软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(10):4636−4660 [doi: 10.13328/j.cnki.jos.006666]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

第三方库依赖冲突问题研究综述*

李 硕[1,2], 刘 杰[1,2,3], 王 帅[1], 田浩翔[1,2], 叶 丹[1,2,3]

[1](中国科学院 软件研究所, 北京 100190)
[2](中国科学院大学, 北京 100049)
[3](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)
通信作者: 刘杰, E-mail: ljie@otcaix.iscas.ac.cn; 叶丹, E-mail: yedan@otcaix.iscas.ac.cn

摘 要: 软件开发过程中, 开发人员通过大量使用第三方库来实现代码复用. 不同第三方库之间存在依赖关系, 第三方库间的不兼容会导致第三方库的安装、加载、调用时出现错误, 进而导致系统异常, 这类问题称之为第三方库依赖冲突问题. 依赖冲突的根本原因是加载的第三方库无法覆盖软件引用的必需特性 (例如: 方法). 依赖冲突问题会在第三方库的下载安装, 项目编译和运行时中出现, 且定位困难. 依赖冲突问题的修复要求开发人员对使用的第三方库版本间差别具有准确的理解, 并且第三方库之间复杂的依赖关系增加了修复难度. 为了能够在软件运行前, 发现软件中存在的依赖冲突, 并且能够响应和处理运行过程中由依赖冲突引发的系统异常, 国内外学者展开了各种针对依赖冲突问题的研究. 从依赖冲突问题的 4 个方面, 对当前已有研究工作进行了梳理, 包括: 第三方库的使用实证分析、依赖冲突原因分析、依赖冲突检测方法以及依赖冲突常用修复方式. 最后对该领域未来值得关注的研究问题进行了展望.
关键词: 依赖冲突; 第三方库; 软件生态系统; 依赖管理; 函数接口兼容性
中图法分类号: TP311

中文引用格式: 李硕, 刘杰, 王帅, 田浩翔, 叶丹. 第三方库依赖冲突问题研究综述. 软件学报, 2023, 34(10): 4636–4660. http://www.jos.org.cn/1000-9825/6666.htm
英文引用格式: Li S, Liu J, Wang S, Tian HX, Ye D. Survey on Dependency Conflict Problem of Third-party Libraries. Ruan Jian Xue Bao/Journal of Software, 2023, 34(10): 4636–4660 (in Chinese). http://www.jos.org.cn/1000-9825/6666.htm

Survey on Dependency Conflict Problem of Third-party Libraries

LI Shuo[1,2], LIU Jie[1,2,3], WANG Shuai[1], TIAN Hao-Xiang[1,2], YE Dan[1,2,3]
[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)
[3](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

Abstract: During software development, developers use third-party libraries extensively to achieve code reuse. Due to the dependencies among different third-party libraries, the incompatibilities among them lead to errors during the installing, loading, or calling of those libraries and ultimately result in system anomalies. Such a problem is called a dependency conflict (DC, also referred to as conflict dependency or CD) issue of third-party libraries. The root cause of such issues is that the third-party libraries loaded fail to cover the required features (e.g., methods) cited by the software. DC issues often occur during the download and install, project compiling, and running of third-party libraries and are difficult to locate. Fixing DC issues requires developers to know the differences among the versions of the third-party libraries they use accurately, and the complex dependencies among the third-party libraries increase the difficulty in this work. To identify the DC issues in the software before its running and to deal with the system anomalies caused by those issues during

* 基金项目: 国家重点研发计划 (2017YFA0700603); 国家自然科学基金 (61972386)
收稿时间: 2021-10-09; 修改时间: 2021-12-28, 2022-01-06, 2022-01-16; 采用时间: 2022-02-18; jos 在线出版时间: 2022-05-24
CNKI 网络首发时间: 2023-04-06

© 中国科学院软件研究所 http://www.jos.org.cn

running, researchers around the world have conducted various studies on such issues. This study presents a systematic review of this research topic from four aspects, including the empirical analysis of third-party library usage, the cause analysis of DC issues, and the detection methods and common fixing ways for such issues. Finally, the potential research opportunities in this field are discussed.
Key words: dependency conflicts; third-party libraries; software ecosystem; dependency management; API compatibility

1 引 言

1.1 研究背景

第三方库 (又称第三方依赖或软件包) 是一种重要的可复用软件资源, 在一定条件下, 可以独立于其他第三方库进行安装和删除[1]. 软件开发过程中调用第三方库是一种提高开发效率的普遍做法, 特别是随着现代信息技术的迅速发展, 软件规模不断增加, 软件开发对第三方库的依赖也在不断增加. 已有研究工作表明一个项目直接依赖于多个不同的库[2]. 此外, 由于第三方库之间存在依赖关系, 一个项目往往会被隐式的加入更多的依赖关系, 导致依赖更多的库[2–5].

第三方库的使用大大提升了软件开发效率, 但也给软件开发带来了潜在的风险. 当加载的第三方库无法覆盖所在项目的必需特性 (例如, 方法) 时, 就会发生依赖冲突问题. 结合软件工程的开发实践, 开发人员可能会采用 build tools 自动完成第三方库的下载安装和项目编译, 也可能根据配置文件, 手动下载安装第三方库, 再执行项目编译. 因此, 依赖冲突问题会在第三方库下载安装、项目编译和运行过程中出现, 比如: 克隆了 GitHub 上的某个项目, 需要下载第三方库时, 由于没有满足引用的第三方库之间的约束条件, 导致第三方库下载安装失败; 在项目编译时, 由于本地环境中, 存在与引用的第三方库重名的代码文件, 导致无法加载正确的第三方库或使用了第三方库中不存在的相关方法, 项目编译报错; 在运行阶段, 由于加载的第三方库版本与所需版本不一致, 提示找不到某些类或方法, 抛出异常, 程序运行失败.

目前对依赖冲突的解决方式多为开发人员借助经验知识去处理, 会消耗大量的时间和人工成本. 运行过程中的报错信息在多数情况下不足以支撑开发人员找到问题根源, 并且由于第三方库之间存在复杂的调用关系, 更改某一第三方库同时可能引起其他第三方库的不适用, 这些问题增加了定位和解决依赖冲突问题的难度. 同时依赖冲突问题会伴随如下特点: 难以重现问题出现的场景、编译器提示的问题报告中描述信息不够清晰、第三方库代码缺失、第三方库代码复杂难以调试等. 这些问题在开发过程中给开发人员带来了巨大挑战.

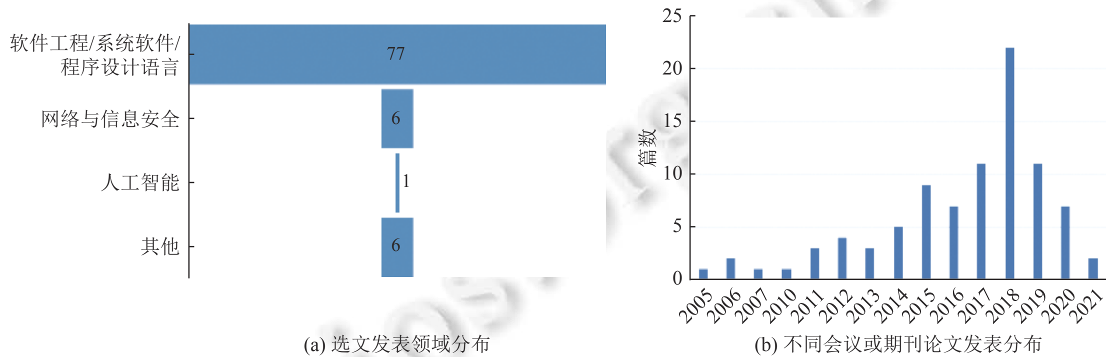
依赖冲突问题的检测和修复成为当前软件开发中的重要技能. 为了避免依赖冲突问题的出现, 第三方库的开发人员提出了一种声明机制, 即在 metadata 中对库本身依赖的其他第三方库信息进行声明. 但是由于 metadata 信息需要开发人员手动维护, 难以保证其时效性和准确性. Maven[6–8]、Pip[9]等成熟的软件管理工具也提供了依赖关系管理功能, 帮助开发人员在开发过程中选择依赖的第三方库版本. 但由于多种复杂因素, 这些工具不能保证加载的是最契合开发人员需求的版本, 并且由于开发人员可能缺乏对依赖的第三方库的了解, 导致错误的几率增加.

1.2 文献检索方式

针对以上分析, 研究人员对第三方库的使用情况和依赖冲突问题的产生原因、检测方法、修复方式都开展了大量研究. 为了对该问题的已有研究工作和成果进行系统的梳理, 我们首先选择了文献搜索的关键词: dependency conflict、conflict dependency、third-party library、第三方库、依赖冲突、包管理. 随后在谷歌学术搜索引擎 (Google Scholar)、ACM Digital Library、IEEE Explore、Springer、Elsevier 以及 CNKI 等搜索引擎和数据库上, 通过上述关键词来检索与本文主题相关的论文. 对上述检索的文献进行筛选, 筛选条件包括: 与本文主题的相关性、文献质量. 通过文献标题、关键词、摘要等信息, 识别并移除与主题无关的文献; 在与主题相关的文献中, 选择发表会议/期刊属于 CCF 推荐的国际学术会议/期刊的文章. 通过 Google Scholar、ACM Digital Library、IEEE Explore 等学术搜索引擎来查阅论文的被引用情况和相关研究人员已发表论文的清单, 进一步补充与本文研究主题相关的文献集. 最终确定了与本文主题相关的高质量论文.

© 中国科学院软件研究所 http://www.jos.org.cn

经过筛选后, 我们发现尚未有对依赖冲突问题进行全面综述的文献. 早在 2006 年 Mancinelli 等人[10]、2012 年 Artho 等人[1], 介绍的依赖冲突有关问题, 是针对 Linux 中的依赖冲突问题. 而后学术界和工业界逐渐对依赖冲突问题进行关注. 为了使得本文内容更加完整, 且更好地反映相关研究的发展脉络, 本文中所介绍论文与已有论文可能会存在少部分重叠. 最终, 本论文选取了 90 篇相关论文进行详细介绍, 图 1 展示了搜集论文统计结果. 参照 CCF 推荐领域划分标准, 图 1(a) 展示了所选论文主要集中在软件工程/系统软件/程序设计语言领域. 图 1(b) 展示了所选文章发表会议/期刊的分布情况, 其中包括会议论文 71 篇, 期刊论文 16 篇; 大多数论文是所涉及领域的高质量会议和期刊, 例如 ICSE 会议 (14 篇)、ESEC/FSE 会议 (11 篇)、ASE 会议 (5 篇)、CCS 会议 (2 篇)、OOPSLA (1 篇)、ECOOP 会议 (1 篇)、NDSS 会议 (2 篇)、ESE 期刊 (4 篇)、TSE 会议 (4 篇)、IST 期刊 (2 篇)、《软件学报》(3 篇).



(a) 选文发表领域分布 (b) 不同会议或期刊论文发表分布

图 1　选文统计结果

1.3　研究问题

本文在对当前已有研究成果梳理基础上, 对依赖冲突问题进行了全面的综述, 包括 4 个方面: 第三方库的使用情况, 依赖冲突原因分析, 有关检测方法以及当前开发人员采用的修复方式, 并对该领域未来潜在的值得关注的研究方向进行展望. 本文主要选择了 GitHub 使用语言排行榜[11]中的前 3 种使用最为广泛的语言 (JavaScript、Java 和 Python) 以及多个生态系统中依赖冲突问题作为研究对象进行分析, 并尝试回答以下 4 个问题.

问题 1: 有关第三方库使用情况以及开发情况是怎样的? 与依赖冲突问题的关系是怎样的?

问题 2: 依赖冲突问题发生的原因是什么, 可以被划分为哪些模式?

问题 3: 依赖冲突问题检测方法有哪些, 特点是什么?

问题 4: 目前开发人员修复依赖冲突的方式是什么?

本文第 2 节给出整体研究框架. 第 3 节对依赖冲突问题的相关实证分析, 进行总结分类. 第 4 节系统分析依赖冲突问题发生的根本原因. 第 5 节对依赖冲突问题的检测方法进行分类阐述, 并对工具进行对比. 第 6 节总结依赖冲突问题的修复方式. 最后对该领域未来潜在值得关注的研究方向进行展望.

2　第三方库依赖冲突问题与研究框架

2.1　第三方库依赖冲突问题简介

大量引用第三方库极易引发依赖冲突问题. 此类问题的出现是由于当同一个库的多个版本存在于相同路径下时, Java、Python、JavaScript 等第三方库的加载机制规定了第三方库在运行时的加载位置[12], 多数情况下当两个第三方库无法共存时, 那么将只有一个第三方库被加载, 而与之存在冲突的其他第三方库无法安装或无法加载[5], 当项目在运行时引用不兼容的、被隐藏掉的第三方时, 将引发运行时异常 (例如, ClassNotFoundException 和 NoSuchMethodError[2]). 此类问题称之为依赖冲突问题 (conflict dependency, 或是 dependency conflict). 除两个第三方库包含内容直接导致冲突外, 也可能因为其依赖的底层第三方库互相冲突. 因此, 两个看似毫无关联的第三方

© 中国科学院软件研究所 http://www.jos.org.cn

库也可能因为依赖冲突而无法安装[13]. 第三方库依赖冲突的表现形式是依赖的第三方库和实际使用的第三方库不一致.

图 2 为 Keras 框架中出现的依赖冲突问题实例[14], 当开发人员加载的 TensorFlow 版本为 1.X 时, 程序无法正常运行, 提示 AttributeError, 其原因为加载的类不包含项目需要引用的方法, 此实例为典型的依赖冲突问题, 此种类型常出现在运行阶段异常[2,15−18], 其他类型的依赖冲突问题也可能出现在第三方库的安装阶段和项目编译过程[17,19]. 伴随着大型系统的普及, 第三方库之间的冲突问题更加广泛频繁出现. 原因在于大型系统广泛依赖于第三方库, 并且大型系统的开发多采用微服务、多模块的开发方式; 一个大型系统, 分为多个服务, 由不同的团队进行开发, 开发过程中, 协调依赖的第三方库、保证库以及对应的版本的一致性非常困难. Jibesh 等人[16]进行的实证研究分析发现, 1/4 的第三方库存在潜在的冲突, 可能导致软件崩溃或其他意外的行为. 已有研究工作主要集中在对依赖冲突问题的实证研究、对依赖冲突表现形式的分析、对依赖冲突问题的检测方法的研究以及对依赖冲突问题的解决方法的研究.

```
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py in placeholder(shape, ndim, dtype, sparse, name)
539 x = tf.sparse_placeholder(dtype, shape=shape, name=name)
540 else:
--> 541 x = tf.compat.v1.placeholder(dtype, shape=shape, name=name)
542 x._keras_shape = shape
543 x._uses_learning_phase = False

AttributeError: module 'tensorflow' has no attribute 'placeholder'
```

图 2　依赖冲突实例

常见的依赖冲突问题表现形式多样: 安装过程中的表现形式为安装第三方库失败, 提示异常信息; 运行过程中的表现形式为编译器提示无法找到需要的类或方法; 或通过了编译过程, 但无法得到正确的运行结果[18].

依赖冲突问题的原因复杂多样, 本文将其概括为 3 种类别, 包括: 版本冲突、版本误用、名称冲突. 版本冲突包含两种类型: 同一个第三方库之间的冲突和不同第三方库之间的冲突. 同一个第三方库之间的冲突可以划分为直接依赖与传递依赖之间的冲突、传递依赖之间的冲突、所需第三方库与本地安装的第三方库之间的冲突. 不同第三方库之间的冲突, 可以划分为直接依赖间的冲突、直接依赖与传递依赖之间的冲突、传递依赖之间的冲突. 版本误用表现形式为对第三方库版本的误用导致的依赖冲突. 名称冲突指项目代码和第三方库之间的命名冲突, 此种类型较为少见.

目前对依赖冲突问题的检测工具相对较少, 主要集中在对第三方库的冲突问题进行预判, 提示开发人员可能存在的风险[15,19,20], 也有部分工作针对存在依赖冲突问题的项目生成测试用例, 报告出现问题的原因[2,16−18]. 对于依赖冲突问题的修复, 目前主要依靠人工修复方式完成.

2.2　研究框架

本文的整体研究框架如图 3 所示. 按照 GitHub 使用语言排行榜[11], 主要关注排行榜中前 3 名的语言 JavaScript、Python、Java. JavaScript 社区、Java 社区和 Python 社区包含了对应语言以及对语言有关问题的讨论. 本文提及的 npm、PyPI 代指 npm 生态系统、PyPI 生态系统. npm 生态系统、Maven 生态系统和 Eclipse 社区、PyPI 生态系统为对应语言提供了第三方库的下载以及管理通道. 其他较大规模社区, 例如 R/CRAN 社区, Android 社区以及 Linux 社区等, 是在本文的研究过程中, 发现的相关社区, 作为补充内容.

JavaScript 和 Python 是动态类型的脚本语言, Java 是静态类型的程序设计语言. 3 种语言具有各自的特性, 包管理模式间也存在差异, 但三者在依赖冲突问题上可以抽取共性. 依赖冲突问题发生在第三方库的使用过程中, 因此本文首先对第三方库使用的相关研究工作进行了分类汇总; 其次对导致依赖冲突的原因进行了分类分析; 接着参照依赖冲突的原因对依赖冲突问题的检测方法进行分类, 并对比总结检测工具, 然后参照开发人员采取的 7 种修复方式进行分类, 最后进行总结并指出未来的潜在研究问题.

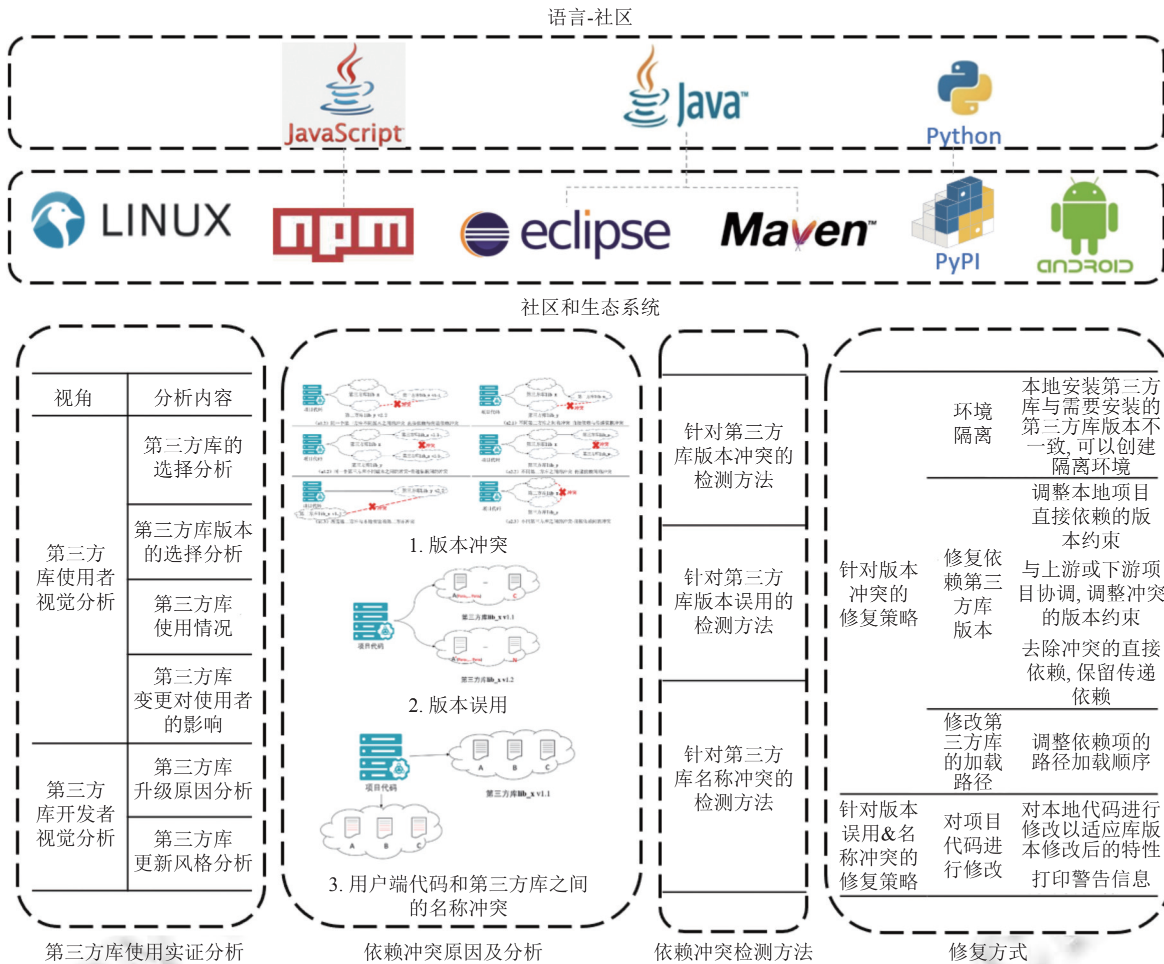


图 3　研究框架

3　第三方库使用实证分析

针对问题 1, 本文对第三方库使用的相关研究工作进行整合, 分别从第三方库的使用者视角和第三方库的开发者视角, 对依赖冲突问题进行总结分析, 主要分为 6 个分析对象, 对应的分析内容如表 1 所示.

对于第三方库使用者视角分析, 本文从第三方库的选择 (选择哪个第三方库)、第三方库版本的选择 (具体版本的选择)、第三方库使用情况, 以及第三方库变更对使用者的影响 4 个方面进行阐述. 本文发现, 对于第三方库的选择, 更倾向于选择被广泛使用的第三方库, 并且开发人员对第三方库的依赖性逐渐增强, 但对于使用第三方库带来的潜在影响以及风险关注度较低; 对于第三方库版本的选择, 开发人员对版本的选择具有不同的倾向, 集中在对项目性能的影响方面的关注; 对于第三方库使用情况分析发现, 第三方库 API 使用不均衡; 第三方库版本变更带来的影响, 由于内容的变更程度不同, 其对本地项目的影响范围亦有所不同.

第三方库开发者视角下的分析内容主要集中在第三方库升级原因和第三方库更新风格两个部分. 第三方库升级原因主要包括实现新需求、提升可维护性、偿还技术债 3 个方面; 不同语言、不同生态系统的库开发者对第三方库的更新风格不一致, Java 语言的 Eclipse 生态系统更倾向于保持向后兼容的升级, R/CRAN 社区更倾向于协作交流, 而 Node.js/npm 生态系统更倾向于对内容大幅度变更, 对兼容性保持较低.

© 中国科学院软件研究所 http://www.jos.org.cn

表 1　第三方库使用分析

视角	分析对象	结论	有关文献
第三方库使用者视角	第三方库的选择	● 开发人员对第三方库的依赖性增强, 并且倾向于使用成熟度较高、使用更加广泛的第三方库. 但开发人员对当前第三方库管理工具提供的语义版本控制使用较少 ● 根据不同使用情况选择不同的衡量指标, 同一种语言的库的衡量指标也有多个, 例如使用GitHub排名、开发社区下载数量排名等 ● 开发人员在选择第三方库时, 关注点在于库是否提供的特定功能, 但对性能、使用库带来的其他潜在风险的关注度较低	[21−32]
	第三方库版本的选择	● 对于不同语言、不同生态系统、不同阶段的开发人员而言, 其采用的版本具有不同的倾向. 总体而言, 开发人员更倾向于选择较为稳定的版本, 并且对有关语义版本控制工具的采用较少 ● 在使用第三方库时, 时常会面临版本更新的权衡问题: 是为了保持项目稳定, 将第三方库版本进行限定; 还是不断迭代更新第三方库的版本, 以使用新特性. 使用人员更新或者不更新第三方库版本的原因是不尽相同的 ● 但由于语言特性、使用场景以及社区规约不同等因素, 开发人员对是否更新第三方库的偏好也有所不同. 版本升级的周期也有所不同	[24,26,28,33−47]
	第三方库使用情况	● 开发者对第三方库中API的利用率不均衡, 即使加载完整的第三方库相较于开发人员手动编写代码实现有关功能, 会消耗更多资源, 但第三方库的运行效率通常会远高于开发人员手动编写代码 ● 由于开发人对API的使用规约了解甚少, 开发人员常会使用一些已经被声明为将被弃用的类或方法, 并且常会误用 API	[16,33,48−52]
	第三方库变更对使用者的影响	● 库版本的变更会对库使用者带来巨大的影响, 因为其版本变更常存在不兼容的更改, 容易造成依赖冲突问题, 需要本地代码进行重构来适配库版本的变更 ● 研究表明, 第三方库的更改大多数情况下是不兼容的更改, 研究人员有意避免版本变更带来的影响, 但由于测试不充分等原因, 仍可能造成依赖冲突问题. 开发人员会检查库的变更对代码的影响, 必要时候进行代码的重构来适配库的变更 ● 研究人员从不同角度分别提出了识别第三方库中API发生变化的有关方法、对被声明为弃用的API寻找替代方案、检测API的修改是否影响了开发人员的调用	[26,46,48,53−65]
第三方库开发者视角	升级原因	● 对于不同语言、不同生态系统, 第三方库的升级是常见的, 库的升级原因基本一致, 主要包括: 实现新需求, 提升可维护性, 偿还技术债	[28,66−68]
	更新风格	● 不同语言、不同生态系统的第三方库开发者对库的修改风格是不同的, 但大多数都会考虑到代码兼容性问题, 防止依赖冲突问题的出现	[46,48,69−75]

对第三方库使用的相关研究工作分析整理后发现, API 兼容性问题是导致依赖冲突的重要原因之一; 第三方库的升级往往是对其中的 API 进行了修改, 这种修改是导致版本不兼容的重要原因. 因此, 对第三方库升级和 API 兼容性进行分析, 是依赖冲突问题分析中, 不可或缺的一部分. 例如, Artho 等人[1]指出第三方库的升级会导致库之间的依赖冲突问题, Wang 等人[18]揭示了在第三方库依赖冲突问题中, 存在用户期望使用的第三方库与实际加载的不一致, 这种不一致是由于加载的版本不符合用户的需求. 第三方库版本的升级需要向前兼容, 升级版本没有做好兼容性处理, 可能会导致依赖冲突问题的出现. 由此可见, 依赖冲突问题与 API 兼容性问题、第三方库升级存在不可忽视的相关性, 因此本文对第三方库升级、API 兼容性问题等与依赖冲突有关的内容进行了分析.

3.1 第三方库使用者视角分析

本节从第三方库的使用者角度对第三方库的使用情况, 包括第三方库的选择、第三方库版本的选择、第三方库内容的使用情况、第三方库变更带来的影响这 4 个方面进行阐述.

3.1.1 第三方库的选择分析

对于第三方库使用者而言, 第三方库的使用已经成为开发过程中不可或缺的一部分, 开发人员更倾向于使用第三方库, 并且对第三方库的依赖性不断增强. 不论是何种语言、何种生态系统, 每个项目都会使用大量的第三方库[22], 项目之间的依赖性在演进过程中不断增加[23,24], 项目依赖的包数量一般遵循指数级增长趋势[25−27]. 并且对于经验较少的开发人员倾向使用“trival”的包[22], 这类包的特点是实现较为简单的任务; 经验丰富的开发人员相反, 更倾向于使用成熟度高、使用较为广泛的第三方库. 但总体而言, 不论是经验丰富还是经验较少的开发人员, 很大

程度上依赖于一组使用较为广泛的库.

在使用过程中如何选择具体的第三方库, 对于不同语言的社区, 衡量指标具有差异. 开发人员往往根据不同使用情况选择不同的衡量指标, 同一种语言的库的衡量指标也有多个. 例如, Wittern 等人[28]发现 npm 生态系统中, 对第三方库的使用情况有 3 种不同的衡量指标: 第三方库排名 (类比 PageRank 排名)、下载数量排名、GitHub 排名. 不同的衡量标准不能相互替代, 它们可以用来描述特定类型包的流行程度. 排名机制的不同, 直接影响对第三方库推荐工具的设计. 并且, 对每一种排名来说, 新的第三方库并不能直接进入排名前列, 但有一些第三方库随着时间推移仍保留在排名靠前的位置.

开发人员在第三方库的选择过程中, 主要关注于是否实现了需要的功能, 但对性能、使用库带来的潜在风险关注度较低. Wang 等人[21]发现超过一半的项目使用了包含安全缺陷的第三方库, 并且 2/3 的第三方库包含安全漏洞. 这些安全漏洞的普遍存在表明, 如果开发人员不知道已使用库中的安全问题, 则项目可能面临潜在风险. Zimmermann 等人[27]通过分析第三方库之间的依赖关系和已被发现的安全问题来研究 npm 生态系统中的安全风险, 研究发现第三方库的可用性使开发人员依赖于越来越多的第三方库, 多个第三方库间的关联性增加, 使得单个库能够影响其他库的安全性, 从而增加了第三方库存在漏洞的可能性, 使项目受到攻击的风险大大增加. Black Duck[76]软件中的 Codecenter 部分, 提示开源代码中是否存在安全漏洞问题. 现实世界中已经发生了多起由第三方库漏洞引发的严重事故, 如: OpenSSL 中出现的心脏滴血漏洞 (heartbleed)、GNU Bash 出现的破壳漏洞 (shellshock) 和 Java 中的反序列化漏洞 (deserialization). Veracode[77]、SAP[65]以及 OWASP[78,79]通知开发人员使用的库版本中的安全错误. Pfretzschner 等人[80]将分析工具集成到 OpenWhisk (一个开源的无服务器云平台), 为开发人员提示使用了用于恶意行为的依赖库. VAS[81]建立在 OWASP[78]之上, 从开发人员的依赖配置文件中抽取直接依赖的第三方库, 与有漏洞的第三方库的名称进行匹配. 当名称匹配成功时, 认为项目是有漏洞的. 现有对 Java[30]和 Android 应用[31,32]的漏洞分析主要关注的就是第三方库带来的安全风险, 并且发现了第三方库代码中存在潜在的安全威胁. Linares-Vásquez 等人[29]研究发现, 与不成熟的应用程序使用的 API 相比, 较为成熟的应用程序所使用的 API 明显更不容易出错.

3.1.2 第三方库版本的选择分析

开发人员对于第三方库的选择具有不同的倾向, 在开发过程中, 开发人员会根据实际需求, 对第三方库的版本选择进行考量; 此外, 使用不同语言的开发人员, 对第三方库版本更新的偏好亦有所不同. 本节对开发人员选择和更新第三方库版本的偏好进行分析.

不同语言、不同生态系统、不同阶段的开发人员, 选择的第三方库版本具有不同的倾向. 总体上看, 开发人员更倾向于选择较为稳定的版本, 但对有关语义版本控制工具的采用较少. Blincoe 等人[24]对数百万种依赖关系进行了深入研究. 他们发现有经验的开发人员倾向于放弃灵活的版本控制, 实际上大多数库管理工具都鼓励第三方库遵循语义版本控制, 但开发人员很少使用现有库管理工具中的语义版本控制[21], 因为他们经历过与兼容性相关的错误, 更喜欢使用稳定的版本, 而且他们重视稳定的 (可重现的) 项目构建, 因此大多数项目声明了它们所引用的第三方库的固定版本. 不同语言的开发人员对版本的管理有所不同, 通常 JavaScript 语言的开发人员不会对版本进行强制约束, 并且多达一半的开发人员偏向自动依赖于最新版本[28]; 而 Java 语言的开发人员考虑到稳定性、健壮性和安全性, 更倾向于选择以前的稳定版本, 而不是最新的版本[33]; 此外, Python 语言开发者也倾向于使用较为稳定的版本. 实际上, 多达一半的开发人员在开发过程中会自动依赖于最新版本, 并且随着时间的增长, 开发人员倾向于使用更新的第三方库. Zhang 等人[34]发现库的版本变更中, API 的修改升级可能是深度学习程序中各种 bug 的根源. Wu 等人[35]提出了一种识别框架演化规则的方法. 即使第三方库的维护人员采用了各种编号约定库版本, 但版本号本身并不能代表库成熟度. 然而在实际使用中, 开发人员会倾向于认为版本号较高的库性能更加稳定.

开发人员在使用第三方库时, 时常会面临版本更新的权衡问题: 是为了保持项目稳定, 将第三方库版本进行限定; 还是不断迭代更新第三方库的版本, 以使用新特性. 使用人员更新或者不更新第三方库版本的原因是不尽相同的. 开发人员选择不更新第三方库版本的原因, 包括: 更新第三方库可能会引入兼容性问题, 开发人员需要对项目进行大量修改; 可能带来未知的 bug 或安全问题[36,37]; 可能造成上下游项目的依赖冲突问题[37,40]; 更新第三方库所

© 中国科学院软件研究所 http://www.jos.org.cn

需要的传递依赖的版本, 可能不符合项目约束, 导致无法更新[41], 或者更新后的版本与当前版本差距较小, 对项目没有任何影响, 徒增更新成本和风险[38,39,42–44]. 开发人员选择更新第三方库版本的原因, 包括: 更新库的版本可以使用库的新特性[36,43]; 修复旧版本第三方库存在的 bug 或安全漏洞[41]; 进行预防性维修任务[39]; 由于开源社区 (生态系统) 的政策规约, 开发人员需要定期对使用的第三方库进行更新[37]; 由于使用版本控制工具, 自动升级相库[45].

除了一些常见的共性原因外, 不同社区 (生态系统) 的开发人员由于语言特性、使用场景以及社区规约不同, 对是否更新第三方库的偏好也有所不同. npm 社区 (生态系统) 中, Decan 等人[37]研究发现, 社区政策会影响开发人员对第三方库的更新. npm, Python, CRAN 和 RubyGems 这 4 个生态系统中, 当采用了版本控制工具时, 第三方库版本的更新频率会增加[35,42,45]. 对于 3 个主流语言社区进行调研发现[53], 开发人员执行升级是因为考虑到机会成本, 例如维护过时代码的成本、围绕已知 bug 进行修改的成本、推迟新特性的成本; 并且项目中某一依赖版本的升级, 可能会影响其他依赖版本的升级, 否则会引发一系列的连锁反应. Maven 社区中, Kula 等人[41]以及 Cox 等人[39]研究发现对于安全性相关问题关注度较低, 因此没有更新第三方库. Android 社区中, Salza 等人[43]研究发现, 库的更新与库的用途息息相关, 例如开发人员对于 APP 的 GUI 等外观相关的特性较为关注, 为了迎合大众审美需求, 对第三方库相关版本更新较快; 同时 Derr 等人[82]研究发现由于开发人员的粗心大意, 或进行版本升级的代价相对较高, 或无法衡量升级后的版本是否存在兼容性问题, 开发人员决定不更新第三方库版本. Bavota 等人[26]进行的定量和定性分析发现, 当项目升级依赖的第三方库版本发生更新时, 他们不会对所有库执行升级, 平均会升级 60% 的新可用版本. 项目的大小与项目升级频率之间没有显著的相关性. Hora 等人[40]研究了 Pharo 生态系统 (拥有大约 3600 个不同的系统) 的库修改升级问题, 发现开发人员通常不能快速对这些升级做出反应 (平均反应时间为 34 天), 往往需要较多的时间应对版本的更新. 对于不同的生态系统而言, 其更新周期也有所不同. McDonnell 等人[47]研究了 Android 框架中 API 的稳定性和演变引发的问题, 发现平均每月有 115 个 API 更新, 应用中有 28% 的 API 使用应该被更新, 但由于多种原因没有进行更新, 滞后中值为 16 个月.

3.1.3 第三方库使用情况

开发者对第三方库中 API 的利用率不均衡. 研究表明复用第三方库的主要目的是使用第三方库中的 API, 因此第三方库 API 的使用是软件开发中不可分割的一部分. 开发项目中存在大量 API 的使用, 平均 3 行代码就会引用一些 API[49]. 相对于开发框架这种特殊类型的第三方库, Wu 等人[48]发现对于一些常用的框架, 其 API 的平均使用率在 35%. 即使第三方库中功能使用率较低, 在多数情况下, 使用第三方库的 API, 运行效率会远高于开发人员手动编写代码[50]. Qiu 等人[33]对 Java 中的 API 使用情况进行调查研究, 发现在使用第三方库的过程中, 一些核心库的 API 利用率不高, 而对核心库的使用需要加载其全部 API, 增加了程序运行过程的资源浪费.

由于开发人对 API 的使用规约[51]了解甚少, 开发人员通常会使用一些已经被声明为将被弃用的类或方法, 并且常会误用 API. API 误用导致的依赖冲突问题存在多种情况: 例如多余的 API 调用、遗漏的 API 调用、错误的 API 调用参数、缺少前置条件判断、忽略异常处理等[83]. Patra 等人[16]发现 1/4 的第三方库存在潜在的冲突, 超过 17% 的第三方库出现过至少一次依赖冲突问题, 这些冲突会导致系统崩溃和其他意外事件. Cai 等人[52]研究了 62894 个 Android 应用程序的依赖冲突问题, 分析了这些问题的表现形式和原因, 研究表明在 Android 应用程序中, 依赖冲突问题在安装时和运行时均可能发生: 安装时冲突问题主要由于 SDK 版本问题, 运行时的冲突问题主要与底层 API 相关.

3.1.4 第三方库变更对使用者的影响

第三方库版本的变更会给库的使用者带来巨大影响, 很容易造成依赖冲突问题, 开发人员需要对项目代码进行重构来适配库版本的变更. Kula 等人[53]实现了一个可视化工具, 利用从 Maven 存储库中提取的依赖项统计数据来调查库更新的历史. 即使库中内容的变更不是破坏性更改, 开发项目也会受到影响; 依据变更程度, 其影响范围亦有所不同. 其中最突出的影响表现为, 当项目代码引用第三方库中的某一类的类型或方法发生变化时[23,26], 本地代码需要进行重构. 版本变更的情况之一是版本升级, 第三方库的版本升级中, 类和方法的变更比其他变更发生得更为频繁, 因此对项目代码的影响更大. Wu 等人[48]分析了 Apache 和 Eclipse 生态系统第三方库升级中 API 的变更情况, 同样发现与其他更改类型相比, 第三方库缺少类和方法的情况更加常见, 对项目代码的影响也更大.

© 中国科学院软件研究所 http://www.jos.org.cn

为了探究第三方库版本是否发生变更, 研究人员从不同角度分别提出了: 识别第三方库中 API 发生变化的有关方法、对被声明为弃用的 API 寻找替代方案、检测 API 的修改是否影响了开发人员的调用. Meng 等人[54]提出一种基于历史的匹配方法, 来识别和理解 Java 框架的 API 演变. Xing 等人[55]认识到框架中 API 的变化, 并提出了弃用 API 的合理替代方案. Jezek 等人[56]对 109 个 Java 开源程序和 564 个程序版本研究发现, API 是不稳定的, 因为不兼容的库更改是常见的. Raemaekers 等人[84]对 Maven Central 中的第三方库进行研究发现, 接近 1/3 的库都发生了不兼容的更改. Wu 等人[48]以及 Bogart 等人[46]在研究中发现当使用的第三方库中的 API 发生变化时, 大约 11% 的变更会在项目代码中引起连锁反应. Li 等人[58]研究了 Web 服务 API 的演变及其对项目代码的影响. Sawant 等人[59]提供了关于推荐 API 使用的研究. Mezzetti 等人[60]提出了类型回归测试, 以确定库更新是否影响其公共接口. 有研究表明开发人员在使用有关库前, 会对其特性进行了解. Wei 等人[61]发现, Android 开发人员经常在调用 API 之前检查 API 更改情况, 以避免兼容性问题. Dietrich 等人[62]研究了从集成构建到部分库升级是否会给 Java 程序带来新的错误类别, 并阐述了二进制和源代码之间兼容性差别, 详细介绍了基于 OSGi 的系统中的二进制兼容性问题. Xavier 等人[23]发现, 14.78% 的 API 更改破坏了与以前版本的兼容性, 并且库升级中破坏性更改 (或称之为 breaking change) 发生的频率随着时间推移而增加; 破坏性更改发生频率越高的系统, 规模越大、越受欢迎、越活跃. 这种破坏性更改, 极易造成依赖冲突问题的出现. Zhang 等人[63]也发现 Python 开发人员在使用 API 时会检查相应的库版本, 以避免依赖冲突问题. 即使开发人员有意识地检查有关库以避免依赖冲突问题, 当开发人员对升级后的第三方库测试不充分时, 容易带来依赖冲突、安全威胁、资源泄露等问题[73,85]. Cox 等人[57]从开发人员的角度揭示了, 任何开发人员都不太可能为每一个依赖的第三方库进行详细的测试. 并且, Xavier 等人[23]发现当一个第三方库拥有更多的维护者时, 其更容易包含一些导致依赖冲突问题的危险元素. 第三方库中 API 的数量、维护人员提交数量以及发布数量都会对依赖冲突问题的产生具有一定影响, 但这些影响的因素较小.

3.2 第三方库开发者视角分析

3.2.1 第三方库升级原因

对于不同的语言和生态系统, 第三方库的升级是常见的, 升级原因主要包括实现新需求、提升可维护性、偿还技术债. Wittern 等人[28]发现, 高度活跃的开发社区, 库与库之间的依赖关系飞速增长. 第三方库的使用周期随着时间的变化可以利用高阶模型进行建模[66]. Bagherzadeh 等人[67]对 2005 年 4 月–2014 年 12 月, Linux 系统依赖的第三方库的 8 770 次修改进行了实证研究, 研究每次库更改的大小、手动识别更改的类型和所做的错误修复工作. 研究发现内核级的系统调用的改变总是由于技术债, 并且调用 API 的数量增长是缓慢的. Brito 等人[68]调研了 59 个第三方库所作的不兼容的修改, 并要求开发人员解释他们决定改变 API 背后的原因, 发现原因是实现新的需求、提升可维护性等.

3.2.2 第三方库升级风格分析

不同语言、不同生态系统的第三方库开发者对库的修改风格是不同的, 但大多数都会考虑到代码兼容性问题, 防止依赖冲突问题的出现. Dig 等人[69,70]发现, 现有应用程序 80% 以上的不兼容性更改是第三方库代码重构造成的. Wu 等人[48]分析了 Apache 和 Eclipse 生态系统, 发现第三方库中出现 API 弃用的情况较少. 但 Wang 等人[71]对 Python 有关第三方库进行研究发现, API 弃用问题较为常见, 并且约有 1/4 的更改没有被明显标记, 但通常在升级后的版本会对被弃用的 API 提供替代方案. Li 等人[72]调查了 Android 框架源代码, 发现有一类常见的 API, 既不向前兼容, 也不向后兼容. Bogart 等人的研究表明[46], 不同语言与生态系统具有不同的价值系统影响库的更改, 例如, Eclipse 更倾向于稳定性, 保持向后兼容的升级, R/CRAN 社区更倾向于协作交流, 而 Node.js/npm 生态系统更倾向于采用不兼容的更改来对抗技术债, 同时也更容易出现版本的迭代变更, 版本间改动幅度更大. Li 等人[73]发现, Android 社区中对 API 的弃用会有平滑的过渡期, 以帮助开发人员更好适应改变, 但其他社区的库开发者通常不会遵循此建议; API 被弃用的情况更倾向于发生在较为流行的库中. Rivières 等人[74]研究了 Java 语言中第三方库中 API 的更改, 并讨论了从开发人员角度如何在保证 API 更改的同时, 保持与现有开发人员编写的代码的兼容性. Lamothe 等人[75]发现在升级和迁移第三方库的过程中, Android 生态系统的使用文档相较于任何其他语言或生态系统都提供了更加丰富的相关 API 的迁移信息. Wang 等人[21]研究表明, 第三方库开发人员通常不遵循语义版

本控制, 这极易引入依赖冲突问题.

3.3 小 结

从本节对第三方库的使用实证分析可以发现, 依赖冲突问题在多数情况下, 是由于使用者对第三方库的了解不足、使用不当造成的; 而第三方库的开发缺少一致性的编写与管理规则, 导致很多第三方库代码的可读性、易理解性和变更兼容性较差, 增加了依赖冲突发生的概率. 依赖冲突问题出现在第三方库的使用过程中, 通过对第三方库的使用情况进行汇总分析, 本文对依赖冲突问题的表现形式有了较为全面的了解. 在此基础上, 第 4 节对依赖冲突问题的类别和原因, 进行总结分析.

4 第三方库依赖冲突原因及分析

为了回答问题 3, 本文通过调研大量有关文献, 研究流行语言的依赖冲突表现形式, 例如 JS、Python、Java 这 3 种语言, 构建了通用的依赖冲突模型. 不同语言对应的第三方库管理工具, 涉及不同的库加载机制, 导致依赖冲突问题的出现模式有所不同, 但可以对问题抽象简化构建依赖冲突模型. 为了简化表示, 我们使用图 4 进行总结.

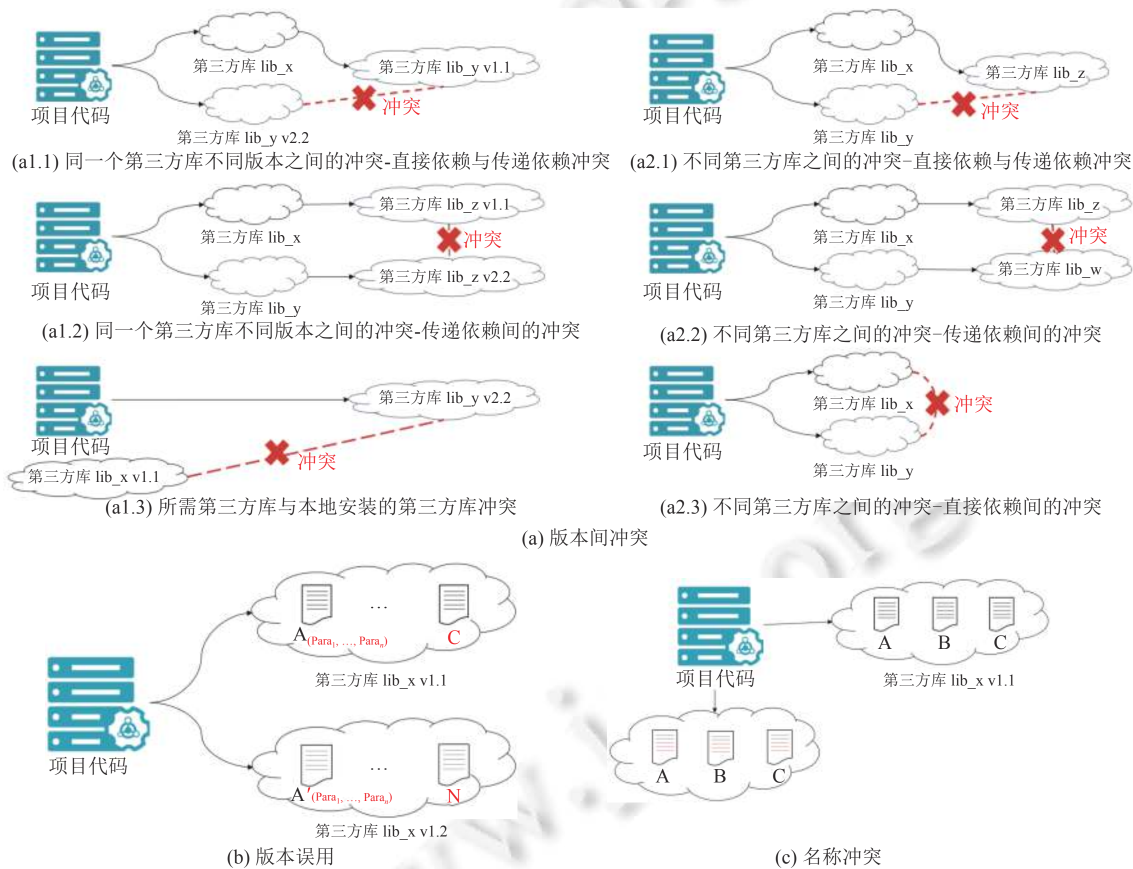


图 4 软件系统的通用依赖冲突模型

将出现在不同语言中的依赖冲突问题原因划分为 3 类: 版本冲突、版本误用以及名称冲突. 我们对这 3 类依赖冲突问题, 可能出现的阶段, 进行了调研和划分. 3 类依赖冲突问题的分类角度, 思路源于文献 [2], 但不仅局限于其总结的在 Java 语言中出现的相关问题, 而是对包括 GitHub 使用语言排行榜中的前 3 种使用最为广泛的语言

JavaScript、Java 和 Python 的问题模式和修复方式, 参考了文献 [1,17,19–21,23] 等多篇参考文献进行了研究总结; 同时, 文献 [2] 阐述的 3 种依赖冲突模式, 均可以被本文分类模式中的子模式涵盖, 通过参考相关文献, 同时考虑冲突发生的阶段本文将依赖冲突原因划分为 3 大类, 10 种细分类别, 相比之下更加全面具体, 研究角度与文献 [2] 也有所不同. 划分后的具体依赖冲突原因模式及对应的发生阶段如表 2 所示.

表 2　依赖冲突原因模式划分

类型	原因	形式	阶段	相关文献
版本冲突	同一个第三方库不同版本之间的冲突	直接依赖与传递依赖冲突 传递依赖间的冲突 所需第三方库与本地安装的第三方库冲突	第三方库下载安装、项目编译、项目运行阶段	[1,2,19−21]
	不同第三方库之间的冲突	直接依赖与传递依赖冲突 传递依赖间的冲突 直接依赖间的冲突	第三方库下载安装、项目编译、项目运行阶段	
版本误用	类的修改	增加类/删除类	项目编译、项目运行阶段	[17,20,23,28,46,57,62,63,69,86]
	方法的修改	增加方法/删除方法		
	方法内部的修改	方法增加参数/减少参数/参数类型改变/方法内部进行重构		
名称冲突	项目代码和第三方库冲突	项目代码和第三方库之间的名称冲突	项目编译、项目运行阶段	[1,2]

4.1　类别 1: 版本冲突

此种依赖冲突表现形式为版本间的依赖冲突. 其典型示例如前文图 4(a) 中所示, 包含同一个第三方库之间的冲突、不同第三方库之间的冲突[2,19,21]. 每种冲突具体表现形式如下.

第 1 种形式为同一个第三方库的不同版本间冲突, 可能出现在第三方库安装、项目编译和项目运行阶段. 可以具体划分为直接依赖与传递依赖之间的冲突、传递依赖之间的冲突、所需第三方库与本地安装的第三方库之间的冲突, 分别如图 4(a1.1)、(a1.2)、(a1.3) 所示, 其具体形式如下.

(1) 同一第三方库之间的直接依赖与传递依赖冲突如图 4(a1.1) 所示: 项目代码中依赖于第三方库 lib_x 和 lib_y, 由于 lib_x 传递依赖于第三方库 lib_y v1.1 版本, 项目代码直接依赖于 lib_y v2.2 版本, 因此 lib_x 和 lib_y 之间存在同一个第三方库 lib_y 的直接依赖与传递依赖冲突, 二者在同一个项目中进行安装、编译和运行时会报错.

(2) 同一第三方库之间的传递依赖冲突如图 4(a1.2) 所示: 项目代码中依赖于第三方库 lib_x 和 lib_y, 由于 lib_x 传递依赖于第三方库 lib_z v1.1 版本、lib_y 传递依赖于第三方库 lib_z v2.2 版本, 因此 lib_x 和 lib_y 之间存在同一个第三方库 lib_z 的传递依赖冲突, 二者在同一个项目中进行安装、编译和运行时会报错.

(3) 所需第三方库与本地安装第三方库间的冲突如图 4(a1.3) 所示: 项目所需要的第三方库为 lib_y v2.2 版本, 而本地已加载了 lib_x v1.1 版本, 且 lib_y v2.2 版本与 lib_x v1.1 版本存在不同特性, 因此同一环境下再加载 lib_y v2.2 版本就会报错, 或者在已加载的 lib_x v1.1 版本中使用 lib_y v2.2 版本的特性就会报错. 此种类型经常出现在项目编译和运行阶段.

第 2 种形式为不同第三方库之间的冲突, 可能出现在第三方库下载安装、项目编译和项目运行阶段, 可以具体划分为直接依赖间的冲突、直接依赖与传递依赖之间的冲突、传递依赖之间的冲突, 分别如图 4(a2.1)、(a2.2)、(a2.3) 所示, 其具体形式如下.

(1) 不同第三方库之间的直接依赖与传递依赖冲突如图 4(a2.1) 所示: 项目代码依赖于第三方库 lib_x 和 lib_y, lib_x 传递依赖于第三方库 lib_z, lib_y 和 lib_z 中存在冲突, 导致 lib_x 或 lib_y 的安装, 或在项目的编译运行阶段出现错误.

(2) 不同第三方库之间的传递依赖间的冲突如图 4(a2.2) 所示: 项目代码依赖于第三方库 lib_x 和 lib_y, lib_x 传递依赖于第三方库 lib_z, lib_y 传递依赖于第三方库 lib_w, 由于 lib_z 和 lib_w 之间存在冲突, 因此 lib_x 或

© 中国科学院软件研究所 http://www.jos.org.cn

lib_y 在无法在同一个项目中安装或在项目的编译运行阶段出现错误.

(3) 不同第三方库之间的直接依赖冲突如图 4(a2.3) 所示: 项目代码依赖于第三方库 lib_x 和 lib_y, lib_x 和 lib_y 中存在冲突, 在安装或运行时, 会产生错误, 例如 lib_x 中定义了方法 M, lib_y 中对方法 M 进行了重写, lib_x 和 lib_y 的方法 M 不一致, 当调用方法 M 时, 可能产生依赖冲突问题. 此种类型经常出现在项目编译和运行阶段.

4.2 类别 2: 版本误用

此种依赖冲突表现形式为对第三方库版本的误用导致的依赖冲突, 会出现在项目编译、运行阶段. 主要表现形式为: 同一第三方库的多个版本共同存在于项目配置的路径中, 但是加载机制决定了只能有一个第三方库的版本可以被加载到路径中. 这导致了期望使用的版本与实际使用的版本不一致[2,20,63,69]. 依赖冲突模式如图 4(b) 所示. 项目代码在开发过程中协商使用第三方库 lib_x v1.1 版本, 但由于多种原因, 实际开发过程中, 加载的版本为 lib_x v1.2, lib_x v1.2 版本兼容 lib_x v1.1 版本但是由于 lib_x v1.2 中语义发生了变化, 使用 lib_x v1.2 可能造成意想不到的问题. 这种变化是难以被察觉的. 由于开发人员通常不会对版本进行强制约束, 并且多达一半的开发人员习惯于自动依赖最新版本[28], 增加了成此种类别问题的出现. 程序开发过程中, 会对代码进行升级维护, 在升级维护过程中, 对依赖的第三方库版本进行修改是极其常见的, 这也可能对程序造成巨大的危害, 因为修改后的版本中开发人员期望使用的 API 已经发生了变化, 这需要对代码进行重构来适应这种改变. 然而这种改变难以察觉, 开发人员对依赖的第三方库了解不够或对代码测试不充分, 难以发觉此类问题. 此种模式是版本误用的一种特殊表现形式, Dig 等人[69]发现 80% 以上的依赖冲突问题中, 导致程序无法正确运行是由于版本变更中的代码重构. 根据 Cossette 等人的实证分析[86], 版本升级的大部分更改都没有记录在相关的发布说明中, 也没有在记录在源代码注释中, 这给开发人员发现问题带来了巨大挑战.

版本变更带来的具体的变化可根据不同层次进行进一步划分, 如图 5 所示: 自顶向下依次分类, 按照类、方法、方法内部的变化进行划分. 对类的修改包括: 增加类, 删除类, 对类内的方法进行重构. 进一步, 对类内的方法进行重构又可以划分为增加方法, 删除方法, 对方法进行修改. 对方法的修改可以进一步划分为方法的签名是否发生改变: 方法签名没有发生更改, 但方法内部语义发生了变化. 修改方法签名可以进一步划分为: 增加参数, 删除参数, 对参数传递类型或参数的默认值进行了改变[15,23,46,48,62,63,69].

在新版本中与上一版本相比, 如果增加了类, 可能会导致兼容性问题, 因为方法的增加可能添加了对其他方法的调用, 潜在地更改了开发人员期望使用方法的有关值, 但是此种问题是极其少见的.

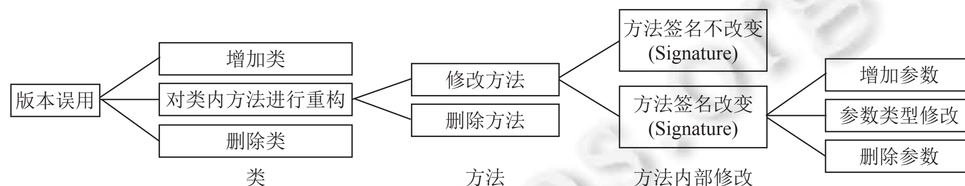


图 5 依赖冲突模式 2-根据修改内容的层级进一步划分类型

不改变方法签名的方法, 最难以察觉[18]. 运行过程中很少会直接出现报错, 只是得到的结果不符合开发人员预期, 但可以顺利通过编译过程或可以正常运行. 举例而言, 开发人员对第三方库的两个版本间细微差别没有注意到时, 直接使用版本 lib_x v1.2 后, 执行结果可能发生了改变, 如图 6 所示的在真实环境中出现的问题[87]. 开发人员期望引用 commons-httpclient:commons-httpclient:jar:3.1 中的 available() 方法, 其可以对 IOException 做多种处理, 但实际加载的包为 commons-httpclient:commons-httpclient:jar:3.0.1, 其仅对 IOException 进行返回值为 0 的处理. 当 closed 状态为 False 时, 其返回值已经发生了变化. 当开发人员传入的 this.closed 状态为 True 时, 两个第三方库不会有任何差异, 但是状态为 False 时, 其结果具有明显差异, 这种差异难以捕捉. 上述情况为第 2 种冲突模式.

© 中国科学院软件研究所 http://www.jos.org.cn

```
//MiA-mahout-0.5 需要使用 commons-httpclient:commons-httpclient:jar
public ProgressMonitorInputStream(Component parentComponent, Object message,  InputStream in){
    super(in);
    try{
        size = in.available(); //期望调用 available 方法
    }catch (IOException IOE){
        size = 0;
    }
    monitor = new ProgressMonitor(parentComponent,  message,  null,  0,  size);
}
//实际加载的包 commons-httpclient:jar: 3.0.1
public int available() throws IOException{
    return 0;
}
//期望加载的包 commons-httpclient:jar: 3.1
public int available() throws IOException{
    if(this.closed){
        return 0;
    }
    int avail = this.wrappedStream.available();
    if(this.pos + avail> this.contentLength){
        avail = (int)(this.contentLength  -  this.pos);
    }
    return avai;
}
```

图 6　方法签名不发生变化, 但内部语义发生了变化的实例

4.3 类别 3: 名称冲突

此种模式如图 4(c) 所示, 项目代码和第三方库之间的命名冲突, 会出现在项目编译、运行阶段. 表现形式为, 第三方库代码与本地代码使用的有关方法以及类的命名一致且加载路径一致[2]. 如果项目代码和第三方库 lib_x 包含重复的类 A、B 和 C, 那么在运行过程中只会加载那些包含在第三方库 lib_x 中的类. 如果此时开发人员期望引用仅在项目的类 A、B 或 C 中定义的特性, 则系统编译过程中会抛出异常或错误. 此错误的出现是由于各种语言加载机制在执行过程中, 优先加载环境配置路径, 因此优先引用第三方库中定义的类, 当类中没有开发人员期望使用的方法时, 就会出现错误.

此类问题中较为特殊的一个表现形式是, 项目代码与第三方库中定义了重复的类, 但是自定义的类的路径与第三方库路径不相同. 当开发人员调用项目代码时报错, 因为项目代码对第三方库部分内容进行了重载, 导致无法找到开发人员实际期望加载的内容. 例如在 STORM-2382 问题[88]中, 最新的 Storm 使用的日志系统已经从 Log4j 切换到了 slf4j+logback, slf4j 提供了桥接工具 log4j-over-slf4j, 提供与 Log4j 完全相同的类名和接口, 但底层是 slf4j 的实现; 在依赖了 Storm 的工程中使用 Log4j, 比如 org.apache.log4j.Logger, 实际找到的是 log4j-over-slf4j 中的完全同名的类, 导致了运行时失败.

4.4 小　结

依赖冲突发生的主要原因是第三方库之间的依赖关系没有被满足、需要的空间等资源被抢占. 现有针对依赖冲突原因分析主要关注在某一特定语言或生态系统, 例如 Arth 等人[1]关注于 Debian 和 RedHat 中的依赖冲突问题, 将原因划分为资源访问上的冲突、配置数据上的冲突, 以及不常见包组合之间的交互等; Wang 等人[19]关注于 PyPI 生态系统中的依赖冲突, 将原因划分为远程依赖更新引起的冲突以及受本地环境影响的冲突. 现有研究分析原因的角度不同, 每种依赖冲突原因的分析集中在特定场景上, 本节对有关研究进行了归纳, 得到依赖冲突原因总结.

5 第三方库依赖冲突问题检测方法

针对问题 3, 本节主要针对 3 种主流语言中依赖冲突问题的检测工具有关文献进行总结. 按照检测目标和解

© 中国科学院软件研究所 http://www.jos.org.cn

决的依赖冲突原因类别, 将检测方法划分为 3 类: 针对版本冲突的检测方法、针对版本误用的检测方法、针对名称冲突的检测方法. 最后对现有工作进行总结对比, 并给出一些新发现.

5.1 针对第三方库版本冲突的检测方法

这类方法主要对依赖冲突问题进行检测, 并尝试对依赖冲突问题生成崩溃信息的堆栈跟踪.

EVOSUITE[89,90]是一个面向 Java 语言的开源工具, 采用遗传算法 (GA)[91]为给定的目标类生成测试用例集. 在依赖冲突问题中, 常见的初始化个体由第三方库中, 被调用方法的参数组合构成; 通过自定义的适应度函数以及设计的交叉运算和变异函数, 加速搜索过程, 指导第三方库被调用方法的测试用例的生成, 提高对第三方库方法间冲突检测的有效性和效率. Wang 等人[17]提出的 RIDDLE 基于静态分析, 构造控制流依赖关系, 对依赖关系生成变体进行测试, 这种变体的生成采用了 EVOSUITE 的基于搜索的策略, 改进了 EVOSUITE 的覆盖标准, 以触发由依赖冲突问题引起的故障. 生成的测试集, 用于收集依赖冲突问题导致崩溃的堆栈跟踪信息, 以促进依赖冲突问题的诊断.

Patra 等人[16]提出的 ConflictJS 针对的是 JavaScript 语言中, 由于第三方库共享相同的全局命名空间导致的依赖冲突问题. 这类问题的出现, 是由于不同库对同一命名空间进行写操作. ConflictJS 首先动态地分析单个第三方库写入全局命名空间的位置, 基于库的全局写入位置进行匹配. ConflictJS 假设写入相同位置的库会导致依赖冲突问题, 为了验证此种假设, 进一步比较库中方法的行为, 当且仅当该方法发现有不同行为时, 报告依赖冲突问题.

Wang 等人[19]提出的 Watchman 是一个持续监测 PyPI 生态系统依赖冲突的工具. 通过收集每个第三方库的 metadata (元数据) 并进行广度优先搜索, 为项目依赖的第三方库构建完整的依赖调用关系图. 依赖关系图的实际呈现形式为有向无环图, 图中的节点代表第三方库, 节点间的有向边代表着两个库之间的依赖关系, 有向边箭头指向的节点代表被依赖的第三方库. 对依赖关系图中有多个有向边所指向的节点, 即为被多个库同时依赖的第三方库; 分析其传入的边的集合, 对边之间约束的版本关系进行匹配分析, 若传入边之间约束的交集为空, 证明存在依赖冲突题.

Huang 等人[92]通过分析项目代码中各个模块的第三方库版本, 检测项目中的依赖冲突问题. 遍历所有模块所依赖的库版本, 对每一个库进行版本匹配, 分析这些库的版本是否一致, 如果不是, 那么检测到了库版本不一致的问题, 并报告其所影响的模块. 根据匹配结果, 按照最少改动的原则, 进行版本的推荐.

Mancinelli 等人[10,93]提出的 EDOS 是基于形式化方法, 对第三方库中的依赖冲突关系进行解析, 将第三方库的可安装性问题转化为约束求解问题, 并设计为自动化的检测工具. 从发行 (基于 Linux 发行版-rpm 和 deb) 第三方库的角度, 降低库之间冲突出现的频率, 提高了发布的第三方库的稳定性.

5.2 针对第三方库版本误用的检测方法

5.2.1 直接相关方法

这类检测方法通过对比项目代码的行为 (如调用关系等), 识别与调用关系不一致的行为. 这一类检测方法中, 总体是基于匹配的思想, 对比不同版本第三方库的运行结果, 或构建项目方法的依赖调用图或抽象语法树, 进行调用关系的对比匹配进行依赖冲突问题的检测.

Foo 等人[94]提出的 JTEXPERT 是一种自动化软件测试数据生成方法, 它实现了单元类测试的高代码覆盖率. 其核心思想是基于遗传算法, 通过类实例生成器和种子策略来优化搜索, 并通过静态分析方法从搜索空间中删除不相关的输入变量加速搜索过程. 其生成的测试数据可用于检测不同版本的第三方库和调用关系不一致的行为.

语义冲突问题是版本误用的一种常见表现形式, 具体表现为, 开发人员期望加载的版本和实际加载的版本中, 使用的方法具有相同的签名, 但是两者程序运行行为不一致, 即为第 3 节中依赖冲突问题的第 2 种表现形式. Wang 等人[18]提出的 SENSOR 使用遗传算法来检测 API 之间的语义冲突. SENSOR 首先从源代码中提取每个对象的构造函数和 API 调用的上下文, 并利用它们来构造类实例池和 API 参数池; 基于 GUMTREE[95]迭代检测得到的代码差异, 在细粒度级别上识别异构的冲突 API 对. SENSOR 认为异构的冲突 API 对可能会导致语义冲突. SENSOR 方法将类实例调用的种子策略与 EVOSUITE[89]结合起来, 生成测试用例集来触发相关的库 API, 并检查

© 中国科学院软件研究所 http://www.jos.org.cn

它们在不同版本中的行为是否一致.

Ghorbani 等人[20]提出的 DARCY 针对 Java-9 开发的应用, 设计检测依赖冲突问题的工具. DARCY 将实际加载与声明的依赖进行匹配对比, 发现不一致的情况, 并报告给开发人员. 根据开发人员编写的代码分析出其使用的第三方库的方法, 与不一致的方法进行对比, 若某版本满足开发人员需求, 则将此版本写入依赖声明文件中.

PYCOMPAT[63]运用静态分析的方法, 检测由第三方库中 API 更改引起的依赖冲突问题; 具体来说, 是为了检测由 API 重命名和参数重命名引起的问题. 其检测分为两个阶段: 第 1 阶段抽取第三方库中的 API 更改信息, 当 API 的更改信息无法进行自动化抽取时, 采用人工抽取; 对抽取出的信息, 构建知识库. 第 2 阶段将 API 知识库作为输入, 并对给定的 Python 源文件执行静态分析, 构建抽象语法树 (AST), 遍历 AST 以获得框架中定义的 API 的调用, 进而通过定义的匹配规则来检查对 API 的调用, 是否使用了改进的 API, 定位潜在的依赖冲突问题.

Foo 等人[96]运用静态分析方法来检查第三方库在升级后, 是否引入了不兼容库中 API 的方法. 其首先构建程序的库调用图, 利用 Myers 算法提取升级前后库中 API 的差异. 对于版本的更改, 可能直接跳跃多个版本. 算法计算每个相近版本间 API 的差异, 进而得到最终的差异结果, 并结合依赖调用图来判断是否发生了不兼容的更改. 通过解析配置文件, 对改动最少的版本进行推荐, 通过匹配名称, 对配置文件中对应依赖信息的版本, 进行修改.

Wang 等人[2]提出的 Decca 通过分析库依赖管理脚本 (如 pom.xml、build.gradle) 提取库依赖树, 根据依赖树和字节码 (JAR 或类文件) 识别重复的库和类, 最后将实际加载的第三方库中的全部方法、期望加载的第三方库中的全部方法, 与开发人员实际调用的方法进行对比匹配, 得到方法的交集. 识别依赖冲突问题的同时, 根据交集的不同情况评定危险等级, 为开发人员提供建议.

5.2.2 间接相关方法

第三方库版本误用主要是由于第三方库升级更新过程中, API 兼容性问题所导致的. 现有解决第三方库版本误用的研究工作, 主要是针对版本更新不及时和库迁移升级的识别问题. 本文中介绍的第三方库 API 兼容性问题的相关研究工作, 可以为检测依赖冲突问题提供参考性建议. 具体而言, 对第三方库的升级进行建模, 可以帮助开发人员快速了解升级前后两个版本之间的差异, 进而修改自己的代码. 当开发人员面临第三方库版本误用导致的依赖冲突问题时, 可以使用 APIwave[97]与 HiMa[54]快速发现新版本中的 API 来修改本地项目代码, 减少版本误用的情况发生; 或者通过静态分析, 首先快速识别开发人员前后使用的第三方库版本, 使用 NoRegrets[60]、NoRegrets+[98]、Ponomarenko 等人[99]和 Lamothe 等人[75]提出的算法或工具, 对比两个版本的第三方库中 API 是否兼容, 提示不兼容的 API 信息, 帮助开发人员快速修复代码, 避免依赖冲突问题中版本误用情况的出现, 同时可以避免在长时间运行后, 程序由于此 bug 意外崩溃, 导致大量的计算资源、时间和人力等资源的浪费.

较为成熟的几种解决方法如下.

Meng 等人[54]提出的 HiMa 基于历史的匹配方法来识别和理解 Java 框架的 API 演变.

Hora 等人[97]提出的 APIwave 是一个基于匹配规则, 识别跟踪 API 流行度和迁移的大型工具. 与 HiMa 的不同点在于, APIwave 的使用范围更广, 可以跟踪 API 的流行程度. APIwave 由两部分构成, 第 1 部分是预处理阶段, 其接收存储在源代码库 (例如, GitHub 项目) 中的项目历史作为输入, 输出 API 数据库, 其中包括关于 API 流行度和迁移的信息, 以及一个关于此类 API 的源代码示例的数据库; 第 2 部分将获取到的两个数据库提供视图给开发人员. 其匹配过程体现在对一个源代码文件的两个版本之间的比较. APIwave 与 HiMa 对第三方库的对比, 可以帮助开发人员快速了解两个版本之间的差异, 进而修改自己的代码. 当开发人员面临第三方库版本误用导致的依赖冲突问题时, 可以使用此工具快速发现新版本中产生变更的 API, 来修改本地项目代码, 减少版本误用的情况发生.

Ponomarenko 等人[99]提出了一种在二进制级别的、可适用于多种语言的、自动检测第三方库的向后兼容性问题的新方法. 此方法除了分析组件二进制文件中的符号外, 还可以通过比较从组件头文件中获得的函数签名和类型定义, 来验证向后兼容性问题. 当开发人员将使用的第三方库的版本更新后, 可能出现版本误用导致的依赖冲突问题. 此工具经过改进, 可以应用于依赖冲突问题的检测上, 具体而言, 通过静态分析, 快速识别开发人员使用的两个版本的第三方库中的 API 是否兼容. 这样可以避免在长时间运行后, 程序由于此 bug 意外崩溃, 导致大量的计算资源、时间和人力等资源的浪费.

© 中国科学院软件研究所 http://www.jos.org.cn

NoRegrets[60]是一个回归测试工具, 可以用来确定第三方库更新后是否影响了有关 API 的使用, 并对更新前后 API 的兼容性进行对比. NoRegrets+[98]通过分析第三方库被其他第三方库复用的情况, 自动判断第三方库的更改是否影响公共 API 的使用, 并自动生成测试用例, 来查找库中的破坏性更改.

Dependabot[100]是 GitHub 上的第三方库的依赖管理工具, 集成了 GitHub 的漏洞数据库[101], 自动下载本地配置的依赖文件、解析并检查是否存在任何弃用的或不安全的依赖关系. 在将所有的第三方库依赖更新到新版本时, 其无法保证不会引入依赖冲突问题, 但是其会对第三方库之间依赖关系进行分析.

Lamothe 等人[75]基于文档和历史代码更改信息, 对 Android 中 API 的迁移升级方法进行检测. 首先提取每个 Android API 的所有 JavaDoc 代码注释. 对每个代码注释, 使用 API 名称作为关键字, 自动搜索注释中是否提到了已更改 (添加、删除或弃用) 的 API 的名称; 同时提取 git 存储库中每两个连续版本之间的所有代码提交及其提交消息. 最后查询官方 Android API 文档中, 每个版本应用的编程接口列表的修改记录. 结合 3 种注释信息, 对 API 迁移升级提供建议. 此种方法可以对 51%–98.4% 的删除或弃用的 API 提供升级的意见.

5.3 针对第三方库名称冲突的检测方法

目前, 据收集到的资料, 尚未出现第 3 种依赖冲突类别 (项目代码与第三方库之间的名称冲突) 的检测方法, 其可能的原因是开发人员在开发过程中, 很容易发现此类问题, 并且更重要的是此类问题出现频率较低, 远不及前两类问题出现的频率. 此类问题的解决方式的工具设计可以考虑, 将第三方库的名称与本地方法路径进行匹配, 匹配命名一致的方法, 视为冲突.

5.4 总　结

表 3 对所有依赖冲突问题有关的检测方法, 根据所检测依赖冲突问题的原因类别, 从 3 大主流语言的角度, 进行了分类; 对各类别的检测方法, 进行了较为全面的总结, 包括方法提出的时间、方法检测的对象, 并将各检测方法对应到其检测对象所属的依赖冲突原因的细分子类. 通过总结和对比, 可以发现以下结论.

表 3　检测方法总结

类别	语言	相关文献	年份	检测对象
针对版本冲突的检测方法	JavaScript	Patra等人[16]	2018	检测由于共享相同命名空间导致的依赖冲突问题
	Java	Wang等人[17]	2019	为有依赖冲突问题的项目生成测试用例以及造成崩溃堆栈跟踪信息
		Huang等人[92]	2020	检测项目中第三方库版本不一致问题, 同时推荐并量化统一库版本的维护代价
	Python	Wang等[19]	2020	监测PyPI生态系统中依赖冲突问题
	Linux	Mancinelli等人[93]	2006	对Linux中发行的第三方库的依赖冲突问题进行形式化表达求解, 提高发行版的稳定性
针对版本误用的检测方法	Java	Meng等人[54]	2012	基于历史信息, 识别第三方库中API变化
		Hora等人[97]	2015	基于构建的API数据集, 识别第三方库中API变化
		Wang等人[2] (直接)	2018	对开发人员使用的方法与库进行匹配, 并提供警示信息
		Ghorbani等人[20] (直接)	2019	针对Java-9开发的应用, 将实际加载与声明的第三方库中方法对比, 报告依赖冲突问题
		Wang等人[18] (直接)	2021	检测程序语义问题, 生成测试用例
	JavaScript	Mezzetti等人[60]	2018	第三方库更新是否影响程序, 并生成测试用例
	Python	Zhang等人[63] (直接)	2020	检测由API重命名和参数重命名引起的依赖冲突问题
	Android社区	Lamothe等人[75]	2018	基于文档和历史代码更改信息, 对Android中API进行迁移升级
	多个生态系统	Foo等人[96] (直接)	2018	检查Maven-Central, PyPI, RubyGems生态系统中升级的版本中是否引入了兼容性问题
	多种语言	Ponomarenko等人[99]	2012	二进制库版本向后兼容的问题
		Dependabot[100]	2019	检查弃用或不安全的依赖关系

(1) 早期对依赖冲突问题的检测关注较少, 从检测项目中是否存在依赖冲突问题, 到生成测试用例触发依赖冲突问题, 到目前可以对有关配置文件进行修复意见的提出, 其发展随着开发人员的需求不断变化, 更加符合开发人员的真实需求; 并且检测方法的丰富程度与相关问题出现的频率密切相关.

(2) 从检测阶段对依赖冲突问题的检测方法进行考量, 有关研究包括: 运行前的依赖冲突问题检测; 运行时的依赖冲突问题检测; 为依赖冲突问题生成测试用例或展示触发问题出现的执行路径.

(3) 针对版本冲突的检测方法相对较少, 并且当前解决方法尚不能解决传递依赖导致的依赖冲突问题. 根据检测方法提出的时间, 从早期只能检测是否存在依赖冲突问题, 到为有依赖冲突风险的项目生成测试并收集崩溃的堆栈跟踪, 到生成测试用例发现语义冲突问题, 再升级到对生态系统有关冲突进行监测, 并检查期待安装的第三方库与项目配置文件中的有关库是否存在冲突. 检测方法的用途逐渐精细化, 更加符合程序员的需求.

(4) 针对版本误用的检测方法丰富多样, 其目标也更加丰富, 包括为依赖冲突问题生成测试用例, 对比框架的 API 演变、检测对兼容性问题的同时提出风险提示、对配置文件进行修改、对第三方库的修改进行建模.

(5) 从语言角度看, 目前的检测方法主要面向 Java 语言, 其可能原因是, 依赖冲突问题常见于大型系统中, 而当前大型系统常用 Java 语言编写. 大型系统的特点是由多模块构成, 模块间通过接口交互, 模块内部反复迭代. 正是由于上述特点, 大型系统不同模块间的依赖关系复杂, 模块内部不断迭代, 第三方库的复杂性不断增加, 容易引入依赖冲突问题.

除了上述针对 3 种主流语言的依赖冲突检测方法, 还有其他面向不同生态的依赖冲突问题检测方法: FicFinder[102]和 CiD[103]检测在 Android 上误用特定平台或频繁进化的 API 引起的潜在兼容性问题; MUTAPI[104]通过突变分析发现 API 误用的模式.

总体来说尽管目前对依赖冲突问题的重视程度在不断提高, 但对于 Python 以及 JavaScript 这些语言的检测方法较少, 而且已有方法的实用性与理论性还有待提升. 当前工具多集中在检测项目中是否存在依赖冲突问题, 并不支持提供更精细的追踪信息, 很多情况下不能给开发人员提示如何进行修复.

6 第三方库依赖冲突问题修复方式

针对问题 4, 本文对有关文献中提及的开发人员修复依赖冲突问题的方式进行总结. 通过总结发现, 现有研究工作中, 支持自动修复或提示修复方式的有关工作非常少. 本节结合已发表的研究工作和软件工程的开发实践, 对依赖冲突问题的修复方式进行总结. 如表 4 所示.

表 4　实际开发过程中维修人员修复策略

目标	类别	修复方式	总结	相关文献
针对版本冲突的修复策略	环境隔离	第三方库本地安装版本与需要安装的版本不一致, 可以创建隔离环境	是解决版本冲突的可行解决方案, 是开发人员经常采用, 简单高效的解决方案	[19]
	修改依赖的第三方库版本	调整本地项目直接依赖的版本约束 与上游或下游项目协调, 调整冲突的第三方库版本约束 去除冲突的直接依赖, 保留传递依赖	通过对依赖的版本进行修改, 可以快速解决有关依赖冲突问题, 此种解决方式需要开发人员对有关版本有详细的了解. 实际开发过程中, 开发人员经常采用此策略	[2,18,19,21,63]
	修改第三方库的加载路径	调整依赖项的路径加载顺序	采用较少, 多数情况下, 通过另外3种类别中的修复方式即可满足开发人员有关需求	[2,12,18]
针对版本误用以及名称冲突的修复策略	对项目代码进行修改	对本地代码进行修改以适应库版本修改后的特性 打印警告信息	项目对第三方库的相关版本具有较强的约束性, 因此可以通过修改项目代码完成特定需求	[18,48,63]

即使不同语言具有各自的特性和库管理机制, 但解决依赖冲突问题的方式具有较多共同之处, 主要参考文献 [2,19] 中涉及的修复策略, 结合其他相关参考文献, 按照本文提出的依赖冲突类别, 对修复方式进行抽象总结.

© 中国科学院软件研究所 http://www.jos.org.cn

同时根据采用的修复方式, 分为 4 大类: 环境隔离、修改依赖的第三方库版本、修改第三方库的加载路径、对项目代码进行修改, 如表 4 所示. 前 3 类解决方式针对版本冲突问题, 最后一类针对版本误用以及名称冲突问题, 前两类对本地代码基本没有改动, 也是开发人员最常采用的解决方式. 第 3 类对本地代码修改较少, 但现实世界中开发人员对此解决方式采用较少. 最后一类对代码改动最多, 开发人员采用较少.

(1) 类别 1: 环境隔离

开发过程中, 会涉及日常开发环境、测试环境、预发布环境和线上环境, 不同项目的应用环境也不尽相同, 多种语言的库管理工具均提供了环境隔离机制. 环境隔离就是将开发中的环境分隔开, 以便于保障所开发模块的运行. 针对不同语言提出的库管理工具, 环境隔离机制的表现形式相似, 为当前项目创建独立的运行环境, 但具体实现中存在一定的差异.

JavaScript 常见的库管理工具有 npm、yarn. 其环境隔离机制为项目加载特定第三方库, 创建独立运行环境时, 可以直接使用本地的 js 版本, 不需要另外下载 js 的其他版本.

Python 常见的库管理工具有 Conda, Anaconda (其内置了 Conda) 和 pip. 前两者与 Python 环境管理工具 virtualenv, 均可以创建独立的环境 (Anaconda 由于内置了 Conda, 无特殊提示, 下文统一使用 Conda 表示). Conda 和 pip 可进行第三方库的管理. 最常见的管理工具为 Conda, 开发人员为每一个项目自定义环境, 加载需要的第三方库, 并安装指定版本的 Python. Conda 为了保证环境隔离的纯洁性, 对每一个环境都会下载指定的 Python 版本. 开发人员根据需要下载第三方库, 不同环境隔离使用, 不产生交互.

Java 常见的第三方库管理工具有 Maven、Gradle、Ant. Maven 和 Gradle 用于环境管理, 其中最常见的管理工具为 Maven. Maven 根据开发人员的声明, 将项目代码依赖的第三方库下载到统一位置, 其中某些库的其他版本可能已经存在于本地中, Maven 仍旧执行下载过程. 但 Conda 与之相反, 为每一个环境单独下载依赖的第三方库. 使用时, Maven 的环境隔离体现为, 每个项目直接按需提取所依赖的第三方库.

环境隔离为每一个项目创建独立的开发环境, 是解决直接依赖冲突、直接依赖与传递依赖冲突、传递依赖冲突的可行方案[19], 也是开发人员经常采用的、简单高效的第三方库管理办法.

(2) 类别 2: 修改依赖的第三方库版本

修改依赖的第三方库版本是开发人员经常采用的修复策略[2,18,21,63], 其根据调整的位置不同, 可以划分为 3 类: 调整本地依赖配置中直接依赖版本的约束、调整远程依赖的配置信息、根据项目需求调整上下游的依赖配置. 调整本地依赖配置中直接依赖版本的约束包括修改版本约束和删除本地直接依赖冲突版本、保留传递依赖版本[19].

此种修复方式通常针对版本间冲突问题, 通过对依赖的版本进行修改, 可以快速解决有关依赖冲突问题, 但需要开发人员对有关版本具备详细的了解. 实际开发过程中, 开发人员经常采用此策略.

(3) 类别 3: 修改第三方库的加载路径

此修复方式常见于需要在同一项目的相同模块、使用同一个第三方库的多个不同版本这种特殊场景. 此种方式需要对依赖的第三方库的加载路径进行调整[2,18], 以满足开发人员需求. 此种方式较为少见, 因为大多数情况下, 通过上述 3 种方式即可满足开发人员有关需求.

(4) 类别 4: 对项目代码进行修改

针对版本依赖问题, 根据所使用第三方库版本的特殊特性, 当上述提及的修复方式一和修复方式二均无法满足开发需求时, 开发人员需要对本地代码进行修改, 以使用特定版本的第三方库特性[18,48]. 打印警告信息[63]也是一种对项目代码进行修改来解决依赖冲突问题的方式, 这种方式一般为下游任务提出警示.

对项目代码进行修改这种修复方式往往对有关版本具有较强的约束性, 因此可以通过修改项目代码完成特定需求.

7 问题与展望

在代码中添加第三方库进行代码复用, 是减少开发负担、提高开发效率的常用方式. 但是第三方库之间的依

赖调用关系复杂, 不正确的调用会引入第三方库的依赖冲突问题. 目前, 依赖冲突问题仍是一个开放性研究问题. 本文针对该主题进行了系统的综述, 以便研究人员更好地了解这一主题的最新研究进展. 本文在分析过程中回答了引言中提出的 4 个有关问题.

本文首先对目前依赖冲突的相关研究工作中, 第三方库实证分析内容进行总结整理, 从库的使用者视角和库开发者视角进行了较为全面的分析. 分析发现第三方库之间依赖调用关系错综复杂, 开发人员不正确调用会引入第三方库依赖冲突问题. 问题发生的根本原因是第三方库间的不兼容. 本文第 2 步是对开发过程中依赖冲突问题发生的原因进行了系统、全面的研究, 将其划分为 3 类: 版本冲突, 版本误用以及名称冲突; 对每种冲突模式的具体类别进行了详细阐述. 随后根据依赖冲突模式, 对现有依赖冲突检测方法进行划分, 从不同维度总结和对比现有检测工具. 最后对实际开发过程中, 开发人员对依赖冲突问题的修复方式, 进行总结分类; 根据依赖冲突原因以及对本地项目的修改程度, 将修复方式分为: 环境隔离, 修改依赖的第三方库版本, 对本地代码进行修改, 修改加载路径.

依赖冲突问题引起了广泛关注, 自动检测该问题也取得了一系列高质量的研究成果, 但该问题仍有很多值得进一步关注的研究点, 具体包括以下内容.

(1) 构建一种更加普适性的依赖冲突问题检测方法. 这种普适性是指针对某种语言或者生态系统同时对版本冲突以及版本误用问题进行检测. 本文发现已有工作中, 对依赖冲突问题检测的相关研究较少, 随着越来越多的研究人员关注到此问题, 部分检测方法对依赖冲突问题特定数据集的检测准确率, 可以达到 0.898. 但更加普适的数据集上的表现效果尚未可知. 因此需要一种更加普适性的检测方法.

(2) 建立统一的衡量标准. 通过分析已有工作, 本文发现大部分工作在对检测结果进行评估时, 采用了手工评估的方法. 但这种评估方法较为主观. 不同文章的评估指标也不尽相同. 因此如何寻找更加客观的评测指标仍是一个开放性问题.

(3) 深度学习技术火热发展, 被广泛应用于各个领域中, 学术界以及工业界提出了有关基于深度学习技术进行程序分析的方法和技术, 有关研究证明了将深度学习应用于程序分析技术的可行性与有效性[105−109]. 依赖冲突检测, 属于程序分析的相关研究领域. 但目前基于深度学习的依赖冲突问题检测, 仍是一项空白. 结合本文总结的第三方库依赖冲突问题, 考虑使用基于深度学习的方式, 对依赖冲突问题进行预判. 例如, 搜集开源社区或生态系统中出现的依赖冲突问题, 运用深度学习的方法, 对错误提示提供详细信息乃至是修复方式. 如将定位错误所涉及的有关的代码、方法, 错误的原因和解决方案等, 进行分析, 提取关键信息; 进而对各依赖冲突模式中, 不同的错误类型进行分类, 挖掘出每个类别的共性特征. 基于分析和挖掘结果, 构建第三方库间的代码关联分析模型, 根据学习到的特征, 检测出不同第三方库之间, 潜在的可能出现依赖冲突的部分, 并识别错误原因和所属类型, 提供相应的解决方案. 同样可以应用在依赖冲突发生之后, 根据错误信息、定位的代码和方法, 分析得到错误原因和所属类型, 给出相应的解决方案, 为依赖冲突的修复提供参考, 提高错误修复的效率.

(4) 除了基于深度学习的依赖冲突问题检测方法, 知识图谱可以为依赖冲突问题的检测提供额外信息, 辅助检测. 例如基于开源社区和生态系统上第三方库间依赖调用关系, 构建知识图谱作为全面的辅助知识, 对依赖冲突研究具有重要价值. 开源社区上的第三方库的依赖调用关系极其复杂, 若想了解第三方库的依赖关系, 只能通过下载到本地进行解析获取 metadata 中声明的依赖信息, 或阅读其在开源平台上发布的代码中的有关依赖配置信息. 如果事先构造有关开源的第三方库的依赖调用关系的知识图谱, 可以有效辅助用户对依赖关系的获取.

(5) 同时, 在已有研究工作中, Horton 等人[110]构建知识图谱, 从代码中推断依赖的第三方库及其版本, 证明了使用知识图谱解决有关第三方库版本推断的可行性与有效性. 因此未来可以尝试基于开源代码中依赖调用关系, 构建知识图谱, 结合本地项目中第三方库的依赖调用关系, 进行依赖关系的匹配对比, 识别潜在的依赖冲突关系.

(6) 此外, 知识图谱的高质量、连续化表示结构, 和高有效性的、全面的领域信息, 可以为深度学习模型提供高质量、高适配的数据来源, 进而提高模型的训练效率和学习效果. 因此, 可以考虑将依赖冲突知识图谱与依赖冲突深度学习模型相结合, 依赖冲突知识图谱的不断丰富, 可以为基于深度学习的依赖冲突分析模型, 提供重要的先验知识, 辅助深度学习模型的训练, 提高模型的学习效率和表现性能.

© 中国科学院软件研究所 http://www.jos.org.cn

(7) API 兼容性问题是导致依赖冲突的重要原因之一, 当前针对第三方库中 API 的兼容性有关工作虽然无法直接用于依赖冲突问题检测, 需要做相当工作量的改进才可应用, 但可以为依赖冲突问题的检测与修复提供重要的参考价值.

(8) 针对依赖冲突题的检测方法是针对特定语言进行的, 对研究工作有效性的评价数据集, 是单独构建的, 对检测效果的评估也是在各自构建的数据集上进行的, 缺少统一的评价数据集. 因此构建针对不同语言的、统一的评价数据集, 对于促进有关研究是十分有必要的.

(9) 提出更多、更有效的修复依赖冲突问题的方法. 目前只有少数工作对依赖冲突问题的修复进行了研究, 多数工作集中在依赖冲突的检测上, 依赖冲突问题修复的研究, 仍存在较多欠缺之处.

本文希望针对上述挑战可以提出更多更有效的解决方案, 可以使开发人员在依赖冲突问题的检测与修复问题上节约大量的时间, 同时自动为项目代码的维护提供可靠保障.

References:

[1] Artho C, Suzaki K, Di Cosmo R, Treinen R, Zacchiroli S. Why do software packages conflict? In: Proc. of the 9th IEEE Working Conf. on Mining Software Repositories (MSR). Zurich: IEEE, 2012. 141–150. [doi: 10.1109/MSR.2012.6224274]

[2] Wang Y, Wen M, Liu ZW, Wu RX, Wang R, Yang B, Hai Y, 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]

[3] Vasilakis N, Karel B, Roessler N, Dautenhahn N, DeHon A, Smith JM. BreakApp: Automated, flexible application compartmentalization. In: Proc. of the 25th Annual Network and Distributed System Security Symp. San Diego, 2018. [doi: 10.14722/NDSS.2018.23131]

[4] Kula RG, Ouni A, German DM, Inoue K. On the impact of micro-packages: An empirical study of the npm JavaScript ecosystem. arXiv:1709.04638, 2017.

[5] Liang S, Bracha G. Dynamic class loading in the Java virtual machine. ACM SIGPLAN Notices, 1998, 33(10): 36–44. [doi: 10.1145/286942.286945]

[6] Maven™. 2021. http://maven.apache.org/

[7] Ikkink HK. Gradle Dependency Management. Packt Publishing, 2015.

[8] Varanasi B, Belida S. Maven dependency management. In: Varanasi B, Belida S, eds. Introducing Maven. Berkeley: Apress, 2014. 15–22.

[9] Resolving package dependencies with the new version of pip. 2021. https://podcasts.apple.com/gh/podcast/resolving-package-dependencies-with-the-new-version-of-pip/id1501905538?i=1000493348880

[10] Di Cosmo R. EDOS deliverable WP2-D2.1: Report on formal management of software dependencies. Technical Report, HAL-Inria, 2006. https://hal.inria.fr/hal-00697468/

[11] 10 most active programming languages in GitHub. 2015. https://learnworthy.net/10-most-active-programming-languages-in-github/

[12] Coblenz M, Nelson W, Aldrich J, Myers B, Sunshine J. Glacier: Transitive class immutability for Java. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Buenos Aires: IEEE, 2017. 496–506. [doi: 10.1109/ICSE.2017.52]

[13] Baidu. Dependency hell. 2021 (in Chinese). https://baike.baidu.com/item/%E7%9B%B8%E4%BE%9D%E6%80%A7%E5%9C%B0%E7%8B%B1/8368103

[14] Issue #13923 of project keras. 2021. https://github.com/keras-team/keras/issues/13923

[15] Jia ZY, Li SS, Yu TT, Zeng C, Xu EC, Liu XD, Wang J, Liao XK. DepOwl: Detecting dependency bugs to prevent compatibility failures. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 86–98. [doi: 10.1109/ICSE43902.2021.00021]

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

[17] 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]

[18] Wang Y, Wu RX, Wang C, Wen M, Liu YP, Cheung SC, Hai Y, 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]

© 中国科学院软件研究所 http://www.jos.org.cn

[19] Wang Y, Wen M, Liu YP, Wang YB, Li ZM, Wang C, Hai Y, 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: ACM, 2020. 125–135. [doi: 10.1145/3377811.3380426]

[20] Ghorbani N, Garcia J, Malek S. Detection and repair of architectural inconsistencies in Java. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 560–571. [doi: 10.1109/ICSE.2019.00067]

[21] Wang Y, Chen BH, Huang KF, Shi BW, Xu CY, Peng X, Wu YJ, Liu Y. An empirical study of usages, updates and risks of third-party libraries in Java projects. In: Proc. of the 2020 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Adelaide: IEEE, 2020. 35–45. [doi: 10.1109/ICSME46990.2020.00014]

[22] Abdalkareem R, Nourry O, Wehaibi S, Mujahid S, Shihab E. Why do developers use trivial packages? An empirical case study on npm. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. Paderborn: ACM, 2017. 385–395. [doi: 10.1145/3106237.3106267]

[23] Xavier L, Brito A, Hora A, Valente MT. Historical and impact analysis of API breaking changes: A large-scale study. In: Proc. of the 24th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Klagenfurt: IEEE, 2017. 138–147. [doi: 10.1109/SANER.2017.7884616]

[24] Dietrich J, Pearce D, Stringer J, Tahir A, Blincoe K. Dependency versioning in the wild. In: Proc. of the 16th IEEE/ACM Int'l Conf. on Mining Software Repositories (MSR). Montreal: IEEE, 2019. 349–359. [doi: 10.1109/MSR.2019.00061]

[25] Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S. The evolution of project inter-dependencies in a software ecosystem: The case of Apache. In: Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance. Eindhoven: IEEE, 2013. 280–289. [doi: 10.1109/ICSM.2013.39]

[26] Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S. How the Apache community upgrades dependencies: An evolutionary study. Empirical Software Engineering, 2015, 20(5): 1275–1317. [doi: 10.1007/s10664-014-9325-9]

[27] Zimmermann M, Staicu CA, Tenny C, Pradel M. Smallworld with high risks: A study of security threats in the npm ecosystem. In: Proc. of the 28th USENIX Security Symp. Santa Clara: USENIX Association, 2019. 995–1010.

[28] Wittern E, Suter P, Rajagopalan S. A look at the dynamics of the JavaScript package ecosystem. In: Proc. of the 13th Int'l Conf. on Mining Software Repositories. Austin: ACM, 2016. 351–361. [doi: 10.1145/2901739.2901743]

[29] Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Di Penta M, Oliveto R, Poshyvanyk D. API change and fault proneness: A threat to the success of Android Apps. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering. Saint Petersburg: ACM, 2013. 477–487. [doi: 10.1145/2491411.2491428]

[30] Reif M, Eichberg M, Hermann B, Lerch J, Mezini M. Call graph construction for Java libraries. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 474–486. [doi: 10.1145/2950290.2950312]

[31] Backes M, Bugiel S, Derr E. Reliable third-party library detection in Android and its security applications. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. Vienna: ACM, 2016. 356–367. [doi: 10.1145/2976749.2978333]

[32] Li MH, Wang W, Wang P, Wang S, Wu DH, Liu J, Xue R, Huo W. LibD: Scalable and precise third-party library detection in Android markets. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Buenos Aires: IEEE, 2017. 335–346. [doi: 10.1109/ICSE.2017.38]

[33] Qiu D, Li BX, Leung H. Understanding the API usage in Java. Information and Software Technology, 2016, 73: 81–100. [doi: 10.1016/j.infsof.2016.01.011]

[34] Zhang YH, Chen YF, Cheung SC, Xiong YF, Zhang L. An empirical study on TensorFlow program bugs. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 129–140. [doi: 10.1145/3213846.3213866]

[35] Wu W, Guéhéneuc YG, Antoniol G, Kim M. AURA: A hybrid approach to identify framework evolution. In: Proc. of the 32nd Int'l Conf. on Software Engineering. Cape Town: IEEE, 2010. 325–334. [doi: 10.1145/1806799.1806848]

[36] Decan A, Mens T, Grosjean P. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering, 2019, 24(1): 381–416. [doi: 10.1007/s10664-017-9589-y]

[37] Decan A, Mens T, Constantinou E. On the evolution of technical lag in the npm package dependency network. In: Proc. of the 2018 IEEE Int'l Conf. on Software Maintenance and Evolution. Madrid: IEEE, 2018. 404–414. [doi: 10.1109/ICSME.2018.00050]

[38] Decan A, Mens T, Constantinou E. On the impact of security vulnerabilities in the npm package dependency network. In: Proc. of the 15th Int'l Conf. on Mining Software Repositories. Gothenburg: ACM, 2018. 181–191. [doi: 10.1145/3196398.3196401]

[39] Cox J, Bouwers E, van Eekelen M, Visser J. Measuring dependency freshness in software systems. In: Proc. of the 37th Int'l Conf. on Software Engineering. Florence: IEEE, 2015. 109–118. [doi: 10.1109/ICSE.2015.140]

[40] Hora A, Robbes R, Anquetil N, Etien A, Ducasse S, Valente MT. How do developers react to API evolution? The pharo ecosystem case.

© 中国科学院软件研究所 http://www.jos.org.cn

In: Proc. of the 2015 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Bremen: IEEE, 2015. 251–260. [doi: 10.1109/ICSM.2015.7332471]
[41] Kula RG, German DM, Ouni A, Ishio T, Inoue K. Do developers update their library dependencies? Empirical Software Engineering, 2018, 23(1): 384–417. [doi: 10.1007/s10664-017-9521-5]
[42] Kula RG, German DM, Ishio T, Inoue K. Trusting a library: A study of the latency to adopt the latest Maven release. In: Proc. of the 22nd Int'l Conf. on Software Analysis, Evolution, and Reengineering. Montreal: IEEE, 2015. 520−524. [doi: 10.1109/SANER.2015.7081869]
[43] Salza P, Palomba F, Di Nucci D, D'Uva C, De Lucia A, Ferrucci F. Do developers update third-party libraries in mobile APPs? In: Proc. of the 2018 Int'l Conf. on Program Comprehension (ICPC). Gothenburg: IEEE, 2018. 255–265.
[44] Zerouali A, Constantinou E, Mens T, Robles G, González-Barahona J. An empirical analysis of technical lag in npm package dependencies. In: Proc. of the 17th Int'l Conf. on Software Reuse. Madrid: Springer, 2018. 95–110. [doi: 10.1007/978-3-319-90421-4_6]
[45] Trockman A, Zhou SR, Kästner C, Vasilescu B. Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Gothenburg: IEEE, 2018. 511−522. [doi: 10.1145/3180155.3180209]
[46] Bogart C, Kästner C, Herbsleb J, Thung F. How to break an API: Cost negotiation and community values in three software ecosystems. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 109–120. [doi: 10.1145/2950290.2950325]
[47] McDonnell T, Ray B, Kim M. An empirical study of API stability and adoption in the Android ecosystem. In: Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance. Eindhoven: IEEE, 2013. 70–79. [doi: 10.1109/ICSM.2013.18]
[48] Wu W, Khomh F, Adams B, Guéhéneuc YG, Antoniol G. An exploratory study of API changes and usages based on Apache and Eclipse ecosystems. Empirical Software Engineering, 2016, 21(6): 2366–2412. [doi: 10.1007/s10664-015-9411-7]
[49] Lämmel R, Pek E, Starek J. Large-scale, AST-based API-usage analysis of open-source Java projects. In: Proc. of the 2011 ACM Symp. on Applied Computing. Taichung: ACM, 2011. 1317–1324. [doi: 10.1145/1982185.1982471]
[50] Tao YD, Tang S, Liu YP, Xu ZW, Qin SC. How do API selections affect the runtime performance of data analytics tasks? In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). San Diego: IEEE, 2019. 665–668. [doi: 10.1109/ASE.2019.00067]
[51] Zhong H, Zhang L, Mei H. Mining invocation specifications for API libraries. Ruan Jian Xue Bao/Journal of Software, 2011, 22(3): 408−416 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3931.htm [doi: 10.3724/SP.J.1001.2011.03931]
[52] Cai HP, Zhang ZY, Li L, Fu XQ. A large-scale study of application incompatibilities in Android. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 216–227. [doi: 10.1145/3293882.3330564]
[53] Kula RG, De Roover C, German D, Ishio T, Inoue K. Visualizing the evolution of systems and their library dependencies. In: Proc. of the 2nd IEEE Working Conf. on Software Visualization. Victoria: IEEE, 2014. 127–136. [doi: 10.1109/VISSOFT.2014.29]
[54] Meng SC, Wang XY, Zhang L, Mei H. A history-based matching approach to identification of framework evolution. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). Zurich: IEEE, 2012. 353–363. [doi: 10.1109/ICSE.2012.6227179]
[55] Xing ZC, Stroulia E. API-evolution support with Diff-CatchUp. IEEE Trans. on Software Engineering, 2007, 33(12): 818–836. [doi: 10.1109/TSE.2007.70747]
[56] Jezek K, Dietrich J, Brada P. How Java APIs break—An empirical study. Information and Software Technology, 2015, 65: 129–146. [doi: 10.1016/j.infsof.2015.02.014]
[57] Cox R. Surviving software dependencies. Communications of the ACM, 2019, 62(9): 36–43. [doi: 10.1145/3347446]
[58] Li J, Xiong YF, Liu XZ, Zhang L. How does Web service API evolution affect clients? In: Proc. of the 20th IEEE Int'l Conf. on Web Services. Santa Clara: IEEE, 2013. 300–307. [doi: 10.1109/ICWS.2013.48]
[59] Sawant AA, Bacchelli A. A dataset for API usage. In: Proc. of the 12th IEEE/ACM Working Conf. on Mining Software Repositories. Florence: IEEE, 2015. 506–509. [doi: 10.1109/MSR.2015.75]
[60] Mezzetti G, Møller A, Torp MT. Type regression testing to detect breaking changes in Node.js libraries. In: Proc. of the 32nd European Conf. on Object-oriented Programming (ECOOP 2018). Dagstuhl: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. 7:1–7:24.
[61] Wei LL, Liu YP, Cheung SC, Huang HX, Lu X, Liu XZ. Understanding and detecting fragmentation-induced compatibility issues for Android APPs. IEEE Trans. on Software Engineering, 2020, 46(11): 1176–1199. [doi: 10.1109/TSE.2018.2876439]
[62] Dietrich J, Jezek K, Brada P. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In: Proc. of the 2014 Software Evolution Week-IEEE Conf. on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). Antwerp: IEEE, 2014. 64–73. [doi: 10.1109/CSMR-WCRE.2014.6747226]

© 中国科学院软件研究所 http://www.jos.org.cn

[63] Zhang ZX, Zhu HC, Wen M, Tao YD, Liu YP, Xiong YF. How do Python framework APIs evolve? An exploratory study. In: Proc. of the 27th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). London: IEEE, 2020. 81–92. [doi: 10.1109/SANER48275.2020.9054800]

[64] Plate H, Ponta SE, Sabetta A. Impact assessment for vulnerabilities in open-source software libraries. In: Proc. of the 2015 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Bremen: IEEE, 2015. 411–420. [doi: 10.1109/ICSM.2015.7332492]

[65] Ponta SE, Plate H, Sabetta A. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In: Proc. of the 2018 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Madrid: IEEE, 2018. 449–460. [doi: 10.1109/ICSME.2018.00054]

[66] Kula RG, German DM, Ishio T, Ouni A, Inoue K. An exploratory study on library aging by monitoring client usage in a software ecosystem. In: Proc. of the 24th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Klagenfurt: IEEE, 2017. 407–411. [doi: 10.1109/SANER.2017.7884643]

[67] Bagherzadeh M, Kahani N, Bezemer CP, Hassan AE, Dingel J, Cordy JR. Analyzing a decade of Linux system calls. Empirical Software Engineering, 2018, 23(3): 1519–1551. [doi: 10.1007/s10664-017-9551-z]

[68] Brito A, Xavier L, Hora A, Valente MT. Why and how Java developers break APIs. In: Proc. of the 25th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Campobasso: IEEE, 2018. 255–265. [doi: 10.1109/SANER.2018.8330214]

[69] Dig D, Johnson R. How do APIs evolve? A story of refactoring. Journal of Software Maintenance and Evolution: Research and Practice, 2006, 18(2): 83–107. [doi: 10.1002/smr.328]

[70] Dig D, Johnson R. The role of refactorings in API evolution. In: Proc. of the 21st IEEE Int'l Conf. on Software Maintenance (ICSM 2005). Budapest: IEEE, 2005. 389–398. [doi: 10.1109/ICSM.2005.90]

[71] Wang JW, Li L, Liu K, Cai HP. Exploring how deprecated Python library APIs are (not) handled. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. New York: ACM, 2020. 233–244. [doi: 10.1145/3368089.3409735]

[72] Li L, Bissyandé TF, Le Traon Y, Klein J. Accessing inaccessible Android APIs: An empirical study. In: Proc. of the 2016 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Raleigh: IEEE, 2016. 411–422. [doi: 10.1109/ICSME.2016.35]

[73] Li L, Gao J, Bissyandé TF, Ma L, Xia X, Klein J. Characterising deprecated Android APIs. In: Proc. of the 15th Int'l Conf. on Mining Software Repositories. Gothenburg: ACM, 2018. 254–264. [doi: 10.1145/3196398.3196419]

[74] des Rivières J. Evolving Java-based APIs. 2007. http://wiki.eclipse.org/Evolving_Java-based_APIs

[75] Lamothe M, Shang WY. Exploring the use of automated API migrating techniques in practice: An experience report on Android. In: Proc. of the 15th IEEE/ACM Int'l Conf. on Mining Software Repositories. Gothenburg: IEEE, 2018. 503–514.

[76] Black duck by synopsys. 2021. https://www.blackducksoftware.com

[77] Veracode. 2021. https://www.veracode.com

[78] OWASP dependency-check. 2021. https://owasp.org/www-project-dependency-check/

[79] OWASP top 10 home page. 2014. https://www.owasp.org/index.php/Top_10_2013-Table_of_Contents

[80] Pfretzschner B, ben Othmane L. Identification of dependency-based attacks on Node.js. In: Proc. of the 12th Int'l Conf. on Availability, Reliability and Security. Reggio: ACM, 2017. 68. [doi: 10.1145/3098954.3120928]

[81] Cadariu M, Bouwers E, Visser J, van Deursen A. Tracking known security vulnerabilities in proprietary software systems. In: Proc. of the 22nd IEEE Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER). Montreal: IEEE, 2015. 516–519. [doi: 10.1109/SANER.2015.7081868]

[82] Derr E, Bugiel S, Fahl S, Acar Y, Backes M. Keep me updated: An empirical study of third-party library updatability on Android. In: Proc. of the 2017 ACM Conf. on Computer and Communications Security. Dallas: ACM, 2017. 2187–2200. [doi: 10.1145/3133956.3134059]

[83] Wang X, Chen C, Zhao YF, Peng X, Zhao WY. API misuse bug detection based on deep learning. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5): 1342–1358 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5722.htm [doi: 10.13328/j.cnki.jos.005722]

[84] Raemaekers S, van Deursen A, Visser J. Semantic versioning versus breaking changes: A study of the Maven repository. In: Proc. of the 14th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation. Victoria: IEEE, 2014. 215–224. [doi: 10.1109/SCAM.2014.30]

[85] Jin WX, Cai YF, Kazman R, Zhang G, Zheng QH, Liu T. Exploring the architectural impact of possible dependencies in Python software. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Melbourne: IEEE, 2020. 758–770.

[86] Cossette BE, Walker RJ. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In: Proc. of

the 20th ACM SIGSOFT Int'l Symp. on the Foundations of Software Engineering. Cary North: ACM, 2012. 55. [doi: 10.1145/2393596.2393661]

[87] Issue #11 of project mia. 2021. https://github.com/tdunning/MiA/issues/11

[88] STORM-2382. 2021. https://issues.apache.org/jira/browse/STORM-2382

[89] Fraser G, Arcuri A. EvoSuite: Automatic test suite generation for object-oriented software. In: Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of Software Engineering. Szeged: ACM, 2011. 416–419. [doi: 10.1145/2025113.2025179]

[90] Panichella A, Campos J, Fraser G. EvoSuite at the SBST 2020 tool competition. In: Proc. of the 42nd IEEE/ACM Int'l Conf. on Software Engineering Workshops. Seoul: ACM, 2020. 549–552. [doi: 10.1145/3387940.3392266]

[91] Wikipedia contributors. Genetic algorithm. In Wikipedia, the free encyclopedia. 2021. https://en.wikipedia.org/w/index.php?title=Genetic_algorithm&oldid=1032041921

[92] 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. New York: ACM, 2020. 518–529. [doi: 10.1145/3368089.3409689]

[93] Mancinelli F, Boender J, di Cosmo R, Vouillon J, Durak B, Leroy X, Treinen R. Managing the complexity of large free and open source package-based software distributions. In: Proc. of the 21st IEEE/ACM Int'l Conf. on Automated Software Engineering. Tokyo: IEEE, 2006. 199–208. [doi: 10.1109/ASE.2006.49]

[94] Sakti A, Pesant G, Guéhéneuc YG. Instance generator and problem representation to improve object oriented code coverage. IEEE Trans. on Software Engineering, 2015, 41(3): 294–313. [doi: 10.1109/TSE.2014.2363479]

[95] Falleri JR, Morandat F, Blanc X, Martinez M, Monperrus M. Fine-grained and accurate source code differencing. In: Proc. of the 29th ACM/IEEE Int'l Conf. on Automated Software Engineering. Vasteras: ACM, 2014. 313–324. [doi: 10.1145/2642937.2642982]

[96] Foo D, Chua H, Yeo J, Ang MY, Sharma A. Efficient static checking of library updates. 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. 791–796. [doi: 10.1145/3236024.3275535]

[97] Hora A, Valente MT. Apiwave: Keeping track of API popularity and migration. In: Proc. of the 2015 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Bremen: IEEE, 2015. 321–323. [doi: 10.1109/ICSM.2015.7332478]

[98] Møller A, Torp MT. Model-based testing of breaking changes in Node.js libraries. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 409–419. [doi: 10.1145/3338906.3338940]

[99] Ponomarenko A, Rubanov V. Backward compatibility of software interfaces: Steps towards automatic verification. Programming and Computer Software, 2012, 38(5): 257–267. [doi: 10.1134/S0361768812050052]

[100] Dependabot. 2021. https://dependabot.com

[101] GitHub. GitHub advisory database. 2021. https://github.com/advisories

[102] Wei LL, Liu YP, Cheung SC. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android APPs. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: IEEE, 2016. 226–237.

[103] Li L, Bissyandé TF, Wang HY, Klein J. CiD: Automating the detection of API-related compatibility issues in Android APPs. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 153–163. [doi: 10.1145/3213846.3213857]

[104] Wen M, Liu YP, Wu RX, Xie X, Cheung SC, Su ZD. Exposing library API misuses via mutation analysis. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 866–877. [doi: 10.1109/ICSE.2019.00093]

[105] Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M. Automated vulnerability detection in source code using deep representation learning. In: Proc. of the 17th IEEE Int'l Conf. on Machine Learning and Applications (ICMLA). Orlando: IEEE, 2018. 757–762. [doi: 10.1109/ICMLA.2018.00120]

[106] Li Z, Zou DQ, Xu SH, Ou XY, Jin H, Wang SJ, Deng ZJ, Zhong YY. VulDeePecker: A deep learning-based system for vulnerability detection. In: Proc. of the 25th Annual Network and Distributed System Security Symp. San Diego, 2018.

[107] Hu X, Li G, Liu F, Jin Z. Program generation and code completion techniques based on deep learning: Literature review. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5): 1206–1223 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5717.htm [doi: 10.13328/j.cnki.jos.005717]

[108] Pradel M, Sen K. Deep learning to find bugs. Technical Report, TUD-CS-2017-0295, TU Darmstadt, Department of Computer Science, 2017.

[109] Cummins C, Fisches ZV, Ben-Nun T, Hoefler T, Leather H. ProGraML: Graph-based deep learning for program optimization and analysis. arXiv:2003.10536, 2020.

© 中国科学院软件研究所 http://www.jos.org.cn

[110] Horton E, Parnin C. DockerizeMe: Automatic inference of environment dependencies for Python code snippets. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 328–338. [doi: 10.1109/ICSE.2019.00047]

附中文参考文献:

[13] 百度百科. 相依性地狱. 2021. https://baike.baidu.com/item/%E7%9B%B8%E4%BE%9D%E6%80%A7%E5%9C%B0%E7%8B%B1/8368103

[51] 钟浩, 张路, 梅宏. 软件库调用规约挖掘. 软件学报, 2011, 22(3): 408–416. http://www.jos.org.cn/1000-9825/3931.htm [doi: 10.3724/SP.J.1001.2011.03931]

[83] 汪昕, 陈驰, 赵逸凡, 彭鑫, 赵文耘. 基于深度学习的API误用缺陷检测. 软件学报, 2019, 30(5): 1342–1358. http://www.jos.org.cn/1000-9825/5722.htm [doi: 10.13328/j.cnki.jos.005722]

[107] 胡星, 李戈, 刘芳, 金芝. 基于深度学习的程序生成与补全技术研究进展. 软件学报, 2019, 30(5): 1206–1223. http://www.jos.org.cn/1000-9825/5717.htm [doi: 10.13328/j.cnki.jos.005717]

李硕(1997－), 女, 博士生, CCF 学生会员, 主要研究领域为智能化软件工程.

田浩翔(1994－), 男, 博士生, CCF 学生会员, 主要研究领域为自动驾驶, 仿真测试, 数据挖掘.

刘杰(1982－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为大数据, 分布式系统, 软件工程.

叶丹(1971－), 女, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为网络分布式系统, 软件工程.

王帅(1982－), 男, 博士, 高级工程师, 主要研究领域为大数据分析处理, 人工智能, 系统集成.

www.jos.org.cn
www.jos.org.cn