

|  |
| --- |
| 硕 士 学 位 论 文 |

**（专业学位）**

|  |
| --- |
| **基于Neo4j的BIM数据库研究** |

|  |  |
| --- | --- |
| **Research on BIM Database Based on Neo4j** | **英文论文题目** |

|  |  |
| --- | --- |
| 院 系： | 计算机科学与技术学院 |
| 专 业： | 计算机技术 |
| 姓 名： | 张亚迪 |
| 指 导 教 师： | 王　鹏　副教授 |
| 完 成 日 期： | 2017年9月8日 |

**目录**

[摘要 IV](#_Toc494988977)

[Abstract V](#_Toc494988978)

[第1章 绪 论 1](#_Toc494988979)

[1.1 研究背景及意义 1](#_Toc494988980)

[1.2 国内外研究现状 2](#_Toc494988981)

[1.3 研究内容和本文贡献 3](#_Toc494988982)

[1.4 组织结构 4](#_Toc494988983)

[第2章 相关技术基础 5](#_Toc494988984)

[2.1 IFC基础 5](#_Toc494988985)

[2.1.1 IFC标准 5](#_Toc494988986)

[2.1.2 IFC文件格式 7](#_Toc494988987)

[2.2 全文检索技术 8](#_Toc494988988)

[2.2.1 全文检索基础理论 8](#_Toc494988989)

[2.2.2 Lucence概述 9](#_Toc494988990)

[2.2.3 ElasticSearch技术 11](#_Toc494988991)

[2.3 图数据库技术 12](#_Toc494988992)

[2.3.1 图数据与图数据库 12](#_Toc494988993)

[2.3.2 Neo4j概述 13](#_Toc494988994)

[第3章 系统总体架构 17](#_Toc494988995)

[3.1 总体架构设计 17](#_Toc494988996)

[3.2 子模块设计要点 18](#_Toc494988997)

[第4章 ElasticSearch存储关键技术研究 19](#_Toc494988998)

[4.1 IFC实体的属性数据模型概述 19](#_Toc494988999)

[4.1.1 属性 19](#_Toc494989000)

[4.1.2 属性集 21](#_Toc494989001)

[4.1.3 属性集与IFC实体关联 23](#_Toc494989002)

[4.2 半结构化存储算法 23](#_Toc494989003)

[4.2.1 冗余量估算模型 24](#_Toc494989004)

[4.2.2 生成JSON数据 25](#_Toc494989005)

[第5章 Neo4j存储关键技术研究 29](#_Toc494989006)

[5.1 IFC实体的空间关系数据模型概述 29](#_Toc494989007)

[5.2 动态建立索引算法 31](#_Toc494989008)

[5.3 基于BloomFilter的特定路径查询算法 33](#_Toc494989009)

[5.3.1 特定路径查询场景及问题 33](#_Toc494989010)

[5.3.2 BloomFilter原理 35](#_Toc494989011)

[5.3.3 基于BloomFilter的查询 37](#_Toc494989012)

[5.3.4 构建BloomFilter 39](#_Toc494989013)

[第6章 系统实验 40](#_Toc494989014)

[6.1 属性数据相关测试 40](#_Toc494989015)

[6.1.1 实验环境 40](#_Toc494989016)

[6.1.2 存储算法实验 40](#_Toc494989017)

[6.1.3 系统可用性测试 42](#_Toc494989018)

[6.2 空间关系数据相关测试 42](#_Toc494989019)

[6.2.1 实验环境 42](#_Toc494989020)

[6.2.2 动态建立索引算法实验 43](#_Toc494989021)

[6.2.3 基于BloomFilter的特定路径查询实验 43](#_Toc494989022)

[6.2.4 系统可用性测试 44](#_Toc494989023)

[第7章 总结与展望 45](#_Toc494989024)

[7.1 总结 45](#_Toc494989025)

[7.2 展望 45](#_Toc494989026)

[参考文献 47](#_Toc494989027)

# 摘要

BIM系统是一种全新的信息化管理系统，目前正越来越多应用于建筑行业中。BIM全称为Building Information Model，即建造信息模型。BIM系统要求将建筑全生命周期中的所有信息整合在统一的数据库中。然而，随着时代的发展，智慧城市概念的提出，BIM数据有了长足的增长，以往的BIM数据库存储和查询手段已经渐渐越来越不能应付数据的长足提升。所以为海量BIM数据提供一种高效快速的存储和查询手段是有必要和有意义的。

结合国内外BIM数据库研究的发展方向，可以看到，BIM数据库研究的目标是支持大数据量的存储和查询、支持更为灵活的查询方式。

基于此目标，本文选择了开源弹性搜索框架ElasticSearch并展开了研究，通过增加冗余的方式，满足用户对于属性数据的高效查询，避免了多表连接的问题。在试验中，着重解决了IFC文件解析后的数据到JSON数据的映射问题，和ElasticSearch的索引优化等问题，最终实现了IFC实体的属性数据的高效存储和查询。

其次，本文还深入研究了IFC数据标准，提出了IFC实体空间关系数据模型的存储设计。本文基于开源的Neo4j图数据库实现了IFC实体空间关系数据模型的存储，从而除了属性值查询之外，为用户提供了另外一种维度的查询方式，即空间关系的查询方式。

除此之外，本文进一步关注了一种特定的路径查询，并针对这种查询提出了基于BloomFilter的解决方案，使得查询效率得到了提升，尤其是在空值查询时能够提前失败，给用户响应。

**关键词**：建筑信息模型（BIM），IFC标准，ElasticSearch，Neo4j，BloomFilter

# Abstract

BIM system is a new information management system, and now more and more used in the construction industry. It is called Building Information Model. The BIM system requires that all information in the entire lifecycle of the building be integrated into a unified database.

However, with the development of the times, the wisdom of the city concept, BIM data has grown by leaps and bounds, the previous BIM database storage and query means has gradually become increasingly unable to cope with the data greatly improved. So it is necessary and meaningful to provide a fast and efficient means of storage and query for mass BIM data.

Combining the development direction of BIM database research both at home and abroad, it can be seen that the goal of BIM database research is to support the storage and query of large data volume and support more flexible query mode.

Based on this goal, this paper chooses the open resilience search framework ElasticSearch and studies it. It can solve the problem of multi-table join and satisfy the efficient query of attribute data by increasing the redundancy. In the experiment, we focus on solving the problem of mapping parsed IFC data to JSON data and ElasticSearch index optimization, and finally realize the efficient storage and querying of attribute data of IFC entity.

Secondly, this paper also studies the IFC data standard in depth, and puts forward the storage design of IFC entity spatial relation data model. Based on the open source Neo4j graph database, this paper realizes the storage of the IFC entity spatial relation data model, and provides the query method of the other dimension, that is, the query method of the spatial relation, other than the attribute value query.

In addition, this paper further focuses on a specific path query, and proposed a solution based on BloomFilter for this query, so that the query efficiency has been improved, especially when it is null value query it can fail in advance, and response to the user.

Keywords：BIM，IFC，ElastciSearch，Neo4j，BloomFilter

# 绪 论

## 研究背景及意义

互联网和信息技术正在变革建筑业的未来。近年来，BIM（Building Information Model，即建筑信息模型）技术在国内外建筑行业得到广泛和应用。BIM技术以建筑工程项目的各项相关信息数据作为基础，建立起三维的建筑模型，通过数字信息仿真模拟建筑物所具有的真实信息。基于BIM技术建立的系统为建筑的全生命周期管理提供平台，参建各方利用统一的建筑信息模型，在设计、施工、项目管理、项目运营等各个过程中，实现项目协同管理，减少错误、节约成本、提高质量和效益。BIM技术开启建筑信息化第二次革命，让建筑信息化从CAD时代迈入BIM时代[1]。

在此背景下，BIM数据库作为建筑产品与BIM技术衔接应用的关键性基础数据平台，具有重要的价值和很好的发展前景，建立BIM数据库意义重大[2]。BIM数据库系统建立完整的建筑信息模型数据库，模型包含建筑所有构件、设备等几何和非几何信息以及之间关系的信息，模型信息随着建设阶段，不断深化和增加。同时，BIM数据库实现了建筑信息模型数据的开放利用，基于BIM数据库，建筑信息提供者、项目管理者与用户之间可以实时、方便地进行信息交互。

实现BIM数据库的基础是信息的标准化表达。IFC（Industry Foundation Class）标准，作为国际通用的数据表示和交换标准，定义了建筑工程项目全生命周期的数据格式，内容丰富、涉及领域广，成为BIM技术的主流数据交换标准。目前，国内外BIM数据库的研究主要基于IFC标准展开。IFC标准是一个类似面向对象的建筑数据模型。IFC模型包括建筑整个生命周期内的各方面的信息，其中包含的信息量非常大而且涵盖面很广。为此，IFC标准的开发人员充分地应用了面向对象分析和设计方法，基于若干原则将这些信息进行分层和模块化，形成了IFC的整体框架。面向对象的设计方法使得IFC建筑模型数据具有很强的特殊性，表现在对象之间的关系十分复杂，这给BIM数据库的设计带来了不小的难度。

随着智慧城市的提出、建筑体量大型化趋势的到来，建筑数据有了长足的增长。2015年11月6日完工的广州周大福金融中心工程总高度530米，占地面积2.6万平方米，总建筑面积50.77万平方米，产生项目图纸5万多张，图纸变更3000余次 [1]。如此大体量的建筑数据为BIM数据库的设计提出了更高的要求。如何解决海量BIM数据的存储和查询问题迫在眉睫。

## 国内外研究现状和热点

### BIM数据库研究现状

一直以来，传统关系型数据库在数据库领域占据着重要的地位。基于传统关系型数据库的BIM数据库研究中，由于IFC标准面向对象的特殊性，如何建立IFC对象模型到关系模式的映射是关键。文献[2] 基于Microsoft SQL Server数据库技术开发了“BIM信息集成平台”，其中采用的映射方式是对IFC对象模型中可实例化的实体类型建立关系表，关系表的字段与IFC标准大纲中实体的定义基本一致，对于引用其它对象类型的字段，存储被引用对象的引用ID。文献[3][4][5]使用同样的映射方式，基于Oracle数据库技术实现IFC数据的存储。由于IFC标准定义的实体类型丰富，针对不同的实体类型都建立关系表将会产生大量的关系表，同时由于实体间关系复杂，在进行关系查询时可能产生大量的多表连接操作，导致查询效率低下。文献[53]考虑到查询特点，提出了查询代价估算模型，并基于此模型的代价估算情况做映射选择：实体类型是单独建立关系表，还是嵌入到父类型（引用类型）表内，从而能有效地降低了查询时的连接操作次数，提高了查询效率。但是由于仍然会保留不少的表，对于复杂关系查询时依然存在多表连接操作导致的效率低下的问题，效率提升空间有限。

面向对象型数据库由于可以支持对象模型的相关概念如继承、抽象等，也被用于BIM数据库的研究与应用。文献[6]开发的WISPER系统采用面向对象数据库ObjectStore来存储IFC数据。文献[55]利用面向对象数据库EDM(express data manager)存储IFC数据。文献[7] 采用Versant Object Database 8来存储IFC数据，并指出，面向对象型数据库与关系型数据库相比，建模工作量少，在IFC数据量较大（超过10万条记录）时处理效率更高。但是面向对象型数据库存在着理论不完善等问题，且可扩展性较NoSQL数据库差。

对象关系型数据库是关系数据库技术与面向对象技术相结合的产物，使传统关系型数据库得到了扩充，能够存储和检索对象数据[56]。Kang等利用对象关系数据库Cubrid存储IFC数据[8]。但目前对象关系数据库尚不成熟，没有广泛使用的商业软件。

随着智慧城市的提出、建筑体量大型化趋势的到来，BIM数据有了长足的增长，传统的关系型数据库在可扩展方面存在不足，已无法满足海量BIM数据管理的需要，越来越多的学者将NoSQL数据库技术应用到BIM数据库的研究中。与传统型数据库相比，NoSQL数据库技术有着更高效的海量数据处理能力和更灵活的数据模型等优点[58]。根据NoSQL数据库的存储方式的不同，基于NoSQL数据库技术建立的BIM数据库有：

1. **键值式BIM数据库**

键值式BIM数据库将BIM数据组织为键-值对形式进行索引和存储。开源项目BIMServer采用小巧但可靠、高性能的嵌入式数据库Berkeley DB来存储IFC数据[60]。BIMServer赋予每一个IFC实体对象一个全局唯一的ID作为key，将IFC实体对象的所有字段值解析为字节数组作为value，将key/value对存储于Berkeley DB中，能够支持基于IFC实体ID的快速查询和应用。其局限在于键值对存储IFC实体信息的方式比较单一，不能够支持任何关系数据的存储，也不能支持基于字段的查询。文献[9]为了解决IFC模型在Web应用中的访问效率问题，使用分布式键值对存储系统Redis建立了IFC数据的高速缓存。它将IFC文件以数据流方式写入Redis缓存，利用IFC文件的行编号进行索引，大幅提升了IFC文件的解析效率。同时在提取IFC实体对象信息时，结合内存中的EXPRESS字典使用JSON格式构建IFC对象模型，从而支持面向Web的IFC模型对象的传输和查询等应用。基于Redis的BIM数据库可以支持基于列值的查询问题，但依然不能很好地支持关系数据的查询。

1. **列式BIM数据库**

文献[10]基于开源的分布式列式数据库Hbase来解决基于列值的查询问题和关系数据的查询问题。基于Hbase的BIM数据库以列为单位存储数据，擅长以列为单位存取数据，并且支持动态添加列，从而支持半结构化的BIM模型设计，具体设计方式是，如果被引用实体对象是资源层实体对象，则将数据转化为二进制序列合并到引用它的实体的字段中，否则独立存储。这样根据IFC实体的键值可以直接提取其所有的信息，避免了多表连接操作，极大地提高了信息提取速度。但是由于一个实体对象信息可能会被其它多个实体引用时存在重复存储，这种方式会带来冗余。同时，将被引用对象的信息序列化的存储方式比较单一，并且存在深层嵌套，不支持对于被引用对象信息的查询。

1. **文档式BIM数据库**

文献[59]基于开源的分布式文档式数据库MongoDB来实现IFC数据的存储和查询，具体实现是将IFC实体对象的信息转化为BSON序列化格式存储为一条文档，对于IFC实体间的引用是否进行BSON嵌套存储的问题，采用分类处理的方法：1）被引用对象是资源层实体对象的，嵌入到引用它的实体的BSON文档中；2）被引用对象是几何信息相关类型的，独立存储为一条BSON文档，并在引用它的实体的文档中存储其引用；3）部分被引用的对象由开发者决定是独立存储还是嵌套存储。所以，这也是基于半结构化存储方式来解决关系数据查询问题的思路，依旧存在冗余问题。

### 图数据库的发展

## 研究内容和本文贡献

以往的关系型BIM数据库已经越来越不能满足日益增长的BIM数据的存储和管理需要，面向对象型数据库、对象关系型数据库由于理论不完善、不成熟且可扩展性差等局限性，无法成为BIM数据库应用的主要技术。而近年来蓬勃发展的NoSql类型的BIM数据库，有着更高效的海量数据处理能力和更灵活的数据模型等优点，基于半结构化的存储设计一定程度上解决了BIM数据关系查询时的效率不高的问题，但是也带来了数据冗余问题。另一方面，当前的半结构化的存储设计均采用嵌套的方式存储被引用的对象信息，由于IFC标准基于面向对象的设计而成，实体间的引用十分复杂，简单的嵌套规则无法支持更复杂的查询场景。以往在关系数据库查询研究中提出的基于查询代价估算的映射规则需要对存储的目标数据进行大量查询代价估算和对实际查询需求的调研，在NoSQL类型的数据库中实现起来存在较大的难度。

基于以上所述问题以及IFC实体对象的引用关系复杂的特点，本文在深入研究了Neo4j图数据库的基础上，提出了基于Neo4j的BIM数据库方案。本文将IFC数据模型映射为Neo4j的图数据模型，利用Noe4j图数据库对图数据的查询、管理等方面的优势，解决IFC关系数据的查询效率问题。具体贡献如下：

1. 本文研究了IFC文件和数据模型，实现了IFC文件的解析、到Neo4j图数据模型的转换、存储和索引的过程，实现了基于Neo4j的BIM数据库。
2. 本文研究了Neo4j数据库内核源码，阐述了其查询缓存模型，并在此基础上提出了面向图数据的查询缓存算法，并在Noe4j内核中进行了实现。实验结果表明，增加的缓存层比原来的系统的查询效率提高了50%。（预装载缓存）
3. 本文提出了特定路径的查询问题，并提出了基于BloomFilter的特定路径查询算法，并基于插件接口设计实现了此查询的服务接口。实验结果表明，本文提出的特定路径查询算法的查询效率比Cypher查询提高了50%。

本文基于Neo4j实现的BIM数据库能够有效地支持基于IFC实体对象的键值查询、列值查询等查询场景，同时增加的缓存层使得查询效率得到了进一步的提升，实现的插件接口拓展了Neo4j的接口服务，有效地支持了特定路径的查询，提升了查询效率。

## 组织结构

本文的组织结构大致如下：

**第1章**，描述了本文的研究背景与意义，阐述了国内外研究的现状和热点，并简单介绍了研究内容和贡献。

**第2章**，主要为本文的相关技术基础与背景知识给出了一个大致的介绍，内容涉及到IFC相关知识、图数据和Neo4j图数据库等。

**第3章**，主要介绍了系统的总体架构。该系统从IFC文件解析、数据模型转换、数据存储和查询等模块进行介绍。

**第4章**，主要是基于Neo4j内核的源码研究，给出了Neo4j的查询缓存模型，并提出了面向图数据的查询缓存算法和面向特定路径查询的算法。

**第5章**，主要是对本文提出的算法进行实验测试，并分析了结果，证明了本文方法的可行性和有效性。

**第6章**，总结本文的主要研究内容，指出当前系统的不足，并提出展望。

# 相关技术基础

## IFC基础

### IFC标准

IFC是由buildingSMART以工业的产品资料交换标准STEP编号ISO-10303-11的产品模型信息描述,用EXPERSS语言为基础，基于BIM中AEC/FM相关领域信息交流所指定的资料标准格式。有专家认为IFC如同网络通信标准HTML一样，IFC不属于任何BIM软件专有，而加入IFC标准认证的各领域及不同软件也日益增加。许多公司或教育单位也加入研究并开发相应的应用，同时提供免费试用源代码，以此吸引更多人参与IFC的研究与发展。基于BIM的IFC标准已经发展10年有余，渐渐受到学术界与业界重视，IFC不断发展会是AEC相关信息交换的重要标准。

IFC标准同样也是一个类似面向对象的建筑数据模型。IFC 模型包括建筑整个生命周期内的各方面的信息，其中包含的信息量非常大而且涵盖面很广。IFC标准的目的是支持用于建筑的设计、施工和运行的各种特定的软件的协同工作。正因为如此，IFC标准是目前对建筑物信息描述最全面、最详细的规范。这证明了IFC 模型是建筑工业和设备制造工业之间的数据模型交换的最好方法。

为此，IFC标准的开发人员充分地应用了面向对象分析和设计方法，并设计了一个总体框架和若干原则将这些信息包容进来并加以很好地组织，形成了IFC的整体框架。IFC的总体框架是分层和模块化的，整体可分为四个层次，从下到上依次为资源层、核心层、共享层、领域层。每个层次内又包含若干模块，每个模块内又包含了不少信息。图2-1是IFC模型总体结构图。

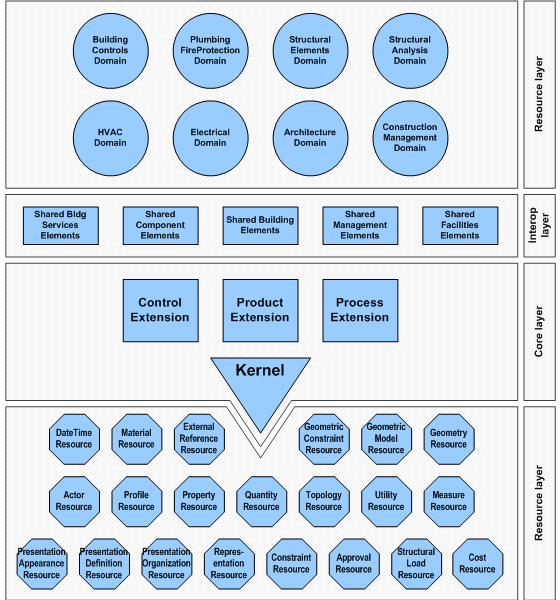


图2-1 IFC模型总体结构图

其中：

1. 信息资源层（Resource layer）描述标准中用到的基本信息，如几何、尺寸、材料等基本元素信息，是整个信息模型的基础。这些信息可与其上层（核心层、共享层和领域层）的实体连接，用于定义上层实体的特性。
2. 核心层（Core Layer）定义了建筑工程信息的整体框架。这个层次提炼了一些适用于整个建筑行业的抽象概念，不仅包括建筑对象的位置和几何形状等，同时也定义了建筑对象之间的关系，其可以反映现实世界的结构。
3. 共享层（Interoperability layer）分类定义了一些适用于建筑项目各领域（如建筑设计、施工管理、设备管理等）的通用概念，以实现不同领域间的信息交换。
4. 领域层（Domain Layer）：定义了一个建筑项目不同领域（如建筑、结构、暖通、设备管理等）特有的概念和信息实体，形成领域内的专题信息。

### IFC文件格式

IFC标准采用了EXPRESS语言作为描述语言，来描述IFC模型信息。一个完整的IFC模型包括类型定义、函数、规则、及预定义属性集组成。其中，类型定义是IFC模型的主要组成部分。类型定义分为定义类型（Defined Types）、枚举类型（Enumeration）、选择类型（Select Types）和实体类型（Entities）。IFC标准的当前版本（IFC 2X3）中包含了117个定义类型、164个枚举类型、46个选择类型和653个实体类型。实体采用面向对象的方式构建，比如门、窗，都是一个实体的实例。而定义类型、枚举类型、选择类型通常作为属性值出现在实体的实例中。下面图2-2中展示的是IfcDoor实体类型的EXPRESS定义。

图2-2

首行“Entity”表示是实体类型的定义。第二行表示它是IfcBuildingElement类型的子类。由于实体是采用面向对象的方式构建的，当然具有继承的特性。第三行和第四行则定义了两个属性OverallHeight（高度）和OverallWidth（宽度）。“OPTIONAL”表示这个属性是可选的，“IfcPositiveLengthMeasure”是一个定义类型（Defined type），它用来表示OverallHeight（高度）和OverallWidth（宽度）这个属性的值是大于0的长度测量值。但是IfcDoor的属性绝不是只有这两个，它也继承了IfcBuildingElement的属性，同时IfcBuildingElemnt也继承了更上一层父类的属性。所以，IfcDoor的属性是自身的定义的属性与所有父类的属性的集合，且严格按照顺序。

那么一个IFC实体的实例的信息是如何在文件中存储的呢？这就要引出IFC SPF文件。

IFC SPF(STEP Physical File)文件是IFC标准所使用的主要数据交换文件。其内容紧凑，无冗余信息，大大地减少了数据存储占用空间。一个完整的IFC SPF文件包含两个部分：

1. 标头段，包含一些文件信息。
2. 数据段，包含IFC整个模型的信息。

图2-3



图2-3中的解释是：

1. 行1是文件开始语句，表示ISO发布的STEP标准。对应文件最后一行（第92行）的END-ISO-10303-21。
2. 行2-6，表示文件信息，以HEADER关键字开头，以ENDSEC关键字结束。包括文件描述、文件名称、文件使用的IFC标准。
3. 行7-92是数据段。以DATA关键字开头，以ENDSEC关键字结尾。中间的每一行表示一个IFC实体的实例数据。#后面的数字是一个唯一的编号，可以不连续，而且不一定是增序，标识一个IFC实体的实例。等号右端首先是IFC实体的类型名，括号内是按照EXPRESS定义的包括继承属性的所有属性的属性值，值的类型可能是定义类型、选择类型、枚举类型。

### IFC数据模型

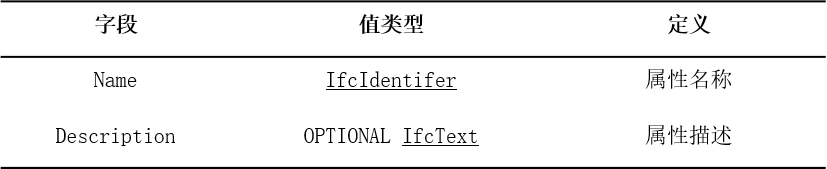
#### 属性数据模型

#### 空间关系模型

## IFC实体的属性数据模型概述

### 属性

属性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实体类型本身的各个字段可以描述IFC实体对象，还定义了不同的属性类型，为描述IFC实体对象提供了一种更加灵活的可扩展的方式。这些属性类型均继承自抽象基类IfcProperty。表4-1分别给出了抽象类IfcProperty的正向字段定义。

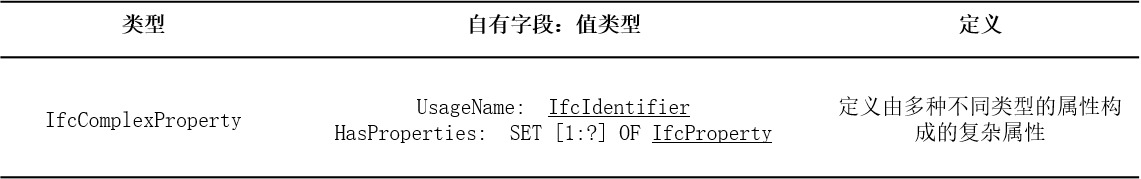
表4-1 IfcProperty定义

根据属性的值类型的不同，IfcProperty具有不同类型的派生类。图4-1给出了IfcProperty的派生子类图，图中的斜体内容表示此类是抽象类。



图4-1 Property派生子类图

IfcSimpleProperty定义简单属性类型，它的派生子类定对应定义了不同的属性值。它们的自有字段的定义如表4-2表示，可以看到，自有字段的值类型比较复杂，有定义类型（Defined Type）如IfcValue、IfcText等，有枚举类型（Enumeration）如IfcPropertyEnumeration，有选择类型（Select Type）如IfcObjectReferenceSelect，也有实体类型（Entity Type）如IfcUnit。

IfcComplexProperty定义复杂属性类型，它包含一个不同类型的属性的集合，也可以嵌套复杂属性类型。它的自有字段及定义如表4-3所示。

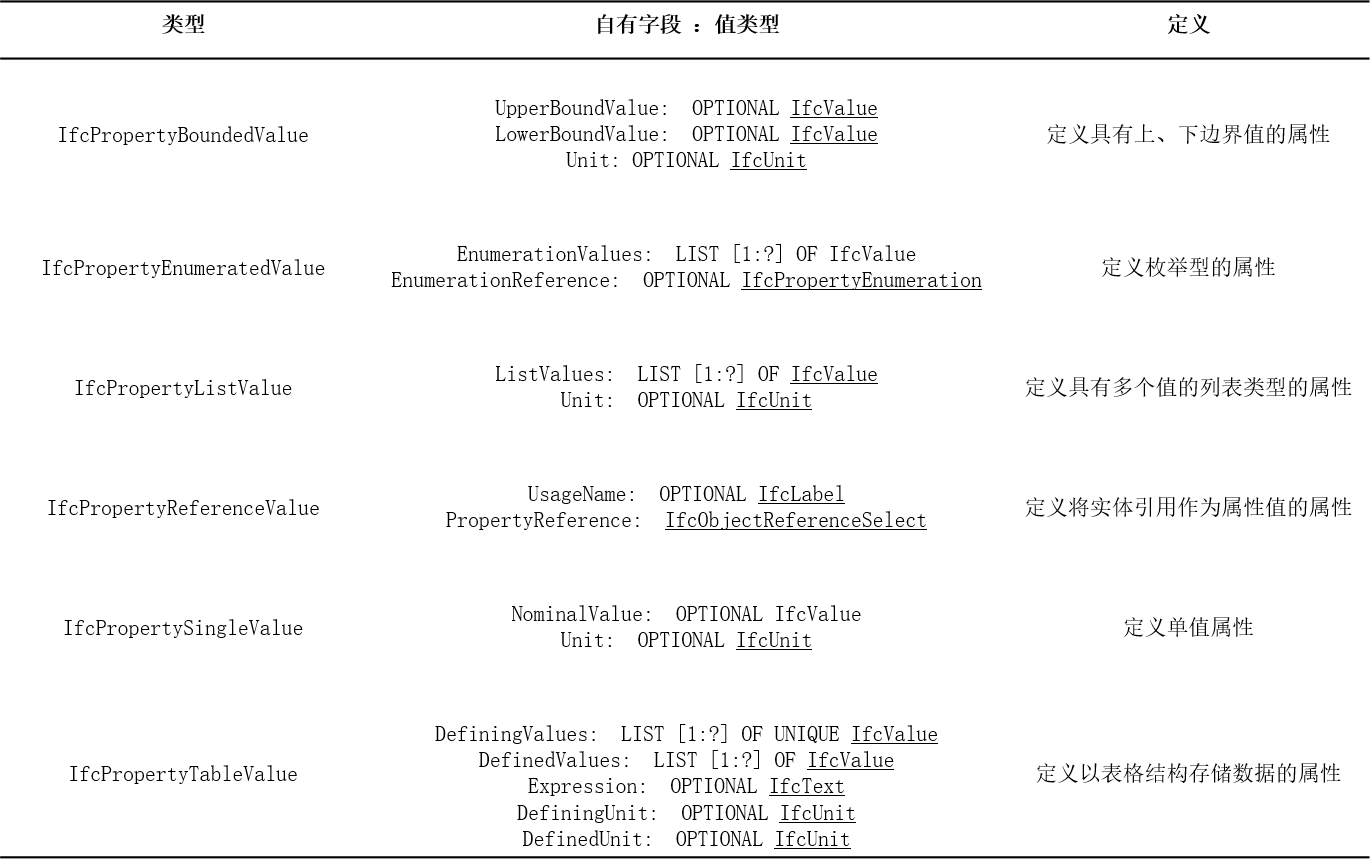
表4-2 IfcSimpleProperty派生子类的自有字段及定义

表4-3 IfcComplexProperty的自有字段及定义

### 属性集

IFC标准用属性集来组织对象的属性。IFC标准定义了不同的属性集类型。属性集的分类如图4-1所示。所有的属性集可以分为静态属性集和动态属性集，动态属性集又分为预定义属性集和自定义属性集。所有的属性集类型都继承自抽象基类IfcPropertySetDefinition。IfcPropertySetDefinition派生子类图如图4-2所示。

1. 静态属性集

以自有字段的方式定义一组属性，只能用来描述特定IFC实体类型。静态属性集的类型及定义如表4-4所示。其中，以IfcDoorLiningProperties为例，其自有字段定义如表4-5所示。

1. 动态属性集

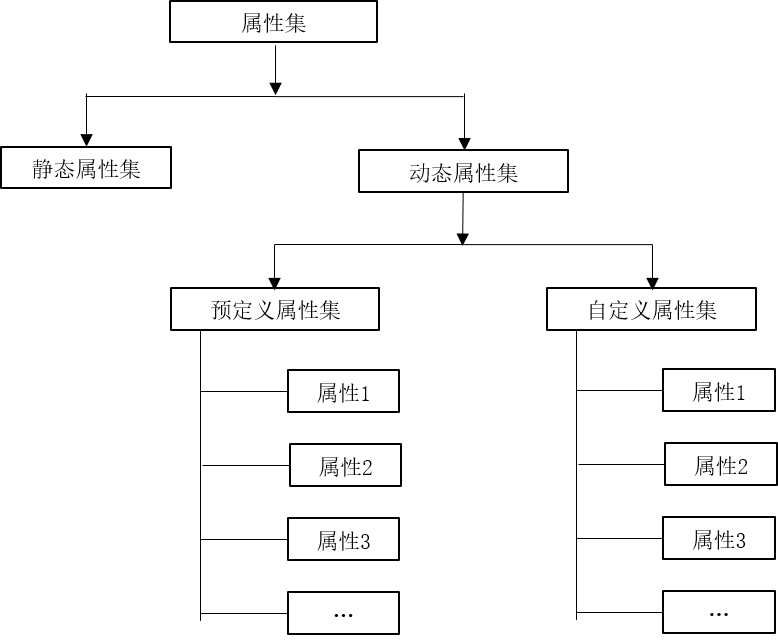
以包含一组不同的属性类型的方式来定义属性集。属性集类型是IfcPropertySet，其字段定义如表4-6所示。

* 1. 预定义属性集

由IFC标准定义，属性集名称以“Pset\_”开头，包含一组预定义的属性类型集合，用来描述特定的IFC实体。以Pset\_RailingCommon为例，其适用于实体类型为IfcRailing，属性定义如表4-7所示。

* 1. 自定义属性集

可以由用户自定义属性集名称和一组属性类型集合，它并不限于描述某一类IFC实体，具有更强大的表达能力。



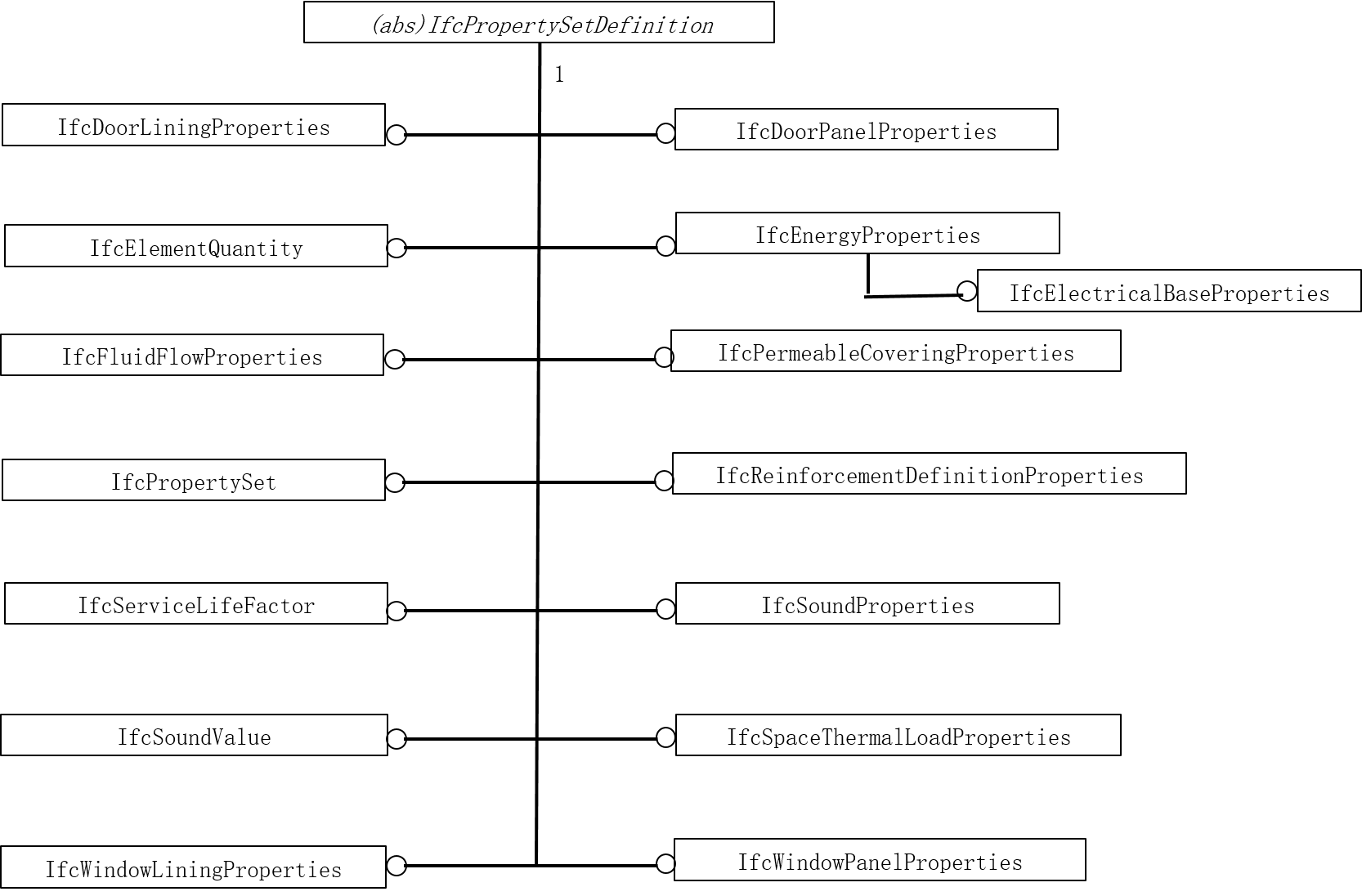
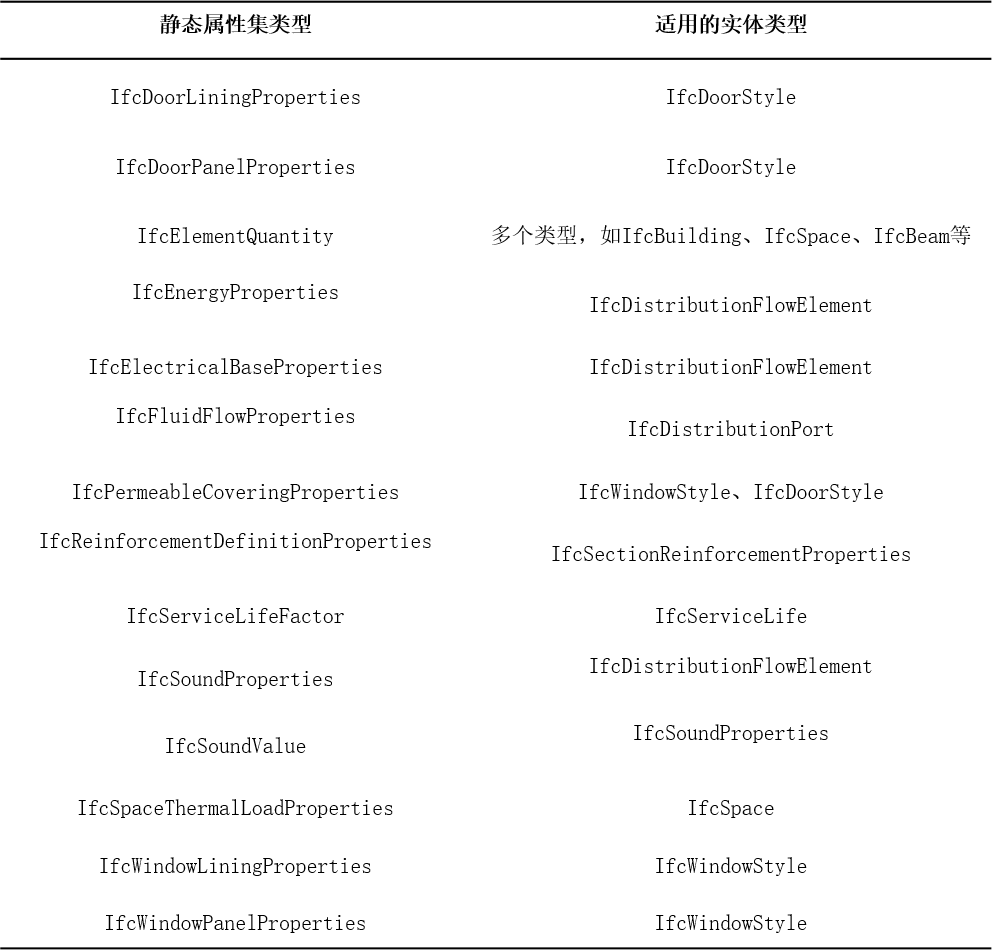
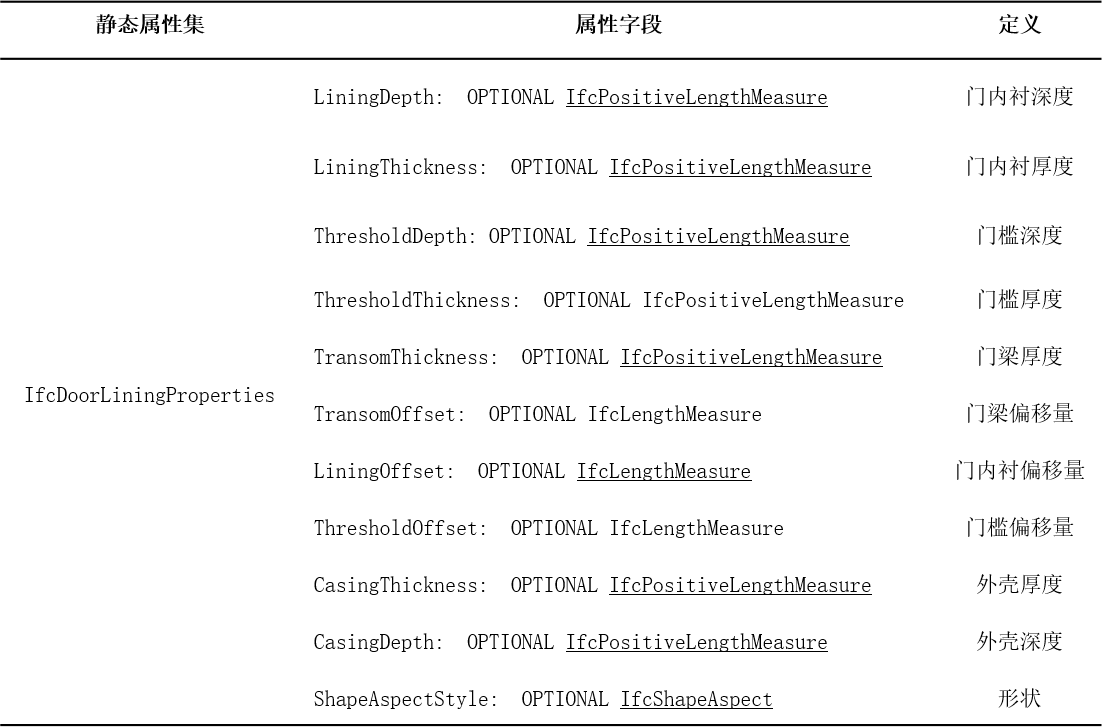
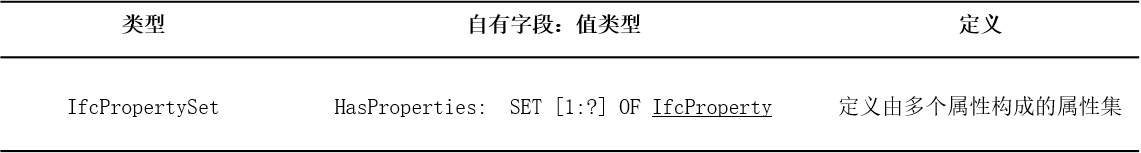
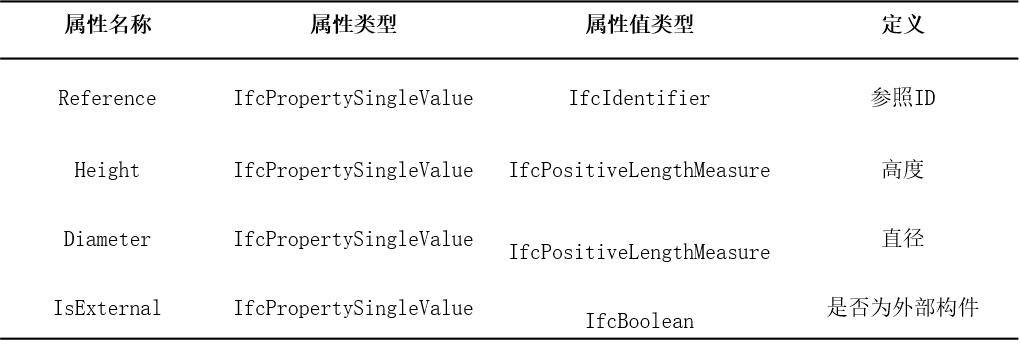
图4-1 属性集分类

图4-2 IfcPropertySetDefinition派生子类图

表4-4 静态属性集类型及适用的实体类型

表4-5 IfcDoorLiningProperties自有字段定义

表4-6 IfcPropertySet自有字段及定义

表4-7 预定义属性集Pset\_RailingCommon的属性定义

### 属性集关联机制

在IFC标准中属性集描述的信息通过以下两种方式关联到IFC实体：

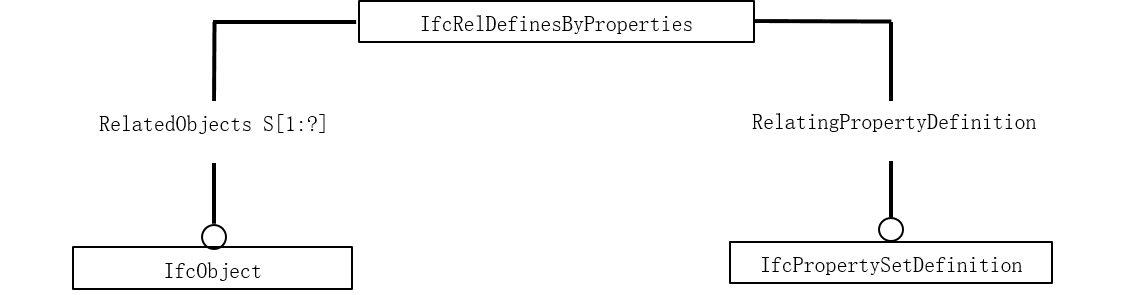
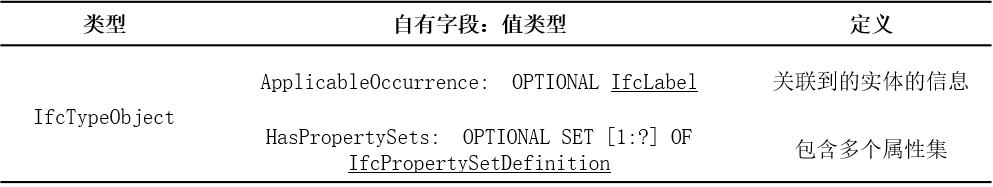
1. 通过IfcRelDefinesByProperties关系实体将IfcPropertySetDefinition对象与IfcObject对象关联。多个具有相同属性集的IFC对象可以通过同一个IfcRelDefinesByProperties关系实体关联到一个属性集，这样可以实现属性集的重复使用。同时一个IFC对象可以通过多个IfcRelDefinesByProperties来关联到不同的属性集，这样就可以借助多个属性集来表达丰富的属性信息。IfcRelDefinesByProperties关联机制如图4-3所示。

图4-3 IfcRelDefinesByProperties关联机制

1. IfcTypeObject的HasProperties字段直接引用一个或者多个IfcPropertySetDefinition对象，如表4-8所示。

表4-8 IfcTypeObject关联属性集

### 属性数据特点总结

IFC标准通过属性、属性集的定义提供了灵活的描述IFC实体的能力。总结属性数据的特点如下：

1. 类型丰富，结构复杂。属性包括简单属性和复杂属性，复杂属性支持嵌套。涉及到的属性集定义包括静态属性集、动态属性集，动态属性集又包括预定义属性集和自定义属性集。
2. 与IFC实体关联关系复杂。多个IFC实体可以关联到一个相同的属性集，一个IFC实体也可以与多个属性集关联。

以上特点为属性数据的存储带来了一定的难度。在4.2小节，本文将介绍如何进行属性数据的存储。

## IFC实体的空间关系数据模型概述

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。其中核心层定义了IFC实体模型的整体框架，不仅包括IFC实体的位置和几何形状等，同时也定义了IFC实体之间的关系。这些关系表现为空间结构上的组成、包含等多种关系，与IFC实体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，某两面墙相互连接等。

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实体之间的空间关系就定义在IfcRelationship当中。

IfcRelationship表示了不同的

IfcRelationship的派生类有IfcRelAssigns、IfcRelAssociates、IfcRelConnects、IfcRelDecomposes和IfcRelDefines五类。它们的定义如下：

* 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。
* 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。
* 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。
* 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。
* 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实体的空间关系，因此在设计空间关系数据的存储模型时，重点考虑对这两类及其派生类的存储。

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实体之间的空间关系数据的存储和查询相关技术。

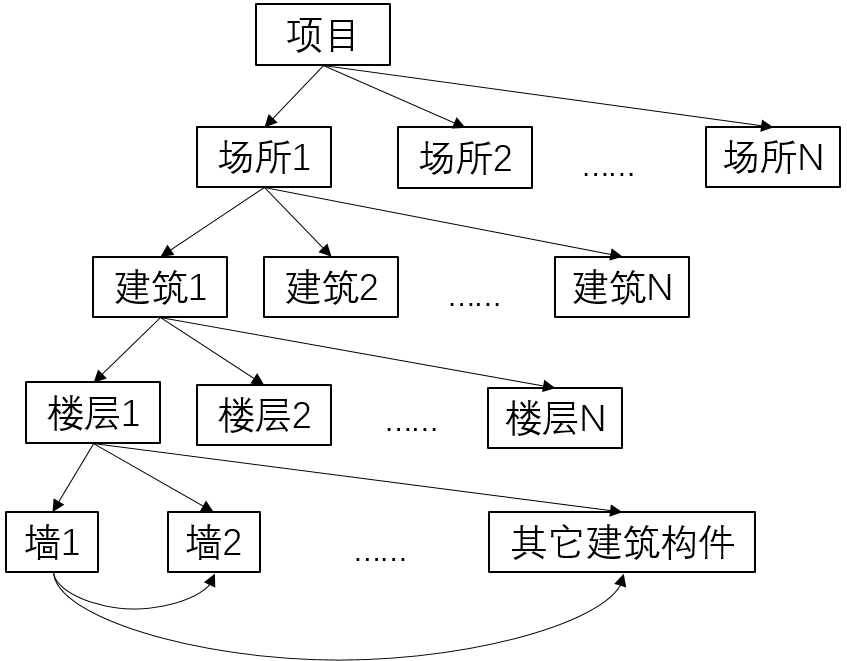
在现实世界中，建筑元素的关系可以是：某个项目包含很多场所，每个场所包含很多建筑，每个建筑包含很多楼层，每个楼层包含很多建筑构件，如门、窗、墙、梁、板、洞等等。而建筑构件之间也有多元化的关系，比如某两面墙相连接、

图5-1现实世界中的建筑元素的空间关系图

某面墙上有四扇窗等等。这些关系可以形象化地展示为图5-1.

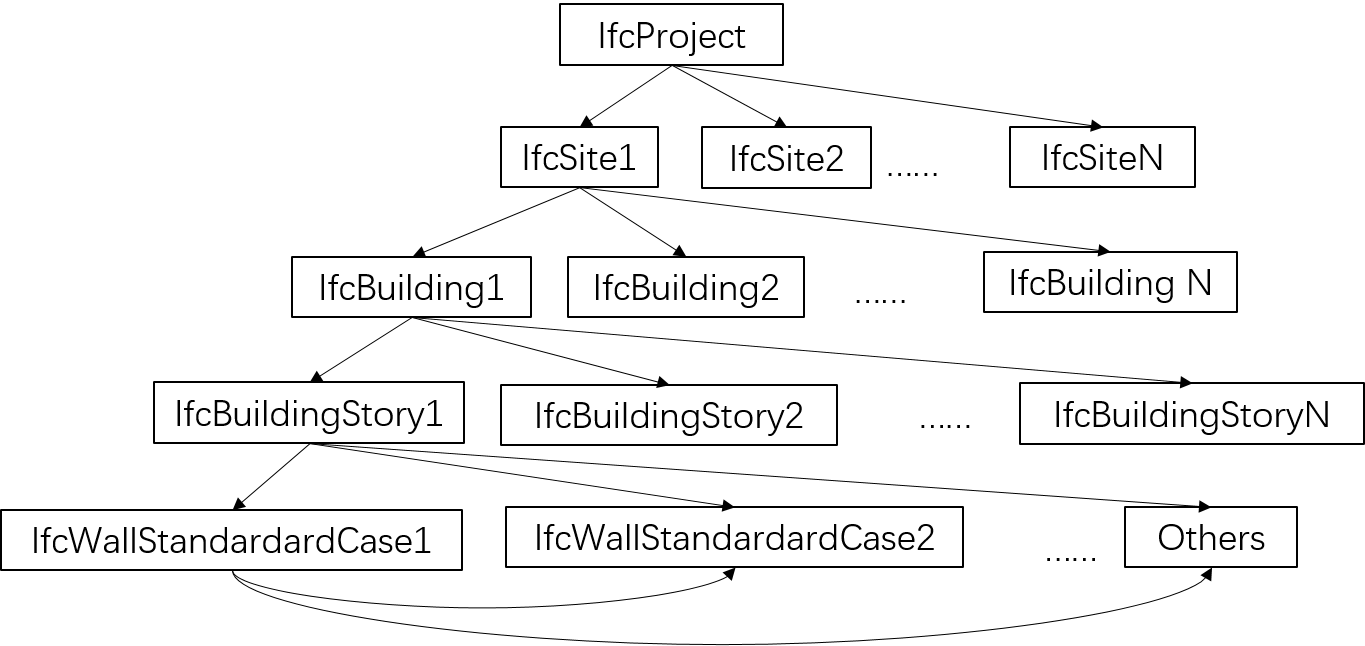
依照4.1小节对于IFC标准中关系的介绍，如果用IFC对象来表示上面图中的建筑元素，则对应的是图5-2。

图5-2 IFC对象的空间关系示意图

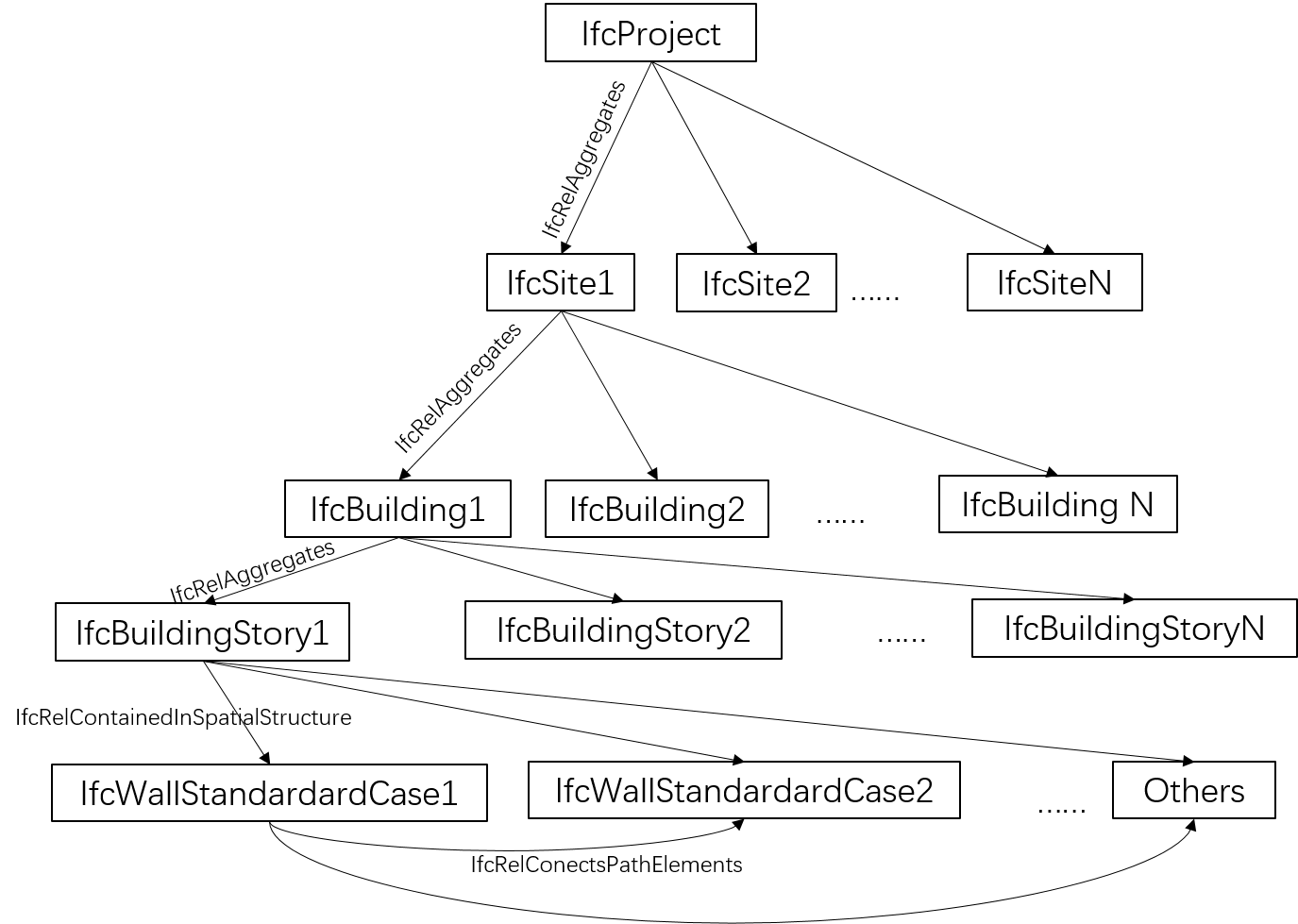
在上图中我们用一条带箭头的连线来代表IFC实体之间的空间关系。而这些关系在IFC标准中是用IfcRelationship的派生类来定义的。例如，IfcProject包含了很多IfcSite，这个包含关系用IfcRelAggregates来表示。而IfcWallStandardCase连接了另一个IfcWallStandardCase，这个连接关系用

图5-3 IFC实体的空间关系数据模型

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实体的空间关系模型，如图5-3。为了简洁，只标出了部分关系。

图5-3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5-3中所展示的IFC实体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实体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实体的空间关系的查询和应用。

## 图数据库技术

### 图数据与图数据库

图数据库源起欧拉和图理论，也可称为面向/基于图的数据库，对应的英文是Graph Database。图数据库的基本含义是以“图”这种数据结构存储和查询数据，而不是存储图片的数据库。它的数据模型主要是以节点和关系（边）来体现，也可处理键值对。它的优点是快速解决复杂的关系问题。

图具有如下特征： 包含节点和边；节点上有属性（键值对）；边有名字和方向，并总是有一个开始节点和一个结束节点；边也可以有属性。图可以说是顶点和边的集合，或者说更简单一点儿，图就是一些节点和关联这些节点的联系（relationship）的集合。图将实体表现为节点，实体与其他实体连接的方式表现为联系。

图将实体表现为节点，实体与其他实体连接的方式表现为联系。我们可以用这个通用的、富有表现力的结构来建模各种场景，从宇宙火箭的建造到道路系统，从食物的供应链及原产地追踪到人们的病历，甚至更多其他的场景。

通常，在图计算中，基本的数据结构表达就是：

图数据库存储一些顶点、边与表中的数据。它们用最有效的方法来寻找数据项之间、模式之间的关系，或多个数据项之间的相互作用。

一张图里数据记录在节点，或包括的属性里面。最简单的图是单节点的，记录了一些属性。一个节点可以从单属性开始，成长为成千上亿，虽然会有一点麻烦。从某种意义上讲，将数据用关系连接起来分布到不同节点上才是有意义的。

图计算是在实际应用中比较常见的计算类别，当数据规模大到一定程度时，如何对其进行高效计算即成为迫切需要解决的问题。大规模图数据，例如支付宝的关联图，仅好友关系已经形成超过1600亿节点、4000亿边的巨型图。要处理如此规模的图数据，传统的单机处理方式显然已经无能为力，必须采用由大规模机器集群构成的并行图数据库。

在处理图数据时，其内部存储结构往往采用邻接矩阵或邻接表的方式，在大规模并行图数据库场景下，邻接表的方式更加常用，大部分图数据库和处理框架都采用了这一存储结构。

随着数据量的不断变大，图数据库也变成了大数据时代的研究热点。

### Neo4j概述

Neo4j是基于Java开发的高性能的、高可靠的、高可扩展的NOSQL图形数据库。Neo4j支持所有的图数据变更操作放在事务中处理，确保数据的一致性。Neo4j单个服务器实例可以应对有着大量节点和关系的复杂图数据模型，处理能力可达到数十亿级别。此外，Neo4j还提供了非常快的图算法，如图遍历、最短路径等。Neo4j可以作为嵌入式的图形引擎用于各种需要快速开发的图应用当中，其轻量、高性能的优势使其越来越受到关注。

下面分别介绍Neo4j的基本概念、查询语言和索引。

1. Node、Relationship、Property、Label

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。同一类型的节点还可以被赋予Label（标签）。Neo4j会根据Label将节点组织为不同的set（集合），基于Label来进行节点的查询将使得查询不再是面向整个图，而是面向某些set。Label是可选的，一个节点可以没有Label。同时，一个节点也可以有多个Label。通过节点、关系属性和标签可以构建出一个大型的图结构，再通过一些列的图操作来进行数据的管理和应用。这就是Neo4j的图数据模型。下图是一个简单的图模型：

图2-9 简单图模型

图2-9是一个含有3个节点的图模型,两个带有“Person”标签的节点，分别有两个属性，“name”和“born”。还有一个带有“Movie”标签的节点，有两个属性“title”和“released”。“Tom Hanks”节点（我们暂且以属性值标识这个节点）与“Forrest Gump”有一个“ACTED\_IN”的关系，且关系具有属性“roles”，其语义是Tom Hanks出演了电影Forrest Gump，出演的角色是“Forrest”。另一个关系“DIRECTED”，表示“Robert Zemeckis”执导了影片“Forrest Gump”。

Neo4j提供了图遍历的traverse API，以某一个节点为起始节点，可以采用深度优先或者广度优先的遍历算法遍历整个图中的节点。

1. Cypher查询语言

Neo4j提供了Cypher查询语言。Cypher是一种声明式的图查询语言，允许对图数据进行有效的查询和更新。Cypher是一个比较简单但语言非常强大的语言。非常复杂的数据库查询可以很容易地通过Cypher来表达，这很大程度上方便了开发者或用户对Neo4j做点对点模式（ad-hoc）的查询操作和其它数据库操作。Cypher参考了SQL的结构，使用各种子句建立查询。子句连接在一起，并且彼此之间提供中间的结果集。例如，一个MATCH子句的匹配结果是下一个子句的上下文。下面是部分查询子句：

MATCH：匹配模式，这是从图形中获取数据的最常用方法。

WHERE：过滤条件。

RETURN：返回所需要的。

基于图2-9的模型，如果要查找Tom Hanks演过的电影，Cypher语句可以写为如图2-10形式：

图2-10

需要注意的是，Cypher并不是一个静态的语言，不断发展的新版本会加入一些新的功能，有时候还会删除一些旧的功能。例如在Cypher 2.2之前的版本中，START子句被用来查找指定的ID的节点作为起始节点，但是在2.2及以后的版本中，START被完全禁止了，取而代之的是用MATCH子句来查找起始节点。每一个版本的Neo4j支持某些版本的Cypher，如Neo4j 3.2支持的Cypher版本有3.2、3.1、2.3。当升级Neo4j版本时，需要注意查看相应的Cypher版本是否支持。

1. Legacy Index和Schema Index

Neo4j提供了属性上的索引，使得用户可以通过属性值快速找到节点。索引包含Legacy Index 和Schema Index两种类型，它们都是基于Lucene实现的在Neo4j 2.0版本之前的索引被称为Legacy Index。Legacy Index能够提供全文检索的能力，而在Schema Index当中没有。这也是Neo4j 2.0版本及以后的版本还保留着Legacy Index的原因之一。使用Legacy Index往往需要一个“起始节点”，使用Cypher语句时需要借助START子句来调用。

在Neo4j 2.0 及以后的版本中Legacy Index不再是官方推荐的索引，而是推荐使用Schema Index。Schema Index需要基于Label和Property使用。它进行了一些优化，使得MATCH子句中查询节点变得更快。

创建Schema Index的Cypher语法如图2-11：

图2-11

其中的Label和Property分别对应具体的标签和属性。

# 系统总体架构

由于IFC标准采用面向对象的方法设计而成，具有面向对象的复杂特性，IFC标准定义的实体类型多样、结构复杂、实体之间相互引用复杂，当前已有的BIM数据库无法解决IFC关系数据查询效率低下的问题，本文提出了基于Neo4j的BIM数据库方案，将IFC数据模型转换为图数据模型，充分利用Neo4j在查询、管理图数据等方面的优势，并通过增加缓存层和扩展插件服务接口来进一步解决IFC关系数据的查询效率等问题。

本章主要是介绍基于Neo4j的BIM数据库系统的数据处理过程的设计和实现。首先介绍系统的总体架构，然后逐一介绍各个模块的设计。

## 系统总体架构

按照数据处理过程，该系统的模块可以分为：IFC文件解析模块、数据模型转换模块、数据存储和查询等模块，其总体架构如下图3-1：



图3-1 系统总体架构（这个图应该更清晰全面）

系统中各个模块的执行流程是：

1. 使用IFC文件作为数据来源，读取IFC文件到系统。
2. 文件解析模块将解析IFC文件为内存中的IFC实体对象数据集合。
3. 数据模型转换模块将对IFC实体对象数据进行处理，转换为Neo4j的图数据模型。
4. 存储模块将图数据存储和索引到Neo4j数据库中去。

## 子模块设计与实现

### 文件解析模块

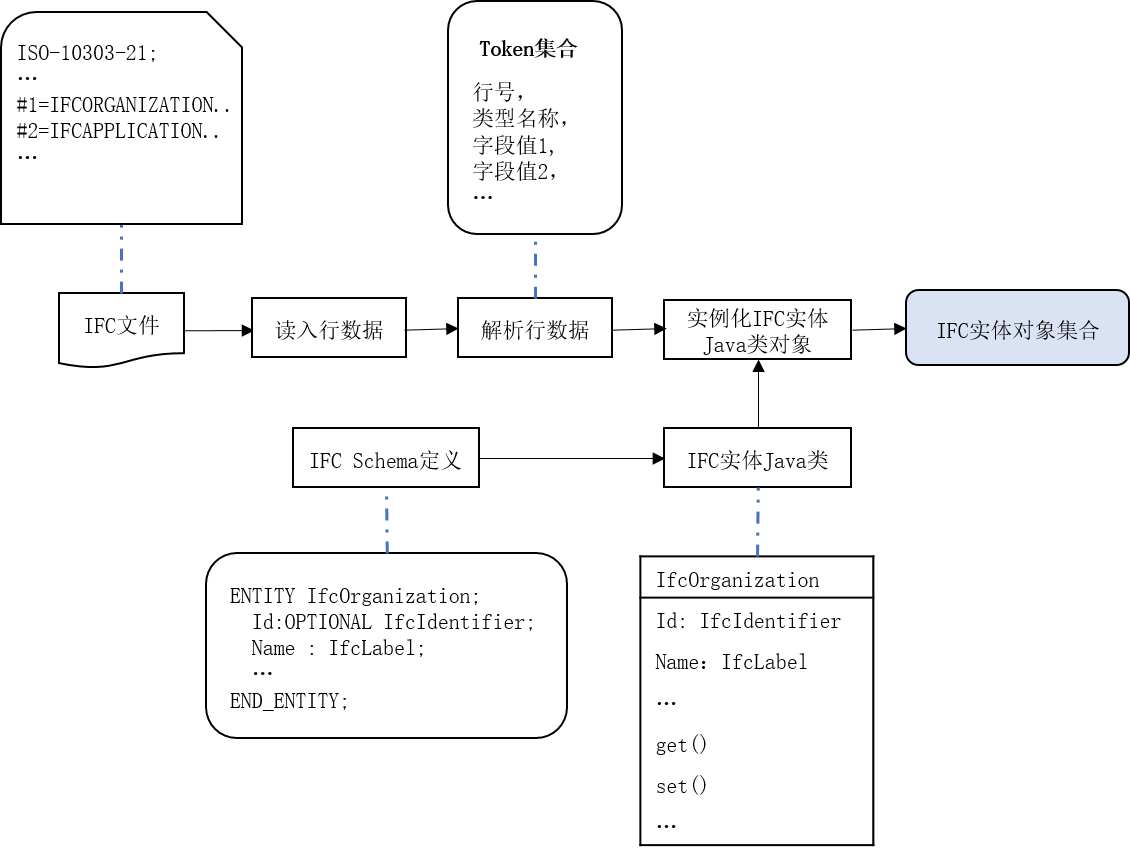
IFC文件解析模块的主要目的是解析IFC文件，得到IFC实体的Java类对象的集合。解析过程如下：逐行读入IFC文件的每条数据字符串，解析获得token集合，包括行号、类型名称、多个字段值等。行号是IFC对象的ID，类型名称是IFC对象的类型，字段值对应了IFC Schema中对IFC实体类型的Express定义。基于IFC schema的Express定义语言，构建IFC实体类型、定义类型、枚举类型、选择类型对应的Java类。然后结合Token集合中的IFC实体的数据实例化相应的IFC类对象。所有行数据解析完毕，最终得到IFC实体对象集合。解析流程图如图3-2所示。

图3-2 IFC文件解析流程

### 存储和索引模块

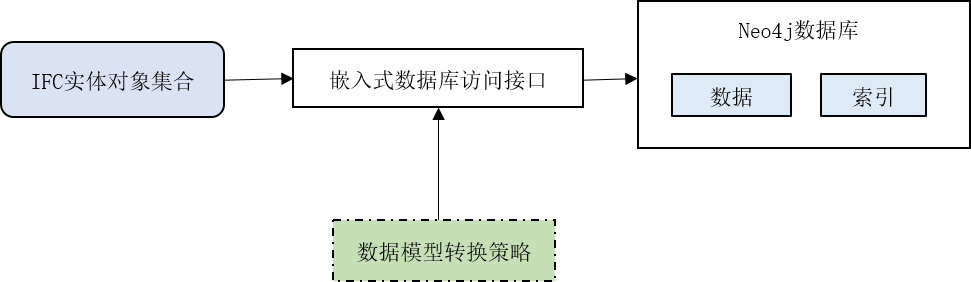
存储和索引模块的主要目的是将IFC实体对象的数据按照一定的策略转化为对应的图数据模型，存储到数据库中去，并建立索引以加速查询。这个过程如图3-3所示。

图3-3 存储和索引（不需要这个策略图了）

在此模块，内存中的IFC实体数据模型包含不同类型的实体对象，不同类型的实体对象按照不同的方式转换为Neo4j中的节点、关系、属性等，然后采用批量处理接口来存储和索引。

其中最重要的是数据模型转换的策略，实体类型不同，转换策略也不同。IFC数据模型中的实体类型可以用下面的集合来表示：

M = {R，E}

其中RL表示IfcRelationship的派生子类型，E表示除了RL以外的实体类型，转换的策略是：

1）属于E类型的实体对象*e*，直接创建一个*node n*，*e*的基本值类型字段作为*n*的属性*property*。对于*e*的引用类型字段*filed*引用了其它实体对象*e­­1*，为*filed*创建一个*relationship*，*relationship*始于*e*且指向*e­­1*。

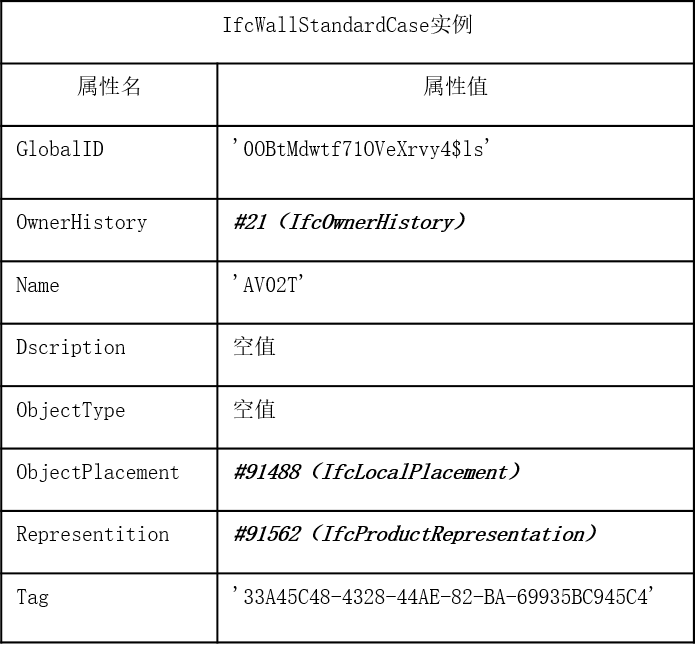
例如，图3-4是IfcWall对象实例的数据，图3-5对应在Neo4j中创建的节点和关系。解释一下这个规则。

图3-4 IfcWallStandardCase实例信息

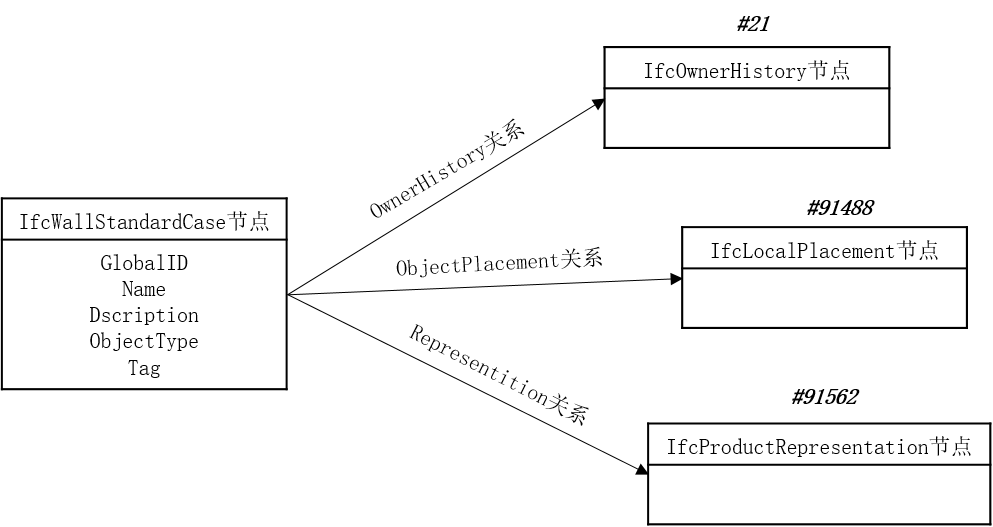


图3-5 IfcWallStandardCase实例对应在Neo4j中创建的节点和关系

2）属于R类型的实体对象*r*，直接创建一个*node n*，*r*的基本值类型字段作为*n*的属性*property*。对于*r*的引用类型字段*filed*引用了其它实体对象*e­­1*，为*filed*创建一个*relationship*，*relationship*始于*r*且指向*e­­1*。对于*r*的“Relating”字段引用的其它实体对象*e­­2*和“Related”字段引用的其它实体对象集合*set*，针对*set*中的每一个实体对象*e3*，创建一个始于*e2*且指向*e3­­*的*relationship*。

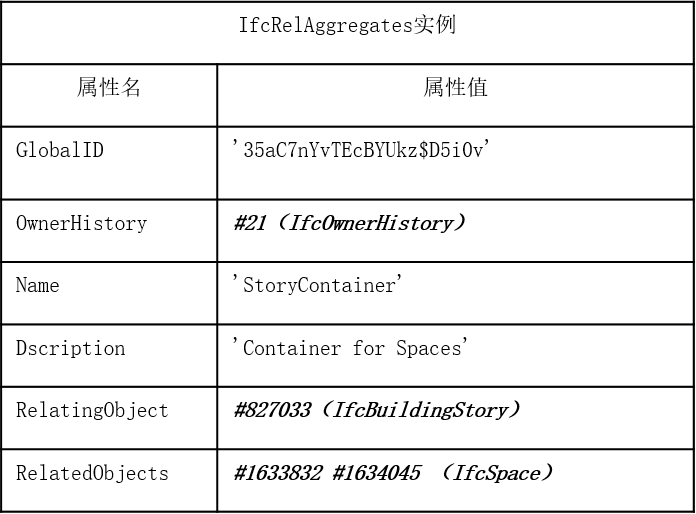
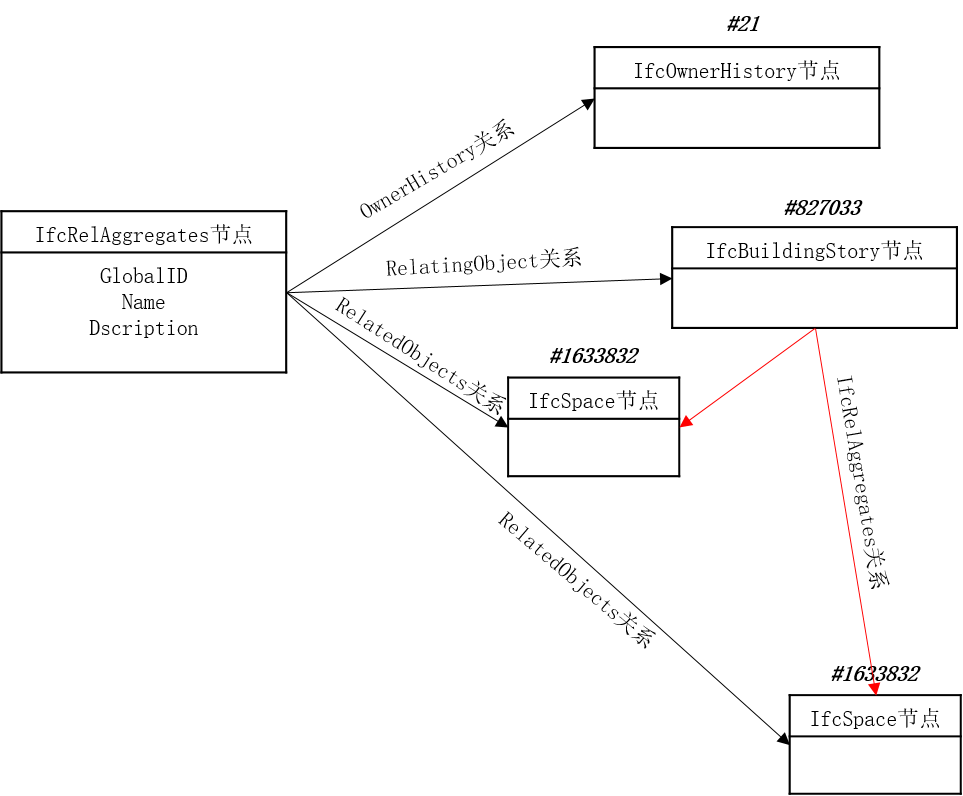
例如，图3-6是IfcRelAggregates的数据，图3-7对应在Neo4j中创建的节点和关系。

图3-6 IfcRelAggregates实例信息

图3-7 IfcRelAggregates实例对应在Neo4j中创建的节点和关系

### 查询访问模块

查询访问模块为用户提供了访问数据库的界面。数据库可以通过7474端口进行访问，提供了cypher语句和REST访问接口服务。本文基于插件机制拓展了数据库端的REST服务，使得用户可以直接访问GET接口来进行特定路径的查询。

对插件服务进行介绍。

# 基于Neo4j的BIM数据库关键技术研究

以往的BIM数据库在关系数据查询方面存在效率低下的问题，而基于Neo4j的BIM数据库采用将IFC数据模型转换为图数据模型的思路，利用Neo4j在查询、管理图数据方面的优势，实现IFC关系数据的高效查询。为了更近一步地降低IFC关系数据查询的响应时间，在分析了Neo4j有关源码和Neo4j的缓存模型之后，本文阐述了其缓存模型的局限性，并提出了Neo4j内核中增加读缓存层的实现方法，并采用了面向图数据的缓存策略，这种缓存策略对于其它面向图数据的缓存算法具有借鉴意义。同时，本文研究了针对IFC关系模型的特定路径查询场景及问题，提出了基于BloomFilter的查询算法，提高了特定路径查询的效率。

## 缓存技术研究

### Neo4j页缓存架构

缓存技术是提高系统性能的一种重要手段，在数据库系统领域有着广泛的使用。缓存的目的是将相应数据存储起来以避免数据的重复创建和处理，降低系统的资源消耗，提升系统的响应速度。Neo4j的缓存架构如图4-1所示。

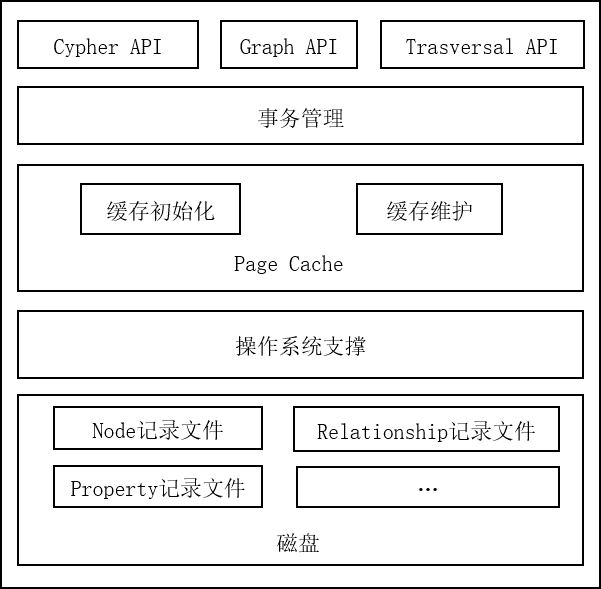


图 4-1 neo4j缓存架构

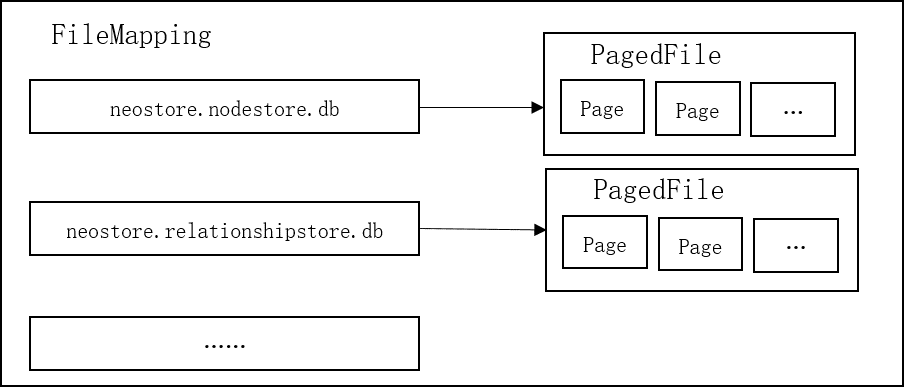
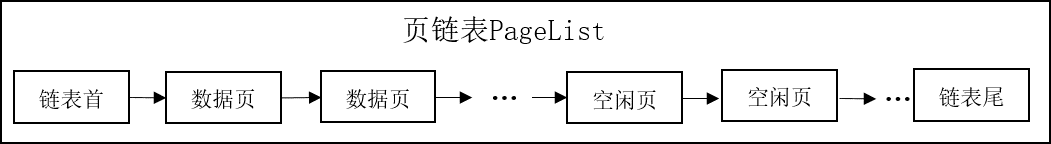
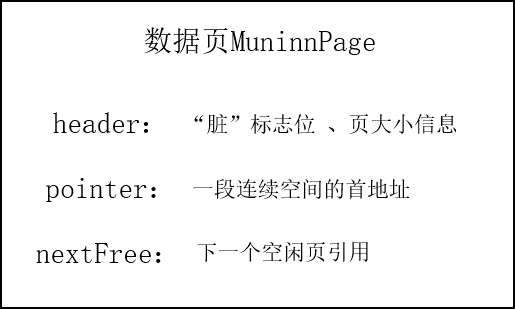
Neo4j采用Page Cache缓存层，将磁盘上的数据记录文件映射到内存堆中的文件。映射的内存文件由页组成，每个页固定大小，由系统配置文件决定。Neo4j根据每个记录文件的大小计算出文件所对应的页数，形成页面链表，管理空闲页面和脏页面。当空闲页面数量小于阈值（阈值不超过总页数的一半，默认为30）时，根据LFU（Least Frequently Used）算法将脏页面置换出内存。Page Cache层文件映射的底层实现依靠操作系统的内存映射技术，页面的置换也交给操作系统来实现，也就是说，最终由操作系统决定哪些页面留在内存中、哪些页面被置换出内存。Page Cache的主要内存结构如图4-2所示。

图 4-2 PageCache层的主要内存结构

1. 文件映射FileMapping：保存磁盘文件到内存文件PagedFile的映射，PagedFile相当于磁盘文件在内存中的形式，通过游标来访问；
2. 页链表PageList：包含数据页MuninnPage和空闲页FreePage两种页对象，数据页在链首，空闲页在链尾，在PageCache初始化时创建。由系统配置的PageCache内存占用的大小（不能小于240KB）与页大小（默认为8KB）可以确定页总数，其中空闲页数量由配置确定（规定不小于30）。
3. 数据页MuninnPage：包含了页的状态信息、描述信息等，并有一个指向一段连续空间的首地址，缓存的数据就在这段连续的空间上。

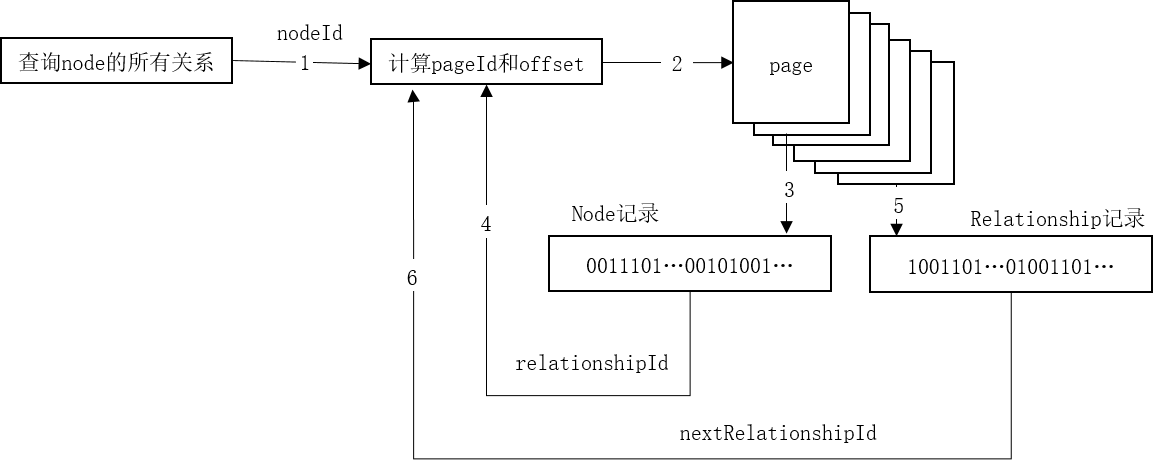
当需要查找某个Node关联的所有关系时，首先由nodeId和节点文件对应的内存文件PagedFile的页大小计算出页地址pageId和页内偏移量offset，此时使用游标技术（cursor），定位到内存中相应位置读取node记录，解析出relationshipId。然后根据relationshipId计算出页地址PageId和页内偏移量offset，使用游标定位到内存中相应位置读取relationship记录。之后通过relationship记录中的nextRelId值继续查找下一个relationship记录，相当于遍历relationship链表结构，重复2、5、6步骤，直到所有的relationship处理完毕。过程如图4-3所示。

图4-3 查询node的所有关系的过程

由于PageCache直接将文件内容缓存到内存堆中，pageCache分配的内存越大，页面换入换出的次数越少，从而性能表现越好，但是这也造成了巨大的资源浪费。实际上，页面中只有一部分数据才经常被访问，而其它数据被访问的可能性较小。Neo4j默认的缓存区大小结合了系统的实际运行时的内存占用情况，设置为运行时可用内存的50%，但仍然存在很大的资源浪费。另一方面，在pageCache命中时虽然避免了对磁盘上数据的访问，提高了查询的效率，但是每次需要重复进行计算Id->访问数据页->解析数据的过程，依然消耗了不少的时间。

描述一下页缓冲和单缓冲的局限性。

为了能够更高效地利用内存资源，并进一步降低关系数据查询的响应时间，本文提出了在PageCache层之上增加读缓存层，读缓存层面向图数据模型中的关系数据进行缓存。

### 对象级缓存策略

缓存策略与页级缓存的访问协同，创建缓存的过程、替换的过程、查询的过程。

缓存算法通常是将用户多次请求的、不改变的数据缓存起来，减少数据的重复处理对系统资源的消耗，同时降低用户请求的响应时间。由于计算机的内存空间限制，缓存区需要实现缓存内容的替换。好的替换策略有利于发现热点数据，将最近经常被访问的数据保留在内存中。关于缓存替换策略已经有了较为长期的研究，包括LRU（Least-Recently-Used）最近最少使用替换策略、LFU（Least-Frequently-Used）最少使用频率替换策略、Clock时钟替换策略等。其中LRU替换策略是目前数据库系统中普遍采用的设计。它的基本思想是如果数据被最近使用过，那么将来被访问的概率也高，如果最近没有被访问，那么将来被访问的概率也比较低。基于这个思想，当存在热点数据时，LRU的缓存效率很好，并且LRU算法具有实现简单的优点，因此在数据库系统等各领域得到了广泛的应用。但是同时LRU算法固有的缺陷也不容忽视，也就是没有考虑到访问频度的问题。在短期内发生的偶发性的批量数据访问往往使得大量新数据项进入缓冲区，而大量在长期内频繁被访问的数据被替换出缓冲区，造成“缓存污染”。LFU算法则解决了这种“缓存污染”问题。LFU的基本思想是如果数据过去被频繁访问，那么将来被访问的概率相对较高。LFU算法对每个数据项保留一个引用计数，所有数据项按照引用计数排序，替换发生时优先替换出引用计数最少的数据项，从而能避免了短期内发生的偶发性的操作造成的“缓存污染”。这两种缓存替换策略对于本文面向图数据的缓存替换策略具有启发意义。

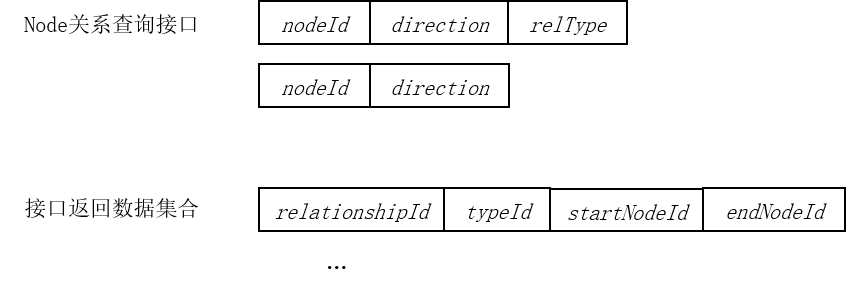
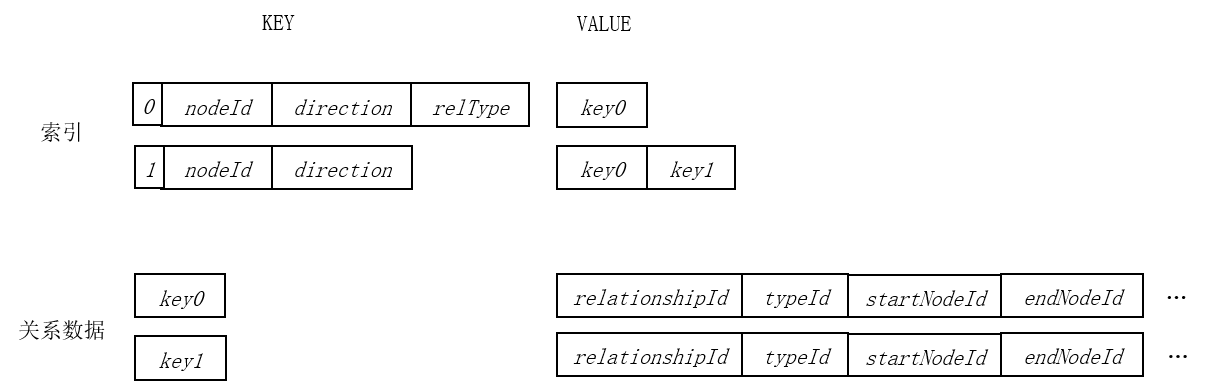
以上所介绍的缓存替换策略关注的是缓存数据项地位一致的缓存管理，而在Neo4j中节点度数不同，一个节点可能没有关系，也可能有成千上万个关系，节点对应的关系数据量随节点度数增加，不同节点缓存项的内存开销与节点度数成正比。研究neo4j内核中对于节点的关系数据访问接口，其中涉及到的接口参数和关系数据结构如图4-4所示。查询接口分为对特定direction（有in、out、both方向）和特定关系类型relType的关系数据查询。接口返回的Node的每项关系数据是固定格式和大小的。

图4-4 Neo4j中Node关系查询接口的数据结构

在设计缓存数据结构时，考虑到有两种查询接口，因此增加索引结构，形成的缓存数据结构如图4-5所示。索引部分，KEY由四部分组成，其中0和1表示两种数据访问方式。索引的VALUE是节点的关系数据的引用，通过一个或者一组引用值可以在关系数据部分中查找到所有符合条件的关系。

图4-5 缓存项NodeRel数据结构

缓存结构中主要是关系数据的部分，一个节点的关系的数量即节点的度数直接决定了其需要缓存的数据项的内存开销。在考虑节点的缓存替换时，替换出访问频度低的节点有利于将访问频度高、未来访问概率大的节点保留在缓冲区中，提高命中次数。而替换出度数大的节点则可以为缓冲区提供更过的空闲空间，使得可以加入更多的缓存项，从而提高命中次数。结合节点的访问频度、缓存项的内存占用两个因素，本文提出了节点的缓存收益率这个概念。节点的缓存收益率的定义公式如公式（4-1）：

（4-1）

其中，表示节点缓存的收益，表示节点缓存的开销。本文将访问频度作为收益，考虑到节点的缓存项的内存占用与度数成正比，因此将节点度数作为开销，有效地简化了计算，缓存收益率的计算公式如（4-2）：

（4-2）

其中，表示节点的访问频度，表示节点的度数。

节点缓存收益率体现了缓存所能带来的性能收益的性价比，改进的缓存策略旨在将缓存收益率高的缓存项留在缓存区中，从而能在内存空间有限的情况下，从整体上提高缓存的效率。

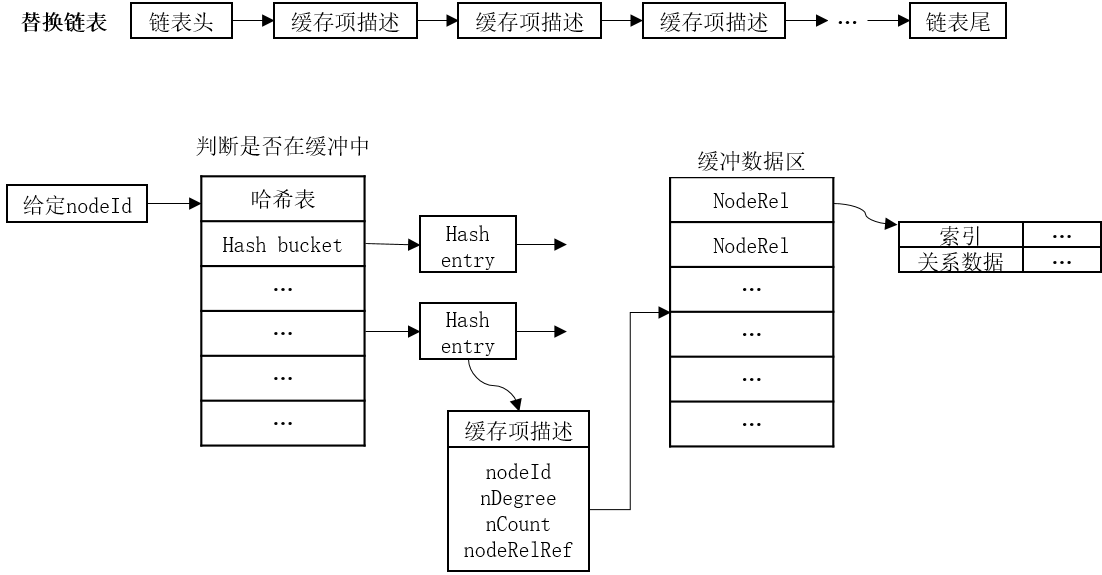
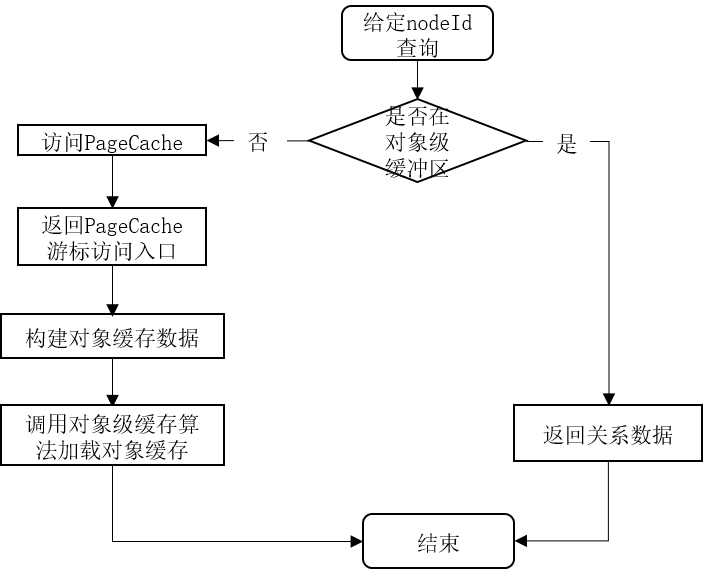
缓存替换策略维护一个单链表，表中每一项存储节点的的描述信息和缓存数据项NodeRel的相关信息，包括节点的nodeId、节点度数nDegree、访问计数nCount、缓存项nodeRel的引用、下一个表项引用next，初始为空链表。初始时，新加入的表项放在链表尾部，访问计数为1。当数据访问命中时，对应表项的访问计数增加1。当数据访问没有命中时，如果有空闲空间可以分配，直接分配空间给新的NodeRel数据项，同时增加新的表项至替换链表的尾部。如果没有空闲空间，则需要替换出一个或者若干NodeRel数据项，直到替换出能够满足需求的缓冲空间为止。替换时，首先计算当前链表中节点的缓存收益率并进行排序，低收益率的表项放在表头。淘汰时从表头开始，依次将对应的NodeRel数据项从缓存中删除，直到有足够的缓存空间来存储新的数据项。为了能够更快速地查找链表中的缓存数据是否命中，增加Hash表，存储节点的nodeId到链表中表项数据的映射，避免对链表进行遍历。缓存区的数据结构示意图如图4-6所示：

图4-4 缓存区结构示意图

改进的缓存策略并不像传统的缓存策略一样，在缓冲区满之后，每次缓存新的数据项都需要进行替换，当替换出节点度数较大的缓存收益率低的缓存项时可以带来较大的空闲空间，减少了后续的替换次数。由于替换时需要进行收益率计算和排序，会带来不小的开销。考虑到实际应用时，节点的收益率短时间内并不会发生急剧地变化，本文引入了一个替换周期T。在本次周期之内，并不重新进行收益率的计算和排序，替换时，按照上次计算后的链表顺序依次往链表尾部进行替换。替换周期T的大小是影响缓存实际运行效果的关键因素。T过大将会造成不能及时地更新数据项，可能会错误地换出历史数据中被频繁访问的数据，T过小则并不能有效地减少替换操作给系统带来的开销。在一个访问密度（单位时间内访问次数）比较稳定的系统内，T可以是一个固定值，而在访问密度变化较大的系统内，T则需要根据实际情况进行调整。

结合Neo4j系统，缓存访问的具体实现流程如图4-5所示。

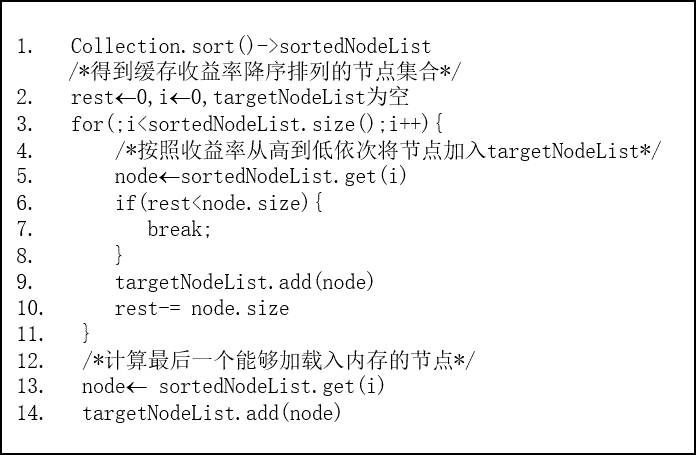
1. 给定nodeId进行关系查询时，数据访问接口首先查找对象级缓冲区，如果在对象级缓冲区命中，直接返回已缓存的关系数据。
2. 如果对象级缓冲区没有命中，就去访问系统原有的Page Cache。由于Neo4j采用游标技术来实现对数据的访问，因此访问Page Cache接口获取的是游标的入口。这部分是Neo4j原有的逻辑。
3. 将数据加载入对象级缓存区之前需要首先构建面向对象的缓存数据项，也就是前文提到的NodeRel数据。NodeRel数据需要通过访问系统的存储层接口，获取原始的Node记录、Relationship记录，解析出relationshipId、typeId、startNodeId、endNodeId，并通过其他接口获取节点的degree等信息，构建出上文所描述的缓存数据。

图4-5 访问缓存流程

### 缓存预装载策略

缓存预装载技术是指在数据访问之前将数据加载进入缓存，它可以在系统运行初期有效地利用缓存空间，提高系统运行效率。本文基于前文所提出的缓存收益率的概念，提出了缓存预装载策略，以提高系统在重启之后的运行效率。基本思想是根据系统的访问日志统计出节点访问次数，并计算出节点的缓存收益率。在固定的内存空间内加载不同内存占用的节点，使得缓存收益率总和最大，这是一个0-1背包问题。本文采用贪心算法来求解预装载节点集合的问题。

算法的伪代码如图4-6所示：

图4-6 求解预装载节点集

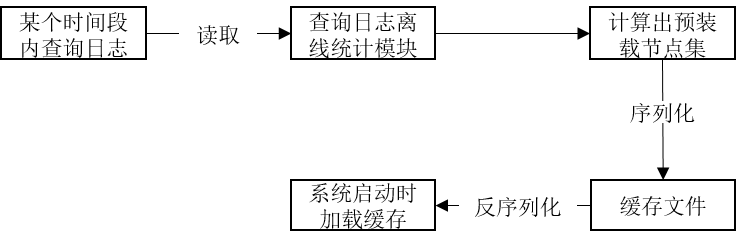
预装载策略的实现基于数据查询日志的统计。本文

图4-7 预装载框架

## 特定路径查询技术研究

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。

### 特定路径查询场景及问题

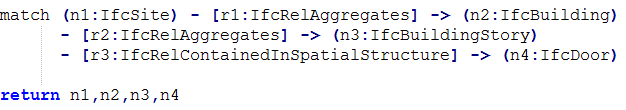
IFC标准中有一种特定的查询是，用户需要依次确定项目、场地、建筑、楼层，然后再找到楼层中满足一定条件的建筑构件，如门、窗、家具或其它零件。图3展示了这种关系在neo4j中的表示。在Neo4j图数据模型的定义中，从项目节点到建筑构件节点形成了一个路径，这个查询是一个特定的路径查询。其Cypher语句的简单形式如图5-7：

图5-7

又由IFC标准的定义可知，r1、r2、r3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表达式应为图5-8：

图5-8

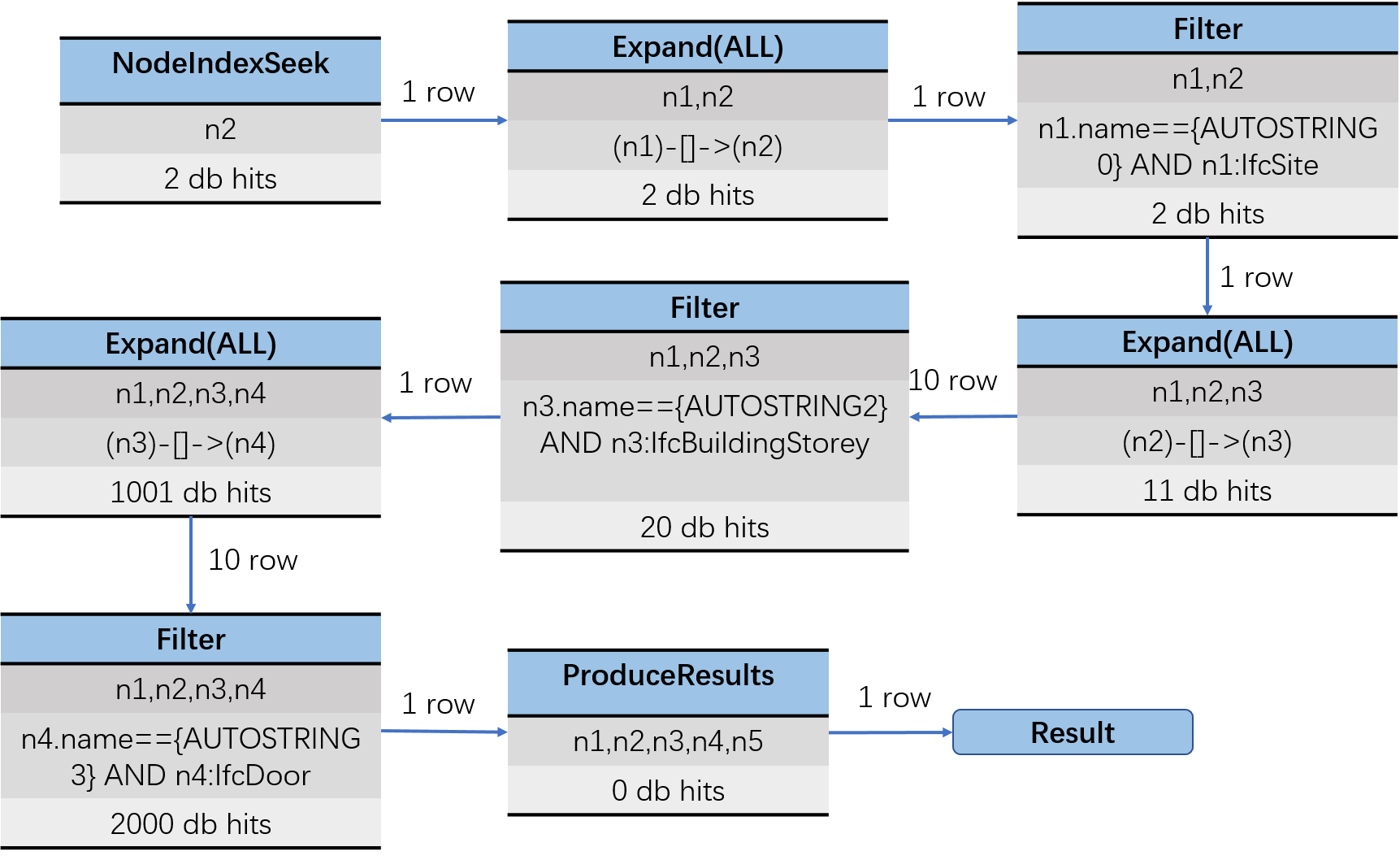
这个Cypher查询语句的完整的执行计划如图5-9：

图5-9

1. 执行NodeIndexSeek。访问索引“index on :IfcBuilding(name)”，返回节点n2。在这里假设只有一个IfcBuilding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IfcBuilding只有一个IfcSite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IfcSite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。

### 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。

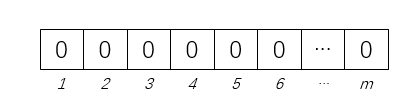
对于4.6.1小节中提到的特定的路径查询来说，如果我们能够把符合空间组成关系的IfcSite、IfcBuilding、IfcBuildingStory、IfcDoor（以及其它建筑构件）的ID按照顺序拼接起来，维护在BloomFilter当中。当查询来到时，将满足条件的各个节点的ID拼接起来，查看是否在BloomFilter当中，就可以知道，这个结果集是否存在。BloomFilter存在一定的误判。也就是说，如果这条路径并不在neo4j存储的图中，但是也有可能会判定为存在，此时用户得到的结果将会是由原始的Cypher查询得到的正确结果。同时，BloomFilter不会漏掉判断。这对我们是相当有利的，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：

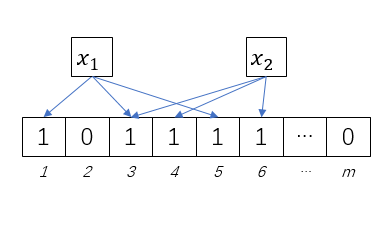
初始状态时，BloomFilter是一个包含m位的bit数组。每个bit都为0，如下：

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：

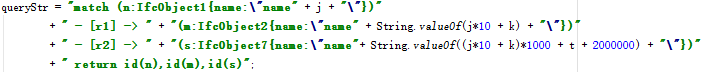
误判率与插入BloomFilter的元素个数n、bit数组大小m以及Hash函数个数k有关。研究表明，当Hash函数个数k时错误率最小。在错误率不大于E的情况下，m至少要等于才能表示任意n个元素的集合。

### 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足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查询分解为针对节点的Cypher查询。原始的查询格式比较固定，会是如下这种情况：

而分解后的查询会是如下这种情况：







1. 分别执行以上单个针对节点的查询，也就是3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。

以上便是基于BloomFilter进行特定路径查询的算法及相关实现。

### 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> 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IFC实体的ID与其所在的IfcBuildingStory的ID的映射。

基于这3个数据结构，可以构建一个IFC对象getStoryID()、getBdgID()、getSiteID()，得出IFC对象所对应的IfcBuildingStory、IfcBuilding、IfcSite的ID。然后按照从Ifcproject到IFC实体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。

# 系统实验

本章将对前文所阐述的关键技术点进行实验测试和分析，验证其有效性。

## 实验环境

本文以Neo4j community edition 3.0.6数据库作为研究对象，并在此基础上进行了改进。本文仍以Neo4j community edition 3.0.6作为验证实验的数据库。实验机器配置如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 运行环境 | CPU | MEM | OS |
| 单机 | Intel(R)CORE i5 | 8G | Win10 |

## 缓存算法实验

本文在Neo4j PageCache页缓冲机制的基础上引入了面向对象的缓存区，并提出基于节点收益率的改进的缓存策略。所以本小节的实验从两个方面来进行测试和分析，一个方面是比较原系统和引入对象级缓冲区之后的系统的关系查询性能表现，另一个方面是比较基于节点收益率的缓存策略与传统的缓存策略的命中次数。

### 数据集

实验数据集采用200M的IFC数据文件，解析、存储和索引到Neo4j数据库中。包含多少个节点，多少关系，平均关系是多少。

### 实验与分析

1. **对象级缓冲策略和原系统的缓存策略的对比实验与分析**

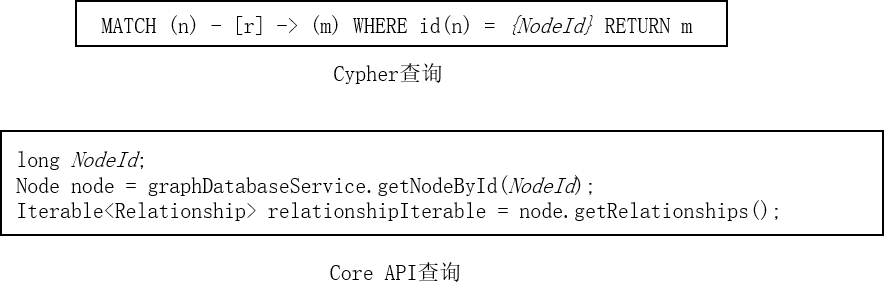
实验中随机选取1000个节点，进行“节点的所有关系”10000次查询。本文的缓存策略是在Neo4j kernel的数据访问层实现的，因此可以服务于上层的所有查询接口。缓冲区设置的大小。为了验证缓存策略的有效性，本小节的查询实验采用Cypher语言和core API两种实现，如图5-1所示：

图5-1 Cypher和Core API实现“节点的所有关系”查询

表5-1 页缓存查询耗时和对象缓存+页缓存查询耗时结果表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 页缓存 | 对象缓存+页缓存 | | | | |
| 20% | 40% | 60% | 80% | 100% | |
| Cypher查询 | ms |  |  |  |  |  | |
| Core API查询 |  |  |  |  |  |  | |

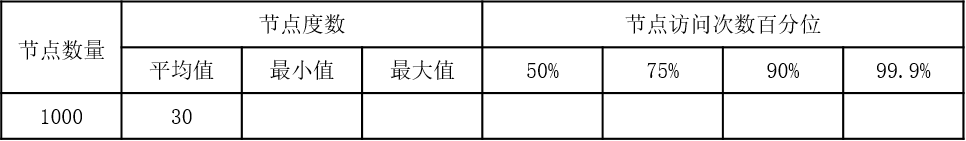
到底是占比好呢？还是数据好？

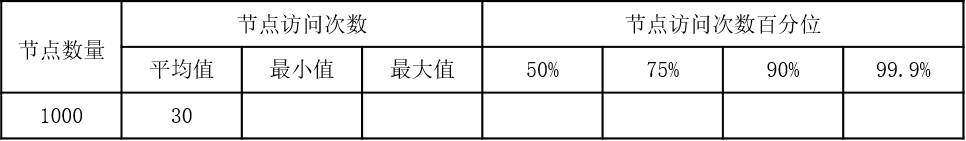
由表5-1可以看出，当缓存空间小，出现无法命中的状况，需要重新去pagecache加载缓存，Iterator应该可以构建的吧

展示成表5-2。

1. **改进的缓存策略和传统LRU缓存策略的对比实验与分析**

实验中将对象级缓冲区大小固定为1M，选取1000个节点进行查询实验，保证节点的所有关系数据超过缓冲区大小，当需要加载新的缓存数据时发生空闲空间不够的情况下，将发生缓存项的替换。改进的缓存策略的基本思想是，缓存中尽量保留缓存收益率高的缓存项，从而整体上提高缓存的效率。为了能够测试出节点收益率的变化对于实验结果的影响，需要保证对节点的访问次数较大。本文对这1000个节点的进行“节点的所有关系”模拟查询，其中对部分节点进行重点重复访问，例如IfcProduct的派生类型节点，模仿实际应用中的数据访问模式。选取的节点度数的统计信息如表5-2所示。节点的访问次数的统计信息如表5-3所示。

表5-2 节点度数统计信息

表5-3 节点访问次数统计信息

分别采用改进的缓存策略与传统的LRU缓存策略进行 “节点的所有关系”查询，查询采用 Cypher语言实现。对比两种策略的缓存命中次数如图5-2所示。

图5-2两种缓存策略的缓存命中数对比

## 特定路径查询算法实验

### 数据集

实验采用生成的模拟数据集进行试验。数据集共包含四类节点，用A、B、C、D代表。每类节点处于一层，四类节点形成路径长度为3的图。每类节点的数量按照路径深度的增加成倍增长，这主要是近似模拟场地IfcSite、楼IfcBuilding、楼层IfcBuildingStory、建筑构件（IfcBuildingElement的派生子类）。数据集的具体信息如表5-4所示。

表5-4 不同类型的节点数量

### 实验与分析

分别采用原始Cypher语句和基于BloomFilter的查询算法在上表中三个数据集上进行特定路径查询实验，实验的结果如表5-5所示。

表5-5 原始Cypher查询与基于BloomFilter的查询的平均用时对比

# 总结与展望

## 总结

当今世界，一场新的制造业竞争已然拉开序幕：美国力促高端制造业回归、德国倾力打造工业4.0。而作为制造业大国，中国则推出了"中国制造2025"行动计划。建筑行业作为制造业的重要组成部分，得到了广泛关注，作为建筑行业的行业数据的标准，BIM行业得到了广泛关注。BIM系统是一种全新的信息化管理系统，目前正越来越多应用于建筑行业中。然而，随着时代的发展，智慧城市概念的提出，BIM数据有了长足的增长，以往的存储和查询手段已经渐渐越来越不能应付数据的长足提升。所以为海量BIM数据提供一种高效快速的存储和查询手段是有必要和有意义的。

结合国内外BIM数据库研究的发展方向，可以看到，BIM数据库研究的目标是支持大数据量的存储和查询、支持更为灵活的查询方式。

基于此目标，本文选择了开源弹性搜索框架ElasticSearch并展开了研究，通过增加冗余的方式，满足用户对于属性数据的高效查询，避免了多表连接的问题。在试验中，着重解决了IFC文件解析后的数据到JSON数据的映射问题，和ElasticSearch的索引优化等问题，最终实现了IFC实体的属性数据的高效存储和查询。

其次，本文还深入研究了IFC数据标准，提出了IFC对象空间关系数据模型的存储设计。本文选择了开源的Neo4j图数据库进行IFC实体空间关系数据模型的存储，从而在属性值查询的基础上，为用户提供了另外一种维度的查询方式，即空间关系（或者成为路径）的查询方式。

除此之外，本文进一步关注了一种特定的路径查询，并针对这种查询提出了基于BloomFilter的解决方案，使得查询效率得到了提升，尤其是在空值查询时能够提前失败，给用户响应。

## 展望

本系统基于弹性搜索框架ElasticSearch和图数据库Neo4j，实现了IFC实体属性数据和空间关系数据的存储和查询。但是目前，两个子系统在查询模块是独立的，用户只能通过各自的web界面来访问数据库，这带来了一定的不变。未来将考虑提供统一的访问界面。

另外，本系统中Neo4j的节点只存储了IFC实体类型本身的自有属性，这也是因为本文将研究的重心放在了IFC实体的空间关系信息的存储和查询上。如果用户需要满足一定空间关系的节点的某些热点属性（即用户关注的属性）的话，需要根据Neo4j结果集中的ID到ElasticSearch中查询数据。未来，可以考虑给IFC实体节点数据增加冗余的设计，将部分热点属性数据集成到节点数据当中去，这将使得用户可以直接在Neo4j找到热点属性数据，而不必在Neo4j做完关系查询之后再到ElasticSearch中查询。

# 参考文献

[1] Jorgensen K A,Skauge J,Christiansson P, Use of IFC Model Servers:Aalborg University and Aarhus School of Architecture, 2008

[2] 张洋，基于BIM的建筑工程信息集成与管理研究，清华大学，2009

[3] Sciences N I O B National Building Information Modeling Standard Verion1-Part1:Overview,Principles,and Methodologies,2011-9-10 ed 2007

[4] Fu C;Aouad G;Lee A IFC model viewer to support nD model application[外文期刊] 2006(2)

[5] Mell P;Grance T,The NIST Definition of Clouding Computing: National Institute of Standards and Technology

[6] Faraj I;Alshawi M;Aouad G.An Industry foundation classes Web-based collaborative construction computer environment:WISPER 2000(1)

[7] 陆宁，马智亮，利用面向对象数据库与关系数据库管理IFC数据的比较，清华大学学报（自然科学版），2012

[8] Kang H;Lee G,Development of an Object-Relational IFC Server 2009

[9] 刘强，张建平，胡振中，基于键-值缓存的IFC模型Web应用技术，清华大学学报（自然科学版），2016

[10] 余芳强，张建平，刘强，赵文忠，基于云计算的半结构化BIM数据库研究，土木建筑工程信息技术，2013(6)

[11] 张建平，BIM技术的研究与应用，2011

[12] 周雅莉，胡珉，喻钢，基于IFC标准的隧道工程信息传递研究，土木建筑工程信息技术，2015(12)

[13] 佐佐木达也，NoSql数据库入门，2012

[14] 岳莉，基于Lucene的全文检索系统的研究与应用，西安电子科技大学，2010

[15] 徐财应，基于Lucene的搜索引擎技术的研究与改进[D].中国科学技术大学，2014

[16] 陈亚杰,王锋,邓辉,刘应波, ElasticSearch分布式搜索引擎在天文大数据检索中的应用研究,天文学报，2016(3).

[17] Rafa l K, Marek R. ElasticSearch可扩展的开源弹性搜索解决方案. 时金桥, 柳厅文, 徐菲, 等译.电子工业出版社, 2015: 25

[18] 龙慧芬，移动社交网络中的数据库应用[J].山西青年管理干部学院学报，2013,26(3):106-108

[19] 王余蓝，图形数据库Neo4j的内嵌式应用研究[J].现代电子技术，2013,35(22):36-38

[20] 王余蓝，图形数据库Neo4j与关系数据库的比较研究[J].现代电子技术，2012,35(20):77-79.

[21] Huang H,Dong Z. Research on architecture and query performance based on distributed graph database Neo4j[C]. Consumer Electronics,Communications and Networks(CECNet),2013 3rd International Conference on .IEEE,2013,533-536

[22] Robin Hecht，Stefan Jablonski. NoSql evaluation：A use case oriented survey [A]. Proceedings of 2011 International Conference on Cloud and Service Computing [C]. 2011.336-341

[23] Sacco G M. Inverted index and inverted list process for storing and retriving information, U.S.Patent 8,738,631[P].2014-5-27.

[24] See R, Karshoej J,Davis D. An Integrated Process for Delivering IFC Based Data Exchage[J]. 2012-09-16

[1] 张建平. BIM技术的研究与应用. 施工技术（资讯），2011，（02）

[2] 建筑产品BIM数据库的价值及发展前景. 李美华