

|  |
| --- |
| 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： | 2018年3月6日 |

**目录**

[摘要 IV](#_Toc508136065)

[Abstract V](#_Toc508136066)

[第1章 绪 论 1](#_Toc508136067)

[1.1 研究背景及意义 1](#_Toc508136068)

[1.2 国内外研究现状和热点 2](#_Toc508136069)

[1.2.1 BIM数据库研究现状 2](#_Toc508136070)

[1.2.2 图形数据库的发展 4](#_Toc508136071)

[1.3 研究内容和本文贡献 4](#_Toc508136072)

[1.4 组织结构 5](#_Toc508136073)

[第2章 相关技术基础 7](#_Toc508136074)

[2.1 IFC基础 7](#_Toc508136075)

[2.1.1 IFC标准 7](#_Toc508136076)

[2.1.2 IFC文件格式 9](#_Toc508136077)

[2.1.3 IFC数据模型 10](#_Toc508136078)

[2.2 图数据库技术 20](#_Toc508136079)

[2.2.1 图数据与图数据库 20](#_Toc508136080)

[2.2.2 Neo4j概述 21](#_Toc508136081)

[第3章 系统总体架构 24](#_Toc508136082)

[3.1 系统总体架构 24](#_Toc508136083)

[3.2 子模块设计与实现 25](#_Toc508136084)

[3.2.1 文件解析模块 25](#_Toc508136085)

[3.2.2 存储和索引模块 25](#_Toc508136086)

[3.2.3 查询访问模块 28](#_Toc508136087)

[第4章 基于Neo4j的BIM数据库关键技术研究 29](#_Toc508136088)

[4.1 缓存技术研究 29](#_Toc508136089)

[4.1.1 Neo4j页缓存架构 29](#_Toc508136090)

[4.1.2 对象缓存策略 31](#_Toc508136091)

[4.1.3 缓存预装载策略 35](#_Toc508136092)

[4.2 特定路径查询技术研究 36](#_Toc508136093)

[4.2.1 特定路径查询场景及问题 36](#_Toc508136094)

[4.2.2 特定路径查询算法 38](#_Toc508136095)

[第5章 系统实验 40](#_Toc508136096)

[5.1 实验环境 40](#_Toc508136097)

[5.2 缓存算法实验 40](#_Toc508136098)

[5.2.1 数据集 40](#_Toc508136099)

[5.2.2 实验与分析 40](#_Toc508136100)

[5.3 特定路径查询算法实验 42](#_Toc508136101)

[5.3.1 数据集 42](#_Toc508136102)

[5.3.2 实验与分析 42](#_Toc508136103)

[第6章 总结与展望 44](#_Toc508136104)

[6.1 总结 44](#_Toc508136105)

[6.2 展望 44](#_Toc508136106)

[参考文献 45](#_Toc508136107)

# 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。然而，随着时代的发展，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，主要问题在于无法解决IFC模型关系数据查询效率低下的问题。

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在图数据的查询、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Neo4j页缓存模型，提出了面向图数据的对象缓存策略，并在Noe4j内核中进行了实现。实验结果表明，增加对象缓存之后的Neo4j数据库有效地提高了关系数据的查询效率; 3. 本文提出了缓存预装载策略，给出了在Neo4j中的实现，预装载缓存策略可以有效地提高数据库系统在重启后运行初期的效率; 4. 本文研究了特定路径查询的问题，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比Cypher查询更高效，在空值查询时能够更快给用户响应。

**关键词**：建筑信息模型（BIM），IFC标准，Neo4j，页缓存，对象缓存，预装载，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. As a key basic data platform for the connection between building products and BIM technology, BIM database is of great value and good development prospects. It is of great significance to establish a BIM database. However, with the development of the times, the past relational BIM databases have been increasingly unable to meet the needs of the growing storage and management of BIM data. Object-oriented database or object relational database cannot be the main technology of BIM database because of its limitations such as imperfect and immature theory, and poor extensibility. In recent years, the NoSQL BIM database is booming. The main problem of it is that it cannot solve the problem of low query inefficiency of relational IFC data.

Based on the thorough research of the Neo4j graph database, this paper proposes a Neo4j based BIM database scheme. This paper maps IFC data model into Neo4j graph data model and uses the advantage of Noe4j in graph data query and management to solve the query efficiency of IFC relational data. Specific contributions are as follows:

1. This paper studies IFC file and data model, realizes the process of IFC file analysis, conversion, storage and index to Neo4j data model, and realizes BIM database based on Neo4j. 2. This paper studies the Neo4j database kernel source code, expounds the Neo4j page cache model, proposes object-oriented caching strategy for graph data, and implements it in Noe4j kernel. The experimental results show that the Neo4j database after adding the object-level cache effectively improves the query efficiency of relational data. 3. This paper presents a cache preloading strategy, and the realization of the preloading cache strategy in Neo4j which can effectively improve the efficiency of the initial operation after the restart of the database system. 4. This paper studies the problem of specific route query, proposes a specific path query algorithm based on BloomFilter, and designs and implements the service interface of this query based on the plug-in interface. The experimental results show that the proposed algorithm is more efficient than Cypher query and can respond to users faster in null query.

Keywords：BIM，IFC，Neo4j，Page Cache，Object Cache，PreLoaded Cache，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。如何设计出高效的BIM数据库迫在眉睫。

## 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。张洋博士[3]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。还有学者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[4,5,6]。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。徐锐博士[7]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。Faraj等人[8]开发的WISPER系统采用面向对象数据库ObjectStore来存储IFC数据。Ali等人[9]利用面向对象数据库EDM(express data manager)存储IFC数据。陆宁等人[10]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，即超过10万条记录时，处理效率更高。但是面向对象型数据库面临着理论不完善、可扩展性较差的问题，无法满足当下BIM数据库的发展需求。

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[11]。Kang等[12]采用对象关系数据库Cubrid存储IFC数据。但是受限于对象关系数据库尚不成熟，还没有被广泛应用在商业软件上，因此不适合应用到BIM数据库中。

随着BIM数据的长足增长，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[13,14,15]。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[14]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。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。刘强等人[15]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**

余芳强等人[16]基于开源的分布式列式数据库HBase[17]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**

Jia-Rui等人[18]基于开源的分布式文档式数据库MongoDB[19]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。

NoSQL类型的BIM数据库具有传统的关系数据库取法比拟的海量数据处理能力，但是受限于本身的存储模型，目前还不能很好地支持关系数据查询。许多学者依靠半结构化的存储来解决部分关系查询问题，依旧无法应对更复杂的关系数据查询。

### 图形数据库的发展

随着计算机软硬件和互联网应用的飞速发展，不同领域都积累了大量类型复杂化、结构多样化的应用数据，传统的关系型数据库无法提供理想的数据存储和管理方案。以常见的社交网络为例，涉及大量存在于人与人之间的关系数据，错综复杂的社交网络数据模型的存储在传统关系型数据库的存储模式下难以实现，而面对“好友的好友”这种多层复杂的关系查询问题传统关系型数据库的响应时间更长、处理效率十分低下。而图形数据库的出现为这种数据模型的存储和查询提供了解决方案[20]。图形数据库依托图论为理论基础，将数据组织为节点、关系的形式，存储在图结构中，在存储和管理复杂的数据模型以及模型中数据之间的内在关系方面有着传统关系型数据库无可比拟的优势。

Neo4j[21,22]作为图形数据库中的重要一员，近年来受到了各领域学者和工程人员的重视。Chad等人[23]对比了MySQL和Neo4j的优势和劣势，指出了MySQL在处理结构化数据时有优势，而Neo4j则更擅长处理关系复杂的数据。王余蓝[24]则从成熟度、安全性、可扩展性3个方面对关系型数据库和Neo4j图数据库进行了对比分析和实例验证，指出了Neo4j图数据库在成熟度、安全性等方面存在劣势，在处理关联关系复杂、关系动态变化等社交性数据则存在优势。Neo4j已经在包括社交网络、智能电网、地理信息、大数据组织、元数据管理等领域的数据解决方案中被广泛研究和应用[25,26,27,28,29,30]。

## 研究内容和本文贡献

综合当前国内外BIM数据库方面的研究可知，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。

为了解决IFC模型的关系数据查询效率低下的问题，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在图数据的查询、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Neo4j页缓存模型，在此基础上提出了面向图数据的对象缓存策略，并在Noe4j内核中进行了实现。实验结果表明，增加对象缓存之后的Neo4j数据库有效地提高了关系数据的查询效率。
3. 本文提出了预装载缓存策略，给出了在Neo4j中的实现，预装载缓存策略可以有效地提高数据库系统在重启后运行初期的效率。
4. 本文研究了特定路径查询的问题，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比Cypher查询更高效，在空值查询时能够更快给用户响应。

## 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，阐述了Neo4j的页缓存模型，并提出了面向图数据对象的对象缓存策略、预装载策略和面向特定路径查询的算法以及实现。

**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模型结构图。



图2-1 IFC模型结构图[31]

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 IfcDoor的EXPRESS定义

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。

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 IFC文件



图2-3中的解释是：

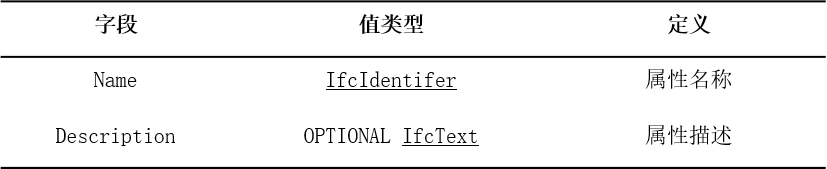
1. 行1是文件开始语句，表示ISO发布的STEP标准。对应文件最后一行（第92行）的END-ISO-10303-21。
2. 行2-6，表示文件信息，以HEADER关键字开头，以ENDSEC关键字结束。包括文件描述、文件名称、文件使用的IFC标准。
3. 行7-92是数据段。以DATA关键字开头，以ENDSEC关键字结尾。中间的每一行表示一个IFC实体的实例数据。#后面的数字是一个唯一的编号，可以不连续，而且不一定是增序，标识一个IFC实体的实例。等号右端首先是IFC实体的类型名，括号内是按照EXPRESS定义的包括继承属性的所有属性的属性值，值的类型可能是定义类型、选择类型、枚举类型。

### IFC数据模型

#### 属性数据模型

1. **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2-1分别给出了抽象类IfcProperty的正向字段定义。

表2-1 IfcProperty定义

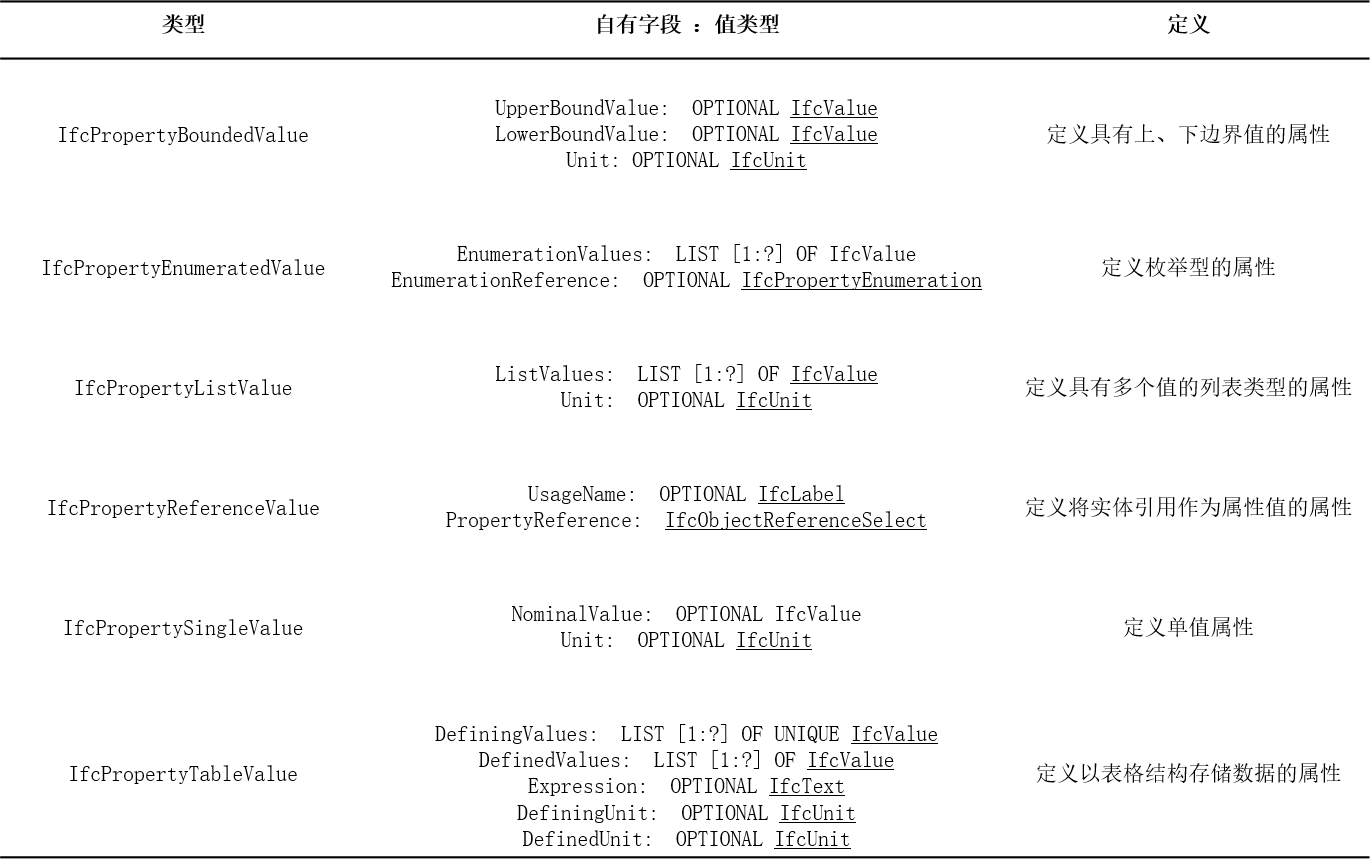
根据属性的值类型的不同，IfcProperty具有不同类型的派生类。图2-4给出了IfcProperty的派生子类图，图中的斜体内容表示此类是抽象类。

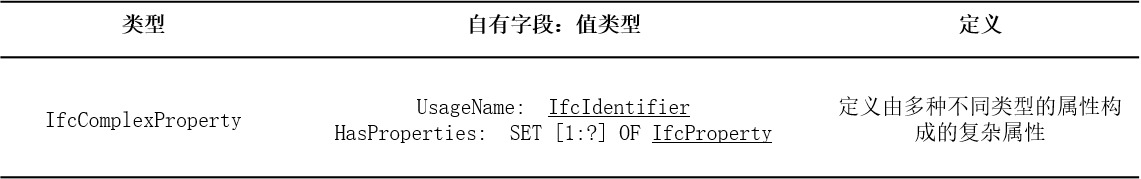


图2-4 Property派生子类图

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2-2所示。

表2-2 IfcSimpleProperty派生子类的自有字段及定义

表2-3 IfcComplexProperty的自有字段及定义

1. **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2-5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2-6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2-4所示。其中，以IfcDoorLiningProperties为例，其自有字段定义如表2-5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2-6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2-7所示。

* 1. 自定义属性集

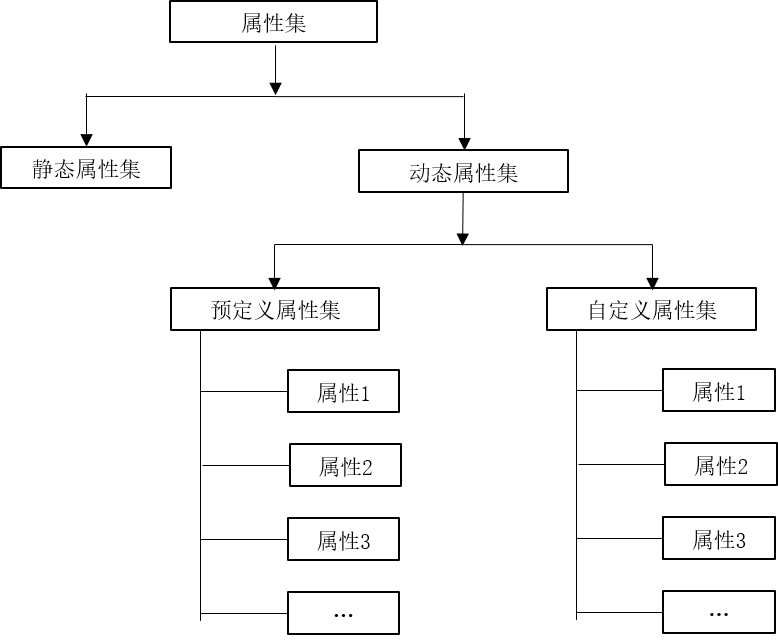
可以由用户自定义属性集名称和一组属性类型集合，它并不限于描述某一类IFC实体，具有更强大的表达能力。

图2-5 属性集分类

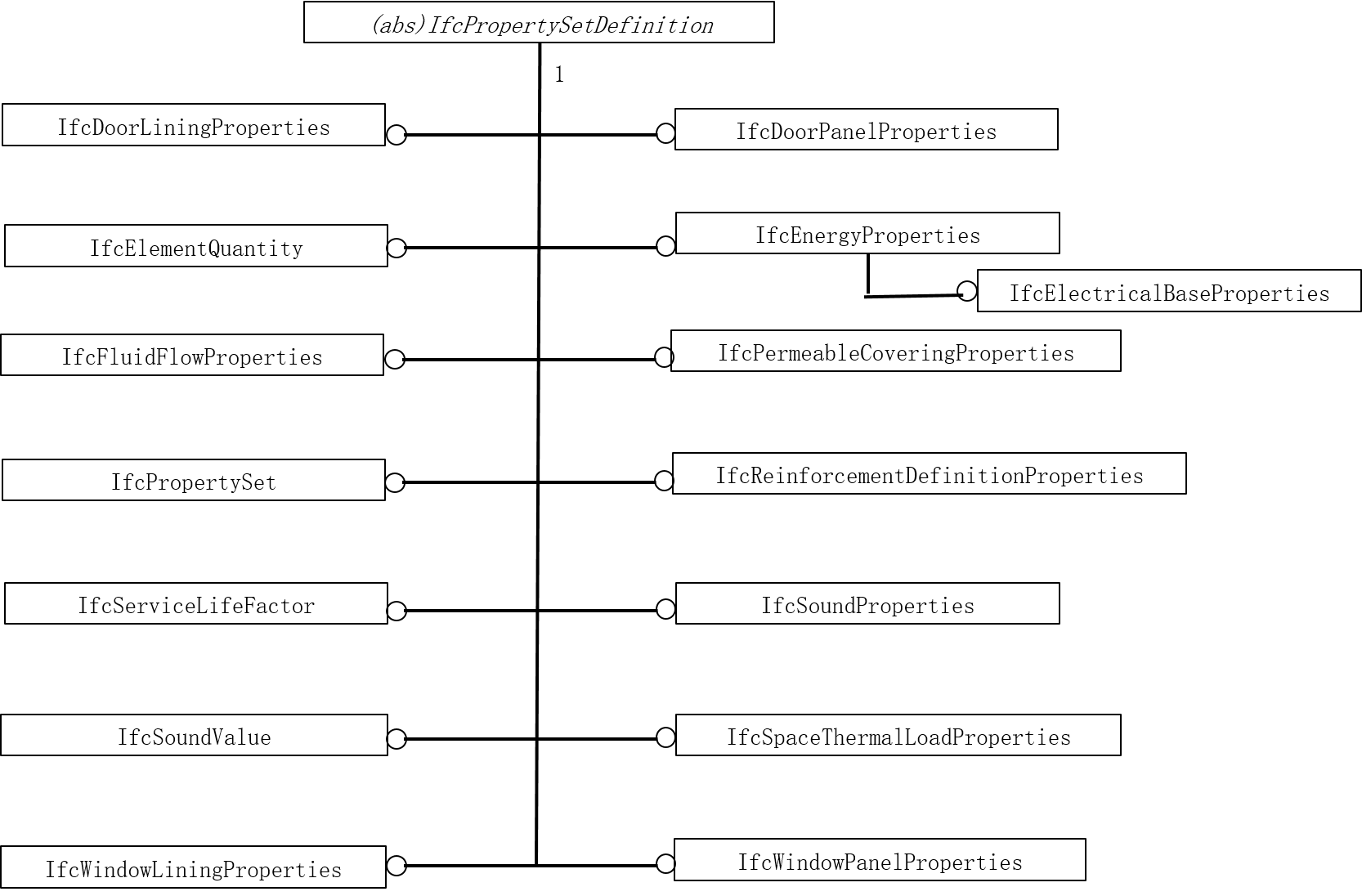
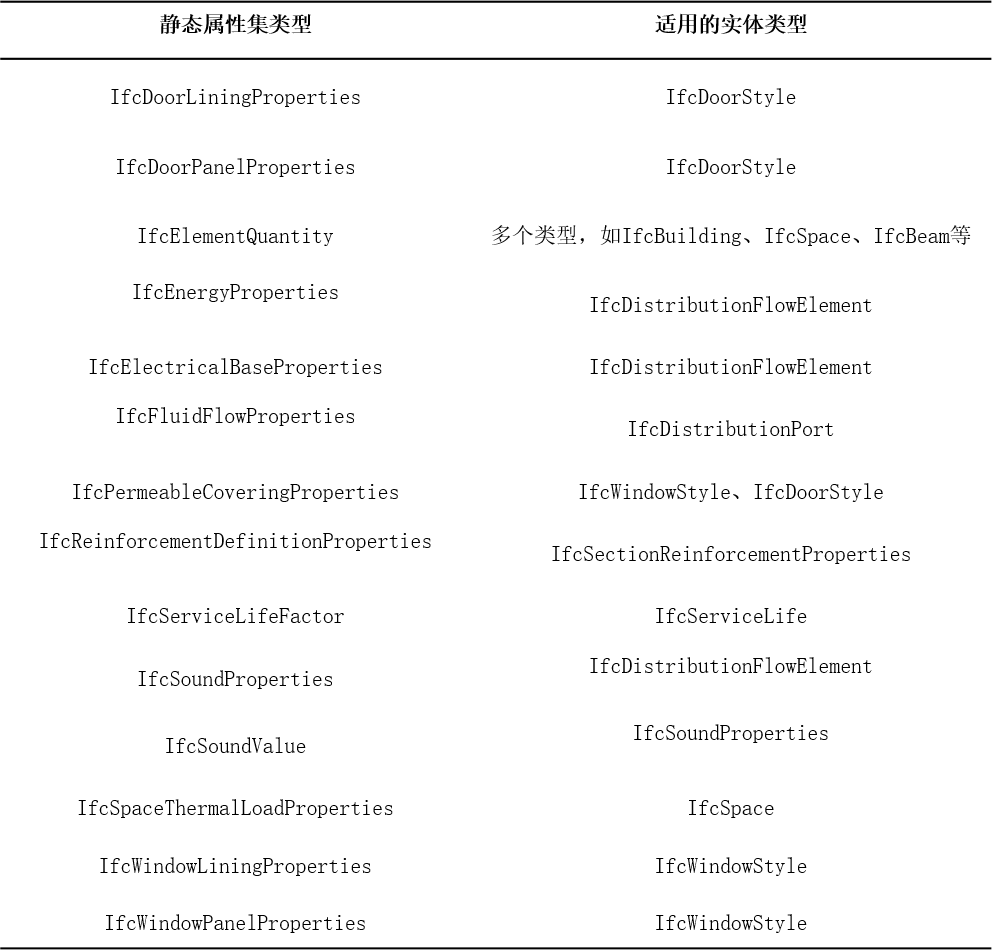
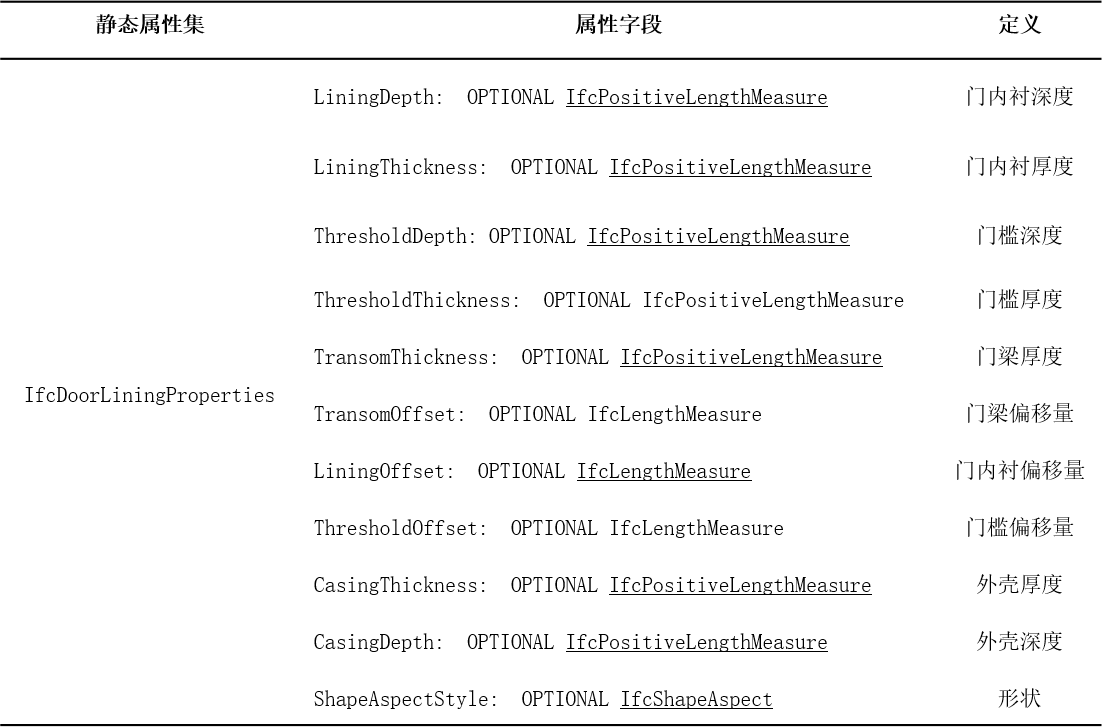
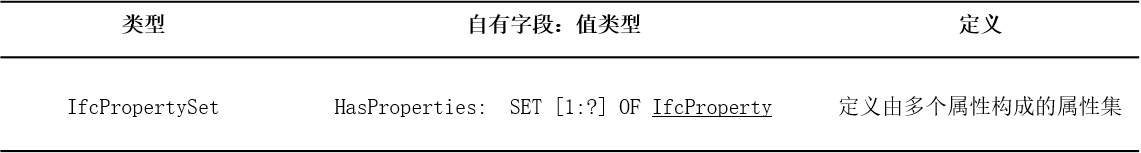
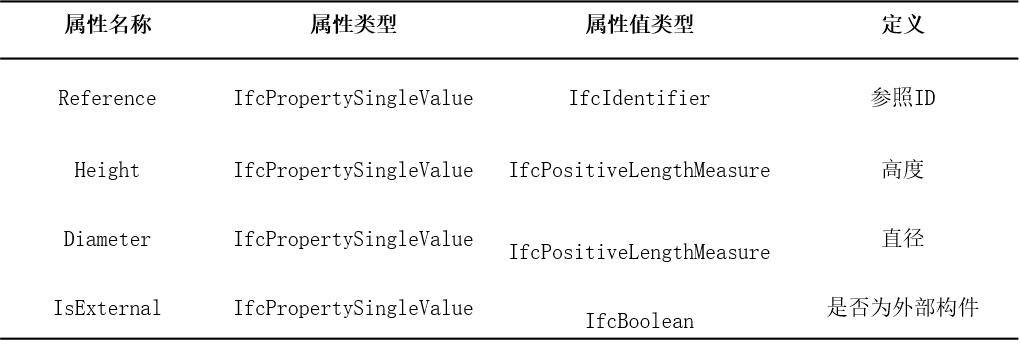


图2-6 IfcPropertySetDefinition派生子类图

表2-4 静态属性集类型及适用的实体类型

表2-5 IfcDoorLiningProperties自有字段定义

表2-6 IfcPropertySet自有字段及定义

表2-7 预定义属性集Pset\_RailingCommon的属性定义

1. **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：

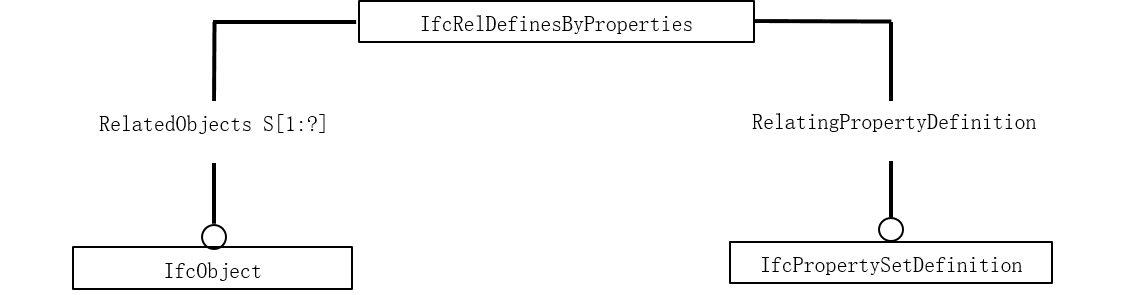
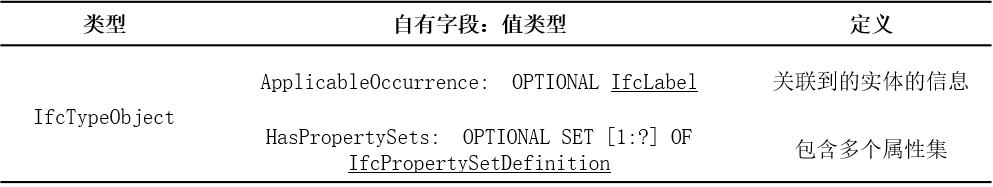
1. 通过IfcRelDefinesByProperties关系实体将IfcPropertySetDefinition对象与IfcObject对象关联。多个具有相同属性集的IFC对象可以通过同一个IfcRelDefinesByProperties关系实体关联到一个属性集，这样可以实现属性集的重复使用。同时一个IFC对象可以通过多个IfcRelDefinesByProperties来关联到不同的属性集，这样就可以借助多个属性集来表达丰富的属性信息。IfcRelDefinesByProperties关联机制如图2-7所示。

图2-7 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2-8所示。

表2-8 IfcTypeObject关联属性集

1. **属性数据特点**

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。

#### 空间关系模型

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。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，IfcRelConnects和IfcRelDecomposes及其派生类涉及到Ifc实体的空间关系。 下面的例子表示了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。建筑构件之间的空间关系是使用关系对象来间接表示的。

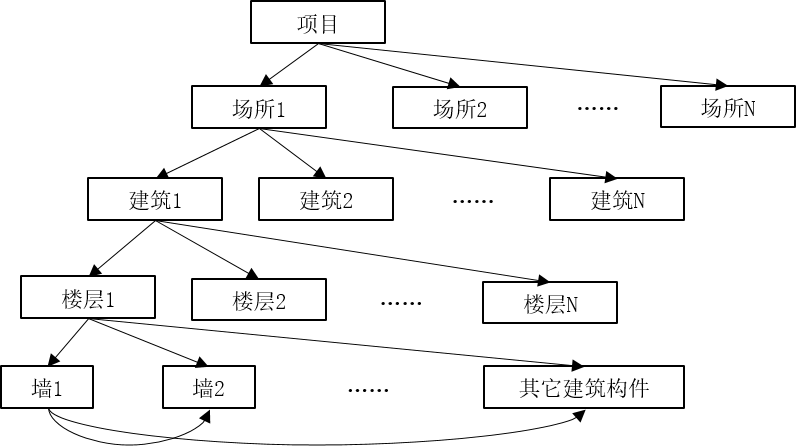
在现实世界中，建筑元素的关系可以是：每个建筑包含很多楼层，每个楼层包含很多建筑构件，如门、窗、墙、梁、板、洞等等。而建筑构件之间也有多元化的关系，比如某两面墙相连接、某面墙上有四扇窗等等。图5-1展示了现实世界中的建筑元素的空间关系图.

图2-8现实世界中的建筑元素的空间关系图

## 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、高可扩展的图数据库。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。

下面分别从基本概念、查询语言和索引、底层存储结构四个方面介绍Neo4j数据库。

1. **基本概念：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。图就是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. **查询语言：Cypher**

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 Cypher查询语句实例

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，并删除一些旧的功能。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. **索引：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，底层基于Lucene实现。在Neo4j 2.0版本之前的索引被称为Legacy Index。Legacy Index能够支持全文检索，相比之下，Schema Index没有全文检索的能力。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”，可以借助Cypher语句中的START子句的调用来实现。

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 创建Schema Index的Cypher语句

其中的Label和Property分别对应具体的标签和属性。

1. **底层存储结构**

Neo4j图数据库将不同类型的数据信息存储在不同的文件里，文件以数组作为核心存储结构，存储相同结构的记录。文件中每条记录都会分配一个唯一的ID，在存储时以该ID作为数组的下标。查找某条记录时，可以通过记录大小、ID值实现快速定位。以Neo4j community 3.0.6源码中的数据格式为参考，分别介绍节点、关系、属性记录的数据结构。

1. 节点

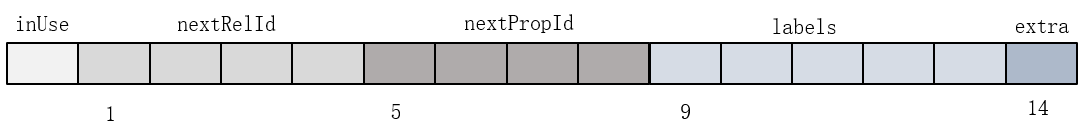
节点信息存储在文件neostore.nodestore.db中。节点记录的结构如图2-12所示。

图2-12 Neo4j的节点记录结构

节点记录长度为15字节。第一个字节为标识位，表示该记录是否正在被用于存储节点数据。不被使用时，可以被Neo4j回收用于存储新的节点。nextRelId占用4字节，表示该节点关联的第一个关系记录的Id。nextPropId占用4字节，表示该节点的第一个属性记录的Id。labels占用5个字节，表示该节点的label的id。extra占用一个字节，表示一些附加信息，如是否是dense节点（节点度数超过系统设定的阈值）。

1. 关系

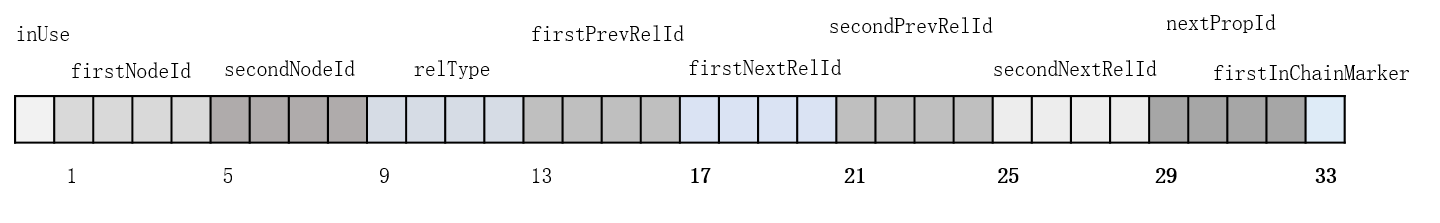
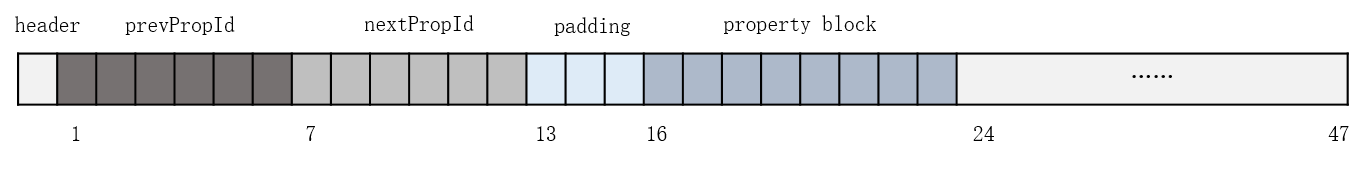
关系信息存储在文件neostore.relationshipstore.db中。关系记录的结构如图2-13所示。

图2-13 Neo4j的关系记录结构

关系记录长度为34字节，inUse是“是否正在被使用”的标志位，接下来的字节依次表示起始节点Id、结束节点Id、关系类型Id、起始节点的上一个关系Id和下一个关系Id、结束节点的上一个关系Id和下一个关系Id、属性Id。firstInChainMarker表示是否是关系链中的首个关系。所有的关系组成一个双向链表，通过Id指针可以迅速地遍历链表中的所有关系和节点。

1. 属性

属性信息存储在文件neostore.propertystore.db中。属性记录的结构如图2-14所示。

图2-14 Neo4j的属性记录结构

属性记录长度为48字节。header包含inUse标志位、属性块计数等信息。prevPropId、nextPropId表示上一个和下一个属性记录的Id，padding作为对齐字节，在解析时直接跳过。Property Block表示一个属性块，占8字节，包含属性名和属性值的指针，指向属性类型文件和属性索引文件中的记录。一条属性记录中最多可以有4个属性块。多个属性记录通过指针的方式形成链表，可以表示节点的所有属性信息。

# 系统总体架构

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在管理图数据等方面的优势，实现IFC数据的存储和管理，并通过增加对象缓存层和缓存预装载策略来进一步提升IFC关系数据的查询效率。同时本文提出了特定路径查询的算法，通过插件服务接口扩展了Neo4j的接口服务。

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。其中关键算法包括对象缓存策略、缓存预装载策略、基于BloomFilter的特定路径查询算法在第四章进行详细介绍。

## 系统总体架构

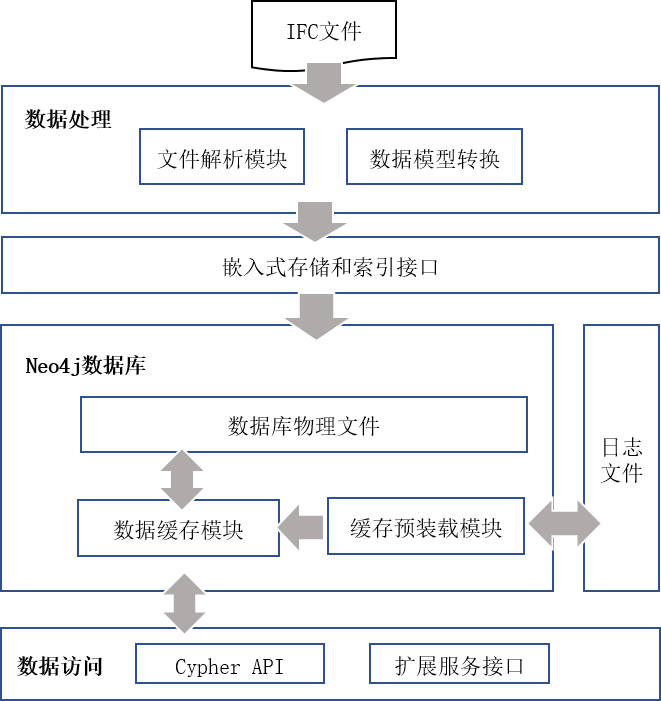
系统总体架构如图3-1，主要包括数据处理、Neo4j数据库、数据访问三个部分。

图3-1 系统总体架构

## 数据处理

系统以IFC文件作为数据源。IFC文件中存储的IFC数据模型需要被解析并转化为图数据模型，以便存储和索引到Neo4j数据库中。

### 文件解析模块

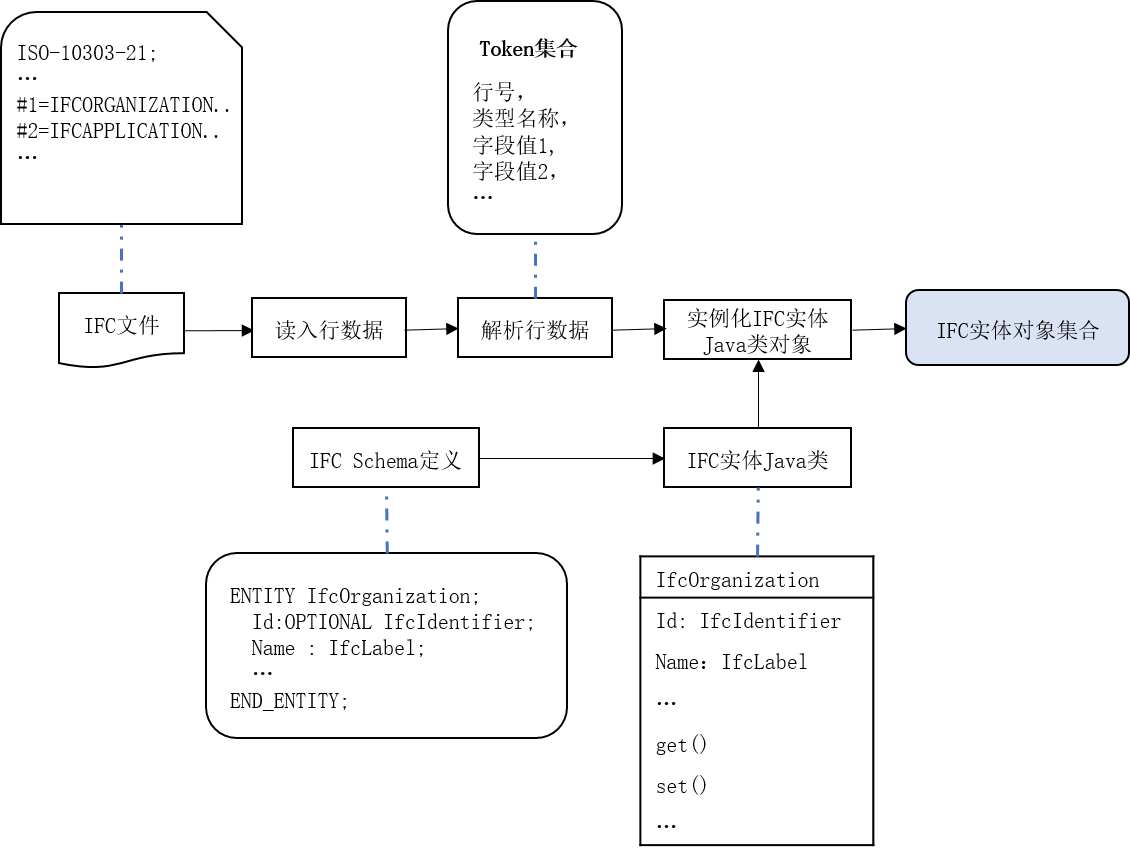
IFC文件解析模块的主要目的是解析IFC文件，得到IFC实体的Java类对象的集合。解析的具体过程是：逐行读入IFC文件的每条数据字符串，解析获得token集合，包括行号、类型名称、多个字段值等。行号是IFC对象的ID，类型名称是IFC对象的类型，字段值对应了IFC Schema中对IFC实体类型的Express定义。基于IFC schema的Express定义语言，构建IFC实体类型、定义类型、枚举类型、选择类型对应的Java类。然后结合Token集合中的IFC实体的数据实例化相应的IFC类对象。所有行数据解析完毕，最终得到IFC实体对象集合。解析流程图如图3-2所示。

图3-2 IFC文件解析流程

### 数据模型转换

数据模型转换的主要目的是将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，为数据存储和索引做准备。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数据按照不同的方式转换为Neo4j中的节点、关系、属性等，以采用批量处理接口来存储和索引。

为了描述数据模型转换的策略，定义M来表示IFC数据模型，其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*。

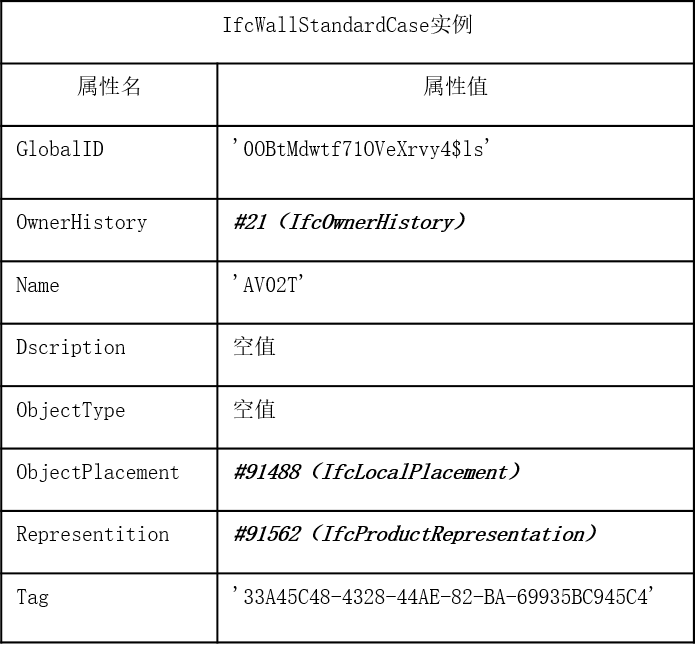
例如，图3-4是IfcWall对象实例的数据，图3-5对应在Neo4j中创建的节点和关系。

图3-4 IfcWallStandardCase实例信息

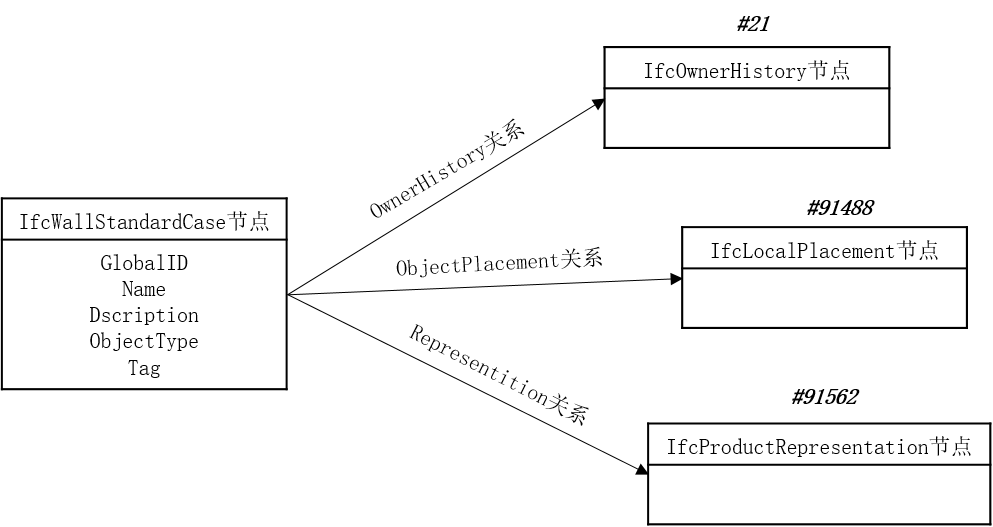


图3-5 IfcWallStandardCase实例对应在Neo4j中创建的节点和关系

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*。

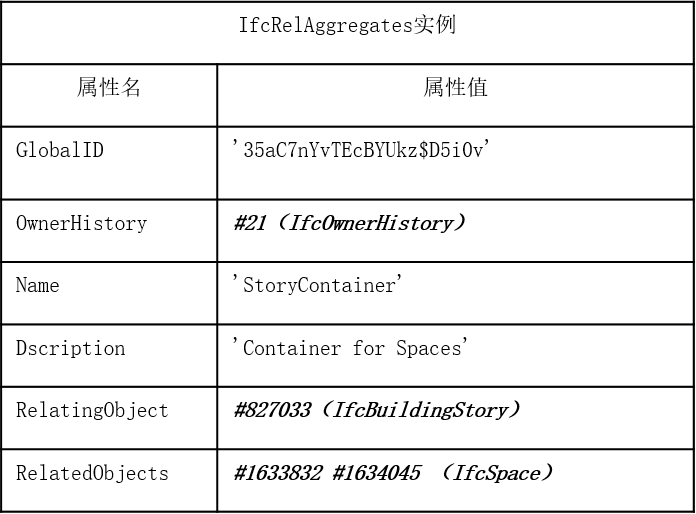
例如，图3-6是IfcRelAggregates的数据，图3-7对应在Neo4j中创建的节点和关系。

图3-6 IfcRelAggregates实例信息

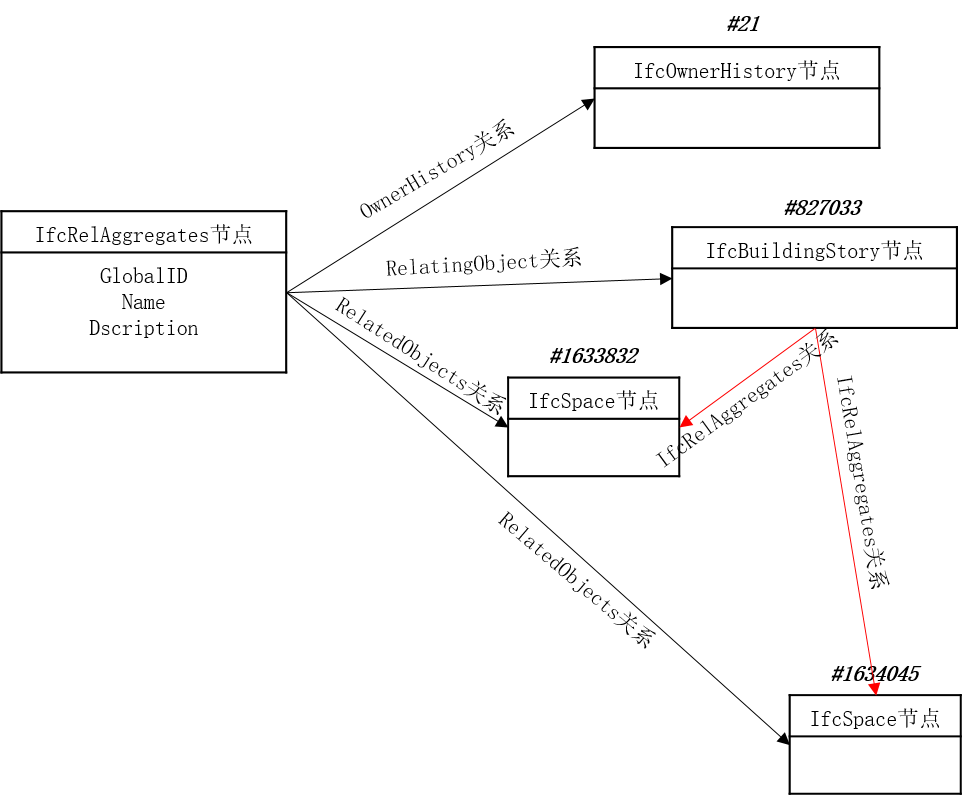
图3-7 IfcRelAggregates实例对应在Neo4j中创建的节点和关系

图3-7中的红色箭头代表将IfcRelAggregates对象中的“RelatingObject”字段引用的IfcBuildingStory对象与“RelatedObjects”字段表示的对象集合中的两个IfcSpace对象建立直接关联的关系。在查询IfcBuilding包含的建筑构件时，可以通过直接关联的关系进行查找，而不必再通过IfcRelAggregates对象进行两层路径查找，这样可以有效地提高关系查询效率。

## Neo4j数据库

Neo4j数据库部分是本文研究工作的重点。原有Neo4j系统中，事务通过页缓存访问数据库数据。页缓存作为数据库物理文件在内存中的映射，具有与物理文件相同的数据结构，在关系查询中需要重复定位数据、解析数据。为了减少重复计算，在页缓存层之上增加对象缓存层。对象缓存层面向节点，通过缓存节点的所有关系数据，在对于节点的关系数据的查询请求到来时直接返回结果，而不必访问文件的原始记录进行遍历和解析。对象缓存层采用一种基于改进的缓存策略，旨在内存空间有限的情况下提高缓存的命中次数。本文将在第四章详细介绍Neo4j的页缓存架构和对象缓存策略。

缓存预装载模块实现了系统重启之后缓存空间的初始化。通过读取本地的日志文件，计算出预装载的节点数据集。日志文件是节点关系查询的历史记录。预装载的节点数据集体现了历史访问的热点数据，并基于一定的策略进行选择。本文将在第四章详细介绍缓存预装载策略的算法和实现。

## 数据访问

数据访问层为用户提供了访问数据库的界面。Neo4j提供了访问数据库的web交互界面，可以接受用户输入的Cypher语句作为数据访问请求，并展示请求结果。同时，web界面可以采用REST API方式访问数据库数据。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。

# 基于Neo4j的BIM数据库关键技术研究

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的页缓存模型之后，本文阐述了页缓存模型的局限性，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对象缓存策略，这种缓存策略基于缓存收益率来替换缓存项，对于其它面向图数据的缓存算法具有借鉴意义。同时，本文实现了基于日志统计的预装载缓存策略，加速系统重启之后运行初期的查询效率。另外，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。接下来，将对以上所述的关键技术点进行详细介绍。

## 缓存技术研究

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中使用缓存层来加速对磁盘数据的访问。

### Neo4j页缓存架构

#### 页缓存架构概述

Neo4j数据库采用页缓存-Page Cache，缓存架构如图4-1所示。所有的上层数据访问接口包括Cypher API、Graph API、Traversal API对数据的操作转化为一系列事务操作。页缓存层则为上层事务管理模块提供数据访问接口。页缓存区存放的是在磁盘上的数据库数据记录文件分别被读入到内存中形成的映射文件。页缓存层的底层I/O实现依靠操作系统的内存映射技术，内存映射技术将磁盘上的物理文件与进程中逻辑地址空间中一块大小相同的区域一一对应，当需要访问磁盘上的数据时，不同于传统的I/O模式需要经过从磁盘拷贝到操作系统内核空间和从操作系统内核空间拷贝到用户空间这两个数据拷贝过程，而是只进行一次从磁盘到用户空间的拷贝，I/O效率更高。

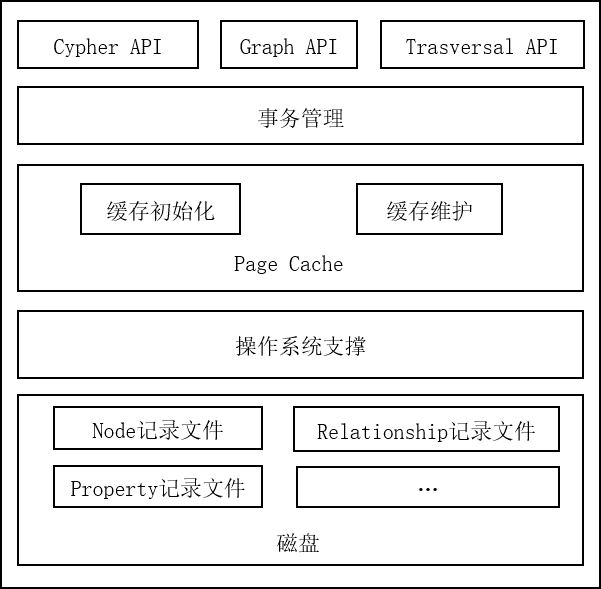


图 4-1 neo4j缓存架构

Neo4j数据库实例启动过程中，首先加载系统配置文件，根据配置参数来分配缓冲区相关数据结构的大小，包括数据缓冲区大小、缓存页大小、空闲页面数阈值等，并建立数据库记录文件的内存映射。数据库实例运行中，缓冲区存放了要访问的大量数据页。缓冲区维护了一个链表来管理数据页和空闲页。当缓冲区的空闲页面数小于阈值同时有新的数据页需要加入时，由于页缓存的文件映射依靠操作系统的内存映射技术实现，因此页面的置换也交给操作系统来实现，也即是说，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。

Neo4j页缓冲区主要包括以下数据结构，如图4-2所示：

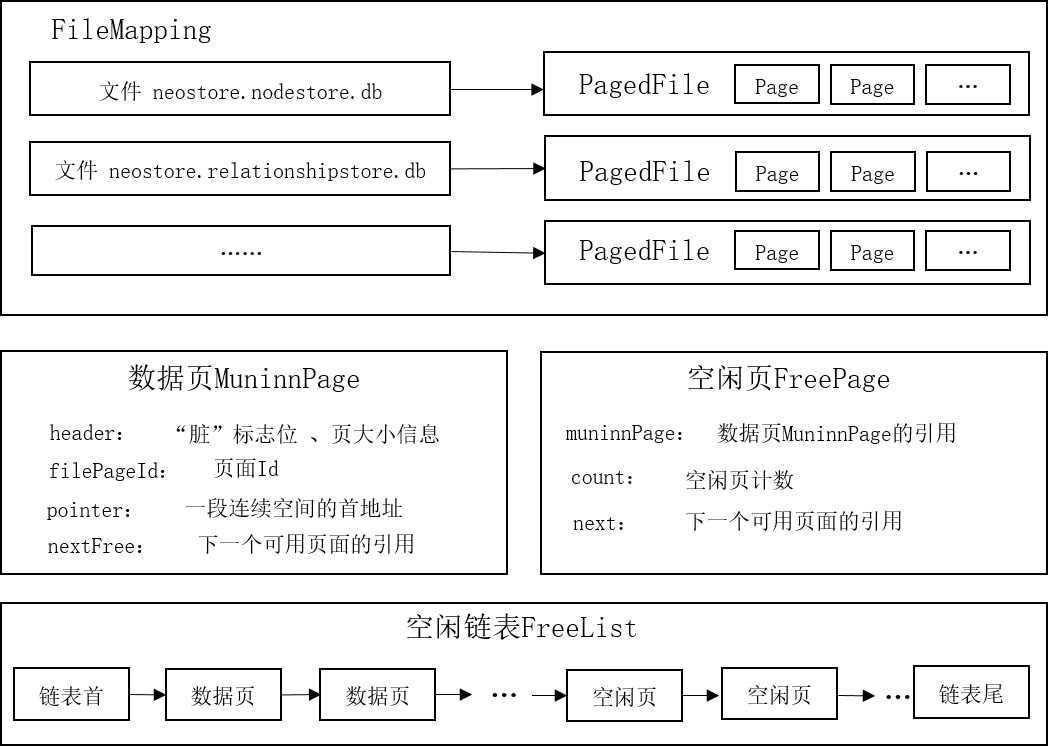
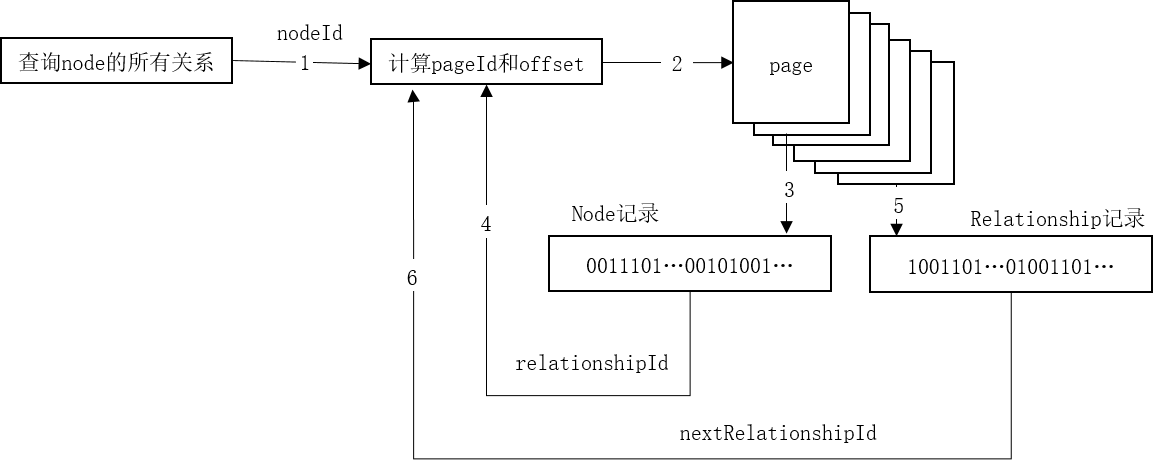
1. 文件映射FileMapping：保存磁盘文件到内存文件PagedFile的映射，PagedFile相当于磁盘文件在内存中的镜像。文件访问接口通过访问PagedFile来获取数据。
2. 数据页描述符MuninnPage：包含了页的状态信息、页Id等，并有一个指向一段连续空间的首地址，缓存的数据就在这段连续的空间上。
3. 空闲页描述符FreePage：是MuninnPage的封装，MuninnPage被置换出后清空数据，回收为FreePage，并加入空闲链表FreeList以供再次加载数据。
4. 空闲链表FreeList：在缓冲区初始化时建立缓冲区空闲管理链表FreeList。FreeList包含两种对象，包含MuninnPage和FreePage。初始状态中，MuninnPage在链首，FreePage在链尾。加载新的数据时，从链表头获取页面，向其指向的地址空间加载数据。FreeList逐渐变短，直到FreePage的数量低于阈值，将从缓冲区中按照clock算法选择MuninnPage的内容清除，同时封装为FreePage加入到FreeList中，实现FreeList的扩展。

图4-2 页缓冲区主要数据结构

页缓存区的页地址空间作为数据库物理文件在内存中的映射单元，具有与物理文件相同的数据结构，因此在关系查询中需要一系列的定位数据、解析数据的处理操作。当查找某个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。整个过程如图4-3所示。

图4-3 查询node的所有关系的过程

#### 页缓存缺点分析

由于页缓存直接将文件内容缓存到内存堆中，页缓存分配的内存越大，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的配置文件设定了页缓存空间的最小值，在缓存区初始化时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，在页缓存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。

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，本文提出了在页缓存层之上增加对象缓存层，并通过一定的缓存替换策略提高缓存区的命中次数。由于对象缓存层管理的缓存项是对象的关系数据，在查询时可以直接返回命中的缓存项数据，而不需要进行页缓存的计算和解析过程，使得查询响应时间进一步降低。

### 对象缓存策略

#### 缓存策略概述

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[32]（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。

#### 对象缓存策略设计

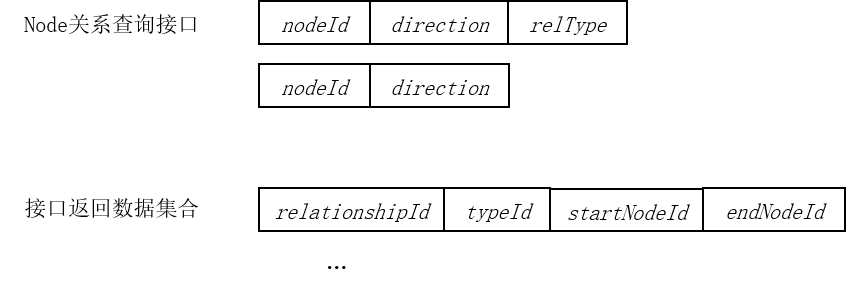
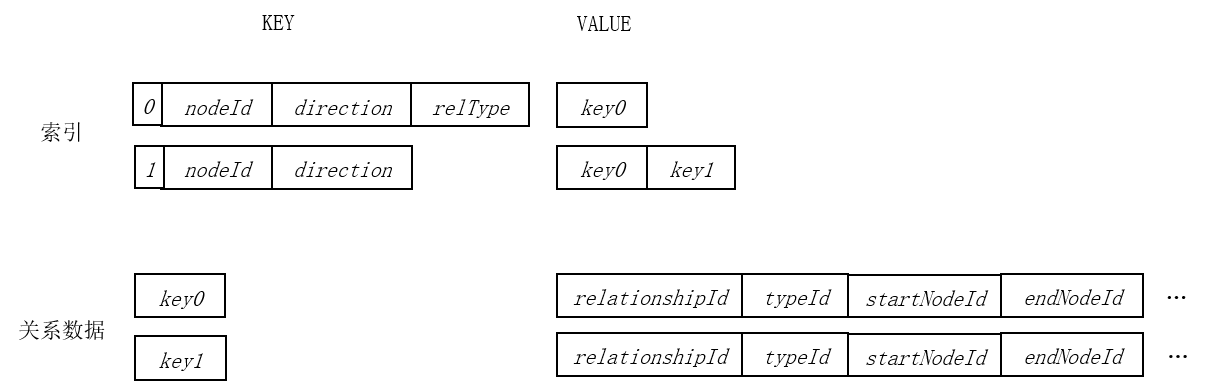
LRU缓存替换策略关注的是缓存数据项地位一致的缓存管理，而在Neo4j中节点度数不同，一个节点可能没有关系，也可能有成千上万个关系，节点对应的关系数据量随节点度数增加，不同节点缓存项的内存开销与节点度数成正比。研究neo4j内核中对于节点的关系数据访问接口，其中涉及到的接口参数和关系数据结构如图4-4所示。查询接口分为对特定direction（可以是INCOMING、OUTGOING、BOTH方向）和特定关系类型relType的关系数据查询。接口返回的Node的每项关系数据是固定格式和大小的。

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，相较于传统LRU替换策略，基于节点收益率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。相较于页缓存而言，在关系查询中，可以省去在页面中定位数据、解析数据的重复计算，降低了系统的计算开销，加快了对查询请求的响应。另外，本文的对象缓存策略是面向读操作的缓存，缓存数据在数据库中并不会被更新、删除等，所以并不涉及缓存数据的一致性维护等问题。

#### 对象缓存策略实现

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，包括节点度数nDegree、访问计数nCount、收益率nRatio、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，并更新收益率。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：

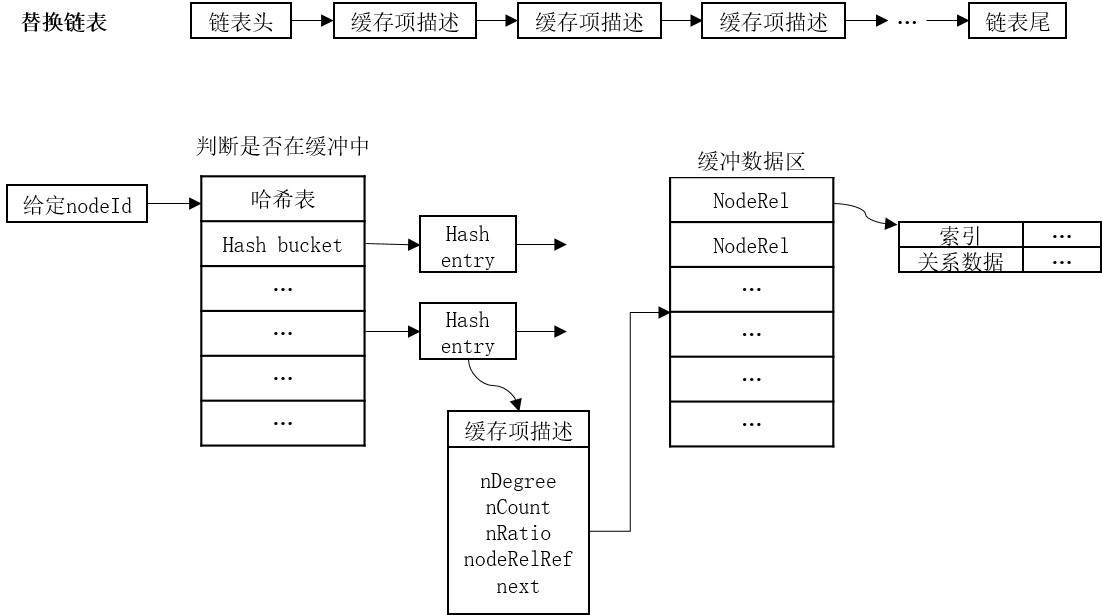


图4-6 对象缓存区结构示意图

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，可以引入一个替换周期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4-7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缓冲区，如果在对象缓冲区命中，直接返回已缓存的关系数据。
2. 如果对象缓冲区没有命中，则向下访问页缓存。
3. 根据页缓存数据构建对象缓存项，调用对象缓存算法加载入对象缓存区。
4. 返回数据结果。

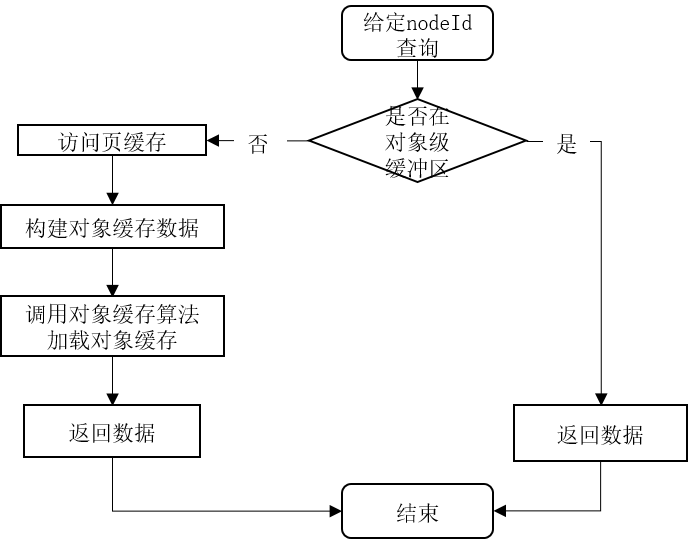


图4-7 缓存访问流程

### 缓存预装载策略

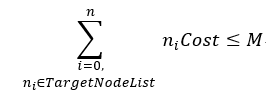
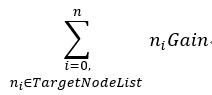
缓存预装载技术是指在数据访问之前将数据加载进入缓存，它可以在系统运行初期有效地利用缓存空间，在早期查询请求到来前准备好数据，有效避免从磁盘加载数据，提高系统运行效率。由于缓存区大小有限，缓存预装载策略需要预测哪些数据会被用户频繁访问。预测的准确率越高，缓存的命中率越高，对系统的查询性能的提升越明显。而面向图数据的缓存预装载策略同时需要考虑到缓存项的内存占用大小不同。单一地将访问次数高的数据留在缓存区中，并不能高效地利用缓存区的空间。本文接下来的小节将介绍面向图数据的缓存预装载策略的设计与实现。

#### 缓存预装载策略设计

本文的缓存预装载策略是基于用户的访问日志的，当用户在历史一段时间内的频繁访问某些数据，将来访问这些数据的可能性较大。同时考虑到节点关系数据占用内存大小的差异，利用前文所提出的缓存收益、缓存开销等概念，在历史访问数据中找到缓存收益总和最大的节点集合作为要加载的目标节点集。

求解目标节点集的问题定义如下：

在图G中，针对节点集合N定义缓存收益、缓存开销、缓存收益率，求解目标节点集 ，对于给定的缓存空间大小M，使得节点集的内存占用总和



且总收益最大。

求解目标节点集TargetNodeList的问题是一个0-1背包问题。动态规划算法可以找出问题的最优解，其时间复杂度为线性，而空间复杂度达到了，考虑到实际应用中数据规模可能较大如千万级别，M可以达到上百MB甚至更多，导致求解时的内存开销较大，因此本文选择贪心算法得到背包问题的近似解，其空间复杂度为，对单机内存的要求大大减少，时间复杂度为，满足了离线计算的耗时要求。

#### 缓存预装载策略实现

预装载策略的实现是基于用户查询日志的。在Neo4j的节点关系数据查询接口加入日志记录功能，对于每一次针对节点的关系查询，记录日志（节点Id，访问时间），离线统计模块可以计算出某段时间每个节点的访问次数，作为节点的缓存收益。通过嵌入式数据库接口Graph API访问数据库存储节点的信息，获得节点度数作为缓存开销。由此根据贪心算法求解目标节点集，其伪代码如图4-6所示。针对目标节点集，再次通过Graph API访问数据库，获取节点的所有关系数据，构建如图4-5的缓存数据项，并序列化到缓存物理文件中，以便系统再次启动时直接读取缓存文件内容加载入缓存区，避免在启动过程中通过访问数据库底层存储来构建缓存项，加速系统启动过程。预装载策略实现流程如图4-7所示。

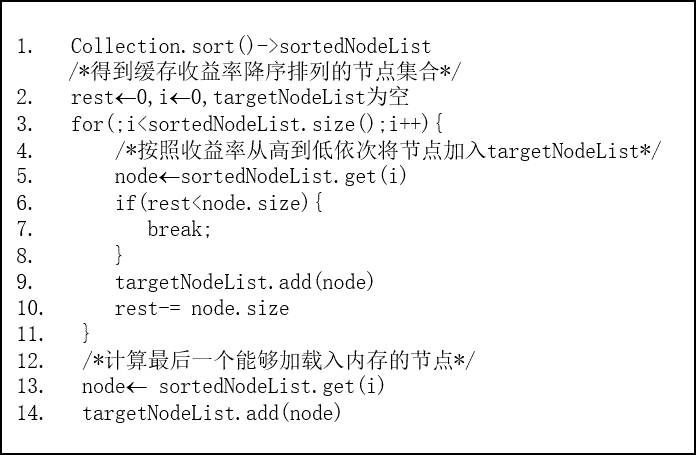
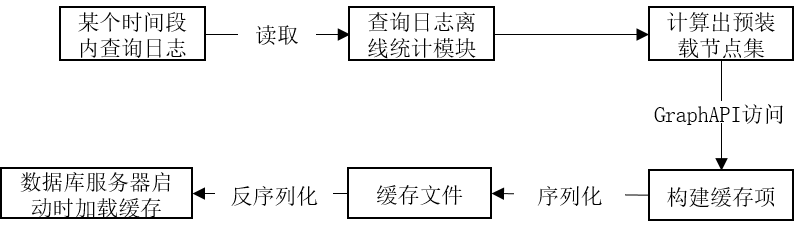
图4-6 求解预装载节点集

图4-7 预装载策略实现流程

## 特定路径查询技术研究

本小节将针对一种特定的路径查询问题和算法进行研究。

### 特定路径查询场景及问题

特定路径查询的场景是，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2-7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语句的表达式如图4-8，Cypher查询语句的完整的执行计划如图4-9。

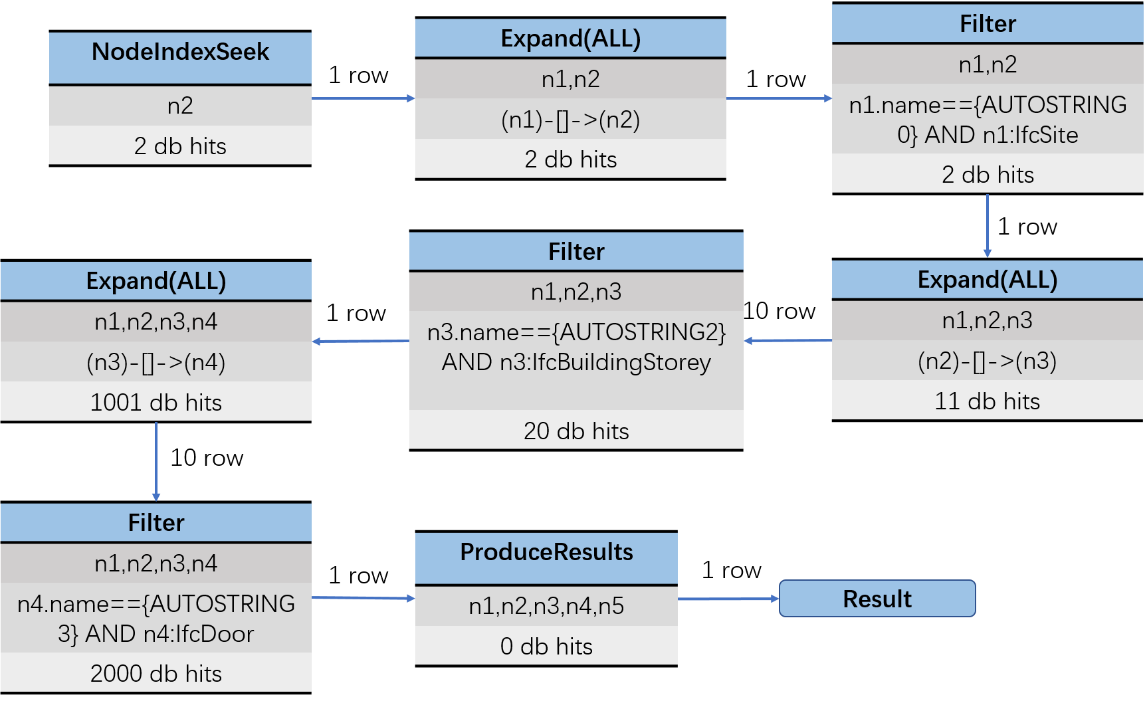
图4-8 特定路径查询Cypher语句

图4-9 特定路径查询执行计划

1. 执行NodeIndexSeek。访问索引“index on :IfcBuilding(name)”，返回节点*n2*。
2. 执行Expand(ALL)，访问节点*n2*的关系链表，解析出关联的节点。
3. 执行Filter。过滤步骤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。直到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。

查询过程只有在整个执行计划的第一步使用了index，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，并对这些节点进行过滤。其具体问题总结如下：

1. 没有充分利用index
2. 当节点所关联的关系数量增长时，链表结构的遍历访问效率会使得查询效率降低。
3. 查询计划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。

### 算法设计

本文提出了新的特定路径查询的实现算法，基本思想是：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。

1. **BloomFilter介绍**

BloomFilter[33,34,35]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。

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。

1. **算法设计**

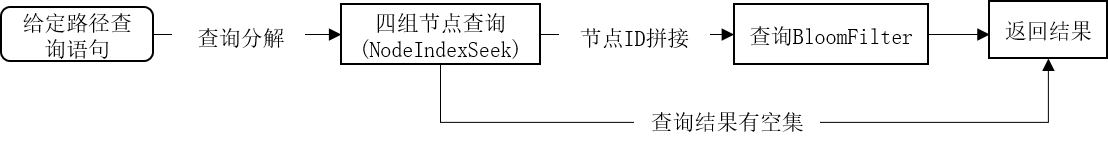
基于BloomFilter的特定路径查询算法中，将IfcSite、IfcBuilding、IfcBuildingStory、IfcDoor（以及其它建筑构件）的ID按照顺序拼接为字符串，存入BloomFilter中。当查询来到时，首先分解整个查询为四个针对节点的查询，如果有结果集为空，则可以直接结束整个查询，返回给用户空值；如果都不为空，将满足条件的节点ID拼接起来形成字符串。查看字符串是否在BloomFilter当中，就可以知道这个结果集是否存在。BloomFilter存在一定的误判。当这条路径并不在Neo4j存储的图中时，可能会判定为存在，此时用户得到的结果将会是由原始的Cypher查询得到的正确结果。同时，BloomFilter不会漏掉判断，只要判定这个字符串在BloomFilter当中，那么它一定存在。基于BloomFilter的特定路径查询算法流程如图4-10所示。

图4-10 基于BloomFilter的特定路径查询算法流程

算法的优点有以下三点：

1. 将查询分解为多个对节点的查询，可以充分利用index；
2. 由于分解查询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。

### 算法实现

基于BloomFilter的特定路径查询算法在Neo4j中的实现包括构建BloomFilter、路径查询接口实现等主要问题。

BloomFilter的构建依赖于IfcSite、IfcBuilding、IfcBuildingStory、建筑构件等的信息。在内存中解析IFC模型后借助以下3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，可以任何一个建筑构件IFC对象所对应的IfcBuildingStory、IfcBuilding、IfcSite的ID，拼接ID为长字符串，插入到BloomFilter中。BloomFilter将被序列化到物理文件中，在数据库实例启动之后加载入内存。由于BloomFilter占用的内存空间相对较小，因此可以常驻内存。

通过实现Neo4j的插件接口实现基于BloomFilter的查询，扩展Neo4j的REST服务。数据库实例在启动时自动安装插件，使得用户可以在web界面通过REST API提交查询参数进行特定路径查询时。

# 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。本文以改进的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。

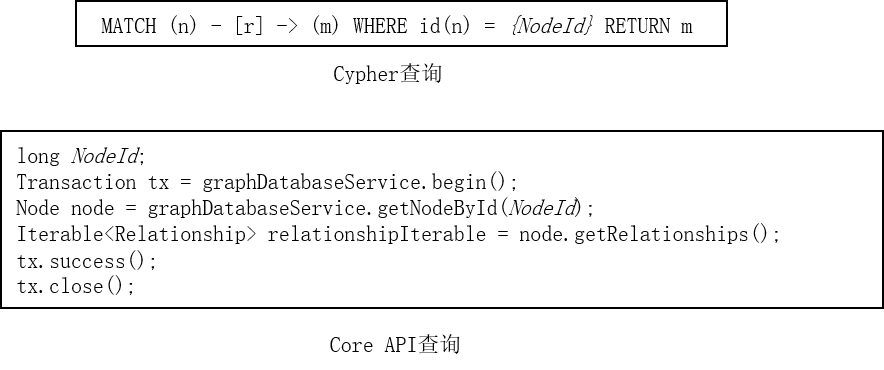
### 数据集

实验数据集采用229M的IFC数据文件，解析、存储和索引到Neo4j数据库中。数据库中包含4,079,353个节点，9,102,634个关系。

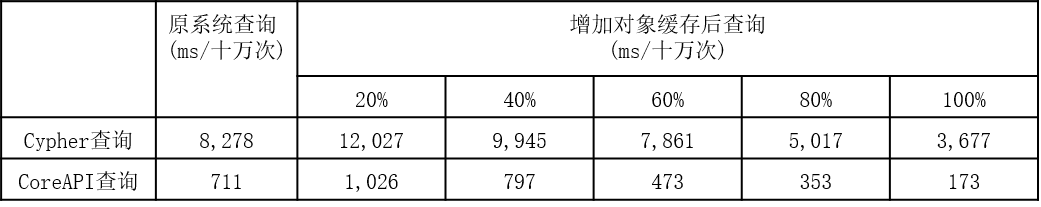
### 实验与分析

1. **对象缓冲策略和页缓存策略的对比实验与分析**

实验中随机选取100,000个节点，进行“节点的所有关系”500,000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。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Graph API两种实现，如图5-1所示：

图5-1 Cypher和Graph API实现“节点的所有关系”查询

缓冲区大小的设置直接影响了查询的响应时间，缓冲区较大时，可以缓存更多的数据，命中次数更多，且不需要频繁的替换旧数据项和加载新的数据项。根据实验选择的100000个节点的关系数据查询请求的数据的总内存占用MemSize，本文将缓冲区的大小设置为MemSize的20%、40%、60%、80%、100%，100%表示查询请求的所有节点的关系数据都会加载入对象缓冲区，在随后的访问中直接命中，而不需要访问页缓存，因此能够反映出直接访问对象缓存的效率。

表5-1 原系统查询耗时和增加对象缓存后查询耗时结果表

由表5-1可以看出，当对象缓存空间很小，出现大多数情况下都无法命中的状况时，由于需要重新去页缓存加载相关数据来构建对象缓存的缓存项，包括构建索引数据和关系数据，这反而会使查询请求的耗时比原系统的耗时要长。但是，当对象缓存空间越来越大时，大部分情况下查询请求可以命中对象缓存中的数据并返回，查询耗时会显著减少。实验结果的折线图如图5-2所示。当对象缓存100%命中时，Cypher查询和Graph API查询的响应时间都减少了超过50%，相比于原系统的查询性能有了巨大的提升。

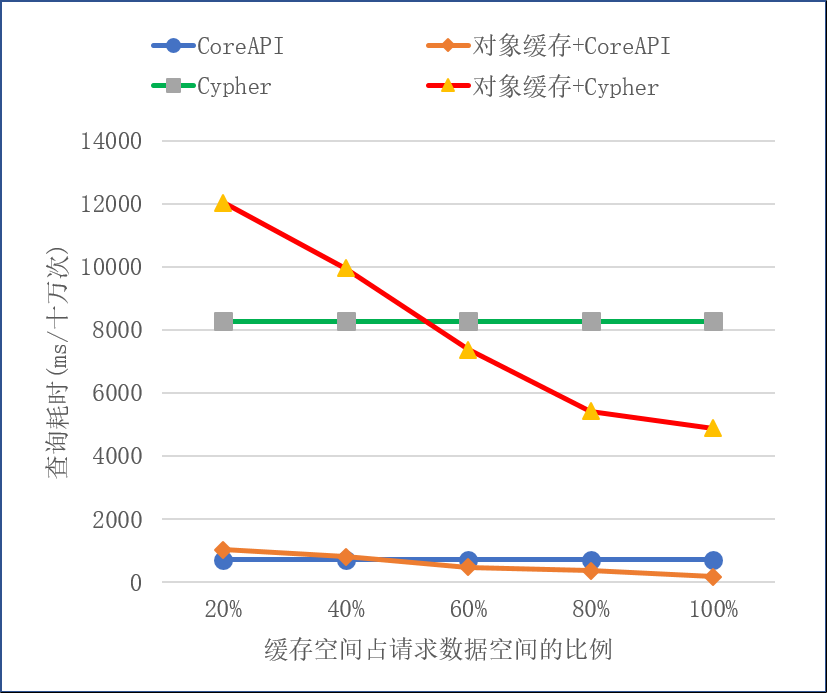
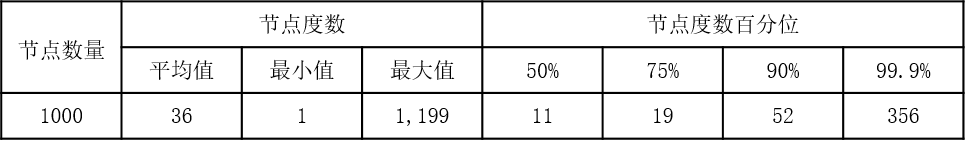


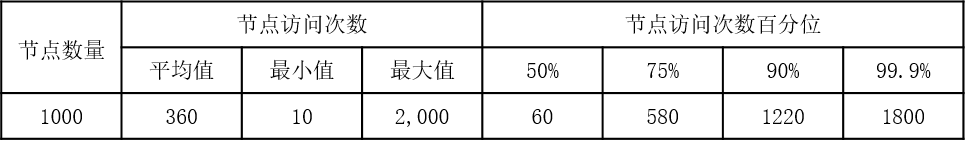
图5-2 原系统查询耗时和增加对象缓存后查询耗时对比

超出的时间应该怎么算呢，上下不一致

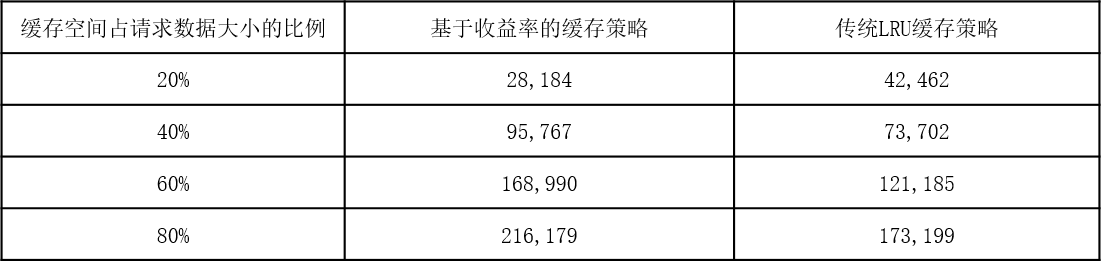
1. **基于收益率的缓存策略和传统LRU缓存策略的对比实验与分析**

实验中将对象缓冲区大小固定为1M，选取1,000个节点进行查询实验，保证节点的所有关系数据的内存占用MemSize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1,000个节点的进行“节点的所有关系”模拟查询，共293,360次。其中对部分节点重点进行重复访问，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 Graph API实现。对比两种策略的缓存命中次数如表5-3所示。

表5-4两种缓存策略的缓存命中数实验结果表

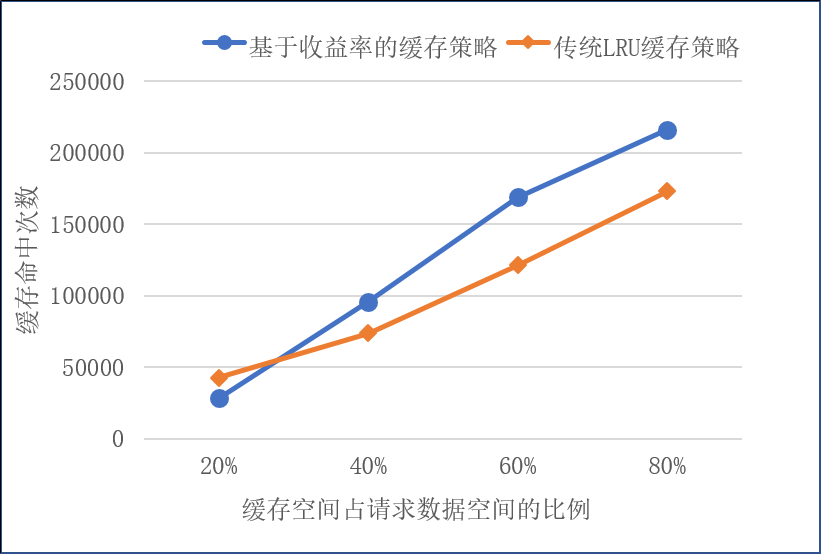


图5-3 两种缓存策略的缓存命中数对比图

## 特定路径查询算法实验

### 数据集

实验采用模拟生成的数据集进行实验。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

表5-5的数据表明，基于BloomFIlter的特定该路径查询算法耗时更短，特别是在节点数量较大时，访问BloomFilter比访问链表的效率更高，因此效率提升更加明显。

## 数据访问实验

7474界面、以及REST API 访问接口；

# 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技术得到了广泛关注。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。

为了解决以往BIM数据库存在的对于IFC模型关系数据查询效率低下的问题，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，并对Neo4j进行了关键点改进，包括对象缓存策略、预装载策略、基于BloomFilter的特定路径查询插件服务，进一步提升了Neo4j的查询性能，以满足BIM数据库的需要。

## 展望

本文针对Neo4j中的关键技术点进行了研究和改进，有效地提升了基于Neo4j的BIM数据库的查询效率，更加适应BIM数据库发展的需要，但是依然存在一些改进空间：

1. 本文提出的对象缓存采用固定大小的缓冲区空间分配，在系统实际运行中内存资源不停地发生变化，需要自适应的缓冲区分配算法以便更合理、充分地利用内存资源。
2. 本文将Neo4j中存储的节点属性全部建立了索引以便加速查询过程，但是由于IFC模型中存在热点访问对象和冷对象，冷对象上的索引并没有被利用，导致了一定的磁盘空间的浪费。更进一步的研究需要从基于访问日志动态建立节点索引方面展开。

# 参考文献

[1] 张建平. BIM技术的研究与应用. 施工技术，2011(2).

[2] 李美华. 建筑产品BIM数据库的价值及发展前景[J]. 建设科技，2015(23):42.

[3] 张洋. 基于BIM的建筑工程信息集成与管理研究[博士学位论文]. 清华大学，2009

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

[5] Fu C, Aouad G, Lee A. Mashall A. IFC model viewer to support nD model application[J]. Automation in Construction, 2006, 15(2):178-185.

[6] Mell P, Grance T. The NIST Definition of Clouding Computing[J]. National Institute of Standards and Technology, 2011, 53(6):50

[7] 徐锐. 基于IFCXML的建筑数据共享平台的研究与设计[硕士学位论文]. 复旦大学，2006

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

[9] Ali M T, Ghassan A. Moving beyond the fourth dimension with an IFC-based single project database[J]. Automation in Construction, 2005, 14(1):15-32.

[10] 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），2012,52(6):836-842.

[11] 李卓玲. 对象关系型数据库应用策略. 沈阳工程学院学报（自然科学版），2001(4):30-33.

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

[13] 陈明. NoSQL数据库系统[J]. 计算机教育. 2013(11)

[14] BIMServer, The Building Information Modelserver (short: BIMServer) projects.

<http://bimserver.org/>. 2012.

[15] 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(4):348-353

[16] 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[J]. 土木建筑工程信息技术. 2013, 5(6):1-5

[17] Khetrapal A, Ganesh V. HBase and Hypertable for large scale distributed storage systems[J]. Dept. of Computer Science, Purdue University, 2006.

[18] Jia-Rui Lin, Zhen-Zhong Hu, Jian-Ping Zhang, Fang-Qiang Yu. A Natural-Language-Based Approach to Intelligent Data Retrieval and Representation for Cloud BIM. Computer-Aided Civil and Infrastructure Engineering, 2016(31):18–33.

[19] Boicea A, Radulescu F, Agapin L I. MongoDB vs Oracle-Database Comparison[C]//EIDWT. 2012:330-335.

[20] R. Angles, C. Gutierrez. Survey of graph database models[J]. ACM Computing Surveys (CSUR), 2008, 40(1):1-6.

[21] Neo4j Com. The Neo4j manual v3.2. [DB/OL]. [2017-5-22]. <https://neo4j.com/docs/> .

[22] Miller J J. Graph Database Applications and Concepts with Neo4j[C]//Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA March 23rd-24th. 2013.

[23] Vicknair C, Macias M, Zhao Z, et al. A comparison of a graph database and a relational database: a data provenance perspective[C]//Proceedings of the 48th annual Southeast regional conference. ACM, 2010:42.

[24] 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.

[25] 张凤军. 基于Neo4j图数据库的社交网络数据的研究与应用[硕士学位论文]. 湖南大学，2016.

[26] 马义松，武志刚. 基于Neo4j的电力大数据建模及分析[J].电工电能新技术，2016, 35(2):24-30.

[27] 廖理. 基于Neo4j图数据库的时空数据存储[J]. 信息安全与技术，2015，6(8).

[28] 陆鹏. 基于Neo4j的大数据组织检索研究与应用[硕士学位论文]. 东南大学，2015

[29] 康杰华，罗章璇. 基于图形数据库Neo4j的RDF数据存储研究[J].信息技术，2015(6):115-117.

[30] 靳永超，吴怀谷. 基于Neo4j处理大数据中元数据溯源的研究. 现代计算机（专业版），2015, 08:61-64.

[31] Industry Foundation Classes [S/OL]. [2018-3-7]. <http://www.buildingsmart-tech.org/>.

[32] A. Dan, D. Towsley. An approximate analysis of LRU and FIFO buffer replacement schemes. ACM SIGMETRICS Conf, 1990:143-149.

[33] Bloom B H. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970, 13(7):422-426.

[34] 肖明忠，代亚非. Bloom Filter及其应用综述[J]. 计算机科学, 2004, 31(4):180-183

[35] Bloomfilter. http://wwwse.inf.tu-dresden.de/xsiena/bloom\_filter