计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
[doi: 10.15888/j.cnki.csa.009863] [CSTR: 32024.14.csa.009863]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

面向用户需求的 Python 库依赖冲突检测和解决方法①

王　文[1], 牟　令[1], 杨德超[1], 姜　凯[2], 贾欣宇[3], 周　宇[3]

[1](湖北能源集团新能源发展有限公司, 武汉 430070)
[2](南瑞继保电气有限公司, 南京 211102)
[3](南京航空航天大学 计算机科学与技术学院, 南京 211106)
通信作者: 王　文, E-mail: wang_wen1@ctg.com.cn

摘　要: 第三方库是软件开发中关键的可复用资源, 它们显著减轻开发人员重复实现常见功能的负担, 从而进一步提高开发效率. 然而, 这些库的迭代更新可能导致在 Python 项目中使用过时的版本, 进而引发依赖冲突问题从而导致项目构建失败, 而开发人员往往对此风险缺乏足够认识. 为了解决这个问题, 本文对 103 个 Python 开源项目的第三方库更新及依赖冲突问题进行了全面的定量分析. 研究不仅关注了第三方库的版本更新频率, 还从项目角度深入探讨依赖冲突问题的具体表现形式. 基于实证研究, 本文提出了一种面向用户需求的 Python 库依赖冲突检测和解决方法, 旨在通过解决依赖冲突问题来简化开发人员的决策过程. 实验结果表明该方法在检测和解决依赖冲突方面具有显著优势和实用性.

关键词: Python 库; 依赖更新; 依赖分析; 第三方库; 依赖冲突

引用格式: 王文,牟令,杨德超,姜凯,贾欣宇,周宇.面向用户需求的 Python 库依赖冲突检测和解决方法.计算机系统应用. http://www.c-s-a.org.cn/1003-3254/9863.html

User-centric Approach for Detecting and Resolving Python Library Dependency Conflict

WANG Wen[1], MOU Ling[1], YANG De-Chao[1], JIANG Kai[2], JIA Xin-Yu[3], ZHOU Yu[3]

[1](Hubei Energy Group Co. Ltd., Wuhan 430070, China)
[2](NR Electric Co. Ltd., Nanjing 211102, China)
[3](College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China)

Abstract: Third-party libraries are essential reusable resources in software development. They significantly reduce the burden of repeatedly implementing common functions, thus improving development efficiency. However, the iterative updates of these libraries can result in the use of outdated versions in Python projects, potentially causing dependency conflicts that lead to project build failures. Developers often lack sufficient awareness of this risk. To address this issue, a comprehensive quantitative analysis of 103 Python open-source projects regarding third-party library updates and dependency conflicts is conducted. This study focuses not only on the frequency of version updates but also examines the specific manifestations of dependency conflicts from a project perspective. Based on empirical research, a user demand-oriented method for detecting and resolving Python dependency conflicts is proposed, aimed at simplifying the decision-making process for developers by resolving dependency issues. Experimental results demonstrate that the proposed method offers significant advantages and practical utility in detecting and resolving dependency conflicts.

Key words: Python library; dependency update; dependency analysis; third-party library; dependency conflict

① 收稿时间: 2024-11-05; 修改时间: 2024-11-28, 2024-12-17; 采用时间: 2024-12-24; csa 在线出版时间: 2025-04-28

随着互联网和大数据技术的迅猛发展, Python 语言凭借其代码简洁性和易学性, 在国际范围内获得广泛认可和应用[1,2]. Python 生态系统拥有丰富的第三方库, 这些库能够轻松实现多样化的功能, 满足不同应用场景的需求. Python 在科学计算[3]、数据分析[4]、图像处理、人工智能等多领域均有显著的应用成果. 同时, Python 的 Web 框架[5]如 Django 和 Flask 也成为开发 Web 应用的首选之一, 其简洁性和高效性得到了业界的广泛认可. 第三方库[6]是软件开发过程中不可或缺的可复用资源, 项目开发者通常从 PyPI (Python package index) 等服务器端中央存储库获取所需的第三方库. 在 PyPI 上托管的第三方库动态生态系统中, 通常可以使用同一个库的多个版本, 并通过版本号来区分. 要使用特定的库版本, 开发人员必须同时指定库的名称和所需的版本. 利用针对 PyPI 的官方库管理工具 pip[7], 开发人员可以根据相关配置轻松地检索和安装预期版本. 然而, Python 项目及其依赖的第三方库是独立开发的, 这导致开发人员不得不手动编写版本约束, 但是缺乏适当的指导. 此外, 第三方库的快速增长要求开发人员花费大量精力来维护这些版本约束以跟进库的发展. 如果不能做到这一点, Python 库的依赖可能会发生冲突, 从而导致项目构建失败 (即依赖冲突问题[8]). 因此, 研究对于依赖冲突问题检测和修复方法变得非常重要.

在实际软件开发过程中, 依赖冲突问题会导致项目构建失败或者运行时出现错误, 严重的话甚至会导致出现安全漏洞, 给开发者和企业带来巨大的经济损失和时间成本消耗. 有效的依赖冲突检测和解决策略可显著提高开发效率, 减少维护成本, 并确保软件项目的顺利进行. 此外, 随着开源项目增多, 依赖冲突问题也对开源社区发展和健康产生了影响, 有效的解决方案可以促进开源项目的稳定性和可用性. 通过自动化的依赖冲突检测和解决, 开发者能快速识别和解决潜在的依赖问题, 从而节省调试和修复时间, 提高开发效率. 有效依赖管理可以减少因依赖问题导致的运行时错误和系统崩溃, 从而提高软件的稳定性和可靠性. 对于用户而言, 软件的稳定性和性能直接关乎使用体验. 通过检测和解决依赖冲突问题, 可以提供更流畅稳定的用户体验. 依赖冲突问题的及时检测和解决有助于避免运行时错误, 确保软件的质量和可靠性.

软件开发过程中, 第三方库集成已成为提高开发效率和质量的关键因素. 目前, 在 Python 库的依赖冲突检测和解决的相关研究中主要有两类方法: 一种是致力于在约束求解过程中检测和解决不同第三方库之间潜在依赖冲突[9–11]. 通过构建不同库和版本约束之间的依赖关系图, 然后通过使用 SMT 求解器进行库的可用版本分配. 另一种是通过为 PyPI 上的第三方库构建相应的知识图谱, 来为新 Python 项目构建运行时的环境, 从而进行依赖冲突问题的检测和解决[12–14]. 尽管如此, 第三方库的迭代更新和潜在的依赖冲突问题一直是开发者面临的挑战. 以往的研究[9]主要集中关注于检测给定 Python 项目中包含的依赖冲突, 而忽略了对于用户需求 (用户提供的与本地环境最接近的依赖库版本及 Python 版本) 的全面考虑, 这些研究未能充分理解开发者在实际开发过程中所面临的具体问题和挑战, 也未能根据用户实际需求来提供针对性解决方案.

为解决上述问题, 本文首先通过开展一项实证研究, 深入探索 103 个 Python 开源项目中的第三方库版本更新情况以及依赖冲突问题. 本研究不仅量化了第三方库的版本更新频率, 还从项目角度出发详细分析了依赖冲突问题的具体表现形式. 此外, 基于对用户给定的实际依赖库需求及本地环境中库依赖的综合考虑, 本文提出了一种面向用户需求的 Python 库依赖冲突检测和解决 (user-oriented Python library dependency resolution, UPLDR) 方法. 具体来说, 该方法通过从 PyPI 官方网站获取第三方库依赖相关的元数据, 构建结合 Python 领域的库版本信息知识库. 面向用户给定的依赖声明需求文件和从用户本地环境中解析出来的依赖项集合, 从知识库中进行依赖的匹配进而完成依赖冲突问题的检测, 如果存在依赖冲突问题, 则将修复问题转化为图搜索问题, 通过贪婪搜索来寻找与用户给出的需求最接近的第三方库兼容版本, 以达到修改最小的可行固定解的解决方案.

综上所述, 本工作主要贡献如下.

1) 通过对 103 个 Python 开源项目全面定量分析, 深入探讨第三方库的更新频率以及依赖冲突问题的具体表现形式, 同时提供对现有问题更为细致的理解.

2) 基于对用户给定的包含所需依赖库版本约束及本地环境约束的全面考虑, 本文提出了一种面向用户需求的 Python 依赖冲突检测与解决的 UPLDR 方法, 旨在提供更贴近用户的个性化精准的解决方案.

3) 本文在相关数据集上进行了全面的实验探究, 在构建的数据集中将 UPLDR 与相关基线方法进行比

较. 结果证明了 UPLDR 在检测和解决 Python 依赖冲突问题方面的有效性和实用性.

1 相关工作

在现代软件开发实践中, 集成他人编写的代码库已成为加速开发进程的常规做法[15]. 特别是在 Python 生态系统中, 第三方库重用不仅促进了开发效率, 也带来了项目间错综复杂的依赖关系, 这些依赖关系的有效解析和依赖冲突问题检测和解决同样至关重要.

1.1 Python 生态系统

在实践中, 基于 Python 生态系统中内部依赖关系, 项目之间彼此共同发展, 共同促进生态系统繁荣发展. Hoving 等人[16]通过对 Python 开源生态系统分析其自身不同特征和可用数据集, 来为 Python 生态系统项目的持续增长与健康发展提出建议. Zhang 等人[17]通过对 Python 框架 API 演变及用法特点分析, 设计实现工具 PYCOMPAT 来解决 API 兼容问题. Vu 等人[18]对集成到 Python 生态系统的库上传过程中的有限自动化控制进行研究, 并识别潜在的恶意库工具目标, 同时提出了一种自动识别组合抢注和拼写错误库名称的方法, 从而确保 Python 生态系统中上传和分发库的安全性. Valiev 等人[19]对影响开源 Python 项目可持续性的生态系统层面因素的混合方法进行定量研究, 同时证明了项目联系数量和依赖关系网络相应位置对持续项目活动的重大影响. Alfadel 等人[20]和 Latendresse 等人[21]对 Python 生态系统中库的漏洞报告和生态系统中的 Python 库进行实验研究, 并且通过改进发现、修复和管理软件库漏洞的过程中提供措施来保证 Python 生态系统的安全.

1.2 依赖推断

有大量研究工作致力于自动推断软件的环境依赖关系[22]. Pipreqs[23]通过分析代码中的 import 语句来为 Python 项目构建 requirements.txt 文件. 对于运行环境中的依赖推断问题, DockerizeMe[13]首先创新性地提出了通过从 PyPI 官方第三方包存储库中离线获取 Python 库的相关资源和依赖关系, 然后通过图推理过程构建 Docker 规范, 精准推断执行 Python 代码片段所需依赖关系, 从而有效避免导入环境中的错误. V2[24]通过探索 Python 代码片段可能配置来进一步拓展 DockerizeMe, 它通过检测配置漂移的离散实例来识别过时的代码片段. V2 利用反馈驱动的搜索机制, 大幅减少了需要验证的潜在环境配置数量, 提高了依赖推断的效率. SnifferDog[25]则专注于 Jupyter notebook 执行环境的精确复现与恢复. 它通过构建 API 相关知识库并解析 Python 文件来记录库到 API 的映射, 确保 notebook 执行环境的精确复现.

知识图谱的引入进一步推动了依赖推断的进展. Cheng 等人[12]提出的 PyCRE 方法, 基于领域知识图谱自动推断 Python 兼容的运行时环境, 同时考虑了导入的第三方模块及其被调用的属性. 针对依赖求解的问题, PyCRE 提出一种启发式图搜索遍历算法来推断兼容的运行时环境. 针对不断更新的 Python 解释器和第三方库给 Python 运行时的环境推理带来了许多挑战. Cheng 等人[14]提出 ReadPyE 的 Python 运行环境推理方法. 通过利用丰富的代码信息定义一种命名相似性度量来匹配未知模块的候选包, 并为多个候选包设置优先级, 从而根据验证日志中匹配的异常模板对推断出的环境进行迭代验证和调整. 在 3 个真实数据集上的实验结果表明, ReadPyE 可以帮助程序员减少在推断 Python 运行环境上花费的时间, 并促进自动化软件配置管理. 本文不仅集成已有研究成果, 还进一步考虑到用户实际需求和本地环境约束, 从而给出更贴近用户需求的个性化解决方案, 极大提升了检测和解决的有效性和时间效率.

1.3 依赖冲突检测和解决

针对依赖冲突问题检测方面, Wang 等人[8]通过对真实环境中依赖冲突问题的表现模式和修复策略进行实证研究, 发现其产生冲突问题的关键因素. 同时设计并实现了一种持续监控 PyPI 生态系统依赖冲突的技术 Watchman. 它通过为每个库构建完整依赖图来模拟 Python 项目安装的过程, 并随着库在 PyPI 上的演变进行迭代更新, 从而进行直接依赖和间接依赖之间冲突问题的检测, 最后可向产生依赖冲突项目的开发者报告问题并提出常用的解决方案作为修复建议. Riddle 方法[26]基于实证研究[27]通过自动化生成测试用例, 在依赖冲突的库版本中触发具有相同签名的 API 的不一致行为, 从而解决依赖冲突的问题. Jia 等人[28]对 278 个流行 Python 库项目的直接与潜在依赖关系的分布和相关性进行全面研究, 同时分类并分析了 316 个依赖冲突问题, 此外还对依赖冲突问题的检测工具进行深入的全面研究. Peng 等人[29]针对 Python 生态系统中配置问题进行实证研究, 同时对于潜在配置问题提出

PyCon 源码检测器, 对第三方库的设置和使用阶段进行不同检查, 从而完成有效的依赖推理.

针对依赖冲突问题解决方面, Wang 等人[30]通过考虑回溯策略来探索不同模式下依赖冲突问题可行的约束求解解决方案, 离线则是为 PyPI 中托管的所有库构建本地依赖知识库, 并根据给定 Python 项目的配置文件生成安装脚本, 并最终管理所有所需库的安装. 然后将本地库的版本约束转化为 SMT 表达式, 使用 Z3 约束求解工具进行依赖求解, 得到满足的依赖项的相关信息. Wang 等人[31]提出了一种基于行为一致性的版本扩展方法 LooCo, 用来解决由低质量版本约束导致的依赖冲突问题. 该方法通过 Python 项目和第三方库的静态分析, 获取除原始版本约束外可以被项目安全使用的其他库版本, 从而扩展项目对于第三方库的版本约束, 并可以保证扩展后的第三方库在项目中的行为与扩展前保持一致, 从而解决依赖冲突问题. Ye 等人[32]提出一种基于知识的 Python 依赖推理技术 PyEGo, 它能够自动推断出 Python 程序第三方库、Python 解释器和兼容版本的系统库依赖关系. PyEGo 通过构建可表示相关关系和约束的依赖知识图谱, 然后通过提取程序特征查询程序相关子图, 最后通过求解子图中约束, 输出最新兼容的依赖版本. 其中, 方法 PyDFix[33]专注于检测和修复由第三方库错误导致的 Python 构建中的不可重现性. 通过将原始和当前的构建日志作为输入, 并通过迭代地尝试相同库的其他版本来修复错误. 结果表明, 通过调整依赖约束来修复依赖声明问题是有效的.

2 方法实现

图 1 详细展示了面向用户需求的 Python 库依赖冲突解决方法的整体架构图. 该方法包括 4 个关键部分. 首先通过从 PyPI 官方平台获取 Python 第三方库包含库版本约束等信息的元数据, 并对第三方库之间的不同依赖库名称、对应的库版本约束及 Python 版本等信息进行依赖解析, 通过调用 Libraries.io 接口来追踪第三方库的更新情况, 确保知识库中信息的迭代更新, 从而构建出 Python 第三方库的专属相关领域知识库 (第 2.1 节). 然后通过提取用户提供的包含所需依赖库版本及 Python 版本等需求的文件内容, 解析出其中包含的依赖第三方库和对应版本, 并通过结合 Pipdeptree 命令获取到的用户本地环境中所包含的依赖信息, 从而获取项目运行包含的所有依赖项集合 (第 2.2 节). 此外, 根据获取到的依赖项集合,并结合知识库中知识的查询, 来检测其中是否存在依赖冲突问题 (第 2.3 节). 一旦发现依赖冲突问题, UPLDR 方法将构建依赖图, 并通过智能图论算法解决这些冲突. 在冲突问题解决之后, UPLDR 方法将提供一份经过严格筛选的推荐解决方案列表; 若依赖冲突问题无法解决, 则会提供详尽的失败信息.

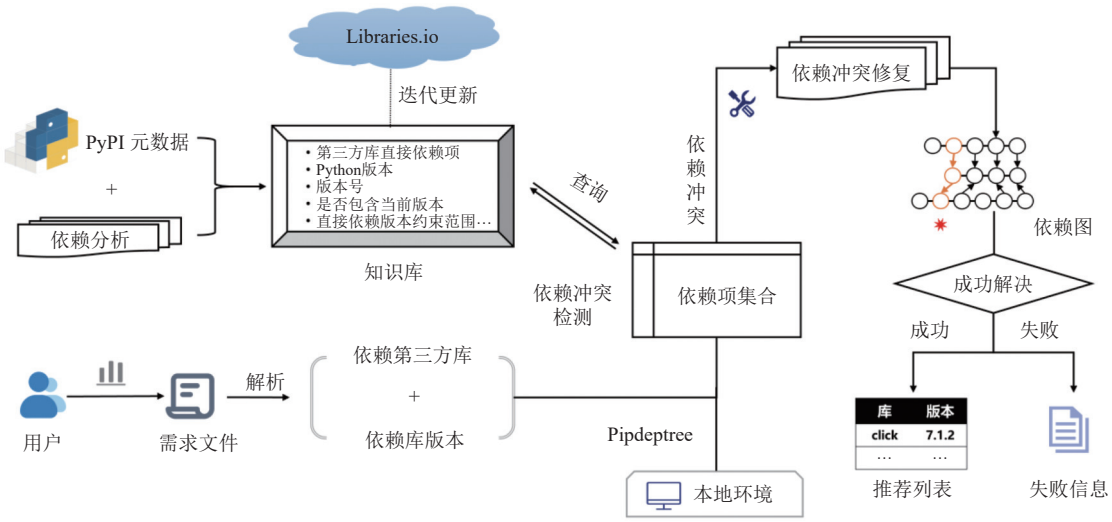

图 1 UPLDR 整体架构图

2.1 知识库构建

知识库的构建是 UPLDR 方法的核心组成部分, 它为依赖冲突检测和解决提供了必要的数据支持. 以下是构建知识库的具体步骤: 1) 选择 PyPI 作为主要的数据源, 因为它是 Python 开发者官方资源宝库, 提供了丰富的声明式元数据. 可以通过特定的 URL 结构, 例

如对于一个库 package, 则可以搜索官方网址如“https://pypi.org/[package]/json”来访问特定包的 JSON 数据节点, 从而获取到该库的详细元数据. 这些元数据不仅包括了库名称和版本号, 还涵盖依赖关系和其他关键信息, 如所需 Python 版本等. 2) 解析从 PyPI 获取的 JSON 数据, 提取出库的直接依赖项、适配的 Python 版本、版本号以及版本约束范围等信息. 3) 通过静态分析这些元数据, 将结构化后的 JSON 数据存储到本地文件系统中, 每个库的元数据对应一个 JSON 文件, 同时以库名作为目录名, 将对应 JSON 文件进行存储. 本文能够深入理解每个库的依赖结构和兼容性要求. 具体来说, 通过表 1 展示 Python 库元数据的属性描述, 可以提供清晰的视角来审视和利用 PyPI 上的信息资源, 从而能够更加高效地进行软件开发, 确保依赖关系的明确性和项目的稳定性.

表 1 Python 库的元数据属性描述

属性	描述
potential_require	第三方库直接依赖项
python_require	适配的Python版本
version	版本号
skip_start, skip_end	是否包含当前版本
start_node, end_node	直接依赖项的版本约束范围

以 requests 库的 2.27.1 版本为例, 它的直接依赖库为 urllib3 (<1.27, ⩾1.21.1), 可以适用于 Python 3.8. 将其数据处理后可以表示为 Python 库的元数据属性: potential_require 为 urllib3, python_require 为 py38, version 为 2.27.1, start_node 为 1.27, end_node 为 1.21.1, skip_start 为 True (跳过该版本, 即不包含 1.27 版本), skip_end 为 False (不跳过该版本, 即包括 1.21.1 版本), 从而可以简化知识存储, 节省空间成本.

通过将这些数据规范化并存储, 本文可以构建一个本地 Python 依赖知识库. 这个知识库不仅能够帮助开发者快速了解和选择合适的库, 还能够在项目开发过程中提供依赖管理和冲突解决的参考. 同时考虑到第三方库的快速迭代更新, 则通过调用 Libraries.io 接口定期查询 PyPI 上的数据. 通过比较知识库中数据与其中官方文档包含的依赖版本约束的不同, 对于新添加的库版本, 则将其相应的元数据添加到知识库中, 从而不断及时获取到更新的信息.

2.2 用户需求获取

用户可以通过 requirements.txt 的形式提供构建 Python 环境中所需要的依赖库和 Python 版本, 在获取到用户提供的依赖项文件后, 对每一行中的相关版本约束分析, 包括依赖第三方库和对应的版本内容.

首先通过 Python 的 open 函数打开 requirements.txt 配置文件, 以逐行读取文件中的依赖项. 在读取的过程中, 需要注意忽略带有 “#” 开头的注释行, 因为它们不包含有效的依赖信息. 接着利用 Requirements 库来解析每一行的依赖项, 该库能够识别并验证依赖项的标识符、名称及版本范围的合法性. 这一步骤是确保依赖项正确无误的关键环节. 解析完成后, 可以得到一个包含所有依赖库及其对应版本的集合. 这个集合不仅包括了用户明确指定的依赖项, 还可能包括了这些依赖项的间接依赖.

为获得一个完整的依赖视图, 还需要考虑本地环境中已经存在的依赖库. 通过调用 Pipdeptree 命令, 能够递归地列出项目中所有依赖库的层次结构. 通过这个命令, 能够获取到本地环境中除了用户给出的需求列表之外的其他依赖库的配置信息. 最后, 通过将解析得到的用户需求文件中的依赖项与获取到的本地环境中的依赖项进行合并处理, 并解决可能出现的版本冲突问题. 通过这种方式, 不仅能够确保依赖项的完整性, 还可以为后续的依赖冲突检测和解决提供坚实的基础.

2.3 依赖冲突检测

在构建 Python 项目过程中, 不仅要关注用户给定需求中明确指定的依赖库, 还需要深入挖掘这些库的传递依赖, 即那些间接依赖的库. 这些库可能隐藏着版本不兼容的风险, 从而影响整个项目的稳定性和兼容性. 为全面掌握项目依赖的全貌, 需要将第 2.2 节获取的用户需求和本地环境中提取的库组件依赖共同组成依赖项集合, 与知识库中的元数据依赖项进行匹配查询, 以识别每个库的所有传递依赖关系. 在这一基础上, 构建一种算法, 用于检测依赖项之间是否存在依赖冲突问题. 该算法不仅能够识别直接的版本依赖冲突, 还能够深入分析依赖树, 发现并报告那些可能不易察觉的间接冲突, 如算法 1 所示.

算法 1. ConflictCheck

输入: 依赖项集合 *dependencies*, 知识库 *kb*.
输出: 依赖冲突报告 *dc_issues*.

1. *visited*←{}; // Initialize an empty set
2. *dc_issues*←[];
3. **for** *d* in *dependencies* **do**

```
4.   if d is not in visited then
5.     visited.add(d);
6.     for d in kb do:
7.       d_tree←kb[d]; //获取依赖
8.     for direct and indirect in d_tree do
9.       if isConflict(direct, indirect) then
10.          report(direct, indirect);
11.          dc_issues.add(report);
12.       end if
13.  end if
14. return dc_issues
```

算法 1 展示了用于检测依赖项之间冲突问题方法 ConflictCheck 的伪代码. 该算法主要实现通过遍历查询并与知识库交互的过程. 该算法首先初始化 *visited* 访问节点空集合, 从而避免在依赖项集合中重复检查相同依赖性, 同时初始化冲突问题 *dc_issues*, 用来存储检测到的依赖冲突报告 (第 1–2 行). 然后, 通过遍历获取到的依赖项集合 *dependencies*, 处理每一个未访问的依赖项. 其中, 对于每个未访问的依赖项, 先将其标记为已访问, 并从知识库 *kb* 中获取相关的依赖约束信息 (第 3–7 行); 随后算法通过遍历上一步骤中获取的依赖项的直接和间接依赖库的版本约束信息, 判断是否存在依赖冲突问题, 如果存在冲突, 则生成对应的冲突报告, 并将其添加到 *dc_issues* 中 (第 8–13 行). 最后返回有关依赖冲突问题集合 (第 14 行).

函数 *isConflict*(*direct*, *indirect*) 用于检测两个依赖项之间是否存在版本冲突. 该函数是依赖冲突检测算法的核心部分, 它通过比较直接依赖 (*direct*) 和间接依赖 (*indirect*) 的版本范围来判断是否存在冲突, 其伪代码如算法 2 所示.

算法 2. *isConflict*(*direct, indirect*)

输入: 直接依赖 *direct*, 知识库 *kb*, 间接依赖 *indirect*.
输出: 返回布尔值, 存在冲突则返回 True, 否则 False.

```
1. d_version←kb.getVersionRange(direct);
2. ind_version←kb.getVersionRange(indirect);
3. if not (d_version.max≥ind_version.min and d_version.min≤int_version.max) then
4.   return True;
5. end if
```

首先, 获取版本的范围, 这包括每个依赖项的最小和最大允许版本 (第 1, 2 行). 然后, 通过比较两个依赖项的版本范围来判断是否存在冲突. 冲突存在的情况以下两种: 1) 直接依赖的最大版本小于间接依赖的最小版本; 2) 直接依赖的最小版本大于间接依赖的最大版本 (第 3 行). 最后如果检测到上述任一冲突条件, 函数立即返回 True, 表示存在冲突. 如果所有条件都不满足, 即两个依赖项的版本范围有重叠, 函数返回 False, 表示不存在冲突 (第 4, 5 行).

依赖冲突检测算法 (ConflictCheck) 的核心创新点在于其面向用户需求的设计和实现方式, 以及它在处理 Python 依赖冲突时的全面性和实用性. 以下是该算法相对于现有依赖冲突检测算法的创新性特点: 一方面, 该算法结合了知识库的冲突检测. 通过预先构建的包含依赖项间的复杂关系的知识库, 来进行依赖冲突问题检测, 结合领域知识的检测方法可能比现有的仅基于当前依赖树的算法更为精确和全面. 另一方面, 可迭代更新知识库. 通过调用 Libraries.io 接口定期查询 PyPI 上的数据, 这种动态更新机制可以确保依赖冲突检测的准确性和实效性.

2.4 依赖冲突修复

对于给定出现依赖冲突的 Python 依赖关系声明文件, UPLDR 方法首先根据解析得到的依赖关系版本约束, 通过搜索知识库来获取所有依赖项的传递依赖关系, 同时考虑它们与 Python 版本的兼容性. 然后, 通过 Python 版本和依赖关系约束的关系来构造依赖图, 其中每个节点代表一个特定的 Python 版本或第三方库版本. UPLDR 将依赖冲突修复问题转化为一个图搜索问题. 寻找是否存在一条能够满足所有版本约束的全连通路径. 如果这样路径不存在, UPLDR 会识别并报告依赖声明中的问题和具体冲突的依赖项. 为了解决这些冲突, UPLDR 采用贪婪搜索算法, 寻找能够形成全连通路径的最近版本. 这种方法确保了依赖图的连通性, 同时尽可能地满足原始的版本约束. 算法 3 提供了 UPLDR 方法的详细过程.

算法 3. DependencyFix 依赖冲突修复

输入: 依赖项集合 *dependencies*.
输出: 解决报告 *resolved* 或失败原因 *conflict_issues*.

```
1. dgraph←{};
2. visited_nodes←{}; // Initialize the set
3. for d in dependencies do
4.   if d is not in visited_nodes then
5.     dgraph.addNode(d);
6.     visited_nodes.add(d);
7.     for node in dgraph.nodes do
8.       for other in dgraph.nodes do
```

```
9.          if not dgraph.hasEdge(node, other) then
10.             dgraph.addEdge(node, other)
11.         end if
12.         FullyConnectedPath ←dfs_search(dgraph);
13.     if path is None then
14.         for conflict in ConflictCheck(dgraph) do
15.             report(conflict)
16.     else // If a fully connected path is found
17.         nearest_versions←greedySearch(dgraph);
18.         update(nearest_versions)
19.     end if
20. end if
21. return resolved or conflict_issues
```

算法 3 展示了用于进行出现依赖冲突问题的解决方法 DependencyFix 的伪代码. 该算法首先初始化构建依赖图及遍历节点的集合 (第 1, 2 行); 通过遍历上述产生依赖冲突问题的依赖项集合, 根据其进行依赖图的构建, 按照每个节点分别获取它的相邻节点并得出相应的依赖边 (第 3–11 行); 根据构建的依赖图进行深度搜索寻找全连接路径, 若无法找到全连接路径, 则报告出相应的依赖冲突问题 (第 12–15 行); 反之则对依赖图进行贪婪搜索获取最近用户所需的依赖项最接近的版本; 最后进行解决方案或出错具体信息的内容输出 (第 16–21 行).

3 实验与结果分析

Python 生态系统中库的更新频率对开发者和项目维护者至关重要[34]. 当一个库被开发出后, 为满足实际的需求或进行修复其中的缺陷漏洞, 开发者需要不断进行更新, 因此 Python 库的版本更新信息对于使用者来说是一项重要的信息, 使用者通常不会选择一个无人维护的 Python 库来使用[35]. 对于 Python 项目依赖声明问题以往是研究 pip 安装策略引起的依赖冲突, 本文则从依赖声明文件的项目总体角度出发研究其中不同的表现形式. 同时旨在通过一系列实验全面评估 UPLDR 方法在处理 Python 项目依赖冲突问题上的性能表现. 本文研究聚焦于以下 4 个核心问题.

RQ1: 不同 Python 库有怎样的版本更新频率?

RQ2: Python 项目中依赖声明问题常见表现形式?

RQ3: UPLDR 方法在解决 Python 依赖冲突方面的有效性如何? 相较于现有的解决方案, 其在依赖冲突解决方面的表现具有哪些优势?

RQ4: UPLDR 方法在不同 Python 项目中, 解决依赖冲突的时间效率表现如何?

本文第 3.1 节中详细介绍实验设置; 第 3.2–3.5 节中, 分别针对上述研究问题提供深入的分析和解答; 最后对 UPLDR 方法实际应用中的优势进行呈现和分析.

3.1 实验设置

● 实验环境: 本研究所有实验均在 Linux 服务器上进行, 该服务器具备高性能的硬件架构, 配置 32 核的 Intel® Xeon® CPU E5-2640 v3 处理器, 具备 2.60 GHz 的处理速度, 确保了处理复杂计算任务时的高效率. 此外, 服务器配备了高达 128 GB 的内存, 为大规模数据处理和多任务并行处理提供了充分支持. 在软件配置方面, 实验环境采用 Ubuntu 22.04 操作系统, 确保了系统的稳定性和安全性. 使用 Python 3.6 作为编程语言, 以利用其丰富的库和框架资源. 同时采用 PyCharm 作为计算机编程的集成开发环境.

● 基线对比: 本文采用两种公开源代码和数据的基线工具进行复现, 然后将其作为对比分析的基准. 接下来则对这些工具进行简要描述.

Watchman: 是一个为 Python 库生态系统设计的自动化工具, 用于持续监控和诊断依赖冲突问题. 它通过构建和分析完整的依赖图, 能够及时发现并预警因版本冲突导致的构建失败, 同时提供诊断信息帮助开发者修复这些问题. Watchman 基于对大量真实世界 Python 项目的实证研究, 不仅能够检测当前的依赖冲突, 还能预测潜在的未来冲突, 从而提高 Python 项目构建的稳定性和可靠性.

PyEGo: 是一个自动化工具, 用于推断 Python 程序的环境依赖性, 其中包括第三方包、Python 解释器和系统库. 它首先构建依赖知识图谱 PyKG, 然后通过抽取其中子图, 并结合程序特征提取和约束求解, 自动识别兼容版本的依赖项, 从而使该方法精确推断 Python 程序所需的环境依赖.

● 依赖冲突实验数据集: 本文参考 Python 开发社区中对于文献[30,31]数据收集的经验实践, 数据收集工作主要通过以下步骤在著名开源项目托管平台 GitHub 上进行.

步骤 1: 收集 Python 项目中的依赖冲突问题. 通过 GitHub API 收集其中包含关键词 “requirement”或“version constraint”或“dependency” (不区分大小写) 的问题报告, 然后筛选掉没有依赖声明文件 (setup.py 或 requirements.txt) 的项目中包含的依赖冲突问题, 同时

删除掉重复且干扰的问题, 最后保留符合上述条件所有依赖冲突的问题报告.

步骤 2: 进一步筛选确立依赖冲突问题. 通过对问题报告及相关讨论中筛选出符合特定条件的问题报告, 如果保留则需要满足以下两个条件: 一方面, 该报告的问题不是由于用户操作不当导致的, 而是专注于明确指出依赖冲突问题核心的案例; 另一方面, 对于依赖冲突问题中对应版本及包含对于问题的完整解决方案的讨论. 这不仅可以帮助开发者理解依赖冲突问题的具体情况, 还为他们提供了宝贵的参考, 以便在遇到类似问题时候能够迅速找到解决方案. 对于出现质疑的问题, 由作者讨论并达成一致. 针对上述标准的筛选后, 最后保留 103 个依赖冲突问题报告进行后续的研究.

● 评估指标: 通过对每个相邻版本之间的间隔时间进行求和平均计算, 这种方法提供了一个更平衡的视角, 因为它考虑了所有版本的更新间隔, 而不是仅依赖于最近或最早的更新. 这有助于更准确地评估库的维护频率和开发者的活跃度. 平均更新间隔较短可能意味着库的维护者非常活跃, 频繁地发布新版本. 相反, 如果平均更新间隔较长, 可能意味着库的维护不够活跃, 或者库已相对稳定, 不需频繁更新. 对于版本更新平均间隔时间, 按照式 (1) 进行计算:

$$interval = \frac{\sum (V_n - V_{n-1})}{N-1} \tag{1}$$

其中, V_n 代表该库发布的第 n 个版本的具体发布日期, N 代表该库的发布版本总数.

在对依赖冲突问题解决的有效性进行评估时, 本文采用了一个基于实际解决的问题数量与数据集中真实问题总数之间比例的度量方法. 具体而言, 式 (2) 定义了依赖冲突问题解决的有效性:

$$accuracy = \frac{Inf(declaration)}{total} \tag{2}$$

其中, $Inf(declaration)$ 表示数据集中成功解决依赖冲突问题的依赖声明文件的数量, 而 $total$ 则代表数据集中所有依赖声明文件的总数. 通过计算这一比例, 能够量化地评估依赖冲突问题解决的效果.

3.2 RQ1: 库版本更新频率

对于 Python 库的版本更新信息的获取, 本文通过专门提供第三方库开源信息平台 Libraries.io 获得信息. 首先, 根据衡量受欢迎程度的 SourceRank 指标进行排名前 100 的库的元数据获取, 其中包括截至目前库版本更新的数量及不同版本之间的更新时间. 然后, 通过列出版本发布日期, 进行计算相邻版本之间时间间隔数据的整理.

通过对收集到的 103 个 Python 库的依赖版本的平均更新间隔时间进行研究分析, 本文可得到平均数和中位数等一些关键相关统计数据, 这些数据可以帮助揭示 Python 库的发行版本数量和平均发行版本间隔时间的分布情况.

如图 2(a) 所示, 平均每个库有大约 125 个版本, 反映了 Python 库的一般更新频率, 中位数 98.5 则表明一半的库拥有超过 98 个版本, 这显示了 Python 库通常具有较多的版本迭代, 同时也表明开发者一直在积极进行库版本的维护. 如图 2(b) 所示, 平均更新周期约为 50 天, 可为开发者提供关于库更新频率的参考. 中位数 46.5 则意味着一半的库在大约 47 天内会发布一个新版本. 由此可见, 开发者不仅一直在维护这些库并且更新维护的频率也很高.

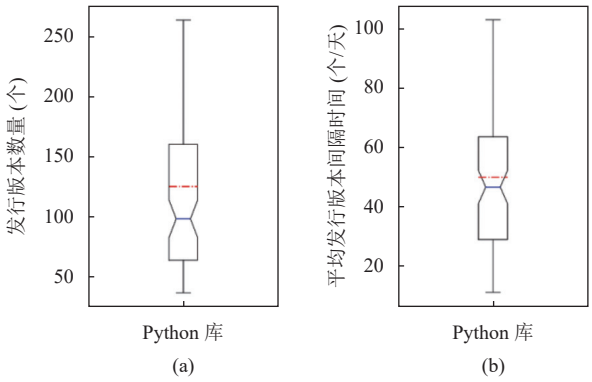

图 2 库发行版本数量及平均间隔时间

通过上述分析, 可以看到 Python 库的版本更新频率和发行版本数量对依赖管理具有重要影响. 对于更新频繁的库, 开发者需要不断检查和更新依赖, 以确保兼容性和安全性. 库频繁更新的版本控制要求开发者精确控制依赖版本, 以避免因自动更新导致的不兼容问题. 开发者可以根据这些数据来制定合理的依赖管理策略, 以确保项目的稳定性和可维护性.

3.3 RQ2: 依赖冲突表现形式

在对 103 个 Python 库的依赖声明报告深入分析的过程中, 本文根据冲突发生的环境和涉及的实体, 从项目角度出发将问题分为几个主要类别, 如图 3 所示. 分类可以帮助我们更精确地理解依赖冲突的性质和可能的解决方案. 以下是对不同表现形式的详细描述.

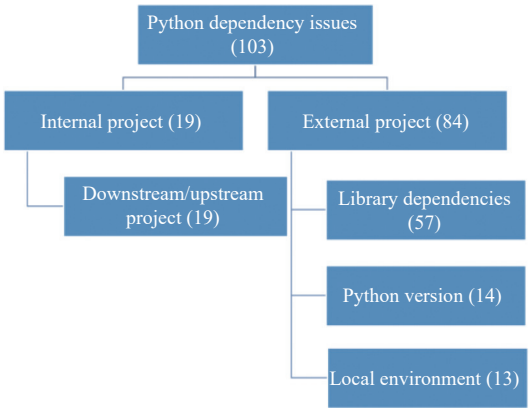

图 3 Python 依赖声明问题分类

(1) 项目之间 (上下游项目之间冲突)

这一类别的冲突发生在不同的项目之间, 通常涉及项目的依赖关系链. 在这种情况下, 一个项目可能依赖于另一个项目, 而这些项目又依赖于其他库或模块, 导致依赖版本之间的不一致或冲突. 这类冲突通常涉及多个项目的依赖树, 可能需要协调多个项目维护者来解决. 例如, 项目 A 依赖于项目 B 的特定版本, 而项目 B 又依赖于库 C 的另一个版本, 这可能导致与项目 A 期望的库 C 版本不兼容. 解决策略则是需要重新设计依赖关系, 或者通过虚拟环境、容器化技术来隔离不同项目的依赖.

(2) 项目之内的冲突问题

1) 第三方库之间依赖冲突: 这类冲突发生在单个项目内部, 涉及项目直接依赖的不同第三方库之间的版本冲突, 它们通常由于库之间的间接依赖关系引起, 例如, 两个库可能依赖于同一库的不同版本. 例如, 一个项目同时依赖于库 X 的 1.0 版本和库 Y 的 2.0 版本, 而库 Y 又需要库 X 的 2.0 版本. 解决策略则是可以通过升级或降级某些库的版本, 或者使用依赖管理工具来解决这些冲突.

2) 第三方库和 Python 版本依赖冲突: 这一类冲突涉及项目依赖的第三方库与 Python 解释器版本之间的不兼容性问题. 常见的特点为某些库可能只支持特定版本的 Python, 而项目其他部分可能需要不同版本的 Python. 例如, 一个项目需要使用 Python 3.8 的特性, 但某个库只支持到 Python 3.6. 解决策略则是等待该第三方库更新, 或寻找替代库共同可用 Python 版本.

3) 本地环境之间冲突: 这类冲突涉及项目依赖与本地开发或部署环境之间的不匹配. 由于本地环境缺少某些库的依赖, 从而导致环境配置与项目期望的配置不一致. 解决策略则是确保所有环境的一致性, 使用环境管理工具或虚拟环境来标准化部署配置.

通过这种分类, 可以更系统地理解和解决 Python 项目中的依赖冲突问题, 从而提高项目的稳定性和可维护性. 这种分析也有助于开发者在项目规划和维护阶段做出更明智的决策.

3.4 RQ3: 解决依赖冲突的有效性

本实验核心目标是通过定量分析, 评估 UPLDR 方法与其他基线方法在解决依赖冲突问题方面的有效性. 本文通过对比不同方法在解决依赖冲突数量上的差异, 对其在实际应用中的表现进行量化评估. 表 2 展示了 3 种方法在 103 个依赖声明文件上的解决问题的结果, 其中包括它们解决问题的数量以及指标评估的内容. 此表格还详细列出了解决问题数量的不同分类, 其中包括第三方库之间的依赖冲突问题、第三方库与 Python 版本之间的依赖冲突问题和第三方库与本地环境之间的依赖问题的数量.

表 2 依赖冲突问题解决有效性

方 法	解决问题数量 (个)				*accuracy* (%)
	第三方库之间	第三方库与Python版本	第三方库与本地环境	总数	
Watchman	57	0	0	57	55.34
PyEGo	48	14	0	62	60.19
UPLDR	51	14	13	78	75.73

此项实验结果不仅证实了 DRPDC 在从第三方库以及实践应用中获取知识的有效性, 同时也突显了其在多层次依赖冲突检测方面的综合优势.

Watchman 在处理特定类型的依赖冲突时具有一定效果, 但在全面性方面有待提升, 其总体准确率为 55.34%, 表明其在处理第三方库之间的依赖冲突时表现出色, 但在处理第三方库与 Python 版本及第三方库与本地环境的依赖冲突问题上未能取得进展. PyEGo 在解决第三方库与 Python 版本之间的依赖冲突问题上取得了进展, 解决了 14 个问题, 总体准确率为 60.19%, 显示了其在特定领域的优势. UPLDR 在所有类型的依赖冲突问题上均表现出色, 共解决了 78 个问题, 准确率高达 75.73%, 在 3 种方法中表现最佳, 这表明 UPLDR 在解决依赖冲突问题上具有较高的全面性和有效性.

与此同时, 将依赖冲突问题的解决数量占比进行直观可视化, 如图 4 所示.

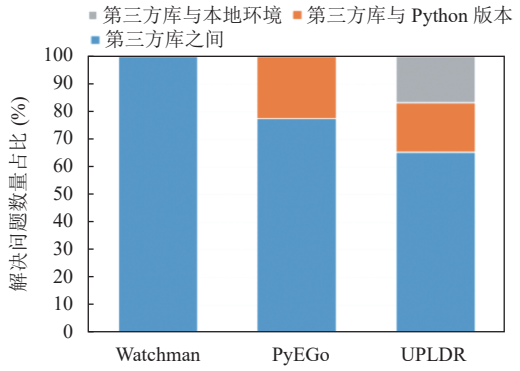

图 4 解决依赖冲突问题数量占比

根据实验结果, UPLDR 方法在解决依赖冲突问题上显示出了显著的优势, 无论是在数量上还是在准确率上, 都超越了其他基线方法. 该发现强调了 UPLDR 在实际应用中的潜力, 尤其是在需要处理多种类型依赖冲突的复杂项目中. UPLDR 高准确率和广泛适用性使其成为一个强有力的工具, 能够有效支持 Python 项目的依赖管理.

3.5 RQ4: 解决依赖冲突的时间效率

本实验旨在评估不同方法在解决依赖冲突问题时的时间效率. 通过比较不同工具解决依赖冲突所需时间, 可以量化它们的性能, 并确定它们在实际应用中的实用性和效率.

表 3 记录了每种工具解决成功和解决失败的依赖冲突问题所需的平均时间和总时间. 在实验中, UPLDR 成功解决了 78 个问题, 平均耗时仅为 1.39 s, 总耗时 108.42 s, 显著优于 PyEGo 和 Watchman 方法, 后两者分别解决了 62 个和 57 个问题, 平均耗时分别为 2.68 s 和 2.26 s, 总耗时分别为 166.16 s 和 128.82 s, 这表明 UPLDR 在处理依赖冲突时不仅效率高, 而且在时间成本上也具有明显优势.

表 3 解决依赖冲突问题的时间对比

方法	总数量	成功解决			解决失败		
		Num (个)	Average (s)	Total (s)	Num (个)	Average (s)	Total (s)
UPLDR	103	78	1.89	147.42	25	1.88	47.00
PyEGo	103	62	2.68	166.16	41	1.94	79.54
Watchman	103	57	2.26	128.82	46	2.13	97.98

UPLDR 在解决依赖冲突问题上表现出了较高的效率, 平均耗时最短, 且总耗时也相对较低. 这表明 UPLDR 在处理依赖冲突问题时不仅有效, 而且速度快. PyEGo 虽然在解决依赖冲突的数量上略低于 UPLDR, 但其平均耗时较长, 总耗时也较高, 这可能意味着在处理某些复杂依赖冲突时, PyEGo 需要更多时间. Watchman 解决依赖冲突问题上的平均耗时和总耗时均高于 UPLDR, 但低于 PyEGo, 表明其在时间效率上介于两者之间.

根据实验结果, UPLDR 在解决依赖冲突问题的时间效率上表现最佳, 不仅解决了最多的依赖冲突问题, 而且所用时间也最短. 这表明 UPLDR 是一个高效且有效的工具, 特别适合需要快速解决依赖冲突的场景. PyEGo 和 Watchman 虽然也能有效解决依赖冲突, 但在时间效率上不如 UPLDR, 可能需要进一步优化以提高处理速度.

4 结论与展望

本研究通过深入分析 103 个 Python 开源项目的第三方库更新和依赖冲突问题, 揭示了 Python 生态系统中存在的一些关键挑战. 本文的分析不仅量化了第三方库的更新频率, 而且从项目角度深入探讨了依赖冲突的具体表现形式, 为理解这一问题提供了新的视角. 基于这些发现, 本文提出了 UPLDR 方法, 这是一种面向用户需求的检测和解决 Python 依赖冲突的策略. 同时通过与现有方法的对比实验, 证明了 UPLDR 在检测和解决依赖冲突方面的有效性和实用性. 未来, 我们计划进行更全面的探索自动化更新机制, 同时确保兼容性和最小化冲突. 通过增强 UPLDR 的安全性分析能力, 以识别和缓解因依赖更新可能引入的安全漏洞. 这将有助于减少开发人员在库更新过程中面临的依赖冲突问题.

参考文献

1 Bogdanchikov A, Zhaparov M, Suliyev R. Python to learn programming. Journal of Physics: Conference Series, 2013, 423(1): 012027.
2 TIOBE. https://www.tiobe.com/tiobe-index/. [2024-11-05].
3 Virtanen P, Gommers R, Oliphant TE, *et al*. SciPy 1.0: Fundamental algorithms for scientific computing in Python. Nature Methods, 2020, 17(3): 261–272.
4 Teoh TT, Rong Z. Python for data analysis. Artificial Intelligence with Python. Singapore: Springer, 2022. 107–122.
5 Ghimire D. Comparative study on Python Web frameworks: Flask and Django. https://www.theseus.fi/handle/10024/339796, 2020. [2024-11-05].

6 李硕, 刘杰, 王帅, 等. 第三方库依赖冲突问题研究综述. 软件学报, 2023, 34(10): 4636–4660. [doi: 10.13328/j.cnki.jos.006666]

7 Python Software Foundation. The pip tool. https://pypi.org/project/pip/. [2024-11-05].

8 Wang Y, Wen M, Liu YP, *et al*. Watchman: Monitoring dependency conflicts for Python library ecosystem. Proceedings of the 42nd International Conference on Software Engineering. Seoul: IEEE, 2020. 125–135.

9 Artho C, Suzaki K, Di Cosmo R, *et al*. Why do software packages conflict? Proceedings of the 9th IEEE Working Conference on Mining Software Repositories. Zurich: IEEE, 2012. 141–150.

10 Patra J, Dixit PN, Pradel M. ConflictJS: Finding and understanding conflicts between JavaScript libraries. Proceedings of the 40th International Conference on Software Engineering. Gothenburg: IEEE, 2018. 741–751.

11 Soto-Valero C, Benelallam A, Harrand N, *et al*. The emergence of software diversity in maven central. Proceedings of the 16th International Conference on Mining Software Repositories. Montreal: IEEE, 2019. 333–343.

12 Cheng W, Zhu XR, Hu W. Conflict-aware inference of Python compatible runtime environments with domain knowledge graph. Proceedings of the 44th International Conference on Software Engineering. Pittsburgh: ACM, 2022. 451–461.

13 Horton E, Parnin C. DockerizeMe: Automatic inference of environment dependencies for Python code snippets. Proceedings of the 41st International Conference on Software Engineering. Montreal: IEEE, 2019. 328–338.

14 Cheng W, Hu W, Ma XX. Revisiting knowledge-based inference of Python runtime environments: A realistic and adaptive approach. IEEE Transactions on Software Engineering, 2024, 50(2): 258–279. [doi: 10.1109/TSE.2023.3346474]

15 沈阚, 黄凯锋, 陈碧欢, 等. 基于静态分析的 Python 第三方库 API 兼容性问题检测方法. 软件学报, 2025, 36(4): 1435–1460. [doi: 10.13328/j.cnki.jos.007224]

16 Hoving R, Slot G, Jansen S. Python: Characteristics identification of a free open source software ecosystem. Proceedings of the 7th IEEE International Conference on Digital Ecosystems and Technologies. Menlo Park: IEEE, 2013. 13–18.

17 Zhang ZX, Zhu HC, Wen M, *et al*. How do Python framework APIs evolve? An exploratory study. Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering. London: IEEE, 2020. 81–92.

18 Vu DL, Pashchenko I, Massacci F, *et al*. Typosquatting and combosquatting attacks on the Python ecosystem. Proceedings of the 2020 IEEE European Symposium on Security and Privacy Workshops. Genoa: IEEE, 2020. 509–514.

19 Valiev M, Vasilescu B, Herbsleb J. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena: ACM, 2018. 644–655.

20 Alfadel M, Costa DE, Shihab E. Empirical analysis of security vulnerabilities in Python packages. Empirical Software Engineering, 2023, 28(3): 59. [doi: 10.1007/s10664-022-10278-4]

21 Latendresse J, Mujahid S, Costa DE, *et al*. Not all dependencies are equal: An empirical study on production dependencies in NPM. Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. Rochester: ACM, 2022. 1–12.

22 Wang Y, Qiao L, Xu C, *et al*. Hero: On the chaos when PATH meets modules. Proceedings of the 43rd International Conference on Software Engineering. Madrid: IEEE, 2021. 99–111.

23 pipreqs. https://github.com/bndr/pipreqs. [2024-11-05].

24 Horton E, Parnin C. V2: Fast detection of configuration drift in Python. Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. San Diego: IEEE, 2019. 477–488.

25 Wang JW, Li L, Zeller A. Restoring execution environments of Jupyter notebooks. Proceedings of the 43rd International Conference on Software Engineering. Madrid: IEEE, 2021. 1622–1633.

26 Wang Y, Wen M, Wu RX, *et al*. Could I have a stack trace to examine the dependency conflict issue? Proceedings of the 41st International Conference on Software Engineering. Montreal: IEEE, 2019. 572–583.

27 Wang Y, Wen M, Liu ZW, *et al*. Do the dependency conflicts in my project matter? Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 319–330.

28 Jia XY, Zhou Y, Hussain Y, *et al*. An empirical study on Python library dependency and conflict issues. Proceedings of the 24th International Conference on Software Quality,

Reliability and Security. Cambridge: IEEE, 2024. 504–515.

29 Peng Y, Hu RD, Wang RK, *et al*. Less is more? An empirical study on configuration issues in Python PyPI ecosystem. Proceedings of the 46th International Conference on Software Engineering. Lisbon: IEEE, 2024. 1–12.

30 Wang C, Wu RX, Song HH, *et al*. SmartPip: A smart approach to resolving Python dependency conflict issues. Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. Rochester: ACM, 2022. 1–12.

31 Wang HY, Liu SG, Zhang LY, *et al*. Automatically resolving dependency-conflict building failures via behavior-consistent loosening of library version constraints. Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. San Francisco: ACM, 2023. 198–210.

32 Ye HJ, Chen W, Dou WS, *et al*. Knowledge-based environment dependency inference for Python programs. Proceedings of the 44th International Conference on Software Engineering. Pittsburgh: IEEE, 2022. 1245–1256.

33 Mukherjee S, Almanza A, Rubio-González C. Fixing dependency errors for Python build reproducibility. Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2021. 439–451.

34 Zulunov RM, Soliev B N. Importance of Python language in development of artificial intelligence. Al-Farg'oniy Avlodlari, 2023, 1(1): 7–12.

35 程弘正, 杨文华. R 语言程序包依赖关系与更新情况的实证研究. 计算机科学, 2024, 51(6): 1–11. [doi: 10.11896/jsjkx.230200069]

(校对责编: 王欣欣)