**毕业论文**

1. 绪论

1.1 课题研究背景及意义 BIM数据的增长、BIM产业的发展、数据共享的重要性

1.2 国内外研究现状 关系数据库、BIMServer、内存数据库redis、Hbase的技术、基于web的技术；关系数据处理研究、引出图数据库

1.3 本文贡献 基于es存储属性数据、基于neo4j存储空间结构、改进neo4j路径查询

1.4 组织结构

第二章 相关技术基础

2.1 IFC相关技术

2.1.1 IFC标准

2.1.2 IFC文件格式 （不适用schema，而是使用内存中的树结构）

2.2 全文检索技术

2.2.1 全文检索基础理论

2.2.1 Lucence概述

2.2.2 ElasticSearch技术（全文检索、关键词检索）

2.3 图数据库技术

2.3.1 图的概念

2.3.2图数据的发展

2.3.3 Neo4j概述

第三章 基于ElasticSearch的IFC对象属性数据的存储和检索研究

3.1 IFC对象属性数据模型概述

3.1.1 IFC对象属性数据

3.2 数据提取技术

3.2.1 IFC文件解析

3.2.2 JSON数据生成

3.3 数据索引技术

3.3.1数据模式映射

3.3.2索引过程

3.3.3索引优化（批量索引bulk）

3.4 数据检索技术

3.4.1 全文检索

3.4.2 关键词检索

3.5 自定义ID字段

第四章 基于Neo4j的IFC对象空间关系的存储和查询研究

4.1 IFC对象空间关系模型

4.2 图数据模型设计

4.3 图数据提取技术

4.3 图数据索引技术

4.4 图数据检索技术

4.5 基于BloomFilter的路径查询优化

第五章 系统集成设计与实现

5.1 系统框架

5.2 文件上传模块

5.3 属性数据检索模块

5.4 空间数据检索模块

5.5 数据展示模块

第六章 系统实验

6.1 属性数据的索引与检索

6.1.1 索引效率

6.1.2 检索效率

6.2 空间关系数据的索引与检索

6.2.1 索引效率

6.2.2 检索效率

6.3 系统集成测试

第七章 总结与展望

应该只做属性数据的管理。否则的话，需要考虑信息存储的完整性。而在本文里，想属性信息的其它字段是不需要的，只留下名称和值。

# 第三章 基于ElasticSearch的IFC对象属性数据的存储和检索研究

本文在绪论中已经介绍了IFC标准。在本章节将详细介绍IFC对象的属性数据。

## 3.1 IFC对象属性数据模型概述

### 3.1.1 属性

属性就是对事物以及概念描述的一个基本单位。在IFC标准中，除了IFC对象类型本身的各个字段可以称之为属性，IFC标准还定义了不同的属性类，为描述IFC对象提供了一种更加灵活的可扩展的方式。这些属性类均继承自抽象的基类IfcProperty。根据buildingsmart官网的文档， IfcProperty的定义如下图：

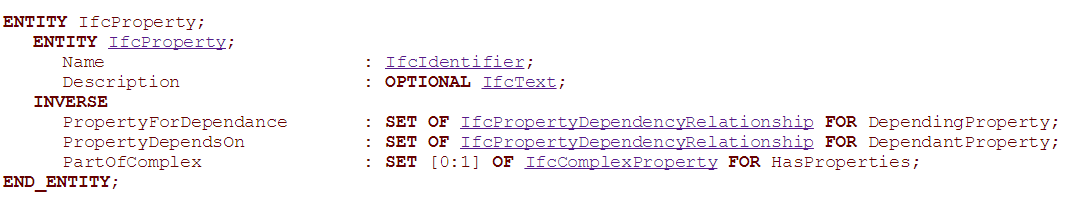


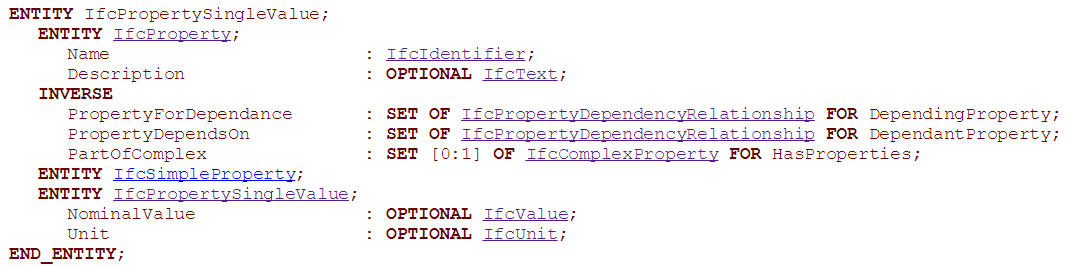
图1 IfcProperty定义

其中Name字段存储属性的名称，对应的值类型是IfcIdentifier，Description存储属性的说明，对应的值类型是IfcText。IfcProperty是抽象基类，它的派生子类如下图：

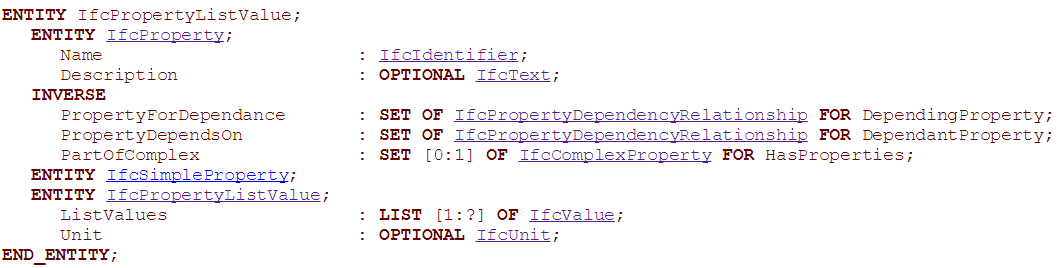


图2 IfcProperty派生子类图

其中，IfcSimpleProperty是简单属性。根据属性值类型的不同又有6个派生子类。比如，IfcPropertySingleValue面向单值属性，它的定义是：

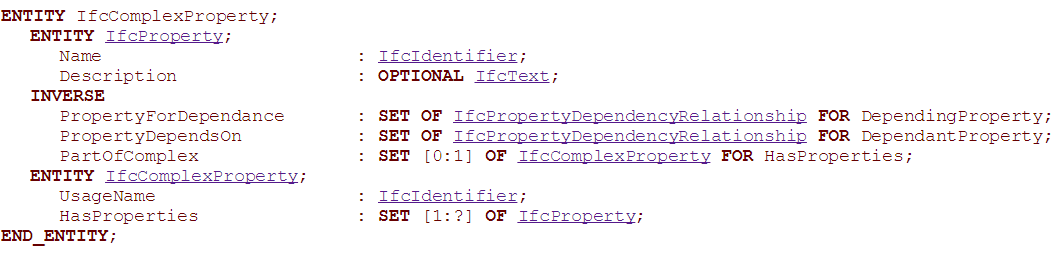


其中的NormalValue就是Name属性字段所对应的属性值。又比如IfcPropertyListValue，面向具有列表值类型的属性，它的定义是：



其中的ListValue就是Name字段所对应的属性值，是一个列表类型。其它如IfcPropertyBoundedValue、IfcPropertyEnumeratedValue、IfcPropertyReferenceValue、IfcPropertyTableValue分别面向上下边界区间类型、枚举类型、实体引用类型、二维表格结构类型的属性，再次不再一一介绍。

相对于IfcSimpleProperty的简单属性类型而言，IfcComplexProperty是复杂属性。它的定义如下图：



其中HasProperties存储了IfcProperty的集合，而UsageName是使用说明。IfcComplexProperty是可以嵌套的，集合中包含的IfcProperty也可以是一个IfcComplexProperty。

### 3.1.2 属性集

IFC标准用属性集来组织对象的属性。所有的属性集都继承自抽象基类IfcPropertySetDefinition。它的派生子类图如下：

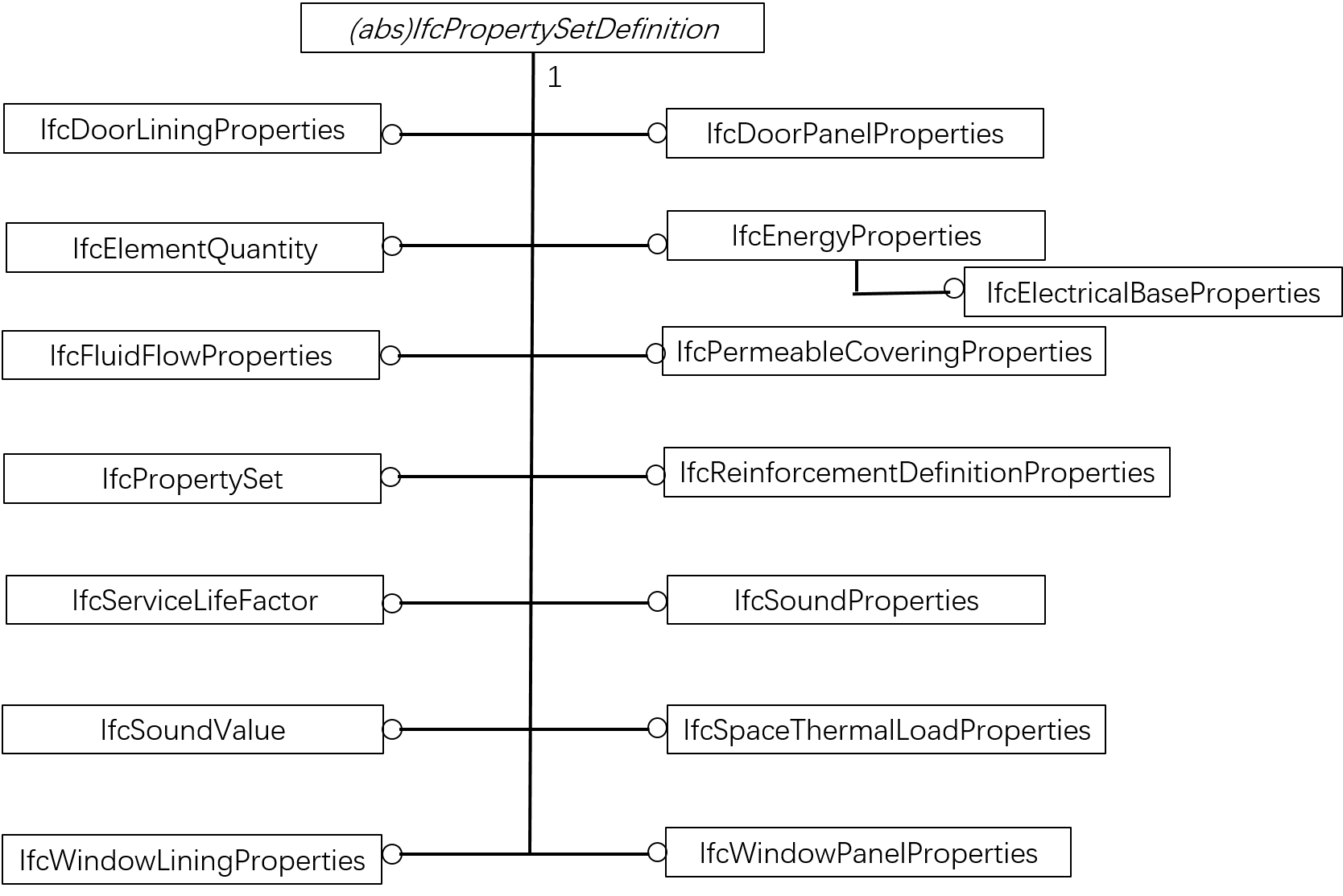
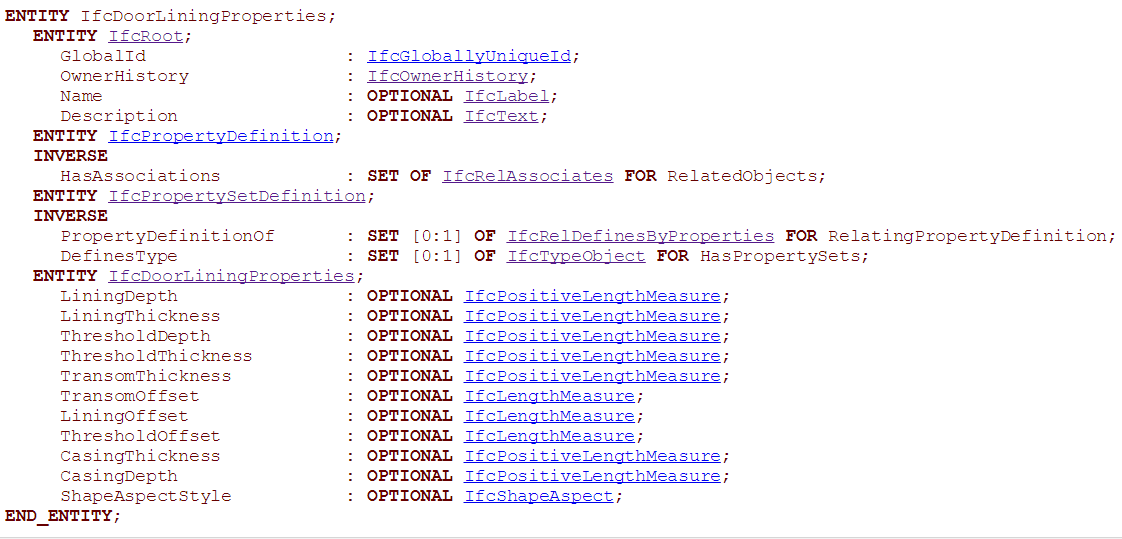


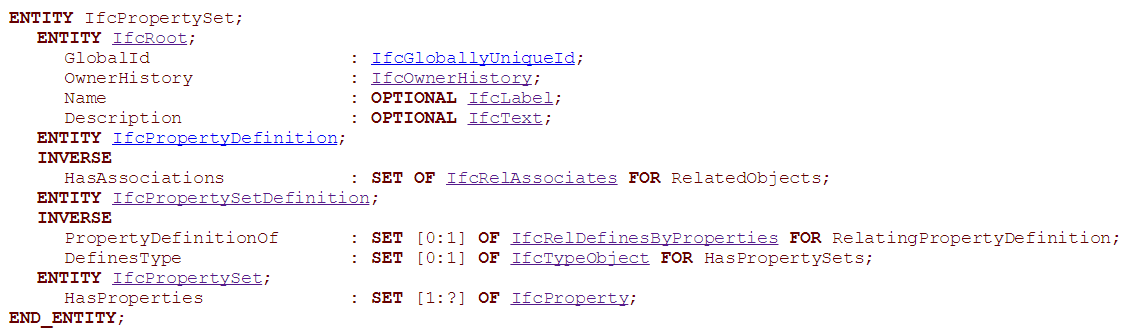
图3 IfcPropertySetDefinition派生类图

所有的这些属性集分为静态属性集和动态属性集。静态属性集往往面向特定的IFC实体类型，如IfcDoorLiningProperties应用于IfcDoorStyle类型，IfcWindowLiningProperties应用于IfcWindowStyle类型。举例IfcDoorLiningProperties的定义如下：



可以看到它包含了一系列的属性如LiningDepth、LiningThickness等来描述门框的信息。所以它是面向IfcDoorStyle的静态属性集。

事实上，以上派生类中除了IfcPropertySet之外都是静态属性集，在此不做一一介绍。IfcPropertySet是动态属性集。它的定义是：



其中HasProperties存储了一个IfcPropertySet的集合。

不要混淆了IfcComplexProperty与IfcPropertySet。前者作为IfcProperty的子类，属于属性范畴，后者是属性集。在IFC标准中需要通过IfcRelDefinesByProperties关系类型来将IFC对象类型和IfcPropertySet关联。也就是说，而IfcComplexProperty必须包含在一个属性集IfcPropertySet中才能关联到IFC对象。

### 3.1.3 属性集与IFC对象关联

在IFC标准中通过IfcRelDefinesByProperties关系实体将IfcPropertySetDefinition描述的信息与IFC对象关联。这种关联是一对多的映射，多个具有相同属性集的IFC对象可以通过同一个IfcRelDefinesByProperties关联到一个属性集。当然，一个IFC对象可以通过多个IfcRelDefinesByProperties来关联到不同的属性集。如图3所示：

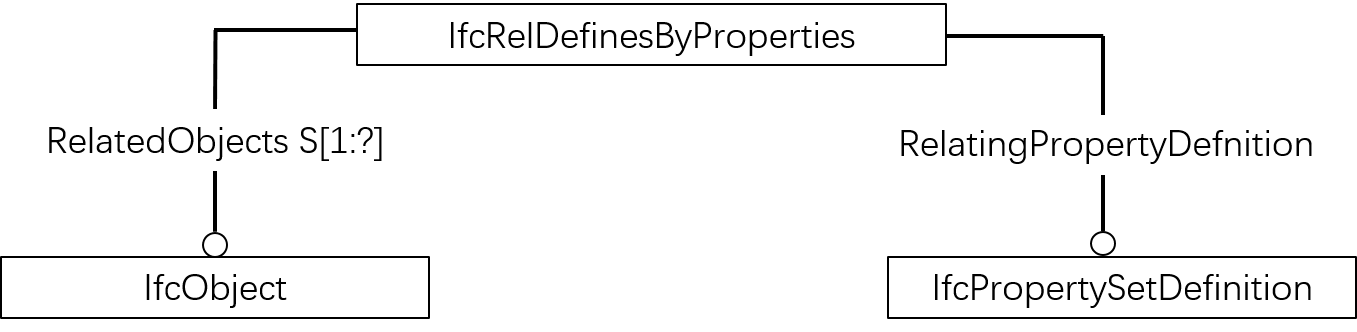


图3 IfcRelDefinesByProperties关联机制

需要举一个具体的例子。

以上便是IFC对象属性数据模型的介绍。对于如何将属性数据存储狭隘，接下来将会讲述。

## 3.2 数据提取技术

本小节将详细介绍从解析IFC文件到内存中的JAVA对象，到ElasticSearch所支持的JSON数据的过程。

### 3.2.1 IFC文件解析

本文在2.1小节中对比了IFC模型解析工具IFC Java ToolBox和BimServer解析模块，最后选择前者来进行IFC文件的解析。

IFC Java ToolBox主要有三个组成部分：

1. 针对IFC实体类型建立的Java类；
2. IFC文件解析器，将文件内容解析为内存中的Java类型；
3. IFC对象模型IfcModel，提供所有IFC对象的访问；

原生的IFC Java ToolBox API并不能访问属性字段，所以我们需要定义一个attributes数组，存储每个对象类型的属性名，并通过Java反射机制来进行属性字段的访问。

举例分析如下：

下面以IfcWallStandardCase实例来进行JSON映射的介绍：

#155=IFCWALLSTANDARDCASE('0o\_qQXI0r5UQtbU$t93HI7',#41,'\X2\57FA672C5899\X0\:CW 102-50-215p:309591',$,'\X2\57FA672C5899\X0\:CW 102-50-215p:412',#124,#151,'309591');

#264=IFCPROPERTYSINGLEVALUE('\X2\65CF\X0\',$,IFCLABEL('\X2\57FA672C5899\X0\: CW 102-50-215p'),$);

#265=IFCPROPERTYSINGLEVALUE('\X2\65CF4E0E7C7B578B\X0\',$,IFCLABEL('\X2\57FA672C5899\X0\: CW 102-50-215p'),$);

#266=IFCPROPERTYSINGLEVALUE('\X2\7C7B522B\X0\',$,IFCLABEL('\X2\5899\X0\'),$);

#267=IFCPROPERTYSINGLEVALUE('\X2\7C7B578B\X0\',$,IFCLABEL('\X2\57FA672C5899\X0\: CW 102-50-215p'),$);

#268= IFCPROPERTYSINGLEVALUE('\X2\7C7B578B\X0\ ID',$,IFCLABEL('\X2\

57FA672C5899\X0\: CW 102-50-215p'),$);

#284=IFCPROPERTYSET('1jxqaKVd18p9OAJYxrc1qw',#41,'\X2\51764ED6\X0\',$,(#264,#265,#266,#267,#268));

#289=IFCRELDEFINESBYPROPERTIES('1j9gkIXEf1yvBd$sOSwe8K',#41,$,$,(#155),#284);

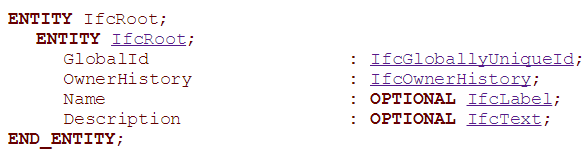
#155是IfcWallStandardCase对象。#264、#265、#266、#267、#268是四个IfcPropertySingleValue对象，分别表示单值属性。#284是IfcPropertySet对象，也就是一个动态属性集，根据IFC Schema定义，它的HasProperties字段存储了所包含的IfcProperty，在这里就是（#264,#265,#266,#267,#268）。#289是IfcRelDefinesByProperties对象，根据IFC Schema定义，RelatingPropertyDefinition表示一个属性集，在这里是#284，而RelatedObjects表示属性集关联到的IfcObject集合，在这里是只包含了一个元素（#155）的集合。

对这样的一个IFC实体，结合其自身Schema定义和属性集数据，将映射为下面的JSON格式：是不是可以将真是数据直接拿过来的？？

### 3.2.2 JSON数据映射

经过解析后的IFC对象数据需要转换为JSON数据，才能够通过Restful API被ElasticSearch索引。我们需要定义这个映射过程。

前面介绍过IFC标准，其分为四个功能层次，资源层、核心层、交互层、领域层。其中IfcRoot是核心层及以上层次中全部实体类型的抽象基类型。IfcRoot的定义如下：



其中GlobalId字段是一个全局唯一的GUID值。IfcRoot的所有派生类因为有了这个全局标识属性而具有了信息交换过程中的独立性。相比之下，资源层中的实体类型如IfcAddress（描述地址的实体），没有GlobalId属性，不能独立用于信息交换。而用户更为关系的是这些可用于交换的实体类型。IfcRoot的部分派生类如图4：

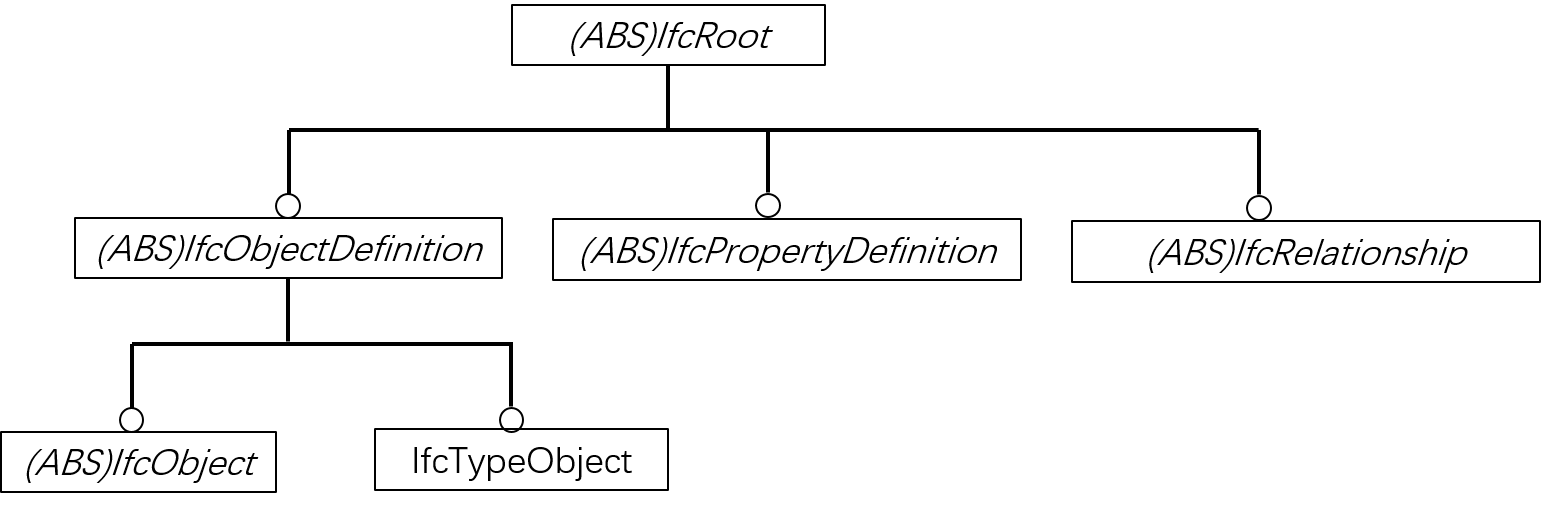


图4 IfcRoot部分派生类图

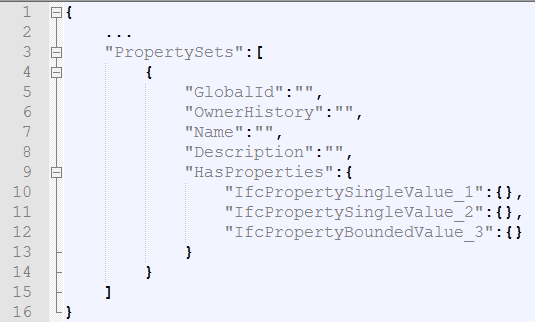
可以看到IfcRoot的直接派生类为IfcObjectDefinition、IfcPropertyDefinition、IfcRelationship三个类型。而3.1.3小节中讲到的IfcRelDefinesByProperties就是IfcRelationship的一个派生类，它将IfcObject类型与IfcPropertySetDefinition定义的属性集关联起来。由此，也就引出了本文ElasticSearch属性数据存储的重要研究对象：IfcObject。

本文的设计是将IfcObject的属性集数据集成到IfcObject本身，实现属性集的高效访问。而对于其它的IFC实体，按照IFC Schema定义中的字段一一映射为JSON格式。

设计的难点在于一个IfcObject可以关联多个IfcPropertySet实例，一个IfcPropertySet实例又关联了多个IfcProperty的派生类实例，而IfcProperty的派生类有6种，如3.1.1小节介绍，它们之间存在差异。

可能的设计方案：

1. 如果采用JSON对象嵌套的方法，如下：



其中行1表示省略IfcObject自身定义的字段，行3表示IfcPropertySet数组，行4-14表示一个IfcPropertySet对象，行9表示IfcPropertySet所关联的多个IfcProperty。这种方式能够完整地保留IfcPropertySet和IfcProperty的每一个字段。但是它的局限在于，需要为每一个IfcProperty实例命名，这个命名必须是数组内唯一的，这是非常困难的。另外，通过字段名访问数据也变得困难。

1. 第二种方案，侧重考虑用户对于字段的访问的便利。由于IfcPropertySet

的Name字段往往是用户关心的，所以可以把Name字段的值作为JSON中的字段名。对于IfcProperty也是一样，把Name字段的值作为JSON的字段名。如下图：



其中，“标识数据”是IfcPropertySet的Name字段的值，标识属性集名称，行4-6是IfcPropertySet的字段。“标高”是IfcPropertySingleValue的Name字段的值，表示属性名称，后面对应属性值，“偏移量”也是一样。这种方案使得用户能够根据日常知识方便地访问想要的数据。

但是它有一个问题是，Name字段不具有唯一性，也就是说可能有多个IfcPropertySet属性集名称都为标识数据。例如，属性集A，名称为“标识数据”，包含属性“标高”“偏移量”，属性集B，名称为“标识数据”，包含属性“类型名称”“部件代码”。在这种情况下希望得到的处理结果是，属性集A和属性集B的四个属性合并为一个属性集。如果是用第二种方案，那么会出现，行4、5、6的数据会因为重叠被覆盖掉。

经过对需求的考虑认为，GlobalId、Description、OwnerHistory在用户的查询中并无重要意义，所以去掉。

基于第二种方案，加上进行扩展，可以得到JSON格式如下：



图5 属性数据JSON格式

其中，行11-28是另一个名称为“Pset\_XXXXCommon”的属性集，行11-14是一个IfcPropertyBoundedValue属性实例，行15-17是一个

IfcPropertyEnumeratedValue属性实例，行18-20是一个IfcPropertyListValue属性实例，行21-23是一个IfcPropertyReferenceValue实例，行24-27是一个IfcPropertyTableValue属性实例。对于所有的字段，如果是引用类型，则只存储引用对象的ID值。另外，如果是IfcPropertySet的Name字段没有值，则存入“Others”JSON字段中。

同样对于IfcTypeObject也是如此。这段介绍应该放在属性集关联那里去。

如果是IfcComplexProperty类型，则嵌套表示。

其代码逻辑是：

1. 遍历IfcModel中的每一个IFC对象；
2. 针对每一个IFC类型创建JSON文件，以类型名为文件名称。由于解析IFC文件是逐行进行，所以需要维护一个Map结构存储文件句柄。
3. 针对解析后的每一个IFC对象，判断是否是IfcTypeObject或IfcRelDefinesByProperties的子类实例，如果都不是，则根据attributes属性名数组获取属性名XXXX，利用Java的反射机制执行getXXXX方法名，获取属性值，从而形成JSON格式字符串写入相应的文件。
4. 如果是IfcTypeObject的子类实例，那么可以对于“HasProperties”字段将遍历每一个所关联到的IfcPropertySet子集，按照上面图5所设计的JSON格式来进行字段和字段值的输出，写入到对应的IfcTypeObject子类类型的文件中去。
5. 如果是IfcRelDefinesByProperties的子类实例，那么将解析出RelatedObjects（是IFC对象实例集合）和RelatingPropertyDefinition（是一个IfcPropertySet的子类实例），获取IfcPropertySet的属性数据，然后依次获取RelatedObjects中的IFC对象类型的文件句柄，将属性数据写入到文件中去。

其代码部分如下：

因为bulk机制，所以所有的记录都应该是这个样子的。

至此，已经将所有IFC对象的数据写入到了JSON文件。

支持中英文的问题。

## 3.3 数据索引技术

### 3.3.1数据模式映射

ElasticSearch中mapping介绍。

分词器介绍。

### 3.3.2索引过程

Bulk过程。索引脚本。

### 3.3.3索引优化（批量索引bulk）

## 3.4 数据检索技术

3.4.1 全文检索

3.4.2 关键词检索

## 3.5 自定义ID字段

# 第四章 基于Neo4j的IFC对象空间关系数据的存储和查询研究

在绪论中本文对IFC标准进行了介绍。IFC标准的核心层中不仅定义了IFC对象的位置、几何形状等，同时也定义了IFC对象之间的关系。这些关系表现为空间结构上的组成、包含等多种关系，与IFC对象在物理世界中的关系一致。对此，本章将详细介绍IFC标准是如何定义IFC对象的空间关系的。

其次，当人们关心某一栋楼的某一层的某一面墙时，实际上指定了一个空间结构关系。为了满足用户对于IFC对象的空间关系的查询，本文采用了图数据库Neo4j来进行IFC对象的空间关系数据的存储和查询。具体的内容分为图数据模型设计、图数据提取技术、图数据存储技术和图数据查询技术4个部分来进行详细地介绍。

最后，本文基于BloomFilter技术提出了一种针对特定空间关系的查询优化，提高了查询效率，同时对于空值查询能够快速失败，提前给用户响应。

## 4.1　IFC对象空间关系数据模型

IFC标准划分为四个层次，包括资源层、核心层、共享层和领域层。资源层定义了基础信息，如几何、尺寸、材料等基本元素信息。核心层定义了IFC对象模型的整体框架，不仅包括IFC对象的位置和几何形状等，同时也定义了IFC对象之间的关系。这些关系表现为空间结构上的组成、包含等多种关系，与IFC对象在物理世界中的关系一致。例如一栋楼包含了许多楼层，而一层楼包含了许多建筑构件如墙、门、窗等。又比如一面墙上有两个空缺（OpeningElement），这个空缺对应了一面窗户等等。~~这些建筑构件对应IFC标准中的实体对象。而这些实体对象均由IfcRoot继承。~~

在IFC标准中，IfcRoot是核心层及以上层次中全部实体类型的抽象基类。IfcRoot的派生子类有三种类型，分别是IfcObjectDefinition、IfcPropertyDefinition和IfcRelationship。而IFC对象之间的空间关系就定义在IfcRelationship当中。

IfcRelationship的派生类有IfcRelAssigns、IfcRelAssociates、IfcRelConnects、IfcRelDecomposes和IfcRelDefines五类。这些类定义并非都是表示IFC对象之间的空间结构关系。它们的定义如下：

1. IfcRelAssigns及其派生类：表示Ifc实体（IfcObjectDefinition）与角色（IfcActor）、控制（IfcControl）、过程（IfcProcess）、资源（IfcResource）等之间的映射关系。
2. IfcRelAssociates及其派生类：表示Ifc实体（IfcObjectDefinition）与约束（IfcConstraint）、文档（IfcDocument）、库（IfcLibrary）、材料（IfcMaterial）等的映射关系。
3. IfcRelConnects及其派生类：表示Ifc实体对象（IfcObjectDefinition）之间基于某个标准的连接关系（connectivity）。例如，两面墙基于一条线相连接。
4. IfcRelDecomposes及其派生类：表示Ifc实体对象（IfcObjectDefinition）之间的组成关系（composition）。例如，一栋建筑有五层楼。
5. IfcRelDefines及其派生类：表示Ifc实体对象（IfcObjectDefinition）与属性类型（IfcProperty）、类型定义（Type）之间的映射关系。

根据这五类子类型的定义，只有IfcRelConnects和IfcRelDecomposes及其派生类涉及到Ifc对象的空间关系，因此在设计空间关系数据的存储模型时，只考虑这两类及其派生类。而其他类型的信息会存储在ElasticSearch当中以供检索。

下面将举一个例子来介绍Ifc实例对象之间的空间关系。

#94=IFCPROJECT('0DVPyye2z0deMtQa1pIata',#41,'\X2\987976EE7F1653F7\X0\',$,$,'\X2\987976EE540D79F0\X0\','\X2\987976EE72B66001\X0\',(#83,#91),#78);

#592=IFCSITE('0DVPyye2z0deMtQa1pIatc',#41,'Default',$,'',#591,$,$,.ELEMENT.,(39,54,57,601318),(116,25,58,795166),0.,$,$);

#679=IFCRELAGGREGATES('35aC7nYvTEcBYUkz$D5i0v',#41,$,$,#94,(#592));

#104=IFCBUILDING('0DVPyye2z0deMtQa1pIatb',#41,'',$,$,#32,$,'',.ELEMENT.,$,$,#100);

#683=IFCRELAGGREGATES('30UiOLvXbBv9bj$ovhCyLU',#41,$,$,#592,(#104));

#113=IFCBUILDINGSTOREY('0DVPyye2z0deMtQa2CjRCZ',#41,'\X2\68079AD8\X0\ 1',$,$,#111,$,'\X2\68079AD8\X0\ 1',.ELEMENT.,0.);

#687=IFCRELAGGREGATES('27PCKGLxT4mxtV9cw6mgBW',#41,$,$,#104,(#113));

#155=IFCWALLSTANDARDCASE('0o\_qQXI0r5UQtbU$t93HI7',#41,'\X2\57FA672C5899\X0\:CW 102-50-215p:309591',$,'\X2\57FA672C5899\X0\:CW 102-50-215p:412',#124,#151,'309591');

#672=IFCRELCONTAINEDINSPATIALSTRUCTURE('3Zu5Bv0LOHrPC10066FoQQ',#41,$,$,(#155,#367,#430,#550),#113);

#94代表项目，#592代表场地，#679代表项目（#94）“包含”场地（#592）。#104代表建筑，#683代表场地（#592）包含建筑（#104）。#113代表楼层，#687代表建筑（#104）包含楼层（#113）。（#155）代表墙，#367、#430、#550代表其它建筑构件，#672代表楼层（#113）包含这四个建筑构件。

这是一个典型的Ifc对象之间的空间包含关系的例子。如果用户需要指定特定楼层的某个建筑构件，这就需要通过Ifc对象之间的空间关系数据来进行查找。由于这种空间关系是使用关系对象来间接表示的，所以在一定程度上使得空间关系的查询并不容易。但是这种关系天然适合图数据库来进行存储。Neo4j是一个高性能的图数据库，它将整个复杂的关联数据集存储在一个大型的网络结构中，再采用一系列的图操作来实现对数据的管理和应用。

接下来本文将介绍基于Neo4j图数据库的Ifc对象之间的空间关系数据的存储和查询相关技术。

## 4.2　图数据模型设计

在现实世界中，建筑元素的关系可以是：某个项目包含很多场所，每个场所包含很多建筑，每个建筑包含很多楼层，每个楼层包含很多建筑构件，如门、窗、墙、梁、板、洞等等。而建筑构件之间也有多元化的关系，比如某两面墙相连接、某面墙上有四扇窗等等。这些关系可以形象化地展示为图1.

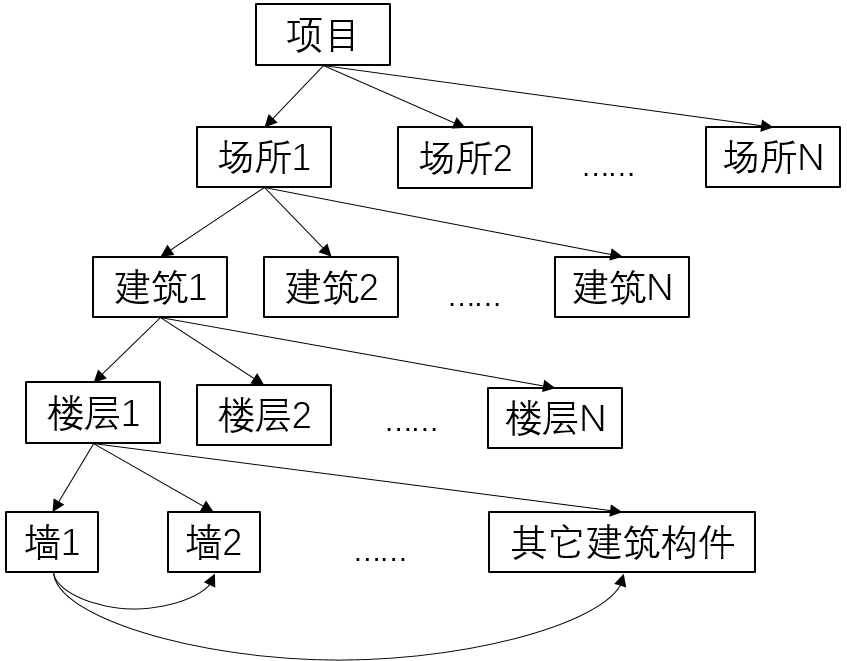


图1 现实世界中的建筑元素的空间关系图

依照4.1小节对于IFC标准中关系的介绍，如果用IFC对象来表示上面图中的建筑元素，则对应的是下面这张图2。

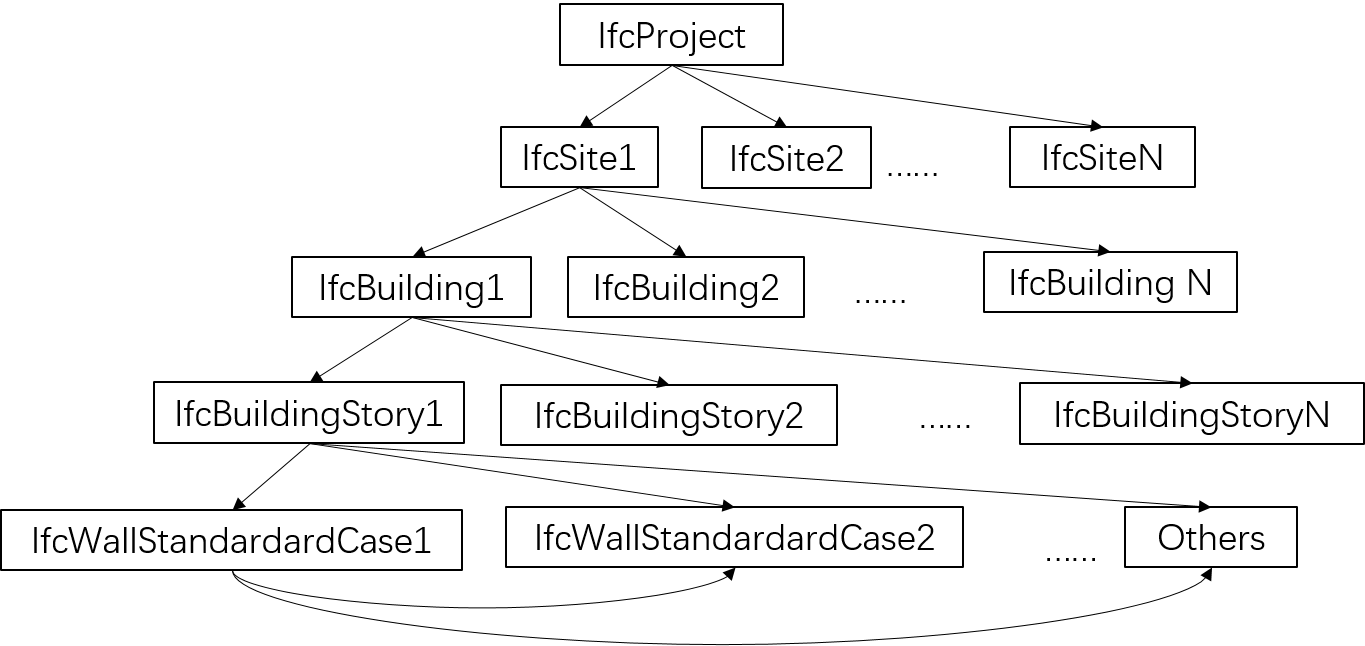


图2 IFC对象的空间关系示意图

在上图中我们用一条带箭头的连线来代表Ifc对象之间的空间关系。而这些关系在IFC标准中是用IfcRelationship的派生类来定义的。例如，IfcProject包含了很多IfcSite，这个包含关系用IfcRelAggregates来表示。而IfcWallStandardCase连接了另一个IfcWallStandardCase，这个连接关系用IfcConnects关系类型来定义。

如果使用IfcRelationship派生类来替代上图中带箭头的连线，就是一个完整的IFC对象的空间关系模型，如图3。为了简洁，只标出了部分关系。

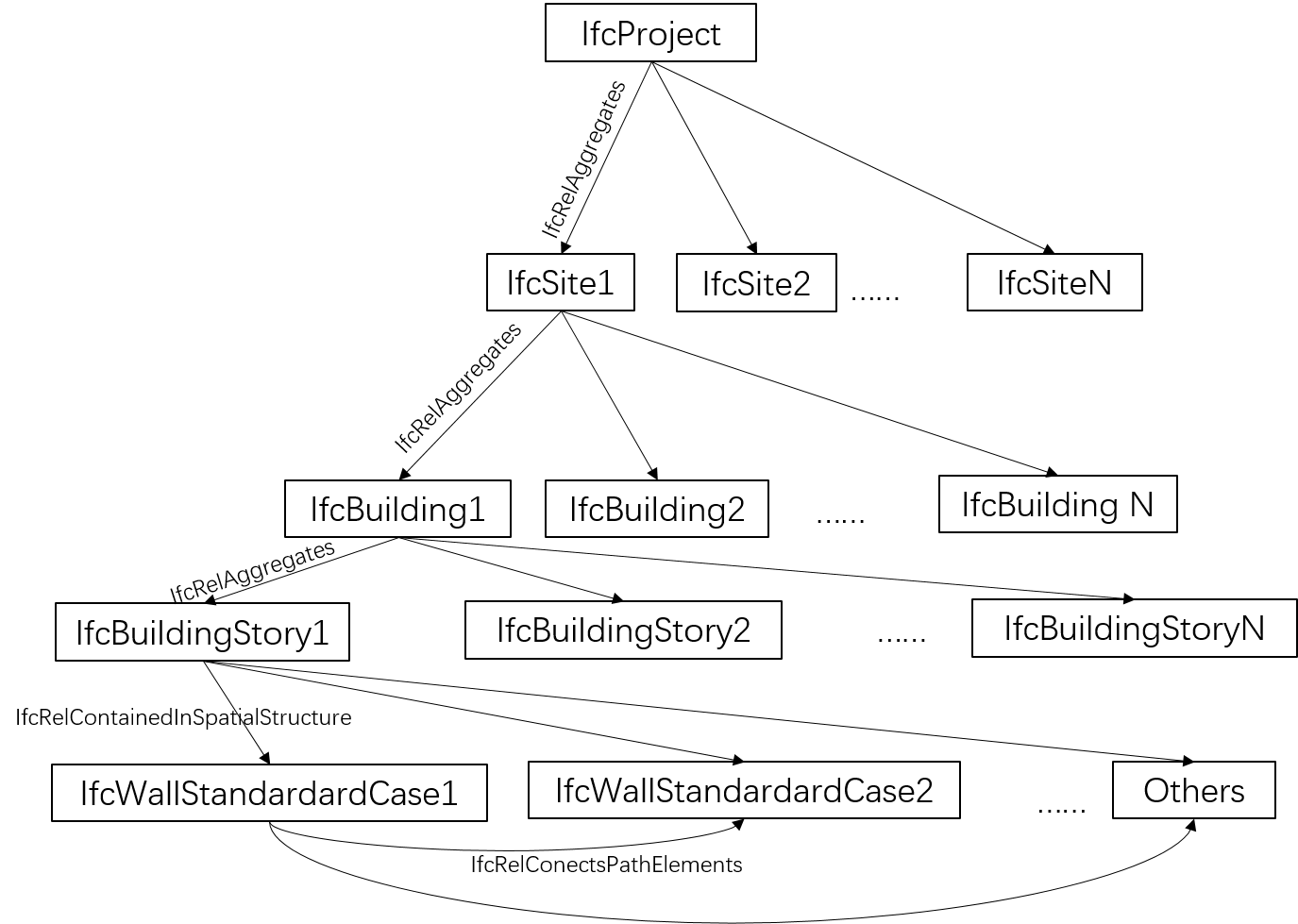


图3 IFC对象的空间关系数据模型

图3展示的模型可以直接对应到Neo4j的图数据模型。在Neo4j中，有两种基本数据类型，Node（节点）和Relationship（关系）。Node通过Relationship所定义的关系连接起来。同时可以在Node和Relationship上赋予key/value形式的属性，来表示详细的信息，以便对Node和Relationship进行查询。如果指定某个节点，从此节点出发，通过关系找到另外一个节点，这就形成了一条路径。这就是Neo4j的图数据模型。由此可知，图3中所展示的IFC对象的空间关系数据模型，天然适合Neo4j的图数据模型。每一个IFC对象对应图模型中一个Node，每一个IfcRelationship派生类对应图模型中的relationship。由此可以实现IFC对象的空间关系数据的存储，并依托Neo4j对于图数据的操作来进行IFC对象的空间关系的查询和应用。

## 4.3　图数据提取技术

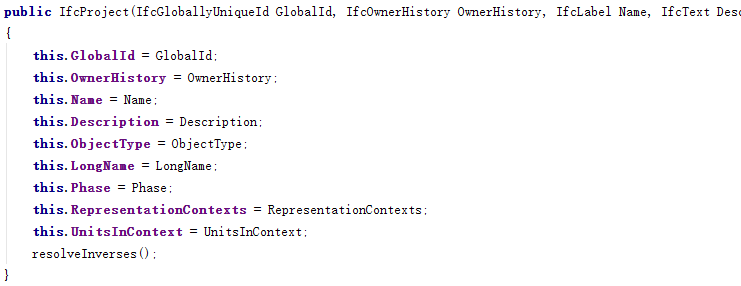
本小节将针对Ifc文件数据如何解析到内存中进行详细介绍。以三条IFC数据为例：

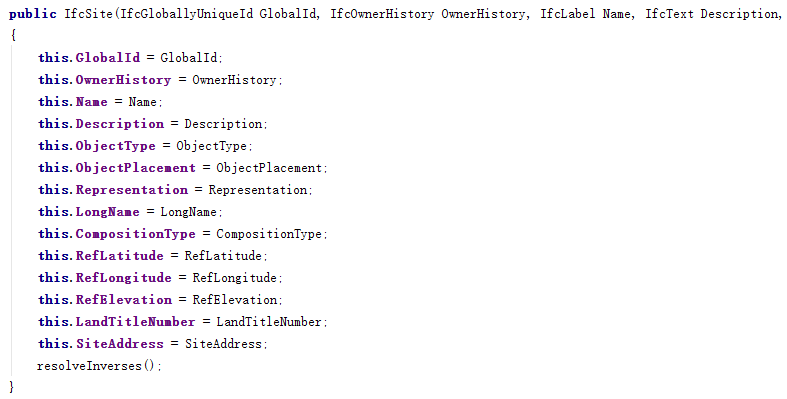
#94=IFCPROJECT('0DVPyye2z0deMtQa1pIata',#41,'\X2\987976EE7F1653F7\X0\',$,$,'\X2\987976EE540D79F0\X0\','\X2\987976EE72B66001\X0\',(#83,#91),#78);

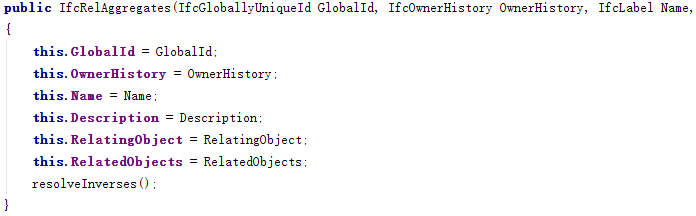
#592=IFCSITE('0DVPyye2z0deMtQa1pIatc',#41,'Default',$,'',#591,$,$,.ELEMENT.,(39,54,57,601318),(116,25,58,795166),0.,$,$);

#679= IFCRELAGGREGATES('35aC7nYvTEcBYUkz$D5i0v',#41,$,$,#94,(#592));

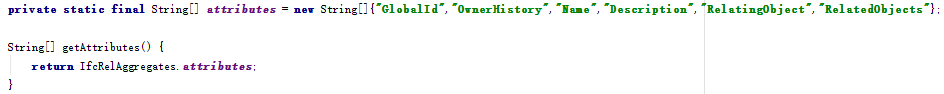
基于3.2.1小节介绍的IFC文件解析过程，这三条数据将被解析为内存中三个Java对象。







Java对象对应了IFC对象，Java对象的fields对应了IFC对象的字段。这些Java对象对应Neo4j的Node，Java对象的字段和字段值将作为key/value pair，存储到Neo4j数据库中，作为Node的属性。

IfcJavaToolBox解析器并不支持获取对象的字段名，所以需要为每一个对象类型定义一个attributes数组结构，如IfcRelAggregates.java代码中的定义：

在此需要说明，因为这些Java对象的fields是继承父类的，反射的方法在此行不通，所以才采用了这种方法。attributes数组中的值必须严格按照IFC标准的schema当中的属性定义的顺序，这样才能够解析出正确的key/value pair。

至于id字段可以借鉴bimserver的方法，内存中建立一个对象和id的映射的map，已达到自定义id的目的。

## 4.4　图数据存储设计

在上一小节中介绍了IFC文件中的三条数据解析到了内存中。这已经完成了数据准备。这就来到了Neo4j开发的部分。

本小节将介绍如何将内存中的Java对象数据存储到Neo4j当中。Neo4j提供了服务器版本和嵌入式版本。但由于后者更有利于接口自定义设计，且不受网络数据传输效率的影响，本文采用嵌入式版本来进行开发。

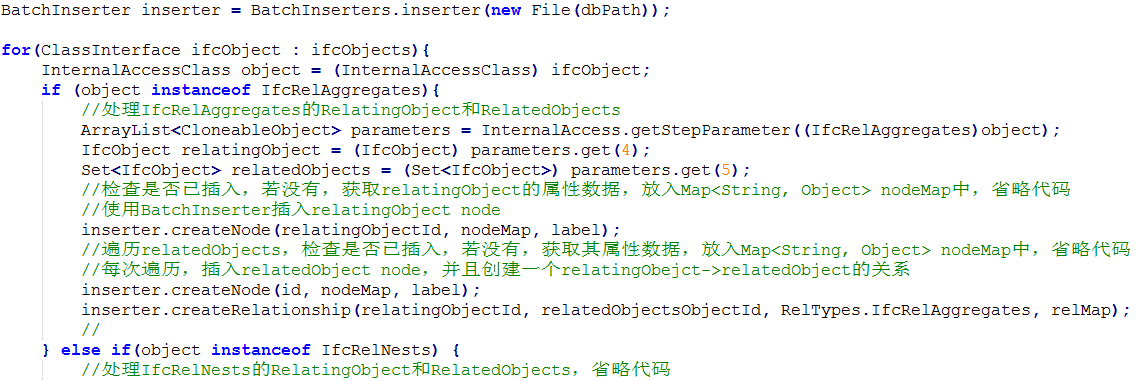
Neo4j存储图数据的步骤是：

1. 构造Map结构，存储属性的key/value pair。
2. 指定DB PATH，在该PATH下创建EmbeddedGraphDatabase实例。
3. 调用createNode()方法生成Node实例。
4. 实现Relationship接口，定义枚举类型表示关系。
5. Node实例调用createRelationship()方法，创建节点之间的关系。
6. 调用PropertyContainer 接口中的方法setProperty()将Map结构的key/value，赋值给Node和Relastionship实例，并可通过 getProperty()获取属性。

这是创建少量节点时可以采用的方式。在本文中需要存储的IFC对象数量巨大，所以采用批量插入——BatchInserter的方式，以提高存储效率。其算法思想是：

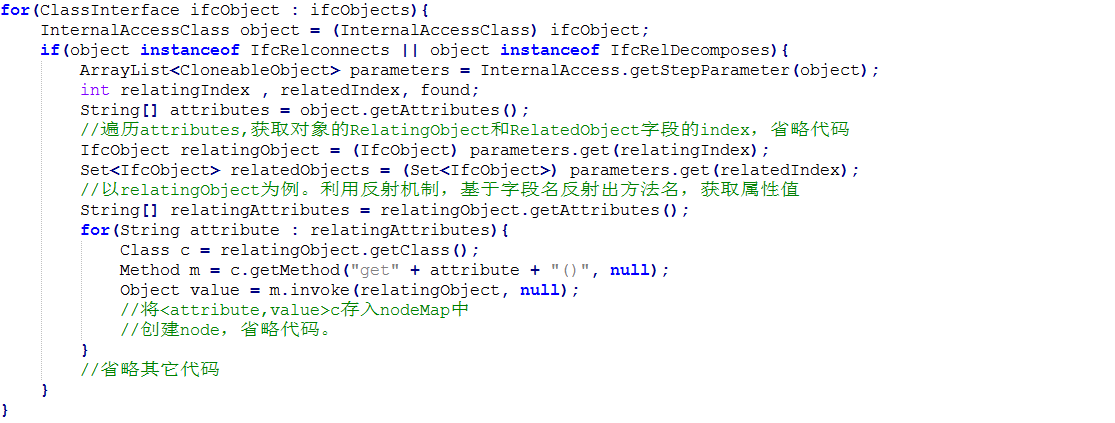
1. 遍历内存中构建的Java对象集合Collection<ClassInterface> ifcObjects，找到IfcRelConnects和IfcRelDecomposes的派生类实例（为什么限于这两类，已经在4.1小节介绍）。
2. 对于1中找到的每一个关系实例，其relatingObejct字段指向一个关系的开始节点，而relatedObjects指向一个关系的结束节点，结束节点可能是一个集合。此时可以使用Neo4j的批量插入接口BatchInserter来创建节点和关系了。
3. 需要注意的是，节点可能出现在多个关系实例里。所以，需要维护一个List(Long)类型的结构，存储已经创建的节点的ID。每次创建节点之前检查是否已经创建，从而避免重复创建节点。

其代码逻辑如下：



这是使用原生的IfcJavaToolBox的API所写的代码，因为本文关心的IfcRelConnects和IfcRelDecomposes的派生类有13种，而每一种的字段类型都不相同，要从中找到relatingObject和relatedObjects需要针对不同的类型各写一个if代码块。同时，每一个IfcObject的字段也千差万别，所以这样写代码实现起来十分冗杂。

因此想到，结合本文为每个IFC JAVA对象定义的attibutes数组，再使用JAVA反射机制，调用method.invoke(), 可以很容易获取字段值，从而大大简化代码。代码逻辑如下：



这样就实现了将内存中已经解析出的IFC JAVA对象的数据存储到Neo4j的过程。

Batch方式的插入效率？

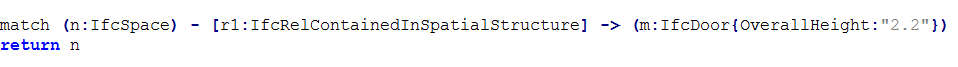
## 4.5 图数据查询设计

用户使用Neo4j的一个经典场景是依据节点或者关系的属性数据找到相关节点以及关联的节点。例如，用户需要找到已经安装了高度为2.2m的门的墙。在IFC对象关系数据模型中，门与墙的关系如图：



图4

于是其查询逻辑使用Cypher如表达式1：



表达式1

其查询执行逻辑是：

1. 访问节点文件和属性文件，找到Label为IfcDoor的节点，遍历IfcDoor类型节点链表，读取其属性数据，找到满足OverallHeight为2.2的节点。
2. 访问关系文件，以1）中找到的节点为起点，遍历关系链表，过滤条件是找到Label为IfcRelContainedInSpatialStructure的关系，且关系的终止节点的label为IfcSpace。
3. 返回符合条件的终止节点。

可以看到，整个查询过程可以分为两个部分，第一步是先找到起始节点，

第二步是以起始节点开始，遍历其关系，过滤目标节点。

关系遍历的规模是起始节点的出度。而第一步找到其实节点的遍历规模则是整个Label类型的节点。这时候需要大量的随机读写，如果没有索引，效率将非常低下。为了获得更好的性能，我们可以在字段属性上构建索引，这样任何查询操作都会使用索引，从而大幅度提升查询性能。

在第二章中我们已经介绍了Neo4j的索引。本文将使用Schema索引，来为节点和关系创建索引。下面是创建索引的Cypher语句示例：

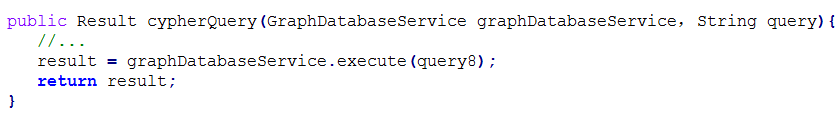


由于Schema索引是依据Label和属性字段来创建的，且IFC对象类型多、属性字段差异大，手动创建十分麻烦。所以本文采用的方法是，在初次向Neo4j中导入所有IFC对象的数据之后，依据IFC对象的数据模型一次性创建所有IFC类型的索引。

其代码逻辑是：

1. 维护一个已导入的IFC对象类型集合，可以防止对不存在的类型进行索引；
2. 针对每个IFC对象类型，获取属性名数组attributes，进行Cypher语句字符串的拼接；
3. 执行Cypher索引语句。

对于Cypher语言的执行，使用嵌入式Neo4j的GraphDatabaseService API，其代码如下：



创建索引以后，当查询指定属性值的节点时，Schema索引将自动被调用，以提高查询性能。

关系是否要被索引呢？

Cypher语言介绍放在前面的部分。

Neo4j是同时启用线程共享的。所以可以另开线程，将

使用cypher语言的代码，语句如何转化？

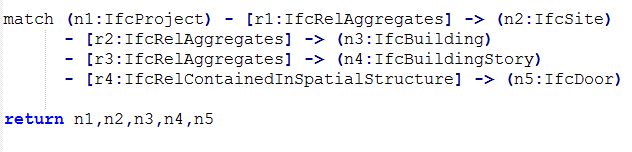
直接将代码搬出来？可是没有多长。需要将用户的查询语言转化为cypher语言进行查询。不用了。这个查询语言已经很简洁了。重点放在查询的快速失败上。如果是空值查询，是不是比较快呢？试验证明是的。

## 4.6 基于BloomFilter的路径查询优化

前面几个小节已经讲述了如何基于Neo4j图数据库来进行IFC对象的空间关系数据的存储和查询。这已经能够满足用户对于某个对象或者某些对象以及它们之间的空间关系的查询。本小节将针对一种特定的路径查询场景提供优化方案。

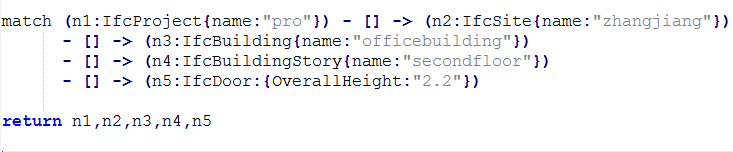
4.6.1 特定路径查询场景

IFC标准中有一种特定的查询是，用户需要依次确定项目、场地、建筑、楼层，然后再找到楼层中满足一定条件的建筑构件，如门、窗、家具或其它零件。图3展示了这种关系在neo4j中的表示。在Neo4j图数据模型的定义中，从项目节点到建筑构件节点形成了一个路径，这个查询是一个特定的路径查询。其Cypher语句的简单形式是表达式1：



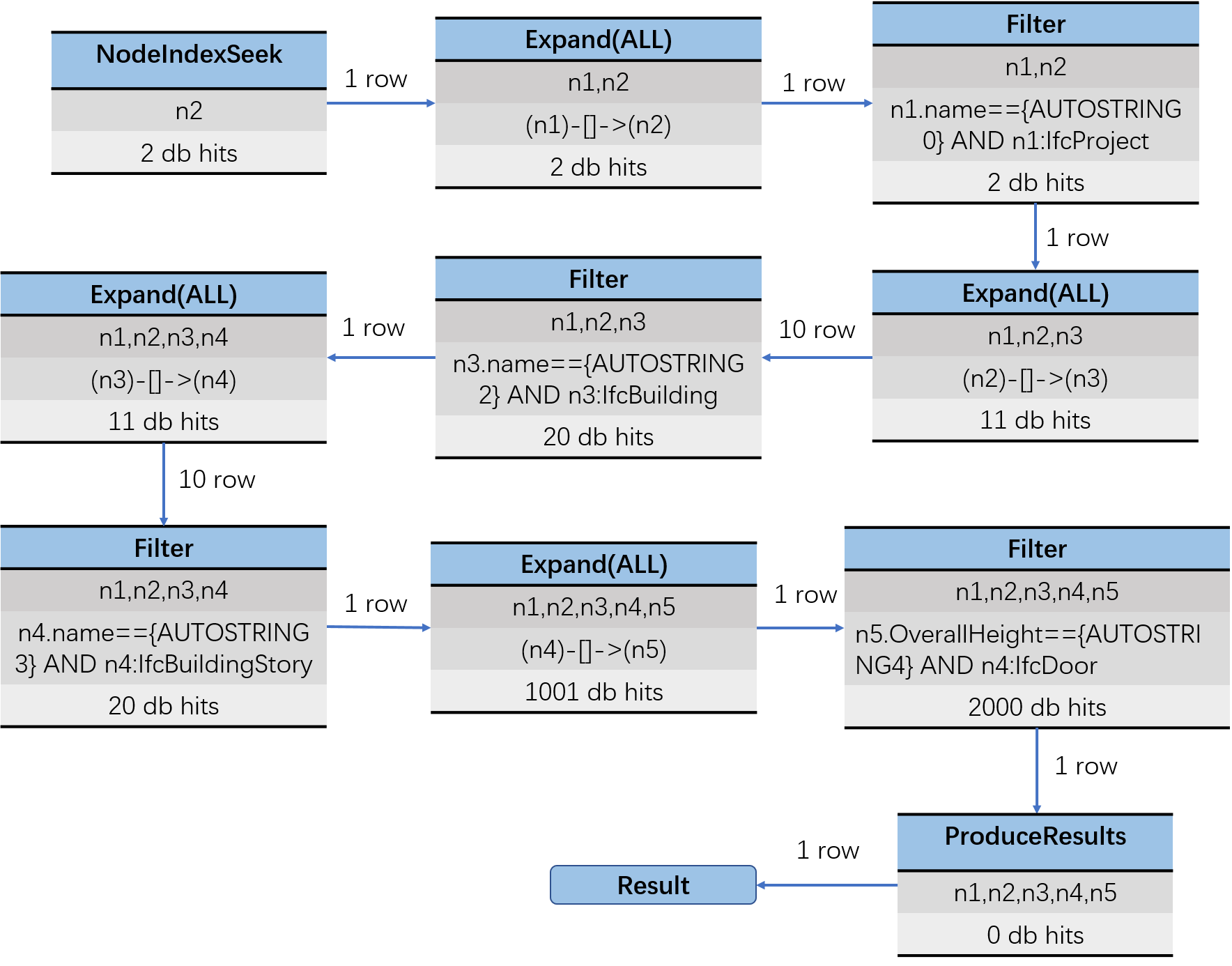
表达式2

又由IFC标准的定义可知，r1、r2、r3、r4的类型是固定的，所以可以在Cypher表达式中省略掉类型。同时，用户需要对路径中的每一个节点加以属性值的约束，本文的示例暂且以“name”属性约束作为这个路径查询中对节点的约束条件。所以Cypher表达式应为式2：



表达式3

这个Cypher查询语句的完整的执行计划是：



1. 执行NodeIndexSeek。访问索引“index on :IfcSite(name)”，返回节点n2。在这里假设只有一个IfcSite节点满足条件而返回。
2. 执行Expand(ALL)，访问指向节点n2的关系链表。因为一个IfcSite只有一个IfcProject指向它，也就是说指向n2的关系只有一个。所以访问db一次。
3. 执行Filter。过滤2）中关系所关联的节点的类型和属性，返回满足条件的IfcProject节点n1。
4. 然后从节点n2开始，依次执行Expand(ALL)和Filter过程。知道最后一个关系所关联的节点完成filter，找到满足条件的节点为止。
5. 返回结果。

分析这个过程，可以得出这样的结论：只有在整个执行计划的第一步使用了index，先找到一个切入点作为起始节点。而之后是通过访问关系链表数据找到关联的节点，而后对这些节点进行过滤。

这里有几个值得讨论的问题：1）没有充分利用index 2）当节点所关联的关系数量增长时，链表结构的访问效率会成为影响查询效率的不容忽视的因素。3）由于是从起始节点依次开始访问关系链表和节点，很有可能出现当执行到最后一个filter环节时才发现没有满足条件的节点而返回空值，此时前面的查询操作已消耗了大量时间。

对此，本文提出用查询分解的方式，先将原始的查询分解为多个针对节点的查询，找出所有的满足条件的ID之后，然后将ID进行拼接，利用BloomFitler做结果过滤。

下面小节的安排是，先介绍关键的BloomFilter数据结构，然后介绍如何构建存储用于本小节所介绍的特定路径查询的BloomFilter，最后会介绍如何基于BloomFilter进行查询。

### 4.6.2 BloomFilter原理

BloomFilter于1970年由Burton H. Bloom提出, 利用位数组标识一个集合，可以以较低的误判率来判断一个元素是否属于这个集合。因此这种数据结构适合应用在能容忍低错误率的场合。相较于传统的哈希函数映射和存储元素的方式，BloomFilter更加节省空间，从而能够满足数据量更大的应用场景。

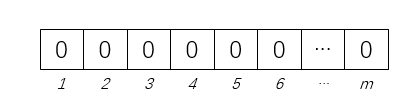
对于4.6.1小节中提到的特定的路径查询来说，如果我们能够把符合空间组成关系的IfcProject、IfcSite、IfcBuilding、IfcBuildingStory、IfcDoor（以及其它建筑构件）的ID按照顺序拼接起来，维护在BloomFilter当中。当查询来到时，将满足条件的各个节点的ID拼接起来，查看是否在BloomFilter当中，就可以知道，这个结果集是否存在。BloomFilter存在一定的误判。也就是说，如果这条路径并不在neo4j存储的图中，但是也有可能会判定为存在，此时用户得到的结果将会是由原始的Cypher查询得到的正确结果。同时，BloomFilter不会漏掉判断。这对我们是相当有利的，因为不会漏判意味着，只要判定这个字符串在BloomFilter当中，那么它一定存在。此时，我们只需要返回这个字符串就可以了。另外，BloomFilter用很少的内存就可以支撑上亿条记录，基本可以满足我们的存储和查询需求。

BloomFilter的核心思想就是利用多个不同的Hash函数来解决单一Hash带来的“冲突”。传统的Hash表利用同一个Hash，计算不同的字符串得到的值可能相同时，这就是“冲突”。如果要减少冲突，必须要将位数组的长度扩大。这样会消耗大量内存。为了解决这个问题，BloomFilter支持选择k个不同的哈希函数，这样就可以大大地减少冲突，同时保证低的内存消耗。

BloomFilter的原理要点：位数组、k个独立的Hash函数。

1. 位数组：

初始状态时，BloomFilter是一个包含m位的bit数组。每个bit都为0，如下图：



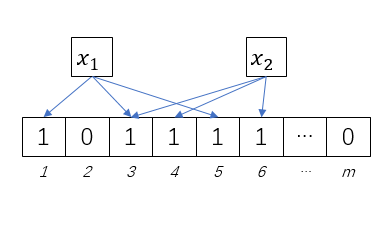
1. 基于K个独立Hash函数，添加元素：

待添加的元素设为 。当加入任何一个元素x的时候，BloomFilter使用k个互相独立的Hash函数，得到k个Hash值，对应的范围内的k个位置。将这个m bit数组的对应k个位置的值置为1。如果这个位置被多次置为1，那么只有第一次会起作用。

例如，k=3，,,,

,,.

加入，则bit数组为：



1. 判断元素是否存在：

在判断元素y是否属于这个集合时，只需要对y使用k个Hash函数得到k个hash值，对应bit数组中k个位置。如果这k个位置都是1，那么就认为y是集合中的元素，否则不是。显然这不是100%正确的。如果y事实上并不在集合里，但映射到的k个位置刚好恰好已经置为1了，那么就会出现False Positive，也就是误判。

1. BloomFilter的参数：

误判率与插入BloomFilter的元素个数n、bit数组大小m以及Hash函数个数k有关。研究表明，当Hash函数个数k时错误率最小。在错误率不大于E的情况下，m至少要等于才能表示任意n个元素的集合。

对于在本文IFC数据存储应用中的BloomFilter的参数选择将在接下来的构件BloomFilter中介绍。

### 4.6.3 构建BloomFilter

根据4.6.1小节对于改进算法的描述可知，如果要想实现这样的查询，必须先基于IfcProject、IfcSite、IfcBuilding、IfcBuildingStory、IfcDoor（等）的空间组成关系，将所有的这些ID拼接字符串存入到BloomFilter当中。这需要在内存中解析IFC对象时就把它们的关系对应起来。为了达到这个目的，需要借助于以下几个Map结构。

1. Map<Long, Long> siteToPro;

存储IfcSite的ID与其所在的IfcProject的ID的映射。

1. Map<Long, Long> bdgToSite;

存储IfcBuilding的ID与其所在的IfcSite的ID的映射。

1. Map<Long, Long> storyToBdg;

存储IfcBuildingStory的ID与其所在的IfcBuilding的ID的映射。

1. Map<Long, Long> elmtToStory

存储IfcElement的ID与其所在的IfcBuildingStory的ID的映射。

基于这4个数据结构，可以构建一个IFC对象getStoryID()、getBdgID()、getSiteID()、getProID()方法，得出IFC对象所对应的IfcBuildingStory、IfcBuilding、IfcSite、IfcProject的ID。然后按照从Ifcproject到IfcElement的空间组成顺序拼接ID为长字符串，插入到BloomFilter中。

关于上面的步骤，会有人有疑问：通过几个get方法不是已经可以找到上文所介绍的路径查询的结果了吗？的确是这样。但是问题就在于，Map结构非常占用内存。这也是本文为什么需要BloomFilter数据结构的原因，它能大大地缩小内存占用，从而可以常驻内存。在后面的试验中，本文会对内存占用做进一步的比较分析。

构建BloomFilter；内存中维护几个map结构，这几个map结构之后就销毁了。总共的内存开销是多少，这个需要计算出来吧。

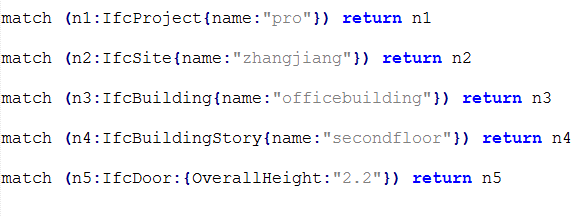
### 4.6.4 基于BloomFilter的查询

回顾4.6.1节中表达式2的Cypher查询所存在的问题，本文改进的想法和目标是：

1. 将查询分解为多个对节点的查询，从而可以充分利用index；
2. 由于1)步骤改变了查询计划执行的顺序，从而使得当节点本身不满属性要求时可以快速失败，返回结果；
3. 使用BloomFilter的过滤机制，而不是链表的顺序访问机制，可以有效提升查询速度。

具体的算法逻辑是：

1. 将上面的Cypher查询分解为四个对节点的Cypher查询：



1. 分别执行以上四个查询，也就是四个NodeIndexSeek。如果结果集为空，则可以直接结束整个查询，返回给用户空值；
2. 如果结果集不为空，则将节点的ID字段拼接为一个字符串。
3. 已创建的BloomFilter包含了所有的可能的结果字符串。查看3）中的字符串是否包含在BloomFilter中。如果在，则返回ID对应的节点。如果不在，则说明查询结果为空。

对于误判的处理