一种基于UML关系的Java代码库构造方法

姜人和[1] 郑晓梅[2] 朱晓倩[1] 潘敏学[1] 张 天[1]
(南京大学计算机软件新技术国家重点实验室 南京 210023)[1]
(南京中医药大学信息技术学院 南京 210023)[2]

摘 要 关系信息是体现代码结构和语义的最重要的一类信息,如继承、聚合、组合、依赖、调用和创建实例等。为了更好地支持开源代码的理解与复用,提出了一种基于UML2关系的代码库构造方法。它以图数据库为实现平台,采用语言工程中经典的抽象语法树作为基础,并针对Java语言的特性和机制,设计富语义的Java代码属性图数据模型,在此基础上使得Java代码的图结构持久化。同时,为了屏蔽各种编程语言社区对代码中关系信息理解的差异性,采用UML2.4国际标准版本中定义的关系类型及语义解释,设计相应的代码关系抽取算法,为图节点添加对应的关系边。针对代码图化后的膨胀及代码库的空间存储消耗情况,选取9个常见的开源项目进行了实验评估。最后,给出了基于此代码库的查询应用实例。

关键词 代码库,UML,图数据库,代码查询
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.11.011

Method of Java Code Repository Construction Based on UML Relationship

JIANG Ren-he[1] ZHENG Xiao-mei[2] ZHU Xiao-qian[1] PAN Min-xue[1] ZHANG Tian[1]
(Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China)[1]
(School of Information Technology, Nanjing University of Chinese Medicine, Nanjing 210023, China)[2]

Abstract Relationship information is the most important information representing code structure and semantics, such as inheritance, aggregation, composition, dependency, call and creation. This paper provided a method constructing source repositories based on relationships defined in UML2 for better comprehension and reuse of the open source code. The graph database is used as the implementation plantform of the approach, and the abstract syntax tree is adopted as the base of graph schema. In addition, the schema is designed specifically according to the Java language so that semantics can be well presented. The key point of the approach is that the relationship definitions are prescribed strictly according to the UML2.4 specifications, which is an ISO version, so that the differentiation between different language communities about the comprehension of relationship can be eliminated. Each category of the relationships was studied and the corresponding recognition algorithms were designed. During the construction of the repository, the relationships are added as different kind of edges. The evaluation experiments of 9 open source projects were conducted to illustrate the expansion of the code graph and the consumed space. At last, two simple case studies of querying on the repository were given.

Keywords Code repository, UML, Graph database, Code query

1 引言

在开源浪潮的推动下,开源社区不断涌现,版本控制工具使得协同开发和代码管理更加方便。随着越来越多的代码被上传到开源代码库中,目前在软件产业界,很多软件开发人员在开发相应的软件系统之前都会搜寻一些开源软件项目作为系统构建的依据。由此可见,如今的开源代码库已经不仅仅是代码在云端的简单备份,它需要提供更加强大的代码存储和查询功能,以满足软件开发人员日益复杂的查询需求,从而更加高效地支持代码复用。

在理解代码的结构和语义方面,关系信息是最重要的一类代码信息,因此也是代码查询的主要关注点。然而,代码中

到稿日期:2016-10-09 返修日期:2016-12-31 本文受基于MDE的异构数据建模及转换研究(61472180),基于场景规约的中断驱动系统的建模与验证技术研究(61502228),基于SysML和MARTE的异构数据模型转换方法研究(BK20141322),中断驱动系统的建模与分析(BK20150589)资助。

姜人和(1994-),男,硕士生,主要研究方向为软件工程;**郑晓梅**(1978-),女,硕士,副教授,主要研究方向为软件工程;**朱晓倩**(1990-),女,硕士,主要研究方向为软件工程;**潘敏学**(1983-),男,博士,助理研究员,主要研究方向为形式化方法和软件工程;**张 天**(1978-),男,博士,副教授,主要研究方向为软件工程。

关系信息的种类繁多(如继承、聚合、组合、依赖、调用和创建实例等),这给查询的实现带来难度,也使查询的效率受到很大影响。针对这一问题,目前主要的解决方式是在构造代码库时就提取相应的关系进行保存,从而加速代码的查询。这种方式的主要问题是增加了设计持久化代码的数据模型的复杂性,这在使用关系型数据库和文档式数据库时尤为明显。

此外,更严重的问题是代码层的很多关系并没有统一的标准,因此各个语言社区的理解颇有偏差,这给代码查询和理解带来了很大困扰,同时也令代码查询工具的开发工作更加复杂。

针对代码结构和语义方面关系类型的复杂性,以及关系定义上的差异性,本文提出一种基于 UML 关系的 Java 代码库构造方法,采用图数据库 Neo4j[1] 作为底层实现平台,借助 Eclipse的 Java 开发工具(Java Development Tooling,JDT)[2] 获得 Java 代码的抽象语法树(Abstract Syntax Tree,AST),面向查询和 Java 语言的自身特点,以属性图为建模基础,对代码的数据模型进行定制。

在此基础上,为了屏蔽各种编程语言社区对代码中关系信息理解的差异性,本文采用了工业界广泛使用的统一建模语言(Unified Modeling Language,UML)[3] 作为理解关系的规范,具体采用 UML2.4 国际标准版本中定义的关系类型及语义解释,设计相应的代码关系抽取算法,对图结构的存储模式进行扩展,为图节点添加对应的关系边,实现对基于 UML 标准的关系查询的支持。

本文第 2 节介绍代码库的存储模式及其扩展;第 3 节介绍面向代码库的查询;第 4 节对代码库进行实验评估;第 5 节进行基于代码库的查询应用实例演示;第 6 节介绍相关工作;最后总结全文,并对未来研究方向进行展望。

2 代码库的构造

2.1 图结构存储模式

2.1.1 抽象语法树

将文本形式的代码转化为图结构时需要对代码的 AST 进行解析。AST 的关键元素是树节点,对于每一种树节点,可以知道树节点的类型、树节点拥有的子节点及树节点与子节点之间的联系类型。给定一个树节点和一个属性键,可以得到对应的属性值,定义 1 给出了 AST 的形式化定义。

定义 1(AST) 一个语法树是一个八元组 $\langle N_{AST}, A, NT, K, V, \varphi, \tau, \upsilon\rangle$,其中:

(1)N_{AST} 是语法树中所有树节点的集合,A 是树节点与子节点间所有可能联系类型的集合,NT 是所有树节点类型的集合,K 是所有树节点属性键的集合,V 是所有树节点属性值的集合。

(2)$\varphi: N_{AST} \times A \mapsto 2^{N_{AST}}$ 是联系子节点的函数。$\forall n \in N_{AST}, a \in A$,如果树节点 n 有联系类型为 a 的子节点,则 $\varphi(n, a)$返回这种子节点集合,否则返回 $\emptyset$。

(3)$\tau: N_{AST} \mapsto 2^{NT}$ 是获取树节点类型的函数。$\forall n \in N_{AST}$,$\tau(n)$返回树节点所属类型集合。

(4)$\upsilon: N_{AST} \times K \mapsto V$ 是获取树节点属性值的函数。$\forall n \in N_{AST}, k \in K$,如果树节点 n 拥有属性键 k,则 $\upsilon(n, k)$返回属性键 k 对应的属性值,否则返回 NULL。

定义 2(G-AST) 令 $\langle N_{AST}, A, NT, K, V, \varphi, \tau, \upsilon\rangle$ 表示一个语法树,*shadow* 是一个将树节点映射到图节点的单射函数,那么 $G\text{-}AST = \langle N, E, T, L, P, \mu, \eta\rangle$ 是其对应的图数据,其中:

(1)$N = \{shadow(n) \mid n \in N_{AST}\}$,$T = \{AST\}$,$L = NT$,$P = (K \times V) \cup (NAME \times A)$。

(2)$\forall n1, n2 \in N_{AST}$,如果 $\exists a \in A \wedge \varphi(n1, a) \supseteq n2$,则 $e = \langle shadow(n1), AST, shadow(n2)\rangle \in E \wedge \eta(e) = \langle NAME, a\rangle$。

(3)$\forall n \in N_{AST}, n' \in N$,并且 $shadow(n) = n'$,则 $\mu(n') = \tau(n)$。

(4)$\forall n \in N_{AST}, n' \in N$,并且 $shadow(n) = n'$,则 $\eta(n') = \{\langle k, \upsilon(n, k)\rangle \mid k \in K \wedge \upsilon(n, k) \neq \emptyset\}$。

示例代码 1:

```
void get(int a){if(a>3){}else{}}
```

首先定义一个名为 G-AST 的图结构(见定义 2),G-AST中的图节点与 AST 中的树节点是一一对应的关系。如果两个树节点之间有父子关系,则这两个树节点对应的图节点之间存在一条边,这条边从父节点对应的图节点指向子节点对应的图节点,边的类型为“AST”,拥有属性键为“NAME”的属性,属性值为父节点与子节点之间的联系类型。G-AST 中的图节点按照对应树节点的类型分类,即将 G-AST 中图节点的标签设置为对应树节点的类型。G-AST 中图节点的属性是其对应树节点的属性键值对集合,图 1 给出了示例代码 1 对应的 G-AST。

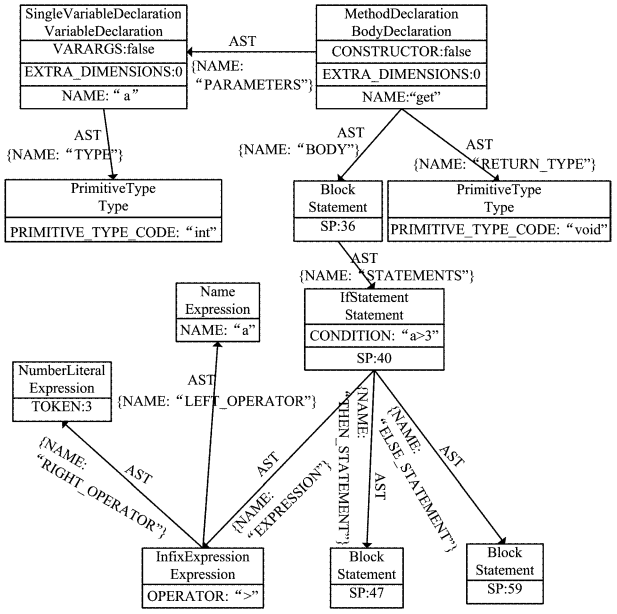


图 1 示例代码 1 对应的 G-AST

2.1.2 键值的引入

在 Java 程序中,不同包中的类可能具有相同的类名,方法的重写与重载机制也使得一个方法名可能对应多个不同的方法,同名变量更是不胜枚举。在得到的 G-AST 中无法直接区分同名实体,这给程序员理解代码增加了困难。尽管理论上可以通过图节点之间的层次关系分析出具体实体,但由于同名实体的数目以及程序复杂程度的不确定性,这种分析效率极低。为了方便快速地区分不同的实体,在 G-AST 的基础

上增加了一类新的图节点（键节点），从而产生了新的图结构 G-AST$^+$（见定义 3）。

定义 3（G-AST$^+$）　令 $L_{dk}=\{Type, AbstractTypeDeclaration, VariableDeclaration, MethodDeclaration\}$，$G\text{-}AST=\langle N,E,T,L,P,\mu,\eta\rangle$，$N_{dk}=\{n\mid n\in N$ 并且 $\mu(n)\cap L_{dk}\neq\varnothing\}$；$\Sigma$ 表示值的集合，$\kappa: N_{dk}\mapsto\Sigma$ 是从集合 N_{dk} 到集合 Σ 的一个单射；$\gamma: N\times N\mapsto\{\text{true},\text{false}\}$ 是一个函数，$\forall n1,n2\in N$，如果 $n1$ 使用 $n2$，则 $\gamma(n1,n2)=\text{true}$，否则 $\gamma(n1,n2)=\text{false}$。那么 $G\text{-}AST^+=\langle N^+,E^+,T^+,L^+,P^+,\mu^+,\eta^+\rangle$ 是一个图数据，其中：

(1) $N\subseteq N^+$，$E\subseteq E^+$，$T^+=T\cup\{KEY\}$，$L^+=L\cup\{Key, TKey, Vkey, Mkey\}$，令 $P'=\{VALUE\}\times\{\kappa(n)\mid n\in N_{dk}\}$，$P^+=P\cup P'$。

(2) $\forall n_{dk}\in N_{dk}$，有且仅有一个 $n_k\in N^+-N\wedge\langle n_{dk}, KEY, n_k\rangle\in E^+-E$。

(3) $\forall n\in N, n_{dk}\in N_{dk}$，如果 $\gamma(n,n_{dk})=\text{true}$，且 $\langle n_{dk}, KEY, n_k\rangle\in E^+-E$，则 $\langle n, KEY, n_k\rangle\in E^+-E$。

(4) $\forall n\in N, \mu^+(n)=\mu(n)$；$\forall n_k\in N^+-N$，并且 $\langle n_{dk}, KEY, n_k\rangle\in E^+-E$，令 $v=\mu(n_{dk})\cap L_{dk}$：

1) if $v\subset\{Type, AbstractTypeDeclaration\}$ $\mu^+(n_k)=\{Tkey, Key\}$。

2) if $v=\{VariableDeclaration\}$ $\mu^+(n_k)=\{VKey, Key\}$。

3) if $v=\{MethodDeclaration\}$ $\mu^+(n_k)=\{MKey, Key\}$。

(5) $\forall n\in N, \eta^+(n)=\eta(n)$；$\forall n_k\in N^+-N, \eta^+(n_k)=\langle VALUE, \kappa(n_{dk})\rangle$；$\forall e\in E, \eta^+(e)=\eta(e)$。

我们为每个类型、方法、变量计算一个全局唯一的键值来区分不同的实体。类型、方法、变量对应的标签无外乎是 *Type*，*AbstractTypeDeclaration*，*VariableDeclaration* 和 *MethodDeclaration*，我们为 G-AST 中所有具有这些标签的图节点添加一个键节点，并添加图节点到键节点的边，边的类型为“KEY”。为该键节点添加一个属性，属性键为“VALUE”，属性值由一个从图节点到属性值的单射函数生成，以确保该属性值能唯一标识一个图节点。键节点得到的标签由对应的图节点的标签决定，如果标签中包含 *Type* 或 *AbstractTypeDeclaration*，则说明图节点表示的实体为类型或者类型定义，对应键节点的标签为 *TKey* 和 *Key*；如果标签中包含 *VariableDeclaration*，则说明图节点表示的实体为变量定义，对应键节点的标签为 *VKey* 和 *Key*；如果标签中包含 *MethodDeclaration*，则说明图节点表示的实体为方法定义，对应键节点的标签为 *MKey* 和 *Key*。

假设实体间的使用关系是可知的，如果一个图节点对应的实体使用了类型、方法、变量对应的实体，则添加从该图节点到类型、方法、变量对应的图节点对应的键节点的边，边的类型为“KEY”。由此可见，通过为类型、方法、变量对应的图节点添加键节点，可以在产生同名冲突的情况下迅速进行节点定位，图 2 给出了示例代码 1 对应的 G-AST$^+$。

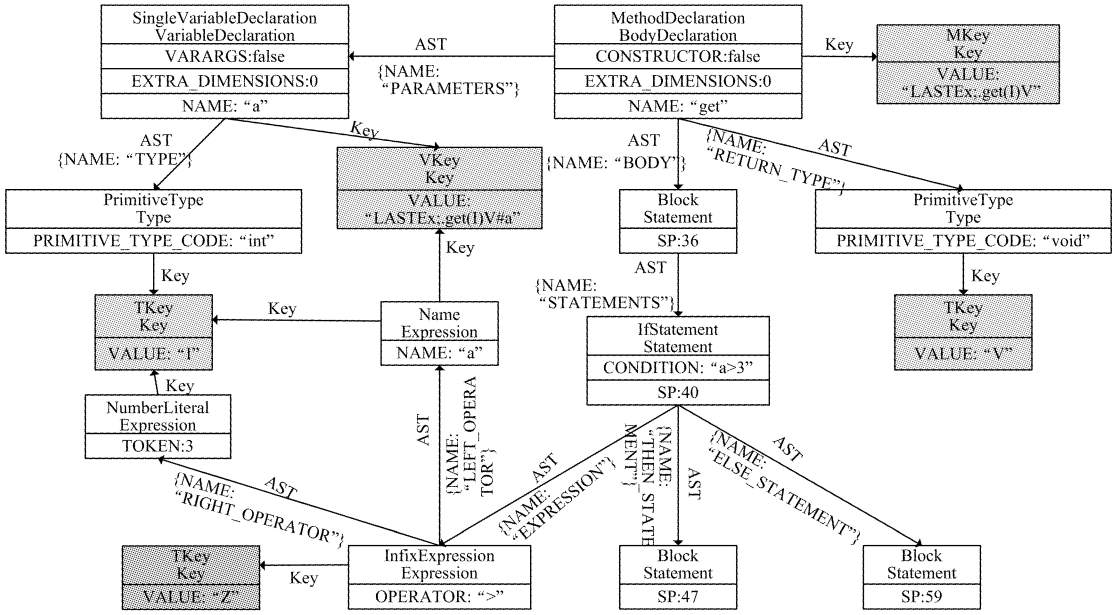


图 2　示例代码 1 对应的 G-AST$^+$

2.1.3　图节点的合并

尽管 Neo4j 支持的数据规模很大，但随着数据库中数据的不断膨胀，查询效率会受到影响。我们发现，G-AST$^+$ 中存在一些反复出现的图节点，而这些图节点并没有提供新的有用信息，例如修饰符“public”“protected”“private”等对应的图节点。这些图节点是冗余的，因此我们考虑将这些图节点进行合并以缩小图的规模，将压缩后的图结构称为 G-AST^{++}（见定义 4）。

定义 4（G-AST^{++}）　令 $G\text{-}AST^+=\langle N^+,E^+,T^+,L^+,P^+,\mu^+,\eta^+\rangle$，$L_c=\{Modifier, PackageDeclaration, Type, BooleanLiteral, CharacterLiteral, NullLiteral, NumberLiteral, StringLiteral, TypeLiteral\}$，$N_c=\{n\mid n\in N^+\wedge\mu(n)\cap L_c\neq\varnothing\}$。$equal: N^+\times N^+\mapsto\{\text{true},\text{false}\}$ 是判断图节点相等的函数，$\forall n1,n2\in N^+$，如果 $\mu^+(n1)=\mu^+(n2)$ 且 $\eta^+(n1)=\eta^+(n2)$，则 $equal(n1,n2)=\text{true}$；否则，$equal(n1,n2)=\text{false}$。将 N_c 划分为等价类，$\forall n1,n2\in N_c$，如果 $equal(n1,n2)=\text{true}$，则 $n1,n2$ 属于同一等价类，令 A 表示所有等价类的集合，那么 $G\text{-}AST^{++}=\langle N^{++},E^{++},T^{++},L^{++},P^{++},\mu^{++},$

$\eta^{++}\rangle$是一个图数据，其中：

(1)令 $N_r=N^+-N_c$，则 $N_r\subseteq N^{++}$；令 $E_c=\{e=\langle n1,X,n2\rangle|e\in E^+\wedge n1,n2\notin N_c\wedge X\in T^+\}$，$E_r=E^+-E_c$，则 $E_r\subseteq E^{++}$，$T^{++}=T^+$，$L^{++}=L^+$，$P^{++}=P^+$。

(2)$\forall a\in A$，有且仅有一个 $n_e\in N^{++}-N_r$，且对任意$n_a\in a$，$equal(n_e,n_a)=$true。

(3)$\forall n_c\in N_c$，$n\in N_r$，且 $e_c=\langle n,AST,n_c\rangle\in E_c$，如果$n_e\in N^{++}-N_r\wedge equal(n_e,n_c)=$true，那么 $e_e=\langle n,AST,n_e\rangle\in E^{++}-E_r$，并且 $\eta^{++}(e_e)=\eta^+(e_c)$。

(4)$\forall n_c\in N_c$，$n_k\in N^+$，并且 $e_c=\langle n_c,KEY,n_k\rangle\in E_c$，如果 $n_e\in N^{++}-N_r\wedge equal(n_e,n_c)=$true，那么 $e_{ek}=\langle n_e,KEY,n_k\rangle\in E^{++}-E_r$，并且 $\eta^{++}(e_{ek})=\eta^+(e_c)$。

(5)$\forall n_{c1},n_{c2}\in N_c$，并且 $e_{cc}=\langle n_{c1},AST,n_{c2}\rangle\in E_c$，如果 $n_{e1},n_{e2}\in N_e$，并且 $equal(n_{e1},n_{c1})=$true，$equal(n_{e2},n_{c2})=$true，那么，$e_{ee}=\langle n_{e1},AST,n_{e2}\rangle\in E^{++}-E_r$，并且 $\eta^{++}(e_{ee})=\eta^+(e_{cc})$。

(6)$\forall n\in N_r$，$\mu^{++}(n)=\mu^+(n)$，$\eta^{++}(n)=\eta^+(n)$；$\forall e\in E_r$，$\eta^{++}(e)=\eta^+(e)$。

若两个图节点拥有完全相同的标签和属性集合，则这两个图节点是相等的。我们需要的合并后的冗余节点就在相等的图节点中产生，但显然并不是所有相等的图节点都是冗余的。在此列出需要压缩的图节点包含的标签：Modifier，PackageDeclaration，Type，BooleanLiteral，CharacterLiteral，NullLiteral，NumberLiteral，StringLiteral，TypeLiteral。

选择这些图节点进行压缩是因为压缩后不会产生副作用，它们都处于图的边缘，即这些图节点都没有出边。需要注意的是，Type 类型的图节点并不满足这一特点，但对于一个代码中的类型，它在图结构中对应的子图结构是固定的，将这个子图看作一个整体，它没有出边。

我们将具有上述标签的图节点按照图节点相等划分为等价类，对于每个等价类，创建一个新的图节点（也属于该等价类）作为合并后的图节点。在不需要合并的图节点中，如果有指向需要合并的图节点的边，那么创建一条新边指向合并后的图节点，边的标签和属性与原来的边相同。

对于 Type 类型的图节点，它们存在一条关系类型为“KEY”的出边，合并后这些图节点也应添加指向键节点的边。对于复合类型而言，Type 类型的图节点之间也有边，在合并后的图节点之间也应添加这些边，边的标签和属性与原来的边相同。

最后，将被合并的节点以及与之相关联的边从图结构 G-AST$^+$ 中删除就得到了压缩之后的图结构 G-AST^{++}，即最终使用的 Java 代码的图结构，图 3 给出了示例代码 2 对应的 G-AST^{++}。

示例代码 2：

public class MergeTest{public int a;public int b;}

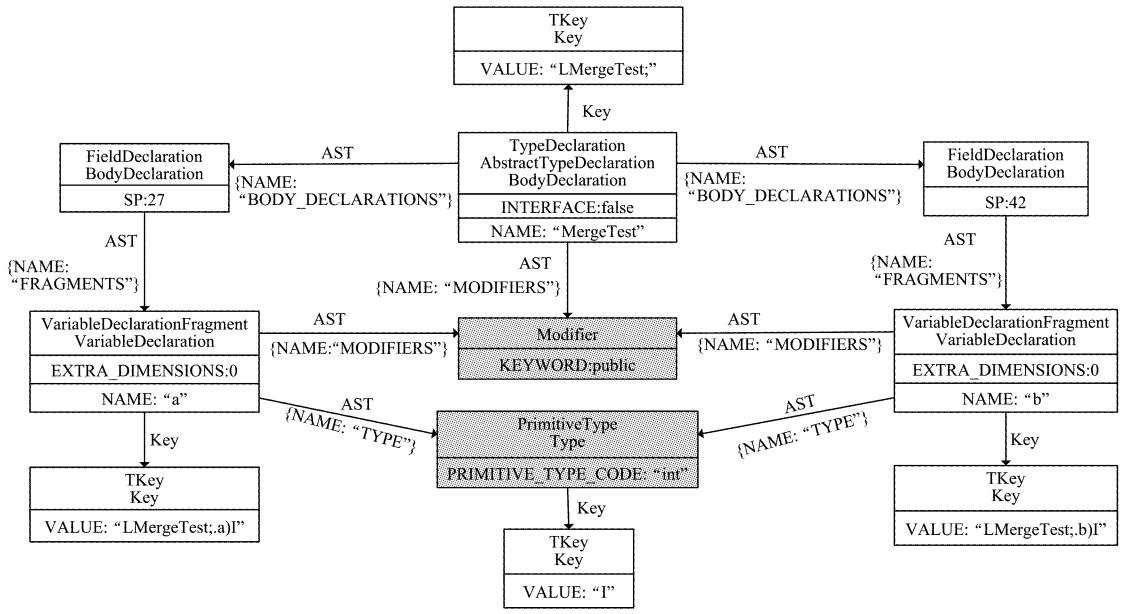


图 3 示例代码 2 对应的 G-AST^{++}

2.2 基于 UML 规范的关系扩展

为了支持对代码元素间关系的查询，需要对图结构存储模式进行扩展，扩展的方式是在 G-AST^{++} 的基础上在图节点之间添加边，边的标签和属性值标识了可供查询的关系类型。

为了使关系查询标准化、规范化，基于 UML 标准对关系进行归类，对每一种关系给出明确的描述。对照 UML 2.4.1 标准，UML 的类间关系主要分为 4 类：依赖关系（Dependency）、关联关系（Association）、实现关系（Realization）和泛化关系（Generalization），其中一部分类间关系还包含有相对应的衍型（见图 4）。

针对每一种关系提出对应的代码模式，即这些关系在 Java 代码中的体现方式，根据每一种关系的代码模式，在 AST 上寻找代码模式中每一个元素对应的树节点，然后将这些树节点归纳成可用于关系提取的节点路径，建立每一种代码模式与树节点路径之间的映射关系列表。根据映射关系列表设计 AST 的遍历算法，查看 AST 上是否存在与某种关系的代码模式相对应的树节点路径，如果存在，则将该种关系的边添加到对应的图节点之间。表 1 列出了本文涉及到的 UML 关系的具体描述。

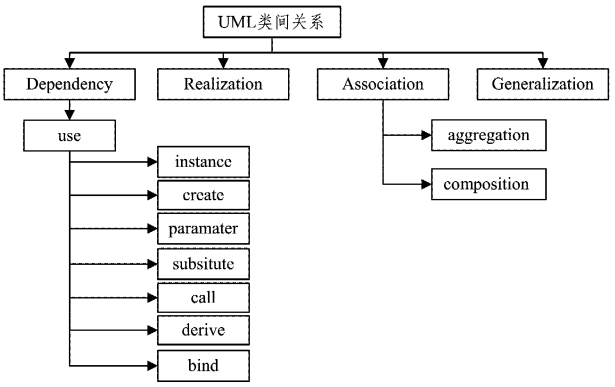


图 4　UML 关系分类

表 1　UML 关系描述

关系类型	关系描述
泛化关系	泛化关系是用来表示类元一般描述和具体描述之间联系的类间关系。
接口实现关系	接口实现关系是用来描述说明和现实的关联的类间关系。
关联关系	关联关系主要用来表述类与类之间的联结，是一种 hasa 的关系。
实例化依赖关系	实例化依赖是用来描述一个类的方法创建了另一个类的声明的衍型。
创建型依赖关系	创建型依赖是用来描述一个类创建了另一个类的实例的衍型。
参数依赖关系	参数依赖是用来描述一个类的方法的参数是另一个类实例的依赖关系的衍型。

2.2.1　泛化关系

泛化关系(Generalization)是一种表示类元一般描述和具体描述之间联系的类间关系。类元的具体描述建立在一般描述的基础之上，是对一般描述的实现和扩展。具体描述具有一般描述的所有特性、成员和关系，是对一般描述的一种补充。一般描述被称为父类，具体描述被称为子类，最常见的泛化关系是继承关系。

泛化关系的代码模式描述的 Java 代码是有以下特征：代码中如果存在泛化关系，那么代码中必须存在关键字“extends”；如果被继承的类元的类型是类，那么被继承的类的个数必须为 1，同时继承的类元的类型也必须是类；如果被继承的类元的类型是接口，那么被继承的接口的个数可以多于 1 个，但是继承的类元的类型必须是接口。

泛化关系的代码模式对应的抽象语法树路径有以下两条：1)获取继承类元即当前类的信息的节点路径：

CompilationUnit－＞TypeDeclaration－＞NAME－＞SimpleName

2)获取被继承类元的信息的节点路径：

CompilationUnit－＞TypeDeclaration－＞SUPERCLASS_TYPE－＞SimpleName

基于路径匹配的关系提取算法有多种实现方式，下面以 Cypher 语句[4]为例来演示泛化关系的提取算法对应的一种具体实现。

语句 1　类继承的情况

match (C1: TypeDeclaration {INTERFACE: {false}})-[: AST {NAME: 'SUPERCLASS TYPE'}]－＞()-[:KEY]－＞(:TKey)＜－[:KEY]-(C2:TypeDeclaration{INTERFACE:{false}})

create (C1)-[:EXTENDS]－＞(C2)

语句 2　接口继承的情况

match (I1: TypeDeclaration {INTERFACE: {true}})-[: AST {NAME: 'SUPER INTERFACE TYPES'}]－＞()-[:KEY]－＞(:TKey)＜－[:KEY]- (I2:TypeDeclaration {INTERFACE:{true}})

create (I1)-[:EXTENDS]－＞(I2)

2.2.2　接口实现关系

接口实现关系(Realization)是一种用来描述说明和实现的关联的类间关系。接口是一种对行为的说明，而与接口相关联的类则是对接口的一种实现。一个接口可以与多个类存在接口实现关系，每个类分别对接口进行各自的实现。接口实现关系是 Java 语言中最重要的类间关系之一，因为接口为实现类提供了标准和模板，实现了构件的可插入性，极大地增强了 Java 程序的扩展性能。

接口实现关系的代码模式描述的 Java 代码实现具有以下特征：在代码实现中如果存在接口实现关系，那么代码中就一定会存在关键字“implements”；接口实现关系对关系两端的类元的类型都有严格的要求，实现端的类元类型必须是类，不能是接口，与此相对的被实现端的类元类型必须是接口，不能是类；实现端的类的个数必须是 1，而被实现端的接口则可以有多个。

接口实现关系的代码模式对应的抽象语法树路径有以下两条：

1)获取实现类即当前类的信息的节点路径：

CompilationUnit－＞TypeDeclaration－＞NAME－＞SimpleName

2)获取被实现接口的信息的节点路径：

CompilationUnit－＞TypeDeclaration－＞SUPER_INTERFAC_TYPES－＞SimpleName

2.2.3　关联关系

关联关系(Association)主要用来表述类与类之间的联结，在这种类间关系中一个类可以知道另一个类的公有属性和方法，是一种 hasa 的关系。在实际运用中，关联关系还分为单向关联和双向关联两种，单向关联表示只有一个类知道另一个类的公有方法和属性，而双向关联则表示两个类相互知晓对方的公有方法和属性。

关联关系主要有两个衍型，即聚合关系(Aggregation)和组合关系(Composition)。聚合关系是一种用来描述整体和部分的关系的关联关系的衍型。相较于普通的关联关系，它是一种更强的关系，表明整体拥有部分，但是部分仍然可以离开整体单独存在。组合关系同样也是一种用来描述整体和部分关系的关联关系的衍型，但是它比聚合关系还要强，由整体对部分进行生命周期的管理，部分不能脱离整体而单独存在。

聚合关系和组合关系的主要区别在于它们的生命周期不同，在代码中的主要表现形式是类属性的实例是否被程序主体显示地销毁。但是在 Java 程序中，受 Java 虚拟机垃圾回收机制的影响，程序员在编写 Java 程序时不需要再显示地对实例进行回收销毁，因此在 Java 程序中关联关系和其两种衍型的代码模式相同。

关联关系的代码模式描述的 Java 代码实现具有以下特征：存在关联关系的类元中类的属性个数必须大于或等于 1，即类元中必须有类属性存在；类元的类属性类型没有强制要

求，但是考虑到在类元与基本类型（如 Integer，Double 等）之间建立关联关系的意义不大，而且关系会显得冗余，本文指定发掘的关联关系所指向的类元必须是项目中的类或接口；一个类属性的拥有者只能有一个，而且这个类属性的拥有者的类型必须是类。

关联关系的代码模式对应的抽象语法树路径有两条：

1）获取关联关系的当前类元的相关信息：

CompilationUnit－＞TypeDeclaration－＞NAME－＞SimpleName

2）获取被关联类的相关信息：

CompilationUnit－＞TypeDeclaration－＞BODY_DECL-ARATIONS－＞FieldDeclaration－＞TYPE－＞SimpleType－＞SimpleName

2.2.4 实例化依赖关系

实例化依赖（Instantiate Dependency）是一种用来描述一个类的方法创建了另一个类的声明的衍型。其通常表现为在一个类的方法体内部对另一个类进行实例化，生成一个属于另一个类的局部变量，同时在方法运行完毕后对这个实例进行销毁，其他方法无法调用该实例。

实例化依赖关系的代码模式描述的 Java 代码实现是有以下特征：该关系是在类元的方法内建立的，其所有者是类元的方法，而该方法的拥有者必须是类元中的类；在方法中必须存在至少一个实例的创建操作，在该实例的创建操作中必须存在关键字“new”，而且关键字的数目为 1，同时还需要有一个用于构造产生实例的构造函数。

实例化依赖关系的代码模式对应的抽象语法树路径有两条：

1）获取当前类元的相关信息：

CompilationUnit－＞TypeDeclaration－＞NAME－＞SimpleName

2）搜索类的方法中的实例化操作，获取实例化操作的相关信息：

CompilationUnit－＞TypeDeclaration－＞BODY_DECLARATIONS－＞MethodDeclaration－＞BODY－＞Block－＞ClassInstanceCreation－＞TYPE－＞SimpleType－＞SimpleName

2.2.5 创建型依赖关系

创建型依赖（Create Dependency）是一种用来描述一个类创建了另一个类的实例的衍型。与实例化依赖非常相似，它同样是在一个类的方法体内部对另一个类进行实例化，区别在于创建型依赖最终将这个实例返回，以供其他的方法和类进行调用。

创建型依赖关系的代码模式描述的 Java 代码实现有以下特征：该关系是在类元的方法内建立的，其所有者是类元的方法，而该方法的拥有者必须是类元中的类；依赖关系的代码实现中，存在创建型依赖关系的方法必须具有一个返回值，且本文指定该返回值的类型必须是工程中的类或者接口，不能是基础类型，这样可以避免大量的冗余关系；在方法中必须存在至少一个实例的创建操作，在该实例的创建操作中必须存在关键字“new”，而且关键字的数目为 1，同时还需要有一个用于构造产生实例的构造函数。

创建型依赖关系的代码模式对应的抽象语法树路径有两条：

1）获取当前类元的相关信息：

CompilationUnit－＞TypeDeclaration－＞NAME－＞SimpleName

2）搜索类的方法中的创建操作，检测方法的返回值类型：

CompilationUnit－＞TypeDeclaration－＞BODY_DECLARATIONS－＞MethodDeclaration－＞RETURN_TYPE2－＞SimpleType－＞SimpleName

2.2.6 参数依赖关系

参数依赖（Parameter Dependency）是一种用来描述一个类的方法的参数，是另一个类的实例的依赖关系的衍型。参数依赖较为常见，通过这种方法，一个类可以使用另一个类的方法对自身的信息进行加工处理，或者一个类可以将自身的数据以参数的方式提供给另一个类使用，以此实现类间信息的交互。

参数依赖关系的代码模式描述的 Java 代码实现较为简单，共有以下特征：由于建立关系的对象是方法的参数，因此该关系的拥有者之一必然是方法；方法的参数必须具有类型声明，而且声明的类型应不少于 1 个，为了防止在模型中加入大量的冗余关系，参数的类型不能是基础类型，参数的类型应是项目中的类或接口。

参数依赖关系的代码模式对应的抽象语法树路径有两条：

1）获取当前类元的相关信息：

CompilationUnit－＞TypeDeclaration－＞NAME－＞SimpleName

2）搜索类的方法的参数类型：

CompilationUnit－＞TypeDeclaration－＞BODY_DECLARATIONS－＞MethodDeclaration－＞PARAMETERS－＞SingleVariableDeclaration－＞SimpleType－＞SimpleName

3 代码查询

Cypher 语言是 Neo4j 提供的查询语言，也是一种声明式（declarative）图查询语言。Neo4j 能够直接执行 Cypher 语句的增、删、改、查，无需进行其他处理，并且十分高效。因此，我们考虑使用 Cypher 语言来进行代码查询。下面给出 4 个查询示例，以说明如何使用 Cypher 语句在代码库中进行查询。

语句 3 查询继承树

```
match ()-[r:EXTENDS*..1]->() return r
```

语句 4 查询类中所有可见性为 public 的属性

输入：A 类的类名 className

输出：A 类中所有的 public 属性

```
match (:TypeDeclaration {NAME:className})-[:BODY_DECLARATIONS]->(d:FieldDeclaration)-[:MODIFIER]->(m:Modifier {KEYWORD:"public"})
return d
```

语句 5 查询类中的定义时被初始化的属性

输入：A 类的类名 className

输出：A 类中所有在定义时被初始化的属性

match (:TypeDeclaration {NAME:className})-[:BODY_DECLARATIONS]->(:FieldDeclaration)-[:FRAGMENTS]->(m:VariableDeclarationFragement{INTIALIZER:notnull})

return m

语句 6　查询类中具有指定返回类型的 method

输入：A 类的类名 AclassName，T 类的类名 TclassName

输出：A 类中所有返回值为 T 的 method

match (:TypeDeclaration {NAME:AclassName})-[:BODY_DECLARATIONS]->(d:MethodDeclaration)-[:RETURN_TYPE2]->()->[:Key]->(m:Key{QualifiedName:TclassName})

return d

4　代码库评估

4.1　实验对象

我们选取了 9 个开源的 Java 工程，这些工程的详细信息如表 2 所列。

表 2　实验对象的基本信息

工程	版本	Java	纯代码	备注	大小/M
ant	1.9.4	861	107007	92045	8.97
ivy	2.4.0	474	50430	17489	3.73
AOI	3.0.1	489	116634	20124	5.91
hsqldb	2.3.2	467	151914	69540	9.67
jedit	5.2.0	536	109236	45448	5.67
JSPwiki	2.10.1	353	40754	28929	3.3
lucene	5.3.1	2615	345479	141872	25.29
Vuze	5.7.0.1	3559	611863	127405	31.44
weka	3.6.12	1053	260290	179374	16.71
总计		10407	1793607	722226	110.51

对实验工程的 Java 文件数、代码行数以及存储空间（纯代码）进行统计，本文只对功能性代码进行实验，测试代码部分不包含在内。由表 2 可知，实验工程包含的 Java 文件数最少为 467 个，最多为 3559 个，总计 10407 个。

实验工程包含的纯代码最少为 40754 行，最多为 611863 行，总计 1793607 行，平均每个 Java 文件包含 2083 行代码。实验工程包含的备注最少为 17489 行，最多为 179374 行，平均每个 Java 文件包含 69 行备注。实验工程存储空间消耗最少为 3.73M，最多为 37.54M，总计 126.62M。

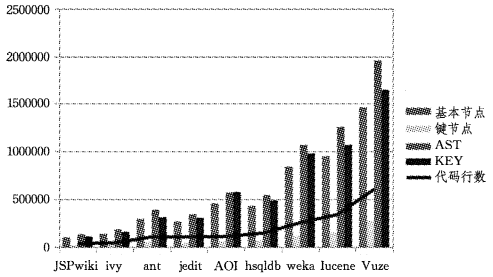
4.2　Neo4j 数据库存储能力的评估

本实验目的在于评估 Neo4j 数据库的存储能力，使用的 Neo4j 的版本号是 2.3.1。表 3 列出了实验工程对应的基本存储模式中图节点和关系的数目，我们将图节点分为基本节点（与语法树相关）和键节点。

表 3　节点关系数目

工程	图节点		关系		Javadoc 节点
	基本	键	AST	KEY	
ant	293443	57015	3906956	312080	12879
ivy	137919	28748	185345	159068	1156
AOI	456390	64368	571785	575019	4832
hsqldb	430781	68057	546151	487953	3945
jedit	267696	47372	344511	306499	3651
JSPwiki	99918	20842	133813	111802	3192
lucene	950662	156539	1257157	1067971	13527
Vuze	1460423	278250	1957052	1644618	8028
weka	840080	118658	1068795	979461	25322
总计	4937312	839849	6455565	5644471	76496

图 5 给出了图节点和关系数随纯代码行数变化的趋势，从图 5 可以看出，其变化步调基本一致。我们试图找到图节点、关系数目与纯代码行数之间的关系，并给出图节点、关系数目的估算方法。

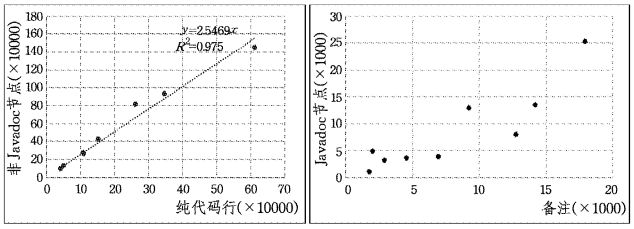


注：每一栏 4 条柱从左往右分别表示基本节点，键节点、AST、KEY，折线表示代码行数

图 5　图节点、关系与纯代码行数的关系

在存储代码时，除了纯代码会产生对应的图节点和关系外，备注行也会产生，在表 3 中“Javadoc 节点”统计了备注行对应的图节点（包含在基本节点中）的数目。在存储代码时，对于备注行，只存储文档类的备注，块备注和行备注并未被存储，并且每块文档备注在数据库中也对应一个图节点。因此，尽管备注行可能较多，但其在数据库中对应的图节点的数目远少于备注行数。

图 6 示出了代码行与图节点之间的关系。由图 6(a)可知，纯代码行与非 Javadoc 节点之间基本为线性关系，线性回归系数(y)为 2.5469，R^2 的值为 0.975。因此给定一个工程，可以利用其纯代码行数大致估计出非 Javadoc 节点的数目。而图 6(b)中的数据点比较散乱，备注行与 Javadoc 节点之间没有明显的关系。因此，使用备注行数来估计备注节点数目的准确性不高。从实验数据来看，Javadoc 节点在基本节点中所占的最高比例仅为 4%，因此在估算基本节点数目时，可以忽略备注行对图节点和关系数目的影响，认为非 Javadoc 图节点数约等于基本节点数。



(a)　(b)

图 6　代码行与图节点之间的关系

图 7 分析了键节点、AST 类型边以及 KEY 类型边的数目与基本节点数目之间的关系。

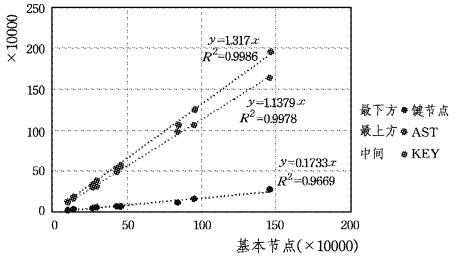


图 7　键节点、AST、KEY 与基本节点的关系

图 7 中最下方数据点(C)描绘了基本节点与键节点之间的关系,它们基本呈线性关系,线性回归系数为 0.1733,R^2 的值为 0.9669。图中最上方数据点(A)描绘了基本节点与 AST 之间的关系,它们基本呈线性关系,线性回归系数为 1.317,R^2 的值为 0.9986。图中中间数据点(B)描绘了基本节点与 KEY 之间的关系,它们基本呈线性关系,线性回归系数为 1.1379,R^2 的值为0.9978。

综上,基本存储模式中图节点和关系的数目估算方式如下:

Number(基本节点)≈纯代码行×2.5469

Number(键节点)≈Number(基本节点)×0.1733

Number(AST)≈Number(基本节点)×1.317

Number(KEY)≈Number(基本节点)×1.1379

Number(总图节点)≈纯代码行×3.0

Number(总关系)≈纯代码行×6.3

实验结论:Neo4j 数据库的代码存储能力主要取决于数据库中关系的数目,一个 Neo4j 数据库最多能存放约 550000 万行代码量。如果存储的 Java 工程纯代码行在 10 万行左右,则一个 Neo4j 数据库大约能存放 55000 个 Java 工程,这对普通个人或企业用户来说已经足够。对于存储更多工程的需求,用户需要借助集群来实现。

4.3 存储空间消耗评估

本实验的目的在于评估代码转换为基本存储模式后存储空间的消耗。在 Neo4j 数据库中,图节点、关系、属性等存放在不同的文件组中。其中,图节点和关系分别对应一个文件组;属性根据属性值的类型被分为基本类型、字符串类型和数组类型且分别存放在不同的文件组中(基本存储模式中不涉及数组类型的属性)。每个文件中的数据都有各自特定的格式,因此存储开销只与文件中的记录数目有关(除了字符串属性,还与字符串内容相关)。

我们将每个实验工程存放到独立的数据库中,分别统计数据库中的图节点、关系、属性、标签占用的存储空间,以评估代码存储到数据库后存储空间的膨胀情况。

表 4 统计了各个工程存储空间消耗的数据。同时,我们还统计了将所有工程存放到同一数据库中的存储空间消耗(对应于表中的 allInOne)。从表中数据可以发现,allInOne 每部分的存储空间消耗与将实验工程单独存储后计算得到的各部分空间消耗的总和基本相等。

表 4 存储空间消耗

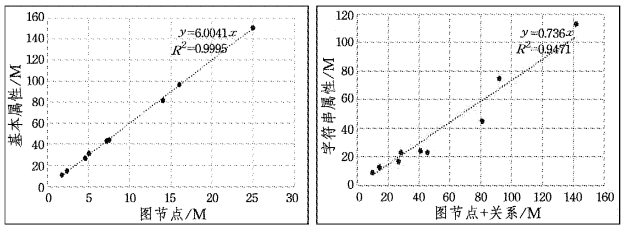
惠穗	图节点	关系	属性		标签	
			基本	字符串		
ant	5.1	23	31	23	1.6	83.7
ivy	2.4	12	15	13	0.7	43.1
AOI	7.5	38	44	23	2	114.5
hsqldb	4.6	22	27	17	1.4	72
jedit	1.8	8	11	8.6	0.5	29.9
JSPwiki	1.8	8	11	8.6	0.5	29.9
lucene	16	76	97	75	4.5	268.5
Vuze	25	117	151	113	7.3	413.3
weka	14	67	82	45	3.9	211.9
总计	83.6	397	501	341.6	23.9	1347.1
allInOne	83	393	496	337	24	1333

注:所有空间消耗单位为 M

在 Neo4j 中存储一个图节点的开销为 15 个字节,存储一个关系的开销为 34 个字节。因此,根据图节点和关系数能够估计出图节点和关系的空间消耗。标签的空间消耗占总空间消耗的比例很小,估算时可忽略不计。而属性需要的空间消耗并没有这么直接,因为每个图节点包含的属性个数以及类型各不相同。但直觉上,各种类型的图节点在整个图数据中的比例是相对稳定的,因此从理论上能大致估算出属性的空间消耗。

由于基本属性仅出现在图节点中,因此我们猜想基本属性的空间消耗与图节点的空间消耗有关,图 8(a)给出了两者之间的关系。从图中能够明显看出基本属性的空间消耗与图节点的空间消耗之间存在线性关系,线性回归系数为 6.0,R^2 的值为 0.9995。

由基本存储模式可知,字符串属性在图节点和关系中都有出现,因此我们猜想字符串属性的空间消耗与图节点和关系的空间消耗之和有关,图 8(b)给出了两者之间的关系。从图中大致能够看出字符串属性的空间消耗与图节点和关系的空间消耗之和之间存在线性关系,线性回归系数为 0.736,R^2 的值为 0.9472。



(a) (b)

图 8 存储空间消耗

综上,基本存储模式存储空间消耗(单位:字节)的估算方式为:

Space(图节点)≈Number(总图节点)×15

Space(关系)≈Number(总关系)×34

Space(基本属性)≈Space(图节点)×6.0041

Space(字符串属性)≈sum(Space(图节点),Space(关系))×0.736

总空间消耗量≈纯代码行×742

实验结论:代码转换为基本存储模式后,总存储空间消耗大约为纯代码行数乘以 742 字节。其中,存储关系以及属性消耗了绝大部分空间。总空间消耗的估计是略低的,因为在估计时忽略了标签的空间消耗。

4.4 存储和查询性能评估

本实验的目的在于通过分析图结构 G-AST^{++} 对存储空间的压缩效果以及在数据库中进行查询所需要的时间,评估基本存储模式的存储性能和查询性能。

首先将实验工程的 AST 完整地存入 Neo4j 数据库,然后将其转化为基本存储模式再存入 Neo4j 数据库,比较上述两种方式产生的数据库的存储空间,评估基本存储模式的存储性能。在得到的数据库中进行查询,为了更好地反映查询性能,查询语句选择语句 3 来查询继承树,根据查询时间评估代码库的查询性能。

表 5　存储空间和查询时间

工程	AST/M	G-AST++/M	查询继承树用时/ms
ant	178	83.7	1546
ivy	88.6	43.1	990
AOI	197	114.5	1286
jedit	169	29.9	1072
JSPwiki	88.1	29.9	1157
平均	144.14	60.22	1210.2

实验结论：在存储性能方面，基本存储模式对存储空间起到了明显的压缩效果，直接存储完整的 AST 所需的存储空间平均为基本存储模式的 2.4 倍；在查询性能方面，查询继承树平均用时为 1210.2ms，在实际情况中，具体的查询场景会获得更短的查询时间。

5　实例应用

5.1　继承深度计算

继承深度(DIT)是面向对象程序设计的重要度量指标，在单继承的情况下，一个类的继承深度等于类与继承树根节点之间的距离。类继承是代码复用的重要方法，但随着继承树深度的增加，也会带来一些问题。类的继承深度越大，类继承的方法越多，类的行为越难预测。同时，在设计软件时，继承树越深，涉及到的类和方法越多，设计的复杂性越大。

语句 7 的功能是计算工程中的类的平均继承深度和最大继承深度，表 6 列出了各实验工程中类的平均继承深度和最大继承深度。

语句 7　继承深度计算

```
match p=(n:TypeDeclaration{P_ID:1,INTERFACE:false})-
[:EXTENDS*0..]->()
with n,max(length(p)) as DIT
return avg(DIT),max(DIT)
```

表 6　继承深度

工程	平均 DIT	最大 DIT
ant	1.23	6
ivy	0.62	6
AOI	0.47	4
hsqldb	0.47	5
jedit	0.27	3
JSPwiki	0.49	3
lucene	0.97	5
Vuze	0.54	5
weka	0.68	5

5.2　循环调用

在阅读代码时，由于方法之间存在调用关系，因此常常要在方法之间跳转来理解代码。在跳转过程中涉及到的方法变得越来越多，从而增加了理解的难度。一些方法之间存在特殊的调用关系，方法之间通过调用关系连接成环，我们称之为循环调用。递归调用是最简单的循环调用。当方法之间存在循环调用关系时，代码变得更加难以阅读理解。而循环调用有时表示一个完整的工序循环，对于这种情况，如果能够找到工程中的循环调用，将有助于我们了解工程的实现逻辑。

语句 8 的功能是查找工程中特定个数函数构成的循环调用，表 7 列出了从 1 个到 7 个函数组成的循环调用。

语句 8　循环调用

```
match p=(n:MethodDeclaration{P_ID:1})-[:CALL*6]->(n)
with p,tail(nodes(p)) as ns
where all (x in ns where 1=length(filter(m in ns where m=x)))
return p
```

表 7　循环调用

工程	函数个数						
	1	2	3	4	5	6	7
ant	198	20	0	2	0	0	0
ivy	56	13	4	3	0	0	0
AOI	68	4	1	1	1	0	0
hsqldb	102	19	15	9	5	5	3
jedit	86	16	18	17	5	11	2
JSPwiki	4	2	3	6	2	1	0
lucene	126	20	30	0	1	0	2
Vuze	218	67	31	41	23	9	4
weka	292	47	5	0	0	0	0

6　相关工作

代码查询技术在代码分析和理解上起着重要的作用，代码查询技术的应用包括软件架构分析[5]、一致性检查[6]、缺陷定位[7]、模式发现[8]等。代码查询技术通常包含 3 个步骤：1)信息抽取，将源代码映射到某种中间结构；2)查询，对中间结果进行处理获得感兴趣的信息；3)显示，结果信息(文本、图形等)的展示。其中，信息抽取和查询是代码查询的关键步骤。信息抽取的中间结果常常存放在数据库中，这使得许多逆向工程的相关工具能够在数据库中的数据的基础上进行开发，而不用重新分析源代码。

源代码映射到的中间结果可能有多种结构类型，其中使用最多的是关系结构，中间结果保存在关系数据库中。Linton[9]于 20 世纪 80 年代开发了工具 OMEGA，为一种类 Pascal 语言提供了相关视图(版本、调用图、配置等)。OMEGA使用的是关系数据库，它设计的存储模式中包含 58 种关系，存储的数据包含了代码的全部信息。尽管这使得 OMEGA能够支持更加一般化的应用，但同时 OMEGA 的使用效率也受到了很大影响。

随后，Chen 等人[10]于 1990 年提出了 C 信息抽象系统，他们从 C 语言中抽取了函数、宏、全局变量、文件、类型之间的 11 种关系并存放到关系数据库中。这些关系是对程序结构进行分析的基础，主要应用于图形化视图显示、子系统抽取、无用代码删除等。由于数据库中数据的规模较小，C 信息抽象系统的使用效率相较于 OMEGA 得到了一定程度的提升。

随着面向对象语言的流行，代码查询技术逐渐向面向对象语言迁移。为了对 C++代码进行可达性分析并检测无用代码，Chen[11]于 1998 年设计了一种针对 C++语言的数据模型。通过解析 C++代码的语法树，他们从代码中抽取出实体之间的继承、友元、包含、引用关系，并将这些关系存放到关系数据库中。涉及到的实体包括函数、变量、类型、宏、文件等一般实体，以及用于修饰函数、变量、类型的实体，例如 public，private 等。

在使用面向对象语言的过程中，为了减少开发的工作量，程序员常常要使用已有框架或类库提供的 API，但 API 的使

用方式有时并不显而易见。Holmes 等人[12]于 2006 年开发了 Strathcona 工具来为程序员推荐 API。他们首先从 API 使用示例代码中抽取出关键实体和关系，并将其存放到关系数据库中。其中实体包括工程、类型、方法、属性，关系包括方法之间的调用关系、方法对属性的访问关系、类型之间的继承关系、方法对类型的使用关系。当程序员在编程过程中需要了解 API 的使用方式时，Strathcona 会对当前的上下文进行分析，同样抽取出上述实体和关系，并将其与示例库中的信息进行比较，返回最接近的示例。

同样，Bajracharya 等人[13]于 2006 年开发了代码查询引擎 Sourcerer，该成果的期刊版本于 2014 年出版[14]。他们从开源代码库中爬取了大量 Java 工程，从代码中抽取出 7 种实体和 11 种关系，并为实体关联了关键字以方便用户查询。Sourcerer 使用的也是关系数据库，由于它存储的代码结构信息粒度较细，如果把每个实体和关系存放在单独的表中，则将影响查询效率，因此 Sourcerer 仅使用两个表来分别存放所有的实体和关系，并采用排序的方式优化查询效率。

从以上工作中可以看出，使用关系数据库存储代码结构信息通常要面临对存储粒度和查询效率之间的权衡。因此关系数据库存储的信息通常粒度较粗并且面向特定应用。

除了关系结构，XML 也是表示代码结构的常用方式。纯文本格式的代码与自然语言相近，因此它便于程序员书写和阅读。同时，纯文本代码还广泛应用于文件系统、版本控制系统等。但对计算机来说，纯文本代码并不容易理解。使用 XML 格式存放代码结构信息的最大优点是便于计算机处理。

Badros[15]于 2000 年提出了一种 XML 格式表示的 Java 代码信息——JavaML，它通过嵌套的标签反映了代码的结构信息。Aguiar 等人[16]于 2004 年对 JavaML 进行了丰富，提出了 JavaML2.0。除了代码的结构信息，JavaML2.0 添加了符号之间的交叉引用信息、类型之间的依赖信息、代码的格式信息以及备注信息，支持与原文本代码之间无丢失的转换。Eichberg[17]于 2005 年开发了 SEXTANT 工具以支持代码理解，SEXTANT 底层使用了 XML 数据库来存放代码结构信息。

随着数据库技术的发展，一些新型的数据库也被应用到代码查询中。这些新型数据库各有优势，未来的研究空间很大。

Panchenko 等人[18]于 2011 年开发了 ACS 工具来查询满足要求的 ABAP 代码片段，ACS 底层使用了一种新型的面向列的内存数据库来存放代码的语法树信息。这种数据库中的信息也是以表的形式存放的，所有语法树节点都存放在一张表中，表中一行数据表示一个树节点，每行数据都记录了当前树节点在语法树中的先序和后序遍历的编号，以及其父节点的先序遍历编号。与关系数据库不同，ACS 使用的数据库对列的访问效率很高，加快了遍历语法树的速度，从而优化了查询效率。

Yamaguchi 等人[19]于 2014 年提出了一种新型的源代码表示方法，即代码属性图(Code Property Graph)。这种图结合了语法树、控制依赖图、数据依赖图。他们将 C 语言代码转换为代码属性图并存放到一种图数据库 Neo4j 中，目的是发现代码中的缺陷。通过对数据库的查询，他们最终发现了内存溢出、整型溢出、字符串形式缺陷以及内存泄露等多种缺陷。

Urma 等人[20]于 2015 年提出了利用 Neo4j 图数据库进行代码查询的方法，并说明了该方法在软件使用和演化上的应用。他们将 Java 代码的语法树信息存放到数据库中，并提出了在语法树上添加视图(类型层次、方法重写、方法调用、数据流等)的想法。但这些视图的添加是前向的，即需要在解析代码时添加。尽管能够通过查询数据库的方式添加视图，但这种方式添加的视图是不准确的。该项工作仍处于比较初期的阶段。

同样，Zhang[21]于 2015 年提出了一种对异构代码库进行语义查询的方法，使用一种文档数据库 MongoDB 来存放 JSON 格式的文档。作者将不同类型的面向对象语言的代码统一转换成 JSON 格式表示的语法树并存放到 MongoDB 中，从而克服了对异构代码进行查询时的困难。

结束语 本文提出了一种新的代码存取方法，这种方法以图为基本结构，存储于 Neo4j 数据库中，适用于目前主流的 Java 语言程序。

本文设计了适用于 Neo4j 的 Java 代码的基本存储模式 G-AST^{++}，G-AST^{++} 以语法树为基础，保证了足够细的存储粒度。同时，G-AST^{++} 为类型、方法、变量添加了键节点来区分同名实体，解决了代码中可能出现的同名问题。G-AST^{++} 对边缘图节点进行了合并，缩小了图数据的规模。

我们还提出了一种增量式的扩展存储模式的方法，并提供了支持 UML2.4 标准的关系的扩展方法，可以在 Neo4j 提供的查询语言 Cypher 的基础上对代码的关系信息进行查询。

本文工作还有继续完善的空间，尤其是在查询这一步骤上。尽管 Cypher 语言的查询能力很强，但实现同一目标的 Cypher 语句在执行性能上差异较大。尽管在 Cypher 语句的构成上有一些推荐的惯例，但这对用户的要求过多。同时，若要充分利用存储的代码信息，就必须对基本存储模式有充分的理解，这进一步加大了用户的负担。因此，完善查询语言是未来工作的一个重要方向。

另一方面，我们已经拥有了基本存储模式以及将代码转换为基本存储模式的自动化工具，这为在基础模式的基础上开发应用打下了良好的基础。未来，我们将在基本存储模式的基础上发现一系列应用，例如缺陷定位、模式发现等。

参 考 文 献

[1] Neo4j[OL]. http://neo4j.com.

[2] Eclipse JDT[OL]. http://www.eclipse.org/jdt.

[3] OMG UML, Superstructure. Version2.4.1[OL]. http://www.omg.org/spec/UML/2.4.1.

[4] Cypher[OL]. http://neo4j.com/docs/stable/cypher-query-lang.html.

[5] FEIJS L, KRIKHAAR R, VAN OMMERING R. A relational approach to support software architecture analysis[J]. Software: Practice and Experience, 1998, 28(4): 371-400.

[6] VERBAERE M, HAJIYEV E, DE MOOR O. Improve software quality with semmlecode: an eclipse plugin for semantic code search[C]// Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion. ACM, 2007: 880-881.

[7] YAMAGUCHI F, GOLDE N, ARP D, et al. Modeling and discovering vulnerabilities with code property graphs[C]// Proceedings of the 2014 IEEE Symposium on Security and Privacy.

IEEE,2014:590-604.
[8] PANCHENKO, KARSTENS J, PLATTNER H, et al. Precise and scalable querying of syntactical source code patterns using sample code snippets and a database[C]// Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension. IEEE,2011:41-50.
[9] LINTON M A. Implementing relational views of programs [C] // Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. ACM,1984:132-140.
[10] CHEN Y, NISHIMOTO M Y, RAMAMOORTHY C V. The c information abstraction system[J]. IEEE Transactions on Software Engineering,1990,16(3):325.
[11] CHEN Y, GANSNER E R, KOUTSOFIOS E. A C++ data model supporting reachability analysis and dead code detection [C]// Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Springer,1997:414-431.
[12] HOLMES R, WALKER R J. Approximate structural context matching: an approach to recommend relevant examples [J]. IEEE Transactions on Software Engineering,2006,32(12):952-970.
[13] BAJRACHARYA S, NGO T, LINSTEAD E, et al. Sourcerer: a search engine for open source code supporting structure-based search[C]// Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. ACM,2006:681-682.
[14] BAJRACHARYA S, OSSHER J, OSSHER J. Sourcerer: an infrastructure for largescale collection and analysis of open-source code[J]. Science of Computer Programming,2014,79:241-259.
[15] BADROS G. Javaml: A markup language for java source code [J]. Computer Networks,2000,33(1):159-177.
[16] AGUIAR, DAVID G, BADROS G. Javaml 2.0: enriching the markup language for java source code[M]// XML: Aplicac Oese Tecnologias Associadas,2004:1-12.
[17] EICHBERG M, HAUPT M, MEZINI M, et al. Comprehensive software understanding with sextant[C]// Proceedings of the 21st IEEE International Conference on Software Maintenance. IEEE,2005:315-324.
[18] PANCHENKO, KARSTENS J, PLATTNER H, et al. Precise and scalable querying of syntactical source code patterns using sample code snippets and a database[C]// Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension. IEEE,2011:41-50.
[19] YAMAGUCHI F, GOLDE N, ARP D, et al. Modeling and discovering vulnerabilities with code property graphs[C]// Proceedings of the 2014 IEEE Symposium on Security and Privacy. IEEE,2014:590-604.
[20] URMA R, MYCROFT A. Source-code queries with graph databases-with application to programming language usage and evolution[J]. Science of Computer Programming,2015,97:127-134.
[21] ZHANG T, PAN M, ZHAO J, et al. An open framework for semantic code queries on heterogeneous repositories[C]// Proceedings of the 2015 International Symposium on Theoretical Aspects of Software Engineering. IEEE,2015:39-46.

(上接第 68 页)

参 考 文 献

[1] RODRIGUEZ L R, LAWALL J. Increasing Automation in the Backporting of Linux Drivers Using Coccinelle[C]// Dependable Computing Conference. 2016:132-143.
[2] MACKENZIE D, EGGERT P, STALLMAN R. Comparing and Merging Files With Gnu Diff and Patch. Network Theory Ltd [OL]. http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html.
[3] MULLER G, PADIOLEAU Y, LAWALL J L, et al. Semantic patches considered helpful[J]. ACM Sigops Operating Systems Review,2006,40(3):90-92.
[4] BRUNEL J, DOLIGEZ D, HANSEN R R, et al. A foundation for flow-based program matching using temporal logic and model checking[C]// POPL 2009. 2009:114-126.
[5] PADIOLEAU Y, LAWALL J, HANSEN R R, et al. Documenting and automating collateral evolutions in Linux device drivers [C]// EuroSys 2008. Glasgow, Scotland: ACM,2008:247-260.
[6] PADIOLEAU Y, LAWALL J L, MULLER G. Under standing collateral evolution in Linux device drivers[C]// EuroSys. 2006:59-71.
[7] LAWALL J L, BRUNEL J, PALIX N, et al. WYSIWIB: exploiting fine-grained program structure in a scriptable API-usage protocol-finding process[J]. Software: Practice and Experience, 2013,43(1):67-92.
[8] ENGLER D R, CHELF B, CHOU A, et al. Checking system rules using system-specific, programmer-written compiler extensions[C]// OSDI'00. San Diego, CA,2000:1-16.
[9] CHOU A, YANG J, CHELF B, et al. An empirical study of operating systems errors[C]// Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01). New York, NY, USA, ACM,2001:73-88.
[10] PALIX N, THOMAS G, SAHA S, et al. Faults in Linux: ten years later[C]// 16th International Conference on Architectural Support for Programming Languages and Operating Systems. Newport Beach, CA, USA,2011:305-318.
[11] CORBET J, RUBINI A, KROAH G. Linux Device Drivers (third edition)[M]. USA: O'Reilly,2006:35-45.
[12] BOVET D P, CESATI M. Understanding the Linux Kernel (third edition)[M]. USA: O'Reilly,2007:525-534.
[13] GALLAGHER K B, LYLE J R. Using program slicing in software maintenance[J]. Transactions on Software Engineering, 1991,17(18):751-761.
[14] WEISER M. Program slicing[C]// ICSE 1981. 1981:439-499.
[15] REN X, SHAH F, TIP F, et al. Chianti: A tool for change impact analysis of Java programs[C]// OOPSLA'04. Vancouver, BC, Canada,2004:432-448.
[16] ANDERSEN J, LAWALL J L. Generic patch inference[J]. Automated Software Engineering,2010,17(2):119-148.
[17] MENG N, KIM M, MCKINLEY K S. LASE: locating and applying systematic edits by learning from examples[C]// ICSE 2013. 2013:502-511.