arXiv:2301.04563v1 [cs.SE] 11 Jan 2023

我依赖你，而你却伤害了我：客户端包中显现的破坏性变更实证研究

DANIEL VENTURINI,巴拉那联邦科技大学（UTFPR），巴西FILIPE ROSEIRO COGO,华为技术有限公司，加拿大IVANILTON POLATO,巴拉那联邦科技大学（UTFPR），巴西MARCO A GEROSA,北亚利桑那大学（NAU），美国IGOR SCALIANTE WIESE,巴拉那联邦科技大学（UTFPR），巴西

复杂的软件系统拥有依赖网络。开发者通常会配置包管理器（例如 npm），使其在每次发布包含错误修复和新功能的新版本时自动更新依赖项。当某个依赖项的发布版本引入向后不兼容的更改（通常称为破坏性变更）时，依赖包可能无法再构建。这可能会间接影响下游包，但破坏性变更的影响，以及依赖包如何从这些破坏性变更中恢复，仍不清楚。为弥补这一研究空白，我们研究了 npm 生态系统中破坏性变更的显现情况，重点关注包的构建受到其依赖项中的破坏性变更影响的案例。我们衡量了破坏性变更对依赖包的影响程度。分析结果表明，在依赖项的非主版本发布更新期间，约有 12% 的依赖包和 14% 的相关发布版本受到破坏性变更的影响。我们观察到，在所有显现的破坏性变更中，44% 同时在次版本和补丁版本发布中引入，而这两类版本原则上都应保持向后兼容。客户端在半数案例中自行从这些破坏性变更中恢复，最常见的做法是在不更改包管理器中的版本配置的情况下，升级或降级提供者版本。我们希望这些结果能帮助开发者了解此类更改可能造成的影响，并从中恢复。

CCS 概念：• **软件及其工程** → **软件演化**。

附加关键词和短语：破坏性变更、语义化版本、npm、依赖管理、变更影响

ACM 参考格式：
Daniel Venturini、Filipe Roseiro Cogo、Ivanilton Polato、Marco A Gerosa 和 Igor Scaliante Wiese。2021。我依赖了你，而你却让我崩溃：客户端包中显现破坏性变更的实证研究。*ACM* 测量、分析与计算系统会议录37，4，文章 111（2021年10月），26 页。https://doi.org/10.5281/zenodo.5558085

1 引言

复杂软件系统通常建立在依赖关系之上，其中客户端包会复用提供方包的功能，而后者又依赖于其他包。为了自动化安装过程，

作者地址：Daniel Venturini，danielventurini@alunos.utfpr.edu.br，巴拉那联邦科技大学（UTFPR），坎普莫朗，巴拉那州，巴西；Filipe Roseiro Cogo，filipe.cogo@gmail.com，华为技术有限公司，加拿大金斯顿；Ivanilton Polato，ipolato@utfpr.edu.br，巴拉那联邦科技大学（UTFPR），坎普莫朗，巴拉那州，巴西；Marco A Gerosa，Marco.Gerosa@nau.edu，北亚利桑那大学（NAU），美国亚利桑那州弗拉格斯塔夫；Igor Scaliante Wiese，igor@utfpr.edu.br，巴拉那联邦科技大学（UTFPR），坎普莫朗，巴拉那州，巴西。

允许为个人或课堂使用而免费制作本作品全部或部分内容的数字或纸质副本，前提是这些副本不得用于或分发以获取利润或商业利益，并且副本须附有本声明及首页上的完整引文。本作品中由 ACM 以外的其他方拥有版权的组成部分，必须遵守其版权规定。注明出处后可进行摘要引用。若要以其他方式复制、重新出版、发布到服务器或分发至列表，须事先获得明确许可和/或支付费用。请向 permissions@acm.org 申请许可。© 2021 计算机协会。稿件已提交至 ACM

升级、配置和移除依赖项时，npm、Maven、pip和Cargo 等包管理器得到了广泛采用。尽管复用提供方包带来了诸多好处，但客户端包面临的主要风险之一是破坏性变更 [21]。破坏性变更是提供方包所做的向后不兼容的更改，会导致客户端包构建出错（例如，提供方 API 的变更）。当客户端包将包管理器配置为自动接受一定版本范围内的提供方包更新时，破坏性变更可能造成严重后果，让客户端措手不及。例如，在npm中，大多数包都遵循语义化版本规范 [23], ，客户端会采用自动更新其提供方的次版本和补丁版本发布的配置。原则上，这些类型的发布版本不应包含任何破坏性变更，因为语义化版本规定只有主版本更新才应包含破坏性变更。然而，次版本或补丁版本发布偶尔也会引入破坏性变更；当这些变更在客户端显现时，会在客户端包中引发意外错误。由于包管理器中的依赖项具有传递性，意外的破坏性变更可能影响依赖网络中的很大一部分，导致多个包无法成功构建。

研究表明，提供方有时会错误地使用语义化版本规范 [15]。在 npm 生态系统中，先前的研究表明，提供方包确实会发布包含破坏性变更的版本 [14, 15, 18, 19]。然而，这些研究对于此类破坏性变更的普遍程度提供的信息有限，着重研究 API 破坏性变更，却未阐明客户端包如何解决这些变更造成的问题。在本文中，我们通过对托管在 GitHub 上的 npm 项目开展实证研究，填补这一空白，验证显现在客户端包中并导致缺陷的破坏性变更的频率和类型，以及客户端如何从中恢复。npm 是 JavaScript 编程语言的主要包管理器，拥有超过一百万个包。据估计，97% 的 Web 应用来自 npm [1], ，这使其成为规模最大的依赖网络 [9]。我们采用混合方法，识别并分析显现的破坏性变更 的类型——即提供方发布版本中的变更，这些变更会导致客户端的构建出现缺陷——以及客户端包在其项目中如何应对这些变更。本文不研究破坏性变更未在其他项目中显现的案例。我们的研究回答以下问题：

研究问题1：破坏性变更在多大程度上会显现在客户端包中?

我们采用随机抽样方法选取了 *384* 个软件包（置信水平为 *95%*，置信区间为 *±5%*），以筛选至少有一个提供方的客户端包。我们发现，显现的破坏性变更影响了 11.7% 的客户端包（不考虑其发布版本），以及 13.9% 的发布版本。此外，2.6% 的提供方引入了显现的破坏性变更。

研究问题2：提供方包中的哪些变更会显现为破坏性变更?

显现的破坏性变更主要由特性修改、依赖项之间的变更传播以及数据类型修改导致。我们还证实，次版本发布和补丁版本发布中引入的显现的破坏性变更比例相同（两种发布类型中均约为 *44%*）。提供方修复了次版本发布和补丁版本发布中引入的大多数显现的破坏性变更案例（分别为 *46.4%* 和 *61.5%*）。最后，*78.1%* 的案例中的显现的破坏性变更记录在问题报告、拉取请求或变更日志中。

研究问题3。客户端包如何从显现的破坏性变更中恢复?

在 *39.1%* 的案例中，客户端包从显现的破坏性变更中恢复了；当提供方没有修复破坏，或客户端先行恢复时，恢复过程耗时约 *134* 天。对于显现的破坏性变更，提供方发布修复版本的中位用时为七天。升级提供方是客户端包从显现的破坏性变更中恢复的最常见方式。

本文通过提供有关 npm 生态系统中显现的破坏性变更这一现象的定量和定性实证证据，为相关研究作出了贡献。我们的定性研究有助于开发者了解哪些类型的变更会在客户端包中显现为缺陷，以及人们采用哪些策略从破坏性变更中恢复。我们还提出了若干建议，说明客户端和提供方如何提升其发布流程的质量。作为另一项贡献，我们针对尚未解决的真实显现的破坏性变更案例创建了拉取请求，其中一半已被合并。

2 定义、范围与示例说明

本节定义本文使用的术语，并介绍本研究的示例说明。

2.1 术语表定义

以下将介绍本文中使用的术语及其定义，这些内容基于相关研究 [7, 11, 17]。

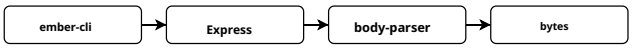


图 1. 依赖树示例，其中包含客户端和提供方。

• **提供方包版本** 是指为其他包的发布版本提供功能和资源以供使用的包版本。在图 1 中，包 Express 是 ember-cli 的提供方，body-parser是 Express 的提供方，依此类推。当我们想强调某个提供方包 P 还有其他提供方包时，会将其称为传递依赖提供方。 P 例如，在图 1 中，body-parser 是 Express 的提供方；body-parser 也有 bytes 作为提供方。在这种情况下，我们将body-parser 视为传递依赖提供方。

• **客户端包版本** 是指使用提供方包版本所提供的功能和资源的包版本。在图 1 中，Express是 body-parser 的客户端，body-parser 是 bytes 的客户端，依此类推。
• **直接提供者版本** 是指客户端直接使用的版本，也就是说，客户端明确声明为依赖项的包。在图 1 中，Express 是 ember-cli 的直接提供方，而 bytes 是body-parser的直接提供方。

• **间接提供者版本**是至少被其某个提供方使用的包发布版本。换言之，它是至少一个直接客户端的提供方的提供方。在图 1 中，body-parser和 bytes 都是 ember-cli的间接提供方，而 bytes是 express的间接提供方。

• **传递提供者版本**是引入破坏性变更的包发布版本与客户端之间的包发布版本。例如，如果 bytes在图 1中引入了一项破坏性变更，并影响客户端 ember-cli，那么包 express 和 body-parser 都是传递依赖提供方。这是因为破坏性变更经过这些包（body-parser 和 express）传递，最终到达客户端 ember-cli。这些传递依赖提供方也都会受到该破坏性变更的影响。

• **版本声明**：客户端可以在 *package.json* 中指定其提供方的版本。package.json 是 npm 用于指定提供方及其版本等信息的元数据文件。版本声明包含提供方可接受的版本。例如，在以下元数据 {" dependencies": {"express": "^4.10.6"}} 中的版本声明定义了客户端要求 express 使用版本 ^4.10.6。

• **版本范围**：在版本声明 上，客户端可以指定其接受的提供方版本/发布版本范围。范围分为三种：

– **全部（>=，或 *）**：使用此范围时，客户端声明支持/接受提供方发布的所有新版本，并可下载这些版本，即使其中包含破坏性变更。– **插入符号（^）**：使用此范围时，客户端声明支持/接受并可下载提供方发布的所有包含新功能和错误修复的新版本；必须避免破坏性变更。这是 npm 安装依赖项时使用的默认范围。– **波浪号范围（~）**：此范围表示，支持/接受并可下载提供方发布的所有仅包含错误修复的新版本；必须避免破坏性变更和新功能。– **固定范围**：此范围始终解析为某个特定版本，也称为特定范围。也就是说，版本声明不包含范围，而是指定一个特定版本。npm 允许使用命令行选项 –save-exact 以固定范围安装。

• **隐式更新和显式更新**：当客户端由于*package.json*中的版本范围而接收到新的提供方版本时，就会发生隐式更新 。对于采用版本范围定义的版本声明，例如^4.10.6，当 npm 安装了符合该范围的 4.10.9 版本时，就会发生隐式更新。显式更新则发生在客户端直接手动更新*package.json*中的版本声明时。

• **显现的破坏性变更** 是指显现为客户端包故障的提供方变更，最终导致客户端构建失败。先前文献采用的破坏性变更定义 [3–6, 8, 15, 19, 21]涵盖了一些不被视为破坏性变更的案例（例如，API 的某项变更实际上并未被客户端包使用）。反过来，显现的破坏性变更也包括先前破坏性变更定义未涵盖的案例（例如，由于提供方包的使用方式并非提供方开发者所预期，该提供方新发布版本中引入的一项符合语义化版本规范的变更，在客户端包中导致了预期的错误）。

2.2 示例说明

我们在人工分析中发现了以下两个显现的破坏性变更示例（在以下每个代码清单中，红色行表示从源代码中移除的内容，蓝色行表示插入到源代码中的内容）。我们的人工分析（第 3.2.1 节）包括运行该客户端各个发布版本的测试套件，并分析所有遇到错误的执行情况。

客户端 assetgraph-builder@7.0.0 有一个提供方 assetgraph@6.0.0 ，而该提供方有一个 terser@^4.0.0 版本，但由于版本范围的缘故，npm 安装了 terser@4.6.10。terser 4.3.0 版本 引入了一项变更：默认情况下，解析时会启用函数包装，如代码清单1所示。[1]

代码清单 1. terser@4.2.1 与 terser@4.3.0 默认行为之间的差异。

```
// terser@4 .2.1 未启用默认包装行为 foo(函数(){});


// terser@4 .3.0 默认包装行为 foo((函数(){}));
```

此更改破坏了 assetgraph-builder@7.0.0 的测试。[2] 此功能成为默认行为后，客户端 assetgraph-builder@8.0.0 会调整其测试，使其兼容terser 的行为，如代码清单 2。[3]

图 2. assetgraph@8.0.0 的客户端测试为适应破坏性变更所做的差异。

[1] https://github.com/terser/terser/compare/v4 0 2 21 43. . ..v . . https://github.com/terser/terser/issues/

[3] https://github.com/assetgraph/assetgraph-builder/commit/e4140416e7feaa3d088cf3ad0229fd677ff36dbc 提交至 ACM 的稿件

```
    expect(javaScriptAssets[0].text,'匹配',-       /SockJS=[\s\S]*define\("
main",function\(\)\{\}\);/+     /SockJS=[\s\S]*define\("main",
\(?function\(\)\{\}\)     ?\);/);
```

有时，提供方变更可能在引入很久之后才导致客户端出现问题。这种情况发生在客户端包ember-cli-chartjs@2.1.1中。在图 2中，发布版本 1.0.4 的 ember-cli-qunit （左侧依赖树）引入了一项变更，但并未导致破坏性变更。然而，近三年后，ember-cli-qunit 与提供方1.3.1版本的 broccoli-plugin（中间依赖树）一同使用时，显现出了一项破坏性变更。

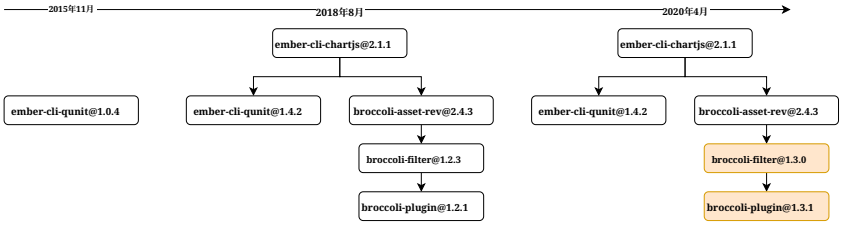


图 2. 发布时（中间依赖树）以及本研究中执行该发布版本相关测试时（右侧依赖树），ember-cli-chartjs@2.1.1 的依赖树（已解析版本）的演变。

2015年11月，提供方ember-cli-qunit@1.0.4 修复了代码中的一个错误，将函数 lintTree 返回的对象类型进行了更改[4]，如代码清单 3所示。尽管这是一次类型更改，但发布时并未破坏客户端，并且这一修复在后续的 ember-cli-qunit 发布版本中得以保留。

清单 3。ember-cli-qunit@1.0.4 对象类型更改。

```
    lintTree:  函数(type , tree) { // 跳过 如果 useLintTree    ===  为 false。如果
(this.options['ember-cli-qunit']&& ... ) {- 返回 tree;  +  // 模拟一个空的 Broccoli 树+
返回 { inputTree: tree , rebuild: 函数() { 返回  []; } }; }
```

将近三年后，即 2018 年 8 月，提供方 broccoli-plugin@1.3.1 发布了该版本（图 2 中的 middle-tree），用于修复一个缺陷，[5] 如代码清单 4所示。

清单 4。broccoli-plugin@1.3.1 验证函数增强。

```
    函数 isPossibleNode(node){- return typeof node
 === 'string' ||- (node !== null && typeof node
 === 'object')+ var type = typeof node;+ if (node
 === null){+     returnfalse;+ } else if (type   ===
'string') {...+ }else {
```

[4]https://github.com/ember-cli/ember-cli-qunit/commit/6fdfe7d [5]
https://github.com/broccolijs/broccoli-plugin/commit/3f9a42b

```
+       return false;
+   }
```

发布版本 1.3.1 的 broccoli-plugin 包因提供方 ember-cli-qunit@1.0.4，[6] 近三年前发布的一个修复而出现了显现的破坏性变更。如图 2所示，随着时间推移，ember-cli-chartjs 的依赖树因版本范围而不断演变，从而导致了这一问题。当 ember-cli-chartjs@2.1.1 于 2020 年 4 月（我们的分析日期）安装时，由于 broccoli-plugin@1.3.1 的变更被集成到 ember-cli-qunit中，安装失败。十五天后，ember-cli-qunit@1.4.3 再次更改 ember-cli-qunit的对象类型后，修复了这个问题。[7] 在显现的破坏性变更持续未解决的十五天期间，broccoli-plugin从 *384k*个 npm下载量。这一情形表明，即使是受欢迎且成熟的项目也可能受到破坏性变更的影响。尽管我们承认下载量未必能反映某个包的受欢迎程度，但我们仍以此指标作为示例，说明提供方包可能影响多少客户端包。

3 研究设计

本节介绍　我们如何收集数据（第 3.1 节），以及各研究问题的动机与方法（第 3.2 节）　.

3.1 数据收集

3.1.1 从 *npm* 包中获取元数据。图 3 的第一部分展示了我们对数据库进行抽样的方法。我们首先收集了 *package.json* 文件，即截至 *npm* 注册表中 2010年12月20日 至 2020年4月1日 期间发布的包的所有元数据文件，共计 1,233,944 个包。此时间范围分别对应我们能够获取的最早检查点，以及开始本研究时的最新检查点。我们忽略了 *package.json* 中没有任何提供方的包，因为这类包不能视为客户端包，因此不会遭受破坏性变更。过滤掉不含提供方的包后，我们的数据集包含 987,595 个 *package.json* 元数据文件。对于每个包的每个发布版本，我们记录了其发布时间戳，以及提供方的名称和对应的版本声明。

我们解析了所有版本声明，并确定了每个客户端版本发布时解析后的提供方版本。以往研究在考察依赖管理时也采用了类似的方法 [7, 29]。对于每个客户端版本中的每个提供方，我们检索了满足该客户端在该版本中指定范围的最新提供方版本；也就是说，检索解析后的版本。利用该解析后的版本，我们判断提供方在两个客户端版本之间是否更改了版本。换言之，我们通过在客户端发布时解析 提供方版本，重现了所有提供方采用的版本。

为进一步筛选样本，我们分析了数据集中客户端包最新版本对应的*package.json* 快照中的两个条件：

(1) *package.json* 快照中的“测试脚本”字段应包含非空条目，且该条目应与默认值“错误：未指定测试”不同。我们设定这一标准，是为了运行作为检测显现的破坏性变更方法一部分的自动化测试。应用这一标准后，总计剩余 488,805 个包。

[6] https://github.com/broccolijs/broccoli-merge-trees/issues/
[7] https://github.com/ember-cli/ember-cli-qunit/commit/59ca6ad 提交至 ACM 的稿件

(2) *package.json* 快照中应包含一个条目，其中列有该包的代码仓库 URL，因为我们希望从软件包代码库中获取信息。应用这一标准后，数据集中剩余 410,433 个包。

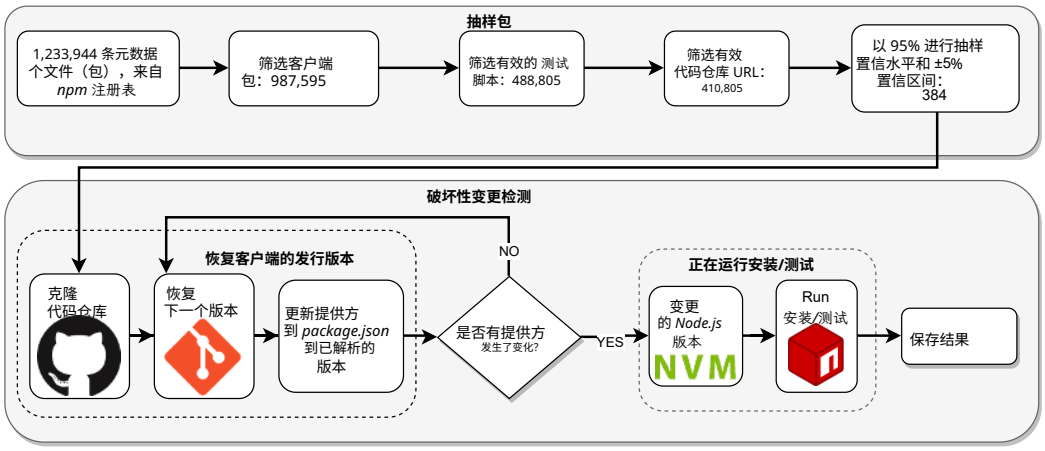


图 3. 从数据库中抽样并使用客户端版本执行相关测试的方法。

3.1.2 运行客户端测试。鉴于数据集规模庞大（超过 410,000 个客户端包），我们对随机样本运行了测试。在 95% 的置信水平和 ±5% 的置信区间下，我们随机抽取了 384 个软件包。样本中每个包的发布版本数中位数为 5.5 个，直接依赖项中位数为 9 个。我们选择研究随机样本，是因为对大型数据集进行人工分析耗时较长（第3.1.3节）；我们花了一个月的时间在样本上执行该方法。我们没有根据发布版本数、提供方数量或任何其他指标忽略任何包。我们对所有发布版本少于四个的入选包（384 个中的 130 个）进行了人工检查，查看其代码仓库，并尝试移除并非真实项目、缺少测试、缺少代码或属于示例项目等包。移除一个包后，我们会依据上述两项标准重新抽取一个。

图3 的第二部分展示了我们针对 384 个客户端的每个发布版本运行测试脚本的方法。对于每个客户端包，我们克隆其代码仓库——所有客户端代码仓库均托管在GitHub 上——并使用各自的版本标签（例如“v1.0.0”）还原所有发布版本的工作树。对于没有版本标签的发布版本，我们使用其 package.json 元数据中提供的时间戳还原工作树（即，将发布版本的时间戳与 master 分支中最接近的现有提交相匹配）。我们进行了分析并确认，在 94% 带有版本标签的发布版本中，标签和时间戳指向同一提交，因此，基于时间戳检出未标记的发布版本是可靠的。

恢复客户端版本的工作树后，我们使用特定的解析后提供方版本更新了相关 *package.json*条目中的全部版本声明（参见第 3.1.1 节）。随后，我们排除了名为 *package-lock.json* 的文件，该文件会 锁定提供方和间接提供方的版本。每当某个发布版本中的提供方包发生变更时，我们还会在该客户端包的一个发布版本上执行相关测试，因为这可能导致显现的破坏性变更。提供方变更包括：1）在 *package.json* 中新增提供方；或 2）客户端包的前一发布版本与当前发布版本之间，某个提供方的解析后版本发生变化。

我们力求复现提供方发生变更时所处的相同构建环境。因此，在执行客户端包的测试之前，我们尽最大努力确定客户端包采用的 Node.js 版本

——即提供方发生变更时客户端包所使用的版本。这是因为 Node.js 每六个月就会发布一个新的主版本。[8] 由于我们希望复现客户端包发布版本时的测试结果，因此在执行客户端包测试之前，我们更改了 Node.js 版本。我们采用两种不同的方法来选择 Node.js 版本。首选方法是选择 package.json 文件的 engines→node 字段中指定的相同 Node.js 版本。[9] 此字段允许开发者手动指定构建特定发布版本时运行相关代码所使用的 Node.js 版本。如果未设置此字段，我们就选择客户端包发布版本发布时可用的最新 Node.js 版本。[10] 因此，我们更改 Node.js 版本，执行安装脚本，并分别使用 npm install 和 npm test 命令运行测试。如果安装或测试命令因与所选 Node.js 版本不兼容而失败，或耗时超过 10 分钟，我们就切换到前一个 Node.js 主版本，直到安装和测试命令成功为止。我们使用 Node 版本管理器（NVM）工具切换 Node.js 版本。此外，我们还根据 Node.js 版本更改了 npm 版本。npm 是用于管理 Node.js 包的包管理器，并负责执行安装和测试脚本。在安装和测试过程中，我们采用相同的流程选择要使用的 npm 版本。最后，我们执行安装/测试脚本，并记录每个客户端版本的结果（成功或错误）。

执行样本中 384 个客户端包的安装/测试脚本后，我们剔除了 33 个包，因为错误导致这些包的任何发布版本都无法执行安装/测试脚本：15 个客户端缺少某个必需文件；11 个包的测试脚本无效（例如，{"test": "no test"}）；4 个包在*.gitignore* 文件中列出了某些必需文件——该文件用于指定 Git 应忽略的未跟踪文件；[11] 2 个包需要无法配置的特定数据库配置；还有 1 个包需要密钥才能访问服务器。我们按照前述标准随机替换了这 33 个包。

表1 展示了对 384 个客户端包及其 3,230 个发布版本执行安装/测试脚本的结果。由于 2,727 个发布版本关联的提供方版本没有变化，因此没有对这些发布版本执行测试。最后，我们将所有客户端包和发布版本的安装/测试脚本均执行失败的情况视为可能显现破坏性变更的案例。

表1. 安装/测试脚本的执行结果。

测试	客户端包	发布版本
已执行	384	3,230
未执行	0	2,727
成功	181	1,954
失败	203	1,276
总计	384	5,957

包含我们的客户端包样本、测量工具、脚本以及已识别出的显现的破坏性变更的复现实验包，可从以下网址下载：https://doi.org/10.5281/zenodo.5558085。

[8] https://github.com/nodejs/node#release-types [9] https://docs.npmjs.com/files/package.json#engines [10] https://nodejs.org/en/download/releases [11] https://git-scm.com/docs/gitignore 提交至 ACM 的稿件

3.1.3 失败案例的人工检查：检测显现的破坏性变更。对于执行安装/测试脚本时出现的全部失败案例（203 个客户端和 1,276 个发布版本），我们人工分析了其中哪些确实是显现破坏性变更的案例。为了识别在客户端包中显现出来的破坏性变更，我们利用了执行第 npm 运行 安装 和 测试 脚本时生成的输出日志。这些日志是执行第 3.1.2 节所述方法的结果（见图 3的后半部分）。对于每个测试失败结果，我们都获取了错误描述及其关联的堆栈跟踪。随后，我们区分了由客户端包相关问题（例如，客户端引入的缺陷）导致的测试失败结果，以及由提供方包变更（例如，提供方函数返回类型发生变化）导致的测试失败结果。根据获取的堆栈跟踪，我们判断是否调用了提供方包中的任何函数，并对肯定的案例进行了人工调查。在人工调查过程中，我们力求确认测试失败是由提供方包引入的显现的破坏性变更所致。

第一作者负责运行测试，并识别显现的破坏性变更及相关发布版本和提交。第一作者还对每项显现的破坏性变更进行了人工分析，并记录了以下信息：受影响的客户端版本数量；是否有任何文档提及该显现的破坏性变更；负责处理该破坏性变更的包（提供方或客户端）；受该显现的破坏性变更影响的客户端版本；引入该破坏性变更的提供方版本；以及对破坏性变更显现原因的文字说明（例如，“提供方函数被误改了名称”，“提供方 normalizeurl@1.0.0 引入[了]一个新函数，客户端 assetgraph 使用[了] 它。但客户端忘记在 package.json 中更新提供方版本。”，“提供方在空的请求正文中插入了一个空格”）。在此过程中，作者之间进行了多轮讨论，运用持续比较 [22] 和协商一致 [13]来完善分析。在协商一致过程中，研究人员讨论了用于对每段代码进行分类的依据，直至达成共识 [13]。更具体地说，我们利用记录的每项显现的破坏性变更的信息，对引入的破坏性变更进行一致分类（研究问题2和研究问题3），并指导新一轮人工分析。

更具体地说，我们在手动调查期间执行了以下一系列操作：

• **分析执行流程：**为了确定与测试失败相关的函数是在提供方代码中还是客户端代码中执行的，我们利用堆栈跟踪来识别测试失败时调用的是哪个函数。具体而言，我们在提供方和客户端包的代码中添加插桩，以输出分析执行流程所需的信息。我们在客户端包调用提供方函数的代码各处添加对 console.log() 和 console.trace() 函数的调用，以分析变量内容。例如，假设出现了以下错误：“类型错误：*myObject.callback* 不是一个函数”。为了查看变量的内容，我们使用命令 console.log(myObject) 检查 myObject 变量是否已更改、是否为 null，或是否接收到了其他值。

• **分析持续集成（CI）流水线的状态：**我们比较了最初构建的发布版本的 CI 流水线状态与手动调查时 CI 流水线的状态。由于在原始发布版本和我们分析中安装的版本之间，客户端包的源代码保持不变，因此我们将 CI 流水线状态的差异作为额外证据，证明测试失败是由提供方版本变更导致的。并非所有客户端都有 CI 流水线，但对于有 CI 流水线的客户端而言，这项信息很有帮助。

• **搜索客户端修复提交：**我们手动搜索了客户端包已安装版本与先前发布版本之间的提交历史，查找修复提交。每当我们识别出一条修复提交时

（通过阅读提交信息），我们就会判断错误是由客户端代码还是提供方代码导致的。例如，我们发现了一些案例：客户端在一次测试失败的发布版本中更新了提供方。我们还观察到，在后续提交中，提供方被降级，且提交信息为“降级提供方”或“修复破坏性变更”。在这些案例中，我们认为测试失败是由显现的破坏性变更导致的。

• **搜索相关问题报告和拉取请求：**我们假设，显现的破坏性变更会影响不同的客户端，进而促使它们提交缺陷报告，或进行修复并向提供方包的代码库提交拉取请求。因此，我们搜索了包含堆栈跟踪中相同错误消息的问题报告和拉取请求。随后，我们收集了有关该错误的详细信息，以确认它是否由提供方包引入的显现的破坏性变更导致。

• **之前和之后的提供方版本：**如果测试错误是由显现的破坏性变更导致的，那么降级到之前的提供方版本或升级到之后的提供方版本可能会修复该错误，前提是提供方已经修复了它。之后的提供方版本是指所有符合版本声明且高于引入显现的破坏性变更的提供方版本（即测试失败时采用的提供方版本）。在这种情况下，我们卸载当前版本，安装之前和之后的版本，然后重新执行测试脚本。例如，如果客户端将提供方 p 指定为 {"p": "^1.0.2"}，而该版本引入了破坏性变更，例如 1.0.4，我们就安装 p@1.0.2 、 p@1.0.3 和 p@1.0.5，以验证这些版本是否仍会出现该错误。

3.2 研究问题：动机与方法

本节介绍各项研究问题的动机和研究方法。

3.2.1 研究问题***1***：显现的破坏性变更在客户端包中出现的程度如何？

动机： 默认情况下，npm 将插入符范围设为默认的版本声明，该声明会自动更新次版本和补丁版本。因此，在次版本和补丁版本发布中引入的显现的破坏性变更，可能会意外导致每天被下载数十万次的软件包发生停机，影响大量软件开发者。了解 npm 等热门软件生态系统中显现的破坏性变更的普遍程度，有助于开发者评估接受自动次版本和补丁版本更新的风险。尽管以往研究关注的是 API 破坏性变更的频率 [3], ，但破坏性变更也可能由不同原因引起。确定更广泛类型的破坏性变更的普遍程度，仍是一个尚待解决的研究问题。**方法：**对于所有导致安装 / 测试脚本出错的案例，我们都确定了错误类型（客户端、提供方、未发现）。在 384 个软件包和 3,230 个发布版本中，我们计算了经确认属于显现的破坏性变更的案例百分比。针对客户端最新发布版本中的所有提供方，我们计算了引入显现的破坏性变更的提供方百分比。此外，我们还计算了每个提供方至少引入一次显现的破坏性变更的次数（发布版本数）。

3.2.2 研究问题***2***：提供方包中的哪些问题会导致显现的破坏性变更？

研究动机： 此前关于 npm 生态系统中破坏性变更的研究仅限于 API 的破坏性变更[14]。然而，提供方包在次版本和补丁版本发布中引入的其他问题，也可能显现为破坏性变更。投稿至 ACM 的稿件

为帮助开发者分析显现的破坏性变更，了解其根本原因至关重要。

方法： 在此研究问题中，我们分析了引发显现的破坏性变更的提供方包所引入的变更类型。根据提供方包的名称和版本，我们手动分析了提供方的代码仓库，以找出导致破坏的确切变更。我们采用以下方法来查找提供方引入的具体变更：

- **使用差异工具：** 我们使用差异工具分析提供方两个发布版本之间引入的变更。例如，假设在发布版本 provider@1.2.5 中引入了显现的破坏性变更。在这种情况下，我们会获取先前版本（如 provider@1.2.4）的源代码，并对这些版本进行差异比较，以手动检查变更后的代码。

- **分析提供方的提交：** 我们利用提供方的提交分析各发布版本之间的变更。对于提供方 p 中的显现的破坏性变更，我们会检查其代码仓库，并手动分析引入显现的破坏性变更的发布标签提交之前或之后的提交。
- **分析变更日志：** 变更日志包含有关包历史记录中所有相关变更的信息。我们利用这些变更日志了解客户端包某个发布版本中引入的变更，并核实是否有记录显现的破坏性变更修复情况。

我们还查阅了问题报告和拉取请求，以了解显现的破坏性变更的成因。在发现引入破坏性变更的提供方变更后，我们对常见问题进行了分析、分类和归组。例如，所有与对象类型更改相关的问题都被归入名为对象类型变更的类别。此外，我们分析了在提供方和客户端包中引入并修复/恢复显现的破坏性变更的语义化版本级别，以核实显现的破坏性变更与非主版本发布之间的关系。

我们分析了修复显现的破坏性变更的发布版本编号，以及显现的破坏性变更记录在何处（变更日志、问题报告等）。此外，我们还分析了引入显现的破坏性变更的提供方的依赖树深度，因为 25% 的 npm 包至少有 95 个传递依赖项，这一点见 2016 [10]。

3.2.3 研究问题***3***。客户端包如何从显现的破坏性变更中恢复？

动机： 破坏性变更可能通过隐式或显式更新影响客户端包。客户端恢复可通过更新其代码、等待提供者的新发布版本，或对提供者版本执行降级/升级来实现。破坏性变更可能由直接或间接提供方引起，因为客户端包依赖少量直接依赖项和大量间接依赖项 [11]。如果未得到修复，破坏性变更可能级联至传递依赖。即使客户端包可以通过升级到较新版本的提供方包来从破坏性变更中恢复，客户端包仍可以手动解决可能存在的不兼容性 [12]。了解破坏性变更如何在客户端包中显现，有助于开发者理解如何从中恢复。

方法： 我们从客户端代码库中检索了与此研究问题相关的全部信息。我们搜索了有关错误以及客户端包如何从显现的破坏性变更中恢复的信息。我们分析了以下信息：

• **提交：** 我们手动检查了在引入相应显现的破坏性变更的提供方发布版本之后，推送到客户端包代码仓库中的后续提交。具体而言，我们搜索了修改 *package.json* 文件的提交。在文件历史中，我们检查了提供方是否被降级、升级、替换或移除。

• **变更日志：** 我们分析了客户端的变更日志和发布说明，查找其中提及提供方更新/降级的内容。约 48% 的客户端在其代码仓库中维护了变更日志或发布说明。
• **拉取请求/问题报告：** 我们在客户端代码仓库中搜索了包含显现的破坏性变更相关信息的拉取请求和问题报告。例如，我们找到了标题中包含“更新提供方”和“修复提供方错误”的拉取请求和问题报告。

针对每个显现的破坏性变更案例，我们还原了提供方的依赖树。例如，在我们的第二个启发性示例（第 2 节）中，我们还原了从客户端到引入显现的破坏性变更的包的依赖树，其结果为 broccoli-asset-rev→ broccoli-filter→broccoli-plugin (图 2）。我们调查了由直接依赖项和间接提供方引入的破坏性变更案例数量、显现的破坏性变更的引入和修复/恢复时间、修复/恢复该变更的包，以及修复/恢复方式。我们还核实了客户端包如何更改提供者版本，以及与显现的破坏性变更相关的文档与修复所需时间之间的关系。

3.3 范围与局限

由于我们对显现的破坏性变更的定义包含了先前对破坏性变更的定义未涵盖的案例（见第 2.1 节），本文无意直接比较这两种现象。因此，本文提出的研究问题并未说明清单破坏性变更中有多大比例实际上符合既有文献对破坏性变更的定义（例如，提供方进行的 API 变更）。此外，由于提供方包很少附带其预期行为的任何形式化规范，因此在我们的研究规模下，不可能区分客户端包中因破坏性变更而显现的错误，与因客户端包以特定方式使用提供方而显现的错误。因此，本文提出的研究问题的结果无法用于评估客户端包是否只需更新到提供方的较新版本即可修复构建。

4 结果

本节介绍各项研究问题的相关发现。

4.1 研究问题1：显现的破坏性变更在客户端包中出现的频率如何?

发现 1： ***11.7%*** 的客户端包（无论其发布版本如何）和 ***13.9%*** 的客户端发布版本受到了显现的破坏性变更的影响。 在全部 384 个客户端包中，有 45 个（11.7%）至少在一个发布版本中因显现的破坏性变更而出现失败测试。在执行了测试的 3,230 个客户端发布版本中，有 1,276 个失败，我们对全部错误进行了人工分析。在 450 个（13.9%）发布版本中，错误由提供方包引发，表明存在显现的破坏性变更。在 86 个（2.7%）发布版本中，我们无法确定是哪个包引发了错误。

我们检测到有 261 个（8.1%）发布版本遭遇了一种特定类型的错误，我们称之为因外部变更导致的破坏。这些发布版本使用了一个依赖外部 API/服务（例如 Twitter）所提供的数据/资源的提供方，而这些数据/资源已不再

可用，从而影响所有客户端版本。提供方无法修复此错误，因为它并不拥有该资源。这些案例表明，通过运行客户端测试来检测清单破坏性变更可能会引入假阳性，而我们在人工分析中直接忽略了这些假阳性。我们还考虑了提供方包从npm 中被移除，因而属于 因外部变更导致的破坏的案例。表 2 展示了按发布版本进行分析的结果。

表2. 版本分析的结果。

结果		**发布版本（数量）**	**(%)**
成功		1954	60.5
Fail	客户端错误	479	14.8
	显现的破坏性变更	450	13.9
	因外部变更导致的破坏	261	8.1
	未识别的错误	86	2.7
总计		**3230**	**100**

发现 2： ***92.2%*** 的提供方引入了一项显现的破坏性变更。 在我们的样本中，51 个提供方里有 47 个（92.2%）仅引入了一个包含显现的破坏性变更的发布版本，另有 4 个提供方引入了两个包含显现的破坏性变更的发布版本。我们检测到提供方引入的 55 个独特的显现的破坏性变更案例，其中一些影响了多个客户端。例如，归类为不兼容的提供方版本 （发现 3）的破坏性变更影响了 6 个客户端。因此，客户端包中显现了 64 个显现的破坏性变更案例。最后，所有客户端的最新版本中共有 1,909 个提供方，其中引入显现的破坏性变更的提供方占 2.6%（1,909 个中有 51 个）。

- 约有 11.7% 的客户端和 13.9% 的客户端发布版本受到显现的破坏性变更影响。
- 我们检测到，因 2% 的提供方进行了变更而导致测试失败。
- 在引入了显现的破坏性变更的对象中，超过 90% 仅通过一个包含显现的破坏性变更的发布版本引入了此类变更。

4.2 研究问题2：提供方包中的哪些问题导致破坏性变更显现?

发现 3： 我们发现了 ***8*** 类问题。我们根据根本原因（问题）将每个显现的破坏性变更归为八个类别。表 3 列出了各类别、发生次数以及受影响的客户端发布版本数量。

表3. 显现的破坏性变更的类别。

类别	案例		发布版本	
	(#)	(%)	(#)	(%)
功能变更	25	39.1	101	22.4
不兼容的提供方版本	15	23.4	64	14.2
对象类型变更	9	14.1	213	47.3
未定义对象	5	7.8	28	6.2
语义错误代码	5	7.8	14	3.1
提供方更新失败	2	3.1	24	5.3
重命名的函数	2	3.1	2	0.4
找不到文件	1	1.6	4	0.9
总计	**64**		**450**	

下面，我们将介绍各个类别，并展示一个在人工分析中发现的示例。

- **功能变更：** 此类别中显现的破坏性变更与提供方功能的修改有关（例如，变量的默认值）。一个例子发生在 request@2.17.0——此版本虽已从npm中移除，但引入的变更仍保留在包中——当开发者在代码中引入新的决策规则时[12] 如代码清单5所示。

代码清单 5。一个被归类为功能变更的显现的破坏性变更示例。

```
debug('emitting complete',self.uri.href)+ if(response.body == undefined && !self._json){+ response.body ="";+ }
self.emit('complete', response, response.body)
```

在代码清单 5中，提供方 request 将空字符串赋给 `response.body` 变量，而不是保留`response.body` 的默认`undefined` 值。

- **不兼容的提供方版本：** 在此类别中，客户端因间接提供方的变更而出现故障。一个例子发生在 babel-eslint 和 escope 这两个包中，其中 escope 是 babel-eslint 的间接提供方。

列表 6. 不兼容的提供方版本示例。

```
  }
-   },
-   visitClass: {
+ }, {
+   key: 'visitClass',
        值:     函数 visitClass(node) {
```

发布版本 escope@3.4 引入了代码清单6中所示的变更。此变更影响了包 babel-eslint，[13] 尽管 escope 此前并非 babel-eslint的直接提供方。[14] 这一显现的破坏性变更持续了一天仍未解决，在此期间，babel-eslint 从 *npm* 获得了约 80k 次下载。

[12] https://github.com/request/request/commit/d05b6ba [13] https://github.com/babel/babel-eslint/issues/ [14] https://github.com/estools/escope/issues/99#issuecomment-178151491 提交至 ACM 的稿件

- **对象类型变更：**我们检测到 9 个（14.06%）案例，其中提供方更改了对象的类型，导致客户端包发生破坏性变更。

列表7：对象类型变更示例。

```
    此.setup();-this.sockets  = [];+
this.sockets  ={};this.nsps  ={};this.connect
Buffer  = [];}var socket  = nsp.add(this,
function() {-   self.sockets.push(socket);+
self.sockets[socket.id] = socket;self.nsps[
nsp.name] = socket;
```

在列表 7中，提供方 socket.io@1.4.0 将数组转换为对象，[15]。这一简单变更破坏了许多socket.io的客户端，甚至包括包 karma，[16] 一个浏览器测试运行器，被迫更新其代码[17] 并发布 karma@0.13.19。在这一天内，显现的破坏性变更一直未得到解决，karma被从*146k* npm 下载了约npm。

- **未定义对象：**在此类别中，未定义对象会导致运行时异常，从而破坏提供方；提供方会将该异常抛给客户端包。

列表8。未定义对象代码示例。

```
+ 应用程序选项  =应用程序选项  || {};应用程序选项.babel  =应用程序选项.
babel  || {}; 应用程序选项.babel.plugins  =应用程序选项.babel.plugins  || [];
```

这个错误发生在提供方 ember-cli-htmlbars-inline-precompile@0.1.3中，该提供方按代码清单 8[18]所示解决了这个问题。

- **提供方更新失败：** 在此类别中，提供方A 更新其提供方B，但提供方A 没有更新其代码以适配新的提供方B。我们检测到此类别的两个案例。除显式更新外，此类别中的一个提供方A 将其提供方B 指定为接受全部 范围（>=）。随着时间推移，其提供方B 发布了一个引入显现的破坏性变更的主版本。尽管提供方A 指定了接受全部 范围，却没有考虑提供方B 的隐式更新，导致客户端遭遇错误。

- **语义错误代码：**此类别中显现的破坏性变更发生于提供方编写了语义错误代码，导致其运行时进程[19] 出错并影响客户端。这些错误在编译型语言中可以在编译时捕获，但在 JavaScript 中则会在运行时发生。这种情况出现在提供方 front-matter@0.2.0 及其他四个案例中。

列表9：语义错误代码示例

```
    。常量 分隔符  = [ '---','= yaml  =]; 常量模式  = '^('+
'((= yaml  =)|(---))'
```

[15]https://github.com/socketio/socket.io/commit/b73d9be [16] https://github.com/socketio/socket.io/issues/[17] https://github.com/karma-runner/karma/commit/3 6 18 78ab d https://github.com/ember-cli/ember-cli-htmlbars-inline-precompile/pull/5 95 19 3/commits/bfaf https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

在代码清单 9中，提供方在其声明中重复使用了变量名（模式），从而产生了语义错误。尽管这个错误很容易被发现并修复，正如提供方在[20] 代码清单 9中所做的那样，但提供方花了将近一年才修复它（front-matter@0.2.2）。与此同时，front-matter 在此期间获得了约 *366* 次下载。

- **重命名的函数：** 此类别中显现的破坏性变更发生在函数被重命名时。我们的分析发现了 2 个函数被重命名的案例。重命名案例是我们的第一个示例（第 2 节）；下面介绍第二个案例。

代码清单 10. 重命名函数的代码示例。

```
- RedisClient.prototype.send_command = 函数 (command, args, callback) {
- var args_copy, arg, prefix_keys;
+ RedisClient.prototype.internal_send_command = 函数 (command, args, callback) {
+ var arg, prefix_keys;
```

提供方 redis@2.6.0-1 重命名了一个函数，如代码清单10。[21] 然而，客户端包 fakeredis[22] 使用了这个函数，因此受到了此变更的影响。客户端包 fakeredis@1.0.3 通过将 redis 降级至 redis@2.6.0-0，从这个错误中恢复。[23] 在这项显现的破坏性变更尚未修复的五天内，fakeredis 从 npm 获得了约*2.3k* 次下载。

- **找不到文件：** 在此类别的案例中，提供方会移除某个文件，或将其添加到版本控制忽略列表（*.gitignore*）中，而客户端尝试访问该文件。在我们样本中此类别的唯一案例里，提供方引用了一个被添加到忽略列表中的文件。

发现 4： 显现的破坏性变更通常是在补丁版本发布中引入的。如表 4所示，在我们分析的 64 个显现的破坏性变更案例中，有 3 个案例是在主版本发布中引入的，26 个是在次版本发布中引入的，28 个是在补丁版本发布中引入的，5 个是在预发布版本中引入的。尽管我们只分析了次版本发布和补丁版本发布中的显现的破坏性变更，但有 3 个案例中的显现的破坏性变更是在间接提供方的主版本级别引入的，并由此传递性地影响了客户端包——例如 jsdom@16 案例（见第 2节）。

表 4. 每个语义化版本级别中的显现的破坏性变更 级别。

级别	(#)	(%)
主版本	3	4.7
次版本	28	43.75
补丁版本	28	43.75
预发布版本	5	7.8
总计	64	100

预发布版本先于稳定版本发布，且被视为不稳定；在稳定版本发布之前，任何内容都可能发生变化。[24] 在所有检测到的预发布版本中的破坏性变更中，提供方都在预发布版本中引入了不稳定变更

[20] https://github.com/jxson/front-matter/commit/f16 01 21 fc
https://github.com/NodeRedis/node-redis/commit/861749f [22]
https://github.com/NodeRedis/node-redis/issues/1030 205379483 23 #
issuecomment- https://github.com/hdachev/fakeredis/commit/01 99 24 1de
https://semver.org/#spec-item-9

并将这些变更传递到了稳定版本。一个例子是预发布版本 redis@2.6.0-1（详见第 3.2.2 节），其中函数重命名被传递到了稳定版本，导致客户端包发生故障。

发现5： 显现的破坏性变更的修复/恢复由客户端和/或提供方引入。 我们进行了搜索，以确定由哪个包（客户端还是提供方）修复/恢复了显现的破坏性变更，以及修复/恢复该变更的发布版本属于哪个级别，如图 4所示。

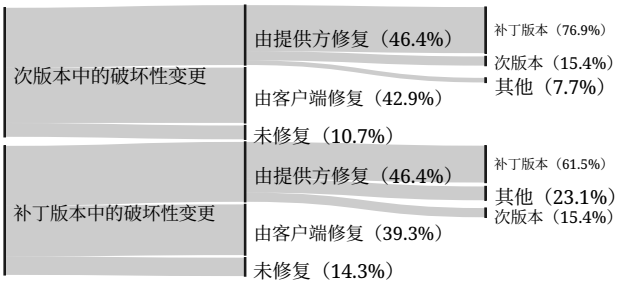


图4. 按提供方包和客户端包以及修复/恢复发布版本的相应语义化版本级别划分的显现的破坏性变更修复/恢复比例。

图 4 显示，客户端包会恢复近一半在次版本更新中引入的显现的破坏性变更。反过来，提供方在次版本发布中引入的显现的破坏性变更，有76.9%会在补丁版本发布中得到修复。提供方会修复补丁版本发布中引入的大多数显现的破坏性变更（占46.4%），通常通过补丁版本发布进行修复（占61.5%）。

发现6： ***21.9%***的显现的破坏性变更没有记录。 虽然客户端和提供方经常会在问题报告、拉取请求或变更日志中记录显现的破坏性变更的发生或修复情况，但仍有五分之一以上的显现的破坏性变更未被记录。

表5. 汇总　　显现的破坏性变更在引入时和修复/恢复时的记录比例　　.

文档	引入	修复/恢复	比例（%）
问题报告	–	32	64
拉取请求	5	15	44
变更日志	23	16	78

表 5 显示，客户端和提供方包在 78.1% 的所有显现的破坏性变更中记录了相关文档。在所有有文档记录的案例中，70% 包含不止一种类型的文档。例如，提供方收到了问题报告，修复了显现的破坏性变更，并将其记录在变更日志中。记录显现的破坏性变更及其修复有助于客户端恢复（第 3.2.3节）。

发现7： ***57.8%***的显现的破坏性变更由间接提供方引入。 间接提供方也可能引入显现的破坏性变更，随后这些变更可能传播到客户端。表6 展示了引入显现的破坏性变更的各提供方在依赖树中的深度级别。约42.2%的显现的破坏性变更

由客户端的*package.json*中直接依赖项引入。这些提供方是客户端直接安装并在自身代码中调用函数的提供方；它们位于依赖树的第一个深度级别。

表 6. 在依赖树中，引发显现的破坏性变更的提供方包与客户端之间的距离有多深。

深度	(#)	(%)
1	27	42.2
2	30	46.9
>3	7	10.9
总计	64	100

深度级别大于 1 的间接提供方引入的显现的破坏性变更占案例的 57.8%。其中，六个案例位于第三个深度级别，一个案例位于第四个深度级别。客户端不会直接安装这些提供方；相反，它们是由直接提供方引入的。在这些案例中，客户端包可能完全无法察觉显现的破坏性变更，因为它们通常并不知道这些提供方（或无法直接控制其安装）。

- 提供方包引入显现的破坏性变更时，最常见的问题是功能变更、不兼容的提供方以及对象类型变更。
- 提供方包在次版本发布和补丁版本发布中引入显现的破坏性变更的比率相近。

- 提供方修复的显现的破坏性变更中，大多数是在补丁版本发布中修复的。
- 78.1% 的案例记录了显现的破坏性变更，主要记录在问题报告中。
- 在大多数案例中，显现的破坏性变更是由间接提供方引入的。

4.3 研究问题3：客户端包如何从显现的破坏性变更中恢复？

发现8：在 39.1% 的案例中，客户端和传递依赖提供方会从破坏性变更中恢复。在依赖树中，传递依赖提供方位于引入显现的破坏性变更的提供方与该变更显现的客户端之间（参见第 2.1 节）。表 7 展示了针对每个显现的破坏性变更案例，哪个包进行了修复/恢复。提供方包修复了大多数显现的破坏性变更。由于破坏性变更是由它们引入的，因此从理论上讲，这正是预期的处理方式。20.3% 的案例中，客户端包从显现的破坏性变更中恢复；18.8% 的案例中，传递依赖提供方从显现的破坏性变更中恢复。当引入显现的破坏性变更的提供方未予修复时，传递依赖提供方可能会进行修复，从而解决客户端的问题。

表7. 修复错误/从错误中恢复的包。

修复者/恢复者	(#)	(%)
提供方	32	50
客户端	13	20.3
传递依赖提供方	12	18.8
客户端 + 传递依赖提供方	25	39.1
未修复/恢复	7	10.9
总计	64	100

由于传递依赖提供方也是引入显现的破坏性变更的提供方的客户端，因此，在 39.1% 的案例中，客户端（客户端和传递依赖提供方）从这些破坏性变更中恢复了过来。这一观察表明，引入显现的破坏性变更时，客户端包有时必须处理补丁版本，因为在 39.1% 的案例中，客户端和传递依赖提供方需要采取措施，以从显现的破坏性变更中恢复。**发现 9：**传递依赖提供方修复显现的破坏性变更的速度比其他包更快：引入显现的破坏性变更时，应由引入该变更的提供方或传递依赖提供方进行修复。在少数案例中，客户端包也会从中恢复。表 8 展示了各个包修复破坏性变更所需的时间。通常，提供方包会在七天内修复显现的破坏性变更。即使在这段相对较短的时间内，许多直接和间接客户端也会受到影响。

表 8. 各软件包修复/恢复显现的破坏性变更所用天数的中位数。

修复/恢复	Days
提供方	7
客户端	134
传递依赖提供方	4
客户端 + 传递依赖提供方	82.4

传递依赖提供方修复显现的破坏性变更的速度比客户端甚至提供方都快。由于显现的破坏性变更只有在客户端包中被提出时才会出现，传递依赖提供方会最先受到影响，因此需要快速修复；传递依赖提供方通常花四天时间修复破坏。与此同时，引入显现的破坏性变更的提供方，修复所需时间的中位数为 7 天。在提供方未能引入修复，或修复耗时超过客户端的情况下，客户端包从显现的破坏性变更中恢复所需的时间也相当长，为 134 天（平均值 286 天；标准差 429 天）。根据表 7，直接提供方和传递依赖提供方修复了大部分显现的破坏性变更，约占 78.8%，因为客户端的恢复速度可能较慢。

然而，由于传递依赖提供方本身也是客户端，我们可以分析客户端和传递依赖提供方修复/恢复显现的破坏性变更所花费的时间。客户端和传递依赖提供方大约用 82 天恢复了显现的破坏性变更。

发现 10： 升级是从显现的破坏性变更中恢复的最常见方式。 表9 描述了客户端如何从破坏性变更中恢复。在 48 个案例中，依赖项版本发生了变化。在大多数案例中，

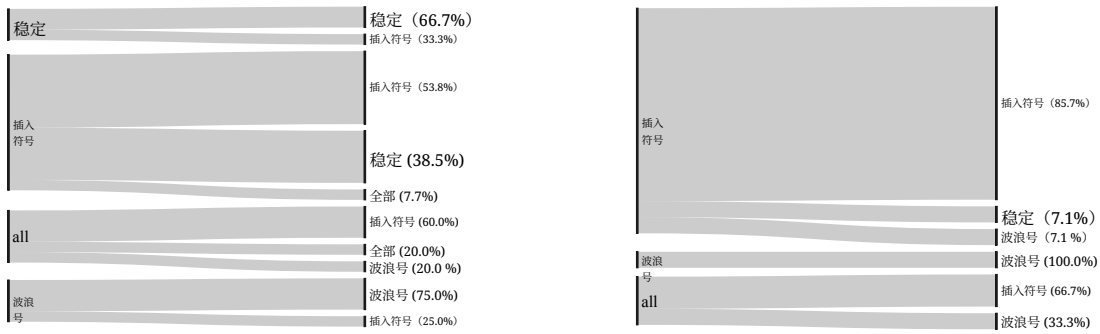
（71.4%）的客户端包升级了其提供方的版本。我们分析了所有客户端和传递依赖提供者在提供方修复错误之前，通过更改提供者版本而从显现的破坏性变更中恢复的案例。在 23 个案例中，有 12 个（52.2%）观察到了升级。因此，在客户端和传递依赖提供者修复/恢复显现的破坏性变更的案例中，超过一半的案例表明，提供方包已有更新版本，但客户端并未使用提供方包的任何后续版本。

表 9. 显现的破坏性变更后，客户端包如何更改提供方的版本。

更改方式	**升级总数**			**降级**		**替换**		**移除**	
		(#)	(%)	(#)	(%)	(#)	(%)	(#)	(%)
客户端	28	20	71.4	6	21.4	1	3.6	1	3.6
传递依赖提供方	20	9	45	10	50	01	5	—	—

传递依赖提供方中的降级次数，或许可以解释为什么它们比客户端包更快地从显现的破坏性变更中恢复。由于传递依赖提供方也是提供方，它们应尽快修复显现的破坏性变更，避免显现的破坏性变更所导致的错误传播。因此，将提供方降级到稳定版本，是传递依赖提供方从显现的破坏性变更中恢复的最常见方式。最后，发生破坏性变更时，只有少数情况下会替换或移除提供方——两种情况合计约占 7.2%。

发现 11： 为了从显现的破坏性变更中恢复，客户端通常会更改所采用的依赖项版本，而不更改自动接受的版本范围。当破坏性变更显现时，客户端通常会更新提供者版本。图 5 展示了客户端和传递依赖提供方何时更新其提供方版本。



(a) 客户端

(b) 传递依赖提供方

图 5. 客户端和传递依赖提供方更改提供者版本的情况。每张图的左侧显示引入显现的破坏性变更时的范围级别，右侧则显示修复同一显现的破坏性变更时的范围级别。

我们验证了传递依赖提供方从未将其提供方版本设为固定版本。当传递依赖提供方出现破坏性变更时，它们会在提供者版本中使用一个范围。不过，有一个传递依赖提供方将该范围从插入符范围改为固定范围（例如 ^1.2.1 → 1.2.1），以修复显现的破坏性变更。尽管如此，当客户端使用插入符范围且出现破坏性变更时，在 38.5% 的案例中，它们会将提供方降级为固定版本。

大多数显现的破坏性变更是在客户端和传递依赖提供方使用插入符号范围（^）时引入的。这是 npm 在将提供方添加为客户端包的依赖项时，默认插入到*package.json* 中的范围声明。在超过一半的案例中，这些客户端将提供者版本改为另一个插入符范围。较少使用接受全部范围（>= 或 *），更新时使用这种范围的情况也更少。

在 60.5% 的案例中，客户端和传递依赖提供方保留了范围类型并对其进行更新。范围类型（全部、插入符号、波浪号或稳定）保持不变，但提供方版本被更新或降级。例如，某个客户端包指定了提供方 p@^1.2.0，随后在 p@1.3.2 中遇到破坏性变更。每当提供方修复代码后，客户端包就会将其更新为例如 p@^1.4.0，但不会将其改为其他范围类型，例如全部、波浪号或稳定范围。

- 在 39.1% 的案例中，客户端包通过更新客户端和传递依赖提供方，修复了显现的破坏性变更。
- 提供方通过更新自身修复显现的破坏性变更的速度，比客户端包从显现的破坏性变更中恢复的速度更快；客户端更倾向于更新提供方，而不是将其降级。
- 发生破坏性变更后，可以更新或降级提供方的范围，但在约 60% 的案例中，范围类型并未改变。

5 讨论

本节讨论我们的研究发现对依赖管理实践的影响（第 5.1 节），以及客户端和提供方可以遵循哪些最佳实践来减轻显现的破坏性变更造成的影响（第 5.2 节）。我们还将讨论破坏性变更的显现，以及 npm 生态系统中的语义化版本相关方面（第 5.3 节）。

5.1 依赖管理

管理依赖项时，客户端包可以使用 GitHub 中的依赖机器人，例如 Snyk 和 Dependabot，以便在提供者发布新版本时收到自动拉取请求 [27]。这些机器人会持续检查是否有新版本，以及提供方是否修复了缺陷或漏洞。它们会在客户端的代码仓库中创建拉取请求，更新*package.json*，其中包含变更日志以及提供方新版本的信息。Mirhosseini 和 Parnin [16] 表明，使用此类机器人的包更新依赖项的速度比手动验证快 1.6 倍。此外，升级提供方发布版本时，JSFIX [20] 等工具也会有所帮助，尤其是当这些版本包含显现的破坏性变更或属于主版本发布时。JSFIX 工具旨在调整客户端代码以适配提供方的新版本，为升级提供方提供一种安全的方式。

我们证实，只有一小部分客户端通过移除或替换提供方来应对显现的破坏性变更（参见发现 10）；如果客户端使用了提供方包中的多项功能或资源，这种做法可能很困难 [2]。相反，客户端包往往会暂时降级到稳定的提供方版本。为简化提供方升级或降级流程并避免意外，客户端应查阅提供方的变更日志

针对重大变更。正如我们在发现6中所验证的，大多数显现的破坏性变更都会记录在变更日志、问题报告或拉取请求中。依赖机器人还可以分析变更日志和问题报告的内容，针对提及显现的破坏性变更的文档发出通知等警示。

最后，客户端包可以使用*package-lock.json* 文件来更好地管理依赖项。我们在发现7中观察到，间接提供方——即依赖树中第二层和第三层的提供方——导致了影响客户端包的显现的破坏性变更中的57.8%。借助 *package-lock.json* 文件，客户端包可以随时了解所有提供方最近一次成功构建所用的版本。当提供方因版本范围而升级，且新发布版本在客户端显现出破坏性变更时，客户端仍可安装所有曾成功构建该客户端的提供方版本。

5.2 最佳实践

通过使用静态分析工具，可以避免我们在人工分类显现的破坏性变更时发现的若干问题（第3.2.2节）。这类工具通常可以捕获被归类为语义错误代码和函数重命名的问题。客户端和提供方的开发者都可以使用这类工具。对于JavaScript这类动态语言，这些工具有助于避免一些问题 [26]。可供选择的JavaScript 工具包括jslint、jshint 和standard。Tómasdóttir 等人 [26] 以及 Tómasdóttir 等人 [25] 指出，开发者使用代码检查器主要是为了预防错误、缺陷和失误。

然而，由于 JavaScript 的动态特性，静态分析工具无法验证继承对象的属性。它们无法捕获归类为更改一条规则、对象类型更改和未定义对象的错误，也无法捕获对象属性函数中的函数重命名 。因此，开发者应重视创建测试用例，让代码与提供方的功能一同运行，因为只有这样，他们（客户端开发者）才能发现影响自身代码的破坏性变更。许多可用的框架，例如 mocha、chai 和 ava，都支持这些任务。每当开发者提交并推送新更改时，也应在集成环境中运行这些测试。对此，有多种可用工具，例如 Travis、Jenkins、Drone CI 和 Codefresh。借助代码检查器和持续集成系统，开发者可以在发布新版本之前捕获大多数此类错误。

最后，对于 npm 包而言，一个良好的实践是维护变更日志，或者在问题报告和拉取请求中记录破坏性变更及其修复方案。这一做法应当继续推行并得到更广泛的采用，因为目前约有五分之一的提供方并未这样做（参见发现6）。这也有助于开发用于处理破坏性变更的自动化工具（例如机器人）。提供方可以创建问题报告和拉取请求模板，以便客户端对发现的问题作出一致的描述。

5.3 破坏性变更的显现与语义化版本控制

破坏性变更在 npm 生态系统中经常发生，并影响客户端包（参见发现 1）。大多数显现的破坏性变更案例都来自间接提供方，也就是说，来自依赖树第二层或更深层的提供方。Decan 等人 [10] 的研究结果表明，2016 年，npm 中一半的客户端包至少有 22 个传递依赖（间接提供方），四分之一的客户端包至少有 95 个传递依赖。在这种情况下，客户端可能难以诊断显现的破坏性变更源自何处，因为当破坏性变更由间接提供方引入时，客户端可能并不知道该提供方。

我们的结果表明，提供方包会在次版本和补丁版本中引入显现的破坏性变更，而根据语义化版本规范，这些版本原则上只能包含向后兼容更新。语义化版本是一项建议，提供方可以选择是否采用 [4, 8]。如果提供方不遵守语义化版本规范，提交至 ACM 的稿件

就可能引入多种错误，正如我们在发现 4 中观察到的，预发布版本中所有显现的破坏性变更都会传递到稳定版本（参见发现 4）。一种假设是，提供方可能不了解语义化版本规则的正确用法，这或许可以解释为什么他们会将不稳定的变更传递到稳定版本。最后，npm 可以提供徽章，让提供方包能够明确表明自己了解并遵守语义化版本。Trockman [24] 指出，开发者会使用徽章等可见信号（特别是在 GitHub 上）来表明项目质量。这样一来，客户端就能更好地选择提供方，并优先选择了解语义化版本的提供方。

6 相关工作

本节介绍 npm 及其他生态系统中破坏性变更的相关工作。

破坏性变更发生于 npm： Bogart 等人 [5] 对 npm 和 CRAN 生态系统中依赖项的稳定性进行了调查。作者采访了七位软件包维护者，了解软件变更情况。本文中，受访者强调，遵循语义化版本有助于避免依赖项更新带来的问题。近期，作者又调查了 18 个软件生态系统中的政策与实践，发现全部生态系统都重视稳定性和兼容性等价值观，但在其他价值观上存在差异 [4]。Kraaijeveld [14] 研究了三个提供方包中的 API 破坏性变更。作者使用 *3k* 个客户端包，通过解析提供方和客户端的文件，检测 API 破坏性变更及其对客户端的影响。研究发现，9.8% 至 25.8% 的客户端发布版本受到 API 破坏性变更的影响。

Mezzetti 等人 [15] 提出了一种名为类型回归测试 的技术，用于验证 API 返回对象的类型，并将其与另一个提供方发布版本中的返回类型进行比较。作者选取了最受欢迎的 12 个提供方包及其主版本，并将该技术应用于第一个主版本更新中的全部补丁/次版本发布。他们在 9.4% 的次版本或补丁版本发布中验证出了类型回归。我们的研究聚焦于各种显现的破坏性变更，并分析了客户端包和提供方包；有 13.9% 的发布版本受到显现的破坏性变更影响。

Mujahid 等人 [19] 着重检测会导致破坏的第三方依赖项版本。作者分析了*29 万*个 npm 包。他们将提供方版本中的每次降级都标记为可能的破坏性变更。作者使用客户端测试对这些提供方版本进行了测试，并发现更新后有 4.1% 的测试失败，导致了降级。与这些作者类似，我们为每个发布版本解析了各客户端的提供方依赖项，但只要至少有一个提供方版本发生变化，我们就会运行测试。

Møller 等人 [17] 提出了一种工具，该工具使用提供方描述的破坏性变更模式，并修复客户端代码。他们分析了一个包含十个最常用 npm 包的数据集，并在变更日志中查找所描述的破坏性变更。我们可以将我们的分类（发现 3）与他们的分类进行比较。他们发现了 153 个在主版本发布中引入的破坏性变更案例。他们声称，大多数破坏性变更（85%）都与特定的软件包 API 项有关，例如模块、属性和函数变更。根据我们的分类（发现 3），功能变更、对象类型变更、未定义对象和重命名的函数也可以归类为软件包 API 变更；若如此，我们认为 64.06% 的显现的破坏性变更与软件包 API 有关。

其他生态系统中的破坏性变更：Brito 等人 [6] 在 116 天内研究了 Maven 仓库中的 400 个提供方。他们根据 GitHub 上的受欢迎程度选择提供方包，并查找了在此期间引入 API 破坏性变更的提交。他们询问开发者发生破坏性变更的原因。我们的论文得出了相似的结果：作者声称，新功能 是引入破坏性变更最常见的方式

而我们认为功能变更是主要的破坏性变更类型（发现 3）。此外，作者同样发现，破坏性变更经常会记录在变更日志中（发现6）。

Foo 等人。 [12] 介绍了一项关于 Maven、PyPI和 RubyGems 生态系统中 API 破坏性变更的研究。该研究重点关注通过计算两个发布版本代码之间的 差异来检测破坏性变更。他们在 26% 的提供方包中发现了 API 破坏性变更，其方法表明可以自动升级 10% 的包。我们的方法不局限于 API 破坏性变更；我们发现，11.7% 的客户端包受到显现的破坏性变更的影响。

7 效度威胁

内部效度：检测到破坏性变更时，我们会验证提供方包引入的变更类型，并将这些变更归类。然而，有些案例可能属于多个类别。例如，提供方包更改对象的类型，以改变或改进其行为。这种情况可能同时属于功能变更和对象类型变更。因此，我们将该案例归入最能代表该错误的类别。在这种情况下，由于对象是通过功能变更而改变的，最合适的类别是功能变更。

我们归类为因外部变更导致的破坏 的错误案例，是指客户端或提供方使用或依赖随时间发生变化的网站和 API 提供的外部数据/资源的情况（见发现 1）。这些案例约占客户端版本的 8.1%；在这些案例中，由于无法执行发布测试，我们无法查找显现的破坏性变更。毕竟，测试所需的数据/资源已不再可用。因此，约 8% 的客户端发布版本可能受到破坏性变更的影响，但我们无法对其进行分析。

构念效度：在我们检测破坏性变更的方法中，只有客户端测试失败时，我们才会进行分析。如果客户端使用的依赖项版本存在破坏性变更，但客户端没有调用导致该破坏性变更的函数，或者没有测试来执行相关代码，我们就无法检测到该破坏性变更。这就是为什么我们将所有案例称为显现的破坏性变更。

因此，我们可能没有检测到全部 API 破坏性变更，因为我们只能检测 API 名称变更和 API 移除。参数变更可能无法检测到，因为 JavaScript 允许使用任意数量的参数调用 API。[25]

对于每个发布版本，我们都将工作树索引恢复到开发者为该发布版本标记的相应提交。我们列出了代码仓库中的所有标签，并使用检出 相应的标签。不过，对于未标记的发布版本，我们会检出*package.json*中引用的时间戳对应的版本。在确认带标签的代码仓库中有 94% 的案例里，标签和时间戳指向同一提交后，我们便信任该时间戳。

最后，我们的研究没有提及 npm-shrinkwrap.json 文件。该文件旨在像 package-lock.json 文件一样控制传递依赖更新，但它可能会与软件包一同发布。不过，npm 强烈建议避免使用该文件。此外，npm-shrinkwrap.json 文件的存在对我们的研究并无太大影响，因为根据我们采用的研究方法，它们不会影响研究结果。因此，我们没有将其纳入研究。

外部效度：我们随机选取了在发布版本号、客户端、提供方和规模方面各不相同的客户端包。然而，由于我们只分析了托管在 GitHub 项目中的 npm 包，因此研究发现无法直接推广到其他情境。还需要指出，代表性也可能受到限制，因为 npm 每天都会增加

[25]https://eloquentJavaScript.net/03_函数.html#p_

包和发布版本的数量。未来的研究可以在其他平台和生态系统中重复我们的研究。最后，由于样本中的项目数量较少，我们没有足够的统计功效来对涉及包级比较的结果进行假设检验。

结论有效性：结论有效性是指由于数据样本量不足而无法得出统计显著结论。不过，由于我们的研究采用定性方法，因此我们通过对发布版本少于四个的全部客户端包的代码仓库进行合理性检查，降低了潜在的结论有效性威胁。这确保了所有包都计划用于生产环境（第3.1.2节）。最后，我们对本研究中声称显现的所有破坏性变更均进行了人工分析，以确认它们确实是会影响现实世界中客户端的破坏性变更（第3.1.3节）。

8 结论

软件复用是一种广泛采用的实践，而 npm 等包生态系统支持复用软件包。然而，破坏性变更是软件复用带来的负面影响。文献中已对多个软件生态系统中的破坏性变更及其影响进行了研究 [3, 6, 18, 28]。少数论文从客户端包的角度考察 npm 生态系统中的破坏性变更，即运行客户端测试以验证破坏性变更的影响[5, 15, 19]。在本研究中，我们从客户端和提供方的角度分析了 npm 生态系统中显现的破坏性变更，并对次版本和补丁版本中的破坏性变更进行了实证分析。

从客户端的角度出发，我们分析了显现的破坏性变更所造成的影响。我们发现，11.7% 的客户端会受到此类变更的影响，并提出了一些建议，帮助客户端和自动化工具开发者发现、避免并应对显现的破坏性变更。客户端可以使用依赖机器人加快升级其提供方的流程，也可以查看变更日志文件，留意任何不希望发生的更新，例如破坏性变更。从提供方的角度出发，我们分析了显现的破坏性变更最常见的成因。我们发现，最常见的原因包括：提供方更改了在过去几个发布版本中一直稳定的功能的某些规则或行为；对象类型发生变更；以及运行时出现了非预期的未定义对象。维护者在代码审查阶段应留意这些问题。未来的研究可以探讨软件包特征和度量指标与破坏性变更发生之间的相关性。

9 致谢

本研究　　部分由美国国家科学基金会通过编号为 IIS-1815503 的资助项目，以及 CNPq/MC 资助　　-TI/FNDCT（资助项目 #408812/2021-4）和 MCTIC/CGI/FAPESP（资助项目 #2021/06662-1）。

参考文献

[1] 2018。JavaScript 年度回顾：2018 年回顾及 npm 对 2019 年的预测。https://blog.npmjs.org/post/180868064080/this-year-in-javascript-2018-in-review-and-npms.html[2] Hussein Alrubaye 和 Mohamed Wiem Mkaouer。2018 年。《在函数级别自动检测第三方 Java 库迁移》。载于第 *28* 届计算机科学与软件工程年度国际会议论文集（*CASCON '18*）。加拿大安大略省万锦市，60–71 页。[3] Christopher Bogart、Christian Kästner、James Herbsleb 和 Ferdian Thung。2016 年。《如何破坏 API：三个软件生态系统中的成本协商与社区价值观》。载于*2016* 年第 *24* 届 *ACM SIGSOFT* 软件工程基础国际研讨会论文集（*FSE 2016*）。美国华盛顿州西雅图，109–120 页。https://doi.org/10.1145/2950290.2950325[4] Chris Bogart、Christian Kästner、James Herbsleb 和 Ferdian Thung。2021 年。《何时以及如何进行破坏性变更：18 个开源软件生态系统中的政策与实践。*ACM* 软件工程与方法论汇刊30 卷，4 期，文章编号 42（2021 年 7 月），56 页。https://doi.org/10.1145/3447245[5] C. Bogart、C. Kästner 和 J. Herbsleb。2015 年。《坏了就是坏了：生态系统开发者如何看待依赖项的稳定性》。载于*2015* 年第 *30* 届 *IEEE/ACM* 自动化软件工程国际会议研讨会（*ASEW*）。美国内布拉斯加州林肯市，86–89 页。https://doi.org/10.

1109/ASEW.2015.21
[6] Xavier, L., Brito, A., Hora 和 M. T. Valente。2018。Java 开发者为何以及如何破坏 API。载于《2018 年 IEEE 第 25 届国际软件分析、演化与再工程会议》（SANER）。意大利坎波巴索，穆利塞，255–265。[7] F. R. Cogo、G. A. Oliva 和 A. E. Hassan。2019。npm 生态系统中的依赖项降级实证研究。《IEEE 软件工程汇刊》（2019 年 11 月），1–13。[8] A. Decan 和 T. Mens。2019。包依赖项能告诉我们哪些关于语义化版本的信息？《IEEE 软件工程汇刊》（2019 年 5 月），1226–1240。[9] Alexandre Decan、Tom Mens 和 Maelick Claes。2016。包依赖网络拓扑：三种编程语言生态系统的比较。载于《第十届欧洲软件架构会议研讨会论文集》（ECSAW '16）。丹麦哥本哈根，文章 21，共 4 页。https://doi.org/10.1145/2993412.3003382[10] A. Decan、T. Mens 和 M. Claes。2017。开源软件包生态系统中依赖问题的实证比较。载于《2017 年 IEEE 第 24 届国际软件分析、演化与再工程会议》（SANER）。奥地利克拉根福，2–12。[11] Alexandre Decan、Tom Mens 和 Philippe Grosjean。2019。七种软件包生态系统中依赖网络演化的实证比较。《实证软件工程师》24，1（2019 年 2 月），381–416。https://doi.org/10.1007/s10664-017-9589-y[12] Darius Foo、Hendy Chua、Jason Yeo、Ming Yi Ang 和 Asankhaya Sharma。2018。高效的库更新静态检查。载于《2018 年第 26 届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集》。美国佛罗里达州布埃纳维斯塔湖，791–796。https://doi.org/10.1145/3236024.3275535[13] D Garrison、Martha Cleveland-Innes、Marguerite Koole 和 James Kappelman。2006。重新审视文本转录分析中的方法问题：协商式编码与可靠性。《互联网与高等教育》9，1（2006），1–8。[14] Michel Kraaijeveld。2017。检测 JavaScript API 中的破坏性变更。硕士论文。代尔夫特理工大学软件技术系，荷兰代尔夫特。http://resolver.tudelft.nl/uuid:56e646dc-d5c7-482b-8326-90e0de4ea419[15] Gianluca Mezzetti、Anders Møller 和 Martin Toldam Torp。2018。类型回归测试：检测 Node.js 库中的破坏性变更。载于《第 32 届欧洲面向对象编程会议论文集》（ECOOP）（莱布尼茨国际信息学会议论文集（LIPIcs））。荷兰阿姆斯特丹，7:1–7:24。[16] S. Mirhosseini 和 C. Parnin。2017。自动化拉取请求能否鼓励软件开发者升级过时的依赖项？载于《2017 年第 32 届 IEEE/ACM 自动化软件工程国际会议》（ASE）。伊利诺伊州厄巴纳-香槟，84–94。[17] Anders Møller、Benjamin Barslev Nielsen 和 Martin Toldam Torp。2020。检测 JavaScript 程序中受破坏性库变更影响的位置。《ACM 程序设计语言》4，OOPSLA，文章 187（2020 年 11 月），25 页。https://doi.org/10.1145/3428255[18] Anders Moller 和 Martin Torp。2019。对 Node.js 库中的破坏性变更进行基于模型的测试。载于《2019 年第 27 届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集》。爱沙尼亚塔林，409–419。https://doi.org/10.1145/3338906.3338940[19] Suhaib Mujahid、Rabe Abdalkareem、Emad Shihab 和 Shane Mcintosh。2020。使用他人的测试识别破坏性更新。载于《国际软件仓库挖掘会议》。https://doi.org/10.1145/3379597.3387476[20] Benjamin Barslev Nielsen、Martin Toldam Torp 和 Anders Møller。2021。用于使 JavaScript 程序适应演化中库的语义补丁。载于《第 43 届国际软件工程会议论文集》（ICSE）。[21] S. Raemaekers、A. van Deursen 和 J. Visser。2014。语义化版本与破坏性变更：Maven 仓库研究。载于《2014 年 IEEE 第 14 届源代码分析与操作国际工作会议》。加拿大不列颠哥伦比亚省维多利亚，215–224。https://doi.org/10.1109/SCAM.2014.30[22] Anselm Strauss 和 Juliet Corbin。1998。定性研究技术基础。加利福尼亚州千橡市：Sage 出版社。[23] Jacob Stringer、Amjed Tahir、Kelly Blincoe 和 Jens Dietrich。2020。主要包管理器中的依赖项技术滞后。载于《第 27 届亚太软件工程会议论文集》（APSEC）。228–237。https://doi.org/10.1109/APSEC51365.2020.00031[24] Asher Trockman。2018。为社交编程增添光彩：npm 生态系统中仓库徽章的实证研究。载于《2018 年 IEEE/ACM 第 40 届国际软件工程会议：伴随会议》（ICSE-Companion）。524–526。[25] K. F. Tómasdóttir、Maurício Aniche 和 Arie Deursen。2018。JavaScript 代码检查工具在实践中的采用：ESLint 案例研究。《IEEE 软件工程汇刊》PP（2018 年 9 月），26。https://doi.org/10.1109/TSE.2018.2871058[26] K. F. Tómasdóttir、M. Aniche 和 A. van Deursen。2017。JavaScript 开发者为何以及如何使用代码检查器。硕士论文。代尔夫特理工大学软件技术系，荷兰代尔夫特。[27] Mairieli Wessel、Bruno Mendes De Souza、Igor Steinmacher、Igor S Wiese、Ivanilton Polato、Ana Paula Chaves 和 Marco A Gerosa。2018。机器人之力：刻画并理解开源软件项目中的机器人。《ACM 人机交互论文集》2，CSCW（2018），1–19。[28] Jooyong Yi、Dawei Qi、Shin Hwei Tan 和 Abhik Roychoudhury。2013。通过软件变更契约表达并检查预期变更。载于《2013 年国际软件测试与分析研讨会论文集》（ISSTA 2013）。瑞士卢加诺，1–11。https: //doi.org/10.1145/2483760.2483772[29] Ahmed Zerouali、Eleni Constantinou、Tom Mens、Gregorio Robles 和 Jesus Gonzalez-Barahona。2018。npm 包依赖项中技术滞后的实证分析。https://doi.org/10.1007/978-3-319-90421-4_6