小 型 微 型 计 算 机 系 统
Journal of Chinese Computer Systems
DOI:10.20009/j.cnki.21-1106/TP.2024-0196
2025 年 6 月 第 6 期
Vol.46 No.6 2025

Java 第三方库依赖冲突自动化修复框架

毛祥煜,施游堃,张 源

(复旦大学 计算机科学技术学院,上海 200438)
E-mail:xymao21@m.fudan.edu.cn

摘 要: 现代 Java 项目通常依赖于数量庞大的开源第三方库,导致依赖冲突问题频繁出现,影响项目稳定性.由于依赖冲突问题成因复杂,人工修复成本极高,但现有工作仅能为开发者提供有限的依赖冲突修复建议,且存在策略单一的缺陷,在面对复杂的项目依赖关系时难以降低修复成本.对此,本文实现了一个基于多样化修复策略的 Java 第三方库依赖冲突自动化修复框架 DCSolver,利用新增与移除依赖关系边等多种操作灵活调整项目依赖结构,构建依赖冲突问题的修复方案,并完成自动化部署.实验结果表明,DCSolver 能够高效地帮助开发者修复依赖冲突问题,同时保证项目兼容性与修复方案的轻量性.

关 键 词: Java;第三方库;依赖管理;依赖冲突;兼容性分析
中图分类号: TP311 **文献标识码:** A **文 章 编 号:** 1000-1220(2025)06-1458-07

Automated Resolution Framework for Dependency Conflict of Java Third-party Libraries

MAO Xiangyu, SHI Youkun, ZHANG Yuan
(School of Computer Science, Fudan University, Shanghai 200438, China)

Abstract: Modern Java projects often rely on a vast number of open-source third-party libraries, leading to frequent occurrence of dependency conflicts that affect project stability. Due to the multifaceted origins of dependency conflicts, manual resolution incurs extremely high costs. Nevertheless, existing approaches only offer developers limited suggestions for resolving dependency conflicts, and they suffer from the limitation of single-minded strategies, making it difficult to reduce the cost of resolution when facing complex project dependency relationships. In this regard, this paper proposes DCSolver, an automated resolution framework for dependency conflicts of Java third-party libraries based on diversified strategies. By employing various operations such as adding and removing dependency relationships, DCSolver can flexibly adjusts project dependency structures, constructs resolutions for dependency conflicts, and completes automated deployment. Experimental results demonstrate that DCSolver efficiently assists developers in resolving dependency conflicts while ensuring project compatibility and lightweight resolutions.

Keywords: Java; third-party library; dependency management; dependency conflict; compatibility analysis

0 引 言

随着 Java 项目多样性的不断增加,越来越多的开发者选择借助开源第三方库(third-party library)实现代码复用,以减少冗余工作量,提高项目开发效率和可维护性.由于第三方库之间存在依赖关系,项目在使用依赖库时通常会引入大量的传递依赖库,增加了依赖管理的复杂性.据相关调查[1]显示,单个 Java 项目平均依赖于 148 个第三方库,在如此庞大而错综复杂的依赖关系中,极易出现同一个库的多个不同版本被同时引入的情况,即依赖冲突问题(dependency conflict).当不同版本间存在代码差异时,依赖冲突问题可能导致项目运行时出现异常.因此,及时解决第三方库依赖冲突问题,对于维护项目稳定性至关重要.

在真实场景下的大型项目中,依赖冲突问题通常表现隐蔽、根因复杂,修复时涉及对依赖关系或源代码的调整,但开发者仅能控制项目本身的依赖配置文件,对间接依赖关系的控制能力较弱,限制了调整能力;同时,存在依赖冲突问题的第三方库往往又与其他库存在关联,修复依赖冲突时可能对项目依赖关系产生间接影响,进而破坏兼容性.在这些因素的共同作用下,人工修复依赖冲突问题将消耗大量分析成本,影响修复效率.举例如 Hadoop 项目中的一个依赖冲突问题[2],在被提出 879 天后内才得到完全解决.

前人工作[3-6]多聚焦于依赖冲突问题的检测与定向测试用例构建,或提供简单的版本调整等修复策略建议.这些工作的主要问题在于不支持为开发者提供完整的、可直接应用的依赖冲突修复方案,且考虑的修复策略较为单一,难以处理实际场景中复杂的项目依赖关系.故多数情况下,项目开发者仍需自行分析修复方案,这可能导致依赖冲突问题修复滞后,长期影响项目稳定性.

为提供便捷、易用的依赖冲突解决方案,帮助开发者高效地修复项目中的依赖冲突问题,本文提出了一种基于多样化策略的 Java 依赖冲突修复方法,并实现了面向 Maven 项目的

收稿日期:2024-03-29 收修改稿日期:2024-05-10 基金项目:国家自然科学基金项目(62172105)资助;上海市青年科技启明星计划项目(21QA1400700)资助;上海市基础研究特区计划项目(21TQ1400100:21TQ012)资助. 作者简介:毛祥煜,男,1999 年生,硕士,研究方向为系统软件与安全;施游堃,男,1997 年生,博士,研究方向为系统软件与安全;张 源,男,1987 年生,博士,教授,CCF 会员,研究方向为系统软件与安全.

依赖冲突问题自动化修复框架 DCSolver. 其核心思路为引入新增、移除依赖关系边等策略,与已有的调整版本策略相结合,在仅需改动项目依赖配置文件的前提下,实现对依赖关系结构的灵活调整,进而修复依赖冲突问题. 在此基础上,为避免修复方案破坏项目兼容性,或引入过于复杂的改动而影响依赖管理,DCSolver 利用预定义的原子化修改操作构建修复方案,并利用影响分析与依赖结构迭代调整等算法进一步优化修复方案. 基于真实项目的评估结果显示,DCSolver 能够在较短的分析时间内以轻量级方案有效地修复所有依赖冲突问题,同时未对项目兼容性产生影响.

1 背景与相关工作

1.1 Maven 依赖管理与依赖冲突问题

以 Maven 和 Gradle 为代表的 Java 依赖管理工具提供了诸多依赖管理机制,并基于以 Maven Central 为代表的公共存储仓库简化了开源第三方库的维护、分发和使用流程. 其中以 Maven 最为流行,以下对其依赖管理机制进行介绍.

Maven 通过 pom. xml 配置文件管理项目依赖,为了在项目中引入指定的库,开发人员需要通过 group、artifact 和 version 三元组指定依赖库的唯一标识符,Maven 将自动下载库文件并将其配置为项目依赖. 基于 pom. xml 文件中声明的所有依赖关系,Maven 在构建项目时将执行递归解析,获取项目直接依赖的所有库,以及这些库所需的传递依赖库. 在多模块项目中,Maven 支持子模块直接继承父模块中指定的依赖关系,实现依赖库的集中化管理.

由于不同的第三方库之间存在复杂的依赖关系,在默认引入所有传递依赖库的机制下,可能出现多个不同版本的同一依赖库共存于项目中,导致依赖冲突问题. 对此 Maven 遵循近者优先策略,从依赖关系图中选择与项目根节点具有最短路径的依赖项版本,并遮蔽该依赖项的其他版本. 然而,该策略并不能保证所选版本符合项目需求,若项目在运行时引用了被遮蔽版本中独有的类、方法、字段等代码元素,将引发 NoClassDefFoundError 或 NoSuchMethodError 等运行时异常. 依赖冲突问题通常不会在项目构建时导致编译错误,问题表现较为隐蔽,这进一步增加了人工分析与解决成本.

1.2 相关工作

作为开源软件生态中的常见问题之一,Java 依赖冲突受到越来越多研究者的关注. Wang[3] 等人提出了依赖冲突检测工具 Decca,通过分析项目依赖关系识别冲突的第三方依赖库或代码类,并推导加载与隐藏的方法集合,结合项目引用的方法集合评估依赖冲突问题的严重性; RIDDLE[4] 采用基于遗传算法的搜索策略,生成可触发依赖冲突运行时异常的定向测试用例;Sensor[5] 则基于对象构造函数和调用上下文信息,构建测试用例触发与冲突相关的语义不兼容问题. 这些工具能够为开发者感知依赖冲突问题与风险提供一定的帮助,但其局限性在于未提供更加通用的依赖冲突自动化修复方案支持,开发者仍需花费较高的成本自行完成修复。而本文提出的 DCSolver 框架可在检测依赖冲突问题基础上,自动化分析冲突修复方案并完成部署,进而显著降低依赖冲突问题的修复成本. 此外,Huang 等人提出的 LibHarmo[6] 工具可检测项目中版本不一致的依赖库声明,并分析统一其版本所需的代码改动开销,依据最小改动原则推荐库版本. 但该工具存在修复策略单一的缺陷,难以应对实际场景下的复杂项目依赖关系. 而本文的 DCSolver 框架引入了新增、移除依赖关系等多样化修复策略,极大增加了灵活性与适用性.

在其他编程语言中,也存在众多与依赖冲突检测与修复技术相关的研究工作. 例如针对 Python 生态系统的依赖冲突监测工具 Watchman[7],其思路为根据第三方依赖库元数据进行广度优先搜索,从中提取不同依赖库的版本约束进行交集分析;Wang 等人提出了工具 SmartPip[8],利用预先构建的依赖关系知识库求解版本约束,从而克服新版本 PyPI 依赖解析策略在处理依赖冲突时的不准确性;ConflictJS[9] 则致力于解决 JavaScript 依赖冲突问题,通过动态匹配各个依赖库对命名空间的写入操作来检测依赖冲突. 由于不同语言的依赖管理机制与依赖冲突问题的成因均存在差异,针对其他语言的依赖冲突检测与修复技术无法直接迁移至 Java 语言项目. 本文提出的 DCSolver 框架针对 Java 生态系统中最为常用的 Maven 工具,通过深入分析其依赖管理机制,设计合理的依赖冲突检测与修复方法,可较好地解决 Java 依赖冲突问题的修复滞后性.

2 系统设计与实现

为帮助开发者高效地修复项目中的依赖冲突问题,本文设计了一种基于多样化策略的 Java 依赖冲突自动化修复框架 DCSolver. 本章将对其设计与实现进行介绍.

2.1 基本思路与研究挑战

根据已有工作[3-5,10] 可知,开发者对于依赖冲突问题采用的人工修复方式可主要分为修改依赖配置文件与修改项目代码两类方式. 考虑到自动化修改代码难以保证可靠性,本文选择基于修改依赖配置文件的修复方式. 采用该方式的主要难点在于,开发者无法控制传递引入的依赖关系,导致多数情况下难以直接调整冲突依赖库的版本. 本文观察发现,在基于 Maven 的项目依赖管理中,开发者常用的依赖配置文件改动方式除调整依赖库版本外,还包括新增依赖库、删除依赖库、排除传递依赖库等策略,可将其抽象为针对依赖关系边的增、删操作. 利用这些策略与已有的调整版本策略相结合,可极大地增强对依赖冲突的修复能力.

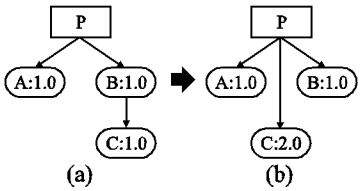

图 1 依赖关系调整

Fig. 1 Adjustment of dependency relationship

以图 1 所示的依赖关系为例,为调整依赖库 C 的版本,可在项目 P 中排除传递依赖项后,重新添加依赖库 C. 这一方法看似将破坏 B 与 C 之间的依赖关系,实际上利用了 Java 的依赖加载机制:在项目运行前,所有库依赖的相关文件均被加载至项目类路径中,Java 虚拟机在运行时将在类路径中进行

查询,而非在依赖关系图中进行查询. 当涉及到依赖库之间的调用时,仍然遵循相同的查询机制,故在调整后,依赖库 B 在调用依赖库 C 时仍然能在类路径中找到方法定义,从而实现了对传递依赖关系的调整.

基于此思路,本文实现了面向 Maven 项目的 Java 第三方库依赖冲突自动化修复框架 DCSolver. 在使用上述策略自动化修复依赖冲突问题时,仍存在两方面的挑战:

兼容性挑战. 调整版本操作通常仅影响单个依赖库,但新增或移除单个依赖关系边带来的变化可能沿依赖关系图进一步传播,对项目产生潜在的兼容性影响. 因此在修复依赖冲突问题时,需进行完备的兼容性分析,避免修复后破坏项目正常功能. 对此,本文将利用影响分析算法,确定修复策略对项目的兼容性影响并进行筛选.

复杂性挑战. 在利用多样化修复策略解决依赖冲突时,直观思路是将所有传递依赖库直接引入,杜绝依赖冲突. 但该思路需要大幅度调整依赖关系,实际上与 Maven 的便捷依赖管理相悖,将降低开发者的接受度. 因此,依赖冲突问题的修复方案需避免过于复杂的改动. 对此,本文将采用导向性的结构调整方式,使用尽可能少的操作实现对依赖冲突问题的修复.

2.2 总体架构概览

DCSolver 的整体架构如图 2 所示,根据功能可将其分为预处理、修复方案构建和修复方案应用模块. 各个模块的简要功能介绍如下:

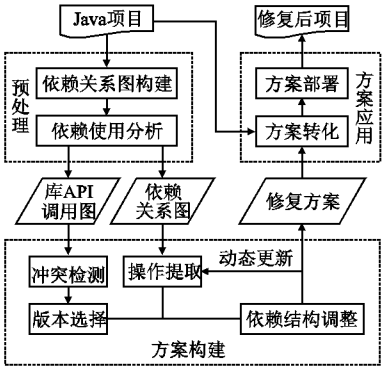

图 2 DCSolver 架构图

Fig. 2 Architecture of DCSolver

1)*预处理模块*:构建依赖关系图,并执行依赖使用分析,为后续模块提供分析参考;

2)*方案构建模块*:检测项目中的所有依赖冲突问题,并确定可用的修复版本;从依赖关系图中提取可用的原子化修改操作并进行组合,将冲突依赖库统一为修复版本,构建得到依赖冲突问题的修复方案;

3)*方案应用模块*:将修复方案转化为对项目依赖配置文件的修改操作序列并自动化部署,完成修复流程.

2.3 预处理

预处理模块分别以项目依赖配置文件 pom. xml 和项目代码编译后得到的字节码文件为输入,执行依赖关系图构建、依赖使用分析两项任务.

2.3.1 依赖关系图构建

DCSolver 使用有向无环图 $DG(V,E)$ 存储项目的完整依赖关系,V 为图中的节点集合,用于表示项目与依赖库,即 $V=P\cup D$,其中 $P=\{p_1,p_2,\cdots,p_n\}$ 代表项目,或多模块项目中的各个模块;$D=\{d_1,d_2,\cdots,d_n\}$ 则表示项目的所有依赖库,根据三元组信息进行区分. E 表示图中的有向边集合,用于表示继承关系与依赖关系,即 $E=\{(u,v)\mid u,v\in V\}$,其中有向边(u,v)代表3 种含义:1)项目不同模块间的继承关系边;2)项目与库之间的依赖关系边;3)库与库之间的依赖关系边.

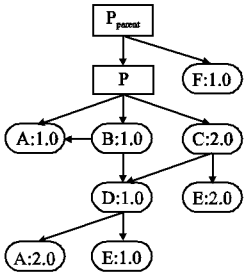

图 3 依赖关系图示例

Fig. 3 Example of dependency graph

在实现上,DCSolver 基于 Maven Dependency Graph[11] 工具,解析项目的依赖配置文件,提取其中的模块继承声明与依赖库声明,并向上递归解析父模块的配置文件、向下递归解析依赖库的配置文件,构建如图 3 所示的完整依赖关系图(其中依赖库节点略去了 group 信息). 后续模块将以此为基础,检测依赖冲突问题并分析修复方案.

2.3.2 依赖使用分析

依赖使用分析的目标为获取项目对各个依赖库的具体使用信息,进而在后续流程中判断是否影响项目正常功能,保证修复方案的兼容性. 考虑到 API 调用为项目对依赖库的常见使用方式,可基于全局调用图确定项目直接或间接调用的依赖库 API. 具体实现上,DCSolver 首先以项目和依赖库的字节码文件为输入,基于 Soot[12] 框架提供的类层次分析(CHA)算法构建调用图. 为获取被调用的库 API,DCSolver 以项目中的方法为入口点执行可达性分析,并将所有可达的库 API 与其所属依赖库节点进行映射.

分析优化. 出于增强分析能力与提高分析效率等因素考虑,DCSolver 额外实现了如下优化:1)在调用图中删除对项目方法或 Java 标准库 API 的调用边,提高可达性分析效率;2)为处理 Java 中常见的反射调用等动态特性,额外执行常量分析步骤,基于字节码文件中的常量池与动态特性相关 API(如 method. invoke)进行建模,提取潜在的库 API 调用,使调用图更加完整;3)收集所有调用点与目标方法位于不同依赖库内的方法调用,便于在依赖关系图变化时,快速执行相应的调用图动态更新.

2.4 方案构建

方案构建模块基于依赖关系图与库 API 调用图,为项目构建依赖冲突问题修复方案. 为便于介绍模块的整体分析流程,以下首先给出依赖关系图中的若干概念定义:

定义 1. 依赖路径

给定依赖关系图 $DG(V,E)$,v_i 与 v_j 为 V 中的两个不同节点,则从 $v_i\sim v_j$ 的一条依赖路径可定义为 $Path=(v_i,v_{i+1},\cdots,v_{j-1},v_j)$,其中 $v_i,v_{i+1},\cdots,v_{j-1},v_j$ 为路径上互不相同的节点,

且对于所有 $i \leqslant k < j$,(v_k, v_{k+1}) 为 E 中的依赖关系边. 依赖路径中包含的有向边数量称为该路径的长度. 在复杂的依赖关系图中,节点 v_i 与 v_j 之间可能存在多条依赖路径.

定义 2. 最短距离

记 PathSet(v_i, v_j) 为节点 v_i 与 v_j 之间的所有依赖路径集合,本文将最短距离 ShortestDis(v_i, v_j) 定义为 PathSet(v_i, v_j) 中所有路径长度的最小值. 若节点 v_i 与 v_j 之间不存在依赖路径,则 ShortestDis(v_i, v_j) 取值为无穷大. 最短距离体现了项目对该依赖库的依赖程度,Maven 加载依赖时的近者优先原则,实际上也等价于优先选择最短距离较小的依赖库节点.

定义 3. 直接依赖与间接依赖

考虑依赖关系图中的依赖库节点 d,由依赖解析机制易知其必然与某个项目节点 p 之间存在至少一条依赖路径. 若 ShortestDis$(p, d) = 1$,则称 d 为直接依赖,反之为间接依赖. 后文中分别以 D_{direct} 和 $D_{transitive}$ 表示依赖关系图中的直接依赖库集合与间接依赖库集合.

2.4.1 冲突检测

DCSolver 首先在依赖关系图中根据继承规则,确定当前项目下的所有依赖库节点,并从中检测同时存在多个版本的依赖库节点,即三元组信息中 group、artifact 相同但 version 不同的节点集合,将其作为待解决的依赖冲突问题,这些节点被称为与依赖冲突问题相关的冲突节点.

若检测到多个依赖冲突问题,DCSolver 将基于依赖关系图进行拓扑排序,优先处理对项目影响较大的依赖冲突问题. 具体而言,DCSolver 考虑与依赖冲突问题相关的冲突节点,按照以下标准从未排序的依赖冲突问题中优先选取:不存在从其他依赖冲突问题的冲突节点至当前冲突节点的依赖路径,或冲突节点的最短距离值总和较小.

2.4.2 版本选择

在检测到依赖冲突问题后,需为其确定修复版本,后续以此为目标调整项目依赖关系,构建修复方案. DCSolver 首先将所有的冲突节点版本共同作为修复版本候选,该策略可避免引入额外的依赖库版本,降低方案整体复杂度. 在此基础上,DCSolver 根据依赖使用分析结果,获取项目对当前依赖库的所有 API 调用,将其与各个候选版本的 API 方法实现进行逐一匹配,以判断可满足项目使用需求的库版本. 例如,项目中存在对 int foo(java. lang. String a,int b)这一库方法的调用,则满足调用需求的库版本代码中必须包含方法名称为 foo、返回类型为 int 与参数类型为[java. lang. String,int]的方法实现.

当多个库版本均可满足调用需求时,DCSolver 将根据版本发布先后顺序,选择其中较新的版本作为调整目标版本. 极端情况下,项目同时调用了多个库版本中的独有 API,此时将不存在任何依赖库版本满足调用需求. 对此,DCSolver 利用 Maven Shade Plugin[14] 同时保留多个库版本,从而在不破坏项目兼容性的前提下修复依赖冲突.

2.4.3 操作提取

通过分析软件工程实践中常见的依赖管理与依赖冲突问题的人工修复方式,DCSolver 总结了 4 类针对依赖关系图的原子化修改操作,如表 1 所示. 此外,为便于将修改操作映射至对依赖配置文件的实际修改,需对其适用范围进行限制. 以下对 4 种类型的修改操作及适用范围进行介绍:

表 1 原子化修改操作定义

Table 1 Definition of atomic edit operations

操作名称	适用范围规则	操作内容
EditVersion(v, ver)	$v \in D_{direct}$	$v.\ version \rightarrow ver$
Rename(v)	$v \in D$	$v.\ group \rightarrow$ "shaded"
AddEdge(v_1, v_2)	$v_1 \in P$ $v_2 \in D_{transitive}$	$E \rightarrow E \cup (v_1, v_2)$
RemoveEdge(v_1, v_2)	$v_1 \in P \cup D_{direct}$ $v_2 \in D$ $(v_1, v_2) \in E$	$E \rightarrow E \setminus (v_1, v_2)$

· 调整版本 EditVersion(v, ver):此操作修改依赖库节点的版本信息,调整的目标版本并非随意指定,需基于现有依赖关系图进行选择. 在 Maven 项目中,开发者仅能修改项目的配置文件,而无法直接修改依赖库的配置文件,故该操作适用范围被限制为所有直接依赖库节点.

· 依赖库重命名 Rename(v):此操作基于 Maven Shade Plugin[14] 实现,针对依赖库节点修改其名称,主要用于保留同一依赖库的不同版本. 其适用于所有依赖库节点,但可能增加后续依赖管理的复杂性,故使用时应慎重考虑.

· 新增依赖关系边 AddEdge(v_1, v_2):此操作将添加额外的依赖关系边,在适用范围上,依赖关系边的起始节点需为项目节点、目标节点需为已有的间接依赖库节点,这可避免引入额外依赖库节点或带来冗余的依赖关系.

· 移除依赖关系边 RemoveEdge(v_1, v_2):此操作将移除依赖关系图中原有的依赖关系边. 由于 Maven 支持排除传递依赖项,所有从项目节点或直接依赖节点出发的依赖关系边均可被移除,相较于新增操作的适用范围更加广泛.

DCSolver 将首先根据以上定义,扫描依赖关系图并记录其中所有符合规则的修改操作. 为解决兼容性挑战,DCSolver 基于影响分析算法,筛选出所有执行后不影响项目正常功能的修改操作,即可保证后续由这些操作组合得到的修复方案同样不会破坏项目兼容性.

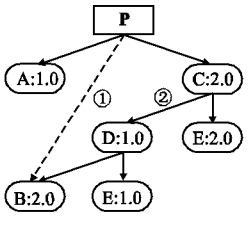

图 4 影响分析示例

Fig. 4 Example of impact analysis

影响分析算法的工作原理如下:对于每个原子修改操作,DCSolver 计算其执行后的依赖关系图,同步更新各个依赖库节点的最短距离,结合 Maven 在加载依赖库时的近者优先策略,确定所有加载版本发生变化的依赖库. 举例而言,在如图 4 所示的依赖关系图中,新增依赖关系边①的操作仅改变依赖库节点 B 的最短距离,未对依赖库的加载版本产生影响;移除依赖关系边②的操作则将使得多个库的加载版本为空,相较于原有版本发生变化.

之后,DCSolver 将基于 GumTree[13] 工具,针对变化前后

版本的依赖库代码执行抽象语法树层级的差异分析,获取版本变化后被移除、方法签名或方法实现改变的所有 API. DCSolver 在差异分析中采取保守策略,当不同版本的 API 方法签名相同但方法内部实现不一致时,同样认为 API 发生变化. 结合依赖使用分析结果,DCSolver 即可判断发生变化的库 API 是否被项目调用,进而判断修改操作是否将影响项目兼容性,排除破坏兼容性的修改操作. 在执行影响分析后,操作提取模块将输出一个可兼容修改操作集合,用于后续调整依赖关系图结构.

2.4.4 依赖结构调整

DCSolver 依照 2.4.1 节中的顺序逐个处理依赖冲突问题,利用可兼容修改操作在依赖关系图中进行结构调整,得到依赖冲突修复方案. 在调整时,DCSolver 遵循最小化改动原则,使修复方案更加轻量易用,解决复杂性挑战.

迭代调整. 对于一个待解决的依赖冲突问题,DCSolver 关注其冲突节点中非修复版本的节点,记为目标节点 d. 若 d 为直接依赖,可直接调整其版本;若 d 为间接依赖,则需要首先将其转化为直接依赖,但若直接在项目节点 p 与 d 之间新增依赖关系边,将导致调整版本时引入额外的依赖冲突问题. 因此,DCSolver 考虑 d 的最短距离属性,以最少的修改操作进行转化. 具体而言,DCSolver 利用图遍历算法获取从项目节点至目标节点的所有依赖路径集合 PathSet(p,d),从中选择最短距离对应的依赖路径. 之后,DCSolver 采取启发式思路迭代缩短该路径,实现转化.

图 5 为迭代调整过程的具体示例. DCSolver 首先寻找路径中的直接依赖节点 d_1 及其后继节点 d_2,之后利用新增操作 AddEdge(p,d_2)与移除操作 RemoveEdge(d_1,d_2)的组合,将 d_2 转化为直接依赖,同时将 d_1 从路径中移除. 重复此迭代过程,即可将 d 转化为直接依赖. 若可兼容修改操作集合中不存在相应的修改操作,DCSolver 将迭代其他依赖路径. 完成迭代后,DCSolver 遍历 p 与 d 之间的其他依赖路径,移除这些路径中与 d 相连的依赖关系边,避免引入额外的依赖冲突问题.

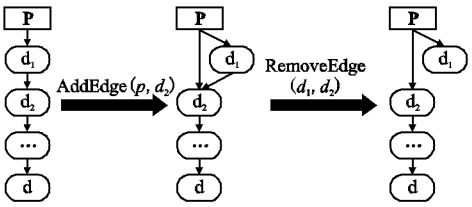

图 5 依赖路径迭代过程

Fig. 5 Iteration process of dependency path

最后,DCSolver 利用调整版本操作,完成对 d 的调整. 考虑到修改操作将改变项目依赖关系图的结构,为保证修复方案的全局准确性,避免后续方案部署时出现无法被执行的修改操作,DCSolver 将在单个目标节点调整完成后,根据选择的修改操作同步更新整体依赖关系图、库 API 调用图与可兼容修改操作集合. 在更新时,DCSolver 采取差分思路,仅对修改操作所影响的依赖关系边与过程间调用进行更新,降低分析开销.

额外优化. 为进一步降低修复方案的整体复杂性,DCSolver 在调整依赖结构时,将执行以下两类额外优化:

1)若使用分析结果表明,目标节点 d 未被项目所调用,DCSolver 将沿所有依赖路径移除连接 d 的依赖关系边,排除该节点及其传递引入的所有子节点,精简项目依赖关系.

2)对于需利用依赖库重命名操作进行修复的依赖冲突问题,DCSolver 将跳过迭代调整过程,直接在修改操作序列中为目标节点添加重命名操作,这有助于进一步降低修复方案的复杂性. 在完成所有依赖冲突问题的处理后,依赖结构调整模块将输出由若干修改操作构成的有序列表,作为依赖冲突问题的修复方案.

2.5 方案应用

在分析得到依赖冲突问题修复方案后,方案应用模块将其自动化部署至项目依赖配置文件中,完成修复流程. 对于修复方案中的各个修改操作,DCSolver 根据其作用的依赖库节点,定位需修改的项目依赖配置文件与依赖库声明. 若当前项目的依赖配置文件中未包含该依赖库节点声明,DCSolver 将沿依赖关系图中的模块继承关系边,反向搜索确定其声明位置. 之后,DCSolver 根据修改操作的具体类型,在声明位置处执行相应改动,完成修复方案部署. 表 2 展示了部署时的部分特殊改动方式,以下将进行详细介绍.

表 2 配置文件改动示例

Table 2 Modification examples of configuration files

操作类型	修改方式
调整版本	< properties > - < slf4j. version > 1. 7. 30 < /slf4j. version > + < slf4j. version > 1. 7. 36 < /slf4j. version > < /properties > < dependency > < groupId > org. slf4j < /groupId > < artifactId > slf4j-api < /artifactId > < version > ${slf4j. version} < /version > < /dependency >
移除依赖关系边	< dependency > < groupId > org. apache. hadoop < /groupId > < artifactId > hadoop-common < /artifactId > < version > ${hadoop. version} < /version > + < exclusion > + < groupId > commons-logging < /groupId > + < artifactId > commons-logging < /artifactId > + < /exclusion > < /dependency >
依赖库重命名	< plugin > < groupId > org. apache. maven. plugins < /groupId > < artifactId > maven-shade-plugin < /artifactId > < version > 3. 2. 4 < /version > < phase > package < /phase > < goals > < goal > shade < /goal > < /goals > < relocation > + < pattern > org. codehaus. plexus. util < /pattern > + < shadedPattern > org. shaded. plexus. util < /shadedPattern > < /relocation > < /plugin >

对于调整版本操作,考虑到真实项目中开发者常使用 < properties > 等方式对依赖库版本进行集中管理,DCSolver 将遵循项目原有的版本管理方式不变,在其版本定义的实际

位置进行改动,同时分析改动是否使得项目中其他依赖库的版本发生变化,将相应信息反馈至项目开发者.

对于移除依赖关系边操作,在需要移除间接依赖项时,DCSolver 定位将其引入的直接依赖项,并利用 <exclusion> 元素实现排除间接依赖.

对于依赖库重命名操作,DCSolver 将遵循最小化修改原则,将冲突版本中独有的部分代码类进行重命名.这些类将在项目编译过程中与项目源代码进行同时打包,并在调用时重定向.考虑到该操作可能对后续依赖管理产生副作用,DCSolver 将其反馈至项目开发者进行额外确认.

3 实验与评估

本章将从以下几个研究问题出发,评估 DCSolver 在依赖冲突自动化修复任务上的表现.

研究问题 1. DCSolver 是否能有效地解决项目中的依赖冲突问题? 此问题旨在评估 DCSolver 的整体有效性;

研究问题 2. DCSolver 修复后的项目兼容性如何? 此问题旨在验证 DCSolver 是否能够解决兼容性挑战,即在修复依赖冲突的同时保持项目正常功能不受影响;

研究问题 3. DCSolver 是否能以轻量化的方案完成依赖冲突修复? 此问题旨在评估 DCSolver 是否能够解决复杂性挑战,实现轻量化修复方案;

研究问题 4. DCSolver 是否能高效地完成依赖冲突自动化修复任务? 此问题旨在评估 DCSolver 的修复分析效率.

3.1 实验数据集与环境

为构建具备代表性的真实项目数据集,本文在 GitHub 开源仓库中随机选择以 Maven 作为依赖管理工具,且具备一定流行度(收藏数量高于 100)的 Java 项目,同时要求项目中的第三方依赖库总数大于 10,以保证其代表性.最终,实验评估使用的数据集由 150 个项目组成,平均每个项目中包含 15415 行 Java 代码、136 个不同的第三方依赖库与 347 条依赖关系边.实验环境为一台配备 Intel Xeon Gold 6242 处理器和 245 GB 内存的 Ubuntu 20.04 机器,实验时使用的 Maven 版本为 3.9.5,使用的 JDK 版本为 8,11 和 17.

3.2 修复有效性评估

为验证 DCSolver 对于依赖冲突问题的修复效果,本文利用 Maven Dependency Plugin[15] 提供的项目完整依赖树查看命令(mvn dependency:tree -Dverbose),分别统计修复前后项目中存在的依赖冲突问题,由此判断 DCSolver 可成功修复的依赖冲突问题数量.

表 3 修复有效性评估结果

Table 3 Evaluation result of effectiveness

存在依赖冲突的项目数	依赖冲突问题数	修复成功数
138 / 150	787	787

表 3 展示了数据集所有项目中包含的依赖冲突问题数量,以及 DCSolver 的修复效果.统计结果显示,150 个项目中的 138 个项目包含了总共 787 个依赖冲突问题,且 DCSolver 修复了所有依赖冲突问题,修复成功率达到 100%.这充分证明了 DCSolver 得益于其采取的多样化修复策略,具备较强的依赖冲突问题修复能力.

图 6 展示了依赖冲突问题与项目的对应关系,根据项目所包含的依赖冲突问题数量进行分类.从中可知,仅有 8% 的项目中不存在依赖冲突问题,且 83% 的项目包含了多个依赖冲突问题,单个项目中最多同时存在 21 个依赖冲突问题.这进一步体现了 Java 依赖冲突问题的普遍性,以及针对依赖冲突问题研究自动化修复技术的必要性.

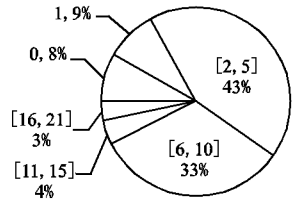

图 6 依赖冲突问题分布统计

Fig. 6 Distribution statistics of dependency conflict

3.3 修复兼容性评估

为确定 DCSolver 在修复依赖冲突问题时是否影响项目兼容性,本实验首先为各个项目应用依赖冲突修复方案,之后分别执行由 Maven 提供的项目编译命令(mvn compile)和单元测试命令(mvn test),根据命令执行结果判断项目是否受修复影响而产生编译错误、运行时异常等问题.部分项目在执行编译或测试命令时要求特殊环境配置,如指定 JDK 版本或指定额外参数.对此,本文将遵循项目文档中提供的具体说明,正确执行相应命令.

表 4 兼容性评估结果

Table 4 Evaluation result of compatibility

兼容性维度	执行成功	执行失败	未知
项目编译	138	0	0
单元测试	101	0	37

表 4 展示了 DCSolver 为存在依赖冲突问题的 138 个项目完成修复后,项目的兼容性评估结果.由结果可知,所有 138 个项目在修复后均可正常通过编译,且其中 101 个项目在修复后可通过单元测试,未出现修复后编译失败的项目.本文通过分析发现,表 4 中单元测试结果为"未知"的 37 个项目包含以下两类情况:1)项目中未提供测试套件,或测试套件不完整;2)DCSolver 为项目修复依赖冲突问题之前与之后,项目单元测试执行均未通过.这些情况下,无法判断单元测试结果是否受修复所影响,本文将其标记为未知.总体而言,DCSolver 在修复依赖冲突问题的同时,较好地保持了项目兼容性不受影响,解决了兼容性挑战.

3.4 方案复杂性评估

为评估 DCSolver 是否能够解决复杂性挑战,以轻量级方案完成对依赖冲突问题的修复,本节对 DCSolver 给出的修复方案中包含的修改操作数量进行统计,并对其分布情况进行人工分析.

统计结果显示,对于 138 个项目中的 787 个依赖冲突问题,DCSolver 给出的修复方案共包含 2756 次修改操作,平均修复每个依赖冲突问题仅需 3.5 次修改操作.本文通过深入分析发现,DCSolver 取得如此表现的主要原因有两点:1)DCSolver 采取了拓扑排序机制,优先处理对项目影响较大的依赖冲突问题,且在修复过程中实时更新依赖关系图,这对于减

少后续解决其他依赖冲突问题时的修改操作存在一定帮助;2)DCSolver 采取了包含移除依赖关系在内的修复策略,在冲突依赖库未被项目调用时,可通过排除操作解决依赖冲突问题,进一步简化修复方案.

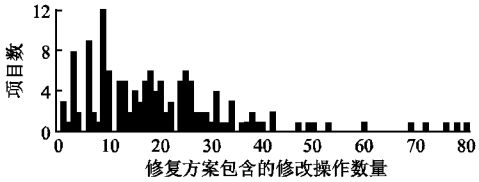

图 7 修改操作数量分布统计

Fig. 7 Distribution statistics of modification operations

图 7 展示了各个项目的依赖冲突问题修复方案中,包含的修改操作数量分布情况. 可以发现 DCSolver 为大多数项目给出的修复方案仅包含 30 次以内的修改操作. 考虑到数据集中项目的庞大依赖关系图规模(136 个不同的第三方依赖库、347 条依赖关系边),可认为 DCSolver 成功解决了复杂性挑战,其给出的依赖冲突问题修复方案可充分满足轻量化需求.

3.5 修复效率评估

为评估 DCSolver 在依赖冲突问题修复任务上的分析效率,本文通过代码插桩的方式,统计 DCSolver 完成依赖冲突修复任务的整体运行时间,以及各个模块的时间开销.

表 5 分析效率评估结果(单位:秒)

Table 5 Evaluation result of performance(unit:seconds)

工具模块	运行时间(平均值)	运行时间(中位数)
预处理	41.7	39.4
方案构建	113.9	98.1
方案应用	3.8	4.0
端到端总用时	159.4	145

表 5 展示了 DCSolver 为 150 个项目的运行时间统计结果,平均每个项目仅需 159.4 秒即可完成所有依赖冲突问题的修复. 进一步分析可发现,预处理模块中在执行依赖使用分析时,需对项目与依赖库代码进行完整扫描,故存在一定的时间开销;此外,方案构建模块中需要对不同版本的依赖库进行代码差异分析,其运行时间随着依赖冲突问题数量与冲突节点数量增加而增加. 考虑到这些分析开销使得 DCSolver 在 3.3节的修复兼容性评估中取得了优异表现,本文认为 DCSolver 可高效地完成依赖冲突自动化修复任务,帮助开发者降低修复成本.

4 总 结

由第三方库引入的依赖冲突问题可能影响项目稳定性,需及时进行修复. 为解决依赖冲突人工修复成本较高的问题,本文提出并实现了 Java 依赖冲突自动化修复框架 DCSolver,利用多样化修复策略实现对项目依赖结构的灵活调整,构建依赖冲突修复方案并自动化部署. 在真实项目数据集上的实验结果表明,DCSolver 能够以轻量级的修复方案,有效地帮助开发者及时修复依赖冲突问题,同时保证项目兼容性不受影响,对于增强开源软件生态稳定性具有积极意义.

References:

[1] Sonatype Inc. Open source supply and demand[EB/OL]. https://www. sonatype. com/state-of-the-software-supply-chain/open-source-supply-and-demand,2023-12.

[2] Apache Software Foundations. HADOOP-11656 [EB/OL]. https://issues. apache. org/jira/browse/HADOOP-11656,2024-03.

[3] Wang Y,Wen M,Liu Z,et al. Do the dependency conflicts in my project matter? [C]//Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering(ESEC/FSE),2018:319-330.

[4] Wang Y,Wen M,Wu R,et al. Could i have a stack trace to examine thedependency conflict issue? [C]//Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE),2019:572-583.

[5] Wang Y,Wu R,Wang C,et al. Will dependency conflicts affect My program′s semantics? [J]. IEEE Transactions on Software Engineering,2021,48(7):2295-2316.

[6] Huang K,Chen B,Shi B,et al. Interactive,effort-aware library versionharmonization[C]//Proceedings of the 28th ACM Joint Meeting on EuropeanSoftware Engineering Conference and Symposium on the Foundations of Software Engineering(ESEC/FSE),2020:518-529.

[7] Wang Y,Wen M,Liu Y,et al. Watchman:monitoring dependency conflicts forpython library ecosystem [C]//Proceedings of the ACM/IEEE 42nd InternationalConference on Software Engineering (ICSE),2020:125-135.

[8] Wang C,Wu R,Song H,et al. Smartpip:a smart approach to resolving pythondependency conflict issues [C]//Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering(ASE),2022:1-12.

[9] Patra J,Dixit P N,Pradel M. Conflictjs:finding and understanding conflictsbetween javascript libraries[C]//Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE),2018:741-751.

[10] LI S,LIU J,WANG S,et al. Survey on dependency conflict problem of third-party libraries[J]. Journal of Software,2023,34(10):4636-4660.

[11] Ferstl. Depgraph maven plugin[EB/OL]. https://github. com/ferstl/depgraph-maven-plugin,2024.

[12] ValléeRai R,Co P,Gagnon E,et al. Soot:a java bytecode optimizationframework[M]. Mississauga Ontario Canada:CASCON First Decade High Impact Papers,2010:214-224.

[13] Falleri J R,Morandat F,Blanc X,et al. Fine-grained and accurate source codedifferencing[C]//Proceedings of the 29th ACM/IEEE International Conference onAutomated Software Engineering (ASE),2014:313-324.

[14] Apache Maven Project. Apache maven shade plugin[EB/OL]. https://maven. apache. org/plugins/maven-shade-plugin/,2024-03.

[15] Apache Maven Project. Apache maven dependency plugin[EB/OL]. https://maven. apache. org/plugins/maven-dependency-plugin/,2024-03.

附中文参考文献:

[10] 李 硕,刘 杰,王 帅,等. 第三方库依赖冲突问题研究综述[J]. 软件学报,2023,34(10):4636-4660.