Computer Science and Application 计算机科学与应用, 2022, 12(12), 2958-2970
Published Online December 2022 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2022.1212300

Hans汉斯

一种筛选和识别供应链软件的方法

孙　晴[1]，田春岐[1]，王　伟[2]

[1]同济大学计算机科学与技术系，上海
[2]华东师范大学数据科学与工程学院，上海

收稿日期：2022年11月26日；录用日期：2022年12月23日；发布日期：2022年12月30日

摘　要

筛选和识别开源供应链软件是软件供应链安全的前置条件，也是帮助用户和企业甄选可靠软件的必要手段。同时识别整个生态的供应链，是探究生态特点、查找生态隐患的重要方法。本文通过追溯不同编程语言管理外部依赖方法的发展史，概括出当今四种常见的外部依赖管理方式，并提出了一种通用的开源软件供应链构建算法，通过实验证明了该方法的有效性。

关键词

开源软件供应链，包管理器，代码依赖关系

A Method to Filter and Identify the Supply Chain Software

Qing Sun[1], Chunqi Tian[1], Wei Wang[2]

[1]Department of Computer Science and Engineering, Tongji University, Shanghai
[2]School of Data Science & Engineering, East China Normal University, Shanghai

Received: Nov. 26th, 2022; accepted: Dec. 23rd, 2022; published: Dec. 30th, 2022

Abstract

Filtering and identifying open source supply chain software are the front conditions for the security of the software supply chain, and it is a necessary means to help users and enterprises select reliable software. At the same time, identifying the supply chain of the entire ecology is a vital way to explore the ecological characteristics and find hidden dangers. By tracing the development history of external dependencies in different programming language management, this article summarizes the four common external dependencies management methods today and proposes a universal open source software supply chain construction algorithm. The effectiveness of this method is proved by experiments.

文章引用: 孙晴, 田春岐, 王伟. 一种筛选和识别供应链软件的方法[J]. 计算机科学与应用, 2022, 12(12): 2958-2970.
DOI: 10.12677/csa.2022.1212300

Keywords

Open Source Software Supply Chain, Package Manager, Code Dependency Relationship

1. 引言

长期以来，编程语言的发展经历了从低级到高级的发展历程，其发展的主旋律在于“让人更容易编程”。随着科学技术的不断发展和需求的增多，软件的规模也在不断增大，传统的软件构建模式虽然可以满足不同功能需求，但是开发效率却不尽如人意，也会导致开发人员对功能相同的软件进行重复性开发。模块化开发是实现代码复用，避免重复工作的有效方法——通过把特定功能的代码抽取出来，封装成一个独立的模块、类或者函数，在构建新的程序时就可以对曾经编写过的代码加以利用。这种模块化，低耦合以及数据封装的软件构建方法就可以将开发者从费时费力的重复劳动中解放出来。

近年来，随着 Git 等协作开发技术的发展，开源软件得到了快速的发展和应用[1]，这也使得软件的开发模式经历了从集中到分散的转变，一方面体现在打破了地理距离的限制，参与开发的组织和个人可以位于世界各地；另一方面可被复用的软件种类和数量迅速增加，每个软件又可能拥有众多版本，物理上存储在互联网上不同的地方。其他用户可以相对容易的获取、安装和使用这些开源软件。

这种细粒度的、广泛的软件复用的是当今软件开发最明显的特征之一。然而，对系统架构而言，外部系统依赖往往是系统质量属性的最大风险，对软件自身亦是如此。将一个包作为依赖项添加自己的程序中，该程序间接拥有了依赖项中的所有风险和缺陷，因为它完全依赖于这些下载的代码。采用不良依赖的成本可以看作是每个不良结果的成本乘以其发生的可能性之和，因此软件依赖有着严重的潜在风险，而这些风险常常会被忽视[2]。开发者可能尚未理解如何有效选择和使用依赖关系，甚至不清楚其使用的软件包含哪些依赖关系。

这种依赖关系也可以视为一种“供应”关系——上游软件(被依赖的软件)和下游软件(含依赖项的软件)通过依赖关系形成了一个错综复杂的供应网络。在本文中，我们首先对常见的四种不同语言的供应链进行分析，提出了一种通用的筛选和识别供应链软件算法。本文主要的贡献包括：总结了四种不同语言生态的供应链构建和管理方法；提出了一种通用的供应链软件筛选和识别算法；最后以开源的形式提出了两个新的供应链软件数据集用于验证提出的供应链算法。

2. 开源软件供应链的定义

2.1. 软件依赖

在软件开发中，依赖项是程序员想要调用的外部代码，用来设计、测试和调试等，这个代码单元通常被称为包，或者库，或者模块等。软件间的依赖关系是普遍存在的，且存在以下特性：1) 多依赖，一个软件可能直接依赖多个外部模块；2) 传递依赖，一个软件的依赖项也可能有自己的依赖项，从而形成比较长的依赖链；3) 区分版本，软件依赖于依赖项的特定版本，使用不同的版本可能会导致错误；4)循环依赖，如果版本为 v1 的软件 A 依赖了 B，B 依赖了 C，C 最终却依赖了版本为 v2 的 A，那么这种环状的依赖关系就形成了循环依赖，在一些复杂的开源系统中，这样的关系并不少见[3]。

正是由于依赖依赖关系层次很深且结构复杂，最终加大了软件维护的难度。因此识别一个软件的供应链是评估软件安全性能的前提，识别一个语言生态的供应网络是分析这一生态的基础。

2.2. 开源软件供应链

供应链这一理念在传统的工业领域已经被广泛深入地研究了多年[4]。在商业领域，成功的供应链管理代表着，一个公司在商业物流中的规划、组织和控制等方面保持着竞争力[5]。

软件开发特别是开源软件开发蕴含了很多相同或类型的特性，这些特性使得我们可以从供应链的视角来发现和研究问题：1) 开源软件的贡献者分布在全球各地，但是通过 github 等协作平台紧密地联系在一起；2) 大部分软件产品都会重用之前一些成熟项目的代码、参考其他项目的成功设计或者涉及其他项目的核心开发成员，这意味着开发实际上是建立在这些成熟软件之上。

供应链管理理论已经在商业领域得到了广泛重视，它能有效降低产品原料分散带来的风险。我们希望能够将供应链的理论或实际迁移到软件开发尤其是开源软件这个领域中。但是并没有一个成熟的软件供应链的定义，因此我们类比传统供应链对软件供应链做了定义：软件供应链涵盖开发者和团队，这些开发者或者团队通常由企业进行支持，软件项目和包之间的关系类比于一条流动的“链条”，任何对源代码的修改都会通过这个链条将影响传递下去。

开源软件供应链涉及所有开源软件上游社区、源码包、二进制包、包管理器、存储仓库以及开发者和维护者、社区、基金会等，本文仅做软件层面的分析。

开源软件和商用软件的区别在于：开源软件通常来源于开源社区或各大代码托管平台，安全性与合规性难以得到保证。开源软件同样无法获得与商业和闭源选项相同级别的技术支持，如果开发人员遇到问题，无法直接向供应商寻求帮助，只能依赖社区论坛。

此外，由于源代码开放等特性，开源软件还容易遭受软件供应链攻击，供应链攻击的特征是向软件包中注入恶意代码，以破坏供应链下游的相关系统。近年来，许多供应链攻击在软件开发过程中充分利用了包管理器，它在整个软件生命周期中自动解析，下载和安装数百万个开源代码包，从而进一步加剧了供应链攻击带来的影响，如 SolarWinds 漏洞，Codecov 攻击等。

对开源软件的使用也会涉及到知识产权风险。开源许可证(License)规定了使用者的权力和义务，使用者必须完全按照许可证的要求进行分发、使用，如果使用者没有按照规定使用开源软件，就会导致版权侵权。常见的许可协议有 GPL，GNU 以及 BSD 等，每一种协议都代表了不同开源社区的标准。如果托管在开源社区的代码本来就是不具备开源许可证的，那么这一开源软件本身就带有版权问题，用户使用这样的代码造成的侵权责任，需要自己承担。

代码复用为 IT 行业带来了极大的便利，提高了开发效率，降低了成本。其中，开源是软件供应链的重要组成部分，新思科公司的报告显示，现在 99%的软件至少依赖了一款开源软件，而且开源软件占当今代码总量的 90%以上。然而由于开源软件的依赖和引用关系较为复杂，其安全性也往往缺少审查和管理，因此，开源软件也增加了软件供应链的复杂性和安全风险。开源软件复杂的供应链关系、不断增加的安全漏洞与恶意软件包，以及开源许可证的风险，已成为不可忽视、亟需管控的领域。

使用安全的依赖项，是开发出安全的软件的基石；了解软件的软件供应链，是评价软件安全性的前提。因此，本文从不同的语言生态出发，分析它们软件供应链的特点，给出筛选和识别供应链软件方法。同时，将一些典型的软件系统作为案例分析。

3. 不同语言生态的供应链软件识别

随着人们需要用软件来解决实际应用中越来越复杂多样的问题，计算机语言自身在迅速发展。由于软件规模的急剧膨胀以及工业界追求越来越快的生产速度，管理外部依赖项方式的也在迅速发展，不同

的语言生态的依赖项管理方式是不同的，从而导致了其供应链的追踪方法也是不同的[6]。本章旨在通过探讨多种语言的依赖项管理方式，挖掘出其内在的规律。

3.1. 手动配置外部依赖

传统的软件安装需要经过获取源代码、解压缩、配置、编译和安装等步骤，用户需要自己解决软件的安装条件，如编译环境和依赖的关系等，才能使软件在当前系统上成功编译和安装。本节以 C++语言为例，分析外部依赖的引用方式和供应链软件的识别方式。

3.1.1. C++引用外部依赖的方式

C++有以下两种使用外部依赖项的方式，分别为源代码分发和 DLL 技术，具体描述如下：

1) 以源代码形式分发复用。用户可以把一个外部实现的 C++类的静态库添加到工程路径中，用编译器重新编译包含类库的源代码，类库的代码模块就和用户自己编写的程序链接为同一可执行代码，这种方法也称为静态链接方式。使用这种方式生成的可执行文件可以独立运行，同时静态链接以目标文件为单位的。若源文件引用了静态库中的 printf()函数，链接器就会把库中包含 printf()函数对应的目标文件进行连接。若多个函数都放在一个目标文件中，可能导致很多不需要的函数都被一起被链接进了输出结果中，从而造成代码冗余以及更新困难等问题。

2) DLL 技术。静态链接的方式会导致引用的外部库函数失去模块化特征以及代码冗余，一种解决方法是将类库封装成动态链接库(DLL)。由于 DLL 只有在运行时才进行动态链接，可以使多个程序共享同一 DLL，避免了不必要的磁盘和内存浪费。另一方面，如果想要更新类库中函数或者类的实现方法，在原有接口不变的前提下，只需要重新生成和分发一个新的 DLL，用户不需要做任何修改就能更新程序[7]。DLL 方法使 C++的类和函数成为可替换，可重用的组件，不足之处在于执行速度和独立性较差。

3.1.2. 如何识别 C++软件的供应链

早期的 C++生态没有完整的包管理器支持，用户需要自己下载安装静态库和动态库，并解决各种依赖关系和冲突。在这种情况下，无法直接获取一个软件完整的供应链，一般要通过源代码分析、makefile 文件解析或是执行某些指令的方式。

Linux 提供了一系列指令来查看软件动态链接库：1) ldd 指令，ldd 指令通过调用动态链接器去查找程序的库文件依赖关系，但某些版本的 ldd 指令可能采用直接调用可执行程序的方式，从而产生安全性问题；2) pldd 指令，可以显示一个运行中的进程载入的所有共享对象；3) pmap 指令，用来报告一个运行中的进程的内存映射，也能显示出该进程的库文件依赖；4) objdump 指令，用查看目标文件或者可执行的目标文件的构成的 gcc 工具，使用-x 选项同样可以查看目标文件需要的动态库文件。

windows 下查看软件的动态链接库的方法主要有两种：1) dumpbin 工具，微软的二进制文件转储器，可以用来来检查 COFF 对象文件、COFF 对象的标准库、可执行文件和动态链接库；2) 进程查看器(ProcessExplorer)。可以用来查看进程(实时运行)依赖的 dll 文件。DependencyWalker 工具：可以用来查看 dll 或 exe 依赖的 dll 文件，并建立所有相关模块的分层树形图(依赖树)。

目前尚未有查看一个 C++可执行程序依赖的所有静态库的工具，由于使用静态库必须在代码中用 include 关键字来显式声明使用了一个头文件，因此统计该项目中所有用 include 声明的、非内部实现的头文件，即可获得该项目所有的外部依赖头文件(包括静态库和动态库)。

用 include 关键字声明引用头文件是 C/C++语言描述依赖关系的一种方法，makefile 则是另一种描述文件依赖关系的方法。makefile 描述了项目的整体编译组织关系，包括各个目标文件的实现和他们的依赖关系。通过使用 make 命令来解释 makefile 文件，实现程序的完全自动化编译。其中关于依赖关系的语法如下。

1) 直接定义

makefile 的基本语法是：

$(TARGET):$(OBJS)

$(CC) -o $(TARGET) $(OBJS)

$(TARGET)作为目标，它依赖于$(OBJS)，下面一句表示 make 需要执行的，命令。如果没有明确说明$(OBJS)依赖的文件，也没有写明命令，make 的隐式规则开始生效，对于$(OBJS)中的每个目标，make 自动执行：

(CC)(CPPFLAGS) -c $< -o $@

其中， $<代表源码文件，$@为目标文件。

2) 间接定义

.c 文件生成的.o 文件被其他文件所依赖，形成了传递依赖关系。目录递归调用：根目录中的 makefile 文件和子目录中的 makefile 文件存在递归调用关系。在根目录中执行 make 命令之后，make 会逐层向下查询文件间的依赖关系，最终生成目标文件。

根据依赖语法从 makefile 中解析依赖关系的方法可以描述为图 1 [8]：遍历目录下的所有 makefile 文件，并对每个文件进行分析；解析文件中的规则语句，用树形结构保存其层次；提取编译选项中与文件的对应关系，并保存在 map<string,list<string>>数据结构中；分析 map：如果 key 对应的 list 为空，则遍历其他 key 对应的 list；如果存在依赖文件对应的描述结构，则添加对应结构的依赖文件，否则新建并添加对应结构的依赖文件，并重新排序。

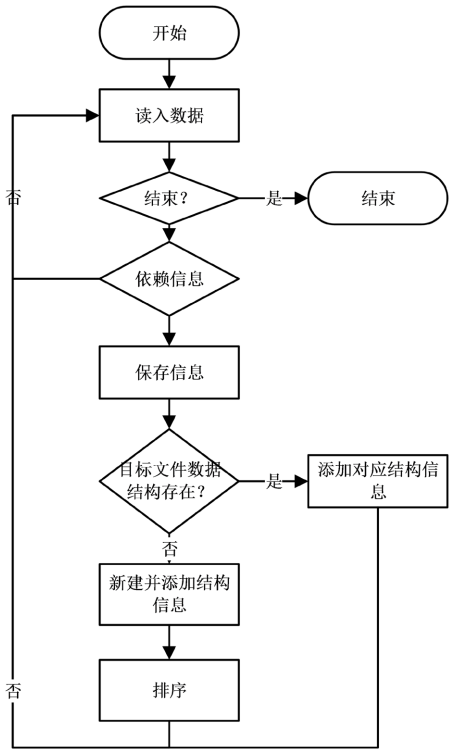

Figure 1. Extract the dependency flowchart according to the makefile
图 1. 根据 makefile 提取依赖关系流程图

值得注意的是，上述大部分方法只能用来查看软件的直接依赖项，对于特定软件的完整供应链(包括依赖项的依赖项)，则需要不断递归使用查看代码依赖项的方法，直到该层代码没有依赖项为止。若想要

分析整个 C++生态的供应链，则需要获取该生态所有软件的依赖树，再将它们根据公共节点组合成一个完整的供应链，这无疑是一项困难的工作。

对于托管在 GitHub 上的 C++项目，可以采用 git submodule 的方式来管理所需要的依赖，这样虽然比上述方法容易一点，但是仍有弊端：每次 clone 项目需要花费大量的时间在拉取依赖项目上；每次编译都要从依赖项开始编译。鉴于长久以来 C++难以进行包管理这一痛点，开发者也开始构建 C++的包管理器，比如 mamba，Conan，vcpakg 等。除了 C++之外，其他很多编程语言也都保留了可以手动配置依赖项这一方式，用在包管理器无法加载安装包的时候。

3.2. 使用包管理器管理依赖

工业界引入“软件包”这一概念作为软件集成的基本单位，其中包括了软件的依赖信息，用来解决原始的源代码安装方式不能包含依赖关系的状况。包管理器的出现将开发者从手动配置外部依赖的复杂性中解放了出来，包管理器也称为依赖管理器，是一种允许用户在操作系统上安装、删除、升级、配置和管理软件包的工具。

包管理器一般基于 C/S 模型，软件分发端用来文件数据库的形式保存软件包，客户端用来根据用户的请求来从软件分发端下载/上传软件包。一般具有以下特性。1) 统一管理软件资源。2) 支持集群服务器。3) 实时更新。由于开源软件自身频繁更新，为了保证包管理器中的软件是最新版本，需要软件的开发者主动向包管理器中的提交新的版本。4) 软件依赖管理：维护依赖描述信息元数据(metadata)。

依赖管理器进一步缩小了开源代码重用模型的规模。现在，开发人员可以在由数十行代码组成的单个函数的粒度上共享代码，这是一项重大的技术成就。

包管理器解决软件之间依赖关系是基于对用户试图安装的软件及其依赖执行图 2 所示的检查，检查过程是采用深度优先遍历算法遍历软件依赖关系所形成的“依赖树”，从而判断安装过程中软件依赖关系是否满足。

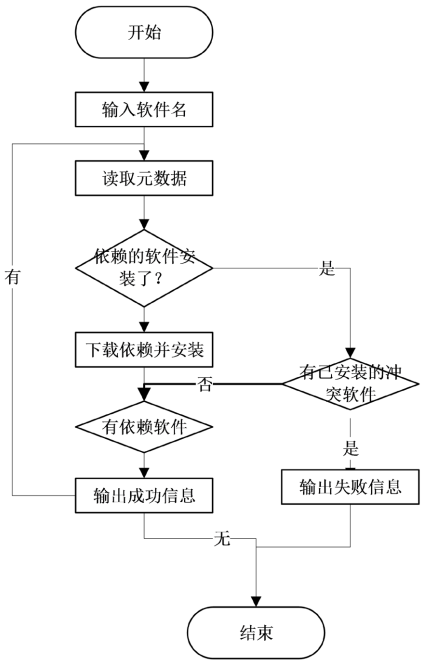

Figure 2. Software package management technology identifies software dependency flowcharts
图 2. 软件包管理技术识别软件依赖流程图

本节将以 maven 为例，阐述包管理器的工作机制以及如何通过包管理器追踪软件的依赖关系。

3.2.1. Java 包管理器

Java 语言的中央仓库 maven 作为当今最大的软件包存储器之一，它的核心是远程仓库、本地仓库和 pom.xml。Maven 基于项目对象模型(project object model，简称 POM)的理念来管理项目，在 pom.xml 文件中描述项目间的依赖管理。Maven 将 pom.xml 中定义的 jar 包从远程仓库下载到本地仓库，同一个版本的 jar 包只需要下载一次，由多个项目共享。

Maven 支持传递依赖(transitive dependencies)，如果一个项目直接依赖一个库，而这个库又依赖了几个或者其他更多库，开发者不必找出所有依赖并把他们写进 pom.xml 中，只需加入直接依赖项，Maven 就可以隐式的把那些间接依赖的库也引入到项目中。同时 Maven 还能处理依赖中潜藏的冲突，开发者可以指定默认的传播行为，也可以去掉指定的传递依赖[9] [10]。

3.2.2. 如何识别 java 软件的供应链

Maven 提供了查看单个软件依赖树的指令：mvndependency:tree，用户可以在项目根目录下执行该指令查看当前项目的依赖树。Maven 主页也提供了包含中央仓库所有软件数据的索引包，用户可以下载这些索引包并使用 Luke 或 Marple 等查询数据，根据制品的链路信息可以获取整个 java 子生态的供应链网络。

Maven 中软件包提供的元数据有限，包括 Artifact Id (产品标识)、version (版本)、packing (打包方式)、compiler lever (jre 的版本)、License (许可证)等，且制品为 jar 包的形式，很难提供其他层面的信息。

3.3. 提供上游协作仓库的包管理器

代码托管平台是开源生态的重要组成部分，用于软件、文档及其他作品的源码托管，开发者使用代码托管平台对代码进行维护修订和版本控制，可被公开或私有访问。代码托管平台在开源生态中具有价值：首先为开源软件开发提供协作环境，是快速实现软件迭代更新的必要条件；其次作为代码托管基础设施，汇聚了大量的开源项目，成为开源代码数据储备资源池；最后还能设立活跃度、受欢迎程度等指标，折射出技术热点和发展趋势。

国内外主要的代码托管平台有 GitHub、SourceForge、Bitbucket、GitLab 和 Gitee 等。代码托管和社区协作是代码托管平台提供的两大主要服务，从代码平台中，我们不仅可以对软件进行源码层面的分析，还可以对代码审查、Bug 跟踪、邮件列表、人员组成和提交记录等方面进行分析。因此对软件的上游协作仓库进行分析，也是研究软件供应链的一个重要方法。

Maven 等包管理的元数据中并不具备表征上游地址的字段，从软件制品出发，寻找其上游协作仓库也是一件非常困难的事情。其他语言生态，诸如 C#的 NuGet，Node.js 的 NPM，PHP 的 Composer，Python 的 PyPI 等都是可选填入上游地址的。从元数据中提取出上游地址，根据上游地址找到软件的代码托管平台，能够对供应链信息做进一步的补充。

对于提供了上游协作仓库的包管理器中的软件，其供应链追踪方式可以描述为图 3。

首先从包管理器中获取软件的基础信息，比如软件包的名字、大小、协议、上游地址、版本等。这一步可以利用包管理提供的指令或者 API，也可以从 libraries.io 在线搜索存储库中获取(https://libraries.io/)。libraries.io 中收集了 32 个包管理器，超过 537 万个软件包的元数据，包括 npm、Maven、PyPI 和 Go 等；同时还监控 github.com, gitlab.com 和 bitbucket.org 三个代码托管平台超过 3300 万个存储库；解析了项目的依赖项描述，并存储了项目间的依赖关系。用户可以使用 libraries.io 官方提供的 API 来获取数据，也可以直接下载其归档文件。

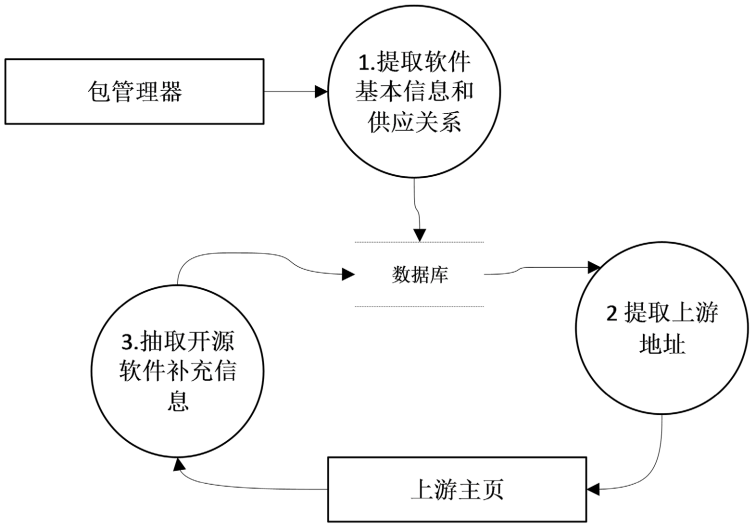

Figure 3. Ways to obtain software supply chain data
图 3. 获取软件供应链数据的方式

获取到软件包的基础数据之后，便可以从中提取上游软件的地址。它可能是软件主页或者代码托管平台地址。对于前者，需要通过爬虫和 NLP 等技术解析到该项目真正的代码托管平台地址；对于后者，从不同的代码托管平台采集数据的方式不同，应该制定不同的策略。

最后访问每个软件的代码托管平台地址，克隆源码，抽取软件的历史版本信息，包括每个版本的版本号、参与开发人员信息、版本特性、已知问题等。以 GitHub 为例，GitHub 提供了获取各种数据的 API，用户可以使用 API 来获取数据，弊端在于难以一次性获取大量项目的信息。所幸，GH Archive (https://www.gharchive.org/)是一个记录公共 GitHub 活动的项目。GitHub 提供了 20 多种事件类型，GH Archive 将这些事件汇总到每小时的存档中，用户可以使用任何 http 客户端进行访问、下载。定时获取 GH Archive 数据并解析也可以获取 GitHub 所有可以被公开访问项目的信息。

3.4. 通过代码托管平台获取外部依赖

Google 在 2009 年 11 月 10 日将 go 语言正式对外开源，Go 语言发展至今已经过去十年多了，是目前最流行的新兴语言，云计算领域的首选语言，而且目前随着区块链的流行，Go 再次成为了这个领域的第一语言，以太坊，IBM 的 fabric 等重量级的区块链项目都是基于 Go 开发。

Go 语言的包管理机制经历了 GOPATH、Vendor 以及 Modules 等阶段，发展趋势在于支持多版本控制、支持项目之间依赖包重用和自动化。虽然 Go Modules 有时也会被称为 Go 的依赖管理器，但是和 3.2、3.3 节中阐释的包管理器有着本质区别。图 4 展示了 3.2、3.3 节中包管理器的工作原理，图 5 展示了 Go Modules 的工作原理。

包管理器的分发端存放着可以提供给用户使用的软件，有专门的维护人员从供应源中导入、更新软件包，最终将封装好的软件包以制品名称、版本号等标志存放到存储集合中。Go 的第三方包是没有中央库统一管理的，所以不存在分发端的概念。对于包的获取，就是用 go get 等命令借助代码管理工具通过远程拉取或更新代码包及其依赖包，并自动完成编译和安装。这样做的好处是，直接跳过了包管理中央库的约束，让代码的拉取直接基于版本控制库，开发者的协作管理都是基于这个版本依赖库来互动；去掉冗余，直接复用最基本的代码基础设施。Go 社区一直遵循“尽量简单”的原则，这种做法很大程度上减轻了开发者对包管理的复杂概念的理解负担，但是对于现实过程中的开发者来说，仍

然有其痛苦的地方。1) 第三方包没有内容安全审计，很容易引入代码 Bug。2) 依赖的完整性无法校验，程序编译时无法保障百分百成功。3) 如果这个第三方库的开发者很活跃，代码更新更快，很难决定如何升级引用的代码。

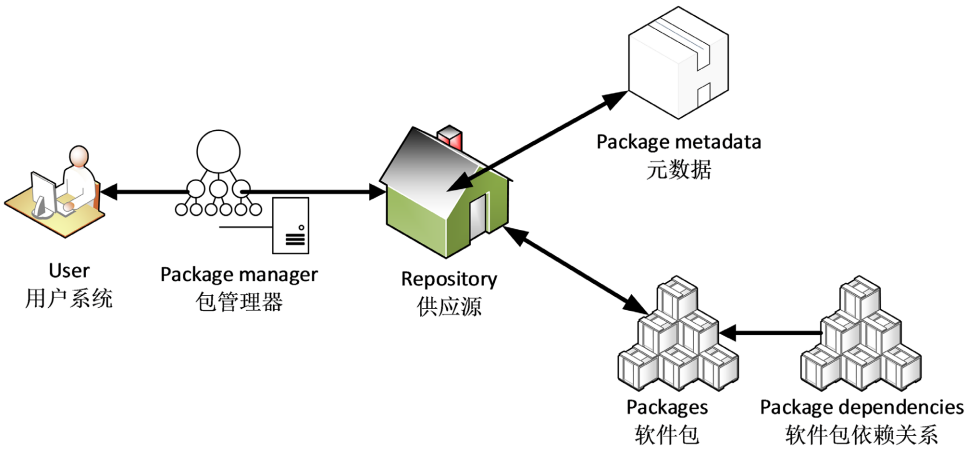

Figure 4. How package managers work
图 4. 包管理器工作原理

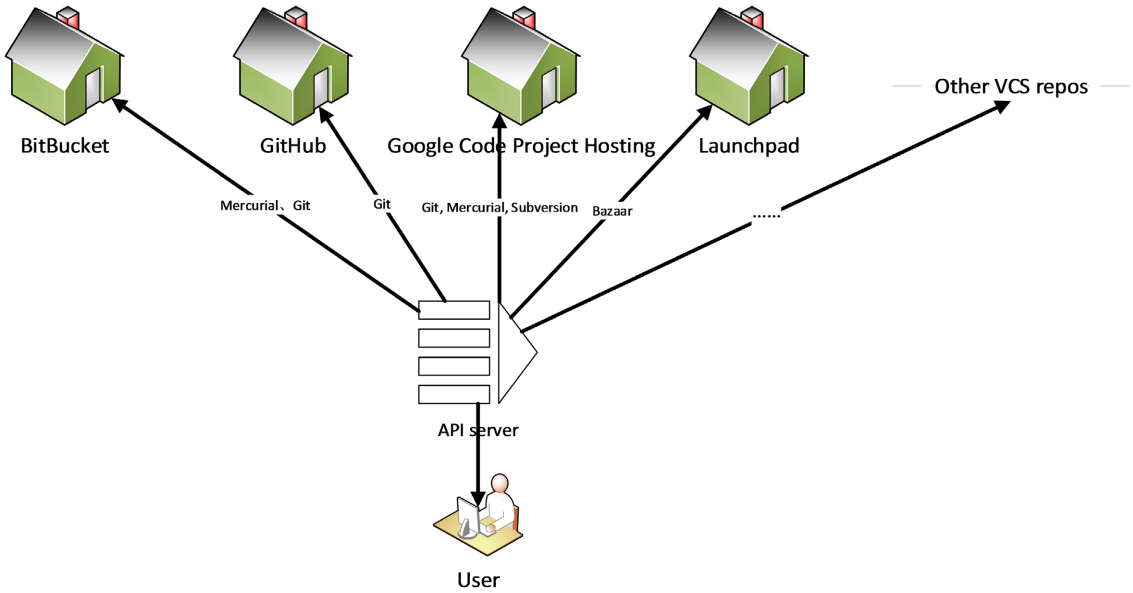

Figure 5. The principle of how Go Modules works
图 5. Go Modules 的工作原理

对单个 Go 项目供应链追踪方法包括以下几种：1) 使用包管理工具提供的指令 go mod graph (Go Modules)、dep status-dot (Go dep)；2) 开源工具，go-callvis (https://github.com/ofabry/go-callvis)、gmchart (https://github.com/PaulXu-cn/go-mod-graph-chart)、depth (https://github.com/KyleBanks/depth/releases)等；3) 对 go 项目的源码进行静态分析，在调用一个 package 的时候，需要在源码开头使用 import 语句，可以通过分析源代码中的 import 语句块来得到包内部的调用关系。

由于 Go 的包管理是去中心化的，对该生态全域的软件进行供应链分析会遇到和 C++子生态相同的问题——软件零散的分布在互联网的各个代码托管平台上，很难收集、链接。

4. 筛选和识别供应链软件的通用算法

通过上述四种编程语言的开源软件供应链的构建和管理进行总结和分析之后，提出了用于安装开源软件包的算法 1：

```
Input: package P
Begin
    Initial stack S
L.push(P)
While(S is not empty)
        s = L.pop()
        get package s metadata from a central repository or distribute repositories
        if s has not been Installed then
            download package s
            print successful information
        else if s conflicts with other Installed package
            print error information
            break
        for t in package s dependencies list
L.push(t)
End
```

虽然算法 1 已经可以用来建构简单的开源软件供应链，但是在实际构建的过程中，还是会遇到以下问题：

- 获取 package P 的特定版本数据，如果指定版本则获取该版本对于 meta data，如果未指定版本则默认拉取最新版本的 metadata。
- Package P 的 meta data 中并没明确依赖的特定版本号，例如对于 JavaScript 的 package.json 中的 dependencies 字段。其依赖的版本并没有明确指定，而是通过 semver 语义版本区间方式给出；3)算法 1 仅仅能够获取一个 package 的依赖信息，而一个软件供应链中的下游包可以有多个，需要同时对其进行溯源从而获得完整的供应链数据。
- 本节提出了一种更加通用的构建算法 2，该算法可以对整个生态数据进行分析。
- 可以同时计算多个 package 的供应链数据，实现整个生态的供应链进行构建。
- 与之前的供应链[11]研究不同的是，我们使用多叉树这一数据结构进行存储供应链中的每一个节点。构建过程可以分为多个步骤，1) 拉取当前 package 的元数据以及其子依赖的数据。2) 拉取子依赖的所有元数据，根据 semver 语义进行解析，得到最适合的版本号。3) 使用算法 1 中 DFS 方式进行迭代构建。
- 使用层序遍历的方式遍历已经构建好的供应链依赖树，并将其保存在 json 文件中。

算法 2：

```
Input: package list PL // PL 中的 package 不仅仅包含包名还包含特定的版本号
Begin
    initial queue<depNode> Q
    initial depNode root
    initial map<string>noLoop
    for p in PL
root.clidrenList.push(p)
Q.push(root)
While(Q is not empty)
        s = Q.pop()
```

Continued

```
            get package s record data from database
            use record data enrich s
Intial list depList from record data // depList 中的 package 包含包名和版本区间
            for dep in depList //  将 semver 语义版本区间转为特定版本号
                get package dep meta data from database
                use semver rules clean the dep's version
                if noLoop[dep.name] == 0 || noLoop[dep.name] == s.layer +1
s.clidrenList.push(dep)
Q.push(dep)
noLoop[dep.name] = s.layer + 1
        use LevelOrder save the tree to json
End
```

5. 实验结果与分析

不同于直接从 npm 或者 composer 等中心仓库中拉取包信息，我们使用 xlab 实验室开发的 OpenDigger (https://github.com/X-lab2017/open-digger)这一开源数据采集工具，将每一个开源软件包的数据分为 record data 和 meta data 两个部分。record data 中记录了包的各种元数据信息，包括包的名称、版本号、维护者信息、许可证以及依赖数据等等。在构建供应链一个节点时候，通过 record data 对节点进行数据丰富，并且提取出子依赖的名称以及 semver 版本号信息。而 meta data 则包含了 package 的所有历史发行版本和日期，通过发行日期限制我们可以很方便地避免依赖回溯(当前 package 依赖到了未来发行的子依赖包，这对于各种指标计算很有帮助)这一问题，而 meta data 最大的作用就是获得“干净”的版本号。以 JavaScript 的 package.json 为例，其记录的子依赖的版本遵循 semver 语义规则，这意味着无法直接获得一个确定的版本号。这里我们使用 JavaScript 中默认的规则，即符合 semver 语义的最大版本号。例如 semver 版本号 1.x，则取最大的版本号 1.9，而忽略版本 2.0。

在构建整个供应链树时候，算法 2 使用队列辅助从上至下进行迭代计算，需要注意的是，每一个开源软件包在供应链中都有唯一的层级，一般认为离制品软件最接近也就是层级越低的软件，其传递漏洞的可能性越高，即重要性越高。这里我们规定供应链下游出现过的开源软件，不会在上游进行重复计算，这不仅避免了依赖重要性混淆这一问题，同时也解决了潜在重复依赖成环导致构建失败的可能性。这就是算法 2 中 noLoop 的作用，只有开源软件包没有被计算过或者其层级没有降低，我们才把当前软件包加入到供应链节点中。

本文同时还发布了两个全新的供应链数据集，NPM15K 和 COMPOSER15K。表 1 展示了两个数据集的详细情况。NPM15K 是随机采样的 15K 个 JavaScript 生态下开源软件包数据，每一项软件包数据中包括了包名称以及在 2021 年发行的所有版本号和发行时间。而 COMPOSER15K 则是随机采样的 15K 个 PHP 生态下开源软件包数据，含有相同的数据类型。对这两个数据集进行分析表明，JavaScript 生态的开源软件包 2021 年平均发布 10.94 个新版本(包括 patch)，而 PHP 生态的开源软件包 2021 年平均发布 5.55 个新版本。JavaScript 在如今的受欢迎程度要远远高过 PHP，这充分反应在了其开源软件生态中。

我们使用开发的工具，在这两个数据集上进行供应链构建，实验的机器 CPU i7-6700HQ，内存为 16G，网络速度 100 Mbps，数据存储在阿里云的 mongodb 云数据库中。

实验结果表示，JavaScript 与 PHP 生态下的开源软件包具有需要相似的特性。JavaScript 生态下的供应链和 PHP 生态下供应链特性相似，其平均依赖数和构建得到的平均供应链深度都很接近。图 6 显示这两个生态下的软件依赖数分布均接近幂律分布，但 PHP 下包的供应链深度分布与 JavaScript 相比较为均

匀。并且我们在 15K 个 JavaScript 开源包即 NPM15K 上进行供应链构建仅耗时 15 h，在 COMPOSER15K 数据集上更是只用时 8 h，这证明我们提出的算法 2 具有优秀的供应链构建效率。

Table 1. Supply chain construction details
表 1. 供应链构建详情

数据集	平均发行版本数	平均依赖数	平均供应链深度	构建时长(h)
NPM15K	10.94	7.17	3.61	15.60
COMPOSER15K	5.55	7.23	3.60	8.14

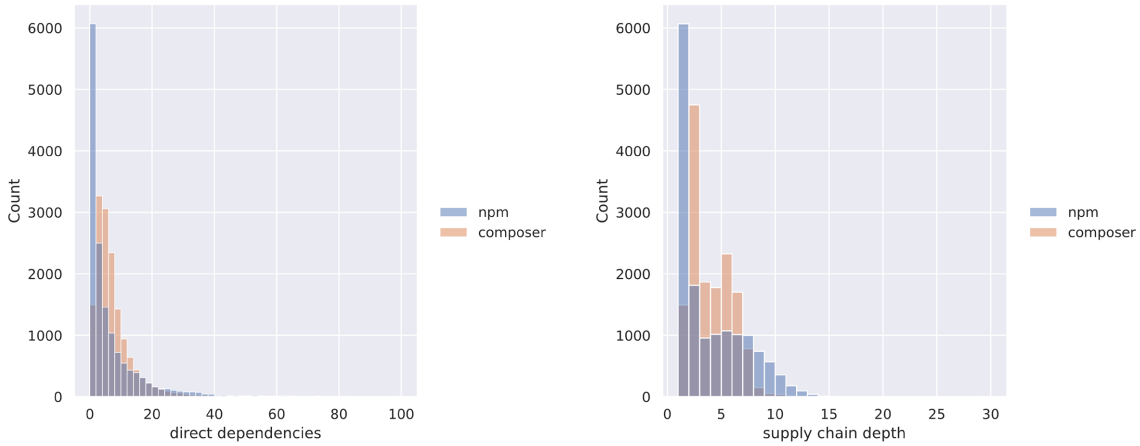

Figure 6. Dependency distribution and supply chain depth distribution of packages on two datasets
图 6. 两个数据集上 package 的依赖分布和供应链深度分布

6. 总结与展望

在本文中，我们归纳总结了四种常见的开源依赖策略，并基于此提出了一种通用的开源供应链软件筛选和识别算法。在两个数据集 NPM15K 和 COMPOSER15K 上的实验结果表明，该算法具有优秀的性能表现，十分适合运用到整个编程语言生态的探索中。当前各种生态管理外部依赖的方式差异很大，软件供应链追踪方式尚缺乏充分的形式化分析，这些都有待做更多工作。

参考文献

[1] 梁冠宇, 武延军, 吴敬征, 赵琛. 面向操作系统可靠性保障的开源软件供应链[J]. 软件学报, 2020, 31(10): 3056-3073. https://doi.org/10.13328/j.cnki.jos.006070

[2] Spinellis, D. (2012) Git. *IEEE Software*, **29**, 100-101. https://doi.org/10.1109/MS.2012.61

[3] 李学彬. 开源软件依赖可满足性识别方法研究与实现[D]: [硕士学位论文]. 沈阳: 东北大学, 2008.

[4] Ballou, R.H. and Srivastava, S.K. (2007) Business Logistics/Supply Chain Management: Planning, Organizing, and Controlling the Supply Chain. Pearson Education, New York .

[5] Tan, K.C., Kannan, V.R., Handfield, R.B., *et al*. (1999) Supply Chain Management: An Empirical Study of Its Impact on Performance. *International Journal of Operations & Production Management*, **19**, 1034-1052. https://doi.org/10.1108/01443579910287064

[6] Belguidoum, M. and Dagnat, F. (2007) Dependency Management in Software Component Deployment. *Electronic Notes in Theoretical Computer Science*, **182**, 17-32. https://doi.org/10.1016/j.entcs.2006.09.029

[7] 向胜军, 周树杰. C++代码复用技术之分析[J]. 北京石油化工学院学报, 2003, 11(4): 41-44.

[8] 谷凤伟. 基于 Makefile 文件依赖的源码分析工具设计与实现[D]: [硕士学位论文]. 南京: 南京大学, 2016.

[9] Varanasi, B. (2019) Introducing Maven: A Build Tool for Today's Java Developers. Apress, New York. https://doi.org/10.1007/978-1-4842-5410-3

[10] 董晓光, 喻涛. 使用 Maven 构建 java 项目[J]. 电子技术与软件工程, 2014(10): 105.

[11] Decan, A., Mens, T. and Grosjean, P. (2019) An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems. *Empirical Software Engineering*, **24**, 381-416. https://doi.org/10.1007/s10664-017-9589-y