软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(7):3041−3086 [doi: 10.13328/j.cnki.jos.007338] [CSTR: 32375.14.jos.007338]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

Java 依赖异味的实证研究与统一检测技术*

孙伟杰[1,2], 许 畅[1,2], 王 莹[3]

[1](计算机软件新技术全国重点实验室 (南京大学), 江苏 南京 210023)

[2](南京大学 计算机学院, 江苏 南京 210023)

[3](东北大学 软件学院, 辽宁 沈阳 110169)

通信作者: 许畅, E-mail: changxu@nju.edu.cn

摘 要: Java 语言因丰富的依赖库和便捷的构建工具 (如 Maven 和 Gradle) 已成为当今最流行的应用项目开发语言之一. 然而, 随着依赖库规模的持续增大, Java 项目的依赖管理变得愈益复杂, 也不断超越现有工具的管理能力, 其潜藏问题容易在未预期情况下触发, 严重影响当前项目及所在 Java 生态中其他项目的构建和运行, 如造成构建错误、运行崩溃或语义冲突等后果. 针对现有调研和技术工作对 Java 语言依赖管理问题分析不足的缺陷, 提出依赖异味 (dependency smell) 的概念, 统一建模此类问题, 并对涉及 Maven 和 Gradle 构建工具所有类别的依赖管理问题开展大规模实证研究, 分析来自开源社区 (如 GitHub)、官方文档 (如 Maven 依赖管理手册) 和系列调研及技术论文的各类依赖管理问题, 最终总结出 13 类依赖异味及其触发根源和影响特征等. 基于该实证研究发现, 设计了面向 Java 项目依赖异味的统一检测算法, 并实现了适配于 Maven 和 Gradle 构建工具的专项检测工具 JDepAna. 实验结果表明, 对已知依赖异味, JDepAna 达到 95.9% 的检测召回率, 对新的上百个 Java 项目, JDepAna 检测出 30 689 个依赖异味实例, 从中选出 360 个实例, 人工验证真阳率达到 96.1%, 其中, 进一步汇报 48 个实例给开发者, 42 个已被快速确认, 21 个已被及时修复, 充分验证了所提出的 Java 依赖异味检测算法和工具的效果和实用性以及对 Java 项目质量保障的有效支撑.

关键词: 依赖管理; 软件生态; 质量保障; 实证研究; 静态分析

中图法分类号: TP311

中文引用格式: 孙伟杰, 许畅, 王莹. Java依赖异味的实证研究与统一检测技术. 软件学报, 2025, 36(7): 3041–3086. http://www.jos.org.cn/1000-9825/7338.htm

英文引用格式: Sun WJ, Xu C, Wang Y. Empirical Study and Unified Detection Technique of Dependency Smells in Java. Ruan Jian Xue Bao/Journal of Software, 2025, 36(7): 3041–3086 (in Chinese). http://www.jos.org.cn/1000-9825/7338.htm

Empirical Study and Unified Detection Technique of Dependency Smells in Java

SUN Wei-Jie[1,2], XU Chang[1,2], WANG Ying[3]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](School of Computer Science, Nanjing University, Nanjing 210023, China)

[3](Software College, Northeastern University, Shenyang 110169, China)

Abstract: Java has become one of the most popular programming languages for application project development nowadays, due to its rich dependency libraries and convenient build tools such as Maven and Gradle. However, with the continuous increase in the scale of dependency libraries, the dependency management of Java projects becomes increasingly complex and constantly exceeds the management capabilities of existing tools. The potential problems are likely to be triggered unexpectedly, seriously affecting the building and running of the current project and other projects in the Java ecosystem, such as causing build errors, runtime crashes, or semantic conflicts. This

* 基金项目: 国家自然科学基金 (62141210); 江苏省前沿引领技术基础研究专项 (BK20202001)
本文由“新兴软件与系统的可信赖性与安全”专题特约编辑向剑文教授、陈厅教授、杨珉教授、周俊伟教授推荐.
收稿时间: 2024-08-25; 修改时间: 2024-10-15; 采用时间: 2024-11-25; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-04-24

study aims to address the gaps in the analysis of dependency management issues found in existing research and technical literature by introducing the concept of “dependency smell”, to build a unified model for these challenges. This study conducts a comprehensive empirical study on dependency management issues, covering all categories of Maven and Gradle related problems. This study analyzes diverse dependency management issues gathered from open-source communities (e.g., GitHub), official documentation (e.g., Maven manual), as well as various surveys and technical papers. Finally, 13 types of dependency smell, as well as their triggering roots and impact characteristics, are summarized. Based on the findings of this empirical study, a unified detection algorithm for dependency smells in Java projects is designed, and a special detection tool JDepAna suitable for Maven and Gradle build tools is implemented. Experimental results demonstrate that for known dependency smells, JDepAna achieves a detection recall rate of 95.9%. For hundreds of new Java projects, JDepAna detects 30 689 instances of dependency smells. 360 instances are selected, and the true positive rate of manual verification reaches 96.1%. Additionally, this study reports 48 instances to developers, with 42 instances promptly confirmed and 21 promptly fixed, thereby validating the efficacy and practicality of the proposed Java dependency smell detection algorithm and tool in facilitating quality assurance for Java projects.

Key words: dependency management; software ecosystem; quality assurance; empirical study; static analysis

Java 语言因为丰富的依赖库而广泛被使用于应用开发[1], 开发者可以通过复用依赖库来提高开发效率和软件质量[2]. 为了管理应用开发过程中繁复且多变的依赖库, Java 提供了便捷的构建工具 (如 Maven 和 Gradle)[3], 开发者只需在对应配置文件中进行配置, 构建工具就会自动化地获取依赖库并进行集中管理.

然而随着依赖库的持续演进与规模扩张, Java 项目的依赖管理任务日益复杂化, 其中隐含的问题可能在非预期时刻被触发, 进而造成严重危害[4]. 鉴于此, 本文引入了依赖异味 (dependency smell) 的概念, 指的是依赖配置文件或代码中潜在的对软件项目或生态系统演化产生负面影响的不良依赖管理模式或特征. 这些模式或特征可能导致对软件的理解、维护或扩展存在挑战, 从而影响软件的整体质量和稳定性. 依赖异味作为隐藏在依赖管理深处的隐患, 虽可能不直接导致程序缺陷 (Bug), 但增加了引入缺陷的风险, 最终可能在特定情境下会被触发进而影响项目的构建和运行, 导致诸如构建失效、运行崩溃或语义冲突等一系列问题. 以 GitHub 中的问题报告 (Issue) #EL-1849[5]为例, 项目 Jersey[6]在代码中使用了依赖库 eclipse:asm:9.4.0, 但却未在配置文件中声明. 与此同时 Jersey 的另一依赖库 eclipse:moxy:4.0.0 明确声明了对 asm:9.4.0 的依赖关系, 这起初使得 Jersey 能够经由 moxy:4.0.0 间接获得 asm:9.4.0 并维持正常运作. 然而 moxy:4.0.1 移除了对 asm:9.4.0 的依赖, 随着 Jersey 将 moxy:4.0.0 升级为 moxy:4.0.1, 其构建过程中无法找到 asm:9.4.0, 最终导致构建时出现 ExceptionInInitializerError 异常.

值得注意的是, Java 项目依赖管理中的类似问题已经引起了研究者的广泛关注, 例如对 Maven 项目中依赖冲突和依赖冗余问题的分析[7−11]. 然而, 目前针对 Java 中依赖异味的研究仍存在空缺. 首先, 目前研究大多关注依赖管理已对项目造成的危害, 但未充分考虑依赖管理中潜在的问题. 此外, 目前的研究主要关注于使用 Maven 进行依赖管理的 Java 项目. 但在传统的构建工具 Maven 之外, 新兴的构建工具 Gradle 同样被广泛使用. Gradle 提供了更多的灵活性和定制化选项, 使开发者能够更精细地控制项目的依赖关系, 但也增加了研究的难度[12]. 在 Java 语言之外, Cao 等人[13]和 Jafari 等人[14]分别对 Python 和 JavaScript 中的依赖异味进行了调研, 发现了依赖异味在这两种语言中的普遍性. 虽然由于语言特性和构建工具配置方式的显著差异, Python 和 JavaScript 中的异味并不直接适用于 Java, 但也启发我们 Java 依赖异味研究存在对应空缺.

针对现有工作对 Java 语言依赖管理中潜藏问题分析不足的缺陷, 我们综合官方文档[15,16]、学术论文[7−14,17−29]以及开源社区 GitHub[30]中的问题报告开展实证研究, 最终发现 Maven 和 Gradle 两大构建工具中的 13 类依赖异味. 我们详细分析了这些依赖异味对应的特征和具体的实例, 并探讨它们的潜在危害及触发场景. 基于实证研究结果, 我们设计了针对 Maven 和 Gradle 项目的依赖模型以及检测上述依赖异味的统一检测算法, 并实现了适配于 Maven 和 Gradle 的专项检测工具 JDepAna.

为了评估 JDepAna 的检测效果, 我们收集了包含 147 个已知依赖异味实例的基准集. JDepAna 在此基准集上的表现卓越, 达到了 95.9% 的召回率和 96.1% 的精确率. 此外, 我们在流行的 73 个 Maven 项目和 51 个 Gradle 项目中进一步验证, JDepAna 在这些实际项目上的精确率同样维持在 96.1% 的高水平. 我们据此向开发者报告了 48 例检测出的依赖异味实例, 其中 42 例 (87.5%) 得到了确认, 21 例 (43.8%) 已被快速修复. 这一系列的实验证明了

我们的研究及其配套检测工具 JDepAna 具有显著的实际应用价值.

总的来说, 本文主要做出以下贡献: (1) 发现了 Java 项目依赖管理中可能存在的 13 类依赖异味; (2) 设计了适用于 Maven 和 Gradle 项目依赖管理的依赖模型, 并提出了针对各种依赖异味的检测算法; (3) 实现了同时适配于 Maven 和 Gradle 的依赖异味检测工具 JDepAna, 并通过实验证明了其有效性和实用性.

本文第 1 节介绍 Java 项目依赖管理中的基本概念. 第 2 节介绍针对 Java 项目中依赖异味进行的实证研究以及经过研究获得的 Java 依赖管理中可能存在的依赖异味和对应影响. 第 3 节基于实证研究结果, 介绍依赖异味的检测流程和检测工具的具体实现. 第 4 节介绍对前述检测工具的检测效果的实验评估和实验过程中的效度威胁性. 第 5 节介绍相关研究工作. 第 6 节对全文进行总结, 并对未来研究工作进行展望.

1 背 景

为了便于后续讨论, 本节将介绍 Java 项目的依赖管理和常用构建工具 Maven 和 Gradle 的依赖解析机制.

1.1 Java 项目依赖管理

Java 项目广泛使用 Maven 和 Gradle 进行依赖管理. 作为两种主流的构建工具, 它们的构建最小单元都是模块 (module). 每个模块包含其专属的源代码、测试代码及配置文件 (Maven 对应 pom.xml, Gradle 对应 build.gradle). 这些模块在构建过程中生成的构件 (artifact) 通常是 JAR 文件, 通过三元组标识符 (G:A:V) 予以区分, 分别对应组织名称 (G)、构件名称 (A) 及版本信息 (V). 在配置文件中, 模块同样以 (G:A:V) 的形式声明源代码和测试代码中所使用的依赖库. 配置文件中声明的依赖库依据其功能角色, 在不同的使用场景下发挥作用, 如被用于源代码编译或进行测试. Maven 和 Gradle 都提供了依赖库的对应配置选项, 供开发者精细化控制依赖库的使用场景 (主要包括构建、运行和测试这 3 类使用场景), 我们称这类依赖库的配置为依赖范围 (dependency scope). 例如 Maven 中的依赖范围 compile 就对应着构建、运行和测试这 3 类使用场景. Maven 和 Gradle 会解析配置文件, 获取模块所需的依赖库, 并构建模块的依赖树. 依赖树展示了模块所依赖的外部库及其相互关系, 包括主动引入的依赖 (直接依赖) 和由直接依赖引入的其他依赖库 (传递依赖).

在单一模块之外, 多模块 (multi-module) 项目同样是构建复杂 Java 应用程序的常见模式. 这种结构设计允许项目按功能或服务划分成多个独立的模块, 每个模块专注于特定任务的开发、测试与维护, 最终所有模块协同工作, 共同构建出一个完整且功能丰富的应用程序. 多模块项目体系通常围绕一个中心父项目构建, 该父项目虽然本身不承载业务逻辑代码, 却扮演着配置中心的角色, 通过集中管理的配置文件来统辖各子模块的构建行为与依赖规则. 图 1 展示了一个典型的多模块 Maven 项目结构及其依赖管理方式.

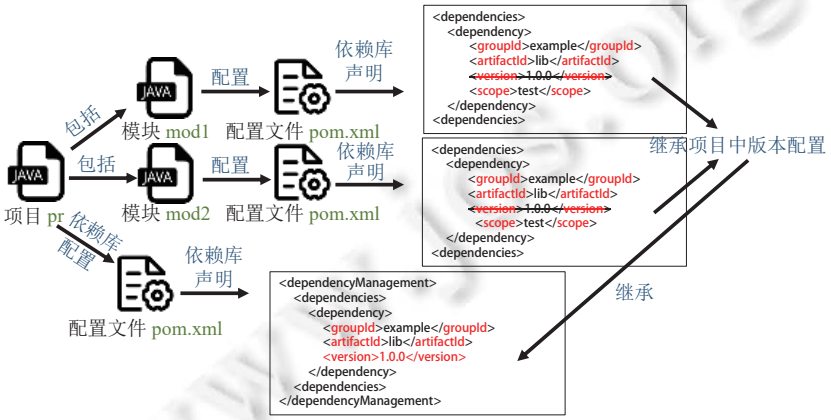

图 1 使用 Maven 的多模块项目

从图 1 中可以看到, 项目 pr 包含两个模块 mod1 和 mod2, 每个模块都有自己的配置文件 pom.xml. 在这些配置文件中, 开发者需要声明所需的依赖库, 并指定其版本和范围. 例如, 模块 mod1 和 mod2 都依赖于 example:lib:

1.0.0 这个库. 值得注意的是, 项目 pr 的配置文件 pom.xml 中包含<dependencyManagement>标签, 用于统一管理依赖版本. 这样一来, mod1 和 mod2 可以继承这个版本信息, 而不需要在各自的 pom.xml 文件中重复声明版本号.

为了确保 Java 项目在不同环境中的构建一致性和可重现性, Maven 和 Gradle 提供了构建工具封装器 (wrapper) 来启动相应版本的构建工具. 封装器包括启动脚本和 JAR 文件 (Maven 使用 maven-wrapper.jar, Gradle 使用 gradle-wrapper.jar). 在构建工具封装器正常工作的情况下, 开发者无须在本地安装 Maven 和 Gradle, 只需运行脚本, 即可调用对应的 JAR 文件, 自动下载并使用相应版本的构建工具.

1.2 构建工具依赖解析机制

Maven 和 Gradle 极大地简化了依赖管理的复杂性. 开发者只需要在配置文件中对项目的直接依赖进行配置, 构建工具会对配置文件中声明的依赖库进行解析, 下载并管理所有的直接依赖和引入的间接依赖. 在解析过程中, 它们会遵循依赖仲裁 (dependency mediation) 机制来确保解析结果的一致性, 主要包括以下两点.

● 版本仲裁机制: 当依赖树中出现同一依赖库的不同版本时, Maven 和 Gradle 仅会选择其中一个版本, 并忽略其他版本. Maven 使用“最短路径优先”策略, 优先选择依赖树中离模块最近的版本; 相比之下, Gradle 默认采用“最高版本优先”策略, 在冲突时选择最高的版本. 例如, 在图 2(a) 中, 模块 mod 同时依赖于 lib:v1.0.0 和 lib:v2.0.0, 此时 Maven 会选择距离较近的 lib:v1.0.0, 而 Gradle 则会选择版本较高的 lib:v2.0.0.

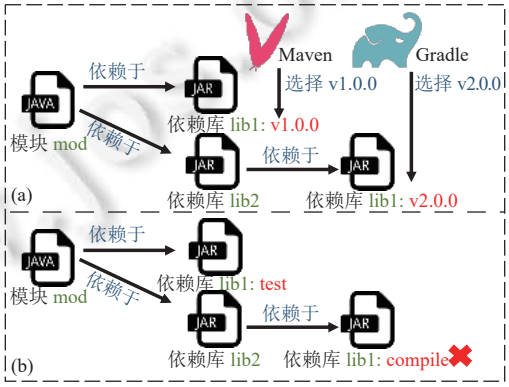

图 2 构建工具解析机制

● 范围仲裁机制: 当依赖树中出现同一依赖库的不同依赖范围时, Maven 仅会选择直接依赖的范围而忽略传递依赖的范围. 例如, 在图 2(b) 中, 依赖库 lib 既是直接依赖又是间接依赖, 其作为直接依赖的范围与作为传递依赖的范围不一致, 最终传递依赖的范围会被忽略.

2 依赖异味的实证研究

为了深入研究 Java 项目依赖管理中潜藏的问题, 我们针对 Java 项目中可能存在的依赖异味及其影响开展实证研究. 接下来我们将介绍数据收集流程并展示研究结果.

● RQ1: Java 项目中可能存在哪些依赖异味类别?

● RQ2: 这些依赖异味会在什么情境下产生怎样的危害?

为了回答 RQ1 和 RQ2, 我们选择了 3 类调研对象, 分别是官方文档、学术论文和开源社区. 我们从官方文档和学术论文中总结出关键词, 根据关键词从开源社区中搜索问题报告, 在人工分析后最终筛选出 47 个依赖异味相关问题报告. 下面我们将详细介绍数据收集过程和调研结果.

2.1 数据收集

步骤 1: 收集相关问题报告. 为了全面调研 Java 项目依赖管理中潜藏的问题, 我们选择了以下 3 类调研对象: (1) 官方文档: Maven、Gradle 的依赖管理手册[15,16]; (2) 学术论文: 依赖管理相关的论文[7−14,17−29]; (3) 开源社区:

GitHub[30]中的项目与问题报告. 这样的选择出于以下考虑: 首先, 官方文档是深入探究 Java 依赖管理的权威途径. Maven、Gradle 的依赖管理手册会帮助我们全面理解 Java 项目依赖管理的核心概念和最佳实践. 其次, 学术论文有助于我们了解现有研究的边界. 依赖管理相关的论文使我们的研究能够包括最新的学术进展. 最后, 开源社区的采用能够有效提供实际问题的案例支持, GitHub 提供了丰富的 Java 代码仓库和相应问题报告供我们开展研究.

我们综合 3 类调研对象, 从而深入探讨 Java 项目依赖管理中的潜藏问题. 首先, 我们调研官方手册和科研论文, 总结出如表 1 中依赖管理相关的关键词, 由于直接以关键词进行搜索则会包含许多无关信息, 例如项目中常规对依赖版本升级的工作, 所以我们根据论文中描述的问题和依赖管理规范, 围绕关键词组织出表 1 中适合于搜索的 7 个搜索子句, 其中 4 个子句来源于官方文档, 3 个来源于相关学术论文. 为了解决 GitHub 搜索对逻辑符数量的限制, 我们将搜索子句分为两组, 分别包含 3 个和 4 个子句, 将这些搜索子句组合成符合 GitHub REST API 的形式在 GitHub 中进行搜索, 返回结果按 GitHub 中的匹配度指标排序, 并分别从中选取前 200 个问题报告. 较高的匹配度能够确保我们所分析的问题报告具有较高的相关性和代表性, 排名 200 之后的问题报告在匹配度指标上相对较低, 也更可能包含无关内容 (例如常规的依赖库版本升级). 这种筛选方法帮助我们聚焦于最具代表性的问题报告, 从而提升研究的有效性. 随后, 我们对两次搜索的结果进行了去重处理, 最终获得了 397 个独立的问题报告. 表 2 中展示这些问题报告对应的项目数据, 可以看到所考察项目具有明显的差异性: (1) 有一定规模 (平均代码行数 519.8k, 小至 200 行, 大至 1 200 万行); (2) 维护良好 (平均 5.5k 次提交); (3) 有一定关注度 (平均有 1k 个 Star).

表 1　依赖异味相关关键词

关键词	搜索子句
“Maven”; “Gradle”; “dependencies”; “dependency”; “dependency scope”; “dependency tree”; “dependency version”; “module”; “class”; “library”; “wrapper.properties”; “wrapper.jar”; “checksum”	“dependency scope” ∧ (“override” ∨ “change”); “conflict” ∧ ((“library” ∧ “class”) ∨ “dependency version”); “modules” ∧ “dependency” ∧ “same”; “dependency” ∧ ((“undeclared” ∧ “used”) ∨ (“unused” ∧ “declared”)); “Maven” ∧ “Gradle” ∧ “dependency”; (“wrapper.jar” ∨ “wrapper.properties”) ∧ “add”; “wrapper.jar” ∧ “checksum” ∧ “correct”

表 2　目标项目信息 (k)

统计指标	Star数目	Commit数目	代码行数
平均值	1	5.5	519.8
最大值	13.7	77	12 067
最小值	0.002	0.025	0.2

步骤 2: 数据分析. 为了回答 RQ1 和 RQ2, 我们深入分析每个问题报告中的开发者讨论以及对应的修复方案. 我们首先排除与依赖异味无关的问题报告, 并对剩余问题报告中反映的问题特征进行总结与分类. 具体来说, 我们排除了与项目日常维护相关的问题报告共 74 个, 例如依赖库的常规版本升级. 接着我们排除了那些缺少明确解决方案, 或仅涉及功能代码修改而不涉及依赖配置文件 (如 pom.xml) 的问题报告共 276 个. 最终, 我们筛选出 47 个与依赖异味相关的问题报告. 对于这些问题报告, 我们通过关键词分析识别其所对应的危害类型. 例如, 增加构建成本的相关关键词包括“Performance regression”; 而运行时错误则涉及“NoSuchMethodError”或“ClassNotFoundException”等. 为判断危害的来源是当前模块还是下游模块, 我们分析了问题报告中是否提及下游模块. 如果问题源于当前模块作为独立应用程序运行时引发, 则视为对当前模块的危害; 反之, 如果问题发生在下游模块使用当前模块作为依赖库时, 则视为对下游模块的危害. 在分析过程中, 我们采用了开放编码 (open coding) 过程[31], 这是一种广泛应用于定性研究的方法, 用于对问题报告中讨论的问题进行分类. 在初始阶段, 本文的两位作者独立分析了 199 个问题报告的修复方案和开发者讨论. 第 1 轮分析和分类后, 两位作者汇总比较并讨论他们的结果, 以调整分类方法, 并在第 3 位作者的协助下解决冲突. 通过这种方式, 我们建立了初步的分类法. 接下来, 前两位作者继续对剩余 198 个问题报告的修复方案或讨论进行分类, 并迭代地完善结果和分类策略. 分类中的冲突再次在会议期间讨论, 并由第 3 位作者解决. 我们调整了初步的分类法, 并最终得到了结果.

2.2 调研结果

我们将调研结果展示在表 3 中, 它包含了调研得到的依赖异味类别, 各个类别对应的问题报告和构建工具以及可能造成的危害. 我们用表 4 中的编号来表示对应危害. 接下来我们将结合表格介绍我们对 RQ1 和 RQ2 的调研结果. 由于篇幅限制, 本文中仅介绍部分依赖异味类别, 剩余部分参见附录 A.

表 3 实证研究调研结果

异味编号	异味名称	异味粒度	对应构建工具	对应问题报告数	主要危害
1.1	内外类冲突	模块粒度	Maven|Gradle	2/47	1, 3, 4
1.2	外部类冲突	模块粒度	Maven|Gradle	7/47	1, 3, 4
1.3	库版本冲突	模块粒度	Maven|Gradle	7/47	1, 3, 4
1.4	未声明依赖	模块粒度	Maven|Gradle	6/47	1, 3, 4
1.5	未使用依赖	模块粒度	Maven|Gradle	6/47	2, 5, 7, 9
1.6	依赖范围误用	模块粒度	Maven|Gradle	12/47	1, 2, 3, 5, 6, 7, 8, 9
1.7	依赖范围冲突	模块粒度	Maven	2/47	1, 3
1.8	依赖树冲突	模块粒度	Maven&Gradle	1/47	1, 3, 4, 11
2.1	构建工具配置缺失	项目粒度	Maven|Gradle	2/47	1
2.2	构建工具封装器JAR缺失	项目粒度	Maven|Gradle	6/47	1
2.3	构建工具封装器JAR异常	项目粒度	Maven|Gradle	1/47	2, 10
2.4	模块间库重复	项目粒度	Maven|Gradle	2/47	11
2.5	模块间库冲突	项目粒度	Maven|Gradle	1/47	11

表 4 依赖异味主要危害

编号	潜在危害	详细说明
1	构建错误	模块构建时出现错误
2	构建时间增加	模块构建所需时间增加
3	运行时错误	模块运行时出现异常
4	运行时语义冲突	模块运行结果与预期不一致
5	构建产物冗余	模块构建产物 (可执行JAR) 体积增大
6	下游模块构建错误	当前模块的下游模块构建时出现错误
7	下游模块构建时间增加	当前模块的下游模块构建所需时间增加
8	下游模块运行时错误	当前模块的下游模块运行时出现异常
9	下游模块构建产物冗余	当前模块的下游模块运行结果与预期不一致
10	安全隐患	项目在使用构建工具时执行被植入的恶意代码
11	依赖库维护难度增加	项目无法统一调整多个同名依赖库的版本而需多次调整

2.2.1 RQ1: Java 项目中可能存在哪些依赖异味类别?

最终我们筛选出 47 个依赖异味相关的问题报告, 总结出 13 类特征各异的依赖异味. 已有研究尚未关注到其中 7 类依赖异味 (异味 1.4、1.7–1.8、2.1–2.4), 而仅从 Maven 的角度对其余 6 类依赖异味进行了部分探讨. 我们的研究不仅在依赖异味的类别上有所扩展, 还综合考虑了 Maven 和 Gradle 这两大主流构建工具. 为了方便研究, 我们把这些依赖异味分为两种粒度, 即模块粒度和项目粒度. 这两种粒度分别代表了依赖异味出现在单个模块中还是整个项目中. 需要说明的是, 模块粒度关注单个模块与其依赖库之间的关系, 而项目粒度则关注项目中多个模块之间的关系, 因此这样的划分方式确保了分类的独立性, 它们之间没有重叠.

1) 模块粒度

在模块级别的分析中, 我们发现了 36 个 (占比 36/47=76.6%) 问题报告中包含的 8 种 (占比 8/13=61.5%) 依赖异味, 对应于表 3 中的 1.1–1.8. 在进行分类时, 我们结合问题报告中涉及的依赖管理的具体实体 (如依赖范围) 进行分析. 最终, 我们总结出 8 类依赖异味, 它们全面概括了 36 个问题报告中的特征. 其中值得注意的是, 异味 1.7 仅在使用 Maven 中出现, 而异味 1.8 仅在同时使用 Maven 和 Gradle 时才会出现, 其余异味在使用 Maven 或

Gradle 时都可能出现. 接下来, 我们将给出异味 1.1–1.8 的定义. 由于篇幅限制, 我们对最后的 3 类异味 (即 1.6–1.8) 将给出详细介绍 (包括例子分析; 而其他类别异味的详细介绍和例子分析参见附录 A), 它们都是先前研究未涉及或未充分讨论的, 并且分别对应了不同的构建工具.

(1) 类别 1.1 内外类名冲突: 模块与依赖库中包含完全限定名相同的类 (2/47). 模块源代码中的类与模块所使用的依赖库中的类完全限定名相同.

(2) 类别 1.2 外部类名冲突: 模块的依赖库之间包含完全限定名相同的类 (7/47). 模块所使用的依赖库之间包含完全限定名相同的类.

(3) 类别 1.3 库版本冲突: 模块的依赖树中存在同一依赖库的不同版本 (7/47). 模块的依赖树中存在组织名和构件名都相同但版本不同的依赖库.

(4) 类别 1.4 未声明依赖: 模块直接使用了未在配置文件中声明的依赖库 (6/47). 模块源代码中使用某依赖库中的类, 但此依赖库并未在项目的配置文件中得到明确声明.

(5) 类别 1.5 未使用依赖: 模块在配置文件中声明的依赖库实际上并未被使用 (6/47). 模块源代码中并未使用某依赖库中的任何类, 但此依赖库却在配置文件中声明.

(6) 类别 1.6 依赖范围误用: 依赖库的实际依赖范围与预期依赖范围不一致 (12/47). 依赖库在配置文件中的依赖范围, 也即实际依赖范围, 与其在模块中的真实使用场景所对应的预期依赖范围不一致. 在图 3(a) 对应的 #CNNIP-4[32]中, 项目 Cavisson-ns-nd-integration-plugin 在声明依赖库 jenkins-ci:jenkins-test-harness:2.25 时未指定其依赖范围, 导致 Maven 自动将其依赖范围设置为默认的 compile. 然而 jenkins-test-harness 事实上是一个与测试相关的依赖库, 应该仅在测试阶段使用, 因此其预期范围应为 test, 导致实际依赖范围与预期不符. 这意味着依赖库会在项目的构建、运行和测试阶段均被引入, 从而导致了依赖的冗余. 为了解决这个问题, 开发者最终将 jenkins-test-harness 的依赖范围修改为预期的 test, 确保它仅在测试阶段被引入.

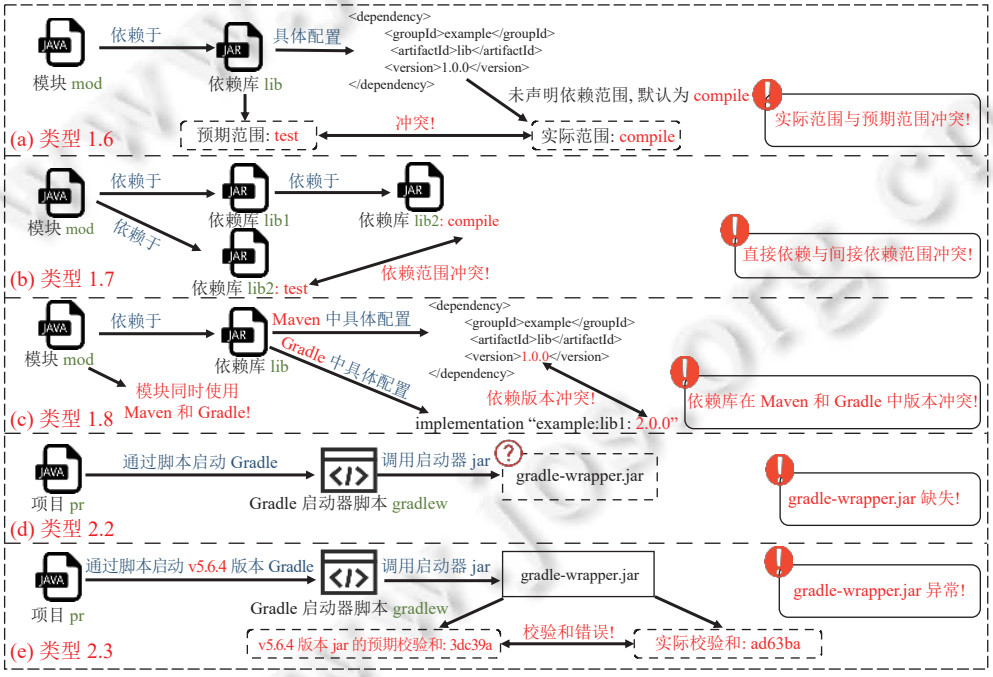

图 3 出现依赖异味特征的案例

这种异味在 Maven 和 Gradle 中都存在, 其产生原因主要在于开发者对依赖库具体使用场景的认知错误, 从而无法选择正确的依赖范围. 虽然 Maven 和 Gradle 中都存在依赖范围的概念, 但由于它们对依赖范围的实现和定义不同, 依赖范围误用在 Maven 和 Gradle 中表现出不同的形式. 由于 Maven 中仅有 6 个固定的依赖范围, 因此

Maven 中的依赖范围误用表现形式较为单一. 在 9 个 Maven 依赖范围误用的问题报告中, 有 8 例中依赖库的预期依赖范围为 test, 而实际依赖范围为 compile.

(7) 类别 1.7 依赖范围冲突: 直接依赖在依赖树中存在冲突的依赖范围 (2/47). 依赖库同时作为模块的直接依赖和间接依赖存在, 但直接依赖对应的依赖范围与间接依赖对应的依赖范围不一致. 在图 3(b) 对应的#DW-3769[33]中, 模块导入 dropwizard:dropwizard-dependencies 中引入的依赖库 jakarta:jakarta.xml.bind-api:2.32 和 jakarta:jakarta.xml.bind-api:2.32 为 test 范围的直接依赖. 然而, 这两个依赖库在传递依赖中同样存在, 并且它们的依赖范围被设置为 compile. 这造成了依赖范围的冲突, 导致直接依赖中的测试范围覆盖了传递依赖中的 compile 范围. 因此, 尽管这两个依赖库预期在构建、运行和测试阶段都被使用, 但最终只在测试阶段中被引入使用. 最终开发者将直接依赖的范围同样设置为 compile 以解决冲突.

这类异味仅存在于 Maven 中, 其产生原因在于 Maven 的内部结构中隐藏的一个深层漏洞#MNG-8041[34]. 相比之下, Gradle 并不受此类漏洞的影响, 因此不会产生类似的异味. 这个 Maven 中的缺陷广泛存在于其所有的 3.x 版本中, 其结果是导致项目在处理依赖库冲突时无法正常运作. 值得注意的是, 这类异味很容易被忽略, 因为在将出现此异味的项目作为库使用时, 依赖范围冲突并不会立即显现出问题; 只有当模块自身作为应用程序被使用时, 才会出现相应危害.

(8) 类别 1.8 依赖树冲突: Maven 和 Gradle 解析出的依赖树不一致 (1/47). 模块同时包含 Maven 和 Gradle 的配置文件, 且 Maven 和 Gradle 根据各自配置文件解析出的依赖树不一致. 在图 3(c) 对应的#MSJC-192[35]中, 模块 Msgraph-sdk-java-core 同时使用 Maven 和 Gradle 进行依赖管理, 然而在这两个构建工具对应的配置文件中, 对依赖库 com.azure:azure-identity 和 com.google.guava:guava 的版本声明却不一致. 最终开发者将 Maven 配置文件中的版本配置调整为与 Gradle 一致, 并且引入自动化机器人以确保 Maven 中依赖版本保持与 Gradle 中一致.

这类异味仅存在于同时使用 Maven 和 Gradle 作为构建工具的项目中, 其产生原因主要在于开发者对项目配置文件的修改未能同步至所有相关的构建工具对应的配置文件中. 以模块 Msgraph-sdk-java-core 为例, 其选择使用 Gradle 进行依赖管理, 但也同时维护了 Maven 的配置文件, 以确保项目能够兼容不同的环境. 在 MSJC-176[36]中, 项目中对依赖库 guava 进行升级, 但仅在 Gradle 对应的配置文件 build.gradle 中进行了修改, 而并未将修改同步至 Maven 对应的 pom.xml, 造成了 Maven 和 Gradle 依赖树的冲突. 最终在 MSJC-192[35]中, 开发者发现并修复了此问题.

2) 项目粒度

在项目级别, 我们发现了 11 个 (占比 11/47=23.4%) 问题报告中的 5 种 (占比 5/13=38.5%) 依赖异味类别, 它们分别对应表 3 中的类别 2.1–2.5. 为了完整地进行分类, 我们基于 Maven 和 Gradle 为项目管理提供的手段进行考虑, 并关注其中涉及的依赖管理的具体实体 (如构建工具封装器). 最终我们总结出 5 类异味, 它们涵盖了 11 个问题报告, 并且都同时存在于 Maven 和 Gradle 中. 接下来, 我们将给出异味 2.1–2.5 的定义. 由于篇幅限制, 我们将对其中异味类别 2.2 和 2.3 进行详细介绍 (其他异味类别的详细分析也参见附录 A), 它们都是先前研究未涉及或未充分讨论的, 并且分别对应了不同的构建工具.

(1) 类别 2.1 构建工具配置缺失: 项目中缺少对其所使用构建工具的配置 (2/47). 项目中缺少所使用的构建工具的版本等关键信息的配置文件.

(2) 类别 2.2 构建工具封装器 JAR 缺失: 项目构建工具封装器对应的 JAR 包缺失 (6/47). 项目中所使用的构建工具封装器所需的 JAR 包未出现在封装器对应路径下. 在图 3(d) 对应的#PENNA-16[37]中, 项目 Penna 在上传至 GitHub 时没有将 gradle-wrapper.jar 与项目的其他代码一起上传.gradle-wrapper.jar 是 Gradle 封装器的一部分, 它用于自动下载和配置 Gradle 的特定版本, 以确保项目的构建与特定版本的 Gradle 兼容. 由于 gradle-wrapper.jar 的缺失, 因此在尝试通过封装器脚本 gradlew 启动 Gradle 时, 脚本无法找到 gradle-wrapper.jar, 最终导致了 ClassNotFoundException 异常. 开发者通过上传对应的 gradle-wrapper.jar 解决了此问题.

这类异味在 Maven 和 Gradle 中均存在, 其出现的主要原因是 JAR 包是二进制文件, 在版本控制系统的管理中通常会忽略二进制文件, 因此构建工具封装器 JAR 更容易缺失. 以#LEGAL-570[38]为例, ASF 项目中允许源代码

发布中包含许多类型的构建工具, 但 Maven 和 Gradle 的构建工具封装器 JAR 由于其二进制文件的形式而被禁止.

(3) 类别 2.3 构建工具封装器 JAR 异常: 项目构建工具封装器的 JAR 包实际校验和与官方提供的预期校验和不一致 (1/47). 项目所采用构建工具封装器的 JAR 文件的实际二进制校验和与官方发布版本所声明的预期二进制校验不符. 在图 3(e) 对应的#BISQ-3422[39]中, Bisq 开发者在升级 Gradle 版本至 5.6.4 后, 只修改了 Gradle 封装器的版本配置, 而未更新 gradle-wrapper.jar 文件, 导致 gradle-wrapper.jar 的版本仍然是升级前的 4.10.2. 这样的操作使得 gradle-wrapper.jar 的实际校验和与官方提供的 5.6.4 版本对应的预期校验和冲突. 其产生的主要原因为, 在进行 Gradle 版本升级时, 第 1 次升级命令会更新 gradle-wrapper.properties 文件中的版本配置, 但不会更新实际的 JAR 文件. 第 2 次运行才会生成新的 JAR 文件, 使得实际的 JAR 文件与配置文件中指定的 Gradle 版本对应 JAR 一致. 最终开发者通过二次运行升级命令更新 JAR 文件, 解决了此问题.

这类异味在 Maven 和 Gradle 中均存在, 其主要有两种产生原因. 一种是上例中提到的未按照官方提出的两次升级规范. 在这种情况下, 尽管 JAR 包的校验和与对应版本的官方校验和不一致, 但可能与其他版本的官方校验和一致. 另一种情况是通过植入 JAR 包进行的攻击, 以一次针对 Minecraft Online 的攻击为例[40], 攻击者伪造了构建工具封装器 JAR 并试图植入对应仓库, 以获取使用者的个人信息. 这类异味往往难以被发现, 因为 JAR 包以二进制形式存在, 不使用特定工具难以确定替代 JAR 包的具体内容和来源. 因此, Maven 和 Gradle 发布了各个版本的封装器 JAR 包的官方校验和, 并提供了利用校验和检验是否存在异常的功能.

(4) 类别 2.4 模块间库重复: 项目多个模块中包含未进行统一管理的同一依赖库 (2/47). 项目多个模块中使用同一依赖库, 但并未使用构建工具提供的版本管理机制对其版本进行统一管理.

(5) 类别 2.5 模块间库冲突: 项目多个模块中使用了不同版本的同一依赖库 (1/47). 项目的多个模块中使用了组织名和构件名都相同但版本不同的依赖库.

基于以上发现, 我们回答研究问题 RQ1 如下: 我们发现了 Java 项目中 13 种特征各异的依赖异味, 它们存在于两大主流的项目构建工具 Maven 与 Gradle 中, 既包括单个模块内部的依赖库使用情况, 也触及了模块间的控制, 体现了 Java 项目中依赖异味的多样性和普遍性.

2.2.2 RQ2: 这些依赖异味会在什么情境下产生怎样的危害?

为了全面分析这些依赖异味可能带来的危害, 我们从项目的整个生命周期进行了综合考察, 包括项目构建、运行、产出和项目维护等. 此外, 我们不仅关注模块自身, 还考虑了其作为一个依赖库被下游模块使用时可能带来的影响.

在研究中, 我们发现了 11 类可能的危害, 包括对模块自身以及下游模块的构建、运行等可能造成的危害. 详细结果呈现在表 4 中. 这些危害可能由不同类别的依赖异味在不同的使用场景下被触发而引起. 针对每种危害, 我们都能提供相应的问题报告或图示作为证明. 由于构建和运行是项目最重要的生命周期, 接下来我们将介绍结合实际问题报告介绍异味 1.4、1.6 和 1.7 对模块自身和下游模块的构建和运行可能造成的危害以及对应触发场景.

1) 类别 1.4 未声明依赖: 模块直接使用了未在配置文件中声明的依赖库 (6/47). 模块中存在直接使用的依赖库, 然而这些依赖库并未在项目的配置文件中得到明确声明. 这类异味可能导致构建错误、下游模块运行时错误等一系列危害. 接下来我们将结合实际问题报告介绍这些危害的触发场景.

(1) 构建错误: 触发场景为模块构建时使用的依赖库在依赖树中不存在. 模块的源代码中使用了某依赖库, 但这些依赖库既未在项目的配置文件中明确列出, 也未作为传递依赖被引入. 因此, 在构建过程中, 由于无法定位到这些依赖库而导致构建错误. 在图 4(a) 对应的#EL-1849[5]中, 项目 Jersey 没有明确声明对依赖库 eclipse:asm:9.4.0 的依赖关系, 但通过 eclipse:moxy:4.0.0 间接引入, 从而能够访问 asm 中的类以进行构建和运行. 然而, 在 moxy 版本升级到 4.0.1 后, moxy 不再依赖于 asm, 导致 Jersey 无法通过传递依赖间接访问 asm, 从而导致构建错误. 未声明依赖时常难以被发现, 这主要因为虽然模块未直接声明使用的依赖库, 但传递依赖中恰好存在, 于是便错误地依赖于传递依赖进行正常使用. 由于此时构建工具并不知道模块对此依赖库的使用, 开发者也难以管理自身依赖, 而是被引入的传递依赖控制. 而当传递依赖不再存在时, 则可能触发模块自身构建错误.

2) 类别 1.6 依赖范围误用: 依赖库的实际依赖范围与预期依赖范围不一致 (12/47). Maven 和 Gradle 有多个不

同的依赖范围, 而不同依赖范围之间的误用造成的危害也不尽相同. 因此, 此类异味有大量触发场景和对应的危害. 我们分别选取 Maven 和 Gradle 中主要的依赖范围, 将不同依赖范围之间误用所对应的危害列于表 5 和表 6 中, 其中, 表行代表依赖库的预期依赖范围, 表列代表实际依赖范围, 表的内容是出现实际和预期依赖范围不一致时可能造成的后果. 从表中可以看出, 依赖范围误用可能造成许多种类的危害. 接下来我们将结合实例, 分别举例介绍依赖范围误用对模块自身运行和下游模块构建可能造成的危害和对应的具体触发场景.

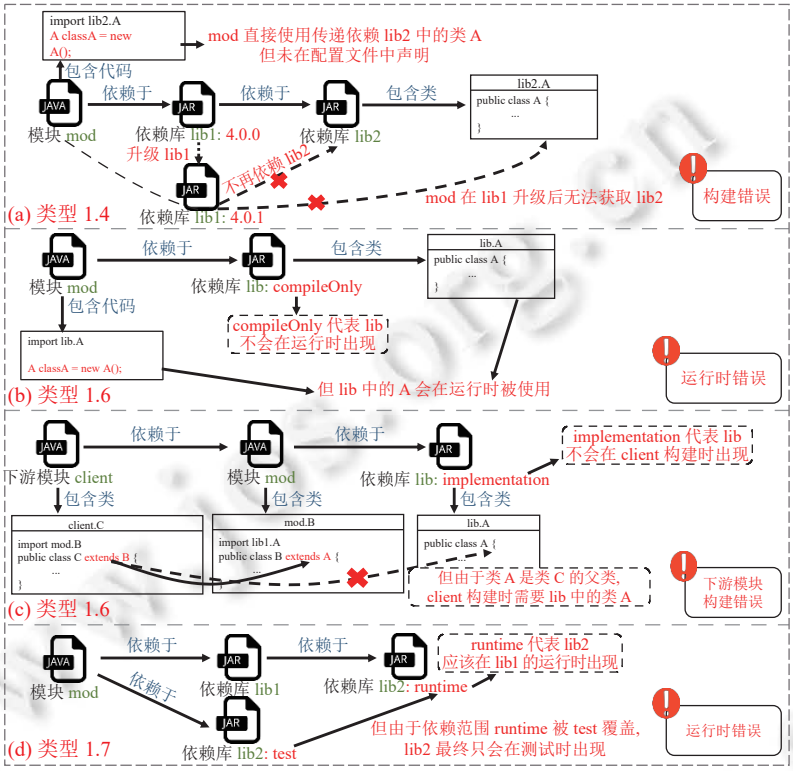

图 4　造成危害的依赖异味案例

表 5　Maven 中依赖范围误用的触发场景和危害

实际范围	预期范围			
	compile	runtime	provided	test
compile	—	2, 7	5, 9	2, 5, 7, 9
runtime	1	—	1	5, 9
provided	3, 6, 8	3, 8	—	2
test	1	3, 8	1	—

表 6　Gradle 中依赖范围误用的触发场景和危害

实际范围	预期范围				
	api	implementation	runtimeOnly	compileOnly	test
api	—	7	2, 7	5, 9	2, 5, 7, 9
implementation	6	—	2, 7	5, 9	2, 5, 7, 9
runtimeOnly	1	1	—	1	5, 9
compileOnly	3, 8	3, 8	3, 8	—	2
test	1	1	3, 8	1	—

(1) 运行时错误: 触发场景为依赖库预期依赖范围是 implementation, 而实际依赖范围是 compileOnly. 某依赖库的预期依赖范围是 implementation, 这意味着在真实情况下, 它不仅参与构建过程, 还应被包含在运行时环境中. 然而, 如果该库的实际依赖范围被设置为 compileOnly, 这就意味着它仅在构建阶段被调用, 而不会出现在运行时出现. 这种配置错误会导致在运行时无法访问该依赖库的功能, 从而引发运行时错误. 在图 4(b) 对应的#PMTS-24[41]中, 模块 Payment-service 将依赖库 commons-codec:commons-codec 的依赖范围设置为 compileOnly, 但事实上此依赖库在构建和运行时都被使用, 其预期范围应为 implementation. 此时, 项目可以正常进行构建, 但是在运行时试图访问 commons-codec:commons-codec 中的类 DigestUtils 时出现异常 NoClassDefFoundError.

(2) 下游模块构建错误: 触发场景为下游模块构建时使用依赖库中特性, 且依赖库预期依赖范围是 api, 而实际依赖范围是 implementation. 具体来说, 某依赖库的预期依赖范围是 api, 这意味着它不仅在构建过程中被使用, 还会作为传递依赖被引入到所有依赖它的下游模块中. 然而, 如果该库的实际依赖范围被设置为 implementation, 这就意味着它不会作为传递依赖被引入下游模块的构建中. 这种配置错误会导致下游模块在构建时无法访问依赖库的特性, 从而引发构建错误. 在图 4(c) 对应的#JAVACLIENT-2019[42]中, 模块 Java-client 将依赖 org.seleniumhq.selenium:selenium-support:5.0 的依赖范围设置为 implementation. 对于使用 Java-client 的下游模块来说, 这个范围意味着 selenium-support 不会作为传递依赖被引入 client 的构建中. 然而, 由于 Java-client 中的类 AppiumFluentWait 继承了 selenium-support 中的类 FluentWait, 下游模块 client 在试图使用 AppiumFluentWait 时也必须能够访达 FluentWait, selenium-support 的预期范围应该是 api. 此时, 由于依赖库 selenium-support 中的 FluentWait 未被引入, 下游模块构建错误.

3) 类别 1.7 依赖范围冲突: 直接依赖在依赖树中存在冲突的依赖范围 (2/47). 与依赖范围误用类似, 不同依赖范围之间的冲突可能造成不同的危害. 我们将最终结果展示在表 7 中, 其中, 表列代表直接依赖的依赖范围, 表行代表被覆盖的传递依赖的依赖范围, 表的内容是对应依赖范围出现冲突时的后果. 可以看到, 与依赖范围误用不同, 依赖范围冲突不会对下游模块造成危害, 因为 Maven 中的漏洞仅会影响模块自身依赖解析. 此外, 从表格的右上部分可以看出, 当直接依赖的使用场景包含了传递依赖的使用场景时也不会造成危害. 但依赖范围冲突同样可能对模块自身造成危害. 接下来我们将结合实例, 介绍依赖范围冲突引发的运行时错误问题.

表 7 Maven 中依赖范围冲突的触发场景和危害

直接范围	传递范围			
	compile	runtime	provided	test
compile	—	—	—	—
runtime	1	—	1	—
provided	3	3	—	—
test	1, 3	3	1	—

(1) 运行时错误: 触发场景为模块运行时使用依赖库中特性, 且依赖库作为直接依赖的依赖范围是 test, 覆盖了其作为传递依赖的依赖范围 runtime. 具体来说, 某依赖库的依赖范围被设置为 test, 这意味着该库仅在测试阶段被引入. 然而, 如果该库在依赖树中同样以传递依赖形式存在, 其依赖范围为 runtime, 这意味着引入此传递依赖的依赖库在运行时需要该库的功能. 由于 test 范围会覆盖 runtime 范围, 最终导致该库仅在测试阶段被引入, 而不包含在运行时环境中. 这种配置错误会导致模块在运行时无法访问依赖库的功能, 从而引发运行时错误. 在图 4(d) 对应的以#COMMON-152[43]为例, 直接依赖 org.slf4j:slf4j-log4j12 的依赖范围被设置为 test, 但与此同时, 其在依赖树中存在另一个依赖范围 runtime, 但由于 test 会覆盖 runtime, 最终导致 slf4j-log4j12 仅会在测试时被引入. 在这种情况下, 模块运行时无法找到 slf4j-log4j12, 因此会导致项目运行时错误, 无法产生任何日志.

基于以上发现, 我们回答研究问题 RQ2 如下: 依赖异味可能在多种使用场景中被触发, 进而导致诸如构建错误和运行时错误等 11 类危害. 更重要的是, 依赖异味的影响远不止于单一项目范畴, 它还可能在 Java 生态系统内部引发一系列连锁反应. 因此, 及时识别并解决依赖异味对于保障项目开发的质量与促进 Java 生态系统的持续健康发展至关重要.

3 依赖异味检测

根据我们在 RQ2 中的发现, Java 项目中存在多种依赖异味, 它们可能在特定条件下被触发, 对项目造成各种危害. 因此, 及时发现项目中潜藏的依赖异味显得格外重要. 尽管现有工具可用于部分依赖异味的检测, 但它们主要针对单一构建工具中的几类依赖异味, 缺乏全面而综合的检测能力. 这促使我们设计并实现一个能够检测我们在 RQ1 中观察到的所有异味, 并且适用于 Maven 和 Gradle 项目的检测工具 JDepAna.

我们结合 RQ1 中识别的各类依赖异味及其特征, 以及 RQ2 中总结的不同触发场景, 为每种异味设计了相应的检测算法. 这些检测算法依赖于类似的数据输入, 因此能够将这些共同的数据特征抽象为构建工具无关的依赖模型. 依赖模型使得统一的检测算法能够在该模型上进行操作, 而无须关心底层构建工具的细节差异所带来的检测逻辑差异. 由于 Maven 与 Gradle 在依赖管理机制上存在显著差异, 若不合理隔离这些差异, 依赖异味的检测逻辑将需在两种构建工具中分别实现. 例如, 在处理依赖树的冲突时, Maven 优先使用距离目标模块路径最近的依赖版本; 而 Gradle 则倾向于选择最高版本来解决冲突. 此时如果没有依赖模型, 异味 1.8 依赖树冲突的检测逻辑需要根据两者特性分别实现. 此外, 依赖模型的抽象设计进一步提升了可扩展性, 它解耦了检测逻辑与具体构建工具, 使得在出现新的依赖异味时, 能够基于依赖模型提供的丰富信息高效实现检测算法, 并将其集成进 JDepAna 中, 而无须考虑构建工具复杂的底层机制.

最终基于依赖模型, 我们设计并实现了用于 Java 项目依赖异味检测的统一算法, 并开发了适配 Maven 和 Gradle 的检测工具 JDepAna, 将各类异味检测功能集成在一个可扩展的框架之下. 接下来, 我们将详细介绍 JDepAna 的依赖异味检测流程和实现细节.

3.1 依赖异味检测流程

我们将依赖异味检测流程展示在图 5 中, 主要可以分为两步: 首先对项目或模块的信息进行解析以构建依赖模型, 其次在构建好的依赖模型上运行各类异味对应的算法以进行依赖异味分析. 我们将从依赖模型构建和依赖异味分析两个方面进行介绍.

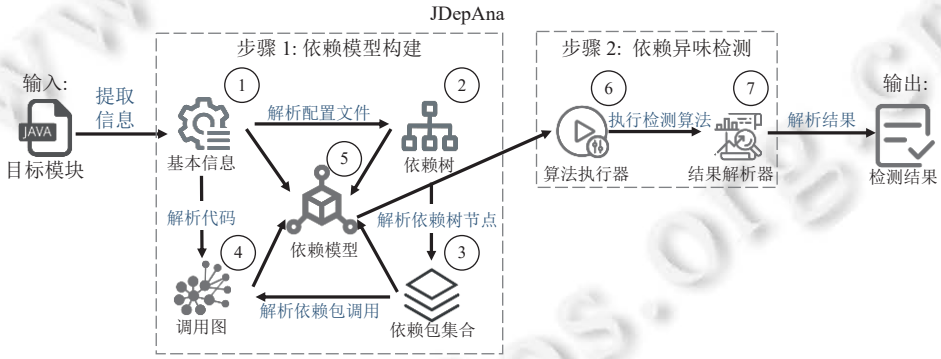

图 5 依赖异味检测流程

3.1.1 依赖模型构建

各类依赖异味检测算法主要以项目中 4 类依赖管理相关数据作为输入, 包括基本信息、依赖树、依赖包集合和调用图, 我们将它们抽象为依赖模型. 接下来我们分别介绍这 4 类数据和它们的作用.

1) 基本信息. 这类信息包括项目或模块的配置文件以及组织结构等原始数据. 其中包括源代码路径、测试代码路径、编译产出路径和依赖库缓存等信息. 这些数据为我们提供了项目或模块的基础信息, 为后续数据的获取提供了支持.

2) 依赖树. 依赖树是包含目标模块对应所有依赖库信息的树形结构. 在依赖树中, 包括了目标模块的直接依赖和传递依赖, 每个依赖库在依赖树中对应一个节点. 通过依赖树, 我们能够准确地定位依赖库及其出现的次序和深

度. 这对于诸如确定依赖库加载顺序等需求提供了重要的支持.

3) 依赖包集合. 依赖包集合是目标模块所有依赖库的具体构件组成的集合. 其中的每个元素对应了各个依赖库的具体实现, 以 JAR 文件的形式存储. 相较于依赖树, 依赖包集合更加关注每个依赖库的具体实现, 而不是其层次关系. 通过依赖包集合, 我们能够获取依赖库的具体实现, 进而深入分析其内容.

4) 调用图. 调用图表示目标模块对依赖库的使用情况. 在此, 我们仅关注目标模块对依赖库的调用, 其中调用图中的节点对应依赖库. 如果模块中存在对某个依赖库的调用, 则会存在连接从目标模块到对应依赖库的边. 在我们的调用图中, 存在 3 类调用边, 即构建调用、运行调用和测试调用, 它们分别代表目标模块在不同场景下对依赖库的使用. 通过调用图我们可以从代码的角度分析依赖库的预期使用场景, 并将其与依赖库在配置文件中声明的依赖范围所对应的实际使用场景进行对比. 值得注意的是, 调用图并不聚焦于具体的依赖范围, 而是关注 3 类主要使用场景——构建、测试和运行. 这一选择源于 Maven 与 Gradle 在依赖范围定义上的差异. 为了将调用图与特定的构建工具分离, 我们关注依赖范围的本质, 即对具体使用场景的划分. 例如, Maven 中的 provided 范围与 Gradle 的 compileOnly 范围虽然在名称上存在差异, 但二者均对应于构建和测试场景.

由于依赖模型中的数据之间存在一定依赖关系, 因此我们需要按照图 5 中的顺序来处理这些数据, 最终依赖模型构建的流程如下. 首先, 我们提取配置文件, 分析其中对项目或模块的配置, 得到基本信息. 其次, 我们对配置文件进行解析, 构建模块对应的依赖树. 然后, 我们对依赖树中的节点进行解析, 获取其对应的 JAR 包并组织成依赖包集合. 接着, 根据基本信息中对应的项目, 找到模块对应的源代码、测试代码和编译后的字节码, 使用静态分析来确定其中对外部库的调用, 这 3 类代码中对外部库的调用分别对应调用图中的构建调用、测试调用和运行调用, 从而形成了调用图. 最后, 我们将这些数据综合并组织为依赖模型.

3.1.2 依赖异味分析

从图 5 中可以看出, 依赖异味分析流程主要分为两步: 首先, 我们在依赖模型之上运行依赖异味检测算法; 其次, 我们对算法执行结果进行解析, 最终得到所有依赖异味. 我们针对每一类依赖异味设计了对应的依赖异味检测算法, 它们有着不同的核心逻辑, 但都以依赖模型中的部分数据为输入. 我们将各类检测算法核心和所需依赖模型中的数据列在表 8 中.

表 8 依赖异味检测算法简介

异味编号	核心逻辑	所需数据
1.1	判断模块构建产物和依赖库中的类是否存在交集	1, 3
1.2	判断依赖库之间类是否存在交集	3
1.3	判断依赖树中是否存在不同版本的同一依赖库	2
1.4	判断模块是否直接使用了传递依赖	2, 4
1.5	判断模块的直接依赖是否未被使用	2, 4
1.6	判断依赖库预期范围与实际范围是否一致	1, 2, 4
1.7	判断模块的直接依赖是否在依赖树中存在不同的范围	2
1.8	判断模块不同配置文件中依赖库版本是否一致	1, 2
2.1	判断项目对应路径下是否存在构建工具封装器脚本	1
2.2	判断项目对应路径下是否存在构建工具封装器 JAR	1
2.3	判断项目构建工具封装器 JAR 的校验和是否与官方一致	1
2.4	判断项目模块之间是否存在未统一管理版本的依赖库	1
2.5	判断项目模块之间是否声明不同版本的同一依赖库	1

由于篇幅限制, 我们以异味 1.6 依赖范围误用的检测算法 1 为例进行介绍, 更多检测算法见附录 A. 该算法接受模块的依赖树 *DT* 和调用图 *CG* 作为输入, 并输出所有出现依赖范围误用的依赖库集合 *JS*. 算法的工作流程如下: 首先, 初始化依赖库集合 *JS*, 并创建用于存储各个依赖库预期使用场景的哈希表 *expectedScenesMap* 以及模块在调用图中对应的节点 *source* (第 1–3 行). 接下来, 算法遍历调用图 *CG* 中所有以 *source* 为起点的调用边 *edge*, 并

记录相应的终点 *target* (第 4–5 行). 由于调用边 *edge* 的类型对应于从 *source* 到 *target* 的具体场景, 算法将 *edge* 的类型 *edge.type* 添加至 *target* 对应的预期使用场景 *expectedScenesMap*[*target*] (第 6 行). 然后, 算法对依赖树 *DT* 中深度为 1 的依赖库 (即直接依赖库)*dep* 进行遍历, 并提取模块中声明的 *dep* 的实际依赖范围 *declaredScope* (第 8–9 行). 随后, 利用 *getCorrespondingScenes* 函数获取依赖范围 *declaredScope* 对应的使用场景 *actualScenes* (第 10 行), 并从 *expectedScenesMap* 中取出预期依赖范围 *expectedScenes* (第 11 行). 通过比较依赖范围对应的使用场景 *actualScenes* 与预期使用场景 *expectedScenes*, 若存在不一致情况, 则将依赖库 *dep* 加入集合 *JS* 中 (第 12–13 行).

算法 1. 依赖范围误用检测.

输入: *DT*: 依赖树; *CG*: 调用图;
输出: *JS*: 出现依赖范围误用的依赖库集合.

1. *JS* ← {};
2. *expectedScenesMap* ← {};
3. *source* ← *CG.source*;
4. **for** *edge* ∈ *CG* **and** *edge.from* = *source* **do**
5. *target* ← *edge.to*;
6. *expectedScenesMap*[*target*].*add*(*edge.type*);
7. **end for**
8. **for** *dep* ∈ *DT* **and** *dep.depth* = 1 **do**
9. *declaredScope* ← *dep.scope*;
10. *actualScenes* ← *getCorrespondingScenes*(*declaredScope*);
11. *expectedScenes* ← *expectedScensMap*[*target*];
12. **if** *expectedScenes* != *null* **and** *actualScenes* != *expectedScenesMap* **then**
13. *JS.add*(*dep*);
14. **end if**
15. **end for**

总的来说, 此算法通过分析调用图中不同类型的调用边来确定依赖库的预期使用场景, 并将其与依赖库的实际依赖范围对应的使用场景进行比较, 输出所有不一致的依赖库. 以图 6 为例, Maven 项目中使用了依赖库 org.projectlombok:lombok, 并将其依赖范围设置为 compile. 依赖库 lombok[44]是一个能够通过在源代码中简单注解自动生成样板代码的依赖库. 由于它在构建时生成代码, 而不会出现在字节码中, 因此其依赖范围应该设置为 provided, 然而实际上却设置为 compile, 这就引发了依赖范围误用. 根据我们的检测算法, 由于 lombok 的类不会出现在字节码中, 因此调用图中不会存在指向 lombok 的运行调用边. 因此, 预期使用场景不包括运行时. 然而, 实际依赖范围为 compile 的使用场景却包括了运行时, 这就导致了不一致. 因此, 检测算法会将 lombok 标记为出现依赖范围误用的依赖库, 从而成功进行检测.

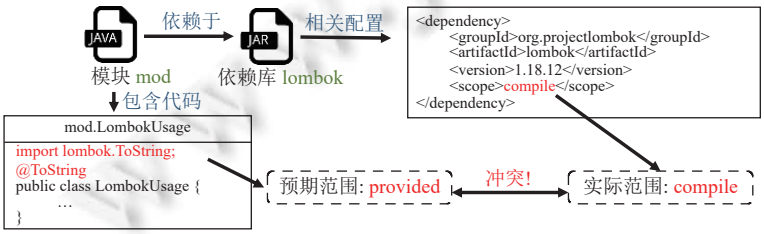

图 6 出现依赖范围误用的案例

3.2 检测工具实现

基于上述依赖异味检测流程, 我们开发了名为 JDepAna 的依赖异味检测工具. 它支持对前述所有依赖异味进行检测, 并且可以应用于 Maven 和 Gradle 项目.

由于 Maven 和 Gradle 在依赖模型的具体实现上存在显著的差异, 我们采取了如图 7 所示的结构进行实现, 以尽可能复用检测算法的同时, 确保 JDepAna 能够正常检测 Maven 和 Gradle 项目. 我们将依赖模型的定义和检测算法作为检测核心 JDepAna-core, 在其中定义依赖模型为抽象类, 包含获取依赖管理信息的接口. 各种检测算法可以通过该接口获取依赖模型数据, 实现自身的检测逻辑, 而无须关心依赖模型构建的细节. 此外, 我们以同样的 JDepAna-core 为核心, 分别实现了 Maven 和 Gradle 两种构建工具下的插件 JDepAna-maven-plugin 和 JDepAna-gradle-plugin. 在插件中, 我们根据 Maven 和 Gradle 依赖管理的不同特性, 分别对依赖模型中定义的函数进行重载, 以实现先前定义的获取依赖管理信息的接口.

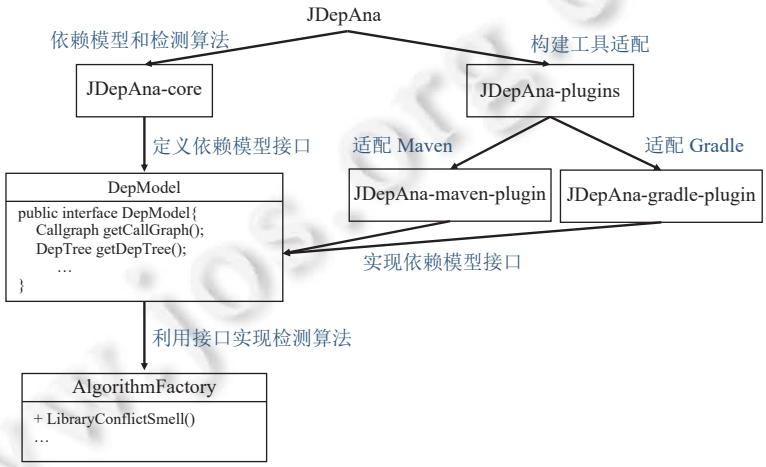

图 7 JDepAna 的实现

在实现对 Maven 和 Gradle 的适配过程中, 为了实现检测流程的完全自动化, 我们通过 Maven 和 Gradle 的插件特性, 介入 Maven 和 Gradle 的依赖解析过程, 将依赖异味检测功能像项目的构建任务一样无缝嵌入项目的自动化流程中, 从而实现了基本信息的提取、依赖树的解析以及依赖包集合的解析等功能. 而在构建调用图时, 我们主要采用了静态分析框架 Soot[45], 同时结合 JavaParser[46]和 Javassist[47]分别对源码和字节码进行静态分析.

4 实验评估

在实验中, 我们通过以下两个研究问题来评估 JDepAna 的有效性和实用性, 由于现有研究中缺少可以同时分析 Maven 和 Gradle 项目的工具, 我们更多地注重验证我们方法的检测能力和范围.

- RQ3 (有效性): JDepAna 能否检测出已知的依赖异味问题?
- RQ4 (实用性): JDepAna 能否在流行的 Java 项目中检测出新的依赖异味问题?

为了回答 RQ3, 我们选择了在第 3.1 节中收集的问题报告, 这些报告涉及实际识别的依赖异味问题. 基于这些报告, 我们构建了一个基准集, 并应用 JDepAna 工具于基准集中的相应项目和模块. 我们通过分析这些检测结果, 评估 JDepAna 是否能有效识别这些已知的依赖异味问题.

为了回答 RQ4, 我们收集了 GitHub 上广受欢迎的 Maven 和 Gradle 项目, 并使用 JDepAna 对这些项目进行分析, 以探测当前未被识别的依赖异味实例. 筛选并人工验证这些检测到的实例后, 我们从中挑选出具有代表性且具价值的问题报告, 并以 GitHub 问题报告的形式提交给开发者, 以便进一步确认.

值得注意的是, RQ3 专注于复现性验证, 即检验工具是否能准确地识别出已被记录并确认的依赖异味问题.

而 RQ4 则侧重于探索性发现, 旨在揭示 JDepAna 在实际、复杂的项目环境中对新问题的检测能力. 这两部分不仅相互形成对比, 还互为佐证, 全面展示了检测工具的性能.

4.1 RQ3: 有效性

4.1.1 实验设置

我们对实证研究中收集到的 47 个问题报告进行了进一步筛选, 排除其中两类不属于我们检测对象的问题报告: 第 1 类是报告中的依赖异味是下游项目使用中遇到的问题, 而不是当前项目; 第 2 类是由于时间跨度较大, 报告对应版本的项目已无法获取当时的依赖库, 导致无法进行正常构建和构建调用图. 最终我们得到了 28 个问题报告, 其中 16 个与 Maven 相关, 12 个与 Gradle 相关.

为了对 JDepAna 的检测结果进行精细评估, 我们从问题报告对应的修复中提取了相应的依赖异味实例. 这些实例被定义为包含依赖异味特征的最小单元, 以元组<项目, 模块, 依赖异味编号, 问题说明>来表示, 分别代表了异味出现的项目和模块、所属类别以及具体信息. 依赖异味实例的引入能够避免宽泛的检测标准, 确保工具不仅能识别问题的存在, 还能细致检测呈现出依赖异味特征的具体依赖库. 以图 8 中#ASJV-4539[48]为例, 项目 aws-sdk-java-v2 中的模块 core/auth 中出现了依赖异味未声明依赖, 其中两个依赖库 software.amazon.awssdk:http-auth-spi 和 soft-ware.amazon.awssdk:checksums 都被使用但未在配置文件中声明. 若没有依赖异味实例的细分, JDepAna 只需检测出未声明依赖即可视为成功. 然而, 基于依赖异味实例的划分, 该问题实际包含两个实例<aws-sdk-java-v2, core/auth, 1.4, software.amazon.awssdk:http-auth-spi>和<aws-sdk-java-v2, core/auth, 1.4, software.amazon.awssdk:checksum>, 这意味着 JDepAna 需分别检出这两个具体依赖库的未声明特征, 才能算作全面检测. 这些实例中 aws-sdk-java-v2 对应于项目名, core/auth 代表出现依赖异味的具体模块, 1.4 对应于未声明依赖的异味编号, software.amazon.awssdk:http-auth-spi 和 soft-ware.amazon.awssdk:checksums 对应于问题说明也即被使用但未被声明的具体依赖库. 值得注意的是, 不同类别的依赖异味虽然都会被划分为依赖异味实例, 但在问题说明中的描述方式有所不同. 例如对于外部类冲突这类异味, 当多个依赖库中存在冲突类时, 问题说明会列出所有包含这些冲突类的依赖库, 并将其作为一个实例处理; 而在异味 1.4 未声明依赖中, 问题说明则对应于使用但未声明的具体依赖库.

图 8 依赖异味实例

按照以上流程, 我们最终提取出了 16 个 Maven 相关问题报告中的 125 个实例和 12 个 Gradle 相关问题报告中的 22 个实例, 这 147 个实例构成了我们的基准集, 覆盖了 13 个异味类别中的 10 类. 对于基准集中的依赖异味

实例, 我们在实例对应的项目或模块中运行 JDepAna, 收集实例对应依赖异味类别的检测结果. 将检测结果同样以依赖异味实例的形式表示, 最终得到 Maven 中的 192 个实例和 Gradle 中的 40 个实例, 这 232 个实例构成了我们的检测结果集. 值得注意的是, 由于实证研究中的部分问题报告不属于我们的检测对象, 最终有 3 类依赖异味不存在于基准集中, 但 JDepAna 对这些类别同样能有效进行检测, 我们将在 RQ4 中讨论其对这些类别的检测能力.

4.1.2 评估指标

我们用召回率 (*recall*) 和精确率 (*precision*) 来评估 JDepAna 的有效性, 包括以下指标.

- 基准真阳 (*TPB*): 被检测为依赖异味, 且存在于基准集中.
- 真阳 (*TP*): 被检测为依赖异味, 且存在于基准集中或不在基准集中但人工验证为真.
- 假阳 (*FP*): 被检测为依赖异味, 但不存在于基准集中且人工验证为假.
- 假阴 (*FN*): 未被检测为依赖异味, 但存在于基准集中.

基于以上指标, 我们如下定义召回率和精确率, 精确率能评估 JDepAna 是否能准确地检测项目中实际存在的依赖异味, 召回率则评估 JDepAna 是否能检测基准集中的所有依赖异味问题.

$$Precision = TP/(TP+FP) \tag{1}$$

$$Recall = TPB/(TPB+FN) \tag{2}$$

4.1.3 实验结果

我们将综合 Maven 和 Gradle 后的实验结果展示在表 9 中. JDepAna 的召回率达到 95.9%, 精确率达到 96.1%. 表 10 和表 11 中展示了 Maven 和 Gradle 中的实验结果. 可以看到, Maven 基准集中的 125 个实例中有 120 个实例被检出, Gradle 基准集中的 22 个实例有 21 个实例被检出, 召回率分别为 96% 和 95.4%, 精确率分别为 95.3% 和 100%.

表 9 整体检测结果

异味类别	基准集	检测结果集	基准集中的检出数 (召回率 (%))	不在基准集中的检出数 (占比 (%))	
				人工验证为真的检出数	人工验证为假的检出数
1.1	1	1	1 (100)	0	0
1.2	1	2	1 (100)	1 (100)	0
1.3	5	23	5 (100)	18 (100)	0
1.4	56	68	56 (100)	12 (100)	0
1.5	13	49	13 (100)	27 (75)	**9 (25)**
1.6	23	19	17 (73.9)	2 (100)	0
1.7	0	0	0	0	0
1.8	6	6	6 (100)	0	0
2.1	4	4	4 (100)	0	0
2.2	2	2	2 (100)	0	0
2.3	0	0	0	0	0
2.4	36	58	36 (100)	22 (100)	0
2.5	0	0	0	0	0
合计	147	232	**141 (95.9)**	**82 (90.1)**	9 (9.9)

表 10 Maven 项目检测结果

异味类别	基准集	检测结果集	基准集中的检出数 (召回率 (%))	不在基准集中的检出数 (占比 (%))	
				人工验证为真的检出数	人工验证为假的检出数
1.1	0	0	0	0	0
1.2	1	2	1 (100)	1 (100)	0
1.3	2	3	2 (100)	1 (50)	0
1.4	56	68	56 (100)	12 (100)	0
1.5	13	49	13 (100)	27 (75)	**9 (25)**
1.6	17	12	12 (71)	0	0
1.7	0	0	0	0	0

表 10 Maven 项目检测结果 (续)

异味类别	基准集	检测结果集	基准集中的检出数 (召回率 (%))	不在基准集中的检出数 (占比 (%))	
				人工验证为真的检出数	人工验证为假的检出数
1.8	0	0	0	0	0
2.1	0	0	0	0	0
2.2	0	0	0	0	0
2.3	0	0	0	0	0
2.4	36	58	36 (100)	22 (100)	0
2.5	0	0	0	0	0
合计	125	192	**120 (96)**	**63 (87.5)**	9 (12.5)

表 11 Gradle 项目检测结果

异味类别	基准集	检测结果集	基准集中的检出数 (召回率 (%))	不在基准集中的检出数 (占比 (%))	
				人工验证为真的检出数	人工验证为假的检出数
1.1	1	1	1	0	0
1.2	0	0	0	0	0
1.3	3	20	3 (100)	17 (100)	0
1.4	0	0	0	0	0
1.5	0	0	0	0	0
1.6	6	7	5 (71.4)	2 (100)	0
1.7	0	0	0	0	0
1.8	6	6	6 (100)	0	0
2.1	4	4	4 (100)	0	0
2.2	2	2	2 (100)	0	0
2.3	0	0	0	0	0
2.4	0	0	0	0	0
2.5	0	0	0	0	0
合计	22	40	**21 (95.4)**	**19 (100)**	0

● 假阴分析. 我们进一步分析基准集中未被检出的异味实例 (假阴), 发现它们都属于异味 1.6 依赖范围误用. 未被检出的主要原因是项目中存在其他未解决的异味问题, 导致 JDepAna 错误地将对应的异味认定为其他类型的异味, 从而导致假阴. 举例来说, #INLONG-4771[49]中的模块 inlong-manager/manager-plugins, 其依赖库 org.junit.jupiter:junit-jupiter 出现了异味 1.6 依赖范围误用, 预期范围应为 test, 但实际范围为 compile. 然而, 该模块中还存在与 junit-jupiter 相关的异味 1.4 未声明依赖和 1.5 未使用依赖的特征. 因此, JDepAna 会优先检测依赖库 org.junit.jupiter:junit-jupiter 是否出现异味 1.5, 但在解决了异味 1.4 和 1.5 之后, JDepAna 可以正常检出异味 1.6.

● 假阳分析. 我们进一步分析了不在基准集中但在检测结果集的实例. 在 Maven 中, 这类实例共计 72 个, 其中 9 个 (12.5%) 经人工验证为假; 而在 Gradle 中, 这类实例共 19 个, 经人工验证均为真. 对于被验证为真的实例, 它们是在提出问题报告时仍然存在但尚未被发现的问题, 但由于问题报告对应的项目版本与项目当前版本有较大的差异, 我们没有寻求开发者的反馈, 开发者反馈会在 RQ4 中讨论. 而对于被验证为假的 9 个实例 (假阳), 我们发现它们都属于异味 1.5. 主要原因在于异味 1.5 的检测对项目调用图的精度要求较高. 如果项目中大量使用诸如反射和动态特性等功能, 静态分析的缺陷可能导致调用图与实际不一致, 从而使得误报更容易发生. 以项目 Hapi-Fhir-Jpaserver-Starter[50]为例, 7 个假阳实例均出现在此项目中. 主要是因为项目使用了 Spring 框架[51], 而 Spring 框架大量使用 Java 的反射和动态类加载等特性, 导致静态分析无法准确分析.

基于以上发现, 我们回答研究问题 RQ3 如下: 在我们收集的包含 147 个依赖异味实例的基准集中, JDepAna 达到了 95.9% 的召回率和 96.1% 的精确率, 在确保高精确度的同时保持了较高的召回率, 有效识别并报告了项目中的依赖异味.

4.2 RQ4: 实用性

4.2.1 实验设置

为了回答 RQ4, 我们首先从 GitHub 中选取了符合以下标准的 Maven 和 Gradle 项目各 100 个: (1) 项目的 Star 数目大于 100; (2) 项目在最近 1 个月里有更新 (2023 年 7–8 月); (3) 项目包含的测试数量大于 10; (4) 项目为常规 Java 项目 (不是 Spring 或者 Android 项目). 在排除了实验环境下无法正常构建和无法正常使用 Soot 构建调用图的项目后, 最终我们得到了 73 个 Maven 项目和 51 个 Gradle 项目. 我们将这些项目及其所有子模块作为我们的实验对象后, 最终得到 73 个 Maven 项目和对应的 1 614 个模块, 以及 51 个 Gradle 项目和对应的 416 个模块. 这些项目的信息如表 12、表 13 所示, 可以看到这些项目都相当流行并且有一定规模. 我们在所有实验对象中运行了 JDepAna, 并收集了检测结果, 最终结果如表 14–表 15 和后文表 16 所示, JDepAna 共检出 30 689 个依赖异味实例, 其中 Maven 和 Gradle 项目中分别检出 21 417 和 9 272 个. 由于最终异味实例数量众多, 为了进行人工验证, 我们从中进行了采样. 我们挑选能通过所有自带测试的模块和对应项目, 收集其检测结果中的异味实例并进行复杂度排序, 根据各个异味类别实例的数量, 按一定比例选取复杂度较大的异味实例. 最终在 Maven 和 Gradle 中分别采样到 188 个和 172 个异味实例, 我们对这些异味实例进行人工验证.

表 12 Maven 项目信息 (k)

统计指标	Star 数目	Fork 数目	代码行数
平均值	**3.8**	**1**	**650**
最大值	27	7.3	26 532
最小值	0.3	0.1	1.5

表 13 Gradle 项目信息 (k)

统计指标	Star 数目	Fork 数目	代码行数
平均值	**3.3**	**0.5**	**199.5**
最大值	47.3	7.8	1 495
最小值	0.1	0.03	2.7

表 14 整体检测结果

异味类别	异味数目
1.1	70
1.2	4 048
1.3	11 160
1.4	4 970
1.5	7 932
1.6	1 021
1.7	818
1.8	2
2.1	56
2.2	62
2.3	25
2.4	395
2.5	130
合计	**30 689**

表 15 Maven 检测结果

异味类别	异味数目
1.1	65
1.2	3 035
1.3	8 566
1.4	3 099
1.5	4 738
1.6	587
1.7	818
1.8	2
2.1	55
2.2	61
2.3	0
2.4	346
2.5	45
合计	21 417

4.2.2 评估指标

我们使用公式 (1) 中定义精确率来评估 JDepAna 的实用性, 具体指标定义如下.

- 真阳 (TP): 被检测为依赖异味且人工验证为真.
- 假阳 (FP): 被检测为依赖异味且人工验证为假.

4.2.3 实验结果

我们将综合 Maven 和 Gradle 检测结果后的实验结果展示在表 17 中, 可以看到 JDepAna 检出的 360 个依赖异味实例中有 346 个被人工验证为真, 精确率为 96.1%. 表 18 和表 19 对应于 Maven 和 Gradle 中的实验结果, 在 Maven 项目的 188 个异味实例中, 有 180 个 (95.7%) 被人工验证为真; 而在 Gradle 项目的 172 个异味实例中, 有 166 个 (95.6%) 被人工验证为真. 值得注意的是, 虽然 RQ3 数据集中缺失了 3 类依赖异味 (1.7、2.3、2.5), 但这些异味在 RQ4 的数据集中均有出现, 并且都被人工验证为真, 这进一步说明了 JDepAna 具备有效的检测能力.

表 16 Gradle 检测结果

异味类别	异味数目
1.1	5
1.2	1 013
1.3	2 594
1.4	1 871
1.5	3 194
1.6	434
1.7	0
1.8	0
2.1	1
2.2	1
2.3	25
2.4	49
2.5	85
合计	9 272

表 17 整体人工验证结果

异味类别	检测结果集	真阳实例数 (精确率 (%))
1.1	15	15 (100)
1.2	45	45 (100)
1.3	41	41 (100)
1.4	47	47 (100)
1.5	45	32 (71.1)
1.6	32	31 (96.9)
1.7	13	13 (100)
1.8	2	2 (100)
2.1	11	11 (100)
2.2	12	12 (100)
2.3	16	16 (100)
2.4	49	49 (100)
2.5	31	31 (100)
合计	359	**345 (96.1)**

表 18 Maven 项目人工验证结果

异味类别	检测结果集	真阳实例数 (精确率 (%))
1.1	10	10 (100)
1.2	23	23 (100)
1.3	18	18 (100)
1.4	18	18 (100)
1.5	24	16 (66.7)
1.6	14	14 (100)
1.7	13	13 (100)
1.8	2	2 (100)
2.1	10	10 (100)
2.2	11	11 (100)
2.3	0	0
2.4	24	24 (100)
2.5	21	21 (100)
合计	188	**180 (95.7)**

表 19 Gradle 项目人工验证结果

异味类别	检测结果集	真阳实例数 (精确率 (%))
1.1	5	5 (100)
1.2	22	22 (100)
1.3	23	23 (100)
1.4	29	29 (100)
1.5	21	20 (95.2)
1.6	18	13 (72.2)
1.7	0	0
1.8	0	0
2.1	1	1 (100)
2.2	1	1 (100)
2.3	16	16 (100)
2.4	25	25 (100)
2.5	10	10 (100)
合计	171	**165 (96.5)**

● 假阳分析. 我们进一步分析出现检测结果中被人工验证为假的实例 (假阳), 发现它们主要集中在异味类别 1.5 和 1.6 中. 其中, 有 13 个类别为 1.5 的假阳实例, 其产生原因与我们在 RQ3 中分析的原因类似, 都是由于动态特性的使用导致项目调用图的构建不够精确; 仅有 1 个类别为 1.6 的假阳实例, 其产生原因主要是项目 Btrace[52] 添加了额外的自定义脚本进行检查以使得依赖库满足其特殊需求, 而 JDepAna 只对常规代码进行分析, 因此出现误报.

● 开发者反馈. 尽管 JDepAna 可以自动化检测依赖异味问题, 但向开发人员报告这些问题涉及大量的人工工作, 例如与开发人员沟通及帮助他们提交修复等. 因此, 我们从人工验证为真的实例中挑选出有代表性且有价值的部分实例, 将它们以问题报告的形式提交给开发者. 问题报告中包括了问题描述、可能危害以及可能的解决措施. 最终, 我们提交了 25 个问题报告, 其中包含了 48 个实例 (一个问题报告中可能包含多个实例), 其中 22 个问题报告中的 42 个实例已经被确认, 16 个问题报告中的 21 个实例已经被修复. 我们将结果展示在表 20 中.

绝大多数开发者对我们提交的问题报告都给予了正面反馈. 我们在下面展示了一些例子, 这些例子对应了不同的异味类别, 说明了我们检测工具的全面性. 在 Issue #2189 (异味 1.3)[53]中, 开发者在我们的帮助下找到了问题并成功解决, 对我们的帮助表示了感谢: “I found the issue with your help and here is the PR to get the issue resolved. Thanks a lot for your help.” 在 Issue #2001 (异味 1.4)[54]中, 开发者先前并不了解此类异味, 但我们的报告促使开发

者深入思考了这类问题: “It was a good discussion and I never got a chance to think about it in depth. Thanks.” 在 Issue #2021 (异味 2.1 和异味 2.2)[55]中, 开发者感谢我们的帮助, 并提到通过使用构建封装器生成新项目为用户提供了很大的便利: “Thanks for your help. We generate new projects with the Maven wrapper which is very convenient for users.” 在 Issue #1655 (异味 2.3)[56]中, 开发者感谢我们指出问题, 并承认这个问题可能表明他们的构建工具封装器的过时或错误设置: “Thanks for highlighting the checksum mismatch with our ‘gradle-wrapper.jar’. You’re right—This discrepancy suggests that our wrapper might be outdated or incorrectly set up.” 在 Issue #2546 (异味 2.4 和异味 2.5)[57]中, 开发者对依赖管理的改进展示出兴趣, 并希望进一步查看其他可能改进: “I like the improvement for the dependency version management. I do want to take a closer look into other pom files that might be able to take advantage of this.” 这些总结展示了开发者对我们检测工具的正面反馈, 并强调了我们的工具在发现和解决问题中的有效性.

表 20 提交的问题报告

问题报告 (项目名, 报告编号)
apache/seatunnel, #6772▲; apache/tinkerpop, #2546♠; line/armeria, #5581♠; GoogleContainerTools/jib, #4234▲; Discord4J/Discord4J, #1200★; networknt/light-4j, #2189★; alkacon/opencms-core, #787★; tchiotludo/akhq, #1657★; SerCeMan/jnr-fuse, #181★; MobilityData/gtfs-validator, #1655★; wiremock/wiremock, #2572★; btraceio/btrace, #668★; javapathfinder/jpf-core, #432★; networknt/light-4j, #2021★; LibrePDF/OpenPDF, #995♠; apache/flink-cdc, #2753♠; stleary/JSON-java, #829★; networknt/light-4j, #2001♠; apache/tinkerpop, #2349★; TooTallNate/Java-WebSocket, #1370★; oracle/opengrok, #4471★; apache/httpcomponents-client, #500★; networknt/light-4j, #1948★; googleapis/google-cloud-java, #10037♠; MobilityData/gtfs-validator, #1725▲;

★: 依赖异味实例已被确认并修复; ♠: 依赖异味实例被确认但尚未被修复; ▲: 问题报告仍在处理中

基于以上发现, 我们回答研究问题 RQ4 如下: JDepAna 在广泛使用的 Java 项目中达到了 96.1% 的精确率, 我们据此向开发者报告了 48 例检测出的依赖关系实例, 其中 42 例 (87.5%) 得到了确认, 21 例 (43.8%) 已被快速修复. 开发者的积极反馈反映了 JDepAna 识别问题的准确性, 也进一步证实了该工具在实际应用中的有效性和实用性.

5 相关研究工作

5.1 软件依赖管理

软件依赖管理是软件开发过程中至关重要的一个方面, 其挑战和问题随着需求变化以及软件库规模和复杂性的增加而不断增加[58−60]. 由于 Java 语言拥有成熟的构建工具和丰富的依赖库, 其依赖管理问题备受关注[7−11,18,22,28,29,61−64]. Jezek 等人[62]提出了 Maven 在解析传递依赖过程中可能导致的冗余、不一致和重复等问题. Wang 等人[8]系统研究 Maven 项目中的依赖冲突问题, 提出工具 DECCA 以对依赖冲突问题的严重程度进行评估, 并提出 RIDDLE[9]和 SENSOR[7]以自动生成测试用例触发依赖冲突导致的语义冲突和程序崩溃. Soto-Valero 等人[10,11,22]针对 Maven 项目中冗余的依赖关系进行定性和定量分析并提出 DEPCLEAN 和 DEPTRIM 用以检测和消除冗余的依赖库. Huang 等人[17]针对 Maven 项目中模块之间依赖库版本冲突的问题进行调研, 并提出 LIBHARMO 以自动化解决这类问题. 本文继承并深入探讨了现有研究中提及的相关问题, 更进一步将视野延伸至涉及 Gradle 项目, 以全面地理解 Java 项目的依赖管理. Yang 等人[63]针对 Maven 项目中依赖范围设置错误的根因进行调研并给出依赖范围设置的建议. 其工作针对 Maven 中依赖范围误用开展实证研究, 但并未提供对应的检测方法, 而本文提出了 JDepAna 以实现对这些问题的自动化检测, 并考虑了 Gradle 项目中更为复杂的依赖范围. Vassallo 等人[28]提出 CD-Linter, 一种语义检查工具, 其能够自动识别持续交付 (continuous delivery) 配置文件中的异味, 并通过大规模长期研究验证了其有效性和可行性. Zhang 等人[29]提出 BuildSonic, 其通过分析 Java 项目的持续集成 (continuous integration) 基础设施中的配置文件检测并修复 15 种持续集成中的异味. Weeraddana 等人[65]分析了冗

余依赖项更新导致持续集成时间浪费, 发现 55.88% 的依赖项更新相关构建时间是无效的, 并提出 Dep-sCImitar 工具, 通过跳过无效构建减少了 68.34% 的时间浪费. 尽管这些研究也关注于 Java 项目的异味, 但主要侧重于项目的持续集成和持续交付, 而我们则着眼于项目开发过程中的依赖管理. 总的来说, 目前关于 Java 依赖管理的研究主要集中在 Maven 上, 而本文则致力于综合分析 Java 项目中两种最常用的构建工具 Maven 和 Gradle, 全面总结了这两类项目可能存在的依赖管理问题. 通过这种综合分析, 本文对现有研究进行了继承与扩展, 从而更深入地探究 Java 项目中的依赖管理问题.

在 Java 之外, 其他语言如 Python 和 JavaScript 同样拥有广泛的用户群体和庞大的依赖库规模, 所以其依赖管理问题也得到了广泛的研究[13,14,18,21,23−25,66−69]. Cao 等人[13]和 Jafari 等人[14]分别对 JavaScript 项目和 Python 项目中可能存在的依赖异味进行了调研, 并发现依赖异味在这两类语言中同样普遍存在. 本文的研究目的与前述工作相契合, 但由于 Java 是静态类型语言, 其依赖管理机制更为严格. 且相较于 Python 和 JavaScript 的构建工具, Maven 和 Gradle 在语法和配置方式上存在着显著的不同. 因此, 本文特别结合了 Maven 和 Gradle 的特性, 对可能在 Java 项目中出现的依赖异味进行了针对性的适配和扩展. Patra 等人[18]提出了 ConflictJS 以分析在客户端 Web 应用中常用的第三方 JavaScript 库之间的冲突. 文献 [23] 对 PyPI 生态系统中 Python 项目的依赖冲突问题进行调研, 并提出 WATCHMAN 以持续监控并报告潜在的依赖冲突. 文献 [25] 提出了 LooCo, 其通过自动放宽 Python 项目的库版本约束来减少依赖冲突导致的构建错误问题. Vázquez 等人[21]通过从 JavaScript 应用中删除冗余功能来分析捆绑在 JavaScript 应用中的第三方软件库, 以提高网页加载速度. Drosos 等人[66]对 PyPI 生态系统中的冗余依赖现象进行了大规模细粒度分析, 发现超过 50% 的依赖存在冗余问题, 且 15% 的漏洞集中在这些冗余代码中. Peng 等人[67]提出了 PyConf 工具, 针对 PyPI 生态系统中的配置问题进行源码级检测, 发现 68% 的问题无法通过现有版本级检查发现, 并且现有的依赖推断工具在处理依赖冲突和依赖库缺失时存在明显不足. Qian 等人[24]提出了 Slimium 框架, 通过静态、动态和启发式分析的混合方法, 对 Chromium 浏览器进行代码去膨胀, 以减少安全攻击面, 实验证明该框架能够在 40 个流行网站上平均去除 94 个 (61.4%) CVE, 削减 23.85 MB 的代码. 前述工作主要关注于 Python 和 JavaScript 项目中的依赖冲突和冗余依赖问题, 这类问题同样会在 Java 项目中出现, 说明了这些问题的普遍性因此, 本文也对 Java 项目中可能出现的这类问题进行了讨论.

5.2 开源软件生态

随着开源软件生态的不断发展, 流行的软件库往往关系着一系列下游项目, 任何问题都很容易传播造成广泛影响[70]. 因此, 研究软件库在生态系统中的影响至关重要[71,72]. Ochoa 等人[73,74]发现大多数破坏性变更 (breaking change) 影响的代码并未被任何下游项目使用, 而仅有 7.9% 的下游项目受到破坏性变更的影响. Jayasuriya 等人[75,76]发现传递依赖的更新是引入破坏性变更的主要因素, 并且几乎一半的破坏性变更违反了语义化版本 (semantic versioning), 并发现 2.30% 的变更引发了下游项目的测试失败. Zhang 等人[77]则提出了 Sembid 工具以检测依赖库中不同版本接口语义不一致的问题. Dann 等人[78]提出了 UPCY, 其通过对依赖树的图操作帮助开发人员在更新项目依赖时减少不兼容性问题. Li 等人[79]探讨了 Go 语言生态系统第三方库升级中的破坏性变更问题, 并发现检测工具和数据集的缺乏使得开发者难以识别和应对这些变更的影响. 前述研究聚焦于项目接口的动态更新过程, 着眼于随着时间推移, 项目如何不断变化以适应新的需求和环境. 相比之下, 本文专注于如何在项目的特定状态下, 通过消除依赖管理中潜藏问题以确保项目的稳定和安全. Wu 等人[80]通过收集大量漏洞实例以分析上游项目中漏洞对下游项目的潜在威胁. Zhang 等人[19,81]针对 Maven 生态系统中上游项目阻塞补丁导致漏洞持久化的问题进行调研, 并提出工具 RANGER 和 CORAL 以自动将依赖库恢复至安全版本. Mir 等人[82]探讨了依赖的传递性和调用图粒度对 Maven 生态系统中漏洞传播分析精度的影响. Pashchenko 等人[83]提出 Vuln4Real 以解决学术界和工业界在报告开源软件中易受攻击的依赖关系时存在的夸大问题. Zhao 等人[20]提出了综合评估模型, 针对 Java 的软件成分分析工具进行了依赖解析能力和漏洞检测效果的全面评估. Fourné等人[84]聚焦于软件供应链安全, 探讨了可重复构建 (reproducible build) 如何为构建工具提供强有力的防御基础. Keshani 等人[85]开发了自动化工具 AROMA, 通过自动查找库的源代码和原始发布环境信息, 以提高 Maven 生态系统的可重复构建性. 类似的,

Gao 等人[86]提出了 PyRadar 框架, 通过结合元数据和源代码分发, 显著提高了源代码仓库信息的检索和验证准确性. Hu 等人[87]研究了 Golang 生态系统中的漏洞生命周期, 发现 66.10% 的模块受漏洞影响, 漏洞修复传播过程中存在两种滞后现象, 通过及时发布和索引补丁版本, 可以显著提升生态系统的安全性. Guo 等人[88]分析了 PyPI 生态系统中的恶意代码生命周期, 收集了 4 669 个恶意包文件, 发现超过 50% 的恶意代码具有多重恶意行为, 74.81% 的恶意包通过源码安装成功进入终端用户项目, 且 72% 以上的恶意包在发现后仍长期存在于 PyPI 镜像服务器中. 前述研究着重关注于检测项目源代码和所使用依赖库中存在的漏洞以及探究这些漏洞对整个生态系统的影响. 然而即便项目中不存在漏洞, 依赖管理中的潜藏问题仍然可能对项目和生态造成严重危害, 本文通过检测这些潜藏问题以确保 Java 生态系统的持续健康发展.

6 总结和展望

本文聚焦于 Java 项目在使用 Maven 和 Gradle 进行依赖管理时出现的依赖异味问题. 我们通过开展大规模实证研究, 综合分析了开源社区、官方文档以及学术论文中关于依赖管理的各类问题, 最终发现了 13 类特征各异的依赖异味, 并探究了它们的触发根源和潜在危害. 基于实证研究成果, 我们提出了统一的依赖异味检测算法, 并开发了适用于 Maven 和 Gradle 的依赖异味检测工具 JDepAna. 经过实验评估, JDepAna 在已知依赖异味的检测中达到了 95.9% 的召回率; 在流行的 Java 项目中, 其准确率达到了 96.1%. 我们从新检出的依赖异味中进一步汇报了 48 个实例给开发者, 其中 42 个已被确认, 21 个已被修复, 充分证明了我们 Java 依赖异味检测算法和工具的效果和实用性. 未来, 我们计划扩展 JDepAna 的检测功能, 使其适用于使用 Gradle 进行依赖管理的 Android 项目. 此外, 我们希望设计并实现一套自动化技术, 在 JDepAna 报告信息的基础上为 Java 项目提供依赖异味自动修复功能. 也值得说明的是, Java 项目中的依赖管理机制也类似存在于其他编程语言中, 因此本文提出的依赖异味可以通过扩展和适配应用于其他语言的依赖管理. 我们希望未来能够进一步拓展 JDepAna 的功能, 使其不仅适用于 Java 项目, 还能为其他语言的依赖管理提供有效的检测和修复工具, 从而提高更广泛的软件项目的质量和可靠性.

References:

[1] TIOBE Index. TIOBE. 2024. https://www.tiobe.com/tiobe-index/

[2] Krueger CW. Software reuse. ACM Computing Surveys, 1992, 24(2): 131–183. [doi: 10.1145/130844.130856]

[3] Cox R. Surviving software dependencies: Software reuse is finally here but comes with risks. Queue, 2019, 17(2): 24–47. [doi: 10.1145/3329781.3344149]

[4] Gkortzis A, Feitosa D, Spinellis D. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. Journal of Systems and Software, 2021, 172: 110653. [doi: 10.1016/j.jss.2020.110653]

[5] No any ASM service available [Moxy 4.0.1 in Jersey]. #1849, 2024. https://github.com/eclipse-ee4j/eclipselink/issues/1849

[6] Eclipse Jersey. 2024. https://projects.eclipse.org/projects/ee4j.jersey

[7] Wang Y, Wu RX, Wang C, Wen M, Liu YP, Cheung SC, Yu H, Xu C, Zhu ZL. Will dependency conflicts affect my program's semantics? IEEE Trans. on Software Engineering, 2022, 48(7): 2295–2316. [doi: 10.1109/TSE.2021.3057767]

[8] Wang Y, Wen M, Liu ZW, Wu RX, Wang R, Yang B, Yu H, Zhu ZL, Cheung SC. Do the dependency conflicts in my project matter? In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 319–330. [doi: 10.1145/3236024.3236056]

[9] Wang Y, Wen M, Wu RX, Liu ZW, Tan SH, Zhu ZL, Yu H, Cheung SC. Could I have a stack trace to examine the dependency conflict issue? In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 572–583. [doi: 10.1109/ICSE.2019.00068]

[10] Soto-Valero C, Durieux T, Baudry B. A longitudinal analysis of bloated Java dependencies. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 1021–1031. [doi: 10.1145/3468264.3468589]

[11] Soto-Valero C, Harrand N, Monperrus M, Baudry B. A comprehensive study of bloated dependencies in the Maven ecosystem. Empirical Software Engineering, 2021, 26(3): 45. [doi: 10.1007/s10664-020-09914-8]

[12] Hassan F, Mostafa S, Lam ESL, Wang XY. Automatic building of Java projects in software repositories: A study on feasibility and

challenges. In: Proc. of the 2017 ACM/IEEE Int'l Symp. on Empirical Software Engineering and Measurement (ESEM). Toronto: IEEE, 2017. 38–47. [doi: 10.1109/ESEM.2017.11]

[13] Cao YL, Chen L, Ma WWN, Li YH, Zhou YM, Wang LZ. Towards better dependency management: A first look at dependency smells in Python projects. IEEE Trans. on Software Engineering, 2023, 49(4): 1741–1765. [doi: 10.1109/TSE.2022.3191353]

[14] Jafari AJ, Costa DE, Abdalkareem R, Shihab E, Tsantalis N. Dependency smells in JavaScript projects. IEEE Trans. on Software Engineering, 2022, 48(10): 3790–3807. [doi: 10.1109/TSE.2021.3106247]

[15] Welcome to apache Maven. 2024. https://maven.apache.org/

[16] Gradle build tool. 2024. https://gradle.org/

[17] Huang KF, Chen BH, Shi BW, Wang Y, Xu CY, Peng X. Interactive, effort-aware library version harmonization. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 518–529. [doi: 10.1145/3368089.3409689]

[18] Patra J, Dixit PN, Pradel M. ConflictJS: Finding and understanding conflicts between JavaScript libraries. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering. Gothenburg: IEEE, 2018. 741–751. [doi: 10.1145/3180155.3180184]

[19] Zhang LY, Liu CW, Xu ZZ, Chen S, Fan LL, Zhao LD, Wu JH, Liu Y. Compatible remediation on vulnerabilities from third-party libraries for Java projects. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2540–2552. [doi: 10.1109/ICSE48619.2023.00212]

[20] Zhao LD, Chen S, Xu ZZ, Liu CW, Zhang LY, Wu JH, Sun J, Liu Y. Software composition analysis for vulnerability detection: An empirical study on Java projects. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 960–972. [doi: 10.1145/3611643.3616299]

[21] Vázquez HC, Bergel A, Vidal S, Díaz Pace JA, Marcos C. Slimming JavaScript applications: An approach for removing unused functions from javascript libraries. Information and Software Technology, 2019, 107: 18–29. [doi: 10.1016/j.infsof.2018.10.009]

[22] Soto-Valero C, Tiwari D, Toady T, Baudry B. Automatic specialization of third-party Java dependencies. IEEE Trans. on Software Engineering, 2023, 49(11): 5027–5045. [doi: 10.1109/TSE.2023.3324950]

[23] Wang Y, Wen M, Liu YP, Wang YB, Li ZM, Wang C, Yu H, Cheung SC, Xu C, Zhu ZL. Watchman: Monitoring dependency conflicts for Python library ecosystem. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: IEEE, 2020. 125–135. [doi: 10.1145/3377811.3380426]

[24] Qian CX, Koo H, Oh CS, Kim T, Lee W. Slimium: Debloating the chromium browser with feature subsetting. In: Proc. of the 2020 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2020. 461–476. [doi: 10.1145/3372297.3417866]

[25] Wang HY, Liu SG, Zhang LY, Xu C. Automatically resolving dependency-conflict building failures via behavior-consistent loosening of library version constraints. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 198–210. [doi: 10.1145/3611643.3616264]

[26] Konat G, Erdweg S, Visser E. Scalable incremental building with dynamic task dependencies. In: Proc. of the 33rd IEEE/ACM Int'l Conf. on Automated Software Engineering. Montpellier: IEEE, 2018. 76–86. [doi: 10.1145/3238147.3238196]

[27] Mitchell N, Sevitsky G. The causes of bloat, the limits of health. In: Proc. of the 22nd Annual ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages and Applications. Montreal: ACM, 2007. 245–260. [doi: 10.1145/1297027.1297046]

[28] Vassallo C, Proksch S, Jancso A, Gall HC, Di Penta M. Configuration smells in continuous delivery pipelines: A linter and a six-month study on GitLab. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 327–337. [doi: 10.1145/3368089.3409709]

[29] Zhang C, Chen BH, Hu JH, Peng X, Zhao WY. BuildSonic: Detecting and repairing performance-related configuration smells for continuous integration builds. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2022. 18. [doi: 10.1145/3551349.3556923]

[30] GitHub. 2023. https://github.com/

[31] Creswell JW. Qualitative Inquiry and Research Design: Choosing Among Five Approaches. 3rd ed., SAGE Publications Inc., 2012.

[32] Cavisson NS-ND integration performance publisher bundles Jenkins test harness, leading to java.lang.NoClassDefFoundError and memory leaks. JENKINS-66060, 2024. https://issues.jenkins.io/browse/JENKINS-66060

[33] The dropwizard-dependencies bom has declared some dependencies with scope. #3769, 2024. https://github.com/dropwizard/dropwizard/issues/3769

[34] Maven Core bug regarding resolution scopes for Mojos. MNG-8041, 2024. https://issues.apache.org/jira/browse/MNG-8041

[35] Feature/devx dashboard. #192, 2024. https://github.com/microsoftgraph/msgraph-sdk-java-core/pull/192
[36] Bump guava from 30.1-jre to 30.1.1-jre. #176, 2024. https://github.com/microsoftgraph/msgraph-sdk-java-core/pull/176
[37] Add missing gradle-wrapper.jar. #17, 2024. https://github.com/hkupty/penna/pull/17
[38] Can gradle-wrapper.jar be included into “INCLUDING BUILD TOOLS” list? LEGAL-570, 2024. https://issues.apache.org/jira/browse/LEGAL-570
[39] Bump gradle wrapper to version 5.6.4. #3422, 2024. https://github.com/bisq-network/bisq/pull/3422
[40] Gradle wrapper attack report. 2024. https://blog.gradle.org/wrapper-attack-report
[41] bug/PMTS-24-change-dependencies-scope-to-runtime. #25, 2024. https://github.com/qnocks/payment-service/pull/25
[42] Fix: Change scope of selenium-support dependency to compile. #2019, 2024. https://github.com/appium/java-client/pull/2019
[43] slf4j-log4j12 in test scope causes downstream issues. #152, 2024. https://github.com/confluentinc/common/pull/152
[44] Project lombok. 2024. https://projectlombok.org/
[45] Soot. 2024. http://soot-oss.github.io/soot/
[46] About JavaParser. 2024. https://javaparser.org/about.html
[47] Javassist by jboss-javassist. 2023. http://www.javassist.org/
[48] Add used but undeclared dependencies to pom.xmls. #4539, 2024. https://github.com/aws/aws-sdk-java-v2/pull/4539
[49] [INLONG-4771][manager] change junit-jupiter dependency to test scope. #4772, 2024. https://github.com/apache/inlong/pull/4772
[50] hapifhir/hapi-fhir-jpaserver-starter. 2024. https://github.com/hapifhir/hapi-fhir-jpaserver-starter
[51] Spring | home. 2024. https://spring.io/
[52] Btrace—A safe, dynamic tracing tool for the Java platform. 2024. https://github.com/btraceio/btrace
[53] Possible dependency conflict due to version conflict in /client. #2189, 2024. https://github.com/networknt/light-4j/issues/2189
[54] Undeclared dependency identified in client module. #2001, 2024. https://github.com/networknt/light-4j/issues/2001
[55] Add Maven wrapper for light-4j. #2021, 2024. https://github.com/networknt/light-4j/issues/2021
[56] The gradle-wrapper. jar is not up-to-date with gradle version. #1655, 2024. https://github.com/MobilityData/gtfs-validator/issues/1655
[57] Manage the versions of snappy-java and jackson-* centrally to avoid version conflict. #2546, 2024. https://github.com/apache/tinkerpop/pull/2546
[58] Abate P, Di Cosmo R, Treinen R, Zacchiroli S. Dependency solving: A separate concern in component evolution management. Journal of Systems and Software, 2012, 85(10): 2228–2240. [doi: 10.1016/j.jss.2012.02.018]
[59] Xu C, Qin Y, Yu P, Cao C, Lü J. Theories and techniques for growing software: Paradigm and beyond. SCIENTIA SINICA Informationis, 2020, 50(11): 1595–1611 (in Chinese with English abstract). [doi: 10.1360/SSI-2020-0079]
[60] Jin Z, Zhou MH, Zhang YX. Open source software and its eco-systems: Today and tommorow. Science & Technology Review, 2016, 34(14): 42–48 (in Chinese with English abstract). [doi: 10.3981/j.issn.1000-7857.2016.14.005]
[61] Ponta SE, Fischer W, Plate H, Sabetta A. The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application. In: Proc. of the 2021 IEEE Int’l Conf. on Software Maintenance and Evolution (ICSME). Luxembourg: IEEE, 2021. 555–558. [doi: 10.1109/ICSME52107.2021.00056]
[62] Jezek K, Dietrich J. On the use of static analysis to safeguard recursive dependency resolution. In: Proc. of the 40th EUROMICRO Conf. on Software Engineering and Advanced Applications. Verona: IEEE, 2014. 166–173. [doi: 10.1109/SEAA.2014.35]
[63] Yang HL, Chen L, Cao YL, Li YH, Zhou YM. Towards better dependency scope settings in Maven projects. In: Proc. of the 14th Asia-Pacific Symp. on Internetware. Hangzhou: ACM, 2023. 90–100. [doi: 10.1145/3609437.3609447]
[64] Song XH, Wang Y, Cheng X, Liang GT, Wang QX, Zhu ZL. Efficiently trimming the fat: Streamlining software dependencies with Java reflection and dependency analysis. In: Proc. of the 46th IEEE/ACM Int’l Conf. on Software Engineering (ICSE). Lisbon: IEEE, 2024. 1261–1272.
[65] Weeraddana NR, Alfadel M, McIntosh S. Dependency-induced waste in continuous integration: An empirical study of unused dependencies in the npm ecosystem. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 116. [doi: 10.1145/3660823]
[66] Drosos GP, Sotiropoulos T, Spinellis D, Mitropoulos D. Bloat beneath Python’s scales: A fine-grained inter-project dependency analysis. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 114. [doi: 10.1145/3660821]
[67] Peng Y, Hu RD, Wang RK, Gao CY, Li SQ, Lyu MR. Less is more? An empirical study on configuration issues in Python PyPI ecosystem. In: Proc. of the 46th IEEE/ACM Int’l Conf. on Software Engineering. Lisbon: ACM, 2024. 202. [doi: 10.1145/3597503.3639077]

[68] Li ZM, Wang Y, Lin ZQ, Cheung SC, Lou JG. Nufix: Escape from NuGet dependency maze. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 1545–1557. [doi: 10.1145/3510003.3510118]

[69] Wang Y, Sun P, Pei L, Yu Y, Xu C, Cheung SC, Yu H, Zhu ZL. Plumber: Boosting the propagation of vulnerability fixes in the *npm* ecosystem. IEEE Trans. on Software Engineering, 2023, 49(5): 3155–3181. [doi: 10.1109/TSE.2023.3243262]

[70] Mojica IJ, Adams B, Nagappan M, Dienst S, Berger T, Hassan AE. A large-scale empirical study on software reuse in mobile Apps. IEEE Software, 2014, 31(2): 78–86. [doi: 10.1109/MS.2013.142]

[71] Wang Y, Wu YX, Gao T, Chen ZY, Xu C, Yu H, Cheung SC. Survey on governance technology of open-source software library ecosystem: Twenty years of progress. Ruan Jian Xue Bao/Journal of Software, 2024, 35(2): 629–674 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6983.htm [doi: 10.13328/j.cnki.jos.006983]

[72] Liang GY, Wu YJ, Wu JZ, Zhao C. Open source software supply chain for reliability assurance of operating systems. Ruan Jian Xue Bao/Journal of Software, 2020, 31(10): 3056–3073 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6070.htm [doi: 10.13328/j.cnki.jos.006070]

[73] Ochoa L, Degueule T, Falleri JR. BreakBot: Analyzing the impact of breaking changes to assist library evolution. In: Proc. of the 44th IEEE/ACM Int'l Conf. on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). Pittsburgh: IEEE, 2022. 26–30. [doi: 10.1145/3510455.3512783]

[74] Ochoa L, Degueule T, Falleri JR, Vinju J. Breaking bad? Semantic versioning and impact of breaking changes in Maven central: An external and differentiated replication study. Empirical Software Engineering, 2022, 27(3): 61. [doi: 10.1007/s10664-021-10052-y]

[75] Jayasuriya D, Terragni V, Dietrich J, Ou S, Blincoe K. Understanding breaking changes in the wild. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: Association for Computing Machinery, 2023. 1433–1444. [doi: 10.1145/3597926.3598147]

[76] Jayasuriya D, Terragni V, Dietrich J, Blincoe K. Understanding the impact of APIs behavioral breaking changes on client applications. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 56. [doi: 10.1145/3643782]

[77] Zhang LY, Liu CW, Xu ZZ, Chen S, Fan LL, Chen BH, Liu Y. Has my release disobeyed semantic versioning? Static detection based on semantic differencing. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2022. 51. [doi: 10.1145/3551349.3556956]

[78] Dann A, Hermann B, Bodden E. UpCy: Safely updating outdated dependencies. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 233–244. [doi: 10.1109/ICSE48619.2023.00031]

[79] Li WK, Wu F, Fu C, Zhou F. A large-scale empirical study on semantic versioning in Golang ecosystem. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Luxembourg: IEEE, 2023. 1604–1614. [doi: 10.1109/ASE56229.2023.00140]

[80] Wu YL, Yu ZL, Wen M, Li Q, Zou DQ, Jin H. Understanding the threats of upstream vulnerabilities to downstream projects in the Maven ecosystem. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 1046–1058. [doi: 10.1109/ICSE48619.2023.00095]

[81] Zhang LY, Liu CW, Chen S, Xu ZZ, Fan LL, Zhao LD, Zhang YR, Liu Y. Mitigating persistence of open-source vulnerabilities in Maven ecosystem. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Luxembourg: IEEE, 2023. 191–203. [doi: 10.1109/ASE56229.2023.00058]

[82] Mir AM, Keshani M, Proksch S. On the effect of transitivity and granularity on vulnerability propagation in the Maven ecosystem. In: Proc. of the 2023 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Taipa: 2023. 201–211. [doi: 10.1109/SANER56733.2023.00028]

[83] Pashchenko I, Plate H, Ponta SE, Sabetta A, Massacci F. Vuln4Real: A methodology for counting actually vulnerable dependencies. IEEE Trans. on Software Engineering, 2022, 48(5): 1592–1609. [doi: 10.1109/TSE.2020.3025443]

[84] Fourné M, Wermke D, Enck W, Fahl S, Acar Y. It's like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security. In: Proc. of the 2023 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2023. 1527–1544. [doi: 10.1109/SP46215.2023.10179320]

[85] Keshani M, Velican TG, Bot G, Proksch S. AROMA: Automatic reproduction of Maven artifacts. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 38. [doi: 10.1145/3643764]

[86] Gao K, Xu WW, Yang WH, Zhou MH. PyRadar: Towards automatically retrieving and validating source code repository information for PyPI packages. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 115. [doi: 10.1145/3660822]

[87] Hu JC, Zhang LY, Liu CW, Yang S, Huang S, Liu Y. Empirical analysis of vulnerabilities life cycle in Golang ecosystem. In: Proc. of the

46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 212. [doi: 10.1145/3597503.3639230]
[88] Guo WB, Xu ZZ, Liu CW, Huang C, Fang Y, Liu Y. An empirical study of malicious code in PyPI ecosystem. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Luxembourg: IEEE, 2023. 166–177. [doi: 10.1109/ASE56229.2023.00135]
[89] org.json classes conflict with the current json library. #8, 2024. https://github.com/SAP/gigya-java-sdk/issues/8
[90] Dependency tree issue. #127, 2024. https://github.com/SpaiR/imgui-java/issues/127
[91] google-auth-library-credentials version collision in dependency tree. #79, 2024. https://github.com/spring-attic/spring-cloud-gcp/issues/79
[92] Evaluate Maven-wrapper to fix inconsistencies in dependencies.md. #441, 2024. https://github.com/exasol/project-keeper/issues/441
[93] Centralize versions for common dependencies. #235, 2024. https://github.com/stargate/stargate/pull/235
[94] Duplicate classes caught by maven-enforcer-plugin. #2642, 2024. https://github.com/apache/pulsar/issues/2642

附中文参考文献:

[59] 许畅, 秦逸, 余萍, 曹春, 吕建. 可成长软件理论方法和实现技术: 从范型到跨越. 中国科学: 信息科学, 2020, 50(11): 1595–1611. [doi: 10.1360/SSI-2020-0079]
[60] 金芝, 周明辉, 张宇霞. 开源软件与开源软件生态: 现状与趋势. 科技导报, 2016, 34(14): 42–48. [doi: 10.3981/j.issn.1000-7857.2016.14.005]
[71] 王莹, 伍盈欣, 高天, 陈子莺, 许畅, 于海, 张成志. 开源软件库生态治理技术研究综述: 二十年进展. 软件学报, 2024, 35(2): 629–674. http://www.jos.org.cn/1000-9825/6983.htm [doi: 10.13328/j.cnki.jos.006983]
[72] 梁冠宇, 武延军, 吴敬征, 赵琛. 面向操作系统可靠性保障的开源软件供应链. 软件学报, 2020, 31(10): 3056–3073. http://www.jos.org.cn/1000-9825/6070.htm [doi: 10.13328/j.cnki.jos.006070]

附录 A

在本附录中, 我们将继续探讨 Java 项目中的依赖异味, 描述正文中未涉及异味的特征, 可能危害与触发场景以及对应检测算法. 通过实证研究, 我们确定了 13 类依赖异味, 而正文已经涵盖了其中 5 类异味的特征, 3 类异味的部分危害与相应触发场景以及 1 类异味的检测算法. 接下来, 我们将详细介绍剩余未涉及的异味和对应检测算法.

A1 依赖异味分类与特征

在正文中, 我们已经对异味 1.6、1.7、1.8、2.2 和 2.3 进行了详细的介绍, 下面将继续介绍剩余的异味.

A1.1 模块粒度

1) 类别 1.1 内外类名冲突: 模块与依赖库中包含完全限定名相同的类 (2/47). 模块源代码中的类与模块所使用的某依赖库中的类名相同. 模块源代码中实现的类与模块所使用的某依赖库中的类名相同. 在图 A1(a) 对应的#GJS-8[89]中, 模块 gigya-java-sdk 中声明类 org.json.JSONObject, 而这个类同样在其依赖库 json 中存在, 从而导致模块与依赖库 json 中包含了完全限定名相同的类. 这导致用户在试图使用与 gigya-java-sdk 中版本不同的 JSONObject 时出现 NoSuchMethodError, 最终 gigya-java-sdk 中移除了冲突的类.

这类异味在 Maven 和 Gradle 中皆存在, 产生原因主要是 Java 虚拟机 (JVM) 在提供的路径中发现多个同名类时, 只会加载其中一个, 并忽略其他类. 这类异味往往不易被察觉, 尤其在构建可执行 Jar 包时, 由于配置文件未将当前模块自身声明为依赖库, 当遇到冲突类时, Maven 和 Gradle 会优先选择依赖库中的类进行打包, 而忽略当前模块中声明的类.

2) 类别 1.2 外部类名冲突: 模块的依赖库之间包含完全限定名相同的类 (7/47). 模块所使用的依赖库之间包含完全限定名相同的类.在图 A1(b) 对应的#IJA-127[90]中, 模块引用直接依赖 imgui-java-app 和被其引入的传递依赖 imgui-java-binding, 但 imgui-java-app 在打包时将一系列 imgui-java-binding 中的类也包含在其中, 导致两个依赖库之间出现了完全限定名相同的类, 最终开发者移除了 imgui-java-binding 中的类解决了类冲突.

这类异味在 Maven 和 Gradle 中皆存在, 产生原因和异味 1.1 相似. 不同之处在于, 这种异味涉及依赖库之间

的冲突, 而不是模块与依赖库之间的冲突, 因此不会出现因为冲突类在模块中而被忽略的情况. 在这种情况下, Maven 和 Gradle 项目都倾向于选择依赖树前序遍历序列中靠前的依赖库中的类.

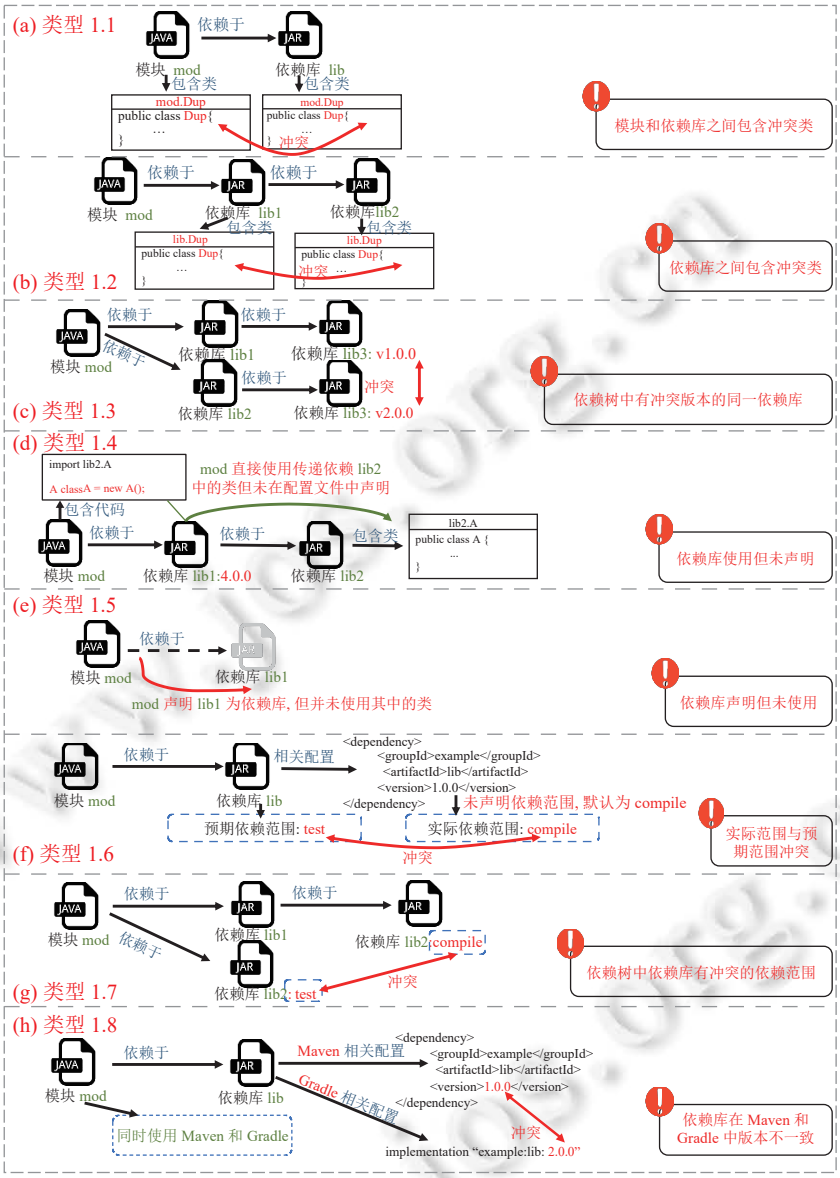

图 A1 模块粒度依赖异味特征实例

3) 类别 1.3 库版本冲突: 模块的依赖树中存在同一依赖库的不同版本 (7/47). 模块的依赖树中存在组织名和构件名都相同但版本不同的依赖库. 在图 A1(c) 对应的#SELENIDE-1652[42]中, 模块的直接依赖 browsermob-core 和 selenide 都引入了 netty-all 作为传递依赖, 但 netty-all 的版本不同, 最终只有一个版本的 netty-all 被选择, 其余会被忽略, 最终导致 NoSuchMethodError. 最终开发者移除了直接依赖 browsermob-core 解决了版本冲突.

这类异味在 Maven 和 Gradle 中皆存在, 产生原因主要在于依赖管理工具的版本调停机制. 当依赖树中存在同一依赖库的不同版本时, 最终只有一个版本的依赖库会被选择. 然而, 在 Maven 和 Gradle 下, 这类异味有不同的表

现形式, 因为它们有着不同的版本调停机制. 在出现版本冲突时, Maven 会选择依赖树中离模块 (也就是树根) 最近的依赖库; 而 Gradle 会选择较高版本的依赖库. 以#SCG-79[91]为例, 模块的依赖树中同时存在 google-auth-library-credentials:0.4.0 和 google-auth-library-credentials:0.7.0. 此时, Maven 选择版本 0.4.0, 而 Gradle 选择版本 0.7.0. 由于模块中实际使用的是 0.7.0 版本, 引用了 0.4.0 版本中不存在的类 ServiceAccountSigner, 因此依赖管理工具若为 Maven 则会出现 ClassNotFoundException, 若为 Gradle 则不会.

4) 类别 1.4 未声明依赖: 模块直接使用了未在配置文件中声明的依赖库 (6/47). 模块源代码中使用某依赖库中的类, 但此依赖库并未在项目的配置文件中得到明确声明. 在图 A1(d) 对应的#EL-1849[5]中, 模块使用了依赖库 asm 但未在自身的配置文件中声明, 但直接依赖 moxy 引入了 asm 作为其传递依赖, 从而让模块能够正常访问 asm 中的类以进行构建和运行. 最终开发者将 asm 声明为直接依赖, 解决了依赖库未声明的情况.

这类异味在 Maven 和 Gradle 中皆存在, 产生原因在于未声明的依赖库同时作为传递依赖存在, 这使得在模块构建或者运行时并不会直接报告缺少声明的依赖库. 这种情况下, 开发者会误以为依赖库已经被正确地声明了, 因为模块能够正常工作.

5) 类别 1.5 未使用依赖: 模块在配置文件中声明的依赖库实际上并未被使用 (6/47). 模块源代码中并未使用某依赖库中的任何类, 但此依赖库却在配置文件中声明. 在图 A1(e) 对应的#HFJS-581[50]中, 项目在构建文件中声明了依赖库 javax-samples, 但实际上并未引用其中类. 最终开发者在配置文件中移除了依赖库 javax-samples.

这类异味在 Maven 和 Gradle 中皆存在, 产生原因主要是 Maven 和 Gradle 只会在无法找到对应依赖库时报错, 而不会在出现冗余依赖库时报错. 这意味着当模块中存在未被使用的依赖库时, 构建工具并不会发出警告或报错信息. 因此, 开发者很容易忽视这些未使用的依赖库, 错误地认为它们是模块所需的一部分.

A1.2 项目粒度

1) 类别 2.1 构建工具配置缺失: 项目中缺少对其所使用构建工具的配置 (2/47). 项目中缺少所使用的构建工具的版本等关键信息的配置文件. 在图 A2(a) 对应的#PK-441[92]中, 项目中缺少了对 Maven 必要的文件如 maven-wrapper.properties, 导致无法准确确定使用的 Maven 版本. 最终这种不一致性导致了开发环境与持续集成 (CI) 中 Maven 版本的不匹配, 进而导致项目输出的不一致.

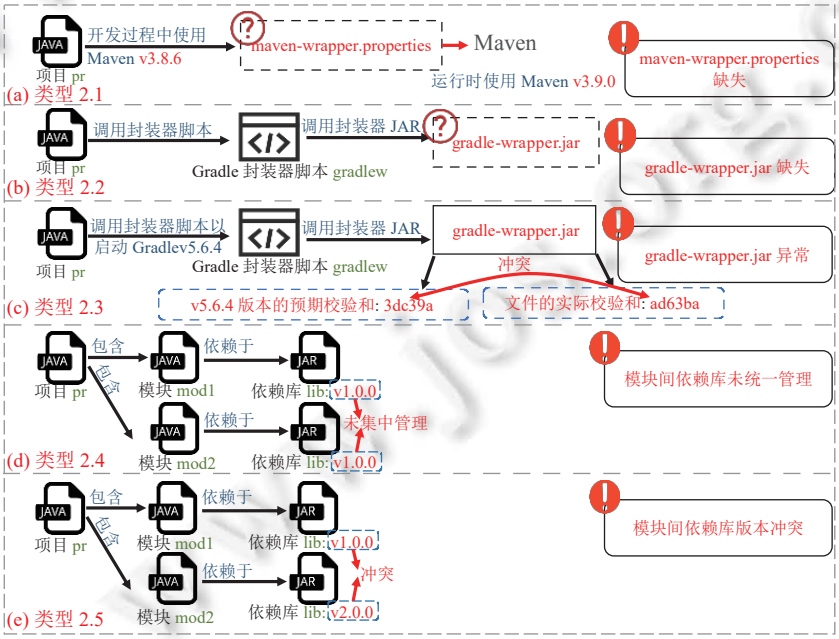

图 A2 项目粒度依赖异味特征实例.

这类异味在 Maven 和 Gradle 中皆存在, 其产生原因是主要是项目开发者未对项目构建工具启动器进行配置. 然而, 相较于 Gradle, 在 Maven 中这类问题更容易出现. 这主要是因为 Gradle 的启动器概念早在其诞生之初就已经存在, 而且官方文档和教程一直都在强调使用启动器的重要性. 此外, 在创建新项目时, Gradle 会自动添加启动器配置, 进一步推广了其使用. 相反, Maven 是在借鉴了 Gradle 后才添加了启动器功能, 这导致了相对于 Gradle 项目, Maven 项目中配置缺失的情况更为普遍.

2) 类别 2.4 模块间库重复: 项目多个模块中包含未进行统一管理的同一依赖库 (2/47). 项目多个模块中使用同一依赖库, 但并未使用构建工具提供的版本管理机制对其版本进行统一管理. 在图 A2(d) 对应的#STARGATE-235[93]中, stargate 在不同的模块中有一系列相同的依赖库, 如 jetty-servlet 等, 但他们的版本声明都分散在各自模块的配置文件中, 而没有利用 Maven 提供的 dependencyManagement 功能在项目中进行统一管理. 最终开发者取消了各模块中分散的版本声明, 而是统一进行管理.

这类异味在 Maven 和 Gradle 中皆存在, 其产生原因主要是开发者对 Maven 和 Gradle 提供统一依赖管理机制的忽略, 它们都提供了相关机制来集中管理项目中的依赖版本, 以确保在不同模块中使用相同的库版本.

3) 类别 2.5 模块间库冲突: 项目多个模块中使用了不同版本的同一依赖库 (1/47). 项目的多个模块中使用了组织名和构件名都相同但版本不同的依赖库. 在图 A2(e) 对应的#PULSAR-2 642[94]中, 项目 pulsar 的模块 pulsar-client 和模块 pulsar-client-schema 中分别使用了 protobuf 的 2.0 和 3.0 版本. 由于 protobuf:3.0 相较于 2.0 版本能力更强, pular 计划最终将所有模块中 protobuf 版本升级为 3.0, 但目前尚未完成这一过程, 最终导致模块之间出现依赖库的版本冲突.

这类异味在 Maven 和 Gradle 中皆存在, 其主要有两种产生原因. 首先, 项目可能考虑到旧版本依赖库兼容性的考虑或者正处于依赖库的升级过程中, 从而被迫在不同模块中使用不同版本的依赖库, 上例即对应这类产生原因. 其次, 另一种产生原因是项目中出现了依赖库分散管理的情况 (即异味 2.4), 从而对单一模块依赖库版本的修改未同步到其他模块, 进而引发模块间的依赖库版本冲突问题.

A2 依赖异味危害与触发场景

在正文中, 我们已经介绍了异味 1.4、1.6 和 1.7 的部分危害与对应触发场景. 下文将继续介绍剩余异味的可能危害以及异味 1.4、1.6 和 1.7 未在正文中被介绍的危害, 并介绍这些危害对应的触发场景.

1) 类别 1.1 内外类冲突

内外类冲突可能引发一系列严重的后果, 包括但不限于模块构建错误、运行时错误以及运行时语义冲突. 以下将详细阐述这些危害的具体触发场景.

(1) 构建错误: 其触发场景为模块源码中调用了依赖库中同名冲突类独有的方法. 以图 A3(a) 为例, 模块 mod 和依赖库 lib 中均存在名为 Dup 的冲突类, 但方法 methodLib 仅在 lib 中的 Dup 类中定义. 若模块 mod 试图调用该方法时, 由于该方法在模块自身的 Dup 类中不存在, 会导致构建错误.

(2) 运行时错误: 其触发场景为模块运行时调用了依赖库中同名冲突类独有的方法. 以图 A3(b) 为例, 与上例类似, 模块 mod 和依赖库 lib 中同样存在冲突类 Dup, 并且方法 methodLib 同样仅在 lib 中的 Dup 中存在. 但此时 mod 并未直接调用 methodLib, 而是调用了 RunDup 中的 runMethodLib 方法, 间接调用了 methodLib. 此时虽然模块能正常构建, 但在运行时则无法找到对应方法, 进而造成运行时错误.

(3) 运行时语义冲突: 其触发场景为模块运行时调用了冲突类中签名相同但实现不同的方法. 以图 A3(c) 为例, 模块 mod 和依赖库 lib 中存在冲突类 Dup 和同签名方法 methodLib, 但其具体实现不同. 当 mod 调用 methodLib 时就会出现语义冲突, 预期调用的是 mod 中的 Dup 类, 对应的 methodDup 返回“Module”, 而实际调用的是 Lib 中的 Dup 类, 对应的 methodDup 返回“Lib”.

2) 类别 1.2 外部类冲突

外部类冲突可能引发一系列严重的后果, 包括但不限于模块构建错误、运行时错误以及运行时语义冲突. 以下将详细阐述这些危害的具体触发场景.

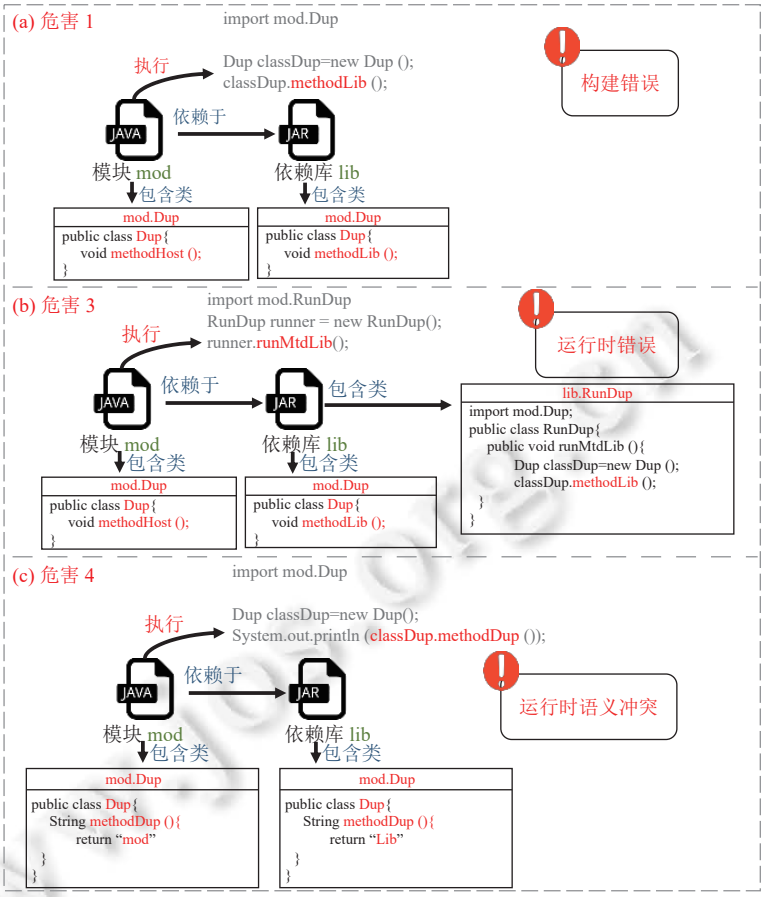

图 A3 异味 1.1 可能危害与对应触发场景

(1) 构建错误: 其触发场景为模块源码中调用了仅存在于被遮蔽的同名冲突类中的方法. 以图 A4(a) 为例, 模块 mod 的依赖库 lib1 和 lib2 中存在冲突类 Dup, mod 试图调用仅存在于 lib2 中的方法 methodLib2, 但由于 lib1 在依赖树中与 mod 距离更近, 最终 lib2 中的类 Dup 被忽略, 导致构建错误.

(2) 运行时错误: 其触发场景为项目运行时调用了仅存在于被遮蔽的同名冲突类中的方法. 以图 A4(b) 为例, 模块 mod 的依赖库 lib1 和 lib2 中存在冲突类 Dup, mod 试图调用 lib2 中的方法 runMtdLib, 其间接调用了 Dup 中的方法 methodLib2. 由于并未直接调用 methodLib2, 模块可以正常进行构建. 但在运行时, JVM 会忽略 lib2 中的类 Dup, 因此在运行时则无法找到 lib 对应方法, 导致运行时错误.

(3) 运行时语义冲突: 其触发场景为模块运行时调用了冲突类中签名相同但实现不同的方法. 以图 A4(c) 为例, 模块 mod 的依赖库 lib1 和 lib2 中存在冲突类 Dup 和同签名方法 methodLib, 但其具体实现不同. 当 mod 调用 methodLib 时就会出现语义冲突, 预期调用的是 lib2 中的 Dup 类, 对应的 methodDup 返回"lib2", 而实际调用的是 lib1 中的 Dup 类, 对应的 methodDup 返回"lib1".

3) 类别 1.3 库版本冲突

库版本冲突可能引发一系列严重的后果, 包括但不限于模块构建错误、运行时错误以及运行时语义冲突. 以下将详细阐述这些危害的具体触发场景.

(1) 构建错误: 其触发场景为模块源码中调用了仅存在于被遮蔽的库中的特性 (例如方法). 以图 A5(a) 为例, 模块 mod 的依赖树中同时存在 lib1 的 1.0.0 和 2.0.0 版本. 根据 Maven 和 Gradle 的版本冲突解决机制, 最终 1.0.0

版本会被遮蔽, 但 mod 在源码中试图调用仅存在于 1.0.0 版本中的方法 methodV1, 这导致模块构建错误.

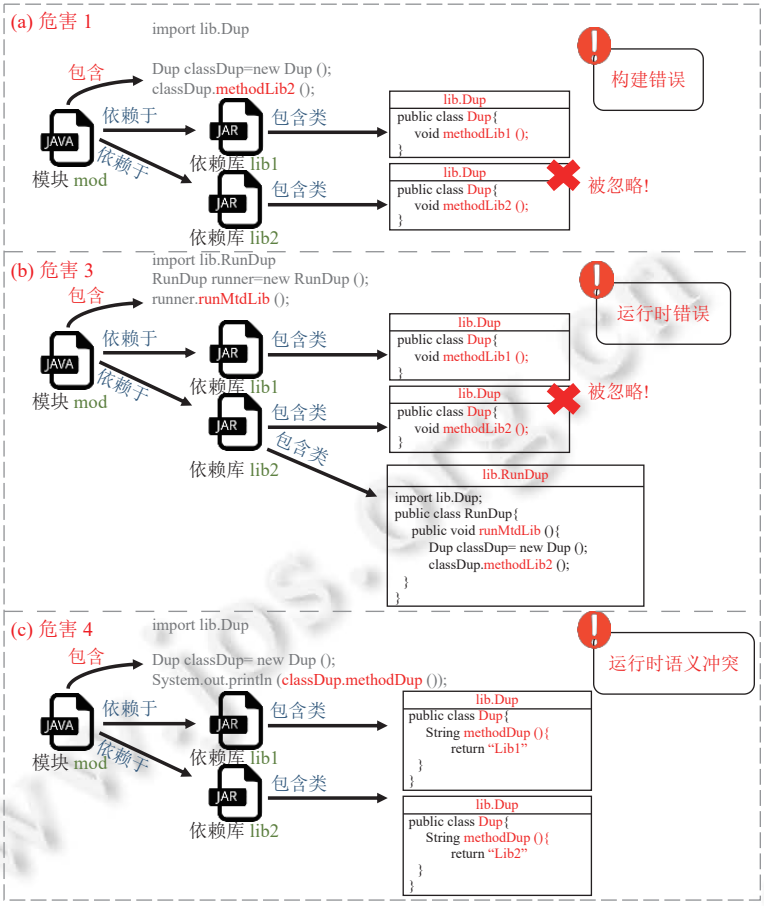

图 A4 异味 1.2 可能危害与对应触发场景

(2) 运行时错误: 其触发场景为模块运行时调用了仅存在于被遮蔽的库中的特性. 以图 A5(b) 为例, 模块 mod 直接依赖于 lib1:2.0.0 和 lib2, lib2 又依赖于 lib1:1.0.0, 并且在 runMtdLib 中调用了仅存在于 lib1:1.0.0 中的方法 methodV1. 当 mod 试图调用 runMtdLib 时, 由于 lib1:1.0.0 被遮蔽, 最终出现运行时错误.

(3) 运行时语义冲突: 其触发场景为模块运行时调用了被遮蔽库中签名相同但实现不同的方法. 以图 A5(c) 为例, 模块 mod 的依赖树中同时存在 lib1 的 1.0.0 和 2.0.0 版本, 两个版本的 lib1 中都包含签名相同的 methodDup 方法, 但其具体实现不同. 当 mod 调用 methodDup 时就会出现语义冲突, 预期调用的是 2.0.0 版本中的 methodDup, 应返回“V2”, 而实际调用的是 1.0.0 版本中的方法, 返回的是“V1”.

4) 类别 1.4 未声明依赖

未声明依赖可能引发一系列严重的后果, 包括但不限于模块构建错误、运行时错误以及运行时语义冲突. 以下将详细阐述这些危害的具体触发场景.

(1) 构建错误: 其触发场景为模块源码中使用但未声明的依赖库未在依赖树中出现. 以图 A6(a) 为例, 模块 mod 中引用依赖库 lib2 中的类 A, 但并未在配置文件中声明 lib2, 而是通过依赖库 lib1:4.0.0 引入 lib2 作为传递依赖. 然而 lib1 升级到 4.0.1 后不再依赖于 lib2, mod 的依赖树中不再存在 lib2, 无法找到对应类, 导致构建错误.

(2) 运行时错误: 其触发场景为模块运行时使用但未声明的依赖库未在依赖树中出现. 以图 A6(b) 为例, mod 利用反射特性引用依赖库 lib2 中的类 A, 但并未在配置文件中声明. 和上例类似, 在 mod 依赖树中不再存在

lib2 后, 由于 mod 并未在源码中而是利用反射特性动态引入类 A, 所以 mod 不会出现构建错误, 但在运行时则会因为无法找到类 A 而出现运行时错误.

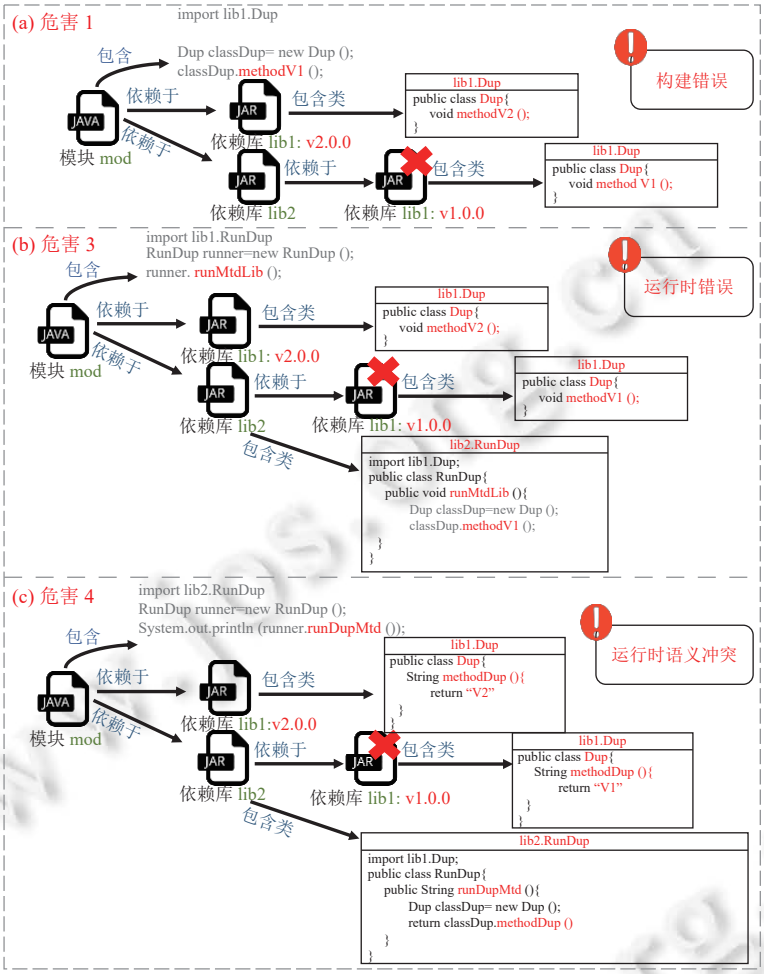

图 A5　异味 1.3 可能危害与对应触发场景

(3) 运行时语义冲突: 其触发场景为模块直接使用但未声明的依赖库版本升级后出现签名相同但实现不同的情况. 以图 A6(c) 为例, 模块 mod 中直接使用 lib2:1.0.0 中的方法 getVersion, 但并未在配置文件中声明 lib2, 而是通过依赖库 lib1:4.0.0 引入 lib2:1.0.0 作为传递依赖. 然而 lib1 升级到 4.0.1 时同时升级 lib2 版本, 引入传递依赖 lib2:2.0.0, 其中方法 getVersion 的签名并未改变, 但其具体实现不同. 当 mod 调用 version 时就会出现语义冲突, 预期调用的是 1.0.0 版本中的 getVersion, 应返回"V1", 而实际调用的是 2.0.0 版本中的方法, 返回的是"V2".

5) 类别 1.5 未使用依赖

未使用依赖不会如前述异味一般造成构建错误等破坏性的危害, 但会造成构建时间增加、构建产物冗余等危害. 以下将详细阐述这些危害的具体触发场景.

(1) 构建时间增加: 这类危害在出现依赖未使用时始终存在. 以图 A7(a) 为例, 模块 mod 并未使用依赖库 lib 中的特性, 但却在配置文件中进行声明, 导致在构建时该依赖会被引入, 需要从中央存储仓库获取对应依赖库, 进而导致构建时间增加, 即增加构建成本.

(2) 构建产物冗余: 其触发场景为模块可执行 JAR 的打包. 以图 A7(b) 为例, 模块 mod 并未使用依赖库 lib 中

的特性, 但却在配置文件中进行声明. 而 Maven 和 Gradle 在将 mod 打包为可执行 JAR 时, 所有的依赖库都会被涵盖, 包括实际上并未被使用的 lib, 导致构建产物冗余.

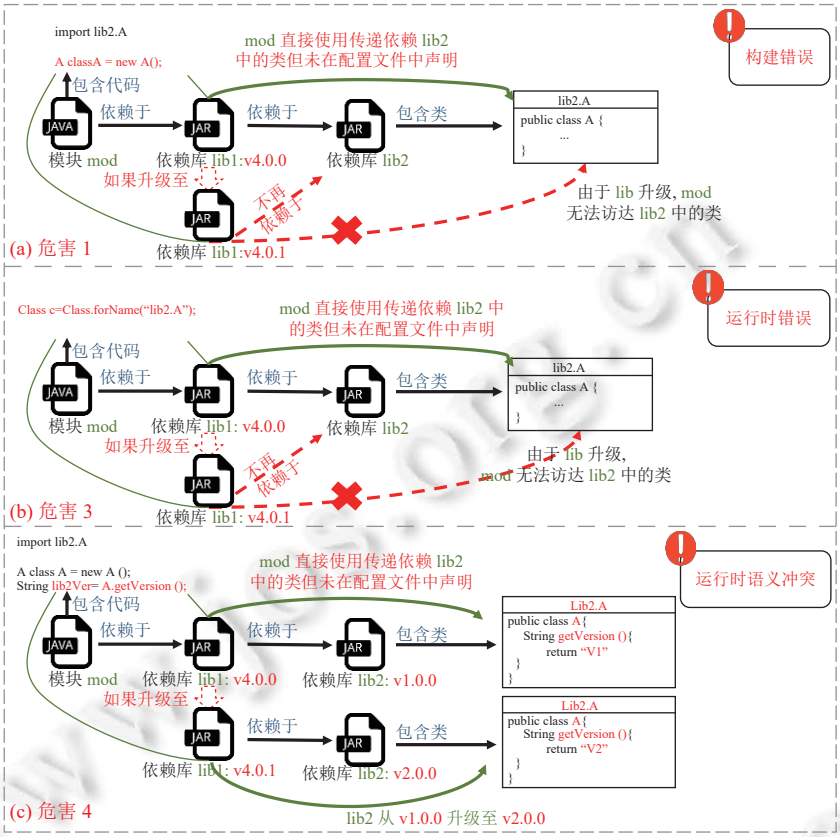

图 A6 异味 1.4 可能危害与对应触发场景

(3) 下游模块构建时间增加: 这类危害在出现依赖未使用时始终存在. 以图 A7(c) 为例, 模块 mod 并未使用依赖库 lib 中的特性, 但却在配置文件中进行声明. 下游模块 client 在使用模块 mod 的同时, 也就引入了 mod 的依赖库 lib 作为传递依赖, 其会在构建时被引入, 需要从中央存储仓库获取对应依赖库, 进而导致构建时间增加, 即增加下游模块 client 的构建成本.

(4) 下游模块构建产物冗余: 其触发场景为下游模块可执行 JAR 的打包. 以图 A7(d) 为例, 模块 mod 并未使用依赖库 lib 中的特性, 但却在配置文件中进行声明. 下游模块 client 在使用模块 mod 的同时, 也引入了 mod 的依赖库 lib 作为传递依赖, 而 Maven 和 Gradle 在将 client 打包为可执行 JAR 时, 所有直接依赖和传递依赖库都会被包括, 导致下游模块 lib 构建产物冗余.

6) 类别 1.6 依赖范围误用

Maven 与 Gradle 均定义了多种依赖范围, 若未按预期正确使用这些范围, 可能引发多种类型的危害. 我们在正文表 5 和表 6 中对这些危害进行了梳理. 下文将详细阐述正文中未能详述的各类危害及其触发场景.

(1) 构建错误: 以预期依赖范围为 compile, 实际范围为 test 为例, 在图 A8(a) 中, 模块 mod 在源码中直接引用 lib 中的类 A, 因此其预期依赖范围应该是 compile, 但其实际依赖范围被设置为 test, 代表其只会在测试时出现而不会在构建时出现, 这就导致 mod 构建时无法找到类 A, 最终构建错误.

(2) 构建时间增加: 以预期范围为 test, 实际范围为 compile 为例, 在图 A8(b) 中, 模块 mod 将直接依赖 lib 的依

赖范围设置为 compile, 代表其在构建、运行和测试时都被使用, 但其预期依赖范围是 test, 代表其不应该在构建时出现. 因此在 mod 进行构建时, 需要从中央存储仓库获取事实上不必要的 lib, 导致构建时间增加, 即增加构建成本.

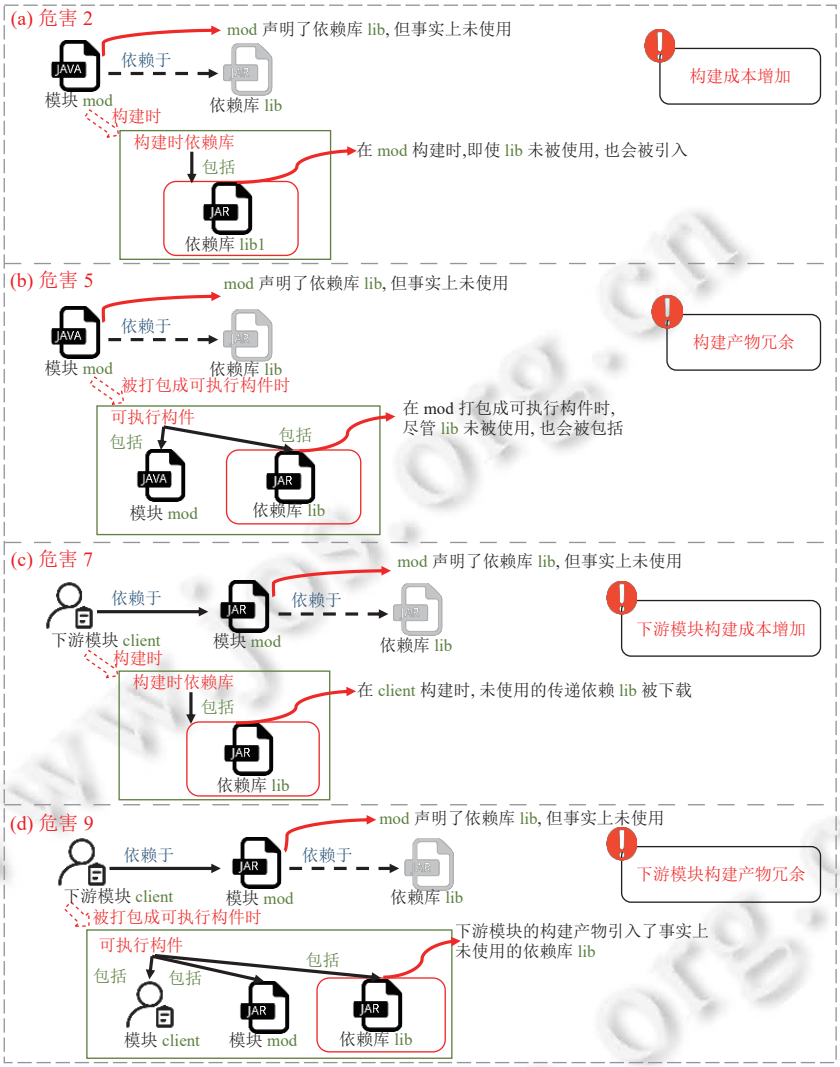

图 A7 异味 1.5 可能危害与对应触发场景

(3) 构建产物冗余: 以预期范围为 test, 实际范围为 compile 为例, 在图 A8(d) 中, 模块 mod 将直接依赖 lib 的依赖范围设置为 compile, 但其实际上仅在测试时被使用, 预期依赖范围应为 test. 在 mod 打包为可执行 JAR 时, 所有依赖范围为 compile 的依赖库都会被涵盖, 而会忽略依赖范围为 test 的依赖库, 这就导致 lib 被错误地包括进构建产物中, 最终导致构建产物冗余.

(4) 下游模块构建时间增加: 以预期依赖范围为 implementation, 实际范围为 api 为例, 在图 A9(b) 中, 模块 mod 将依赖库 lib 的依赖范围设置为 api, 此范围意味着下游模块 client 在构建时引入 mod 就会同时引入 lib, 但 lib 仅用于 mod 内部实现, 其预期范围是 implementation, 不应该出现在 client 的构建时. 因此 client 在构建时需要从中央存储仓库获取事实上不必要的 lib, 导致构建时间增加, 即增加构建成本.

(5) 下游模块运行时错误: 以预期依赖范围为 runtime, 实际范围为 test 为例, 在图 A9(c) 中, 模块 mod 将依赖库 lib 的依赖范围设置为 test, 此范围意味着 test 不会在运行时被引入, 但事实上 mod 中的方法 runMtdLib 以反射

形式获取 lib 中的类 lib.A, 其预期范围应该是 runtime. 由于 lib.A 以反射形式被引入, 此时 mod 可以正常进行构建, 但当下游项目运行时使用 runMtdLib 时, 则会因为 lib 未在运行时被引入, 无法找到 lib.A, 从而导致下游模块运行时错误.

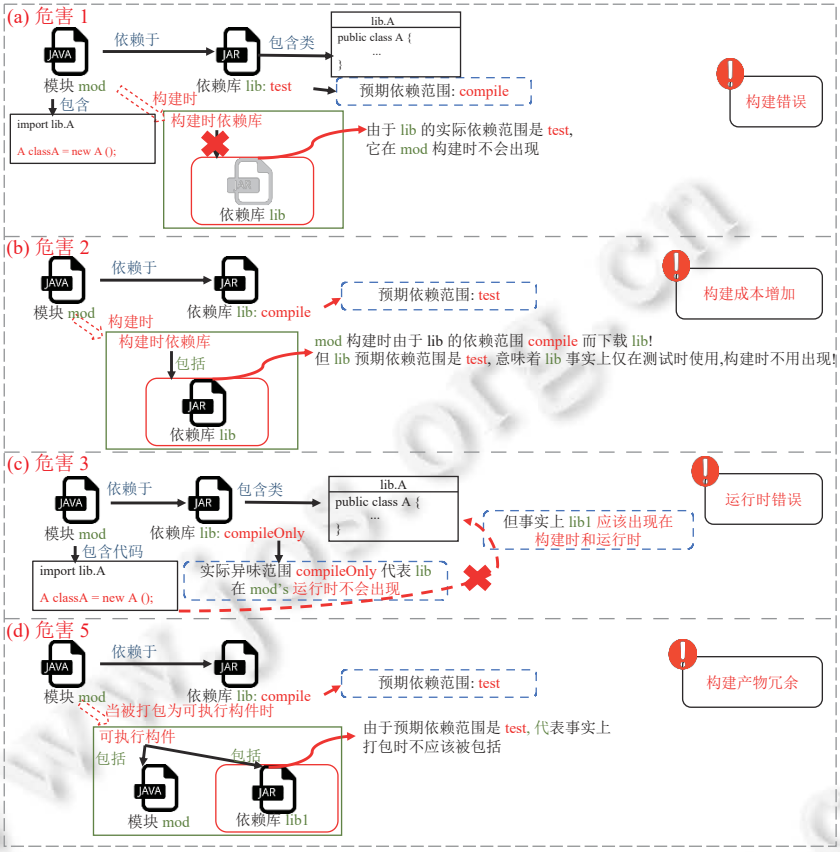

图 A8　异味 1.6 对当前模块的危害与对应触发场景

(6) 下游模块构建产物冗余: 以预期范围为 test, 实际范围为 compile 为例, 在图 A9(d) 中, 模块 mod 将直接依赖 lib 的依赖范围设置为 compile, 但其实际上仅在测试时被使用, 预期依赖范围应为 test. 下游模块 client 在使用模块 mod 的同时, 也引入了 mod 的依赖库 lib 作为传递依赖. 将 client 打包为可执行 JAR 时, 直接依赖和传递依赖中依赖范围为 compile 的依赖库都会被包含, 但事实上 lib 仅在测试时使用而不需被打包, 这就导致下游模块 lib 构建产物冗余.

7) 类别 1.7 依赖范围冲突

与异味 1.6 依赖范围误用类似, 不同依赖范围之间的冲突也可能造成多样的危害. 我们将详细结果汇总于正文表 7. 下文将针对正文中未能详述的各类冲突可能造成的危害及其触发场景进行详细介绍.

(1) 构建错误: 其触发场景为直接依赖的 runtime 和 test 范围覆盖了传递依赖的 compile 和 provided 范围. 以图 A10(a) 为例, 模块 mod 有直接依赖 lib1 和 lib2, 而 lib1 也引入 lib2 作为传递依赖. 在 mod 的依赖树中, lib2 作为直接依赖的范围是 test, 而作为传递依赖的范围是 compile. 此时, lib2 的最终范围是 test, 仅会在测试时出现. 但在 mod 中虽然并未直接引入 lib2 中的类, 但其中的类 C 以 lib2 中的类 A 为父类, 这就导致 mod 构建时无法找到类 A, 进而出现构建错误.

8) 类别 1.8 依赖树冲突

依赖树冲突的危害与异味 1.3 类似, 其差别在于异味 1.3 关注于单一依赖树中的库版本冲突, 而异味 1.8 则关

注于 Maven 和 Gradle 之间依赖树的冲突, 下面我们将详述各类危害的具体触发情境.

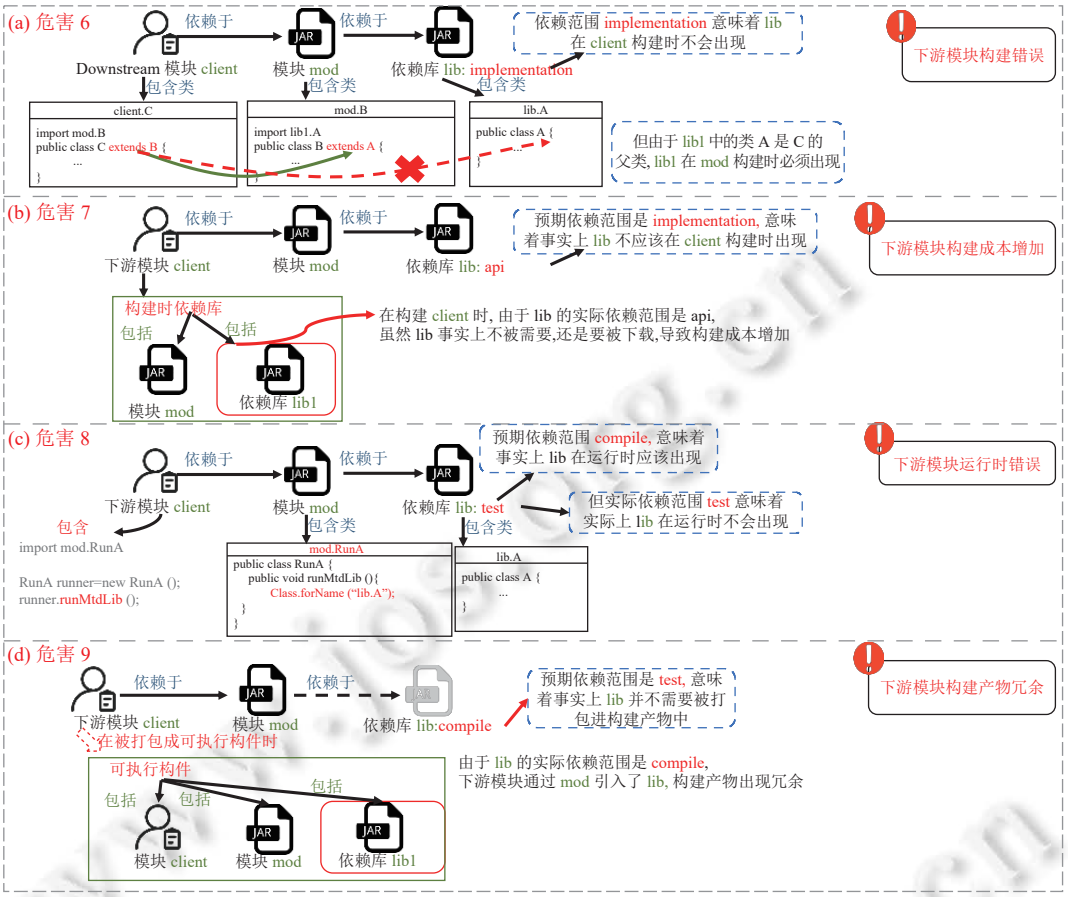

图 A9 异味 1.6 对下游模块的危害与对应触发场景

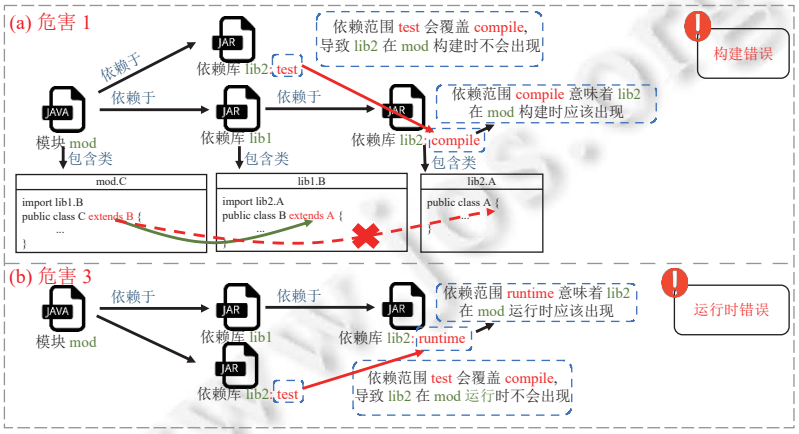

图 A10 异味 1.7 可能危害与对应触发场景

(1) 构建错误: 其触发场景为模块源码中调用了某冲突依赖库中独有的特性. 以图 A11(a) 为例, 模块 mod 同时维护了 Maven 和 Gradle 对应的配置文件, 但依赖库 lib 在 Maven 配置文件中的版本声明为 2.0.0, 而在 Gradle 配

置文件中则为 1.0.0. 由于 mod 在源码中调用了仅存在于 1.0.0 版本中的方法 methodV1, 试图 A 使用 Maven 进行构建时无法找到对应方法, 因此出现构建错误.

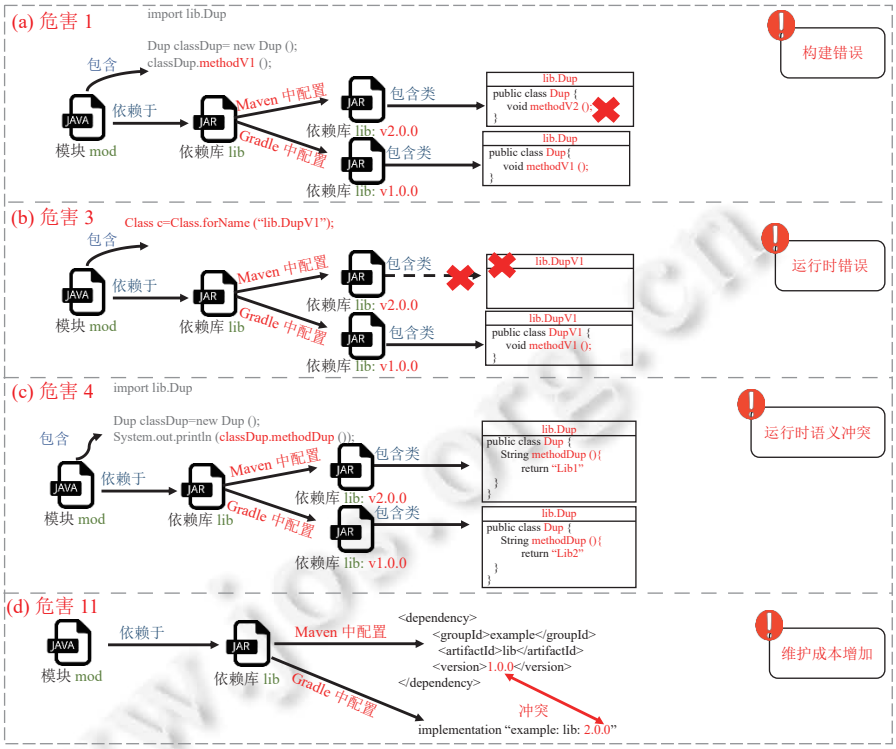

图 A11 异味 1.8 可能危害与对应触发场景

(2) 运行时错误: 其触发场景为模块运行时调用了某冲突依赖库中独有的特性. 以图 A11(c) 为例, 与上例类似, mod 在 Maven 中依赖于 lib:1.0.0 而在 Gradle 中依赖于 lib:2.0.0. 此时 mod 以反射形式获取仅存在于 1.0.0 版本中的类 lib.DupV1, 能正常通过构建, 但在运行 Maven 对应构建产物时, 会因找不到 lib.DupV1 而出现运行时错误.

(3) 运行时语义冲突: 其触发场景为模块运行时调用了冲突依赖库中签名相同但实现不同的方法. 以图 A11(d) 为例, 与上例类似, mod 在 Maven 中依赖于 lib:1.0.0 而在 Gradle 中依赖于 lib:2.0.0. 虽然类 Dup 中的方法 methodDup 在两个版本中都存在, 但其具体实现不同. 当 mod 试图调用 methodDup 时会出现语义冲突, 在运行 Maven 构建产物时的输出为“Lib1”, 而运行 Gradle 构建产物时输出则为“Lib2”.

(4) 依赖库维护难度增加: 这类危害在出现异味时始终存在. 以图 A11(a) 为例, 模块 mod 同时维护了 Maven 和 Gradle 对应的配置文件, 但依赖库 lib 在 Maven 配置文件中的版本为 1.0.0, 而在 Gradle 配置文件中为 2.0.0. 开发者在进行版本升级时需要同步更新至所有构建工具对应的配置文件, 导致维护难度增加.

9) 类别 2.1 构建工具配置缺失

构建工具配置缺失可能导致无法获取预期版本的构建工具, 进而导致构建错误.

(1) 构建错误: 其触发场景为项目实际使用的构建工具版本与预期不一致, 且使用了仅存在于预期版本中的特性. 以图 A12(a) 为例, 项目预期使用 Gradle 4.1 版本及其中新引入的 java-library 特性, 但实际上未对 Gradle 版本进行配置, 导致实际使用的是本地的 Gradle 3.1 版本. 因此, 无法识别 java-library 特性, 导致构建错误.

10) 类别 2.2 构建工具启动器 JAR 缺失

构建工具启动器 JAR 缺失与异味 2.1 类似, 同样可能导致无法获取预期版本的构建工具, 进而导致构建错误.

(1) 构建错误: 其触发场景为项目使用构建工具启动器运行构建工具. 以图 A13(a) 为例, 项目预期使用 Gradle 4.1

版本及其中新引入的 java-library 特性, 但本地仅存在 Gradle 3.1 版本而无法进行构建. 在使用构建工具启动器时, 由于项目中仅存在启动器脚本 (gradlew), 而缺少启动器 JAR (gradle-wrapper.jar), 因此无法找到 gradle-wrapper.jar 而导致构建错误.

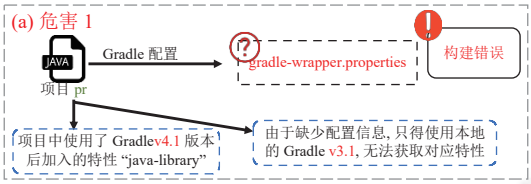

图 A12　异味 2.1 可能危害与对应触发场景

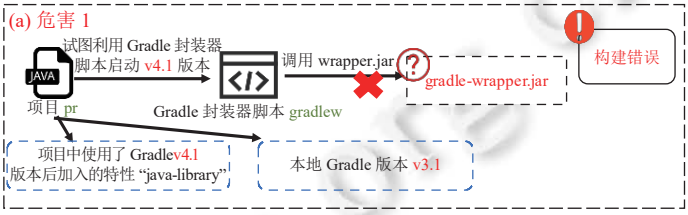

图 A13　异味 2.2 可能危害与对应触发场景

11) 类别 2.3 构建工具启动器 JAR 异常

构建工具启动器 JAR 异常可能造成构建时间增加和构建工具使用时的安全隐患, 以下将详细阐述这些危害的具体触发场景.

(1) 构建时间增加: 其触发场景为项目构建工具启动器 JAR 版本过旧, 导致性能问题. 以图 A14(a) 为例, 项目使用的是 Gradle 3.0 版本, 对应的 gradle-wrapper.jar 应该是 3.0 版本, 预期的校验和为 42d7a2, 但实际的校验和为 695 089, 对应于 2.6 版本. 然而, Gradle 2.6 版本存在严重的性能下降问题, 影响了 Gradle 包装器的启动时间, 进而导致构建时间增加, 而在 3.0 版本中此问题已被修复.

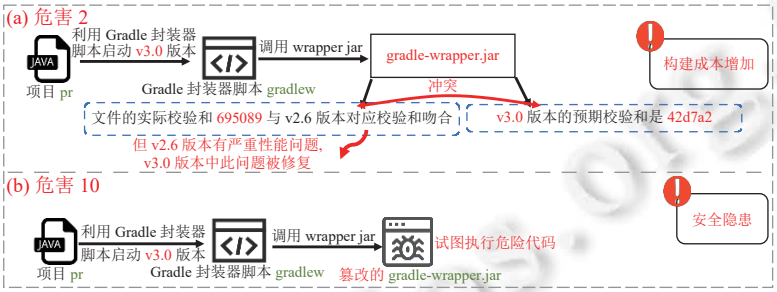

图 A14　异味 2.3 可能危害与对应触发场景

(2) 安全隐患: 其触发场景为项目构建工具启动器 JAR 被篡改. 以图 A14(b) 为例, 项目中的 gradle-wrapper.jar 校验和与任何官方发布的已知版本都不相符. 其被修改并添加了恶意代码, 当开发者试图通过构建工具启动器启动构建工具时, 可能会导致安全隐患, 例如开发者的信息被窃取等.

12) 类别 2.4 模块间库重复

模块间库重复可能会造成项目维护成本的增加, 其触发场景如下.

(1) 依赖库维护难度增加: 其触发场景为项目对未集中管理的依赖库进行版本升级. 以图 A15(a) 为例, 项目中的模块 mod1 和 mod2 之间存在同一依赖库 lib 的 1.0.0 版本, 但其版本分别在各自配置文件中进行声明, 而并未进行统一指定. 在试图将 lib 升级至 2.0.0 版本时, 需要将修改同步至所有模块, 这就导致项目维护成本的增加.

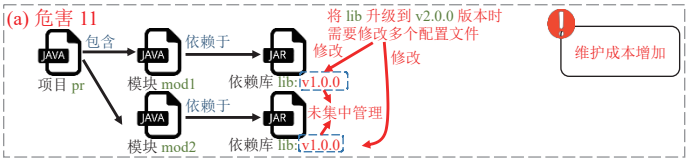

图 A15 异味 2.4 可能危害与对应触发场景

13) 类别 2.5 模块间库冲突

模块间库冲突的危害和模块间库重复类似, 同样会增加项目维护成本, 并可能增加下游项目出现依赖冲突的概率.

(1) 依赖库维护难度增加: 其触发场景为项目对未集中管理的依赖库进行版本升级. 以图 A16(a) 为例, 项目中的模块 mod1 和 mod2 分别依赖于依赖库 lib 的 1.0.0 和 2.0.0 版本. 在试图将 lib 升级至 3.0.0 版本时, 需要将修改同步至所有模块, 这就导致项目维护成本的增加.

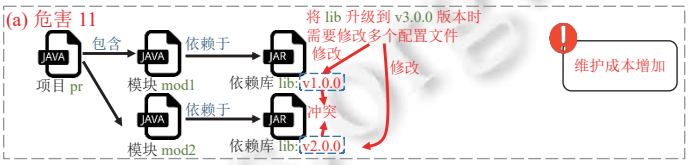

图 A16 异味 2.5 可能危害与对应触发场景

A3 依赖异味检测算法

在正文中, 我们已经介绍了异味 1.6 依赖范围误用的检测算法, 下文将继续介绍剩余异味对应的检测算法.

1) 类别 1.1 内外类冲突 (如算法 A1 所示)

算法 A1. 内外类冲突检测.

输入: *DJ*: 依赖包集合; *BI*: 基本信息;
输出: *JS*: 出现内外类冲突的依赖库集合.

```
1.  JS ⇐ {}
2.  moduleClasses ⇐ BI.sourceClasses
3.  for depJar ∈ DJ do
4.      depClasses ⇐ depJar.sourceClasses
5.      for depclass ∈ depClasses do
6.          if depclass ∈ moduleClasses then
7.              JS.add(depJar)
8.          end if
9.      end for
10. end for
```

该算法接受依赖包集合 *DJ* 和基本信息 *BI* 作为输入, 并输出出现内外类冲突的依赖库集合 *JS*. 算法的工作流程如下: 首先, 初始化依赖库集合 *JS* (第 1 行). 接着, 从基本信息 *BI* 中获取模块的类集合 *moduleClasses* (第 2 行). 然后, 算法遍历依赖包集合 *DJ* 中的每一个依赖包 *depJar* (第 3 行), 并提取该依赖包的类集合 *depClasses* (第 4 行). 接下来, 算法遍历依赖包 *depClasses* 中的每一个类 *depclass* (第 5 行). 如果当前类 *depclass* 存在于模块的类集合 *moduleClasses* 中, 算法将依赖包 *depJar* 加入集合 *JS* (第 6–8 行). 最终, 算法结束对依赖包的遍历 (第 9 行), 并完成内外类冲突的检测.

2) 类别 1.2 外部类冲突 (如算法 A2 所示)

算法 A2. 外部类冲突检测.

输入: *DJ*: 依赖包集合;
输出: *JS*: 出现外部类冲突的依赖库对集合.

```
1.  JS ⇐ {}
2.  classToJarMap ⇐ {}
3.  for depJar ∈ DJ do
4.      depClasses ⇐ depJar.sourceClasses
5.      classToJarMap[depClasses].add(depJar)
6.  end for
7.  for class ∈ classToJarMap, keys do
8.      conflictJars ⇐ classToJarMap[class]
9.      if conflictJars.size() > 1 and not conflictJars ∈ JS then
10.         JS.add(conflictJars)
11.     end if
12. end for
```

该算法接受依赖包集合 *DJ* 作为输入, 并输出出现外部类冲突的依赖库对集合 *JS*. 算法的工作流程如下: 首先, 初始化依赖库对集合 *JS* (第 1 行). 接着, 创建一个哈希表 *classToJarMap* 用于存储类与依赖包的映射关系 (第 2 行). 然后, 算法遍历依赖包集合 *DJ* 中的每一个依赖包 *depJar* (第 3 行), 并提取该依赖包的类集合 *depClasses* (第 4 行). 接下来, 算法将依赖包 *depJar* 添加到哈希表 *classToJarMap* 中对应类集合 *depClasses* 的映射关系中 (第 5 行). 完成依赖包遍历后, 算法接着遍历 *classToJarMap* 的所有类 (第 7 行). 对于每一个类 *class*, 算法获取对应的依赖包集合 *conflictJars* (第 8 行). 如果 *conflictJars* 的大小大于 1 且不在集合 *JS* 中, 算法将 *conflictJars* 添加到集合 *JS* 中 (第 9–10 行). 最后, 算法结束对类的遍历 (第 12 行), 并完成外部类冲突的检测.

3) 类别 1.3 库版本冲突 (如算法 A3 所示)

算法 A3. 库版本冲突检测.

输入: *DT*: 依赖树;
输出: *JS*: 出现库版本冲突的依赖库集合.

```
1.  JS ⇐ {}
2.  depToVersionMap ⇐ {}
3.  for dep ∈ DT do
4.      depName ⇐ dep.name
5.      depVer ⇐ dep.version
6.      depToVersionMap[depName].add(depVer);
7.  end for
8.  for dep ∈ depToVersionMap.keys do
9.      if depToVersionMap[dep].size() > 1 then
10.         JS.add(dep)
11.     end if
12. end for
```

该算法接受依赖树 *DT* 和调用图 *CG* 作为输入, 并输出使用但未声明的依赖库集合 *JS*. 算法的工作流程如下: 首先, 初始化未声明依赖库集合 *JS* (第 1 行). 接着, 创建一个集合 *directDeps* 用于存储直接依赖库 (第 2 行). 然后, 算法遍历依赖树 *DT* 中深度为 1 的依赖库 *dep* (第 3 行), 并将这些直接依赖库添加到集合 *directDeps* 中 (第 4 行). 完成依赖树的遍历后, 算法接着遍历调用图 *CG* 中直接调用的依赖库 *dep* (第 6 行). 对于每一个调用的依赖库 *dep*, 如果该依赖库不在集合 *directDeps* 中, 算法将其添加到未声明依赖库集合 *JS* 中 (第 7–9 行). 最后, 算法结束对调用图的遍历 (第 10 行), 并完成未声明依赖的检测.

4) 类别 1.4 未声明依赖 (如算法 A4 所示)

算法 A4. 未声明依赖检测.

输入: *DT*: 依赖树; *CG*: 调用图;
输出: *JS*: 使用但未声明的依赖库集合.

```
1.  JS ⇐ {}
2.  directDeps⇐ {}
3.  for dep ∈ DT and dep.depth = 1 do
4.      directDeps.add(dep)
5.  end for
6.  for dep ∈ CG.directCalledDeps do
7.      if not dep ∈ directDeps then
8.          JS.add(dep)
9.      end if
10. end for
```

该算法接受依赖树 *DT* 和调用图 *CG* 作为输入, 并输出未使用的依赖库集合 *JS*. 算法的工作流程如下: 首先, 初始化未使用依赖库集合 *JS* (第 1 行). 接着, 创建一个集合 *directDeps* 用于存储直接依赖库 (第 2 行). 然后, 算法遍历依赖树 *DT* 中深度为 1 的依赖库 *dep* (第 3 行). 对于每一个直接依赖库 *dep*, 算法检查该依赖库是否不在调用图 *CG* 的调用依赖集合 *CG.calledDeps* 中 (第 4 行). 如果该依赖库未被调用, 则将其添加到直接依赖库集合 *directDeps* 中 (第 5 行). 最后, 算法结束对依赖树的遍历 (第 7 行), 并完成未使用依赖的检测.

5) 类别 1.5 未使用依赖 (如算法 A5 所示)

算法 A5. 未使用依赖检测.

输入: *DT*: 依赖树; *CG*: 调用图;
输出: *JS*: 使用但未声明的依赖库集合.

```
1. JS ⇐ {}
2. directDeps ⇐ {}
3. for dep ∈ DT and dep.depth = 1 do
4.     if not dep ∈ CG.calledDeps then
5.         directDeps.add(dep)
6.     end if
7. end for
```

该算法接受依赖树 *DT* 和调用图 *CG* 作为输入, 并输出未使用的依赖库集合 *JS*. 算法的工作流程如下: 首先,

初始化未使用依赖库集合 *JS* (第 1 行). 接着, 创建一个集合 *directDeps* 用于存储直接依赖库 (第 2 行). 然后, 算法遍历依赖树 *DT* 中深度为 1 的依赖库 *dep* (第 3 行). 对于每一个直接依赖库 *dep*, 算法检查该依赖库是否不在调用图 *CG* 的调用依赖集合 *CG.calledDeps* 中 (第 4 行). 如果该依赖库未被调用, 则将其添加到直接依赖库集合 *directDeps* 中 (第 5 行). 最后, 算法结束对依赖树的遍历 (第 7 行), 并完成未使用依赖的检测.

6) 类别 1.7 依赖范围冲突 (如算法 A6 所示)

算法 A6. 依赖范围冲突检测.

输入: *DT*: 依赖树;
输出: *JS*: 出现冲突依赖范围误用的直接依赖集合.

```
1.  JS ⇐ {}
2.  depToScopeMap ⇐ {}
3.  for dep ∈ DT and dep.depth = 1 do
4.      depToScopeMap[dep] = dep.scope
5.  end for
6.  for dep ∈ DT and dep.depth ≠ 1 do
7.      if dep ∈ depToScopeMap.keys and dep.scope ≠ depToScopeMap[dep] then
8.          JS.add(dep)
9.      end if
10. end for
```

该算法接受依赖树 *DT* 作为输入, 并输出出现冲突依赖范围误用的直接依赖集合 *JS*. 算法的工作流程如下: 首先, 初始化冲突依赖集合 *JS* (第 1 行). 接着, 创建一个哈希表 *depToScopeMap* 用于存储直接依赖库及其对应的依赖范围 (第 2 行). 然后, 算法遍历依赖树 *DT* 中深度为 1 的依赖库 *dep* (第 3 行), 并将每个直接依赖库的依赖范围 *dep.scope* 添加到哈希表 *depToScopeMap* 中 (第 4 行). 完成直接依赖库的遍历后, 算法继续遍历依赖树中深度不等于 1 的依赖库 *dep* (第 5 行). 对于每一个依赖库, 算法检查该依赖库是否在哈希表 *depToScopeMap* 的键中, 且其依赖范围 *dep.scope* 是否与哈希表中记录的依赖范围不一致 (第 6 行). 如果存在不一致, 算法将该依赖库添加到冲突依赖集合 *JS* 中 (第 7 行). 最后, 算法结束对依赖树的遍历 (第 9 行), 并完成依赖范围冲突的检测.

7) 类别 1.8 依赖树冲突 (如算法 A7 所示)

算法 A7. 依赖树冲突检测.

输入: *DT*: 依赖树; *BI*: 基本信息;
输出: *JS*: Maven 和 Gradle 下出现不一致的直接依赖库.

```
1. currentToolMap ⇐ {}
2. otherToolMap ⇐ resolveDepFile(BI.otherToolDepFile)
3. for dep ∈ DT and dep.depth = 1 do
4.     currentToolMap[dep.depName] = dep.depVer
5. end for
6. JS ⇐ difference(currentToolMap, otherToolMap)
```

该算法接受依赖树 *DT* 和基本信息 *BI* 作为输入, 并输出 Maven 和 Gradle 下出现不一致的直接依赖库集合 *JS*. 算法的工作流程如下: 首先, 初始化一个哈希表 *currentToolMap* 用于存储当前工具 (如 Maven 或 Gradle) 的直

接依赖库及其版本 (第 1 行). 接着, 调用 *resolveDepFile* 函数解析其他工具的依赖文件, 并将结果存储在哈希表 *otherToolMap* 中 (第 2 行). 然后, 算法遍历依赖树 *DT* 中深度为 1 的直接依赖库 *dep* (第 3 行), 并将每个直接依赖库的名称 *dep.depName* 和版本 *dep.depVer* 添加到哈希表 *currentToolMap* 中 (第 4 行). 完成依赖树的遍历后, 算法通过调用 *difference* 函数计算当前工具与其他工具之间的差异, 并将结果赋值给不一致的直接依赖库集合 *JS* (第 6 行). 最后, 算法结束, 返回 Maven 和 Gradle 下的不一致依赖库.

8) 类别 2.1 构建工具配置缺失 (如算法 A8 所示)

算法 A8. 构建工具配置缺失检测.

输入: *BI*: 基本信息;
输出: *MC*: 项目是否缺失构建工具配置.

1. *buildToolConfigs* ⇐ *parseConfigFile*(*BI.builtToolConfigPath*)
2. *MC* ⇐ *buildToolConfig.empty*()

算法接受基本信息 *BI* 作为输入, 并输出项目是否缺失构建工具配置的标志 *MC*. 算法的工作流程如下: 首先, 调用 *parseConfigFile* 函数解析构建工具配置文件, 获取配置结果并存储在 *buildToolConfigs* 中 (第 1 行). 接着, 算法通过检查 *buildToolConfigs* 是否为空来确定项目是否缺失构建工具配置, 并将结果赋值给标志 *MC* (第 2 行). 最后, 算法结束, 返回项目缺失构建工具配置的状态.

9) 类别 2.2 构建工具启动器 JAR 缺失 (如算法 A9 所示)

算法 A9. 构建工具封装器 JAR 缺失检测.

输入: *BI*: 基本信息;
输出: *MJ*: 项目是否缺失构建工具封装器 JAR.

1. *buildToolConfigs* ⇐ *parseConfigFile*(*BI.builtToolConfigPath*)
2. *wrapperJAROnline* ⇐ *buildToolConfigs*[‘*onlineWrapperJAR*’].*empty*()
3. *wrapperJAROffline* ⇐ *BI.defaultWrapperJAR.exist*()
4. *MJ* ⇐ *wrapperJAROnline* **or** *wrapperJAROffline*

该算法接受基本信息 *BI* 作为输入, 并输出项目是否缺失构建工具封装器 *JAR* 的标志 *MJ*. 算法的工作流程如下: 首先, 调用 *parseConfigFile* 函数解析构建工具配置文件, 并将结果存储在 *buildToolConfigs* 中 (第 1 行). 接着, 算法检查在配置文件是否指定了在线下载封装器 *JAR* 是否为空, 并将结果赋值给布尔变量 *wrapperJAROnline* (第 2 行). 随后, 算法检查默认本地封装器 *JAR* 是否存在, 并将结果存储在布尔变量 *wrapperJAROffline* 中 (第 3 行). 最后, 算法通过逻辑或运算将两个布尔值结合, 确定项目是否缺失构建工具封装器 *JAR*, 并将结果赋值给标志 *MJ* (第 4 行).

10) 类别 2.3 构建工具启动器 JAR 异常 (如算法 A10 所示)

算法 A10. 构建工具封装器 JAR 异常检测.

输入: *BI*: 基本信息;
输出: *AC*: 项目构建工具封装器是否与构建工具版本一致.

1. *buildToolConfigs* ⇐ *parseConfigFile*(*BI.builtToolConfigPath*)
2. *buildToolVer* ⇐ *buildToolConfigs*[‘*builtToolVersion*’]

```
3. wrapperJAROffline ⇐ BI.defaultWrapperJAR
4. if not wrapperJAROffline.exist() then
5.     Return
6. end if
7. localChecksum ⇐ calculateLocalChecksum(wrapperJAROffline)
8. officialChecksum ⇐ getOnlineChecksum(buildToolVer)
9. AC ⇐ localChecksum == officialChecksum
```

该算法接受基本信息 *BI* 作为输入, 并输出项目构建工具封装器是否与构建工具版本一致的标志 *AC*.算法的工作流程如下: 首先, 调用 *parseConfigFile* 函数解析构建工具配置文件, 并将结果存储在 *buildToolConfigs* 中 (第 1 行). 接着, 算法从配置中提取构建工具版本, 并将其赋值给变量 *buildToolVer* (第 2 行). 随后, 算法获取默认封装器 *JAR*, 并将其存储在变量 *wrapperJAROffline* 中 (第 3 行). 接下来, 算法检查 *wrapperJAROffline* 是否存在, 若不存在则提前返回, 结束算法 (第 4–6 行). 如果存在, 算法计算本地封装器 *JAR* 的校验和, 并将结果存储在变量 *localChecksum* 中 (第 7 行). 然后, 算法获取对应版本的官方校验和, 并存储在变量 officialChecksum 中 (第 8 行). 最后, 算法通过比较本地校验和与官方校验和来确定构建工具封装器是否一致, 并将结果赋值给标志 *AC* (第 9 行).

11) 类别 2.4 模块间库重复 (如算法 A11 所示)

算法 A11. 模块间库重复检测.

输入: *BI*: 基本信息;
输出: *JS*: 未统一版本管理的依赖库集合.

```
1.  JS ⇐ {};
2.  moduleDirectDepsCnt ⇐ {}
3.  for module ∈ BI.modules do
4.      for dep ∈ resolveDepFile(module.depFile) do
5.          moduleDirectDepsCnt[dep] += 1
6.      end for
7.  end for
8.  controlledDeps ⇐ BI.managedDeps
9.  for dupDep ∈ moduleDirectDepsCnt.keys do
10.     if moduleDirectDepsCnt[dupDep] > 1 and not dupDep ∈ controlledDeps then
11.         JS.add(dupDep)
12.     end if
13. end for
```

该算法接受基本信息 *BI* 作为输入, 并输出未统一版本管理的依赖库集合 *JS*. 算法的工作流程如下: 首先, 初始化集合 *JS* 用于存储未统一版本管理的依赖库 (第 1 行). 接下来, 创建一个字典 *moduleDirectDepsCnt* 用于记录各个依赖库在模块中的直接依赖计数 (第 2 行). 算法遍历多模块项目中的每个模块 (第 3 行), 并对每个模块的依赖文件调用 *resolveDepFile* 函数以解析其依赖 (第 4 行). 对于每个解析到的依赖 *dep*, 算法将其计数加 1 (第 5 行). 完成模块遍历后, 算法从基本信息中获取受控依赖集合, 并将其赋值给变量 *controlledDeps* (第 8 行). 随后, 算法遍历 *moduleDirectDepsCnt* 字典中的每个重复依赖 *dupDep* (第 9 行), 并检查该依赖的计数是否大于 1 且不在受控依赖集合中 (第 10 行). 如果条件满足, 算法将该依赖 *dupDep* 添加至集合 *JS* 中 (第 11 行).

12) 类别 2.5 模块间库冲突 (如算法 A12 所示)

算法 A12. 模块间库冲突.

输入: *BI*: 基本信息;
输出: *JS*: 各模块间出现版本冲突的依赖库集合.

```
1. JS ⇐ {};
2. moduleDirectDepsVers ⇐ {}
3. for module ∈ BI.modules do
4.     for dep ∈ resolveDepFile(module.depFile) do
5.         moduleDirectDepsVers[dep.depName].insert(dep.depVer)
6.     end for
7. end for
8. for conflictDep ∈ moduleDirectDepsCnt.keys do
9.     if moduleDirectDepsVers[conlictDep].size() > 1 then
10.        JS.add(conflictDep)
11.    end if
12. end for
```

该算法接受基本信息 *BI* 作为输入, 并输出各模块间出现版本冲突的依赖库集合 *JS*. 算法的工作流程如下: 首先, 初始化集合 *JS* 用于存储发生版本冲突的依赖库 (第 1 行). 接下来, 创建一个字典 *moduleDirectDepsVers* 用于记录每个依赖库在各模块中的版本信息 (第 2 行). 算法遍历多模块项目中的每个模块 (第 3 行), 并对每个模块的依赖文件调用 *resolveDepFile* 函数以解析其依赖 (第 4 行). 对于每个解析到的依赖 *dep*, 算法将其版本 *dep.depVer* 插入到字典 *moduleDirectDepsVers* 中, 使用依赖名称 *dep.depName* 作为键 (第 5 行). 完成模块遍历后, 算法遍历 *moduleDirectDepsVers* 字典中的每个依赖 *conflictDep* (第 8 行), 并检查该依赖的版本集合大小是否大于 1 (第 9 行). 如果条件满足, 算法将该依赖 *conflictDep* 添加至集合 *JS* 中 (第 10 行).

孙伟杰(2001－), 男, 博士生, CCF 学生会员, 主要研究领域为开源治理技术.

王莹(1987－), 女, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为开源治理技术, 软件测试及分析学.

许畅(1977－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为大数据软件工程, 智能软件测试与分析, 自适应和自控软件系统.