查，获得该表达式的类型type3； 5. “type2 | type3”的运算结果即为整个FXQL表达式的类型。 |

XQuery形式语义中，各种XQuery表达式的静态类型检查过程的描述方式，与“表 3‑1”是相同的，显然，根据这种形式化的描述方式，很容易得到对应的FXQL表达式的静态类型检查算法。我们只需要知道要进行类型检查的FXQL表达式，所对应的XQuery语句是什么，然后在XQuery形式语义中找到这些语句的类型检查描述，就可以得到对这些FXQL表达式进行静态类型检查的算法。

## 静态类型检查算法设计

静态类型检查模块对于FXQL表达式的处理包含以下两方面内容：

* + 类型推导：根据FXQL表达式的内容，推导出表达式的类型，并将类型信息标注到表达式上。
  + 类型检查：检查FXQL表达式中是否存在类型错误。

下面将给出静态类型检查的输入、输出说明，以及算法描述。

### 输入与输出说明

**输入：**

静态类型检查模块的输入包括四部分内容：待处理的FXQL表达式，变量映射表，自定义函数映射表，XML Schema映射表。下面逐一给出简要说明。

1. **待处理的FXQL表达式**

通过翻译XQuery查询语句所获得的FXQL表达式，是静态类型检查模块的主要处理对象。FXQL的抽象语法域如“”所示，而该模块所要处理的FXQL表达式的语法规则如“”所示。

从可以看出，静态类型检查模块所要处理的FXQL表达式，都是由“”中的常量表达式、变量表达式、路径表达式、函数调用表达式等这些基本的表达式推导得出的，因此，以只要能够对“”中所有的表达式进行处理，那么也就可以对任何FXQL表达式进行类型检查。

1. **变量映射表**

该表保存FXQL表达式中变量的类型信息，保存“<变量名 → 变量类型>”这一映射关系。

1. **自定义函数映射表**

该表保存XQuery序言中的用户自定义函数所对应的FXQL表达式，保存“<自定义函数名 → 函数定义>”这一映射关系，其中“函数定义”包括使用FXQL表示的自定义函数的参数类型声明、函数体、返回类型声明等信息。当对自定义函数的函数调用进行类型检查时，使用函数名从该表获得函数定义，通过比较形参与实参的类型、使用函数体计算结果类型、比较返回类型声明与计算所得到的结果类型等步骤，即可完成对这一函数调用的类型检查。

1. **XML Schema映射表**

该表保存当前系统中已导入的Schema所包含的类型信息。

**输出：**

静态类型检查模块的输出是带有类型信息、通过类型检查的FXQL表达式。

### 基本算法

任何FXQL表达式都是可以由“”中所给出的基本表达式推导而来，因此首先给出对“”中各种表达式的主要检查过程，之后将给出形式化的算法描述。

为方便说明，首先对静态类型检查的各种输入（参见“”）做如下约定：

表 3‑4 静态类型检查输入预定义

|  |  |
| --- | --- |
| 名称 | 说明 |
| *e* | 当前正处理的FXQL表达式 |
| *v* | 变量映射表 |
| *w* | 自定义函数映射表 |
| *u* | XML Schema映射表 |

对“表 2‑10”中各种表达式的基本处理过程如下：

1. **常量表达式（*<Exp>* → *Id*）**

直接根据常量的值，判断出常量的类型，并将该类型返回。

1. **变量表达式（*<Exp>* → *Const*）**

根据变量名，在变量映射表*v*中查找是否存在相应的映射，若存在，则为该变量表达式标注上相应类型，然后返回；若不存在则抛出相应的类型错误。

1. **条件表达式（*<Exp>* → if *<Exp>* then *<Exp>* else *<Exp>*）**

首先对该表达式中的逻辑判断子句进行静态类型检查，看其是否为“boolean”类型，如果不是，则报错；如果是“boolean”类型，则继续对“then”子句和“else”子句进行类型检查，最后将“then”子句和“else”子句的类型做“|”运算（参见“2.3.2 ”），运算结果作为该表达式的结果类型，将其标注到条件表达式上。

例如，假设现在对“表 3‑5”中的FXQL表达式进行类型检查。

表 3‑5 FXQL条件表达式

|  |
| --- |
| if plus(3, 2)  then plus(4, 3)  else 6.0 |

首先会对子表达式“plus(3, 2)”进行类型检查，显然，这个子表达式的结果类型为数值类型，不是“boolean”类型，发生了类型错误，应停止检查，报出相应的错误信息。

1. **函数定义表达式**

在XQuery中，用户可以在序言部分声明自定义函数，经过“翻译”模块处理后，这些自定义函数也会被翻译为相应的FXQL表达式。值得注意的是，XQuery中，自定义函数的参数类型声明和返回类型声明都不是必须的，那么，相应地，FXQL中的自定义函数声明也就不一定包含有参数类型声明和返回类型声明。

在“”中，函数定义表达式都是作为where表达式的一部分出现的（“*<Prog>* **→** *<Exp>* where *Id := fun(Id\*)<Exp>*”中，“where”关键字后面的“*Id := fun(Id\*)<Exp>*”即表示函数定义表达式）。对自定义函数的静态类型检查包含两部分内容：对函数定义进行类型检查，对自定义函数的函数调用进行类型检查。之所以对函数定义进行类型检查，主要是基于两方面的考虑：

* 可以提早发现函数定义中存在的类型错误。
* 在对自定义函数的函数调用进行类型检查时，如果传递给函数调用的实参类型与函数定义中声明的参数类型完全一致，那么就可以直接使用之前对函数定义进行类型检查所得到的结果，这样减少静态类型检查的计算量，提高处理速度。

下面详细介绍如何对自定义函数进行静态类型检查：

1. **对函数定义的类型检查**

**情形1:形参没有类型声明：**

如果函数定义中，没有声明参数的类型，那么跳过类型检查，直接将“函数名 → 函数定义”加入到*w*映射表中，等到要处理该函数的函数调用时，再进行类型检查；

**情形2:形参有类型声明：**

如果函数定义中声明了参数类型，那么将声明的类型代入，对函数体进行类型检查，参考XQuery形式语义给出的静态类型检查规则，推导出结果类型，并将该结果保存起来，以便在函数调用中使用。如果函数定义中声明了返回类型，还需要把推导出的结果类型和声明的返回类型做比较，看是否存在类型错误。

1. **对自定义函数的函数调用进行类型检查**

首先需要对传递给该函数的各个实参进行类型检查，以获取实参的类型，接着看实参个数与形参个数是否一致，之后的检查视情况而定：

**情形1：函数定义中声明了参数类型**

如果函数定义中声明了参数类型，那么逐个比较实参类型和对应的类型声明，看是否存在类型错误，然后将实参类型代入到函数体中，计算结果的类型；

**情形2：函数定义没有声明参数类型**

如果函数定义没有声明参数类型，则直接将实参类型代入函数体，计算结果的类型。

另外，不管是上面哪种情形，如果函数定义中声明了返回类型，都需要把计算出的结果类型和声明的返回类型做比较，看是否存在类型错误。

除此之外，正如之前所提到的，当函数调用中，实参的类型与自定义函数声明的参数类型完全一致时，可以跳过对这个函数调用的类型检查，直接使用函数定义的检查结果。

例如，假设要对“表 3‑6”中的FXQL表达式进行类型检查。

表 3‑6 一个含有自定义函数的FXQL表达式

|  |
| --- |
| plus\_func($var1 as decimal, $var2 as decimal) as decimal{  plus($var1, $var2)  }  plus\_func(2, “a”) |

该表达式中包含两部分内容：自定义函数plus\_func的函数定义，对plus\_func函数的函数调用。那么，首先对函数定义进行检查：参数声明中两个参数的类型均为“decimal”，因此将其直接代入到函数体中，由于“plus()”是加法操作，根据XQuery形式语义，两个“decimal”类型的数相加，其结果也是“decimal”类型，而这一推导结果与函数定义中声明的返回类型一致，因此，plus\_func函数定义表达式的类型就是“decimal”类型，将其标注到函数定义表达式上（注意，这个“decimal”是经过类型推导计算得出的结果类型，是要标注到函数定义表达式上的类型，并不是返回类型声明中的“decimal”，二者是分开保存的。）。

然后对该表达式中的函数调用进行类型检查：实参分别是“2”、“a”，分别为“integer”类型、“string”类型。而在FXQL类型体系中，“integer”是“decimal”的子类型（参见“ ”），因此第一个参数的类型是合法的。第二个参数为“string”类型，不是“decimal”类型，与函数定义中的类型声明不符，则应该停止类型检查，报出错误信息。

1. **函数调用表达式（*<Exp>* → *Id( <Arg>…<Arg> )*）**

FXQL中，函数调用分为两种：对自定义函数的函数调用，对原语函数的函数调用。对于前者的类型检查，在“（4）函数定义表达式”中已经阐明，这里不再赘述；对于原语函数的函数调用，其静态类型检查处理思路比较简单：首先对实参进行静态类型检查，获得实参的类型，然后根据所调用的原语函数的具体作用，在XQuery形式语义规范中找到对应于该原语函数的XQuery语义要求和类型计算规则，检查实参类型是否满足这些语义要求，如果满足，那么按照规范中的类型计算规则，计算出结果类型，并返回；如果实参类型不满足这些语义要求，则报错。

例如，要对FXQL表达式“plus(1，‘a’)”进行类型检查，具体过程如下：

1. 对实参进行类型检查：两个实参都是常量，类型分别为“integer”、“string”。
2. 找出对应的XQuery语义：“plus()”原语函数对应于XQuery中的算术加法运算。
3. 进行类型检查：首先看实参是否语义要求，显然，第二个实参的类型“string”不满足算术加法对参数的要求，检查出类型错误，报出相应的错误信息。
4. **where表达式**

从“”可以看出，where表达式主要由两部分组成，即“where”关键字后面的绑定部分和“where”关键字前面的表达式主体，并且绑定部分可以是变量绑定，也可以是自定义函数绑定。因此，这里分情况讨论：

1. **绑定部分为变量绑定**

**（*<Exp>* → *<Exp>* where *Id := <Exp>*）**

当绑定部分为变量绑定（“*Id* := *<Exp>*”）时，首先对绑定中的子表达式“*<Exp>*”进行类型检查，“*<Exp>*”的类型就是变量表达式“*Id*”的类型。在推导出“*<Exp>*”的类型后，将该类型标注到变量表达式“*Id*”上，并将 “*Id*变量名 **→** *<Exp>*类型”这一映射关系保存到变量映射表*v*中。

1. **绑定部分为自定义函数绑定**

**（*<Prog>* → *<Exp>* where *Id := fun(Id\*)<Exp>*）**

当绑定部分为自定义函数绑定（“*Id* := *fun(Id\*)<Exp>*”，其中的“*Id*”即为自定义函数的函数名，“*Id\**”表示参数声明列表，“*<Exp>*”表示函数体）时，首先对自定义函数的函数定义“*fun(Id\*)<Exp>*”进行类型检查，具体的处理方式之前已经提到过（参考“（4）函数定义表达式”），在对函数定义的类型检查结束后，将映射“<函数名 🡪 函数定义>”保存到函数定义映射表*w*中。

在处理完绑定部分后，再处理“where”关键字前面的表达式主体，即可完成对where表达式的类型检查。例如，假设要对“”中的FXQL表达式进行类型检查。

表 3‑7 一个FXQL where表达式

|  |
| --- |
| plus(var, “a”) where var = plus(4, 2) |

可以看出，这是一个绑定部分为变量绑定的where表达式。那么，首先对绑定中的子表达式“plus(4, 2)”进行类型检查，得到该表达式的类型“integer”，然后将“integer”类型标注到变量表达式“var”上，并将映射“<var **→** integer>”保存到变量映射表*v*中，之后再对“plus(var, “a”)”进行类型检查：由于“var”是一个变量，那么就需要到*v*中查找该变量的类型信息，而之前在处理绑定部分的时候，已经将“<var **→** integer>”保存到了*v*中，因此这时就可以获得“var”的类型“integer”，再对另一个参数“a”进行检查，发现它是“string”类型，不满足“plus()”对参数类型的要求，因此停止检查，报出错误信息。这样就完成了对这个where表达式的类型检查。

1. **轴操作与节点测试（*<Exp>* → *Axis(<Exp>, NodeTest)*）**

首先对要进行轴操作与节点测试的FXQL表达式“<Exp>”进行类型检查，获得其具体的类型，接着，根据XQuery形式语义中轴操作的语义，计算出该节点类型进行轴操作所得的结果类型。之后，对轴操作的结果进行节点测试（包括名称测试和类型测试），获得最终结果。

1. **路径表达式**

**（*<Exp>* → *Id <Step>+*，*<Step>* → *Axis <NodeTest> [<Arg>]\**）**

对路径表达式进行静态类型检查的处理步骤大致如下：

1. 从上下文中获取路径表达式需要处理的节点类型。
2. 根据路径表达式中的每一“步”（“*Step*”）所包含的轴操作（“*Axis*”）和节点测试（“*NodeTest*”），计算出结果类型。例如，对于“a / b”这个路径表达式，如果此时的“a”节点下有一个孩子节点“b”，则从“a”中取出这个节点，作为结果类型返回；如果节点“a”下面没有“b”孩子，则返回“empty”，表示没有任何结果。具体算法描述参见“”。

需要说明的是，路径表达式中的每一“步”中都可能包含有谓词表达式（参见“”），但就静态类型检查而言，它并不影响路径表达式的类型结果。因此在处理每一“步”时，只需要单独处理谓词，不必根据谓词对每“步”的类型结果进行“筛选”。有关路径表达式更详细的分析过程请参见“”，这里不再举例说明。

1. **实参表达式（*<Arg>* → *<Exp>*）**

直接对实参所包含的“参数内容”——“Exp”进行类型检查，但是在检查完“Exp”、为其标注上类型信息后，还需要将“Exp”的类型信息标注到实参表达式“Arg”上。

1. **匿名函数表达式（*<Arg>* → *fun(Id\*)<Exp>*）**

首先对匿名函数的每个参数（“*Id\**”）进行类型检查，将所得结果保存到变量映射表*v*中。然后，对匿名函数的函数体（“*Exp*”）进行类型检查，并计算出的结果类型标注到匿名函数表达式“Arg”上。

1. **FXQL程序（*<Prog>* → *<Exp>*）**

首先对“FXQL程序”所包含的表达式“*Exp*”进行类型检查，并且在检查结束后，需要将“*Exp*”的类型信息标注到“*Prog*”上。

静态类型检查算法的形式化描述如“”所示。需要说明的是，“*Prog*”、“*Arg*”本质上仍是FXQL表达式（有关“*Prog*”、“*Arg*”的说明参见“”、“”。），因此也可以作为静态类型检查算法的输入。

表 3‑8 静态类型检查算法doTypeCheck

|  |
| --- |
| **算法**：doTypeCheck（*e/ Prog / Arg, v, w, u*）  **输入**：  *e/ Prog / Arg* FXQL表达式。  *v* 映射表（变量 **→** 类型）。  *w* 映射表（函数名 **→** 参数及其类型，主体表达式，返回值类型）。  *u* 映射表（文件名 **→** XML Schema)。  **输出**：*Type* 表达式*e*（或是*Prog、Arg*）的类型。  **功能**：为FXQL表达式标注类型， 并检查表达式中是否存在类型错误。 |
| **//（1）处理常量表达式**  If *e* **→** *Const* Then  *e’* = annotate( *e*, type(*Const*) ) //为表达式e标注上类型  Return type( *e’*) //返回常量的类型  End If  **//（2）处理变量**  If *e* **→** *Id* Then  *e’* = annotate( *e*, type( *v*[ *Id* ] ) )  //从映射表*v*中查找变量*Id*的类型信息，标注到表达式e上  Return type( *e’*) //返回变量的类型  End If  **//（3）处理条件表达式**  If *e* **→** if *e1* then *e2* else *e3* Then  *t1* = doTypeCheck(*e1*) //首先检查逻辑判断子句是否为boolean类型  If match( type(*e1*), Boolean) Then  doTypeCheck(*e2, v, w, u*) //计算then子句的类型  doTypeCheck(*e3, v, w, u*) //计算else子句的类型  *t* = getUnionType(type(*e2*), type(*e3*)) //then、else子句的类型做“|”运算  annotate(*e, t*) //将“|”运算的结果标注到条件表达式上  Return *t*  Else  Return *error*  End If  End If  **//（4）处理函数调用表达式**  If *e* **→** *Id( e1, ..., en )* Then  *type1* = doTypeCheck(*e1, v, w, u*)  annotate(*e1, type1*)  …  *typen* = doTypeCheck(*en, v, w, u*)  annotate(*en, typen*) //对所有实参进行类型检查  If match( type(*e1*), *t1* ) and …, and match(type(*en*), *tn*) Then  //如果实参类型符合函数对参数类型的要求，则对函数体进行类型检查  *bodyType* = doTypeCheck(*body, v’, w, u*)  annotate(*body, bodyType*)  Where //对函数体进行类型检查前，先将实参类型保存到*v*中，得到*v’*  *v'* = < *x1* **→** type(*e1*) >, ... , < *xn* **→** type(*en*) >  Return *bodyType*  Else  Return *error*  Where //对函数调用进行检查的前提是函数映射表*w*中有该函数的定义  *w[Id]* = <*x1:t1, ..., xn:tn, body*>  End If  End If  **//（5）处理where表达式：绑定部分为变量绑定的情况**  If *e* **→** *e1* where *Id* := *e2* Then  *t2* = doTypeCheck( *e2 , v, w, u*)  *v’* = addVar(*v , Id , t2*) //将<*Id🡪t2*>保存到变量映射表*v*，获得*v’*  *t1* = doTypeCheck(*e1, v’, w, u*)  annotate(*e, t1*)  Return *t1*  End If    **//（6）处理where表达式：绑定部分为函数绑定的情况（包括了对自定义函数的处理）**  If *Prog* **→** *e1* where *FuncId ‘:=’ fun(Id\*)e2* Then  *type1* = doTypeCheck(*Id1, v, w, u*)  …  *typen* = doTypeCheck(*Idn, v, w, u*) //获得形式参数的类型  *funcType* = doTypeCheck(*e2, v’, w, u*) //对函数体进行类型检查  *w’* = addNewFunc(*w, FuncId, <Id1:type1, ..., Idn:typen>, e2, funcType*)  *progType* = doTypeCheck(*e1, v’, w’, u*)  annotate(*e1, progType*)  annotate(*Prog, progType*) //将结果类型标注到*Prog*上  Return *progType*  End If  **//（7）处理轴操作和节点测试**  If *e* **→** *Axis(e1, NodeTest)* Then  *t1* = doTypeCheck(*e1, v, w, u*)  *e1’* = annotate(*e1, t1*)  If match(*t1, Node*) Then //只有节点类型才可以做轴操作和节点测试  *t2* = doAxisAndNodeTest( *t1, Axis, NodeTest* )  annotate(*e, t2*)  Return *t2*  Else  Return *error*  End If  End If  **//（8）处理路径表达式**  If *e* **→** *Id<Step>+* Then  *sourceType* = type( *v[ Id ]* ) //从*v*中获得要处理的节点类型  If match(*sourceType, Node*) Then  *dotType* = *sourceType*  //StepNum表该路径表达式中的Step数量  For(i = 0; i < StepNum; i++){  //进行轴操作和节点测试  *dotType* = doAxisAndNodeTest( *dotType*,  *Step[i].Axis, Step[i].NodeTest* )  // ArgsNum 表Step[i]所包含的谓词表达式的数量  If *Step*[i].ArgsNum > 0 Then // Args表Step[i]的所有谓词表达式  For(j = 0; j < *Step*[i].Args.length; j++)  doTypeCheck(*Step[i].Args[j]*, *v, w, u*)  End If  }  annotate(*e, dotType*)  Retrun *dotType*  Else  Return *error*  End If  End If  **//（9）处理实参表达式**  If *Arg* **→** *e1* Then  *t1* = doTypeCheck(*e1, v, w, u*); //对实参的具体内容进行类型检查  annotate(*e1, t1*)  annotate(*Arg, t1*) //将结果类型标注到实参表达式上  Retrun *t1*  End If  **//（10）处理匿名函数**  If *Arg* **→** *fun(Id\*) e1* Then  *type1* = doTypeCheck(*Id1, v, w, u*)  …  *typen* = doTypeCheck(*Idn, v, w, u*) //对函数的参数做类型检查，获得参数类型  *funcType* = doTypeCheck(*e1, v’, w, u*)  Where //对函数体进行类型检查前，先将参数类型添加到*v*中，获得映射表*v’*  *v'* = < *Id1* **→** *type1* >, ... , < *Idn* **→** *typen* >  annotate(*Arg, funcType*)  Return *funcType*  End If  **//（11）处理FXQL程序**  If *Prog* **→** *e1* Then  *t1* = doTypeCheck(*e1, v, w, u*);  annotate(*e1, t1*)  annotate(*Prog, t1*) //将结果类型标注到*Prog*上  Retrun *t1*  End If  **辅助说明：**  算法中用到的辅助函数说明参见“”。 |

静态类型检查算法中所用到的轴操作与节点测试算法“doAxisAndNodeTest”，具体的算法描述如“”所示。需要说明的是：

（1）doAxisAndNodeTest算法中使用“empty”用于表示“空”类型，表明没有得到任何结果。由“ ”可知，“empty”对于“|”运算是幺元，因此不会对类型计算的结果产生任何影响。

（2）doAxisAndNodeTest算法只处理了ChildAxis等4种轴操作。虽然在XQuery中，定义了多种轴操作，例如ParentAxis、AncestorAxis，但在FXQL中，为了便于优化，ChildAxis、DescendantAxis、DescendantOrSelfAxis、AttributeAxis会被翻译为相应的“Step”表达式，其他的轴则是以原语函数的形式表示，例如AncestorAxis在FXQL中使用原语函数“ancestor（）”表示。因此，在算法doAxisAndNodeTest中，只给出处理ChildAxis等4种轴的流程，其他轴操作的类型检查过程与处理原语函数的过程类似，不再赘述。

表 3‑9 算法辅助函数预定义

|  |  |  |
| --- | --- | --- |
| 操作 | 表示 | 说明 |
| 获取表达式类型 | type(*e*) | 返回表达式*e*的类型。 |
| 给表达式标注类型 | annotate(e, t) | 为表达式*e*标注上类型*t*，返回标注后的表达式。 |
| 类型“并”运算 | getUnionType(*t1, t2*) | 对类型*t1*和*t2*进行“|”类型运算（参见“ ”），返回结果类型。 |
| 保存变量类型信息 | addVar( *v, Id, t*) | 向<变量名 **→** 类型>映射表*v*中添加新的映射：*<Id* **→** *t>*，并返回添加新映射后的映射表。 |
| 保存自定义函数 | addNewFunc(*w*,*FuncId*, <*Id1:type1,...,Idn:typen*>,  *e,funcType*) | 将名为“*FunId*”的自定义函数加入到自定义函数映射表*w*中，并返回添加后的映射表。其中“*<Id1:type1,...,Idn:typen>*”表示形参名以及形参类型声明，*e*表示自定义函数的函数体表达式。 |
| 比较两类型是否为同一类型 | match(*t1, t2*) | 比较类型*t1*与*t2*是否为同一类型，相同则返回*true*，否则返回*false*。 |
| 获得某节点的所有孩子节点 | getChildren(*t*) | 返回节点类型*t*的所有孩子节点类型。 |
| 判断两类型是否为存在继承关系 | isSubSet(*incr, result*) | 判断类型*incr*是否为在类型*result*的子类型，如果是则返回*true*，否则返回*false*。 |
| 获得某节点的所有属性节点 | getAttrs(*type*) | 返回节点类型*t*的所有属性节点。 |

表 3‑10 轴操作与节点测试算法doAxisAndNodeTest

|  |
| --- |
| **算法：**doAxisAndNodeTest( *type , axis , nodeTest* )  **输入：** *type* 节点类型  *axis* 轴操作标志位  *nodeTest* 节点测试，包括名称测试和类别测试两部分  **输出：** *Type* 结果类型  **功能：** 对*type*进行轴操作和节点测试，获得结果类型。 |
| **//轴操作是*ChildAxis*的情况**  If *axis* 🡪 *ChildAxis* Then  //首先判断type能否进行孩子轴操作：只有元素节点和文档节点才可以  If match(*type*, *Element*) == True || match(*type*, *Document*) == True Then  *childList* = getChildren(*type*); //获得*type*的所有孩子节点的类型  For(i = 0; i < *childList*.length; i++){ //对所有的孩子类型进行节点测试  If doNodeTest(*childList*[i], *nodeTest*) == True Then  *resultList*.addToList(*childList*[i])  //通过节点测试的类型加入到结果列表*resultList*中  }  Return *resultList*; //返回结果类型  Else  Return *empty*;  End If  End If  **//轴操作是*DescendantAxis*的情况**  If *axis* 🡪 *DescendantAxis* Then  *incr* = doAxisAndNodeTest(*type, ChildAxis, nodeTest*);  while(! isSubSet(*incr, result*)){ //通过迭代的方式获得所有后代节点的类型  *result* = getUnionType(*result, incr*);  *type* = *result*;  *incr* = doAxisAndNodeTest(*type, ChildAxis, nodeTest*);  }  Return *result*;  End If  **//轴操作是*DescendantOrSelfAxis*的情况**  If *axis* 🡪 *DescendantOrSelfAxis* Then  *t1* = doAxisAndNodeTest(*type, DescendantAxis, nodeTest*); //获得所有后代节点类型  *t2* = *type*; //保存自身的类型 *result* = getUnionType (*t1, t2*); //两者做“|”运算即得到结果类型  Return *result*;  End If  **//轴操作是*AttributeAxis*的情况**  If *axis* 🡪 *AttributeAxis* Then  //首先判断type能否进行属性轴操作：只有元素节点才可以，其他类型将返回空  If match(*type, Element*) Then  *attrList* = getAttrs(*type*); //获得type的所有属性节点的类型信息  For(i = 0; i < *attrList*.length; i++)  If doNodeTest(*attrList*[i], *nodeTest*) Then  *resultList*.addToList(*attrList*[i]);  //通过节点测试的类型加入到结果列表*resultList*中  Return *resultList*;  Else  Return *empty*;  End If  End If |

而“表 3‑10”中所用到的节点测试算法doNodeTest，具体描述如“表 3‑11”所示。

表 3‑11 节点测试算法doNodeTest

|  |
| --- |
| **算法：**doNodeTest( *type , axis , nodeTest* )  **输入：** *type* 节点类型  *nodeTest* 节点测试，包括名称测试和类别测试两部分  **输出：** *Boolean*  节点类型*type*是否通过了节点测试的判断结果  **功能：** 对*type*进行节点测试，如果通过了类型测试和名称测试，返回true；否则返回 false。 |
| If match(*type, nodeTest.type*) Then //判断*type*是否与*nodeTest*给出的测试类型相同  If *type.name == nodeTest.name* Then  //通过类型测试，并且*type*的名称与*nodeTest*给出的测试名称相同，返回true  Return True  End If  Else  Return False  End If |

### 算法复杂度

1. **时间复杂度**

假设在静态类型检查中，处理一个FXQL子表达式的时间为K，处理轴操作与节点测试的时间为P（即算法“doAxisAndNodeTest”所用的时间，参见“表 3‑10”），路径表达式中谓词的数量为a，处理一个谓词所需要的时间为Q，一个FXQL语句中的子表达式个数为N，那么，根据算法“doTypeCheck”（参见“表 3‑8”），很容易得到对FXQL中各种表达式进行类型检查的时间复杂度，具体结果如“表 3‑12”所示。

表 3‑12 检查各种表达式的时间复杂度

|  |  |
| --- | --- |
| FXQL表达式 | 时间复杂度*O*(m) |
| 常量 | m = 1 |
| 变量 | m = 1 |
| 条件表达式 | m = 3K |
| 函数调用 | m = NK + K |
| where表达式（变量绑定） | m = K + K |
| where表达式（函数绑定） | m = NK + K + K |
| 轴操作 | m = K + P |
| 路径表达式 | m = N(P + aQ) |
| 实参表达式 | m = K |
| 匿名函数 | m = NK + K |
| FXQL程序 | m = K |

就一般情况而言，P、Q的数量级要远小于N、K，而N与K的数量级比较接近，那么，由“表 3‑12”可以看出，类型检查算法“doTypeCheck”的时间复杂度近似为*O*()。

1. **空间复杂度**

从“表 3‑8”很容易看出，静态类型检查算法所占用的存储空间主要用于保存待处理的FXQL表达式、变量映射表、自定义函数映射表、XML Schema映射表，算法运行过程中临时变量所占用的存储空间与上述这4种数据所占用的空间相比，可以忽略不计。设一个FXQL表达式平均所占用的存储空间为e，需要做类型检查的FXQL表达式的个数为N，变量映射表、自定义函数映射表、XML Schema映射表这些映射表中每条记录所占用的存储空间平均大小为e’，3个映射表所保存的记录总数为M，那么，算法“doTypeCheck”所占用的存储空间大小为Ne + Me’。而一般情况下，N和M的数量级比较接近，那么，静态类型检查算法“doTypeCheck”的空间复杂度近似为*O*(N(e + e’))。

### 其他需要说明的问题

**（1）对于自定义函数中递归的处理**

XQuery自定义函数允许出现递归定义，因此，在FXQL中，也应该考虑如何对递归定义的自定义函数进行类型检查。具体的解决方法如下：

设置一个栈userFunc用于保存自定义函数名、参数列表（包含参数个数和参数类型这些信息）。

在对用户自定义函数的函数定义或是函数调用进行类型检查时，首先检查userFunc中是否有函数名相同并且参数列表相同（参数个数相同，并且对应位置的参数类型相同）的记录：

* 如果没有相同的记录，说明没有出现递归或是已出现递归但还不需要终止，则将当前分析的函数名以及相应的参数列表保存到栈userFunc中，继续对该函数调用进行类型检查，检查结束后将该记录弹出栈userFunc；
* 如果有相同记录，那么说明出现了递归，并且应该在此处终止递归。此时，将该函数调用的结果类型置为该自定义函数声明的返回类型（如果没有声明返回类型，则置为“anyType”，表示返回类型可以为任意类型），然后对表达式中的剩余部分进行类型检查。

**（2）退出静态类型检查的情况**

之前曾提到过，只有在引入XML文档指定的XML Schema的情况下，才能获取XML节点有关结构、内容的约束信息，基于这些信息才能对XML节点进行精确地类型检查，但不是所有的XQuery查询中都会明确提出要导入哪些Schema，XQuery所查询的XML文件也可能没有Schema，或者即使有Schema，但由于各种原因导致暂时无法获取这个Schema。

因此，这就要求在静态类型检查时，如果对节点的类型检查需要Schema但相应Schema无法获得，那么就应该将该节点置为缺省类型（一般是“anyType”，表示任意类型），继续处理后续的表达式，如果能继续进行类型检查则继续做，不能做则逐层退出。

为满足这一要求，在对FXQL中各种原语函数、表达式进行类型检查的代码中，都包含有形式如“”所示的代码片段：

表 3‑13 处理没有Schema情况的算法

|  |
| --- |
| if(检查对象是节点类型并且相应的Schema无法获得){  将检查对象的类型置为anyType，并返回；  }else if(检查对象是anyType){  返回anyType；  } |

上面的伪代码其具体含义是：当碰到对节点进行类型检查，需要Schema但相应Schema无法获得的情况，则直接将该节点的类型置为“anyType”；当检查对象是“anyType”类型时，则退出检查，直接返回“anyType”。这样就为节点类型的检查设置了一个“安全通道”，当不能检查时就可以逐层退出类型检查。而之所以选择“anyType”设置“安全通道”，则是因为“anyType”是“|”运算的零元（参见“ ”），即使参与到类型计算中，其结果仍为“anyType”，不会对类型计算产生影响。

**（3）异常处理**

静态类型检查模块运行的过程中，会出现异常的情况有2种：

* FXQL表达式中出现了类型错误。
* XML Schema解析过程出现了错误。

对于第一种情况，静态类型检查模块会根据类型错误的种类，从上下文中取出合适的参数，生成相应的错误信息，输出给用户，以方便用户调试。实际上，对这种情况的处理正体现出该模块对于FXQL表达式起到了“类型检查”的作用。

对于第二种情况，静态类型检查模块也将根据错误出现的位置、错误的类型，生成合适的信息，输出给用户。

更详细的处理过程请参见“”。

## XML Schema导入算法

在当前系统实现中，有两个位置可能发生XML Schema的导入：

1. 进行类型检查之前

在XQuery查询的序言部分可能包含了XML Schema导入语句，在进行类型检查之前需要将这些Schema导入，以便在对FXQL表达式的类型检查过程中，使用Schema中的信息，增强类型检查的精度；

1. FXQL表达式中调用“doc()”函数的位置

“doc()”函数为XQuery内置函数，用于获取特定XML文档的文档节点，FXQL也包含有同名并且作用相同的原语函数。当FXQL表达式中“doc()”函数所使用的XML文档，包含了该文档所使用的Schema所在的位置，就应该将这个Schema导入到当前的系统中。

下面给出Schema导入的基本算法。

### 基本算法

在当前系统中，我们使用了Apache Xerces作为XML Schema解析器，这样就不必再对Schema进行语法分析。但Apache Xerces解析的结果是以树形结构保存的，并且Schema的类型信息都是以Apache Xerces自定义的一套类型系统来表示的，需要我们将其转换成GeoQuery系统能处理的类型表示。另外，Xerces解析的结果中，简单类型定义的类型信息很明确，但是复杂类型定义的类型信息则需要进一步计算。因此，Schema导入主要包含两部分内容：简单类型的翻译，复杂类型的计算。

**（1）简单类型的翻译**

Apache Xerces中定义的基本类型，与XQuery中的类型基本一致，而FXQL的类型系统是参考XQuery类型系统而创建的，因此，Apache Xerces中定义的基本类型，大部分可以直接对应到FXQL类型。对于可以对应到一起的类型，翻译相对简单，例如对于Apache Xerces中定义的“integer”类型，直接返回FXQL中的“integer”类型即可。因此，对于简单类型的翻译，其基本算法如“”所示：

表 3‑14 简单类型翻译算法tranSimpleType

|  |
| --- |
| **算法：**tranSimpleType(*type1*)  **输入：***type1* Apache Xerces中定义的基本类型  **输出：***type2* FXQL类型  **功能：**将Apache Xerces类型翻译为FXQL类型 |
| switch(*type1*){  ……  case Apache Xerces中的“integer”类型：*type2* = FXQL中的“integer”类型；  case Apache Xerces中的“decimal”类型：*type2* = FXQL中的“decimal”类型；  ……  }  return *type2*; |

Apache Xerces中还定义了日期类型，XQuery中也支持日期类型，但是目前的FXQL尚未提供支持，因此这些日期全部翻译为FXQL类型系统中的空类型“empty”，由“ ”可知，“empty”对于类型运算“|”是幺元，因此“empty”将不会影响“|”的类型计算结果。

另外，Apache Xerces中还定义了“union”和“list”两种类型，这是XML中支持的两种节点，但XQuery中没有对应的类型（相应地，FXQL中也不存在对应的类型）。因此，笔者在翻译这两种类型时，根据“union”和“list”的形式语义，将其看做是复杂类型，先翻译其内部包含的子类型，然后将这些子类型进行做类型运算“|”，其计算结果，作为“union”和“list”的类型表示。

**（2）复杂类型的计算**

由于在Apache Xerces的Schema解析结果中，一个复杂类型定义经常是由名为“particle”、“modelGroup”的类对象相互嵌套构成的。例如，假设现在对“”中的Schema复杂类型进行解析。该复杂类型通过Apache Xerces解析所得的结果，大致结构如“表 3‑16”所示。

表 3‑15 一个Schema复杂类型的定义

|  |
| --- |
| <xsd:complexType name="complex">  <xsd:sequence>  <xsd:element name="childElem" type="xsd:string">  </xsd:element>  </xsd:sequence>  </xsd:complexType> |

表 3‑16 Xerces解析结果的结构

|  |
| --- |
| particle  └─ modelGroup  └─ particle  └─ childElem元素的具体定义 |

“表 3‑16”的“└─”表示包含关系，也就是说，Apache Xerces解析所得的结果中，“particle”包含一个“modelGroup”，“modelGroup”中又包含一个“particle”，最内层的“particle”包含了子元素“childElem”的具体类型信息，而子元素“childElem”的类型通过简单类型的翻译即可获得。因此，计算这个名为“complex”的复杂类型最终归结于对其子节点的翻译。

一般来讲，与上面的例子类似，对于复杂类型的计算，到最后都可以归结为对于其子节点类型的翻译。那么，对复杂类型的计算，大致算法如“”所示。

将Schema中的元素、属性等节点的类型信息转换为FXQL类型表示的处理过程相对简单，不再给出算法描述。具体的表示方式请参见“”。

### 算法复杂度

1. **时间复杂度**

显然，绝大多数情况下，简单类型的翻译算法“tranSimpleType”的时间复杂度为*O*(1)，但当处理“union”和“list”两种类型时，由于会出现遍历操作，此时算法“tranSimpleType”的时间复杂度将会变为*O*(N)。

而由“表 3‑17”很容易得出，算法“tranComplexType”的时间复杂度总是*O*(N)。

1. **空间复杂度**

在算法“tranSimpleType”中，只需要创建一个新的FXQL类型来保存翻译Apache Xerces类型所得到的类型信息，因此，算法的空间复杂度为*O*(1)。

算法“tranComplexType”在计算复杂类型的过程中，需要为不断为子节点创建新的FXQL类型以保存其类型信息，因此，算法的空间复杂度为*O*(N)。

### 递归的处理

Schema的复杂类型定义可能出现递归定义的情况，如“”所示。

在这个Schema中，“bookType”出现了递归定义，如果按照“”中的算法进行处理，将会出现程序无法终止的情况。为解决这一问题，设置一个名称栈用于保存处理过的类型名，具体解决方法如“”所示。

需要说明的是，表示出现了递归定义的ElementType，其属性和内容的值为“null”，而其他情况下，ElementType属性和内容的值都不是“null”，而是保存相应信息的类对象，如果一个元素没有属性也没有子节点，其对应的类对象就不包含任何信息，但并不是“null”，这样就可以和出现递归的情况区分开。

表 3‑17 复杂类型计算算法tranComplexType

|  |
| --- |
| **算法：**tranComplexType  **输入：**xsComplex 表示Apache Xerces中的复杂类型定义。  **输出：**fType 表示使用FXQL类型表示的复杂类型定义。  **功能：**把Xerces给出的复杂类型定义转换为FXQL类型表示。 |
| For(i = 0; i < xsComplex.childNum; i++){  If( xsMatch(xsComplex.children[i], xsElement)) Then  fType.addChild( xsTransElem(xsComplex.children[i]) );  //将翻译所得的FXQL元素节点添加到fType中，作为其孩子节点  Else If( xsMatch(xsComplex.children[i], xsAttribute)) Then  fType.addChild( xsTransAttr(xsComplex.children[i]) );  //将翻译所得的FXQL属性节点添加到fType中，作为其孩子节点  End If  }  Return fType;  **辅助说明：**  xsElement：表示Apache Xerces中用于保存元素类型信息的数据结构。  xsAttribute：表示Apache Xerces中用于保存属性类型信息的数据结构。  xsComplex.children：表示保存xsComplex内部所有类型信息的数组。  xsMatch(t1, t2)：判断t1与t2是否为同一种数据结构。  xsType：表示Apache Xerces中保存类型信息的数据结构。  xsTransElem(xsType)：将xsType翻译为使用FXQL类型系统表示的元 素类型，返回翻译结果。限于篇幅，具体算法从略。  xsTransAttr(xsType)：将xsType翻译为使用FXQL类型系统表示的属性 类型，返回翻译结果。限于篇幅，具体算法从略。 | |

表 3‑18 Schema中出现递归定义的情况

|  |
| --- |
| <xsd:complexType name='bookType'>  <xsd:sequence>  <xsd:element name='references' minOccurs='0' maxOccurs='1'>  <xsd:complexType>  <xsd:sequence>  <xsd:element name='book' type='bookType' minOccurs='0' maxOccurs='unbounded'/>  </xsd:sequence>  </xsd:complexType>  </xsd:element>  </xsd:sequence>  </xsd:complexType> |

表 3‑19 处理Schema递归定义的算法checkRecursive

|  |
| --- |
| **算法：**checkRecursive(*curRETypeParent* , *type , nameStack*)  **输入：** *curRETypeParent* 当前正在解析的节点的父类型所对应的FXQL 类型表示。  *type* 当前正在解析的节点在Schema中的类型信息。  *nameStack* 节点类型名称栈，保存目前已经解析过的 Schema节点类型的名称。  **输出：** 无  **功能：** 处理Schema递归定义。 |
| If (nameStack.findSameName(*type*)){  *newElem* = getNewElementType(*type.name, null, null*);  //表示创建一个仅包含名称，属性和内容的值为“null”的FXQL元素节点  curRETypeParent.addChild(*newElem*);  }else{  nameStack.push(*type.name*);  //将节点名称入栈，*type.name*表示节点类型*type*的名称  tranSchemaType(*type*); //表示继续分析*type*所包含的类型信息  nameStack.pop();  }  **辅助说明：**  **getNewElementType(*arg1, arg2, arg3*)：**用于创建一个新的FXQL类型系统中的元素节点，三个参数依次为新建节点的元素名称、属性列表、具体内容。  **addChild(*arg1*)：**用于表示向当前节点类型添加孩子节点，其中*arg1*表示使用FXQL类型系统表示的节点类型。 |

# 静态类型检查模块的设计与实现

## 静态类型检查模块设计



图 4‑1 静态类型模块的设计

静态类型检查模块的总体设计如“图 4‑1”所示。下面给出具体的说明。

**（1）输入与输入**

静态类型检查模块的输入包含两部分内容：

* 不含有类型信息的FXQL表达式：静态类型检查的主要对象。
* XML Schema：用于提供XML节点的类型信息，以提高类型检查的精度。由于Schema导入在静态类型检查中是可选的，因此XML Schema并不是必须的。

输出为通过类型检查、带有类型信息的FXQL表达式。

**（2）组成**

由“图 4‑1”可以看出，静态类型检查模块主要由两个功能模块组成：类型检查模块和Schema导入模块。下面分别给出这两个功能模块的设计。

**类型检查模块**

类型检查模块由以下三个部分组成：

1. **表达式类型检查器**：对FXQL表达式进行静态类型检查。

该部分首先计算子表达式的类型，将子表达式的类型与期望的类型进行比较，如果符合要求则继续进行类型检查，最终推导出整个表达式的类型，完成类型检查；如果不符合要求，则抛出相应的类型错误。

1. **函数类型检查器**：对FXQL原语函数以及用户自定义函数进行静态类型检查。

该部分首先检查传入参数的个数与函数声明是否一致，然后计算所有实参的类型，将其与函数声明中对应的形参类型比较，如果符合要求，则根据实参的类型、函数体，计算出结果的类型并返回；如果不符合要求，则抛出相应的类型错误。

1. **基本类型运算集**：实现常用的类型运算。

如“ ”中所提到的几个类型运算，但不同的是，这里的类型运算都是针对FXQL表达式的类型。

这三个部分之间是调用与返回的关系：

* “表达式检查器”调用“函数检查器”，表明表达式中允许函数调用。当正在处理的表达式中出现函数调用时，“表达式检查器”会将这个函数调用提交给“函数检查器”处理，“函数检查器”处理结束后，将结果返回给“表达式检查器”。
* “函数检查器”调用“表达式检查器”，表明函数的参数可以是表达式。当参数是表达式时，“函数检查器”会将表达式提交给“表达式检查器”处理，得到“表达式检查器”返回的结果后，再继续对函数调用进行类型检查。
* “表达式检查器”、“函数检查器”都会调用“基本类型运算集”，表示这两个组成部分都会用到基本的类型计算。

每个部分具体的输入、输出说明如下：

①：需要进行类型检查的FXQL表达式。

②：通过类型检查的FXQL表达式。

③：通过类型检查的FXQL函数调用或是函数定义。

④：需要进行类型检查的FXQL函数调用或函数定义。

⑤、⑦：需要进行类型运算的FXQL类型信息。

⑥、⑧：作为类型运算结果的FXQL类型信息。

需要说明的是，之所以将表达式与函数的静态类型检查分开处理，是因为表达式与函数的检查方式差异较大，而不同的函数、不同的表达式，虽然具体内容不同，但是检查的方式基本相同，因此将相应的处理代码放到一起，便于管理与维护。

**Schema导入模块**

Schema导入模块主要由两部分组成：Apache Xerces，翻译与计算模块。各组成部分功能定义如下：

1. Apache Xerces：一款开源的Schema解析器，其输入为XML Schema文本文件，输出为一棵带有Schema文件内所有类型信息的DOM（Document Object Model，文档对象模型）树。
2. 翻译与计算：根据Apache Xerces输出的解析结果，计算出该Schema内的节点类型信息，保存起来，供类型检查模块使用。

需要说明的是，由于Apache Xerces的解析结果使用的是Xerces自定义的一套类型表示，使用起来很不方便，因此加上一个“翻译与计算”模块，将Xerces解析结果转换成FXQL类型表示（FXQL类型系统的表示方式参见“ ”），以便类型检查模块能够直接使用。而保存转换后类型信息的数据结构设计，详细说明请参见“”。

## 数据结构设计

### FXQL类型系统

要进行静态类型检查，首先需要解决的问题就是如何表示类型。参考XQuery类型系统，FXQL类型系统的数据结构基本设计如“”所示。与XQuery类型类似，FXQL类型也包括4大类：原子类型（AtomicType及其子类），节点类型（NodeType及其子类），序列类型（SequenceType），用户自定义类型（REType及其子类）。下面给出各个类的详细说明。



图 4‑2 FXQL类型系统类设计

所有的类型都派生自Type类，下面对各个类型类做详细介绍。

**（1）AtomicType**

目前，FXQL类型系统中定义了30多种原子类型，更详细的原子类型类设计请参见“”。XQuery的原子类型，除日期类型外，都有与之对应的FXQL类型。FXQL类型系统中，所有属于原子类型的类都从AtomicType类派生。每种原子类型都独自形成一个类，并提供一个默认的构造方法。

例如，现在想要创建“integer”这个类型，那么就先创建一个IntegerType类，并使其派生自AtomicType；然后，再在IntegerType类中提供一个合适的构造函数，供外界调用。其他的类型也可以类似地根据具体情况逐一实现。

此外，在具体实现中，空类型也被归类到原子类型中（使用“”中的“NONE”来表示），因为一个表达式是空类型，则说明该表达式的类型是确定并且是唯一的，而且空类型又是一个简单的类型，从这个角度来看，空类型和其他原子类型的性质是一样的，因此将空类型归类到原子类型是合理的。

**（2）NodeType**

参考XQuery类型系统，FXQL中定义了7种节点类型：文档节点（document node），元素节点（element node），属性节点（attribute node），文本节点（text node），注释节点（comment node），处理指令节点（processing instruction node），命名空间节点（namespace node）。这七种节点类型都从NodeType派生，如“”所示。



图 4‑3 FXQL节点类型类设计

节点类型的构造算法同原子类型是类似的。不过在节点类型中，我们添加了nodeTest这一属性，这个属性存放的是当前节点的QName（Qualified Name的简称），以及节点的种类（NodeKind），当需要获知节点有关结构、内容的约束信息时，使用这个属性就可以到XML Schema中找到相应的信息。

需要说明的是，尽管之前曾提到过，节点类型有关结构、内容的约束信息保存在XML Schema中，但笔者认为，这些信息应该体现在FXQL表达式中（具体的表示方式请参见“”），而不应该体现在类型系统的数据结构上。

这是因为如果在NodeType中保存节点有关结构、内容的约束信息，那么在对每个表达式进行静态类型检查时就需要先判断是否为节点类型，如果是节点类型就需要特殊处理，但大部分数据类型不是节点类型，显然，这样做将会增加类型检查的时空开销，影响其处理效率。

因此，NodeType只被用于表明是哪种节点类型（是元素节点，属性节点，还是其他节点类型），而节点有关结构、内容的约束信息则用其他FXQL类型的组合来表示——REType及其子类正是为了更细致地节点的这些信息，才加入到FXQL类型系统中的。

**（3）ItemType**

在XQuery中，“项”（Item）是原子类型和节点类型的通称，相应地，我们在FXQL类型系统中，设计ItemType类，作为AtomicType和NodeType的公共父类，表示在FXQL类型系统中，原子类型、节点类型都是“项”类型。

**（4）SequenceType**

SequenceType用于保存序列中所有类型的信息，“types”数组用于保存所有类型信息，“occ”用于表示出现描述符（包括“1”、“？”、“+”、“\*”，参见“ ”）。需要特别说明的是，XQuery中，一个序列类型内部不能再嵌套序列类型，而为了便于程序的优化，FXQL中的序列类型则采用了广义表的结构，允许序列类型内再继续嵌套序列类型。比如说，在FXQL表达式可能出现“（2，（3，5））”这样的序列，并且这个序列的类型就是“（integer，（integer，integer））”，而XQuery查询中这样的一个序列的类型就是“（integer，integer，integer）”，即XQuery无法保存嵌套的序列类型。正是为了支持序列的嵌套，才选择基类“Type”作为“types”数组的类型——这使得FXQL的序列中既可以包含原子类型、节点类型，也可以包含序列类型。

另外，上文中已经指出，FXQL的类型检查主要依据的是XQuery形式语义规范，FXQL中的类型运算与XQuery的类型运算基本上是一致的，也有prime、quantifier等类型运算（参见“ ”）。而“occ”主要是通过“quantifier”运算得到的，由于代数结构**（**quantifier, {“1”，“？”，“+”，“\*”}**）**与代数结构**（**逻辑“或”, {0，1，2，3}**）**是同构的（证明较为简单，略），因而为了快速得到“occ”，将“occ”设计成一个取值范围在0到3之间的数值型变量，“occ”的值为0、1、2、3分别对应于“1”、“？”、“+”、“\*”，并使用逻辑“或”操作来完成“quantifier”运算。

**（5）Type**

Type是FXQL类型体系中所有类型的父类型，Type类中基本上不定义任何的属性和方法，但所有ItemType在Type类中都被定义为静态常量，并通过静态的构造方法来获取。

例如，要想定义简单类型“integer”，则可以在Type中定义常量“static final Type INTEGER”，然后在Type类中，添加相应的静态构造方法即可。

这样处理以后，如果在类型检查的过程中计算出某个表达式的类型是“integer”这样的简单类型，则可以直接使用语句“Type.INTEGER”获得Integer这个类的实例，而不必再临时构造了。另外，由于该对象是静态的，因此在程序运行的过程中就会有且只有一个实例，所以在进行类型比较的运算时，处理起来也很方便。

另外，对于ItemType所表示的简单类型，将其设计成静态的也是类型表示的内在要求。例如，对于FXQL表达式“plus(4, 3)”，显然“4”和“3”都是“integer”类型，两者的类型完全相同，没有任何本质的区别。如果在具体实现类型的时候，将“integer”这些简单类型都设计为动态的，那么为表示“4”和“3”的类型，将会创建两个不同的表示“integer”类型的对象，而这种做法实际上就默认“4”和“3”的类型有区别——显然，这样做所传达的信息，并不是我们实际要表达的类型信息。因此，只能将简单类型设计成静态表示。

对于除ItemType以外的类型，则将它们看成是动态的。例如，如果两个类型同为SequenceType，并不能说明这两个类型就是同一个类型，因为SequenceType里面的内容并不一定完全一致。

此外，需要说明的是，FXQL中的“anyType”（表示任意类型，在静态类型检查模块中用作缺省类型。），与“integer”这样的简单类型一样，也是在Type中以常量的形式定义的，由于“anyType”表示表达式的类型可以是任何一种类型，显然放到Type的任何一个子类中都不合适，只能放到Type类中。

*注：下面将使用Type.INTEGER、Type.ELEMENT这样的形式表示具体类型。*

**（6）REType**

正如之前所提到的，REType主要用于Schema相关的类型表示，以表示节点类型有关结构、内容的约束信息。REType及其子类的大体设计如“”所示。



图 4‑4 REType及其子类的大体设计

其中ElementType、AttributeType分别用于表示Schema中元素、属性的类型信息，ElementType中的“attrList”用于保存属性列表，“content”保存具体内容的类型信息，AttributeType中的“type”用于保存属性的具体类型。例如，对于“<book year=’2000’>The Bible</book>”这样的一个XML节点，对应的REType表示就是一个’”name”值为“book”的ElementType对象，并且该对象的属性列表“attrList”中包含一个’”name”值为“year”、“type”值为“integer”类型的AttributeType对象。REType及其子类的具体设计，以及Schema类型信息的具体表示方式请参见“”。

### Schema导入的数据结构

XML Schema保存着对XML节点有关结构、内容的约束信息，在FXQL中将使用上文中提到的REType及其子类来表示Schema所包含的类型信息。显然，如果每次碰到节点类型才进行Schema的导入，I/O开销将会非常大，比较好的做法是，在进行静态类型检查前，将所有涉及到的XML Schema一次性导入到上下文环境中，这样在进行类型检查时，若碰到节点类型并需要获取该节点的Schema内容时，直接从上下文环境中就可得到。在上下文环境中用于保存Schema内容的数据结构如“”所示。



图 4‑5 保存Schema导入信息的数据结构

考虑到一个FXQL表达式中可能需要导入多个XML Schema，因此使用二级存储结构：SchemaTypes用于保存一个XML Schema的类型信息，SchemaPool则用于保存所有已导入的Schema所对应的SchemaTypes对象。

显然，SchemaPool需要频繁进行的操作是根据给定的shemaURI获取相应的SchemaTypes对象，或是根据给定的SchemaURI和节点名返回对应的节点类型信息。因此，在SchemaPool中使用一个以SchemaURI为关键字的哈希表schemaPool保存所有SchemaTypes对象，并且提供了方法getREType(QName qname)用于获取节点的类型信息，其中的qname由SchemaURI和节点名称组成，这样就可以很方便地获得节点的类型信息。

之前曾提到过，XML Schema所包含的内容中，我们经常用到的是类型定义，元素声明和属性声明。在SchemaTypes中，为了处理方便，将其分开保存：哈希表globalTypes用于保存类型定义，globalNodes则保存节点类型的声明。

有关SchemaPool和SchemaTypes更详细的说明参见“”。

### 表达式相关的数据结构

每个FXQL表达式都是以MExpr类对象的形式表示的，因此，为了在FXQL表达式中保存类型信息，在MExpr类中添加如下内容：



图 4‑6 MExpr类中与类型有关的数据结构

type和nodeType配合表示FXQL表达式的类型，candidateNodes用于保存当前可能的节点类型，下面给将逐一详细介绍。

1. **type和nodeType**

如果FXQL表达式的类型是节点类型，则type用于表示该表达式是哪种节点类型，nodeType则保存该节点具体的结构和内容。

需要说明的是，在涉及到节点类型的静态类型检查过程中，有时只需要使用type，有时则要使用nodeType。比如说，FXQL原语函数“addContent()”用于为一个元素节点添加内容，要求第一个参数必须是元素类型，那么此时只需要使用实参表达式的type判断出这个参数是否为元素即可；而当原语函数“plus()”的一个实参是元素类型时，由于“plus()”要求其实参的类型必须是数值类型，这时就需要使用nodeType来获得该元素的具体内容（这个过程实际上相当于XQuery中的“原子化”），以判断这个实参是否符合函数对参数类型的要求。因此，在FXQL表达式中的type、nodeType的赋值、更新必须同步进行。

当FXQL表达式的类型不是节点类型时，表示方式如下：

1. 如果FXQL表达式的类型是原子类型，那么nodeType显然没有任何用处，因此将其置为“anyType”，表明nodeType不会被用到。
2. 如果FXQL表达式的类型是序列类型，由于序列中可能包含节点类型，因此nodeType将保存一个类型列表，对应于序列中相应位置的类型：若序列中的某一项是原子类型，则将nodeType列表中对应位置赋值为“anyType”；若序列中的某一项是节点类型，则将nodeType列表中对应位置赋值为该节点的Schema类型信息。

例如，在静态分析中，当通过类型计算获知一个表达式expr的类型为“title”元素类型，并且假设“title”元素的声明与“表 2‑3”中的“title”相同，那么该表达式的类型信息表示如“表 4‑1”所示。其中，“title”对应的ElementType对象不包含属性，但规定其内容必须是“string”类型。

表 4‑1 节点类型的表示实例

|  |
| --- |
| expr.type = Type.ELEMENT  expr.nodeType = title对应的ElementType对象 |

1. **candidateNodes**

候选节点candidateNodes是和nodeType配套使用的，因为在路径表达式中，我们在获得当前节点的类型之前，应该根据Schema导入的结果首先检查一下该节点是否存在于上下文的环境中。例如，给出路径表达式doc(“example.xml”)/a，静态类型检查模块首先通过“example.xml”这一XML文档获得相应Schema，将Schema内所有全局定义的节点类型放入到当前的candidateNodes中，接着分析“./a”这一孩子轴操作，如果candidateNodes中有名为“a”的节点，那么取出这个节点的类型信息，作为结果类型；如果没有，那么类型检查的结果类型就为“empty”，表示没有得到任何结果。

在比较简单的路径表达式中，candidateNodes和当前节点名称是一样的，但是在使用了后代轴操作这样的情况下，其candidateNodes的内容可能就不是唯一的了。例如给出路径表达式doc(“example.xml”)/a//b，在分析“a//b”这一“步”时，首先要从“a”节点的子孙节点中要找出所有候选节点，显然，如果“a”的子孙中有多个名为“b”的节点，那么candidateNodes中也就会有多个节点，并且这些节点都是合法的，都需要保存起来。为方便查找，candidateNodes使用哈希表来保存所有可能的结果。为区分不同的结果，将candidateNodes的存储对象作为它的关键字。

## 错误处理机制

对FXQL表达式进行静态类型检查的主要目的，就是为了在静态分析阶段，提早发现错误。因此，如何处理FXQL表达式中存在的类型错误，如何给出适当的错误信息以方便用户纠正这些错误，是静态类型检查模块中十分重要的一环。

由于静态类型检查是FXQL预处理流程中的一个环节，不能决定是否终止程序运行，因此，当发现当前处理的FXQL表达式中存在类型错误时，比较合适的做法是：根据上下文的信息，生成合适的错误信息，然后退出对FXQL表达式的静态类型检查，将错误信息提交给上一级模块处理。

另一方面，在类型检查的过程中，可能需要进行XML Schema的导入操作，以便获取XML节点的类型信息，增强类型检查的精度。那么，当Schema导入发生错误时，也应该给出相应的错误信息，提示用户Schema导入失败。但是，由于Schema导入是FXQL静态类型检查中一个可选的环节，不能决定是否终止程序运行，因而，发生Schema导入错误时，应该将这个错误提交给类型检查模块来处理。

基于以上考虑，并参考GeoQuery中已有的错误处理机制，错误处理流程的设计如“”所示。具体的处理步骤如下：

1. 由“类型检查模块”对FXQL表达式进行类型检查。
2. 当“类型检查模块” 需要XML Schema中的类型信息时，则调用“Schema导入模块”完成Schema的导入操作，使用Schema导入的结果做进一步的类型检查。
3. 如果“Schema导入模块”读取XML Schema的过程中出错，则由“Schema导入错误处理器”负责生成相应的错误信息，并将该错误返回给“类型检查模块”。“类型检查模块”在得知发生导入错误后，将停止检查，将这个错误提交给“FXQL错误处理器”，以便做进一步处理。
4. 如果在静态类型检查过程中，检测到了类型错误，则“类型检查模块”将停止检查，由“FXQL静态类型错误处理器”负责生成相应的错误信息，“类型检查模块”则把类型错误提交给“FXQL错误处理器”，以便做进一步处理。
5. 如果不存在任何错误，FXQL表达式通过了类型检查，则将检查后的带有类型信息的FXQL表达式提交给后续模块，做进一步处理。



图 4‑7 静态类型检查模块错误处理流程

### 静态类型检查的错误处理

W3C给出的XQuery规范中，给出了在XQueyry静态分析过程中，抛出类型错误的条件。为方便起见，我们在FXQL中直接使用XQuery规范中所定义的类型错误。抛出类型错误的具体实现方法如下：

1. **终止检查的实现方法**

在具体实现中，设计一个异常类StaticTypeException，当发生类型错误时，创建一个StaticTypeException对象，根据发生错误的位置，传递相应的参数给这个对象，以生成对应的错误信息。

1. **错误信息的生成**

W3C制定的XQuery规范对各种错误进行了分类，并给出了对应的错误编号，如“参数个数与函数声明不一致”这个错误的编号是“XPST0017”。因此，在创建StaticTypeException对象时，根据已出现的静态类型错误的类型，将相应的错误编号传递给StaticTypeException对象，同时从上下文中取出合适的参数传递给该对象。在StaticTypeException类内部，针对每个静态类型错误都给出了错误信息的模板，这样就可以根据错误编号、传入的参数，自动生成相应的错误信息。

例如，FXQL函数调用语句plus(“a”,4)中，第一个参数类型不是数值类型，发生类型错误，在创建StaticTypeException对象时，就应该将错误编号“XPTY0004”（表示“表达式的静态类型与当前上下文不匹配”这一类型错误的错误编号，参见“”）、“a”（类型错误的表达式）、“integer”（上下文所期望的数据类型）传递给该对象，StaticTypeException内部会根据这些参数自动生成形如“The type of expression ‘a’ is invalid, the type should be integer.”这样的错误报文。接着，将这个StaticTypeException对象传递给上一级的错误处理模块，以便进一步处理。静态类型检查所能处理的类型错误参见“”。

### Schema导入的错误处理

这一部分使用了GeoQuery系统中已有的Schema导入错误处理器SchemaErrorHandler对Schema导入错误进行处理。SchemaErrorHandler仿照Apache Xerces自带的defaultErrorHandler，当Apache Xerces对XML Schema进行解析的时候，所有的错误信息全部都会写到SchemaErrorHandler提供的一个缓冲区中。Schema导入结束后，只需要查看一下SchemaErrorHandler的缓冲区，如果缓冲区中有错误信息，则终止类型检查，该错误抛给上一级的错误处理器；如果缓冲区中没有错误信息，则说明XML Schema解析成功，继续进行类型检查即可。

# 测 试

## 功能测试

功能测试采用了W3C给出的XML Test Suite作为测试用例集，所覆盖的测试用例范围参见“”所示。

图 5‑1 功能测试用例所覆盖的范围

在对静态类型检查模块的功能测试中，选取了XML Test Suite中的48个用例进行测试，并且这些用例全部通过了测试。具体所选取的用例请参考“”。 XML Test Suite有关静态类型检查的用例中，没有测试的用例含有self轴操作或是parent轴操作，而目前GeoQuery系统尚在开发中，尚未对这两个轴操作提供支持，因此也就没有对这些用例进行测试。

## 性能测试

为了计算出在整个对XQuery查询语句的处理过程（从输入XQuery查询语句到输入查询结果）中，对FXQL表达式的静态类型检查所花费的时间开销，笔者使用了GeoQuery系统中已有的测试用例来做性能测试。笔者根据XQuery查询的类别，分别统计了不同种类的XQuery查询语句所对应的FXQL表达式进行静态类型检查所需要的时间，以及每种查询从输入到得到查询结果所需要的总的处理时间，统计结果如“”所示。限于篇幅，不再给出测试用例的具体内容。

从“”可以看出，对于大部分的XQuery查询语句而言，对其相应的FXQL表达式做静态类型检查的时间开销是比较小的，平均只占去了总处理时间的12.20%，可见静态类型检查模块的性能还是比较让人满意的。同时，也可以看出，当前路径表达式的类型检查时间占总处理时间的43.64%，时间开销偏大，因此，静态类型检查模块对于路径表达式的处理上仍有待改进。

图 5‑2 静态类型检查所需要的时间开销（单位：毫秒）

# 结 论

北京工业大学软件与理论实验室正在研发一款XQuery查询引擎GeoQuery，由于从用户输入的原始XQuery查询中分析出用户的查询意图较为困难，课题组设计了一种函数式查询语言FXQL作为中间语言用于描述XQuery查询计划。目前，课题组希望在FXQL表达式提交给执行引擎前，对FXQL表达式进行静态类型检查，以提前发现类型错误。因此，本文在参考XQuery类型系统和静态类型检查过程的基础上，研究了如何对FXQL表达式进行静态类型检查。

笔者参考XQuery类型系统，设计并实现了FXQL类型系统，并根据W3C给出的XQuery形式语义，设计并实现了对FXQL的静态类型检查算法，该算法能为FXQL表达式标注上类型信息，并能够在FXQL表达式提交给执行引擎执行前，检查出FXQL表达式中存在的类型错误。

此外，为了获得XML节点有关结构、内容的约束信息，笔者还改写了GeoQuery原有的XML Schema导入算法，使得该算法能将XML Schema中的类型信息转化为FXQL类型表示，保存到系统中，供静态类型检查模块以及后继模块使用。

由于GeoQuery系统仍在开发过程中，因此，系统中仍有很多不足和有待完善的地方，就静态类型检查而言，下面的功能还有待实现：

1. 对提取树模式之后的表达式进行类型检查

实际上，XQuery到FXQL的“翻译”是分两步进行的：第一步，将XQuery翻译为相应的FXQL表达式；第二步，对第一步翻译的结果进行处理，提取出“树模式”，以便对FXQL表达式进行进一步的优化。限于时间和精力，本文所设计并实现的静态类型检查，所处理的是第一步翻译所得的FXQL表达式。而在第二步提取出“树模式”后，所得的FXQL表达式与第一步翻译的结果有所区别。因此，在后续的版本中，静态类型检查应该将第二步翻译后所得到的FXQL表达式作为处理对象。

1. 对日期类型的支持

XQuery的类型系统支持日期类型（如xs:dateTime），但由于GeoQuery执行引擎尚未提供对日期类型的支持，因此本文所设计并实现的FXQL类型系统也就没有包含日期类型，静态类型检查也没有提供相应的支持。

# 致 谢

首先，我要感谢苏航老师。作为我的毕设指导老师，从背景知识的学习，算法的设计，系统的实现，再到论文的撰写，苏老师都给予了我充分的指导与帮助，总是能让我对自己所要做的工作能够有一个整体上的认识与把握。

其次，我要感谢廖湖声老师。廖老师做事严谨认真，在百忙之中抽出时间，检查我的毕设进展情况，并指出其中所存在的问题，多次的面对面指导，使我充分认识到自己哪些地方存在不足，让我感到受益匪浅。

另外，我还要感谢高红雨老师，感谢他在这个学期中，及时地向我们发布各种有关毕业的通知、注意事项，使我们能顺利地走过大学生活中的最后一个学期。

此外，我还要特别感谢GeoQuery课题组中的张晓博，李燕宾两位学长，感谢他们耐心细致地向我讲解了他们所负责的功能模块。同时，也要感谢罗卿和陈海涛两位学长，正是有了他们的热心帮助，才使得我对XML Schema有了比较清楚的认识，帮助我解决了Schema导入部分中碰到的问题。

最后，感谢论文评审委员会的老师对我论文的指正。

# 主要参考文献

1. Erik T.Ray. XML入门. 北京：中国电力出版社，2001年11月
2. Priscilla Walmsley. XQuery权威指南. 北京：电子工业出版社，2009年4月
3. 任宇. 面向空间数据集成的XQuery语言与执行引擎技术的研究，2005年5月
4. Chelsea Valentine, Lucinda Dykes, Ed Tittel. XML Schema数据库编程指南. 北京：电子工业出版社， 2002年5月
5. Ashok Malhotra, Jim Melton, Norman Walsh, etc. XQuery 1.0 and XPath 2.0 Functions and Operators(Second Edition).

http://www.w3.org/TR/2010/REC-xpath-functions-20101214/ ,2010年12月

1. Denise Draper, Michael Dyck, Peter Fankhauser, etc. XQuery 1.0 and XPath 2.0 Formal Semantics(Second Edtion). http://www.w3.org/TR/2010/REC-xquery-semantics-20101214/ ,2010年12月
2. 胡轶. 基于Schema的XQuery静态类型检查机制的设计与实现，2005年6月
3. 廖伟. 面向空间数据集成的XQuery语言与查询分解技术的研究，2005年5月
4. 张晓博，廖湖声 支持XML查询代数和树模式查询的XQuery系统框架，2010年4月
5. Tim Bray, Dave Hollander, Andrew Layman, etc. Namespaces in XML 1.0(Third Edition). www.w3.or/TR/xml-names，2009年8月
6. 丁璇. XML正规式类型类体系设计，2011年2月
7. 杨科朝. FXQL2.0抽象语法树类体系设计文档，2009年3月
8. Jonathan Robie, Don Chamberlin, Michael Dyck, etc. XQuery 1.0: An XML Query Language(Second Edtion). http://www.w3.org/TR/2010/WD-xquery-30-20101214/， 2010年12月
9. Michael Rorke, Karuna Muthiah, Ravindranath Chennoju, etc. XML Query Test Suite. http://dev.w3.org/2006/xquery-test-suite/PublicPagesStagingArea/ ，2010年9月
10. 万常选，刘喜平. XML数据库技术. 北京：清华大学出版社，2008年

# 附录 测试用例

下面给出在功能测试中所用到的W3C测试用例，共计48个，所用编号与W3C用例规范完全一致。因篇幅所限，用例只列出了用例名和测试目标。

（1）有关空类型的测试

|  |  |
| --- | --- |
| 用例名 | 测试内容 |
| generalexpressionst1.xq | 两个空序列做普通比较运算“=” |
| generalexpressionst100.xq | 两个空序列做普通比较运算“!=” |
| generalexpressionst101.xq | 空序列和原子值做普通比较运算“!=” |
| generalexpressionst199.xq | 两个空序列做普通比较运算“<” |
| generalexpressionst2.xq | 空序列和原子值做普通比较运算“=” |
| generalexpressionst200.xq | 空序列和原子值做普通比较运算“<” |
| generalexpressionst298.xq | 两个空序列做普通比较运算“<=” |
| generalexpressionst299.xq | 空序列和原子值做普通比较运算“<=” |
| generalexpressionst397.xq | 两个空序列做普通比较运算“>” |
| generalexpressionst398.xq | 空序列和原子值做普通比较运算“>” |
| generalexpressionst496.xq | 两个空序列做普通比较运算“>=” |
| generalexpressionst497.xq | 空序列和原子值做普通比较运算“>=” |

（2）有关FLWOR表达式的测试

|  |  |
| --- | --- |
| 用例名 | 测试内容 |
| statictyping-1.xq | 含有“where”子句的FLWOR表达式，算法加法运算 |
| statictyping-2.xq | abs函数 |
| statictyping-3.xq | avg函数 |
| statictyping-4.xq | max函数 |
| statictyping-5.xq | min函数 |
| statictyping-6.xq | sum函数 |
| statictyping-7.xq | boolean函数、true函数的测试 |
| statictyping-8.xq | 算术减法，普通比较运算“=” |
| statictyping-9.xq | 算术乘法 |
| statictyping-10.xq | 算术除法div的测试 |
| statictyping-11.xq | 整数除法idiv的测试 |
| statictyping-12.xq | 取模运算mod的测试 |
| statictyping-13.xq | 含有“let”子句的FLWOR表达式，值比较运算eq |
| statictyping-14.xq | 值比较运算le |
| statictyping-15.xq | 值比较运算lt |
| statictyping-16.xq | 值比较运算gt |
| statictyping-17.xq | 值比较运算ne |
| statictyping-18.xq | 对序列进行算术加法运算 |
| statictyping-19.xq | 对序列进行算术减法运算 |
| statictyping-20.xq | 对序列进行算术乘法运算 |
| statictyping-21.xq | 含有“let”子句的FLWOR表达式中使用了未声明变量 |
| statictyping-22.xq | 序言部分的变量声明中使用了未声明变量 |
| statictyping-23.xq | 对嵌套的FLWOR表达式使用了未声明变量 |
| statictyping-24.xq | not函数的测试 |

（3）路径表达式的测试

|  |  |
| --- | --- |
| 用例名 | 测试内容 |
| ST-Axes007.xq | 对属性节点做descendant-or-self轴操作 |
| ST-Axes008.xq | 对属性节点做descendant-or-self轴操作和名称测试 |
| ST-Axes009.xq | 对属性节点做descendant-or-self轴操作和名称测试，其中名称测试使用该属性节点的名字作为测试内容 |
| ST-Axes010.xq | 对文本节点（text node）做descendant-or-self轴操作 |
| ST-Axes011.xq | 对属性节点做attribute轴操作 |
| ST-Axes012.xq | 对文档节点做attribute轴操作 |
| statictypingaxis-1.xq | 对非节点类型做child轴操作 |
| statictypingaxis-2.xq | 对非节点类型做self轴操作 |
| statictypingaxis-3.xq | 对非节点类型做attribute轴操作 |
| statictypingaxis-4.xq | 对非节点类型做parent轴操作 |
| statictypingaxis-5.xq | 对非节点类型做descendant轴操作 |
| statictypingaxis-6.xq | 对非节点类型做descendant-or-self轴操作 |

# 附录 Schema导入设计方案

**一、FXQL类型系统说明**

FXQL类型系统的设计请参见“”，下面给出进一步的说明，以便阐明Schema导入的设计。

论文的正文中曾提到过，FXQL类型系统中REType及其子类主要用于表示XML Schema中保存的用户自定义类型，这些类的设计是基于丁璇的XML正规式类型类体系设计，做了一些修改得到的，REType及其子类具体的类设计如“附图 1”所示。



附图 1 REType及其子类设计

所修改的内容如下：

（1）SimpleType加入Type类型变量type：用于保存Schema中的简单类型定义。

（2）XMLType中加入ElementType类型变量parent：该变量用于保存父子关系，方便查找当前元素或属性的父元素。

（3）添加TypeDefName：在Schema中，一个元素声明是另一个schema中已定义的类型时，用于保存这个元素与另一个类型定义的关系。下面举例说明。

附表 1 一个Schema文档的片段

|  |
| --- |
| <xs:complexType name=*"personinfo"*>  <xs:sequence>  <xs:element name=*"name"* type=*"xs:string"*/>  ……  </xs:sequence>  <xs:attribute name=*"id"* type=*"xs:int"* use=*"required"*/>  </xs:complexType>  <xs:element name=*"person"* type=*"personinfo"*/> |

对于“”中的person元素，其类型为personinfo，因此person元素的content就应该保存一个TypeDefName对象，表明这个元素的具体类型声明需要根据TypeDefName保存的名字，到类型定义哈希表中查找。

REType及其子类的简要说明如“附表 2”所示。

附表 2 REType以及子类的说明

|  |  |  |
| --- | --- | --- |
| 类 | 属性 | 说明 |
| REType |  | 正则类型基类。 |
| XMLType | String name：保存元素或属性声明的名字  ElementType parent：保存父亲元素 |  |
| TypeItem | 无 |  |
| ElementType | TypeItem content：保存该元素的具体内容  ArrayList attrList：保存该元素的属性列表 | 保存元素节点的类型信息。 |
| AttributeType | boolean required：保存该属性在元素中是否为“required”  TypeItem attrType：保存属性的类型（SimpleType或是TypeDefName） | 保存属性节点的类型信息。 |
| StarItem | ElementType contentType：保存对应元素 |  |
| CountItem | int minOccur：元素最少出现次数  int maxOccur：元素最大出现次数 | maxOccur = -1表示该元素的maxOccur刻面的值为”unbounded”。 |
| OptionItem | 无 |  |
| PlusItem | 无 |  |
| ItemList | TypeItem[] items：保存合成器的内容 | 合成器的基类。 |
| AllList | 无 | 用于表示Schema中的“all”合成器。 |
| SequenceList | 无 | 用于表示Schema中的“sequence”合成器。 |
| ChoiceList | 无 | 用于表示Schema中的“choice”合成器。 |
| SimpleType | Type simpleType：保存简单类型  String simpleTypeName：保存该简单类型的名字 | 用于保存简单类型。 |
| TypeDefName | String typeDefName：保存类型定义的名字 | 若一个元素声明的内容是一个TypeDefName对象，利用这个typeDefName到全局类型定义哈希表中查找即可。 |

**二、用例说明**

假设现在有文件”store.xsd”，其中的schema定义如“”所示。

附表 3 Schema示例

|  |
| --- |
| <?xml version="1.0" encoding="ISO-8859-1" ?>  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">  <xsd:element name='store' type='storeType'/>  <xsd:complexType name='storeType'>  <xsd:sequence>  <xsd:element name='book' type='bookType' minOccurs='0'  maxOccurs='unbounded'>  </xsd:element>  </xsd:sequence>  <xsd:attribute ref='name' use="required" />  </xsd:complexType>  <xsd:complexType name='bookType'>  <xsd:sequence>  <xsd:element name='name' type='xsd:string'/>  <xsd:element name='author' minOccurs='1' maxOccurs='unbounded'/>  </xsd:sequence>  <xsd:attribute name='id' use="required" type="xsd:long" />  <xsd:attribute name='lang' use="optional" type="xsd:string" />  </xsd:complexType>  <xsd:attribute name='name' type="xsd:string" />  </xsd:schema> |

那么，以上述的类型体系表示，如“”、“”所示：

***注：限于篇幅，没有在“”中标明元素、属性的父子关系，但实际编码时肯定会保存这些信息。***

**ElementType**

**String:name**

**TypeItem: content**

**AttributeType []:attrList**

**ElementType： parent**

**“store”**

**SequenceList**

**TypeItem[]:items**

**AttributeType**

**String:name**

**Boolean：required**

**TypeItem：attrType**

**ElementType：parent**

**SimpleType**

**Type:SimpleType**

**String:simpleTypeName= null**

**True**

**“name”**

**StringType**

**attrList[0]**

**ElementType**

**String:name**

**TypeItem:content**

**AttributeType[]:**

**attrList**

**ElementType：parent**

**“book”**

**CountItem**

**ElementType: contentType**

**Int:minoccur=0**

**Int:maxoccur=-1**

**items[0]**

<store nane=’a’>

<book ID=1234 lang=EN>

<name>XXX</name>

< author >XXX</author >

< author >XXX</author >

< author >XXX</author >

</book>

……

<book>

……

</book>

</store>

**-1表示正无穷**

**保存对父元素的引用**

**null**

**全局元素类型定义中，父元素为空**

**保存对父元素的引用**

**“book”元素的具体表示参见“**附图 3**”。**

附图 2 store.xml的REType表示（1）

**ElementType**

**String:name=”book”**

**TypeItem:content**

**AttrType[]: attrList**

**ItemList**

**TypeItem[]:items**

**AttributeType**

**String:name=”ID”**

**Boolean：required=”True”**

**SimpleType：type**

**SimpleType**

**Type: simpleType**

**String:name=null**

LongType

**attrList[0]**

**attrList [1]**

**ElementType**

**String:name=”name”**

**TypeItem: content**

**AttrType[]:attrlist**

**AttributeType**

**String:name=”lang”**

**Boolean：required =”false”**

**TypeItem：type**

**SimpleType**

**Type:simpleType**

**String:name=null**

**StringType**

**SimpleType**

**Type:simpleType**

**String:name=null**

**StringType**

**ElementType**

**String:name=”author”**

**TypeItem: content**

**AttrType[]:attrlist**

**SimpleType**

**Type:simpleType**

**String:name=null**

**StringType**

**PlusItem**

**ElementType: contenttype**

**CountItem**

**ElementType:**

**contentype**

**Int:minoccur=1**

**Int:maxoccur=1**

**items[0]**

**items[1]**

<name>

XXXX

</name>

<author>

XXXX

</author >

<book ID=1234 lang=’EN’>

<name>XXX</name>

< author >XXX</author >

< author >XXX</author >

< author >XXX</author >

</book>

**1~n个author**

**表示必须存在一个name**

**必须存在一个名为ID的属性**

**一个名为lang的可选属性**

附图 3 store.xml的REType表示（2）

**三、数据结构设计**

**1、对单个Schema的保存**

创建SchemaTypes类保存单个Schema的内容，“”中所示，这个类使用2个哈希表保存Schema内的信息，每一个哈希表的具体作用将在下面阐述。

1. **保存类型定义**

定义：Hashtable globalTypes

这张哈希表保存Schema内全局类型定义，具体结构如“”所示。

附表 4 globalTypes所保存的信息

|  |  |
| --- | --- |
| key | value |
| 第1个SimpleType的name | 第1个SimpleType对象 |
| …… | …… |
| 第n个SimpleType的name | 第n个SimpleType对象 |
| 第1个复杂类型定义的name | 第1个ComplexTypeContent对象 |
| …… | …… |
| 第m个复杂类型定义的name | 第m个ComplexTypeContent对象 |

复杂类型定义包括两方面内容：属性列表、除属性外的其他信息。因此定义一个类ComplexTypeContent，用于保存这些信息，具体设计如“”所示。



附图 4 ComplexTypeContent类设计

其中attrList用于保存属性列表，content用于保存除属性外的其他信息。

1. **保存节点声明**

定义：Hashtable globalNodes

这张哈希表保存一个Schema中的节点类型声明，哈希表结构如“附表 5”所示。

附表 5 globalNodes所保存的信息

|  |  |
| --- | --- |
| key | value |
| 第1个ElementType的name | 第1个ElementType对象 |
| …… | …… |
| 第n个ElementType的name | 第n个ElementType对象 |
| 第1个AttributeType的name | 第1个AttributeType对象 |
| …… | …… |
| 第m个AttributeType的name | 第m个AttributeType对象 |
| 第1个AttributeGroup的name | 第1个AttributeType对象的链表 |
| …… | …… |
| 第p个AttributeGroup的name | 第p个AttributeType对象的链表 |
| 第1个Group的name | 第1个ElementType对象的链表 |
| …… | …… |
| 第q个Group的name | 第q个ElementType对象的链表 |

因此，“”中的用例，对应的globalNodes内容如下所示：

附表 6 对应于store.xsd的globalNodes哈希表

|  |  |
| --- | --- |
| key | Value |
| “store” | ElementType对象 |
| “name” | Attribute对象 |

而“store”对应的ElementType对象的具体结构参见“二、用例说明”中的图示。

**2、所有schema的保存**

创建SchemaPool类保存所有导入的schema信息，如“”中所示，由于可能会导入多个Schema，这个类中包含一个哈希表schemaPool，用于保存所有当前导入的Schema，schemaPool结构如“”所示。

附表 7 schemaPool哈希表所保存的信息

|  |  |
| --- | --- |
| key | value |
| 第1个Schema的URI | 保存第1个Schema的哈希表  （SchemaTypes类对象） |
| …… | …… |
| 第n个Schema的URI | 保存第n个Schema的哈希表  （SchemaTypes类对象） |

之所以使用schema的URI作为关键字，是因为两个不同的Schema，其名称空间可能是相同的，若用名称空间为关键字，可能会发生冲突。

# 附录 FXQL静态类型错误整理

目前所实现的FXQL静态类型检查中，所涉及到的类型错误如“”所示。

附表 8 FXQL静态类型错误

|  |  |
| --- | --- |
| 错误代码 | 说明 |
| XPTY0004 | 表达式的静态类型与当前上下文不匹配。 |
| XPST0005 | 除()（空表达式）和data(())之外的表达式被赋予空类型。 |
| XPST0008 | 表达式引用的元素名，属性名，schema类型名，命名空间前缀或是变量名在上下文没有定义。 |
| XPST0017 | 函数调用中参数的数量与静态上下文中的定义不符。 |
| XPTY0019 | 路径表达式的某一步（不是最后一步）是原子值。 |
| XPTY0020 | 指定了相对路径表达式，但当前项不是一个节点。 |
| XQST0045 | 试图在xml、xs、xsi、fn等命名空间中声明自己的函数。 |
| XQST0055 | 序言中包含有超过一个的拷贝命名空间声明。 |
| XPST0081 | 查询中的限定名QName具有一个未声明的前缀，或在一个作用域外引用的前缀。 |

# 附录 原子类型的类设计

FXQL类型系统实现了30多种原子类型，具体的类设计参见“”。



附图 5 FXQL原子类型类设计

# 附录 路径表达式分析处理过程说明

**一、数据说明**

首先给出需要用到的示例文件的具体内容。

（1）假设有Schema文件store\_path.xsd，其具体内容如“”所示。

附表 9 用于说明路径表达式的Schema实例

|  |
| --- |
| <?xml version=*"1.0"*?>  <xsd:schema xmlns:xsd=*"http://www.w3.org/2001/XMLSchema"*>  <xsd:element name=*"book"*>  <xsd:complexType>  <xsd:sequence>  <xsd:element name=*"author"* type=*"xsd:string"*>  <xsd:complexType>  <xsd:sequence>  <xsd:element name=*"firstName"* type=*"xsd:string"*></xsd:element>  <xsd:element name=*"midName"* type=*"xsd:string"*></xsd:element>  <xsd:element name=*"familyName"* type=*"xsd:string"*></xsd:element>  </xsd:sequence>  <xsd:attribute name=*"id"* type=*"xsd:integer"*></xsd:attribute>  </xsd:complexType>  </xsd:element>  <xsd:element name=*"title"* type=*"xsd:string"*/>  <xsd:element name=*"description"* type=*"xsd:string"*/>  </xsd:sequence>  </xsd:complexType>  </xsd:element>  <xsd:element name=*"store"*>  <xsd:complexType>  <xsd:sequence>  <xsd:element ref=*"book"*></xsd:element>  </xsd:sequence>  </xsd:complexType>  </xsd:element>  </xsd:schema> |

（2）store\_path.xml文件内容如“”所示。

附表 10 用于说明路径表达式的XML文档

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <storeNS:store xmlns:storeNS="http://www.example.com/storeNS"  xmlns:xs="http://www.w3.org/2000/10/XMLSchema-instance"  xs:schemaLocation="store\_path.xsd">  <storeNS:book>  <storeNS:title>The Code of Beauty</storeNS:title>  <storeNS:description>the content of the book ……  </storeNS:description>  <storeNS:author id="123">  <storeNS:firstName>John</storeNS:firstName>  <storeNS:midName>D</storeNS:midName>  <storeNS:familyName>Lucy</storeNS:familyName>  </storeNS:author>  </storeNS:book>  </storeNS:store> |

**二、路径表达式处理过程说明**

路径表达式的数据源主要由两种：

1. 使用“doc()”函数获得所要查询的数据，例如：

doc(“store\_path.xml”)/book/author/@id。

1. 使用之前声明的变量作为数据源，例如：

$var/book/author

其中的“$var”是在路径表达式之前声明并被赋值的变量，它保存了所要查询的节点类型。

对于第（2）种情况，除了获得节点类型的方式不同外，对路径表达式的处理过程与第（1）种情况下完全相同，因此下面主要以第（1）种情况为例，说明对路径表达式的处理过程。并且，正如之前所提到的，目前在GeoQuery系统中，为了便于优化，ChildAxis、DescendantAxis、DescendantOrSelfAxis、AttributeAxis会被翻译为相应的“Step”表达式，其他的轴操作则是以原语函数的形式表示，这里仅针对能翻译为“Step”表达式的4种轴操作进行说明，其他轴操作的处理的思路与这4种轴操作类似，不再做详细说明。

假设现在要处理如下XQuery查询语句：

**doc(“store.xml”)/book/author/@id**

该查询语句被翻译为FXQL表达式后，翻译结果如“”所示。

附表 11 路径表达式翻译结果

|  |
| --- |
| $var/element(book)/element(author)/@attribute(id)  where $var:= fs:doc("store\_path.xml") |

要处理上面的FXQL表达式中，首先需要做的就是根据“doc()”函数的参数值“store\_path.xml”，进行XML Schema的导入操作，并保存变量“$var”到这个Schema的映射关系。因此，上下文环境中应该设有一个映射表来保存“<变量名 → Schema内容>”这一映射关系。为方便说明，预定义如“”所示。

附表 12 预定义变量

|  |  |
| --- | --- |
| 名称 | 说明 |
| varToSchema | 用于保存“<变量名 → Schema内容>”这一映射关系的映射表。 |
| nsToSchemaURI | 用于保存“<名称空间 → SchemaURI>”这一映射关系的映射表。 |
| sourceType | 用于表示保存当前需要进行轴操作的节点类型。 |

由于翻译结果是一个where表达式，因此，对路径表达式的处理分两步进行：第一步，处理“where”关键字后面的绑定表达式；第二步，处理“where”关键字之前的表达式。具体过程如下：

1. **变量绑定部分的分析：**根据doc函数的值，读取相应XML文件，看是否给出了对应的Schema所在的位置。如果给出了Schema的URI（XML中用“schemaLocation”属性来保存），那么据此找到对应的Schema文件，将其导入，并将Schema的名称空间到SchemaURI的映射保存到映射表nsToSchemaURI中，将变量名到Schema内容这一映射保存到映射表varToSchema中。
2. **表达式部分的分析：**根据变量名，从上下文中获取将要进行轴操作的节点类型sourceType，按顺序逐个处理路径表达式（“*Id <Step>+*”）中的每一个“Step”，不同的“Step”表达式，处理思路如下（下文中涉及到FXQL路径表达式的type、nodeType、candidateType参见“”中的说明）：
3. 对Child Step：将sourceType的孩子元素保存到当前Step表达式的candidateNodes中。
4. 对Descendant Step：将sourceType所有后代元素加入到当前Step表达式candidateNodes中。（简单起见，假设元素没有重名，如果子元素的名字已经保存过，说明是递归调用，则不用再保存。）
5. 对DescendantOrSelf Step：将sourceType所有后代元素以及sourceType自身加入到Step表达式的candidateNodes。（简单起见，假设元素没有重名，如果子元素的名字已经保存过，说明是递归调用，则不用再保存。）
6. 对Attribute Step：将当前元素节点sourceType的属性保存到Step表达式的candidateNodes
7. 对其他Step：FXQL会将其翻译对应的函数调用，如ancestor::author会翻译为ancestor(author)，因此按照对应的函数调用进行分析即可，这里不再赘述。

需要说明的是，在对路径表达式的分析过程中，一般只需要使用candidateNodes来进行对下一“步”（“Step”）的分析，在所有“Step”都分析完后，再根据candidateNodes中的内容，给整个路径表达式的type、nodeType赋值即可。

以“”为例，具体的处理过程如下：

1. **变量绑定部分的分析：**得到文件名为”store\_path.xml”后，可以得知其对应的SchemaURI为"store\_path.xsd"，则将该Schema导入，并将映射“http://www.example.com/storeNS → store\_path.xsd”保存到nsToSchemaURI中，将映射“var → store\_path.xsd内容”保存到varToSchema中，其中的“store\_path.xsd内容”包含有如下的映射表，以保存这个Schema内的所有类型信息。

附表 13 “store\_path.xsd”的内容

|  |  |
| --- | --- |
| 关键字 | 值 |
| book | book元素对应的ElementType对象 |
| store | store元素对应的ElementType对象 |

1. **表达式部分的分析：**根据翻译规则可知，易知“”由三个“Step”组成，依次保存到数组steps中，分析过程以伪代码的形式给出，参见“附表 14”。

附表 14 对路径表达式的处理实例

|  |
| --- |
| //**第一步**：从上下文中获取路径表达式要处理的节点类型，可以得到如下结果。  curCandidateNodes = getSourceType();包含book、store两个元素声明信息的哈希表  //**第二步**：按顺序逐个处理每个“Step”，其中，steps用于表示保存所有“Step”  //的数组。  **//对steps[0]处理(即对“当前信息/ book”的处理)**  steps[0].candidateNodes = getAxisNodeTest(curCandidateNodes,  steps[0].axis, steps[0].nodeTest);  //进行轴操作和节点测试, steps[0]的candidateNodes此时仅包含book元素节点。  **//重置“当前”类型信息**  curCandidateNodes = steps[0].candidateNodes ;  //此时的curCandidateNodes包含book元素声明的信息, 即book对应的 //ElementType对象  **//对steps[1]处理(即对“当前信息/ author”的处理)**  steps[1].candidateNodes = getAxisNodeTest(curCandidateNodes,  steps[1].axis, steps[1].nodeTest);  //获取steps[1]的结果，此时steps[1].candidateNodes将包含author元素的类型信 //息，即author对应的ElementType对象  **//重置当前类型信息**  curCandidateNodes = steps[1].candidateNodes ;  //此时的curCandidateNodes包含author元素声明的信息  **//对steps[2]处理(即对“当前信息/ @id”的处理)**  steps[2].candidateNodes = getAxisNodeTest(curCandidateNodes,  steps[2].axis, steps[2].nodeTest);  //steps[2]. candidateNodes将包含id属性类型信息，即属性id对应的AttributeType //对象  **//第三步：**获取最终结果  //最后一个”step”的结果，即为整个路径表达式的最终结果  path.type = collectType(steps[2]. candidateNodes);  //此时的path.type保存的是Type.ATTRIBUTE  path.nodeType = collectNodesType(steps[2]. candidateNodes);  //此时的path.nodeType保存的是“id”对应的AttributeType对象  path.candidateNodes = steps[2].candidateNodes  **辅助说明：**  curCandidateNodes：用于保存当前需要处理的候选节点。  path:表示整个FXQL路径表达式  getSourceType()：用于从上下文中获得路径表达式所要处理的节点的类型信息。  getAxisNodeTest(arg1, arg2, arg3)：用于进行轴操作和节点测试，以便得到所有候选节点的辅助函数，arg1表示当前要处理的所有候选节点的集合，arg2表示当前要对arg1中所有节点进行的轴操作的标志位，arg3表示节点测试。具体算法与“表 3‑10”类似，限于篇幅，不再给出详细说明。  collectType(arg1)：用于将候选节点的集合转化成合适的FXQL类型表示（arg1表示作为候选节点集合）：   * + 1. 若集合中只有一个节点，则根据这个节点的类型，返回相应的NodeType类型（比如说，该节点是一个ElementType对象，则返回Type.ELEMENT，参见“”）；     2. 若有多个节点，则创建一个FXQL序列，并将对应于这些节点的NodeType类型加入到序列中（对应关系参见i.中阐述内容）；     3. 若集合为空，则返回FXQL中的空类型“empty”，表示没有任何结果。   collectNodesType(arg1)：用于将候选节点的集合转化成合适的FXQL类型表示（arg1表示作为候选节点集合）：   * + 1. 若集合中只有一个节点，则直接将该节点返回；     2. 若有多个节点，则创建一个FXQL序列，并将所有这些节点加入到序列中；     3. 若集合为空，则返回FXQL中的空类型“empty”，表示没有任何结果。 |