硕士学位论文

（学术学位论文）

面向代码质量评估的变更影响分析方法研究

RESEARCH ON CHANGE IMPACT ANALYSIS METHODS FOR CODE QUALITY ASSESSMENT

李美娜

哈尔滨工业大学

2024 年 12 月

国内图书分类号：TM301.2　　学校代码：10213
国际图书分类号：62-5　　密级：公开

硕士学位论文

面向代码质量评估的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：2024年12月
授予学位单位：哈尔滨工业大学

Classified Index: TM301.2
U.D.C: 62-5

Dissertation for the Master's Degree

RESEARCH ON CHANGE IMPACT ANALYSIS METHODS FOR CODE QUALITY ASSESSMENT

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

摘　要

摘要的字数（以汉字计），硕士学位论文一般为 500 ~ 1000 字，

关键词：代码质量分析；变更影响；代码审查

Abstract

An abstract

Keywords: code quality analysis, change impact, code review

目　录

第 1 章　绪论

1.1　课题研究的背景和意义

随着计算机和软件技术的不断进步，软件系统的规模和复杂性也随之增长，尤其是那些被称为遗留系统（legacy system）或遗产软件的项目。这些遗留系统由于其庞大的规模和维护升级的难度，成为了代码质量分析的重要领域。根据 Chiu 等人的定义 [1]，遗留系统是指那些维护和升级难度较高的大规模软件系统。而 Carvalho 等人 [2] 则将遗留系统描述为依赖过时技术的系统，这些系统经过多年的广泛维护，导致其架构衰变和退化。遗留系统的复杂性不仅体现在它们由紧密相连且相互依赖的组件构成，难以扩展和创新 [3]，还在于它们对维护资源的巨大需求，这阻碍了数字化转型的步伐。尽管如此，这些系统代表着长期的大规模投资，包含企业的重要数据和技术流程，完全抛弃可能会导致重要资产流失，不可轻易放弃。因此，为了保持竞争力和保护现有投资，许多公司都选择对这些遗留系统进行继续维护或者重构。

除了遗留系统之外，对于一般的软件项目，持续维护也是最重要的代码活动之一。调查显示，软件系统的维护成本在历年的软件维护预算中占 60% 到 80% 的比重 [4]。标准的代码维护工作的流程图如图 1-1 所示，主要分为以下三个步骤：

（1）开发人员进行代码开发。修改项目代码后，针对修改部分进行回归测试，验证功能是否符合需求，同时避免缺陷的引入。

（2）测试通过后，由审查人员进行代码审查。代码审查的主要目的是确保代码质量，通过识别潜在的错误和优化点，提升代码的可维护性与可靠性，同时保证开发风格的统一。

（3）审查通过后，则可与原始项目进行合并。

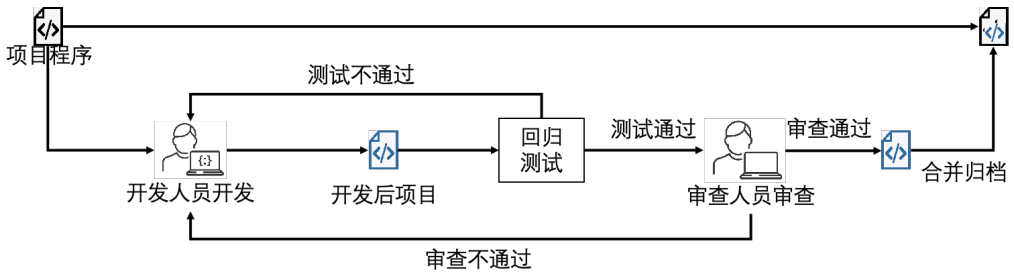

图 1-1 示例代码

软件项目由于历史原因，往往存在大量低质量代码和技术债务。在对软件项目进行维护或重构的过程中，代码质量分析是具有指导性意义的关键一环。代码质量分析可以帮助开发者识别代码质量问题和问题所在的位置，提供有针对性的优化以及重构方案，从而降低成本和风险。通过有效的代码质量分析，企业可以逐步提升软件系统的稳定性和可维护性，为实现数字化转型打下坚实基础。这不仅能够保护企业的长期投资，还能提升整体竞争力，确保在快速变化的市场中保持领先地位。

代码变更影响分析是代码质量分析的重要方法之一。通过代码变更影响分析，能够确定当某处代码进行变更时，系统中的其他哪些部分会受到影响，这可以让开发人员提前了解与变更部分有依赖关系的其他模块，避免变更时引入问题。

对于大型软件系统，代码变更影响分析更显得尤为关键。大型软件系统往往融合了不同开发者的个性化风格，随着时间的推移，系统的模块化设计原则逐渐被繁复和庞大的代码结构所淹没。这种情况下，精准地识别变更后有影响的代码模块，对于系统的有效重构和维护至关重要。然而，传统依赖经验丰富的专家进行手工提取的方法，在面对日益复杂化的系统结构时，显得力不从心[5]。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[6]。这种方法为软件系统的维护和升级提供了一种更为科学和系统的解决策略，有助于实现软件质量的持续提升和维护成本的有效控制。

本文将从代码变更影响分析入手进行方法研究，面向代码质量评估，同时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，帮助开发者和审查人员更直观地理解整个项目，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、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。

代码质量分析方法按照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、代码故障等概念也属于代码缺陷研究的范畴。软件代码中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，对系统和应用的安全造成影响。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[1-2]或基于模式的检测器[3-5]。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[6-8]，作为分类器，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[9]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。基于传统度量指标的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、CK度量的基础上，结合新型指标进行复杂度评估。除此之外，研究表示软件代码可以作为复杂网络来进行研究[10-11]，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。此外，基于认知复杂度也是代码复杂度的研究方法之一。认知复杂度衡量执行任务所需的人类努力[12-13]，是Shao和Wang[14]引入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，其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，使得系统更易于扩展和测试。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，有少量研究是对传统度量指标的改进。[15] 提出一种基于程序依赖性分析的类内聚度度量方法, 它从属性与属性之间、属性与方法之间以及方法与方法之间三个方面对类进行全面分析. 这三个方面既可以单独度量类的内聚度, 也可以综合使用. 证明了提出的方法满足 Briand 提出的关于优良的内聚度度量准则的四个基本性质。QU[16] 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，因此能够反映类的内聚程度。研究表明，这些指标可以提供现有类内聚性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。马健等人[17]，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 代码质量度量套件的国内外研究现状

根据不同的研究主题，演化出了很多代码度量套件，这些度量被广泛应用于与源代码相关的各种应用程序和实验中 (例如，故障预测、测试、重构)，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[18]，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。

CK 套件主要是针对面向对象语言设计的度量套件，具体含义如表 1-1。CK 套件中共有六个核心指标，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

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

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[19]，主要用于评估程序的复杂度、维护性以及开发成本等度量套件。LH 套件中有一部分指标和 CK 套件是一样的,除此之外，还有另外五个核心指标。

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）是软件工程中用于评估代码变更对系统其他部分可能产生影响的一种技术。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[20]，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[21] 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。在代码变更之前，对其影响进行分析，可以估计变更可能造成的负面影响。优点可总结如下：

（1）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）有助于代码质量与合规性要求。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）来进行分析和推理。在静态分析中，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[22-23]，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[24-26]。

在此领域的研究中，Schrettner 等人[27] 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，可以作为影响分析的基础。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缺陷，显著提高了静态分析的准确性和实用性。

Sun 等人[28] 则提出了变更类型与影响机制相结合的影响分析方法。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，以期提高变更影响分析的准确性和健壮性[29]。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集合。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，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（OOP）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受到不精确性的影响。为了提高动态分析的精确性，Huang 等人[30] 提出了一种专门针对面向对象程序的精确动态影响分析方法。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，显著减少了影响集合的规模，从而提升了分析的效率和精度。

在动态影响分析的精确性和可靠性方面，Cai 等人[31-32] 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[33]。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。DISTIA 方法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。

Basri 等人[34] 则提出了一种结合静态和动态分析的混合影响分析方法，旨在支持在软件工件状态不一致时的影响估算。在这一方法中，静态分析用于估算部分开发工件的工作量，而动态分析则用于已完成工件的工作量估算。这种方法通过综合静态分析和动态分析的优势，不仅提升了分析的精度，还能有效处理开发过程中不一致的工件状态，提供了一种灵活且高效的影响分析策略。

（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的实现，而是转向了软件变更的历史库[25,35-38]。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[25] 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集的生成技术。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[37] 则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[38] 则致力于改进现有的耦合分析算法，尤其是在软件变化的上下文中。TARMAQ 算法是他们提出的一种新型算法，在性能上明显优于 ROSE[?] 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。

Dit 等人[39] 提出了一种自适应组合方法 Impactminer，通过静态文本分析、动态执行追踪和软件库挖掘的结合来估算变更影响集。该方法能够通过对源代码的静态分析和运行时信息的结合，挖掘出更准确的变更影响关系。与传统静态分析方法相比，Impactminer 能更好地捕捉到软件系统中的动态行为和变更的实际影响，因此能够提供更为可靠的变更影响估算结果。

Huang 等人[40] 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。

1.3 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。这三部分各自的主要研究内容如下：

（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

随着软件开发迭代的加速和代码复杂度的增加，代码变更对软件质量的影响变得愈加复杂。尤其是在持续集成和快速迭代的开发模式下，如何高效评估

和管理这些变更对软件质量的影响，成为了提升软件可靠性和维护性的重要挑战。

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）代码审查图生成

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、重复性等方面的指标，能够有效反映项目的质量状况；其次是变更影响关系，该部分通过分析软件变更对其他模块和功能的影响，帮助开发者预测和规避潜在的风险；最后是模块标签，这基于软件项目的实际模块结构，通过使用大语言模型进行预测和分析，从而为开发者提供对软件模块化质量的客观评价。

在代码审查图中，节点代表项目中的关键元素，如方法、函数以及全局变量；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。

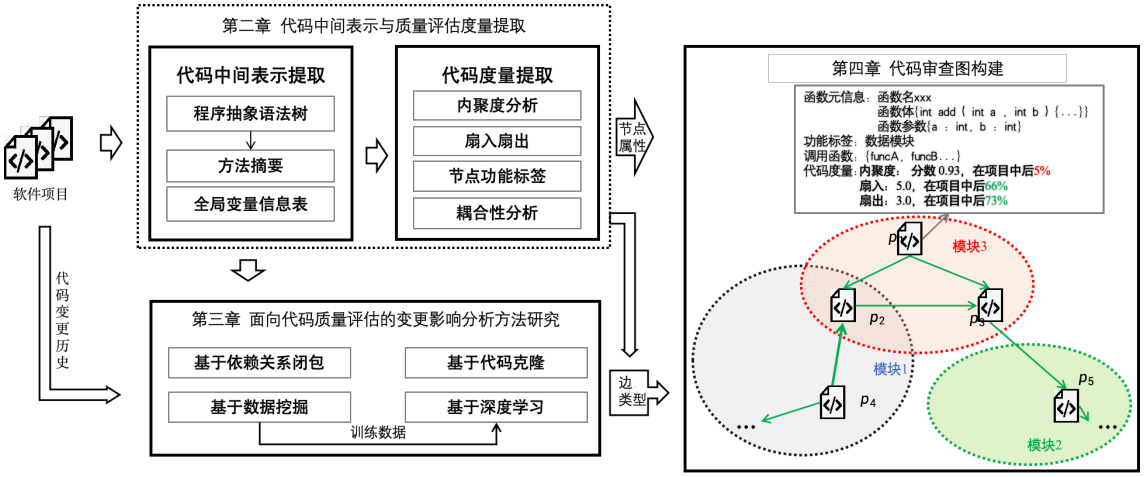

图 1-2 章节安排

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　代码中间表示与质量评估度量提取

2.1　引言

2.2　基于 clang 的抽象语法树生成方法

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。从这棵树的根节点出发，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。在这棵树中，节点之间的连接仅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 [7]。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 [8]。本文使用了 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了两个函数和一个全局变量，主函数用于计算两个数的和。这段代码由 Clang 解析生成抽象语法树后，得到的树结构如下图 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、函数的参数列表、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

```c
#include <stdio.h>

int globalVar = 10;

int addNumbers(int a, int b) {
    return a + b;
}

void main() {
    int num1 = 5;
    int num2 = 7;
    int sum = addNumbers(num1, num2);
    printf("Sum of %d and %d is: %d\n", num1, num2, sum);
}
```

图 2-1 示例代码

的类型（如是否为方法定义、变量定义、变量引用等）、获取 cursor 所代表的源代码元素的名称、类型、位置等信息、获取 cursor 的父节点或子节点等。本文通过操作游标，遍历 AST，获取整个 AST 的结构。

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 指的是函数定义，在 clang 中是不区分函数声明和函数定义的，统一用 Function_Decl 来标识，两个区分主要看是否有函数体，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，函数 addNumbers 有两个参数 a 和 b。CompoundStmt 代表大括号，函数实现、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 标签表示函数调用 Expr，子节点有调用的参数列表。ReturnStmt 表示返回语句。

2.3 方法调用链提取与分析

本文以整个代码项目为分析对象，代码中的方法为分析单元，方法之间的调用关系则是本文的分析基础。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，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

```
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

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时，对变量进行作用域的判断。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。具体的计算流程如算法 2-1 所示。

算法 2-1 Scan and Analyze Code Files

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

分析结束后，将会获得每个方法的方法调用链，对应于一个方法摘要表，包括项目代码中所有的方法和方法之间的调用关系，除此之外还包括方法的参数表、方法主体和所在文件等其他信息。

2.4 全局变量定义-使用链提取与分析

在 C/c++ 代码中，全局变量的定义、作用域、生命周期和方法是类似的，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。

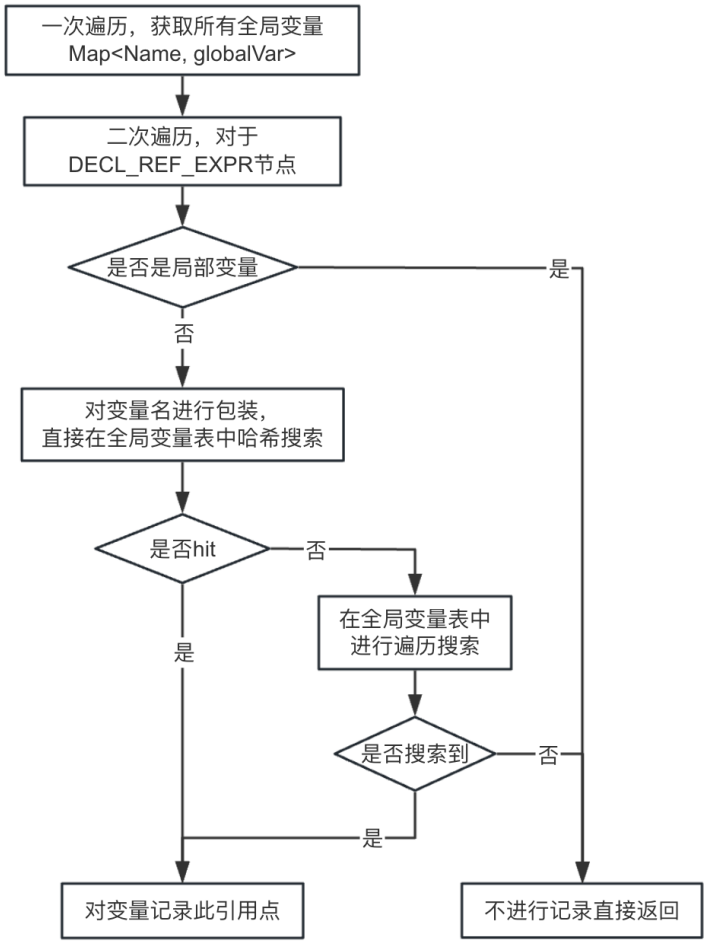

图 2-3 示例代码对应的抽象语法树结构

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函数，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，则获得了每个全局变量的定义-引用链，对应于一个全局变量信息表，包括项目代码中所有的全局变量和变量的引用点，除此之外还包括全局变量的类型、作用域和所在文件等其他信息。

2.5 基于方法特征的代码度量提取

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质量。代码内聚度指的是模块（如函数、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 [9]。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最小化，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 [10]。

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。

2.5.1 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。这里一共计算以下四个指标：

（1）LCOM1，含义是不引用相同字段的方法对数目 [11]。计算公式如式 (2-1)

$$LCOM1 = \begin{cases} P - Q, if P \geq Q \\ 0, otherwise \end{cases} \tag{2-1}$$

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。在计算时，对于每个文件首先构建方法耦合图（method coupling graph），将提取到的方法作为图中节点，对方法两两进行判断，如果两个节点都引用相同的字段，则它们之间用一条无向边连接，按式 (2-2) 计算，

$$LCOM1 = choose(n, 2) - e \tag{2-2}$$

其中 n 是文件中的方法总数，e 是图中的实际边数，即图中可能的最大边数减去实际的边数，得到的值即为 LCOM1

（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 [12]。LCOM2 相比于 LCOM1，考虑到了类中所有方法和变量的相互作用，其计算公式如式 2-3：

$$LCOM2 = \frac{1}{a}\sum(a - m_a) \tag{2-3}$$

其中，a 是类中属性的数量，m_a 是访问属性 a 的方法数量。在计算时，首先构建方法属性图（method-attribute graph），将提取到的方法和全局变量作为图中节点。如果方法引用了变量，则有一条有向边从方法节点指向属性变量，计算公式如式 2-4，

$$LCOM2 = 1 - \frac{e}{n \times a} \tag{2-4}$$

其中 n 为方法数，a 为变量数。假设 e 是图中的实际边数，得到的值即为 LCOM2。

（3）LCOM3，含义是以方法为顶点，两方法引用相同字段则有边构成的无向图的连通分支数 [12]。LCOM3 是对 LCOM2 的进一步改进，其计算公式如式 2-5：

$$LCOM3 = (\frac{\sum_{i=1}^{n}(m_i - a_i)}{m}) - a \tag{2-5}$$

其中，m 是方法的数量，a 是变量的数量，m_i 是第 i 个方法访问的属性数量，a_i 是被第 i 个方法访问的属性数量。LCOM3 试图通过分析方法和属性之间的关系来提供更细致的内聚度度量。

实际计算时，如式 2-6，基于上述计算出的 LCOM2 指标可以快速地计算出 LCOM3

$$LCOM3 = \frac{n}{n-1} \times LCOM2 = \frac{n - \frac{e}{a}}{n-1} \tag{2-6}$$

（4）LCOM4，含义是以方法为顶点，两方法引用相同字段或有调用关系则有边构成无向图的连通分支数 [13]。

计算时，构建方法耦合图，节点是方法。如果两个节点都引用相同的字段，则它们之间用一条无向边连接，除此之外，如果两个节点有调用关系，则也用一条无向边将它们连接。根据深度搜索的方式，计算图中的连通分支数，得到的值即为 LOCM4.

2.5.2 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 [14]。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。计算时，对于每个文件首先构建方法耦合图，对方法两两进行判断，如果两个节点都引用相同的字段，则它们之间用一条无向边连接，计算公式如式 2-7，其中 n 是文件中的方法总数，e 是图中的实际边数

$$TCC = \frac{e}{choose(n, 2)} \tag{2-7}$$

（2）LCC，基于方法间接引用共同字段关系的传递闭包计算 [14]。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。

计算时，对于每个模块首先构建方法耦合图，对方法两两进行判断，如果两个节点都引用相同的字段，则它们之间用一条无向边连接表示直接连接，如果方法之间有调用，则也将方法进行连接，并连接当前节点和调用节点的所有邻居节点。计算公式如式 2-8，

$$LCC = e + \frac{e_{indirect}}{choose(n,2)} \tag{2-8}$$

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。

2.5.3 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，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，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 2-1 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	是
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。

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函数，结合这两个信息，即可确定两个方法是否存在着标记耦合关系。

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。

2.5.4 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。

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函数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。

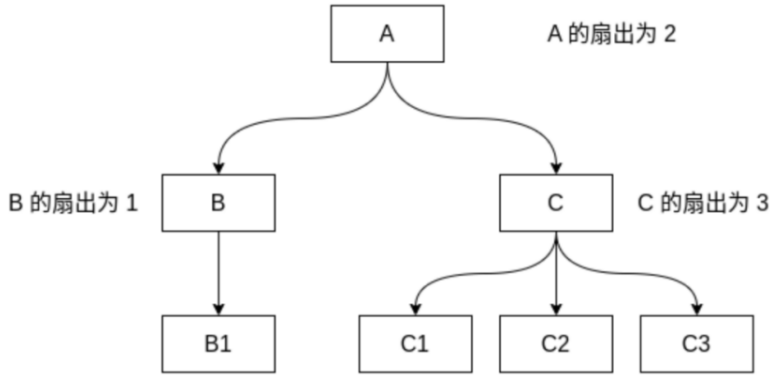

图 2-4 扇入扇出示例图

2.5.5 结合静态检测工具提取缺陷

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了全面的检测和分析。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。通过集成 Cppcheck，我们不仅能够评估代码的功能性和正确性，还可以进一步揭示代码中潜在的安全隐患和性能瓶颈。检测结果中包括了诸如错误、警告和建议等不同级别的信息，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。

此外，Cppcheck 提供的报告与其他度量指标（如内聚度、耦合度等）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。

2.6 实验结果与分析

2.6.1 实验数据描述与分析

本文收集了表 2-2 中所示的软件项目为示例项目，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。除此之外，这些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，以供后续的实验分析。

表 2-2 示例项目

项目名称	项目简介	收藏数
antiword	用于提取 Microsoft Word 文档内容的工具	13k
librdkafka	Apache Kafka 的 C/C++ 客户端库，用于与 Kafka 集群进行通信	18k
TheAlgorithms	各种算法的开源实现的集合，涵盖了计算机科学、数学和统计学、数据科学、机器学习、工程等各种主题	57k
libbpf	linux 内核观测技术的一个脚手架库	1.9k
FFmpegKit	FFmpeg 工具包	3.7k
jemalloc	通用的 malloc(3) 实现，强调碎片避免和可扩展的并发支持	9k

对示例项目提取基于内聚度缺乏度和基于连通性的内聚度指标、耦合性以及扇入扇出，这些指标一定程度上反映了代码质量。内聚度在本文中是一个模块级别的指标，它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。内聚度高的模块意味着模块内的各个部分紧密合作以完成某一特定任务，模块的职责单一，功能聚焦。本文中计算了两类共六种内聚度指标，在同一项目中，人工检查每种内聚度在最差的前 5% 的模块，判断是否能被开发者接受，以此来验证内聚度指标的有效性。

对于耦合性，本文中提取了五种不同的耦合类别，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。这里对于耦合性的判别标准需要根据是否在同一个模块进行区分。对于同一模块内，代码往往有共同的上下文和职责范围，因此要求尽量采用数据耦合，因为通过参数传递数据，方法的输入输出明确，易于测试和维护；允许标记耦合和外部耦合，允许通过传递数据结构优化参数列表简洁性；减少公共耦合，模块内共享状态可提高效率，但要避免复杂的依赖关系。对于不同模块的方法，则要求尽量采用数据耦合，数据耦合方法之间的独立性强，模块职责单一，代码易于测试和复用；减少标记耦合，优化接口设计，避免通过标记变量传递复杂的变量；杜绝公共耦合、外部耦合：避免全局变量的滥用，防止影响模块之间的依赖关系和模块内部的可测试性。同样也是通过人工检验不符合要求的耦合性，判断是否能被开发者接受，以此来验证耦合性指标的有效性。

对于扇入扇出也是类似的，传统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。这里也是通过人工检查两种指标在最差的前 5% 的方法，判断是否能被开发者接受，以此来验证扇入扇出指标的有效性。

为了更准确地衡量代码质量，本文结合了静态分析工具 Cppcheck，对项目代码进行了检测。检测结果作为质量评估的一部分，提供给用户进行判别。

接受率指标计算公式如式 2-9

$$AR = \frac{A}{T} \times 100\% \tag{2-9}$$

其中 AR 表示接受率，A 表示开发者接受的不符要求的示例数，T 表示方法检测到的不符要求的样例数。

2.6.2 实验结果与分析

1. 内聚度接受率实验结果及分析

（1）内聚度接受率实验结果

对示例项目人工检验得到的接受率进行统计，得到的结果如表 2-3。

表 2-3 内聚度接受率

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

对于内聚度的接受率，经过分析可以发现指标基本都在 70% 以上，表现良好。通过对比，发现在 antiword、TheAlgorithms 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. 内聚度接受率实验结果示例分析

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 文件进行分析，该文件作为一个模块来进行内聚度的计算。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。

通过分析可以发现，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. 其他度量接受率实验结果及分析

对示例项目人工检验得到的接受率进行统计，得到的结果如表 2-4。

表 2-4 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
antiword	84.2	56.4	100.0
librdkafka	73.7	43.3	98.8
TheAlgorithms	99.2	87.6	100.0
libbpf	78.3	42.1	100.0
FFmpegKit	64.0	43.3	94.6
jemalloc	87.8	43.3	100.0

从实验结果来看，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）耦合性接受率分析

首先分析耦合性，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，这是因为简单项目结构较为简单，各个模块功能明确，职责单一，所以耦合性不良的样例很少，维护起来也更加容易，尤其对于 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。而在复杂项目中，如 FFmpegKit，耦合性接受率相较其他项目低一些，这是由于对与大型、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。

对耦合性实验结果进行进一步分析，针对不接受的样例，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 2-5 其他度量接受率实验结果

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

除此之外，还有一些被拒绝的样例具有共同的特征：它们涉及的都是工具类方法。例如，在 libbpf 项目中，fail 方法作为一种工具性方法，主要用于处理程序中的内部错误并输出详细的错误信息。由于该方法的功能是对程序中的异常进行集中处理，因此它在项目中会被频繁调用。然而，尽管该方法具有重要作用，但它本身并不具备足够的独立性，难以作为一个单独的模块存在。为了避免对项目结构造成过度拆分，类似功能的工具方法通常会被归纳在一个工具类模块中，以便于集中管理和供外部调用。因此，尽管这些方法的耦合度较高，但它们在实现项目功能时是不可或缺的。通过合理组织这些工具类方法，我们能够保持系统结构的清晰与高效，同时确保错误处理等基础功能能够在项目中得以统一和有效地执行。

（2）扇入扇出接受率分析

对于扇入扇出值，可以发现在这个指标上的接受率是最低的。经过实例分析，发现大部分的项目的开发模式并不完全符合高扇入低扇出的标准。比如低

扇入的模块虽然复用性低，但是更加独立，意味着它们不太依赖于其他模块的实现，因此更容易被单独测试、维护和扩展，考虑到未来软件的发展，这些低扇入的方法也可能被调用，所以不能被直接接受。

对于高扇出方法的检测结果，大部分是可以接受的，但是存在某些高扇出的模块实际上是中心化的模块，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，是项目中扇出度最大的一个，但是并未被用户所接受。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。根据不同的 Word 版本和转换需求，方法会选择不同的处理逻辑，导致总共产生了 30 个方法调用。集中处理所有功能逻辑的设计思路减少了冗余代码，避免了方法调用链过长所带来的潜在性能问题，尤其是在内存栈使用方面。虽然高扇出度可能会导致一定的复杂性，但对于该方法而言，集中式的处理方式有效降低了因方法调用带来的额外开销，并优化了整体系统的效率。

（3）静态工具检验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头文件引用错误、变量未使用等错误等不容易被开发者所接受，因为这并不是代码本身的结构问题。

2.7 本章小结

本章主要探讨了代码中间表示的提取及其质量评估度量的生成过程。首先，本文提出了基于 libclang 工具对软件项目进行抽象语法树提取的方法，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，用于对代码质量进行全面评估。最后，本章通过一系列实验验证了所提出方法的有效性，并对实验结果较差的样例进行了深入分析，探讨了其可能的原因与改进方向。

第 3 章　面向代码质量评估的变更影响分析方法研究

3.1　引言

软件变更是软件维护和发展的关键操作。通常，对软件系统所做的更改可能会导致对系统其余部分的不良副作用或连锁反应。CIA 的目标是识别涟漪效应，并防止拟议变更的副作用。变更影响分析过程的输入是变更集。在进行变更影响分析后，计算影响集。变更集是用户提出的修改请求的集合，或者是软件中要更改的元素的集合。它可以引用代码级方法、语句或变量。影响集是可能受到变更集中元素影响的元素集合。影响集中的元素可以是受变更影响的文档规范、代码和测试用例。

但是本文的目的并不是根据用户的更改识别受影响的其他代码部分，而是通过代码变更影响来反映软件的代码质量，生成代码质量报告反馈给用户，因此在本文中，变更影响分析的对象是方法，识别每一个方法更改后受影响的部分，提取这种关系，反馈给用户。

方法之间的变更影响的类型可以分为以下三种类型:

（1）有直接调用关系这种变更影响关系是显而易见的。比如方法 funcA 调用了方法 funcB，那么当 funcB 的方法签名有所变化时，调用它的方法必须对应地修改代码才能正常调用，否则会调用不成功，甚至直接发生编译错误。

（2）共同调用另一个方法如当 funcA 和 funcC 同时调用了方法 funcB 时，funcA 和 funcC 之间会有变更影响关系。这种变更影响关系实际上是类型 1 的扩展，这种情况通常是 funcB 的更改引起了 funcA 和 funcC 的同时更改。

（3）逻辑上存在变更影响关系这种变更影响关系中，方法之间虽然没有直接的调用关系，也不共同调用某个方法，但它们的实现逻辑或者所操作的数据之间存在某种隐含的关联。这种关系可能是由于它们共同维护某个数据的一致性、共享某些资源或者它们的输出和输入之间存在某种预期的联系。因此，当其中一个方法发生变化时，可能会间接影响到其他方法的行为或结果，即使这种影响在代码的静态结构中不直接可见。

3.2 基于依赖关系闭包的变更影响分析

变更影响分析的主要目标是识别软件系统中由代码修改引发的涟漪效应，从而有效防止变更产生的副作用。这一过程旨在帮助开发者全面评估变更可能带来的直接和间接影响，并制定相应的测试与维护策略。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，是一种基于静态依赖关系的技术手段，能够通过识别代码模块间的关联性，精准划定受变更影响的范围[29]。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。

依赖关系传递闭包分析的基本假设是，软件系统中的各个代码单元（如方法或全局变量）存在不同程度的直接或间接依赖。通过对这些依赖关系的深入分析，可以识别代码变更对系统其他部分的潜在影响。本文以方法和全局变量为基本分析单元进行变更影响分析。

（1）代码预处理

在进行代码变更影响分析之前，首先需要将源代码转化为一种可供进一步分析的程序模型。该模型通常由一组能够描述代码结构和依赖关系的数据结构组成，其目的是通过抽象化和结构化的方式，将复杂的代码逻辑转化为更易于分析的形式。本文提出的预处理模块设计基于这一需求，核心思想是以抽象语法树为基础，构建程序的调用图、全局变量信息表和方法摘要表，为后续的依赖关系分析和影响范围评估提供数据支撑。

首先，使用 libclang 对源代码解析生成抽象语法树。抽象语法树是一种树状数据结构，用于表示源代码的语法结构，其中每个节点对应于代码中的一种语法构造（如变量声明、函数调用等）。基于抽象语法树可以直观地捕获代码的层次结构，为进一步的依赖关系分析提供了良好的起点。在生成抽象语法树之后，进一步构建全局变量信息表和方法摘要表。全局变量信息表用于记录程序中所有全局变量的相关信息，包括变量名、定义位置，以及使用了该变量的方法的名称。通过这些信息，能够明确全局变量与方法之间的依赖关系，从而识别代码中跨模块或跨方法的依赖。方法摘要表旨在对程序中的方法进行全面的结构化描述。该表包含每个方法的签名信息（如方法名和参数列表）、定义位置（如所在文件）、方法内定义的局部变量，以及方法调用的其他方法列表。这些信息不仅可以用于分析方法间的调用关系，还可以帮助区分局部变量和全局变量，从而更精确地提取依赖关系。

这些数据结构共同构成分析所需的程序模型，为依赖关系的提取和后续的

传递闭包计算提供基础支持。

（2）依赖关系图的构建

依赖关系图图的构建依赖于前述提取的全局变量信息表和方法摘要表。在构建过程中，首先需要识别代码中的直接依赖关系，包括函数调用和变量引用等。前面提取的全局变量信息表和方法摘要表已经包含了这些直接依赖关系的信息。接下来，依赖关系被形式化为一个有向图，图中的节点代表代码中的基本元素，例如方法、全局变量和其他代码单元，而有向边则表示这些元素之间的依赖关系。例如，如果方法 A 调用了方法 B，那么在调用图中就会有一条从节点 A 指向节点 B 的有向边。通过这种方式，调用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。

（3）变更影响分析的执行

接下来，基于依赖关系图进行变更影响分析，实际上是对方法之间变更影响关系的全面识别与提取过程。具体而言，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）来识别受影响的方法集 IMS（impacted method set）。

$$\begin{aligned} &RBM = CALL \cup RETURN \text{ where} \\ &(f, g) \in CALL \iff f(\text{ transitively})\text{calls } g, \\ &(f, g) \in RETURN \iff f(\text{ transitively})\text{returns into } g \end{aligned} \tag{3-1}$$

这里以方法 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。

对于依赖关系图中的每个节点，我们需要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，通过图中定义的依赖关系，可以直接或间接到达的所有节点的集合。换言之，传递闭包反映了节点之间的依赖链条以及影响传播的范围。为了高效地计算传递闭包，利用广度优先搜索和图可及性理论递归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，最终形成该节点的完整传递闭包。传递闭包的具体计算公式如式 3-2

$$
\begin{aligned}
&IMS^{(N)} = IMS^{(N)}_{CALL} \cup IMS^{(N)}_{RETURN} \\
&IMS^{(N+1)}_{RETURN} = \bigcup_{define \in \left(IMS^{(N)}_{RETURN} - IMS^{(N-1)}_{RETURN}\right)} IMS(define) \\
&IMS^{(N+1)}_{CALL} = \bigcup_{define \in \left(IMS^{(N)}_{CALL}\right)} IMS(define), define \in \\
&\left(IMS^{(N)}_{CALL} - IMS^{(N-1)}_{CALL}\right)
\end{aligned}
\tag{3-2}
$$

在变更影响分析的具体应用中，我们假设依赖图中的某个方法节点发生了变更，并按上述公式进行计算，最终得到的节点集合中，所有的节点（即方法）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。

以图 2-5 为例，在这个例子中，一共有 4 个方法，其中方法 funcA 调用了 funcB 和 funcD，funcB 调用了 funcC。在对 funcC 进行变更影响分析时，会直接影响到 funcB，根据依赖关系闭包，会间接影响到 funcA。所以与 funcC 有变更影响关系的方法集合为 funcB, funcA。

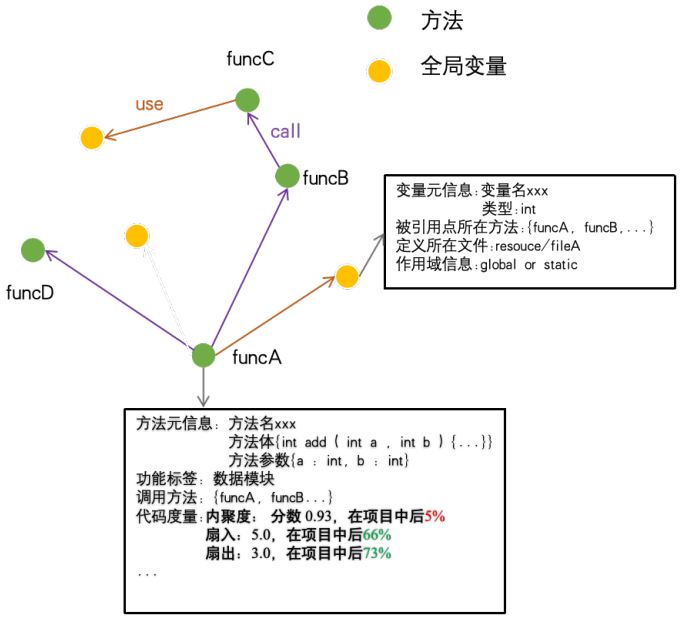

图 3-1 依赖关系示例

3.3 基于代码克隆的变更影响分析

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、函数名或字面量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、添加了功能或进行优化），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。

因此，本文提出了一种基于代码克隆检测的变更影响分析方法。该方法通过识别和检测软件源代码中的代码克隆，揭示了不同代码片段之间的变更影响关系。具体而言，通过检测代码克隆，可以有效地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，从而为后续的代码质量分析提供了关键的依据和数据支持。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得到代码克隆，具体的检测流程如图 3-2 所示。

3.3.1 代码预处理和分块

为了精确有效地检测代码克隆，必须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。

（1）代码标准化

首先，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，去除源代码中的注释是必不可少的步骤，它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。

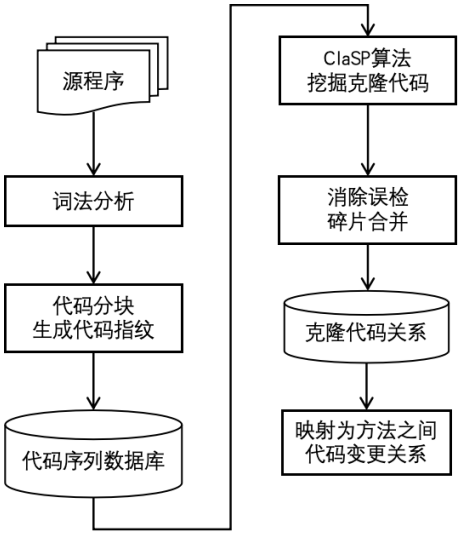

图 3-2 基于代码克隆的变更影响分析方法流程

接下来，去除头文件引用语句。在 C/C++ 程序中，多个源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重复解析相同的头文件，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。因此，合理的头文件处理可以有效地去除冗余，确保分析过程的高效性。

此外，考虑到代码克隆中的语法克隆类型，变量名的标准化处理在克隆检测中具有重要意义。具体而言，为了避免因变量名的变化导致漏检，通常可以在方法内部对变量名进行统一标准化处理。即，通过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。

（2）代码分块

代码分块的步骤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。其核心原因在于减少不同开发者在代码书写风格、格式和结构上的差异对检测结果的影响，确保能够准确识别功能相似或重复的代码块。

本文的代码分块策略分为几步，首先对于基本语句按固定行数分块，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。对于控制语句，识别并根据关键分块词（如 if, while, for, switch, try 等）进行分块，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，而不是因为格式的不同而导致误判。

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。一个关键规则是大括号不被视为代码块的一部分。这一处理策略是基于实际开发中的多样化书写风格所作出的考虑：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。

另外，在分块过程中，为了便于后续的克隆检测和碎片合并，需要详细记录每个代码块的位置信息。这些信息通常包括：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。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，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。

（3）代码指纹提取

这一步通过对代码的特征进行抽象和提取，从而为每个代码片段生成唯一的“指纹”，该指纹能够有效地用于识别代码片段之间的相似性或重复性。通过这种方式，不同的代码片段可以被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，本文选择采用哈希算法作为提取代码指纹的方法，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。

3.3.2 基于 ClaSP 算法的代码克隆检测方法

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。本文利用数据挖掘算法，提取代码克隆序列。

（1）数据挖掘领域基本概念

数据挖掘领域中有一些基本概念：

项集[?]：设有一个集合 $I = \{i_1, i_2, \ldots, i_n\}$，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。

序列[?]：序列是一个由项集组成的有序列表，记作 $s = (s_1, s_2, \ldots, s_n)$，其中每个 s_i 是项集。每个项集 s_i 可以进一步表示为 $(x_1, x_2, \ldots, x_m)$，其中 x_m 是项。

子序列[?]：考虑两个序列 $\alpha = (a_1, a_2, \ldots, a_n)$ 和 $\beta = (b_1, b_2, \ldots, b_m)$，如果存在一组整数 $1 \le j_1 < j_2 < \cdots < j_n \le m$，使得序列 $(a_{j_1}, a_{j_2}, \ldots, a_{j_n})$ 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[?]：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。

频繁子序列[?]：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，即 $\text{support}(\alpha) \ge \text{min\_support}$，则称序列 α 为频繁子序列。所有频繁子序列的集合记为 FS。

序列支持度：记作 $\sigma(\alpha, D)$，是指在输入数据库 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。FS 是所有频繁序列的集合。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。

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。否则，如果一个频繁序列 β 存在一个超序列 γ，使得它们具有完全相同的支持度，则称 β 为非闭合序列，并且 γ 吸收 β。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，表为序列数据库示例，这里设定最小支持度阈值为 2。

表 3-1 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。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。

本文中，克隆代码的指纹是一个项，每个文件是一个序列，所有文件作为序列数据库，在数据库上运行 ClaSP 算法挖掘所有的频繁项集，即克隆代码。

（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

算法 3-1 ClaSP 算法

Input: 序列数据库

Output: 频繁闭合序列集 FCS

$F_1 \leftarrow$ {频繁 1-序列}

$FCC \leftarrow \emptyset, FCS \leftarrow \emptyset$

for all $i \in F_1$ **do**

  $F_{ie} \leftarrow$ {频繁 1-序列的长大于 i 的扩展序列}

  $FCC_i \leftarrow$ DFS-PRUNING(i, F_1, F_{ie})

  $FCC \leftarrow FCC \cup FCC_i$

end

$FCS \leftarrow$ N-ClosedStep(FCC)

首先找到所有的频繁的 1-序列（即长度为 1 的序列），然后，对于所有频繁的 1 序列，递归地调用 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 中出现的非封闭序列。

算法 3-2 DFS-Pruning 算法伪代码

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 ← ∅ Itemp ← ∅ F_i ← ∅ P_s ← ∅ P_i ← ∅
2  if ¬checkAvoidable(p, I(D_p)) then
3      for all i ∈ S_n do
4          if p' = (s_1, s_2, ..., s_n, {i}) is frequent then
5              Stemp ← Stemp ∪ {i}
6              P_s ← P_s ∪ {p'}
7          end
8      end
9      F_i ← F_i ∪ P_s ∪ ExpSiblings(P_s, Stemp, Stemp)
10     for all i ∈ I_n do
11         if p' = (s_1, s_2, ..., s_n ∪ {i}) is frequent then
12             Itemp ← Itemp ∪ {i}
13             P_i ← P_i ∪ {p'}
14         end
15     end
16     F_i ← F_i ∪ P_i ∪ ExpSiblings(P_s, Stemp, Itemp)
17 end
18 return F_i
```

在递归遍历过程中，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，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，实际上这些克隆关系表现为碎片化的形式。为了恢复代码的完整性，进一步对这些碎片进行合并。具体而言，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，提取出的这些克隆关系实际反映和代表了方法之间的变更影响关系。

3.4 基于数据挖掘的变更影响分析

3.4.1 代码变更历史提取

在软件工程中，代码变更历史的分析是理解软件演化、优化代码质量及进行维护的重要手段之一。传统的基于依赖关系闭包的方法能够识别方法之间的浅层变更影响关系，主要侧重于语法层面和直接的依赖关系。然而，这种方法无法揭示更深层次的关系，如方法间的潜在影响和变更传播。因此，本文设计了一种基于数据挖掘技术的变更影响分析方法，主要的挖掘对象是软件项目的变更历史。该方法能够更全面、深刻地提取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。

该方法的核心假设是：在代码变更历史中，频繁同时更改的代码片段或方法对，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语法上的依赖，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，我们可以揭示出更深层的变更关系，为后续的代码优化、重构以及维护工作提供有力支持。

本方法的具体实现过程可以分为几个主要步骤，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。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，如2.6.1节所示的6个项目。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），这里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，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。

3.4.2 基于共现关联挖掘的变更影响关系提取

关联规则（Association Rules）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”是两个共现项。

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。

（1）频繁项集评估标准

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，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.

$$Support(funcA, funcB) = \frac{num(AB)}{num(AllCommits)} \tag{3-3}$$

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。

$$Confidence(funcA \Leftarrow funcB) = P(AB)/P(B) \tag{3-4}$$

（2）本文标准设置

本文将置信度设置为 1。具体而言，对于方法 A，记录其在所有提交记录中出现的次数为 N_A，并统计在这些提交中，其他方法 B 出现的次数为 N_B。如果在所有提交中，方法 A 出现时，方法 B 的出现频率达到 $N_A \backslash N_B = 1$ 则认为方法对 (A, B) 存在变更影响关系。换句话说，置信度为 1 表示每当方法 A 被修改时，方法 B 也必然随之修改，从而确认这两个方法的变更是紧密关联的，并且它们的修改行为是同步的。

本文将支持度设置为 2，意味着方法 A 和方法 B 在多个提交中同时出现的

次数必须至少为 2 次，才能认为这两个方法之间可能存在变更影响关系。支持度的设定要求方法对 (A, B) 在变更历史中有一定的共现频率，从而能够确保所识别的变更影响关系具有一定的统计显著性，避免偶然性因素的干扰。

综上所述，本文通过设置置信度为 1 和支持度为 2，旨在确保识别出的变更影响关系具有较高的确定性和可靠性。这一标准有助于捕捉那些在多个提交中频繁共同变更的方法对，从而为进一步的变更影响分析提供有效的依据。

3.5 基于深度学习的变更影响分析

3.5.1 数据集来源和收集

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，恰巧弥补了这一空缺。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 10000 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函数，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。

最终本文选择了基于表 2-1 中列出的 GitHub 项目。这些项目除了符合上述条件外，还有以下优点。首先，它们的收藏数均在千以上，表明这些项目在开源社区中具有较高的关注度和影响力，并且被广泛使用。其次，这些项目的社区非常活跃，说明项目持续更新和迭代，保证了具有较为丰富的变更历史，能够为我们的变更影响关系分析提供丰富的数据支持。

3.5.2 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，输出为一个介于 [0, 1] 之间的值，表示这两个方法之间存在变更影响关系的概率。该概率越接近 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。

首先，将两个方法体分别输入到预训练模型中，通过模型的嵌入层处理，分别生成每个函数体的代码嵌入表示，分别记为 s1 和 s2。接下来，将 s1 和 s2 这两个向量进行拼接融合，并使用特殊的分隔符 token（在 CodeBERT 模型中是 </s>）来区分两个函数体的嵌入信息，从而保证模型能够区分这两个函数体的独立特性。，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，通过非线性变换学习两个函数体之间的深层次关联，得到融合后的特征向量。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。然后将模型的输出与真实标签之间做交叉熵损失来进行优化，以此来调整模型的参数。

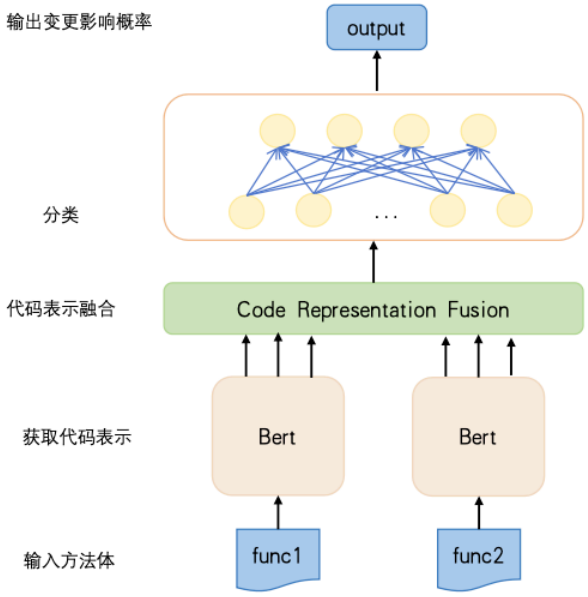

图 3-3 基于代码预训练的深度学习模型架构

3.6 实验结果与分析

3.6.1 实验数据集描述与分析

3.6.2 实验设置

3.6.3 实验结果与分析

对收集到的数据进行分析，这里以真实项目中挖掘到的方法对为例，如图 2-6 中所示，这个项目是 kafka 的一个 C/c++ 客户端。方法一的主要功能是在 Kafka 模拟环境中按照一定的规则生成 Producer ID，而方法二的功能是检查一个 Producer ID 是否有效。当涉及检查的逻辑时，往往需要考虑到它是如何被创建和分配的。在这个例子中需要确保 ID 的创建逻辑与检查逻辑一致，才能确保检查是正确的。这个方法中，对于要检查的 Producer ID，首先会判断事务 ID，然后再检查 epoch，逻辑与创建 ID 是形成对应的。考虑修改的行为，如果对于 Producer ID 的创建和分配逻辑进行了修改，那么通常需要同时修改 Producer ID 检查逻辑，以确保两者保持一致。否则，如果创建和分配逻辑发生了变化，而检查逻辑没有相应地进行调整，就可能导致检查逻辑无法正确地验证新生成的 Producer ID，从而引入错误和不一致性。

实验结果如表 2-2 所示。实验使用的数据集总共 6000 对左右，按照训练、验证、测试集为 6:2:2 ，分别训练了 codebert 系列的两个网络 CodeBERTa-small-v1

和 codebert-base-mlm，具体结果如下：从总体上来看，两个模型的性能均表现出色，但深入分析召回率（recall）和精确率（precision）时，可以发现两个模型各有所长。CodeBERTa-small-v1 模型更加重视精确率，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，但这种方法可能会导致一些正确的情况被漏掉，即存在漏报的情况。由于”small” 模型的规模较小，参数数量较少，这可能会使得模型在训练过程中更容易发生过拟合现象。

另一方面，CodeBERT-base-mlm 模型更加注重召回率，致力于捕捉尽可能多的正样本。这种策略虽然能够覆盖更多的正例，但在这个过程中，也可能会引入一些误报，即将一些实际上是负例的情况错误地判定为正例。大型模型由于具备更强的泛化能力，通常能够更好地适应新的数据集。因此，为了获得更好的泛化性能，它们通常会牺牲一定的精确率。

```
rd_kafka_pid_t rd_kafka_mock_pid_new(rd_kafka_mock_cluster_t *mcluster,
                     const rd_kafkap_str_t *TransactionalId) {
        //计算长度，分配内存
        size_t tidlen = TransactionalId ? RD_KAFKAP_STR_LEN(TransactionalId) : 0;
        rd_kafka_mock_pid_t *mpid = rd_malloc(sizeof(*mpid) + tidlen);
        rd_kafka_pid_t ret;
        //生成一个随机的 Producer ID、设定epoch
        mpid->pid.id    = rd_jitter(1, 900000) * 1000;
        mpid->pid.epoch = 0;
        //将事务ID拷贝到pid里
        if (tidlen > 0)  memcpy(mpid->TransactionalId, TransactionalId->str, tidlen);
        mpid->TransactionalId[tidlen] = '\0';
        //新生成的 Producer ID 添加到模拟集群中的 PID 列表
        rd_list_add(&mcluster->pids, mpid);
        ret = mpid->pid;
        //返回
        return ret;
}

rd_kafka_resp_err_t rd_kafka_mock_pid_check(rd_kafka_mock_cluster_t *mcluster,
                        const rd_kafkap_str_t *TransactionalId,
                        const rd_kafka_pid_t check_pid) {
        rd_kafka_mock_pid_t *mpid;
        rd_kafka_resp_err_t err = RD_KAFKA_RESP_ERR_NO_ERROR;
        //检查TID、epoch
        err = rd_kafka_mock_pid_find(mcluster, TransactionalId, check_pid, &mpid);
        if (!err && check_pid.epoch != mpid->pid.epoch)  err = RD_KAFKA_RESP_ERR_INVALID_PRODUCER_EPOCH;
        //检查失败
        if (unlikely(err))  rd_kafka_dbg(mcluster->rk, MOCK,...);
        return err;
}
```

图 3-4 逻辑上有变更影响关系的方法对示例

经过分析可以看出，通过数据挖掘方法得到的具有变更关系的方法对，是具有一定的准确性的，并且可以弥补静态分析的不足。

3.7 本章小结

第 4 章　代码审查图生成

4.1　引言

对整个项目的分析结果将以代码审查图的方式展示给用户。其中图的节点是代码元素，包括方法和全局变量，方法和方法以及方法和全局变量之间的关系将以边的形式进行展示。

4.2　基于方法功能标签的方法聚类

4.2.1　基于大语言模型的方法功能标签生成

4.2.2　基于功能标签的方法聚类

4.3　代码审查图

4.3.1　代码审查图构建

4.3.2　代码审查图可视化

图形可视化方案基于开源项目 G6，构建并展示最终的代码审查图。G6 是一个图形可视化引擎，它提供了绘制、布局、分析、交互、动画等全方位的图形可视化基础功能。在图形的节点与边被计算出后，将这些数据以 JSON 格式组织起来，这样可以让 G6 加载这些远程数据源进行展示，同时也实现了计算逻辑与图形可视化的有效分离。

如图 2-8 所示，目前实现了节点的分类、设置节点属性，以及边的分类、设置边信息等关键功能。为了提升交互体验，还增加了鼠标悬停时即可查看节点或边信息的便捷功能。

4.4 实验结果与分析

4.4.1 实验数据集描述与分析

4.4.2 实验设置

4.4.3 实验结果与分析

4.5 本章小结

结　论

学位论文的结论作为论文正文的最后一章单独排写，但不加章标题序号。

结论应是作者在学位论文研究过程中所取得的创新性成果的概要总结，不能与摘要混为一谈。博士学位论文结论应包括论文的主要结果、创新点、展望三部分，在结论中应概括论文的核心观点，明确、客观地指出本研究内容的创新性成果（含新见解、新观点、方法创新、技术创新、理论创新），并指出今后进一步在本研究方向进行研究工作的展望与设想。对所取得的创新性成果应注意从定性和定量两方面给出科学、准确的评价，分（1）、（2）、（3）…条列出，宜用“提出了”、“建立了”等词叙述。

参考文献

[1] SHENEAMER, ABDULLAH, ROY, et al. A detection framework for semantic code clones and obfuscated code[J]. Expert Systems with Application, 2018.

[2] 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.

[3] SHAHRIAR H, ZULKERNINE M. Mitigating program security vulnerabilities: Approaches and challenges[J]. ACM Computing Surveys, 2012, 44(3): 1-46.

[4] BASET A Z, DENNING T. Ide plugins for detecting input-validation vulnerabilities [J]. IEEE, 2017.

[5] BALOGLU B. How to find and fix software vulnerabilities with coverity static analysis[C]//Cybersecurity Development. 2016.

[6] SPANOS G, ANGELIS L, TOLOUDIS D. Assessment of vulnerability severity using text mining[J]. ACM, 2017: 1-6.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] NAKAMURA M, HAMAGAMI T. A software quality evaluation method using the change of source code metrics[J]. IEEE, 2012.

[12] MISRA S, AKMAN I, COLOMO-PALACIOS R. Framework for evaluation and validation of software complexity measures[J]. Iet Software, 2012, 6(4): 323-334.

[13] MISRA S, KOYUNCU M, CRASSO M, et al. A suite of cognitive complexity metrics[C]//International Conference on Computational Science Its Applications. 2012.

[14] CONCAS G, MARCHESI M, MURGIA A, et al. Assessing traditional and new metrics for object-oriented systems[J]. ACM, 2010.

[15] 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.

[16] 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.

[17] 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.

[18] 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).

[19] LI W, HENRY S. Object-oriented metrics that predict maintainability[J]. Journal of Systems Software, 1993, 23(2): 111-122.

[20] CHEN W, IQBAL A, ABDRAKHMANOV A, et al. Large-scale enterprise systems: Changes and impacts[J]. lecture notes in business information processing, 2013.

[21] ARNOLD R S. Software change impact analysis[M]. Washington, DC, USA: IEEE Computer Society Press, 1996.

[22] 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.

[23] GALLAGHER K B, LYLE J R. Using program slicing in software maintenance[J]. IEEE Transactions on Software Engineering, 1991, 17(8): 751-761.

[24] JITENDERKUMARCHHABRA, MRINAALMALHOTRA, JITENDERKUMARCHHABRA, et al. Improved computation of change impact analysis in software using all applicable dependencies[J]. Springer, Singapore, 2018.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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).

[30] 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.

[31] 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.

[32] 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.

[33] CAI H, THAIN D. Distia: a cost-effective dynamic impact analysis for distributed programs[C]//the 31st IEEE/ACM International Conference. 2016.

[34] BASRI S, KAMA N, ADLI S, et al. Using static and dynamic impact analysis for effort estimation[J]. Iet Software, 2016, 10(4): 89-95.

[35] 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.

[36] 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.

[37] ZANJANI M B, SWARTZENDRUBER G, KAGDI H. Impact analysis of change requests on source code based on interaction and commit histories[C]//ACM. 2014.

[38] 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.

[39] DIT B, WAGNER M, WEN S, et al. Impactminer: a tool for change impact analysis [J]. ACM, 2014: 540-543.

[40] 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.

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《面向代码质量评估的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

衷心感谢导师 XXX 教授对本人的精心指导。他的言传身教将使我终生受益。

……

感谢哈工大 LaTeX 论文模板 hiThesis !