北京大學

本科生毕业论文

题目：**库依赖变化的大规模挖掘和应用**

Large-Scale Mining of Library Dependency Changes and its Applications

姓　　名：何　　昊

学　　号：1600012742

院　　系：信息科学技术学院

本科专业：计算机科学与技术

指导导师：周 明 辉

二〇二〇 年 五 月

北京大学本科毕业论文导师评阅表

学生姓名		学生学号		论文成绩	
学院（系）				学生所在专业	
导师姓名		导师单位/ 所在研究所		导师职称	
论文题目 （中、英文）					
导师评语 （包含对论文的性质、难度、分量、综合训练等是否符合培养目标的目的等评价） 导师签名： 年　　月　　日					

版权声明

任何收存和保管本论文各种版本的单位和个人，未经本论文作者同意，不得将本论文转借他人，亦不得随意复制、抄录、拍照或以任何方式传播。否则，引起有碍作者著作权之问题，将可能承担法律责任。

摘要

软件复用是现代软件开发中非常重要的一环。最常见的软件复用方式是在软件项目中添加对第三方库的依赖。随着开源软件的蓬勃发展，现代软件项目越来越多使用开源的第三方库来完成某项功能。这种复用既带来了便利，也带来了风险，特别地，项目开发中选择、升级和替换开源库都面临很大的困难，而解决这些困难需要已有开源项目中的依赖变化数据。这些数据既可以用于研究开发者管理库依赖的行为变化，也可以作为企业选择、升级和替换开源库的决策参考。然而，构建一个描述全部开源项目中的库依赖使用和变化历史的数据集并不容易。一方面，面对海量带有噪声的开源数据，数据集的构建算法必须高效、健壮（robust）和可规模化（scalable）。另一方面，为了支持多样化的下游分析任务，最终的数据集必须可靠和易于使用，并且能够持续更新。

有鉴于此，本文的目标在于探讨从大规模开源数据中构建库依赖变化历史数据集的可行性，以及此数据集的潜在应用场景。本文提出了一种并行增量式构建算法，并使用此算法，基于北京大学和美国田纳西大学合作构建的 World of Code 开源数据库，从中选择 60030 个 Java 开源项目（约 20M 个代码提交 Commit），构建了多种粒度下的完整库依赖变化历史，得到了一个原型数据集，数据集最终大小在 500GB 左右。进一步地，论文基于此数据集构造了三类应用场景，展现了应用该数据集提供服务的可行性和普适性：开源库的版本升级推荐、项目的依赖管理开发活动分析、和项目的库迁移情况分析。我们发现，Java 开源项目中的各种依赖管理活动，包括库添加、删除、版本改变和库迁移，都是很频繁的，值得进行更深入的研究。

关键词：开源软件 软件供应链 软件仓库挖掘 实证研究 库迁移

Abstract

Software reuse is critical for modern software development. The most common form of software reuse is by adding third party libraries as “dependencies” to a project. The flourishment of open source software has led to increasing adoption of open source software libraries. The reuse of software libraries is convenient but also risky. More specifically, projects face significant difficulties in choosing, updating and replacing dependent libraries, which can be mitigated by leveraging dependency change data in existing open source projects. A dataset of commit-level dependency changes can be used for in-depth study of dependency management behaviors in software projects, and can also serve as a foundation for industry supply chain management decisions. However, it is not trivial to build such a dataset on large corpus of open source projects. On one hand, the dataset construction algorithm must be robust to noisy data, highly efficient, and scalable to large number of projects. On the other hand, the final dataset must be reliable, easy to use and can be continuously updated.

The main objective of this paper is to study the feasibility of building such a dataset in large corpus of open source projects, and its potential applications. In this paper, we propose a parallel incremental algorithm for dataset construction, and build a prototype database based on 60030 Java projects extracted from World of Code data. The size of final dataset is about 500GB. Finally, we show the potential applications of this dataset through three application scenarios: version recommendation for library update, analysis of dependency management activities, and analysis of library migration in Java projects. We have identified intensive dependency management activities in terms of library addition, removal, migration and version changes, which can serve as a foundation for in-depth researches on the topic of dependency management.

Key Words： Open Source Software, Software Supply Chain, Mining Software Repositories, Empirical Study, Library Migration

全文目录

第一章 引言

1. 开源库的使用现状

现代软件开发离不开对已有软件的复用。已有研究显示，软件复用可以提升软件质量和开发效率，并减少软件交付所需时间，从而能够为软件公司带来整体的效益提升[1][2]。最常见的软件复用方式就是在软件项目中添加第三方库[1]，并利用第三方库提供的 API 来完成软件项目所需要完成的功能。随着开源软件的蓬勃发展和开源模式的成熟，产生了越来越多的开源库。于此同时，以 NPM[2]和 Maven[3]为代表的包管理平台的出现，不仅使得公司和独立开发者可以轻易地发布一个开源库，也使得开发者只需要编写一些配置文件，就能够轻易地在项目中引入一个开源第三方库。

现在，开源库的数量空前增长，并且软件项目也愈发的依赖开源库来完成某些功能。例如，2010 年时，Maven 中不同版本的库数量就已经超过了 260,000[4]，而现在更是超过了 4,000,000 个[5]。一项针对 GitHub 上一千余个大型 Java 项目的统计显示，每个 Java 项目平均会直接依赖 28 个开源库[3]，更不用说，这些开源库可能还会依赖其他的开源库，产生数量更多的间接依赖。项目与库，库与库之间构成了复杂的依赖关系。学术界和工业界会使用开源软件供应链一词，来指代软件项目所依赖的开源库的总和[4]。“供应链”一词形象地描述了开源库与开源库之间的复杂的依赖关系。

然而，开源库的大量使用也伴随着风险。开源库可能会存在质量问题和安全漏洞；开源库可能会被删除；开源库可能会被放弃维护；开源库可能无法满足项目的某些需求；等等。例如，OpenSSL 库中著名的 Heartbleed 漏洞影响了成千上万使用了这个库的网站和服务[6]。又例如，NPM 中一个只有 11 行代码的 leftpad

[1] 除了库（library），工业界还使用包（package）、组件（component）、框架（framework）、依赖（dependency）等用语来指代某个可复用的软件。由于这些概念之间并没有明显的分别，本文将会不加区分地使用这些用语，以指代一个出现在中心包管理平台（central package management platform）上出现的软件或软件模块。
[2] https://www.npmjs.com/
[3] http://maven.apache.org/
[4] https://blog.sonatype.com/2010/12/now-available-central-download-statistics-for-oss-projects/#.Vmgw77_2O7g
[5] https://search.maven.org/stats
[6] https://heartbleed.com/

库的突然删除，由于这个库被大量重要基础设施库使用，从而间接地使得无数下游软件和服务产生了问题[7]。至于放弃维护的开源库[5][6]，和开源库无法满足项目特定需求的例子更是比比皆是[7][8]。因此，库依赖管理的复杂性、开源库的脆弱性和安全性问题，愈发受到工业界和学术界关注。

2. 开源库相关的研究

近年来，软件工程学界对开源库相关的各种主题，进行了广泛的研究。一部分研究聚焦于大量开源库所形成的生态系统，例如刻画库生态系统的活跃度和库的流行度[9][10][11][12][13]；库依赖网络的结构性质[14][15][16][17]、升级延迟[18][19][20]和安全漏洞[21][22][23]；以及库生态系统的上下游之间的相互影响[6][24]、等等。另一部分研究则聚焦于开发者在管理开源库依赖时的行为，包括版本声明行为[26][27][28]、库选择行为[29][30][31]、库升级行为[32][33][34][35]、在相似功能的库之间的迁移行为[7][8][36]、等等。此外，学术界也提出了面向各种开源库管理需求的工具原型[37][38][39]。于此同时，工业界中也出现了各种支撑开源管理的功能和产品。例如，GitHub 可以显示依赖某个开源库的下游项目[8]，而且可以自动提醒开发者升级自己项目中具有安全漏洞的库[9]。又例如，随着 DevOps 在工业界的流行[40]，也出现了面向 DevOps 流程的软件供应链自动管理工具，可以自动化地检测开源库的安全性、许可证兼容性等问题[10]。

这些研究和应用都需要海量的开源数据进行支撑。早期的研究主要通过 GitHub 和 NPM 等网站提供的 Web API 来获取数据。由于这些网站的 Web API 通常会对单个客户端的访问加以流量限制，因此这种获取方式存在难以大规模获取，完整性难以保证的问题[41]。此外，未经处理的开源数据中，也可能存在诸如项目重复[42]、数据冗余[43]、开发者一人多名[44]等数据质量问题。因此，为了支撑学术界的研究和工业界的应用，出现了各种经过整理和清洗的公开数据集。软件供应链研究中常用的数据集包括：包含 GitHub 上项目和开发者信息的 GHTorrent[45]；包含了公开平台上出现的开源库，以及开源项目对这些库的使用

[7] https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/
[8] https://help.github.com/en/github/visualizing-repository-data-with-graphs
[9] https://help.github.com/en/github/managing-security-vulnerabilities
[10] https://www.sonatype.com/

情况信息的 Libraries.io[11]；以及收集了海量开源仓库数据的 World of Code[46]与 Software Heritage[47]。

然而，已有的数据集难以用于分析依赖管理的开发活动，特别是无法用于细粒度地分析开源项目的库使用历史。具体地说，GHTorrent 并不包含项目的 VCS 仓库数据；Libraries.io 虽然记录了库的全部历史版本和之间的依赖关系，却不包含开源仓库中库使用情况的变化历史，只包含了最新版本中的库使用情况；World of Code 虽然包含了海量开源仓库的开发活动数据，但是直接使用 World of Code 数据库进行开源项目库使用历史的分析是非常低效的。据笔者所知，不存在一个公开数据集，可以用于高效地回答以下这些问题：一个开源项目在何时开始使用一个开源库，又在何时不再使用这个开源库？当对库 A 的版本 X 进行升级时，最常见的升级版本是哪一个？开源项目在开发过程中添加、删除和升级库的频率如何？开发者的版本声明行为是如何随时间变化的？等等。如果有了一个这样的数据集，既可以用于研究开发者管理库依赖的行为变化，也可以作为企业选择、升级和替换开源库的决策参考。

3. 本文的工作

3.1 库依赖变化数据的挖掘

构建库依赖变化历史的数据集并不容易。开源项目可能具有大量代码源文件和历史版本，而数据集必须提供对每一个历史版本中的库使用情况的查询。如果一个项目的大小为 1GB，并有 10000 个历史版本的话，那么一个直接保存每个历史版本的简单数据集可能会消耗超过 10TB 的空间！所幸，开源项目的历史版本之间通常具有很强的相关性和信息冗余，因此可以从第一个版本开始,利用 Git Diff 提供的信息，记录每个版本所发生的变化，从而在查询时增量式地构建任意历史版本的完整库使用变更历史。此外，由于 Git 在设计之初并未考虑开发数据分析的需求，直接在 Git 仓库上进行开发历史分析是极其低效的，难以进行大规模分析。因此，本文选择面向软件工程研究的开源数据库 World of Code 作为数据集的原始数据来源。World of Code 使用分布式微服务架构，可以支持大规模

[11] https://libraries.io/data

的并行数据集构建算法。最后，由于在大规模开源数据挖掘中，缺失数据和噪音数据广泛存在，数据集的构建算法必须能够处理数据中的各种噪音，避免数据质量问题对最终的分析结果产生影响。

基于以上需求，本文提出了一种高效、健壮和可规模化的增量式库依赖变化历史构建算法。这个算法能够处理缺失和错误数据，而且可以大规模并行以处理海量数据。然后，本文基于 World of Code 数据库，以总大小为 1528GB 的 60139 个 Java 项目为输入，使用此算法构建了一个库依赖变化历史数据集原型。最终的数据经过压缩后约 500GB，并且可以高效支撑各种对开源项目的库使用历史的分析任务。

3.2 库依赖变化数据的应用

为了展现此数据集的有效性，本文使用此数据集进行了以下三种应用任务：库升级版本的推荐、项目的依赖管理开发活动分析和项目的库迁移情况分析。在库升级版本推荐任务中，使用库依赖变化数据，可以直接对开源项目版本升级历史数据的大规模挖掘，并针对一个版本给出不同历史时间段中各种项目中升级最多的版本；使用库依赖变化数据，可以对开源项目中添加、删除和升级库的情况进行分析，从而对开源项目的依赖管理活动的情况给出一个系统性刻画；最后，使用库依赖变化数据，可以对开源项目中相似库之间的迁移情况进行刻画和分析，从而有助于对库迁移的现象进行更深入的研究，以及对库迁移的目标进行推荐。综上所述，库依赖变化数据可以对多种多样的软件仓库挖掘任务提供支撑，从而可以作为学术研究和实际应用的基础。

4. 本文的组织

本文的余下部分是这样组织的：第二章首先会介绍与本文相关的背景知识，包括包管理器、Git 和 World of Code 等，然后会详细介绍相关工作，以及本文与相关工作的区别；第三章会详细介绍数据集的构建算法和最终数据集的特性；第四章会详细介绍几种数据集的应用场景，以及在这些应用场景上得到的结果；第五章会讨论本文的意义、不足和未来工作，并加以总结。

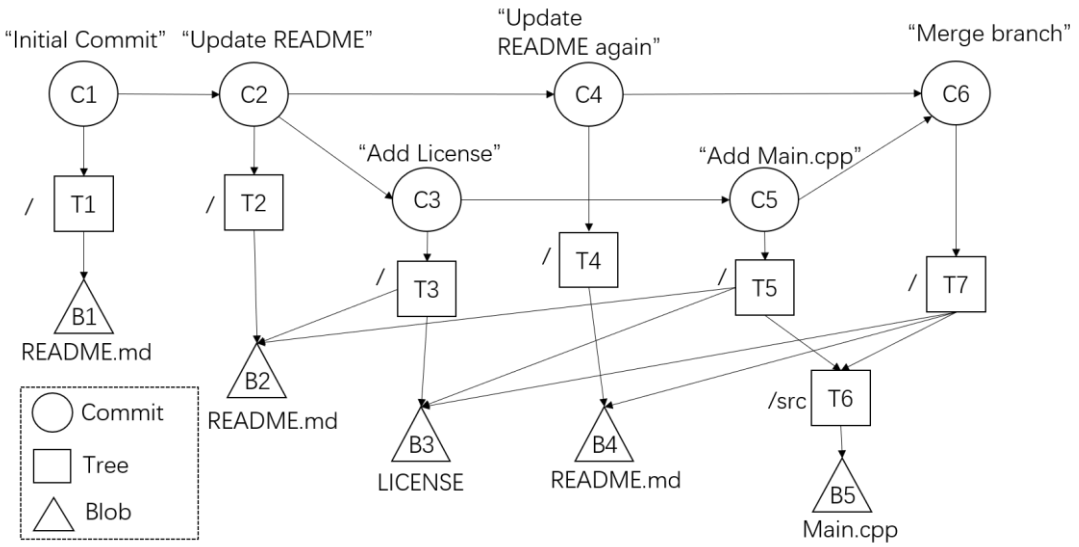

图 1 一个典型的 Git 仓库结构

第二章 工作背景

本章将会介绍与本文有关的背景知识和相关工作。我们会首先介绍 Git、包管理器和 World of Code，然后介绍与本文有关的已有研究，包括对软件供应链和库生态系统的研究；对开发者的依赖管理开发活动的研究；以及支撑上述研究的数据集。

1. 基础知识

1.1 Git 简介

任何一个大型软件项目的开发都离不开版本控制系统。使用版本控制系统，项目可以随时回退到任何一个历史版本，也可以支持多人合作协同开发。此外，任何对软件演化和开发活动的分析，都离不开版本控制系统所记录的软件开发历史。早期的主流版本控制系统是以 SVN 为代表的中心式版本控制系统[12]。由于中心式版本控制系统存在的诸多问题，目前的主流版本控制系统是以 Git 为代表的分布式版本控制系统[13]。除了少数古老的项目之外，几乎所有仍在积极维护的软件项目都使用 Git 进行版本管理。此外，分布式版本控制系统对于软件仓库挖

[12] https://subversion.apache.org/
[13] https://git-scm.com/

掘研究的最大优势在于，任何一个项目的副本中都包含了这个项目的完整开发历史。因此，当前的软件仓库挖掘研究的研究对象一般都是使用 Git 管理的软件仓库。

尽管 Git 一般用于管理软件项目，它事实上可以对一个文件夹下的任意一组文件和文件夹进行版本管理。Git 所管理的文件夹一般被称为 Git 仓库（repository）。为了进行版本管理，Git 会在仓库中建立一个数据库，用于管理以下三种 Git 对象：Commit，Tree，Blob，且每一个 Git 对象都具有唯一的 SHA1 标识符[49]。Git 对象之间会存在一些有向边，来表示对象之间的关系，但是不会出现环。有人将 Git 建立的这种数据库类型，称作 Merkle 有向无环图（DAG，Directed Acyclic Graph）数据库[14]。

图 1 是一个典型的 Git 仓库结构示意图。Git 使用 Commit 来表示被管理文件的一个历史版本。一个 Commit 可能会有一个或多个父 Commit，代表当前版本的上一个版本。如果这个版本由多个历史版本合并（merge）得到的，那么这个 Commit 会有多个父 Commit。同理，一个 Commit 也可能会有一个或多个子 Commit，表示这个 Commit 的下一个版本。如果同时有多人基于此 Commit 对文件进行了修改，并分别独立地提交，那么就会产生多个子 Commit。通过这种方式，Commit 与 Commit 之间形成了一个有向无环图。通常，我们使用 HEAD 来指代最新的历史版本，使用 TAIL 来指代最古老的历史版本。为了保存当前 Commit 下的全部文件版本，一个 Commit 会指向一个 Tree。Tree 对象对应于文件系统中的目录，可以嵌套地指向多个 Tree 和多个 Blob。每一个 Blob 对应于文件系统中的一个文件。由于 Tree 和 Blob 都以 SHA1 作为唯一标识，且多个 Commit/Tree 可以指向同一个 Tree/Blob，因此当 Commit 数很多而被修改的文件版本不多时，可以有效避免多个版本中相同文件和目录带来的冗余。如果想要知道两个 Commit 之间的区别，Git 通过在两个 Commit 各自的 Tree 上运行基于文本行的 Tree Diff 算法[15]，得到在哪些文件上添加和删除了哪些行的 GNU Diff 格式数据[16]。

[14] https://docs.ipfs.io/guides/concepts/merkle-dag/
[15] https://git-scm.com/docs/git-diff-tree
[16] https://www.gnu.org/software/diffutils/

1.2 包管理工具简介

在软件开发中，复制黏贴已有代码是最简单的软件复用方式。然而，代码的大量复制黏贴会给软件维护带来诸多问题，包括容易引入 Bug，难以进行更新和升级等等[50]。因此，现代软件工程的最佳实践通常要求尽可能把项目本身的代码与外部代码，特别是与第三方库的代码加以分离。为了满足这一需求，绝大多数高级编程语言都提供包管理工具。如果使用包管理工具，程序员只用通过编写

```
1.    <project>
2.    ...
3.      <properties>
4.        <mavenVersion>3.0</mavenVersion>
5.      </properties>
6.      <dependencies>
7.        <dependency>
8.          <groupId>org.apache.maven</groupId>
9.          <artifactId>maven-artifact</artifactId>
10.         <version>${mavenVersion}</version>
11.       </dependency>
12.       <dependency>
13.         <groupId>org.apache.maven</groupId>
14.         <artifactId>maven-core</artifactId>
15.         <version>${mavenVersion}</version>
16.       </dependency>
17.     </dependencies>
18.   ...
19.   </project>
```

图 2 Java 语言常用的包管理文件 pom.xml

一个配置文件，指定项目使用的库和其版本，就能够在编译和运行程序时，让包管理工具自动地从包管理平台上下载需要的库，并提供给程序调用。典型的包管理工具和平台有：Java 语言的 Maven、JavaScript 语言的 NPM、Python 语言的 pip 和 PyPI，等等。

图 2 是 Java 语言中，基于 Maven 的包管理文件 pom.xml 的例子。开发者可以在这个文件中添加`<dependency></dependency>`块，在其中以 `groupId`, `artifactId` 和 `version` 三个参数声明此项目需要使用的第三方库，从而可以在编译时，让 Maven 工具自动地下载这些库，与应用程序一起打包到最终生成的

JAR 包中。

不幸的是，绝大多数编程语言都没有强制要求开发者使用包管理器进行库依赖管理。不仅可能存在某个开源库在一个项目中使用了，却没有在包管理配置文件中声明；也可能包管理器中声明的库实际没有在项目中进行使用。另一方面，包管理器中包含的库的版本信息又是不可或缺的。因此，对项目的库使用情况的分析，往往必须将源代码与包管理配置文件相结合来进行。

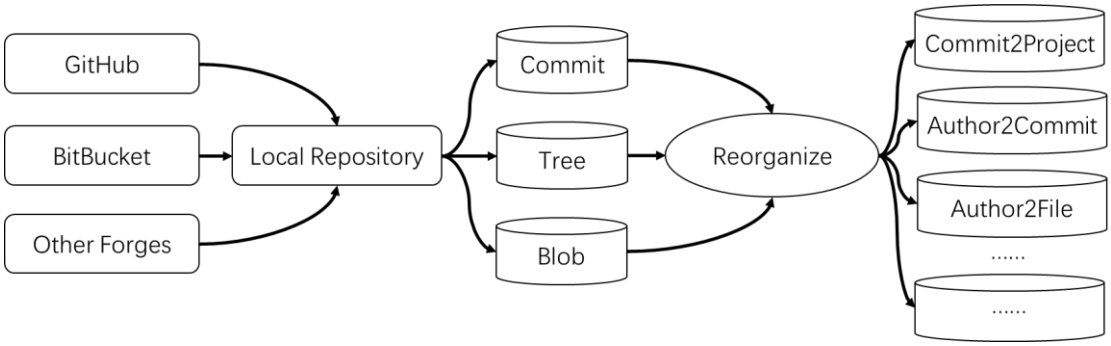

图 3 World of Code 架构示意图

1.3 World of Code 简介

正如前文所说，由于 Git 在设计之初并未考虑软件仓库挖掘和分析的需求，直接在 Git 仓库上进行开发历史分析是极其低效的，难以进行大规模分析；此外，直接挖掘 Git 仓库数据存在各种危险和陷阱[51]。World of Code（以下简称 WoC）是一个专门为大规模软件仓库挖掘和分析所设计，包含了海量开源软件的版本控制数据，并且可以不断更新的数据库[46]。图 3 展现了其构建数据集的基本流程。WoC 首先利用多种启发式方法，从 GitHub 等常见开源平台获取远程 Git 仓库信息；然后，采用一种定制的 git clone 协议，从远程仓库中增量式地获取和更新仓库数据，并将仓库数据分成 Commit、Tree 和 Blob 三类，以 SHA1 作为键值存放于键值对数据库中；最后，以这些原始 Git Object 为基础，将软件仓库挖掘任务所需要的各种数据分割成不同的键值对映射，例如 Author2Commit、Commit2Project、Author2File 等。

WoC 借鉴了分布式微服务架构的设计思想，将数据处理任务和最的数据尽

表 1 本文出现的 WoC 缩写及其解释

缩写	解释
2	To，用于表示映射关系，例如 b2c 表示 Blob to Commit
b	Blob，前文所介绍的 Git Object 的一种
c	Commit，前文所介绍的 Git Object 的一种
f	File，表示一个文件，更精确地说是 Tree 中所保存的 Blob 的文件名
p	Project，表示一个软件项目
pkg	Package，表示一个库
cg	Commit Graph，Commit 之间构成的 DAG，用于表示一个项目的开发历史

表 2 本文中用到的 WoC 全部输入数据

其中，{0-127}和{0-31}表示上述数据文件按 0-127 或者 0-31 编号分块存储

.tch 格式的文件是随机访问数据库，.s 和.gz 格式的文件是顺序访问的压缩纯文本数据

路径	说明
/fast/All.sha1o/sha1.blob_{0-127}.tch	SHA1 索引 Blob 数据
/fast/All.sha1o/sha1.commit_{0-127}.tch	SHA1 索引 Commit 数据
/da0_data/basemaps/p2cFull{ver}.{0-31}.tch	项目到 Commit 的映射
/da0_data/basemaps/gz/c2fbbFullR{0-127}.s	Diff 数据，以 Commit SHA1 排序
/da3_data/basemaps/gz/bbcfFullR{0-127}.s	Diff 数据，以新 Blob SHA1 排序
/da0_data/play/javathruMaps/bQjavapkgs.{0-127}.gz	.java 文件中 import 的 Class 数据

可能分割成独立的模块，使得每种数据都可以使用独立的程序，在不同的服务器上进行处理和更新。WoC 中最核心的数据库是保存从 SHA1 到 Commit/Tree/Blob 的原始 Git Object 数据库，由于数据的规模巨大，为了保证大量随机访问的高性能和低额外开销（low overhead），使用 Tokyo Cabinet[17]作为键值对数据库，并按 SHA1 的前两位十六进制数字进行分块，以控制每个独立键值对数据库的大小。除了随机访问数据库外，每种数据还包含以 gz 格式存储的纯文本压缩文件备份，不仅可以支持高效顺序访问，还可以在不同服务器上保存，从而可以尽可能降低意外事故带来的数据损失风险。

WoC 的微服务架构，使得任何人都可以使用任何技术栈，以 WoC 上存储的特定数据为输入，构建自己所需要的数据。目前，WoC 上已经存储了多种多样

[17] https://fallabs.com/tokyocabinet/

的，为了支撑各种不同软件仓库分析任务的扩展数据。其中，最常用的扩展数据是为了解决开源仓库数据中常见的数据质量问题的扩展数据，包括解决开发者一人多名问题的数据[50]、解决重复项目的数据[42]、等等。此外，本文还用到了一部分 WoC 中预计算的项目 API 使用情况的数据。本文所构建的数据集也是基于 WoC 构建的一种数据扩展。为了便于讨论，笔者会按照 WoC 中常用的命名缩写规则来编码输入和输出数据。表 1 列出了本文中用到的全部缩写及其解释。表 2 列出了本文中用到的 WoC 中的全部输入数据。

2. 相关工作

2.1 软件供应链/库生态系统

“软件供应链”（Software Supply Chain）一词很早就用于指代软件项目所复用的组件的集合[48]。近年来，随着开源软件的兴起和开源库管理平台的兴起，软件供应链一词现在主要用于从软件项目的角度，指代开源库与开源库之间形成的生态系统[4]。学术界已经存在大量对于库生态系统性质的研究。Wittern 等人研究了 NPM 库生态系统的情况和 GitHub 项目使用 NPM 库生态系统的情况，并探索了不同库流行度指标之间的区别[9]；Decan 等人先后研究了库生态系统的依赖网络结构和随时间的变化[16][17]，在库依赖网络中度量技术延迟（Technical Lag）的方法[18]，以及库安全漏洞在依赖网络中的影响[21]等问题；Zerouli 等人研究了在库依赖网络中度量技术债的方法和技术债在依赖网络中的分布情况[20]，还研究了带有旧版本 JavaScript 库的 Docker 镜像中的安全漏洞情况[23]；Kula 等人研究了在库生态系统中可视化库的流行度、接受情况和分布情况的方法[10]；Zimmermann 等人从安全研究的角度系统性地度量了 NPM 生态系统对安全漏洞的风险和脆弱性，并提出了若干解决方案[22]；Bogart 等人研究了库生态系统上下游之间对 API 兼容性的交涉情况[24]；Dey 等人研究了库的流行度与库生态系统提供的指标的相关性[11]；Soto-Valero 等人研究了 Maven 生态系统中库的版本多样性情况[19]；等等。我们的研究与上述研究是互补的关系，因为上述研究的对象是库生态系统，我们数据集的研究对象是利用库生态系统的软件项目，且关注点是软件项目在使用库生态系统时遇到的问题。

2.2 库依赖管理相关的开发活动

软件项目在开发时依赖开源库，虽然能够降低开发成本，但也会带来各种问题。因此，学术界也存在大量研究库依赖管理中存在的问题，以及如何解决这些问题的相关研究。

依赖管理的一大痛点是版本管理和库升级。新的版本可能会有不兼容的 API 和 Bug 带来迁移成本；而保持在旧的版本则存在安全风险。早期的研究主要关注于如何自动化地进行库升级。例如，Kapur 等人研究了如何自动化地在库升级中重构 Java 代码中对一个库的不兼容 API 的引用[55]。然而，一项对自动化库升级研究的回顾性实证研究指出，由于库升级中 API 不兼容场景的复杂性，已有方案几乎都只能在特定的个别场景下有效工作[56]。为了解决库升级场景的复杂问题，GitHub 创始人之一的 Tom Preston-Werner 提出并号召开源库开发者遵循 Semantic Versioning 的最佳实践[18]，通过一定的版本命名规则，来提示新版本与旧版本的区别，以便指导开发者在决策是否升级选择升级到的版本。已有证据显示，Semantic Versioning 已经在开源社区得到了广泛应用[27][28]。然而，Semantic Versioning 也存在规则过于理想化，难以完全得到遵循的问题。例如，Raemeaker 等人发现，尽管 Semantic Versioning 要求 MINOR 版本号不应引入不向后兼容的 API 变化，然而实际的 Maven Artifacts 常常因为种种原因在 MINOR 版本中引入了这样的 API 变化[57]。此外，开发者也会选择不升级一个库而保持在旧的版本。Kula 等人度量了开发者不升级库的情况[32]，探索旧版本库的实际安全漏洞影响[34]，并总结了开发者不升级库的种种原因[35]。另一方面，库与库之间甚至可能存在版本冲突问题[26]。以上研究表明，现实场景中开发者的库升级行为是非常复杂的，并且进行版本管理并不容易。

依赖管理的另一大痛点是库选择与库迁移。具体地说，开发者需要在各种开源库中，选择自己应该使用的库，然后在项目的演化过程中，可能由于各种原因，需要将正在使用的库迁移成另一个功能相同或者相似的库。为了解决库选择问题，Chen 等人研究了使用 Stack Overflow 数据自动化挖掘相似库的方法[38]；Thung 等人研究了如何基于项目正在使用的库推荐其他库的方法[3]；De la Mora 等人研究了如何使用软件供应链指标来比较功能相似的库[29]；Ma 等人研究了哪些因素

[18] https://semver.org

会影响 R 语言开发者选择不同的 Data Frame 库[31]；等等。对于项目演化过程中的库迁移问题，Teyton 等人研究了如何自动化地挖掘库迁移并使用迁移图来描述库迁移[8][36]；Kabinna 等人具体研究了 Apache 项目中 Logging 库之间的库迁移情况，发现性能、易用性、减少依赖和追求新功能都可能是库迁移的原因，库迁移可能会引入 Bug，且库迁移并不总能带来收益[7]。为了支持库迁移过程，Teyton 等人提出了一种自动化地匹配相似库 API 的方法[58]；Alrubaye 和 Chen 先后对此问题提出了改进方法[60][59]；Chen 指出相似库 API 对应关系相当复杂，并不能总是一一对应[59]。Xu 等人提出了一种直接从历史数据中学习代码修改，从而进行自动化库迁移的方法[61]。

我们的数据集可以作为上述对库依赖管理开发活动研究的基础。具体地说，从数据集中可以获得大量开发者在 Commit 粒度的库升级、选择和删除等数据，有力地支持和辅助上述相关主题的更深入研究。

2.3 面向上述研究的数据集

正如前文所说，早期的相关研究主要直接使用 GitHub 和包管理平台网站提供的 API 来获取所需数据。Kalliamvakou 等人指出虽然 GitHub 是一个宝贵的数据源，但从 GitHub 挖掘数据也存在一些困难和问题[41]。为了便于研究者获取和分析 GitHub 的数据，Gousios 等人提出并构建了 GHTorrent 数据集，用于收集和存储 GitHub 上存在的公开软件仓库的信息[45]。然而，GHTorrent 并不包含软件仓库的版本控制数据，这可能是因为收集这部分数据不仅需要很多硬件资源，也存在较大的工程难度。为了收集开源软件的版本控制数据以支撑软件仓库挖掘研究，Ma 等人提出并实现了 World of Code，采用分布式微服务架构收集开源仓库数据，并保存成适合用于进行软件仓库挖掘的数据格式[46]；同时，Pietri 等人出于保存软件数字遗产的动机，在同时期提出并实现了 Software Heritage，将开源仓库数据建模并保存为 Merkle DAG 数据库，并支持进行开源代码的高效存取和查询[47]。为了解决从包管理平台挖掘数据时的种种困难，Tidelift 公司收集并公开了 Libraries.io 数据集[19]，提供了 30 余个主流包管理平台中保存的开源库的基本信息，以及 GitHub、GitLab 和 BitBucket 平台上的软件仓库使用前述开源库的

[19] https://libraries.io/data

情况。同时，Libraries.io 数据集还提供了开源库的全部历史版本信息，以及库与库的不同版本之间的复杂依赖关系的信息。以上数据集被广泛运用于 2.1 节和 2.2 节提及的研究中。然而，以上数据集并不足以支撑对软件项目的库依赖管理变化历史的分析，因为 GHTorrent 和 Libraries.io 都不包含记录软件开发历史的版本控制数据，而 World of Code 虽然包含相关版本控制数据，但是却不能直接用于研究依赖管理的变化历史。因此，本文的主要贡献在于，基于 World of Code，构建了用于研究依赖管理开发活动的库依赖变化数据集。

第三章 库依赖变化数据的挖掘

本章将会介绍我们是如何从大规模开源数据中挖掘库依赖变化的。首先，我们会对库依赖变化挖掘问题给出形式化定义；然后，我们会给出挖掘算法描述，并对算法进行复杂度分析；最后，我们会介绍 World of Code 上具体实现架构和最后的数据集结构，并给出实现中的性能优化、错误处理方法和其他注意要点。

1. 问题定义

正如前文所说，本文的核心目标之一在于构建库依赖变化历史的大规模数据集。那么，到底什么是“库依赖变化历史”呢？本节将会从数学的角度，对数据集的构建目标给出定义。

数据集需要从一组软件项目中构建而得。假设我们关心的软件项目有n个，要构建的软件项目的集合为

$$\mathcal{P} = \{p_1, p_2, ..., p_n\}$$

对于任意一个项目$p \in \mathcal{P}$，由于项目p的全部 Commit 构成 DAG，定义

$$\begin{aligned} p &= \langle C, E \rangle \\ C &= \{c_1, c_2, ..., c_m\} \\ E &= \{\langle c_i, c_j \rangle | c_i \text{ is parent of } c_j\} \end{aligned}$$

其中C是此项目的全部 Commit 集合，E是 Commit 之间的有向边的集合。对于 Commit $c \in C$，每个 Commit 下的 Tree 可能会嵌套包含任意多个 Tree 和任意多个 Blob，但是由于我们只关心包含了库使用情况信息的 Blob 及其内容，因此定义

$$\begin{aligned} c &= \{\langle f_b, L_b \rangle | b \text{ is a blob in } c \text{ and } f_b \text{ is file path of } b\} \\ L_b &= \{\langle l, v \rangle | \langle l, v \rangle \text{ declared in } b,\ l \in \mathcal{L} \wedge v \in V_l\} \end{aligned}$$

其中，f_b表示 Blob 的文件路径，可以使用路径中的后缀名来区分不同类型的 Blob。对于特定的语言而言，我们只会关心与库依赖有关的少数几个文件类型，例如对于 Java 语言而言，如果我们只关心 pom.xml 文件和.java 文件，那么

$$f_b \in \{*/pom.xml, *.java\}，*\text{表示字符串通配符}$$

然后，L_b中包含了 Blob 中使用的库的信息，定义$\mathcal{L}$是我们关心的所有库的集合，V_l是库l的有效版本声明的集合。需要注意的是，由于现实中库的版本声明可能缺失，可能是一个特定的版本，也可能是一个特定的版本范围区间，因此上述声明都是V_l的合法成员。形式化地说，若库l的全部版本$V_l^{'} = \{v_1, v_2, ..., v_n\}$构成全序集，那么

$$V_l = \{\emptyset\} \cup \{v | v \in V_l^{'}\} \cup \{[v_1, v_2] | v_1 \in V_l^{'} \wedge v_2 \in V_l^{'} \wedge v_1 < v_2\}$$

对于一个项目，我们将这个项目的库依赖的**变化历史**定义为在 Commit 上发生的一系列**事件**。我们关心以下三种事件：添加一个库，删除一个库和改变一个库的版本。这些事件必须通过与上一个 Commit 进行比较得到。具体地说，对于某个 Commit，我们必须首先通过分析这个 Commit 下的所有 Blob，得知这个 Commit 中使用的所有库和其版本的集合，并将其与某一个父 Commit 中使用的所有库和其版本的集合进行比较。如果此 Commit 中有某个库而父 Commit 中没有，那么此 Commit 中发生了添加一个库的事件；如果此 Commit 中没有某个库而父 Commit 中有，那么此 Commit 中发生了删除一个库的事件；如果此 Commit 和父 Commit 中都有某个库，但是版本声明有所不同，那么此 Commit 中发生了改变一个库版本的事件。形式化地说，对于$\langle c_i, c_j \rangle \in E$，我们定义

$$L_i = \bigcup_{\langle f_b, L_b \rangle \in c_i} L_b$$
$$L_j = \bigcup_{\langle f_b, L_b \rangle \in c_j} L_b$$
$$Added(c_j) = \{\langle l, v \rangle | \langle l, * \rangle \notin L_i \wedge \langle l, v \rangle \in L_j\}$$
$$Removed(c_j) = \{\langle l, v \rangle | \langle l, v \rangle \in L_i \wedge \langle l, * \rangle \notin L_j\}$$
$$VerChgd(c_j) = \{\langle l, v_1, v_2 \rangle | \langle l, v_1 \rangle \in L_i \wedge \langle l, v_2 \rangle \in L_j\}$$

其中，$\langle l, * \rangle \in L$表示，只要库$l$的任意一个版本属于$L$，此式就为真。

基于以上定义，我们定义数据集的构建问题如下：

- 输入：项目集合$\mathcal{P}$和库集合$\mathcal{L}$
- 输出：$Added(c),\ Removed(c),\ Verchgd(c),\ \forall c \in \bigcup_{p \in \mathcal{P}} p$

2. 算法描述

由于数据集的规模巨大，一个项目可能会有数万乃至数十万个 Commit，加

之 Git 中 Commit 与 Commit 之间存在巨大的信息冗余，因此如果直接按照上述定义，将每个 Commit 的库使用情况集合分别计算和存储，再进行库依赖变化的计算，在计算和存储开销上都是不可接受的。由于 Commit 之间构成的 DAG 的性质，可以使用增量式（incremental）的方法来进行开发历史的构建。算法的形式化描述见图 4。

Algorithm 1 Library Dependency Change History Construction

Input: $\mathcal{P}$, $\mathcal{L}$

Output: $Added(c),\ Removed(c),\ Verchgd(c),\ \forall c \in \bigcup_{p\in\mathcal{P}} p$

1: T is the set of file types. For Java, $T = \{pom.xml, .java\}$
2: **for** $p \in \mathcal{P}$ **do**
3: $\langle C, E\rangle = getCommitGraph(p)$
4: L is a Dict of Dict
5: **for** $t \in T,\ c \in C$ **do**
6: $L[t][c] = \emptyset$ ($L[t][c]$ is multiset)
7: **end for**
8: **for** $c \in C$ in topological order **do**
9: $c_p =$ one of c's parent
10: **if** $c_p = \emptyset$ **then**
11: $Added(c) = L[t][c] = \bigcup_{\langle f,L\rangle\in c\wedge type(f)=t} L$
12: $Removed(c) = VerChgd(c) = \emptyset$
13: **continue**
14: **end if**
15: $D(c, c_p) = \{\langle f, L_i, L_j\rangle | type(f) \in T \wedge \langle f, L_i\rangle \in c_p \wedge \langle f, L_j\rangle \in c\}$
16: **for** $t \in T$ **do**
17: $L[t][c] = L[t][c_p]$
18: **end for**
19: **for** $\langle f, L_i, L_j\rangle \in D(c, c_p)$ **do**
20: $t = type(f)$
21: $L[t][c] = (L[t][c] \cup L_j) - L_i$
22: **end for**
23: $Added(c) = \bigcup_{t\in T}\{\langle l, v\rangle | \langle l, *\rangle \notin L[t][c_p] \wedge \langle l, v\rangle \in L[t][c]\}$
24: $Removed(c) = \bigcup_{t\in T}\{\langle l, v\rangle | \langle l, v\rangle \in L[t][c_p] \wedge \langle l, *\rangle \notin L[t][c]\}$
25: $VerChgd(c) = \bigcup_{t\in T}\{\langle l, v_1, v_2\rangle | \langle l, v_1\rangle \in L[t][c_p] \wedge \langle l, v_2\rangle \in L[t][c]\}$
26: **end for**
27: **end for**

图 4 库依赖变化历史构建算法

这个算法的基本思想是，对于任意一个 Commit c，如果它的某个父 Commit c_p的完整的库使用情况$L[t][c_p]$已知，那么只用计算和这个父 Commit 的 Diff $D(c, c_p)$（第 15 行），就可以递推得到当前 Commit 的库使用情况$L[t][c]$（19-22 行），并且得到相比较于这个父 Commit，项目中发生的库添加、库删除和版本

变化事件（23-25 行）。由于 Commit 之间构成有向无环图，从而可以将全部 Commit 进行拓扑排序，并按照拓扑排序的顺序进行遍历，就可以保证在任意一个 Commit 中，前一个 Commit 的库使用情况一定已经被计算。值得一提的是，为了保证算法递推过程的正确性，$L[t][c]$必须被实现为多重集（允许重复元素）。

这个算法的性能关键在于第 15 行$D(c, c_p)$的计算（也就是 Commit 之间的两两 Diff 计算）。事实上，一个低效的 Tree Diff 算法可能会使得大规模数据集上，上述算法几乎永远无法完成。如果假设 Commit 之间的 Diff 已经预计算好，可以在常数时间内访问，Diff 的遍历可以在常数时间内完成，且一个 Commit 中使用的库的数量不超过一个不大的常数，那么这个算法的时间复杂度为$O(|\mathcal{P}||C|)$。此外，在计算中，由于一个 Commit 所拥有的父 Commit 数量一般不超过两个，且并行分支的数量不会特别巨大，因此在遍历C并进行递推计算时，若c的全部子 Commit c_c的$L[t][c_c]$已经被计算，就可以释放$L[t][c]$所占据的空间。如此一来，此算法的空间复杂度为$O(x|C|)$，x为项目中最大的并行分支数量。最后，由于项目之间的计算完全独立，此算法可以以项目作为基本单位，在计算集群上进行大规模并行计算。

3. 具体实现

在实现中，为了尽可能地接近上述分析中提及的理论时间和空间性能，必须预计算各种相关数据，并以合理的格式进行存储。图 5 展现了数据集的构建流程，图 6 展现了最终数据集的数据组织结构（data schema）。

首先，为了得到项目集合$\mathcal{P}$，我们使用 Libraries.io 数据集[20]中提供的 GitHub 仓库列表，参考已有研究的选择标准[52]，从中选择 Star 数大于 1、Contributor 数大于 1、且不是 Fork 的 Java 项目，共得到 62023 个 Java 项目；然后，利用 WoC 中的项目（p2cFull）和 Commit 数据（sha1.commit.tch），提取相关 Commit 的全部元信息，并构建得到每个项目的 Commit Graph $\langle C, E\rangle$（p2cg）；随后，利用 WoC 上预计算好的 Diff 信息（c2fbbFull），提取每个 Commit 的 Diff D_c；最后，为了得到每个 Blob 的文件名和使用的库情况信息，使用了 WoC 中预计算的，按 Blob SHA1 排序的 Diff 信息（b2bcfFull，注意 Blob 本身并不包含其文件名），过

[20] https://libraries.io/data

滤得到.java 和 pom.xml 后缀的 Blob，然后根据 Blob SHA1 取出其文件内容进行分析，得到每个 Blob 的库使用情况信息L_b。对于 pom.xml 文件而言，我们提取全部的 Maven Artifact 及其版本信息的二元组作为L_b；对于.java 文件，我们提取其在 import 语句中导入的所有 Java Class 作为L_b，并将版本信息留空[21]。在实际中，一个 POM Artifact 可能会对应一个或多个 Java 包。我们将构建 POM Artifact 和 Java 包之间的对应关系作为未来工作。有了以上全部输入后，再运行算法 1，得到每个 Commit 中库添加、删除和改变版本的信息。

上述数据构建流程中，每一步都可能会有出错和缺失数据。首先，Libraries.io 中记录的 GitHub 仓库信息可能在 WoC 中不存在。这可能是因为 WoC 没有发现这个 GitHub 项目，也可能是因为这个 GitHub 项目改名或者被删除了。由于这种情况不常见，且一般都不是比较著名的项目，因此我们选择简单地跳过这个 GitHub 仓库，最终得到 60030 个有效的项目。其次，在构建过程中涉及到的 Blob 数据可能缺失，也可能存在格式或语法错误而无法被分析。由于这种情况非常常见，且如果不加处理的话会显著影响最终的数据质量，因此在算法 1 中，如果 Commit c中的某些 Blob 内容缺失或者存在语法错误，那么$L[t][c]$中会原封不动地继承上一个 Blob 的库使用情况。在递推计算$L[t][c]$时，需要一个额外的词典，保存带有错误的 Blob SHA1 到上一个没有错误的 Blob 中的L_b的映射。

图 5 中，每一个圆柱体表示一个底层数据库。在实际存储时，每个数据库会包含两份拷贝，一份是用于随机访问的键值对数据库，另一份是用于顺序访问的压缩纯文本文件。为了简化对这些数据的访问和分析，在这些数据的基础上还封装了上层的访问接口。图 6 展现了经过封装后，最终数据集的数据组织结构。最终的数据集的相关参数参见表 3。

[21] 这一步我们直接使用了 WoC 上预计算好的 b2pkgJava 数据

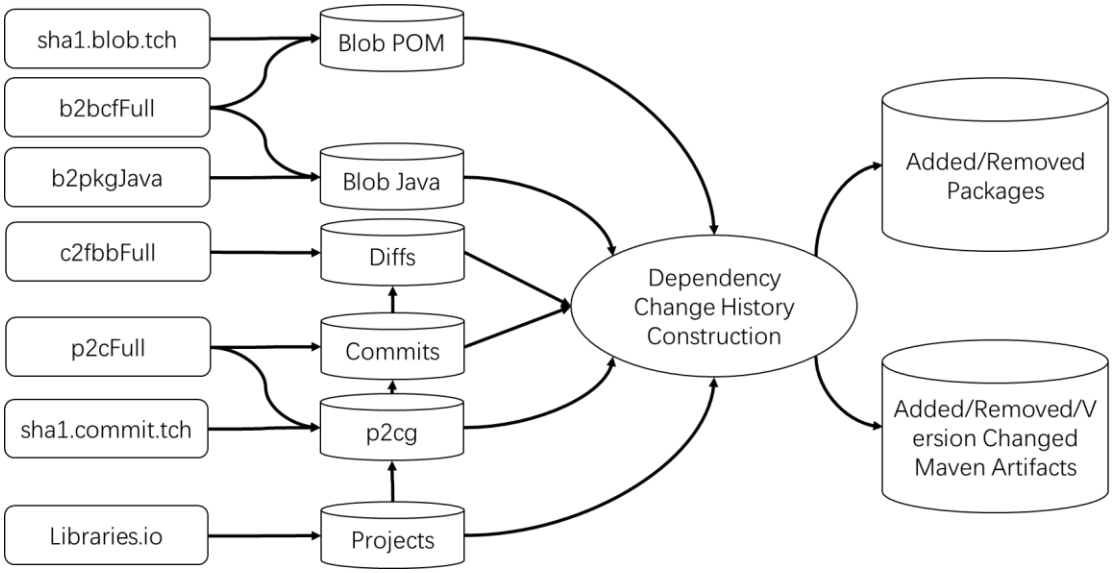

图 5 数据集的构建流程

其中，第一列是输入数据，第二列是中间数据。

关于输入数据的说明请参考第二章 1.3 节的表 1 和表 2。

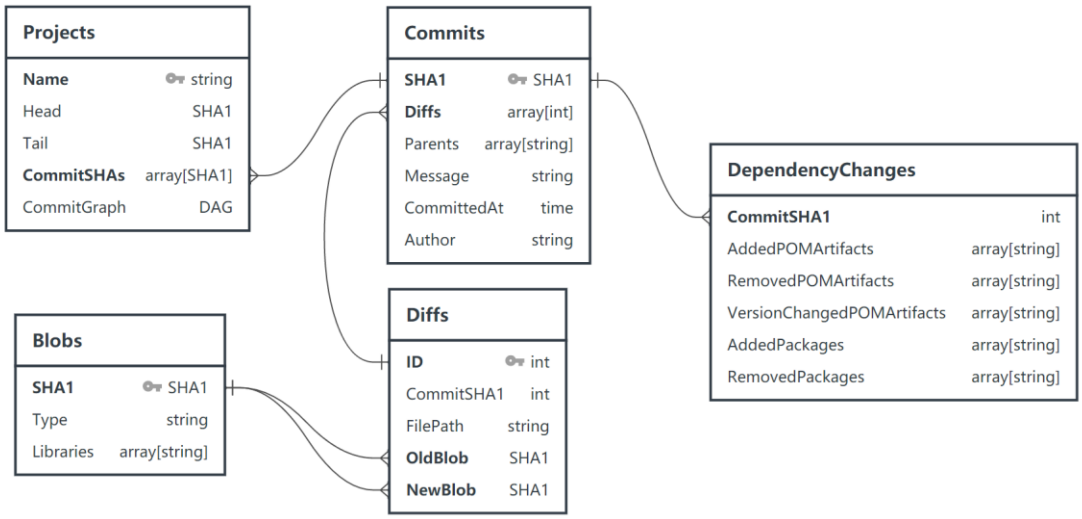

图 6 最终的数据组织结构示意图

表 3 最终的数据集参数

其中，总大小（顺序访问）指的是经过压缩的纯文本数据的大小

总大小（随机访问）指的是使用 TokyoCabinet 构建的键值对数据库的大小

此外，为了便于大规模并行读取，除项目数据外，其他数据都进行了分块处理

数据表	条目数	分块数	总大小（顺序访问）	总大小（随机访问）
Projects	60,139	1	8.1GB	11.5GB
Commits	26,466,383	128	8.0GB	39.8GB
Diffs	454,799,142	128	23.8GB	107.1GB
Dependency Changes	429,119,158	128	7.8GB	56.4GB
Java Blobs	416,703,279	128	48.5GB	217.6GB
POM Blobs	32,255,060	128	3.8GB	34.6GB

第四章 库依赖变化数据的应用

本章将会介绍库依赖变化数据的三种应用场景，以展现使用此数据集支撑下游应用和服务的可行性和普适性。第一个应用场景是开源库的版本升级推荐：给定一个库的版本，通过挖掘已有项目对这个版本的升级情况，给出应当升级到的库的推荐方案；第二个应用场景是项目的依赖开发活动分析：利用数据集，对开源项目中的添加、删除和库升级情况给出一个整体性的刻画，并对开源库的添加、删除、升级和留存情况给出一个整体性的刻画；第三个应用场景是项目的库迁移情况分析：利用数据集和已有研究的数据，对开源项目中相似库之间的迁移情况进行刻画。

1. 开源库的版本升级推荐

1.1 问题背景

拥有大量软件项目的大型科技企业通常都会使用某些科学的规章制度和管理流程，来保证软件项目的工程质量。对于软件项目的供应链管理也不例外。正如第一章所言，大型软件项目通常会使用大量的开源库，而开源库也可能会带来安全漏洞等问题。当一个库的版本被发现存在安全漏洞时，应当把库升级到没有这个安全漏洞的版本[22]。然而，实际中，往往并不容易决定开发中应当升级到的版本。很多时候，项目会出于各种原因保持在一个较老的版本[12]，盲目升级到过新的版本可能会产生 API 不兼容问题，而保持过老的版本可能会缺失一些功能[34]。为了进行科学的库版本迁移决策，需要客观的数据进行支持。

1.2 解决方案

我们认为，要进行库及其版本的推荐，可以采用“Crowd of Wisdom”式的推荐方案。具体地说，对于需求 1 而言，给定不能使用的库和版本$\langle l, v_1 \rangle$，我们可以从数据集中统计这个版本被迁移到的版本的情况及分布，即对如下版本集合进行分析

$$V = \{v_2 | \langle l, v_1, v_2 \rangle \in VerChgd(c), \forall c \in \mathcal{C}\}$$

其中，$\mathcal{C}$表示全部的 Commit 集合。对于以上每一个版本，我们可以统计其出现的比例，时间分布，Commit 信息和相关的项目，一并提供给项目开发者提供决策。

1.3 实验结果

我们选取 Jackson 库的 2.9.0 版本作为我们的研究对象。Jackson 库是 Java 中非常流行的，用于解析 JSON 的库。然而，2018 年底，Jackson 的一个模块 com.fasterxml.jackson.core:jackson-databind 的 2.9.x 版本被发现存在严重安全漏洞，直到 2.9.6 版本才得以解决[22]。因此，如果一个项目是 2.9.0 版本，应当对这个项目的版本进行升级。利用库依赖变化数据集，我们可以得到数据集中，从 2.9.0 版本中进行升级的所有 Commit，并进行上述分析，得到结果如下。

表 4 数据集中 2.9.0 版本的升级情况

全部数据		2019 年之后的数据	
目标版本	出现次数	目标版本	出现次数
2.9.2	62	2.9.8	22
2.9.5	44	2.10.0	9
2.9.7	23	2.9.5	3
2.9.8	22	2.10.1	3
2.9.1	13	2.9.9	1

表 5 数据集中最新的升级 Commit 信息举例

项目	Commit	升级版本	升级时间
HalBuilder/halbuilder-jaxrs	623a97	2.10.0	2019-11-02 23:05:46
Commit Message: Bump jackson.version from 2.9.0 to 2.10.0			
openmrs/openmrs-core	b6f0be	2.9.8	2019-05-21 16:33:44
Commit Message: TRUNK-5533: Upgrade com.fasterxml.jackson.core:jackson Libraries (#2902)			
sstrickx/yahoofinance-api	88a3ee2	2.9.8	2019-05-11, 14:11:54
Commit Message: Fix jackson-databind vulnerability			

[22] https://github.com/advisories/GHSA-qr7j-h6gg-jmgc

表 4 中显示了数据集中，使用了 jackson-databind 库的 2.9.0 版本的项目对 2.9.0 版本的升级情况。表 5 中举例了几个结果中最新的 Commit 及其相关信息。通过检查这些结果，我们可以发现，进入 2019 年之后，从 2.9.0 升级的最多的版本是 2.9.8 和 2.10.0，并且通过 Commit Message，我们也可以观察到，升级到 2.9.8 的原因之一是为了规避 jackson-databind 库 2.9.0 版本的漏洞。有了这些具体的输出结果，可以帮助项目的开发者和维护者对是否升级 2.9.0 版本的 jackson 库，以及应该升级到哪个版本提供帮助。

2. 项目的依赖管理开发活动分析

2.1 问题背景

虽然目前已经有了大量针对软件供应链特性的研究和针对依赖管理开发活动的研究，然而据笔者所知，目前尚没有对开源项目中的依赖管理相关开发活动，进行整体性量化分析的研究。利用库依赖变化数据集，可以对开源项目中的依赖管理开发活动情况进行整体性的刻画。这样的刻画，既可以帮助开发者改进当前的依赖管理开发活动，也可以为设计支撑依赖管理活动的开发工具提供思路。

为了对 Java 开源项目中依赖管理相关的开发活动情况进行一个整体性的刻画，我们针对 Java 项目和 Java 库提出以下研究问题：

- RQ1：项目中库的使用情况是什么样的？
- RQ2：项目的开发活动中，添加库、删除库和改变库版本的情况是什么样的？
- RQ3：项目的 Commit 数量和开发时间长短对项目的上述各种依赖管理开发活动有什么相关性？
- RQ4：不同的库在项目中的被添加、删除和版本改变情况有什么区别？

2.2 研究方法

首先，我们对于 Java 语言中的 Maven Artifact 和 Package 两种不同粒度，定义两种不同的库集合，分别记为$\mathcal{L}_m$和$\mathcal{L}_p$，并通过遍历所有的项目的开发历史数据计算得到$\mathcal{L}_m$和$\mathcal{L}_p$。由于从源代码中提取的 Java Package 不包含版本信息，因此下面计算中，与版本有关的指标都不会对$\mathcal{L}_p$进行计算。

为了回答 RQ1，对于$\mathcal{P}, \mathcal{L}_m, \mathcal{L}_p$，计算以下指标并进行可视化分析：

1. 对于所有$p \in \mathcal{P}$，项目最新的 Commit 中，使用的不同库的数量$LibNum_{head}$和版本的数量$VerNum_{head}$（对于 Java Package 只计算前者）

$$
\begin{aligned}
L_{head}(p) &= \bigcup_{\langle t_b, b\rangle \in c_{head}} L_b \\
VerNum_{head}(p) &= |L_{head}| \\
LibNum_{head}(p) &= |\{l | \langle l, * \rangle \in L_{head}\}|
\end{aligned}
$$

2. 对于所有$l \in \mathcal{L}_m, l \in \mathcal{L}_p$，在最新 Commit 中使用库$l$的项目个数

$$
DependentProjects(l) = |\{p | p \in \mathcal{P} \wedge l \in LibNum_{head}(p)\}|
$$

为了回答 RQ2，我们首先根据 RQ1 的结果选择具有 100 个以上的 Commit、开发时间大于半年、使用 10 个以上 Java Package 和使用 3 个以上 Maven Artifact 的项目子集$\mathcal{P}'$，然后对于每个项目$p \in \mathcal{P}'$，我们分别针对$\mathcal{L}_m$和$\mathcal{L}_p$计算以下指标，并对如下指标进行可视化分析

1. 项目的开发历史中，添加、删除和修改库的版本库的次数

$$
\begin{aligned}
TotalAdded(p) &= \sum_{c \in p} |Added(c)| \\
TotalRemoved(p) &= \sum_{c \in p} |Removed(c)| \\
TotalVerChgd(p) &= \sum_{c \in p} |VerChgd(c)|
\end{aligned}
$$

2. 项目的开发历史中，添加、删除和修改库的版本的 Commit 频率

$$
\begin{aligned}
AddIntensity_c(p) &= \frac{|TotalAdded(p)|}{|p|} \\
RemoveIntensity_c(p) &= \frac{|TotalRemoved(p)|}{|p|} \\
VerchgIntensity_c(p) &= \frac{|TotalVerchgd(p)|}{|p|}
\end{aligned}
$$

3. 项目的开发历史中，添加、删除和修改库的版本的时间频率

$$
\begin{aligned}
\Delta T &= \text{Time interval between HEAD and TAIL in days} \\
AddIntensity_t(p) &= \frac{|TotalAdded(p)|}{\Delta T} \\
RemoveIntensity_t(p) &= \frac{|TotalRemoved(p)|}{\Delta T} \\
VerchgIntensity_t(p) &= \frac{|TotalVerchgd(p)|}{\Delta T}
\end{aligned}
$$

4. 项目的开发历史中，平均对每个添加的库的版本修改次数

$$AvgVerChg(p) = \frac{TotalVerChgd(p)}{TotalAdded(p)}$$

5. 项目的开发历史中，平均每个被添加的库被删除的概率

$$ProbRemove(p) = \frac{TotalRemoved(p)}{TotalAdded(p)}$$

为了回答 RQ3，我们采用项目的开发总时长ΔT和 Commit 数量两种不同的指标来刻画项目的开发体量。然后，对于上述每一个指标，我们对其与ΔT和 Commit 数量计算 Spearman 相关性系数并进行比较。

为了回答 RQ4，我们首先根据 RQ1 的结果，选择在 10 个以上项目中得到使用的库子集$\mathcal{L}'_m$和$\mathcal{L}'_p$，因为这一部分更有可能是开源第三方库。然后，我们对于所有$l \in \mathcal{L}'_m$，计算如下指标并对其进行可视化分析

1. 这个库在项目中被添加、删除和改变版本的次数

$$\begin{aligned} TotalAdded(l) &= |\{p|p \in \mathcal{P} \wedge \exists c \in p, l \in Added(c)\}| \\ TotalRemoved(l) &= |\{p|p \in \mathcal{P} \wedge \exists c \in p, l \in Removed(c)\}| \\ TotalVerChgd(l) &= |\{p|p \in \mathcal{P} \wedge \exists c \in p, l \in VerChgd(c)\}| \end{aligned}$$

2. 库的升级频率

$$UpdateIntensity(l) = \frac{TotalVerChgd(l)}{TotalAdded(l)}$$

3. 库的留存率

$$Retention(l) = 1 - \frac{TotalRemoved(l)}{TotalAdded(l)}$$

由于从源代码中提取的 Java Package 并不包含版本信息，因此对所有$l \in \mathcal{L}'_p$，我们只计算库的添加、删除次数和留存率。此外，我们还选择库在数据集的每个项目的最新版本的 Commit 中被使用的次数、和库在数据集被添加的次数作为库的流行度指标，计算两种库的流行度指标与库的添加情况、删除情况、版本改变情况、升级频率和留存率计算 Spearman 相关性系数，并进行比较。

2.3 研究结果

2.3.1 Java 项目中库的使用情况

数据集中共有 60030 个 Java 项目（$|\mathcal{P}| = 60030$），这些 Java 项目共使用了

211596 个不同的 Maven Artifact（$|\mathcal{L}_m| = 211596$）和 1669058 个不同的 Java Package（$|\mathcal{L}_p| = 1669058$）。

表 6 项目集合$\mathcal{P}$中的开发活动情况和库使用情况

指标	均值	方差	最小值	25%分位	中位数	75%分位	最大值
Commit 数量	779.7	4827.5	1	35	103	329	347910
开发天数	1068.3	1162.7	1	159	714	1619	18338
Java Package	52.36	92.56	0	12	28	57	3846
$VerNum_{head}(p)$	10.37	38.14	0	0	0	7	2008
$LibNum_{head}(p)$	8.20	25.35	0	0	0	7	988

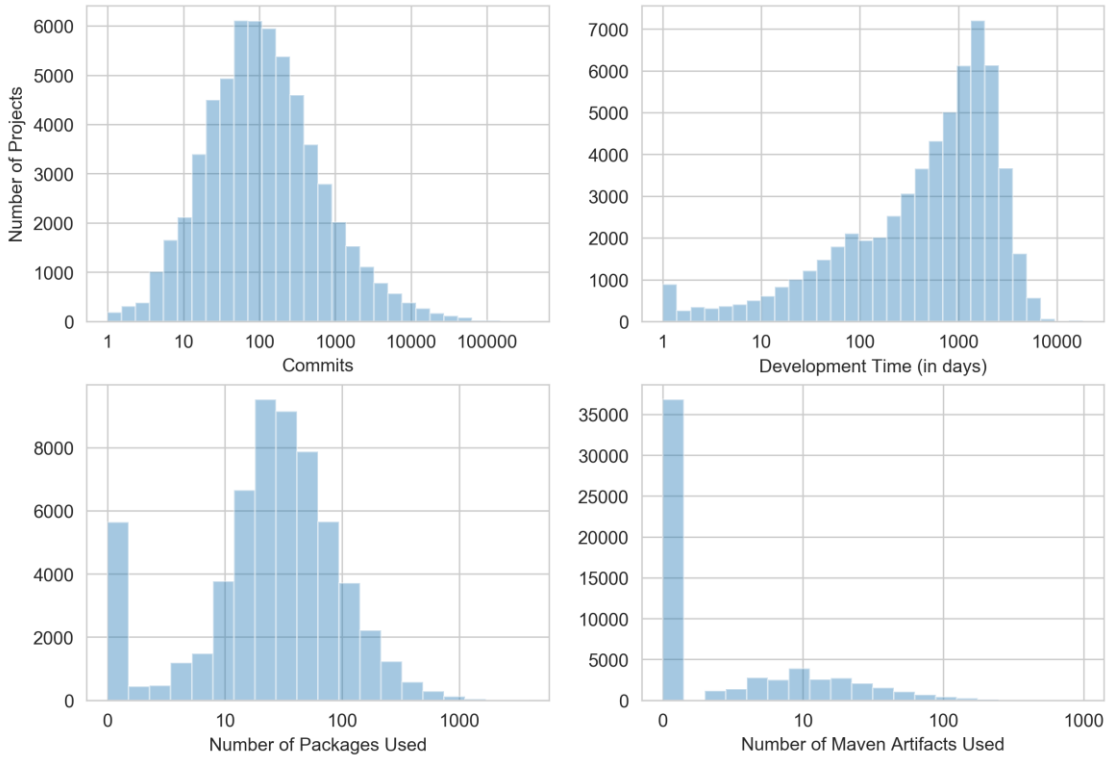

图 7 项目集合$\mathcal{P}$中的开发活动情况和库使用情况

表 6 显示 Java 项目集合$\mathcal{P}$中，Commit 数量、开发天数、使用的 Java Package 数量和 Maven Artifacts 数量的相关统计参数（后两行是使用 Maven Artifact 计算得到的结果）。图 7 显示了以上数据的分布情况（采用对数坐标）。我们可以看到，数据集中的 Java 项目在 Commit 数量上非常不均匀，而在开发时间上普遍较长；此外，Java 项目在使用的 Package 数量和 Maven Artifacts 数量上也呈现非常不均匀的情况，这可能与不同项目的项目体量和开发强度有关。另外，存在相当一部分项目同时使用一个库的多个版本，这个发现与一项最新的研究是一致的[53]。

值得一提的是，存在 9.37%（5635/60030）的项目没有使用任何 Java Package，这样的项目可能是不包含实际 Java 代码的教程类项目；存在 61.15%（36706/60030）的项目没有使用任何 Maven Artifact，这样的项目可能不使用第三方库，可能未使用包管理器，也有可能使用 Gradle 等其他包管理器来管理项目所使用的库。因此，在回答 RQ2 和 RQ3 时，应当排除开发时间较短、不使用第三方库和使用第三方库情况尚不清楚的项目。

表 7 Java Package 和 Maven Artifact 在项目最新版本中的使用情况

指标	*均值*	*方差*	*最小值*	*25%分位*	*中位数*	*75%分位*	*最大值*
Maven Artifact	2.30	48.70	0	0	0	1	14522
Java Package	1.88	86.62	0	0	1	1	50641

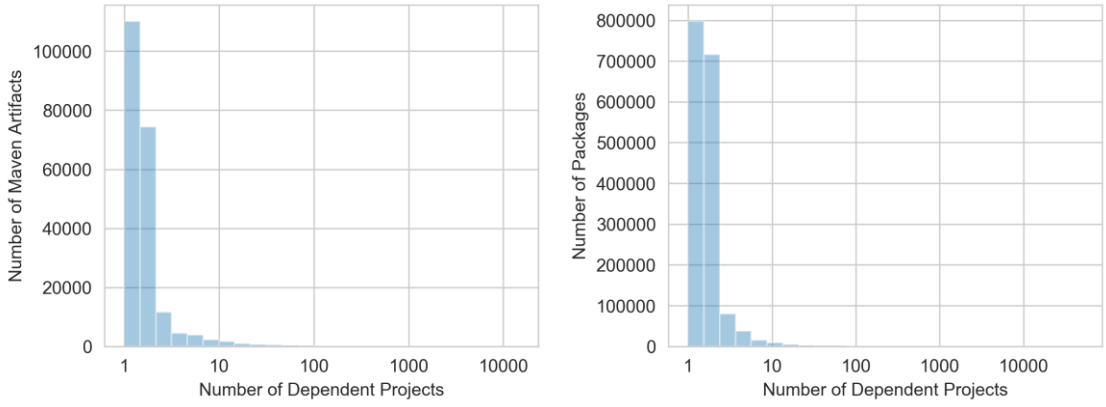

图 8 Java Package 和 Maven Artifact 在项目最新版本中的使用情况的分布

表 7 显示 211596 个 Maven Artifact 和 1669058 个 Java Package 在项目的最新版本中的使用情况（$DependentProjects(l)$）。图 8 则显示了使用情况的分布（使用对数坐标）。表 8 和表 9 分别展示了最流行的 Maven Artifact 和 Java Package。我们可以发现，不管使用哪种粒度来定义一个库，库在项目中的使用情况是极度不均匀的。对于 Maven Artifact 而言，只有 12.8%（27008/211596）的 Maven Artifact 会在多于一个 Java 项目的最新版本中得到使用；2.07%（4386/211596）的 Maven Artifact 会在多于 10 个 Java 项目的最新版本中得到使用；0.24%（523/211596）的 Maven Artifact 会在多于 100 个 Java 项目的最新版本中得到使用。使用 Java Package 统计得到的上述三个比例分别为：9.31%（155484/1669058）、1.08%（18067/1669058）、以及 0.13%（2224/1669058）。出现这种结果的一个原因在于，相当多的 Java Package 和 Maven Artifacts 都是项目为自己开发的，并不是一个具

表 8 数据集中项目最新版本使用最多的 Maven Artifact

Maven Artifact	使用项目个数	功能
junit:junit	14522	单元测试
org.slf4j:slf4j-api	6015	日志
com.google.guava:guava	4383	JSON 解析
commons-io:commons-io	4270	各种常用功能
log4j:log4j	3335	日志

表 9 数据集中项目最新版本使用最多的 Java Package

Java Package	使用项目个数	功能
java.util	50641	各种常用功能
java.io	43769	输入输出
java.net	26477	网络
org.junit	24012	单元测试
java.util.concurrent	20383	并行计算

有广泛复用价值的库。如果一个 Java Package/Maven Artifact 在多个项目中得到使用，才有较大可能性是一个开源第三方库。因此，在回答 RQ3 和 RQ4 时，应选择在多个项目中都得到使用的 Java Packages 和 Maven Artifacts 作为研究对象进行研究。

结论：开源 Java 项目中，项目的开发活动情况多种多样，库的使用情况也多种多样。其中，只有一部分项目会大量使用库，且库中只有一小部分是得到广泛使用的第三方库。对于项目依赖管理开发活动的研究，我们应该关注大量使用库并具有一定开发时间的项目，以及在各种项目中得到广泛使用的库。

2.3.2 Java 项目中添加、删除和改变库版本的情况

我们根据 RQ1 的结果，选择具有 100 个以上的 Commit、开发时间大于半年、使用 10 个以上 Java Package 且使用 3 个以上 Maven Artifact 的项目子集，共得到 11946 个 Java 项目（$|\mathcal{P}'| = 11946$）。表 10 展现了这些 Java 项目中添加、删除和改变 Maven Artifact 的情况。其中，一个处于中位数的 Java 项目会在开发过程中添加 36 个库、删除 11 个库、并修改 23 个库的版本。基于 Commit 来看，一个

表 10 Java 项目中添加、删除和改变 Maven Artifact 的情况

指标	均值	方差	最小值	25%	中位数	75%	最大值
$TotalAdded$	110.6	362.5	3	16	36	89	11848
$TotalRemoved$	55	240	0	3	11	37	8247
$TotalVerChgd$	208.5	2070	0	6	23	79	129247
$AddIntensity_c$	0.139	0.237	0.000	0.040	0.081	0.016	12.75
$RemoveIntensity_c$	0.053	0.107	0.000	0.008	0.024	0.060	5.305
$VerChgIntensity_c$	0.156	0.546	0.000	0.014	0.050	0.139	24.78
$AddIntensity_t$	0.076	0.212	0.000	0.010	0.025	0.068	14.82
$RemoveIntensity_t$	0.030	0.097	0.000	0.000	0.007	0.025	3.541
$VerChgIntensity_t$	0.102	0.887	0.000	0.004	0.015	0.049	49.32
$AvgVerChg$	1.430	4.088	0.000	0.202	0.609	1.440	251.4
$ProbRemove$	0.373	0.387	0.000	0.164	0.342	0.534	24.25

处于中位数位置的 Java 项目平均每 12.5 个 Commit 会添加一个库；平均每 25 个 Commit 会删除一个库；平均每 20 个 Commit 会修改一个库的版本。基于时间来看，一个处于中位数位置的 Java 项目平均每 40 天添加一个库；每 142 天删除一个库；每 66 天改变一个库的版本。平均而言，一个被添加进的库会被修改 1.43 次版本，且有 37.3%的概率会被删除。然而，对于不同的项目而言，上述数据的变化是很大的。使用 Java Package 得到的结果是类似的，在此不再赘述。

结论：Java 项目中添加、删除和改变库版本的情况都非常常见，且不同的项目之间有很大差异。其中，添加库的次数往往是最多的；改变版本的次数少于添加库；删除库的次数最低。不管从时间频率还是从 Commit 频率来衡量，Java 项目的开发者都需要在依赖管理上持续投入精力。

2.3.3 项目 Commit 数量和开发时间与依赖管理开发活动的相关性

表 11 给出了项目 Commit 数量与开发时间对依赖管理开发活动的影响。由于指标的分布是高度偏态的（highly skewed），因此计算 Spearman 相关系数来体现变量之间的相关性。从表中可以看到，项目的开发时间和 Commit 数量与依赖管理开发活动的强度都呈现正相关性。其中，添加、删除和版本改变的数量与

表 11 项目 Commit 数量与开发时间对依赖管理开发活动的影响

	与 Commit 数量的相关性		与开发时间的相关性	
Maven Artifact 指标	相关系数	p 值	相关系数	p 值
$TotalAdded$	0.538	<0.0001	0.142	<0.0001
$TotalRemoved$	0.542	<0.0001	0.235	<0.0001
$TotalVerChgd$	0.470	<0.0001	0.304	<0.0001
$AvgVerChg$	0.153	<0.0001	0.270	<0.0001
$ProbRemove$	0.307	<0.0001	0.241	<0.0001
Java Package 指标	相关系数	p 值	相关系数	p 值
$TotalAdded$	0.724	<0.0001	0.172	<0.0001
$TotalRemoved$	0.684	<0.0001	0.230	<0.0001
$ProbRemove$	0.287	<0.0001	0.172	<0.0001

Commit 的数量呈现中等到较强的相关性，且 Java Package 指标的相关性更强。虽然添加、删除和版本改变的数量与开发时间的相关性较弱，但是三者之间的相关性大小存在明显区别。随着项目开发时间的增长，版本改变和删除库的数量的增长速度比添加库的数量更快，这意味着版本改变和删除库的开发活动更倾向于在开发时间较长的项目中出现。最后，随着项目的开发时间和 Commit 数量的增长，平均每个库被升级的次数越多，一个库被删除的可能性就越大，这意味者删除库和升级库的行为主要出现于开发时间较长的项目。

结论：项目的开发时间越长、Commit 数量越多，项目的依赖管理开发活动就越多，平均每个库被升级的次数就越多，且一个库被删除的可能性就越大。删除库和升级库的行为主要出现于开发时间较长的项目。

2.3.4 不同 Java 库的被添加、被删除和被改变版本的情况

我们首先根据 RQ1 的结果，选择在 10 个以上项目中得到使用的库子集$\mathcal{L}'_m$和$\mathcal{L}'_p$，因为这一部分更有可能是开源第三方库。在经过前述条件筛选后，共得到了 4384 个 Maven Artifact（$|\mathcal{L}'_m| = 4384$）和 18067 个 Java Packages（$|\mathcal{L}'_p| = 18067$）。表 12 分别展现了这些 Java Packages 和 Maven Artifact 的各个项目使用相关指标

表 12 Java 库在项目中的变化情况

Maven Artifact 指标	均值	方差	最小值	25%	中位数	75%	最大值
$TotalAdded(l)$	252.7	903.5	10	46	82	177	36525
$TotalRemoved(l)$	117.7	359.8	0	18	38	83	12901
$TotalVerChgd(l)$	235.9	606.3	0	23	66	203	129247
$UpdateIntensity(l)$	1.704	5.283	0.000	0.291	0.676	1.470	229.3
$Retention(l)$	0.554	0.172	0.000	0.440	0.561	0.667	1.000
Java Package 指标	均值	方差	最小值	25%	中位数	75%	最大值
$TotalAdded(l)$	474.3	2342	10	51	103	253	99433
$TotalRemoved(l)$	137.8	681.6	0	11	28	74	20088
$Retention(l)$	0.687	0.206	0.000	0.570	0.723	0.840	1.000

的情况。从表中可以发现，库与库之间的被添加、删除和改变版本的情况也有很大差别。有的库可能经常需要升级，有的库可能不怎么需要升级；有的库可能留存率很低，有的库可能留存率很高。表 13 展现了库的流行度对库在项目中的留存率和升级情况的影响。我们可以发现，库的升级频率与库的流行度有很弱的负相关性，也就是库的升级频率越频繁，那么这个库越不可能流行。此外，库的留存率也与库的流行度有很弱的负相关性，然而结果并不是显著的，因此可以认为，库的留存率与库的流行度的相关性很弱。这意味着，项目在使用这个库一段时间之后，将其删除的概率与库的流行度无关。因此，使用流行度来决定项目应该使用什么库，可能并不总是合适的。

结论：不同的 Java 库在 Java 项目中被添加、删除和改变版本的情况、库在项目中的留存率和被升级的频率都有很大不同。库的升级频率与库的流行度有很弱的负相关性，而与常识不同的是，库的留存率与库的流行度的相关性很弱，这意味着使用流行度来决定项目应该使用什么库，可能并不总是合适的。

表 13 库的流行度与库项目中的留存率和升级情况的相关性

	$DependentProjects(l)$		$TotalAdded(l)$	
Maven Artifact 指标	相关系数	p 值	相关系数	p 值
$UpdateIntensity(l)$	-0.102	<0.0001	-0.070	<0.0001
$Retention(l)$	-0.007	0.6682	-0.216	<0.0001
Java Package 指标	相关系数	p 值	相关系数	p 值
$Retention(l)$	-0.092	<0.0001	-0.002	0.7525

3. 项目的库迁移情况分析

3.1 问题背景

正如上一节所示，在项目的开发过程中，开发者在依赖管理上投入的开发活动是很频繁的，并且依赖管理相关的各种开发活动分布于项目的各个阶段。因此，我们希望能够知道，开发者对项目所使用的库进行添加、删除和修改的背后原因。尽管由于现实场景的复杂性，项目添加和删除库的原因可能很复杂，不过已有研究显示，其中一个重要的原因很可能是库迁移[36][39][8]（Library Migration）。库迁移是指在项目的开发过程中，因为种种原因，不得不将一个正在使用的库替换成另一个功能相同或相似的库的开发活动。业界认为库迁移的决策是困难的，开发成本是很高的，且收益并不明朗，需要某种自动化的工具加以支持。利用库依赖变化数据集，可以对项目的库迁移的情况进行一个初步的刻画，并为探索库迁移原因的研究和对辅助库迁移的自动化工具提供设计思路。因此，本节回答的问题如下：

- RQ：Java 项目中库迁移的情况是什么样的？

3.2 研究方法

首先，我们需要知道哪些 Java 库之间存在库迁移关系。我们直接选择 Teyton 等人[8]提供的迁移规则（Migration Rules）作为我们研究的基础。从开源数据中挖掘库迁移规则并不是一个简单的问题，我们将设计更好的迁移规则作为未来

工作。由于已有研究中使用 Maven Artifact 作为研究对象，因此我们选择上一节中的$\mathcal{L}'_m$，作为我们研究的库集合。对于$l_1 \in \mathcal{L}'_m, l_2 \in \mathcal{L}'_m$，如果$l_1$和$l_2$功能相近且可以在一个项目中相互替换，那么我们定义$\langle l_1, l_2\rangle$是一条迁移规则。此外，迁移规则具有传递性，如果$\langle l_1, l_2\rangle, \langle l_2, l_3\rangle$都是迁移规则，那么$\langle l_1, l_3\rangle$也是一条迁移规则。我们定义全部迁移规则构成的集合为$\mathcal{M}$。

然后，我们需要知道项目中哪些 Commit 内存在迁移现象。已有研究显示自动化地挖掘与迁移相关的 Commit 也不是一个容易的问题[36][39]，然而，如果一个项目在一个 Commit 中同时添加了库l_1和删除了库l_2且$\langle l_1, l_2\rangle \in \mathcal{M}$，那么这个 Commit 有非常大可能性是一个涉及了库迁移的 Commit。因此，我们在数据集中挖掘如下的 Commit 集合作为迁移相关的实例

$$\mathcal{C}_m = \{c | l_1 \in Removed(c) \wedge l_2 \in Added(c) \wedge \langle l_1, l_2\rangle \in \mathcal{M}\}$$

尽管这样做可能会损失跨越多个 Commit 的迁移实例，我们相信$\mathcal{C}_m$依然是一个具有研究意义的迁移实例集合。我们将设计自动化的库迁移实例挖掘方法作为未来工作。

为了回答 RQ，我们参考已有工作，使用迁移图（Migration Graph）来描述库与库之间的迁移关系。迁移图是一个有向图$\langle L, E\rangle$，节点是库，边描述库与库之间的迁移关系，权重表示这个迁移关系在数据集中出现的次数，也就是

$$\begin{aligned} L &\subseteq \mathcal{L}'_m \\ E &= \{\langle l_1, l_2\rangle | \langle l_1, l_2\rangle \in \mathcal{M}\} \\ Weight(\langle l_1, l_2\rangle) &= |\{c | l_1 \in Removed(c) \wedge l_2 \in Added(c)\}| \end{aligned}$$

3.3 研究结果

从 Teyton 等人提供的数据集中，我们共得到 194 个库和其对应的迁移规则，这 194 个库均存在于$\mathcal{L}'_m$中。我们将原始迁移规则按照传递性进行扩展后再用于进行挖掘。最后，我们从库依赖变化数据集中挖掘得到了 1165 对真实存在的迁移关系（$|\mathcal{M}| = 1194$），这 1165 对迁移关系存在于 25736 个 Commit 中（$|\mathcal{C}_m| = 25736$），这些 Commit 分布于 6145 个项目中。也就是说，数据集中，26.3%（6145/23324）的软件项目存在库迁移。这些迁移关系之间构成一个并不连通的带权有向图，共有 29 个连通分量，每个连通分量表示一组可以相互迁移

的库。为了节省篇幅，本节举例讨论其中的三个连通分量：HTTP 库之间的迁移，Logging 库之间的迁移和 JSON 库之间的迁移。

图 9 显示了 HTTP 库之间的迁移。我们可以发现，在 org.apache.httpcomponents:httpcore 与 org.apache.httpcomponents:httpclient 之间的迁移是很多的，而其他 HTTP 库之间的迁移很少。这一方面是因为 org.apache.httpcomponents 是最流行的 HTTP 库，且前者是底层实现，后者是基于前者的高层 API，项目可能会出于种种原因在两者之间切换。图 10 显示了 Logging 库之间的迁移。我们可以看到，往 org.slf4j:slf4j-api 的迁移是最多的，这是因为 org.slf4j:slf4j-api 是一个对其他所有 Logging 库的上层封装，以解决不同 Logging 库兼容性问题的库。图 11 显示了 JSON 库之间的迁移，JSON 库之间存在复杂的迁移关系。

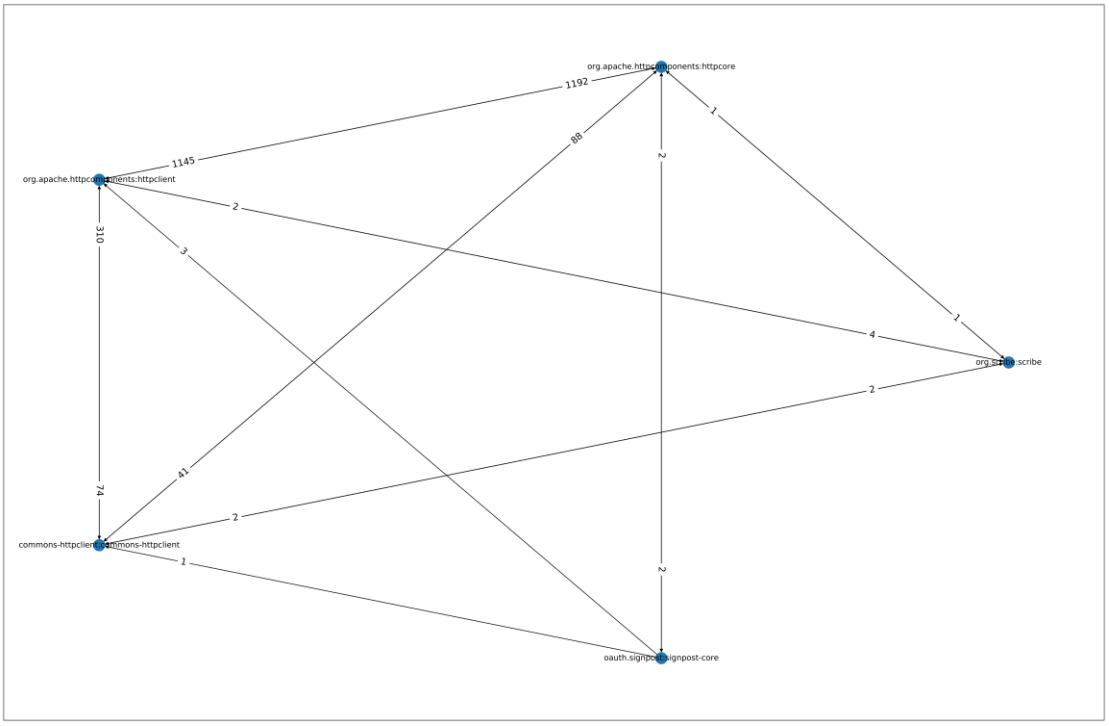

图 9 HTTP 库之间的迁移，其中箭头边的数字代表此种迁移出现的次数

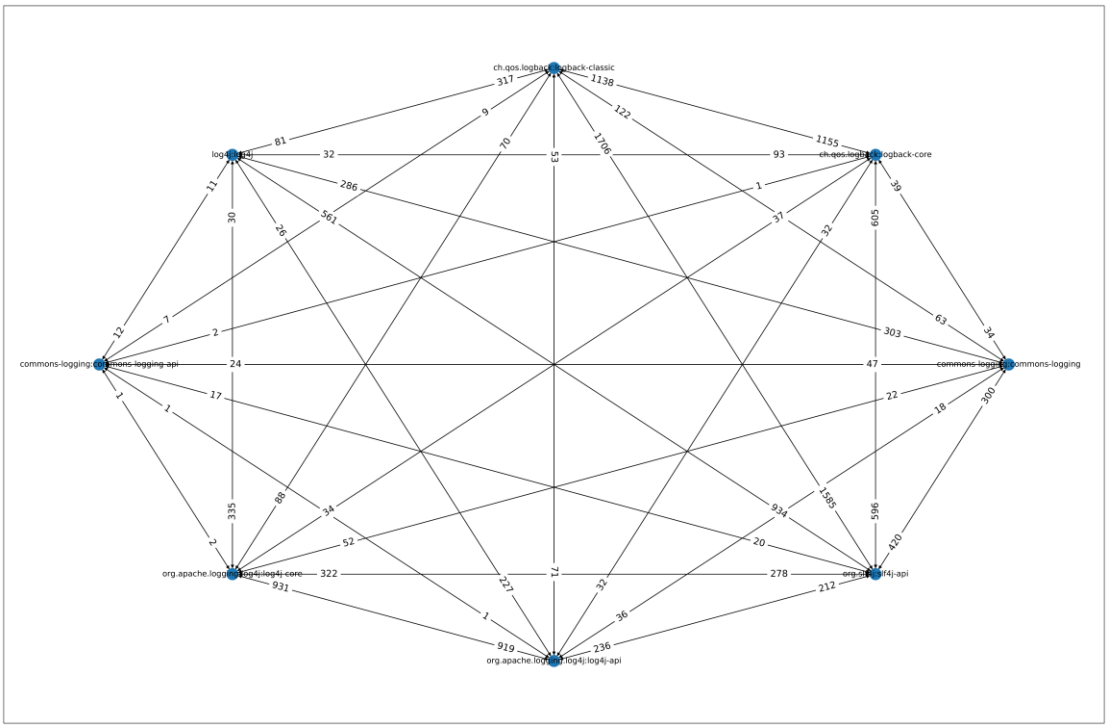

图 10 Logging 库之间的迁移

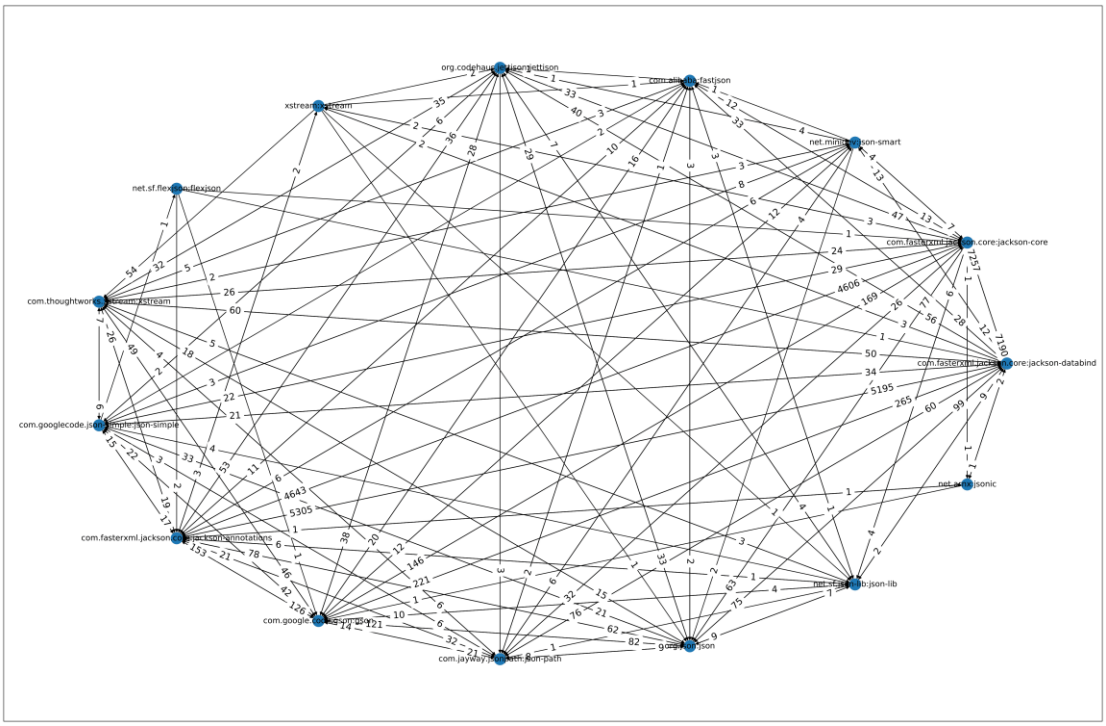

图 11 JSON 库之间的迁移

第五章 讨论与总结

1. 研究的意义（Implications）

对于研究者而言，我们的研究可以作为新的开源库相关研究的基础。我们的研究结果揭示了，项目中对库的版本活动和删除活动是非常频繁的。尽管已有工作对库升级的现状和原因进行了细致的刻画，但是目前尚没有对于库降级、删除和其他版本改变行为的现状进行刻画的研究，也没有系统性探索库删除情况的研究。因此，我们的数据集可以用于对此类依赖管理开发活动，进行更加深入的分析。此外，我们的研究揭示了，相比较于已有工作的结果，当前的 Java 项目中库迁移的现象是更加频繁的，因此，有必要对当前环境下的库迁移现象，进行更深入的研究。最后，对实践者而言，我们的数据集可以作为各种软件供应链管理工具的数据输入，用于对企业提供决策支持。

2. 效度风险（Threat to Validity）

2.1 内部效度

内部效度（internal validity）是指在研究实验测量中，在完全相同的研究过程中复制研究结果的程度。对于本文而言，在我们的具体实现中，World of Code 可能存在缺失数据；我们的数据集构建算法中可能存在 Bug；此外，Git 仓库原始数据也会存在数据不准确的问题；等等。以上这些问题可能会使得最终的数据集存在数据质量问题，并且使得最终的分析结果存在偏差。为了降低以上威胁，我们在实验过程中尽可能地处理了各种 World of Code 和 Git 中的数据质量问题，例如缺失数据等。尽管数据质量问题依然没有得到完全解决，我们认为这不会对本文的研究结果造成显著的偏差。

2.2 外部效度

外部效度（external validity）是指实验结果类推到其他外部环境的有效性。

对于本文而言，本文只选择 Java 语言和 Maven 包管理平台作为研究对象进行数据集构建和研究，研究结果可能无法适用于其他编程语言的软件项目和库生态环境，因为不同的编程语言的库生态环境存在很大的不同[14]。然而，我们相信由于 Java 语言和 Maven 平台的流行性和代表性，我们的研究结果依然是具有价值的。此外，本文中的数据集构建算法并没有对编程语言和平台做出任何假设，因此可以直接应用于其他编程语言的项目和其他包管理平台。对于 Java 语言而言，我们只选择了 60030 个开源项目作为我们的研究对象。因此，我们的研究结果可能不能精确反应开源世界的情况，也可能无法反映企业内部项目的情况。为了降低这个威胁，我们参考已有研究的方法，按照一定标准在 GitHub 上选择具有一定质量的开源项目。我们相信，这部分项目是具有一定代表性的样本集合。

3. 未来工作

首先，我们计划在数据集构建实现中，支持更多的编程语言和更多的包管理平台，例如支持 JavaScript、Python、PyPI、NPM、Gradle 等。然后，我们计划构建库与代码 API 之间的一一对应关系，从而对项目的依赖管理开发活动进行更深更细粒度的研究。最后，我们计划对库迁移现象进行更加深入的研究，包括库迁移的发生的原因、刻画库迁移的成本和收益、等等。

4. 小结

本文面向开源库依赖管理的研究与实际应用，提出了一种从大规模软件仓库中挖掘和构建库依赖变化数据的增量式构建算法，并基于 World of Code 数据库在 60030 个 Java 项目上完成了算法实现和数据集原型构建。基于得到的 Java 项目库依赖变化数据，我们探索了进行版本升级推荐的可行性，对 Java 项目的库添加、删除和升级情况进行了量化分析，并对 Java 项目中的库迁移情况进行了初步刻画。我们的研究结果可以为更深入的软件供应链相关研究打下基础，也可以支撑企业的供应链管理应用。

参考文献

[1] Lim W. Effects of reuse on quality, productivity, and economics. IEEE Software, 1994, 11(5): 23-30

[2] Mohagheghi P, Conradi R. Quality, productivity and economic benefits of software reuse: A review of industrial studies. Empirical Software Engineering, 2007, 12(5): 471-516

[3] Thung F, Lo D, Lawal J, et al. Automated library recommendation. 20th Working Conference on Reverse Engineering (WCRE 2013). Koblenz: IEEE Computer Society, 2013:182-191

[4] Sonatype Inc. 2019 State of the software supply chain [R/OL]. London: Sonatype Inc, 2019 (2019-06-25) [2020-04-28]. https://www.sonatype.com/en-us/2019ssc

[5] Coelho J，Valente M. Why modern open source projects fail. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). Paderborn: ACM, 2017:186-196

[6] Valiev M, Vasilescu B, Herbsleb J. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. Proceedings of the 2018 12th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2018). Lake Buena Vista: ACM, 2018: 644-655

[7] Kabinna S, Bezemer CP, Shang W, et al. Logging library migrations: A case study for the Apache software foundation pojects. Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR 2016). Austin: IEEE, 2016: 154-164

[8] Teyton C, Falleri JR, Palyart M, et al. A study of library migrations in Java. Journal of Software: Evolution and Process, 2014, 26(11): 1030-1052

[9] Wittern E, Suter P, Rajagopalan S. A look at the dynamics of the JavaScript package ecosystem. Proceedings of the 13th International Conference on Mining Software Repositories (MSR 2016). Austin: IEEE, 2016: 351-361

[10] Kula RG, De Roover C, German DM, Ishio T, Inoue K. A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem. 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER 2018). Campobasso: IEEE, 2018: 288-299

[11] Dey T, Mockus A. Are software dependency supply chain metrics useful in predicting change of popularity of NPM packages. Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2018). Oulu: ACM 2018: 66-69

[12] Zerouali A, Mens T, Robles G, et al. On the diversity of software package popularity metrics: An empirical study of NPM. 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER 2019). Hangzhou: IEEE, 2019: 589-593

[13] Saini M, Verma R, Singh A, et al. Investigating diversity and impact of the popularity metrics for ranking software packages [J/OL]. Journal of Software: Evolution and Process: Wiley, 2020 (2020-03-31) [2020-04-29], https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2265

[14] Decan A, Mens T, Claes M. On the topology of package dependency networks: A comparison of three programming language ecosystems. Proceedings of the 10th European Conference on Software Architecture Workshops. Copenhagen: ACM 2016: 1-4

[15] Kikas R, Gousios G, Dumas M, et al. Structure and evolution of package dependency networks. Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017). Buenos Aires: IEEE, 2017: 102-112

[16] Decan A, Mens T, Claes M. An empirical comparison of dependency issues in OSS packaging ecosystems. 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER 2017). Klagenfurt: IEEE, 2017: 2-12

[17] Decan A, Mens T, Grosjean P. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering, 2019, 24(1): 381-416

[18] Decan A, Mens T, Constantinou E. On the evolution of technical lag in the NPM package dependency network. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME 2018). Madrid: IEEE, 2018: 404-414

[19] Soto-Valero C, Benelallam A, Harrand N, et al. The emergence of software diversity in maven central. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR 2019). Montreal: IEEE, 2019: 333-343

[20] Zerouali A, Mens T, Gonzalez-Barahona J, et al. A formal framework for measuring technical lag in component repositories and its application to NPM. Journal of Software: Evolution and Process. 2019, 31(8): e2157.

[21] Decan A, Mens T, Constantinou E. On the impact of security vulnerabilities in the NPM package dependency network. Proceedings of the 15th International Conference on Mining Software Repositories (MSR 2018). Gothenburg: IEEE, 2019: 181-191

[22] Zimmermann M, Staicu CA, Tenny C, et al. Small world with high risks: A study of security threats in the NPM ecosystem. 28th USENIX Security Symposium (USENIX Security 2019). Santa Clara: USENIX, 2019: 995-1010

[23] Zerouali A, Cosentino V, Mens T, et al. On the impact of outdated and vulnerable JavaScript packages in docker images. 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER 2019). Hangzhou: IEEE, 2019: 619-623

[24] Bogart C, Kästner C, Herbsleb J, et al. How to break an API: Cost negotiation and community values in three software ecosystems. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016) Seattle: ACM, 2016: 109-120

[25] Abdalkareem R, Nourry O, Wehaibi S, et al. Why do developers use trivial packages? An empirical case study on NPM. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). Paderborn: ACM, 2017: 385-395

[26] Wang Y, Wen M, Liu Z, et al. Do the dependency conflicts in my project matter. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). Lake Buena Vista: ACM, 2018: 319-330

[27] Dietrich J, Pearce D, Stringer J, et al. Dependency versioning in the wild. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR 2019). Montreal: IEEE, 2019: 349-359

[28] Decan A, Mens T. What do package dependencies tell us about semantic versioning? [J/OL]. IEEE Transactions on Software Engineering. IEEE, 2019 (2019-05-23) [2020-04-29]. https://ieeexplore.ieee.org/abstract/document/8721084

[29] De la Mora FL, Nadi S. An empirical study of metric-based comparisons of software libraries. Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2018). Oulu: ACM, 2018: 22-31

[30] Trockman A, Zhou S, Kästner C, et al. Adding sparkle to social coding: An empirical study of repository badges in the NPM ecosystem. Proceedings of the 40th International Conference on Software Engineering (ICSE 2018). Gothenburg: ACM, 2018: 511-522

[31] Ma Y, Mockus A, Zaretzki R, et al. A methodology for analyzing uptake of software technologies among developers [J/OL]. arXiv:1908.11431: (2019-08-29) [2020-04-29]. https://arxiv.org/abs/1908.11431

[32] Kula RG, German DM, Ishio T, et al. Trusting a library: A study of the latency to adopt the latest maven release. 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015). Montreal: IEEE, 2015: 520-524

[33] Mirhosseini S, Parnin C. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). Urbana: IEEE/ACM, 2017: 84-94

[34] Kula RG, German DM, Ouni A, et al. Do developers update their library dependencies? Empirical Software Engineering, 2018, 23(1):384-417

[35] Zapata RE, Kula RG, Chinthanet B, et al. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for NPM JavaScript packages. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME 2018). Madrid: IEEE, 2018: 559-563

[36] Teyton C, Falleri JR, Blanc X. Mining library migration graphs. 2012 19th Working Conference on Reverse Engineering (WCRE 2012). Kingston: IEEE, 2019:289-298

[37] Hora A, Valente MT. APIWave: Keeping track of API popularity and migration. 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME 2015). Bremen: IEEE, 2015: 321-323

[38] Chen C, Xing Z. SimilarTech: Automatically recommend analogical libraries across different programming languages. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016). Singapore: IEEE/ACM, 2016: 834-839

[39] Alrubaye H, Mkaouer MW, Ouni A. MigrationMiner: An automated detection tool of third-party Java library migration at the method level. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME 2019). Cleveland: IEEE, 2019: 414-417

[40] Leite L, Rocha C, Kon F, et al. A survey of DevOps concepts and challenges. ACM Computing Surveys (CSUR), 2019, 52(6):1-35

[41] Kalliamvakou E, Gousios G, Blincoe K, et al. An in-depth study of the promises and perils of mining GitHub. Empirical Software Engineering. 2016, 21(5): 2035-2071.

[42] Mockus A, Spinellis D, Kotti Z, et al. A complete set of related Git repositories identified via community detection approaches based on shared commits [R/OL]. arXiv:2002.02707: (2020-02-27) [2020-04-29]. https://arxiv.org/abs/2002.02707

[43] Lopes CV, Maj P, Martins P, et al. DéjàVu: A map of code duplicates on GitHub. Proceedings of the ACM on Programming Languages. 2017, 1(OOPSLA):1-28.

[44] Amreen S, Mockus A, Zaretzki R, et al. ALFAA: Active Learning Fingerprint based Anti-Aliasing for correcting developer identity errors in version control systems. Empirical Software Engineering. 2020, 1(3):1-32.

[45] Gousios G, Spinellis D. GHTorrent: GitHub's data from a firehose. 2012 9th IEEE Working Conference on Mining Software Repositories (MSR 2012). Hyderabad: IEEE, 2012: 12-21

[46] Ma Y, Bogart C, Amreen S, et al. World of code: An infrastructure for mining the universe of open source VCS data. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR 2019). Montreal: IEEE, 2019: 143-154

[47] Pietri A, Spinellis D, Zacchiroli S. The software heritage graph dataset: Public software development under one roof. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR 2019). Montreal: IEEE, 2019: 138-142

[48] Dai W, Rubin SH. A supply chain model for software components management. Proceedings Fifth IEEE Workshop on Mobile Computing Systems and Applications. Las Vegas: IEEE, 2003: 69-76

[49] Eastlake D, Jones P. US secure hash algorithm 1 (SHA1) [P/OL]. US Patent: Cisco Systems, 2001 (2001-09-01) [2020-04-28]. https://www.hjp.at/(st_a)/doc/rfc/rfc3174.html

[50] Li Z, Lu S, Myagmar S, et al. CP-Miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on Software Engineering. 2006, 32(3):176-92

[51] Bird C, Rigby PC, Barr ET, et al. The promises and perils of mining git. 2009 6th IEEE International Working Conference on Mining Software Repositories (MSR 2009). Vancouver: IEEE, 2019: 1-10

[52] Hata H, Treude C, Kula RG, et al. 9.6 million links in source code comments: Purpose, evolution, and decay. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2019). Montreal: IEEE, 2019: 1211-1221

[53] Wang Y, Chen B, Huang K, et al. An empirical study of usages, updates and risks of third-party libraries in Java projects [J/OL]. arXiv:2002.11028. (2020-02-25) [2020-05-14] https://arxiv.org/abs/2002.11028

[54] Braun V, Clarke V. Using thematic analysis in psychology. Qualitative Research in Psychology. 2006, 3(2):77-101

[55] Kapur P, Cossette B, Walker RJ. Refactoring references for library migration. Proceedings of the ACM Inernational Cnference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2010). Tahoe: ACM, 2010:726-738

[56] Cossette BE, Walker RJ. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012) Cary: ACM, 2012: 1-11

[57] Raemaekers S, van Deursen A, Visser J. Semantic versioning and impact of breaking changes in the Maven repository. Journal of Systems and Software. 2017, 129:140-58.

[58] Teyton C, Falleri JR, Blanc X. Automatic discovery of function mappings between similar libraries. 2013 20th Working Conference on Reverse Engineering (WCRE 2013). IEEE 2013: 192-201)

[59] Chen C, Xing Z, Liu Y. Mining likely analogical APIs across third-party libraries via large-scale unsupervised API semantics embedding. IEEE Transactions on Software Engineering. 2019 Jan 30.

[60] Alrubaye H, Mkaouer MW, Ouni A. On the use of information retrieval to automate the detection of third-party Java library migration at the method level. 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC 2019). IEEE, 2019: 347-357

[61] Xu S, Dong Z, Meng N. Meditor: Inference and application of API migration edits. 2019

IEEE/ACM 27th International Conference on Program Comprehension (ICPC). IEEE 2019: 335-346

本科期间的主要工作和成果

本科期间发表的短文

1. Hao He. Understanding source code comments at large-scale. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019) Tallinn, 2019 Aug 12, 1217-1219.
2. Yue Wu, Kenuo Xu, Hao He, Zihang Wu, and Chenren Xu. Poster: Retroreflective MIMO Communication, In Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications (HotMobile 2019). Santa Cruz, 2019 Feb 22, 161-161.

致谢

首先，我要衷心感谢我的指导老师周明辉教授，她对我的研究做出了悉心指导，并为我提供了宝贵的业界交流机会。其次，我要感谢田纳西大学的 Audris Mockus 教授，他对 World of Code 的使用给出了非常耐心和详尽的指导。第三，我要感谢我的研究生学长徐玉麟同学，他对我的工作提出了很多宝贵建议。然后，我要感谢华为公司的人员，他们分享的企业内部需求和提出的建议给我带来了很大帮助。最后，我要感谢我的父母，在今年新冠肺炎疫情的特殊环境下，对我居家完成论文给出了各种各样的支持。没有以上各位的支持，我的论文不可能完成，在此再次感谢！