

硕士学位论文

(学术学位论文)

面向质量评估的多粒度变更影响分析方法
研究

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

李美娜

哈尔滨工业大学

2025年1月

国内图书分类号：TP311
国际图书分类号：004.02

学校代码：10213
密级：公开

硕士学位论文

面向质量评估的多粒度变更影响分析方法 研究

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

Classified Index: TP311

U.D.C: 004.02

Dissertation for the Master's Degree

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

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

摘要

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

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

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

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

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

各项质量度量，为开发人员提供了清晰的项目质量状况概览。

关键词：代码审查；代码质量评估；变更影响分析；代码度量

Abstract

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

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

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

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

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

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

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

目 录

摘要	I
Abstract	III
第1章 绪论	1
1.1 课题研究的背景和意义	1
1.2 国内外研究现状及分析	3
1.2.1 代码质量研究主题的国内外研究现状	3
1.2.2 代码质量度量的国内外研究现状	4
1.2.3 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	9
1.3 本文的主要研究内容以及各章节安排	10
1.3.1 主要研究内容	10
1.3.2 章节安排	11
第2章 基于方法间关联关系的代码变更影响分析	13
2.1 引言	13
2.2 代码预处理和中间表示生成	13
2.2.1 基于 clang 的抽象语法树生成	13
2.2.2 方法调用链提取与分析	16
2.2.3 全局变量定义-使用链提取与分析	18
2.3 基于依赖关系的变更影响分析	19
2.4 基于克隆关系的变更影响分析	21
2.4.1 代码预处理和分块	22
2.4.2 基于代码克隆检测的变更影响关系提取	24
2.5 基于变更历史和共现关联关系的变更影响分析	27
2.5.1 代码变更历史提取	27
2.5.2 基于共现关联挖掘的变更影响关系提取	28
2.6 基于深度学习的变更影响分析	30
2.6.1 数据集来源和数据清洗	30
2.6.2 基于代码预训练模型的变更影响关系预测	31

2.7 实验结果与分析	33
2.7.1 实验数据与评价方式.....	33
2.7.2 实验设置与评价方式.....	34
2.7.3 实验结果与分析	35
2.7.4 实验结果与对比分析	36
2.8 本章小结	41
第3章 基于RAG的方法间变更影响分析	42
3.1 引言	42
3.2 本章小结	44
第4章 基于代码审查图的代码架构和质量信息可视化	45
4.1 引言	45
4.2 基于代码中间表示的代码质量度量提取	46
4.2.1 基于内聚度缺乏度的内聚性分析	46
4.2.2 基于连通性的内聚性分析	47
4.2.3 方法间耦合性分析	48
4.2.4 方法扇入扇出度量分析	50
4.2.5 基于静态检测工具的代码缺陷检测.....	51
4.3 代码审查图	51
4.3.1 代码审查图构建	51
4.3.2 代码审查图可视化	54
4.3.3 代码审查报告生成	54
4.4 实验结果与分析	56
4.4.1 实验环境与评价方法.....	56
4.4.2 实验结果分析	56
4.5 代码审查图的实际应用	68
4.6 本章小结	70
结 论	71
参考文献	73
哈尔滨工业大学学位论文原创性声明和使用权限	78
致 谢	79

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。

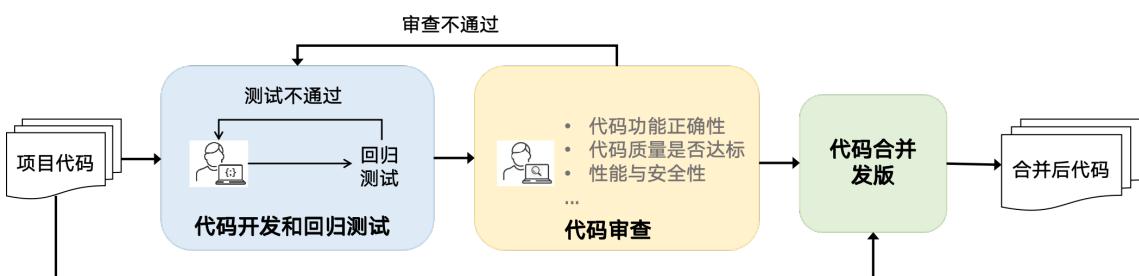


图 1-1 标准代码开发流程

(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不良影响。

传统的代码质量评估分为多种方法。首先是专家进行人工代码审查，这种方法虽最为精确，但是耗时耗力，在面对日益复杂化的系统结构时，显得力不从心^[2]。其次是使用经典度量指标进行评估，虽然能够在一定程度上衡量代码质量，但是这种方法对代码质量的评估仅停留在较小范围内，如类内、模块内的质量，对软件项目整体的质量评估不足。除此之外还有漏洞检测、代码坏味检测等方法从漏洞或规范的角度评估，这类方法评估范围则更小，通常是方法级、切片级、语句级，难以帮助开发者从项目的宏观角度上指导其变更。

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，破除模块和模块间的界限，帮助开发人员更好地理解变更对其他模块或代码的潜在影响，从而做出更加合理的设计和优化决策。因此，从代码变更影响分析的角度出发，不仅能深入理解代码变更对系统整体和各个子模块的潜在影响，还能够提高维护项目的效率^[3]。现有的变更影响分析方法通常只关注基于静态依赖关系的影响，然而软件系统中最难以被用户察觉到，也是最容易导致功能逻辑上变更问题是逻辑上的变更影响关系，正是因为这一关系，才导致大型系统的变更如此棘手，因此如何挖掘这类关系，分析这类关系导致的质量问题，以及如何重构优化成为是亟待解决的难题。

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

本文面向代码质量评估，结合静态分析方法对代码项目进行代码度量提取和缺陷检测，从挖掘代码变更影响入手，提出三种创新方法，弥补传统方法只关注依赖型影响的不足，帮助用户准确变更并给出相应优化方向。最终，提出代码审查图的展示方法，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架构，优化项目的持续维护过程，为维护项目代码提供更有效的保障。

1.2 国内外研究现状及分析

1.2.1 代码质量研究主题的国内外研究现状

软件质量与软件的稳定性、可靠性、健壮性和可维护性等关键特性密切相关，是衡量软件性能和可靠性等指标的重要标准。作为软件质量的重要组成部分，代码质量在整体软件质量中占据着至关重要的地位，直接影响着软件系统的长期可用性与可维护性。根据近年来对代码质量研究的广泛性进行排名^[4]，代码质量研究主题主要可以分为以下几种，

(1) 代码缺陷

代码缺陷是研究最广泛的代码质量主题。相近地，代码漏洞、代码故障等也属于代码缺陷的范畴。软件代码中往往存在大量隐藏的缺陷，这些缺陷可能源于开发人员的疏忽、软件固有的逻辑漏洞或开发语言安全性的脆弱性，如果不进行及时的处理，这些问题可能引发安全漏洞，对软件的安全造成影响。在此主题中，缺陷检测和预测是最主要的研究目的。

目前的研究方法主要分为三种，首先是传统的基于静态分析的方法，如基于代码相似度的检测器^[5-6]或基于模式的检测器^[7-9]。这种检测方法基于一系列预定义的规则、模式或静态模型进行检查，但是通常会产生较高的误报，检测的效果取决于预先定义的规则或模式的质量。其次是基于机器学习的方法。该种方法通过文本分析等方法提取代码特征，结合分类器，如决策树、支持向量机等机器学习技术^[10-12]，在大型数据集上进行预测。随着深度学习技术的发展，逐渐出现了基于深度学习技术的缺陷或漏洞预测方法，这种检测方法主要分为两步，首先通过代码表示学习方法，将代码表示为词嵌入、图表示或序列表示，利用神经网络捕捉代码语义和结构特征。通过端到端的深度学习模型直接从原始代码或中间表示中学习缺陷模式，训练模型后，直接对代码进行缺陷预测。

(2) 代码复杂度

代码复杂度也是代码质量研究领域的重要主题之一。在 Alberto 等人^[4]的研究中统计，代码复杂度在代码质量的主题中研究数量排在第二位。研究方法可分为以下三种，一是基于传统度量指标的研究，这类研究通常是对经典复杂度度量的扩展，如在传统的度量标准，如圈复杂度、Halstead 复杂度等的基础上，结合新型指标进行复杂度评估。二是基于复杂网络来进行研究^[13-14]，这类研究中通常将代码转化为图数据，如抽象语法树、调用图、控制流图、数据流图等，通过分析图结构的复杂性，捕捉代码逻辑的复杂程度。三是基于认知复杂度的方法，认知复杂度衡量执行任务所需的人类努力^[15-16]，是由 Shao 和 Wang^[17]等

人提出的认知指标的重要组成部分，这些度量标准与任何特定的编程范例都没有关联，但是可以从人因工程视角评估代码的认知复杂性，研究代码结构（如嵌套、条件分支）对开发者理解负担的影响，从而衡量代码的复杂性。

(3) 代码内聚度和耦合性

内聚度和耦合性是软件设计中的核心指标，二者之间往往存在着此消彼长的关系，它们的平衡直接影响系统的质量与可维护性。高内聚度表示模块内部功能集中且相关性强，有助于提高代码的可读性和复用性，降低修改时的风险；低耦合性则强调模块间独立性，减少了相互依赖导致的连锁反应，使得系统更容易扩展和测试。二者共同作用，构成了高质量、易维护的软件架构的基础。

近年来对于内聚度的研究大多是从不同的角度提出新的度量标准，有部分研究是对传统度量指标的改进。Chen 等人^[18]提出了基于程序依赖性分析的类内聚度度量方法，该方法从属性与属性、属性与方法、方法与方法之间的依赖关系出发，全面分析类的内聚性。该方法不仅支持单独度量类的内聚度，还可以结合多种度量方式进行综合评估。研究表明，该方法符合 Briand^[19]提出的优良内聚度度量标准的四项基本要求。QU^[20]提出了两个新的类内聚度量 MCC (Method Community Cohesion) 和 MCEC (Method Community Entropy Cohesion)，这两种指标基于类中方法在不同社区间的分布，反映类的内聚程度。研究表明，这两种指标可以提供现有类内聚指标所未反映的额外且有用的信息。

同内聚度类似，对于耦合性的研究大部分也是对传统度量指标的改进。Ma 等人^[21]参考 UML 中类图之间的关系，对 CBO (Coupling Between Object Classes) 度量指标进行了改进，并使用一组形式化评估软件质量性质的定理进行评估，以 JUnit 和 JEdit 为研究对象，对提出度量框架的关联、依赖、泛化关系进行度量研究。结果分析表明，改进后的指标可以较准确地反映面向对象设计中的耦合关系。

除此之外，还包括代码变更、代码可重用性、代码可读性、代码性能和代码安全性等研究主题。

1.2.2 代码质量度量的国内外研究现状

根据不同的研究主题，演化出了很多代码度量套件，这些度量套件被广泛应用于各种软件项目上，在软件生命周期的各个环节（例如，开发、测试、重构）上被使用，以评估软件的整体质量。为了满足不同的度量需求，多年来，人们提出、研究和验证了许多来自不同编程范型的源代码度量，并不断提出新的度量和研究。研究发现^[22]，近年来使用最广泛的是 Chidamber and Kemerer 套件

(CK 套件) 和 Li and Henry (LH) 套件。

表 1-1 CK 套件核心指标

度量	全称	描述
WMC	Weighted Methods per Class	类中的方法的复杂性, 复杂性可以通过方法数量或具体的计算方式(如方法的圈复杂度)来衡量
DIT	Depth of Inheritance Tree	类在继承树中的深度
NOC	Number of Children	一个类直接子类的数量
CBO	Coupling Between Object Classes	类与其他类之间的耦合关系数量
RFC	Response for a Class	类能够响应的所有方法的数量, 包括其直接定义的方法和调用的其他类的方法
LCOM	Lack of Cohesion in Methods	类中方法间缺乏内聚性的程度

CK 套件主要是针对面向对象语言设计的度量套件, 所含度量与其具体含义如表 1-1 中所示。CK 套件中共有六个核心度量, 侧重点各有不同, 其中 WMC 用于衡量类的复杂性。如果值较高, 说明类可能过于复杂, 难以维护。DIT 用于衡量类继承层次的复杂性。较大的 DIT 值表明更深的继承层次, 增加了设计复杂性。NOC 计算了一个类的直接子类的数量, 衡量了类的影响范围, NOC 值较高可能表明父类的职责过于泛化。CBO 衡量了类间的依赖性, 如果值较高, 表示类之间耦合较强, 模块化和重用性可能会较低。RFC 衡量类的复杂性和可能的行为范围, RFC 值较高可能表明类的行为过于复杂。LCOM 衡量类内部方法和字段的相关性, 如果值较高, 说明类可能职责分散, 应考虑拆分或重构优化。CK 套件提供了一种系统化的方法来评估软件质量, 能够帮助开发人员识别潜在问题, 改进代码可维护性和可重用性。虽然 CK 套件主要是应用于面向对象语言的, 但是也可应用于面向过程语言, 只是需要根据具体场景调整权重和解释。

L&H 套件是由 Li 和 Henry 等人提出^[23] 的, 用于评估程序的复杂度、维护性以及开发成本等的度量套件。LH 套件中有一部分度量和 CK 套件是一样的, 除此之外, 还有另外五个核心指标, 具体度量名称和含义见表 1-2。

表 1-2 L&H 套件核心指标

度量	全称	描述
DAC	Weighted Methods per Class	类之间通过抽象数据类型(Abstract Data Type)进行交互的程度
MPC	Message Passing Coupling	类之间通过方法调用进行通信的频率
NOM	Number of Methods	类中所有方法(包括继承和自定义方法)的数量
SIZE2	Second Order Size Metric	通常用类的属性数和方法数的平方和表示
SLOC	Source Lines of Code	源代码行数

其中，较高的 **DAC** 和 **MPC** 值表示类对其他类的依赖程度较高，可能导致耦合度增加。较高的 **NOM** 和 **SIZE2** 可能表明类的职责过多，过于复杂，难以理解和维护。较高的 **SLOC** 值表示代码较为冗长，可能需要优化。

1.2.3 代码变更影响分析方法

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^[24]，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^[25] 提出变更影响分析的概念以来，它一直是代码审查的重要组成部分之一。该方法支持用户在代码变更之前对其影响进行分析，从而估计变更可能造成的负面影响，其优势可总结如下：(1) 能提升代码的稳定性和可靠性。通过分析代码变更对相关模块的影响，开发者可以识别潜在的故障或不一致之处，在变更合入前发现问题，避免引入新的缺陷，从而提高系统的稳定性和可靠性。(2) 有助于模块化设计和低耦合。变更影响分析能够反映出代码模块之间的依赖关系和耦合程度，通过减少不必要的耦合，增强代码的模块化特性，从而提升代码的可维护性和可扩展性。(3) 有助于代码质量的提高。通过变更影响分析能够记录变更过程中的风险评估和解决措施，满足质量保证和审查的需求，提高软件开发过程的透明性和可追踪性。(4) 有助于开发团队协作。变更影响分析为团队提供了清晰的变更范围和影响信息，便于团队成员之间协调工作，减少因沟通不足导致的重复工作或冲突，同时提高代码可读性，有助于开发团队更快速地理解系统，减少后期维护成本。

目前变更影响分析的相关技术主要分为静态分析和动态分析两类。

(1) 静态变更影响分析方法

静态分析方法因其具有高覆盖率和高安全性的优势，广泛应用于对安全性要求较高的软件回归测试中。这类方法通过基于程序的中间表示（如控制流图、调用图等）来进行分析。在静态分析方法中，过程内分析方法通常依赖于程序切片、控制流和数据流等技术^[26-27]，而过程间的影响分析则主要通过计算调用

图或系统依赖图的传递闭包来揭示不同模块之间的依赖关系^[28-30]。

Schrettner 等人^[31]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^[32]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。

近年来，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（CIA）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^[33]。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。

（2）动态变更影响分析方法

尽管静态影响分析在软件工程中因其较高的覆盖率和较好的安全性而被广泛应用，但其分析结果往往存在精确性不足的问题。这是因为静态分析主要依赖程序的中间表示（如控制流图、调用图等）进行推理，而这些模型无法捕捉程序在运行过程中可能出现的实际行为。为了弥补这一不足，一些研究者转向动态影响分析。与静态分析不同，动态影响分析是在程序运行时收集实际执行信息，并基于这些运行时数据计算程序中各个部分的影响关系。尽管动态分析通常能够提供更为精确的结果，但其成本较高，而且在面对复杂的系统时，无法确保分析结果的完全安全性。

尤其是在面向对象编程的系统中，由于程序实体之间的依赖关系较为复杂且难以静态建模，动态分析的结果有时会产生不精确性的影响。为了提高动态分析的精确性，Huang 等人^[34]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^[35-36]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^[37]。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。

(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^[29,38-41]。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^[29]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^[40]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^[41]致力于改进现有的耦合分析算法，尤其是在软件变更的上下文中。TARMAQ 算法是他们提出的一种新型算法，在性能上明显优于 ROSE^[42]和 SVD 等传统算法。TARMAQ 通过挖掘代码库中的耦合关系，能够更精确地揭示源代码之间的依赖和关联，从而提高了变更影响集的生成效率和准确性。

Huang 等人^[43]则提出了一种增强型方法，通过将历史变更模式映射到当前的变更影响分析任务中，解决了跨项目场景中的变更影响分析问题。在许多实际应用中，变更影响分析不仅仅局限于单一项目，而是需要跨项目、跨系统进行分析。Huang 等人的方法通过借助历史变更模式，能够在不同项目间共享和迁移影响分析的知识，进而提升了变更影响分析的普适性和适应性。

1.2.4 现有方法存在的问题与分析

通过对上述方法的分析可以看出，尽管近年来提出了许多新的质量度量指标，但在实际应用中，传统的流行指标依然是使用最多的。这表明，大部分的代码质量评估仍然聚焦于缺陷、内聚性和耦合度等核心特征，这些特征反映了用户最为关注的方面，并且具有较强的客观性。因此，在进行代码质量分析时，应当充分考虑这些主题，并在度量指标中予以涵盖。

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

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

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

1.3 本文的主要研究内容以及各章节安排

1.3.1 主要研究内容

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

(1) 代码中间表示与质量评估度量提取

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

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

(2) 面向代码质量评估的变更影响分析方法研究

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

(3) 基于代码审查图的代码架构和质量信息可视化

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

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

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

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

1.3.2 章节安排

本文的章节安排如图 1-2。

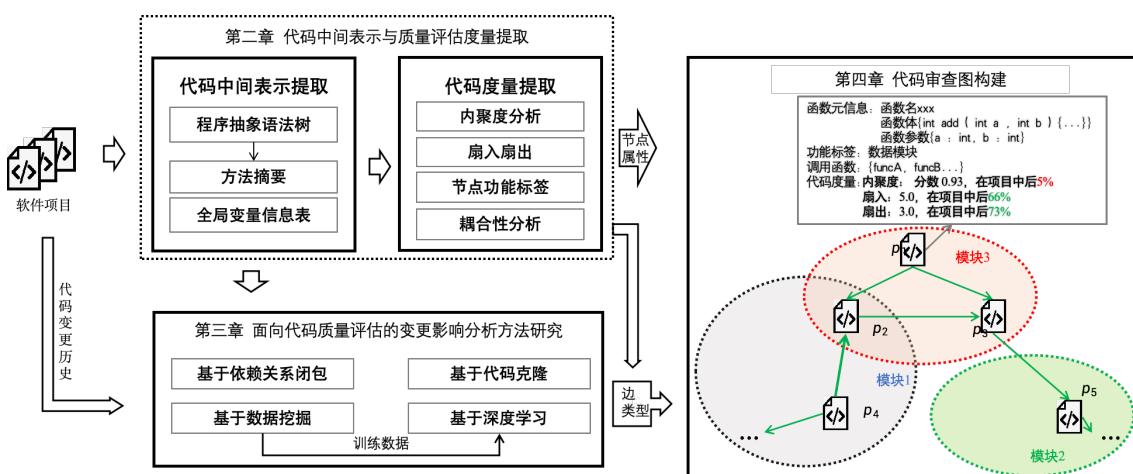


图 1-2 章节安排

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

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

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

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

第2章 基于方法间关联关系的代码变更影响分析

2.1 引言

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

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

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

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

为了解决上述问题，本文以 C/C++ 项目为研究对象，实现了基于依赖关系闭包的传统分析方法，并实现了基于依赖关系、基于克隆关系以及基于变更历史和共现关联关系的的变更影响分析方法，从而能够全面挖掘这依赖型和逻辑型的变更影响关系，并通过实验对比分析了这三种方法在提取这两类影响关系的有效性。

2.2 代码预处理和中间表示生成

2.2.1 基于 clang 的抽象语法树生成

抽象语法树（Abstracted Syntax Tree, AST）是一种用来表现编程语言构造的树状结构，它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。在抽象语法树中，节点之间的连接表明了它们之间的层级关系，即父节点与子节点的关系。通过这样的结构，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^[44]。词法分析通过识别 Token 将程序代码分解成基本单元。语法分析在此基础上识别程序的语法结构，构造抽象语法树。语义分析消除语义模糊，生成属性信息，让计算机生成目标代码。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^[45]。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。

这里通过一个简单的例子来说明 libclang 的使用方式。如图 2-1 所示，这里定义了一个简单的 C 语言文件，文件中声明并定义了两个方法和一个全局变量，主函数用于计算两个数的和。

```

1  #include <stdio.h>
2
3  int globalVar = 10;
4
5  int addNumbers(int a, int b) {
6      return a + b;
7  }
8
9  void main() {
10     int num1 = 5;
11     int num2 = 7;
12     int sum = addNumbers(num1, num2);
13     printf("Sum of %d and %d is: %d\n", num1, num2, sum);
14 }
```

图 2-1 示例代码

这段代码由 Clang 解析生成抽象语法树后，得到的树结构如图 2-2 所示。

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

```

CursorKind.TRANSLATION_UNIT resources/template.c
| CursorKind.VAR_DECL globalVar
| | CursorKind.INTEGER_LITERAL
| | CursorKind.FUNCTION_DECL addNumbers
| | | CursorKind.PARM_DECL a
| | | CursorKind.PARM_DECL b
| | | CursorKind.COMPOUND_STMT
| | | | CursorKind.RETURN_STMT
| | | | | CursorKind.BINARY_OPERATOR
| | | | | | CursorKind.UNEXPOSED_EXPR a
| | | | | | | CursorKind.DECL_REF_EXPR a
| | | | | | | CursorKind.UNEXPOSED_EXPR b
| | | | | | | CursorKind.DECL_REF_EXPR b
| | | | | | CursorKind.FUNCTION_DECL main
| | | | | | CursorKind.COMPOUND_STMT
| | | | | | | CursorKind.DECL_STMT
| | | | | | | | CursorKind.VAR_DECL num1
| | | | | | | | | CursorKind.INTEGER_LITERAL
| | | | | | | | | CursorKind.DECL_STMT
| | | | | | | | | | CursorKind.VAR_DECL num2
| | | | | | | | | | | CursorKind.INTEGER_LITERAL
| | | | | | | | | | | CursorKind.DECL_STMT
| | | | | | | | | | | | CursorKind.VAR_DECL sum
| | | | | | | | | | | | | CursorKind.CALL_EXPR addNumbers
| | | | | | | | | | | | | | CursorKind.UNEXPOSED_EXPR addNumbers
| | | | | | | | | | | | | | | CursorKind.DECL_REF_EXPR addNumbers
| | | | | | | | | | | | | | | CursorKind.UNEXPOSED_EXPR num1
| | | | | | | | | | | | | | | | CursorKind.DECL_REF_EXPR num1
| | | | | | | | | | | | | | | | CursorKind.UNEXPOSED_EXPR num2
| | | | | | | | | | | | | | | | | CursorKind.DECL_REF_EXPR num2
| | | | | | | | | | | | | | | | | CursorKind.CALL_EXPR printf
| | | | | | | | | | | | | | | | | | CursorKind.UNEXPOSED_EXPR printf
| | | | | | | | | | | | | | | | | | | CursorKind.DECL_REF_EXPR printf
| | | | | | | | | | | | | | | | | | | CursorKind.UNEXPOSED_EXPR
| | | | | | | | | | | | | | | | | | | | CursorKind.UNEXPOSED_EXPR
| | | | | | | | | | | | | | | | | | | | | CursorKind.STRING_LITERAL "Sum of %d and %d is: %d\n"
| | | | | | | | | | | | | | | | | | | | | CursorKind.UNEXPOSED_EXPR num1
| | | | | | | | | | | | | | | | | | | | | CursorKind.DECL_REF_EXPR num1
| | | | | | | | | | | | | | | | | | | | | CursorKind.UNEXPOSED_EXPR num2
| | | | | | | | | | | | | | | | | | | | | | CursorKind.DECL_REF_EXPR num2
| | | | | | | | | | | | | | | | | | | | | | CursorKind.UNEXPOSED_EXPR sum
| | | | | | | | | | | | | | | | | | | | | | CursorKind.DECL_REF_EXPR sum

```

图 2-2 示例代码对应的抽象语法树结构

代码元素的名称、类型、位置等信息、获取 cursor 的父节点或子节点等。本文通过操作游标，遍历 AST，获取整个 AST 的结构。

Clang 定义了一套节点类型标识。AST 的顶层节点类型是 Translation_Unit 标签，表示一个翻译单元，对 AST 树的遍历，实际上就是遍历整个 Translation_Unit。Function_Decl 指的是方法定义，在 clang 中是不区分方法声明和方法定义的，统一用 Function_Decl 来标识，两个区分主要看是否有方法体，在 libclang 中提供了程序接口供开发者调用判断。Parm_Decl 是参数节点，上面的例子中，方法 addNumbers 有两个参数 a 和 b。CompoundStmt 代表大括号，方法实现、struct、enum、for 的 body 等一般用此标签包起来。DeclStmt 是定义语句，里边可能有 VarDecl 等类型的定义，VarDecl 是对变量的定义。CallExpr 标签表示方法调用 Expr，子节点有调用的参数列表。ReturnStmt 表示返回语句。

2.2.2 方法调用链提取与分析

在使用 libclang 提取代码的抽象语法树后，遍历整棵树来提取方法之间的调用关系。这部分我们重点关注抽象语法树上的方法节点，以及方法节点内部的调用节点，分别对应着代码中方法的定义和方法内部对其他方法的调用。

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

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

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

算法 2-1 方法调用链提取

Input: 项目中的所有代码文件: *files*

Output: 方法摘要表: *functions*

```

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

```

2.2.3 全局变量定义-使用链提取与分析

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

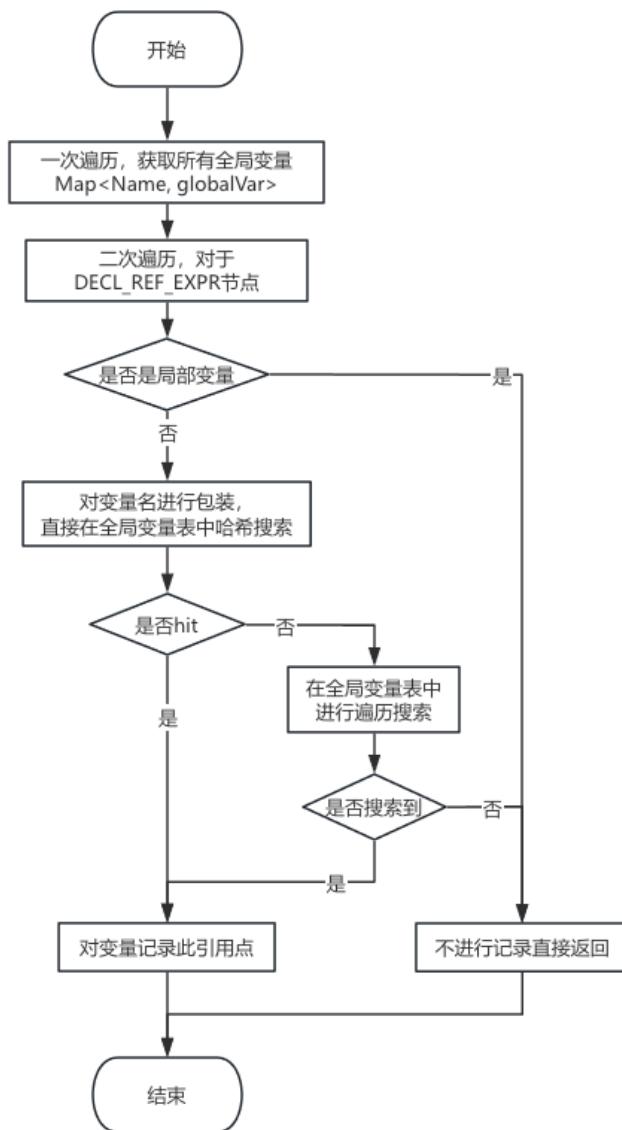


图 2-3 全局变量提取流程图

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

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

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

分析结束后，将全局变量的信息进一步整理，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 *<globalVarID, type, use, scope, file >* 共 5 部分组成，分别表示全局变量的唯一 ID 标识，变量类型，变量的引用点所在的方法，变量的作用域以及变量所在模块。

2.3 基于依赖关系的变更影响分析

在代码变更影响关系分析中，依赖关系传递闭包分析被广泛应用，这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^[33]。其核心思想是利用依赖关系的传递性，通过构建和分析依赖图，揭示所有可能受到影响的代码模块或单元。

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

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

量 C，在图中增加一条从节点 A 指向节点 C 的有向边。

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

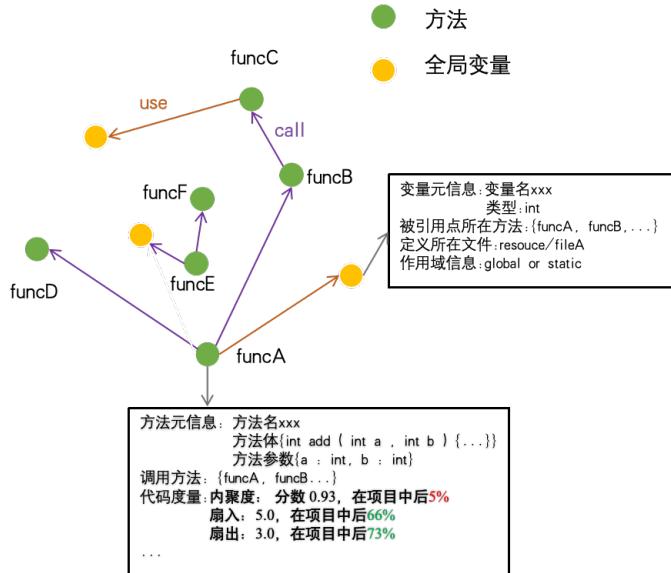


图 2-4 依赖图示例

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

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

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式 2-2。

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

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

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

2.4 基于克隆关系的变更影响分析

代码克隆（Code Clone）是指在代码中存在两段或多段内容相似或完全相同的代码片段。这些代码片段可能由于直接复制粘贴或手动修改而产生，通常是在软件开发过程中为了快速复用功能、减少重复实现或其他原因而引入的。代码克隆虽然可以在短期内提高开发效率，但在长期来看，可能对代码的维护和演化带来负面影响。

代码克隆主要分为以下三种类型：(1) 完全克隆：两段代码完全相同，除了空白符、注释或格式化上的差异，这种克隆通常是直接复制粘贴的结果；(2) 语法克隆：两段代码的结构和功能相同，但在变量名、方法名等命名上存在一些简单修改。(3) 修改克隆：两段代码基本相似，但在部分语句上有较大改动。例如，某些逻辑被修改、删除或添加。这种克隆通常反映了代码的部分复用。

在项目代码中，克隆的代码片段是完全或部分相同的，因此它们在逻辑上往往具有相同的功能或行为。如果对其中一个克隆片段进行了变更（例如修复了一个 bug、添加功能或进行优化），那么在其他地方相同或相似的代码也可能

需要同步修改，否则可能会导致系统的不一致性或错误，这是典型的逻辑型变更影响关系。

因此，本文基于方法间的克隆关系进行变更影响分析。追踪在代码变更过程中，克隆代码之间的相互依赖及其潜在影响，从而为开发者的安全变更提供参考。这一方法不仅有助于揭示代码重复带来的潜在风险，还能为开发人员提供系统的变更影响评估，从而优化代码维护和改进的决策过程，推动软件质量的持续提升。

该方法主要分为两步，首先对源程序进行预处理，通过代码分段及代码指纹提取的方式对源程序进行编码，生成代码序列数据库。随后利用频繁模式挖掘算法 ClaSP 得到克隆代码列表，具体的算法流程如图 3-2 所示。

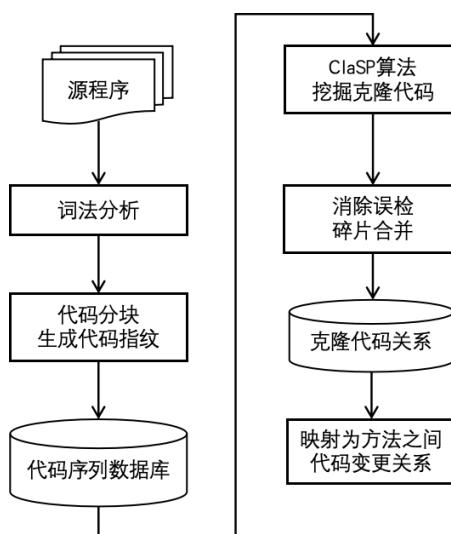


图 2-5 基于代码克隆的变更影响分析方法流程

2.4.1 代码预处理和分块

为了精确有效地检测代码克隆，需先对源代码进行详细的预处理，以消除可能影响克隆检测准确性的各种干扰因素。

代码预处理

- 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。因此，去除注释可以避免注释的差异干扰克隆检测的结果，从而确保算法专注于代码的实际逻辑。

- 去除头文件引用语句：在 C/C++ 程序中，源代码文件往往会包含多个头文件，而不同的源文件可能引用相同的头文件。如果不对头文件进行统一的处

理，算法就可能在不同的源文件中检测头文件引用的代码克隆情况，进而导致无谓的计算冗余。这不仅增加了处理的时间和空间复杂度，还可能影响克隆检测的效率和准确性。

- 程序标准化：为了避免因变量名的变化导致漏检，在方法内部对变量名进行统一标准化处理，将所有局部变量和参数名替换为预定义的标准变量名，从而消除变量名不同带来的影响。这一过程有助于聚焦于代码的逻辑和功能层面，而非单纯依赖变量命名的差异，从而提高克隆检测的准确性和可靠性。

通过上述预处理过程，可以生成一个清晰、标准化的源代码文件，不仅消除了无关的噪声，还确保了源代码的逻辑和结构能够被准确捕捉，为后续的克隆检测提供了高质量的输入数据。

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

- 顺序结构：按固定行数分块，行数可由用户定义，默认为 6 行一块。行数越小则识别结果越精准，越能识别更细小的代码克隆情况。

- 控制结构：将选择结构 (if、then、else、endif、switch)、循环结构 (while、for)、和域结构 ({、}) 共同描述为控制结构识别并根据关键分块词进行分块。这是由于控制语句是代码逻辑的重要分界点，将它们作为分块的标准可以确保检测系统能聚焦于实际功能的逻辑边界。

此外，在代码克隆检测中，还需要遵循一些附加规则以确保检测的准确性和一致性。

- 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，从而确保检测过程更侧重于代码的逻辑结构，而非视觉格式上的不同。

- 分块过程中记录位置信息：包括代码块所在的文件路径、代码块的起始行号和终止行号、代码块的总行数，以及每个块包含的具体行数共 5 个信息。通过这种方式，不仅能够清晰地标识出每个代码块的位置，还能为后续的克隆碎片合并提供必要的上下文支持。

通过这些步骤和规则，代码被拆分成更小、更一致的单元，使克隆检测能够更加高效且精准地识别功能相似的代码块。

代码指纹提取 这一步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。通过这种方式, 不同的代码片段信息被压缩, 转换为具有独特标识符的形式, 从而在后续的分析中实现高效的比较和匹配。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, 以确保在保证高效性的同时, 能够快速、准确地对代码进行相似性分析和重复性检测。

2.4.2 基于代码克隆检测的变更影响关系提取

序列数据挖掘 (Sequence Data Mining, SDM) 是时序数据挖掘领域的一个重要研究方向, 旨在从给定的输入数据库中, 探索在大量对象之间随时间频繁出现的模式。判断一个模式是否具有意义的阈值被称为最小支持度。SDM 已被广泛研究, 并在多个领域得到了应用, 如 DNA 序列中的基序发现、顾客购买序列分析、网页点击流分析等。本文中, 将代码指纹片段作为序列, 利用序列数据挖掘算法, 挖掘频繁出现的模式, 提取对应的代码克隆序列。

(1) 数据挖掘领域基本概念

项集^[46]: 设有一个集合 $I = \{i_1, i_2, \dots, i_n\}$, 其中 i_i 表示一个项。项集即为这个项的非空子集。一个项可以出现在多个项集中, 但同一项集中的每个项只能出现一次, 即不允许项集包含重复的项。在本文中, 项集指的是代码行的指纹。

序列^[46]: 序列是一个由项集组成的有序列表, 记作 $s = (s_1, s_2, \dots, s_n)$, 其中每个 s_i 是项集。每个项集 s_i 可以进一步表示为 (x_1, x_2, \dots, x_m) , 其中 x_m 是项。本文中序列指代码块的指纹。

子序列^[46]: 考虑两个序列 $\alpha = (a_1, a_2, \dots, a_n)$ 和 $\beta = (b_1, b_2, \dots, b_m)$, 如果存在一组整数 $1 \leq j_1 < j_2 < \dots < j_n \leq m$, 使得序列 $(a_{j_1}, a_{j_2}, \dots, a_{j_n})$ 是序列 β 的子序列, 则称序列 α 为序列 β 的子序列, 序列 β 为序列 α 的超序列。

序列数据库^[46]: 序列数据库是由一组元组 $\langle \text{sid}, s \rangle$ 构成的集合, 其中 sid 为序列的标识符, s 为序列。如果序列 α 是元组 $\langle \text{sid}, s \rangle$ 中序列 s 的子序列, 则称该元组包含序列 α 。序列数据库中包含序列 α 的元组的数量被称为序列 α 的支持度。本文中指的是项目代码所有分块的集合。

频繁子序列^[46]: 给定一个正整数 min_support 作为支持度阈值, 如果序列 α 的支持度大于或等于 min_support , 则称序列 α 为频繁子序列。所有频繁子序列的集合记为 FS , 即 Frequent Subsequences。

序列支持度：记作 $\sigma(\alpha, D)$ ，是指在数据库 D 中包含序列 α 的序列的总数。如果一个模式或序列至少出现给定的用户指定阈值 min_support ，即最小支持度，则称其为频繁序列。频繁序列挖掘的问题就是在给定的输入数据库中，找到满足最小支持度阈值的频繁序列集合 FS 。本文中即是找到频繁出现的代码。

闭合序列：如果一个频繁序列 α 没有其他超序列与其具有相同的支持度，则称 α 为闭合序列。所有频繁闭合序列的集合记作 FCS 。更正式地说，若对所有 $\beta \in FS$ ，有 $\alpha \subseteq \beta$ 且 $\sigma(\alpha, D) = \sigma(\beta, D)$ ，则 $\alpha \in FCS$ 。

闭合序列挖掘的问题就是在给定的输入数据库中，找到满足最小支持度阈值的闭合序列集合 FCS 。显然，频繁闭合序列的集合要小于所有频繁序列的集合。这是因为在频繁模式的集合中，很多模式可能是冗余的。例如，某些频繁模式的支持度相同，且其中一个是另一个的超序列。对于这些模式而言，冗余的超序列模式没有提供比其子序列模式更多的信息。闭合频繁模式的引入可以去除这些冗余，保留那些最具代表性和信息量的模式。

这里举例说明，表 3-1 为序列数据库示例，这里设定最小支持度阈值为 2。

表 2-1 序列数据库示例

序列 id (sid)	序列
1	$\langle(a)(ab)(bc)\rangle$
2	$\langle(a)(abc)\rangle$
3	$\langle(d)(a)(ab)(bc)\rangle$
4	$\langle(d)(ad)\rangle$

在这个例子中，第一个序列中有 3 个项集，分别是 (a)，(ab) 和 (bc)，以此类推。根据上述概念计算得到，这个例子中的闭合频繁序列 $FCS=\{\langle(a)\rangle, \langle(d)(a)\rangle, \langle(a)(ab)\rangle, \langle(a)(bc)\rangle, \langle(a)(ab)(bc)\rangle\}$ ，共 5 个，而频繁序列有 27 个。因此识别闭合频繁模式具有重要的压缩性，并且模式信息也是不丢失的。

(2) ClaSP 算法

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

首先找到所有的频繁的 1-序列（即长度为 1 的序列），然后，对于所有频繁的 1 序列，递归地调用 DFS-Pruning 方法来探索相应的子树（通过进行深度优先搜索）。对所有频率为 1 的序列进行此处理，得到 FCC，最后，算法结束去除 FCC 中出现的非闭合序列。

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

```

算法 2-3 DFS-Pruning 算法

Input: 当前模式 p , 候选项集 S_n 和 I_n **Output:** 更新后的频繁模式集 FCC

```

1  $Stemp \leftarrow \emptyset Itemp \leftarrow \emptyset F_i \leftarrow \emptyset P_s \leftarrow \emptyset P_i \leftarrow \emptyset$ 
2 if  $\neg \text{checkAvoidable}(p, I(D_p))$  then
3   for all  $i \in S_n$  do
4     if  $p' = (s_1, s_2, \dots, s_n, \{i\})$  is frequent then
5        $Stemp \leftarrow Stemp \cup \{i\}$ 
6        $P_s \leftarrow P_s \cup \{p'\}$ 
7     end
8   end
9    $F_i \leftarrow F_i \cup P_s \cup \text{ExpSiblings}(P_s, Stemp, Stemp)$ 
10  for all  $i \in I_n$  do
11    if  $p' = (s_1, s_2, \dots, s_n \cup \{i\})$  is frequent then
12       $Itemp \leftarrow Itemp \cup \{i\}$ 
13       $P_i \leftarrow P_i \cup \{p'\}$ 
14    end
15  end
16   $F_i \leftarrow F_i \cup P_i \cup \text{ExpSiblings}(P_s, Stemp, Itemp)$ 
17 end
18 return  $F_i$ 

```

其中 DFS-Pruning 算法的核心流程如 3-2 所示。通过递归生成候选模式（包括 s-扩展和 i-扩展，分别在模式末尾和任意位置添加新元素）并检查其支持度，最终返回以当前模式 p 为前缀的所有频繁模式集。算法输入包括当前频繁模式 p 以及用于执行扩展操作的候选项集 S_n 和 I_n 。

在 s-扩展阶段，算法遍历 S_n 中的每个候选项 i ，检查扩展后的模式是否为频繁模式。若为频繁模式，则将其加入 $Stemp$ 和 P_s 中，并通过递归调用 `ExpSiblings` 方法继续扩展。i-扩展部分则类似，遍历 I_n 中的候选项，检查并加入频繁扩展模式到 $Itemp$ 和 P_i 中，最终进行 i-扩展的剪枝处理。

最终，算法返回以模式 p 为前缀的所有频繁模式集 F_i ，其中包括通过 s-扩展和 i-扩展生成的所有频繁模式。这些操作通过递归遍历模式树，确保高效地发现所有频繁模式。

剪枝时，通过检查对应模式的子序列和超序列的支持度，将序列的节点进行合并，防止继续遍历冗余节点。最终得到的 FCS 即为所有克隆代码集。

(3) 合并碎片

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的相似性。

这种方法间的克隆关系不仅揭示了代码的重复性，还反映了不同方法之间在代码修改过程中的潜在影响。方法与方法之间的克隆关系可以被视为一种变更传播的路径，指示了某一方法的修改可能如何影响其他方法。因此，提取出的克隆关系即代表了方法之间的变更影响关系。

2.5 基于变更历史和共现关联关系的变更影响分析

2.5.1 代码变更历史提取

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，本文设计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。

本方法的具体过程主要分为两部分，首先是对代码变更历史的收集与整理，然后通过数据挖掘技术提取并分析方法之间的变更影响关系。其中代码变更历史提取具体流程如下：

(1) 收集项目代码库及变更历史记录

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，本文的分析主要集中在 Git 项目上。首先，克隆项目的代码库到本地，项目中的.git 文件夹中包含了版本变更历史记录，包括所有提交（commit）记录等。每个提交代表着代码库的一次变更，包含了代码的修改、删除或新增文件等信息。通过 git log 命令获取每次 commit 的详细信息，包括每个提交的哈希值、作者、日期和提交信息等。

(2) 提取每个提交的变更信息

每个提交不仅仅是一个单一的代码变更，往往涉及多个文件的更改。因此，对于每个提交，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。

(3) 定位变更代码行所属的方法

在提取了具体的代码变化后，定位这些变化的代码行所在的方法。通过 lib-clang 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，将变更的代码行的位置与方法位置进行匹配，进而得到变更的代码行所在的方法。

(4) 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。这个列表反映了在特定提交中发生变更的所有方法，并为后续的变更影响分析提供了基础数据。用 Map<commitID, List<Methods>> 的结构存储每个 commit 变更的方法，作为序列数据库，便于后续分析与处理。

2.5.2 基于共现关联挖掘的变更影响关系提取

基于关联规则（Association Rules）的数据挖掘方法是反映事物之间相互依存性和关联性的一个重要数据挖掘技术，旨在从大量数据中挖掘出有价值的项

之间的相关关系。共现关系可以视为关联规则的一种表现形式，它描述了在给定集合中，某一组项（或特征）经常出现在同一事务中。例如，在零售分析中，常见的共现关系是“购买了面包的顾客通常也会购买牛奶”。在这种情况下，“面包”和“牛奶”是一对共现项。

在本文的代码变更影响分析中，共现关系描述的是在一次提交中，哪些方法经常同时发生变更。如果两个方法在多个提交中频繁一起变动，则它们之间可能存在某种依赖关系或变更影响关系。通过分析这些方法在多个提交中被频繁同时修改的情况，我们能够识别方法之间的变更影响路径，从而揭示它们的潜在关联性。

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

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

置信度表示共现项中一个出现后，另一个项出现的概率。变更分析中，置信度度量表示当方法 A 被修改时，方法 B 被修改的概率，计算公式如 3-4。

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

在前文提取的序列数据库中，进行如下步骤的计算，详细流程见算法 3-3。

首先，遍历序列数据库，搜索并得到候选 1 项集，即包含单个元素的项集，这些元素代表在变更历史中被修改过的方法。计算对应的支持度，剪枝去掉低于支持度的 1 项集，得到频繁 1 项集，频繁 1 项集中的元素反映了在变更历史中频繁出现的被修改方法。

基于频繁 1 项集中的元素，通过连接操作生成候选的频繁 2 项集。通过再次遍历序列数据库，计算候选频繁 2 项集的支持度，并筛选去除那些支持度低于设定阈值的项集。经过这一筛选步骤后，得到真正的频繁 2 项集。频繁 2 项集中的元素表示了那些在变更历史中频繁同时被修改的两个方法对。

在得到频繁 2 项集之后，进一步筛选出置信度高于或等于设定阈值的方法对。最终，留下的即为那些频繁同时变更，并且存在强关联关系的方法对，表明它们在变更过程中有着较为显著的相互依赖关系，即方法对之间的变更影响

关系。

算法 2-4 变更影响方法对挖掘算法

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  $change\_impact\_pairs$ 

```

2.6 基于深度学习的变更影响分析

2.6.1 数据集来源和数据清洗

当项目代码拥有丰富的变更历史时, 可以通过前文中介绍的数据挖掘方法, 提取具有变更影响关系的方法对。这种方法通过分析历史提交记录中方法间的共现频率, 识别出在多次变更中相互依赖或相互影响的方法。然而, 并不是所有软件项目都有变更历史可供我们分析, 在仅有项目源代码的情况下, 数据挖掘方法无法直接应用, 但是只使用基于依赖闭包和基于克隆代码的方法, 则无

法识别除这两种关系外更深层次的变更影响。因此本文提出基于深度学习的变更影响分析方法，通过训练深度学习模型，对变更影响关系进行预测，以弥补另两种分析方法的不足。

为了识别那些来源除了依赖关系和代码克隆得到的代码变更影响关系，需要收集包含深层变更影响关系的方法对。而 3.4 节中根据数据挖掘的变更影响关系分析的方法，恰巧具有这一特征。该方法可以识别出那些在多个提交记录中频繁同时发生变更的方法对，由于我们的标准设置十分严格，因此识别的变更影响关系具有较强的可靠性，可作为数据集的正例进行收集。此外，为了构建平衡的数据集，通过对项目中的方法进行随机抽样，从中选取一些不具有变更影响关系的方法对，作为数据集的负例进行收集。

在数据集收集过程中，本文面临了几个挑战。首先，一些看似活跃的项目（例如频繁提交的项目）实际上并未挖掘到大量可用数据。分析原因是由于项目规模较大，文件数量庞大，开发者的工作内容往往不具有交集，大部分的方法不会被二次更改，因此难以出现频繁的样例。其次，对于一些发展时间较长的项目，因为提交记录众多（如数万次提交），所以在收集数据时需要处理大量的提交，这使得数据处理的过程非常耗时。

为了解决这些问题，本文在选择项目时设定了一些标准。首先，优先选择规模适中的项目，这些项目既足够活跃，又不会因为过于庞大的代码库而导致数据处理困难。同时，考虑到提交记录较多的项目可能导致数据量过大，本文限制了采集范围，只选择了最近的提交记录（例如最新的 2000 个提交）。这一策略有两个主要考虑：一方面，确保数据量足够大，以支持后续的挖掘工作；另一方面，聚焦于那些近期频繁变动的方法，排除较为稳定且变化较少的旧函数，从而更好地反映出当前项目的活跃度和变更趋势。

2.6.2 基于代码预训练模型的变更影响关系预测

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

考虑到代码理解的复杂性与深度，本文采用了基于预训练模型的代码表示学习方法，选择了 CodeBERT 作为核心模型。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。模型架构如图 3-3 所示。

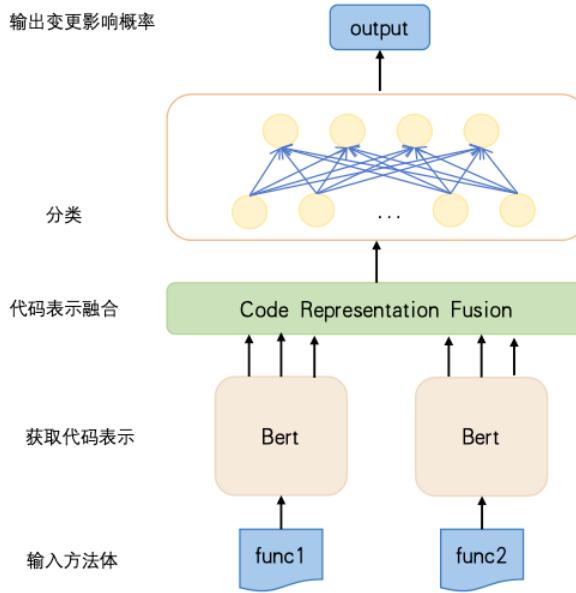


图 2-6 基于代码预训练的深度学习模型架构

首先，将两个方法体 F_a, F_b ，先通过代码预训练模型进行编码

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

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

得到方法的向量化表示 E_a, E_b ，通过拼接融合这两个向量，将融合后的向量表示送入一个由两层组成的多层感知机（Multilayer Perceptron, MLP）中，再通过 softmax 层进行分类处理，将模型的输出转化为一个概率分布，表示两个方法体之间存在关系的概率。

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

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

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

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

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

最后与真实标签计算交叉熵损失，得到 loss，计算梯度，优化模型参数

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

2.7 实验结果与分析

2.7.1 实验数据与评价方式

实验数据 本文从影响力或社区活跃程度的角度出发，收集了表 2-2 中所示的软件项目为被测项目进行实验。这些项目在 `github` 上的收藏数均在千以上，说明这些项目在开源社区中有着一定的影响力，使用范围比较广泛。除 `antiword` 项目之外，其他项目有着比较活跃的社区，说明其还在不断更新迭代过程中，所以能提供较为丰富的变更历史，以供数据挖掘方法的实验分析。

表 2-2 被测项目

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

根据每个示例项目的上一个版本和示例版本间的版本变更，分别各自选取 50 个变更方法，通过分析 `commit` 得到真实的被影响方法集 AIS (Actual Impact Set)，作为测试集，这里分别统计了依赖型和逻辑型的变更影响关系，得到的数据如表 2-3 所示。再分别通过前述方法进行检测，得到估计的被影响方法集 EIS (Estimated Impact Set)，通过评价指标评估方法的有效性。

表 2-3 被测项目数据统计

项目名称	commit 数	变更点数	依赖型 AIS	逻辑型 AIS	总计
TheAlgorithms	-	50	162	34	196
antiword-0.37	-	50	197	89	186
jemalloc-5.3.0	-	50	112	21	133
libbpf-1.1	-	50	322	27	349
librdkafka-2.1.0	-	50	153	43	203
FFmpegKit-5.1.0	-	50	175	27	202
总计		300	1121	241	1362

基于深度学习的变更影响分析方法的数据集收集方式如 3.5.1 节所述，为

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

表 2-4 数据集统计信息

项目名称	正例对数	负例对数	总对数
TheAlgorithms	97	1000	1097
jemalloc-5.3.0	593	1000	1593
libbpf-1.1	401	1000	1401
librdkafka-2.1.0	932	1000	1932
FFmpegKit-5.1.0	17	1000	1017
总计	2040	5000	7040

评价指标 评价指标如式 (3-13) (3-14) (3-15) 所示，真实的被影响方法表示为 AIS (Actual Impact Set)，每种方法检测得到的结果为估计的被影响方法 EIS (Estimated Impact Set)，按逻辑型和依赖型对关系进行划分，根据这两个值计算精确度、召回率和 F-measure，这三种评价指标在信息检索的场景下被广泛使用，本章中用于评价变更影响分析方法的有效性。

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

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

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

2.7.2 实验设置与实验过程

实验设置

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

表 2-5 模型参数设置

超参数	数值
Token embedding size	768
codeBERT learning-rate	1e-5
codeBERT dropout	0.4
Classifier learning-rate	1e-4
Adam β_1	0.95
Adam β_2	0.999
batch_size	64

- 基于共现关系方法置信度：设为 1。
- 基于共现关系方法支持度：设为 2 和 3 分别进行实验，选择表现较好的与其他方法做对比。

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

表 2-6 基于代码预训练模型的变更影响关系在数据集上的实验结果

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

2.7.3 实验结果与对比分析

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

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

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

1. 针对于 RQ1 的实验

三种方法的实验结果如表 3-4 所示。

整体上讲，基于共现关系的方法表现最好，其 F-measure 在所有方法中表

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

表 2-7 变更影响实验结果 - F-measure/召回率/精确度

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

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

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

2. 针对于 RQ2 的实验

RQ1 中从整体的角度上说明了三种方法的有效性。为了回答 RQ2 中三种方法的优势，这里对每种方法检测得到的依赖型和逻辑型的变更影响关系分别



图 2-7 依赖闭包方法迭代路径

进行计算，得到如表的结果。通过对两种类型分别的统计，我们能更直观地发现不同方法的优势和特点。

表 2-8 变更影响实验结果 - F-measure/召回率/精确度

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

基于依赖关系闭包方法只能检测依赖型影响关系。其依赖型召回率高达 96.2%，能查全大部分的依赖型影响关系，但准确率不高，导致整体的表现不好。

基于克隆代码的方法仅能检测逻辑型的变更影响关系，但是其准确率能达到 95%。结合 RQ1 中的分析结果，这表明基于克隆代码方法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。图 3-6 为克隆代码方法挖掘到的一对有变更影响关系的方法。这里展示了这对方法的部分代码，其中绿色高亮的部分表示代码克隆的区域。这两个方法的主要功能是分别对 8 位和 4 位压缩格式的图像进行解码。我们发现，这对方法中的大部分逻辑结构几乎完全相同，只有少部分关键处理逻辑存在差异。由此，我们可以认定，这两个方法的变化过程很可能是同步的，即在实际的维护过程中，当对其中一个方法进行修改时，另一个方法也通常需要同步进行相应的变更，才能保证逻辑的一致性。

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

克隆代码

图 2-8 包含克隆代码片段的一组方法实例

这种代码克隆现象表明，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，因此在代码维护时，特别是针对这类高度相似的方法，必须考虑它们之间的相互依赖关系。这种潜在的同步更新要求在进行变更影响分析时，尤其是在代码质量评估和变更影响分析的研究中，必须给予充分的关注，以确保系统的稳定性和一致性。

基于数据挖掘的方法可以挖掘两种类型的变更影响关系，根据报告比例，可以发现其报告依赖型更多，这是由于对于质量良好的项目来讲，通常项目中的依赖型变更影响关系更多，能够通过编译器或开发工具即可让开发者掌握项目架构。除此之外该方法对于逻辑型的关系更为擅长，表现在挖掘到的逻辑型关系中，准确率能达到 91.2%，而对于依赖型的变更影响，则同样存在误报的情况，根据方法的设计思路，推测这里是由于在项目早期，开发者人数较少，项目发展较快，变更较频繁，所以通常都是大量代码一起提交并且频繁更改造造成的偶然现象，未来或许可通过仅挖掘项目稳定后的变更历史来改善。

为了说明数据挖掘方法的优秀潜力，以 librdkafka 项目中挖掘到的一对方法为例，该项目是 Apache Kafka 的一个高性能 C/C++ 客户端库。图 3-7 展示了通过数据挖掘技术发现的一对存在变更影响关系的方法。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 2-9 逻辑上有变更影响关系的方法对示例-incr 和 decr

从功能上来看，这两个方法是一对典型的协作方法，其作用密切相关。具体来说，incr 方法不仅执行计数器加一操作，还在计数器从零变为一时，执行资源的初始化工作，确保所需资源在后续操作中可用。而 decr 方法则在计数器减为零时，执行资源的释放操作，以清理不再使用的资源，避免资源泄漏。从代码实现可以看出，这两个方法的操作逻辑具有明显的互补性，加一与减一、初始化与释放的功能关系呈现出“镜像”特性。

尽管它们在代码中并未直接相互调用，也未调用相同的函数，但由于它们在功能上承担了计数器的管理和资源的初始化与释放工作，其逻辑关联性非常强。因此，当其中一个方法的实现逻辑发生变化时，另一个方法通常需要进行相应的调整，以保持整体逻辑的一致性。这种方法对的变更影响关系属于典型的逻辑关联型变更影响关系，而非通过直接调用或共享资源显式连接的变更关系。这种逻辑上的关联性表明，数据挖掘方法能够有效捕获代码中隐性的、非显式的变更依赖。

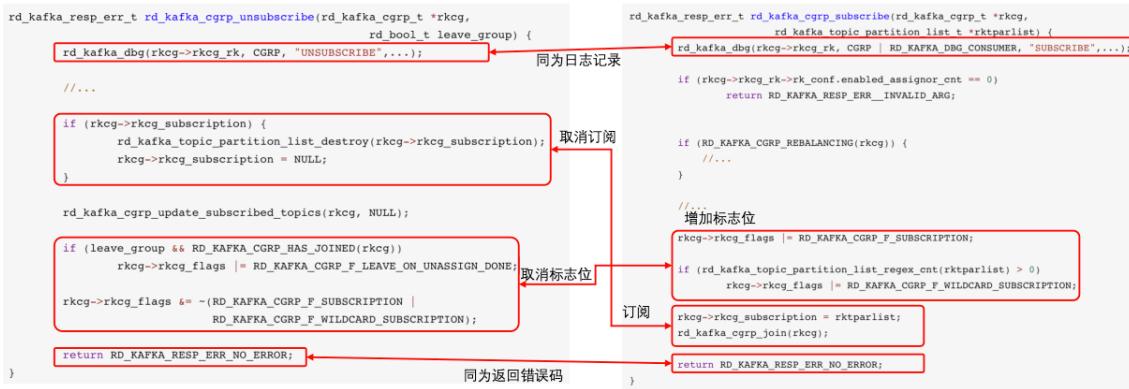


图 2-10 逻辑上有变更影响关系的方法对示例-subscribe 和 unsubscribe

图 3-8 所示是一个更加复杂的例子，这个例子同样是一对有逻辑关联型变更影响关系的方法。这对方法的功能分别是对 Kafka 的主题 (topic) 进行订阅和取消订阅。左侧的方法负责实现对指定主题的取消订阅操作，而右侧的方法则

负责实现订阅操作。尽管这两个方法在代码上并未直接相互调用，但它们在逻辑上具有显著的关联性，尤其是在处理订阅状态的管理上，在订阅操作中，方法会新增订阅标志位、更新订阅的主题列表。而在取消订阅操作中，方法会移除对应的订阅标志位、清空订阅列表，并在必要时清理与订阅相关的组资源。这种逻辑上的互补关系，使得订阅和取消订阅操作形成了一个完整的管理闭环。

从功能视角来看，订阅和取消订阅的行为本质上是对同一资源或逻辑状态的不同操作。这种镜像特性意味着，当订阅逻辑发生变化时，例如新增了订阅标志位的更新规则或修改了主题列表的处理方式，取消订阅逻辑通常也需要进行相应的调整，以保证订阅状态管理的完整性和一致性。

这两对方法体现了典型的逻辑关联型变更影响关系。这种关系不同于通过显式调用或共享资源产生的直接依赖，而是一种通过功能和逻辑流程相互关联的隐性依赖。这种实例表明，即使在方法间没有显式的代码关联，数据挖掘方法仍能够捕获这类隐性逻辑关联，为代码变更影响分析提供了有力支持。通过识别这类方法对，开发者在维护和演化代码时可以更好地理解潜在的影响范围，避免遗漏可能的关联性调整，提高系统维护的可靠性和效率。

表 2-9 变更影响实验结果 - F-measure/召回率/精确度

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

本文提出的三种方法均从逻辑型关系的检测上弥补了传统基于依赖传递闭包方法的不足，为代码变更影响关系的检测提供了多样化的解决方案。这些方法各有侧重，适用于不同的应用场景，同时也具有一定的局限性。总结如表 3-7。

表 2-10 变更影响分析方法对比总结

方法	检测关系	适用场景	缺点
克隆代码	逻辑型中的代码克隆	有无代码变更历史均可	只能检测代码克隆一种关系
数据挖掘	依赖型和逻辑型	有代码变更历史的项目	无法应用于没有变更历史的项目，如果不频繁变更可能无法被检测
深度学习	依赖型和逻辑型	无变更历史的项目	依赖于训练数据的质量

2.8 本章小结

本章实现了对代码中间表示的提取及质量评估度量的计算。首先，本文基于 libclang 工具对软件项目进行抽象语法树提取，并在此基础上进一步构建了方法摘要表和全局变量信息表。随后，基于提取得到的中间表示，分别介绍了基于内聚度缺乏度和基于连通性的内聚度分析，介绍了方法间耦合性分析以及扇入扇出的提取和分析，通过这几种分析手段提取了 8 个代码质量指标和 6 种耦合关系。本章通过一系列实验验证了度量指标的有效性，但是发现其只能聚焦于范围较小的质量问题，无法反映软件系统在架构上的缺陷，难以帮助开发者从软件代码的宏观角度上指导软件维护。

第3章 基于RAG的方法间变更影响分析

3.1 引言

3. 针对于RQ3的实验

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

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

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

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

表3-1 变更影响实验结果-接受率

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

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

关系中，开发者实际上更信任自己通过阅读代码结构得到的一到二层涟漪效应。

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

表 3-2 被用户拒绝实例代码信息

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

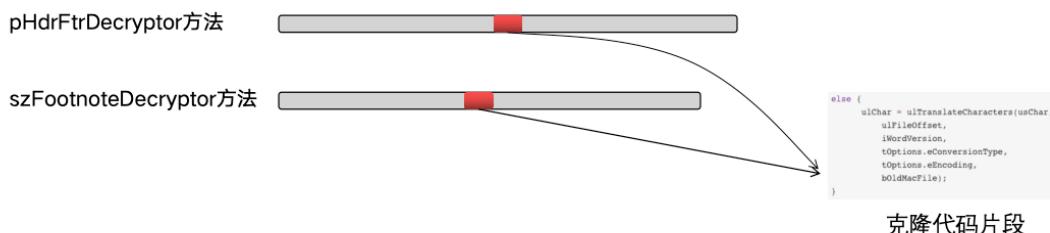


图 3-1 被用户拒绝实例代码长度可视化

数据挖掘方法的实验结果也高于依赖闭包方法，通过对比支持度不同的实验可以发现，支持度为 3 略高于支持度为 2 的接受率，这是由于，支持度为 3 意味着这对方法起码发生共同修改三次才能被挖掘到，这排除了共同修改 2 次的方法对的一些偶然情况，挖掘到的关系更少，但是也更为精准。因此，支持度也是决定数据挖掘方法精确度的重要因素之一。

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

进而提高代码质量。

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

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

其次对于

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

3.2 本章小结

本章介绍了四种变更影响分析方法。首先是传统的基于依赖关系闭包的变更影响分析方法。该方法首先对软件项目进行预处理，提取抽象语法树、方法摘要表和全局变量信息表等数据结构，然后构建依赖关系图，节点表示方法和全局变量，边表示方法之间的调用关系。然后根据依赖闭包对每个方法提取对应影响方法。其次是基于克隆代码的变更影响分析方法。该方法认为代码克隆关系一定程度上反映了变更影响关系，通过 ClaSP 算法进行序列挖掘，提取代码克隆，从而反映变更影响关系。第三种是基于数据挖掘的变更影响分析方法。该方法认为代码变更历史蕴含了一定的变更影响关系，通过提取代码的提交历史，根据频繁模式挖掘理论，挖掘出频繁共现的方法对，从而反映变更影响关系。第四种是基于深度学习的变更影响分析方法。对不存在变更历史的项目，通过数据挖掘得到的数据整理成数据集，训练深度学习模型，对项目方法之间的变更影响关系进行预测。最后，本章通过实验验证了四种方法在提取变更影响关系上的有效性，实验结果表明，除传统的基于依赖关系闭包的方法外，其他方法都取得了良好的检测效果。除此之外还通过实例分析，说明了每种方法的缺点和各自的侧重点。

第4章 基于代码审查图的代码架构和质量信息可视化

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^[47]。审查者不仅需要对变更的代码本身进行分析，还需要理解这些代码所处的上下文，才能做出正确的评估。因此，无论是作为开发者还是审查者，理解软件项目的结构和代码是至关重要的。只有深入掌握软件的整体架构和各模块之间的关系，才能在后续的开发和审查过程中保证代码质量。然而，传统的代码阅读和理解方式不仅需要消耗大量的时间和精力，还难以确保高效性和准确性，尤其是在面对庞大复杂的代码库时。

为了提高代码理解的效率并减少人为错误，本文提出了一种基于代码审查图的代码质量分析展示方式。这一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，并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，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 基于代码中间表示的代码质量度量提取

代码内聚度和代码耦合性是衡量软件设计质量的两个核心指标，它们直接反映了代码模块化质量。代码内聚度指的是模块（如文件、类或组件）内部元素之间的相关性。高内聚度意味着模块内的所有元素都紧密地围绕着一个单一的、明确的功能，代码更容易理解和维护^[48]。代码的耦合性则描述了模块之间的相互依赖。低耦合度意味着模块之间的依赖关系较小，每个模块都可以独立地执行其功能，而不需要过多地依赖其他模块。低耦合度的代码更容易测试和维护^[49]。除此之外我们还考虑了代码的复杂性和代码缺陷，对

基于方法摘要和全局变量信息表，我们计算如下代码度量，用于分析代码质量。

4.2.1 基于内聚度缺乏度的内聚性分析

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^[50]。计算公式如式(2-1)。

$$LCOM1 = C_n^2 - e \quad (4-1)$$

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

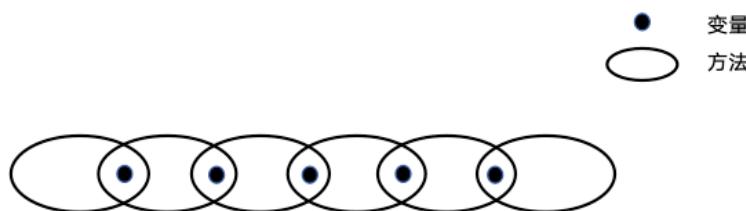


图 4-1 示例模块

(2) LCOM2, 含义是不引用相同字段方法对与引用相同字段方法对数之差^[51]。其计算公式如式 (2-2)。

$$LCOM2 = \begin{cases} P - Q, & \text{if } P \geq Q \\ 0, & \text{otherwise} \end{cases} \quad (4-2)$$

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):

$$LCOM3 = \frac{\left(\frac{1}{a} \sum_{j=1}^a \mu(A_j)\right) - m}{1 - m} \quad (4-3)$$

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。

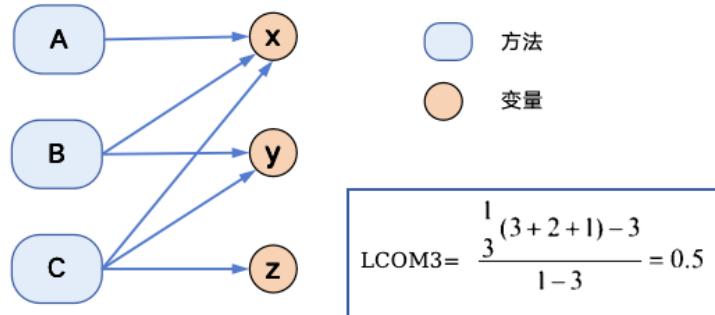


图 4-2 LCOM3 计算示例

(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^[52]。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。

4.2.2 基于连通性的内聚性分析

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

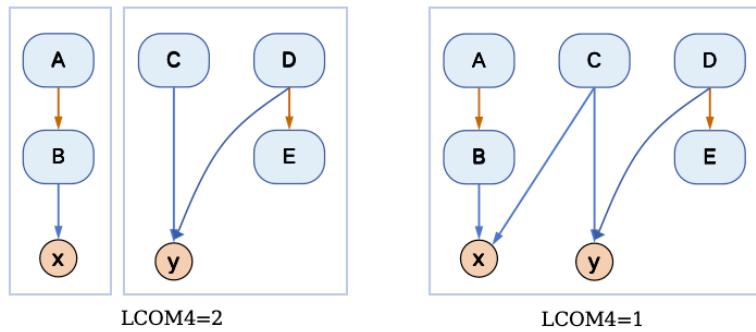


图 4-3 LCOM4 计算示例

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^[53]。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)。

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

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^[53]。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)：

$$LCC = \frac{e + e_{indirect}}{C_n^2} \quad (4-5)$$

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 方法间耦合性分析

耦合是在软件架构中用来描述模块间相互依赖和连接程度的一个重要指标。耦合度的高低直接影响到系统的维护性和可扩展性。在现有的研究和实践中，耦合度通常被细分为六个等级，如表 2-1 所示，这些等级从高到低反映了

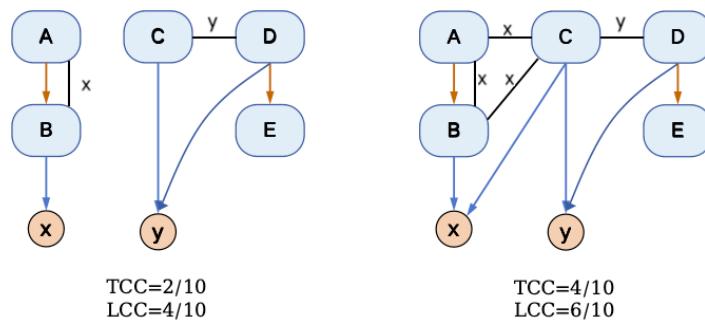


图 4-4 TCC 和 LCC 计算示例

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，方法间的耦合性反映了不同方法之间的依赖关系，它直接影响代码的可读性、可测试性以及后续的维护和扩展。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	6	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 方法扇入扇出度量分析

方法的扇入（Fan-in）和扇出（Fan-out）是软件工程中用于衡量方法复杂性和模块间依赖关系的两个指标。

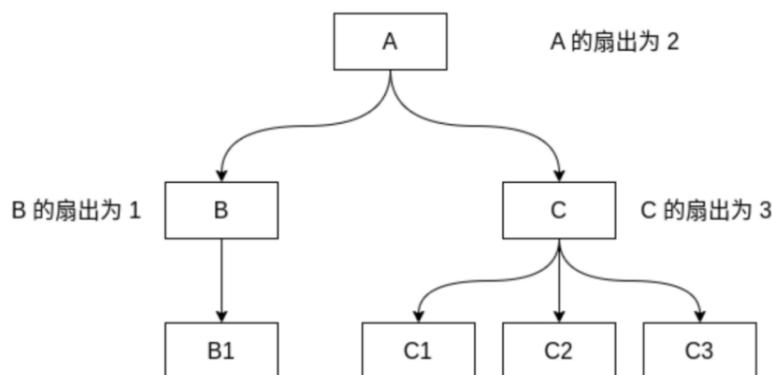


图 4-5 扇入扇出实例

扇入是指调用某个方法的不同方法的数量。它表示了一个方法对其他方法的依赖程度。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。

扇出是指某个方法直接调用的不同方法的数量。它表示了一个方法对其他方法的影响程度。扇出值较高的方法可能更复杂，因为它们需要管理和协调更多的方法调用。高扇出值可能意味着该方法具有较高的责任度，且可能更难以理解和维护。

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缺陷检测

为了更准确地衡量代码质量，本文结合了静态代码分析工具 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++ 代码中的潜在错误和编码规范问题。它通过静态分析技术，扫描源代码，识别可能存在的内存泄漏、空指针解引用、未初始化变量等常见问题，并提供详细的诊断报告。与传统的编译器警告不同，Cppcheck 不依赖于程序的编译过程，它直接分析源代码的结构和逻辑，从而能够发现更广泛的潜在问题，尤其是那些难以通过常规测试手段捕捉到的错误。

在本研究中，Cppcheck 的检测结果被视为代码质量评估的重要组成部分。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
warning	潜在的错误或不推荐的做法
style	代码风格问题，通常与代码格式、命名约定等相关
performance	性能问题，表明当前方式可能效率不高
portability	平台相关问题，可能导致在不同环境出现不同行为
information	额外的信息或建议

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并及时修复，进而提高代码的可维护性和健壮性。

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。

4.3 代码审查图

4.3.1 代码审查图构建

代码审查图主要由两个核心元素构成，即节点和边。其中，节点代表软件项目中的方法或全局变量，边则表示节点之间的各种关系，如耦合关系、变更

影响关系以及依赖调用关系。这些元素的结合能够帮助开发者从全局视角理解和评估项目的结构和质量，尤其是在进行代码审查和变更分析时。

(1) 节点属性

在代码审查图中，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
模型预测方法模块	模型预测的方法应在的模块

这一部分包括方法的名称、所在模块、方法签名、访问修饰符等基本信息，这些信息有助于开发者快速识别和定位方法的功能和作用。例如，方法的名称可以反映其业务功能，所在模块和调用信息则有助于理解方法的上下文和调用约束。

其次是与代码质量相关的度量和信息，如表 4-2 所示。

表 4-4 代码审查图节点属性-质量相关信息

属性类别	属性	描述
扇入扇出信息	扇入	方法的扇入值以及扇入值在项目中的排名比例
	扇出	方法的扇出值以及扇出值在项目中的排名比例
内聚度信息	LCOM1	所在模块的 LCOM1 值以及相应建议
	LCOM2	所在模块的 LCOM2 值以及相应建议
	LCOM3	所在模块的 LCOM3 值以及相应建议
	LCOM4	所在模块的 LCOM4 值以及相应建议
	TCC	所在模块的 TCC 值以及相应建议
	LCC	所在模块的 LCC 值以及相应建议
耦合关系	数据耦合	与本方法存在数据耦合关系的方法
	标记耦合	与本方法存在标记耦合关系的方法
	外部耦合	与本方法存在外部耦合关系的方法
	公共耦合	与本方法存在公共耦合关系的方法
变更影响关系	变更影响关系	与本方法存在变更影响的方法，表明来源为代码克隆、变更历史、模型预测
缺陷	静态代码缺陷	由 cppcheck 检测得到的本方法缺陷

这一部分基于前文方法提取和统计结果，涵盖了一些与代码质量直接相关的指标，如方法的扇入扇出、内聚度、耦合性等。这些度量和指标的结果将结合项目的具体统计数据或检测结果，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位居项目中的前 5%，则可能表明该方法在项目中的依赖关系过于复杂，可能导致高耦合性，进而影响系统的灵活性和可维护性。类似地，如果检测出某方法存在不良耦合，则可能需要开发者重新设计该方法与其他模块的接口，以减少不必要的依赖。

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

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，变更影响关系则根据检测方法的不同，设定为 3 个不同的来源。需要注意的是，由于依赖闭包方法本身是基于依赖图生成的，因此在代码审查图中，这类关系不再单独指出。

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	含代码克隆、变更历史、模型预测三种

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 4-6 **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，这样的级别对于一个图来讲，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。

4.3.3 代码审查报告生成

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

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

在质量信息过于分散、不易聚焦的问题。因此，本研究进一步提出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为主线，系统性地向用户报告代码中的潜在问题。具体来说，报告主要涵盖以下几个方面的内容：

- 内聚度信息：报告中将重点标记内聚度最差的 5% 模块，并提供相应的统计信息。这些模块通常在逻辑结构上松散、职责分散，可能是代码设计需要改进的关键部分。开发者可通过这些信息快速识别项目中存在高维护风险的模块。
- 不良的耦合信息：包括同模块内的公共耦合、不同模块的公共耦合以及外部耦合。这些耦合关系可能导致模块之间的高依赖性和低灵活性，影响系统可维护性。
- 复杂度信息：报告将筛选出扇入扇出指标最差的 5% 模块，提供详细数据说明。这些模块往往由于过多的依赖关系或调用关系而难以维护，是优化的重点对象。
- 代码缺陷与规范信息：cppcheck 检测出的代码缺陷和不符合规范的代码信息，并给出相应的建议。
- 不良变更影响：通过分析代码中的变更影响关系，报告将重点指出项目中存在变更影响关系数量最多的前 5% 方法。这些方法通常具有较高的变更复杂度，是项目维护中的风险点。
- 基于大模型的模块预测结果：报告列出那些标签不属于原始模块的方法名称。此类方法可能存在职责划分不当或模块归属不合理的情况，报告将此信息提供给用户，供其判断是否需要重新划分模块归属，从而优化模块结构。

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，在变更时全面评估影响范围，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。

4.4 实验结果与分析

4.4.1 实验环境与评价方法

本章依旧使用第二章中的项目为示例项目。

(1) 实验环境

实验环境如表 4-6 所示。

表 4-8 实验环境

环境	信息
操作系统	macOS Ventura v13.5.2
IntelliJ pycharm	2021.1.1
python	3.7
Java	1.8
G6	g6.min.js 4.3.11
libclang	15.0.7
pycparser	2.21

(2) 评价方法

对于代码审查图生成方法，首先对示例项目生成代码审查图，验证可视化方案的需求，根据代码审查图分析前文中实验结果中的实例研究，以说明代码审查图相较文字的优越之处。

对于代码审查报告的生成，则是验证审查报告的格式是否符合设计方案。

4.4.2 实验结果分析

将实验项目的 `github` 仓库克隆到本地，对每个项目进行按前文所述步骤提取代码中间表示，包括抽象语法树、方法摘要表和全局变量信息表，再进一步按 2.5 节所述方法，提取代码质量度量和缺陷信息。在提取到质量度量后，筛选质量较差的代码分析结果报告给用户，收集用户的反馈，衡量方法的有效性。报告的标准如下：

(1) 内聚度

内聚度是衡量一个模块内部各个组件（本文中是方法和全局变量）之间关系的紧密程度的指标。内聚度越高，则意味着模块内的各个部分越紧密，职责明确且功能聚焦。然而，仅凭数值的计算，用户可能难以直观理解模块的内聚度高低。因此本文通过对同一项目中各模块内聚度进行排名，向用户报告内聚度最差的前 5% 模块，以帮助用户识别和定位项目中内聚度较差的模块。随着用户不断进行改进，模块的内聚度将逐步提升并收敛，最终整个项目达到平衡的状态。

（2）耦合性

本文中提取了四种不同的耦合类别，每种类别的耦合程度并不相同，数据耦合、标记耦合、外部耦合、公共耦合，耦合程度依次递增。[\[54\]](#) 中提到数据耦合是最佳的耦合类型，而外部耦合和公共耦合则对系统的质量和维护性有负面影响。这是由于数据耦合和标记耦合都通过参数传递数据，方法的输入输出明确，易于测试和维护，而外部耦合和公共耦合共享公共数据，可能会影响程序的可测试性和问题的排查诊断。

本文进一步将代码是否处于同一模块考虑在内，因为对于同一模块内，代码往往有共同的上下文和职责范围，因此可适当对耦合标准放宽。因此对与在同一模块内的方法，报告公共耦合，对于不在同一模块内的方法，报告外部耦合和公共耦合，将这两种情况作为不良耦合报告给用户。

（3）扇入扇出

传统标准中要求尽量高扇入低扇出，高扇入表示其他模块依赖该模块，模块的功能可能是系统中其他模块的核心或基础，复用性较好；低扇出表示该模块没有过多地依赖其他模块，因此它具有更高的独立性，更易维护。同内聚度类似，仅计算数值或统一设置阈值的方式，用户难以直观理解，因此这里也对同一项目中扇入扇出值进行排序，向用户报告最差的前 5% 的方法。

（4）静态分析工具

本文结合了静态分析工具 Cppcheck 对项目代码进行检测。考虑到 Cppcheck 的严重等级分类能从错误、警告、风格、性能、未定义行为等多个角度提供细致分析，每个类别都关注特定的代码问题，不仅帮助开发者准确识别出可能导致崩溃或异常行为的缺陷，还能够进行规范检查，发现影响程序效率、可读性、资源管理等方面的潜在问题。因此本文将向用户报告表 2-2 中六种类型的问题，作为质量评估的一部分，提供给用户进行判别。

总的来讲，这四种度量向用户报告的标准如表 2-4 所示。

为了验证质量度量的可靠性与实用性，本文邀请了四位经验丰富的软件开

表 4-9 各度量报告标准

度量	报告标准
内聚度	6 项指标分别报告内聚度最差的前 5% 模块
耦合性	同模块内报告公共耦合 不同模块报告公共耦合、外部耦合
扇入扇出	分别报告最差的前 5% 方法
静态工具检测	报告 6 种类型问题

发者，对报告的每一项的准确性和可接受性进行评估。评估的标准包括以下几个方面：

- 内聚度差、耦合不良、扇入过低、扇出过高和缺陷的检测结果是否能为用户所接受：评估这些指标的报告内容是否准确反映了代码中的问题，且是否能够为开发者提供有效的指导。
- 指标的解释与建议是否有助于代码优化：不仅是检测结果本身，还需要评估报告中对检测结果的解释是否充分。
- 报告结果与实际代码质量的关联性：通过比较报告中的分析结果与开发者实际编码过程中所遇到的问题，验证报告的实用性。

对于报告的每一项，如果用户认为符合标准，则认为用户接受对应的结果，则将对应的项反馈结果标记为 1，统计用户的接受率，接受率指标计算公式如式（2-6）

$$AR_n = \frac{\frac{1}{N} \sum_{i=1}^N A_i}{T} \times 100\% \quad (4-6)$$

其中 AR 表示接受率，A 表示开发者接受的项数，T 表示报告的总项数，N 表示软件开发者数。

1. 内聚度实验结果及实例分析

(1) 内聚度实验结果

表 2-5 展示了对于内聚度报告项用户的接受率，从实验结果可以看出，该方法表现良好，可靠性较高。具体来讲，指标基本都在 70% 以上。通过进一步对比，发现在 TheAlgorithms、antiword 等比较简单 的项目中，内聚度在 6 项指标中常出现相同的接受率，经过分析发现这是因为在同一项目中，尽管 6 项指标的计算方式和侧重点都不同，但是最差的前 5% 基本上是重合的，这意味着一个模块如果在 LCOM1 中表现的很差，在其他指标中也通常表现得很差，这符合通常认知，同时也侧面验证了指标的准确性。但是对于 FFmpegKit 等较复杂的项目，6 项指标则不太相同，体现了不同内聚度计算方式的侧重点。

(2) 内聚度实例分析

表 4-10 内聚度接受率

项目名称	LCOM1	LCOM2	LCOM3	LCOM4	TCC	LCC
TheAlgorithms	89.2	74.9	74.9	89.2	74.9	74.9
antiword-0.37	91.3	84.2	84.2	91.3	84.2	84.2
jemalloc-5.3.0	82.2	82.2	82.2	82.2	88.6	88.6
libbpf-1.1	87.3	83.4	87.3	87.3	93.7	93.7
librdkafka-2.1.0	73.7	73.7	73.7	73.7	78.4	78.4
FFmpegKit-5.1.0	71.3	69.2	74.9	76.6	89.3	84.9

对内聚度实验结果进行进一步分析，这里选取 antiword 项目中的内聚度表现最差的模块 misc.c 文件模块进行分析。为了更好地理解这一现象，进一步检查了该模块的源代码，并对其结构进行了统计分析。具体来看，misc.c 文件中一共定义了 1 个全局变量和 15 个方法，其中 11 个方法并未被同一模块内的其他方法调用，只有 3 个方法之间存在相互调用关系，而仅有 2 个方法使用了该模块定义的全局变量。

通过这个结果可以发现，尽管该模块包含了一定数量的方法和变量，但模块内各个方法和变量之间的依赖关系极为薄弱。大部分方法相互之间没有调用关系，且仅有少数方法与全局变量发生交互。这样的结构表明，模块内的功能划分较为松散，各个功能单元之间缺乏必要的协作和紧密联系。因此，该模块的内聚度较差，功能的集中性和一致性较低，导致其内聚度值显著较低。因此，通过对 misc.c 文件内聚度的分析，我们不仅可以得出该模块的内聚度较差的结论，还能为未来的重构和优化提供指导意见。例如，增强模块内部方法之间的调用关系，优化变量的使用方式，以提高模块的内聚度，从而提升系统的整体质量和可维护性。

2. 其他度量实验结果及实例分析

表 2-6 展示了对于其他度量报告项用户的接受率，从实验结果可以看出，不同项目在各项度量指标上的接受率存在一定差异。大部分项目的耦合性接受率较高，而扇入扇出的接受率普遍较低，大部分的项目上只达到了 40-50% 的水平，而所有项目对静态工具检测出来的缺陷上的接受率都较高，说明静态工具的可靠性较好。接下来对所有指标的实验结果进行进一步分析。

(1) 耦合性结果分析和实例分析

耦合性的接受率在几个项目中的表现不尽相同。对于结构较为简单的项目，如 antiword 和 jemalloc，接受率均能达到较高的水平，在 TheAlgorithms 项目上甚至能达到 99% 以上的接受率。而对于更大型更复杂的项目，接受率则略微降低。

表 4-11 其他度量接受率实验结果

项目名称	耦合性接受率	扇入扇出接受率	静态工具缺陷接受率
TheAlgorithms	99.2	87.6	100.0
antiword-0.37	84.2	56.4	98.0
jemalloc-5.3.0	77.8	66.3	100.0
libbpf-1.1	68.3	42.1	100.0
librdkafka-2.1.0	63.7	48.7	98.8
FFmpegKit-5.1.0	54.0	43.3	94.6

进一步分析如此表现的原因，对于 **TheAlgorithms**，这个项目是对各种算法的开源实现，模块和模块之间几乎不存在相互依赖和调用的情况，每种算法独立开发，实际上是库函数，仅在需要时供开发者调用。总的来讲，简单项目结构较为简单，各个模块功能明确，职责聚焦，所以耦合性不良的样例很少，共检测出 15 例。维护起来也更加容易，用户更容易接受。

而在复杂项目中，如 **FFmpegKit**，耦合性接受率相较其他项目低一些。这里以 **FFmpegKit** 项目中不被用户所接受的样例进行分析。在这个样例中，方法 `probe_file` 与另外三个模块的方法发生了公共耦合，具体来讲这几个方法共同使用了 `fftools_ffprobe.c` 文件中定义的全局变量 `nb_streams`，这是一个整数。而不被用户所接受的理由是，这个全局变量实际上是一个共享全局状态，它代表了流的数量，需要在不同平台上保持一致，如表中所示，需要在 `android`、`linux`、`apple` 的环境中都保持一致，才能保证程序的正确性和一致性。在这种情况下，多个方法访问并操作该全局变量是合理的，因为这种做法确保了平台间的一致性。

表 4-12 FFmpegKit 项目中不被用户接受的不良耦合实例

共同访问变量	方法名
<code>nb_streams</code>	<code>ffmpeg-kit/apple/src/fftools_ffprobe.c@probe_file</code>
	<code>ffmpeg-kit/android/ffmpeg-kit-android-lib/cpp/fftools_thread_queue.c@tq_alloc</code>
	<code>ffmpeg-kit/linux/src/fftools_thread_queue.c@tq_alloc</code>
	<code>ffmpeg-kit/apple/src/fftools_thread_queue.c@tq_alloc</code>

通过对这个样例的分析可以看出，对与大型、复杂的软件项目来讲，各模块和方法之间的依赖关系可能已经非常复杂，需要大量时间来理解现有结构并分析方法间的相互影响，对于某些功能，也的确缺乏耦合性更低的实现方式，所以接受率更低。

(2) 扇入扇出接受率分析

对于扇入和扇出值指标，观察到该指标的接受率较低。通过实例分析发现，大多数项目的开发模式并不完全符合高扇入、低扇出的设计原则。

特别是在低扇入的报告项中，接受率尤为偏低。实际分析表明，许多方法仅被单次调用，虽然这些方法的复用性较差，但它们更为独立，且较少依赖于其他模块的实现。这种独立性使得这些方法更容易进行单独测试、维护和扩展。考虑到软件的长期发展，这些低扇入的方法未来仍有可能被其他模块调用。因此，低扇入方法在某些情况下是有其优势的，用户难以直接接受此类报告项，因此拉低了整体的接受率。

而对于高扇出的报告项，大部分是可以接受的，但是也存在某些高扇出的方法是合理的，比如有些高扇出方法实际上 是中心化的模块。

```

void
vGetPropertyInfo(FILE *pFile,...)
{
    switch (iWordVersion) {
        case 0:
            //6处调用
        case 1:
            //...
        case 2:
            //8处调用
        case 4:
        case 5:
        case 6:
        case 7:
            //6处调用
        case 8:
            //7处调用
    default:
        //3处调用
    }
}

```

图 4-7 vGetPropertyInfo 方法结构

这里以 antiword 项目中 properties.c 文件的 vGetPropertyInfo 方法为例，该方法主要根据传入的 Word 版本，从文档中提取各种属性信息。该方法的扇出度为 30，是项目中扇出度较大的一个，方法可能复杂度过高，因此被报告属于不良的扇出，但这一报告并未得到用户的接受。其原因在于，该方法的核心结构是一个包含多达 9 种不同情况的 switch 语句，每种情况又细分为多个子情况，因此导致总共产生了 30 个方法调用。该方法的合理性在于通过清单式方式集中处理所有功能逻辑，使得用户能够一目了然。此设计不仅减少了冗余代码，避免了方法调用链过长带来的可读性问题，还降低了因方法调用所带来的额外开销，从而优化了整体系统的效率。因此，尽管该方法具有较高的扇出度，其设计思路仍然是有效的，符合高效代码的要求。

(3) 静态工具检测接受率分析

静态工具检验得到的代码缺陷的接受率是最高的，经过分析发现只有个别错误，如平台相关问题，与特定编译器相关的警告等不容易被开发者所接受，因为这并不是代码本身的质量问题。

经过分析可以发现，这些度量指标有一定的有效性，但是这些度量聚焦的往往是范围较小的质量问题。如模块内、方法内甚至语句级的质量问题。面对更大规模的软件代码时，无法反映其在架构上的缺陷，难以帮助开发者从宏观的角度上指导代码维护。

2. 代码审查图

(1) 代码审查图概览

图 4-4 展示了 Antiword 项目和 TheAlgorithms 项目的代码审查图，直观地反映了两者的结构特征和模块化差异。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。上方的图是未区分模块的全局视图，所有的节点和边混杂在一起，仅从整体上体现了项目的结构复杂度。而下方的图对模块进行了区分，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。

从图中可以明显看出两个项目在结构特征上的显著差异。Antiword 项目整体模块之间高度协作，共同实现一个完整的功能，因此模块之间的联系较为紧密，表现出较强的耦合性。从可视化图上观察，按模块划分后，可以清晰地看到具有相同功能的模块形成了较为紧密的聚集。同一颜色的节点集中分布，进一步体现了模块的内聚性较高以及逻辑结构的清晰性。

相比之下，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。

此外，用户可以通过点击图中的某个节点来查看该节点的详细信息，包括方法或全局变量的具体描述。这一功能能够帮助用户快速定位关键信息，深入了解特定方法或变量的功能和用途，从而更高效地进行代码审查和理解。

(2) 子图展示

代码审查图蕴含了丰富且有价值的信息，通过聚焦于代码审查图中的子图，开发者和审查者可以快速直观地理解代码结构之间的关系。子图清晰地展示了

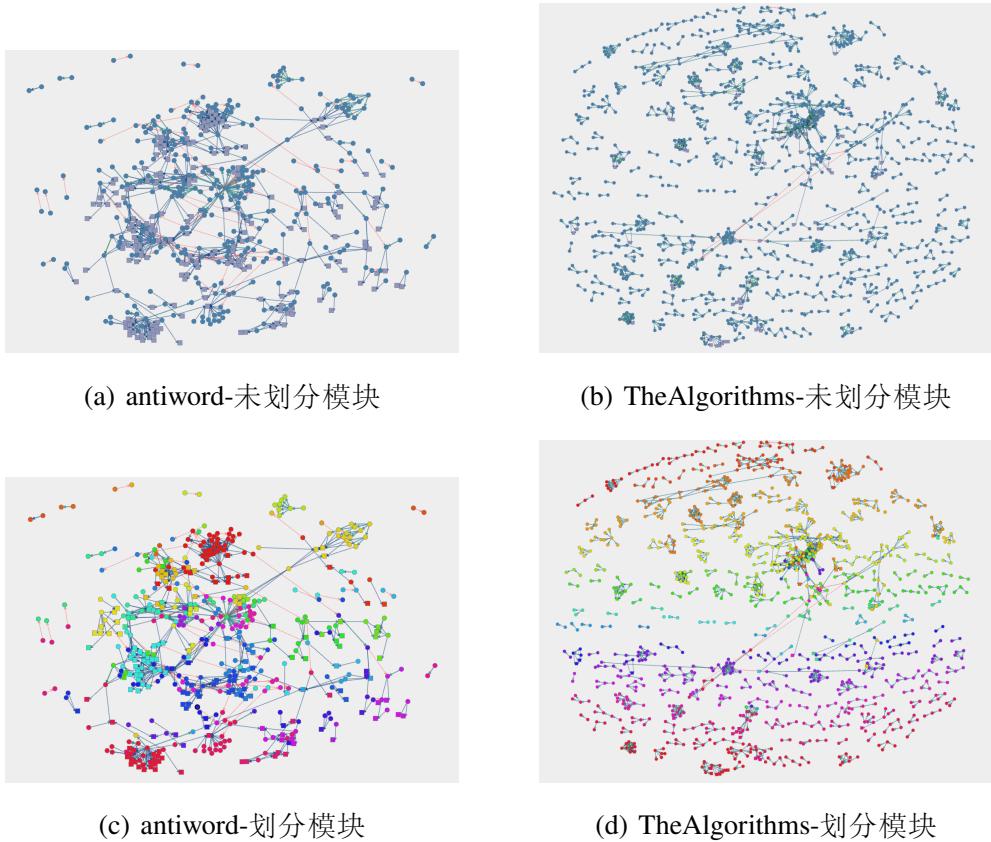


图 4-8 代码审查图

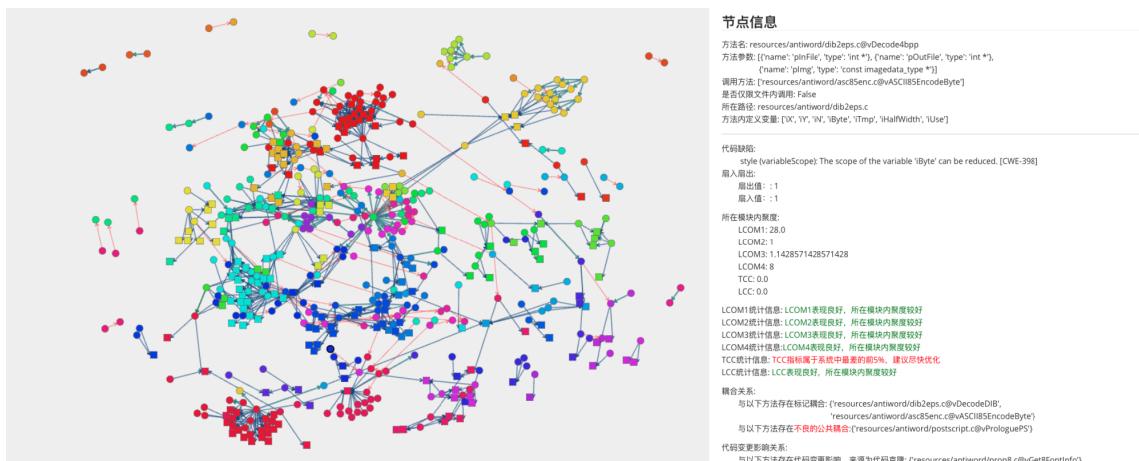


图 4-9 点击节点展开节点信息

代码结构之间的依赖、耦合以及变更影响关系，使得复杂的代码逻辑更加可视化。

以 antiword 项目中的子图为为例，本文聚焦于 `vDecodeDIB` 方法的代码开发和审查场景。若按照传统方法对该方法进行开发或分析，仅通过人工阅读代码，开发者需要深入解析与其相关的依赖关系和变更影响关系，这可能涉及 300 多行代码的逻辑才能全面掌握方法的上下文。然而，通过代码审查图，开发者只

需关注 8 个节点和 9 条边，即可快速获取关键信息。这种直观的可视化显著降低了代码理解的复杂度和成本。

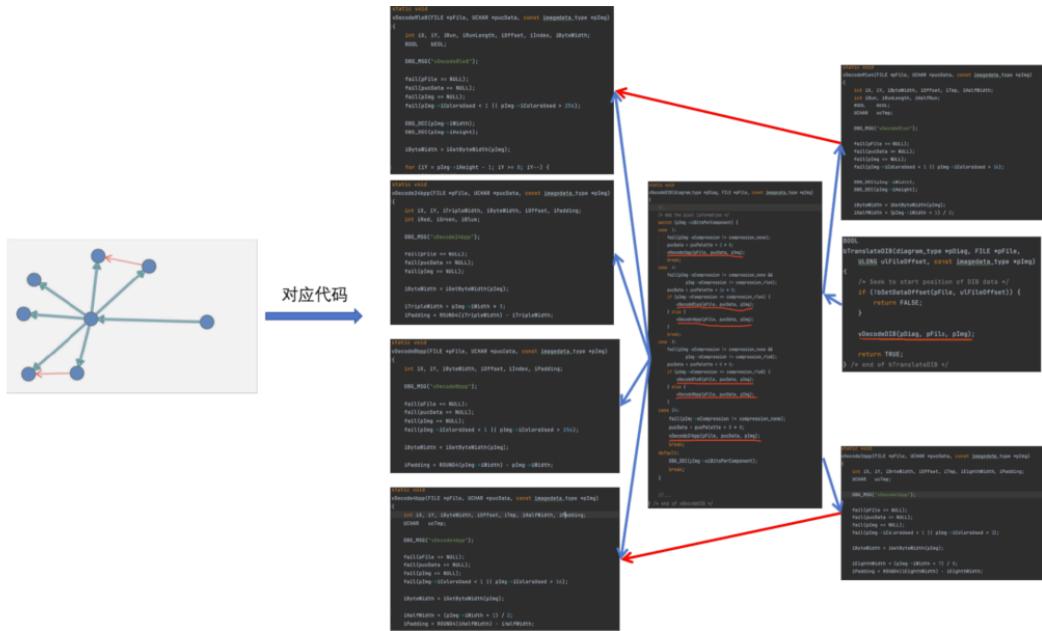


图 4-10 vDecodeDIB 方法上下文

尤其值得注意的是，变更影响关系中包含了两个基于克隆代码的关系。这类关系通常隐匿于代码中，开发者仅靠手动查看代码很难准确发现并记录。代码审查图将这些隐藏的克隆关系显式标出，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。

对开发者而言，代码审查图能够提供更加直观的视角，帮助其快速掌握代码之间的各种依赖关系和耦合情况。在代码开发过程中，审查图不仅有助于理清模块间的关系，还能有效避免变更不完全的问题。而对于代码审查者，审查图能够快速展示代码的上下文结构，简化复杂逻辑的理解过程，同时高效地识别代码中的变更缺陷。

(3) 代码质量属性体现

通过代码审查图，不仅可以直观展示代码的依赖关系和变更影响关系，还能有效地体现代码质量的特征，从而辅助用户理解代码质量较差或较好的具体原因。在第 2.6.2 节中详细解释了 misc.c 文件内聚度表现最差的原因，并统计了该文件内部方法和变量的相关信息。然而，仅依靠文字说明可能难以全面传达问题的全貌，而配合代码审查图，如图 4-7 所示，可以更加直观地展示该模块的结构问题。

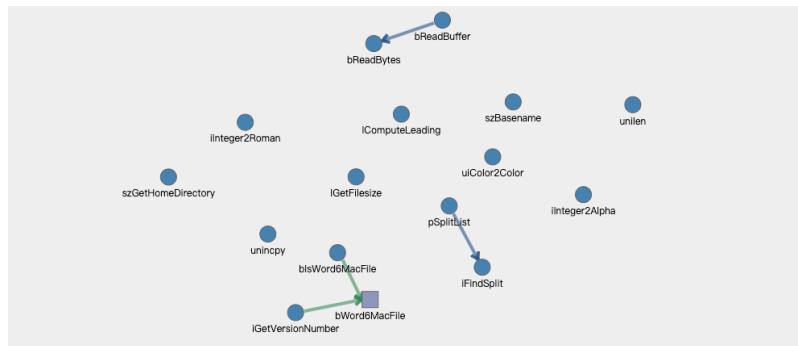
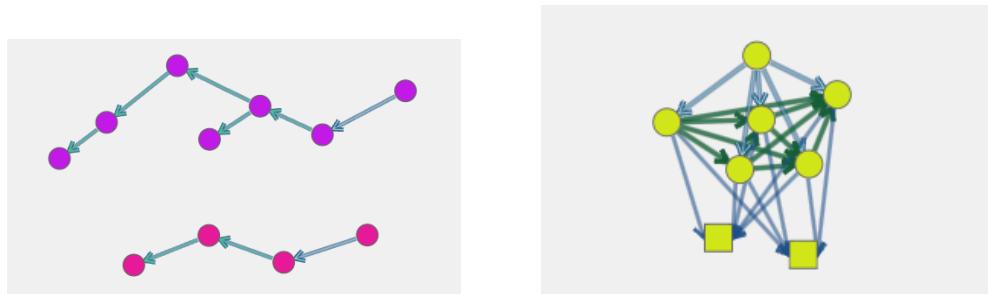


图 4-11 misc.c 文件对应的代码审查图

从图中可以清晰地看出，`misc.c` 文件的结构非常松散，其内部存在大量孤立节点，缺乏足够的联系和协调关系。这些孤立节点既无法与其他节点形成有意义的逻辑关系，也难以体现出模块内部的高内聚性特征。这种松散的结构正是导致其内聚度表现最差的主要原因。

除此之外还有一些内聚度差和好的模块在代码审查图中可以清晰地展现，如图 4-8 所示。图中，左侧展示的是内聚度较差的模块，而右侧则展示了内聚度较好的模块。单纯从代码本身出发，往往难以直观地识别出导致内聚度差或较好的具体原因。然而，通过代码审查图，问题的根源得以清晰呈现。具体来说，左侧内聚度差的模块之所以表现为低内聚度，主要是因为存在过多的链式调用，这种设计使得模块之间的依赖关系过于松散，缺乏足够的内在联系。而右侧内聚度良好的模块，则表现出较强的模块化特征，模块内部功能紧密相关，依赖关系清晰且有序，从而促进了代码的高内聚性。通过这种可视化的方式，代码审查图不仅帮助开发者迅速识别出模块内聚度的优劣，还能够为后续的重构与优化提供建议，如左侧内聚度差的模块，可通过合并方法，减少方法数和调用量，达到提高内聚度的效果。



(a) 内聚度差——存在链式调用

(b) 内聚度良好——关联紧密

图 4-12 内聚度良好和较差的例子

在 2.6.2 节中对高扇出方法的讨论中，高扇出通常意味着方法存在过高的复杂性，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

4-9 所示。图中的中心节点通过大量出边与其他模块或方法产生关联，这种结构不仅揭示了方法的复杂性，也反映了其可能导致的维护难度和潜在的设计缺陷。用户可以通过代码审查图迅速定位到具有高扇出的具体方法，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。当发现高扇出的原因主要源于过多的依赖关系时，开发者可以及时采取相应的优化措施，如重构该方法、简化其职责或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。

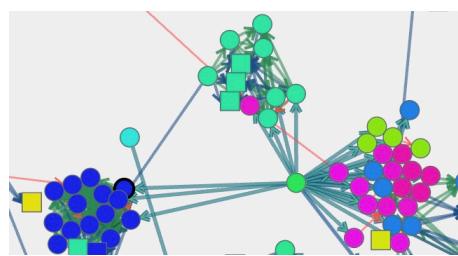


图 4-13 扇出度较高的节点

对于某些方法，其功能可能已经与当前所在模块的职责不再完全匹配，而与其他模块的结合更加紧密。这一问题在代码审查图中得以有效呈现。得益于力导向图的可视化特性，图中会显示出方法之间的依赖关系与耦合关系，紧密依赖的节点往往聚集在一起。由于同一模块内的节点通常会被赋予相同的颜色，若模块划分不合理，则会出现如图 4-10 所示的情况。在图中，聚集的节点颜色并不统一，某些节点（如图中的绿色和蓝色节点）与模块的主色调（橙色）不符。这表明，尽管这些方法本应属于其他模块，但它们与当前模块（即橙色模块）之间的耦合关系更为紧密。通过这种可视化，用户能够明确识别出不合理的模块划分，并采取相应的重构措施，例如将这些方法迁移至更合适的模块中，避免产生不良的耦合关系。这种调整不仅能够优化模块的功能划分，还能提高系统的可维护性与可扩展性，减少未来修改时的复杂度。

在代码审查图中，我们也能看到一些方法之间的不良变更影响关系。除了直接反映变更影响较多的方法外，代码审查图还能够揭示出由于变更影响分析所导致的模块间不良变更传播问题。如图 4-11。该子图涉及三个模块，每个模块的内聚度均较为良好，然而每个模块内都有一个方法与其他两个模块之间存在变更影响关系。这导致当这几个方法变动时，可能影响到不属于同一模块的代码也要跟着改变。更为严重的是，随着变更的涟漪效应，这些变更会进一步扩散，影响到更多模块，造成不必要的连锁反应。这种现象揭示了该系统架构存在潜在的问题——尽管各模块内部结构较为合理，但模块之间的依赖关系过于复杂且紧密，增加了维护和变更时的复杂度和风险。为解决这一问题，优化

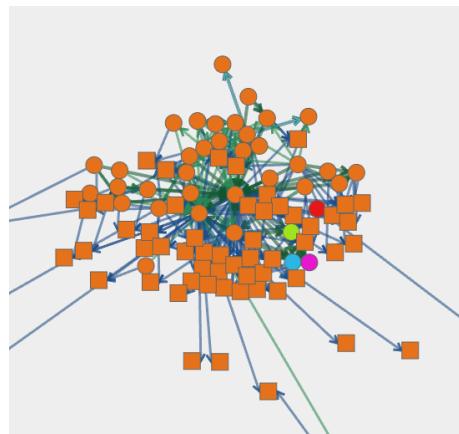


图 4-14 模块划分不合适的方法

的建议是尽量减少或消除这种不良的变更影响关系。具体而言，可以通过提取和复用模块间的共同代码逻辑，消除不必要的代码克隆，进而降低跨模块变更传播的可能性。

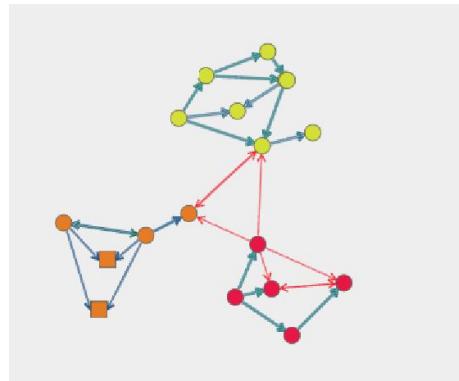


图 4-15 不良的变更影响关系

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

3. 代码质量评估报告

下图展示了一个实际的代码评估报告示例。由于报告内容较长，这里仅展示了其中的一部分信息。通过代码评估报告，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。

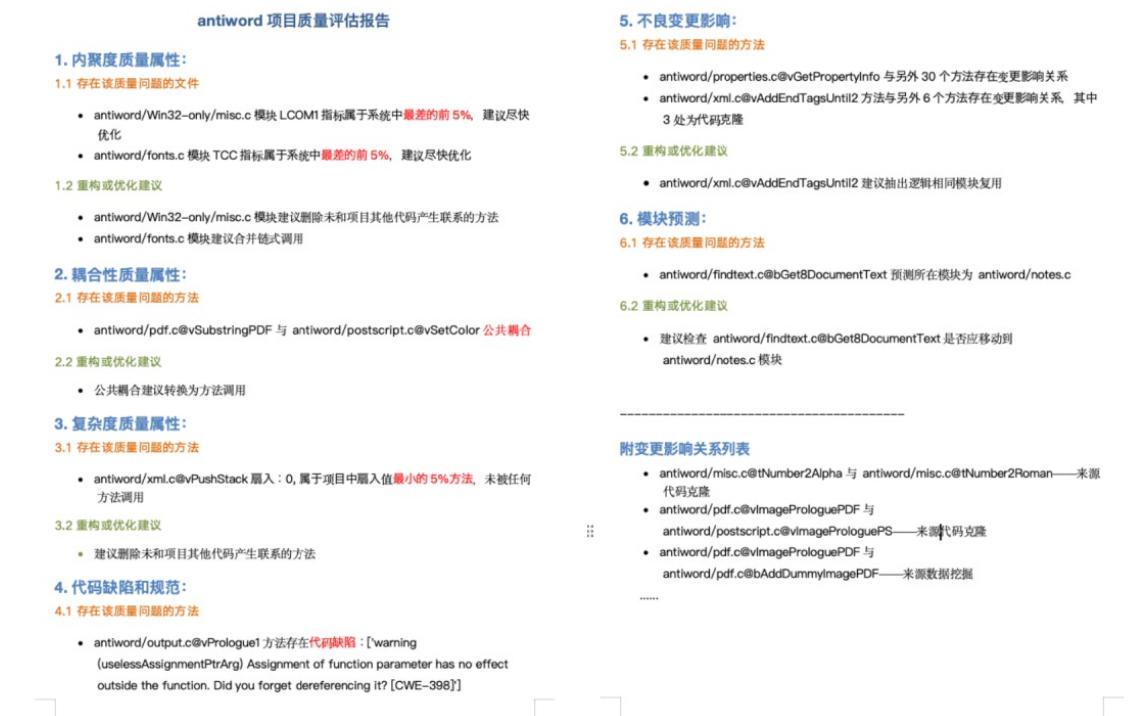


图 4-16 antiword 项目代码质量评估报告节选

4.5 代码审查图的实际应用

在实际应用中，代码审查图可以有以下三种用法。

辅助开发者的代码维护 当用户对软件代码进行开发时，可将复杂庞大的项目先通过系统进行分析，得到代码审查图。

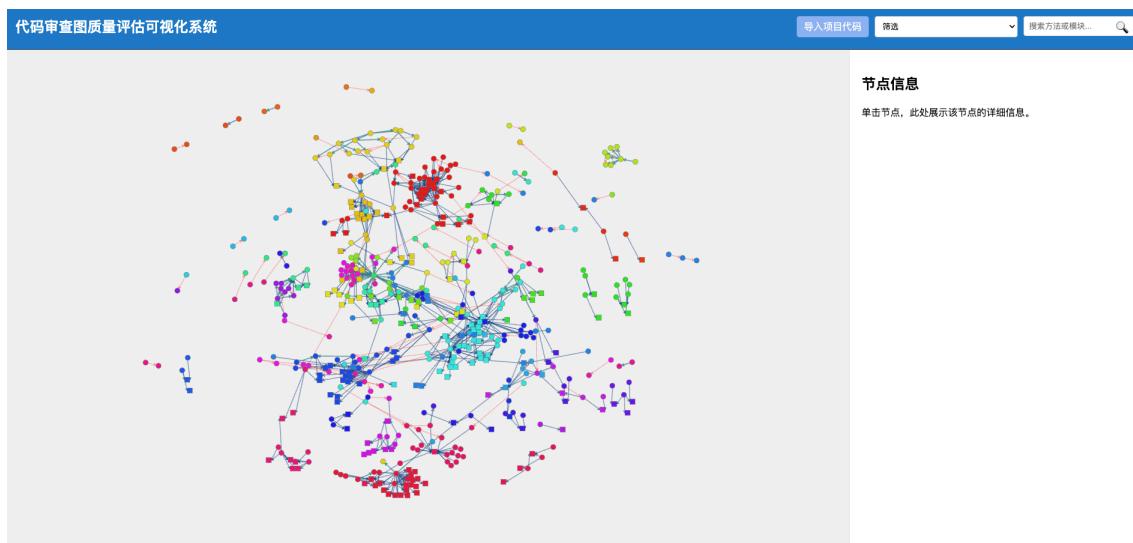


图 4-17 导入项目生成代码审查图

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情况，用户可根据相应的建议进行修改，优化代码质量。



图 4-18 定位开发任务涉及的方法，查看质量信息

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。尤其是变更影响关系，可以帮助用户在进行变更时，提示其依赖型和逻辑型影响的范围，并根据给出的建议，帮助用户安全变更，解决了用户面对复杂软件难以理解、不敢变更、容易变更不完全的痛点。

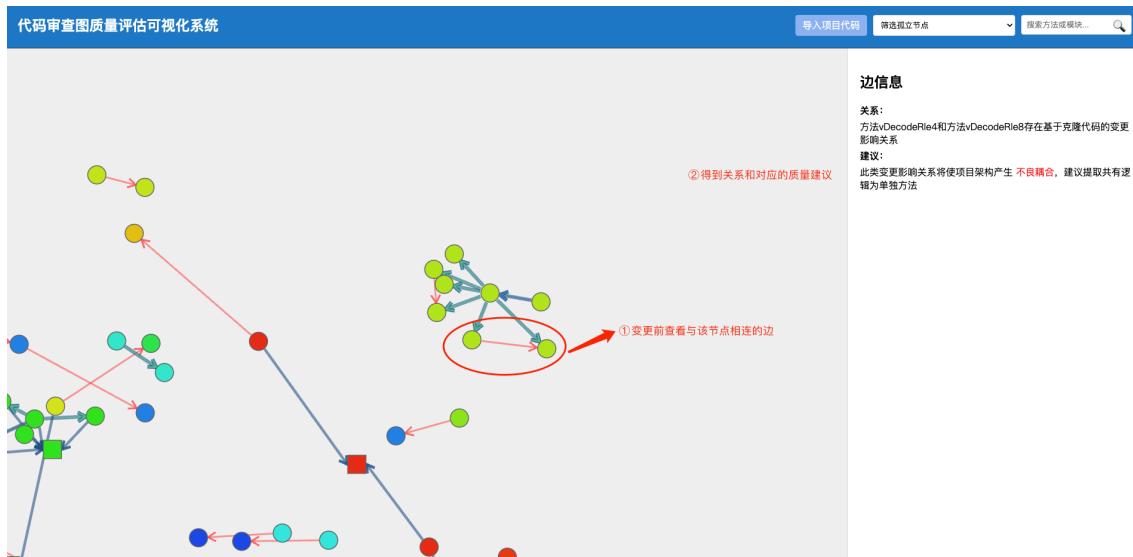


图 4-19 查看方法对应边，按建议安全变更

作为审查工具帮助审查者进行审查 当面对审查任务时，审查者可通过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。

作为质量评估工具对软件代码进行整体的评估 用户可根据代码审查图，观察软件模块划分，也可导出代码质量评估报告，以清单的方式了解项目代码的质量情况。

4.6 本章小结

本章介绍了基于标签生成的方法模块分类，该方法通过大语言模型生成方法的实际标签进行分类，表明模块划分不正确的方法，通过实验证明，该方法具有一定的参考意义。其次介绍了代码审查图，将软件分析结果以图的形式展示给用户，方便用户从宏观的角度了解项目结构，最后介绍了代码质量报告的生成，通过清单式的质量信息报告，方便用户查询和优化。

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] PAHL C, JAMSHIDI P. Microservices: A systematic mapping study[J]. SCITEPRESS - Science and Technology Publications, Lda, 2016.
- [3] DAI P, WANG Y, JIN D, et al. An improving approach to analyzing change impact of c programs[J]. Computer communications, 2022(Jan.): 182.
- [4] NUÑEZ-VARELA A S, PÉREZ-GONZALEZ H G, MARTÍNEZ-PEREZ F E, et al. Source code metrics: A systematic mapping study[J/OL]. Journal of Systems and Software, 2017, 128: 164-197. <https://www.sciencedirect.com/science/article/pii/S0164121217300663>. DOI: <https://doi.org/10.1016/j.jss.2017.03.044>.
- [5] SHENEAMER, ABDULLAH, ROY, et al. A detection framework for semantic code clones and obfuscated code[J]. Expert Systems with Application, 2018.
- [6] LI H, KWON H, KWON J, et al. A scalable approach for vulnerability discovery based on security patches[C]//International Conference on Applications Techniques in Information Security. 2014.
- [7] SHAHRIAR H, ZULKERNINE M. Mitigating program security vulnerabilities: Approaches and challenges[J]. ACM Computing Surveys, 2012, 44(3): 1-46.
- [8] BASET A Z, DENNING T. Ide plugins for detecting input-validation vulnerabilities [J]. IEEE, 2017.
- [9] BALOGLU B. How to find and fix software vulnerabilities with coverity static analysis[C]//Cybersecurity Development. 2016.
- [10] SPANOS G, ANGELIS L, TOLOUDIS D. Assessment of vulnerability severity using text mining[J]. ACM, 2017: 1-6.
- [11] SPANOS G, ANGELIS L. A multi-target approach to estimate software vulnerability characteristics and severity scores[J]. Journal of Systems and Software, 2018, 146(DEC.): 152-166.
- [12] KUDJO P K, CHEN J, MENSAH S, et al. The effect of bellwether analysis on software vulnerability severity prediction models[J]. Software Quality Journal, 2020, 28(1): 1-34.

- [13] QU Y, GUAN X, ZHENG Q, et al. Exploring community structure of software call graph and its applications in class cohesion measurement[J]. Journal of Systems Software, 2015, 108(oct.): 193-210.
- [14] NAKAMURA M, HAMAGAMI T. A software quality evaluation method using the change of source code metrics[J]. IEEE, 2012.
- [15] MISRA S, AKMAN I, COLOMO-PALACIOS R. Framework for evaluation and validation of software complexity measures[J]. Iet Software, 2012, 6(4): 323-334.
- [16] MISRA S, KOYUNCU M, CRASSO M, et al. A suite of cognitive complexity metrics[C]//International Conference on Computational Science Its Applications. 2012.
- [17] CONCAS G, MARCHESI M, MURGIA A, et al. Assessing traditional and new metrics for object-oriented systems[J]. ACM, 2010.
- [18] 陈振强, 徐宝文. 一种基于依赖性分析的类内聚度度量方法[J]. 软件学报, 2003, 14(11): 8.
- [19] BRIAND L C, MORASCA S, BASILI V R. Property-based software engineering measurement[J]. IEEE Press, 1997.
- [20] QU Y, GUAN X, ZHENG Q, et al. Exploring community structure of software call graph and its applications in class cohesion measurement[J/OL]. Journal of Systems and Software, 2015, 108: 193-210. <https://www.sciencedirect.com/science/article/pii/S0164121215001259>. DOI: <https://doi.org/10.1016/j.jss.2015.06.015>.
- [21] 马健, 刘峰, 樊建平. 面向对象软件耦合度量方法[J]. 北京邮电大学学报, 2018, 41(1): 6.
- [22] ARDITO L, COPPOLA R, BARBATO L, et al. A tool-based perspective on software code maintainability metrics: A systematic literature review[J]. Scientific Programming, 2020(8840389).
- [23] LI W, HENRY S. Object-oriented metrics that predict maintainability[J]. Journal of Systems Software, 1993, 23(2): 111-122.
- [24] CHEN W, IQBAL A, ABDRAKHMANY A, et al. Large-scale enterprise systems: Changes and impacts[J]. lecture notes in business information processing, 2013.
- [25] ARNOLD R S. Software change impact analysis[M]. Washington, DC, USA: IEEE Computer Society Press, 1996.
- [26] 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.

- [27] GALLAGHER K B, LYLE J R. Using program slicing in software maintenance[J]. IEEE Transactions on Software Engineering, 1991, 17(8): 751-761.
- [28] JITENDERKUMARCHHABRA, MRINAALMALHOTRA, JITENDERKUMARCHHABRA, et al. Improved computation of change impact analysis in software using all applicable dependencies[J]. Springer, Singapore, 2018.
- [29] 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.
- [30] 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.
- [31] 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.
- [32] 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.
- [33] 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).
- [34] 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.
- [35] 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.
- [36] CAI H, SANTELICES R, XUT. Estimating the accuracy of dynamic change-impact analysis using sensitivity analysis[C]//Eighth International Conference on Software Security Reliability. 2014.
- [37] CAI H, THAIN D. Distia: a cost-effective dynamic impact analysis for distributed programs[C]//the 31st IEEE/ACM International Conference. 2016.

- [38] 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.
- [39] 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.
- [40] ZANJANI M B, SWARTZENDRUBER G, KAGDI H. Impact analysis of change requests on source code based on interaction and commit histories[C]//ACM. 2014.
- [41] 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.
- [42] ZIMMERMANN T, ZELLER A, WEISSGERBER P, et al. Mining version histories to guide software changes[J]. IEEE Transactions on Software Engineering, 2005.
- [43] 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.
- [44] LLVM-ADMIN TEAM. Clang, a c language family frontend for llvm.[EB/OL]. 2024. <https://clang.llvm.org/>.
- [45] LLVM-ADMIN TEAM. libclang: C interface to clang.[EB/OL]. 2024. https://clang.llvm.org/doxygen/group__CINDEX.html.
- [46] GOMARIZ A, CAMPOS M, MARIN R, et al. Clasp: An efficient algorithm for mining frequent closed sequences[Z]. 2013.
- [47] 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.
- [48] ADJOYAN S, SERIAI A D, SHATNAWI A. Service identification based on quality metrics object - oriented legacy system migration towards soa[C]//International Conference on Software Engineering Knowledge Engineering. 2014.
- [49] ABDELKADER M, MALKI M, BENSLIMANE S M. A heuristic approach to locate candidate web service in legacy software[J]. International Journal of Computer Applications in Technology, 2013, 47(2/3): 152-161.
- [50] CHIDAMBER S R, KEMERER C F. A metrics suite for object oriented design[J]. Software Engineering IEEE Transactions on, 1994, 20(6): 476 - 493.

- [51] 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.
- [52] 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.
- [53] BIEMAN J M, KANG B K. Cohesion and reuse in an object-oriented system[C]// Proceedings of the 1995 Symposium on Software reusability. 1995.
- [54] 退曲. 关于软件设计的模块独立性分析[J]. 数字技术与应用, 2011(3): 2.

哈尔滨工业大学学位论文原创性声明和使用权限

学位论文原创性声明

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

作者签名： 日期： 年 月 日

学位论文使用权限

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

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

保密论文在保密期内遵守有关保密规定，解密后适用于此使用权限规定。

本人知悉学位论文的使用权限，并将遵守有关规定。

作者签名： 日期： 年 月 日

导师签名： 日期： 年 月 日

致 谢

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

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

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

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