

硕士学位论文

(学术学位论文)

面向质量评估的代码变更影响分析及其可视化研究

RESEARCH ON CODE CHANGE IMPACT
ANALYSIS AND VISUALIZATION FOR
QUALITY ASSESSMENT

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

面向质量评估的代码变更影响分析及其可视化研究

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

RESEARCH ON CODE CHANGE IMPACT ANALYSIS AND VISUALIZATION FOR QUALITY ASSESSMENT

Candidate:	Li Meina
Supervisor:	Prof. Su Xiaohong
Academic Degree Applied for:	Master of Engineering
Specialty:	Software Engineering
Affiliation:	Faculty of Computer
Date of Defence:	January, 2025
Degree-Conferring-Institution:	Harbin Institute of Technology

摘要

在软件系统的维护过程中，代码变更不可避免，即使是微小的改动，也可能引发连锁反应，对现有代码质量和系统整体架构产生深远影响。尤其是在大型软件系统中，随着开发团队的扩展和系统复杂度的提升，长期维护可能导致代码结构逐步退化，模块化设计逐渐被复杂的耦合结构取代，从而显著增加系统的维护和管理难度。通过变更影响分析，可在变更发生前识别其潜在影响范围，有效协助开发人员实现一致性变更与安全变更，减少变更带来的风险。此外，结合变更影响分析结果，还能够进一步揭示软件代码结构中的质量问题。

本文以 C/C++ 项目为研究对象，围绕代码变更影响分析方法展开研究，并将变更影响分析结果与质量分析结合起来，通过代码审查图的形式展示。该方法不仅能清晰地揭示代码变更可能造成的影响范围与程度，还能帮助开发人员从宏观层面掌握软件代码结构和质量，从而有效降低系统维护成本和风险，确保软件系统的稳定运行。

首先，本文实现了基于依赖关系、克隆关系和历史共现关系的变更影响分析方法，在分析其局限性的基础上，提出了基于代码预训练模型的变更影响分析方法。实验结果表明，所提方法在多个指标上优于其他方法，尤其在对逻辑型变更影响关系的检测上，取得了显著的提升。

其次，针对基于代码预训练模型方法在实际应用中在计算效率和依赖信息丢失方面的缺陷，本文进一步提出了基于依赖链和检索增强生成的变更影响分析方法。通过依赖链弥补了间接依赖信息的缺失，通过离线构建知识库和在线检索的方式，结合大语言模型在代码语义理解上的优越性能，有效提高了计算效率和检测性能。实验结果表明，该方法进一步提高了变更影响的检测性能，尤其是在间接依赖型变更影响关系上的性能得到了显著提升。

最后，本文提出代码审查图的概念，以代码中的关键元素作为节点，元素间关系作为边，将变更影响分析结果结合代码质量和软件代码结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。此外，还生成了详细的代码质量分析报告。实验结果表明，代码审查图能够帮助开发人员快速聚焦于特定开发模块，有助于其发现不良的代码模式。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: 代码质量; 检索增强生成, 变更影响分析; 代码度量, 代码结构可视化

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	5
1.2.3 其他代码变更影响分析方法	6
1.2.4 现有方法存在的问题与分析	7
1.3 本文的主要研究内容以及各章节安排	8
1.3.1 主要研究内容	8
1.3.2 章节安排	9
第2章 基于关系挖掘和预测的代码变更影响分析	11
2.1 引言	11
2.2 代码预处理和中间表示生成	11
2.2.1 基于 clang 的抽象语法树生成	11
2.2.2 方法调用链提取与分析	12
2.2.3 全局变量定义-使用链提取与分析	13
2.3 基于依赖闭包的变更影响分析	15
2.4 基于克隆检测的变更影响分析	17
2.5 基于共现关系挖掘的变更影响分析	20
2.6 基于代码预训练模型的变更影响预测	22
2.6.1 研究动机	22
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	25
2.7.2 评价指标	27

2.7.3 实验设置	27
2.7.4 实验结果与对比分析	27
2.8 本章小结	33
第 3 章 基于依赖链与检索增强生成的代码变更影响分析	34
3.1 引言	34
3.2 研究动机	34
3.3 整体流程设计	36
3.4 基于中间代码表示和关系挖掘方法的数据构建	37
3.5 基于语义向量的变更影响候选方法检索	39
3.5.1 嵌入模型训练与向量化知识库构建	39
3.5.2 查询嵌入与候选检索	41
3.6 基于依赖链与大语言模型推理的生成方法	41
3.7 实验结果与分析	42
3.7.1 实验数据	42
3.7.2 评价指标	43
3.7.3 实验设置	43
3.7.4 实验结果与对比分析	44
3.8 本章小结	49
第 4 章 基于代码审查图的代码结构可视化和质量分析	50
4.1 引言	50
4.2 基于代码度量的代码质量度量模型	51
4.2.1 代码质量度量模型	51
4.2.2 基于内聚度缺乏度和连通性的的内聚性度量	52
4.2.3 方法间耦合性度量	54
4.2.4 方法扇入扇出度量	56
4.2.5 基于静态检测工具的代码安全性和规范性度量	57
4.3 代码审查图的构建和代码结构的可视化	57
4.3.1 代码审查图构建	57
4.3.2 代码审查图可视化	60
4.4 基于代码审查图的代码质量分析	61
4.5 代码质量分析报告生成	65
4.6 实验结果与分析	65
4.6.1 实验数据和实验设计	65

4.6.2 实验环境	66
4.6.3 实验结果分析	66
4.7 代码审查图的应用案例分析	70
4.8 本章小结.....	76
结 论	77
参考文献	79
哈尔滨工业大学学位论文原创性声明和使用权限	83
致 谢	84

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^[1]。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。

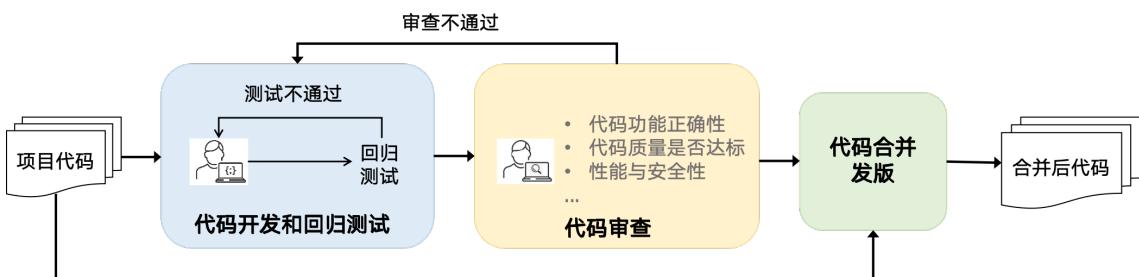


图 1-1 标准代码开发流程

(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, 从变更影响分析的角度出发, 不仅可以深入剖析代码变更对系统整体及各子模块的潜在影响, 还能显著提升项目维护的效率^[2]。然而, 现有的变更影响分析方法大多侧重于基于静态依赖关系的影响分析, 而忽视了逻辑变更关系的复杂性。实际上, 逻辑层面的变更影响往往难以被用户察觉, 却是导致功能性问题的主要原因之一, 这也使得大型系统的变更管理尤为棘手。因此, 如何挖掘和分析逻辑变更关系及其引发的质量问题, 成为亟待解决的关键难题。

尽管变更影响分析能够预测变更的涟漪效应, 并在一定程度上反映代码中的质量问题, 如代码的可维护性等^[3-4], 但现有方法并未有效地将变更影响分析结果与代码质量相结合。此外, 目前的分析结果通常以分散的代码间影响关系呈现, 难以从项目的宏观代码结构上直观展现影响的传播路径及其对代码质量的整体影响。因此, 亟需一种与代码结构深度结合的直观展示方法, 以更高效地帮助开发者理解变更影响的范围和深度, 呈现系统整体的架构和模块间的依赖关系, 从而提升项目维护和质量管理的效率。

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, 帮助开发者更全面地检测变更影响范围, 进而更安全地进行代码维护工作。并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^[5], 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^[6]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^[3]。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，其中静态分析方法中还有一类特殊地关注软件代码变更历史的方法。本文主要从这三方面展开对变更影响分析方法的总结。

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。这类方法主要通过传统的程序分析技术获取代码的语法和语义信息，如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，根据分析粒度的不同可分为过程内和过程间，分别表示分析方法内和方法间的变更影响。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^[7-8] 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^[9-11]。

Schrettner 等人^[12]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^[13]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。

Liang 等人^[14]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。

在静态分析研究中，基于图的分析方法是常见的手段之一。Ufuktepe 等人^[15]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。Peng 等人^[2]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。Yadav^[16]等人提出了一种结合故障预测的变更影响分析方法，利用机器学习技术对面向对象语言的软件进行分析。在其所提出的方法中，采用图的形式构建了一种中间面向对象程序表示，用于检测原始程序和修改后程序之间的差异，随后使用机器学习算法预测并定位类中的故障。

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^[17]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。Jiang 等人^[18]提出了一种基于动态 AST 和 GCN 的代码变更影响范围分析方法。该方法首先对 DAST 的 token 信息进行了拓展，随后，提出了基于 DAST 的代码依赖分析的类型和具体的分析方法。在完成 DAST 节点权重矩阵的构建之后，运用基于 GCN 的代码变更影响范围分析模型，进而确定代码的变更影响范围。根据实验结果，该方法较为有效地分析了方法内变更影响范围。

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^[19]。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误报较高的问题。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。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。

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^[20]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^[21-22]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^[23]。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^[24]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。

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。

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^[10,25-28]。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^[10]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^[27]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^[28]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，在性能上明显优于传统静态分析算法。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^[29]使用传统变更影响方法获得变更初始影响集，将历史变更模式应用于当前的变更影响分析任务，对分析过程进行增强，有效提高了变更影响分析的准确性和效率。在实际应用中，变更影响分析通常需要处理复杂的依赖关系，通过借助历史变更模式，该方法能够提升分析过程的普适性和适应性。

1.2.4 现有方法存在的问题与分析

现有的变更影响分析方法在理论和实践中已取得一定进展，能够较为有效帮助开发者识别代码变更的涟漪效应，并评估其对系统的潜在影响。然而，在实际应用中，这些方法仍然面临一系列复杂挑战，尤其是在对逻辑上的变更影响的检测和与代码质量和代码结构的结合上。

1. 对逻辑上的变更影响挖掘不够深刻

无论是静态方法还是动态方法，其本质上都是通过显式的代码间依赖关系来进行分析，即表现在代码中间表示（如调用图、影响图、抽象语法树等）中，代码元素之间的依赖关系（如图上的连通性、路径的可达性等）。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。在文献^[30]中有提到现有的变更影响分析方法忽视了因代码克隆导致的变更影响，这实际上就是逻辑型变更影响，但只是逻辑型变更影响关系的一种，项目代码仍然存在更多逻辑型的未被揭示的变更影响。正是由于其隐式影响的特点，才导致检测难度大，因此挖掘不够深刻。

而面向软件变更历史的分析方法通过挖掘开发者过往的变更行为模式来识别变更影响，在一定程度上能检测逻辑上的变更影响关系，这是由于过去的变更历史都是由开发者充分理解代码功能和语义的基础上，人工分析后变更，并且经过较为严格的审查环节才完成代码合入的过程，因此其反映的变更影响关系较为可靠和全面。但是这类方法只能利用项目内的变更历史，难以迁移到其他项目中使用。有部分研究通过分析提交信息和变更代码之间的关系来预测潜在影响，但依赖于提交信息的质量，这使得方法的准确性受到影响，尤其在提交信息质量不稳定或缺失的情况下，误差率较高。而且该类无法处理没有变更历史或缺少提交信息的项目，存在明显的局限性。

2. 缺乏与代码质量和代码结构的深入结合

变更影响分析在软件代码维护过程中具有重要意义。如文献^[3]所述，变更影响分析不仅能够优化软件的维护过程，还能提升代码质量与开发效率，同时帮助开发人员更深入地理解软件系统的结构和需求，从而更高效地开展开发与维护工作。然而当前研究在将检测结果与代码结构和代码质量进行深度结合这一方面仍在存在局限性。这一不足体现在多个方面。

首先，现有方法与代码质量的结合不够深入。代码质量涉及可维护性、可复用性、安全性等多个维度，其中代码间的变更影响关系会直接影响到代码的可维护性，但现有研究未能与可维护性指标有机结合。除此之外，不良的隐式

变更影响关系也可能导致用户在变更时遗漏应同步变更的代码，导致代码质量的下降，甚至代码功能逻辑的不一致。

其次，对于软件项目来说，变更影响分析通常是针对整个代码仓库展开的。但是，现有的分析结果往往呈现为过于具体的代码间的关系，没有在宏观的角度上将其与软件代码的结构紧密联系起来。开发人员在面对这些结果时，只能看到一些具体而分散的信息，难以清晰地了解宏观架构层面上的代码结构和内部关系。例如，开发者根据检测结果可得知哪些代码部分可能会受到变更的影响，但无法直观地感知这种影响是如何在代码的模块、方法等结构层次上体现的，无法明确具体的代码元素之间是如何相互关联和相互影响的。缺乏直观的结果展示限制了开发者对变更影响与代码质量关系的理解，难以准确把握变更在代码结构上的传播方式和对代码质量的全面影响。

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

为解决前文变更影响分析方法存在的局限性，本文面向 C/C++ 软件项目，提出基于代码预训练模型和检索增强生成的变更影响分析方法，同时提出基于代码审查图的软件结构和质量可视化方法，旨在帮助用户在软件生命周期的各个环节了解代码各个部分的变更影响关系和代码质量信息，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：基于关系挖掘和预测的代码变更影响分析，基于依赖链和检索增强生成的代码变更影响分析和基于代码审查图的代码结构可视化和质量评估。

(1) 基于关系挖掘和预测的代码变更影响分析

这部分首先将代码转换成抽象语法树（Abstract Syntax Tree，AST）作为中间代码表示，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。在此基础上实现了三种变更影响分析方法。基于依赖关系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。基于克隆关系的方法通过检测代码克隆反映影响关系，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。基于共现关系的方法通过数据挖掘的方式挖掘频繁在变更历史中共同更改的代码对，反映其之间存在代码变更影响关系。在分析这三种方法

局限性的基础上，本文提出基于代码预训练模型的变更影响分析方法，将这三种方法检测得到的影响关系整合为数据集，微调代码预训练模型，通过学习存在变更影响关系的代码之间的模式，对方法间变更影响进行预测。

(2) 基于依赖链和检索增强生成的代码变更影响分析

这部分针对前文方法的局限性，为了弥补基于预测方法的计算效率问题和依赖信息丢失的问题，提出基于代码依赖和检索增强生成的代码变更影响分析方法。该方法共分为三个模块，首先是数据构建模块，构建原始知识库、生成代码依赖图和嵌入模型训练所需要的三元组语料。检索模块首先训练嵌入模型，使具有变更影响关系的方法具有更强的向量相似性。检索时对查询方法进行嵌入，在知识库中检索得到候选的有变更影响关系的方法。最后在生成模块使用大语言模型对候选方法进行判断，得到最终的有变更影响关系的方法。

(3) 基于代码审查图的代码结构可视化和质量分析

本文提出代码审查图的概念，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结构和质量，并做出相应的优化决策。在代码审查图中，节点代表项目中的关键元素，包括方法和全局变量。为了对代码质量进行量化分析，结合变更影响关系，从5个属性和18个度量元来计算代码度量，作为节点的属性。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。

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分析结果还会以代码质量分析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。

第一章为绪论，首先介绍了本文的研究背景和研究现状，分别从静态方法、动态方法和基于代码库的方法三类分析方法总结了变更影响分析现状，进一步分析了这些方法的问题和难点。最后对论文的整体架构进行了介绍。

第二章实现了传统的基于关系挖掘的三种方法，在分析传统方法的局限性的基础上，提出了基于代码预训练模型的变更影响分析方法，最后进行了实验结果和对比分析。

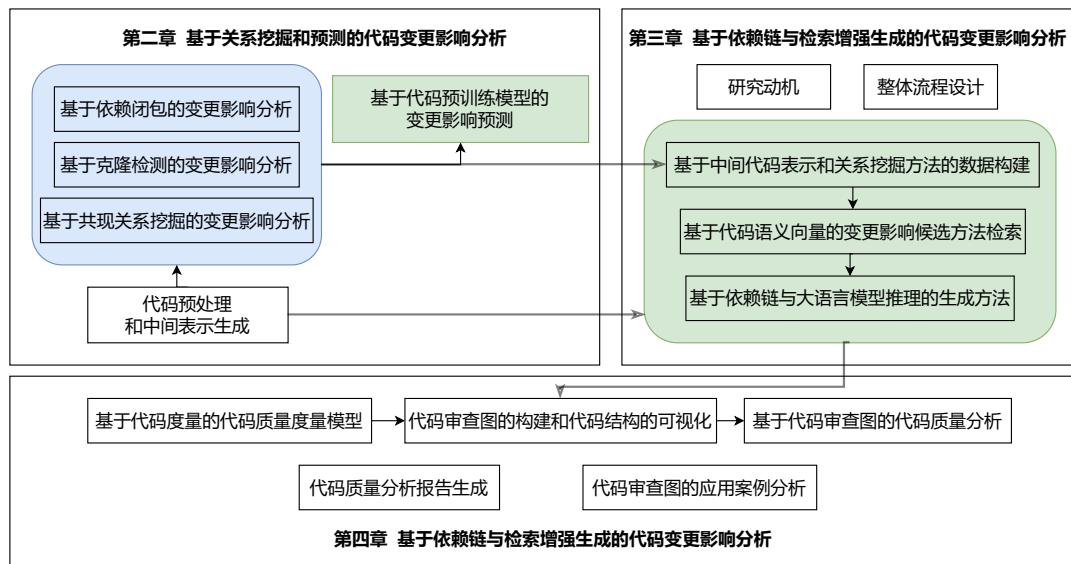


图 1-2 章节安排

第三章在第二章的基础上，提出了基于依赖链和检索增强生成的变更影响分析方法，进一步提高检测的准确率和计算效率，最后通过实验验证了方法的有效性。

第四章提出了代码审查图的概念，将变更影响关系作为节点和边的一部分，通过图可视化引擎 G6 对图进行可视化，并根据可视化的结果进一步对不良的代码质量模式进行总结，最后进行了实验结果展示和分析。

第2章 基于关系挖掘和预测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等方法，得到的静态依赖关系。方法间的依赖型变更影响关系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通过阅读代码进行人工分析，不仅分析难度大，还往往难以被开发者察觉，是导致维护时出现功能性问题的主要原因之一。正是由于逻辑型影响的存在，开发者在维护软件代码时面临着诸多困扰，会出现难以理解软件架构、难以安全进行代码变更等问题。

为了解决上述问题，本文以 C/C++ 项目为研究对象，针对不同的应用场景，实现了基于依赖闭包、基于克隆检测以及基于共现关系挖掘的变更影响分析方法，在研究这三种方法各自的优点和局限性的基础上，提出了基于代码预训练模型的变更影响预测方法，并通过实验验证了基于代码预训练模型的方法的有效性。

2.2 代码预处理和中间表示生成

为了便于后续分析，需要对项目源代码进行预处理，将其转换为适合后续方法处理的中间表示。本文首先将项目源代码解析为抽象语法树（Abstract Syntax

Tree, AST), 并在此基础上进一步提取方法调用链以及全局变量的定义-使用链, 为后续的变更影响分析奠定基础。

2.2.1 基于 clang 的抽象语法树生成

抽象语法树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^[31]。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^[32]。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。

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

本文通过操作游标, 遍历 AST, 获取整个 AST 的结构以及重要节点的详细信息, 在此之上进一步提取代码元素之间的关系。clang 定义的部分重要节点类型如表2-1中所示。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，区分二者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以进行区分。

2.2.2 方法调用链提取与分析

为了进行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。本方法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。

方法调用链提取流程如算法2-1所示。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，排除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。

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。

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。

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

```

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，如果查找到了，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。

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 基于依赖闭包的变更影响分析

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^[2]。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^[19]。该方法主要分为以下两步。

1. 构建依赖关系图

以抽象语法树、全局变量信息表和方法摘要表为基础，构建程序的依赖关系图。本文的依赖关系图是对简单调用图的扩展，增加了方法和全局变量之间的引用关系，方便后续的分析。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。

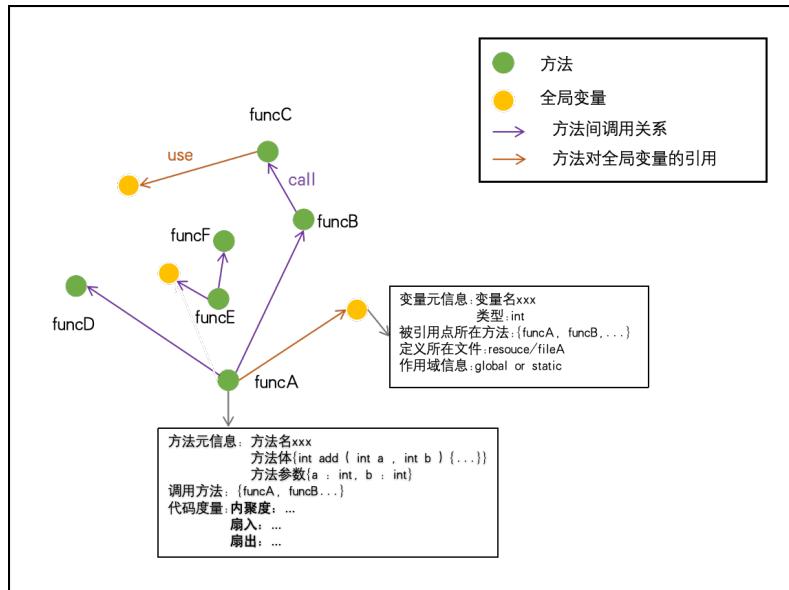


图 2-1 依赖关系图示例

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2-1为依赖关系图示例，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。

2. 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。本文以方法为研究对象，以变更类型为方法变更^[33]，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。

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

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下：

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

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 基于克隆检测的变更影响分析

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。

基于克隆检测识别变更影响的方法在文献^[30]中首次被提出。本文不同于该方法使用机器学习方法进行克隆检测，而是以方法为研究对象，使用频繁模式挖掘技术，以求对克隆代码得到更精准的检测。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

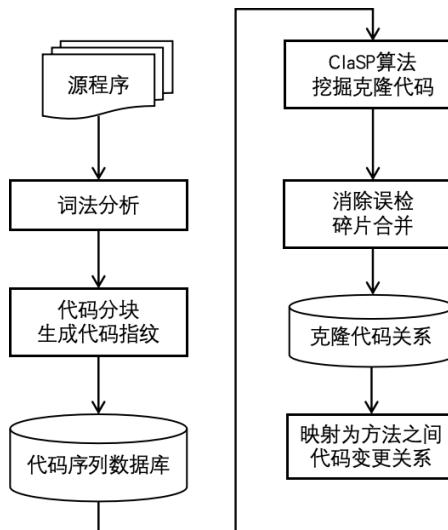


图 2-2 基于克隆检测的变更影响分析方法流程

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2-2所示。

1. 代码预处理

(1) 词法分析。词法分析的主要步骤分为以下几步：

- 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，算法可能在不同的源文件中检测头文件引用的代码克隆情况，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。

(2) 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，识别并根据对应的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。

(3) 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. 基于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顺序频繁出现的模式。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，检测频繁出现的模式，从而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^[34]。具体的流程分为以下两步：

(1) 基于闭合频繁子序列挖掘算法的代码克隆检测。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)。主要的流程如算法2-2所示，接下来是对该算法的详细解释。

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  $FCS \leftarrow \text{N-ClosedStep}(FCC)$ 
```

ClaSP 算法由 Gomariz 等人^[34] 提出，兼具高效性和准确性的优点。在该算法中，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，通过 DFS-Pruning 算法递归地探索相应的子树。DFS-Pruning 算法

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，返回以当前模式 p 为前缀的所有频繁模式集。对所有频率为 1 的序列进行此处理，得到 FCC。

最后去除 FCC 中出现的非闭合序列。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。

(2) 基于克隆代码碎片合并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 基于共现关系挖掘的变更影响分析

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。

1. 基于代码变更历史提取的序列数据库构建

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。具体步骤如下：

- 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。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. 基于共现关系挖掘的变更影响关系提取

基于关联规则（Association Rules）的共现关系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，计算公式如下：

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

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，计算公式如下：

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

在前文所述的基础上，本文设计了如算法2-3所示的频繁共现变更方法对挖掘算法，算法的基本思想是通过挖掘在序列数据库中（即代码提交历史中）中

算法 2-3 频繁共现变更方法对挖掘算法

Input: 序列数据库 D , 支持度阈值 min_sup , 置信度阈值 min_conf

Output: 变更影响方法对 $change_impact_pairs$

```

1  $F_1 \leftarrow \emptyset$  # 频繁 1 项集
2 for all  $f \in D$  do
3   if  $support(f) \geq min\_sup$  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  $Support(f_1, f_2) \geq min\_sup$  then
10     $F_2 \leftarrow F_2 \cup (f_1, f_2)$ 
11   end
12 end
13  $change\_impact\_pairs \leftarrow \emptyset$ 
14 for all  $(f_1, f_2) \in F_2$  do
15   if  $confidence(f_1, f_2) \geq min\_conf$  then
16      $change\_impact\_pairs \leftarrow change\_impact\_pairs \cup (f_1, f_2)$ 
17   end
18 end
19 return  $change\_impact\_pairs$ 

```

的共现关系, 计算得到频繁同时变更的方法对, 这样的关系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 基于代码预训练模型的变更影响预测

2.6.1 研究动机

尽管前文所述的方法在一定程度上能够进行变更影响分析, 但它们仍然存在一些局限性, 主要表现在以下几个方面:

- 冷启动问题: 当项目代码拥有丰富的变更历史时, 可以通过共现关系挖掘方法提取具有变更影响关系的方法, 然而, 并非所有软件项目都具备足够的变更历史支持。例如, 新项目或缺乏完善版本管理的项目中, 历史变更数据的

匮乏使得该方法难以发挥作用。

- 影响类型覆盖不全：在仅有项目源代码的情况下，共现关系挖掘方法无法直接应用，而仅依赖基于依赖闭包和基于克隆检测的方法，虽然能够识别部分影响关系，但对逻辑型变更影响的挖掘仍显不足。逻辑型影响关系往往是系统中隐蔽性最强，同时对功能正确性威胁最大的部分，其难以被捕获的特性极大限制了现有方法的适用性。
- 影响模式难以迁移：共现关系挖掘方法只能从历史变更中学习已经出现的影响关系，而对于未发生变更的方法，无法有效将已知的影响模式迁移到新的场景中。这种局限性导致模型的通用性和适应性不足，尤其在开发者希望预判未来变更影响时，难以提供可靠的支持。

针对上述问题，本文提出了一种基于代码预训练模型的变更影响预测方法。该方法通过整合前述的依赖闭包、克隆检测和共现关系挖掘方法，构建具有代表性的数据集，并对数据进行清洗和增强，通过微调代码预训练模型用于变更影响关系的预测。这种方法不仅能够弥补历史数据不足的问题，还能通过模型的语义理解能力覆盖更广泛的影响类型。此外，模型能够有效迁移已知的影响模式，从而在更广泛的场景中提供可靠的预测能力。通过这一方法，本文旨在提高变更影响分析的全面性和准确性，为复杂软件系统的维护提供更有力的支持。

2.6.2 数据集来源和数据清洗

1. 数据集来源

为了确保代码预训练模型能够有效识别依赖型和逻辑型两种变更影响关系，本文利用依赖闭包、克隆检测和共现关系挖掘方法生成的数据作为原始数据集。具体来说：(1) 正例样本：从上述方法检测到具有变更影响关系的方法对中提取。(2) 负例样本：在项目中随机采样一部分不具有变更影响关系的方法对，确保正负样本比例的平衡。

通过这种方式构建数据集，不仅能够覆盖多种影响关系，还为模型提供了明确的正负分类基础，有助于提高模型的识别能力。

2. 数据清洗

原始数据中可能包含噪声样本，例如依赖闭包方法生成的依赖方法对未必都具有变更影响关系，而共现关系挖掘方法中的支持度较低样本（如支持度为2）可能因偶然共现而导致误报。这些噪声数据会降低模型的训练效果，因此需要进行严格的数据清洗。

为提高数据质量，本文结合当前主流的大语言模型（Large Language Model, LLMs）进行数据清洗。商业化大语言模型拥有较强的代码语义理解能力，可以通过推理较为准确地识别真实的变更影响关系。具体清洗步骤分为三步：（1）Prompt 设计：为大语言模型提供明确的提示，包括变更影响关系的定义及其可能的表现形式，指导模型对样本中方法间的关系进行语义推理。（2）样本过滤：根据模型推理结果，过滤掉那些可能因误报产生的噪声样例。（3）验证数据一致性：通过对模型输出进行人工抽样验证，确保清洗后的数据集具有较高的准确性和可靠性。

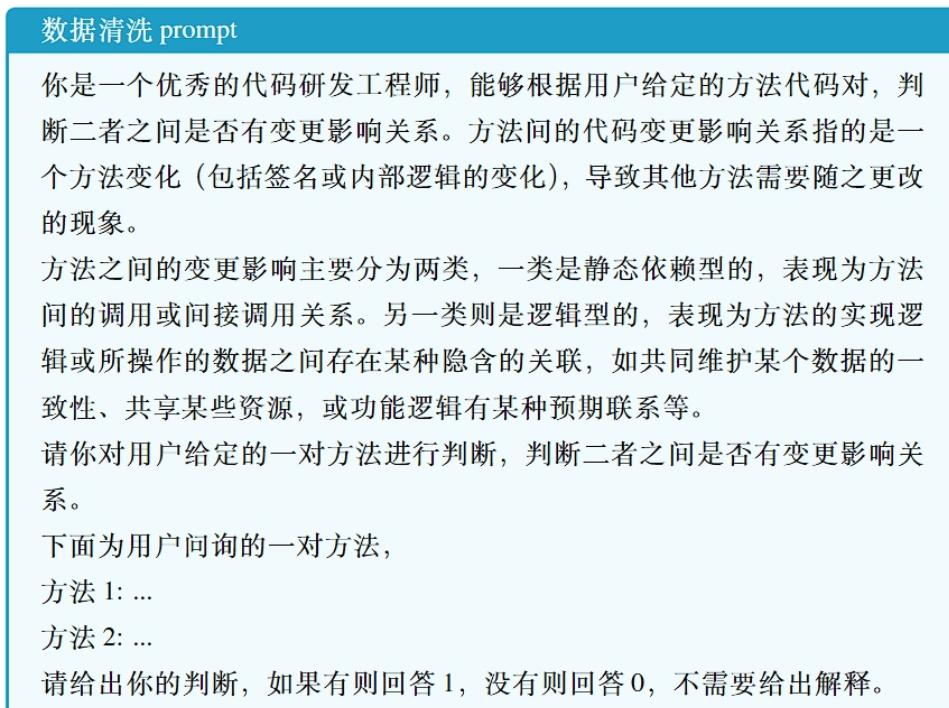


图 2-3 数据清洗所使用的的 Prompt

图2-3展示了本文用于数据清洗的 Prompt 设计。通过结合大语言模型的推理能力，本文显著提高了数据集的质量，为后续模型训练奠定了坚实基础。

2.6.3 基于代码预训练模型的变更影响关系预测

本文提出基于代码预训练模型的变更影响关系预测方法。该任务的输入为两个方法体，通过对两方法之间的变更影响关系的预测，判断是否存在变更影响关系，即输出“存在”或“不存在”。

考虑到代码理解的复杂性与深度，相比于通用领域的如 Bert, T5 等预训练语言模型，面向代码领域进行预训练的 CodeBert 模型更适合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，能够学习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。整个模型架构如图2-4所示。

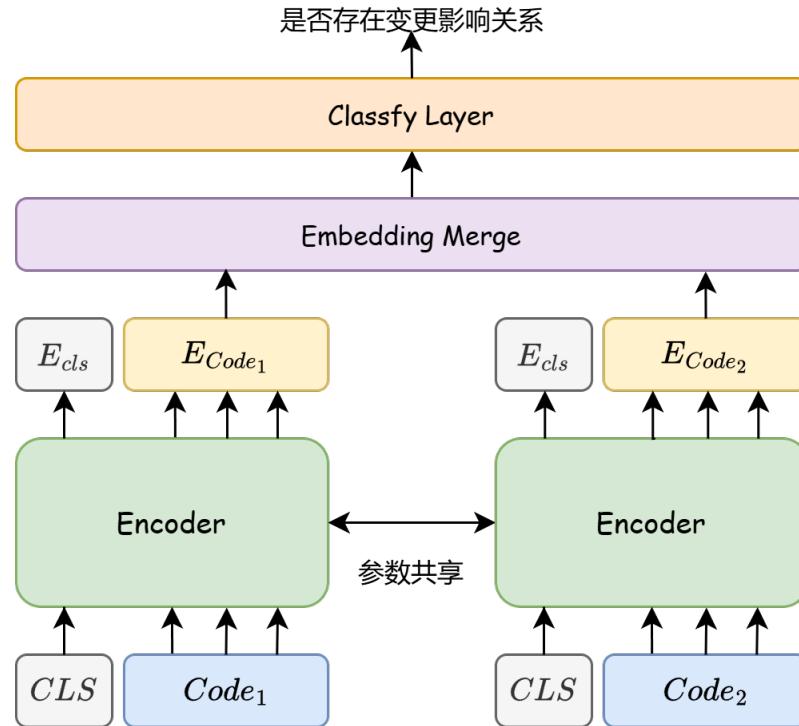


图 2-4 基于代码预训练模型的算法架构

首先，将两个方法体 $Code_1, Code_2$ ，先通过代码预训练模型进行编码

$$H_{Code_1} = \text{Encoder}(Code_1) \in \mathbb{R}^{(len, dim)} \quad (2-5)$$

$$E_{Code_1} = \text{mean}(H_{Code_1}[1 : \dots]) \in \mathbb{R}^{(dim)} \quad (2-6)$$

其中 len 表示 $Code_1$ 的序列长度， dim 为 Encoder 的隐层表示维度，得到方法的向量化表示 E_{Code_1}, E_{Code_2} 。

通过拼接融合这两个向量，得到融合后向量表示 $E_{Code_1, Code_2}$

$$E_{Code_1, Code_2} = \text{Concat}(E_{Code_1}, E_{Code_2}) \in \mathbb{R}^{(dim*2)} \quad (2-7)$$

将融合后的向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。

$$\logits_{Code_1, Code_2} = \text{MLP}(E_{Code_1, Code_2}) = \text{FFN}(\text{ReLU}(\text{FFN}(E_{Code_1, Code_2}))) \quad (2-8)$$

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

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

$$\logits_{Code_1, Code_2} \in \mathbb{R}^{(2)} \quad (2-11)$$

其中, $\logits_{Code_1, Code_2}$ 为模型的 MLP 层最后给出的两种标签 (存在关系, 不存在关系) 各自的概率。

$$loss = CrossEntropyLoss(\logits_{Code_1, Code_2}, Label) \quad (2-12)$$

最后与真实标签 $Label$ 计算交叉熵损失, 得到 loss, 计算梯度, 优化模型参数。

2.7 实验结果与分析

2.7.1 实验数据

1. 实验数据来源

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` 之外, 这些项目还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, 以供共现关系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

2. 训练数据

基于代码预训练模型的变更影响预测方法的训练数据集收集方式如2.6.2节所述, 数据清洗过程中使用的模型为 Doubao (API model name:Doubao-lite-32k-240428), 为了保证测试集和训练集的不重叠性, 在经过收集和清洗后得到的关系中排除测试变更点。得到的训练数据统计信息如表3-1所示, 共 12156 对数

据。

表 2-3 训练数据统计信息

项目名称	正例对数	负例对数	总对数
TheAlgorithms	297	500	797
antiword-0.37	230	500	730
jemalloc-5.3.0	993	2000	2993
libbpf-1.1	801	1500	2301
librdkafka-2.1.0	1432	3000	4432
FFmpegKit-5.1.0	303	600	903
总计	4056	8100	12156

3. 测试数据

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，以代码变更记录作为辅助，以半手工的方式进行标注，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2-4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 2-4 测试数据统计信息

项目名称	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

2.7.2 评价指标

真实的被影响方法表示为 AIS，每种方法检测得到的结果为估计的被影响方法 EIS，按逻辑型和依赖型对关系进行划分，根据这两个值计算精确度、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。

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

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

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

2.7.3 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模型，得到的代码表示为 768 维，融合两组表示后使用的多层 MLP 的维度为 $768*2, 64, 2$ 。骨干模型的学习率设置为 $1e-5$ ，分类器的学习率设置为 $5e-4$ ，优化器 Adam 的两个参数 β_1, β_2 分别设置为 0.95, 0.999, batch_size 为 64。数据收集过程中，共现关系挖掘方法置信度设为 1，支持度设为 2 以获得较多的训练原始数据。

在实际训练过程中，由于 CodeBERTa-small-v1 和 codebert-base-mlm 模型都是类 Bert 模型，允许输入的上下文长度最大为 512，但是经过统计数据集中有 67% 的方法体在分词之后的 Token 数量是大于 512 的，所以本实验在实验中对于超长的方法体按照 512 进行分片，分片进行编码后对多组分片的 *CLS token* 的表示进行平均，来作为方法体的编码表示。

实验使用 PyTorch 和 Transformer 库实现。模型 CodeBERTa-small-v1 和 codebert-base-mlm 均在单个 NVIDIA Tesla V100 GPU 上训练。

2.7.4 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代码预训练模型的变更影响预测方法的性能，这里主要讨论下列三个问题：

RQ1: 本章提出的基于代码预训练模型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（Dependence-Based）和逻辑型（Logic-Based）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的检测是否足够全面？分别适用于哪些特殊场景？又各自有怎样的局限性？

RQ3: 基于代码预训练模型的方法的跨项目迁移表现如何？针对不同代码风格、结构和质量的项目，模型是否能是否能够保持较高的准确性和稳定性？

1. 针对于 RQ1 的实验

四种方法的实验结果如表2-5所示。总的来讲，两个基于代码预训练模型的

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。这说明基于代码预训练模型的方法能够学习到过去变更历史中的行为模式，并能将学习到的变更关系知识进行迁移，用于判断新的影响关系。

表 2-5 变更影响实验结果

方法	F-measure	recall	precision
依赖闭包	36.8	81.9	23.7
克隆检测	3.8	2.3	11.6
共现关系挖掘	52.8	42.3	70.4
依赖 ∪ 克隆 ∪ 共现	47.1	88.4	32.1
codebert-base-mlm	58.8	53.9	64.8
CodeBERTa-small-v1	59.5	55.3	64.4

基于依赖闭包的方法表现为召回率较高而准确率很低，仅为 23.7%。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，这会导致较多误报。



图 2-5 依赖闭包方法迭代路径

如在图2-5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。

其次基于共现关系挖掘的方法的准确率较高，但召回率略低。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共现关系挖掘的结果集中。但是由于一些代码并未存在变更，因此该方法忽略了部分较为稳定的代码之间的变更影响关系，导致了一定程度的漏报。

基于代码克隆检测的方法则整体效果最差。这是由于该方法只能准确的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。本文进一步将三种方法进行结合，尽管结合后召回率为最高，但整体效果表现依旧是基于代码预训练模型为最佳。

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（Dependence-Based）和逻辑型（Logic-Based）的变更影响关系分别进行计算，得到如表2-6的结果，通过对比能更直观地发现不同方法的优势和特点。

表 2-6 两类变更影响关系实验结果

方法	Dependence-Based			Logic-Based		
	F-measure	recall	precision	F-measure	recall	precision
依赖闭包	44.8	100	28.9	-	-	-
克隆检测	-	-	-	31.8	19.1	95.7
共现关系挖掘	56.0	44.6	75.4	57.0	47.3	71.8
依赖 \cup 克隆 \cup 共现	44.1	100	28.3	62.3	53.7	74.1
codebert-base-mlm	61.2	55.1	68.9	70.9	72.6	69.3
CodeBERTa-small-v1	61.9	56.6	68.3	71.7	73.9	69.7

(1) 依赖闭包方法 对于依赖型影响没有漏报，但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。

(2) 克隆检测方法 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。但其缺点在于仅能检测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严重漏报。

为说明克隆检测方法的优势，以其检测到的一对有变更影响关系的方法为例，如图2-6所示。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。

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

克隆代码

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。而克隆检测方法能准确检测出此种影响。

(3) 共现关系挖掘方法 该方法对两种类型的变更影响关系均可检测，且综合表现优于依赖闭包和克隆检测方法。以该方法在 librdkafka 项目中检测到的一对方法为例，如图 2-7 所示，该项目是 Apache Kafka 的一个高性能 C/C++

客户端库，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。



图 2-7 逻辑上有变更影响关系的方法对示例-incr 和 decr

这两个方法是一对典型的协作方法。`incr` 方法负责计数器加一并在计数器从零变为一时初始化资源，而 `decr` 方法则在计数器减为零时释放资源。尽管它们在依赖图上没有直接联系，但由于在功能上互为补充，共同承担了计数器的管理和资源的初始化与释放工作，表现出“镜像”特性。因此若一个方法的实现逻辑发生变化，另一个方法通常也需相应调整，以保持逻辑一致性。这种变更影响关系属于典型的逻辑关联型，体现了该方法在检测逻辑型影响的优秀潜力。

但该方法也存在一定局限性：(1) 要求项目代码必须有变更历史。(2) 挖掘算法的支持度会影响结果。表2-7展示了支持度分别为 2 和 3 时的实验结果。可以看出，支持度越大，误报减少，但漏报增多。这表明，较高的支持度能排除偶然共现的影响，但会限制影响关系的普遍性，导致检测到的关系减少，从而增加漏报。(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，相同的影晌模式之间无法进行迁移。

表 2-7 支持度对共现关系挖掘方法的影响

方法	Dependence-Based			Logic-Based		
	F-measure	recall	precision	F-measure	recall	precision
共现关系挖掘-支持度-2	55.5	51.2	60.7	56.1	58.9	53.5
共现关系挖掘-支持度-3	56.0	44.6	75.4	57.0	47.3	71.8

(4) 基于代码预训练模型方法 该方法在两类变更影响关系上的表现均达到最优，与之前的方法相比，在两类变更影响关系上的 F-measure 分别提升了 5.9% 和 14.7%。该方法有效解决了共现关系挖掘方法在缺乏变更历史时的冷启动问题，且在逻辑型影响的挖掘上能够实现影响模式的迁移。

然而，该方法在依赖型影响的检测上虽也有提升，但相对于逻辑型影响来

说，提升效果较为有限，经过分析发现可能是由于该方法仅依靠两个方法体的特征向量进行预测，不管是直接依赖还是逻辑型依赖，在代码中都存在较为明显的特征，而间接依赖的方法对之间，特征则微乎其微，中间的依赖信息丢失了。因此这样的方法体之间在语义层面的关联性较难识别，从而导致对这种类型依赖关系的识别能力较弱。

此外，由于该方法仅通过方法体对变更影响关系进行预测，因此在实际使用中需要对项目中的所有两两方法的组合进行计算。以 `antiword` 项目为例，该项目中共有 578 个方法体，测试集中的变更点为 30 个。对于每一个变更点，都需要与项目中的所有方法进行一一比对，计算的次数为 $30 * (577) = 17,310$ 次，每次计算均需要通过嵌入和分类过程。大量的计算时间消耗显著影响了该方法的可用性，从而限制了其在大规模项目中的应用效果。

3. 针对于 RQ3 的实验

为了探索基于代码预训练模型在跨项目方面的迁移能力，本节对实验所使用到的六个项目进行划分，将 `TheAlgorithms`, `antiword-0.37`, `librdkafka-2.1.0` 和 `FFmpegKit-5.1.0` 划分为训练集，将这四个项目称为分布内项目，将 `jemalloc-5.3.0` 和 `libbpf-1.1` 划分为测试集，将这两个代码库称为分布外项目，训练时仅使用分布内项目的训练数据，使其更符合应用时的场景设置，即在使用时可能会遇到训练中从未见过的项目。

表 2-8 跨项目迁移能力分析 (F-measure)

方法	<code>TheAlgorithms</code>	<code>antiword</code>	<code>librdkafka</code>	<code>FFmpegKit</code>	<code>jemalloc</code>	<code>libbpf</code>
train on 6 datasets	train	train	train	train	train	train
<code>codebert-base-mlm</code>	59.66	60.43	57.47	57.03	59.67	57.82
<code>CodeBERTa-small-v1</code>	60.23	59.64	60.18	57.87	60.99	58.22
train on 4 datasets	train	train	train	train	test	test
<code>codebert-base-mlm</code>	61.72	61.55	62.03	60.65	42.49*	39.75*
<code>CodeBERTa-small-v1</code>	62.28	65.24	63.45	62.19	41.92*	40.8*

实验结果如表2-8所示，仅在分布内项目上进行训练的模型，在对应测试集上的表现会略有提升，并且在分布外项目的测试集上也能有一定的性能迁移，但相比在六个数据集上进行训练的模型，性能下降在 28.9% 到 31.7% 之间。

4. 方法对比总结

上述四种方法为检测代码变更影响关系提供了多样化的解决方案。每种方法在处理不同类型的变更影响关系时各有侧重，适应于不同的应用场景。然而每种方法也存在一定的局限性。对于这些方法的比较与总结，请参见表2-9。

表 2-9 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
克隆检测	逻辑型-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
共现关系挖掘	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
代码预训练模型	依赖型和逻辑型	两类影响的性能均较好	计算效率问题, 间接依赖丢失问题

2.8 本章小结

本章实现了基于依赖闭包、克隆检测、共现关系挖掘的变更影响分析方法，并提出了基于代码预训练模型的变更影响预测方法。通过实验分析了基于依赖闭包、克隆检测、共现关系挖掘的三种方法的优势以及局限性，实验中基于代码预训练模型的变更影响预测方法在两大类关系的检测中均表现出了最优的性能，并且各个指标上没有明显缺陷，但是由于其对计算量的要求较高，所以在实际场景中的应用价值会有一定程度的折扣。

第3章 基于依赖链与检索增强生成的代码变更影响分析

3.1 引言

由于代码预训练模型优秀的泛化能力和知识迁移能力，第二章中基于代码预训练模型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性能。但是由于其计算效率问题，导致其在实际使用时较为耗时。除此之外，由于该方法仅靠两两方法的代码进行变更影响关系的检测，因此方法代码中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。

检索增强生成（Retrieval Augmented Generation, RAG）技术由 Lewis 等人^[35]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查询（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。

为了进一步提升代码变更影响分析的效率和可用性，本章提出了一种基于依赖链与检索增强生成的代码变更影响分析算法。本方法以代码库中的海量方法为知识库，通过信息检索技术，生成候选方法集合，如果候选方法中有与变更方法存在间接依赖关系的方法，则根据代码依赖图提取调用路径上的中间方法，构造推理信息。利用当前大规模语言模型在各种文本领域上强大的语义理解和生成能力，准确的判断候选集合中的变更影响关系。

3.2 研究动机

上一章节所提出的基于代码预训练模型的变更影响分析方法，尽管在测试基准方面取得了当前最优的效果，然而，其仍然存在两个较为显著的问题，即计算效率问题和间接依赖的语义缺失问题。

计算效率问题：每当需要针对代码库中的一个变更点展开变更影响分析时，

都必须将变更方法与整个代码库的其他方法组对，然后输入模型进行检测。对于规模较大的项目代码而言，这种方法所带来的计算延迟会严重影响其可用性，几乎使其无法正常使用。

针对该问题，可以考虑将对方法的编码与变更关系的分析两个阶段进行解耦。在初始化阶段，运用嵌入模型对代码库的全部方法进行编码并存储。当需要对某个变更点进行分析时，仅需通过向量相似度计算筛选出少量候选方法，利用大语言模型优越的语义理解能力进行其变更影响关系的判断。

间接依赖的语义缺失问题：由于基于代码预训练模型的算法仅依据两个方法的代码来判断变更影响关系，因此当两个方法之间存在间接依赖时，如果不提供中间方法的调用逻辑，相关信息便会缺失，这将导致模型无法合理推断方法之间的变更影响关系。在这种情况下，只有资深工程师凭借对方法功能的了解，结合丰富的经验才能够完成判断，但即便如此，也很难保证判断的正确性。



图 3-1 方法间间接依赖型变更影响关系实例

这里以 `jemalloc` 项目中的一对有变更影响关系的方法为例，说明这类间接依赖型关系的出现原因和判断难度。如图3-1所示，该例中共包含 4 个方法，分别是 `psset_insert`、`psset_stats_insert`、`psset_bin_stats_insert` 和 `psset_bin_stats_insert_remove`，后文简称为方法 A、B、C、D。它们的调用关系为 A → B → C → D。其中方法 D 与方法 A 之间存在变更影响，这是由于在方法 A 中向下游方法传递了变量 `psset` 和 `ps`，在方法 B 和 C 中，均未对这两个变量进行操作，而是继续向下游传递。而在方法 D 中，直接对这两个变量进行了更改，由于这两个变量是指针类型，因此更改直接反映在了对应的内存变量上，而在方法 A 调用返回后的代码逻辑中，存在 6 处直接使用了这两个更改后的变量，一旦这两个变量的操作逻辑在方法 D 中有所改变，那么将会间接影响

到方法 A 中对该变量的访问、操作逻辑，因此方法 A 和方法 D 之间存在由间接依赖导致的变更影响关系，主要表现在数据流传递过程中的变更-访问操作。而方法 B 和 C 与方法 D 之间则不存在变更影响关系，原因就在于其并未访问变更后的变量，只是进行了简单的数据传递。值得注意的是，如果仅依靠方法 A 和方法 D 的方法体，这样的依赖信息是丢失的，尤其是在数据传递的过程中，变量并不一定保持原来的名称，如在方法 B 中到方法 C 的传递中，将变量 `psset→stats.empty_slabs` 转化为了以变量名 `binstats` 标识的变量，判断难度进一步加大。

针对该问题，可以考虑在判断间接依赖的方法之间的变更影响关系时，提供其调用路径中的中间方法，补全缺失信息，以便为模型提供支持，实现准确的判断。

因此，本章提出了基于依赖链与检索增强生成的代码变更影响分析技术。该技术借助检索方式和代码依赖图，分别解决计算效率问题以及间接依赖所带来的语义缺失问题，从而提升变更影响分析在实际应用中的效果。

3.3 整体流程设计

为了解决第二章中基于代码预训练模型方法的计算效率问题与间接依赖语义缺失问题，本章提出了基于依赖链与检索增强生成的变更影响分析方法。图3-2中展示了本章方法的研究框架。本章的方法主要流程分为三个模块：

1. 数据构建模块。主要构建两个部分的数据：（1）依赖关系图。构建项目代码对应的依赖关系图，以支持后续间接调用的检测与查询。（2）原始知识库。将项目代码按方法的粒度进行提取，得到以方法为单元的集合作为原始知识库。（3）嵌入模型训练需要的三元组语料。将完整代码库按照方法的粒度进行切分并作为知识库。以上一章节的依赖闭包、克隆检测、共现关系挖掘三种方法检测出的变更影响关系为原始数据，基于大语言模型与代码依赖图进行数据清洗，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这一垂直领域。

2. 检索模块。在三元组语料上训练嵌入模型，训练后使用嵌入模型对知识库进行预编码。在测试时仅需对查询方法进行编码，并与知识库中的向量计算相似度，找出相似度最高的前 k 个（ $\text{top-}k$ ）作为候选方法。

3. 增强生成模块。构建由查询方法，候选方法集合，调用路径的中间方法（如有）组合成的提示，使用大语言模型对候选方法进行判断，得到最终的有变

更影响关系的方法。

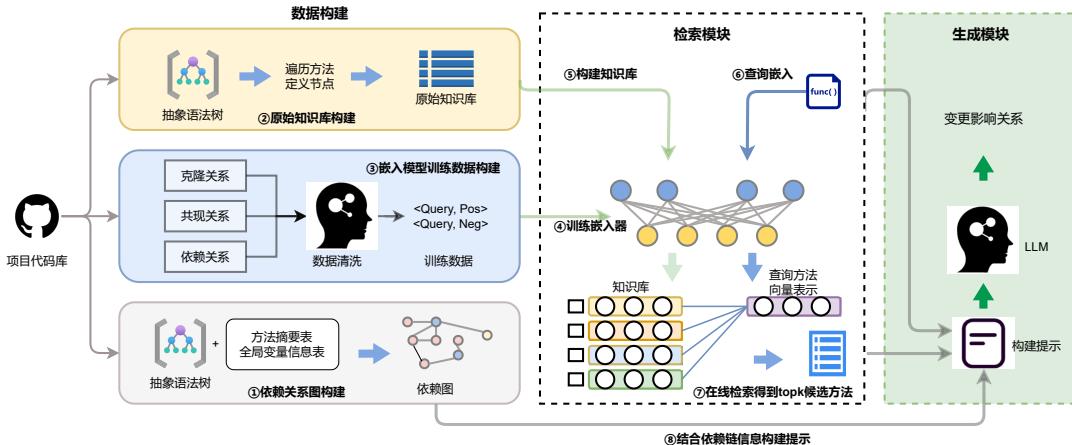


图 3-2 基于依赖链与检索增强生成的变更影响分析方法框架

3.4 基于中间代码表示和关系挖掘方法的数据构建

数据构建主要分为三个部分，分别是依赖关系图构建、原始知识库构建以及嵌入模型训练数据构建。接下来进一步对这三部分数据构建进行介绍。

1. 依赖关系图构建

为在数据清洗过程和生成模块的推理过程中补充对依赖型变更影响关系的间接依赖信息，根据章节2.3所述步骤，基于抽象语法树、全局变量信息表和方法摘要表，构建以方法和全局变量为节点，以方法间的调用关系和方法对全局变量的引用关系为边的依赖关系图。

2. 原始知识库构建

为了将代码以更结构化和可管理的形式存储，便于后续进行嵌入和检索，需要对软件项目代码库进行分块。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。

3. 嵌入模型训练数据构建

为了使嵌入模型专精于检测变更影响关系这一垂直领域，需要对嵌入模型进行训练，使具有变更影响关系的方法在向量空间中更接近，从而检索效果更好。因此需要构建训练嵌入模型所需的训练数据。

这里同第二章中基于代码预训练模型的方法类似，以基于依赖关系、克隆检测、共现关系挖掘三种方法检测出的变更影响关系为原始数据，假设得到的

变更影响关系对集合为 S_{raw} , 其中 $\langle Code_i, Code_j \rangle \in S_{raw}$ 。参考上一章节的实验部分可知, S_{raw} 中存在部分误报关系, 如果不加处理, 会严重影响检索模块的准确性。因此本文使用大语言模型对收集的数据进一步清洗, 以得到足够准确的正例对。

值得注意的是, 第二章中在对数据集进行清洗时仅提示了两个方法的具体信息, 对于存在间接依赖方法的中间调用信息有所缺失, 才导致其对于间接依赖导致的变更影响关系检测效果较差。因此在这一章中我们考虑结合依赖路径进行数据清洗。对于每一对方法 $\langle Code_i, Code_j \rangle$, 数据清洗的具体流程如下:

(1) 判断依赖可达性。根据代码依赖图判断两个方法之间是否存在调用关系, 如果是直接调用, 则无需处理。如果是间接调用, 则需要补充调用路径的中间方法为 $\langle Code_i, Code_{mid_1}, Code_{mid_2}, Code_j \rangle$ 。

(2) 基于大语言模型进行变更影响关系判断。利用代码语义和语言理解能力均较为出色的大语言模型, 为原始数据 S_{raw} 中的每一组数据构建如图3-3的提示, 进行变更影响关系的判断, 剔除大模型认为没有变更影响关系的数据, 剩下的数据整合为 $S_{filtrate}$, 其中 $\langle Code_i, Code_j \rangle \in S_{filtrate}$ 。

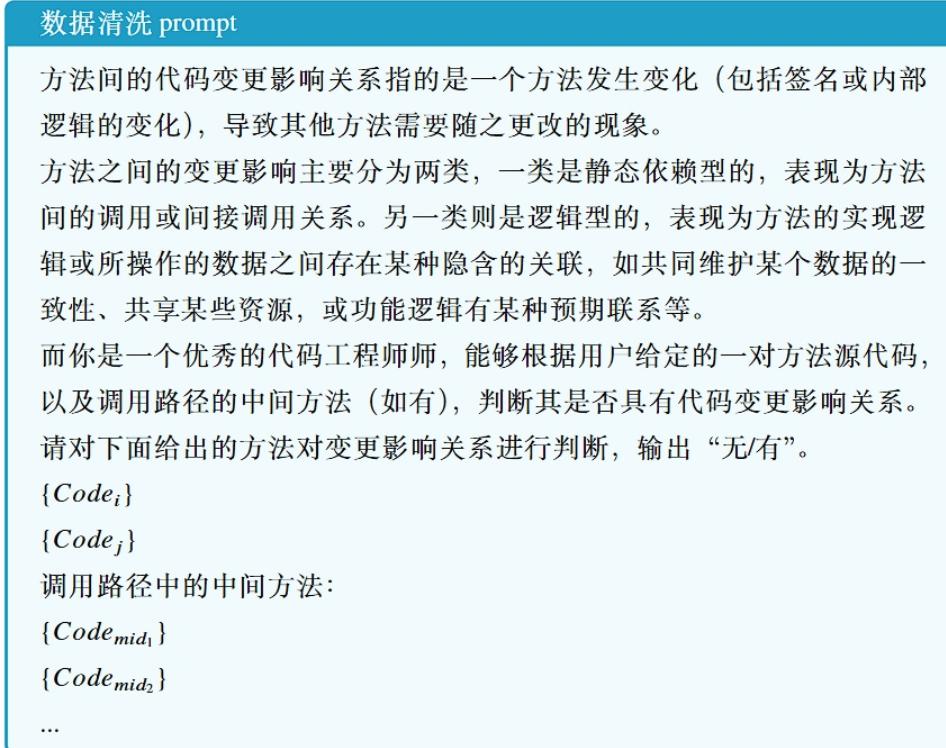


图 3-3 数据清洗所使用的的 Prompt

为训练嵌入模型, 还需构建正例所对应的负例, 得到训练嵌入模型所需的 $\langle Query, Pos, Neg \rangle$ 三元组数据。对于每组正例数据数据 $\langle Code_i, Code_j \rangle$,

随机挑选其中一个方法作为查询 $Code_{query}$, 另一方法作为正例 $Code_{pos}$ 统计 $S_{filtrate}$ 中出现的 $Code_{query}$ 的正例集合 $\{Code_{pos}, Code_k, Code_e, Code_k\}$, 从代码库中除正例集合之外的方法中随机采样方法作为负例 $Code_{neg}$, 将构建好的三元组数据 $\langle Code_{query}, Code_{pos}, Code_{neg} \rangle \in S$ 作为嵌入模型的训练数据, 之后简写为 $\langle Query, Pos, Neg \rangle$ 。

3.5 基于语义向量的变更影响候选方法检索

检索模块主要分为离线和在线两个阶段, 离线阶段涉及到嵌入模型训练与向量化知识库构建, 在线阶段则涉及到嵌入检索与候选生成。接下来详细介绍两个阶段的主要任务和关键步骤。

3.5.1 嵌入模型训练与向量化知识库构建

离线阶段的主要任务是完成嵌入模型的训练, 并对知识库中每个方法进行嵌入。具体而言, 嵌入模型通过对代码库中的每个方法进行编码, 将其转换为具有语义信息的稠密向量表示。这些嵌入向量随后被存储在知识库中, 为后续的检索任务提供数据基础。

1. 嵌入模型训练

本文中嵌入模型用于将方法代码这种长度不一、非对齐的形式转换为固定长度的稠密向量表示, 即将方法的 token 序列 $code = t_1, t_2, \dots, t_n$ 转化为对应的向量表示 E_{code} 。

向量检索的原理是将查询向量与知识库中的向量计算相似度, 返回向量相似度最高的若干条作为候选向量。因此相似度计算的准确性尤为重要。在检测方法间变更影响关系的场景下, 相似度越高则证明方法之间越容易存在变更影响关系, 即这两个方法对应的向量在向量空间中有着更近的距离, 而没有变更影响关系的方法对在向量空间中的距离则更远。

因此在训练嵌入模型时, 对于训练数据中的三元组数据的嵌入 $\langle E_{Query}, E_{Pos}, E_{Neg} \rangle$, 训练目标是将 E_{Query} 与 E_{Pos} 的相似度尽可能地提高, 将 E_{Query} 与 E_{Neg} 的相似度尽可能地降低。本文选择对比损失函数 (Contrastive Loss) 中 InfoNCE Loss (Noise Contrastive Estimation Loss) 作为嵌入模型的损失函数进行训练。其定义为式3-1所示。InfoNCE Loss 是一种用于自监督学习的损失函数, 通常用于学习特征表示或者表征学习。它基于信息论的思想, 通过对比正样本和负样本的相似性来学习模型参数。

$$L_{InfoNCE} = -\log \frac{\exp(sim(E_{Query}, E_{Pos})/\tau)}{\exp(sim(E_{Query}, E_{Pos})/\tau) + \exp(sim(E_{Query}, E_{Neg})/\tau)} \quad (3-1)$$

其中, τ 为温度因子, 用于控制相似度的平滑程度。 $sim(E_{Query}, E_{Pos/Neg})$ 为查询向量和其对应的正例或负例样本的相似度, 在相似度度量上本文选择余弦相似度进行计算, 其定义如式3-4所示。

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

$$A = (x_1, x_2, \dots, x_n) \quad (3-3)$$

$$B = (y_1, y_2, \dots, y_n) \quad (3-4)$$

其中 A 和 B 分别表示方法对的 token 序列编码后的向量表示。通过最小化 $L_{InfoNCE}$ 来优化嵌入模型, 从而提升查询向量与正样本之间的相似度, 同时减少与负样本之间的相似度, 使嵌入模型专精于变更影响关系这一任务。

2. 知识库构建

在训练得到嵌入模型后, 为了提高推理效率, 需要将原始知识库中的方法通过嵌入模型转换为向量表示, 从而构建编码后的知识库。该过程的目的是将每个方法转化为包含语义信息的稠密向量, 以便在推理过程中实现高效检索。具体而言, 利用前文中训练得到的嵌入模型对代码库中的每个方法 $Code_i$ 进行编码, 得到其对应的向量表示 $E_{Code_i} = Embedding_model(Code_i)$, 其中 E_{Code_i} 是一个包含该方法语义信息的稠密向量。编码完成后, 所有方法的向量集合表示为 $\mathbf{E} = \{E_{code_1}, E_{code_2}, \dots, E_{code_n}\}$, 其中 n 是代码库中方法的总数, \mathbf{E} 即为项目代码对应的知识库。

为了提高知识库检索的效率, 本文采用 FAISS (Facebook AI Similarity Search) 作为稠密向量的检索工具。FAISS 是一种高效的向量检索框架, 能够对大规模向量数据进行索引并支持快速检索。通过构建向量索引, FAISS 能够显著提高在大规模数据库中进行相似度搜索的速度。FAISS 支持多种类型的索引结构, 包括平坦索引、倒排文件索引和层次化导航图等, 这些索引结构能够根据数据的规模、向量维度及存储需求, 灵活应对不同查询速度和精度的需求。

鉴于本文的研究对象是软件项目代码, 且分析粒度为方法级别, 代码库中的方法数量通常较少, 最多只有几千个方法。与处理文档类问答生成任务中涉及的大规模数据集不同, 本文的代码库规模相对较小。因此, 在构建索引时, 本

文选择了较为简单的平坦索引结构对方法向量进行检索。

3.5.2 查询嵌入与候选检索

在线阶段主要负责对每个新的查询进行实时处理，该阶段涉及两个关键步骤，旨在为查询方法检索候选的有变更影响关系的方法。以下是对该阶段的详细阐述：

1. 查询嵌入

对于新的方法查询 $Query$ ，使用经过训练的嵌入模型对其进行嵌入操作，将方法转换为可与知识库中存储的向量进行相似度比较的形式。具体来说，通过将 $Query$ 输入到嵌入模型中，得到其对应的嵌入表示，即 $E_{Query} = Embedding_model(Query)$ 。此嵌入表示 E_{Query} 是一个稠密向量，它包含了查询的语义信息，能够有效地表示该查询在高维空间中的位置，为后续的相似度计算和检索过程奠定基础。在这一过程中，嵌入模型发挥着至关重要的作用，它依据自身在训练阶段所学习到的模式和特征，将具有不同语义特征的查询转化为具有独特特征的向量，从而确保在后续的检索和分析中能够准确地找到候选方法。

2. 在线检索

在获得查询的嵌入表示 E_{Query} 之后，利用 FAISS 对知识库中的向量进行检索，由于 FAISS 不直接提供基于余弦相似度的检索方式，根据余弦相似度公式，可以结合其提供的基于点积的检索方式和 L2 范数来达成基于余弦相似度的检索。选择与查询向量 E_{Query} 在余弦相似度上最为接近的 k 个向量。通过这些相似度最高的 k 个向量的索引，可以找到对应的源码方法，将其作为候选方法返回。

$top k$ 值的选择对检索结果的质量与效率有着直接影响。当 $top k$ 值设置得较高时，检索模块能够召回更多的正确关系，因此查全率较高。然而，候选数量的增加会导致后续的大模型需要判断更多的候选方法，导致整个系统的处理速度可能会下降。相反，若 $top k$ 值设置较低，则检索出的候选方法较少，查全率可能会较低，但大模型的计算效率会相对较高。因此， $top k$ 的选择需要根据实际使用场景的需求以及模型的能力进行权衡，以达到查全率与计算效率之间的最佳平衡。

3.6 基于依赖链与大语言模型推理的生成方法

生成模块旨在利用大语言模型出色的自然语言和代码语义理解能力，分析

查询方法 $Code_{query}$ 与检索出来的候选方法集合 $\{Code_1, Code_2, \dots, Code_k\}$ 之间的代码变更影响关系。在进行变更影响判断之前，需要首先通过代码依赖图来判断查询方法与候选方法之间是否存在调用关系，如果是直接调用或逻辑型的依赖，则无需处理。如果是间接调用，则需要补充该调用路径中的中间方法，以形成完整的调用链，如 $<Code_{query}, Code_{1_{mid_1}}, \dots, Code_{1_{mid_2}}, Code_1>$ 。

为根据查询和依赖分析的结果动态构建推理提示，通过槽位填充的方式进行构建，如图所示。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，则说明该组的主要候选方法是与查询方法存在间接调用关系的方法，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。 $[x]$ 表示查询方法， $[z]$ 表示检索到的候选方法，存在间接依赖的方法则对路径信息进行补充，构造得到对应的提示。

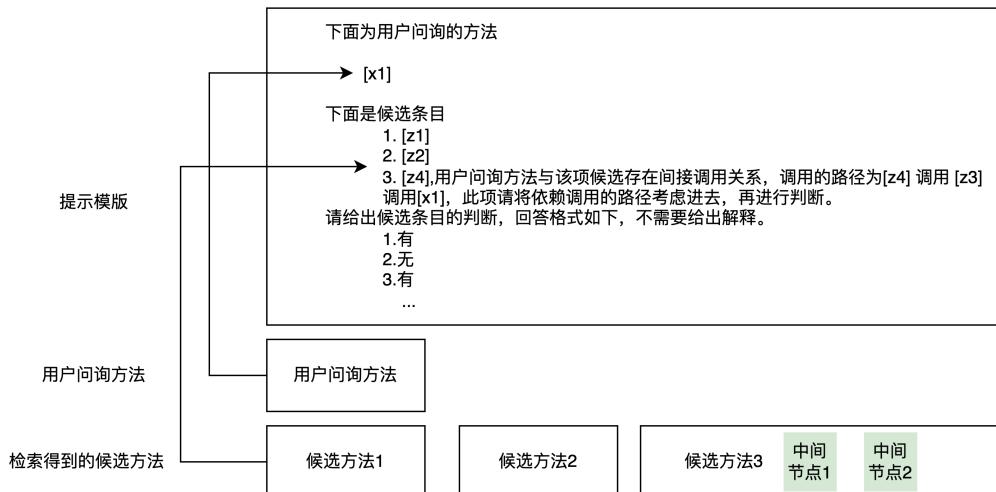


图 3-4 动态构建生成模块所使用的提示

将大语言模型的回复进行解析即可得到与查询方法有变更影响分析的方法。对于有 N 个方法的项目，在实际使用时对于每个查询，需要进行 N 次相似度计算，调用 1 次大模型进行推理，如果上下文长度过长可以对候选方法分片，多次调用大模型，相比于基于代码预训练模型的方法每次需要 N 次的模型前向计算，不仅节省了大量计算时间，还提升了系统的查全率和查准率，在实际应用场景中是更好的选择。

3.7 实验结果与分析

3.7.1 实验数据

为了便于与前文方法进行性能对比，本章的测试数据与第二章相同，具体

信息见表2-4所示。

对于嵌入模型所需的三元组训练数据，本章中数据清洗的方法新增了对依赖路径信息的补充，数据集的收集过程如3.4节中所述。在数据清洗过程中，所使用的模型为 Doubao（API 模型名：Doubao-lite-32k-240428）。最终得到的训练数据的统计信息如表3-1所示，共计包含 4289 对数据。这些数据将用于嵌入模型的训练。

表 3-1 嵌入模型训练数据统计信息

项目名称	三元组数量
TheAlgorithms	314
antiword-0.37	257
jemalloc-5.3.0	1023
libbpf-1.1	835
librdkafka-2.1.0	1524
FFmpegKit-5.1.0	336
总计	4289

3.7.2 评价指标

评价指标与第2.7.2节一致，采用 Precision（精确率）、Recall（召回率）和 F-measure 来全面评估模型在检测变更影响关系中的表现。这些指标可以有效衡量模型在实际应用中的能力，

3.7.3 实验设置

1. 检索模块嵌入模型选择

MTEB Leaderboard^① 是一个实时更新的开源嵌入模型排行榜，专门用于评估不同模型在各种任务中的嵌入能力。为了实现对原始知识库中方法代码的有效编码，本文从 MTEB Leaderboard 中选取适合本任务的嵌入模型。选择标准主要包括以下几点：(1) 模型在开源嵌入能力评价基准上表现优异，这能够体现其在各类通用文本上的嵌入能力。(2) 模型支持的上下文长度大于 4k，考虑到代码任务通常涉及较长的上下文，较短的上下文长度可能无法有效捕捉到代码间的复杂关系。(3) 模型的参数规模控制在 500M 以下，综合考虑计算资源和数据规模，超过 500M 参数规模的模型在使用上的代价较高，难以满足实际应用需求。基于这些选择标准，本文最终选定了如表3-2所示的 5 个模型。这些模型在性能、上下文支持以及资源消耗方面都达到了较好的平衡，适合本文中对

^① <https://huggingface.co/spaces/mteb/leaderboard>

代码片段进行嵌入编码的需求。

表 3-2 嵌入模型信息

嵌入模型	参数数量	上下文长度
gte-large-en-v1.5	434M	8192
jina-embeddings-v3	572M	8192
stella_en_400M_v5	435M	8192
KaLM-embedding-multilingual-mini-instruct-v1.5	494M	131072
nomic-embed-text-v1.5	137M	8192

2. 嵌入模型训练设置

本文在嵌入模型的训练过程中，采用了对比损失函数 InfoNCE Loss 作为目标函数，以确保模型能够有效学习到不同代码片段之间的相似度关系。训练优化器选择了 Adam 优化器，其中超参数 β_1 设置为 0.9, β_2 设置为 0.95。为提高训练效果，学习率采用余弦衰减策略，使得训练过程中学习率逐渐减小，有助于模型在接近收敛时获得更精细的更新。

Batch size 固定为 64，以保证每次梯度更新的稳定性。对于较大规模的模型，考虑到内存限制，采用了梯度累计（Gradient Accumulation）技术，通过多次反向传播累积梯度，从而保证了相同的批次大小。学习率初始值设置为 1e-4。

所有实验均在 PyTorch 框架和 Transformer 库上实现，确保了高效的模型训练和易于扩展的实现平台。训练过程在单个 NVIDIA Tesla V100 32G GPU 上进行。

3. 生成模型选择

本章实验中生成模块使用的大模型包含两类，分别是闭源商业模型和开源模型。其中闭源模型包括：GPT-4o-mini (API model name: gpt-4o-mini-2024-07-18), Doubao (API model name: Doubao-lite-32k-240428), Kimi (API model name: moonshot-v1-32k)。开源模型选择：Qwen2.5-Coder-3B-Instruct , Llama-3.2-3B-Instruct

3.7.4 实验结果与对比分析

RQ1: 经过训练后的检索模块性能如何，与 0 样本嵌入模型相比是否更专注于变更影响关系检测？能否有效检索到与查询方法存在变更影响关系的方法？

RQ2: 本章提出的基于依赖链与检索增强生成的变更影响关系分析方法性能如何，相比于其他方法，其在精确率，召回率和 F-measure 上表现如何？

RQ3: 本章提出的基于依赖链与检索增强生成的变更影响关系分析方法是否能有效解决间接依赖信息缺失的问题？

RQ4: 本章提出的基于依赖链与检索增强生成的变更影响关系分析方法的跨项目迁移表现如何?

1. 针对于 RQ1 的实验

本实验通过计算 $topk$ 中 k 值与召回率的关系，验证检索模块嵌入模型的性能。实验结果如表3-5所示。其中横坐标表示 $topk$ 值，即检索模块在知识库中检索到的相似度排序的前 k 个，纵坐标表示召回率。虚线为未经训练的嵌入模型，实线表示经过训练的嵌入模型。

图 3-5 检索模块的召回率与 Topk 关系

经过分析可以发现，总的来讲，除模型 KaLM-embedding-v1.5 之外，经过训练的嵌入模型在本文应用场景的性能普遍优于未经训练的模型，其中模型 KaLM-embedding-v1.5 较为特殊，在 0 样本的情况下其本身性能差于其他模型，因此训练后虽有提升，但还是难以超越其他模型未经训练的版本。

对于未经训练的嵌入模型来讲，模型 stella_en_400M_v5 表现最好，但在 k 最大为 30 的设置下也只能达到 50% 左右的召回率，这说明对于查询方法来讲，有近一半的与其有变更影响关系的方法未被检索出来，性能一般。而经过训练的嵌入模型普遍优于未经训练的模型，其中 stella_en_400M_v5 模型的综合表现最佳。在 k 为 15 到 30 的区间，召回率均高于其他模型，尤其在 k 为最高 30 时，能达到近 90% 的召回率。因此可以得出结论，训练后的检索模块性能较强，与未经训练的模型相比，能更专注于检测变更影响关系的任务。

对于 $topk$ 的 k 值，可以发现 k 值越大召回率越高，这是由于在召回率较小的时候，检索到的候选方法较少，甚至真实正例数即能超过该 k 值。而 $topk$ 值越大，则检索到的包含真实正例的概率也越大，因此召回率越高。

考虑到训练后嵌入模型的性能和 $topk$ 的 k 值大小对推理速度和召回率产生影响，综合考虑选择模型 stella_en_400M_v5 作为嵌入模型， $topk = 20$ 作为后续实验的设定。

2. 针对于 RQ2 的实验

实验结果如表3-3所示。总的来讲，基于大语言模型的方法的检测性能均优于基于代码预训练模型的方法，即使是最小的开源模型 Qwen2.5-Coder-3B-Instruct，Llama-3.2-3B-Instruct，其在 F-measure、召回率和精确率的表现也可以超过经过上一章节训练的 CodeBERTa 模型。

为了更好地评估大语言模型在变更影响关系检测任务中的重要性，本实验进一步与仅使用检索模块的方案进行了对比分析。从实验结果可以看出，单独使

表 3-3 不同 LLM 的实验表现

方法	F-measure	recall	precision
CodeBERTa	59.5	55.3	64.4
Just Retrieval	32.6	78.1	20.6
RAG-GPT4o	75.8	74.6	77.1
RAG-Doubao	73.4	73.8	73.1
RAG-Kimi	70.2	71.2	69.2
RAG-Qwen2.5	65.4	63.5	67.4
RAG-Llama-3.2	64.9	62.3	67.8

用检索模块的方案在召回率上表现最佳，达到了引入生成模型后性能的上限。这一现象的原因在于，大语言模型只在检索到的候选结果基础上进行进一步判断，因此，它的作用是减少而非增加变更影响关系的检测结果。由于设定了 $topk = 20$ ，检索过程中不可避免地会出现误报，这导致了该方法的准确率相对较低。

然而，在加入生成模块之后，召回率略有下降，但精确度却提升了 47.2% 至 55.5%，这充分证明了大语言模型在代码语义理解基础上判断变更影响关系的强大能力。通过结合检索模块和生成模块，RAG 方法能够在提高查全率的同时，显著提升查准率，确保检测过程中的全面性和准确性。

此外，选择不同的大语言模型作为生成模块，其对检测任务的表现也存在一定差异。总体而言，闭源商业模型的性能优于开源模型。具体来说，以 GPT4o 为生成模块的 RAG 方法展现出了最佳的综合性能，其在召回率和准确率上均表现优异。而 Doubao 和 Kimi 模型的表现稍逊一筹，尽管在召回率上与 GPT4o 接近，但精确率存在明显差距，可能是由于这些模型对代码语义的理解能力相对较弱所致。而开源模型之间的表现相对平衡，尽管它们的整体效果略低于闭源商业模型，这可能与模型规模和架构设计存在差异有关，但相比于基于代码预训练模型的方法，仍然表现出了显著的优势。

综上所述，生成模块在变更影响关系检测中的作用不可忽视，它在变更影响分析任务中的体现出了强大的代码语义理解能力，能够有效提升检测的精确率。尽管闭源商业模型的表现更为优越，但开源模型依然提供了一个具有竞争力的选择，在实际应用中具备较好的实用价值。

为了进一步分析 RAG 方法与其他方法在不同类型的变更影响关系上的检测效果，本实验对比了不同方法在依赖型和逻辑型的表现，得到的结果如表3-4所示。对比可以发现相比于其他方法来讲，基于依赖链与检索增强生成的方法在两个类型的变更影响关系检测上，均拥有更好的性能。这表明，基于依赖链与检索增强生成的方法在处理依赖型和逻辑型变更影响关系时，具有更强的

表 3-4 变更影响实验结果

方法	Dependence-Based			Logic-Based			Overall
	F-measure	recall	precision	F-measure	recall	precision	
依赖闭包	44.8	100	28.9	-	-	-	36.8
克隆检测	-	-	-	31.8	19.1	95.7	3.8
共现关系挖掘	56.0	44.6	75.4	57.0	47.3	71.8	52.8
CodeBERTa	61.9	56.6	68.3	71.7	73.9	69.7	59.5
Just Retrieval	34.5	82.3	21.8	36.6	88.5	23.1	32.6
RAG	79.8	78.4	81.2	86.9	85.9	87.9	75.8

检测能力。

依赖型关系和逻辑型关系有着本质的差异，分别侧重于不同方面的代码间联系。而 RAG 方法在这两类关系的检测上均展现了优越的性能。依赖型影响关系的判定依赖于方法间的直接或间接调用。前文方法在处理间接依赖时存在信息缺失，而 RAG 方法通过结合代码依赖图和检索模块，能够定位到相关方法并补充缺失的调用路径，显著提高了依赖型关系的召回率和准确率。在这一任务中，F-measure 的提升达到 16.9%，有效减少了漏报的情况。而逻辑型影响关系主要关注代码的逻辑层面，涉及程序控制结构和功能实现之间的复杂交互。在这类关系中，对代码的语义理解尤为重要，RAG 方法通过引入大语言模型的语义理解能力，能够深入分析代码的结构和功能逻辑，精准地判断代码变更对系统的影响。逻辑型关系的 F-measure 提升达到 15.2%，表明 RAG 方法在这方面的优势同样显著。

总的来说，RAG 方法通过结合检索模块和大语言模型的优势，在依赖型和逻辑型变更关系的检测中均取得了显著的性能提升。依赖型关系的检测通过更精准的依赖路径识别提高了查全率，而逻辑型关系的检测则通过对代码语义的深度理解提升了精确度。两者结合，使得 RAG 方法在变更影响分析任务中展现了更强的适应性和准确率。

3. 针对于 RQ3 的实验

为了验证依赖路径在变更影响关系检测中的重要作用，并探讨将调用路径中的中间方法纳入提示对间接依赖检测效果的提升，本实验选择了测试集中的间接依赖型变更影响分析子集进行验证。为了深入分析其贡献，设计了依赖路径消融实验，结果如表3-5所示。

分别与第二章方法和未加入依赖路径的 RAG 方法进行对比，根据结果可以看出在提示中加入路径的中间方法对于检测效果的提升非常显著。与第二章方法相比，F-measure、召回率和精确率分别增加了 61.9%、58.4% 和 63.4%，与

表 3-5 加入调用路径的中间方法对间接依赖的检测效果的影响

方法	Indirect-Dependence-Based		
	F-measure	recall	precision
CodeBERTa	10.7	15.1	8.3
RAG without 依赖路径	14.2	12.1	17.1
RAG	72.6	73.5	71.7

未加入依赖路径的方法相比，分别增加了 58.4%、61.4% 和 54.6%，说明方法中间的调用信息对变更影响关系的判断非常重要，如果丢失，则模型无法捕捉方法之间的联系，增加调用链能够补全这部分信息。

与第二章中提出的方法和未加入依赖路径的 RAG 方法分别进行对比，实验结果表明，在提示中加入依赖路径中的中间方法对变更影响关系的检测效果有显著提升。具体而言，与第二章方法相比，F-measure、召回率和精确率分别提高了 61.9%、58.4% 和 63.4%；与未加入依赖路径的 RAG 方法相比，F-measure、召回率和精确率分别提高了 58.4%、61.4% 和 54.6%。这些结果表明，方法调用链中间信息的加入对于变更影响关系的判断至关重要。缺少这些中间调用信息时，模型往往无法正确捕捉方法之间的内在联系，从而影响检测精度。通过引入调用链中的中间方法，模型能够有效弥补这一缺失，补全变更影响分析中的关键依赖信息，显著提高检测效果。

这一实验结果进一步证明了在提示信息中加入依赖路径不仅增强了模型对间接依赖的感知能力，也有助于提升整体变更影响关系分析的准确性和实用性。

4. 针对于 RQ4 的实验

为了验证基于 RAG 的方法在跨项目场景下的泛化能力和知识迁移能力，本章设计了类似第二章的跨项目实验。数据集划分同上一章节一样，将项目分为四个分布内项目和两个分布外项目。在跨项目的实验设置下，模型仅在分布内项目上进行训练，而在测试时则只使用分布外项目进行验证。

实验结果如表3-6所示。通过对分析，上一章节基于代码预训练模型的方法在跨项目测试时，性能下降幅度在 28.9% 到 31.7% 之间，显示出较为明显的性能衰减。而本章所提出的基于依赖路径和 RAG 的变更影响分析方法，在跨项目测试时性能下降幅度仅为 9.2% 到 11.4%，显著低于基于代码预训练模型的方法。

这一结果表明，RAG 方法在跨项目迁移时的性能下降较小，体现出更强的泛化能力，能够有效地将从一个项目中学到的变更影响模式迁移到新项目中，

表 3-6 跨项目迁移能力分析 (F-measure)

方法	TheAlgorithms	antiword	librdkafka	FFmpegKit	jemalloc	libbpf
train on 6 datasets	train	train	train	train	train	train
CodeBERTa	60.23	59.64	60.18	57.87	60.99	58.22
Just Retrieval	32.4	32.91	31.81	36.57	33.78	35.07
RAG	76.0	78.45	76.46	77.66	79.04	76.29
train on 4 datasets	train	train	train	train	test	test
CodeBERTa	62.28	65.24	63.45	62.19	41.92*	40.8*
Just Retrieval	33.33	34.2	34.77	36.89	29.74*	31.09*
RAG	78.24	79.21	78.59	77.98	71.08*	69.66*

即使新项目的特征与源项目存在差异，系统依然能够维持较高的检测精度。这一优势不仅使得本章方法更贴近实际应用中的跨项目场景，也为变更影响分析提供了更广泛的适用性。相比之下，基于代码预训练模型的方法在面对不同项目时的表现较为不稳定，难以有效迁移学习到的模式，导致跨项目应用时性能明显下降。

因此，本章提出的基于依赖路径和 RAG 的变更影响分析方法不仅表现出较高的准确性，在跨项目中的泛化能力同样优秀，充分证明了该方法在多样化应用场景下的可行性和实用性。

3.8 本章小结

本章提出了基于依赖链与检索增强生成的代码变更影响分析方法，通过结合依赖路径提升了方法在间接依赖影响上的检测性能，并通过训练检索模块，大幅提升了测试集上的召回率，最后整个系统的综合表现得到了大幅提升。并且通过实验证明了依赖路径加入的有效性。最后在跨项目迁移能力的实验中，基于依赖链与检索增强生成的代码变更影响分析方法的迁移能力表现优秀，进一步证明了本章所提出的方法的有效性。

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^[36]。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，本文提出了代码审查图的概念及其构建方法，同时提出基于代码审查图的代码结构可视化和质量分析方法。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^[37]，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。这里一共建以下四个指标，

(1) LCOM1，含义是不引用相同字段的方法对数目^[38]。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 $C_6^2 - 5 = 10$ 。

(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^[39]。其计算公式如式 (2-2)。

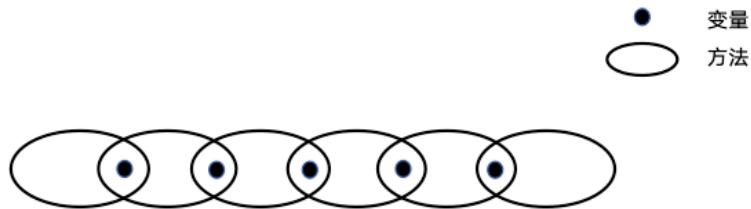


图 4-2 示例模块

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

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。

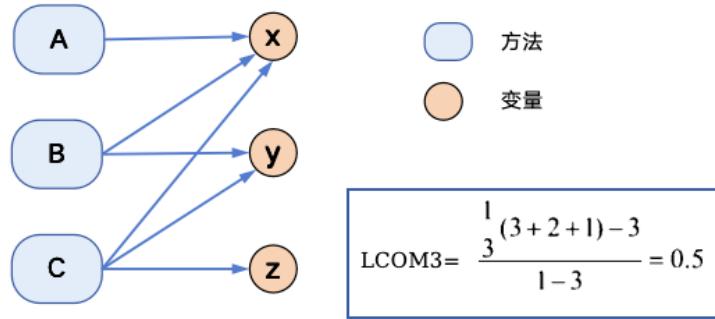


图 4-3 LCOM3 计算示例

(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^[40]。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。

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

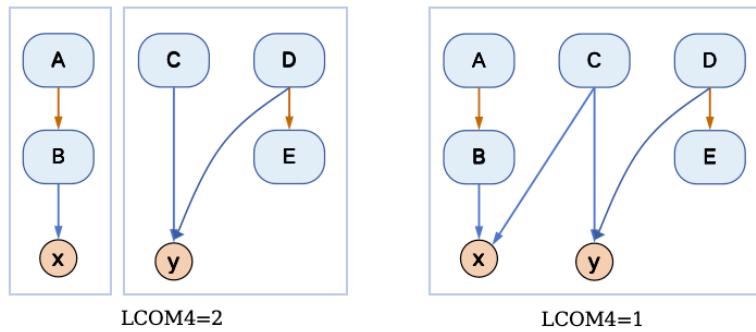


图 4-4 LCOM4 计算示例

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^[41]。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^[41]。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。

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

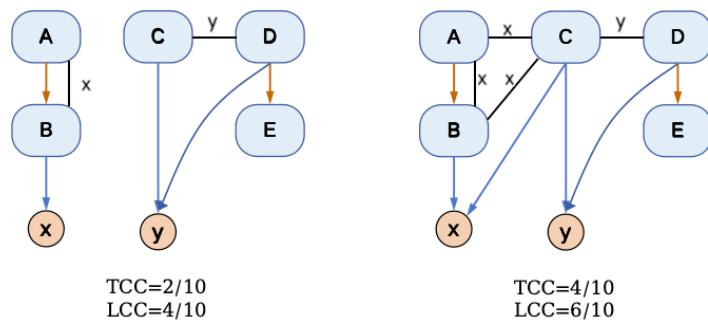


图 4-5 TCC 和 LCC 计算示例

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，可能需要进行分解和增加中间层方法的方式进行重构，以减少对外部方法的依赖，提升其可复用性和灵活性。

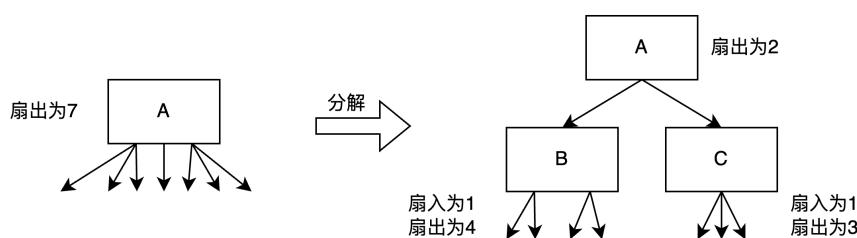


图 4-6 高扇出及其分解示例

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代码安全性和规范性度量

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。

1. 节点设计 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。

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

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

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

2. 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。

1. 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。

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。

2. 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。本节通过分析代码审查图展示的软件代码结构，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。

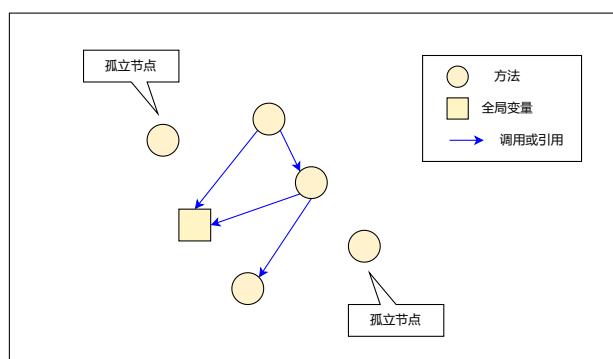


图 4-8 冗余代码图模式

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。

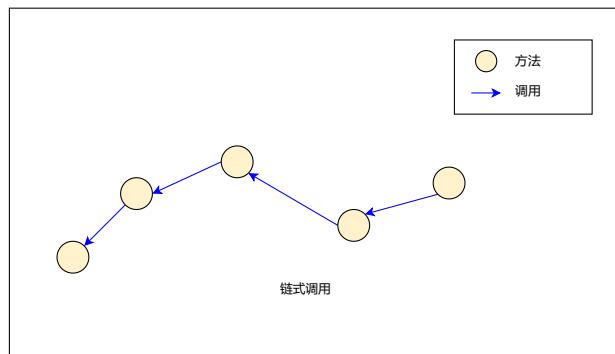


图 4-9 长嵌套调用图模式

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、依赖过多的外部变量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。

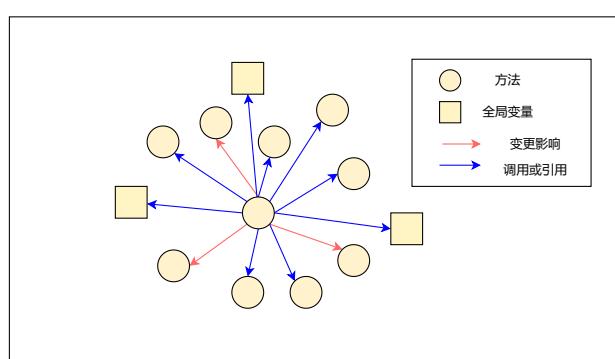


图 4-10 复杂方法图模式

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，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。

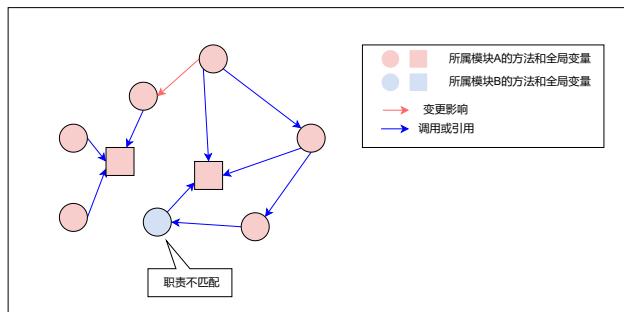


图 4-11 职责不匹配图模式

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. 隐式逻辑型变更模式 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。

这种隐式逻辑型变更模式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，蓝色边表示调用关系。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

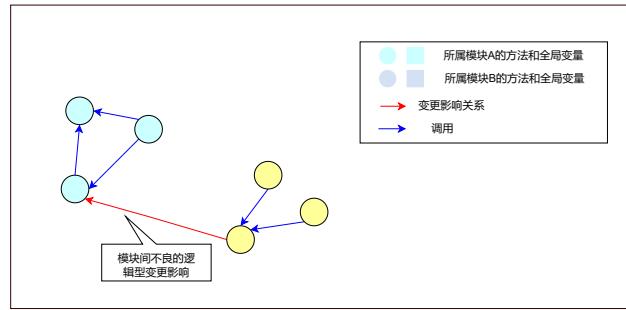


图 4-12 隐式逻辑型变更模式

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。

6. 模块间紧耦合 方法间的耦合关系共分为 4 种，其中公共耦合和外部耦合为耦合性最强的两种耦合，均存在显著的缺点。公共耦合指多个方法共享同一全局数据结构，这种耦合方式会导致一个方法的修改可能对其他方法产生不经意的影响，降低了代码的可维护性和可理解性。外部耦合是指一组方法通过访问同一全局简单变量进行交互，这种耦合方式使得方法之间的依赖关系变得隐晦且难以追踪，变量的改变可能引发多个方法的异常行为，且问题的根源往往难以定位。而不同模块间的这两种耦合则更为严重，由于扩散效应，可能导致两个模块间功能逻辑被影响。

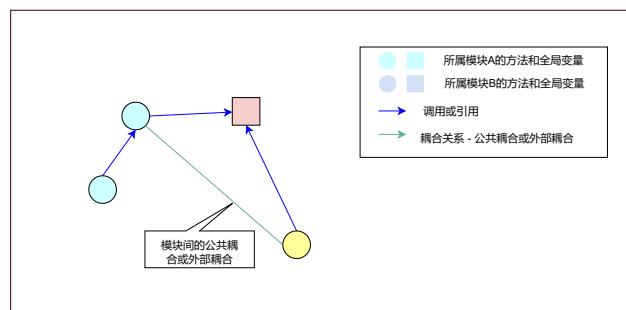


图 4-13 模块间的公共耦合或外部耦合

这种模块间的公共耦合和外部耦合表现为图4-13所示的模式，两个所属模块不同的方法被公共耦合或外部耦合类型的边连接起来。在优化时，建议通过参数传递数据，将公共耦合或外部耦合降级为标记耦合或数据耦合，避免直接访问同一全局变量。

4.5 代码质量分析报告生成

在软件开发和代码审查的过程中，开发者通常可以借助代码审查图聚焦于代码的上下文，帮助发现局部代码的问题。然而，当软件开发完成，开发者希望从全局角度对软件项目的整体质量进行衡量时，仅依靠代码审查图可能会存在质量信息过于分散、不易聚焦的问题。因此，本节进一步通过生成文档化的代码审查报告，为开发者提供统一的代码质量概览，帮助其全面掌握项目的质量状况。

代码审查报告的核心目标是揭示软件项目中存在的关键质量问题，并以本文提取的代码质量属性与子属性为主线，系统性地向用户报告代码中的相对质量子属性为差的模块或方法，并给出相应的建议。为了帮助用户在代码开发过程中防止变更不完全，报告还将列出项目中所有的代码变更影响关系。用户可以参考这些信息，在变更时全面分析影响范围，降低遗漏风险。

通过对这些信息的整合与分析，生成的报告文档为用户提供了一份全面的代码质量概览，既可以帮助用户识别代码中的潜在问题，又能为系统的后续优化与维护提供指导性建议。

4.6 实验结果与分析

4.6.1 实验数据和实验设计

本章实验从软件的历史版本变更角度出发，结合代码度量和不良图模式，探讨在代码历史中修复和重构概率较大的代码与其质量或代码审查图模式之间的关系，从而验证其有效性。

1. 实验数据 本章选取第二章的部分项目进行研究，选择 jemalloc 为示例项目进行代码质量度量模型的分析实验，选取 jemalloc 和 TheAlgorithms 项目进行代码审查图的不良图模式的分析实验。

将项目的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, improve, Improve }。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，能在一定程度上反映用户提高代码质量的操作。
- 进一步按代码变更类型筛选：分析代码变更，判断是否真正包含修复或重构的行为。

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

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：

- (1) 对于实验数据中收集到的提交，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。
- (2) 将该提交中变更的方法与不同分档进行求交集，得到每档中方法变更的数量。
- (3) 观察代码质量度量和变更概率的数量关系，验证其相关性。

3. 验证代码审查图中不良图模式的实验设计 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。

4.6.2 实验环境

实验环境如表 4-6 所示。

表 4-9 实验环境

环境	版本信息
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

4.6.3 实验结果分析

RQ1: 代码质量度量与版本历史中修复和重构相关的代码变更是否存在一定的相关性，在修复或重构过程中，代码质量如何影响开发人员的变更行为？

RQ2: 代码审查图中不良图模式与版本历史中修复和重构相关的代码变更

是否存在一定的相关性? 基于代码审查图的分析能否为代码优化重构提供指导?

1. 针对于 RQ1 的实验

以4.2.1节中质量度量模型的质量属性为分析对象, 得到的实验结果如表所示, 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。

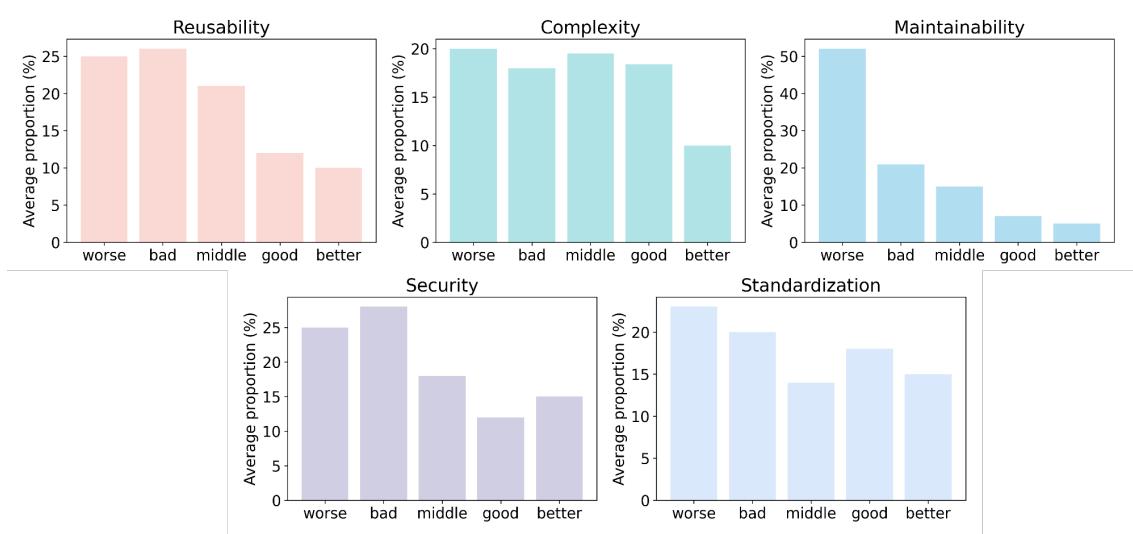


图 4-14 代码质量度量 - 方法变更比例

通过表4-14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。

进一步分析可以发现, 这种现象在可维护性上表现最为明显, 其次是可复用性和安全性, 而在复杂性和规范性上则没有明显的趋势。对于复杂性而言, 原因可能是不同的开发人员对于代码复杂性的感知和接受程度可能有所不同, 并且在某些情况下, 复杂性高的代码并不一定意味着它就需要被优先重构或修复, 因为其可能是为了实现特定的性能要求或者业务逻辑而故意为之。对于规范性而言, 它更多地侧重于代码是否遵循了特定的编码标准和风格指南, 虽然不规范的代码可能会给团队协作和后续的维护带来一定的麻烦, 但在某些情况下, 不影响代码功能和性能的轻微不规范问题可能会被暂时搁置, 开发人员更倾向于优先处理那些会直接影响系统可维护性、可复用性和安全性的代码部分。这种现象也从侧面反映出开发人员在进行代码重构和修复工作时, 往往会优先考虑那些对系统的长期稳定性和扩展性有重要影响的代码属性, 可维护性、可复用性和安全性方面的问题一旦出现, 会对系统的可靠性和后续开发造成更大的潜在风险, 因此更容易受到关注和处理; 而对于复杂性和规范性, 由于其在

某些情况下的模糊性和非紧急性，在代码的重构和修复过程中没有像其他属性那样呈现出明显的变更趋势。

因此，开发者在软件开发或维护过程中，应当更加关注代码的可维护性、安全性以及可复用性。在此基础上，还需留意模块的内聚性、方法合理的扇入扇出情况，避免产生过紧的耦合，并尽量减少逻辑型的变更影响关系。同时，也要确保代码的安全性和规范性。尤其需要注意的是，要避免出现忽视代码在可维护性、安全性和可复用性方面潜在隐患的行为，在处理质量分析报告时，应将涉及这三类的问题赋予较高的优先级。

上述实验结果进一步验证了代码质量度量模型在实际应用时的指导意义，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 的实证分析

本实验验证的基本思想是通过分析项目代码某一版本的代码审查图，发现其不良图模式，观察后续版本中对该不良图模式的变更，尤其是重构或修复的行为，从而验证其不良性，进而验证代码审查图对代码质量分析和优化重构的指导意义。针对 4.4 节总结的不良图模式，在示例项目（TheAlgorithms 项目和 jemalloc 项目）中均出现过对应子图，并且在后续版本的优化中，存在不同程度的重构、变更或优化。

图 (a) 是 jemalloc 项目 5.3.0 版本中的实际存在的子图，该图为 src/pa.c 文件的代码审查图子图。该文件定义了 20 个方法，从图中可以清晰地看出该文件中方法 pa_shard_retain_grow_limit_get_set 并未与任何方法产生联系，因此属于冗余代码。在后续的优化中（commit da66aa3），该方法被删除。同时可以观察到该文件的结构较为松散，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 numerical_methods/ode_midpoint_euler.c 文件和 numerical_methods/ode_forward_euler.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。在提交 8513174 中对其进行了结构上的优化，使其具有更优的模块化结构。

图 (c) 是 jemalloc 项目中 5.1.0 版本中 unit/emitter.c 文件存在的复杂方法

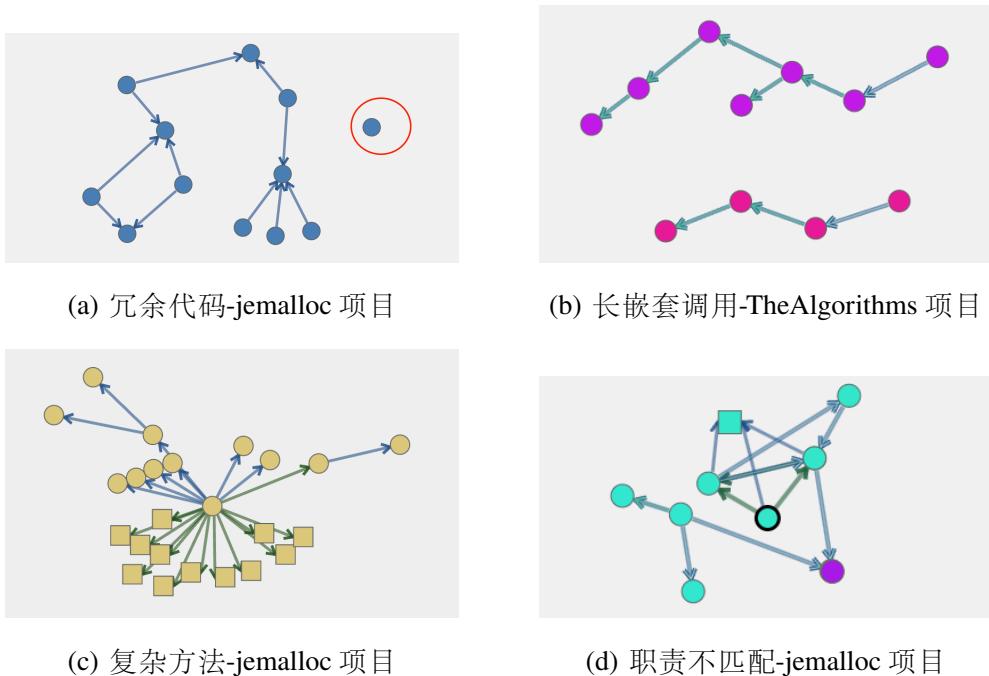


图 4-15 示例项目中实际的

`emit_modal`, 该方法依赖了调用了 8 个其他方法和 12 个全局变量, 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。在后续的优化中, 该方法调用的其他方法进行了组合和重构, 该提交 `eb261e5` 被收录在 v5.2.0 中。

图 (d) 是 `jemalloc` 项目中 `src/hpa.c` 文件声明的方法 `hpa_supported`, 在 5.3.0 版本前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` 等文件的方法, 因此出现了职责不匹配的现象。该现象在 v5.3.0 版本有所改善, 部分文件外方法与该方法进行了解耦, 在如 `d93eeff2` 等提交中对其增加了文件内方法的调用, 使其更专注于本文件内部逻辑。

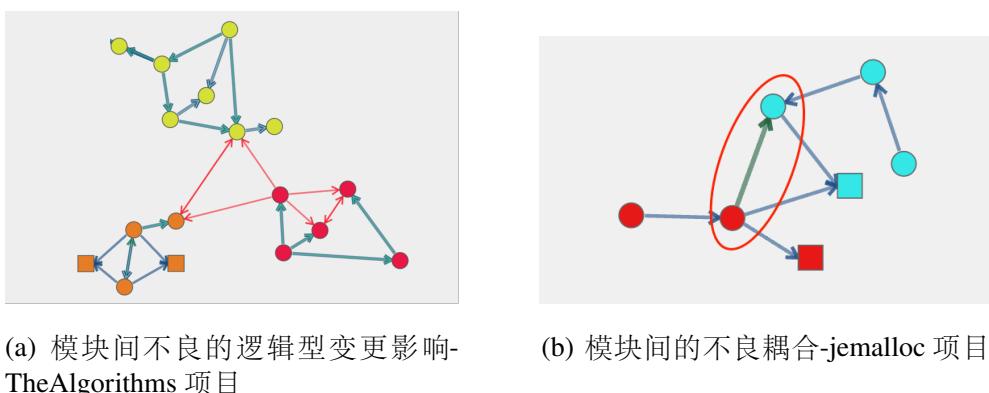


图 4-16 代码审查图

图4-16 (a) 展示了 TheAlgorithms 项目中的一个实际子图，涉及三个模块，分别是 modified_Binary_Search、radix_sort_2 和 lexicographic_Permutations.c。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。在后续的 4b07f0f 等提交中均观察到其同时变更的维护工作。

图4-16 (b) 中展示了 jemalloc 项目中的 src/decay.c 文件与 hpa.c 文件的部分代码审查图，两个模块由于全局变量 h_steps 而产生了外部耦合，这是一种不良的模块间的耦合关系，一个模块内任何改变该变量的操作都可能沿着耦合影响到另一个模块。在 v5.2.0 中将此种耦合进行了优化，在 8229cc7 提交中缩小了该变量的作用域，从全局可使用变为了仅在文件内可使用。

因此，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分析的直观性和说服力。

4.7 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，分别是作为一种静态分析工具对软件代码进行结构可视化和质量分析、作为一种开发工具辅助开发者的代码维护以及作为一种审查工具帮助审查者进行代码审查。

1. 作为一种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，观察代码结构。

图4-19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，体现了代码结构上的相互关联关系。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。

(1) 分析结构特征信息

从图中可以明显看出两个项目在代码结构特征上的显著差异。

- Antiword 项目整体模块之间高度协作，共同实现一个完整的功能，因此

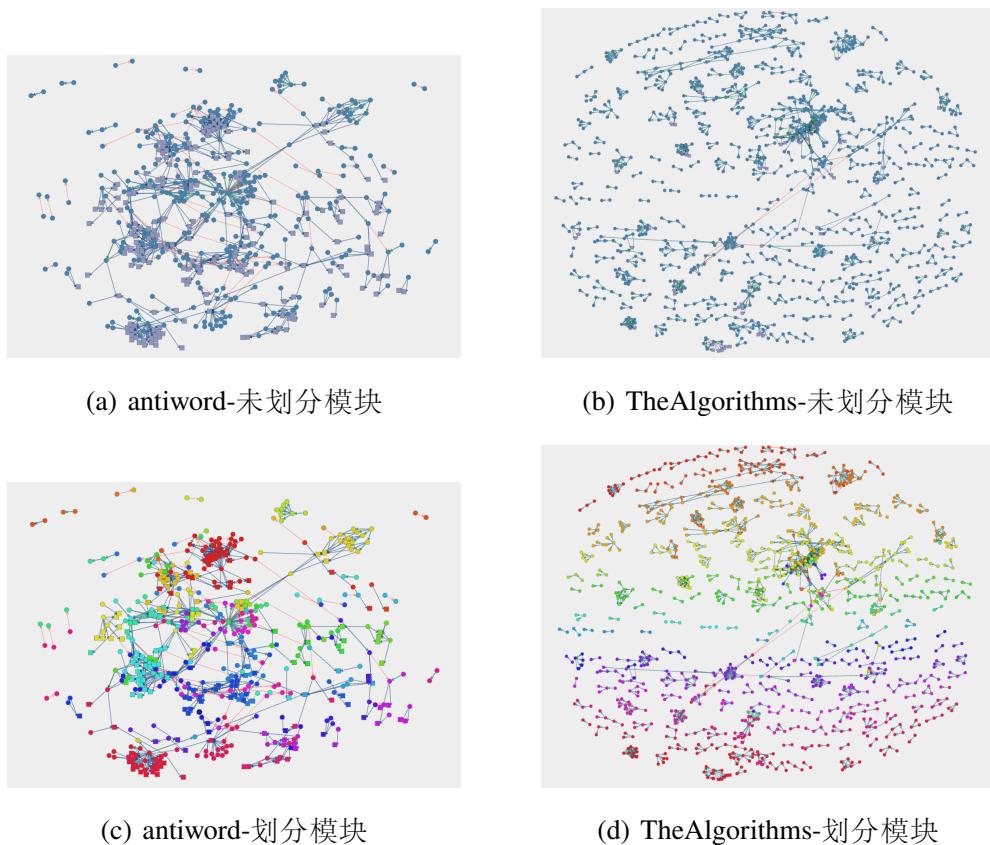


图 4-17 代码审查图

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。

(3) 分析不良的图模式

根据4.4中所述的不良的图模式，分析 **Antiword** 项目的代码审查图，可以发现一些对应的子图，反映了代码中存在的一些质量或结构问题。

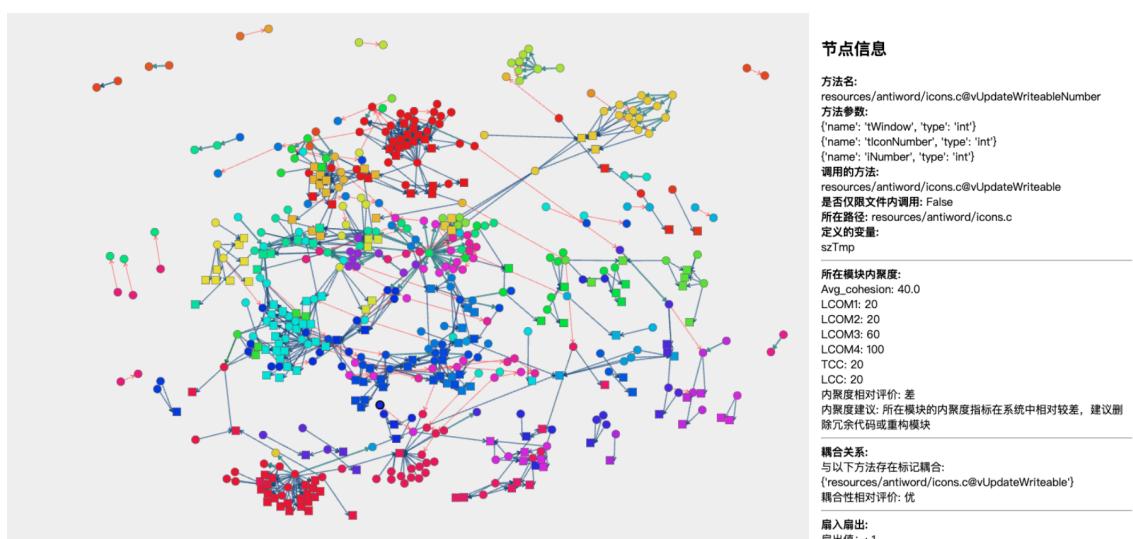


图 4-18 点击节点展开节点信息

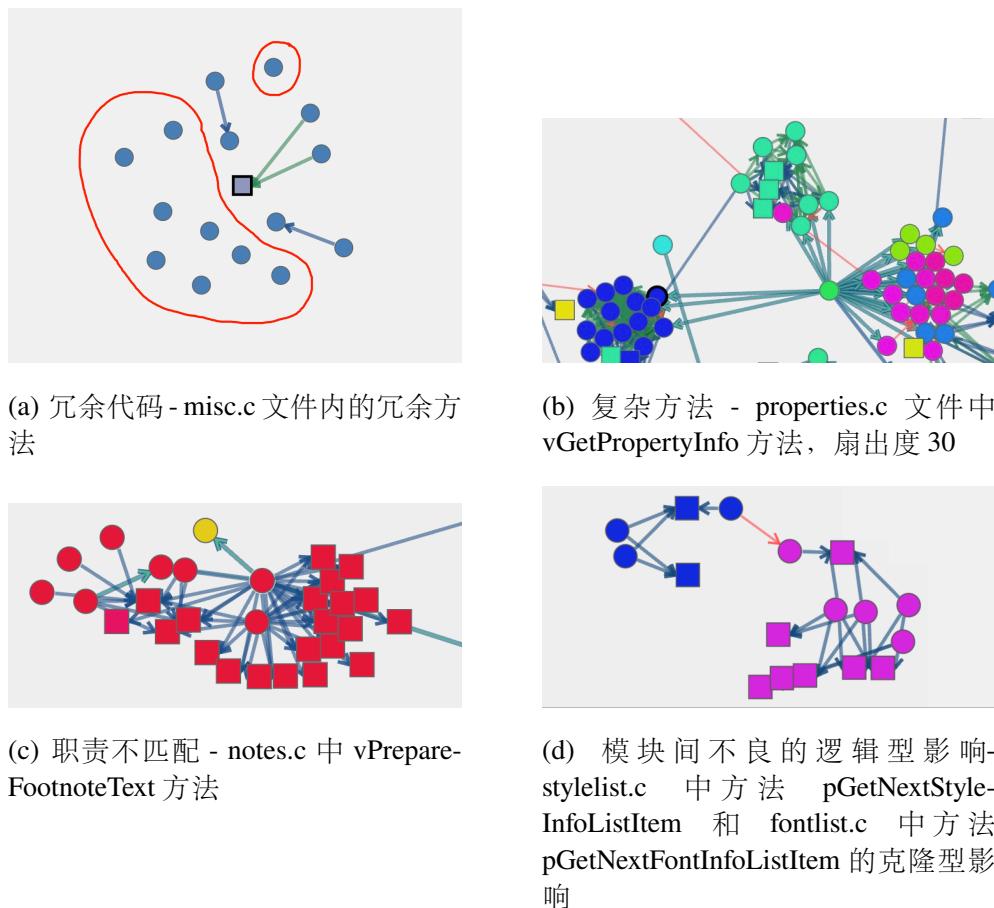


图 4-19 antiword 代码审查图中的部分不良子图

(4) 导出质量分析报告

下图展示了一个实际的代码分析报告示例。由于报告内容较长，这里仅展示了其中的一部分信息。通过代码分析报告，用户能够以结构化的清单形式全

面了解软件项目中存在的各类质量问题及其具体位置。这些报告不仅清晰地列出了每个问题的详细描述，还提供了针对性优化或重构的建议。通过这种方式，开发团队可以快速识别代码中的潜在缺陷或性能瓶颈，并根据报告中提供的指导意见，采取有效的措施进行改进。报告的可视化和条目化呈现，帮助用户直观地理解各项问题的优先级和重要性，从而优化项目的维护流程，并提高软件系统的整体质量。

antiword 项目质量分析报告	
1. 可复用性:	3. 可维护性:
1.1 内聚度质量属性:	3.1 耦合性质量属性:
<ul style="list-style-type: none"> antiword/png2sprt.c 模块相对内聚度评分为 33.33333333333336 相对评价为 极差，建议删除冗余代码或重构模块 	<ul style="list-style-type: none"> 见 1.2 节
1.2 耦合性质量属性:	3.2 变更影响属性:
<ul style="list-style-type: none"> antiword/properties.c@vGetPropertyInfo 方法耦合性相对评价为 极差，与其他方法耦合性较高，建议解耦合或进行合并，与其他方法耦合性较高，建议解耦合或进行合并 	<ul style="list-style-type: none"> antiword/xml.c@vAddEndTagsUntil2 方法变更影响相对评价 极差，与其他方法变更影响关系过多，建议解耦合，在变更时安全变更，与其他方法变更影响关系过多，建议解耦合，在变更时安全变更
1.3 扇入属性:	4. 安全性:
<ul style="list-style-type: none"> antiword/xml.c@vPushStack 方法扇入相对评价为 极差，扇入度较小，建议检查，可复用性不高，建议进一步拆分或删除 antiword/xml.c@ucPopStack 方法扇入相对评价为 极差，扇入度较小，建议检查，可复用性不高，建议进一步拆分或删除 	<ul style="list-style-type: none"> antiword/fonts_u.c@pOpenFontTableFile 方法存在严重缺陷，具体信息为 error (unknownMacro) There is an unknown macro here somewhere. Configuration is required. If ANTIWORD_DIR is a macro then please configure it. [CWE-0]，建议修复
1.4 扇出属性:	5. 规范性:
<ul style="list-style-type: none"> antiword/properties.c@vGetPropertyInfo 方法扇出相对评价为 极差，扇出度较大，建议分解并增加中间层，扇出度较大，建议分解并增加中间层 	<ul style="list-style-type: none"> antiword/xml.c@vPrintSpecialChar 方法存在规范性问题，具体信息为 style (variableScope) The scope of the variable 'tIndex' can be reduced. [CWE-398] 建议更正 antiword/xml.c@vPrintSpecialString 方法存在规范性问题，具体信息为 style (variableScope) The scope of the variable 'usChar' can be reduced. [CWE-398] 建议更正
2. 复杂性:	附变更影响关系列表:
2.1 圈复杂度属性:	<ul style="list-style-type: none"> antiword/xml.c@vAddStartTag 与方法 antiword/xml.c@vAddCombinedTag 存在变更影响关系 antiword/xml.c@vAddEndTag 与方法 antiword/xml.c@vAddCombinedTag 存在变更影响关系 ...
2.2 扇出属性:	
<ul style="list-style-type: none"> 见 1.4 节 	

图 4-20 antiword 项目代码质量分析报告节选

2. 作为一种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4-21所示。

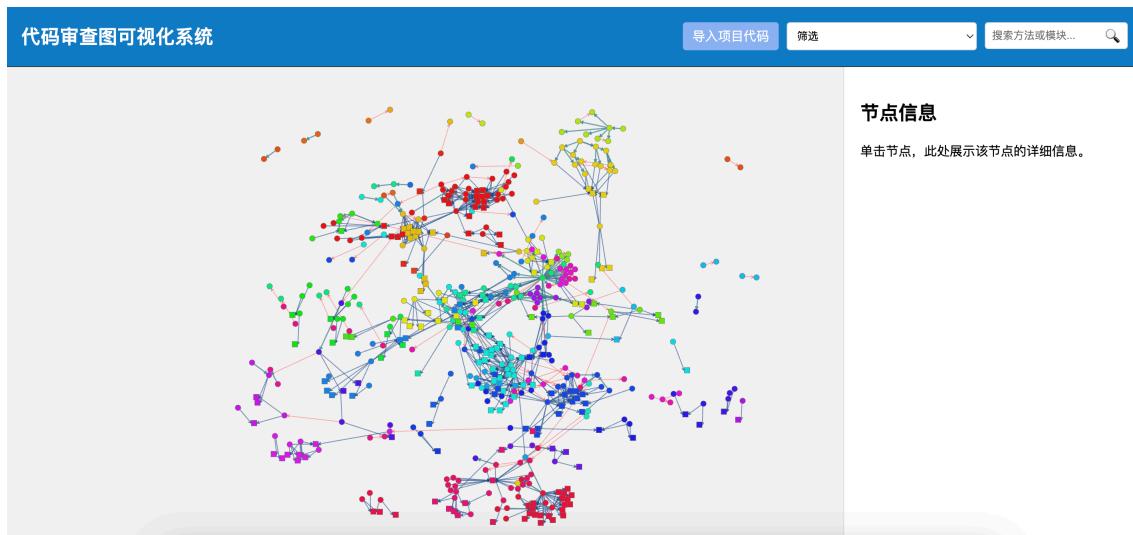


图 4-21 导入项目生成代码审查图

(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4-22所示。

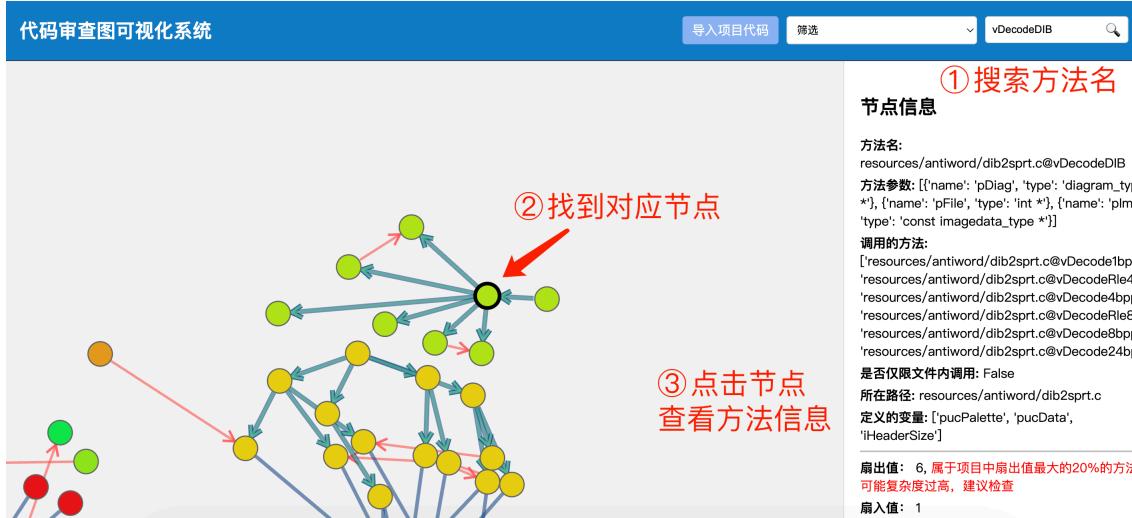


图 4-22 定位开发任务涉及的方法，查看质量信息

(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4-23所示。尤其是变更影响关系，可以帮助用户在进行变更时，提示变更方法所影响的范围，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。

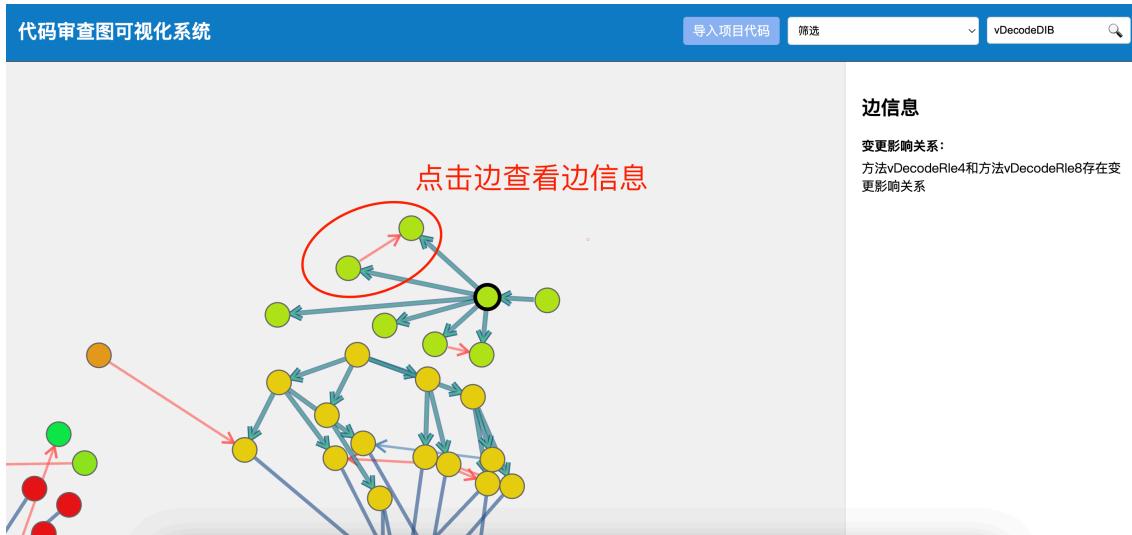


图 4-23 查看方法对应边，按建议安全变更

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。

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

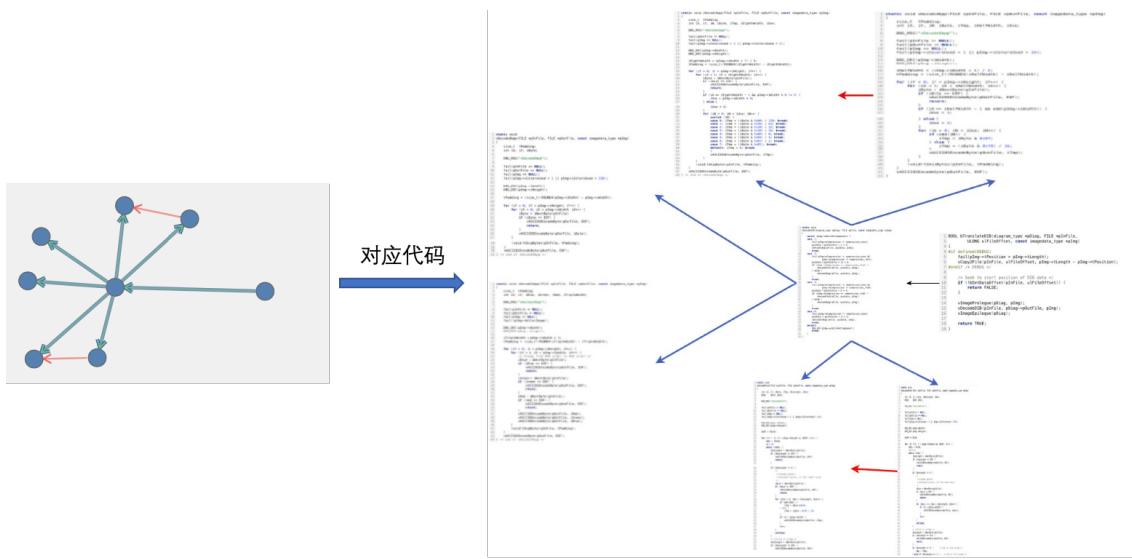


图 4-24 vDecodeDIB 方法上下文

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。

3. 作为一种审查工具帮助审查者进行代码审查 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。

4.8 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。通过实验证明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] KRETSOU M, ARVANITOU E M, AMPATZOGLOU A, et al. Change impact analysis: A systematic mapping study[J/OL]. Journal of Systems and Software, 2021, 174: 110892. <https://www.sciencedirect.com/science/article/pii/S016412122030282X>. DOI: <https://doi.org/10.1016/j.jss.2020.110892>.
- [4] LELOVIC L, HUZINGA A, GOULIS G, et al. Change impact analysis in microservice systems: A systematic literature review[J]. The Journal of Systems & Software, 2025, 219.
- [5] CHEN W, IQBAL A, ABDRAKHMANY A, et al. Large-scale enterprise systems: Changes and impacts[J]. lecture notes in business information processing, 2013.
- [6] ARNOLD R S. Software change impact analysis[M]. Washington, DC, USA: IEEE Computer Society Press, 1996.
- [7] 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.
- [8] GALLAGHER K B, LYLE J R. Using program slicing in software maintenance[J]. IEEE Transactions on Software Engineering, 1991, 17(8): 751-761.
- [9] JITENDERKUMARCHHABRA, MRINAALMALHOTRA, JITENDERKUMARCHHABRA, et al. Improved computation of change impact analysis in software using all applicable dependencies[J]. Springer, Singapore, 2018.
- [10] 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.
- [11] 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.

- [12] 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.
- [13] 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.
- [14] 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.
- [15] UFUKTEPE E, TUGLULAR T. Code change sniffer: Predicting future code changes with markov chain[C]//IEEE Annual Computers, Software, and Applications Conference. 2021.
- [16] YADAV D K, AZAD C, ADHIKARY S D R D. Ciafp:a change impact analysis with fault prediction for object-oriented software[J]. International journal of software innovation, 2022, 10(Pt.3): 933-951.
- [17] 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.
- [18] 姜瑛, 黄培凤, 顾加伟. 基于动态 AST 与 GCN 的代码变更影响范围分析[J]. 昆明理工大学学报 (自然科学版), 2024(4).
- [19] 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).
- [20] 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.
- [21] 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.

- [22] CAI H, SANTELICES R, XU T. Estimating the accuracy of dynamic change-impact analysis using sensitivity analysis[C]//Eighth International Conference on Software Security & Reliability. 2014.
- [23] CAI H, THAIN D. Distia: a cost-effective dynamic impact analysis for distributed programs[C]//the 31st IEEE/ACM International Conference. 2016.
- [24] 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].
- [25] 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.
- [26] 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.
- [27] ZANJANI M B, SWARTZENDRUBER G, KAGDI H. Impact analysis of change requests on source code based on interaction and commit histories[C]//ACM. 2014.
- [28] 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.
- [29] 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.
- [30] 戴鹏. 软件变更影响分析关键技术研究[D]. 北京邮电大学, 2024.
- [31] LLVM-ADMIN TEAM. Clang, a c language family frontend for llvm.[EB/OL]. 2024. <https://clang.llvm.org/>.
- [32] LLVM-ADMIN TEAM. libclang: C interface to clang.[EB/OL]. 2024. https://clang.llvm.org/doxygen/group__CINDEX.html.
- [33] 于长铖, 贺宏良. 面向对象软件变更影响域分析方法[J]. 信息技术与标准化, 2021(04): 41-46.
- [34] 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.
- [35] LEWIS P, PEREZ E, PIKTUS A, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks[Z]. 2020.

- [36] 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.
- [37] 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.
- [38] CHIDAMBER S R, KEMERER C F. A metrics suite for object oriented design[J]. Software Engineering IEEE Transactions on, 1994, 20(6): 476 - 493.
- [39] 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.
- [40] 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.
- [41] 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《面向质量评估的代码变更影响分析及其可视化研究》，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。