

硕士学位论文

(学术学位论文)

面向质量评估的多粒度变更影响分析方法  
研究

**CRESEARCH ON MULTI-GRAINED  
CHANGE IMPACT ANALYSIS METHOD  
FOR QUALITY EVALUATION**

李美娜

哈尔滨工业大学

2025年1月

国内图书分类号：TP311  
国际图书分类号：004.02

学校代码：10213  
密级：公开

## 硕士学位论文

# 面向质量评估的多粒度变更影响分析方法 研究

硕士研究生：李美娜  
导 师：苏小红教授  
申 请 学 位：工学硕士  
学 科 或 类 别：软件工程  
所 在 单 位：计算学部  
答 辩 日 期：2025 年 1 月  
授 予 学 位 单 位：哈尔滨工业大学

Classified Index: TP311

U.D.C: 004.02

## Dissertation for the Master's Degree

# **CRESEARCH ON MULTI-GRAINED CHANGE IMPACT ANALYSIS METHOD FOR QUALITY EVALUATION**

<b>Candidate:</b>	Li Meina
<b>Supervisor:</b>	Prof. Su Xiaohong
<b>Academic Degree Applied for:</b>	Master of Engineering
<b>Specialty:</b>	Software Engineering
<b>Affiliation:</b>	Faculty of Computer
<b>Date of Defence:</b>	December, 2017
<b>Degree-Conferring-Institution:</b>	Harbin Institute of Technology

## 摘要

随着软件系统规模和复杂性不断增加，传统依赖经验丰富的专家进行人工代码审查的方法，已难以满足日益复杂的需求。由于代码审查的主要目的是确保代码质量符合项目要求，因此，自动化代码质量评估方法的重要性愈加凸显。特别是在大型软件系统中，随着开发团队的不断扩大和系统复杂度的提升，代码结构往往在漫长的维护过程中逐渐退化，模块化设计逐步被复杂的代码结构所取代，导致系统的维护和管理变得愈加困难。在软件系统的维护过程中，代码变更几乎是不可避免的，而这些变更可能对现有代码的质量和系统整体结构产生深远的负面影响，从而加剧系统的维护难度。

本文面向代码质量评估，深入研究代码变更影响分析方法，并结合质量评估度量，通过代码审查图的方式展示 C/C++ 项目的质量评估结果，方便开发人员从宏观的角度理解软件架构和代码质量。

首先，本文基于代码中间表示对软件质量度量指标进行计算，基于 clang 提取软件项目代码的抽象语法树，进一步从抽象语法树中提取方法摘要表和全局变量信息表。这些代码中间表示蕴含了代码调用等互相依赖的特征，基于这些中间表示，本文从内聚度、耦合性、代码复杂度和代码缺陷四个角度，提取了相关的代码质量评估度量和信息。这些度量不仅帮助开发人员深入洞察了代码质量，也为后续的代码优化和维护决策提供了优化角度。

其次，本文进一步研究了代码变更影响分析方法，以帮助开发人员更好地了解软件项目在维护过程中可能出现的相互影响的情况。变更影响分析方法能够预测代码变更可能带来的影响，帮助开发人员做出更加合理的设计和优化决策，减少变更带来的风险。本文首先实现了基于传统依赖关系闭包的变更影响分析方法，随后提出了三种新的变更影响分析方法，分别基于代码克隆、数据挖掘和深度学习技术。实验结果表明，这三种新方法均在不同的角度优于传统的依赖关系闭包方法，并能够弥补传统方法的不足之处。

最后，为了便于开发人员和审查人员从宏观角度全面了解软件项目的架构和代码质量，本文将代码质量分析结果通过代码审查图的形式展示给用户，此外，还生成了详细的代码质量检测报告。研究表明，代码审查图能够帮助开发人员快速聚焦于特定开发模块，使其在不需要掌握过多代码上下文的情况下，便能了解代码的整体架构。同时，代码质量检测报告以清单形式展示了项目中

各项质量度量，为开发人员提供了清晰的项目质量状况概览。

**关键词：**代码审查；代码质量评估；变更影响分析；代码度量

## Abstract

As the scale and complexity of software systems continue to increase, the traditional method of relying on experienced experts for manual code reviews has become inadequate to handle the increasingly complex code review demands. As a result, the importance of automated code quality assessment methods has become more prominent. In large software systems, as development teams expand and system complexity increases, the code structure often deteriorates over time during maintenance, with modular design being gradually replaced by complex code structures. This leads to greater difficulty in maintaining and managing the system. Furthermore, code changes are almost inevitable during the maintenance of software systems, and these changes can have a profound impact on the quality of the existing code and the overall system structure, thus exacerbating the difficulty of system maintenance.

This paper focuses on code quality assessment, conducting in-depth research on change impact analysis methods, and combines quality assessment metrics to present the quality evaluation results of C/C++ projects in the form of code review graphs. This approach allows developers to better understand the software architecture and code quality from a macro perspective.

First, this paper calculates software quality metrics based on intermediate code representations, extracting the abstract syntax tree of software projects using Clang, and further extracting method summary tables and global variable information tables from the abstract syntax tree. These intermediate code representations capture dependencies such as code calls, and based on these representations, this paper extracts relevant code quality metrics and information from the perspectives of cohesion, coupling, code complexity, and code defects. These metrics provide developers with deep insights into code quality and offer optimization directions for subsequent code optimization and maintenance decisions.

Second, this paper further explores change impact analysis methods to help developers better understand the potential interdependencies that may arise during the maintenance of software projects. Change impact analysis methods can predict the possible effects of code changes, helping developers make more reasonable design and optimization decisions, thereby reducing the risks brought by changes. The paper first implements

the traditional change impact analysis method based on dependency closure, and then proposes three new change impact analysis methods based on code cloning, data mining, and deep learning techniques. Experimental results show that these three new methods outperform the traditional dependency closure approach and effectively address its shortcomings.

Finally, to help developers and reviewers gain a comprehensive understanding of the software project's architecture from a macro perspective, this paper presents the results of code quality analysis in the form of code review graphs. Additionally, detailed code quality inspection reports are generated. The study shows that code review graphs can help developers quickly focus on specific development modules, enabling them to understand the overall code architecture without needing to grasp excessive code context. Meanwhile, the code quality inspection report presents project quality indicators in a checklist format, providing developers with a clear overview of the project's quality status.

**Keywords:** code review, code quality assessment, change impact analysis, Code Metrics

## 目 录

摘 要 .....	I
Abstract .....	III
第 1 章 绪论 .....	1
1.1 课题研究的背景和意义 .....	1
1.2 国内外研究现状及分析 .....	2
1.2.1 静态变更影响分析方法 .....	3
1.2.2 动态变更影响分析方法 .....	4
1.2.3 其他代码变更影响分析方法 .....	5
1.2.4 现有方法存在的问题与分析 .....	6
1.3 本文的主要研究内容以及各章节安排 .....	7
1.3.1 主要研究内容 .....	7
1.3.2 章节安排 .....	9
第 2 章 基于方法间关系和深度学习的代码变更影响分析 .....	10
2.1 引言 .....	10
2.2 代码预处理和中间表示生成 .....	10
2.2.1 基于 clang 的抽象语法树生成 .....	10
2.2.2 方法调用链提取与分析 .....	11
2.2.3 全局变量定义-使用链提取与分析 .....	14
2.3 基于依赖关系的变更影响分析 .....	15
2.4 基于克隆关系的变更影响分析 .....	17
2.4.1 代码预处理 .....	17
2.4.2 基于代码克隆检测的变更影响关系提取 .....	19
2.5 基于变更历史和共现关联关系的变更影响分析 .....	20
2.5.1 代码变更历史提取 .....	20
2.5.2 基于共现关联挖掘的变更影响关系提取 .....	21
2.6 基于深度学习的变更影响分析 .....	23
2.6.1 研究动机 .....	23
2.6.2 数据集来源和数据清洗 .....	23

---

2.6.3 基于代码预训练模型的变更影响关系预测 .....	24
2.7 实验结果与分析 .....	25
2.7.1 实验数据与评价方式.....	25
2.7.2 实验设置与实验过程.....	27
2.7.3 实验结果与对比分析.....	28
2.8 本章小结.....	33
<b>第3章 基于RAG的方法间变更影响分析 .....</b>	<b>34</b>
3.1 引言 .....	34
3.2 结合依赖路径的关系推理 .....	34
3.3 基于RAG的变更影响分析方法.....	36
3.3.1 研究方案 .....	36
3.3.2 数据和嵌入器准备 .....	37
3.3.3 检索模块 .....	37
3.3.4 推理增强 .....	38
3.4 实验结果与分析 .....	39
3.4.1 实验数据与评价方式.....	39
3.4.2 实验设置 .....	40
3.4.3 实验结果与对比分析.....	40
3.5 本章小结.....	44
<b>第4章 基于代码审查图的代码结构可视化和质量评估 .....</b>	<b>45</b>
4.1 引言 .....	45
4.2 基于代码度量的代码质量度量模型 .....	46
4.2.1 代码质量度量模型.....	46
4.2.2 基于内聚度缺乏度和连通性的的内聚性度量 .....	47
4.2.3 方法间耦合性度量 .....	50
4.2.4 方法扇入扇出度量 .....	51
4.2.5 基于静态检测工具的潜在缺陷度量 .....	52
4.3 代码审查图的构建和代码结构的可视化 .....	53
4.3.1 代码审查图构建 .....	53
4.3.2 代码审查图可视化 .....	55
4.4 基于代码审查图的代码质量分析 .....	56
4.5 实验结果与分析 .....	59
4.5.1 实验设计和实验数据 .....	59

4.5.2 实验环境 .....	60
4.5.3 实验结果分析 .....	60
4.6 代码审查图的应用案例分析 .....	63
4.7 本章小结.....	68
结 论 .....	69
参考文献 .....	71
哈尔滨工业大学学位论文原创性声明和使用权限 .....	75
致 谢 .....	76

# 第1章 绪论

## 1.1 课题研究的背景和意义

在软件的生命周期中，持续的代码维护是保证系统长期稳定发展的关键环节。研究表明，软件系统的维护成本在长期的项目预算中占据了 60% 至 80% 的比例<sup>[1]</sup>。维护过程中，开发人员不仅要对现有代码进行修改，还要确保新增功能或修复的缺陷不会影响系统原有的稳定性和性能。为了确保软件质量，维护工作通常依赖于系统的回归测试和代码审查。具体来说，标准的代码开发流程通常包括以下三个主要步骤，如图 1-1 所示。

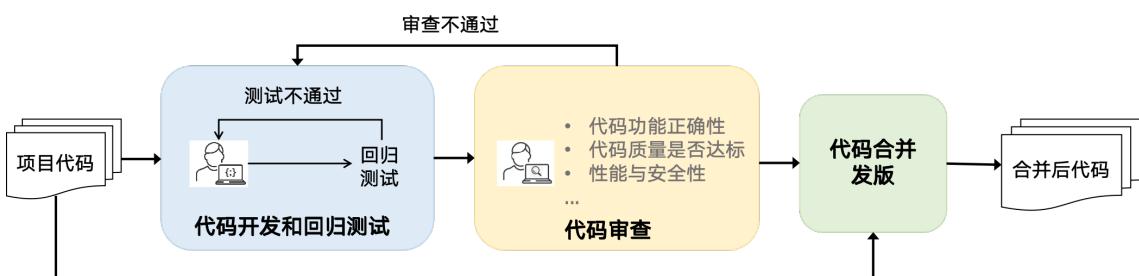


图 1-1 标准代码开发流程

(1) 代码开发与回归测试：开发人员在对项目代码进行修改后，首先对修改部分进行回归测试，验证新功能是否符合需求并避免新缺陷的引入。

(2) 代码审查：当测试通过后，代码将提交给审查人员进行审查。代码审查不仅仅是对代码逻辑正确性的检查，更是一个确保代码质量的重要环节。通过代码审查可以识别潜在的错误，提出代码优化建议，并统一开发团队的编码风格，从而提高代码的可维护性和可靠性。

(3) 代码合并：审查通过后，修改的代码可以与原始项目进行合并，进入下一个开发周期。在此过程中，确保代码的正确性和稳定性是至关重要的，避免因合并而引入新的问题。

不论在哪一个环节，代码质量都是最重要的主题之一。而代码质量评估则是确保软件项目高效、可维护和可扩展的基础手段。随着软件系统的规模和复杂性的不断增长，代码质量直接影响到系统的稳定性、可维护性以及开发过程中的效率。尤其对于遗留系统等大型系统来讲，复杂庞大的结构和长期维护导致的系统架构腐化会让开发者在进行软件维护的时候非常困扰，常常有“牵一发而动全身”的效应，担心代码变更对系统功能的潜在不良影响。

变更影响分析（Change Impact Analysis, CIA）作为一种有效方法，能够在代码变更发生时预测变更可能带来的影响，帮助开发人员更好地理解变更对其他模块或代码的潜在影响，从而做出更加合理的设计和优化决策。因此，从代码变更影响分析的角度出发，不仅能深入理解代码变更对系统整体和各个子模块的潜在影响，还能够提高维护项目的效率<sup>[2]</sup>。而现有的变更影响分析方法通常只关注基于静态依赖关系的影响，然而软件系统中最难以被用户察觉到，也是最容易导致功能逻辑上变更问题的是逻辑上的变更影响关系，正是因为这一关系，才导致大型系统的变更如此棘手，因此如何挖掘这类关系，分析这类关系导致的质量问题成为亟待解决的难题。

在大型软件系统中，开发者在阅读代码、掌握系统架构及模块间的关系时，常常缺乏一种与项目结构深度结合的直观展示方式。通常，开发者只能依赖于肉眼阅读代码或通过开发工具生成的类图等工具进行分析，然而这些方法所提供的信息往往是碎片化的，难以呈现系统整体的架构和模块间的依赖关系。此外，在进行代码变更时，开发者也面临着难以全面、准确地评估变更影响范围的问题，尤其是在大型系统中，单一变更可能对多个模块产生深远影响。现有的分析工具虽然能揭示代码结构或潜在问题，但难以提供系统化的、直观的质量评估。

本文从代码变更影响分析入手，提出了基于代码预训练模型和基于检索增强生成的变更影响分析方法，弥补传统方法只关注依赖型影响的不足，帮助开发者更全面、安全地进行代码维护工作。同时提出了代码审查图的软件结构和质量信息可视化方式，将分析结果结合软件代码架构直观地展示给用户，帮助开发者和审查人员更直观地理解整个项目的质量情况和代码结构，优化项目的持续维护过程，提高代码维护过程的安全性和效率。

## 1.2 国内外研究现状及分析

变更影响分析（Change Impact Analysis, CIA）是软件工程中用于分析代码变更对系统其他部分可能产生的影响的一种技术。研究表明，代码变更对代码质量的影响在大规模软件中尤为显著。Wenchen 等人的研究指出<sup>[3]</sup>，在大型系统中，每个版本的补丁可能影响约 2% 的代码，这对全面的程序回归测试提出了巨大挑战。因此，针对受变更影响的代码区域进行精确测试则是一种既高效又安全的测试方案。此方法不仅能够有效降低时间和资源成本，还能在不牺牲系统质量的前提下，减少因回归测试覆盖不足而引发的潜在漏洞或故障风险。

这一策略强调了将代码变更范围与回归测试策略精细化结合的重要性，为提升软件开发和维护阶段的代码质量提供了有力支持。

自 Arnold 等人<sup>[4]</sup>提出变更影响分析的概念以来，它一直是代码审查的重要组成部分之一。该方法支持用户在代码变更之前对其影响进行分析，从而估计变更可能造成的负面影响，其优势可总结如下：(1) 能提升代码的稳定性和可靠性。通过分析代码变更对相关模块的影响，开发者可以识别潜在的故障或不一致之处，在变更合入前发现问题，避免引入新的缺陷，从而提高系统的稳定性和可靠性。(2) 有助于模块化设计和低耦合。变更影响分析能够反映出代码模块之间的依赖关系和耦合程度，通过减少不必要的耦合，增强代码的模块化特性，从而提升代码的可维护性和可扩展性。(3) 有助于代码质量的提高。通过变更影响分析能够记录变更过程中的风险评估和解决措施，满足质量保证和审查的需求，提高软件开发过程的透明性和可追踪性。(4) 有助于开发团队协作。变更影响分析为团队提供了清晰的变更范围和影响信息，便于团队成员之间协调工作，减少因沟通不足导致的重复工作或冲突，同时提高代码可读性，有助于开发团队更快速地理解系统，减少后期维护成本。

变更影响分析方法主要可以分为两部分，分别是静态变更影响分析方法和动态变更影响分析方法，除此之外，还有一些不专注于软件代码的方法。

### 1.2.1 静态变更影响分析方法

静态分析方法因其具有高覆盖率和高安全性的优势，广泛应用于对安全性要求较高的软件回归测试中。这类方法通过基于程序的中间表示（如控制流图、调用图等）来进行分析。在静态分析方法中，过程内分析方法通常依赖于程序切片、控制流和数据流等技术<sup>[5-6]</sup> 分析语句级别的变更影响关系，而过程间的影响分析则主要通过调用图或系统依赖图进行分析，揭示不同方法之间的依赖关系<sup>[7-9]</sup>。

Schrettner 等人<sup>[10]</sup>提出了一种创新的静态执行后关系（Static Execute After, SEA）方法，这是一种计算高效且足够精确的程序关系，表示代码之间的运行顺序紧密相连的情况，可以作为变更影响分析的基础。SEA 的提出为程序分析提供了新的视角，其实验结果表明，通过 SEA 计算得到的影响关系能够发现大量的实际影响关系，显著提高了静态分析的准确性和实用性。

Sun 等人<sup>[11]</sup>提出了将变更类型与影响机制相结合的变更影响分析方法。他们认为不同类型的软件变更通常会带来不同的影响机制，因此需要根据变更的具体类型来制定相应的影响分析策略。此外，他们还指出，影响关系的精确度

与初始影响关系的精确度密切相关，初始影响关系越精确，基于其计算得到的最终影响关系也会更为准确。这一发现为改进影响分析技术提供了新的思路，强调了初始数据质量对最终分析结果的重要性。

在代码变更影响的研究中，基于图的分析方法是常见的手段之一。Ufuktepe 等人<sup>[12]</sup>针对方法间的依赖关系，利用方法调用图和影响图，提出了一种基于马尔可夫链的变更预测方法。他们通过前向切片信息计算变更后的影响概率，实验结果表明，调用图的精确率更高，而影响图则在少数情况下有更高的召回率。

Liang 等人<sup>[13]</sup>提出了一种影响集的概念，该影响集由包、类、方法、语句和变量五个层级的影响元素组合而成。通过在这五个层级上逐层搜索受影响的节点，并对这些节点进行层次化整理，最终构建出完整的层级影响结构。在此基础上，他们利用变更集定位子语句级依赖关系图中的变更节点，从多个层次分析代码变更的影响范围。

Peng 等人<sup>[2]</sup>针对传统 C 程序影响分析工具仅能在方法粒度上进行分析的局限性，研究了一种基于语句粒度的变更影响分析方法。该方法首先对源代码文件进行编译，提取全局信息以生成控制流图和调用图等结构。随后，将变更代码解析为不同类型变量的组合，并结合程序结构和变量变更信息进行深入的影响分析，从而更精确地评估代码变更带来的影响范围。

近年来，一些研究者尝试利用深度学习模型来研究代码变更的影响分析。Zhang 等人<sup>[14]</sup>提出了一种方法，通过构建更精细的系统依赖图并使用程序切片技术定位变更代码的影响区域，将这些区域提取为变更影响图。随后，他们采用 GCN（图卷积网络）从变更影响图中提取上下文信息，并将其与抽象语法树中的语法变更数据结合起来，最终实现了对不同版本维护类型的识别。

随着软件系统复杂度的增加，单一的变更影响分析方法已经难以满足高效性和准确性的双重需求。因此，研究人员提出了混合变更影响分析技术，通过将多种 CIA 方法结合起来，以提高变更影响分析的准确性和健壮性<sup>[15]</sup>。混合 CIA 技术的核心思想是将不同方法的优势互补，从而弥补单一方法的不足。研究表明，结合至少两种 CIA 技术的混合策略能够显著提高性能，且相比于基线技术，混合 CIA 方法始终表现出更好的性能改进。这一进展为变更影响分析提供了新的解决方案，尤其适用于大型复杂系统的影响分析任务。

### 1.2.2 动态变更影响分析方法

尽管静态影响分析在软件工程中因其较高的覆盖率和较好的安全性而被广泛应用，但其分析结果往往存在精确性不足的问题。这是因为静态分析主要

依赖程序的中间表示（如控制流图、调用图等）进行推理，而这些模型无法捕捉程序在运行过程中可能出现的实际行为。为了弥补这一不足，一些研究者转向动态影响分析。与静态分析不同，动态影响分析是在程序运行时收集实际执行信息，并基于这些运行时数据计算程序中各个部分的影响关系。尽管动态分析通常能够提供更为精确的结果，但其成本较高，而且在面对复杂的系统时，无法确保分析结果的完全安全性。

尤其是在面向对象编程的系统中，由于程序实体之间的依赖关系较为复杂且难以静态建模，动态分析的结果有时会产生不精确性的影响。为了提高动态分析的精确性，Huang 等人<sup>[16]</sup>提出了一种专门针对面向对象程序的精确动态变更影响分析方法。该方法结合了面向对象编程的特性，能够更加准确地确定程序实体之间的实际影响关系。同时，Huang 等人通过排除与变更对象无关的程序部分，显著减少了分析的规模，从而提升了分析的效率和精度。

在动态变更影响分析的精确性和可靠性方面，Cai 等人<sup>[17-18]</sup>进行了深入研究。他们提出了一种实验方法，首先通过敏感性分析来评估变更影响分析的准确性，然后通过实施软件变更并观察这些变更的实际影响，进一步分析其精确度和召回率。这一方法为动态影响分析技术的有效性提供了重要的实证依据，并揭示了在实际应用中可能遇到的挑战。此外，Cai 等人还提出了针对分布式系统的动态影响分析方法——DISTIA<sup>[19]</sup>。该方法通过对分布式系统中各个执行事件进行部分排序，并根据这些排序推断事件之间的因果关系，同时结合消息传递的语义预测影响在不同进程边界内外的传播情况，有效地解决了分布式系统中的影响传播问题，为分布式软件的动态影响分析提供了新的技术路径。

Wang 等人<sup>[20]</sup>通过词法分析器和语法分析器对源代码进行处理，将其转化为令牌和解析树，并为抽象语法树的每个节点设置预定义的权重。他们结合动态分析技术，收集代码运行时数据，并研究抽象语法树与运行时信息之间的关系，得出关联分析结果。在此基础上，构建代码影响图，并对其进行深入分析，研究代码变更如何通过影响图传播，从而得出扩散分析的结论。同时，他们追踪代码变更在抽象语法树中的传播路径，评估其可能带来的影响范围。

### 1.2.3 其他代码变更影响分析方法

除了上述静态和动态分析方法外，另一些研究并未直接关注软件本身，而是将关注点转向软件变更的历史库<sup>[8,21-24]</sup>。这些研究认为，软件变更历史记录中包含了大量与程序及其演化相关的信息，分析和挖掘这些信息能够帮助识别和预测变更对软件系统的潜在影响。这些依赖关系和变更模式可以通过数据挖

掘方法、信息检索技术以及机器学习等手段进行挖掘和分析，从而为变更影响分析提供新的视角和方法。

Gethers 等人<sup>[8]</sup>采用了信息检索、动态分析和数据挖掘方法，基于历史源代码提交记录改进了变更影响方法的生成技术。通过分析过去的源代码提交，研究者能够更好地识别变更和其他程序部分之间的潜在依赖关系，从而生成更加精确的变更影响集。这一方法突出了历史数据的重要性，利用现有的变更历史信息来为未来的变更影响分析提供依据，从而提升了分析的准确性和效率。

Zanjani 等人<sup>[23]</sup>提出了一种结合交互历史和提交历史的方法来分析源代码变更请求。他们的创新之处在于将信息检索、机器学习和轻量级源代码分析相结合，通过构建源代码实体的语料库，来提高变更影响分析的精确度。当给定一个变更请求的文本描述时，该语料库可以被查询，并返回一个按相关性排序的最可能发生变更的源代码实体列表。这种方法能够通过历史变更请求的文本描述，准确预测哪些源代码实体可能受到影响，为开发人员提供有效的决策支持。

Rolfsnes 等人<sup>[24]</sup>致力于改进现有的耦合分析算法，尤其是在软件变更的上下文中。TARMAQ 算法是他们提出的一种新型算法，在性能上明显优于 ROSE<sup>[25]</sup> 和 SVD 等传统算法。TARMAQ 通过挖掘代码库中的耦合关系，能够更精确地揭示源代码之间的依赖和关联，从而提高了变更影响集的生成效率和准确性。

Huang 等人<sup>[26]</sup>则提出了一种增强型方法，通过将历史变更模式映射到当前的变更影响分析任务中，解决了跨项目场景中的变更影响分析问题。在许多实际应用中，变更影响分析不仅仅局限于单一项目，而是需要跨项目、跨系统进行分析。Huang 等人的方法通过借助历史变更模式，能够在不同项目间共享和迁移影响分析的知识，进而提升了变更影响分析的普适性和适应性。

#### 1.2.4 现有方法存在的问题与分析

静态变更影响分析方法主要通过分析软件项目各个模块的依赖关系和耦合性来提取变更影响关系。尽管该方法在某些情况下有效，但其误报率较高，导致在实际应用中可能产生较多的不准确结果。此外，动态变更影响分析方法虽然能够提供更为准确的分析结果，但其实现成本较高，且难以确保软件项目的安全性，同时由于动态运行的局限性，也存在一定程度的漏报问题。无论是静态还是动态方法，其本质上都是通过显式的模块间依赖关系来进行分析。然而，除了这些显式的依赖关系之外，仍然存在大量逻辑上的变更影响关系未被充分挖掘和揭示，这些关系往往对用户能否安全变更至关重要。

另一方面，面向变更历史的方法侧重于分析提交信息与变更代码之间的关联性，并基于这种关联性来预测新的提交可能影响的范围。这种方法分析的是自然语言与代码之间的关系，非常依赖提交信息的质量。然而，由于提交信息的质量往往难以保证，导致该方法在实际应用中可能产生较高的错误率。此外，该方法也无法应用到没有变更历史或没有提交信息的项目之上，具有一定的局限性。

现有的软件项目代码质量分析方法大多数依赖于计算度量和各种工具套件所提供的指标，这些指标虽然能够在一定程度上反映代码质量，但往往缺乏与软件项目结构的深度结合。因此，用户对软件架构的理解仍然较为抽象，缺乏直观的表现形式。

### 1.3 本文的主要研究内容以及各章节安排

#### 1.3.1 主要研究内容

本文面向 C/C++ 软件项目进行代码质量评估，旨在帮助用户在软件生命周期的各个环节全面、直观地了解软件的代码质量，了解代码各个部分的变更影响关系，帮助用户更安全地对软件代码进行维护。主要研究内容分为三个部分：代码中间表示与质量评估度量提取、面向代码质量评估的变更影响分析方法研究和代码审查图生成。

##### (1) 代码中间表示与质量评估度量提取

代码中间表示是一种介于源代码和机器代码之间的抽象表示形式，通常用于程序分析、优化和转换等环节。通过构建合理的中间表示，可以有效地抽象出代码的结构和行为，为质量评估提供准确的依据。本文将代码转换成抽象语法树（Abstract Syntax Tree，AST），并且基于 AST 提取方法定义-使用链和全局变量定义-使用链，分别整理为方法摘要表和全局变量信息表。

方法摘要表和全局变量信息表为代码质量度量的计算提供了简化的、结构化的信息，使得各种质量度量的提取变得更加高效和准确。通过中间表示，代码的复杂结构和行为可以被清晰地捕捉，进而为质量度量提供更为精确的计算基础。代码质量评估度量是用于量化软件质量的各种指标。这些度量可以评估代码的模块化、可维护性、复杂度等多方面的特征。本文基于提取到的方法摘要表提取代码模块的内聚度、耦合度和复杂度相关的度量，用于评估代码的质量。

##### (2) 面向代码质量评估的变更影响分析方法研究

本文实现了传统的静态分析方法，并设计了三种新的变更影响分析方法。基于静态分析方法根据方法摘要表和全局变量信息表计算依赖传递闭包得到变更影响关系。除此之外设计了基于克隆代码的检测方法，克隆代码指的是开发者通过复制和粘贴已有的代码片段，来创建功能类似的代码段。这种代码通常在功能上与原代码重复，因此当代码有变更的时候，这样的代码会被影响。对于有代码变更历史的软件项目，可以根据代码变更历史，通过数据挖掘的方式挖掘频繁共同更改的代码对，认为其之间存在代码变更影响关系。对于缺失代码变更历史的软件项目，将数据挖掘的到的代码对整合为数据集，训练深度学习模型，对变更影响关系进行预测。本文这四种方法结合起来，提取代码中的变更影响关系，评估其对软件质量的影响。

### (3) 基于代码审查图的代码架构和质量信息可视化

为帮助开发者全面了解软件项目的整体架构、模块化情况以及经过深入分析后的代码质量，本文提出了一种基于代码审查图的结果展示方式。代码审查图将整个软件项目的结构、质量度量和变更影响等信息可视化，便于开发者更直观地理解项目的各个方面，并做出相应的优化决策。

代码审查图包括多个重要组成部分：首先是代码质量度量，这涵盖了代码的复杂度、可维护性、重复性等方面指标，能够有效反映项目的质量状况；其次是变更影响关系，该部分通过分析软件变更对其他模块和功能的影响，帮助开发者预测和规避潜在的风险；最后是模块标签，这基于软件项目的实际模块结构，通过使用大语言模型进行预测和分析，从而为开发者提供对软件模块化质量的客观评价。

在代码审查图中，节点代表项目中的关键元素，如方法和全局变量；而边则表示不同节点之间的关系，包括依赖关系、耦合关系和变更影响关系。这些边反映了不同模块或方法之间的交互和依赖，能够揭示出系统架构的潜在问题和优化点。

为了提供更加清晰的视图，代码审查图的可视化工作通过图可视化引擎 G6 完成，G6 引擎能够高效地渲染和展示复杂的图结构，支持交互式查看和深入分析，帮助开发者快速识别问题所在。此外，所有提取和计算得到的质量评估结果还会以代码质量评估报告的形式进行详细总结，报告中将完整展示各项质量指标、分析结果和建议，确保开发者能够全面掌握软件项目的质量状态，从而进行有效的改进与优化。

### 1.3.2 章节安排

本文的章节安排如图 1-2。

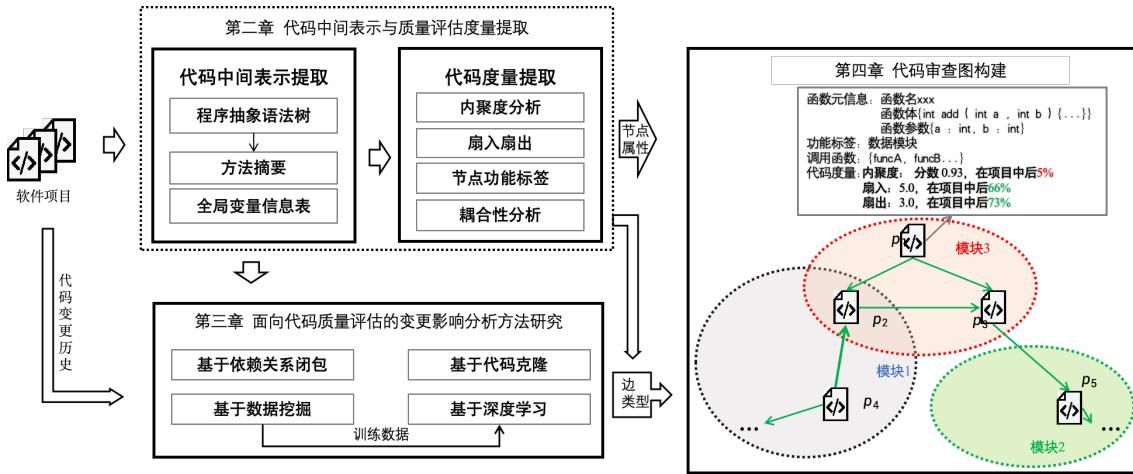


图 1-2 章节安排

第一章为绪论，首先介绍了本文的研究背景和研究现状，首先介绍了软件质量和代码变更影响的背景和意义，然后介绍了代码质量研究主题的国内外研究现状，介绍了代码质量度量套件的研究现状，并分别从静态和动态两类分析方法总结了变更影响分析的研究现状。然后介绍了本文的主要研究内容和章节安排。

第二章介绍了代码中间表示与质量评估度量提取。本章介绍了基于 clang 的抽象语法树生成方法，并且基于 AST 提取方法调用链和全局变量定义-使用链。基于提取的中间表示和特征，本章介绍了代码度量的提取，并进行了实验结果展示和分析。

第三章介绍了面向代码质量评估的变更影响分析方法研究。分别介绍了基于依赖关系闭包的方法、基于代码克隆的方法、基于数据挖掘的方法和基于深度学习的方法。最后进行了实验结果展示和分析。

第四章介绍了代码审查图的生成。首先介绍了利用大语言模型生成模块标签，用于分析代码的模块化质量。其次介绍了代码审查图的构建方法，并通过图可视化引擎 G6 对图进行可视化，最后进行了实验结果展示和分析。

## 第2章 基于方法间关系和深度学习的代码变更影响分析

### 2.1 引言

软件变更是软件维护的核心环节。对软件系统的修改可能引发系统其他部分的不良副作用或连锁反应。而变更影响分析的目标在于识别变更的涟漪效应，帮助开发者安全地进行变更。方法之间的变更影响可以分为以下两种类型：

(1) 依赖型变更影响关系：依赖型指的是能直接体现在代码静态结构中的变更影响关系，如将项目代码组织为抽象语法树、系统依赖图或影响图，通过图结构的可达性分析，得到的静态依赖关系。方法间的依赖型变更影响关系体现为表现为方法间的调用或间接调用关系。

(2) 逻辑型变更影响关系：在这种类型的变更影响关系中，方法之间不存在静态结构之间的关联关系，但它们的实现逻辑或所操作的数据之间存在某种隐含的关联。具体来说，这种关系可能源于它们共同维护某个数据的一致性、共享某些资源，或其功能逻辑有某种预期联系。因此，当某个方法发生变化时，可能会间接影响到其他方法的行为或结果。

依赖型影响关系可在代码的静态结构中直接显现，通过基于依赖关系的传统影响分析方法甚至开发工具即可捕捉，而逻辑型影响关系通常只能通过开发者人工进行分析提取，正是因为逻辑型影响的存在，导致了开发者对软件代码的维护非常困扰，难以理解软件架构、难以安全变更的问题出现。

为了解决上述问题，本文以 C/C++ 项目为研究对象，实现了基于依赖关系、基于克隆关系以及基于变更历史和共现关联关系的变更影响分析方法，提出了基于深度学习的变更影响分析方法，并通过实验验证深度学习方法的有效性，对比了四种方法各自的优点和局限性。

### 2.2 代码预处理和中间表示生成

#### 2.2.1 基于 clang 的抽象语法树生成

抽象语法树（Abstracted Syntax Tree, AST）把代码的语法结构以树形的方式进行了抽象化描述。在这个树形结构中，每一个节点都对应着代码中的某个

元素，比如变量声明、语句或者是表达式等。从抽象语法树的根节点出发，代码逐步被拆解成更小的部分，直到最终到达叶节点，这些叶节点代表了代码中最基本的元素，如操作符或变量等。AST 能够清晰地展示出代码的层次和结构，为编译器或其他工具分析和处理代码提供便利。

Clang 是由苹果公司发起的支持 C、C++、Objective-C 和 Objective-C++ 语言的编译器前端，负责对代码进行词法分析、语法分析和语义分析，对程序代码的分析和理解至关重要<sup>[27]</sup>。而 libclang 是 Clang 编译器的一个重要组成部分，它提供了一套用于解析源代码的程序接口。这些程序接口允许开发者在项目中使用 Clang 的强大语言解析和代码分析功能<sup>[28]</sup>。本文使用 libclang 生成 AST，提取代码中的调用和依赖关系，为后续进一步分析提供基础。

在 libclang 解析得到的抽象语法树中，游标（cursor）是一个核心概念，它作为一个指针或引用存在，每个 cursor 都与 AST 中的一个特定节点相对应，表示了源代码中的一个结构元素。通过操作 cursor，可以遍历整个 AST，访问和分析代码中的各种元素，如获取变量的类型、方法的参数列表、类的成员等。libclang 提供了一系列 API 函数来操作 cursor，例如：遍历 AST 中的 cursor、获取 cursor 的类型（如是否为方法定义、变量定义、变量引用等）、获取 cursor 所代表的源代码元素的名称、类型、位置等信息、获取 cursor 的父节点或子节点等。本文通过操作游标，遍历 AST，获取整个 AST 的结构。

表 2-1 重要 AST 节点类型标识

节点标识	含义
Translation_Unit	一个翻译单元
Function_Decl	方法定义
Parm_Decl	方法的参数定义
Var_Decl	变量定义
Devl_Ref_Expr	变量引用
Call_Expr	方法调用

clang 定义了一系列的节点类型，我们将部分重要节点类型列在表 2-1 中。值得注意的是，在 clang 中是不区分方法声明和方法定义的，统一用 Function\_Decl 来标识，两个区分主要看是否有方法体，在 libclang 中提供了程序接口供开发者调用判断。

## 2.2.2 方法调用链提取与分析

在使用 libclang 提取代码的抽象语法树后，遍历整棵树来提取方法之间的调用关系。这部分我们重点关注抽象语法树上的方法节点，以及方法节点内部

的调用节点，分别对应着代码中方法的定义和方法内部对其他方法的调用。

对抽象语法树的遍历主要分为两次，第一次遍历的目的是获取所有的方法定义。首先提取所有的 Function\_Decl 节点，它表示方法的定义，在该节点中可提取方法签名。在 Function\_Decl 节点下，提取子节点 Parm\_Decl，该节点表示方法的参数列表，在该节点中可提取参数名称和参数类型等参数相关信息。然后提取 Function\_Decl 节点的子节点 VarDecl，该节点表示在该方法内定义的局部变量。在对方法进行分析时，我们本身不关心方法的内部实现，但是由于在 C/C++ 语言中，存在局部变量可以和全局变量重名的情况，在这里提取方法内定义的局部变量，方便后续在提取全局变量的使用时，排出同名局部变量的影响。除此之外，还需提取整个方法的 token 序列，所在文件以及作用域。

第二次遍历的目的是提取方法之间的调用关系。提取 Function\_Decl 节点的子节点 Call\_Expr，该节点标签表示的是调用语句，可提取调用的方法名。注意，由于主要分析该项目中由开发者定义的方法之间的依赖关系，所以对于一些标准库方法的调用选择忽略，不进行提取。具体的提取流程如算法 2-1 所示。

分析结束后，将会获得每个方法的方法调用关系和详细信息，将提取到的信息组织为一个方法摘要表，表的每一项表示一个方法的摘要，每个摘要由  $\langle funcID, token, params, call, scope, file, localvar \rangle$  共 7 部分组成，分别表示方法的唯一 ID 标识，方法体，方法参数列表。方法内调用的其他方法，方法的作用域，方法所在模块和方法定义的局部变量。

## 算法 2-1 方法调用链提取

**Input:** 项目中的所有代码文件: *files*

**Output:** 方法摘要表: *functions*

```

1 Function scanAndAnalyze(files) :
2     functions ← {} # 初始化方法摘要 ;
3     # 第一次扫描: 收集方法的定义 ;
4     foreach file ∈ files do
5         cursor ← libclang.parse(file).cursor # 获取 AST 的根 cursor ;
6         traverse(cursor, 0, functions, file, True) # 遍历
7             AST, 收集方法定义 ;
8     end
9     # 第二次扫描: 分析方法调用情况 ;
10    foreach file ∈ files do
11        cursor ← libclang.parse(file).cursor # 获取 AST 的根 cursor ;
12        traverse(cursor, 0, functions, file, False) # 分析
13            方法调用 ;
14    end
15    return functions ;
16
17 Function traverse(node, depth, functions, filePath, isFirstScan) :
18     if isFirstScan then
19         if node.kind == CursorKind.FUNCTION_DECL then
20             function ← collectionInfo(node) # C 收集方法信息 ;
21             functions.add(function) # 将方法添加到方法摘要 ;
22         end
23     end
24     else if node.kind == CursorKind.CALL_EXPR then
25         parse(node) # 分析被调用的方法 ;
26     end
27     foreach n ∈ node.get_children() do
28         traverse(n, depth + 1, functions, filePath,
29             isFirstScan) # 递归遍历子节点 ;
30     end

```

### 2.2.3 全局变量定义-使用链提取与分析

在 C/C++ 代码中，相同描述符修饰下的全局变量的定义、作用域、生命周期和方法是同级别的，所以在本文中，将全局变量也作为独立的代码单元进行分析。全局变量定义-引用链的提取和方法的定义和调用提取类似，对 AST 的遍历主要也分为两次。具体流程如图 2-3。

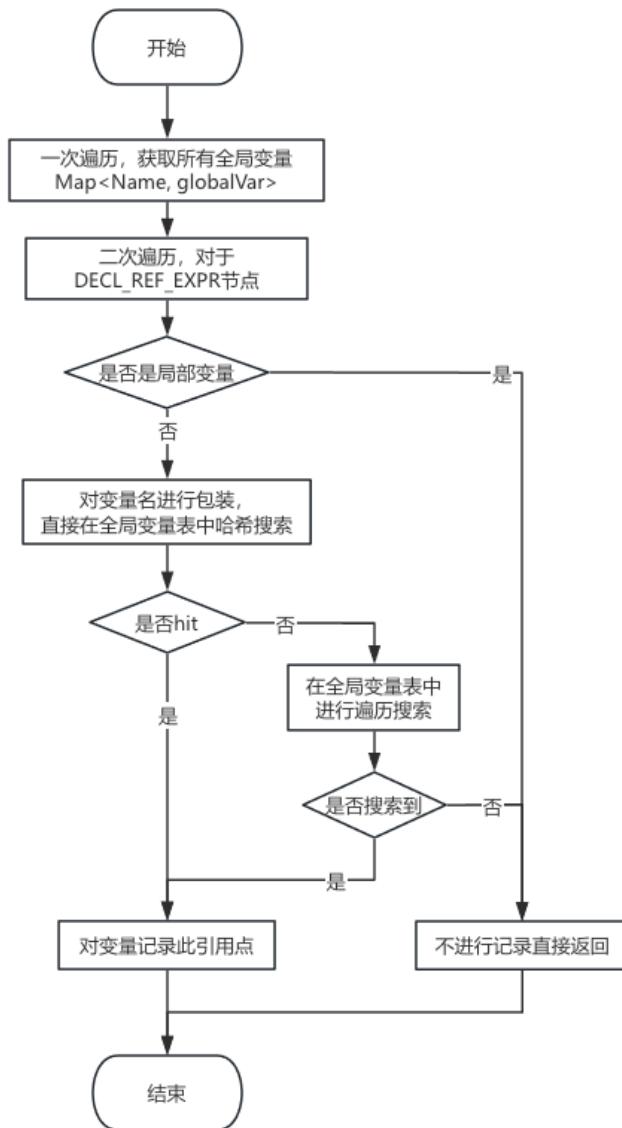


图 2-1 全局变量提取流程图

第一次遍历获取所有的全局定义。首先提取所有的 Var\_Decl 节点，它表示变量定义，然后提取节点中的变量名和变量类型。注意，由于在 AST 中的节点标签中无法区分变量是否是全局的，所以这里根据节点在 AST 中的深度来判断是否是全局变量，并且在变量名前加上针对该项目文件的绝对路径，来保证变

量名的唯一性。在确定其为全局变量后，还需进一步提取该变量的作用域。在 C/C++ 语言中，static 关键字可用于修饰变量和方法，意味着该变量或该方法只能在其所在文件内使用，而不是全局可用，因此需要对其作用域进行判断。一次遍历提取到的结果是一个全局变量表，这里使用哈希表 Map<Name, globalVar> 的数据结构进行存储，方便对全局变量进行查找。

第二次遍历的主要目的是提取全局变量的引用点。在方法节点子树中搜索 Devl\_Ref\_Expr 节点，该类型节点表示对变量的引用，这里首先判断被引用的变量是否是局部变量，根据方法摘要表中该方法的相关信息可以判断，如果是则直接返回，因为我们不关心方法内的局部变量引用。如果不是，则证明使用的是全局变量，首先在哈希表中进行查找该变量名，以节省检索时长，如果找到了，说明是在该文件中定义的全局变量，同时能够保证被 static 修饰的全局变量的判断的准确性。如果没有查找到，则说明引用了别的文件中定义的全局方法，则在哈希表中进行遍历查找，记录该全局变量被引用的方法。具体提取流程如图 2-3 所示。

分析结束后，将全局变量的信息组织为全局变量信息表，表的每一项表示一个全局变量的信息，每条信息由  $\langle globalVarID, type, use, scope, file \rangle$  共 5 部分组成，分别表示全局变量的唯一 ID 标识，变量类型，变量的引用点所在的方法，变量的作用域以及变量所在模块。

### 2.3 基于依赖关系的变更影响分析

依赖关系传递闭包方法是一种基于静态依赖关系的技术手段，通过识别代码模块间的关联性划定受变更影响的范围<sup>[15]</sup>。其核心思想是利用依赖关系的传递性，通过构建和分析依赖图，揭示所有可能受到影响的代码模块或单元。

**构建依赖图** 以抽象语法树、全局变量信息表和方法摘要表为基础，构建程序的系统依赖图。图节点代表代码中的基本元素，本文中是方法和全局变量，而边则表示这些元素之间的依赖关系。依赖关系包括方法调用和全局变量引用，在全局变量信息表和方法摘要表中可直接提取依赖关系。生成边的原则如下：

- 调用边 (call)：方法间的调用关系。如果方法 A 调用了方法 B，在图中增加一条从节点 A 指向节点 B 的有向边。
- 引用边 (use)：方法和全局变量的引用关系。如果方法 A 引用的全局变量 C，在图中增加一条从节点 A 指向节点 C 的有向边。

通过这种方式，依赖关系图不仅能够系统地表示代码中各个元素之间的直接依赖关系，还能为后续的变更影响分析提供结构化的图形模型。如图 2-4 为依赖图示例，该图中共有 6 个方法和 3 个全局变量，其静态依赖关系如图中的边所示。

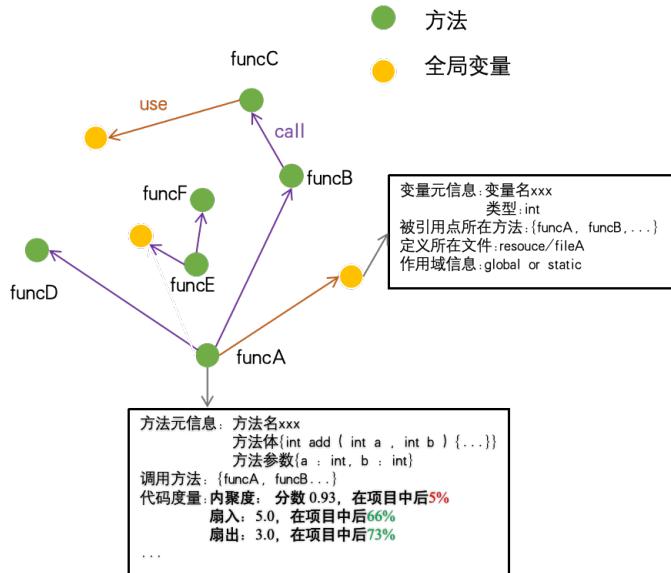


图 2-2 依赖图示例

**执行变更影响分析** 这一过程旨在确定哪些方法在代码变更时可能受到影响，以及这些影响的传播路径。本文基于方法和方法之间的以下两种关系 **RBM** (relationships between methods) 对每一个方法识别当其变更时受影响的方法集 **IMS** (impacted method set)，定义如式 (2-1) 所示，以方法  $f$  和方法  $g$  的关系为例，**CALL** 方法和 **RETURN** 分别代表  $f$  和  $g$  的调用和被调用关系。

$$\begin{aligned}
 RBM &= CALL \cup RETURN \text{ where} \\
 (f, g) \in CALL &\iff f \text{ (transitively) calls } g, \\
 (f, g) \in RETURN &\iff f \text{ (transitively) returns into } g
 \end{aligned} \tag{2-1}$$

对于依赖关系图中的每个节点，计算该节点的传递闭包。传递闭包是指从某个特定节点出发，根据上文定义的依赖关系和图的可达性，可以直接或间接到达的所有节点的集合，反映了节点之间的依赖链以及影响传播的范围。传递闭包的具体迭代模型如式 2-2。

$$\begin{aligned}
 IMS^{(N)} &= IMS_{CALL}^{(N)} \cup IMS_{RETURN}^{(N)} \\
 IMS_{RETURN}^{(N+1)} &= \bigcup_{define \in (IMS_{RETURN}^{(N)} - IMS_{RETURN}^{(N-1)})} IMS(define) \\
 IMS_{CALL}^{(N+1)} &= \bigcup_{define \in (IMS_{CALL}^{(N)})} IMS(define), define \in \\
 &\quad (IMS_{CALL}^{(N)} - IMS_{CALL}^{(N-1)})
 \end{aligned} \tag{2-2}$$

其中  $N$  表示第  $N$  轮迭代，第  $N+1$  轮的迭代受  $N$  和  $N-1$  轮的影响，反映出软件系统中的变更的涟漪效应。为了高效地计算传递闭包，使用广度优先搜索遍历图中的各个节点及其依赖边，进而识别出所有直接或间接依赖于某个节点的其他节点。每次从某个节点出发时，都会跟踪并记录通过依赖关系可到达的所有节点，最终得到的节点集合中，所有的节点都与初始变更的节点存在某种直接或间接的依赖关系。这些方法可以视为受变更影响的范围，意味着它们在该方法变更后，可能会因为依赖关系的传递而受到影响。通过这一分析，我们不仅可以识别出受影响的直接方法，还能揭示出那些通过多次间接依赖而受到影响的方法，帮助开发者全面了解变更的潜在影响范围。

在图 2-1 的例子中，方法 funcA 调用了 funcB 和 funcD，funcB 调用了 funcC。在对 funcB 进行变更影响分析时，会直接影响到 funcA 和 funcC，根据依赖关系闭包，会间接影响到 funcD。所以与 funcB 有变更影响关系的方法集合为 {funcA, funcC, funcD}。

## 2.4 基于克隆关系的变更影响分析

代码克隆（Code Clone）是指在代码中存在两段或多段内容相似或完全相同的代码片段。因此它们在逻辑上往往具有相同的功能或行为。如果对其中一个克隆片段进行了变更（例如修复了一个 bug、添加功能或进行优化），那么在其他地方相同或相似的代码也可能需要同步修改，否则可能会导致系统的不一致性或错误，这是典型的逻辑型变更影响关系。

因此，本文基于方法间的克隆关系进行变更影响分析。该方法主要分为两步，首先对源程序进行预处理，通过代码分段及代码指纹提取的方式对源程序进行编码，生成代码序列数据库。随后利用频繁模式挖掘算法 ClaSP 得到克隆代码列表，具体的处理流程如图 3-2 所示。

### 2.4.1 代码预处理

#### 词法分析

- 去除注释：注释通常用于解释代码的意图，并不直接影响程序的执行，但

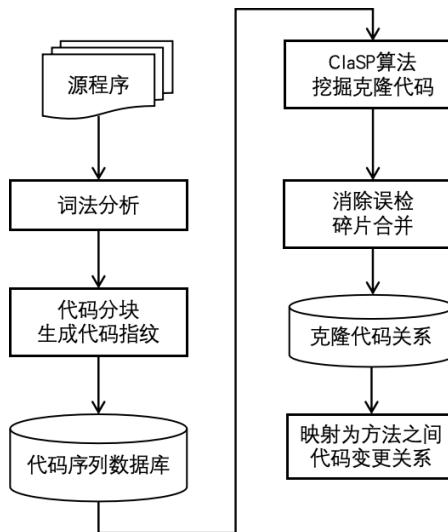


图 2-3 基于代码克隆的变更影响分析方法流程

不同的代码实现中注释内容可能存在差异，这会导致本质相同的代码片段由于注释的不同而被误判为非克隆。

- 去除头文件引用语句：源代码文件往往包含多个头文件，而不同的源文件可能引用相同的头文件。如果不对头文件进行统一的处理，算法就可能在不同的源文件中检测头文件引用的代码克隆情况，可能影响克隆检测的效率和准确性。
- 程序标准化：为了避免因变量名的变化导致漏检，在方法内部对变量名进行统一标准化处理。

**代码分块** 这一步将代码拆解成更小、更易于对比的单元，从而提高克隆检测的准确性。本文的代码分块策略按代码结构的不同分为几类，

- 顺序结构：按固定行数分块，行数可由用户定义，默认为 6 行一块。行数越小则识别结果越精准，越能识别更细小的代码克隆情况。
- 控制结构：将选择结构（if、then、else、endif、switch）、循环结构（while、for）、和域结构（{、}）共同描述为控制结构识别并根据关键分块词进行分块。这是由于控制语句是代码逻辑的重要分界点，将它们作为分块的标准可以确保检测系统能聚焦于实际功能的逻辑边界。

值得注意的是，分块时大括号不被视为代码块的一部分。不同的开发者在代码排版上可能存在差异，例如有的开发者将大括号置于同行，而另一些则习惯将大括号另起一行。为了避免这种格式差异对克隆检测结果产生干扰，大括号被排除在代码块之外。

**代码指纹提取** 遍历代码块的每一行语句，将每行语句转化为数字序列，再将所有数字序列合并，转化为代码块的“指纹”。该指纹将代表代码片段用于识别代码片段之间的相似性或重复性。鉴于哈希算法在计算上的高效性与实现的简便性，它在生成代码指纹方面具有显著的优势。因此，本文选择采用冲突率较低的 hashpjw 算法提取代码指纹。

#### 2.4.2 基于代码克隆检测的变更影响关系提取

序列数据挖掘 (Sequence Data Mining, SDM) 是时序数据挖掘领域的一个重要研究方向，旨在从给定的输入数据库中，探索在大量对象之间随时间频繁出现的模式。判断一个模式是否具有意义的阈值被称为最小支持度。本文中，将代码指纹片段作为序列，合并起来为序列数据库，利用序列数据挖掘算法，挖掘频繁出现的模式，将问题转化为闭合序列挖掘问题。这里对于数据挖掘的一些基本概念不再赘述，具体可参考文献<sup>[29]</sup>。

**闭合序列挖掘——检测克隆代码片段** 主要分为两步，首先生成频繁序列，作为频繁闭合序列的候选 FCC (Frequent Closed Candidates)。第二步执行剪枝，从候选中剔除所有非闭合的序列，最终得到精确的 FCS (Frequent Closed Set)。主要的算法流程如 3-1 所示。

---

##### 算法 2-2 ClaSP 算法

---

**Input:** 序列数据库

**Output:** 频繁闭合序列集 FCS

```

1  $F_1 \leftarrow \{\text{频繁 1-序列}\}$ 
2  $FCC \leftarrow \emptyset, FCS \leftarrow \emptyset$ 
3 for all  $i \in F_1$  do
4    $F_{ie} \leftarrow \{\text{频繁 1-序列的长大于 } i \text{ 的扩展序列}\}$ 
5    $FCC_i \leftarrow \text{DFS-Pruning}(i, F_1, F_{ie})$ 
6    $FCC \leftarrow FCC \cup FCC_i$ 
7 end
8  $FCS \leftarrow \text{N-ClosedStep}(FCC)$ 

```

---

首先找到所有的频繁的 1-序列（即长度为 1 的序列），然后，对于所有频繁的 1 序列，递归地探索相应的子树。对所有频率为 1 的序列进行此处理，得到 FCC，最后，算法结束去除 FCC 中出现的非闭合序列。

DFS-Pruning 算法的通过递归生成候选模式（包括 s-扩展和 i-扩展，分别在模式末尾和任意位置添加新元素）并检查其支持度，最终返回以当前模式  $p$  为前缀的所有频繁模式集。剪枝时，通过检查对应模式的子序列和超序列的支持度，将序列的节点进行合并，防止继续遍历冗余节点。最终得到的 FCS 即为所有克隆代码集。

**合并碎片——提取方法间的变更影响关系** 由于先前的代码分段处理导致克隆代码呈现为片段间的克隆关系，为了恢复代码的完整性，进一步对这些碎片进行合并。基于每段代码的位置信息，将属于同一方法的碎片进行合理整合，从而重建方法间的克隆关系。通过这种方式，最终得到的是方法与方法之间的克隆关系，反映了不同方法之间在代码修改过程中的潜在影响，即方法间的变更影响关系。

## 2.5 基于变更历史和共现关联关系的变更影响分析

在软件工程中，分析代码变更历史是理解软件演化重要手段之一。在开发者对项目进行维护的过程中，通常是以一个提交（commit）为单位进行功能上的变更。当进入新的维护工作时，如对同一功能进行升级等，通常的做法是参考前人的开发历史，对当前开发工作做指导，以防止变更的不完全。基于这一特点，本文实现了基于变更历史和共现关联关系的变更影响分析方法，分析对象是软件项目的变更历史。该方法能够提取蕴含在代码变更历史中的变更影响关系，尤其适用于具有丰富变更记录的软件项目。

该方法的核心原理是在代码变更历史中，频繁同时更改的代码片段，通常存在着某种潜在的变更影响关系。这种变更影响关系不仅仅局限于静态结构上的依赖，还包括功能上的耦合和实现上的相互作用。因此，通过对这些历史变更数据的深入挖掘和分析，我们可以揭示出更丰富的逻辑型变更关系。

### 2.5.1 代码变更历史提取

由于 Git 是现代软件开发中最广泛使用的版本控制工具，因此，本文的分析主要基于由 Git 进行版本管理的项目。

**收集项目代码库及变更历史记录** 克隆项目的代码库到本地，通过 git log 命令获取所有的 commit，包括每个提交的哈希值 commitHash 等信息。

**提取每个提交的变更信息** 对每个提交运行 `git show <commitHash>` 命令，查看该提交引入的代码变更（即“diff”或差异），这会显示哪些文件被修改、添加或删除，本文主要关注标记为“修改”的文件，这些修改的文件中包含了具体的代码变化，即代码行的增、删、改操作，记录该 commit 引起的所有发生变更的代码行。

**定位变更代码行所属的方法** 定位变化代码行所在的方法。通过 libclang 分析变化前文件得到的抽象语法树可获取每个方法对应的代码行，与变更的代码行位置进行匹配，得到变更的代码行所在的方法。

**提取变更方法与提交的关系** 对于每个提交，提取出所有受影响的方法（即发生变化的方法），并将这些方法构成一个变更方法列表，用 `Map<commitID, List<Methods>` 的结构存储每个 commit 变更的方法，作为序列数据库，便于后续分析与处理。

### 2.5.2 基于共现关联挖掘的变更影响关系提取

基于关联规则（Association Rules）的数据挖掘方法是反映事物之间相互依存性和关联性的一个重要数据挖掘技术，旨在从大量数据中挖掘出有价值的项之间的相关关系。共现关系可以视为关联规则的一种表现形式，它描述了在给定集合中，某一组项（或特征）经常出现在同一事务中。例如，在零售分析中，常见的共现关系是“购买了面包的顾客通常也会购买牛奶”。在这种情况下，“面包”和“牛奶”是一对共现项。

在本文的代码变更影响分析中，共现关系描述的是在一次提交中，哪些方法经常同时发生变更。如果两个方法在多个提交中频繁一起变动，则它们之间可能存在某种依赖关系或变更影响关系。

常用的频繁项集的评估标准有支持度和置信度。支持度表示共现项在数据集中出现的次数占总数据集的比重，用于衡量一组项在数据集中的普遍程度。在代码变更分析中，支持度表示某一方法对在多个提交中同时出现的频率，计算公式如 3-3.

$$Support(funcA, funcB) = \frac{num(AB\text{共现})}{num(AllCommits)} \quad (2-3)$$

置信度表示共现项中一个出现后，另一个项出现的概率。变更分析中，置

信度度量表示当方法 A 被修改时，方法 B 被修改的概率，计算公式如 3-4。

$$\text{Confidence}(\text{funcA} \Leftarrow \text{funcB}) = \frac{P(\text{AB 共现})}{P(\text{B 出现})} \quad (2-4)$$

在前文提取的序列数据库中，进行如下算法 3-3 的计算。

---

算法 2-3 变更影响方法对挖掘算法

---

**Input:** 序列数据库  $D$ , 支持度阈值  $\text{min\_sup}$ , 置信度阈值  $\text{min\_conf}$

**Output:** 变更影响方法对  $\text{change\_impact\_pairs}$

```

1  $F_1 \leftarrow \emptyset$  # 频繁 1 项集
2 for all  $f \in D$  do
3   if  $\text{support}(f) \geq \text{min\_sup}$  then
4      $F_1 \leftarrow F_1 \cup f$ 
5   end
6 end
7  $F_2 \leftarrow \emptyset$  # 频繁 2 项集
8 for all  $(f_1, f_2) \in P = \{(f_i, f_j) \mid f_i, f_j \in F_1, i \neq j\}$  do
9   if  $\text{Support}(f_1, f_2) \geq \text{min\_sup}$  then
10     $F_2 \leftarrow F_2 \cup (f_1, f_2)$ 
11   end
12 end
13  $\text{change\_impact\_pairs} \leftarrow \emptyset$ 
14 for all  $(f_1, f_2) \in F_2$  do
15   if  $\text{confidence}(f_1, f_2) \geq \text{min\_conf}$  then
16      $\text{change\_impact\_pairs} \leftarrow \text{change\_impact\_pairs} \cup (f_1, f_2)$ 
17   end
18 end
19 return  $\text{change\_impact\_pairs}$ 

```

---

计算得到的即为那些频繁同时变更的方法对，表明它们在变更过程中有着较为显著的相互依赖关系，反映了方法间的变更影响关系。

## 2.6 基于深度学习的变更影响分析

### 2.6.1 研究动机

前文中所述的方法均有一定的局限性。

- 冷启动问题：当项目代码拥有丰富的变更历史时，可以通过前文中介绍的数据挖掘方法，提取具有变更影响关系的方法对，然而并不是所有软件项目都有变更历史可供我们分析。
- 影响类型覆盖不全：在仅有项目源代码的情况下，数据挖掘方法无法直接应用，只使用基于依赖闭包和基于克隆代码的方法，则无法识别除这两种关系外的逻辑型变更影响。
- 影响模式难以迁移：数据挖掘方法仅能得到变更过的影响关系，对于未变更的方法，无法将相同的影响模式进行迁移。

因此本文提出基于深度学习的变更影响分析方法，通过训练深度学习模型来进行变更影响关系的预测，从而解决上述问题。

### 2.6.2 数据集来源和数据清洗

**数据集来源** 为了保证深度学习方法能够识别依赖型和逻辑型两种影响关系，我们以数据挖掘方法得到的影响关系作为数据集训练深度学习模型。

将数据挖掘方法的置信度设为 1，支持度设为 2 在示例项目中进行检测，检测到有变更影响关系的方法对作为数据集的正例。为了构建平衡的数据集，通过对项目中的方法进行随机采样，从中选取一些不具有变更影响关系的方法对，作为数据集的负例。

**数据清洗** 置信度为 1 表示方法严格共现，但是支持度为 2 并不是十分严格的挖掘标准，因此数据集中会存在一些由于偶然共现导致的误报现象。为了排除此种样例，本文结合大语言模型对数据集进行清洗。

大语言模型拥有较强的代码语义理解能力，能够对真实的变更影响关系进行精准识别。通过提示变更影响关系的类型以及对应的含义，能够帮助大语言模型对方法间的影响关系进行推理，从而排除掉误报样例。

### 数据清洗推理 prompt

你是一个优秀的代码变更影响关系检测师，能够根据用户给定的方法代码对，判断二者之间是否有变更影响关系。方法间的代码变更影响关系指的是一个方法变化（包括签名或内部逻辑的变化），导致其他方法需要随之更改的现象。

方法之间的变更影响主要分为两类，一类是静态依赖型的，表现为方法间的调用或间接调用关系。另一类则是逻辑型的，表现为方法的实现逻辑或所操作的数据之间存在某种隐含的关联，如共同维护某个数据的一致性、共享某些资源，或功能逻辑有某种预期联系等。

请你对用户给定的一对方法进行判断，判断二者之间是否有变更影响关系。

下面为用户问询的一对方法，

方法 1: ...

方法 2: ...

请给出你的判断，如果有则回答 1，没有则回答 0，不需要给出解释。

### 2.6.3 基于代码预训练模型的变更影响关系预测

基于深度学习的影响关系预测任务的输入为两个方法体，输出为两个方法之间存在变更影响关系的概率，概率越接近 1，表明这两个方法之间有关系的可能性越大。

考虑到代码理解的复杂性与深度，本文采用了基于预训练模型的代码表示学习方法，选择了 CodeBERT 作为编码器提取代码的表示。CodeBERT 是一种专为程序代码设计的预训练语言模型，通过大规模的代码语料库预训练，能够学到代码中的丰富语法结构和语义信息。经过 CodeBERT 模型的表示学习，所得到的向量不仅包含了每个方法体的语法特征，还能够编码代码中的语义关系及其他潜在的编程特征。模型架构如图 3-3 所示。

首先，将两个方法体  $F_a, F_b$ ，先通过代码预训练模型进行编码

$$H_a = \text{Encoder}(F_a) \in \mathbb{R}^{(len, dim)} \quad (2-5)$$

$$E_a = \text{mean}(H_a[1 : \dots]) \in \mathbb{R}^{(dim)} \quad (2-6)$$

得到方法的向量化表示  $E_a, E_b$ ，通过拼接融合这两个向量，将融合后的向

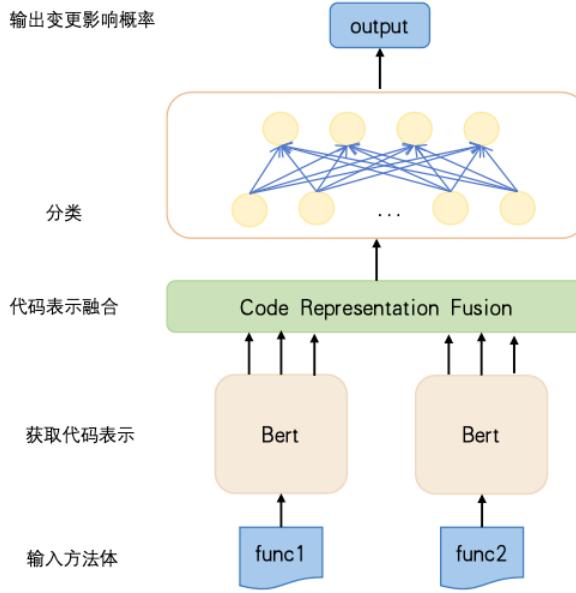


图 2-4 基于代码预训练的深度学习模型架构

量表示送入一个由两层组成的多层感知机（Multilayer Perceptron, MLP）中，再通过 softmax 层进行分类处理，将模型的输出转化为一个概率分布，表示两个方法体之间存在关系的概率。

$$E_{a,b} = \text{Concat}(E_a, E_b) \in \mathbb{R}^{(dim*2)} \quad (2-7)$$

$$\logits_{a,b} = \text{MLP}(E_{a,b}) = \text{FFN}(\text{ReLU}(\text{FFN}(E_{a,b}))) \quad (2-8)$$

$$\text{FFN}(x) = Wx + b \quad (2-9)$$

$$\text{ReLU}(x) = \max(0, x) \quad (2-10)$$

$$\logits_{a,b} \in \mathbb{R}^{(2)} \quad (2-11)$$

最后与真实标签计算交叉熵损失，得到 loss，计算梯度，优化模型参数

$$\text{loss} = \text{CrossEntryLoss}(\logits_{a,b}, \text{Label}) \quad (2-12)$$

## 2.7 实验结果与分析

### 2.7.1 实验数据与评价方式

**实验数据** 本文从影响力或社区活跃程度的角度出发，收集了表 2-2 中所示的软件项目为被测项目进行实验。这些项目在 `github` 上的收藏数均在千以上，说

明这些项目在开源社区中有着一定的影响力，使用范围比较广泛。除 antiword 项目之外，其他项目有着比较活跃的社区，说明其还在不断更新迭代过程中，所以能提供较为丰富的变更历史，以供数据挖掘方法的实验分析。

表 2-2 被测项目

项目名称	项目简介	代码行数	提交数	收藏数
TheAlgorithms	各种算法的开源实现，涵盖了计算机科学、数学和统计学等领域	24645	1536	57k
antiword-0.37	提取 Microsoft Word 文档内容的工具	34725	-	13k
jemalloc-5.3.0	通用的 malloc(3) 实现，强调碎片避免和可扩展的并发支持	83525	3530	9k
libbpf-1.1	linux 内核观测技术的一个脚手架库	127927	2375	1.9k
librdkafka-2.1.0	Apache Kafka 的 C/C++ 客户端库	154951	4430	18k
FFmpegKit-5.1.0	FFmpeg 工具包	450998	369	3.7k

根据每个示例项目的上一个版本和示例版本间的版本变更，分别各自选取 30 个变更方法，通过分析 commit 得到真实的被影响方法集 AIS (Actual Impact Set)，作为测试集，这里分别统计了依赖型和逻辑型的变更影响关系，得到的数据如表 2-3 所示。再分别通过前述方法进行检测，得到估计的被影响方法集 EIS (Estimated Impact Set)，通过评价指标评估方法的有效性。

表 2-3 被测项目数据统计

项目名称	commit 数	变更点数	依赖型 AIS	逻辑型 AIS	总计
TheAlgorithms		30	98	21	119
antiword-0.37	-	30	119	54	173
jemalloc-5.3.0		30	67	14	81
libbpf-1.1		30	194	17	211
librdkafka-2.1.0		30	92	26	118
FFmpegKit-5.1.0		30	105	17	122
总计		180	675	149	824

基于深度学习的变更影响分析方法的数据集收集方式如 3.5.1 节所述，为了保证测试集和训练集的不重叠性，在经过挖掘和清洗后得到的关系中排除上述测试变更点。得到的数据集统计信息如表 3-2 所示，共 7351 对数据，按照训练、验证、测试集为 6:2:2 进行训练和测试。这里排除了 antiword 项目，因为该项目并没有官方维护的 github 仓库，因此无法获得足够的历史变更。

**评价指标** 评价指标如式 (3-13) (3-14) (3-15) 所示，真实的被影响方法表示为 AIS (Actual Impact Set)，每种方法检测得到的结果为估计的被影响方法 EIS (Estimated Impact Set)，按逻辑型和依赖型对关系进行划分，根据这两个值计算

表 2-4 数据集统计信息

项目名称	正例对数	负例对数	总对数
TheAlgorithms	97	1000	1097
jemalloc-5.3.0	593	1000	1593
libbpf-1.1	401	1000	1401
librdkafka-2.1.0	932	1000	1932
FFmpegKit-5.1.0	103	1000	1103
总计	2126	5000	7126

精确度、召回率和 F-measure，这三种评价指标在信息检索的场景下被广泛使用，本章中用于评价变更影响分析方法的有效性。

$$precision = \frac{|EIS \cap AIS|}{|EIS|} \quad (2-13)$$

$$recall = \frac{|EIS \cap AIS|}{|AIS|} \quad (2-14)$$

$$F - measure = \frac{2 \times precision \times recall}{precision + recall} \quad (2-15)$$

## 2.7.2 实验设置与实验过程

### 实验设置

- 代码预训练模型选择：本文使用了 CodeBERTa-small-v1 和 codebert-base-mlm 两个模型分别作为代码表示学习模型，得到的代码表示为 768 维，融合时使用的 MLP 的每层维数为 (768\*2, 64, 2)。根据在数据集上的表现，选择表现较好的模型与其他方法做对比。
- 深度学习模型参数设置模型的参数设置如表 3-3。

表 2-5 模型参数设置

超参数	数值
Token embedding size	768
codeBERT learning-rate	1e-5
codeBERT dropout	0.4
Classifier learning-rate	1e-4
Adam $\beta_1$	0.95
Adam $\beta_2$	0.999
batch_size	64

- 基于共现关系方法置信度：设为 1。

- 基于共现关系方法支持度：设为 2 和 3 分别进行实验，选择表现较好的与其他方法做对比。

**实验过程** 基于深度学习的方法在数据集上的实验结果如表 3-5 所示，从总体上来看，两个模型的性能均表现出色，但模型 CodeBERTa-small-v1 的 F-measure 略高于 codebert-base-mlm，该模型更强调预测出的正例的真实性和准确性。这意味着它在确保预测结果的准确性方面做得很好。由于其良好的综合能力，选择 CodeBERTa-small-v1 模型作为主模型进行对比实验。

表 2-6 基于代码预训练模型的变更影响关系在数据集上的实验结果

模型	F-measure	recall	precision
CodeBERTa-small-v1	91.8	86.1	98.2
codebert-base-mlm	87.1	100.0	77.1

### 2.7.3 实验结果与对比分析

本节将通过实验对比来评估本章中提出的基于深度学习的变更影响分析方法的性能，这里主要讨论下列三个问题：

RQ1: 本章提出的基于深度学习的方法能否有效检测变更影响关系？与其他方法相比，它在精确率，召回率和 F-measure 上表现如何？

RQ2: 四种方法在提取依赖型（DB）和逻辑型（LB）的变更影响关系上各自的优势如何？尤其是对于逻辑型的影响关系是否具有实际意义？分别适用于哪些特殊场景？又各自有怎样的局限性？

#### 1. 针对于 RQ1 的实验

三种方法的实验结果如表 3-4 所示。

整体上讲，基于深度学习的方法表现最好，其 F-measure 在所有方法中表现最优，并且在查全和查准的能力上较为平衡。这说明基于深度学习的方法能够学习到过去变更历史中的行为模式，并能将学到的变更关系知识进行迁移，用于判断新的影响关系。

其次基于共现关系的方法也较为良好。这表明，代码变更历史中方法的共现关系的确蕴含了大量能够有效揭示变更影响关系的信息。这是因为变更历史中都是前人对软件项目进行变更的记录，这样的提交由开发者精确变更，并经历过开源项目中非常严格的审查过程才合入主分支，因此较为准确地反映了代码变更中的实际操作，从而也能将过去的开发模式反映在数据挖掘的结果集中。

表 2-7 变更影响实验结果 - F-measure/召回率/精确度

方法	F-measure	recall	precision
依赖关系	42.2	81.3	28.5
克隆关系	6.4	3.3	92.0
共现关系	62.1	68.7	56.6
深度学习	65.2	69.4	61.4

而另外两种方法表现则较为失衡。基于依赖关系闭包的方法表现为召回率较高而准确率很低，仅为 28.5%。这是由于依赖闭包方法本身的特性决定的，由于变更影响关系随涟漪扩散效应，越向外扩散影响越小，但该方法却平等地认为扩散所至的代码均存在影响关系，这并不符合涟漪效应的特性。如在图 3-4 中所示是 antiword 项目中从 bTranslateImage 方法出发得到的部分依赖图，它层层递进地展示了从 word 中提取 jpec 图片的过程，bTranslateImage 调用 bTranslateJPEG，处理 jpec 图片，再调用 vASCII85EncodeFile，将图片提取为文件，再依次调用 vASCII85EncodeArray 和 vASCII85EncodeByte。当对 Byte 方法进行变更影响分析时，根据 RETURN 关系的涟漪效应，最终会将图中所示的其他 4 个方法都列为影响集。然而实际上，该方法只对 {Array, File} 存在变更影响关系，最显然的，当 Byte 方法的签名发生改变时，将直接影响到 {Array, File}，这两个方法如果不更改将发生编译错误。而对另两种方法的影响则微乎其微，化为了动态运行时内部值的变化。



图 2-5 依赖闭包方法迭代路径

而基于克隆关系的检测方法则更为特殊，其精确率最高，这说明其识别的

正例非常准确，很少出现误报。但是它的召回率极低，表明该方法对大部分的变更影响关系无法识别到。这是由于该方法只能识别由于克隆关系产生的变更影响，而在质量良好的项目代码中，克隆代码的现象很少出现，因此提取到样例本身也较少，就导致其整体效果不佳。在实践中，建议基于克隆关系的方法作为其他方法的补充使用。

## 2. 针对于 RQ2 的实验

RQ1 中从整体的角度上说明了三种方法的有效性。为了回答 RQ2，这里对每种方法检测得到的依赖型和逻辑型的变更影响关系分别进行计算，得到如表的结果。通过对两种类型分别的统计，我们能更直观地发现不同方法的优势和特点。

表 2-8 变更影响实验结果 - F-measure/召回率/精确度

方法	DB-F-measure	DB-recall	DB-precision	LB-F-measure	LB-recall	LB-precision
依赖闭包	44.0	96.2	28.5	0	0	0
克隆代码	0	0	0	31.95	19.1	97.8
历史共现	60.3	65.9	55.6	69.3	81.7	60.2
深度学习	63.0	66.3	60.1	74.4	83.4	67.2

### 依赖关系闭包方法

- 优势：对于依赖型影响几乎没有漏报。召回率高达 96.2%，能查全大部分的依赖型影响关系。
- 局限性：依赖型影响关系的误报较高，导致依赖型的检测表现整体上较差。针对逻辑型的变更，该方法则无法检测到，这是由于逻辑型的影响关系无法在依赖图中产生联系，因此依赖闭包方法无法检测。

### 克隆关系方法

- 优势：逻辑型影响几乎没有误报，其准确率能达到 95%，说明其非常擅长挖掘逻辑型中由于克隆代码导致的变更影响关系。而这种关系在软件长期的维护中容易被忽略。

图 3-6 为克隆代码方法挖掘到的一对有变更影响关系的方法。这里展示了这对方法的部分代码，其中绿色高亮的部分表示代码克隆的区域。这两个方法的主要功能是分别对 8 位和 4 位压缩格式的图像进行解码。我们发现，这对方法中的大部分逻辑结构几乎完全相同，只有少部分关键处理逻辑存在差异。由此，我们可以认定，这两个方法的变化过程很可能是同步的，即在实际的维护过程

中，当对其中一个方法进行修改时，另一个方法也通常需要同步进行相应的变更，才能保证逻辑的一致性。

<pre>static void vDecodeRle4(FILE *pInFile, FILE *pOutFile, const imagedata_type *pImg) {     int iX, iY, iByte, iTmp, iRunLength, iRun;     BOOL bEOF, bEOL;      DBG_MSG("vDecodeRle4");      fail(pInFile == NULL);     fail(pOutFile == NULL);     fail(pImg == NULL);     fail(pImg-&gt;iColorsUsed &lt; 1    pImg-&gt;iColorsUsed &gt; 16);      DBG_DEC(pImg-&gt;iWidth);     DBG_DEC(pImg-&gt;iHeight);      bEOF = FALSE;      for (iY = 0; iY &lt; pImg-&gt;iHeight &amp;&amp; !bEOF; iY++) {         bEOL = FALSE;         iX = 0;         while (!bEOL) {             iRunLength = iNextByte(pInFile);             if (iRunLength == EOF) {                 vASCII85EncodeByte(pOutFile, EOF);                 return;             }             if (iRunLength != 0) {                 iByte = iNextByte(pInFile);                 if (iByte == EOF) {                     vASCII85EncodeByte(pOutFile, EOF);                     return;                 }                 for (iRun = 0; iRun &lt; iRunLength; iRun++) {                     if (odd(iRun)) { </pre>	<pre>static void vDecodeRle8(FILE *pInFile, FILE *pOutFile, const imagedata_type *pImg) {     int iX, iY, iByte, iRunLength, iRun;     BOOL bEOF, bEOL;      DBG_MSG("vDecodeRle8");      fail(pInFile == NULL);     fail(pOutFile == NULL);     fail(pImg == NULL);     fail(pImg-&gt;iColorsUsed &lt; 1    pImg-&gt;iColorsUsed &gt; 256);      DBG_DEC(pImg-&gt;iWidth);     DBG_DEC(pImg-&gt;iHeight);      bEOF = FALSE;      for (iY = 0; iY &lt; pImg-&gt;iHeight &amp;&amp; !bEOF; iY++) {         bEOL = FALSE;         iX = 0;         while (!bEOL) {             iRunLength = iNextByte(pInFile);             if (iRunLength == EOF) {                 vASCII85EncodeByte(pOutFile, EOF);                 return;             }             if (iRunLength != 0) {                 iByte = iNextByte(pInFile);                 if (iByte == EOF) {                     vASCII85EncodeByte(pOutFile, EOF);                     return;                 }                 for (iRun = 0; iRun &lt; iRunLength; iRun++) {                     if (iX &lt; pImg-&gt;iWidth) { </pre>
---	--

图 2-6 包含克隆代码片段的一组方法实例

这种现象表明，在软件的演化过程中，维护人员可能需要对这两个方法进行联动更新。任何对其中一个方法的修改都可能影响到另一个方法的功能或逻辑一致性，在代码维护时必须考虑它们之间的相互依赖关系。因此这种基于代码克隆的方法有着非常重要的实际价值。

- 局限性：仅能检测由于代码克隆导致的逻辑型影响，对于其他逻辑型和依赖型影响存在漏报。

## 数据挖掘方法

- 优势：可以挖掘两种类型的变更影响关系。对于依赖型影响，尽管有一定的漏报，但是误报较少，对比依赖闭包方法有着更强的实用价值。而对于逻辑型影响，其表现优于依赖闭包和克隆代码方法，说明其不仅仅只关注了克隆一种关系，还包括其他逻辑型的关系。

以 librdkafka 项目中挖掘到的一对方法为例，该项目是 Apache Kafka 的一个高性能 C/C++ 客户端库。图 3-7 左侧的方法 rd\_kafka\_global\_cnt\_decr 负责对计数器进行减一操作，而右侧的方法 rd\_kafka\_global\_cnt\_incr 则负责对计数器进行加一操作。

从功能上来看，这两个方法是一对典型的协作方法，其作用密切相关。具体来



图 2-7 逻辑上有变更影响关系的方法对示例-incr 和 decr

说，incr 方法执行计数器加一操作，在计数器从零变为一时，执行资源的初始化工作。而 decr 方法则在计数器减为零时，执行资源的释放操作。从代码实现可以看出，这两个方法的操作逻辑具有明显的互补性，加一与减一、初始化与释放的功能关系呈现出“镜像”特性。

尽管它们依赖图上无法产生联系，但由于它们在功能上承担了计数器的管理和资源的初始化与释放工作，其逻辑关联性非常强。因此，当其中一个方法的实现逻辑发生变化时，另一个方法通常需要进行相应的调整，以保持整体逻辑的一致性，这种方法对的变更影响关系属于典型的逻辑关联型变更影响关系。因此也有较强的实用价值。

- 局限性：

- 要求项目代码必须有变更历史。
- 挖掘算法的支持度对结果有影响。表是支持度分别为 2 和 3 对应的实验结果。可以发现，支持度越大，误报越少，但是漏报越多。这说明支持度越大越排除了一些变更共现的偶然因素，但是失去了对影响关系的普遍性检测，能检测出的关系更少，导致了漏报增加。

表 2-9 变更影响实验结果 - F-measure/召回率/精确度

方法	DB-F-measure	DB-recall	DB-precision	LB-F-measure	LB-recall	LB-precision
历史共现-2	60.3	65.9	55.6	69.3	81.7	60.2
历史共现-3	63.3	54.5	75.4	76.1	69.3	84.3

- 挖掘到的信息属于硬信息。仅依靠变更历史的只能得到变更过的方法之间影响，未变更过或变更次数较少的影响关系无法反映，相同的影响模式之间无法进行迁移。

## 深度学习方法

- 优势：可以挖掘两种类型的变更影响关系且效果较好。解决了没有变更

历史的冷启动问题，对于逻辑型影响也能够实现影响模式的迁移。

- 局限性：首先是计算性能问题。以 antiword 项目为例，方法共 578 个，测试变更点有 30 个，对每个变更点都需与项目所有方法一一计算就是  $30*(578-1)=17310$  次，均需要通过嵌入、分类过程。耗费较长的时间，导致实际可用性大打折扣。第二点是依赖路径信息缺失问题，在依赖型影响关系中，可分为直接依赖和间接依赖，表现为直接调用或通过调用链的间接连接关系。我们发现在深度学习方法在间接依赖的表现很差，这是由于直接依赖关系和逻辑型的关系在代码段中都可以直接体现，即使转化为向量，也保留了一定的特征。而间接依赖在方法段中无法体现，因此丢失了依赖信息，导致其效果不好。

表 2-10 深度学习方法在直接和间接依赖上的实验结果

方法	F-measure	recall	precision
直接依赖	高	高	高
间接依赖	低	低	高
逻辑依赖	高	高	高

上述方法为代码变更影响关系的检测提供了多样化的解决方案，但是各有侧重，适用于不同的应用场景，同时也具有一定的局限性。总结如表 3-7。

表 2-11 变更影响分析方法对比总结

方法	检测关系	优势	局限性
依赖闭包	依赖型	依赖型漏报低	依赖型误报高，且仅能检测依赖型
克隆代码	逻辑型中的代码克隆	逻辑型中克隆关系影响的误报低	只能检测代码克隆一种关系
数据挖掘	依赖型和逻辑型	两类影响的性能均较好	无法应用于没有变更历史的项目，不频繁变更无法被检测
深度学习	依赖型和逻辑型	两类影响的性能均较好	计算性能差，间接依赖丢失

## 2.8 本章小结

本章实现了基于依赖闭包、克隆代码、变更历史共现挖掘的变更影响分析方法，并提出了基于深度学习的影响分析方法。经过实验证明了基于深度学习方法的有效性，并通过分析四种方法在依赖型和逻辑型的表现，总结了四种方法各自的优点和局限性。

## 第3章 基于RAG的方法间变更影响分析

### 3.1 引言

检索增强生成 (Retrieval Augmented Generation, RAG) 技术由 Lewis 等人<sup>[30]</sup>于 2020 年提出，是一种将信息检索与生成式模型（如 GPT 等）相结合的技术，它可以在文本生成过程中有效利用外部知识库或数据库中的信息，以提高生成结果的准确性和上下文相关性。RAG 工作流程可以概括为两步，检索和生成。检索过程中根据输入问询 (query)，从外部知识库（如文档库、网页、数据库）中检索与输入相关的上下文或片段，生成阶段将检索到的信息与用户的输入结合起来，作为生成模型的输入，生成模型根据检索的上下文生成答案或内容。RAG 方法能有效解决端到端问答模型的效率问题和减少事实错误的发生。

由于深度学习模型优秀的泛化能力和知识迁移能力，基于深度学习的变更影响分析方法相较于基于方法间关系的方法具有更优秀的变更影响分析能力。但是由于其性能问题，导致其在实际使用时较为耗时。除此之外，由于深度学习方法仅检测两两方法之间的变更影响关系，因此仅靠方法块无法反映的间接依赖信息在检测时缺失了，导致其在间接依赖的影响关系判断上表现不好。

为了解决上述问题，本文提出结合依赖路径和检索增强生成 (Retrieval Augmented-Generation, RAG) 的变更影响分析方法，通过检索的方法提高效率，并结合了依赖路径，为检测过程提供依赖信息，最后通过实验验证该方法的有效性。

### 3.2 结合依赖路径的关系推理

在前文中提到，深度学习方法对于间接依赖导致的变更影响关系检测效果较差，这是由于深度学习方法判断的是方法两两之间的联系，而对于这对方法中间的依赖路径信息存在缺失导致的。如图是一个例子，这个例子中主要是由于参数传递的数据类型，比如在这个例子中，A 方法的局部变量传递给了 B 方法，又传递给了 C 方法，而 C 方法内又操作了该变量的值，当 C 方法内对于该值的操作逻辑有变更时，导致 A 方法对于该变量的操作也需跟着变更，因此这是一种基于间接依赖的变更影响关系。而在 AC 这两个方法体中，并没有 AC 之间的依赖特征，因此深度学习方法难以捕捉这两个方法的联系。

由此我们提出结合依赖路径的推理方法。主要分为两阶段，具体流程如图所示。

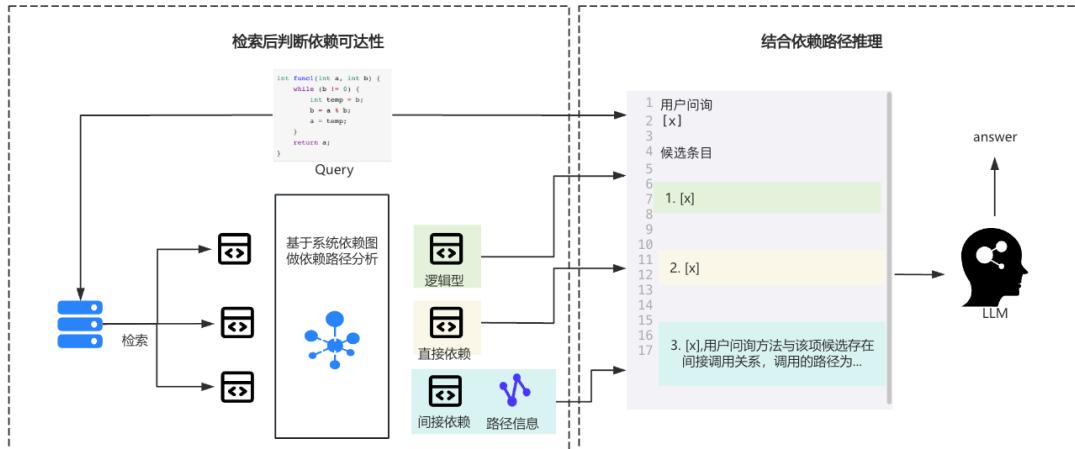


图 3-1 结合依赖路径的关系推理过程

**判断依赖可达性** 这阶段先判断用户问询方法和检索得到方法的类型，如果是逻辑型或直接依赖型，则直接进行推理。如果均不属于，则先判断其在依赖图中是否存在间接可达，如果不可达，则直接交由大模型推理。如果可达，则将方法对依赖路径上的所有方法都进行提取，作为判断的候选。

**结合依赖路径推理** 对于有间接依赖关系的方法对，将提取的依赖路径构造特殊候选提示信息。具体来讲，提取路径上的方法块，将其融入对应候选方法的提示信息中，提示大语言模型用户问询方法与候选方法存在这样一条依赖路径，补全这部分间接依赖信息，再对此候选方法进行推理。值得注意的是，依赖路径的两端分别是用户问询和当前候选方法，而中间路径上的节点可能是其他候选方法，因此我们把路径上的两端和出现过的候选方法只提示为签名，其他方法提示整个方法体，以此来减少提示中的冗余，推理的提示如下图所示。

### 推理 prompt

你是一个优秀的代码变更影响关系检测师，能够根据用户给定的方法代码，在候选条目中判断与用户给定的方法有变更影响关系的方法。方法间的代码变更影响关系指的是一个方法变化（包括签名或内部逻辑的变化），导致其他方法需要随之更改的现象。

方法之间的变更影响主要分为两类，一类是静态依赖型的，表现为方法间的调用或间接调用关系。另一类则是逻辑型的，表现为方法的实现逻辑或所操作的数据之间存在某种隐含的关联，如共同维护某个数据的一致性、共享某些资源，或功能逻辑有某种预期联系等。

请你在给定的候选条目中，一一判断其与用户给定的方法是否有变更影响关系。

下面为用户问询的方法

[x1]

下面是候选条目

1. [x2]

2. [x3]

3. [x4]，用户问询方法与该项候选存在间接调用关系，调用的路径为 [x4] → [x3] → [x1]，此项请将依赖调用的路径考虑进去，再进行判断。

请给出候选条目的判断，回答格式如下，不需要给出解释。

1. 有

2. 无

3. 有

...

通过提示间接依赖的信息，帮助大模型更好的判断方法间的变更影响关系，从而防止间接依赖型的漏报，达到更高的召回率和准确率。

## 3.3 基于 RAG 的变更影响分析方法

### 3.3.1 研究方案

为了解决第二章中深度学习方法的性能问题，本章提出了基于 RAG 的变更影响分析方法。图中展示了本章方法的研究框架。

【有个图】

本章中的方法主要分为三个阶段。

- 数据准备：将完整代码库按照函数的粒度进行切分并作为知识库；构建准确的变更影响关系方法三元组（查询，正例，负例），用于训练嵌入模型，使嵌入模型专精于检测变更影响关系  $zh$  垂直领域。
- 检索：将软件代码中的方法组成的集合作为知识库，经过嵌入模型生成向量表示得到向量数据库。被测方法嵌入得到用户向量，在数据库中检索得到候选方法。
- 增强生成：通过大语言模型对候选方法进行筛选，得到最终的有变更影响关系的方法。

### 3.3.2 数据和嵌入器准备

数据准备共有两部分。

**生成知识库** 生成知识库是为了方便检索。本方法检索的对象是方法，因此将软件代码按照方法进行分块最为合适。通过第二章中提到的代码预处理得到的抽象语法树，遍历其方法定义节点，得到每个方法的方法体，将方法体的集合作为知识库。

**编码器数据** 为了使编码器更专注于变更影响关系检测的任务，我们训练嵌入模型，使具有变更影响关系的方法在向量空间中更接近。将研究项目代码通过上章中的四个方法检测得到的有变更影响关系的方法对作为正例进行收集，经过数据清洗得到准确的正例对，并对正例对中任选其一在项目中随机采样得到对应的负例，得到  $\langle Q, Pos \rangle, \langle Q, Neg \rangle$  的数据。

编码器训练

考虑到对代码语义的理解能力，这里使用 codebert 模型作为编码器进行训练，对其进行微调。模型的架构如图所示。

【有图有公式】

### 3.3.3 检索模块

检索模块主要分为在线和离线两个部分。离线部分将生成的方法块通过编码器编码得到对应的向量，并将其存储在向量数据库中。在线部分则是将用户的输入通过编码器进行编码，在向量数据库中进行基于相似度匹配的检索。

在线部分中的向量检索的原理是将用户向量与向量数据库中的向量进行

相似度匹配，选择最相似的 top-k 个向量作为候选向量。本文选择欧式距离计算向量之间的相似度。其计算公式如式 0 所示。

$$d(A, B) = \sqrt{\sum_{i=1}^n (y_i - x_i)^2} \quad (3-1)$$

其中， $A = (x_1, x_2, \dots, x_n)$  和  $B = (y_1, y_2, \dots, y_n)$  分别表示用户问询问向量和向量数据库中的嵌入向量。值得注意的是，由于我们任务的特殊性，实际上用户问询的向量是存在在向量数据库中的，二者的欧式距离为 0，因此必然会在检索结果中，我们需要在检索的结果中排除掉问询问向量本身。

为提高检索的效率，本文使用 FAISS (Facebook AI Similarity Search) 作为稠密向量检索工具进行检索。FAISS 能够对向量数据库中的向量构建索引，通过索引能够快速地进行检索。FAISS 支持多种类型的索引，包括平坦索引，倒排文件索引，层次化导航图等多种索引，以应对不同规模、维度、存储要求的数据和不同的查询速度需求。由于本文的研究对象是软件项目代码，分析粒度为方法级，而软件项目至多只有几千个方法，相比于文档类问答生成的规模较小，因此本文使用较为简单的平坦索引对方法块进行检索。

### 3.3.4 推理增强

在检索得到候选的与问询方法存在变更影响关系的方法后，进一步生成提示指导大模型进行推理。

**多轮对话** 由于我们的对话任务单一，因此只需提示一次任务主题，后续直接提示问询方法和候选方法即可，指导大语言模型进行格式化的输出，方便后续处理。具体的提示模版如前文所示。

**槽位填充** 根据检索得到的结果，动态地生成当前任务的提示。值得注意的是，为了包含依赖路径关系，这里候选方法是以组的形式出现的，如果组内元素个数不为 1，则说明该组是多跳依赖类型，则提示对应的路径信息，否则直接进行推理。填充的方式如图所示。 $[x]$  表示用户问询方法， $[z]$  表示检索到的信息，根据 3.2 节中提到的依赖路径方法，对间接依赖的方法进行路径信息的补充，组成得到对应的提示词。

将大语言模型给到的格式化回答进行处理即得到与问询方法有变更影响分析的方法。对项目中的所有方法都进行问询，即可得到所有变更影响关系。

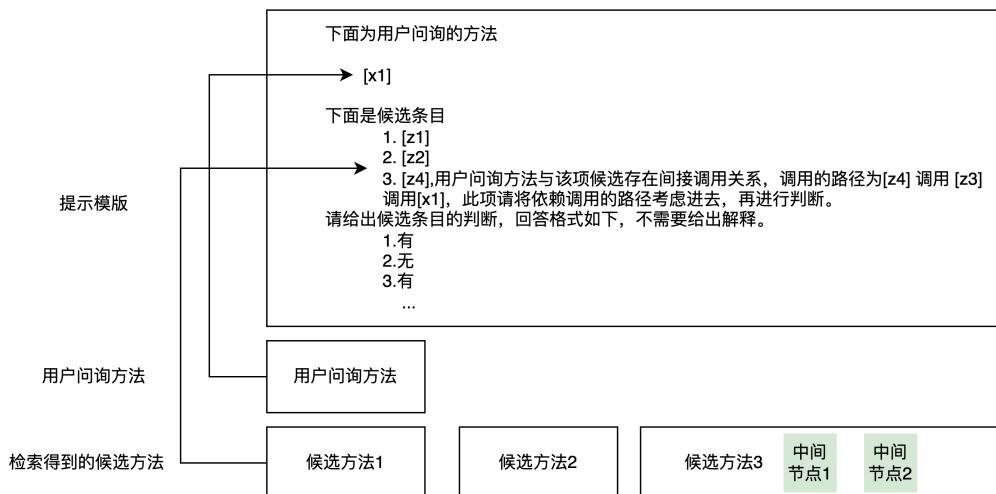


图 3-2 槽位填充

对于有 N 个方法的项目，嵌入次数为 N 次，检索次数为 N 次，对话次数为 N 词，检索过程会进行  $N^2$  次欧式距离相似度的计算，相比于深度学习方法  $N^2$  次的嵌入和多层感知机的计算，不仅节省了大量计算时间，还增加了依赖路径的信息，从而获得更好的性能。

## 3.4 实验结果与分析

### 3.4.1 实验数据与评价方式

**实验数据** 为了方便对比本章方法与前文方法的性能，本章的测试数据与第二章相同。值得注意的是，为验证我们训练的嵌入器的性能，我们进一步将嵌入模块的数据集按 8: 2 分为训练集和测试集，通过计算评价指标评估嵌入器的性能。

#### 评价指标

- **嵌入器评价指标：**为了验证我们训练的嵌入器的性能，本文使用召回率指标进行评价。这是因为在我们的任务中不区分检索结果中的排名，我们关注的是检索模块是否能将有变更影响关系的方法都查全。用 ARU，与分类阈值无关!!!
- **变更影响关系评价指标：**与第二章相同，依旧使用 F-measure、召回率和准确率作为评价指标来评估方法的有效性。具体公式为式 0 0 0。

### 3.4.2 实验设置

- 为了验证训练的检索模块的性能，我们与现有的嵌入模型进行对比。分别选择不同的检索数，分析检索数对召回率的影响，同时确定检索器的候选个数 k。

表 3-1 嵌入模型参数量

嵌入模型	参数量
gte-Qwen2-7B-instruct	7B
gte-Qwen2-1.5B-instruct	1.5B
gte-large-en-v1.5	434M
gte-base-en-v1.5	137M
bge-large-en-v1.5	335M
bge-base-en-v1.5	109M

- 训练嵌入模型参数设置如表。

表 3-2 模型参数设置

超参数	数值
Token embedding size	768
codeBERT learning-rate	1e-5
codeBERT dropout	0.4
Classifier learning-rate	1e-4
Adam $\beta_1$	0.95
Adam $\beta_2$	0.999
batch_size	64

- 为验证结合依赖路径的推理方法是否能有效解决间接依赖信息缺失的问题，本章实验通过结合和不结合依赖路径的方式分别进行实验，根据结果验证二者的差异。

- 本章实验中使用 ChatGPT 模型作为大语言模型进行实验，通过调用 API 的方式使用。

### 3.4.3 实验结果与对比分析

**RQ1：**训练的检索器性能如何，是否能有效地检索到与问询方法存在变更影响关系的方法？

**RQ2：**本章提出的基于 RAG 的影响关系分析方法性能如何，相比于其他方法表现如何？

**RQ3：**本章提出的结合依赖路径的推理方法是否能有效解决间接依赖信息缺失的问题？

**RQ4:** 这三种方法检测到的变更影响关系对软件项目的质量贡献如何？应从什么角度指导开发者对软件项目进行维护？

### 1. 针对于 RQ1 的实验

实验结果如表所示。表中横坐标表示设定的检索器得到的检索结果的数量  $k$ ，纵坐标表示召回率。可以发现，在 6 之前， $k$  越大召回率越高，在 6 之后则基本维持在 1。因此检索器的性能是较好的，能在较少的  $k$  就能达到较高的召回率。同时我们也选择以  $k$  值为 6 进行后续的实验。

【有个图】

### 2. 针对于 RQ2 的实验

在测试集上进行实验，得到的结果如表所示。

表 3-3 变更影响实验结果 - F-measure/召回率/精确度

方法	F-measure	recall	precision
依赖关系	42.2	81.3	28.5
克隆关系	6.4	3.3	92.0
共现关系	62.1	68.7	56.6
深度学习	65.2	69.4	61.4
<b>RAG</b>			

根据实验结果可以看出，本章方法性能优于深度学习方法。这说明基于 RAG 方法总体上讲相比于深度学习方法有更优的检测性能。

进一步对于不同类型的影响关系进行分析，可以发现相比于深度学习方法来讲，基于 RAG 的方法在依赖型的影响关系检测上，拥有更好的性能，尤其在查全率上相比基于深度学习的方法而言，提升了 1，这说明该方法在提示了依赖路径之后，补全了这部分信息，从而能更好地报告多跳依赖的影响关系。

表 3-4 变更影响实验结果 - F-measure/召回率/精确度

方法	DB-F-measure	DB-recall	DB-precision	LB-F-measure	LB-recall	LB-precision
依赖闭包	44.0	96.2	28.5	0	0	0
克隆代码	0	0	0	31.95	19.1	97.8
历史共现	60.3	65.9	55.6	69.3	81.7	60.2
深度学习	63.0	66.3	60.1	74.4	83.4	67.2
<b>RAG</b>						

除此之外，我们还统计了深度学习方法和 RAG 方法在测试集上的计算时间，通过对比我们发现，基于 RAG 的方法能节省大量的时间，相比于深度学习方法两两计算的方式，RAG 的检索功能大幅提高检测效率。

【有个时间图折线，横坐标项目，纵坐标时间】

### 3. 针对于 RQ3 的实验

本实验探讨依赖路径方法对检测效果的影响。对依赖路径进行了消融实验，得到的结果如表所示。

表 3-5 变更影响实验结果 - F-measure/召回率/精确度

方法	DB-F-measure	DB-recall	DB-precision	LB-F-measure	LB-recall	LB-precision
深度学习	44.0	96.2	28.5	0	0	0
深度学习 + 依赖路径清洗	0	0	0	31.95	19.1	97.8
RAG	60.3	65.9	55.6	69.3	81.7	60.2
RAG+ 依赖路径推理	63.0	66.3	60.1	74.4	83.4	67.2

根据结果可以看出结合依赖路径对于基于 RAG 的方法有一定的提升作用，尤其针对依赖型变更影响关系，相比于仅依靠 RAG 的方法，召回率提升了 1%，精确率虽略有下降，但 F-measure 提升了。

### 4. 针对于 RQ4 的实验

在软件维护过程中，代码变更是必要的，而由于变更影响关系的存在，导致用户在变更时必须谨慎更改，防止功能上和逻辑上的不一致或代码架构的恶化。那么变更影响关系本身是否反映质量问题呢？本文从变更影响关系的数量属性和类型属性进行讨论。

**数量属性** 一般而言，一个成熟且高质量的软件项目通常具有清晰的模块化结构，整个软件架构存在模块内高内聚、模块间低耦合的特性，维护起来非常方便。因此对于一个方法而言，如果它的变更会影响到较大的范围，存在“牵一发而动全身”的效应，说明在该方法上可能存在质量问题，从而导致较差的可维护性。

基于上述分析，本章统计了变更影响关系数量位于前 5% 的方法，认为这些方法可能是代码质量较差的特征代表，并将其作为重点信息报告给用户。统计用户的接受率。得到的结果如表 3-8 所示。

表 3-6 变更影响实验结果-接受率

项目名称	依赖闭包	克隆代码	数据挖掘-支持度 2	数据挖掘-支持度 3
TheAlgorithms	36.2	<b>92.3</b>	81.2	89.4
antiword-0.37	33.7	<b>89.9</b>	-	-
jemalloc-5.3.0	33.4	89.3	69.8	<b>92.1</b>
libbpf-1.1	40.4	<b>94.4</b>	76.4	77.3
librdkafka-2.1.0	28.3	<b>83.7</b>	74.3	83.2
FFmpegKit-5.1.0	27.2	84.7	64.7	<b>87.0</b>

实验结果中，基于依赖闭包的方法的接受率依旧为最低，也就是说，依赖闭包方法检测到的变更影响范围较大的那些方法，并不一定是真实的差质量代码。这同样是由于依赖闭包方法的涟漪效应的不准确性导致的误报问题，这类误报使得大量真实和虚假的变更影响关系混同，导致开发者难以分辨哪些方法是真实的差质量代码，也难以进行变更。在根据代码静态结构得到的变更影响关系中，开发者实际上更信任自己通过阅读代码结构得到的一到二层涟漪效应。

而另外三种方法的效果均优于依赖闭包方法，表明了三种方法提取的关系的数量属性能很好的反映代码的质量。实验结果中有4个项目的最高值均来源于克隆代码方法，即使在最大的数量远低于依赖闭包方法的情况下，仍然能让开发者接受，体现了克隆代码方法的准确性。除此之外，对于克隆代码方法，我们还观察到部分未被接受的方法对存在一些共同的特征，进一步揭示了影响方法准确性的其他因素。例如，以 antiword 项目中的一对包含克隆代码片段的方法为例，这两个方法的统计信息如表 3-6 所示，其中克隆代码片段仅占据 8 行。代码长度的可视化形式以及克隆代码片段如图 3-6 所示，包含的重复代码仅为两行单独的语句。之所以在统计时被视为克隆代码，主要是由于编程习惯的影响，每个参数被单独放在一行，这导致了克隆代码的扩展至 8 行。用户认为此例较为牵强，因为克隆代码占整个方法的行数过少，这表明克隆代码所占比例也是开发者决定是否信任该项分析的重要因素之一。

表 3-7 被用户拒绝实例代码信息

方法	代码行数 (Line of code)	克隆代码长度
pHdrFtrDecryptor	125	8
szFootnoteDecryptor	115	8

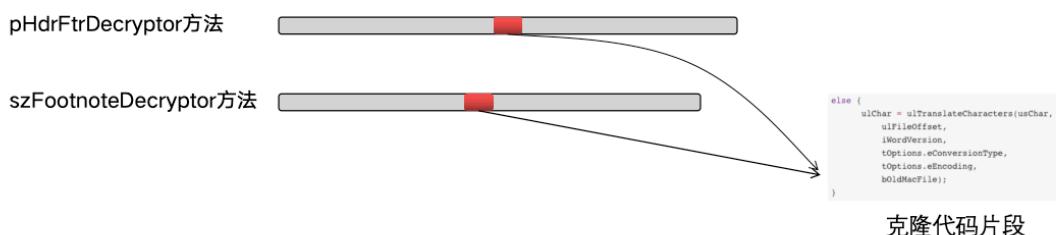


图 3-3 被用户拒绝实例代码长度可视化

数据挖掘方法的实验结果也高于依赖闭包方法，通过对比支持度不同的实验可以发现，支持度为 3 略高于支持度为 2 的接受率，这是由于，支持度为 3 意味着这对方法起码发生共同修改三次才能被挖掘到，这排除了共同修改 2 次

的方法对的一些偶然情况，挖掘到的关系更少，但是也更为精准。因此，支持度也是决定数据挖掘方法精确度的重要因素之一。

总的来讲，三种方法的数量属性均可以反映质量问题，表现为变更影响范围越大的方法，代码质量越差。在这里我们给到的代码优化和重构建议是，用户可通过合并有变更影响分析的方法或解耦合，从而降低方法的影响关系数量，进而提高代码质量。

**类型属性** 本文提出的三种方法中，基于代码克隆方法能检测逻辑型中基于代码克隆的变更影响关系，而基于数据挖掘和基于深度学习的方法能检测更为广泛的逻辑型关系，包括协同方法、镜像方法等。

首先对于代码克隆关系，此种类型的代码在软件项目中一直被认为是较差的代码习惯，主要因为代码克隆通常导致重复性高、维护成本大以及可读性差。代码克隆不仅增加了代码的冗余度，还可能隐藏潜在的缺陷。开发者应使用抽象化的设计模式，通过功能模块化等策略提取重复代码部分，减少重复代码的产生。

其次对于

通过对同一软件的不同版本进行检测，得到的实验结果侧面印证了，具有一定的实际应用价值。

### 3.5 本章小结

本章提出了基于 RAG 和依赖路径的变更影响分析方法，通过结合依赖路径提升了方法在多跳依赖影响上的检测性能，并通过 RAG 检索的方式，提高了检测效率。最后，本章通过一系列实验验证了该方法在提取变更影响关系上的有效性，并通过消融实验验证了结合依赖路径对方法的提升。

## 第4章 基于代码审查图的代码结构可视化和质量分析

### 4.1 引言

在软件开发过程中，开发者通常会经历多个阶段。在项目的初始阶段，开发者需要阅读和理解已有的代码，这是熟悉软件项目的第一步。然而，对于大型项目而言，由于项目代码量庞大，涉及的模块和功能众多，这一过程通常需要耗费大量的时间和精力。除此之外，开发者在对软件进行修改时，如添加新功能或修复缺陷，通常需要深入了解修改代码的上下文信息。如果对上下文理解不清晰，可能会导致变更不完全或不准确，进而影响软件质量。在软件开发的后期，开发者往往需要作为代码审查者参与到代码审查过程中。代码审查的主要目的是评估变更后的代码是否符合质量标准，是否能够顺利地合并到主分支中。这一过程不仅在协作开发中至关重要，也是确保软件质量的有效手段。然而，代码审查往往需要投入大量的时间和精力<sup>[31]</sup>。审查者不仅需要对变更的代码本身进行分析，还需要理解这些代码所处的上下文，才能做出正确的评估。

因此，无论是作为开发者还是审查者，理解软件项目的结构和代码是至关重要的。只有深入掌握软件的整体架构和各模块之间的关系，才能在后续的开发和审查过程中保证代码质量。然而，传统的代码阅读和理解方式不仅需要消耗大量的时间和精力，还难以确保高效性和准确性，尤其是在面对庞大复杂的代码库时。

为了提高代码理解的效率并减少人为错误，本文提出了一种基于代码审查图的代码架构和质量信息可视化的方法。通过将项目中的各个方法和全局变量表示为图的节点，用边表示方法与方法之间、方法与全局变量之间的依赖关系、耦合关系和变更影响关系，从而形成一个结构化的代码关系图。这样的图形化展示方式能够帮助开发者和审查者从宏观的角度掌握整个软件项目的架构和各个模块之间的关系，进而提升对代码的理解效率。通过这种方式，开发者和审查者可以更直观地识别出项目中的关键部分及其相互依赖关系，从而在变更和审查过程中更高效地评估代码的质量和影响。

## 4.2 基于代码度量的代码质量度量模型

### 4.2.1 代码质量度量模型

代码质量度量模型用于评估和量化软件代码质量，本文研究对象是方法以及模块的代码质量，我们选取具有代表性的以下 5 个方面的质量属性和 18 个对应的度量元来进行衡量。

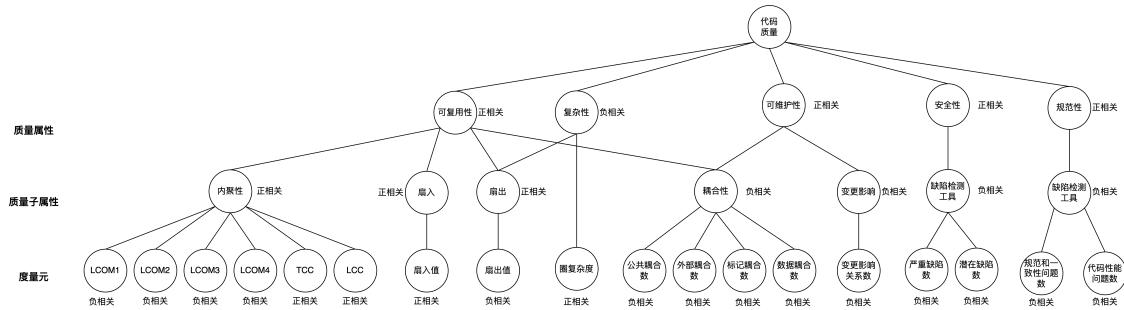


图 4-1 代码质量度量模型

在对代码质量的评价上，我们的分为模块级和方法级的两个层次的度量，其中内聚性为模块级度量，其他度量为方法级。通常评价可分为绝对评价和相对评价两种，对于一组相同设计目的代码，相对评价更为合适<sup>[32]</sup>，一个项目内的模块或方法可以看作设计目的相同的代码，方便用户在项目中关注到相对来说质量较差的代码。

**度量元度量** 使用统计中常用的分档方式，分五个档次对应优、良、中、低、差，按式的方式给定参考分值。对于度量值越小越好的度量元，假设代码  $c$  的质量属性  $k$  中子属性  $i$  度量元  $j$  对应的最大值和最小值分别是  $\max_{ij}$  和  $\min_{ij}$ ，测试值区间  $Z_{ij}$  和测试值  $t_{ij}$  在  $Z_{ij}$  中的位置  $t'_{ij}$  分别为：

$$Z_{ij} = \max_{ij} - \min_{ij} \quad (4-1)$$

$$t'_{ij} = t_{ij} - \min_{ij} \quad (4-2)$$

相对评价的分值  $V_{ij}^k$  如式。

$$V_{ij}^c = \begin{cases} 100, & t'_{ij} \leq 0.2Z_{ij} \\ 80, & 0.2Z_{ij} < t'_{ij} \leq 0.4Z_{ij} \\ 60, & 0.4Z_{ij} < t'_{ij} \leq 0.6Z_{ij} \\ 40, & 0.6Z_{ij} < t'_{ij} \leq 0.8Z_{ij} \\ 20, & 0.8Z_{ij} < t'_{ij} \end{cases} \quad (4-3)$$

对于度量值越大越好的度量元，只需把式中分值倒序。

**质量子属性度量** 一个质量子属性包含一个或多个度量元。在这里将一个子属性下的所有度量元进行组合，总的评估质量子属性。定义代码 c 的子属性度量  $U_i^c$  如式 (4-4)，这里将各度量元的权重设为等值。

$$U_i^c = \sum_j w_{ij} V_{ij}^c, \quad \sum_j w_{ij} = 1 \quad (4-4)$$

**质量属性度量** 通过对质量特性下的度量元进行运算得到代码质量特性的度量，计算公式如下，这里将各子属性的权重设为等值。质量特性度量可以作为代码在某一方面的质量表现情况，在实际情况下可以单独使用，因此这里不再对质量特性度量进行聚合评分。除此之外，每一层级的质量属性均可如式 (4-3) 所示对度量进行相对分档。

$$Q_k^c = \sum_i w_{ki} U_i^c, \quad \sum_i w_{ki} = 1 \quad (4-5)$$

#### 4.2.2 基于内聚度缺乏度和连通性的的内聚性度量

##### 1. 基于内聚度缺乏度的内聚度计算

LCOM (Lack of Cohesion in Methods) 系列指标是根据模块内聚度的缺乏程度来衡量模块的内聚度的指标。在本文中，面向对象语言以类为研究范围进行计算内聚度，非面向对象的语言以文件为研究范围进行计算，类中的成员属性对应文件中的全局变量，类中的成员方法对应文件中定义的方法。LCOM 指标的核心思想是度量一个类中方法对实例变量（属性）的共享程度。不同版本的 LCOM 有着不同的计算方法和含义，体现了不同的侧重点。这里一共建算以下四个指标，

(1) LCOM1, 含义是不引用相同字段的方法对数目<sup>[33]</sup>。计算公式如式(2-1)。

$$LCOM1 = C_n^2 - e \quad (4-6)$$

其中  $n$  是文件中的方法总数,  $e$  是引用相同字段的方法对。以图 2-4 为例介绍计算方式, 其中椭圆表示方法, 点表示变量, 点在椭圆内表示该方法引用了该变量。LCOM1 值为  $C_6^2 - 5 = 10$ 。

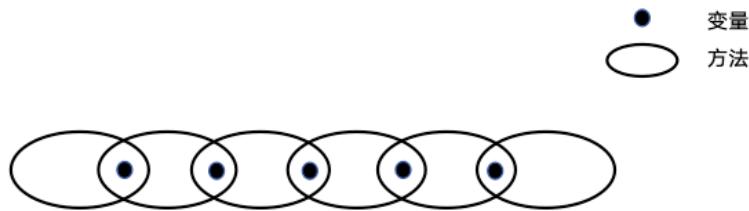


图 4-2 示例模块

(2) LCOM2, 含义是不引用相同字段方法对与引用相同字段方法对数之差<sup>[34]</sup>。其计算公式如式(2-2)。

$$LCOM2 = \begin{cases} P - Q, & \text{if } P \geq Q \\ 0, & \text{otherwise} \end{cases} \quad (4-7)$$

其中,  $P$  是不共享实例变量的方法对的数量,  $Q$  是共享实例变量的方法对的数量。如果 LCOM1 的结果为负数, 则被置为 0。图 2-4 模块中, 不共享变量的方法对  $P$  为 10, 共享变量的方法对  $Q$  为 5, LCOM2 值为  $P-Q=5$ 。

(3) LCOM3 是对前两种指标的进一步改进, 其计算公式如式(2-3):

$$LCOM3 = \frac{\left(\frac{1}{a} \sum_{j=1}^a \mu(A_j)\right) - m}{1 - m} \quad (4-8)$$

其中  $m$  为文件中的方法数,  $a$  表示文件中的变量数,  $\mu(A_j)$  表示的是引用变量  $A_j$  的方法数。如图 2-5 所示的文件中有 3 个方法和 3 个变量, 计算方式如图所示。

(4) LCOM4, 含义是以方法和变量为顶点, 方法引用字段或方法之间有调用关系则两节点之间有条边构成图的连通分支数<sup>[35]</sup>。计算时, 根据深度优先搜索的方式, 计算图中的连通分支数, 得到的值即为 LCOM4。如图 2-6 所示的两个文件的 LCOM4 的值分别为 2 和 1。

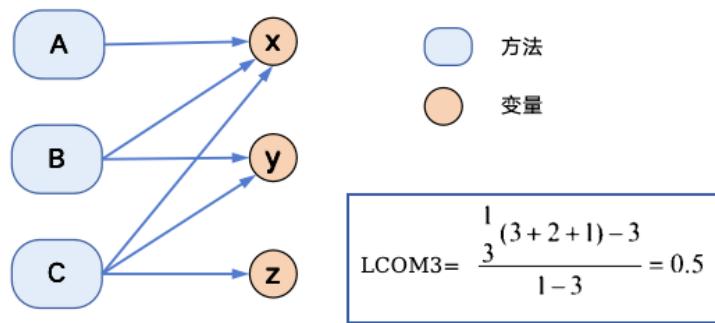


图 4-3 LCOM3 计算示例

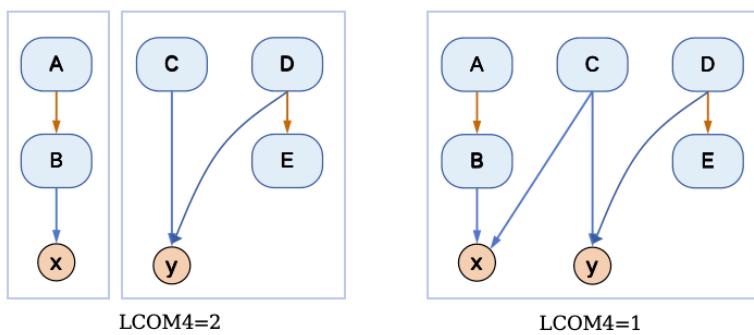


图 4-4 LCOM4 计算示例

## 2. 基于连通性的内聚度计算

TCC (Tight Class Cohesion) 和 LCC (Loose Class Cohesion) 是用于衡量模块内聚度的指标，这两个指标主要关注于模块中方法之间的连通关系，核心思想是通过分析模块中方法如何相互作用以及如何访问共同资源（如全局变量）来评估模块的内聚度。

(1) TCC，含义是有连通关系的方法对数与总方法对数的比值<sup>[36]</sup>。TCC 关注于模块中方法之间的“直接连接”。如果两个方法直接共享访问同一个变量，则认为这两个方法是直接连接的。计算公式如式 (2-4)。

$$TCC = \frac{e}{C_n^2} \quad (4-9)$$

其中  $n$  是文件中的方法总数， $e$  是图中的直接连接边数。

(2) LCC 则基于方法间接引用共同字段的关系进行计算<sup>[36]</sup>。LCC 除了考虑直接连接的方法对外，还包括了间接连接的方法对。如果两个方法不是直接连接，但可以通过一系列的方法调用或变量引用来连接，则认为它们是间接连接的。LCC 的值基于模块中直接或间接连接的方法对占所有可能方法对的比例来计算。因此，LCC 的值通常不低于 TCC 的值，并且提供了一个更宽泛的模块内聚度视角。计算公式如式 (2-5)：

$$LCC = \frac{e + e_{indirect}}{C_n^2} \quad (4-10)$$

其中  $n$  是文件中的方法总数,  $e$  是图中的直接连接边,  $e_{indirect}$  是除直接连接边的边数。如图 2-7 是计算 LCC 和 TCC 的例子, 左图中通过方法 AB 通过变量  $x$  直接连接, 方法 CD 通过变量  $y$  直接连接, 直接连接和间接连接都是 2。而右图中直接连接是 AB、BC、BC 和 CD, 间接连接是 AD 和 BD, 因此计算结果如图。

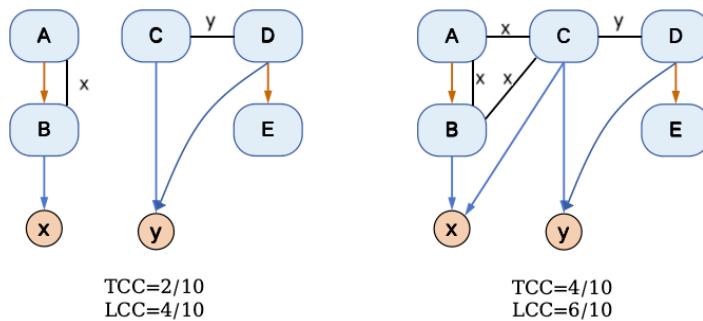


图 4-5 TCC 和 LCC 计算示例

#### 4.2.3 方法间耦合性度量

耦合是在软件架构中用来描述模块间相互依赖和连接程度的一个重要指标。耦合度的高低直接影响到系统的维护性和可扩展性。在现有的研究和实践中, 耦合度通常被细分为六个等级, 如表 2-1 所示, 这些等级从高到低反映了模块间依赖的紧密程度。本文关注的是方法与方法之间的耦合性。通过深入分析方法级别的耦合性, 研究方法如何通过参数传递、调用关系、共享全局变量等方式相互依赖, 我们可以更准确地识别潜在的设计缺陷和优化机会, 从而提高系统的模块化程度, 增强系统的可维护性和可扩展性。

表 4-1 软件架构中耦合性分类

耦合性类别	描述	耦合程度	本文是否分析
内容耦合	模块直接访问或修改另一个模块的内部数据	6	否
公共耦合	模块访问同一公共数据环境	5	是
外部耦合	模块共享全局简单数据结构	4	是
控制耦合	模块传递控制信息, 影响计算流程	3	否
标记耦合	通过参数传递复杂数据结构信息	2	是
数据耦合	通过参数传递简单数据	1	是

内容耦合是耦合度最高的一种形式, 它表示一个模块能够直接访问或修改另一个模块的内部数据和结构。在方法级的耦合分析中, 这种耦合形式通常不

被考虑，因为方法间的直接数据访问往往通过参数传递或者 API 调用实现，而不是直接的内容访问。

公共耦合发生在多个模块共同访问某个全局数据环境时。这种数据环境可能是全局数据结构、全局变量或内存公共区域等。在提取到的全局变量表中，对于复杂数据结构如结构体和数组，其引用点所在的方法之间均存在公共耦合关系。

外部耦合与公共耦合相似，但区别在于它涉及的是对全局简单变量的访问。例如，当多个模块访问或修改相同的全局简单类型变量时，则这些模块之间存在外部耦合。

控制耦合指模块之间传递信息中包含用于控制模块内部的信息。在提取到的方法摘要表中，遍历方法，如果该方法调用其他方法时，对应方法的参数列表中有变量决定了被调用方法中的计算流程，则方法之间存在控制耦合关系。由于本文不考虑分析方法内部的控制逻辑，因此不提取此种耦合。

标记耦合指通过参数表传递数据结构信息，调用时传递的是数据结构。在方法摘要表中提取了方法的参数列表，包括参数名和参数类型，根据参数类型，可以确定参数表中是否包含复杂类型。除此之外，在方法的调用表中，也提取了方法调用的其他方法，结合这两个信息，即可确定两个方法是否存在标记耦合关系。

数据耦合指通过参数表传递简单数据。与标记耦合类似，根据参数类型可以确定参数是否全部为基本类型，结合方法调用表，即可确定两个方法是否存在数据耦合。

#### 4.2.4 方法扇入扇出度量

方法的扇入（Fan-in）和扇出（Fan-out）是软件工程中常用于衡量方法可复用性和方法复杂性的两个指标。

扇入是指调用某个方法的不同方法的数量。扇入值较高的方法通常被认为是重要的或核心的，因为它们被多个其他方法所依赖。高扇入值可能意味着该方法执行了一个基础或共享的任务。可复用性较强。

扇出（Fan-out）指的是一个方法直接调用的其他方法的数量，反映了该方法对外部方法的依赖程度。较高的扇出值通常意味着该方法需要管理和协调更多的外部操作。在软件工程中，较高的扇出值可能导致方法的责任范围过大，进而影响代码的可复用性。具体而言，较高的耦合度可能使得该方法难以在不同的上下文中独立使用或重用。因此，扇出值较高的代码常常提示着潜在的复杂

性问题，可能需要进行重构或拆分，以减少对外部方法的依赖，提升其可复用性和灵活性。

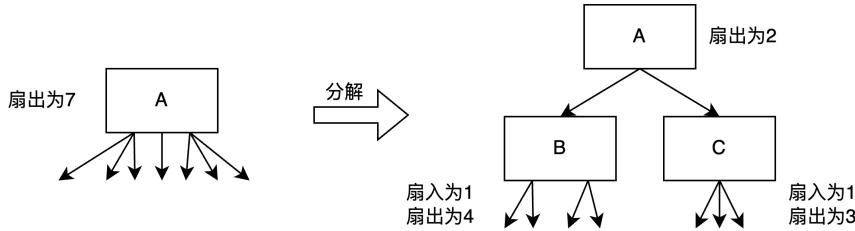


图 4-6 高扇出及其分解示例

本文中对于提取到的方法摘要表，遍历每一个方法，统计其调用方法的数量即可计算出该方法的扇出值，再以该方法名在方法摘要表中搜索调用了该方法的方法，统计总数，得到的值即为扇入值。

#### 4.2.5 基于静态检测工具的度量

在安全性和规范性代码属性中，我们使用静态代码分析工具 Cppcheck，对项目中的源代码进行检测和分析。Cppcheck 是一款开源的静态分析工具，专门用于检测 C/C++ 代码中的缺陷、潜在错误和编码规范问题。Cppcheck 会根据问题的严重性和类别将检测出的问题如表 2-2 所示的六类。

表 4-2 cppcheck 报告问题分类

类别	描述
error	严重错误，通常包括内存泄漏、缓冲区溢出、空指针解引用等致命问题
warning	潜在的错误或不推荐的做法，如不安全的类型转换等
style	代码风格问题，通常与代码格式、命名约定等相关，例如变量命名不规范等
performance	性能问题，表明当前方式可能效率不高，例如重复计算、低效循环等
portability	平台相关问题，可能导致在不同环境出现不同行为，例如操作系统特有的 API 调用、字节序问题等
information	额外的信息或建议，例如推荐的编码实践等

检测结果中包括了诸如错误、警告和代码风格问题等不同级别的信息，这些信息帮助开发者识别代码中的潜在问题。我们将前两类作为安全性度量元，将 style 和 performance 类别的问题数作为规范性度量元进行计算。portability 和 information 由于并不是代码本身的质量问题，因此不进行计算。

Cppcheck 提供的报告与其他度量指标相结合，共同构成了全面的质量评估体系。用户可以根据检测报告对代码质量进行更细致的判断和分析，从而为后续的优化和重构提供科学依据。通过这种方式，本研究实现了对代码质量的多

维度综合评价，为开发者提供了更为精确的质量检测和改进方向。

在复杂性属性中，我们使用 `lizard` 提取方法的圈复杂度，它是一个轻量级的源代码复杂度分析工具，可以按方法或文件等不同的粒度计算圈复杂度，这里我们分析对象为方法级。

## 4.3 代码审查图的构建和代码结构的可视化

### 4.3.1 代码审查图构建

在软件开发过程中，尤其是在代码审查和变更管理的环节，开发者通常需要从全局的角度了解项目的结构、模块之间的关系以及代码的质量。然而，随着项目规模的扩大，代码的复杂性和模块间的依赖关系也会急剧增加，这使得传统的代码审查方法难以有效地展示全局视图，容易忽略一些潜在的问题。因此，为了更好地帮助开发者在复杂的项目中进行有效的代码审查和质量评估、以及更安全地维护变更，本章提出了一种名为“代码审查图”的可视化方式。

代码审查图主要由两个核心元素构成，分别是节点和边。其中，节点代表软件项目中的方法或全局变量，边表示节点之间的各种关系，包括耦合关系、变更影响关系以及依赖调用关系。接下来进一步详细介绍节点和边的构成。

**节点构建** 节点的作用是标识项目中的方法和全局变量。通过前文所述的方法摘要表和全局变量信息表，我们为每个方法和全局变量创建了对应的节点。每个节点都具有多个属性，这些属性能够提供有关节点所代表的方法或变量的关键信息，便于开发者对代码进行全面的审查和分析。

方法属性分为两个主要部分：首先是方法的基本信息，如表 4-1 所示。

表 4-3 代码审查图节点属性-基本信息

属性	描述
方法名	方法名，由方法所在路径和方法名拼接而成，保证唯一
方法参数	方法的参数列表，包括参数的名称和类型
方法内调用方法	本方法内调用的其他方法名
方法可作用域	表明方法是否全局可用
方法所在模块	方法所在模块，目前表示为方法所在文件

这一部分包括方法的名称、所在模块、方法签名、访问修饰符等，这些属性有助于开发者了解方法的基本信息。例如，方法的名称可以反映其业务功能，所在模块和作用域则有助于理解方法的上下文和调用约束等。

其次是与代码质量相关的度量和信息，如表 4-2 所示。

表 4-4 代码审查图节点属性-质量相关信息

质量子属性	度量元	描述
内聚性	LCOM1、LCOM2、LCOM3、LCOM4、TCC、LCC	总的相对评分、相应建议、具体的属性值
耦合性	数据、标记、外部、公共耦合数	总的相对评分、建议、与本方法存在数据、标记、外部、公共耦合关系的方法
变更影响	变更影响关系数	总的相对评分、与本方法存在变更影响的方法
扇出	扇出值	总的相对评分、建议、方法的扇出值
扇入	扇入值	总的相对评分、建议、方法的扇入值
代码缺陷	严重缺陷数和潜在缺陷数	总的相对评分、cppcheck 检测得到的本方法缺陷信息
代码规范	规范和一致性问题数、代码性能问题数	总的相对评分、cppcheck 检测得到的本方法规范性信息
圈复杂度	圈复杂度	总的相对评分、圈复杂度

这一部分将展示前文根据质量度量模型得到的代码的 5 种质量属性情况，包括可维护性、复杂性、可复用性、安全性和规范性五个方面，并且将这五个方面的特性根据度量进行分档，给出分档等级。同时为了进一步指导用户对具体质量子属性进行优化，还报告了质量子属性的相对质量情况，对于较差的子属性向开发者提供有针对性的改进建议。例如，如果某个方法的扇出度较差，则可能表明该方法在项目中的依赖关系过于复杂，可能需要拆分。类似地，如果某模块的内聚性较差，则说明模块内部各元素的依赖关系较为松散，指导用户对该模块进行重构。

对于全局变量的属性则主要包含表 4-3 中的信息。主要是对全局变量基本信息的展示，方便开发者快速了解该变量的作用域、使用情况以及与其他代码部分的关联性。通过这些信息，开发者能够更好地理解变量在整个项目中的作用以及对应的代码上下文。

表 4-5 代码审查图节点属性-全局变量信息

属性	描述
变量名	全局变量名，由所在路径和变量名拼接而成，保证唯一
变量类型	变量类型
被使用方法	使用了本全局变量的方法名
变量可使用域	表明变量是否全局可用
方法所在模块	变量所在模块，目前表示为所在文件

**边的设计** 在代码审查图中，边表示节点与节点之间的关系，这些关系揭示了软件系统中各个方法与全局变量之间的相互依赖和影响。根据其性质，边的类型主要分为三类，具体分类如表 4-4 所示：静态依赖关系、耦合关系和变更影响关系。

其中静态依赖关系分为方法之间的调用关系和方法与全局变量的引用，耦合关系如表 4-2 中所示共 4 类，变更影响关系则是根据第三章方法检测得到的变更影响关系。

每种关系反映了不同层次的代码相互作用，帮助开发者全面理解系统的结构和潜在的质量风险。

表 4-6 代码审查图边分类

属性	描述
静态依赖关系	含方法之间调用、方法引用全局变量两种
耦合关系	含数据耦合，标记耦合，外部耦合，公共耦合四种
变更影响关系	由第三章方法检测得到

### 4.3.2 代码审查图可视化

本节从代码审查图的可视化方案和交互方案两个方面展开介绍。

**可视化方案** 代码审查图的可视化方案基于开源项目 **G6**。**G6** 是一个强大的图形可视化引擎，提供了绘制、布局、分析、交互、动画等全方位的图形可视化基础功能，具有简单易用且完备的特性。**G6** 具有两个显著优势。

(1) 数据与可视化图形分离：在使用 **G6** 时，用户只需将图的数据组织为 JSON 格式，如图 4-3 所示，包括节点信息和边信息，直接传递给 **G6** 即可自动生成对应的力导向图。这种数据与图形的分离不仅简化了开发流程，还提高了数据的灵活性和可操作性，便于进行后续的数据更新和图形重绘。

```
{
  "nodes": [{"id": "node1"}, {"id": "node2"}],
  "edges": [{"source": "node1", "target": "node2"}]
}
```

图 4-7 **G6** 图数据示例

(2) 高度的定制能力：**G6** 提供了丰富的图形展示配置选项，用户可以根据需求自由选择不同的样式和布局方式。如果 **G6** 内置的元素不满足特定需求，它还支持用户自定义节点、边及其他元素，使得图形展示更加贴合实际应用场景。

景。

本文使用 G6 内置节点和边实现代码审查图的可视化。G6 的节点构成共包含 6 部分，其中 label 表示文本标签，通常用于展示节点的名称或描述，本文中将节点属性赋值给 label，便于用户查看属性相关信息。G6 的边的构成共包含 4 部分，label 具有同样的功能，将边的类别用于 label。让 G6 加载此数据源进行展示，就实现了同时也实现了计算逻辑与图形可视化的有效分离。

**交互方案** 对于软件项目这样的分析对象，方法和全局变量的数量常达到千级别，对于一个图来讲，这样的级别很难在图中展示完所有的信息，因此需要用户交互，来展示更详细的信息。表 4-5 展示了目前代码审查图的交互和对应的逻辑设计。

表 4-7 代码审查图交互和逻辑设计

交互方式	业务逻辑
视角缩放	操作鼠标滚轮对图进行缩放
视角移动	鼠标拖拽移动整个代码审查图
聚焦节点	光标悬停在节点上显示节点的方法名/变量名
移动节点	鼠标长按节点拖拽可移动节点
查看节点属性	鼠标点击节点展开节点属性
聚焦关系	光标悬停在边上显示边的类型
查看关系信息	鼠标点击节点展开节点属性
节点筛选	通过点击筛选节点按钮，确认是否筛选掉孤立节点

#### 4.4 基于代码审查图的代码质量分析

代码审查图能直观体现模块或方法层面上与代码质量相关的特征，从而辅助用户理解代码质量较差或较好的具体原因。本节通过分析项目代码历史版本中被修复或重构的代码对应的代码审查图结构，总结了以下 5 种不良的图模式。它们在不同的角度上反映了代码较差的质量属性和对应的优化方向。

**1. 冗余代码元素** 冗余代码元素指的是在代码中定义但从未被使用的代码元素。在代码审查图中表现为孤立节点，如图4-8所示。即某个节点不与任何其他节点存在边的图模式。这意味着该代码元素不和任何其他代码产生依赖、调用等联系，不论是作为全局变量还是方法出现，都会消耗不必要的性能和资源，且影响其所在模块的内聚程度。优化时，可以考虑去掉冗余代码，只保留和软件核心逻辑相关的代码。

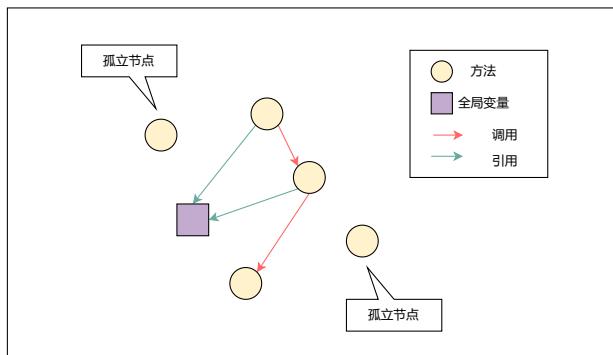


图 4-8 冗余代码图模式

**2. 长嵌套调用** 长嵌套调用在代码审查图中的表现方式是方法间的链式连接，如图4-9所示。长嵌套调用有以下几点危害，1. 可读性差：层次过多的嵌套调用会使代码结构变得复杂，理解代码的执行顺序和逻辑变得困难。2. 可维护性差：由于链式的松散结构，模块的可维护性也会变差，当需要对中间环节进行变更时，可能会影响整个嵌套调用的逻辑。3. 性能问题：方法调用会涉及到栈帧的创建和销毁，长的嵌套调用会导致栈的深度增加，频繁的栈操作可能会影响性能。在代码审查图中可以直观地发现嵌套调用的现象。优化时可以考虑合并部分底层方法，减少嵌套层数来优化代码。

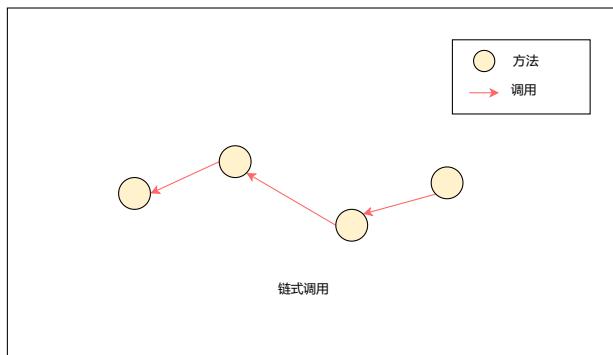


图 4-9 长嵌套调用图模式

**3. 复杂方法** 复杂方法的特性之一是职责过多，通常表现为方法内部调用了过多的其他方法或与大量方法存在变更影响关系，这种特征在代码审查图中表现为具有大量的出边的中心化节点，如图4-10所示。这种方法的危害有以下几点，1. 可维护性差：依赖过多外部方法，任何外部方法的变动都可能导致当前方法的变动，增加维护成本。2. 增加耦合度，降低了系统的灵活性。3. 违背单一职责原则。

通过代码审查图用户可以迅速定位到复杂方法，深入了解其上下文信息，

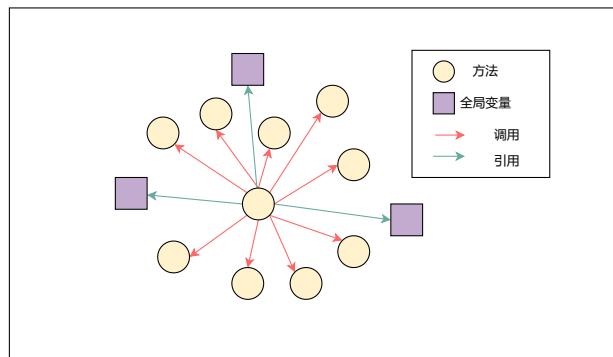


图 4-10 复杂方法图模式

进而判断其是否存在不合理的复杂度。优化时，用户可以考虑通过方法分解和增加中间层的方式重构该方法、简化其职责并调整其与其他模块的关系，从而有效提升系统的可维护性和可扩展性。

**4. 职责不匹配** 在软件项目里，每个代码元素都处于特定的物理位置，比如所在的文件、类、文件夹等。这种物理位置往往体现了软件最初的设计理念。而在静态代码结构中，每个代码元素还处于特定的逻辑位置，这在代码审查图中，表现为与其他代码元素的相对位置关系。对于某些方法而言，会出现一种情况：其承担的职责与它所处的物理位置不再完全适配，反而与其他模块的关联更为紧密。我们将这种问题称作职责不匹配问题。

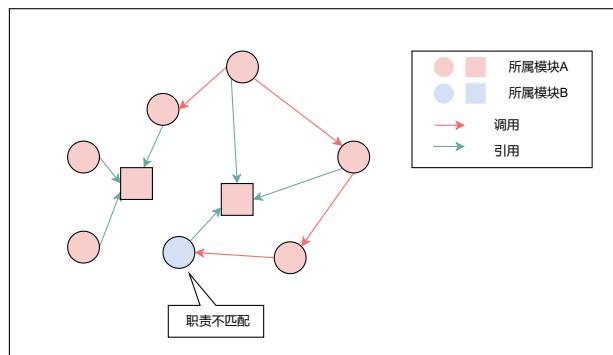


图 4-11 职责不匹配图模式

代码审查图中，相同模块的代码元素呈现为同一颜色，不同模块不同颜色，得益于力导向的可视化特性，逻辑上紧密依赖的节点聚集在一起。职责不匹配的特征在代码审查图中的模式如图4-11所示，表现为逻辑聚集的节点颜色不统一，某些节点（如图4-11中的蓝色节点）与模块的主色调（红色节点）不符。优化时一般有两种角度，如果用户认可最初的设计理念，可将职责不匹配的代码与主模块进行拆分，如果用户认为逻辑位置更为合适，则可将职责不匹配的代

码合并进主模块。通过这样的重构操作，可将依赖和变更影响关系限制在同一模块内，避免变更时跨模块的影响。

**5. 模块间不良的逻辑型变更影响** 对于软件代码而言，模块间的变更影响关系应当以依赖型变更显式地呈现，这有助于开发者在静态结构中清晰地识别依赖关系。相比之下，逻辑型变更影响往往隐匿在代码内部，仅通过代码上下文或前后调用关系难以察觉。模块间的变更影响通常被视为不利因素，它可能导致不良变更传播问题，需要引起维护人员的关注。

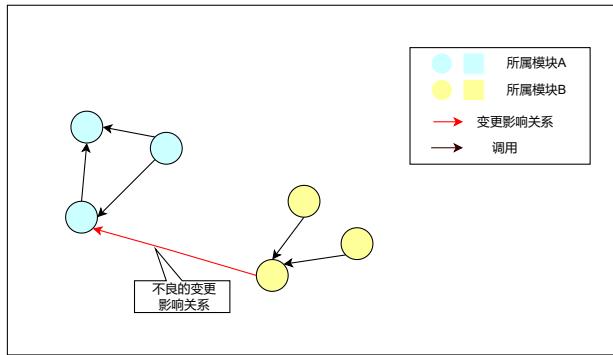


图 4-12 不良的逻辑型变更影响

这种不良的逻辑型变更影响在代码审查图中表现为图4-12所示的模式，其中红色边表示逻辑型变更影响关系，黑色边表示由调用引起的依赖型变更影响关系。两个模块由于逻辑型影响被连接在一起，导致变更时很难察觉，且模块间互相影响。这种现象暴露了系统架构中的潜在问题——虽然各模块内部结构较为合理，但模块间的依赖关系复杂且紧密，增加了维护和变更时的复杂性与风险。为了解决这一问题，优化的建议是尽量减少或消除这些不良的变更影响关系，或者将其转化为依赖型影响，从而减少隐匿的跨模块变更传播的可能性。

## 4.5 实验结果与分析

### 4.5.1 实验数据和实验设计

**1. 实验数据** 本章选取第二章实验部分的项目进行研究，选择 jemalloc 为示例项目进行代码质量度量模型的有效性验证，选取 jemalloc 和 TheAlgorithms 项目进行代码审查图的不良图模式的验证。

1. 代码历史版本收集。将代码历史提交按以下原则进行筛选，筛选后收集提交前版本的项目源代码，提取该提交变更的方法，得到 `<code, list<changed_func>`

的二元组。

- 按关键词筛选提交：保留提交信息中包含以下关键词的提交：{ fix, refactor, Fix, Refactor }。这些关键词通常出现在与缺陷修复和代码重构相关的提交中，能够反映用户提高代码质量的操作。
- 按代码差异类型筛选：进一步分析代码差异，判断是否涉及修复或重构。

表 4-8 jemalloc 数据统计

项目属性	数值/信息
总提交	3530
包含修复提交	653
包含重构提交	53
修复提交抽样	10
重构提交抽样	10

为了验证所提的代码度量模型与基于代码审查图的质量分析的有效性，本实验从软件的历史版本变更角度出发，结合代码度量和图模式，探讨修复和重构概率大的代码与其质量之间的关系，从而验证其有效性。

**2. 验证代码质量度量模型的实验设计** 本实验的核心数据是代码历史版本和对应的提交记录，具体步骤如下：

- 对于每一个提交，对提交前版本的项目源代码进行代码度量计算，不同的质量属性内部进行分档排名，得到方法之间的相对质量关系。
- 将该提交的提交变更方法与不同分档进行求交集，得到每档中方法变更的数量。
- 观察代码质量度量和变更概率的数量关系，验证有效性。

**3. 验证代码审查图的实验设计** 与前述步骤相似，先对提交进行筛选并收集提交前代码版本。对变更前的版本进行代码审查图生成，对于变更部分识别其是否具有不良图模式。

#### 4.5.2 实验环境

实验环境如表 4-6 所示。

#### 4.5.3 实验结果分析

**RQ1:** 代码质量度量模型能否有效反映项目的质量情况？在修复或重构过程中，代码质量如何影响开发人员的修改决策？

表 4-9 实验环境

环境	信息
操作系统	macOS Ventura v13.5.2
python	3.7
Java	1.8
G6	g6.min.js 4.3.11
libclang	15.0.7
pycparser	2.21

RQ2: 基于代码审查图分析得到的不良质量信息是否有效，代码审查图中能反映哪些不良的代码模式？

### 1. 针对于 RQ1 的实验

实验结果如表所示，表中横坐标表示代码度量的相对值，横坐标越大，代码质量越好。纵坐标表示在修复或重构的变更历史中，被变更的方法数量比例。

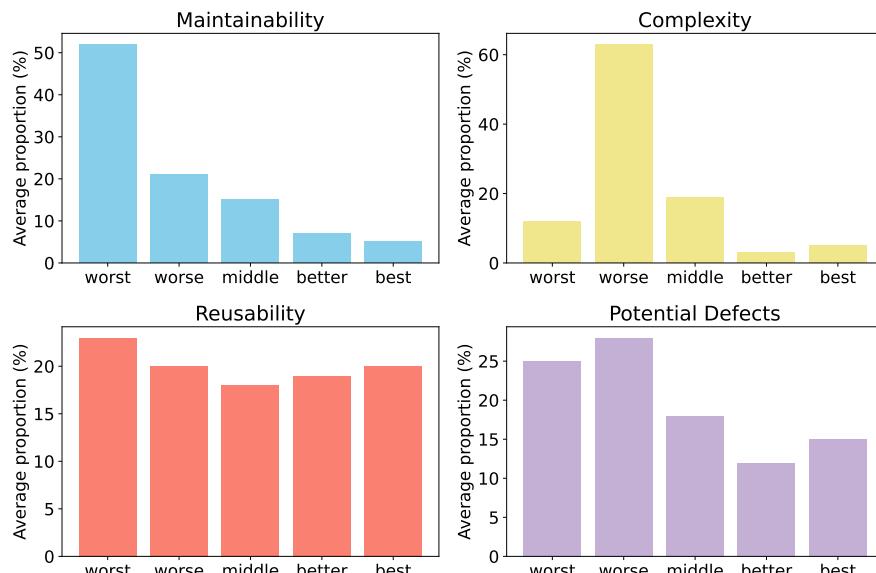


图 4-13 代码质量度量 - 方法变更比例

通过表可以看出，在修复或重构等操作过程中，大多数变更都集中在代码度量较差的部分。这表明，度量较差的代码相较于度量较好的代码，在重构或修复的提交中更容易被修改。这一现象进一步验证了代码质量度量模型的有效性，它能够在项目中识别出质量较差的代码区域，从而帮助开发人员优先关注这些部分，在提升整体代码质量时有针对性地进行优化和重构。因此，代码质量模型不仅提供了对现有代码质量的量化评估，还能为项目团队提供具体的优化方向，确保资源投入最大化地改善代码质量。

### 2. 针对于 RQ2 的实验

针对 4.4 节总结的不良的图模式，在示例项目中均有对应模式的代码审查图子图。这里以 TheAlgorithms 项目和 jemmaloc 项目中实际出现过，后又被重构行为频繁变更的子图为例，说明了代码审查图可以帮助用户发现此类不良的代码结构。

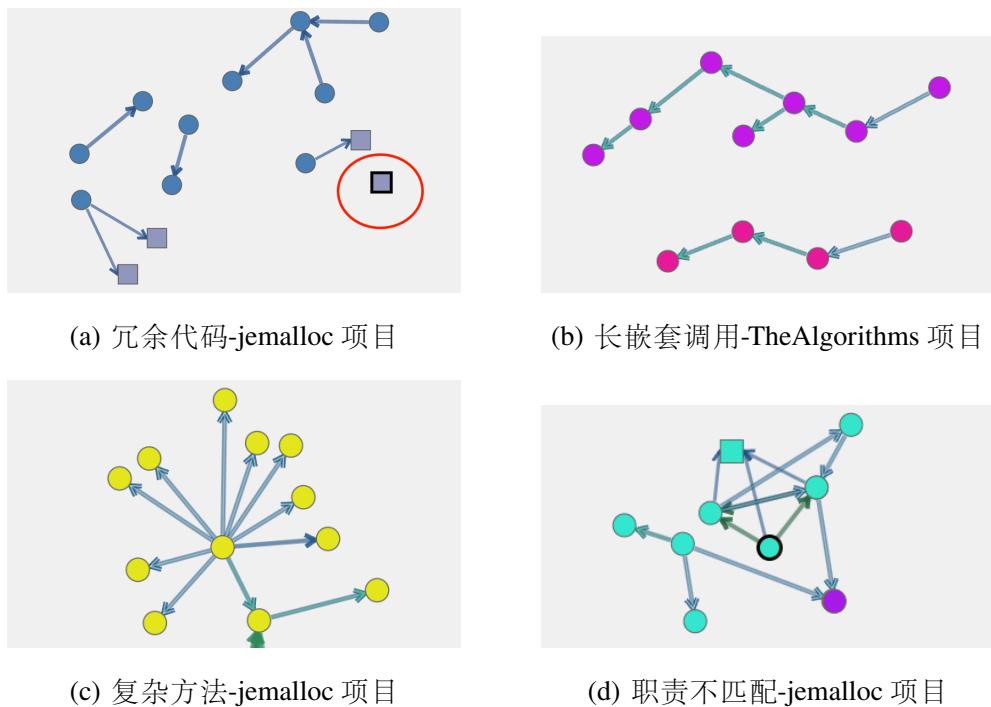


图 4-14 代码审查图

图 (a) 是 jemalloc 项目中的实际存在的子图，该图为 src/san.c 文件的代码审查图。该文件定义了 4 个变量和 10 个方法，从图中可以清晰地看出该文件中存在冗余的全局变量 opt\_lg\_san\_ucf\_align，同时该文件的结构非常松散，缺乏足够的联系和协调关系，这种结构也导致了其内聚度较差。仅依靠代码或文字难以直接反映问题，而配合代码审查图，可以更加直观地展示该模块的结构问题。

图 (b) 是 TheAlgorithms 项目中 cipher/affine.c 文件和 math/euclidean\_algorithm\_extended.c 文件的部分子图，这两个文件内均出现了长嵌套调用，当中间任何一个方法产生变更时，都可能导致上下游的方法跟着改变，因此变更较为频繁。

图 (c) 是 jemalloc 项目中 unit/fxp.c 文件存在的复杂节点 TEST\_BEGIN 方法，该节点共调用了 10 个其他方法，导致其在外部方法变更的同时也要随之变更，因此较为频繁地出现变更行为。

图 (d) 是 jemalloc 项目中 src/hpa.c 文件声明的方法 hpa\_supported，

而它并未与其所在文件的其他方法产生联系，而是更多地服务于 unit/hpa\_background\_thread.c 文件内的方法，因此出现了职责不匹配的现象。

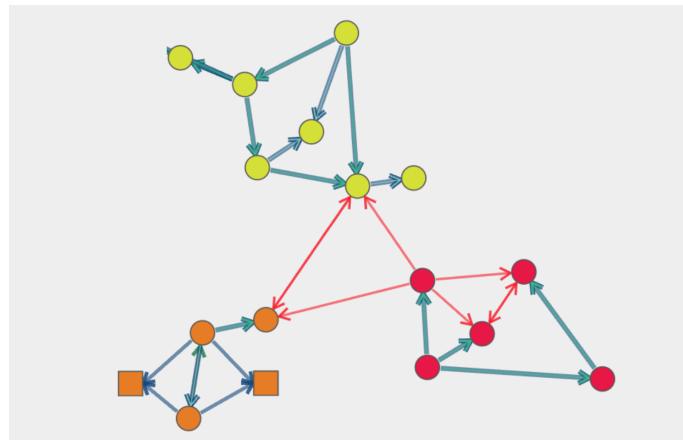


图 4-15 模块间不良的逻辑型变更影响实例

图4-15展示了 TheAlgorithms 项目中的一个实际子图，涉及三个模块。尽管每个模块的内部结构良好，但各有一个方法与其他模块存在逻辑型影响。因此当这些方法发生变动时，不仅当前模块的代码需要修改，其他模块的代码也被影响。更为严重的是，随着变更的涟漪效应，这些变更可能扩展并影响更多的代码，导致不必要的连锁反应。

总而言之，通过代码审查图，不仅能够帮助开发者快速定位模块质量问题，还能直观地理解问题的根源，为后续的优化和改进提供清晰的建议。这种结合文字与图形的分析方式，显著提升了代码质量评估的直观性和说服力。

## 4.6 代码审查图的应用案例分析

代码审查图在实际使用时可以有以下三种用法。

**作为静态分析工具对软件代码进行结构可视化和质量分析** 用户可将代码生成代码审查图，观察软件结构。

图4-18展示了 Antiword 项目和 TheAlgorithms 项目的代码审查图。

图中圆形节点表示方法，方形节点表示全局变量，不同颜色的边则代表了代码元素之间的不同关系：蓝色边表示依赖关系，绿色边表示耦合关系，红色边表示代码变更影响关系。图 a 和图 b 是未区分模块的全局视图，图 c 和图 d 则对模块进行了区分，采用颜色区分不同模块，将属于同一模块的节点用相同颜色进行标注，从而进一步突出模块之间的边界和逻辑关系。

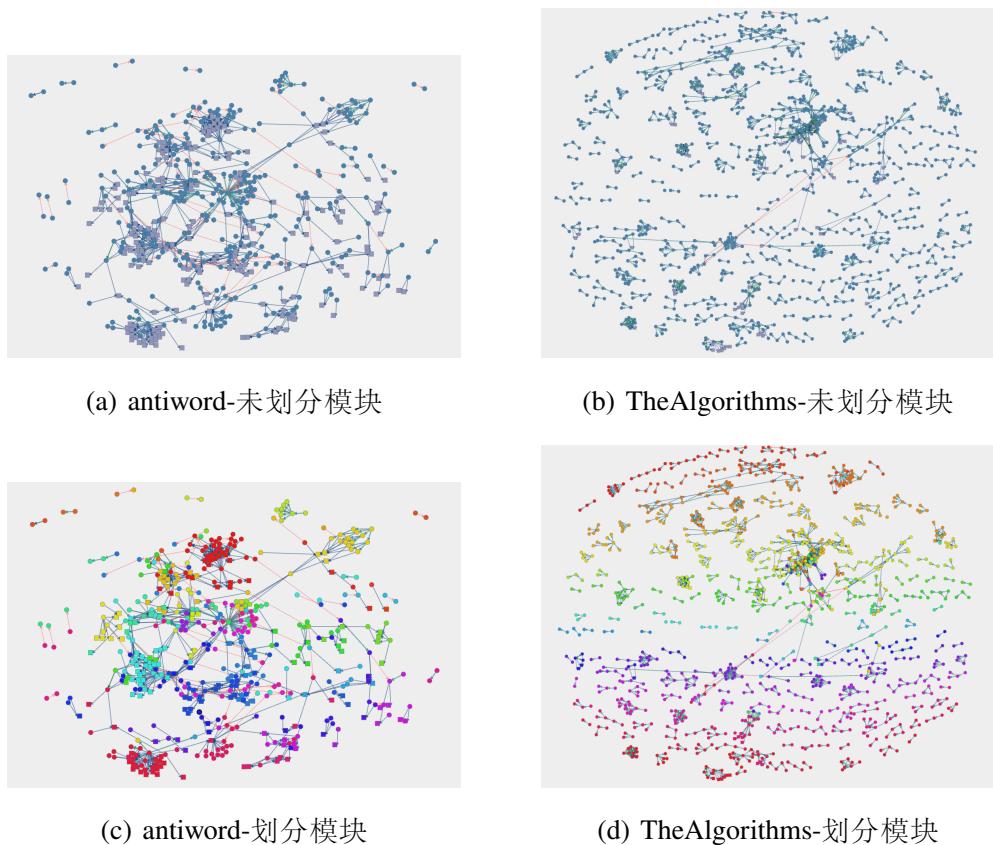


图 4-16 代码审查图

### (1) 分析结构特征信息

从图中可以明显看出两个项目在结构特征上的显著差异。

- **Antiword** 项目整体模块之间高度协作，共同实现一个完整功能，因此模块之间的联系较为紧密，表现出较强的耦合性。从可视化图上观察，按模块划分后，可以清晰地看到具有相同功能的模块形成了较为紧密的聚集。同一颜色的节点集中分布，进一步体现了模块的内聚性较高以及逻辑结构的清晰性。
- **TheAlgorithms** 项目由于其作为算法库的特性，方法之间的耦合性较低，各模块间的联系相对较弱。从图上来看，不同模块呈现出较为分散的分布，模块内部的聚集程度也较低。整体结构表现为由若干独立模块组成，松散而分离，符合库函数式项目的典型特征。

### (2) 查看代码度量

点击图中的某个节点可查看该节点的详细信息，包括方法或全局变量的具体描述，以及代码度量信息。这一功能能够帮助用户快速了解特定方法或变量的功能和用途，并根据代码度量信息了解方法或所在模块的质量情况，从而更高效地进行代码审查和理解。

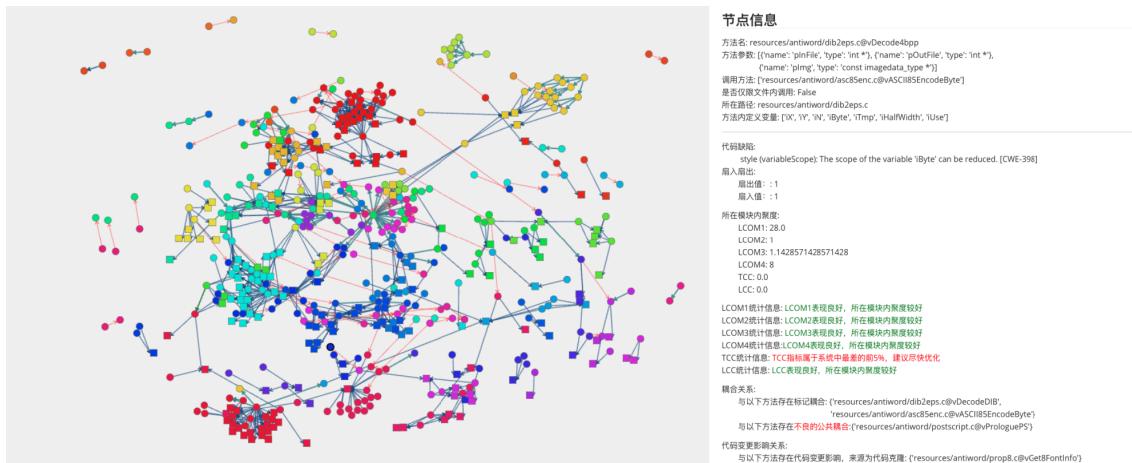


图 4-17 点击节点展开节点信息

### (3) 分析不良的图模式

分析 Antiword 项目，可以发现一些不良的图模式

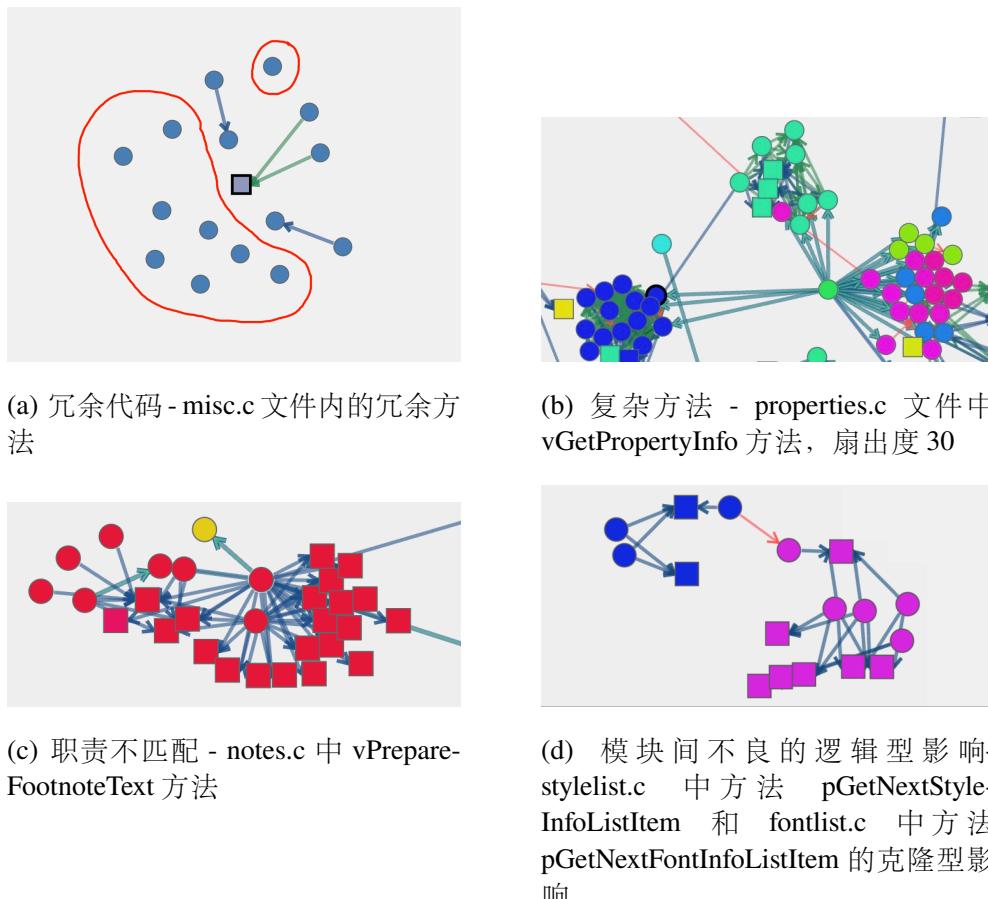


图 4-18 代码审查图

**作为开发工具辅助开发者的代码维护** 当用户对软件代码进行开发时，也可按以下步骤辅助代码维护。

(1) 将复杂庞大的项目先通过系统进行分析，得到对应的代码审查图，如图4-19所示。

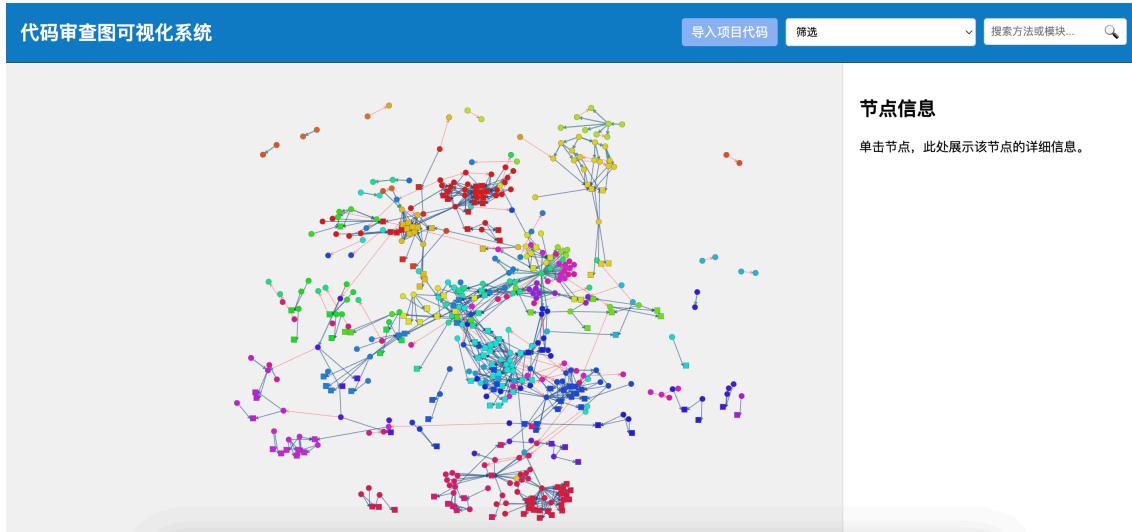


图 4-19 导入项目生成代码审查图

(2) 对于开发任务所在的代码上下文，通过搜索方法名找到其在代码审查图中的位置，点击节点查看方法详细信息和质量度量情况，如图4-20所示。

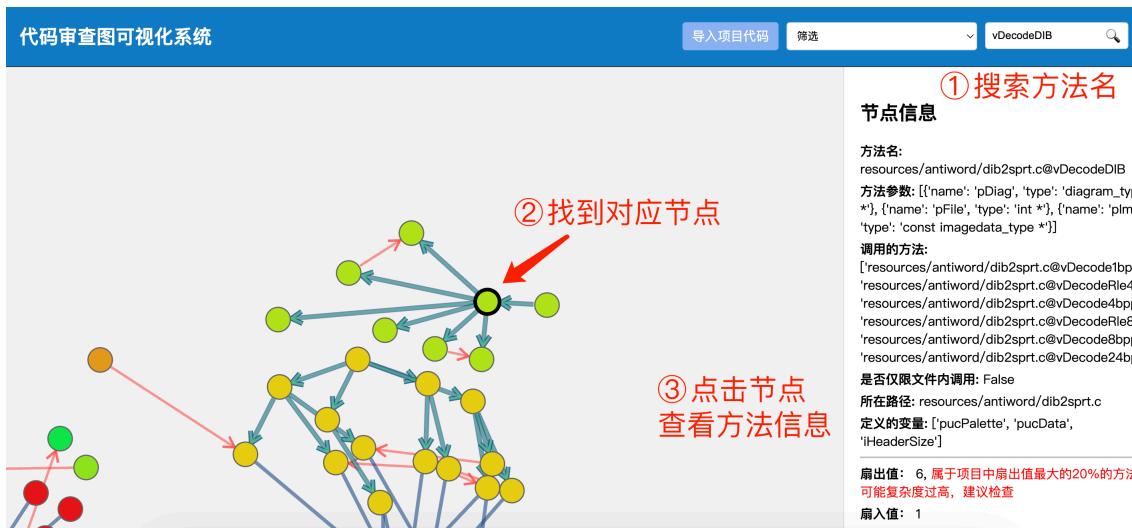


图 4-20 定位开发任务涉及的方法，查看质量信息

(3) 根据代码审查图的边，则可以了解当前代码与其他部分的依赖关系、耦合关系和变更影响关系，如图4-21所示。尤其是变更影响关系，可以帮助用户在进行变更时，提示其依赖型和逻辑型影响的范围，并根据给出的建议，帮助用户安全变更，解决了用户面对复杂软件难以理解、不敢变更、容易变更不完全

的痛点。

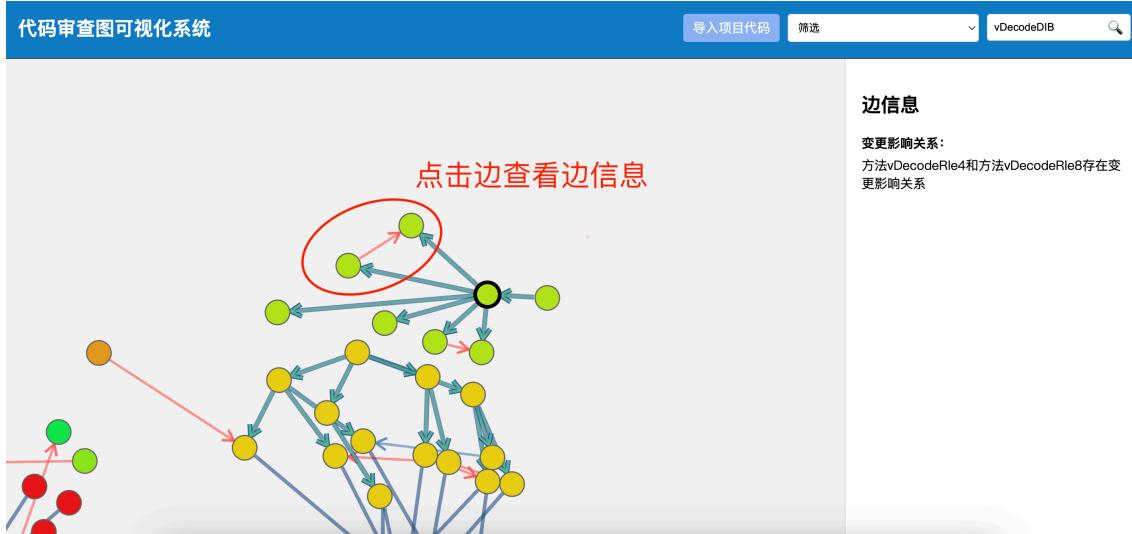


图 4-21 查看方法对应边，按建议安全变更

这里聚焦于 antiword 项目中的 vDecodeDIB 方法的代码开发场景。若按照传统方法对该方法进行开发或分析，人工阅读代码需要涉及 300 多行代码的逻辑才能全面掌握方法的上下文。然而，通过代码审查图，开发者只需关注 8 个节点和 9 条边，即可快速获取关键信息。这种直观的可视化显著降低了代码理解的复杂度和成本。

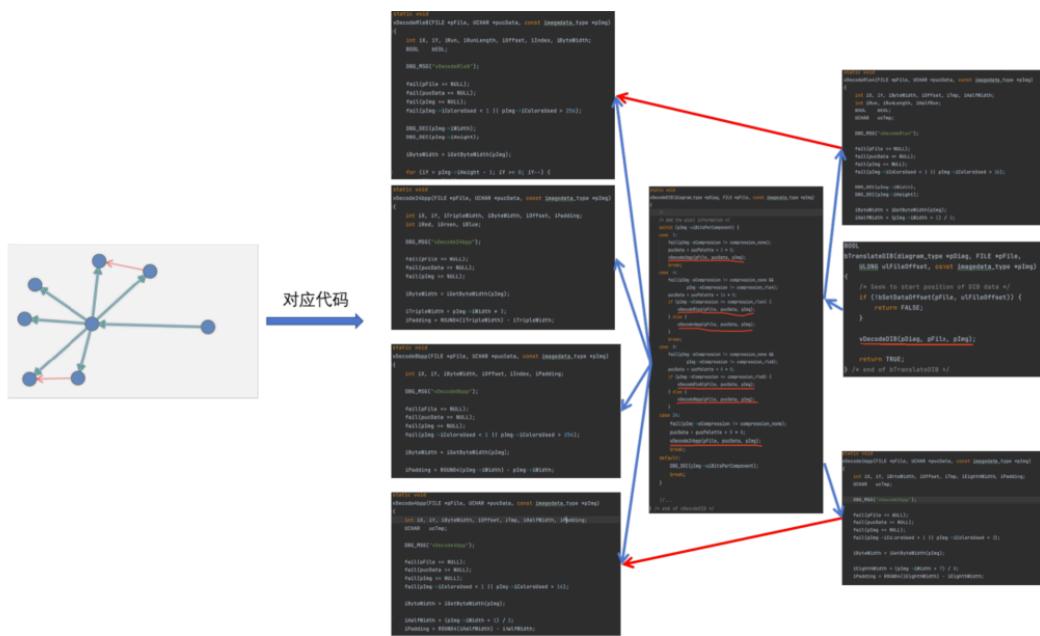


图 4-22 vDecodeDIB 方法上下文

尤其值得注意的是，该方法的上下文中包含了两个由于克隆代码导致的变更影响关系。这类关系通常隐匿于代码中，开发者仅靠手动查看代码很难准确

发现并更改。代码审查图将这些隐藏的逻辑变更影响关系显式标出，使开发者在代码修改时能够快速识别潜在的影响范围，从而避免遗漏变更的风险。

**作为审查工具帮助审查者进行审查** 当面对审查任务时，审查者可通过类似开发的过程，搜索代码审查图定位到当前提交所涉及的节点，通过图中的关系迅速了解代码上下文间的调用以及变更影响关系，通过系统的提示，对开发者的变更进行审查，检查其功能逻辑上是否安全变更，检查其变更操作给软件带来的质量影响是恶化还是优化，从而给出对应的审查结果。

## 4.7 本章小结

本章首先构建了一个代码质量度量模型，从四个维度对软件内部代码的相对质量展开评价。随后提出以代码审查图呈现软件结构和代码质量度量结果的可视化方式。随后，基于代码审查图开展代码质量分析，归纳出五种不良的图模式。借助这种方式，用户不仅能够更直观地把握代码结构，还能从宏观层面洞悉代码质量欠佳的缘由。通过实验验证了该代码质量度量模型能够切实有效地反映代码在重构过程中凸显的问题，进而证明了代码审查图在实际应用中的重要价值。最后，以案例分析的形式，展示了代码审查图在实际运用中的具体方法与显著优势。

## 结 论

随着软件系统规模和复杂性的不断增加，确保代码质量已成为一个日益重要的挑战。传统的依赖经验丰富的专家进行人工代码审查的方法，已经难以满足现代软件开发，特别是大型系统中的需求。随着系统的复杂性增加，代码结构往往逐渐退化，导致系统的维护和管理变得更加困难。因此，自动化、高效的代码质量评估方法显得尤为迫切。本文面向代码质量评估，对代码变更影响分析方法进行研究，取得了如下成果：

(1) 本文通过使用 Clang 提取了三种代码中间表示形式——抽象语法树、方法摘要表和全局变量信息表，并进一步计算了 12 种质量度量。这些度量包括基于内聚度缺乏度和连通度内聚度的 6 项度量，4 种耦合关系度量以及 2 个代码复杂度度量。同时，结合静态分析工具对代码缺陷进行检测。研究结果表明，这些度量指标具有较高的准确性，能够有效帮助开发者全面了解软件的质量状况。

(2) 本文实现了基于传统依赖闭包的代码变更影响分析方法，并提出了三种新的变更影响分析方法。基于代码克隆的方法通过检测软件项目中的代码克隆情况，反映出代码变更影响关系。基于数据挖掘的方法通过挖掘代码变更历史，检测历史中频繁共同更改的方法，提示用户变更影响，避免变更不完全。基于深度学习的方法利用数据挖掘中提取的数据作为数据集进行训练，能够在没有变更历史的情况下，预测代码变更的影响关系。实验结果表明，这三种方法均在不同的角度优于传统方法，并且能有效弥补传统方法只能挖掘基于依赖关系的变更影响关系的不足。

(3) 本文提出了代码审查图的方式，将代码质量分析结果展示给开发者，方便开发者或审查者从宏观的角度了解软件项目架构，聚焦特定模块，减少上下文阅读。本文提出了基于大语言模型的方法模块预测方法，帮助用户识别模块错误划分的方法。除此之外还生成详细的代码质量检测报告，为开发人员提供了清晰的项目质量状况概览。

但本文的研究方法依然存在一些不足，在未来工作中，可考虑进一步在以下方面进行研究：

(1) 对于大规模软件项目，代码审查图可能过于复杂，信息难以辨识。因此，可以考虑分层展示细节，首先按模块级、方法级等不同层级构建图，用户

可以通过点击模块节点查看更为详细的子图，从而实现信息的逐层递进展示。

(2) 基于大语言模型的方法模块划分可进一步尝试融合聚类和大模型预测的方法，提高准确性。

## 参考文献

- [1] KUMAR A, GILL B S. Maintenance vs. reengineering software systems[J]. Global Journal of Computer Science & Technology, 2012.
- [2] DAI P, WANG Y, JIN D, et al. An improving approach to analyzing change impact of c programs[J]. Computer communications, 2022(Jan.): 182.
- [3] CHEN W, IQBAL A, ABDRAKHMANOV A, et al. Large-scale enterprise systems: Changes and impacts[J]. lecture notes in business information processing, 2013.
- [4] ARNOLD R S. Software change impact analysis[M]. Washington, DC, USA: IEEE Computer Society Press, 1996.
- [5] ZHANG X, GUPTA R, ZHANG Y. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams[C]//Software Engineering, 2004. ICSE 2004. Proceedings. 2004.
- [6] GALLAGHER K B, LYLE J R. Using program slicing in software maintenance[J]. IEEE Transactions on Software Engineering, 1991, 17(8): 751-761.
- [7] JITENDERKUMARCHHABRA, MRINAALMALHOTRA, JITENDERKUMARCHHABRA, et al. Improved computation of change impact analysis in software using all applicable dependencies[J]. Springer, Singapore, 2018.
- [8] GETHERS M, KAGDI H, DIT B, et al. An adaptive approach to impact analysis from change requests to source code[C]//IEEE/ACM International Conference on Automated Software Engineering. 2011.
- [9] SUN X, LI B, WEN W, et al. Analyzing impact rules of different change types to support change impact analysis[J]. International Journal of Software Engineering & Knowledge Engineering, 2013, 23(03): 259-288.
- [10] DEPARTMENT, OF, SOFTWARE, et al. Impact analysis in the presence of dependence clusters using static execute after in webkit[J]. Journal of Software: Evolution and Process, 2013, 26(6): 569-588.
- [11] SUN X, LI B, TAO C, et al. Change impact analysis based on a taxonomy of change types[C/OL]//2010 IEEE 34th Annual Computer Software and Applications Conference. 2010: 373-382. DOI: 10.1109/COMPSAC.2010.45.

- [12] UFUKTEPE E, TUGLULAR T. Code change sniffer: Predicting future code changes with markov chain[C]//IEEE Annual Computers, Software, and Applications Conference. 2021.
- [13] ZHEHENG L, JIANMING C, WUQIANG S, et al. Ecia: Elaborate change impact analysis based on sub-statement level dependency graph[C/OL]//2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C). 2023: 471-480. DOI: 10.1109/QRS-C60940.2023.00032.
- [14] ZHANG Z, LIU L, CHANG J, et al. Commit classification via diff-code gcn based on system dependency graph[C/OL]//2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS). 2023: 476-487. DOI: 10.1109/QRS60937.2023.00053.
- [15] SHAKIRAT Y, BAJEH A, ARO T O, et al. Improving the accuracy of static source code based software change impact analysis through hybrid techniques: A review[J]. Universiti Malaysia Pahang Publishing, 2021(1).
- [16] HUANG L, SONG Y T. Precise dynamic impact analysis with dependency analysis for object-oriented programs[C]//Acis International Conference on Software Engineering Research. 2007.
- [17] CAI H, SANTELICES R. A comprehensive study of the predictive accuracy of dynamic change-impact analysis[J]. Journal of Systems and Software, 2015, 103: 248-265.
- [18] CAI H, SANTELICES R, XUT. Estimating the accuracy of dynamic change-impact analysis using sensitivity analysis[C]//Eighth International Conference on Software Security & Reliability. 2014.
- [19] CAI H, THAIN D. Distia: a cost-effective dynamic impact analysis for distributed programs[C]//the 31st IEEE/ACM International Conference. 2016.
- [20] 王海龙, 姜华, 吴晓雯. 一种基于代码树分析的代码影响范围分析方法[Z].
- [21] MARKUS, BORG, KRZYSZTOF, et al. Supporting change impact analysis using a recommendation system: An industrial case study in a safety-critical context[J]. IEEE Transactions on Software Engineering, 2017.
- [22] HATTORI L, JR G P D S, CARDOSO F, et al. Mining software repositories for software change impact analysis: a case study[C]//Brazilian Symposium on Databases. 2008.

- [23] ZANJANI M B, SWARTZENDRUBER G, KAGDI H. Impact analysis of change requests on source code based on interaction and commit histories[C]//ACM. 2014.
- [24] ROLFSNES T, ALESIO S D, BEHJATI R, et al. Generalizing the analysis of evolutionary coupling for software change impact analysis[C]//IEEE International Conference on Software Analysis. 2016.
- [25] ZIMMERMANN T, ZELLER A, WEISSGERBER P, et al. Mining version histories to guide software changes[J]. IEEE Transactions on Software Engineering, 2005.
- [26] HUANG Y, JIANG J, LUO X, et al. Change-patterns mapping: A boosting way for change impact analysis[J]. IEEE Transactions on Software Engineering, 2021, PP (99): 1-1.
- [27] LLVM-ADMIN TEAM. Clang, a c language family frontend for llvm.[EB/OL]. 2024. <https://clang.llvm.org/>.
- [28] LLVM-ADMIN TEAM. libclang: C interface to clang.[EB/OL]. 2024. [https://clang.llvm.org/doxygen/group\\_\\_CINDEX.html](https://clang.llvm.org/doxygen/group__CINDEX.html).
- [29] GOMARIZ A, CAMPOS M, MARIN R, et al. Clasp: An efficient algorithm for mining frequent closed sequences[C]//Advances in Knowledge Discovery and Data Mining: 17th Pacific-Asia Conference, PAKDD 2013, Gold Coast, Australia, April 14-17, 2013, Proceedings, Part I 17. Springer, 2013: 50-61.
- [30] LEWIS P, PEREZ E, PIKTUS A, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks[Z]. 2020.
- [31] 花子涵, 杨立, 陆俊逸, 等. 代码审查自动化研究综述[J]. 软件学报, 2024, 35 (7): 3265-3290.
- [32] 黄沛杰, 杨铭铨. 代码质量静态度量的研究与应用[J]. 计算机工程与应用, 2011, 47(23): 61-63.
- [33] CHIDAMBER S R, KEMERER C F. A metrics suite for object oriented design[J]. Software Engineering IEEE Transactions on, 1994, 20(6): 476 - 493.
- [34] HENDERSONSELLERS B, CONSTANTINE L L, GRAHAM I M. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)[J]. Object Oriented Systems, 1996, 3(3): 143-158.
- [35] HITZ M, MONTAZERI B. Measuring coupling and cohesion in object-oriented systems[C]//Proc. Int. Symposium on Applied Corporate Computing, Oct. 25-27, 1995. 1995.

- [36] BIEMAN J M, KANG B K. Cohesion and reuse in an object-oriented system[C]// Proceedings of the 1995 Symposium on Software reusability. 1995.

## 哈尔滨工业大学学位论文原创性声明和使用权限

### 学位论文原创性声明

本人郑重声明：此处所提交的学位论文《面向质量评估的多粒度变更影响分析方法研究》，是本人在导师指导下，在哈尔滨工业大学攻读学位期间独立进行研究工作所取得的成果，且学位论文中除已标注引用文献的部分外不包含他人完成或已发表的研究成果。对本学位论文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。

作者签名： 日期： 年 月 日

### 学位论文使用权限

学位论文是研究生在哈尔滨工业大学攻读学位期间完成的成果，知识产权归属哈尔滨工业大学。学位论文的使用权限如下：

(1) 学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文，并向国家图书馆报送学位论文；(2) 学校可以将学位论文部分或全部内容编入有关数据库进行检索和提供相应阅览服务；(3) 研究生毕业后发表与此学位论文研究成果相关的学术论文和其他成果时，应征得导师同意，且第一署名单位为哈尔滨工业大学。

保密论文在保密期内遵守有关保密规定，解密后适用于此使用权限规定。

本人知悉学位论文的使用权限，并将遵守有关规定。

作者签名： 日期： 年 月 日

导师签名： 日期： 年 月 日

## 致 谢

时光飞逝，转眼间在哈尔滨工业大学的两年半学习生活接近尾声，在此过程中得到很多老师、同学和朋友的帮助，在这里向他们表达诚挚的谢意。

感谢我的导师苏小红教授对我的悉心指导，从本科四年级开始就跟着苏老师做课题，苏老师在工作上认真负责、严谨治学，而且待人和善，对学生负责，无论是科研还是为人都是我们学习的榜样。感谢师兄魏宏巍和郑伟宁，在组会中对我的汇报和疑问点明方向。

感谢我的室友，朋友和实验室的小伙伴们，生活上有你们的陪伴，让我度过了快乐的两年半。

感谢我的父母，在我的人生道路上给了我极大的自主权，在我失落时给予安慰，从不施加压力。遇见的人多了之后才发现这有多么可贵，谢谢你们的爱。感谢我的姐姐，你的关心是我在最脆弱时候的强心剂。