软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(4):1435−1460 [doi: 10.13328/j.cnki.jos.007224] [CSTR: 32375.14.jos.007224]

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

基于静态分析的 Python 第三方库 API 兼容性问题检测方法*

沈 阚, 黄凯锋, 陈碧欢, 彭 鑫

(复旦大学 计算机科学技术学院, 上海 200433)

通信作者: 黄凯锋, E-mail: kaifenghuang@fudan.edu.cn

摘 要: Python 丰富的开发生态提供了多种多样的第三方库, 极大地提高了开发者的开发效率和质量. 第三方库开发者通过对代码底层的封装, 使得上层应用开发者只需调用 API 就可以使用相关功能, 从而快速完成开发任务. 然而, 第三方库 API 不会保持恒定不变. 由于缺陷修复、代码重构、功能新增等, 第三方库代码会不断更新. 更新后部分 API 发生了不兼容的更改, 从而导致上层应用运行异常终止或者产生不一致的结果. 因此, Python 第三方库 API 的兼容性问题已成为目前开源生态中亟需解决的问题之一. 目前已有相关研究工作对 Python 第三方库 API 兼容性问题展开研究, 但兼容性问题原因的分类覆盖不够完全, 无法输出兼容性问题的细粒度原因. 为此, 对 Python 第三方库 API 兼容性问题的表现形式和产生原因开展了实证研究, 并针对性提出了 Python 不兼容 API 的静态检测方法. 首先, 针对 flask 库和 pandas 库的总共 6 个版本对, 通过收集版本更新日志、运行回归测试相结合的方法, 共收集 108 个不兼容 API 对. 接着, 对收集到的数据开展实证研究, 总结了 Python 第三方库 API 兼容性问题的表现形式和产生原因. 最后, 提出了一种基于静态分析技术的 Python 不兼容 API 的检测方法, 输出句法层面的不兼容 API 问题产生原因. 在 4 个常用 Python 第三方库的共计 12 个版本对上进行了实验评估, 结果表明检测方法具有良好的有效性、泛化性、时间性能、空间性能以及易用性.

关键词: Python; 静态分析; 第三方库; API 兼容性问题; 版本演化

中图法分类号: TP311

中文引用格式: 沈阚, 黄凯锋, 陈碧欢, 彭鑫. 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. http://www.jos.org.cn/1000-9825/7224.htm

英文引用格式: Shen K, Huang KF, Chen BH, Peng X. Detecting Incompatible Third-party Library APIs in Python Based on Static Analysis. Ruan Jian Xue Bao/Journal of Software, 2025, 36(4): 1435–1460 (in Chinese). http://www.jos.org.cn/1000-9825/7224.htm

Detecting Incompatible Third-party Library APIs in Python Based on Static Analysis

SHEN Kan, HUANG Kai-Feng, CHEN Bi-Huan, PENG Xin

(School of Computer Science, Fudan University, Shanghai 200433, China)

Abstract: The rich development ecosystem of Python provides a lot of third-party libraries, significantly boosting developers' efficiency and quality. Third-party library developers encapsulate underlying code, enabling upper-layer application developers to swiftly accomplish tasks by calling relevant APIs. However, APIs of third-party libraries are not constant. Owing to fixes, refactoring and feature additions, these libraries undergo continuous updates. Incompatible changes are seen in some APIs after updates, leading to abnormal termination or inconsistent results in upper-layer applications. Therefore, the API compatibility of the Python third-party library has become one of the issues that needs to be solved. There have been related studies focusing on API compatibility issues of Python third-party libraries, of which reasons have yet to be fully classified so that, the fine-grained cause can not be provided. An empirical study is conducted on the symptoms and causes of API compatibility issues with Python third-party library and a targeted static detection method is proposed. Initially, this study gathers 108 pairs of incompatible API versions by combining version update logs and regression tests across 6 version

* 基金项目: 国家自然科学基金 (62372114); 中国博士后科学基金 (2022M720768)
收稿时间: 2023-12-18; 修改时间: 2024-03-20, 2024-05-06; 采用时间: 2024-05-14; jos 在线出版时间: 2024-07-03
CNKI 网络首发时间: 2024-07-05

pairs of the flask and pandas libraries. Subsequently, an empirical study is conducted on the collected data, summarizing the symptoms and causes of compatibility issues. Finally, this study proposes a static analysis-based detection method for incompatible Python APIs, providing syntactic-level causes of incompatible API issues. This study conducts experimental evaluations on 12 version pairs of 4 popular Python third-party libraries. The results show that the proposed method is good in effectiveness, generalization, time performance, memory performance, and usefulness.

Key words: Python; static analysis; third-party libraries; API compatibility issues; version evolution

作为目前最受欢迎的编程语言之一, Python 以其丰富的第三方库开发生态, 受到了众多开发者的青睐. 第三方库开发者通过对代码底层逻辑的封装, 使得上层应用开发者只需关心相应的 API 调用, 从而可以实现快速开发, 大大提升了开发效率[1]. 然而, Python 的第三方库会因为缺陷修复、功能新增和代码重构等原因而频繁发生更新和迭代. 更新版本的代码中存在对已有版本 API 的变更, 这些变更可能对 API 进行不兼容更改, 从而导致调用该 API 的上层应用项目在更新第三方库版本后出现异常终止或者产生不一致的结果[2]. 因此, 理解 Python 第三方库 API 兼容性问题, 实现相应的不兼容 API 检测方法, 可以有效避免因兼容性问题导致的错误, 保障软件正常可靠运行. 当上层应用开发者调用 Python 第三方库的某一函数、某一类或某一类的成员函数时, 第三方库内部会形成一个调用链, 调用链中涉及此第三方库多个文件中的函数、类以及类的成员函数. 为探求第三方库兼容性问题的细粒度产生原因, 并实现对应的检测方法, 本文中的 API 包括第三方库源代码中的所有函数以及所有类的成员函数.

针对 Python 第三方库 API 兼容性问题, 目前相关的研究工作[2−4]存在一定的局限性. 在 Python 不兼容 API 的数据收集方面, 现有研究通过分析第三方库的更新日志来获取存在兼容性问题的 API. 然而, 这种方式严重依赖于开发者记录更新日志的质量, 往往无法搜集得到所有存在兼容性问题的 API. 在问题产生原因方面, 现有研究对 Python 第三方库 API 兼容性问题的认识还不够充分, 例如, Python 参数包含 5 种类别, 但暂未有工作系统对其中的可变参数、命名关键字参数、关键字参数进行细粒度分析, 未有静态检测方法可以输出这些兼容性问题的细粒度原因. 此外, 除了 Python 生态, 研究人员还对 Java、Android 等生态系统展开了研究[5−15]. 然而, Python 作为动态语言, 与 Java 等静态语言存在显著的区别[16]. 例如 Python 语言的弱类型特性、方法参数可变的特性等, 导致对静态语言第三方库 API 的兼容性问题检测方法无法直接应用到 Python 第三方库中.

为探求 Python 兼容性问题的细粒度产生原因并实现检测兼容性问题的静态方法, 本文首先采用收集版本更新日志与运行回归测试相结合的方法来构建 Python 第三方库不兼容 API 的数据集, 从而弥补文献 [2,3] 因更新日志记录不完整导致的不兼容 API 数据集不全的问题. 最终, 本文在 flask 库[17]和 pandas 库[18]的 6 个版本对共收集到 108 个不兼容 API 对. 其中, 70 个 (64.8%) 不兼容 API 对通过运行回归测试收集, 其余 38 个 (35.2%) 不兼容 API 对通过阅读版本更新日志进行补充. 然后, 本文对 Python 第三方库 API 兼容性问题进行了实证研究, 通过分析 108 个不兼容 API 对, 总结了 API 兼容性问题的表现形式和产生原因. 表现形式是从第三方库调用者的角度进行分析, 而产生原因是从第三方库开发者的角度进行分析. 为提升数据的可信度, 收集不兼容 API 对以及根据表现形式和产生原因分类不兼容 API 对均由本文第一作者和第二作者完成. 当出现分歧时, 由本文第三作者组织小组讨论, 并决定最终结果. 与文献 [2,3] 不同的是, 本文增加了对 API 兼容性问题表现形式的分析, 同时在分析产生原因时增加了与异常有关的兼容性问题, 并对 Python 的 5 种参数进行了逐一分析.

在实证研究的基础上, 本文提出了一种基于静态分析技术的 Python 第三方库 API 兼容性问题检测方法. 方法的输入为 Python 第三方库在更新前后两个版本的源代码, 通过语句级别的数据流、控制流切片分析, 最终输出为句法层面的兼容性问题产生原因集合.

最后, 本文在 flask 库等 4 个常用 Python 第三方库的共计 12 个版本对上进行了实验评估. 评估结果证明本文方法具有较好的有效性、泛化性、时间性能、空间性能以及易用性. 在有效性上, 方法的精准率和召回率分别达到了 92.87% 和 93.79%. 在新构建的数据集上, 精准率和召回率分别为 88.65% 和 94.59%, 这表明方法具有良好的泛化性. 在时间性能和空间性能上, 检测这 12 个版本对的 5 243 个变更 API 共耗时 30 858.09 s, 共占用内存 1 494.24 MB, 平均每个版本对的检测大约需要耗时 42.86 min、占用 124.52 MB 的内存. 此外, 本文邀请了 10 名有经验的 Python 开发者在不兼容 API 上进行人工实验, 结果表明本文方法有良好的易用性.

本文主要贡献总结如下.

(1) 采用将更新日志与第三方库测试用例相结合的方法, 构建了 Python 第三方库 API 兼容性问题的数据集, 包括 flask 库和 pandas 库中 6 个版本对上的 108 个不兼容 API 对.

(2) 开展了关于 Python 第三方库 API 兼容性问题的实证研究, 总结归纳了 Python 第三方库 API 兼容性问题的表现形式和产生原因. 其中, 表现形式是从第三方库调用者的角度进行分析, 而产生原因是从第三方库开发者的角度进行分析.

(3) 提出了基于静态分析技术的 Python 第三方库 API 兼容性问题检测方法, 并通过实验评估证明了所提方法具有良好的有效性、泛化性、时间性能、空间性能以及易用性.

本文第 1 节介绍研究背景和相关工作. 第 2 节介绍 Python 第三方库 API 兼容性问题的实证研究. 第 3 节介绍基于静态分析的 Python 第三方库 API 兼容性问题检测方法. 第 4 节对方法的有效性、泛化性、时间性能、空间性能以及易用性进行评估. 第 5 节讨论方法面临的威胁和应用局限性. 最后总结全文.

1 研究背景与相关工作

1.1 研究背景

1.1.1 API 兼容性问题

如图 1 所示, 上层应用通过 API 调用的形式, 使用了第三方库的代码, 并且在原版本 v_o 上可以正确运行. 然而, 由于第三方库新功能需要、版本存在漏洞或缺陷等原因需要升级或降级版本, 上层应用开发者必然面对切换版本带来的 API 兼容性问题. 不兼容的 API 是指在上层 API 调用不变的情况下, 其在第三方库的原版本 v_o 上可以正确运行, 但在第三方库的其他版本 (v_b 或者 v_a) 运行异常终止或者产生与 v_o 版本不一致的结果[19]. 考虑到版本切换包括升级和降级, API 兼容性问题包括前向兼容性问题和后向兼容性问题. 从代码修改来看, API 兼容性问题是由原 API 调用链中的方法发生了修改导致, 例如语句的增加、删除、更新.

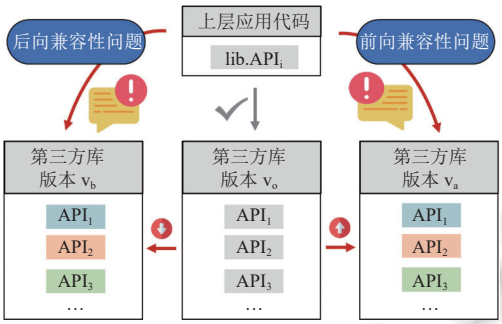

图 1 API 兼容性问题图示

1.1.2 Python 语言特性

Python 语言特性导致其对兼容性问题更为敏感, 本文从解释性语言、参数和返回值等 3 个方面讨论 Python 语言特性及其对兼容性问题的影响.

1.1.2.1 解释性语言

Python 为解释性语言, 在运行时逐行解释源代码, 将代码翻译成机器代码并执行. 与编译性语言在编译阶段就可以检查不同, 解释性语言只有通过动态运行到不兼容 API 时, 才可能发现兼容性问题导致的错误.

1.1.2.2 灵活的参数类别

Python 提供了 5 种类别的参数, 即位置参数、默认参数、可变参数、命名关键字参数以及关键字参数. 相较于其他编程语言, Python 的参数传递方式更加灵活. 在参数变化导致的不兼容 API 中, 可能使得原有 API 调用中的输入参数格式与新 API 声明的参数格式不一致, 引起 API 兼容性问题. 示例代码 1 展示了 Python 中 API 的 5

种参数. 表 1 展示了 5 种参数的使用特点.

示例代码 1. Python 中 API 的参数.

```
1 # a 和 b 是位置参数, c 和 d 是默认参数, args 是可变参数, key1 和 key2 是命名关键字参数, kwargs 是关键字参数
2 def foo (a:str, b, c=0, d=0, *args, key1, key2=0, **kwargs):
3     pass
```

表 1 Python 中 API 的 5 种参数及其特点

参数类别	参数个数	是否有默认值	是否需要传值	传值的顺序	传值的方式
位置参数	≥0	否	是	按定义顺序	直接传或 以键值对传入
默认参数	≥0	是	可选	若省略默认参数名称, 按定义顺序; 若以键值对形式传值, 无顺序要求	直接传或 以键值对传入
可变参数	0或1	否	可选	—	直接传
命名关键字参数	≥0	可选	若无默认值, 是; 否则, 可选	无顺序要求	以键值对传入
关键字参数	0或1	否	可选	无顺序要求	以键值对传入

位置参数按照定义的顺序传入值, 如 foo(‘a’, ‘b’, …); 也可以通过键值对的形式依次传入所有参数的值, 如 foo(a=‘a’, b=‘b’, …).

默认参数在未被显式传值时将使用默认值. 传入值时, 可直接按照定义的顺序传值, 如 foo(…, ‘c’, ‘d’, …). 需要注意的是, Python 优先赋值给顺序在前的默认参数. 例如 foo(…, ‘value’, …) 会将‘value’传递给默认参数 c, 而默认参数 d 为默认值 0. 此外, 还可以通过键值对的形式传递部分默认参数的值. 例如 foo(…, d=‘d’, …), 此时会将‘d’传递给默认参数 d, 而默认参数 c 为默认值 0. 这种方式使得参数传递更加灵活, 可以根据需要灵活赋值给需要提供值的参数.

可变参数允许传递任意数量的值, 这些值在 API 内部被组装成一个元组. 例如 foo(…, 0, 1, 2, …) 调用后 args 变量的值为元组 (0, 1, 2). 需要注意的是, 当 API 定义中有可变参数时, 也可以不传递任何值给它. 例如 foo(‘a’, ‘b’, ‘c’, ‘d’, key1=1) 调用后 args 变量为长度为 0 的元组. 可变参数使得 API 可以灵活接收不确定数量的值.

命名关键字参数可以指定默认值 (如示例代码 1 中的 key2), 也可以不指定默认值 (如示例代码 1 中的 key1). 在对其传递值时, 必须以键值对的形式进行, 例如 foo(…, key1=1, key2=2, …). 注意, 对于没有默认值的命名关键字参数, 必须提供值进行传递, 即 foo(‘a’, ‘b’, ‘c’, ‘d’) 和 foo(‘a’, ‘b’, ‘c’, ‘d’, key2=2) 会运行失败.

关键字参数允许传递任意数量的键值对, 这些键值对将在 API 内部被组装成一个字典. 例如 foo(…, key3=3, key4=4) 调用后 kwargs 为字典{‘key3’:3, ‘key4’:4}.

1.1.2.3 灵活的返回语句使用

在 Python 语法中, 返回语句 (return) 可缺省, 在缺省情况下默认返回 None. 此外, Python 支持返回语句中包含多个变量. 这种方式使得 API 可以灵活返回 API 处理结果.

1.2 相关工作

1.2.1 第三方库的演化

由于缺陷修复、代码重构、功能新增等, 第三方库代码会不断更新演化. 已有研究人员对此展开研究[2,20−23]. Zhang 等人[2]关注 Python 第三方库的演化, 发现 14 种与句法相关的演化规则, 覆盖第三方库的类、API 和字段. 其中, 与 API 有关的演化规则为 API 增删、必选参数增删、参数重排序、可选参数增删、参数默认值增删以及参数默认值改变, 后 3 种是 Python 语言特有的演化规则. Zhang 等人[20]关注深度学习库 TensorFlow 的演化, 发现随着版本更新, 增加的 API 逐渐增多, 删除的 API 逐渐减少, 未有 API 的名称发生改变, 而 API 移动、参数增删、参数重命名、返回值修改等变动的数量在不断波动. Dilhara 等人[21]关注上层应用开发者对机器学习第三方库更

新的反应, 研究发现在 22.04% 的提交中开发者会降低使用的第三方库版本, 在 41.52% 的提交中开发者会同时修改多个机器学习第三方库的使用. 此外, 研究人员还对 Android、Java 生态开展了研究[22,23]. Liu 等人[22]关注 Android 生态中未被第三方库官方文档记录的演化, 发现其中 64.59% 的变更涉及语义变化, 这些变化可能导致其上层应用发生错误. Dig 等人[23]关注 Java 生态中第三方库的演化, 发现超过 80% 的更新会改变程序结构但不改变程序行为, 即重构.

1.2.2 第三方库 API 的弃用

当第三方库 API 有缺陷、效率低时, 第三方库开发者通常会将这些 API 标记为弃用, 不鼓励上层应用开发者使用它们, 并且会在删除前保留一段时间. 如果开发者仍然调用这些被弃用的 API, 那么上层应用可能会发生错误. 已有研究人员对第三方库的 API 弃用展开研究[24−27]. Wang 等人[24]分析了 Python 第三方库开发者对弃用 API 的记录情况以及上层应用开发者对弃用 API 的维护策略. Vadlamani 等人[25]通过分析 Python 第三方库源代码中的 API 装饰器、硬编码警告和代码注释来识别弃用 API. Haryono 等人[26]设计了用于机器学习第三方库的 MLCatchUp 工具, 通过比较弃用 API 的签名和相关非弃用 API 的签名, 自动推断替代 API, 从而为上层应用开发者提供修改建议. Brito 等人[27]关注第三方库开发者弃用 API 时是否提供了可替换的 API 建议以及建议的质量如何. 研究发现, 66.7% 的 Java 弃用 API 和 77.8% 的 C#弃用 API 提供了替换 API 建议, 但这些建议的质量并不会随着第三方库的演化而提高.

1.2.3 第三方库 API 的兼容性问题

已有研究人员对第三方库 API 的兼容性问题展开研究[5−9]. Mostafa 等人[5]、Brito 等人[6]对 Java 第三方库 API 兼容性问题的普遍程度、触发条件、表现形式以及更改目的进行研究. Zhao 等人[7]关注 Android 第三方库 API 兼容性问题, 总结了与操作系统、特定设备以及内部回调有关的兼容性问题. 此外, Xia 等人[8]、Wei 等人[9]关注上层应用开发者对第三方库中不兼容 API 的应对方式. 然而, 兼容性问题的总结与语言特性直接相关, 现有对静态类型语言的兼容性问题研究工作无法直接应用到 Python 生态中. 针对 Python 生态, Zhang 等人[2]总结了 8 种与参数有关的兼容性问题, 分析时按照必选和可选对 Python 的 5 种参数进行划分. 在此基础上, Haryono 等人[3]通过对比机器学习领域的 Python 第三方库 API 源代码, 发现参数输入类型变化、参数输入值变化、参数输入格式变化带来的兼容性问题. 然而, 这些工作[2,3]在收集不兼容 API 时局限于第三方库的更新日志, 导致总结的不兼容 API 类型有遗漏, 忽略了不同参数类别和异常对兼容性问题的影响. 不同的是, 本文首先通过收集版本更新日志, 运行回归测试相结合的方法收集不兼容 API, 并针对 Python 的 5 种参数、异常、返回值分别进行分析, 总结了 API 兼容性问题表现形式和产生原因.

基于对第三方库 API 兼容性问题的认识, 研究人员设计了不兼容 API 的检测方法. 部分研究人员采用了动态分析方法[10,11]. Chen 等人[10]提出了一种跨客户端项目的动态检测方法, 方法首先在不同版本的第三方库上构建客户端项目, 然后通过运行客户端内置的测试代码检测第三方库 API 的兼容性问题. Sun 等人[11]设计了工具 JUnitTestGen, 该工具通过挖掘 API 用例生成检测兼容性问题的单元测试用例. 动态测试的缺点是覆盖范围有限, 因此, 更多的研究人员寻求用静态方法进行检测[12−15]. Zhang 等人[12]关注 Java 第三方库 API 在返回值上的兼容性问题, 提出一种遍历 API 调用链并测量 API 内部语义差异的静态检测方法. Mahmud 等人[13]、He 等人[14]、Yang 等人[15]在 Android 生态中的使用静态分析方法对 API 兼容性问题进行检测, 并取得良好效果. 然而, API 兼容性问题与语言特性直接相关, 现有对静态类型语言的检测工作无法直接应用到 Python 生态中. 针对 Python 生态, Du 等人[4]提出了一种动态静态相结合的方法检测 Python 第三方库的模块、类、API 和属性是否存在兼容性问题, 并按照 API 的断损程度分类. Zhang 等人[2]通过手动检查第三方库的更新日志建立关于 API 参数重命名的知识库, 并据此检测上层应用是否调用了存在参数重命名问题的 API . 不同的是, 本文旨在提出一种静态方法对 API 的兼容性进行检测, 输出细粒度的问题产生原因.

2 关于 Python 第三方库 API 兼容性问题的实证研究

为深入了解 Python 第三方库 API 兼容性问题的表现形式和产生原因, 本文将对其进行实证研究. 首先介绍实

证研究的设计, 然后介绍实证研究中分析数据的来源, 最后介绍实证研究的结果.

2.1 实证研究的设计

本文希望通过对 Python 第三方库 API 兼容性问题进行实证研究, 探索 Python 第三方库 API 兼容性问题的表现形式和产生原因.

RQ1: Python 第三方库 API 兼容性问题的表现形式有哪些? 它们呈现怎样的分布?

RQ2: Python 第三方库 API 兼容性问题的产生原因有哪些? 它们呈现怎样的分布?

首先, 在 flask 库和 pandas 库的 6 组版本对上通过更新日志与回归测试结合的方法收集存在兼容性问题的 API 版本对, 然后对不兼容 API 对逐一分析. 研究 RQ1 是从第三方库使用者的角度出发, 挖掘当其调用不兼容 API 时, 存在兼容性问题的 API 将以怎样的形式在上层应用代码中表现出不兼容, 即兼容性问题的表现形式. 研究 RQ2 是从第三方库开发者的角度出发, 挖掘其对 API 进行的何种更新操作导致 API 不兼容, 即兼容性问题的产生原因.

2.2 实证研究的数据来源

2.2.1 第三方库版本对的选择

第三方库的选择: 分别选择了在 Web 开发领域具有代表性的开源第三方库 flask 和在数据分析领域具有代表性的开源第三方库 pandas 进行分析.

第三方库版本对的选择: Python 第三方库的版本编号遵循语义化版本号 (semantic versioning), 形式为主版本.次版本. 补丁版本 (major.minor.patch). 本文对每个第三方库选择 3 组版本对来模拟软件版本演化和上层应用代码更新的适配过程, 包括跨 major 版本对、跨 minor 版本对和跨 patch 版本对. 选择最新的 Python 第三方库版本 major.minor.patch 作为锚定版本, 向前追溯选择版本作为比较, 具体选择方法为:

- 跨 major 版本对: 找到 major-1 为主版本的版本集合, 选择距离锚定版本最近版本作为跨 major 版本.
- 跨 minor 版本对: 找到 major 为主版本, minor-1 为次版本的版本集合, 选择距离锚定版本最近的版本作为跨 minor 版本.
- 跨 patch 版本对: 找到 major 为主版本, minor 为次版本的版本集合, 选择距离锚定版本最远的版本作为跨 patch 版本.

共选择 6 个版本对, 具体选择结果见表 2 所示.

表 2 第三方库及其版本对的选择集合

第三方库	应用领域	版本对v1→v2	
flask	Web开发	major	1.1.4→2.0.0
		minor	2.1.3→2.2.0
		patch	2.1.0→2.1.3
pandas	数据分析	major	0.25.3→1.0.0
		minor	1.4.4→1.5.0
		patch	1.4.0→1.4.4

2.2.2 不兼容 API 对的收集

对某第三方库版本对 v1→v2, API 兼容性问题指上层应用代码在前一版本 v1 上可以正常运行, 但在后一版本 v2 上运行异常终止或者产生不一致的结果. 本文收集的不兼容 API 对包括不兼容的函数以及不兼容的类成员函数. 本文采用更新日志与回归测试结合的方法收集存在兼容性问题的 API 对. 一方面, 这弥补了单元测试无法覆盖所有测试场景的缺陷. 一是可能没有针对某些 API 的测试用例; 二是可能某些 API 的兼容性问题需要特殊的输入才能被触发, 而这些输入并未在测试用例中被覆盖. 另一方面, 本方法可以有效避免更新日志质量不高对数据收集的干扰.

回归测试: 在第三方库 v2 版本的源代码上运行 v1 版本的测试用例. 如果某些测试用例运行出错, 说明它们所调用的 API 存在兼容性问题. 本文在表 2 中的 6 个版本对上共找到 2 290 个回归测试文件, 其中 58 个文件运行失

败, 发现 70 个不兼容 API 对.

第三方库更新日志: 分析第三方库 v2 版本的更新日志, 若发现日志中出现“减少参数”“增加处理”“不再支持”等关键字, 则认为对应的 API 对存在兼容性问题. 由于跨 patch 版本对并非紧密相连, 分析跨 patch 版本对时, 还会分析 v1 至 v2 中间版本的更新日志, 但仅记录更改保留至 v2 版本的 API . 例如, 除了分析 flask 库版本 2.0.0、2.2.0、2.1.3 的更新日志外, 还会分析版本 2.1.1、2.1.2 中保留至版本 2.1.3 的更新. 通过分析共计 11 个更新日志, 额外发现 38 个不兼容 API 对.

为提升数据的可信度, 不兼容 API 对的收集过程由本文第一作者和第二作者独立完成, 第三位作者负责解决分歧. 具体来说, 当一个不兼容 API 对仅由一位作者收集发现时, 前三位作者开会分析此 API 对的源代码、回归测试、更新日志, 并讨论决定此 API 是否存在兼容性问题. 同样地, 从表现形式 (RQ1) 和产生原因 (RQ2) 这两个角度对不兼容 API 对的分类过程也根据上述过程进行验证.

2.3 实证研究的结果: Python 第三方库 API 兼容性问题的表现形式 (RQ1)

从 Python 第三方库 API 调用者的角度看, 调用不兼容 API 会造成上层应用在两个版本上的运行行为发生改变. 通过对收集到的 108 个不兼容 API 对进行分析, 本文发现 Python 第三方库 API 兼容性问题的表现形式有 6 种, 如表 3、图 2 所示.

表 3 Python 第三方库 API 兼容性问题的表现形式

序号	兼容性问题表现形式	总计 (个)	flask库不兼容API数量 (个)	pandas库不兼容API数量 (个)
S1	输入参数格式错误	28	6	22
S2	非法输入参数值	43	1	42
S3	非法输入参数类型	3	1	2
S4	输出变量值错误	40	8	32
S5	输出变量类型错误	1	0	1
S6	新异常无法捕获错误	6	3	3
	总计	108	17	91

注: 由于一个不兼容API在不同调用中表现形式不同, 总计数据小于各项加和

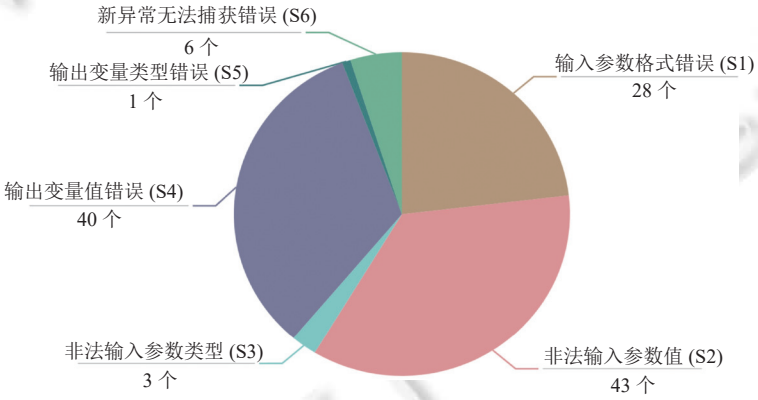

图 2 Python 第三方库 API 兼容性问题的表现形式

28 个 API 的兼容性问题表现为输入参数格式错误 (S1), 即上层调用 API 的格式与第三方库中 API 的签名不匹配. 此时, Python 默认抛出异常 TypeError.

例 1: 示例代码 2 中, 默认参数 encoding 被删除, 导致运行 cls.to_stata(‘fname’, convert_dates=‘convert _dates’, write_index=‘write_index’, encoding=‘latin-1’, byteorder=‘byteorder’) 时抛出异常 TypeError: to_stata() got an unexpected keyword argument ‘encoding’. 此外, 由于必选参数 fname 被重命名为 path, 上层调用 cls.to_stata (fname=‘fname’) 时抛出异常 TypeError: to_stata() got an unexpected keyword argument ‘fname’.

示例代码 2. 输入参数格式错误 (S1).

```
# 版本变更: pandas0.25.3 → pandas1.0.0
1  –  def to_stata(self, fname, convert_dates=None, write_index=True, encoding='latin-1', byteorder=None):
2  +  def to_stata(self, path, convert_dates=None, write_index=True, byteorder=None):
3           ...
4  –        writer = statawriter(fname, ...)
5  +        writer = statawriter(path, ...)
6           writer.write_file()
# 受兼容性问题影响的上层应用代码
1       cls.to_stata('fname', convert_dates='convert_dates', write_index='write_index', encoding='latin-1',
                     byteorder='byteorder')
2       cls.to_stata(fname='fname')
```

43 个 API 的兼容性问题表现为非法输入参数值 (S2), 即上层调用 API 时输入的参数值不合要求, 如示例代码 3. 此时, 抛出的异常类型由第三方库源代码中的 raise 语句决定. 其中, 90.70% 的 API 抛出异常 ValueError, 其余 9.30% 的 API 抛出异常 AssertionError、IndexingError、TypeError 或 KeyError.

示例代码 3. 非法输入参数值 (S2).

```
# 版本变更: pandas1.4.4 → pandas1.5.0
1       def asof(self, where, subset=None):
2  –        if not self.index.is_monotonic:
3  +        if not self.index.is_monotonic_increasing:
4               raise ValueError('asof requires a sorted index')
5           ...
# 受兼容性问题影响的上层应用代码
1       cls.index.is_monotonic = None
2       cls.asof('where')
```

3 个 API 的兼容性问题表现为非法输入参数类型 (S3), 即上层调用 API 时输入的参数类型不合要求. 此时, 抛出的异常类型由第三方库源代码中的 raise 语句决定. 本文收集到的 API 所抛出异常均为 TypeError.

40 个 API 的兼容性问题表现为输出变量值错误 (S4), 即当上层调用 API 的输入不变时, API 返回了不同的输出值.

1 个 API 的兼容性问题表现为输出变量类型错误 (S5), 即当上层调用 API 的输入不变时, API 返回了不同的输出类型.

例 2: 示例代码 4 中, 若第 4 行的条件在 v1 版本 pandas1.4.4 成立, 则 API 会返回 None. 然而在 v2 版本 pandas1.5.0 中, 同样的情况会返回一个 DataFrame 类型的对象.

6 个 API 的兼容性问题表现为新异常无法捕获错误 (S6), 即上层应用在 try-except 结构中调用 API 时, v2 版本抛出的异常无法被正常捕获, 导致上层应用代码的执行路径发生更改.

示例代码 4. 输出变量类型错误 (S5).

```
# 版本变更: pandas1.4.4 → pandas1.5.0
1       def read(self, nrows=None):
```

```
2          ...
3  –       if self._current_row_in_file_index >= self.row_count:
4  +       if nrows > 0 and self._current_row_in_file_index >= self.row_count:
5  –           return None
6  +           return DataFrame()
```

例 3: 示例代码 5 中, 当 ctx 为 None 时, v1 版本 pandas1.4.4 中 render_template 运行至第 5 行时会抛出异常 AttributeError, 这导致上层应用执行第 3、4 行的代码块. 然而, v2 版本 pandas1.5.0 中新增的第 3、4 行导致相同情况下会抛出异常 RuntimeError 而非 AttributeError, 导致上层应用代码中的 except 语句无法捕获异常, 进而异常终止.

示例代码 5. 新异常无法捕获错误 (S6).

```
# 版本变更: pandas1.4.4 → pandas1.5.0
1      def render_template(template_name_or_list, **context):
2          ctx = _app_ctx_stack.top
3  +       if ctx is None:
4  +           raise RuntimeError('This function can only be used when an application context is active.')
5          ctx.app.update_template_context(context)
6          return _render(ctx.app.jinja_env.get_or_select_template(template_name_or_list), context, ctx.app)
# 受兼容性问题影响的上层应用代码
1      try:
2          render_template(template_name_or_list)
3      except AttributeError:
4          pass
```

RQ1 结论: 从 Python 第三方库 API 调用者的角度看, API 兼容性问题的表现形式有 6 种, 分别为输入参数格式错误 (S1/25.93%)、非法输入参数值 (S2/39.81%)、非法输入参数类型 (S3/2.78%)、输出变量值错误 (S4/37.04%)、输出变量类型错误 (S5/0.93%) 和新异常无法捕获错误 (S6/5.56%), 它们最终会反映为上层应用的非正常运行. 其中, 非法输入参数值 (S2) 和输出变量值错误 (S4) 占比最高.

2.4 实证研究的结果: Python 第三方库 API 兼容性问题的产生原因 (RQ2)

通过对收集到的 108 个有兼容性问题的 API 对进行分析, 本文发现 Python 第三方库 API 兼容性问题的产生原因有 7 种, 分布如表 4、图 3 所示. 其中 CI1–CI5 与参数有关, CI6 与异常有关, CI7 与返回值有关.

表 4 Python 第三方库 API 兼容性问题的产生原因

序号	兼容性问题产生原因	总计 (个)	flask库不兼容API数量 (个)	pandas库不兼容API数量 (个)
CI1	参数的增删	26	5	21
CI2	关键字参数的键增删	1	0	1
CI3	参数的重命名	3	1	2
CI4	参数默认值的改变	11	2	9
CI5	参数的范围改变	45	2	43
CI6	抛出不同的异常	6	3	3
CI7	输出不同的返回值	31	7	24
	总计	108	17	91

注: 由于一个API可能由多种原因导致不兼容, 总计数据小于各项加和

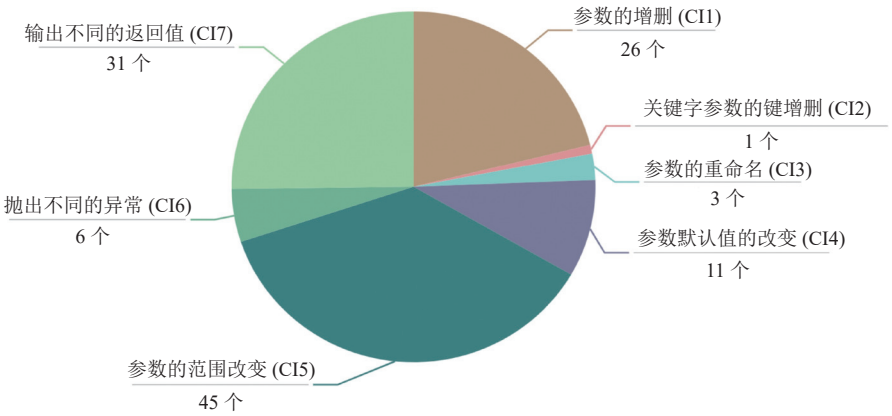

图 3 Python 第三方库 API 兼容性问题的产生原因

从 Python 的语法来看, 第三方库 API 兼容性问题的产生原因还应该包括参数种类的改变和参数位置的改变. 参数种类的改变指从 Python 5 种参数中的任意一种变成另外一种, 如从默认参数变为位置参数. 参数位置的改变有两种, 一种是由参数的增删 (CI1) 导致的副作用, 如示例代码 2 中参数 byteorder 的位置从 6 变为 5; 另一种是两个参数的位置互换. 本文收集到的 108 个不兼容 API 对中未发现由参数种类改变、两个参数位置互换导致的兼容性问题.

2.4.1 与参数有关的兼容性问题 (CI1–CI5)

26 个 API 的兼容性问题由参数的增删导致 (CI1). 参数的增删包括两个方面的内容, 一是参数的增加, 二是参数的删除, 它们导致的兼容性问题均表现为输入参数格式错误 (S1).

(1) 参数的增加: 表 5 展示了 Python 第三方库 API 中 5 种参数的增加对兼容性的影响.

表 5 由于参数增加原因导致的 Python 第三方库 API 兼容性问题

参数类别	增加该类参数是否会造成兼容性问题	示例代码	
		第三方库API代码	受兼容性问题影响的上层应用代码
位置参数	一定是	− def foo(a): + def foo(a, b): pass	foo(‘a’) foo(a=‘a’)
默认参数	可能是	− def foo(a=0, c=2): + def foo(a=0, b=1, c=2): pass	foo(‘a’, ‘c’)
可变参数	否		
命名关键字参数	可能是	− def foo(*, key1=1): + def foo(*, key1=1, key2): pass	foo( ) foo(key1=‘1’)
关键字参数	否		

位置参数: 因为所有位置参数都必须传入值, 所以若第三方库 API 有新增的位置参数, 则上层应用会异常终止.

默认参数: 当上层调用给默认参数传递值的数量小于第三方库 API 默认参数的数量时, Python 默认按照参数定义的顺序进行赋值. 因此, 如果在新增默认参数之后还存在其他默认参数, 可能会导致值的传递发生错位. 以表 5 中的示例代码为例, 在 v1 版本中, 值‘c’将被赋给参数 c. 而在 v2 版本中, 值‘c’将被赋给参数 b, 参数 c 将使用默认值 2.

命名关键字参数: 因为所有无默认值的命名关键字参数都必须传入值, 所以若第三方库 API 有新增的无默认值的命名关键字参数, 则上层应用会异常终止.

可变参数和关键字参数: 可变参数可以接受任意数量的值, 包括不传递任何值, 而关键字参数可以接受任意数量的键值对, 包括不传递任何键值对. 因此, 当第三方库 API 新增可变参数和关键字参数时, 不会对上层应用造成破坏性影响.

(2) 参数的删除: 如表 6 所示, 无论何种类别的参数, 一旦发生删除操作, 都会导致兼容性问题. 这是因为删除参数会导致上层调用中原本用于赋值给被删除参数的值冗余, 从而影响代码正常执行.

表 6 由于参数删除原因导致的 Python 第三方库 API 兼容性问题

参数类别	删除该类参数是否会造成兼容性问题	示例代码	
		第三方库API代码	受兼容性问题影响的上层应用代码
位置参数	一定是	− def foo(a, b): + def foo(a): pass	foo(‘a’, ‘b’) foo(a=‘a’, b=‘b’)
默认参数	一定是	− def foo(a=0, b=1): + def foo(a=0): pass	foo(‘a’, ‘b’) foo(a=‘a’, b=‘b’) foo(b=‘b’)
可变参数	一定是	− def foo(*arg): + def foo(): pass	foo(0, 1, 2)
命名关键字参数	一定是	− def foo(*, key1=1, key2): + def foo(*): pass	foo(key1=‘1’, key2=‘2’) foo(key2=‘2’)
关键字参数	一定是	− def foo(**kwargs): + def foo(): pass	foo(key=‘key’)

1 个 API 的兼容性问题由关键字参数的键增删 (CI2) 导致. 关键字参数的键增删指关键字参数所需传入键的增加或删除, 此种原因导致的兼容性问题表现为非法输入参数值 (S2). 当 API 有关键字参数时, API 内部会对关键字参数的各个键值对进行读写. 当 API 内部需要对键 key 进行读取, 但 API 调用者未对键 key 传值时, API 会抛出异常 KeyError. 若 API 内部不允许传入键 key, 但 API 调用者对键 key 传值时, API 也会抛出异常.

例 4: 示例代码 6 中, 在 v2 版本 pandas1.0.0 中, 新增的第 3、4 行不再允许关键字参数 kwargs 被传入键 labels.

示例代码 6. 关键字参数的键增删 (CI2).

```
# 版本变更: pandas0.25.3 → pandas1.0.0
1      def copy(self, names=None, dtype=None, levels=None, codes=None, deep=False, _set_identity=False,
           **kwargs):
2          name = kwargs.get(‘name’)
3  +       if ‘labels’ in kwargs:
4  +           raise TypeError(“ ‘labels’ argument has been removed; use ‘codes’ instead”)
5          ...
```

3 个 API 的兼容性问题由参数的重命名 (CI3) 导致. 由于在调用 API 时可以使用键值对的形式传递参数, 若第三方库 API 发生参数的重命名, 则会导致上层应用无法运行. 此种原因导致的兼容性问题表现为输入参数格式错误 (S1).

11 个 API 的兼容性问题由参数默认值的改变 (CI4) 导致. 参数默认值的改变是默认参数和命名关键字参数特有的兼容性问题. 若它们的参数默认值发生改变, 则 API 内部与这些参数相关的行为也会随之改变, 进而可能导致 API 的输出内容发生变化, 从而影响上层应用的正常运行. 此种原因导致的兼容性问题表现为输出变量值错误 (S4).

45 个 API 的兼容性问题由参数的范围改变 (CI5) 导致. 参数的范围改变指第三方库 API 更新后对输入参数

的要求范围发生变化, 例如示例代码 3. 此种原因导致的兼容性问题表现为非法输入参数值 (S2) 或非法输入参数类型 (S3).

2.4.2 与异常有关的兼容性问题 (CI6)

6 个 API 的兼容性问题由抛出不同的异常 (CI6) 导致. 抛出不同的异常指第三方库 API 在相同条件下会抛出不同的异常, 例如示例代码 5. 此种原因导致的兼容性问题表现为新异常无法捕获错误 (S6).

2.4.3 与返回值有关的兼容性问题 (CI7)

31 个 API 的兼容性问题由输出不同的返回值 (CI7) 导致. 输出不同的返回值是指当相同输入的情况下, API 输出不同的返回值, 例如示例代码 4. 此种原因导致的兼容性问题表现为输出变量值错误 (S4) 或输出变量类型错误 (S5).

RQ2 结论: 从 Python 第三方库 API 开发者的角度看, API 兼容性问题的产生原因有 7 种, 分别是参数的增删 (CI1/24.07%)、关键字参数的键增删 (CI2/0.93%)、参数的重命名 (CI3/2.78%)、参数默认值的改变 (CI4/10.19%)、参数的范围改变 (CI5/41.67%)、抛出不同的异常 (CI6/5.56%) 和输出不同的返回值 (CI7/28.70%). 其中 CI1–CI5 与参数有关, CI6 与异常有关, CI7 与返回值有关.

3 基于静态分析的 Python 第三方库 API 兼容性问题检测方法

基于实证研究中发现的 Python 第三方库 API 兼容性问题的 7 种产生原因 (CI1–CI7), 本文提出了一种基于静态分析的检测方法, 用于检测 Python 第三方库 API 的兼容性问题. 方法的核心思想是在语句级别切片分析与参数、异常、返回值有关的数据流、控制流, 进而分别检测与参数有关的兼容性问题 (CI1–CI5)、与异常有关的兼容性问题 (CI6)、与返回值有关的兼容性问题 (CI7). 图 4 展示了方法的整体框架. 该方法的输入包括第三方库 v1 版本和 v2 版本的源代码, 输出为兼容性问题的产生原因集合 *CI*. 方法主要分为 3 步, 包括预处理、代码信息提取和兼容性问题检测.

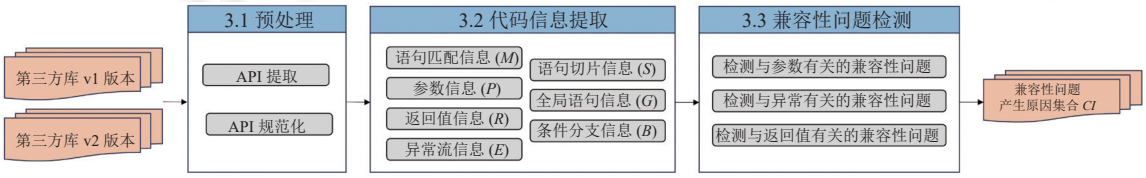

图 4 基于静态分析的 Python 第三方库兼容性问题检测方法的框架示意图

3.1 预处理

3.1.1 API 提取

首先逐个提取 API 源代码和 API 所在文件的全局语句. 作为解释性语言, Python 拥有脚本语言的特性, 即代码语句可以在函数体外声明和运行. 这部分语句具有声明和操作全局变量、导入模块、导入模块中特定对象等作用. 因此, 全局语句会对 API 的行为产生影响, 检测时需要将它们与 API 源代码一同提取出来. 记 v1 版本和 v2 版本中提取的 API 源代码分别为 $APIcode_1$ 和 $APIcode_2$. 本方法仅检测保留在同一路径中的同名 API, 并且不涉及过程间分析, 所以本步骤中仅提取路径和名称均不变的保留 API. 具体来说, 对于保留的函数, 本步骤仅提取路径、函数名均不变的函数; 对于保留的类成员函数, 本步骤仅提取路径、类名、函数名均不变的类成员函数.

3.1.2 API 规范化

源代码通常会包含代码注释、类型注释、空白行与代码换行, 这些元素对 API 的行为没有直接影响. 为提高方法后续分析的准确性, 对 $APIcode_1$ 和 $APIcode_2$ 进行规范化处理, 将代码注释、类型注释、空白行和代码换行统一删除. 规范化时先将源代码转化成抽象语法树, 再从抽象语法树反向解析出源代码, 反向解析时忽略代码注释、类型注释、空白行和代码换行. 若发现规范后的 $APIcode_1$ 和 $APIcode_2$ 完全一致, 即 API 未发生变更, 则认为该

API 不存在兼容性问题, 不再进行后续检测.

3.2 代码信息提取

代码信息提取从预处理后的规范代码 $APIcode_1$ 和 $APIcode_2$ 中抽取语句匹配信息 (M)、参数信息 (P)、返回值信息 (R)、异常流信息 (E)、切片信息 (S)、全局语句信息 (G) 以及条件分支信息 (B). 给定 $APIcode_1$, 抽取的代码信息可分别表示为 M_1, P_1, R_1, E_1, S_1, G_1, B_1. 除提取全局语句信息 (G) 外, 代码信息提取步骤不涉及 API 外部代码信息, 仅分析 API 内部的源代码. 换句话说, 代码信息提取步骤仅涉及过程内分析, 而不涉及过程间分析.

3.2.1 语句匹配信息 (M)

本方法使用 GumTree 工具[28]在两个版本中同一个 API 的抽象语法树上进行语句级别的匹配, 得到的语句匹配信息 M 可以表示为一个四元组<*add*, *del*, *change*, *unchange*>. 其中, *add*、*del*、*change*、*unchange* 分别表示新增语句集合、删减语句集合、更新语句集合以及未变语句集合. 本文用 s_i 表示源代码 *APIcode* 中存在的语句, i 表示其所在的行号. 特别地, 对于 if、for 等控制语句, 方法根据其逻辑条件进行匹配并记录对应的行号. 集合 *M.change*、*M.unchange* 中的元素可以表示为二元组<s_a, s_b>, 表明 v1 版本中的语句 s_a 和 v2 版本中的语句 s_b 存在匹配关系. 记 M_1.*change* 和 M_2.*change* 表示 v1 版本和 v2 版本中的更新语句集合, M_1.*unchange* 和 M_2.*unchange* 表示 v1 版本和 v2 版本中的未变语句集合.

3.2.2 参数信息 (P)

参数信息包括参数顺序、参数名称、参数类别和参数默认值. 参数信息 P 可以表示为五元组<*pst*, *dft*, *abt*, *key*, *kwargs*>, 分别记录位置参数、默认参数、可变参数、命名关键字参数以及关键字参数的信息. 此外, 用 *Name*(P) 记录 API 所有参数的名称, 同理用 *Name*(*P.pst*) 记录 API 所有位置参数的名称.

(1) *pst* 集合记录位置参数的名称与顺序间的对应关系, 由 Map 类型 *name*: *idx* 组成, 从字符串类型映射到整数类型.

(2) *dft* 集合记录默认参数的名称与顺序、默认值间的对应关系, 由 Map 类型 *name*: <*idx*, *value*> 组成, 从字符串类型映射到一个二元组.

(3) *abt* 变量记录可变参数的信息, 若 API 有可变参数, *P.abt* 变量记录可变参数的名称, 否则记 *P.abt* 变量为 NA.

(4) *key* 集合记录命名关键字参数的名称与默认值间的对应关系, 由 Map 类型 *name*: *value* 组成. 特别地, 若命名关键字参数无默认值, *value* 为 NA.

(5) *kwargs* 变量记录关键字参数的信息, 若 API 有关键字参数, *P.kwargs* 变量记录关键字参数的名称, 否则记 *P.kwargs* 变量为 NA.

3.2.3 返回值信息 (R)

返回值信息 R 是包含了所有返回语句 (return) 的集合.

3.2.4 异常流信息 (E)

异常流信息由两部分组成, 分别是抛出异常信息 *E.R* 和捕获异常信息 *E.C*.

(1) 抛出异常信息 *E.R*: 集合 *E.R* 记录抛出异常语句 s_i 与抛出异常类型、抛出异常条件的对应关系, 由 Map 类型 s_i: <*exception*, *condition*> 组成. 元素 *exception* 是记录 s_i 抛出异常类型的变量, 元素 *condition* 是记录 s_i 抛出异常的触发条件的变量. 注意, Python 中的异常分为解释器自动抛出和开发者利用 raise 语句主动抛出两种, 均用 *E.R* 记录.

(2) 捕获异常信息 *E.C*: 集合 *E.C* 记录 try-except 结构进行异常捕获的信息, 由 Map 类型 s_i: <*TryStmts*, *AllStmts*> 组成. s_i 记录 try 语句, 元素 *TryStmts* 是记录 try 语句所处理代码块的语句集合, 元素 *AllStmts* 是记录 try-except 结构中所有语句的集合.

3.2.5 切片信息 (S)

给定参数信息 P 和返回值信息 R, 本方法使用工具[29]对代码进行切片 (slicing), 抽取与 P、R 相关的切片语句. 切片分为前向切片 (forward slicing) 与后向切片 (backward slicing). 具体来说, 本方法将预处理后的 API 代码输入给工具[29], 并指定切片的方向以及切片入口, 则工具[29]会输出与指定参数或返回值有关的控制流、数据流语句

集合.

(1) 有关参数的前向切片信息 *S.FP*: 对每个参数 p 进行前向切片, 抽取与参数 p 存在控制流和数据流关系的语句集合, 记为 *S.FP*[p].

(2) 有关返回值的后向切片信息 *S.BR*: 每个 return 语句 s_i 进行后向切片, 抽取与 s_i 存在控制流和数据流关系的语句集合 (含语句 s_i), 记为 *S.BR*[s_i].

3.2.6 全局语句信息 (*G*)

全局语句信息用集合 *G* 记录源代码 *APIcode* 中的全局语句.

3.2.7 条件分支信息 (*B*)

Python 通过 if-elif-else 结构控制条件分支的跳转, 条件分支信息记录条件分支语句及其跳转条件, 由 Map 类型 s_i: <*condition*, *Stmts*> 构成. 元素 s_i 记录 if 语句或 elif 语句或 else 语句, 元素 *condition* 记录 s_i 的成立条件, 元素 *Stmts* 是记录该条件成立后此条件分支结构中所运行代码块的语句集合.

例 5: 示例代码 5 中, 语句匹配信息为 $M.add=\{s_3, s_4\}$, $M.del=M.change=\{\}$, $M.unchange=\{<s_2, s_2>, <s_5, s_5>, <s_6, s_6>\}$; 参数信息为 $P_1.pst=P_2.pst=$ {'template_name_or_list': 1}, $P_1.abt=P_2.abt$=NA, $P_1.kwargs=P_2.kwargs$= 'context', $P_1.dft=P_2.dft=P_1.key=P_2.key=\{\}$, 其中 $P_1.pst$['template_name_or_list'].*idx*=1; 返回值信息为 $R_1=R_2=\{s_6\}$; 异常流信息为 $E_1.C=E_2.C=\{\}$, $E_1.R=\{s_5$: <AttributeError, AttributeNotFound(ctx, app) OR AttributeNotFound(ctx.app, update_template_context)>}, $E_2.R=\{s_4$: <RuntimeError, ctx is None>}, 其中 $E_2.R[s_4]$.exception=RuntimeError, $E_2.R[s_4]$.condition=ctx is None; 切片信息为 $S_1.FP$['template_name_or_list']=$S_2.FP$['template_name_or_list']= $\{s_6\}$, $S_1.BR[s_6]=\{s_2, s_5, s_6\}$, $S_2.BR[s_6]=\{s_2, s_3, s_4, s_5, s_6\}$; 全局语句信息为 $G_1=G_2=\{\}$; 条件分支信息为 $B_1=\{\}$, $B_2=\{s_3$: <ctx is None, $\{s_4\}$>}.

例 6: 示例代码 7 中, 异常流信息为 $E_1.R=E_2.R=\{\}$, $E_1.C=\{\}$, $E_2.C=\{s_7$: <$\{s_8\}$, $\{s_7, s_8, s_9, s_{10}\}$>}, 其中 $E_2.C[s_7]$.*TryStmts*=$\{s_8\}$, $E_2.C[s_7]$.*AllStmts*=$\{s_7, s_8, s_9, s_{10}\}$; 全局语句信息为 $G_1=\{s_1, s_2, s_3, s_4\}$, $G_2=\{\}$.

示例代码 7. 全局语句移动产生的良性变化语句.

```
# 版本变更: flask2.1.0 → flask2.1.3
1  –  try:
2  –      import dotenv
3  –  except ImportError:
4  –      dotenv = None
5     def load_dotenv(path=None):
6  –      if dotenv is None:
7  +      try:
8  +          import dotenv
9  +      except ImportError:
10             return False
```

3.3 兼容性问题检测

兼容性问题检测包括检测与参数有关的兼容性问题、检测与异常有关的兼容性问题以及检测与返回值有关的兼容性问题. 算法 1、算法 2、算法 3 分别检测与参数有关、与异常有关、与返回值有关的兼容性问题, 最终合并后得到被检测 API 中可能存在的兼容性问题产生原因集合 *CI*.

3.3.1 检测与参数有关的兼容性问题 (CI1–CI5)

检测与参数有关的兼容性问题如算法 1 所示, 其输入为语句匹配信息 *M*、参数信息 *P*、异常流信息 *E* 和切片信息 *S*, 输出为兼容性问题产生原因集合 *CI*, 包括检测参数的增删 (CI1)、关键字参数的键增删 (CI2)、参数的重命名 (CI3)、参数默认值的改变 (CI4) 以及参数的范围改变 (CI5).

算法 1. 检测与参数有关的兼容性问题.

输入: M, P, E, S;

输出: CI.

Function DetectParam:

1. CI = {}
2. P_D = $Name(P_1)$–$Name(P_2)$, P_A = $Name(P_2)$–$Name(P_1)$, P_M = $Name(P_1)$∩$Name(P_2)$
3. P_D, P_A, P_M, IsRename = MATCH(P_D, P_A, P_M, $S_1.FP$, $S_2.FP$, $M.change$, $M.unchange$)
4. CI ← {CI3} **if** IsRename **is** True
5. CI ← {CI1} **if** Len(P_D)>0 **or** Len(P_A∩$Name(P_2.pst)$)>0
6. max_index =**MAX(**$P_2.dft$[$param_2$].idx **for each** $param_2$ **in** P_M∩$Name(P_2.dft)$**)**
7. **for each** $param_2$ **in** P_A∩$Name(P_2.dft)$ **then**:
8. CI ← {CI1} **if** $P_2.dft$[$param_2$].idx < max_index
9. **for each** $param_2$ **in** P_A∩$Name(P_2.key)$ **then**:
10. CI ← {CI1} **if** $P_2.key$[$param_2$].$value$ **is** NA
11. KY1, KN1 = TravelExceptionAboutKwargs($P_1.kwargs$, $S_1.FP$, E_1)
12. KY2, KN2 = TravelExceptionAboutKwargs($P_2.kwargs$, $S_2.FP$, E_2)
13. CI ← {CI2} **if** KY1 != KY2 **or** KN1 != KN2
14. **for each** param **in** $Name(P_1.dft)$∩$Name(P_2.dft)$ **then**:
15. CI ← {CI4} **if** $P_1.dft$[param].$value$!= $P_2.dft$[param].$value$
16. **for each** param **in** $Name(P_1.key)$∩$Name(P_2.key)$ **then**:
17. CI ← {CI4} **if** $P_1.key$[param].$value$!= $P_2.key$[param].$value$
18. **for each** s **in** $E_2.R$ **then**:
19. **if** Raise($E_2.R$[s].$condition$, E_1) **is** False **then**:
20. **for each** $param_2$ **in** P_M **then**:
21. CI ← {CI5} **if** s **in** $S_2.FP$[$param_2$]
22. **return** CI

Function End

算法 1 中, 第 2 行通过比较参数名列表 $Name(P_1)$ 和 $Name(P_2)$, 获得参数增删的初步结果 P_D、P_A 和 P_M, 它们分别表示增加、删除和保留的参数集合. 第 3、4 行检测参数的重命名 (CI3). 具体来说, 第 4 行的 MATCH 逐个对比 v1 版本 P_D 中的删除参数和 v2 版本 P_A 中的增加参数, 通过分析它们的前向切片信息 $S.FP$ 和语句匹配信息 M, 判断 API 是否发生重命名并更新 P_D、P_A 和 P_M. 若对某被删参数 $param_1$ 和某新增参数 $param_2$, 集合 $S_1.FP$[$param_1$]∩($M_1.change$∪$M_1.unchange$) 中的任意语句 s_a 在二元组<s_a, s_b> 中对应的语句 s_b, 均满足 s_b∈$S_2.FP$[$param_2$], 即被删参数 $param_1$ 在 v1 版本中的前向切片保留语句和新增参数 $param_2$ 在 v2 版本中的前向切片保留语句之间均存在对应关系, 则认为 v1 版本中的参数 $param_1$ 在 v2 版本中被重命名为 $param_2$, 并返回 IsRename 为 True. 同时, 更新 P_D=P_D–{$param_1$}, P_A=P_A–{$param_2$}, P_M=P_M∪ {$param_1$, $param_2$}.

第 5–10 行检测参数的增删 (CI1). 其中, 第 5 行的 Len(Set) 表示集合 Set 中的元素个数. 如果仍存在被删除的参数 (第 5 行), 则认为该 API 存在参数删除导致的兼容性问题. 如果增加位置参数 (第 5 行), 或者增加的默认参数后有保留的默认参数 (第 6–8 行), 或者增加无默认值的命名关键字参数 (第 9、10 行), 则认为该 API 存在参数增加导致的兼容性问题.

第 11–13 行检测关键字参数的键增删 (CI2). KY 和 KN 分别表示关键字参数必须传入的关键字集合和不允许

被传入的关键字集合. 第 11、12 行的 TravelExceptionAboutKwargs 表示遍历与关键字参数有关的异常流信息, 若 *P.kwargs* 没有关键字 key 时 API 会抛出异常, 则将 key 存入 KY; 若 *P.kwargs* 有关键字 key 时, API 会抛出异常, 则将 key 存入 KN. 若 v1 版本和 v2 版本的 KY 或 KN 存在差异, 则该 API 存在关键字参数的键增删 (CI2).

第 18–21 行检测参数的范围改变 (CI5). 其中, 第 19 行的 Raise(*condition*, *E*) 表示通过分析 API 的异常流信息 *E*, 判断条件 *condition* 成立时 API 是否会抛出异常. 如果会抛出异常, Raise(*condition*, *E*) 返回相应的异常类型; 否则返回 False. 如果某异常仅在 v2 版本抛出而不在 v1 版本抛出 (第 19 行), 并且该异常的触发条件与某参数的输入有关 (第 20、21 行), 则该 API 存在参数的范围改变 (CI5).

例 7: 示例代码 2 中, 算法 1 第 2 行 P_D、P_A 和 P_M 分别初始化为{‘fname’, ‘encoding’}、{‘path’}和{‘self’, ‘convert_dates’, ‘write_index’, ‘byteorder’}. 由 $M.change \cup M.unchange$={<s_3, s_3>, <s_4, s_5>, <s_6, s_6>}, $S_1.FP$[‘fname’]={s_4, s_6}, $S_2.FP$[‘path’]={s_5, s_6}, 可知参数 fname 和参数 path 前向切片中的保留语句之间存在对应关系. 因此, 第 3、4 行认为参数 fname 被重命名为 path, 并更新 P_D 为{‘encoding’}, P_A 为{}, *CI* 为{CI3}. 由于 Len(P_D)>0, 第 5 行更新 *CI* 为{CI1, CI3}.

例 8: 示例代码 6 中, $E_1.R$={}, $E_2.R$={s_4: <TypeError, ‘labels’ in kwargs>}, 所以算法 1 第 11、12 行中 KN2={‘labels’}, KY1=KY2=KN1={}. 由于 KN1 与 KN2 不相等, 更新 *CI* 为{CI2}.

例 9: 示例代码 3 中, $S_2.FP$[‘self’]={s_3, s_4}, $E_1.R$={s_4: <ValueError, not self.index.is_monotonic>}, $E_2.R$={s_4: <ValueError, not self.index.is_monotonic_increasing>}. 当第 18 行中 s 为 s_4 时, Raise(not self.index.is_monotonic_increasing, E_1) 为 False, 而 $s_4 \in S_2.FP$[‘self’], 所以参数 self 的范围发生改变, 更新 *CI* 为{CI5}.

3.3.2 检测与异常有关的兼容性问题 (CI6)

检测与异常有关兼容性问题如算法 2 所示, 其输入为异常流信息 *E*, 输出为兼容性问题产生原因集合 *CI*, 包括抛出不同的异常 (CI6).

算法 2. 检测与异常有关的兼容性问题.

输入: *E*;
输出: *CI*.

Function DetectException
1. **for each** s **in** $E_2.R$ **then**:
2. exception$_1$ = Raise($E_2.R$[s].*condition*, E_1)
3. *CI* ← {CI6} **if** exception$_1$!= False **and** exception$_1$!= $E_2.R$[s].*exception*
4. **return** *CI*

Function End

算法 2 展示了检测抛出不同的异常 (CI6) 的具体流程. 如果相同条件 $E_2.R$[s].*condition* 下, v1 版本和 v2 版本均抛出异常, 但抛出的异常类型不同, 则认为该 API 存在抛出不同的异常 (CI6).

例 10: 示例代码 5 中, $E_1.R$={s_5: <AttributeError, AttributeNotFound(ctx, app) OR AttributeNotFound(ctx.app, update_template_context)>}, $E_2.R$={s_4: <RuntimeError, ctx is None>}. 当 ctx is None 条件在 v1 版本成立时, 会在 s_5 抛出异常 AttributeError, 即 exception$_1$ 为 AttributeError, 这导致算法 2 第 3 行的条件成立, 更新 *CI* 为{CI6}.

3.3.3 检测与返回值有关的兼容性问题 (CI7)

检测与返回值有关的兼容性问题如算法 3 所示, 其输入为语句匹配信息 *M*、返回值信息 *R*、异常流信息 *E*、切片信息 *S*、全局语句信息 *G* 以及条件分支信息 *B*, 输出为兼容性问题产生原因集合 *CI*, 包括输出不同的返回值 (CI7).

首先, 第 1–23 行计算 API 从 v1 版本更新至 v2 版本中的良性变化语句集合 BSC_1 和 BSC_2. 根据文献 [12], 当第三方库进行更新时, 由于修复缺陷、提升性能、重构方法等原因, 可能会发生良性的语义变化 (benign semantic change), 这些变化通常不被视为兼容性问题. 对于缺陷修复而言, 它不符合兼容性问题定义中的第 1 个条件, 即在

v1 版本中无法正常运行. 而性能提升和方法重构并不会改变 API 的输出. 文献 [12] 中还总结了第三方库良性语义变化的实现方式, 例如替换内部调用的方法 (第 3–6 行)、添加 try-except 结构 (第 9–11 行)、添加或删除条件分支结构 (第 12–17 行)、更改原有条件分支结构的判断条件 (第 18–21 行) 等. CallMethod(stmt) 返回语句 stmt 中调用方法的名称, Parameter(method, stmt) 返回语句 stmt 中方法 method 的参数列表.

然后, 算法 3 第 24–26 行分析匹配的 return 语句<s_a, s_b>, 若 s_a 或 s_b 的后向切片语句中存在非良性变化, 则认为存在输出不同的返回值 (CI7). 第 27 行分析新增和删除的 return 语句, 若存在非良性变化的新增 return 语句或删除 return 语句, 则认为存在输出不同的返回值 (CI7).

算法 3. 检测与返回值有关的兼容性问题.

输入: *M*, *R*, *E*, *S*, *G*, *B*;

输出: *CI*.

Function DetectReturn

1. BSC_1 , BSC_2 = { }, { }
2. $BSC_1 = BSC_1 \cup (M.del \cap G_2)$, $BSC_2 = BSC_2 \cup (M.add \cap G_1)$
3. **for each** <s_a, s_b> **in** *M.change* **then**:
4. m_1 = CallMethod(s_a), m_2 = CallMethod(s_b)
5. **if** m_1 != m_2 **and** Parameter(m_1, s_a) == Parameter(m_2, s_b) **then**:
6. $BSC_1 = BSC_1 \cup \{s_a\}$, $BSC_2 = BSC_2 \cup \{s_b\}$
7. **while** True **then**:
8. tmp_BSC_1 = BSC_1, tmp_BSC_2 = BSC_2
9. **for each** s **in** $M.add \cap E_2.C$ **then**:
10. **if** $E_2.C$[s].*TryStmts* $\subseteq M.add$ **or** $E_2.C$[s].*TryStmts* $\subseteq M_2.unchange \cup BSC_2$ **then**:
11. $BSC_2 = BSC_2 \cup E_2.C$[s].*AllStmts*
12. **for each** s **in** $M.add \cap B_2$ **then**:
13. **if** B_2[s].*Stmts* $\subseteq M.add$ **or** B_2[s].*Stmts* $\subseteq M_2.unchange \cup BSC_2$ **then**:
14. $BSC_2 = BSC_2 \cup \{s\} \cup B_2$[s].*Stmts*
15. **for each** s **in** $M.del \cap B_1$ **then**:
16. **if** B_1[s].*Stmts* $\subseteq M.del$ **or** B_1[s].*Stmts* $\subseteq M_1.unchange \cup BSC_1$ **then**:
17. $BSC_1 = BSC_1 \cup \{s\} \cup B_1$[s].*Stmts*
18. **for each** <s_a, s_b> **in** *M.change* **then**:
19. **if** (s_a **in** B_1) **and** (s_b **in** B_2) **and** ($B_1[s_a].condition$!= $B_2[s_b].condition$) **then**:
20. **if** $B_1[s_a].Stmts \subseteq M_1.unchange \cup BSC_1$ **and** $B_2[s_b].Stmts \subseteq M_2.unchange \cup BSC_2$ **then**:
21. $BSC_1 = BSC_1 \cup \{s_a\}$, $BSC_2 = BSC_2 \cup \{s_b\}$
22. **if** tmp_BSC_1 == BSC_1 **and** tmp_BSC_2 == BSC_2 **then**:
23. **break**
24. **for each** <s_a, s_b> **in** $M.change \cup M.unchange$ **then**:
25. **if** s_a **in** R_1 **and** s_b **in** R_2 **then**:
26. $CI \leftarrow$ {CI7} **if** Len($S_1.BR[s_a]$-BSC_1-$M_1.unchange$)>0 **or** Len($S_2.BR[s_b]$-BSC_2-$M_2.unchange$)>0
27. $CI \leftarrow$ {CI7} **if** Len($M.del \cap R_1$-BSC_1)>0 **or** Len($M.add \cap R_2$-BSC_2)>0
28. **return** *CI*

Function End

例 11: 示例代码 7 中, 算法 3 第 2 行根据全局语句信息更新 BSC_1={}, BSC_2={s_8}. 进行首轮第 7–23 行的循环时, tmp_BSC_1={}, tmp_BSC_2={s_8}. 由于 $E_2.C[s_7].TryStmts$={s_8}⊆$M.add$, 在第 9–11 行后 BSC_2={s_7, s_8, s_9, s_{10}}. 由于 $B_1[s_6].Stmts$={s_{10}}⊆$M_1.unchange$∪BSC_1, 在 15–17 行后 BSC_1={s_6}. 此时第 22 行的条件不成立, 再次进入循环. 第 2 轮循环中 BSC_1 和 BSC_2 没有更新, 第 22 行的条件成立, 跳出循环, 此时 BSC_1={s_6}, BSC_2={s_7, s_8, s_9, s_{10}}. 因此, 在第 24–26 行分析保留 return 语句 s_{10} 时, 第 26 行的条件不成立. 又因为 $M.del$∩R_1=$M.add$∩R_2={}, 所以该 API 不存在输出不同的返回值 (CI7).

例 12: 示例代码 4 中, 算法 3 在第 18–21 行分析匹配的条件分支语句<s_3, s_4>时, $B_1[s_3].Stmts$={s_5}, $M_1.unchange$∪BSC_1={s_2}, 导致第 20 行的条件无法成立, 最终 BSC_1=BSC_2={}. 在第 24–26 行分析匹配 return 语句<s_5, s_6>时, $S_1.BR[s_5]$={s_3, s_5}, $S_1.BR[s_5]$-BSC_1-$M_1.unchange$={s_3, s_5}, 导致第 26 行的条件成立, 更新 CI 为{CI7}.

4 实验评估

本节对检测方法进行实验评估, 首先介绍实验设计, 然后介绍实验数据与评价指标, 最后介绍有效性、泛化性、时间性能和空间性能以及易用性的评估结果.

4.1 实验设计

为了评估本文提出的方法能否有效检测 Python 第三方库 API 的兼容性问题, 设计了如下 4 个研究问题.

RQ3: 本文提出方法的有效性如何?

RQ4: 本文提出方法的泛化性如何?

RQ5: 本文提出方法的时间性能和空间性能如何?

RQ6: 本文提出方法的易用性如何?

4.2 实验数据与评价指标

4.2.1 有效性评估 (RQ3)

4.2.1.1 实验数据

在第 2.2.2 节中本文已经在 flask 库和 pandas 库上收集到 108 个有兼容性问题的 API 对, 即正样本. 为评估方法的有效性, 本文进一步通过更新日志和回归测试相结合的方法, 在表 2 的 6 个版本对上进行负样本的收集, 即收集兼容 API 对. 为确保负样本数据的可靠性, API 被归类为负样本需要满足两个条件: 第一, 该 API 在第三方库 v1 版本中有相应的测试用例, 并且在 v2 版本中成功通过回归测试; 第二, 更新日志中显示变更此 API 的原因包含“缺陷修复”“性能提升”“代码重构”等关键字. 通过上述方法, 本文共收集到 108 个不兼容 API 对 (正样本) 以及 108 个兼容 API 对 (负样本). 为确保有效性评估的真实性, 本文仅对收集到的这 216 个 API 对进行检测. 具体来说, 评估时对方法的输入为第三方库前后两个版本的源代码, 并且指定方法对这 216 个 API 对进行检测.

4.2.1.2 评价指标

兼容性问题产生原因的检测可以抽象为一个多标签分类问题 (multi-label classification), 因此按照多标签分类的精准率 (precision) 和召回率 (recall) 来评估方法的有效性[30].

精准率衡量了数据集在所有被预测为正标签的样本中, 真正为正样本的比例. 在本实验中, 一方面会分别评估 7 种细粒度兼容性问题产生原因 (CI1–CI7) 的精准率; 另一方面计算各产生原因的宏平均精准率, 即 7 个精准率的算数平均值.

召回率衡量了数据集的所有正样本中被成功检测到的比例. 在本实验中, 一方面会分别评估 7 种细粒度兼容性问题产生原因 (CI1–CI7) 的召回率; 另一方面计算各产生原因的宏平均召回率, 即 7 个召回率的算数平均值.

4.2.2 泛化性评估 (RQ4)

4.2.2.1 实验数据

本文在机器学习领域的 sklearn 库[31]以及科学计算领域的 numpy 库[32]上评估方法的泛化性. 首先, 本文根据

第 2.2.1 节的标准选取了跨 major、minor、patch 的 6 个版本对, 分别为 sklearn 0.24.2→1.0.0、sklearn 1.1.3→1.2.0、sklearn 1.1.0→1.1.3、numpy 1.26.4→2.0.0、numpy 1.25.2→1.26.0 以及 numpy 1.26.0→1.26.4. 然后, 采用与 flask 库和 pandas 库相同的方法在这 6 个版本对上进行正样本 (不兼容 API 对) 和负样本 (兼容 API 对) 的收集, 一共收集到 88 个正样本和 71 个负样本. 具体来说, 在 sklearn 库上收集到 58 个正样本和 34 个负样本, 在 numpy 库上收集到 30 个正样本和 37 个负样本. 正样本的具体分布如表 7 所示. 此外, 与 RQ3 相同, 评估时指定方法检测这 6 个版本对上的共计 159 个 API.

表 7 sklearn 库和 numpy 库上的 API 兼容性问题数据

序号	兼容性问题产生原因	总计 (个)	sklearn库不兼容API数量 (个)	numpy兼容API数量 (个)
CI1	参数的增删	19	17	2
CI2	关键字参数的键增删	7	7	0
CI3	参数的重命名	3	2	1
CI4	参数默认值的改变	13	13	0
CI5	参数的范围改变	17	15	2
CI6	抛出不同的异常	1	0	1
CI7	输出不同的返回值	32	8	24
	总计	88	58	30

注: 由于一个API可能由多种原因导致不兼容, 总计数据小于各项加和

4.2.2.2 评价指标

与 RQ3 相同, 本文使用精准率和召回率来评估方法的泛化性.

4.2.3 时间性能和空间性能评估 (RQ5)

4.2.3.1 实验数据

为充分评估方法的时间性能和空间性能, 一方面, 本文指定方法检测上文在 flask 库、pandas 库、sklearn 库、numpy 库的 12 个版本对上收集到的 196 个不兼容 API 对和 179 个兼容 API 对; 另一方面, 使用方法直接检测 12 个版本对中的所有变更 API.

4.2.3.2 评价指标

本文使用对 API 进行检测的平均耗时来评估方法的时间性能, 单位为 s; 本文使用检测时的内存占用来评估方法的空间性能, 单位为 MB. 由于检测时的时间消耗和内存占用有一定的随机性, 本文使用运行 3 次的平均值.

4.2.4 易用性评估 (RQ6)

评估易用性是探讨本文检测方法是否能帮助开发者简便地理解 API 兼容性问题, 从而避免上层应用异常终止或产生不一致的结果. 一方面, 本文从理解时间的缩短以及理解正确率的提升这两个角度进行定量评估; 另一方面, 本文从对定位不兼容代码元素的帮助以及对理解问题产生原因的帮助这两个角度进行定性评估.

4.2.4.1 实验数据

本文检测方法的应用场景是从第三方库两个版本的源代码中为开发者检测不兼容 API, 并提供兼容性问题产生原因. 因此, 本文邀请了 10 名具有 3 年以上 Python 编程经验的开发者在不兼容 API 上进行人工实验, 其中不包含本文的作者. 本文从收集到的 196 个不兼容 API 对中随机抽取 14 个, 记为 API_1–API_{14}. 7 种兼容性问题产生原因 (CI1–CI7) 在 API_1–API_{14} 中均有 2 个. 本文将 10 名参与者平均分成两组, 为他们提供 API 前后两个版本的代码差异对比, 他们需要从 API_1 开始对 API_1–API_{14} 进行标注. 第 1 组需要标注的前 7 个 API 被提供了本文检测方法的结果, 而后 7 个没有; 第 2 组以相反的形式被提供检测方法结果.

具体来说, 10 名实验参与者需要理解 API_1–API_{14} 的兼容性问题产生原因, 然后分别进行标注, 并记录对每个 API 进行标注的时间. 在完成上述任务后, 参与者还需要对 2 个定性问题打分, 分数区间为 0–5, 具体问题如下.

- 问题 1: 与直接看 API 的代码差异相比, 我们的工具更能帮助定位造成兼容性问题的代码元素吗?
- 问题 2: 与直接看 API 的代码差异相比, 我们的工具更能帮助理解造成兼容性问题的产生原因吗?

4.2.4.2 评价指标

基于上述人工实验的结果, 本文使用以下 3 个问题来评估方法的易用性.

● RQ6-1: 本文检测方法能否帮助定位造成兼容性问题的代码元素? 即本文方法能否帮助开发者定位不兼容的参数、语句、异常、返回值等代码元素. 具体来说, 本文使用问题 1 的打分进行评估.

● RQ6-2: 本文检测方法能否帮助理解造成兼容性问题的产生原因? 一方面, 本文使用 API 在有检测方法结果以及没有检测方法结果的两种情况下, 被标注正确率的对比进行评估. 这里的标注正确率指对某一不兼容 API , 10 名参与者中标注正确的占比. 另一方面, 本文使用问题 2 的打分进行评估.

● RQ6-3: 本文检测方法能否帮助缩短发现兼容性问题的所需时间? 具体来说, 本文使用 API 在有检测方法结果以及没有检测方法结果的两种情况下, 被标注所需时间的对比进行评估. 这里的标注时间指对某一不兼容 API , 10 名参与者中标注时间的平均值.

4.3 有效性评估结果 (RQ3)

表 8 展示了本文提出的方法在 flask 库和 pandas 库数据集上的有效性结果. 总的来看, 方法对兼容性问题产生原因的宏平均精准率和宏平均召回率分别达到了 92.87% 和 93.79%. 从更细粒度的角度看, 除输出不同的返回值 (CI7) 的精准率较低, 其余类别的精准率和召回率均不低于 75%. 可见, 本文方法在分析 Python 第三方库 API 兼容性问题产生原因的有效性.

表 8 方法在 flask 库和 pandas 库数据集上的有效性结果

序号	兼容性问题产生原因	不兼容API数量 (个)	精准率 (%)	召回率 (%)
CI1	参数的增删	26	100	100
CI2	关键字参数的键增删	1	100	100
CI3	参数的重命名	3	100	100
CI4	参数默认值的改变	11	100	100
CI5	参数的范围改变	45	93.33	95.45
CI6	抛出不同的异常	6	100	83.33
CI7	输出不同的返回值	31	56.76	77.78
	宏平均		92.87	93.79

本检测方法发生误报的原因主要有两个.

(1) 未充分利用 API 外部代码的语义信息. 对于 API 外部代码, 本文方法仅分析同一文件中的全局语句, 并且分析时仅关注与 API 内部完全一致的全局语句, 将这些语句过滤为良性语义变化 (算法 3 第 2 行), 而没有涉及过程间分析. 然而, API 外部代码中有丰富的语义信息, 如声明和操作全局变量、导入模块中的特定变量、调用外部 API 等. 本文方法忽略了这些外部代码的语义信息, 这导致方法的错误识别, 如例 13 所示.

例 13: 示例代码 8 中, 第 4 行类型判断 isinstance 中的 text_type 被替换为 str. 而 v1 版本中 text_type 通过第 1 行的 from-import 语句导入, _compat.py 文件中语句‘text_type = str’使 text_type 被赋值为 str. 因此, 该 API 本质上未发生修改. 然而, 算法 1 (第 18–21 行) 错误地认为第 8 行 raise 语句的触发条件发生改变, 进而错误地认为参数 rv 的输入范围发生改变, 即错误地认为该 API 存在参数的范围改变 (CI5).

示例代码 8. 未充分利用 API 外部代码的语义信息所导致的错误识别.

```
# 版本变更: flask1.1.4 → flask2.0.0
1  –  from ._compat import text_type
2     def make_response(self, rv):
3         if (not isinstance(rv, self.response_class)):
4  –          if isinstance(rv, (text_type, bytes, bytearray)):
5  +          if isinstance(rv, (str, bytes, bytearray)):
```

```
6                ...
7            else:
8                raise TypeError
9        return rv
```

(2) 未充分利用 API 内部代码的语义信息. 针对 API 的内部代码, 本文方法仅在两个版本的 API 源代码之间进行语句级别的匹配分析, 忽略了语句内部的语义信息以及语句与语句之间的语义信息, 这导致本文方法无法识别兼容性的代码重构, 进而导致方法的错误识别, 如例 14 所示.

例 14: 示例代码 9 中, 第 6 行代码重构为第 7–10 行. 然而, 在分析匹配的 return 语句<s_6, s_{10}> 时, M_1.*unchange*={s_2, s_3, s_4, s_5, s_{11}}, BSC_1={}, S_1.*BR*[s_6]={s_3, s_5, s_6}, 使得 S_1.*BR*[s_6]-BSC_1-M_1.*unchange*={s_6}, 从而导致算法 3 第 26 行的条件成立, 错误地认为该 API 存在输出不同的返回值 (CI7).

示例代码 9. 未充分利用 API 内部代码的语义信息所导致的错误识别.

```
# 版本变更: pandas1.4.4 → pandas1.5.0
1      def is_inferred_bool_dtype(arr):
2          ...
3          if dtype == np.dtype(bool):
4              return True
5          elif dtype == np.dtype('object'):
6  –           return lib.is_bool_array(arr)
7  +           result = lib.is_bool_array(arr)
8  +           if result:
9  +               warnings.warn('In a future version, object-dtype columns with all-bool values will not be
                                  included in reductions with bool_only=True. Explicitly cast to bool dtype
                                  instead.', FutureWarning, stacklevel=find_stack_level(inspect.currentframe()))
10 +           return result
11         return False
```

4.4 泛化性评估结果 (RQ4)

表 9 展示了本文提出的方法在表 7 数据集上的表现. 工具在 sklearn 库和 numpy 库上的表现与表 8 中在 flask 库和 pandas 库上的表现类似. 总的来看, 方法的宏平均精准率和宏平均召回率分别达到了 88.65% 和 94.59%. 从更细粒度的角度看, 除参数的范围改变 (CI5)、输出不同的返回值 (CI7) 的精准率较低, 其余类别的精准率和召回率均不低于 75%. 可见, 本文提出的方法具有良好的泛化性.

表 9　方法在 sklearn 库和 numpy 库数据集上的泛化性结果

序号	兼容性问题产生原因	不兼容API数量 (个)	精准率 (%)	召回率 (%)
CI1	参数的增删	19	100	100
CI2	关键字参数的键增删	7	100	100
CI3	参数的重命名	3	100	100
CI4	参数默认值的改变	13	100	100
CI5	参数的范围改变	17	65.22	83.33
CI6	抛出不同的异常	1	100	100
CI7	输出不同的返回值	32	55.32	78.79
	宏平均		88.65	94.59

4.5 时间性能和空间性能评估结果 (RQ5)

一方面, 实验时指定方法检测上文在共计 12 个版本对上收集到的 196 个不兼容 API 对和 179 个兼容 API 对, 表 10 展示了具体的检测时间开销和内存占用. 在时间开销上, 检测 375 个 API 共耗时 2 036.24 s, 平均每个 API 的检测大约需要 5.43 s. 在内存占用上, 检测 375 个 API 共占用内存 498.04 MB.

另一方面, 实验时使用方法直接检测 12 个版本对中的所有变更 API, 表 11 展示了具体的时间开销和内存占用. 在时间开销上, 检测 12 个版本对中的 5 243 个变更 API 共耗时 30 858.09 s, 平均每个版本对的检测大约需要耗时 42.86 min, 平均每个变更 API 的检测大约需要 5.88 s. 在内存占用上, 检测 12 个版本对中的 5 243 个变更 API 共占用内存 1 494.24 MB, 平均每个版本对的检测大约需要占用内存 124.52 MB.

表 10 方法在指定数据集上的时间性能、空间性能结果

第三方库	版本对v1→v2		代码行数	检测API数量 (个)	检测时间 (s)	平均检测时间 (s/个)	内存占用 (MB)
flask	major	1.1.4→2.0.0	4k→4k	16	65.91	4.12	14.35
	minor	2.1.3→2.2.0	4k→4k	16	48.53	3.03	15.48
	patch	2.1.0→2.1.3	4k→4k	4	20.82	5.21	14.64
pandas	major	0.25.3→1.0.0	78k→77k	87	342.06	3.93	63.41
	minor	1.4.4→1.5.0	104k→111k	73	413.18	5.66	76.23
	patch	1.4.0→1.4.4	101k→104k	20	140.15	7.01	61.42
sklearn	major	0.24.2→1.0.0	54k→64k	41	253.77	6.18	51.09
	minor	1.1.3→1.2.0	66k→68k	39	271.83	6.97	36.80
	patch	1.1.0→1.1.3	64k→66k	12	68.24	5.69	46.93
numpy	major	1.26.4→2.0.0	69k→108k	46	266.54	5.79	45.23
	minor	1.25.2→1.26.0	68k→68k	6	41.35	6.89	37.19
	patch	1.26.0→1.26.4	68k→69k	15	103.86	6.93	35.27
总计				375	2 036.24	5.43	498.04

表 11 方法在所有变更 API 上的时间性能、空间性能结果

第三方库	版本对v1→v2		代码行数	API总数量 (个)	变更API数量 (个)	检测时间 (s)	平均检测时间 (s/个)	内存占用 (MB)
flask	major	1.1.4→2.0.0	4k→4k	324→312	258	1 137.23	4.41	27.09
	minor	2.1.3→2.2.0	4k→4k	315→336	103	447.77	4.35	22.82
	patch	2.1.0→2.1.3	4k→4k	314→315	47	197.15	4.19	17.94
pandas	major	0.25.3→1.0.0	78k→77k	5 125→4 954	1 814	10 565.17	5.82	364.48
	minor	1.4.4→1.5.0	104k→111k	5 885→6 103	1 636	9 373.08	5.74	370.52
	patch	1.4.0→1.4.4	101k→104k	5 881→5 885	94	560.73	5.97	115.46
sklearn	major	0.24.2→1.0.0	54k→64k	2 829→2 939	492	3 101.86	6.30	184.59
	minor	1.1.3→1.2.0	66k→68k	3 075→3 182	587	4 078.09	6.95	195.41
	patch	1.1.0→1.1.3	64k→66k	3 071→3 075	41	263.50	6.43	57.44
numpy	major	1.26.4→2.0.0	69k→108k	3 677→3 448	125	838.39	6.71	51.15
	minor	1.25.2→1.26.0	68k→68k	3 650→3 659	17	106.15	6.24	46.01
	patch	1.26.0→1.26.4	68k→69k	3 659→3 677	29	188.97	6.52	41.33
总计					5 243	30 858.09	5.88	1 494.24

因此, 本文方法具有良好的时间性能和空间性能. 此外, 通过分析代码行数可以看出, 本文方法具有良好的处理大型代码库的能力.

4.6 易用性评估结果 (RQ6)

图 5 和表 12 展示了人工实验的结果. 基于上述结果, 本文使用以下 3 个子问题来评估方法的易用性.

RQ6-1: 本文检测方法是否能帮助定位造成兼容性问题的代码元素?

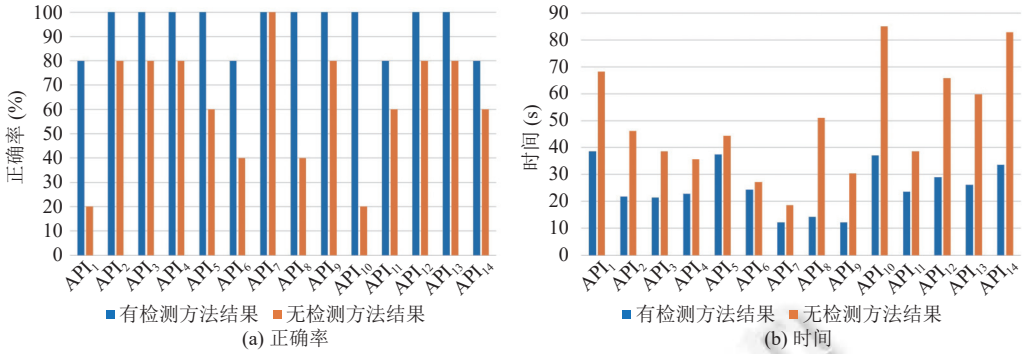

图 5 人工实验的标注正确率和标注时间

表 12 人工实验中两个问题的得分情况

问题	得分个数						平均得分
	(0)	(1)	(2)	(3)	(4)	(5)	
问题1	0	0	0	0	2	8	4.8
问题2	0	0	0	0	2	8	4.8

表 12 第 3 行展示了实验参与者对问题 1 (第 4.2.4.1 节) 的打分. 问题 1 的平均得分为 4.8, 这表明本文检测方法能较好地帮助定位造成兼容性问题的代码元素, 如参数、语句、异常、返回值等.

RQ6-2: 本文检测方法是否能帮助理解造成兼容性问题的产生原因?

一方面, 图 5(a) 展示了不兼容 API 在有检测方法结果以及没有检测方法结果下的标注正确率. 标注正确率指针对某一 API , 10 名参与者中标注正确的占比. 可以看到, 有检测方法结果的标注正确率均不低于无检测方法结果的标注正确率, 并且部分 API 的标注正确率提升了 3 倍. 在检测方法结果的帮助下, 平均每个 API 的标注正确率为 94.29%, 这也表明本文方法的结果受到开发者的认可. 如果没有检测方法的帮助, 平均每个 API 的标注正确率为 62.86%.

另一方面, 表 12 第 4 行展示了实验参与者对问题 2 的打分. 问题 2 的平均得分为 4.8.

从上述两部分的实验结果可以看出, 本文检测方法能较好地帮助理解造成兼容性问题的产生原因.

RQ6-3: 本文检测方法是否能帮助缩短发现兼容性问题的所需时间?

图 5(b) 展示了不兼容 API 在有检测方法结果以及没有检测方法结果下的标注时间. 标注时间是指针对某一 API, 10 名参与者标注时间的平均值. 可以看到, 有检测方法结果的标注时间均低于无检测方法结果的标注时间, 并且部分 API 的标注时间缩短了一半. 当有检测方法结果时, 平均每个 API 的标注时间为 25.31 s, 即在本文检测方法的帮助下, 开发者只需 25.31 s 即可发现兼容性问题的产生原因. 然而, 如果没有本文检测方法的帮助, 开发者平均需要 49.44 s 才能发现兼容性问题的产生原因. 由此可见, 本文检测方法能较好地帮助缩短发现兼容性问题的所需时间.

综合 RQ6-1、RQ6-2、RQ6-3 的结果可知, 本文检测方法对用户来说具有良好的易用性.

5 讨 论

5.1 威胁分析

本文提出的方法虽然能够根据兼容性问题产生原因检测出不兼容 API, 但在有效性和泛化性上存在威胁.

(1) 有效性威胁. 正如第 4.3 节有效性评估结果中指出, 制约方法有效性的主要原因有两个, 即未充分利用 API 外部代码的语义信息和未充分利用 API 内部代码的语义信息. 为了增加方法的有效性, 对于 API 外部代码, 本文提取了全局语句信息 G, 并在检测输出不同的返回值 (CI7) 时将移动全局语句过滤为良性语义变化; 对于 API

内部代码, 本文提取了多种代码信息, 包括取语句匹配信息 M、参数信息 P、返回值信息 R、异常流信息 E、切片信息 S 以及条件分支信息 B. 未来可以考虑增加过程间分析, 通过 API 调用链分析充分利用 API 外部代码, 并对语句语义进行细粒度分析, 从而充分利用 API 外部代码、内部代码的语义信息.

(2) 泛化性威胁. 对本文方法泛化性的威胁主要与 Python 第三方库及其版本对的选择有关. 在构建数据集时, 为确保 API 兼容性标注的准确性, 本文采用将回归测试和更新日志相结合的方法, 构建时间成本较高. 因此, 本文只选择了 4 个常用 Python 第三方库的 12 个版本对进行实验评估. 为了减少第三方库及其版本选择造成的泛化性威胁, 在第三方库的选择上, 本文选择了 4 个不同领域的常用第三方库, 分别是 Web 开发领域的 flask 库、数据分析领域的 pandas 库、机器学习领域的 sklearn 库、科学计算领域的 numpy 库; 在版本对的选择上, 本文对每个第三方库均选取最新的跨 major 版本对、跨 minor 版本对、跨 patch 版本对. 未来可以考虑在更多第三方库、更多版本对上构建数据集, 并进行更深入的实验评估.

5.2 应用局限性分析

本文提出的方法虽然能够根据兼容性问题产生原因检测出不兼容 API, 但需要被提供第三方库前后两个版本的源代码才会进行兼容性问题的检测, 在实际应用中存在局限性. 未来可以考虑通过将本方法集成到持续集成/持续部署 (CI/CD) 流程的工具中来提升方法的实时检测能力. 具体来说, 在构建时自动拉取第三方库 commit 的源代码, 并计算出变更 API 的列表. 当开发者需要某 API 的兼容性问题信息时, 工具立即检测此 API, 从而实现实时或近实时的不兼容 API 检测.

6 总 结

Python 第三方库会频繁发生更新, 其中部分 API 发生了不兼容的更改, 导致上层应用运行出现异常终止或者产生不一致的结果. 本文围绕 Python 第三方库 API 的兼容性问题展开研究. 首先, 本文采用更新日志与回归测试结合的方法在 flask 库和 pandas 库的 6 个版本对上构建兼容性问题的数据集. 接着, 本文根据收集到的数据对 Python 第三方库 API 兼容性问题进行实证研究, 分析兼容性问题的表现形式和产生原因. 最后, 本文提出了一种基于静态分析技术的 Python 第三方库 API 兼容性问题检测方法. 本文在 4 个常用 Python 第三方库的共计 12 个版本对上进行了实验评估, 证明了该方法的具有良好的有效性、泛化性、时间性能、空间性能以及易用性. 本文提出的检测方法能根据兼容性问题的产生原因检测出不兼容 API, 但仍存在改进空间, 对方法的未来展望如下.

(1) 提升方法的有效性. 制约本文方法有效性的主要因素有两个: 一是方法主要分析 API 内部代码, 没有充分利用 API 外部代码, 比如未分析 API 内部所调用的其它 API 代码; 二是方法在语句级别进行匹配分析, 未充分利用 API 内部的语义信息. 未来可以考虑增加过程间分析, 通过 API 调用链分析充分利用 API 外部代码, 并对语句语义进行细粒度分析, 从而充分利用 API 外部代码、内部代码的语义信息.

(2) 提升方法的实用性. 本文方法是从第三方库的版本级别出发, 检测两个不同版本中存在兼容性问题的 API. 未来可以考虑将本文方法的应用场景拓宽到代码提交级别, 将方法集成到持续集成/持续部署 (CI/CD) 流程的工具中. 具体来说, 在构建时自动拉取第三方库代码提交前后的源代码, 计算出变更 API 的列表, 并检测这些变更 API 是否存在兼容性问题, 进一步提升方法的实时检测能力.

(3) 提升方法的通用性. 本文是在 Python 生态中研究了第三方库 API 的兼容性问题, 并提出了基于静态分析的 Python 第三方库 API 兼容性问题的检测方法. 未来可以考虑在 JavaScript、Java 等其它语言生态中收集第三方库的不兼容 API 对, 探究本文实证研究结果的通用性. 然后根据具体语言特性对本文方法进行扩展, 将方法推广到其它语言生态中. 例如, Java 语言没有关键字参数, 所以不存在关键字参数的键增删 (CI2) 导致的兼容性问题.

References:

[1] Islam M, Jha AK, Nadi S, Akhmetov I. PyMigBench: A benchmark for python library migration. In: Proc. of the 20th Int'l Conf. on Mining Software Repositories (MSR). Melbourne: IEEE, 2023. 511–515. [doi: 10.1109/MSR59073.2023.00075]

[2] 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 Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). London: IEEE, 2020. 81–92. [doi: 10.1109/SANER48275.2020.9054800]

[3] Haryono SA, Thung F, Lo D, Lawall J, Jiang LX. Characterization and automatic updates of deprecated machine-learning API usages. In: Proc. of the 2021 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Luxembourg: IEEE, 2021. 137–147. [doi: 10.1109/ICSME52107.2021.00019]

[4] Du XL, Ma J. AexPy: Detecting API breaking changes in python packages. In: Proc. of the 33rd Int'l Symp. on Software Reliability Engineering (ISSRE). Charlotte: IEEE, 2022. 470–481. [doi: 10.1109/ISSRE55969.2022.00052]

[5] Mostafa S, Rodriguez R, Wang XY. Experience paper: A study on behavioral backward incompatibilities of Java software libraries. In: Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Santa Barbara: ACM, 2017. 215–225. [doi: 10.1145/3092703.3092721]

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

[7] Zhao YJ, Li L, Liu K, Grundy J. Towards automatically repairing compatibility issues in published Android Apps. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 2142–2153. [doi: 10.1145/3510003.3510128]

[8] Xia H, Zhang Y, Zhou YT, Chen XT, Wang Y, Zhang XY, Cui SS, Hong G, Zhang XH, Yang M, Yang ZM. How Android developers handle evolution-induced API compatibility issues: A large-scale study. In: Proc. of the 42nd Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 886–898. [doi: 10.1145/3377811.3380357]

[9] 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]

[10] Chen LC, Hassan F, Wang XY, Zhang LM. Taming behavioral backward incompatibilities via cross-project testing and analysis. In: Proc. of the 42nd Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 112–124. [doi: 10.1145/3377811.3380436]

[11] Sun XY, Chen X, Zhao YJ, Liu P, Grundy J, Li L. Mining Android API usage to generate unit test cases for pinpointing compatibility issues. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2022. 70. [doi: 10.1145/3551349.3561151]

[12] Zhang L, 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]

[13] Mahmud T, Che MR, Yang GW. Android compatibility issue detection using API differences. In: Proc. of the 2021 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Honolulu: IEEE, 2021. 480–490. [doi: 10.1109/SANER50967.2021.00051]

[14] He DJ, Li L, Wang L, Zheng HJ, Li GW, Xue JL. Understanding and detecting evolution-induced compatibility issues in Android apps. In: Proc. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering. Montpellier: ACM, 2018. 167–177. [doi: 10.1145/3238147.3238185]

[15] Yang S, Chen S, Fan LL, Xu SH, Hui ZW, Huang S. Compatibility issue detection for Android Apps based on path-sensitive semantic analysis. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 257–269. [doi: 10.1109/ICSE48619.2023.00033]

[16] Peng Y, Zhang Y, Hu MZ. An empirical study for common language features used in python projects. In: Proc. of the 2021 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Honolulu: IEEE, 2021. 24–35. [doi: 10.1109/SANER50967.2021.00012]

[17] flask. https://github.com/pallets/flask

[18] pandas. https://github.com/pandas-dev/pandas

[19] Wang Y, Chen BH, Huang KF, Shi BW, Xu CY, Peng X, Liu YJ, Wu 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]

[20] Zhang ZJ, Yang YM, Xia X, Lo D, Ren XX, Grundy J. Unveiling the mystery of API evolution in deep learning frameworks: A case study of TensorFlow 2. In: Proc. of the 43rd Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP). Madrid: IEEE, 2021. 238–247. [doi: 10.1109/ICSE-SEIP52600.2021.00033]

[21] Dilhara M, Ketkar A, Dig D. Understanding software-2.0: A study of machine learning library usage and evolution. ACM Trans. on Software Engineering and Methodology, 2021, 30(4): 55. [doi: 10.1145/3453478]

[22] Liu P, Li L, Yan YC, Fazzini M, Grundy J. Identifying and characterizing silently-evolved methods in the Android API. In: Proc. of the

43rd Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP). Madrid: IEEE, 2021. 308–317. [doi: 10.1109/ICSE-SEIP52600.2021.00040]

[23] 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]

[24] 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. Virtual Event: ACM, 2020. 233–244. [doi: 10.1145/3368089.3409735]

[25] Vadlamani A, Kalicheti R, Chimalakonda S. APIScanner-towards automated detection of deprecated APIs in python libraries. In: Proc. of the 43rd Int'l Conf. on Software Engineering: Companion Proc. (ICSE-Companion). Madrid: IEEE, 2021. 5–8. [doi: 10.1109/ICSE-Companion52605.2021.00022]

[26] Haryono SA, Thung F, Lo D, Lawall J, Jiang LX. MLCatchUp: Automated update of deprecated machine-learning APIs in Python. In: Proc. of the 2021 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2021. 584–588.

[27] Brito G, Hora A, Valente MT, Robbes R. On the use of replacement messages in API deprecation: An empirical study. Journal of Systems and Software, 2018, 137: 306–321. [doi: 10.1016/j.jss.2017.12.007]

[28] Gumtree. https://github.com/GumTreeDiff/gumtree

[29] Python program analysis. https://github.com/microsoft/python-program-analysis

[30] Pereira RB, Plastino A, Zadrozny B, Merschmann LHC. Correlation analysis of performance measures for multi-label classification. Information Processing & Management, 2018, 54(3): 359–369. [doi: 10.1016/j.ipm.2018.01.002]

[31] Sklearn. https://github.com/scikit-learn/scikit-learn

[32] Numpy. https://github.com/numpy/numpy

沈阚(1999－), 女, 硕士生, 主要研究领域为软件供应链.

陈碧欢(1986－), 男, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为软件供应链, 智能网联汽车, AI 系统工程.

黄凯锋(1994－), 男, 博士, CCF 专业会员, 主要研究领域为软件供应链, 软件演化.

彭鑫(1979－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件智能化开发与运维, 人机物融合泛在计算, 智能网联汽车基础软件.