

基于知识图谱的可视化系统的设计与实现

摘要

近年来，互联网信息安全漏洞造成的风险不断增加。由于漏洞信息的收集与发布工作分散在互联网各处，且通常以纯文本的形式发布共享，这造成网络安全工作者及软硬件开发者，难以轻松直观地获知相关项目或资产的漏洞风险情况。而提供类似可视化漏洞知识图谱服务的多为商业公司、其系统不公开且收费高昂。

基于此背景，本项目设计并实现了使用 GPL-3.0 自由软件许可证授权开源的“基于漏洞知识图谱的可视化系统”。

本文第二章进行数据采集、知识图谱、持久化、可视化等功能的技术调研与理论基础介绍，分析并确定了开发本系统采用的技术栈。第三章针对系统数据流、功能和非功能需求进行了需求分析，细化系统开发目标。第四章进行了系统概要设计、系统功能模块设计、系统详细设计：针对系统数据流分析，设计系统总体及分层结构，将系统划分为五个子系统，形成项目框架；随后划分功能模块、针对每个功能模块明确其功能与职责，并为模块绘制了类图与功能时序图。第五章以流程图形形式阐述各个子系统的设计与实现，并以文字与伪代码形式阐述关键功能模块的设计与实现。第六章对系统运行速度与运行结果进行简单测试，以截图形式展示可视化系统的实现效果。第七章总结了本项目所做的工作，并展望未来系统的发展方向。

关键词 知识图谱 信息安全 漏洞 信息融合 可视化

Design and Implementation of Visualization System Based on Vulnerability Knowledge Graph

ABSTRACT

In recent years, the risks caused by Internet information security vulnerabilities have been increasing. Since the collection and distribution of vulnerability information is scattered all over the Internet and usually shared in the form of plain text, it is difficult for network security researchers and hardware and software developers to easily visualize the vulnerability risk of related items or assets. Most of the companies that provide similar vulnerability knowledge graph visualization services are commercial companies, whose systems are not public and charge high fees.

Based on this background, this project designs and implements a “Vulnerability Knowledge Graph-based Visualization System” open-sourced under the GPL-3.0 Free Software License.

Chapter 2 of this paper introduces the technical research and theoretical foundation of data collection, knowledge graph, persistence, visualization and other aspects of the system, and analyzes and determines the technology stack used to develop this system. Chapter 3 conducts requirements analysis for system data flow, functional and non-functional requirements, and refines the system development objectives. Chapter 4 presents the system outline design, system functional module design, and system detailed design: for the system data flow analysis, the overall and hierarchical structure of the system is designed, and the system is divided into five subsystems to form the project framework. Then the functional modules are divided, their functions and responsibilities are clarified for each functional module, and class diagrams and functional timing diagrams are drawn for the modules. Chapter 5 illustrates the design and implementation of each subsystem in the form of flowcharts, and the design and implementation of key functional modules in the form of text and pseudo-code. Chapter 6 conducts a simple test on the running speed and results of the system, and demonstrates the visualization system in the form of screenshots. Chapter 7 summarizes the work done in this project and looks forward to the future development of the system.

KEY WORDS Knowledge Graph Information Security Vulnerability Information Integration Visualization

目 录

第一章 絮论	1
1.1 项目背景及意义	1
1.1.1 选题背景	1
1.1.2 国内外研究现状	1
1.1.2.1 公开数据库	1
1.1.2.2 商业项目	2
1.1.3 项目意义	2
1.2 开发目标	2
1.3 软件工程方法	3
1.4 授权与许可证	3
第二章 相关技术介绍	4
2.1 数据采集技术	4
2.1.1 Scrapy	4
2.1.2 Pandas	4
2.2 知识图谱构建技术	5
2.2.1 知识图谱概念	5
2.2.2 知识图谱构建过程	5
2.2.3 Ray	6
2.3 持久化技术	6
2.3.1 MongoDB	6
2.3.2 Neo4j	7
2.4 后端技术	7
2.4.1 Flask	7
2.4.2 Advanced Python Scheduler	8
2.5 前端技术	8
2.5.1 Vue.js	8
2.5.2 Vuetify	9
2.5.3 Apache ECharts	9
第三章 系统需求分析	10
3.1 外部数据流分析	10
3.2 功能性需求分析	10
3.2.1 漏洞数据采集	10

3.2.2 漏洞数据处理	10
3.2.3 知识图谱构建	11
3.2.3.1 实体生成	11
3.2.3.2 关系生成	11
3.2.3.3 关系融合	11
3.2.4 知识图谱持久化存储	11
3.2.5 可视化展示知识图谱	11
3.2.6 日志功能	12
3.3 非功能性需求分析	12
3.3.1 性能需求	12
3.3.2 兼容性需求	12
3.4 用户视角的需求分析	12
第四章 系统设计	14
4.1 系统概要设计	14
4.1.1 系统总体设计	14
4.1.2 系统层次结构设计	16
4.1.2.1 表示层	16
4.1.2.2 逻辑层	17
4.1.2.3 持久化层	17
4.2 系统功能模块设计	17
4.2.1 数据采集子系统	17
4.2.2 知识图谱构建子系统	19
4.2.2.1 本体定义	19
4.2.2.2 实体生成	20
4.2.2.3 关系生成	21
4.2.2.4 关系融合	21
4.2.3 后端服务子系统	22
4.2.4 前端子系统	22
4.2.5 数据库设计	22
4.2.5.1 MongoDB	22
4.2.5.2 Neo4j	23
4.2.6 系统接口设计	26
4.3 系统详细设计	27
4.3.1 开发环境配置	27

4.3.2 各核心功能模块详细设计	27
4.3.2.1 类图	27
4.3.2.2 功能时序图	32
第五章 系统实现	33
5.1 工具模块	33
5.1.1 根目录模块实现 bot_root_dir.py	33
5.1.2 日志模块实现 logger_factory.py	33
5.1.3 环境配置文件 secret.py	33
5.2 数据采集子系统	34
5.2.1 Spider 模块实现	34
5.2.1.1 初始化 __init__() 实现	34
5.2.1.2 发起请求 start_requests() 实现	35
5.2.1.3 解析数据 parse() 实现	35
5.2.2 Item Pipeline 模块实现	35
5.2.3 其他模块实现	36
5.3 知识图谱构建子系统	36
5.3.1 实体生成	36
5.3.1.1 init_nodes() 实现	36
5.3.1.2 漏洞实体生成的实现 init_vul_ray()	38
5.3.1.3 资产实体生成的实现 init_asset_ray()	38
5.3.1.4 利用代码实体生成的实现 init_exploit_ray()	39
5.3.1.5 资产家族实体生成的实现 init_asset_family_ray()	39
5.3.2 关系生成	39
5.3.2.1 关系生成的实现 init_rels()	40
5.3.2.2 漏洞 - 资产家族关系生成的实现 create_rel_vaf_ray()	40
5.3.2.3 资产家族 - 资产关系生成的实现 create_rel_afa_ray()	41
5.3.2.4 利用 - 漏洞 - 资产家族关系生成的实现 create_rel_evaf_ray()	41
5.3.3 关系融合实现	42
5.4 持久化子系统实现	42
5.4.1 MyMongo 类实现	42
5.4.2 MyNeo 类实现	43
5.5 后端服务子系统实现	43
5.5.1 知识图谱 API 实现 graph.py	44
5.5.1.1 知识图谱统计数据 API 实现 /api/graph/	44
5.5.1.2 知识图谱可视化数据 API 实现 /api/graph/<limit>	44

5.5.1.3 知识图谱搜索 API 实现 /api/graph/search/<keyword>	45
5.6 前端子系统	46
5.6.1 项目结构	46
5.6.2 概览视图实现	47
5.6.3 可视化视图实现	48
5.6.4 搜索视图实现	48
5.6.5 关于视图实现	48
第六章 系统测试	49
6.1 测试环境	49
6.2 数据采集子系统测试	49
6.3 知识图谱构建子系统测试	49
6.3.1 结点生成	49
6.3.2 关系生成	51
6.3.3 关系融合	51
6.4 持久化子系统与后端服务子系统测试	51
6.4.1 概览视图统计信息获取 /api/graph/	51
6.4.2 可视化视图数据获取 /api/graph/<limit>	51
6.5 前端子系统测试	53
6.5.1 概览视图测试	53
6.5.2 可视化视图测试	53
6.5.3 搜索视图测试	55
6.5.4 关于视图测试	56
第七章 结束语	57
7.1 项目工作总结	57
7.2 未来工作展望	57
参考文献	
致 谢	

第一章 绪论

1.1 项目背景及意义

1.1.1 选题背景

近年来，随着互联网产业迅速发展，互联网安全漏洞问题的显著性也急剧增加。根据公共漏洞和暴露¹、国家信息安全漏洞库²等业内权威的漏洞数据档案库的数据^{[1][2]}，自 1999 年 CVE 漏洞库首次披露安全漏洞以来，互联网安全漏洞年新增数量呈增长趋势。2019 年全年，新增漏洞两万余个；2020 年全年，新增漏洞三万五千余个。随着现代软件系统复杂度提升、互联网加速漏洞信息传播，对攻击者而言，不仅漏洞攻击的学习成本和实施难度下降，其可以利用的漏洞数量也明显增多^[3]；对企业与开发者而言，随着开源化逐渐成为一种潮流趋势，各类计算机软硬件与互联网产品对开源项目的依赖性随之提高。正如近期 Java Log4j 日志组件漏洞造成全球互联网范围的大规模信息安全问题，计算机与互联网产业在快速发展的同时，也面临着与日俱增的信息安全挑战。

1.1.2 国内外研究现状

网络安全意指一组用于保护计算机、网络、程序与数据免受攻击、未授权访问、篡改、破坏的技术与步骤^[4]。提供详实可靠的漏洞信息可以有效地帮助互联网安全研究人员进行网络安全研究，以应对可能存在的攻击和利用。针对互联网漏洞数据，主要分为公开数据库与商业项目。

1.1.2.1 公开数据库

如 cve.mitre.org、nvd.nist.gov、vuldb.com 等。

- 优势：

- 数据更新及时，CVE 与 CPE 为官方数据源
 - 开放数据库，人人可获取

- 劣势：

- 无知识图谱，不具备可视化功能，仅提供 JSON 或 CSV 文件格式存储关系型数据
 - 按照 CVE ID 索引漏洞信息，难以发现漏洞之间联系
 - 信息分散在多个站点，难以整合，为研究人员带来不便

¹Common Vulnerabilities and Exposures, CVE

²China National Vulnerability Database of Information Security, CNNVD

1.1.2.2 商业项目

如百度安全知识图谱¹、绿盟安全知识图谱²等。

- 优势:

- 成熟的企业级解决方案与技术支持
- 具备知识图谱的可视化功能

- 劣势:

- 数据不开放，对安全研究人员及软件开发者意义有限
- 价格高昂

1.1.3 项目意义

本课题针对上述问题，提出一种漏洞知识图谱可视化系统。基于爬虫抽取、知识图谱、图数据库、可视化前端等技术，对多种数据源的互联网公开漏洞数据，包括漏洞描述及风险评估、受影响资产、可利用代码及补丁等信息进行收集与分析，通过对异构数据源抽取的数据进行实体构建、关系构建、关系融合等处理，在图数据库中建立漏洞本体信息及其之间的关联信息，从而形成具有一定知识结构的知识图谱。基于该漏洞知识图谱，搭建基于 B/S 架构的可视化系统，提供易于使用的接口、用户友好的 UI 界面呈现漏洞知识图谱信息、进行创建统计图表、知识筛选等操作。

本系统采用自动化的方式，实现对漏洞信息的持续收集与整理，极大节省了人力资源的消耗。结合抽取关键信息建立互联网信息安全本体、对漏洞间关联性进行分析、构建漏洞知识图谱，将分散的漏洞信息转化为相互联系的图结构，本系统将为开发者提供项目依赖安全性参考、为计算机信息安全研究人员提供逻辑清晰、直观易于理解的统计数据与服务支撑。

本系统作为自由软件^[5]发布，使用 GPL-3.0^[6]许可证授权，自由与开放促进构建更高效安全的互联网环境。

1.2 开发目标

- 设计并实现对异构公开互联网信息漏洞数据源的信息采集子系统。
- 设计并实现基于规则的漏洞知识图谱构建子系统。
- 设计并实现基于图数据库的漏洞知识图谱持久化子系统。
- 设计并实现基于上述漏洞知识图谱的后端服务子系统，提供 RESTful API 访问点。

¹<https://anquan.baidu.com/product/skg>

²https://www.nsfocus.com.cn/html/2022/209_0510/163.html

- 设计并实现基于 Web 服务的漏洞知识图谱可视化子系统，具备独立前端。
- 对上述系统进行系统测试、排错、功能扩展、性能优化，编写文档记录。

1.3 软件工程方法

由于本项目为单人完成，且系统整体结构较为复杂，故采用原型开发与敏捷开发^[7]相结合的方式，先开发最简可行产品^[8]（Minimum Viable Product, MVP）验证核心概念，在此基础上不断细化细节、完善代码、扩展功能。同时，因为单人项目难以在前期进行完善的需求分析与系统设计，采用敏捷开发的方式可以增强灵活性。

例如，本项目立项初期技术选型，计划使用较为成熟的 Java Springboot 框架搭建后端服务。在之后的开发中发现，由于该项目对数据库操作需求较高，其他子系统使用 Python 而后端使用 Springboot 意味着需要使用 Python 和 Java 分别编写两套代码用于控制数据库驱动，增加无意义的项目复杂度、拖慢开发进度。得益于敏捷开发思想“响应变化高于遵循计划”等方针，本项目在知识图谱构建子系统的原型开发完成时即更改后端使用 Python Flask 进行开发，从而保证软件质量的同时提升开发效率。

1.4 授权与许可证

本系统作为自由软件^[5]发布，使用 GPL-3.0^[6] 许可证授权，全部代码在 GitHub¹ 开源。

¹<https://github.com/RiddMa/KnowledgeGraph-Visualization>

第二章 相关技术介绍

根据系统开发目标，可将系统层次结构从信息处理逻辑上分为三个层次：用于持久化存储信息的持久化层，用于对信息进行采集、处理、传输的逻辑层，用于面向用户展示信息的表示层；从功能上分为五类技术：数据采集技术、知识图谱构建技术、持久化子技术、后端技术、前端技术。本章从功能的五个方面分别介绍本项目使用的相关技术。

2.1 数据采集技术

采用的技术或工具：Scrapy、Pandas

2.1.1 Scrapy

Scrapy¹ 是自由且开源的协作式网络爬虫 Python 框架，用于从网站或 API 中提取需要的数据。Scrapy 基于 Twisted，具有快速、易用、可扩展等特性，主要优点是架构清晰、模块间的耦合程度低、通过 Middleware 钩子框架能灵活完成各种需求。Scrapy 项目围绕 Spider 类构建，并具有 Item Pipelines、Downloader、Scheduler 等多个模块，这些模块通过内置的 Scrapy Engine 进行管理与调度。Scrapy 设计理念遵循“一次且仅一次”（Once and only once, OAOO）原则，主张为每个爬虫任务只需要编写一个自包含（self-contained）的 Spider 类，没有更多，没有更少。

相较于 urllib 或 Python Requests 等基础的库，Scrapy 提供任务日志、错误重试、并发控制，具备更灵活完善的机制以应对大型爬虫任务。而相较于 Selenium、Puppeteer、Playwright 等基于无头浏览器（Headless Browser）实现的爬虫功能，Scrapy 基于事件驱动的网络编程框架 Twisted，仅爬取 HTML 文档，默认不提供执行网页中的 JavaScript 代码功能，因此 CPU 与内存开销相较无头浏览器显著减小，是最适合本系统大量数据爬取需求的框架。

2.1.2 Pandas

Pandas² 是适用于 Python 语言的数据操纵与分析库，是使用 BSD 许可证授权发行的自由且开源的软件，尤其擅长数值表格与时间序列的数据结构和运算操作。

本数据采集子系统将从 cve.mitre.org 爬取其提供的包含全部 cve id 信息的 csv 文件，其中包含数十万条 cve 漏洞 id 条目。由于该 csv 文件包含表头信息及一些无效或重复的 cve id，在建立用于从异构数据源爬取项目的 cve id 索引时需要将无关信息删去。该 csv 文件体积较大（约数百兆字节），因此使用 Pandas 提供的流式读取 csv 文件、迭代操作的方法处理此文件。

¹<https://scrapy.org/>

²<https://pandas.pydata.org/>

2.2 知识图谱构建技术

采用的技术或工具：基于规则的实体构建、基于规则的关系生成、Ray。实体构建与关系生成技术将在章节 4.2.2 知识图谱构建子系统展开。

2.2.1 知识图谱概念

近年来，使用知识图谱表达人类知识结构受到了学术界与工业界的极大关注。知识图谱是事实、实体构成、关系和语义的结构化表达，是一种用图表示的数据结构。知识图谱存储实体结点、结点之间的关系，旨在积累并表达现实世界的知识。^[9]

知识图谱的图结点代表被研究的实体，实体可以是现实世界的对象或抽象的概念。实体结点之间的边代表实体之间的关系，以及实体的语义描述，它们的关系具有类型和明确定义的属性。^[9] 用于存储知识图谱数据的图遵从基于图结构的数据模型，如有向的边标记图或属性图。^[10]

通常以三元组的形式表达知识图谱结构，即

$$G = (Entity_{head}, Relation, Entity_{tail})$$

$Entity_{head}$ 为三元组 G 的头实体，是关系的起点。 $Entity_{tail}$ 为三元组 G 的尾实体，是关系的终点。实体表示被建模的现实的对象或抽象的概念。 $Relation$ 则表示 $Entity_{head}$ 到 $Entity_{tail}$ 的语义化关系。^[10]

知识图谱在信息检索中的应用价值主要在于，它能通过推理实现概念检索，以图形化的方式，向用户展示经过分类整理的结构化知识，从而使人从人工筛选、过滤数据寻找答案的模式中解放出来。在漏洞数据库的基础上，构建漏洞知识图谱，挖掘互联网信息安全漏洞内在特性、关联关系等价值，有助于多维度、全方位地理解安全漏洞关系。

2.2.2 知识图谱构建过程

知识图谱主要包含开放知识图谱和领域知识图谱。开放知识图谱如 Wikipedia、Freebase、DBpedia 等，往往采用自底向上方法构建。而互联网信息安全漏洞知识图谱属于领域知识图谱，适宜采用自顶向下构建方法，即通过人工整理出领域知识图谱的实体及关系模型图，进行实体和关系的抽取与知识图谱的构建。在安全漏洞领域中，预先定义的实体与关系模型能极大提高知识图谱的构建质量及应用效率。^[11]

实体抽取：又称命名实体识别（Named Entity Recognition, NER），主要分为基于规则的方法、基于传统机器学习的方法、基于深度学习的方法^[12]，后两者主要用于解决从自然语言文本中抽取未预定义格式信息的问题。本项目相关安全漏洞领域实体具有明确名称与格式，适宜采用基于规则的方法提取。未来系统扩展如博客、新闻等小部分自然语言数据源，可采用基于深度学习的方法如 CNN-Bi-LSTM-CRF 等模型进行命名实体识别。

关系抽取 (Relationship Extraction, RE)：主要分为基于传统规则和模板的方法、基于传统机器学习的方法、基于深度学习的方法^[12]。安全漏洞领域实体如漏洞、资产、利用等类别较少且具备明确语义关系，因此采用基于传统规则的关系抽取方法。

2.2.3 Ray

Ray¹ 是一个开源的通用分布式计算框架，由加州大学伯克利分校的 RISE 实验室开发。Ray 为构建分布式应用提供了一个简单、通用的 API，提供了运行机器学习工作流的 Ray ML 工具箱、运行分布式应用的 Ray Core 核心框架、部署大规模工作负载的 Ray Cluster 集群等，具有通用的分布式计算抽象以及优秀的性能。

相较传统的分布式框架（如 Hadoop、Spark 等），Ray 可直接通过 pip 进行安装，具备轻量级的特性。同时它通过共享内存实现了高效的数据存储和传输，通过全局状态存储服务实现了全局的状态维护、去中心化的高效调度、远程调用。相较基于线程的并行库 threading，Ray 通过进程级并行绕过了 Python 全局解释器锁的限制。相较轻量级的 Python 进程级并行库 multiprocessing，Ray 具备同样简单易用的接口以及更优秀的性能与机器学习框架整合能力。

本项目通过使用 Ray 框架提供的通用分布式计算能力，对上层封装具体实现；将知识图谱构建过程抽象成创建结点、创建关系、关系融合三个操作，重点优化这三个步骤的代码实现，使“数据采集子系统得到的原始数据通过知识图谱构建子系统形成知识图谱”过程得以实现并行化计算，充分利用多核心处理器的计算能力。同时，得益于 Ray 框架对本地多进程与分布式多进程的统一抽象、Ray ML 框架与深度学习模型良好的整合能力，未来可以引入深度学习模型、将系统迁移扩展至分布式计算平台。

2.3 持久化技术

采用的技术或工具：MongoDB、Neo4j

2.3.1 MongoDB

MongoDB² 是一个 C++ 语言编写的基于分布式文件存储的开源文档型数据库系统，是最受欢迎的 NoSQL 数据库。MongoDB 基于 JSON/BSON 格式的文档存储，相较传统 SQL 数据库可以表示灵活的数据结构；具备动态 DDL 能力、没有强 Schema 约束的特性使其支持开发者进行快速迭代；提供基于内存的快速数据查询，实现高并发计算性能；提供数据分片能力，分布式扩展性强。

本项目需求从异构数据源采集互联网信息安全漏洞数据，导致数据格式繁多，在项目初期规定数据所采用的 Schema 格式将大幅增加数据转换工作量，并不现实。因此，传统 SQL 关系型数据库并不适合本项目的开发需求。而 MongoDB 提供面向文档的 JSON

¹<https://docs.ray.io/en/latest/>

²<https://www.mongodb.com/>

存储，适配互联网爬虫数据采集需求。同时，JSON 文件具备良好的序列化能力，且与本项目后端服务子系统 RESTful API 所需的 Application/JSON Response 相匹配。更进一步地，MongoDB 具有优秀的分布式能力。综上原因，采用 MongoDB 作为存储知识图谱所用原始数据的数据库。

2.3.2 Neo4j

Neo4j¹ 是一个高性能的图形数据库，是目前全球范围内最受欢迎、使用人数最多的图数据库，社区版本采用 GPL-3.0 许可证授权。其特点是将结构化数据存储在网状结构上，使得长程、广范围关系查询变得更加容易实现，Neo4j 是目前使用最多的图数据库。本项目知识图谱构建子系统使用 Neo4j 存储图谱数据，将漏洞、资产、利用代码等实体作为结点，为结点之间添加“影响”、“具有”、“攻击”、“被攻击”、“父级”、“子级”等关系，形成知识网络图谱。

2.4 后端技术

采用的技术或工具：Flask、Advanced Python Scheduler

2.4.1 Flask

Flask² 是一个使用 Python 编写的轻量级 Web 应用框架，具备微核心与高扩展性，使用 BSD 许可证授权发行。Flask 是在 Jinja2 模板引擎和 Werkzeug WSGI 工具箱的基础上构建的。Werkzeug 是一个语言网络服务器网关接口（Web Server Gateway Interface, WSGI），为请求、响应等功能实现软件对象；Jinja 是一个适用于 Python 的网页模板引擎。

本项目主要使用 Flask 框架的 WSGI 功能，构建一个 RESTful 风格的 Web API 服务，为 Vue.js 编写的前端提供与数据库的交互功能。本着“高内聚，低耦合”的设计原则，同时兼具部署简洁、外部依赖较少、用户控制灵活的优点，本基于漏洞知识图谱的可视化系统将前述数据采集子系统、知识图谱构建子系统抽象成自包含的服务，仅通过持久化层的 MongoDB 及 Neo4j 数据库进行纯数据交换，最大程度上减少模块间逻辑耦合。

由于数据采集子系统、知识图谱构建子系统、持久化子系统、后端子系统均采用 Python 语言编写，此四个子系统得以共用一个高扇入的数据库驱动模块。这种设计将会导致单个数据库控制类非常庞大，在本项目开发后期该数据库控制类相关代码已接近千行，造成运行时实例化开销昂贵。并且，基于知识图谱数据量庞大的事实，本系统采用由加州大学伯克利分校牵头开发的开源分布式高性能计算库 Ray，实现知识图谱数据构建的并行计算并提供可能的分布式扩展性。为此，数据库控制类需要实现线程安全。

¹<https://neo4j.com/>

²<https://flask.palletsprojects.com/en/2.1.x/>

为解决此问题，针对本系统将频繁进行数据库操作的预期，利用 Python 语言引入模块为单例的特性实现了一个“饿汉模式”的数据库控制类。在其他上级模块中引入数据库控制器类的实例化对象，使每个模块运行时在内存中只有一个实例，减少运行时频繁地创建和销毁类实例的开销。相较使用 Java Springboot 等其他语言框架编写后端服务，采用 Python Flask 并共用单例数据库控制模块的设计在减少重复编码工作的同时，极大增强了代码可维护性。

2.4.2 Advanced Python Scheduler

Advanced Python Scheduler¹(下称 APScheduler)是一个基于 Python 的定时任务库，提供 Cron 风格、间隔执行、单次延时执行三种任务规划模式；内存、SQLAlchemy、MongoDB、Redis 数据库等任务存储方式；asyncio、Twisted、Qt 等 Python 框架集成。APScheduler 并非命令行工具，而旨在向运行中的应用程序增加定时任务功能，且具备 Python 跨平台的优势，因此非常适合作为 Python 后端的定时任务模块。

后端服务作为一个高扇出模块，是整个系统的运行控制中心，同时控制后端 Web API 服务、数据采集服务、知识图谱构建服务。根据本项目需求，为实现知识图谱的动态更新，在 Flask 后端服务中集成 APScheduler 库用于控制和管理数据采集服务、知识图谱构建服务，实现周期性定时运行数据采集与知识图谱构建的功能。相较使用 crontab、Windows Task Scheduler 等依赖操作系统运行环境的外部工具，采用 Flask + APScheduler 进行服务运行管理的设计更加优雅简洁，在保持优秀可迁移性之外，还允许管理员用户透过 HTTPS 协议在前端子系统提供的 Web 应用中对系统运行状态进行远程管理，无需接触物理机或使用 SSH 等协议连接至终端。该设计增强系统安全性并提高了系统易用度。

2.5 前端技术

采用的技术或工具：Vue.js、Vuetify、Apache ECharts

2.5.1 Vue.js

Vue.js² 是一套用于构建用户界面的开源 MVVM 渐进式前端 JavaScript 框架，使用 MIT 许可证授权。通过 Vue.js 提供的模板语法、计算属性和侦听器、条件渲染、组件等功能，可以构建现代化的响应式前端页面。Vue 既可以驱动一套完整的单页应用，也易于与第三方库或既有项目进行整合。

本项目使用 Vue.js 作为前端逻辑的基础框架，结合 Vuex³、Vue Axios⁴、Vue Router⁵、

¹<https://apscheduler.readthedocs.io/en/latest/>

²<https://v3.cn.vuejs.org/>

³<https://vuex.vuejs.org/zh/>

⁴<https://www.npmjs.com/package/vue-axios>

⁵<https://router.vuejs.org/>

register-service-worker¹ 等实用库，构建一套现代化的前端页面，具备响应式、单页应用 (Single Page Application, SPA)、渐进式网页应用 (Progressive Web Application, PWA) 等特性。

2.5.2 Vuetify

Vuetify²是一个 Material 样式的 Vue UI 组件库，使用 MIT 许可证授权。它包含一系列预定义样式的 UI 组件，可用于构建前端应用程序的用户界面。

本项目前端子系统采用 Vuetify 作为 UI 库，致力于搭建一个用户友好、信息密度大、界面简洁易用的前端应用。借助 Vuex 状态管理库及 Vue.js 计算属性，实现尽可能减少跨组件回调函数使用的“Single Source of Truth”模式单向数据流的前端应用。降低组件间逻辑耦合，实现快速开发、轻松扩展。

2.5.3 Apache ECharts

Apache ECharts³（下称 ECharts）是一个自由且开源的 JavaScript 可视化库，采用 Apache License 2.0 许可证授权。ECharts 基于轻量级绘图库 zrender，为浏览器提供直观的、强大的、可交互的、高度自定义的数据可视化功能。

本项目使用 ECharts 中力引导图绘制可视化的知识图谱并提供一定交互功能；使用各类数据统计图，直观地呈现知识图谱的相关统计数据。借助 Vue.js 的模板组件与监听属性能力，实现可复用、响应式的可视化数据呈现能力。

¹<https://www.npmjs.com/package/register-service-worker>

²<https://vuetifyjs.com/zh-Hans/>

³<https://echarts.apache.org/en/index.html>

第三章 系统需求分析

产品需求分析是针对即将开发的软件施加的一种要求或限制。软件产品需求可分为功能性需求、非功能性需求等。^[13]

3.1 外部数据流分析

- 系统输入：来自多种数据源的漏洞相关异构信息数据，包括 HTML 文件、CSV 文件、JSON 文件、XML 文件等。
- 系统输出：
 - 以文本、统计表、统计图、交互式可视化图等形式呈现的知识图谱信息。
 - 系统自身运行状态等统计信息。

3.2 功能性需求分析

3.2.1 漏洞数据采集

本系统需要构建互联网信息安全漏洞相关的知识图谱，为此需要一定的数据源提供整个系统的输入。由于安全感知态势千变万化、CVE 漏洞数据时刻更新，本系统需要实现自动化数据采集功能，向数据库中动态添加数据，而非将现有数据一次性导入。调研筛选出待爬取的网站有：

- cvedetails.com，一个以表格形式简单聚合 CVE 漏洞信息的网站；
- cve.mitre.org，一个官方发布 CVE 漏洞命名的网站；
- cpe.mitre.org，一个官方发布 CPE 资产信息的网站；
- nvd.nist.gov，一个发布 CVE 详细风险评估及受影响 CPE 资产统计的数据库；
- exploit-db.com，一个提供漏洞利用代码的数据库；

3.2.2 漏洞数据处理

数据采集得到的输入数据不满足构建漏洞知识图谱所需数据结构或模式。例如：cvedetails.com 采集到的数据为 HTML 文档，而其中有效信息分散在 HTML DOM 树中；nvd.nist.gov 提供官方 JSON API 接口，采集得到的数据为 JSON 格式，但包含版本控制数据等与漏洞本身无关的信息；在 cve.mitre.org 得到的数据为独立 csv 文件，等。因此需要实现从采集到的数据转化为知识图谱数据的处理过程。

3.2.3 知识图谱构建

3.2.3.1 实体生成

为了构建漏洞知识图谱，需要从爬虫获取的互联网信息安全漏洞数据中获得所需要的实体与关系。首先需要进行实体生成。如前所述，图谱中一个结点代表现实世界中一个实体，适用于此处即为一个与漏洞关联的实体，如漏洞条目、软件资产、硬件资产、资产家族、利用代码等。

由于实体分属于不同种类，因此需要标签（label）进行区分。每个实体还应该具有不同的属性，这些属性应该被存储到图数据库结点中。

3.2.3.2 关系生成

在有向边标记知识图谱中的关系通常只具有一些简洁的标签（label），代表 head 结点是 tail 结点的 label。^[10] 通过这种方式，可以将复杂的关系拆分成若干串简单关系链组成的网络。

对于已有的实体结点，需要为结点之间添加关系，以反映它们在现实世界中的联系。这一步骤既可以在实体生成时执行一个多次迭代的算法计算节点之间的关系，也可以在所有实体生成完毕后按照某一索引（在此情况下，是 cve_id）对结点进行遍历生成关系。

3.2.3.3 关系融合

关系融合模块作为适配使用深度学习模型进行命名实体识别结点生成与关系抽取方案的模块，当前仅在系统数据加工管线中留有接口，以备未来扩展需要。当前系统的结构化数据实体与关系生成已经实现实体结点和关系的消除二义性。

3.2.4 知识图谱持久化存储

经前期调研，cve.mitre.org 现有 cve 条目约 16 万条，涉及漏洞条目、资产、漏洞利用等数据为百万数量级，且这些数据每日不断更新。因此，本系统需要实现一个高性能的数据持久化子系统，用于存取更新图谱数据。

3.2.5 可视化展示知识图谱

作为一个可视化系统，面向用户的最主要需求是以文本、统计表、统计图、交互式可视化图等形式呈现的知识图谱信息。可视化系统需要实现：

- 知识图谱的相关信息展示，如漏洞数量、资产数量、利用代码数量；
- 各种种类漏洞百分比统计；
- 各种受影响资产类型百分比统计；

- 利用代码执行类型百分比统计；
 - 按时间排序的最近漏洞信息、趋势统计；
- 等等。

3.2.6 日志功能

缺乏日志功能将无法跟踪系统运行轨迹，给运维带来困难。记录清晰定位准确的日志系统有效提升开发效率，对于较复杂系统而言是不可或缺的需求。此外，日志功能可以提供系统运行状态有效信息，后端通过获取日志中的信息返回给前端，可以使用户获知当前系统运行状态。

3.3 非功能性需求分析

3.3.1 性能需求

本系统应具备处理大量图数据的能力，并且具备良好的架构以灵活应对不断增长的数据量。这要求本系统能在可以接受的时间内完整建立知识图谱，并且能不断采集信息以更新自身知识图谱数据，因此数据采集及处理周期不应大于源数据更新周期。

3.3.2 兼容性需求

好的系统具备优秀的可移植性。本系统期望达到良好的可移植性，因此应尽可能采用平台无关的语言及框架如 Python、Java 等。可以使用 Advanced Python Scheduler 来避免使用 crontab 之类平台特定命令工具。且本系统使用网页前端作数据展示功能，可视化功能应保证在多种现代浏览器上正常运行。

3.4 用户视角的需求分析

当前系统下，用户均为普通用户，由于本漏洞知识图谱数据来自自动采集的权威数据源，手动修改数据意义不显著。未来可向系统加入管理员角色，主要特权为可以通过前端界面，控制后端服务子系统的 APScheduler 调度服务运行状态，从而实现手动控制数据采集子系统、知识图谱构建子系统的运行或停止。

- 查看概览视图
 - 查看图谱统计数据
 - 查看图谱统计图表
 - 利用代码执行类型百分比统计
 - 按时间排序的最近漏洞信息、趋势统计

- 查看可视化视图
 - 拖拽浏览图谱
 - 点击结点或关系，弹出面板查看详情
 - 搜索实体并查看知识图谱
 - 搜索漏洞
 - 搜索资产
 - 搜索代码利用
 - 搜索关系
 - 查看关于视图：获取系统说明、使用指南信息

根据上述需求，图 3-1 即为本系统用户视角需求分析的用例图。

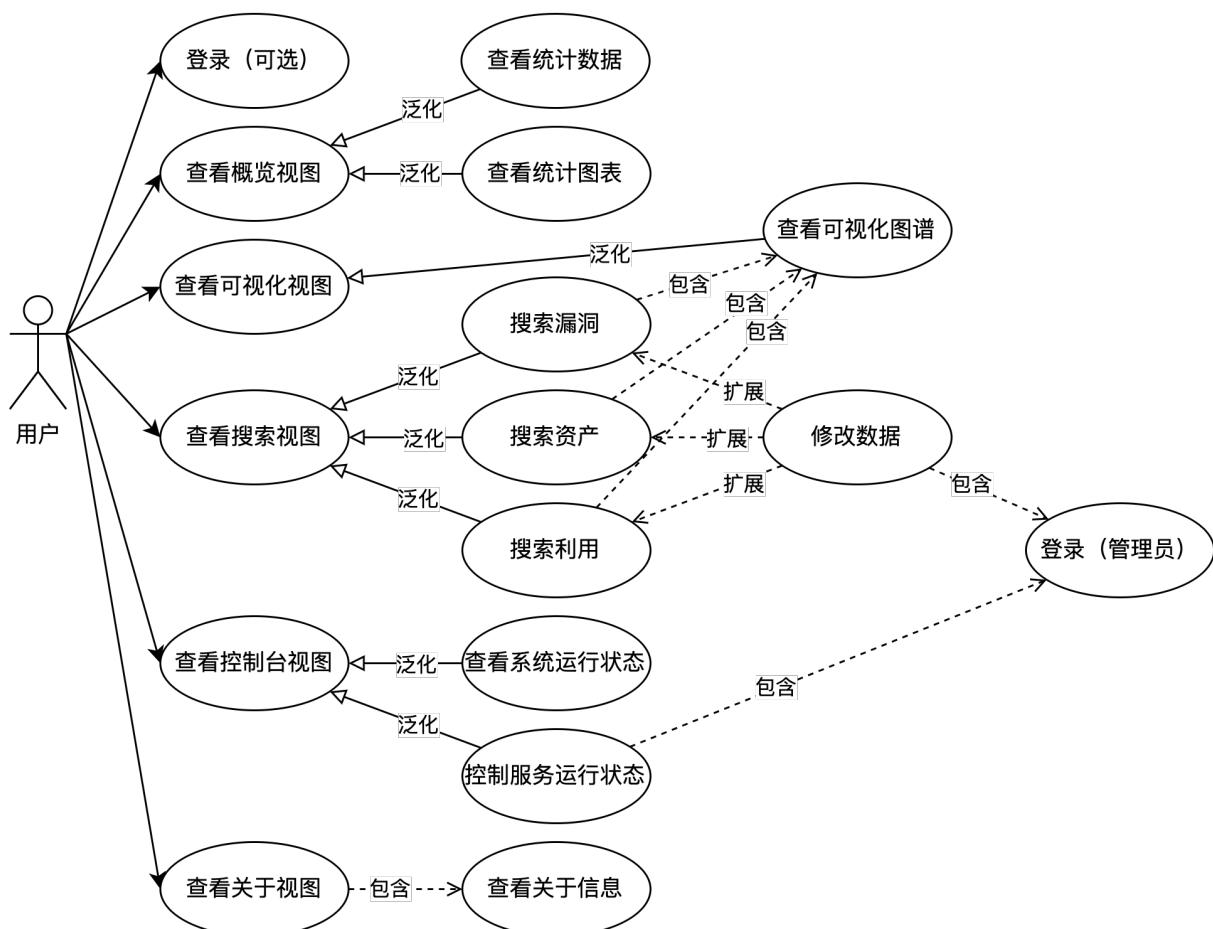


图 3-1 系统用例图

第四章 系统设计

4.1 系统概要设计

4.1.1 系统总体设计

系统数据流图如图 4-1 所示。

系统输入为来自多个数据源¹的异构信息安全漏洞数据，分别经过数据采集子系统、知识图谱构建子系统、后端服务子系统、前端子系统的加工，最终流出系统，呈现给用户。其中：

1. 数据采集子系统的加工细化为：

- (a) 预处理：对获取到的 cve 目录、cpe 目录等文件，使用 Pandas 进行筛选、过滤、重新格式化、存储至 MongoDB 数据库，得到爬虫模块所需的索引文件。
- (b) 爬虫：使用 Scrapy 进行并发爬虫，从前述多个数据源获取数据。
- (c) 数据解析：对爬虫得到的网络数据，编写 Scrapy 数据管线（Item Pipeline）进行解析与格式化，得到与互联网信息安全漏洞相关的结构化数据与自然语言数据，作为原始数据。

2. 知识图谱构建子系统的加工细化为：

- (a) 实体生成：通过基于规则的或基于深度学习模型的方法，对前述爬虫得到的原始数据进行命名实体识别，提取所需实体，及其相关属性。从 nvd.nist.gov 获取的漏洞数据区块提取 cve 条目的 cve id 与危险等级等属性、从资产区块提取受影响资产的 cpe23uri 匹配属性、从 exploit-db.com 爬取的页面中提取漏洞利用代码、代码类型、影响平台等属性。
- (b) 关系生成：为实体结点之间添加关系，从若干孤立结点形成网状结构。加工步骤利用信息安全漏洞本体定义的实体间的逻辑关系进行关系生成。
- (c) 关系融合：消除具有二义性的实体结点与关系边，仅需留有接口。

3. 后端服务子系统的加工细化为：

- (a)：查询数据库，将数据转换成 Python 数据结构。
- (b)：将数据重新组织、添加前端绘图库 ECharts 所需的属性，将查询结果转换为与绘图库兼容的数据结构。
- (c)：将绘图数据序列化成 JSON 字符串，通过 HTTPS 协议发送至前端。

4. 前端子系统的加工细化为：

¹cvedetails.com、cve.mitre.org、cpe.mitre.org、nvd.nist.gov、exploit-db.com 等。

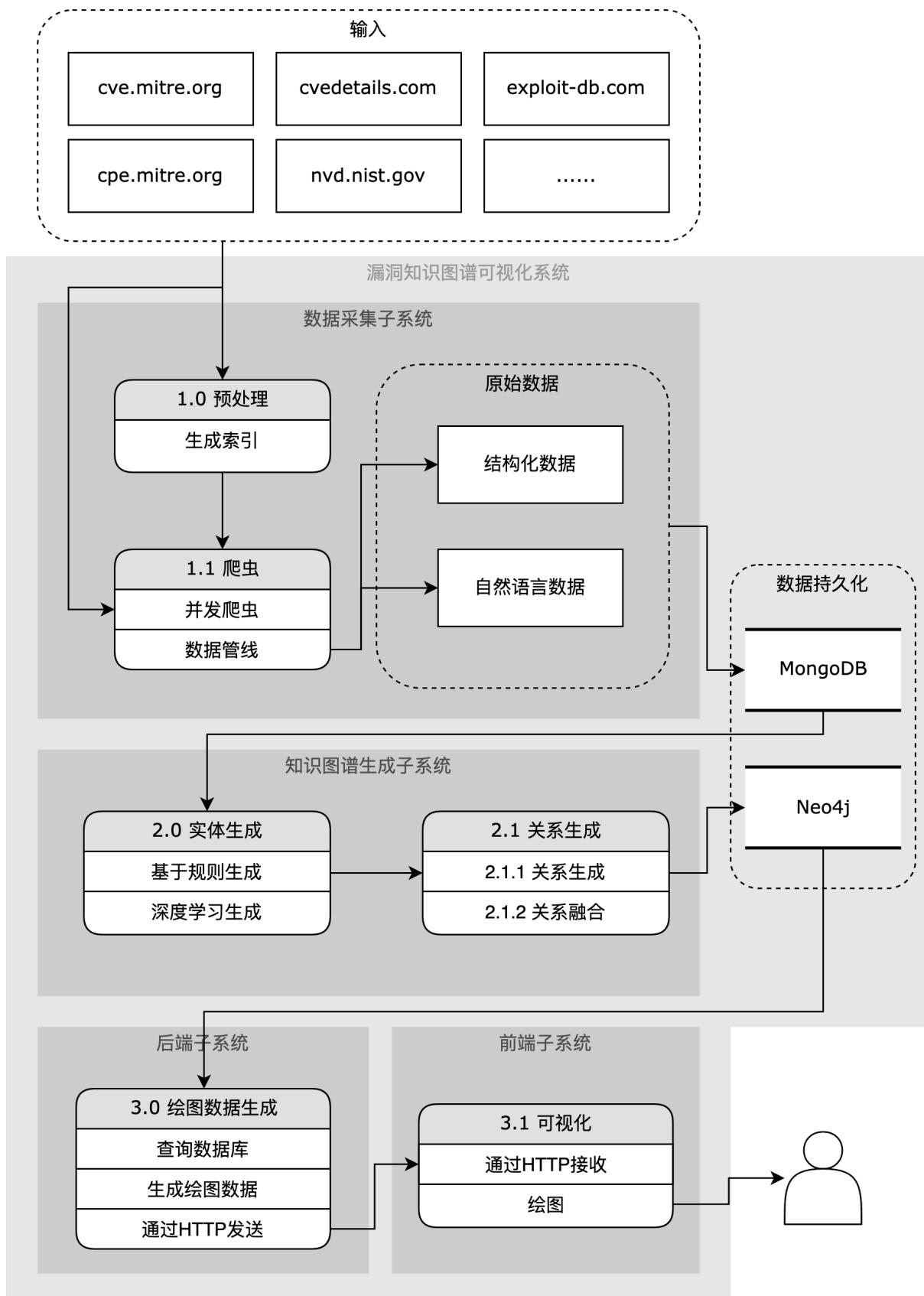


图 4-1 系统数据流图

(a) : 通过 HTTPS 协议从后端接收绘图数据，反序列化成 JSON 对象。

(b) : 调用绘图库使用接收到的绘图数据，在 canvas 上绘制图形。

4.1.2 系统层次结构设计

系统层次结构图如图 4-2 所示。系统可以分为三层：表示层、逻辑层、持久化层。表示层包含前端用户界面；持久化层包含 Neo4j 与 MongoDB 两个数据库；逻辑层包含后端服务、知识图谱构建服务、数据采集服务。借鉴微服务思想，逻辑层之间各服务通过持久化层传递数据，减少服务间耦合度。

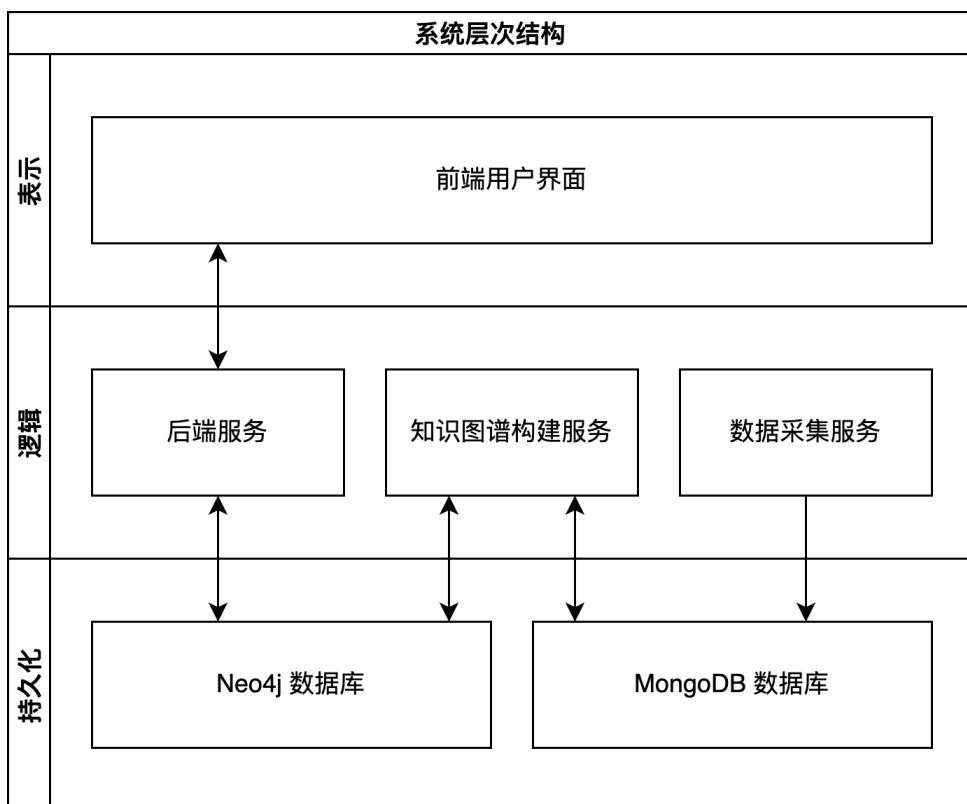


图 4-2 系统层次结构设计图

4.1.2.1 表示层

表示层是用户与本系统交互的介面。用户与前端用户界面交互，从而获得信息。表示层在前端具备简单的数据处理能力，用于对从后端接收到的数据进行处理与转换。例如，将后端提供的适用于扇形图的数据转换为适用于柱状图的相同数据。这种设计可以充分利用前端设备的计算能力，减轻后端服务器负载，并且有助于减少 API 数量，降低复杂度。

4.1.2.2 逻辑层

逻辑层实现系统核心功能，响应表示层传来的用户指令并且回传所需数据，通过与持久化层交互使数据从输入源不断被加工，最终输出给用户。同时，负责整个系统调度工作。

数据采集服务：通过生成索引、爬虫、数据清洗，向持久化层输入数据。数据采集服务在持久化层存储的数据包括用于指引爬虫生成新链接的索引、爬虫从各数据源爬取得到的 HTML 文档、JSON 等原始数据用于存档、经过数据管线处理得到的可用于下一步构建知识图谱的 JSON 格式化数据。数据采集服务与持久化层 MongoDB 进行交互，是系统数据输入界面。

知识图谱构建服务：从 MongoDB 数据库读取格式化数据，经过实体生成、关系生成等步骤，构建互联网信息安全漏洞的知识图谱。知识图谱构建服务与持久化层 MongoDB 和 Neo4j 进行交互，实现原始数据的加工，将图谱相关结点、关系、属性数据存储至 Neo4j 数据库中。

后端服务：与表示层通过 RESTful API 交互，获取存取数据、服务调度等用户指令。与持久化层 Neo4j 数据库进行加护，读取知识图谱相关结点、关系、属性数据，将其通过 HTTPS 协议传递给表示层。此外，后端服务还具有调度功能，通过 APScheduler 调度逻辑层其他服务的运行，从而保障整个系统正常运转。

4.1.2.3 持久化层

持久化层主要包含 MongoDB 数据库与 Neo4j 数据库。

MongoDB 数据库在系统中具备两种功能：存档数据采集服务获取的数据以供知识图谱构建服务进行加工；利用 MongoDB 内存技术特性缓存后端服务向 Neo4j 数据库进行大规模查询得到的结果，前端服务频繁查询直接读取缓存数据，减轻 Neo4j 数据库负载，缓存定时更新。

Neo4j 数据库在系统中主要发挥互联网安全信息漏洞知识图谱的承载功能。

4.2 系统功能模块设计

图 4-3 展示了本系统各个子系统的功能模块设计。系统分为五个子系统：数据采集子系统、知识图谱构建子系统、后端服务子系统、前端子系统。

4.2.1 数据采集子系统

主要实现 Web 爬虫服务，包括请求生成器、异构数据解析器、异构数据管线三个模块。其中异构数据解析器包含包含 HTML 解析器、JSON 解析器、csv 解析器、gzip 解析器等。异构数据管线包含与异构数据解析器对应的各类数据管线。

请求生成器根据 csv 索引生成爬虫请求链接，交由 Scrapy Spider 进行并发爬虫。

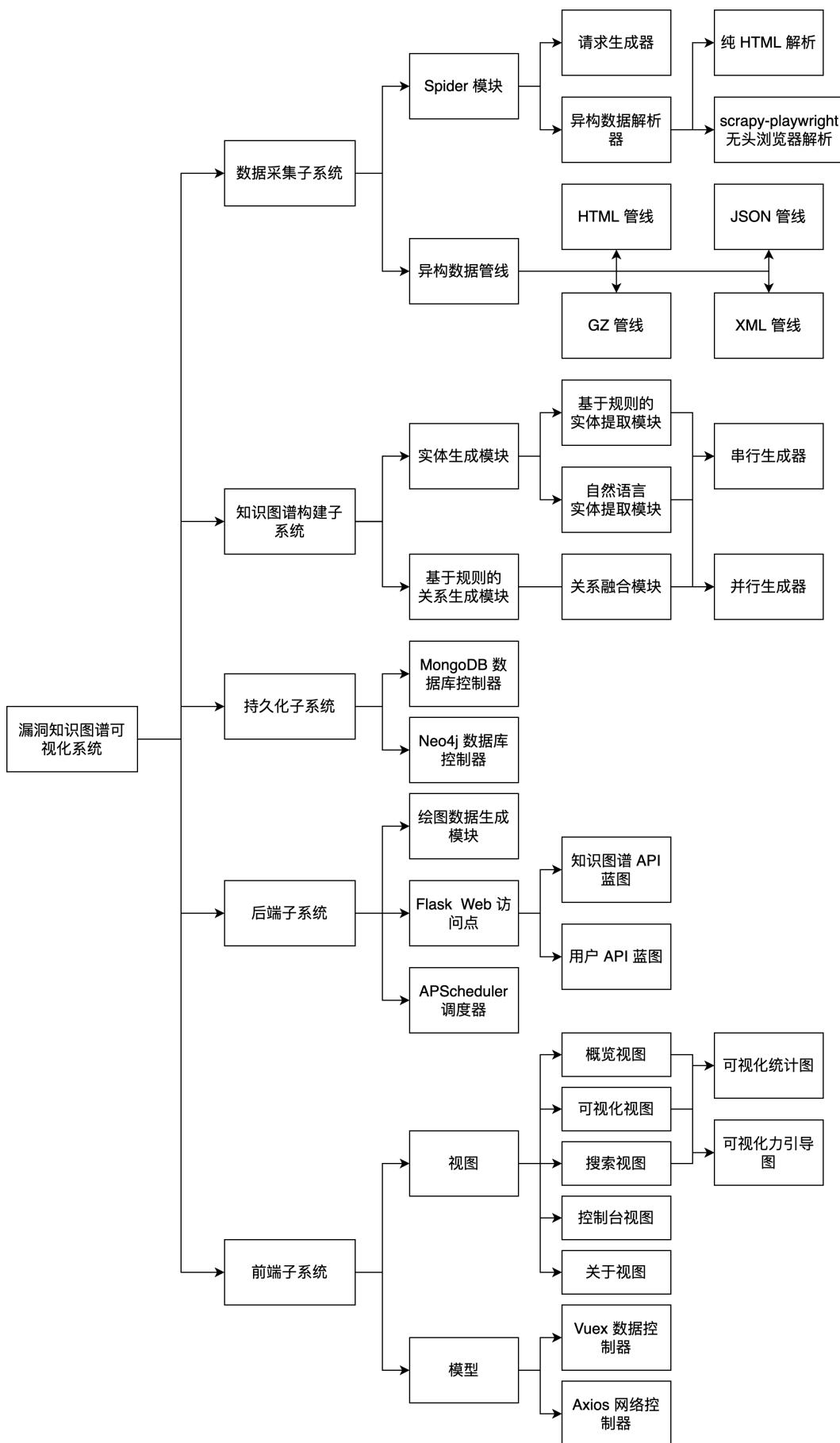


图 4-3 系统功能模块设计图

异构数据解析器负责使用对应解析器解析 Spider 爬虫返回的网络数据，传递给异构数据管线。此外，异构数据解析器中的 HTML 解析器应用在一些网站的爬虫任务时，还会解析页面上包含“下一页”等元素的 <a> 标签，通过 yield 将其中 href URL 交给请求生成器，生成新的爬虫请求。

异构数据管线负责接收其对应的数据解析器传来的数据，提取其中需要的信息进行格式转换等处理，填充固定格式的 Python Dictionary，再调用持久化子系统中的数据库控制器对象，将请求原始数据与 Dictionary 分别存储至不同的 MongoDB 数据库集合(Collection) 中持久化存储，形成原始数据，完成数据采集工作。

4.2.2 知识图谱构建子系统

主要实现知识图谱构建服务，包含实体生成模块、基于规则的关系生成模块、关系融合模块。

4.2.2.1 本体定义

采用自顶向下方法构建信息安全漏洞领域知识图谱，首先需定义信息安全漏洞本体。调研归纳与信息安全漏洞相关概念及术语如下：

- **Vulnerability**: 漏洞或脆弱性，指计算机信息系统在需求、设计、实现、配置等方面缺陷，漏洞被利用会导致计算机信息系统在数据的保密性、完整性、可用性、访问控制等方面面临威胁。^[14]
- **Asset**: 资产，信息技术系统、软甲和软件包等产品统称为资产。CPE 项目列举官方资产列表。
- **Exploit**: 对漏洞的利用代码，通常是程序片段，提供对漏洞利用的概念验证。
- **CVE**: 公共漏洞和暴露，提供一个命名字典，为广泛认同的信息安全漏洞或已经暴露的弱点给出一个公共名称，收集各种信息安全弱点及漏洞并给予编号以便于公众查阅。^[15]
- **CPE**: 通用平台枚举，是信息技术系统、软件和软件包的结构化命名方案。基于统一资源标识符 (URI) 的通用语法，CPE 提供一个官方 CPE 产品字典，枚举商定的官方 CPE 名称列表。^[16]
- **CWE**: 通用弱点枚举，是一个社区开发的具有安全影响的常见软硬件弱点类型列表。标准化分类列举描述软硬件实现、代码、设计或架构中的缺陷、故障、漏洞或其他错误。^[17]
- **CVSS**: 通用漏洞评分系统，提供一种评分方法来捕捉漏洞的主要特征并反映其严重性。评分可以转换为定性的表示（高、中、低等）。^[18]

根据以上概念可推导关系：Vulnerability 漏洞具备 CVE 作为唯一标识、具备 CVSS 属性评价漏洞严重性、漏洞产生的原因对应一个或多个 CWE；Asset 资产具备 CPE 作为唯一标识，资产具有漏洞、漏洞影响资产；Exploit 利用代码无通用唯一标识、一个利用代码有时可对应多个漏洞，利用代码利用漏洞、攻击资产。

图 4-4 所示为信息安全漏洞本体，给出了实体定义及关系定义。分为漏洞 Vulnerability、资产 Asset、利用 Exploit 三类实体，图中给出每类实体作为索引的属性域，该属性具有唯一性。三类实体之间利用上述联系，定义关系边连接。详见下述章节 4.2.2.2 实体生成与 4.2.2.3 关系生成。

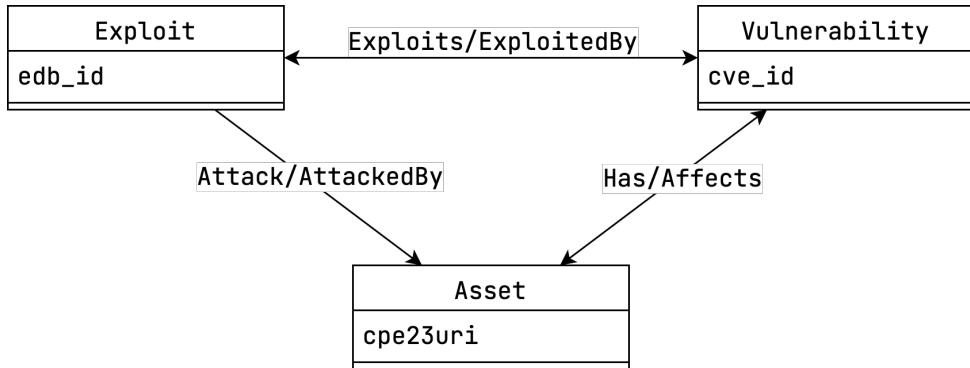


图 4-4 信息安全漏洞本体定义

4.2.2.2 实体生成

目前包含基于规则的实体生成子模块，未来可引入基于深度学习模型的自然语言实体生成子模块。基于规则的实体生成模块应用自顶向下人工构建的互联网信息安全漏洞本体规则，从结构化数据抽取实体及其属性。前述数据采集子系统获得的结构化数据，经处理后包含根据规则定义的属性域，如下：

- 漏洞：cve_id，格式为 CVE-[0-9]+-[0-9]+
- 资产：cpe23uri，格式如代码 4-1 所示。
- 利用代码：edb_id，为 exploit-db.com 的数字序号。因利用代码无通用标识符，需使用 UUIDv4 方法生成本系统内使用的标识符。

代码 4-1 CPE 2.3 格式正则表达式

```

1 cpe:2\.3:[aho\*\-\-](:((\?\*|\*\?)([a-zA-Z0-9\-\_])|(\\\\[\\\\*\?\!\"#\$\%&'\\()\\
\+,/\:;<\=\>@\[\\]\^\{\\\}\~))+(\?\*|\*\?))|[\*\-\-])\{5}(:(([a-zA-Z]\{2,3\})(-([a-
-zA-Z]\{2\}|[0-9]\{3\}))?)|[\*\-\-]):(((\?\*|\*\?)([a-zA-Z0-9\-\_\\])|(\\\\[\\\\*\?\!\"#\$\%&'\\()\\
\+,/\:;<\=\>@\[\\]\^\{\\\}\~))+(\?\*|\*\?))|[\*\-\-])\{4}

```

生成方法根据属性域，区分实体类型，调用相应方法从文档提取属性，调用持久化子系统的数据库控制对象将结点存入 Neo4j 数据库。来自不同数据源的实体存储在

MongoDB 不同集合中, 如来自 NVD JSON API 的数据存储于 nvd_json 集合中。针对每个数据库集合, 在 Python 中实现一个实体生成方法。如 nvd_json 集合对应方法为遍历数据库存储的文档, 对于每个文档, 提取其 cve_id, cwe_id, desc, impact{}, references[], publish_date, last_update_date, version, timestamp 作为 Vulnerability 实体的属性, 提取其 cpe23uri 作为 Asset 实体的属性。对于仅包含资产信息的 cpe 集合, 将每个文档生成一个 Asset 实体, 并与已经存在的 cpe23uri 相同的实体融合。

4.2.2.3 关系生成

基于规则的关系生成模块根据信息安全漏洞本体所表示关系, 在 Neo4j 数据库中匹配对应结点, 在两个结点间生成对应的边, 并为其添加属性:

- 资产 -[具有]-> 漏洞, 漏洞 -[影响]-> 资产
- 利用代码 -[攻击]-> 资产, 资产 -[被攻击]-> 利用代码
- 漏洞 -[被利用]-> 漏洞代码, 利用代码 -[利用]-> 漏洞

因知识图谱中预计将有一百万个结点, 使用最细粒度将每个漏洞结点与每个资产结点连接将产生超过一亿条边, 这对于本项目的性能要求而言是不现实的。因此, 关系生成模块会使用前述步骤生成的资产家族结点, 在漏洞结点与漏洞结点所包含的受影响资产列表中的每条正则匹配项所匹配到的资产对应的资产家族之间添加关系。为了维持该关系指向资产的语义准确, 将正则匹配到的资产列表作为属性添加到该关系边之上:

- 资产家族 -[具有]-> 漏洞, 漏洞 -[影响]-> 资产家族
- 资产 -[是子级]-> 资产家族, 资产家族 -[是父级]-> 资产
- 利用代码 -[攻击]-> 资产家族, 资产家族 -[被攻击]-> 利用代码
- 漏洞 -[被利用]-> 漏洞代码, 利用代码 -[利用]-> 漏洞

如此可将构建图谱所需关系数量从亿级降低到百万级, 在不损失语义的情况下有效提高查询效率, 减小数据库体积。

4.2.2.4 关系融合

关系融合模块作为适配使用深度学习模型进行命名实体识别结点生成与关系抽取方案的模块, 当前仅在系统数据加工管线中留有接口, 以备未来扩展需要。当前系统的结构化数据实体与关系生成已经实现实体结点和关系的消除二义性, 因此无需实现关系融合模块。

4.2.3 后端服务子系统

主要负责提供后端 Web API、对知识图谱的数据进行转换处理、调度各服务运行。包括绘图数据生成模块、Web API 访问点、APScheduler 调度模块。

Web API 访问点由 Flask 支持，使用 Flask 的 Blueprint 功能对注册在 RESTful API 根路径下的第一级 API 进行分组管理，实现模块化设计。在某个 Route 收到请求时，根据用户请求内容，将请求参数作为 CQL 参数，调用持久化层数据库控制对象对数据库进行操作，增删查改知识图谱中的结点和关系。若是可视化相关的 Route，则会对查询到的数据调用绘图数据生成模块进行转换。

绘图数据生成模块将存储在 Neo4j 数据库中的知识图谱数据转化为 ECharts 绘图库可以使用的数据格式。

APScheduler 调度模块：通过调用数据采集子系统和知识图谱构建子系统提供的 Runner 方法，实现爬虫、更新图谱的定时运行。

4.2.4 前端子系统

前端子系统负责管理用户运行时相关数据、通过 UI 向用户展示信息。使用 Vue.js 的 MVVM 架构，主要包含视图与模型两个组成部分。

视图包含概览、可视化、搜索、关于四部分。

- 概览视图为用户提供知识图谱信息概览，包含列表形式的数据展现，以及统计图表形式的数据可视化。
- 可视化视图是一个全屏视图，通过该视图用户可以一览知识图谱全貌。
- 搜索视图支持用户在图谱中搜索感兴趣的内容，并将搜索结果以力引导图可视化的形式展示。
- 关于视图展示本系统介绍信息及用户指南。

模型主要包含 Vuex 数据控制器、Axios 网络控制器两部分。Vuex 数据控制器统一管理前端子系统的数据与状态，是前端子系统的“Single Source of Truth”，为视图的响应式显示提供数据源，还包含在前端对数据进行简单处理功能。Axios 网络控制器封装一个 httpClinet 对象，以及若干与后端通信的方法。Vuex 数据控制器可以调用 Axios 网络控制器，从后端获取数据，更新 Vuex Store 中的状态。

4.2.5 数据库设计

4.2.5.1 MongoDB

图 4-5 所示为 MongoDB 数据模式图。由于 MongoDB 支持动态 DDL，因此数据库在本项目开发后期共有 7 个集合。

- cpe: 存储从 XML 解析的 cpe 资产数据。
- edb_html: 存储从 exploit-db.com 爬虫获得的 HTML 原始数据。
- edb_json: 存储从 exploit-db.com HTML 数据处理得到的 JSON 数据。
- cve_html: 存储从 cvedetails.com 爬虫获得的 HTML 原始数据。
- cve_json: 存储从 cvedetails.com HTML 数据处理得到的 JSON 数据。
- nvd_json_src: 存储从 nvd.nist.gov 数据接口获得的 JSON 原始数据。
- nvd_json: 存储从 nvd.nist.gov JSON 数据处理得到的 JSON 数据。

4.2.5.2 Neo4j

图 4-6 所示为 Neo4j 数据模式图，同时也是知识图谱实体与关系模式图。

Neo4j 数据模式图中，**Vulnerability**, **Asset Family**, **Asset**, **Exploit** 为主要实体结点。**Vulnerability**, **Asset**, **Exploit** 为章节 4.2.2.1 本体定义所述信息安全漏洞本体的实体；**Asset Family** 结点是属于同一类型、同一制造商、同一产品的资产的集合，是一个抽象的概念实体。

Vulnerability 漏洞实体的 **cve_id** 属性为索引，**version** 标识数据格式版本，**timestamp** 记录数据更新时间，**vuln**, **assets[]** 为复合属性。**vuln** 复合属性记录漏洞相关信息，**cve_id** 与索引属性相同，**cwe_id** 为漏洞的 CWE 类型，**desc** 为漏洞的自然语言描述，**impact {}** 为元组复合属性，记录漏洞的 CVSS 信息，**references** 为列表复合属性，记录漏洞相关参考资料。**assets[]** 为列表复合属性，包含一系列用于匹配资产的正则表达式。

Asset 为资产实体，索引属性 **cpe23uri** 为资产的 CPE 标识 URI；**field{}** 元组复合属性对应 **cpe23uri** 的各部分语义；**references[]** 列表复合属性记录一系列关于此资产的参考资料链接及资料类型标签；**title** 为资产的自然语言名称，**timestamp** 记录数据更新时间。

Asset Family 为资产家族实体，为了解决漏洞影响资产过多而导致关系数量庞大的问题而提出。资产家族的索引属性 **cpe23uri** 同为 CPE 标识 URI，与资产 **Asset** 的 **cpe23uri** 属性区别在于，资产家族的该属性仅有 **part**, **vendor**, **product** 三个字段。

Exploit 为利用代码实体，**edb_id** 为 exploit-db.com 的数字序号；**title** 为利用代码描述性自然语言标题；**author** 为代码作者；**type** 为代码执行类型，分类如网络、邻接、本地、物理等；**platform** 为利用代码或漏洞的运行平台；**date** 为该段代码的更新日期；**code** 为代码文本；**cve_ids[]** 是列表复合属性，为 **cve_id** 组成的列表，记录与此利用相关的漏洞编号。

全部实体均有 **eid** 索引属性，**eid** 即 entity id，使用 UUIDv4 方法生成，保证唯一性。实体之间的关系为：

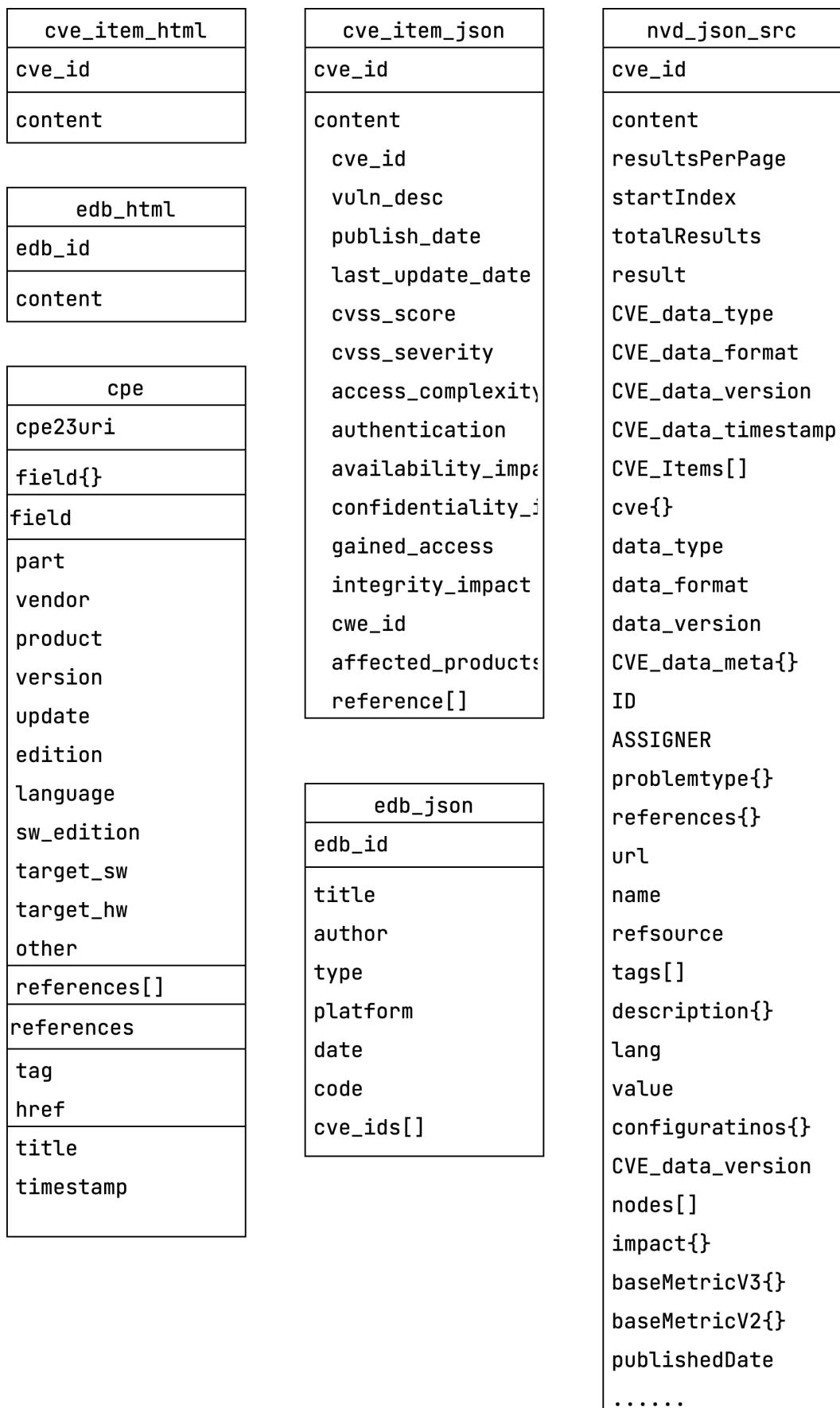


图 4-5 MongoDB 数据模式图

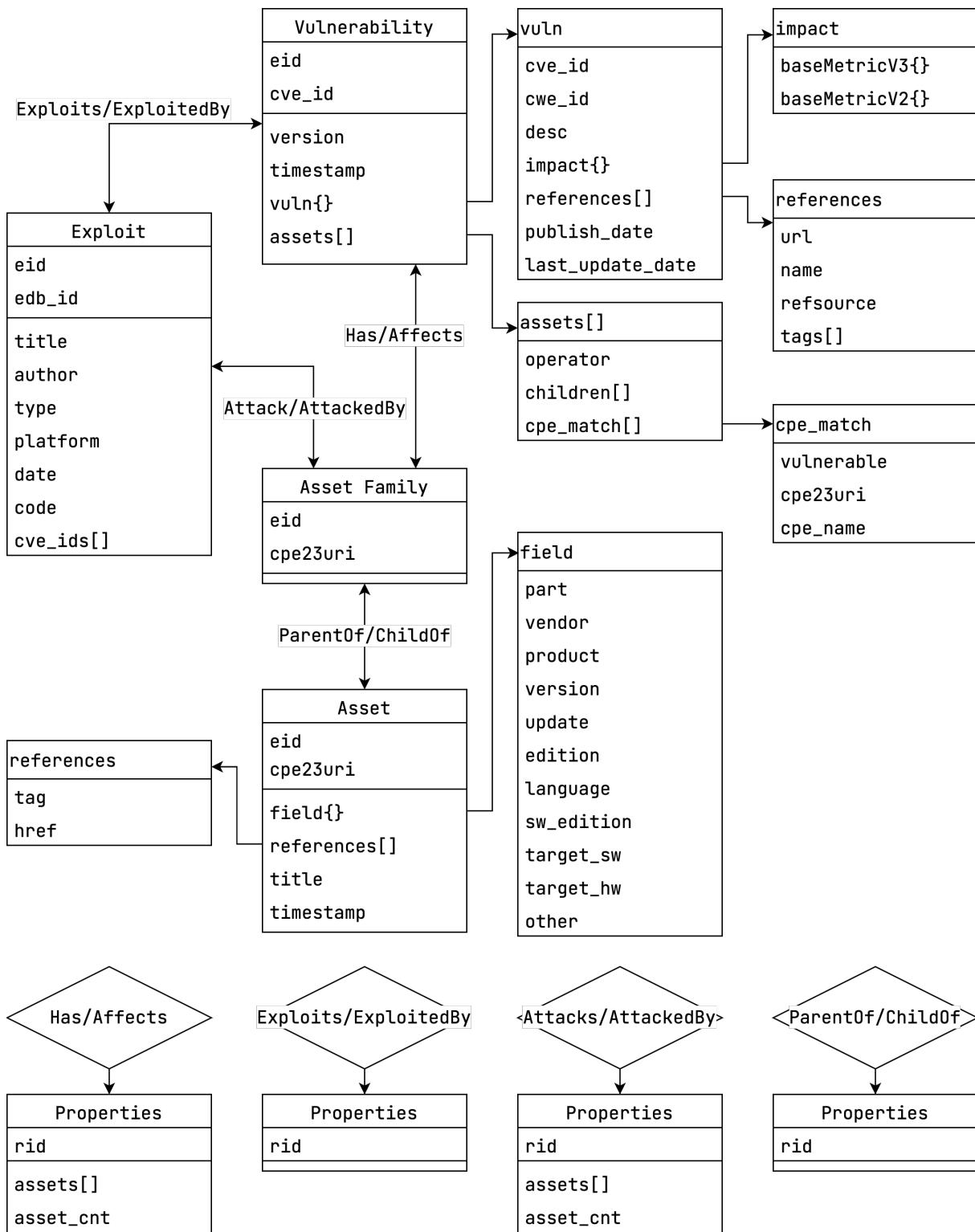


图 4-6 Neo4j 数据模式图

```

1 Vulnerability-[Affects]->AssetFamily, AssetFamily-[Has]->Vulnerability
2 Exploit-[Exploits]->Vulnerability, Vulnerability-[ExploitedBy]->Exploit
3 AssetFamily-[AttackedBy]->Exploit, Exploit-[Attacks]->AssetFamily
4 Asset-[ChildOf]->AssetFamily, AssetFamily-[ParentOf]->Asset

```

所有关系都具有 `rid` 索引属性即 relationship id 作为唯一标识。`Has/Affects` 关系具有的属性为 `assets[]` 列表复合属性记录该条边代表的受影响资产正则匹配表达式, `asset_cnt` 属性记录满足正则匹配规则的全部资产数量。`Attacks/AttackedBy` 关系的属性与 `Has/Affects` 相同。

4.2.6 系统接口设计

类别: 前端路由

- `/welcome`: 欢迎页面
- `/overview`: 概览页面
- `/vis`: 可视化页面
- `/search`: 搜索页面
- `/about`: 关于页面

类别: 后端接口

- `/api/graph/`
 - 方法: GET
 - 参数: 无
 - 描述: 获取知识图谱统计信息, 包含漏洞数量、各类别资产数量、利用数量等
 - 返回值: 含 `vul`, `asset`, `exploit` 三个键的 JSON, 每个键对应的值为实体相关数据 JSON
- `/api/graph/<limit>`
 - 方法: GET
 - 参数: `limit`, 整数字符串
 - 描述: 获取与 `<limit>` 个漏洞结点相关联的结点或关系组成的子图谱
 - 返回值: 含 `categories`, `nodes`, `links` 三个键的 JSON, 每个键对应的值为一个列表, 列表中的每个元素代表一个类别、一个结点或一个关系

- `/api/search/<keyword>`
 - 方法: GET
 - 参数: `keyword`, 搜索关键字, 字符串
 - 描述: 搜索关键字相关的漏洞、资产、利用结点或关系
 - 返回值: 同 GET `/api/graph/<limit>`

4.3 系统详细设计

4.3.1 开发环境配置

- 操作系统: Ubuntu 20.04, Windows 11 Pro
- 数据采集子系统: Python 3.8, Scrapy 2.6
- 知识图谱构建子系统: Python 3.8, Ray 1.11.1, uuid 1.3
- 持久化子系统: Python 3.8, Py2neo 2021.2.3, Pymongo 3.12, Neo4j 4.4
- 后端服务子系统: Python 3.8, Flask 2.0, flask-cors 3.0
- 前端子系统: JavaScript ES6, Vue 2.6, Vuetify 2.6, ECharts 5.3.2, vue-axios 3.4, Vuex 3.4, Moment 2.29.3, Lodash 4.17.21

4.3.2 各核心功能模块详细设计

4.3.2.1 类图

图 4-7 所示为数据采集子系统类图。其中 `Item` 的数据模式与 MongoDB 里同名集合模式相同。

`Spider` 类继承自 `scrapy.spider` 类, 包含异构数据源站点爬虫实现, 内置请求生成器及异构数据解析器。

`Item` 类为 Scrapy 内置数据对象, 可用于定义数据模型并且在 `Spider` 与 `Pipeline` 间传递信息。

`Pipeline` 类为异构数据管线。`GzFilePipeline` 继承自 `scrapy.FilePipeline` 数据管线, 其他管线为单独实现。

图 4-8 所示为知识图谱构建子系统类图。

`init_kg()` 为模块的函数入口, 可被调度系统调用进行图谱更新。其中 `init_nodes()`, `init_rels()` 为支持 Ray 框架的并行化函数, 它们内部各自又实现了生成漏洞结点、生成资产结点、生成漏洞到资产家族关系等一系列并行化函数, 用于并发构建知识图谱。

`VulnEntity` 则实现了一个互联网安全漏洞本体类。

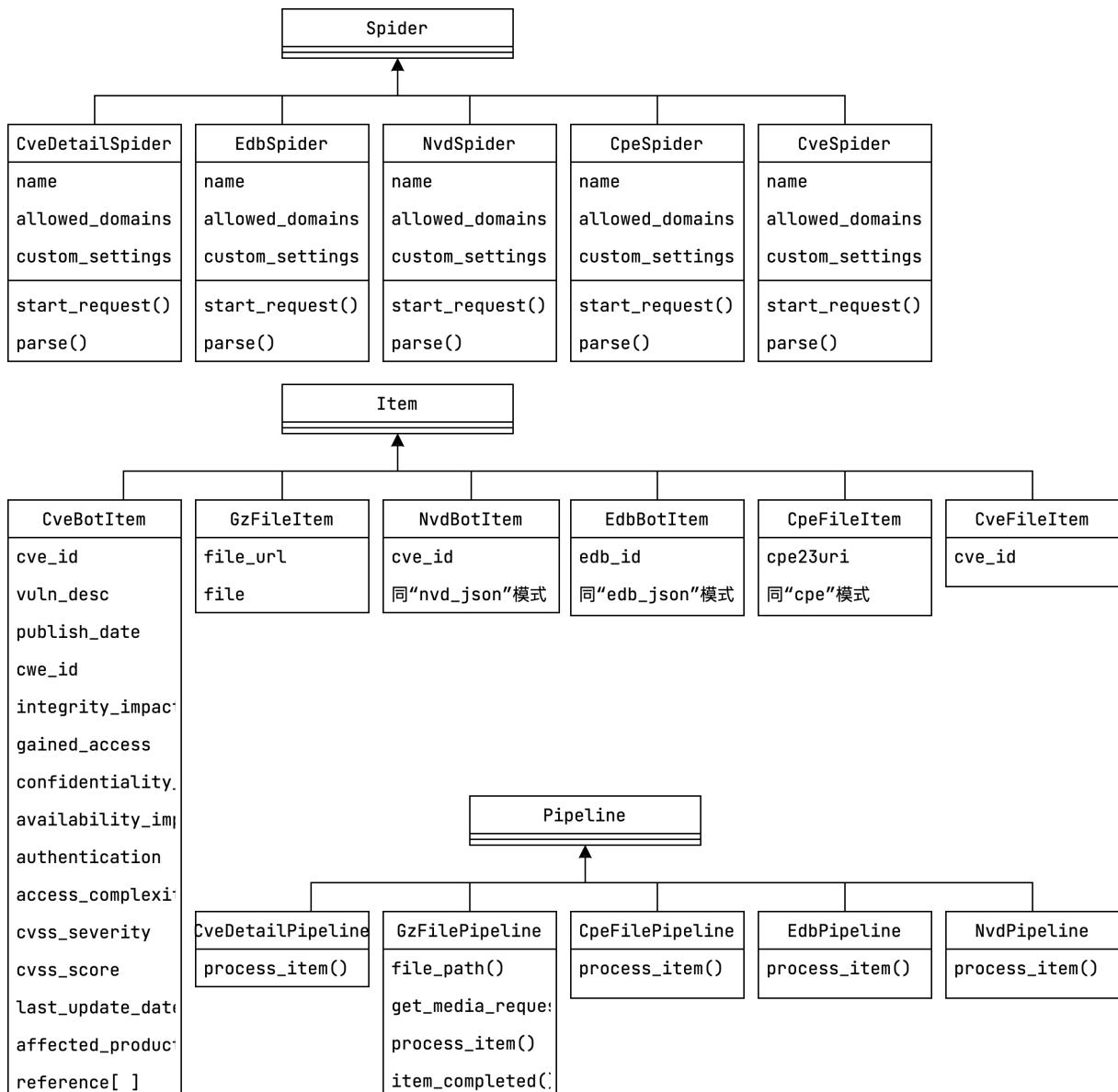


图 4-7 数据采集子系统类图

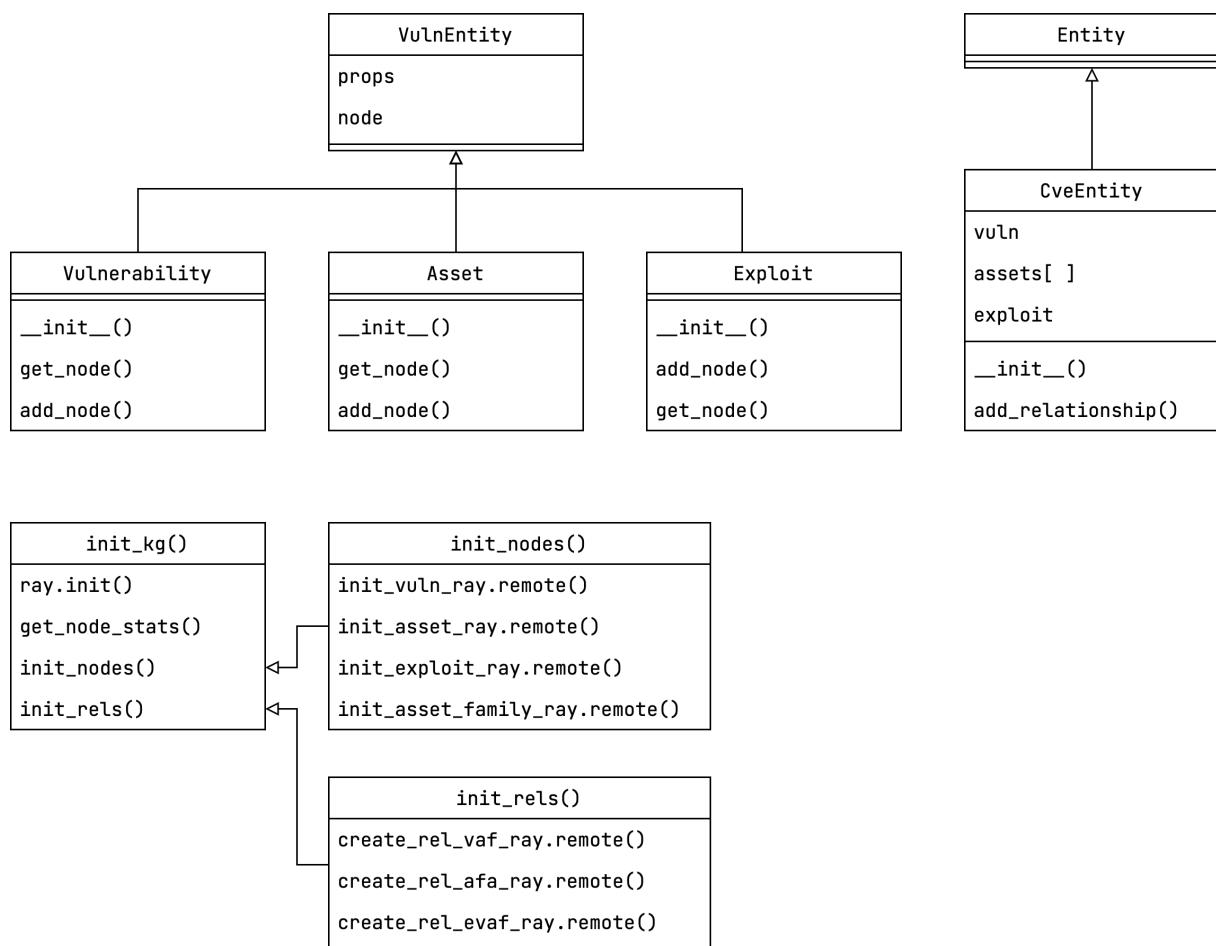


图 4-8 知识图谱构建子系统类图

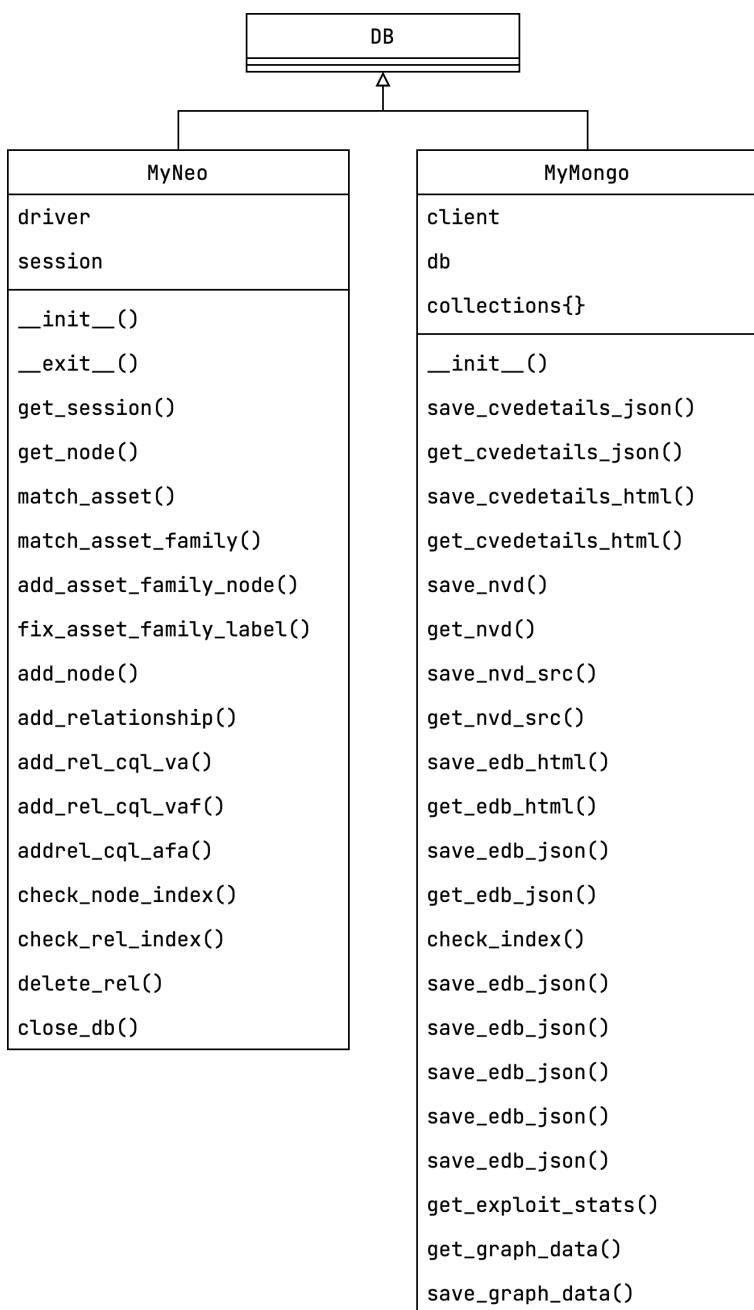


图 4-9 持久化子系统类图

图 4-9 所示为持久化子系统类图。

持久化子系统主要包含 MyMongo，MyNeo 两个数据库控制器类。在 `__init__()` 方法中实现了单例模式；使用自带连接池，实现若干用于读写数据库的方法。

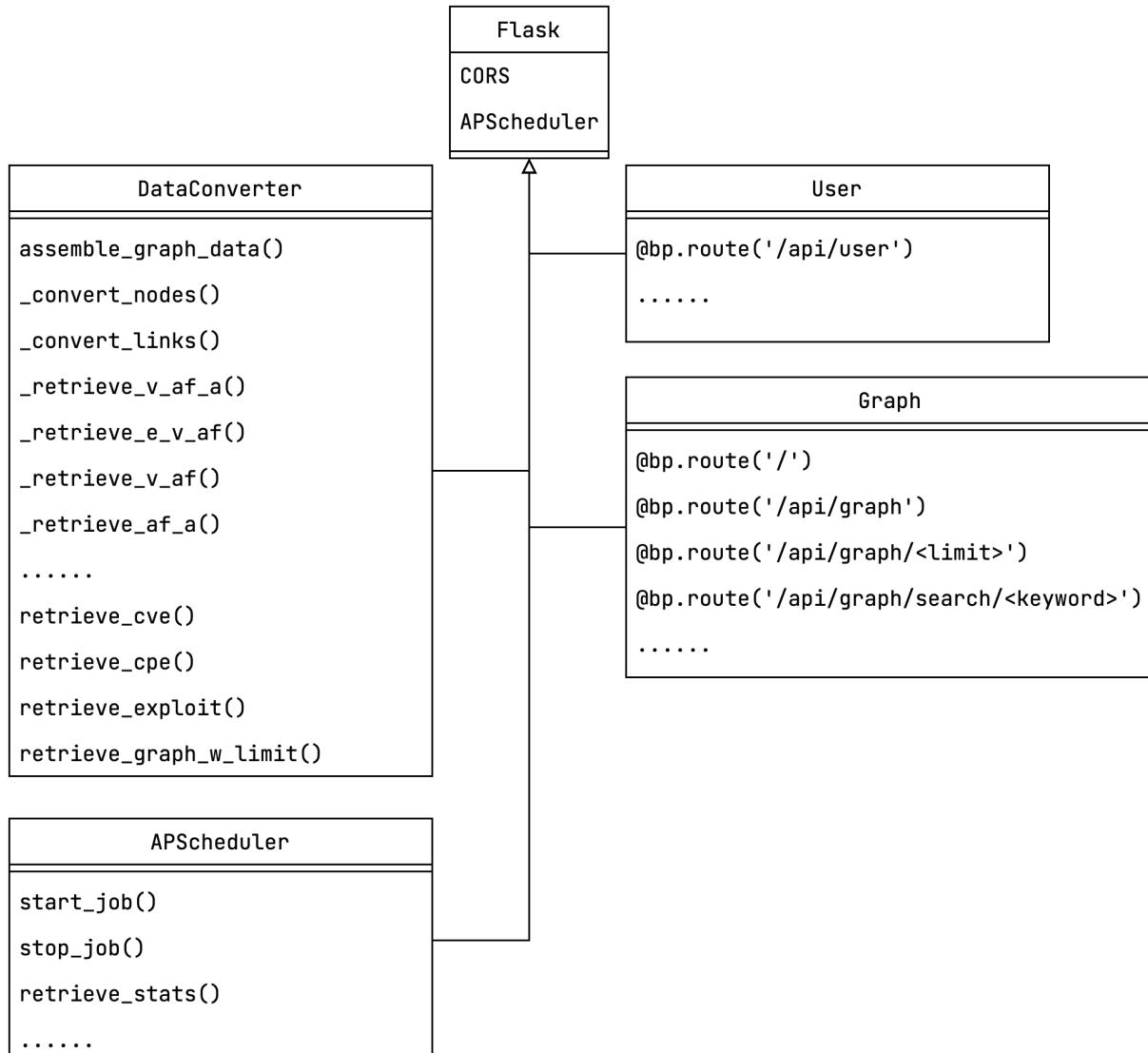


图 4-10 后端服务子系统类图

图 4-10 所示为后端服务子系统类图。

`Flask` 对象中注册了 `CORS` 中间件，包含 `User`, `Graph` 两个模块化的蓝图。`User` 蓝图注册用户相关 API, `Graph` 蓝图注册知识图谱相关 API。后端服务启动，`Flask` 类实例化时调用 `APScheduler` 模块的初始化方法，向 `Flask` 实例挂载计划任务和响应的调度器。

`DataConverter` 是绘图数据生成器模块，包含将结点、关系和若干预定义三元组从知识图谱数据结构转换为 ECharts 可用数据结构的方法。

`APScheduler` 是后端子系统的调度模块，使后端子系统能控制数据采集子系统、知识图谱构建子系统的运行或停止。

4.3.2.2 功能时序图

图 4-11 所示为通用状态下用户操作前端发起请求的时序图。

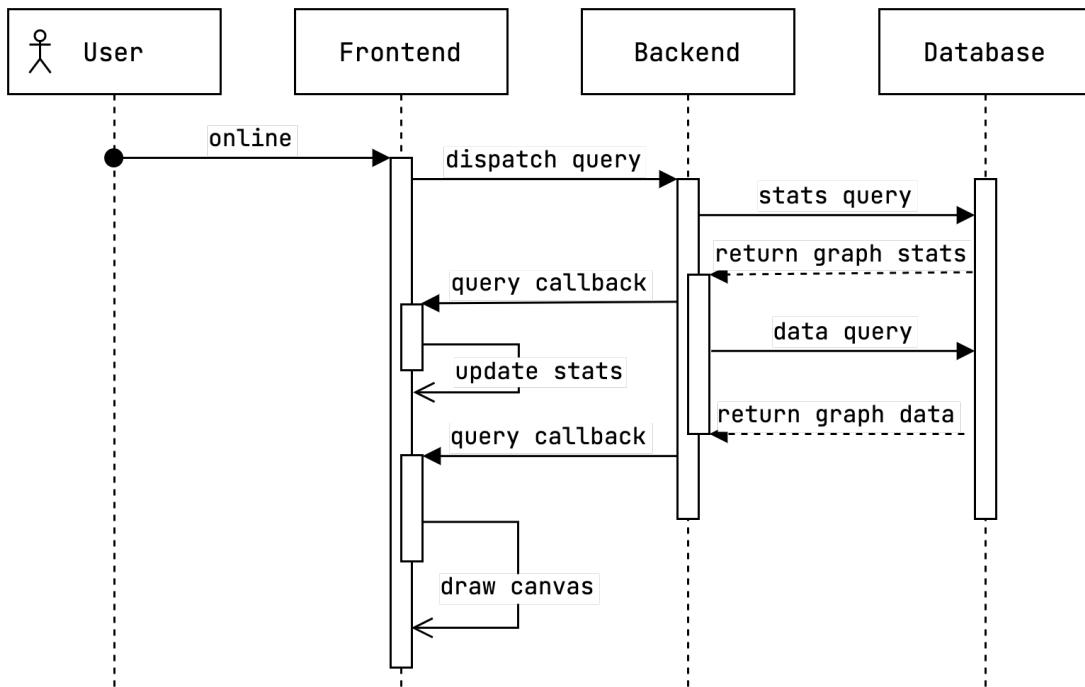


图 4-11 用户请求功能时序图

用户上线后，前端页面被挂载时调用钩子函数向后端发送初始请求。初始请求包含用户信息、系统的状态、数据更新的时间等。这些信息将会在页面顶部一条高 24px 的信息栏上常驻显示。同时，向 Vuex 注册定时器及回调函数，周期性发送心跳包更新数据。

用户进行操作（点击按钮、切换视图页面、搜索图谱等）时，会发送两个请求。一是立即触发定时器回调函数，发送系统状态更新请求。二是发送操作相关请求。系统状态更新请求回调后会更新 Vuex 中的数据，前端视图层响应式更新显示。操作相关请求回调后，更新 Vuex 中的数据，根据操作类型调用 ECharts 实例的重绘方法，重绘视图层 canvas。

由于本系统多数流程为全自动控制，且与用户操作完全异步，不作功能时序图展示。数据采集子系统与知识图谱构建子系统的工作流程可见图 5-1 数据采集子系统工作流程图与图 5-2 知识图谱构建子系统工作流程图。

第五章 系统实现

5.1 工具模块

5.1.1 根目录模块实现 `bot_root_dir.py`

本模块位于项目根目录下。`get_bot_root_dir()` 方法使用 `pathlib` 的 `Path(__file__).parent` 方法获得项目根目录的路径，使得本项目内代码全部使用平台无关的相对路径，从而能在 Linux、macOS、Windows 等多种系统平台运行，保证可移植性。

本模块内包含许多路径相关的方法，如 `get_source_data_dir()`，`get_cve_data_dir()`，`get_log_dir()` 等。这些方法接受字符串形式的路径参数，返回 `Path` 对象。通过传入参数，控制系统生成的文件存放的位置，若目录不存在，则会自动创建所需目录。

本系统所有涉及文件系统的模块都依赖于此模块。

5.1.2 日志模块实现 `logger_factory.py`

在 Python Logging 模块基础上重新实现了自定义的日志功能。具有 `init_log_dir()`，`setup_logger()`，`mylogger()`，`loggers={}` 等方法和属性。

`init_log_dir()` 接受字符串路径可选参数，初始化日志记录路径。若参数为空，则使用默认路径。

`setup_logger(name, log_folder=None, lvl_file=None, lvl_stdout=None)` 是一个工厂函数，创建一个新的 `Logger` 对象，根据参数设置 `FileHandler()`，`StreamHandler()`。支持自定义日志路径、文件名称和不同 `Handler` 使用不同记录等级。

`mylogger()` 实现 `logger` 对象的单例模式^[19]。利用 Python 模块引入为单例的特性，使得在同一模块内多次调用相同 `<name>` 的 `mylogger()` 会返回相同的 `Logger` 对象。当第一次调用 `mylogger(<name>)` 时，会调用 `setup_logger()` 创建一个新的 `Logger` 对象，并且将其保存在全局变量 `loggers` 字典内。之后调用同样的 `mylogger(<name>)` 时，会根据 `<name>` 从 `loggers` 字典查找对应的 `Logger` 对象并返回。

在一个模块内首次调用 `mylogger(<name>)` 会初始化日志，添加日志的文件控制器与标准输出控制器，并且设定输出日志文件位置，以“本模块名 + 当前时分秒”命名。后续再次调用 `mylogger(<name>).log()` 会使用首次创建的 `logger` 对象进行日志记录。

5.1.3 环境配置文件 `secret.py`

以 Python 变量的形式存储 MongoDB 与 Neo4j 数据库的用户名、密码、数据库名、连接 URL 等信息，和一些私有 API 的密钥等信息。此文件被 `.gitignore`，不会同步到远程仓库。

5.2 数据采集子系统

数据采集子系统工作流程如 5-1 所示。

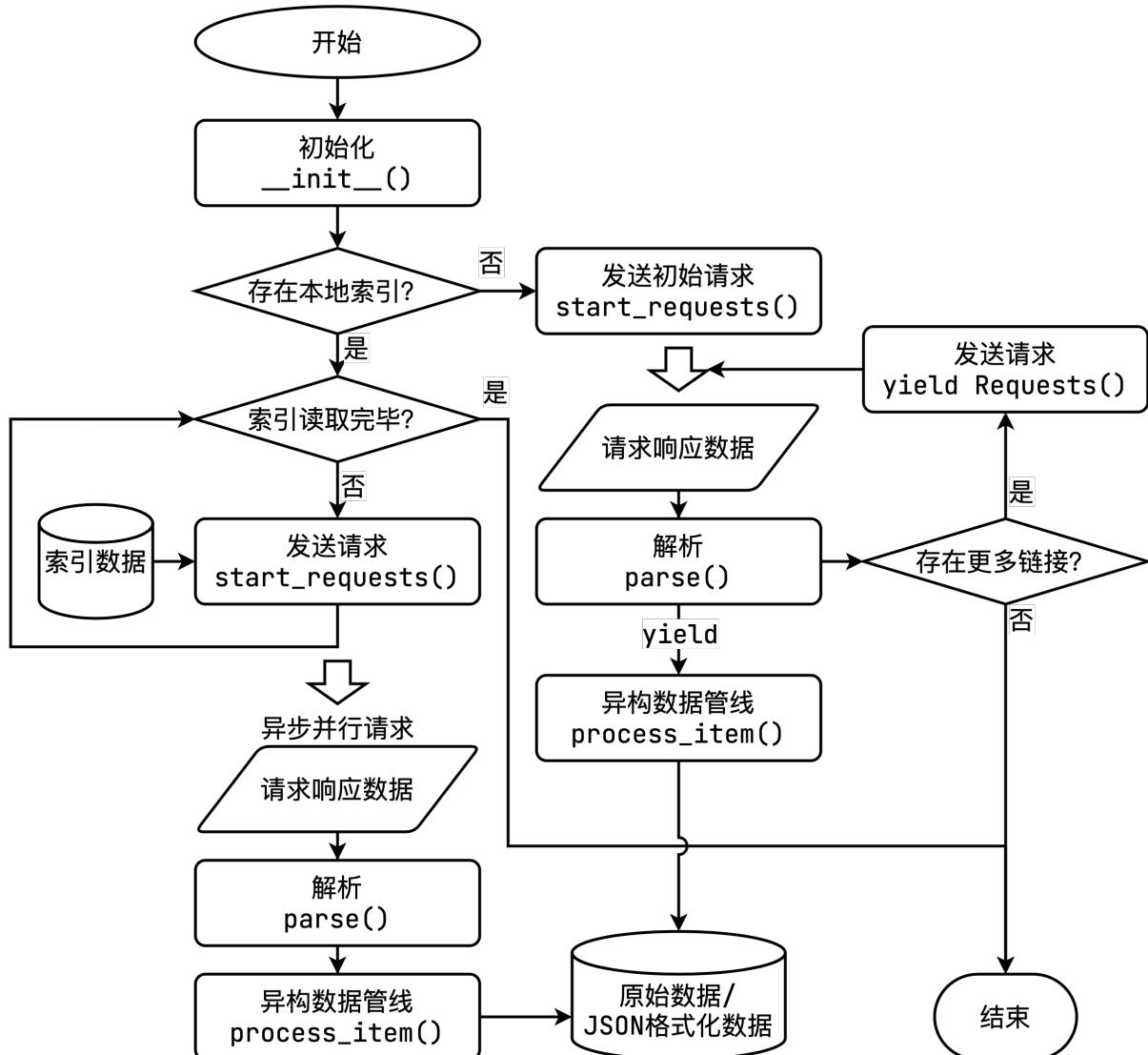


图 5-1 数据采集子系统工作流程图

5.2.1 Spider 模块实现

每个 Spider 继承自 `scrapy.Spider` 类，重写了父类的 `__init__()`, `start_requests()`, `parse()` 方法。

5.2.1.1 初始化 `__init__()` 实现

`__init__()` 方法完成 Spider 类的初始化工作。在调用父类的 `super().__init__()` 方法之后，调用本系统基于 Python Logging 模块重新自定义封装的 `mylogger(<name>)` 函数初始化 logger。

5.2.1.2 发起请求 start_requests() 实现

`start_requests()` 方法描述发起请求的步骤。

对于需要 cve id 索引的 Spider 如 `CveDetailSpider(scrapy.Spider)`, 使用 Pandas 库的 `read_csv()` 方法, 设置 `chunksize` 数值和 `iterator=True`, 分块迭代读取 csv 文件。对每一行 cve id, 生成所需 URL 并使用 `yield scrapy.Request(url=url, callback=self.parse)` 发出请求。

对于需要从爬取页面中获得下一页 URL 的 Spider 如 `EdbSpider(scrapy.Spider)`, `start_request()` 方法中仅指定初始页面发出请求, 在解析收到的 response 时处理下一页的链接, 并发出请求。

对于采用 JavaScript 动态加载页面的站点如 `exploit-db.com`, 使用将 `playwright` 无头浏览器与 Scrapy 框架集成的 `scrapy-playwright` 库, 对爬取到的 HTML 页面及其 JavaScript 代码进行渲染, 将渲染生成的 DOM 返回给 Spider 的 `parse()` 方法。

5.2.1.3 解析数据 parse() 实现

`parse()` 方法实现对请求返回数据的解析。

对于返回数据为 HTML 文档类型, 使用 `BeautifulSoup` 库加载 DOM 进行解析。使用 `soup.select_one()` 方法配合 CSS 选择器提取所需标签的内容。对于需要在页面上查找下一请求 URL 的站点, 使用 `yield scrapy.Request()` 方法发出请求。将解析出的数据封装在 `scrapy.Item` 类中, 使用 `yield item` 方法将数据传递给对应数据管线。

对于返回数据为 JSON 类型, 使用 `response.json()` 获取返回数据, 将其作为参数实例化 `Item` 对象, 传递给 JSON 数据管线。

对于单个请求返回数据为 Gz 压缩的 XML 类型, 使用工具模块 `gz.py` 的 `un_gz()` 方法解压文件得到 XML, 并使用 `lxml` 模块的 `etree.iterparse(filepath, tag='')` 方法迭代遍历该 XML 文件, 填充 `item`, 最终调用持久化子系统的数据库控制对象提供的 `save_cpe()` 方法存储。

将以上代码段包裹在 Python 的 `try...except...finally...` 语句内, 使日志模块能正常 `catch` 抛出的错误, 并使程序不中断继续运行。

5.2.2 Item Pipeline 模块实现

每个 Pipeline 类实现一个 `process_item()` 方法, 同时在 Spider 的 `custom_settings` 内设置对应的 Pipeline 类。Scrapy Engine 将 Spider 内 `yield item` 发送给 Pipeline 对象进行处理。

`GzPipeline` 负责处理压缩成 Gzip 格式的 cve 与 cpe XML 文件。将其解压并返回解压后文件的路径。

`NvdPipeline` 负责处理 NVD API 提供的 JSON, 提取数据并调用持久化子系统的数据库控制对象提供的 `save_nvd_src()` 方法存储原始数据, 使用 `save_nvd()` 方法存储处理

过的数据。

其余 Pipeline 功能类似，根据爬取站点不同，实现有所不同。

5.2.3 其他模块实现

- `scrapy.cfg`: 存放 Scrapy 项目设置选项，如默认设置文件、项目名称等。
- `settings.py`: Python 变量形式存放 Scrapy 爬虫设置选项，如并发度、请求延时、User-Agent 等。
- `crawl_runner` 目录: 包含 `crawl_cpe.py`, `crawl_nvd.py`, `crawl_edb.py` 等。使用 `scrapy.cmdline.execute()` 方法运行爬虫程序。可以被调度模块调用，实现全自动爬虫。

5.3 知识图谱构建子系统

数据采集子系统工作流程如 5-2 所示。

5.3.1 实体生成

实体生成模块主要包含

```

1  init_nodes(vul_num = 0, asset_num = 0, exploit_num = 0)
2  init_vul_ray(skip, _limit)
3  init_asset_ray(skip, _limit)
4  init_exploit_ray(skip, _limit)
5  init_asset_family_ray(skip, _limit)
6  get_step(num)

```

几个方法。

5.3.1.1 `init_nodes()` 实现

`init_nodes(vul_num, asset_num, exploit_num)` 方法接受可选参数漏洞数量、资产数量、利用代码数量，若参数不为零，则将数据根据并行度参数分割并行处理，使用 `ray.get()` 方法等待并行处理结束。示例代码如下：

```

1  '''Ensure Vulnerability, Asset, Exploit nodes exist'''
2  arr = [init_vuln_ray.remote(skip=i, _limit=get_step(vuln_num) + 1) for i in
        range(0, vuln_num, get_step(vuln_num))]
3  arr.extend([init_asset_ray.remote(skip=i, _limit=get_step(asset_num) + 1) for
            i in range(0, asset_num, get_step(asset_num))])
4  arr.extend([init_exploit_ray.remote(skip=i, _limit=get_step(exploit_num) + 1)
            for i in range(0, exploit_num, get_step(exploit_num))])

```

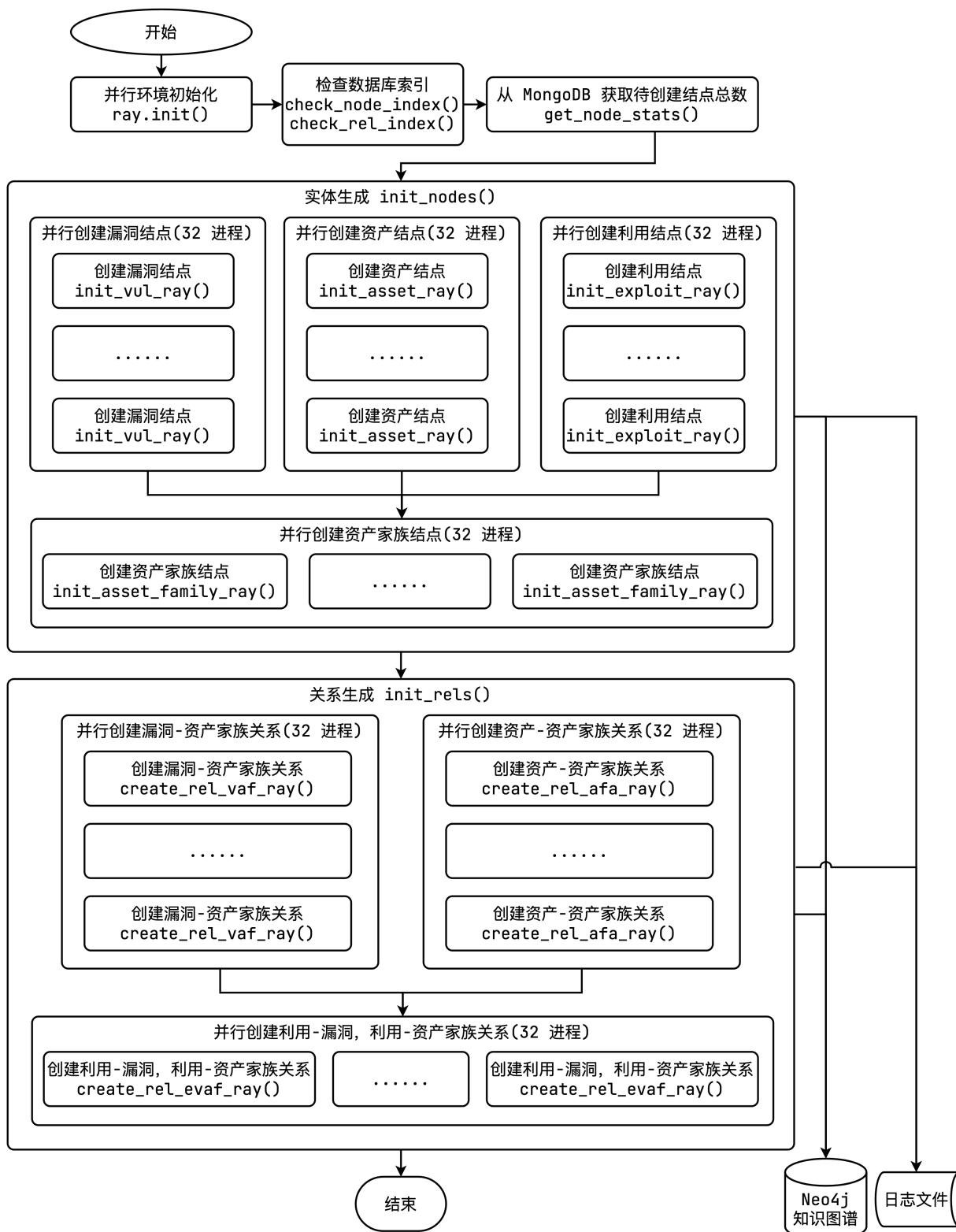


图 5-2 知识图谱构建子系统工作流程图

```

5   ray.get(arr)
6   '''Ensure Asset:Family nodes exist'''
7   family_id = [init_asset_family_ray.remote(skip=i, _limit=get_step(asset_num)
8     + 1) for i in range(0, asset_num, get_step(asset_num))]
9   ray.get(family_id)

```

若不传入漏洞数量等参数， 默认使用单线程串行处理。

5.3.1.2 漏洞实体生成的实现 init_vul_ray()

`init_vul_ray(skip, _limit)` 方法用于生成漏洞实体结点。`skip` 参数指定跳过数据库查询结果中的前多少个记录，`_limit` 参数指定最多处理多少个数据库查询结果。两个参数配合使用，可以实现从第 `<skip>` 个记录开始，处理 `_limit` 个记录。从而实现并行化处理。伪代码：

```

1  cursor = mongo.get_nvd().skip(skip).limit(_limit) # 获取 MongoDB 数据库查询指
   针
2  for doc in cursor:
3      try:
4          doc = split_properties(doc, api_ver=doc['api_ver']) # 根据 API 版本，提取属
           性
5          neo.add_vul(doc) # 调用数据库控制对象添加 doc
6      except BaseException as e:
7          # 异常处理
8  mongo.close_db() # 释放数据库连接

```

创建一个持久化子系统数据库控制器单例对象 `_mg = MyMongo()`，调用方法 `get_nvd(cve_id = None)` 不传入参数，获取 MongoDB 数据库中全部 NVD 数据的迭代指针对象。根据传入参数从指定位置迭代处理查询记录。

对于每个记录，调用 `split_properties(doc, api_ver = ApiVersion.NVDv1)` 方法，`ApiVersion` 是一个自定义枚举类型，用于枚举不同的数据 API 类型。这个方法根据不同的 `api_ver` 使用不同的处理方式，将原始数据中的信息分割为不同语义，并将可用于索引的属性域提至外层。随后调用循环体外创建的持久化子系统数据库控制器单例对象 `neo = MyNeo()` 将实体结点不重复地加入知识图谱中。

将以上循环体内代码包裹在 `try...except...finally...` 语句内保证正确处理异常并且程序不会意外中断。此外，在处理开始、循环体中间、处理结束均使用前述日志模块的 `mylogger()` 方法进行日志输出。

5.3.1.3 资产实体生成的实现 init_asset_ray()

`init_asset_ray(skip, _limit)` 方法用于生成资产实体结点。参数作用同 漏洞实体生成的实现 `init_vul_ray()` 章节，代码实现与 漏洞实体生成的实现 `init_vul_ray()` 章节所

述类似。

5.3.1.4 利用代码实体生成的实现 init_exploit_ray()

`init_exploit_ray(skip, _limit)` 方法用于生成漏洞利用实体结点。参数作用同 漏洞实体生成的实现 `init_vul_ray()` 章节，代码实现与 漏洞实体生成的实现 `init_vul_ray()` 章节所述类似。

5.3.1.5 资产家族实体生成的实现 init_asset_family_ray()

`init_exploit_ray(skip, _limit)` 方法用于生成资产家族实体结点。参数作用同 漏洞实体生成的实现 `init_vul_ray()` 章节。

该方法与 `init_asset_ray()` 方法为串行依赖关系，需要在其之后执行。调用 `NodeMatcher().match()` 方法，从知识图谱中匹配全部资产结点，返回一个迭代指针对象。迭代遍历全部结点，对 `cpe23uri` 资产类型、制造商、产品名称都相同的节点，认为它们同属一个家族。为每个家族不重复地创建一个资产家族实体结点，存入知识图谱。伪代码：

```

1  cursor = neo.match('Asset').skip(skip).limit(_limit) # 查询 Neo4j 所有资产结
   点，返回迭代指针
2  for asset_node in cursor:
3      try:
4          _neo.add_asset_family_node(asset_node['cpe23uri']) # 根据资产结点的
           cpe23uri 向图谱不重复地添加资产家族
5      except BaseException as e:
6          # 异常处理
7  neo.close_db() # 释放数据库连接

```

5.3.2 关系生成

关系生成模块主要包含

```

1  init_rels(vul_num = 0, exploit_num = 0)
2  create_rel_vaf_ray(skip, _limit)
3  create_rel_afa_ray(skip, _limit)
4  create_rel_evaf_ray(skip, _limit)
5  get_step(num)

```

几个方法。

5.3.2.1 关系生成的实现 init_rels()

`init_rels(vul_num, exploit_num)` 方法接受可选参数漏洞数量、利用代码数量，若参数不为零，则将数据根据并行度参数分割并行处理，使用 `ray.get()` 方法等待并行处理结束。示例代码如下：

```

1 arr = []
2 '''Ensure Vulnerability---Family relationships'''
3 arr.extend([create_rel_vaf_ray.remote(skip=i, _limit=get_step(vuln_num) + 1)
             for i in range(0, vuln_num, get_step(vuln_num))])
4 '''Ensure Family---Asset relationships'''
5 arr.extend([create_rel_afa_ray.remote(skip=i, _limit=get_step(vuln_num) + 1)
             for i in range(0, vuln_num, get_step(vuln_num))])
6 ray.get(arr)
7 arr = [create_rel_evaf_ray.remote(skip=i, _limit=get_step(exploit_num) + 1)
         for i in range(0, exploit_num, get_step(exploit_num))]
8 ray.get(arr)

```

若不传入漏洞数量等参数，默认使用单线程串行处理。

5.3.2.2 漏洞 - 资产家族关系生成的实现 create_rel_vaf_ray()

vaf 表示 Vulnerability 和 AssetFamily。`create_rel_vaf_ray(skip, _limit)` 方法用于生成漏洞实体结点与资产家族实体结点之间的关系边。`skip` 参数指定跳过数据库查询结果中的前多少个记录，`_limit` 参数指定最多处理多少个数据库查询结果。两个参数配合使用，可以实现从第 `<skip>` 个记录开始，处理 `_limit` 个记录。从而实现并行化处理。伪代码：

```

1 cursor = neo.match("Vulnerability").skip(skip).limit(_limit) # 查询 Neo4j 所
                  有资产结点，返回迭代指针
2 for vuln_node in cursor:
3     rel_cnt = 0 # 匹配资产的计数器
4     props = json.loads(vuln_node['props']) # 读取序列化存储的属性域
5     for op_dict in props['assets']:
6         try:
7             if op_dict['operator'] == 'OR':
8                 for match in op_dict['cpe_match']:
9                     if match['vulnerable']:
10                         rel_cnt += neo.add_rel_cql_vaf(cve_id=vuln_node['cve_id'], cpe23uri=
11                                         match['cpe23Uri']) # 为一条 cpe23uri 进行匹配，并记录匹配数量
12             except BaseException as e:
13                 # 异常处理
14             neo.add_rel_cql_vaf_cnt(cve_id=vuln_node['cve_id']) # 为关系边增加资产计数器
15             属性

```

```
14 neo.close_db() # 释放数据库连接
```

根据漏洞实体结点属性中存储的受影响资产正则匹配信息，若按照“对于每个正则表达式，匹配知识图谱中符合规则的资产实体结点，为漏洞实体结点与资产实体结点添加两条关系边”的最细粒度做法，知识图谱中的 120 万个结点将生成超过 1 亿条边，且全图正则匹配无法有效利用建立在属性域上的索引，本项目的性能需求是无法接受的。

因此，采用“对于每个正则表达式，匹配知识图谱中符合规则的资产家族实体结点，为漏洞实体结点与资产家族实体结点添加两条关系边，并把真实受影响资产列表存储在关系边属性上”的方法，可以在不改变使用效果的前提下，将构建图谱所需的边数从亿级降低到百万级，提升运行效率、减小数据库体积。

为了进一步加速匹配，把正则表达式中 `.*` 之前的固定模式串切割成子串，使用该固定模式串向数据库进行 `STARTS WITH` 查询，将返回的查询结果在 Python 中进行正则匹配进一步精确筛选。由于 Neo4j 数据库查询时支持 `STARTS WITH` 子句使用属性域索引，采用此种方法查询，使得查询速度提高了 100 倍以上。

5.3.2.3 资产家族 - 资产关系生成的实现 `create_rel_afa_ray()`

`afa` 表示 `AssetFamily` 和 `Asset`。`create_rel_afa_ray(skip, _limit)` 方法用于生成资产实体结点与资产家族实体结点之间的关系边。`skip` 参数指定跳过数据库查询结果中的前多少个记录，`_limit` 参数指定最多处理多少个数据库查询结果。两个参数配合使用，可以实现从第 `<skip>` 个记录开始，处理 `_limit` 个记录。从而实现并行化处理。伪代码与章节 5.3.2.2 漏洞 - 资产家族关系生成的实现 `create_rel_vaf_ray()` 类似。

对于每个漏洞实体结点，遍历其受影响资产正则匹配表达式列表，对每个正则匹配表达式，通过寻找其前缀获得其属于的资产家族，调用持久化子系统数据库控制器对象的 `add_rel_cql_afa()` 方法，为该资产结点和资产家族结点之间添加“资产家族结点是资产结点父级”、“资产结点是资产家族结点子级”两个关系。

5.3.2.4 利用 - 漏洞 - 资产家族关系生成的实现 `create_rel_evaf_ray()`

`evaf` 表示 `Exploit` 和 `Vulnerability` 和 `AssetFamily`。`create_rel_evaf_ray()` 方法用于生成漏洞利用代码结点与漏洞实体结点之间的关系边，并且从这条关系与漏洞实体结点指向的受影响资产家族结点，推理出漏洞利用代码结点与资产家族结点之间的关系。`skip` 参数指定跳过数据库查询结果中的前多少个记录，`_limit` 参数指定最多处理多少个数据库查询结果。两个参数配合使用，可以实现从第 `<skip>` 个记录开始，处理 `_limit` 个记录。从而实现并行化处理。伪代码：

```
1 cursor = neo.match("Exploit").skip(skip).limit(_limit) # 查询 Neo4j 所有利用
   代码结点，返回迭代指针
2 for exploit_node in cursor:
3     cve_ids = exploit_node['cve_ids'] # 获得利用代码对应的 CVE 漏洞编号列表
```

```

4     r_cnt = 0 # 匹配资产的计数器
5     for cve_id_no in cve_ids:
6         cve_id = f'CVE-{cve_id_no}'
7         vuln_node = neo.get_node('Vulnerability', cve_id=cve_id).first() # 查找
               cve_id 对应的漏洞实体结点
8         if vuln_node is not None:
9             r_cnt += neo.add_rel_cql_ev(edb_id=exploit_node['edb_id'], cve_id=cve_id)
                   # 向图谱中加入利用 - 漏洞关系
10        cpe_list = neo.get_rel_cql_vaf(cve_id=cve_id) # 获取受影响资产列表
11        for rel in cpe_list: # 遍历受影响资产列表
12            r_cnt += neo.add_rel_cql_eaf(edb_id=exploit_node['edb_id'], cpe23uri=
                   rel['target'], assets=rel['assets'], asset_cnt=rel['cnt']) # 向图谱
                         中加入利用 - 资产家族关系
13    neo.close_db() # 释放数据库连接

```

遍历漏洞利用代码结点，对于带有 cve id 属性的结点，从图谱中查询对应 cve id 的漏洞实体结点。为漏洞利用结点与漏洞实体结点之间添加“漏洞利用利用漏洞”、“漏洞被漏洞利用利用”两个关系。再从漏洞实体结点出发，寻找所有受影响的资产家族结点。根据“漏洞影响资产家族”关系边上的属性，为漏洞利用结点与资产家族结点添加“漏洞利用针对资产家族”、“资产家族被漏洞利用针对”两个关系，属性继承自前述关系边属性。

5.3.3 关系融合实现

关系融合模块作为适配使用深度学习模型进行命名实体识别结点生成与关系抽取方案的模块，当前仅在系统数据加工管线中留有接口，以备未来扩展需要。当前系统的结构化数据实体与关系生成已经实现实体结点和关系的消除二义性，因此关系融合模块暂未实现。

5.4 持久化子系统实现

持久化子系统利用 Python 默认引入模块默认为单例的特性，使用饿汉模式^[20] 实现灵活的单例数据库控制器类，分别为 class MyMongo 与 class MyNeo。

饿汉模式灵活之处在于，在单线程运行场景下，可直接引入在 db.py 内实例化的数据库控制器对象，如 mg = MyMongo()，neo = MyNeo()，实现单例模式。而在多线程运行场景下，可以引入 class MyMongo 与 class MyNeo 类，并在每个线程内分别实例化。

5.4.1 MyMongo 类实现

实现的方法如下：

```

1 check_index() # 不重复地创建数据库所有集合的索引
2 save_cvedetails_json(cve_id, doc) # 保存 cvedetails JSON 数据
3 get_cvedetails_json(cve_id = None) # 查询 cvedetails JSON 数据
4 save_nvd(cve_id, doc) # 保存 nvd JSON 数据
5 get_nvd(cve_id = None) # 查询 nvd JSON 数据
6 save_cpe(cpe23uri, doc) # 保存 cpe JSON 数据
7 get_cpe(cpe23uri = None) # 查询 cpe JSON 数据
8 save_edb_html(edb_id, doc) # 保存 exploit HTML 数据
9 save_edb_json(edb_id, doc) # 保存 exploit-db JSON 数据
10 get_edb_json(edb_id = None) # 查询 exploit-db JSON 数据
11 get_exploit_stats() # 查询漏洞利用统计信息

```

其中所有 get 方法的可选参数为空时，返回全部数据的一个迭代器指针。

5.4.2 MyNeo 类实现

实现的方法如下：

```

1 get_session() # 获取一个数据库连接
2 check_node_index() # 不重复地创建结点相关属性域的索引
3 check_rel_index() # 不重复地创建关系相关属性域的索引
4 get_node(*args, **kwargs) # 封装的通用查询结点方法
5 add_node(labels, props) # 封装的通用添加结点方法
6 add_relationship(start, _type, end, props = None) # 封装的通用添加关系方法
7 delete_relationship(cve_id = '') # 删除与某漏洞实体结点相关的所有关系
8 match_asset(pattern) # 根据正则表达式，匹配资产
9 match_asset_family(pattern) # 根据正则表达式，匹配资产家族
10 add_asset_family_node(cpe23uri) # 添加资产家族结点
11 add_rel_cql_vaf(cve_id, cpe23uri) # 在漏洞结点与资产家族结点之间添加两条关系，使用 CQL 实现
12 add_rel_cql_afa(asset_uri) # 在资产结点与资产家族结点之间添加两条关系，使用 CQL 实现
13 close_db() # 关闭数据库连接，释放资源

```

5.5 后端服务子系统实现

`create_app.py` 为创建 Flask App 实例的工厂函数^[21]。主要包含创建 Flask 实例、挂载中间件、注册蓝图等步骤。

5.5.1 知识图谱 API 实现 graph.py

知识图谱相关 API 的蓝图。向 Flask 注册 /api/graph/ 路径下的 API，并且实现这些 API 对应的功能。主要实现了以下方法：

```

1  _convert_node(entry, i, node_map) # 从知识图谱结点数据结构，转换为绘图数据结点
   数据结构
2  _convert_link(entry, i, rel_map) # 从知识图谱关系数据结构，转换为绘图数据关系数
   据结构
3  _retrieve_v_af(cve_id) # 查询漏洞 - 资产家族结点与关系
4  _retrieve_af_a(cpe23uri) # 查询资产家族 - 资产结点与关系
5  _retrieve_e_af(edb_id) # 查询利用 - 资产家族结点与关系
6  _retrieve_e_v(edb_id) # 查询利用 - 漏洞结点与关系
7  .....等等
8  _label_to_id_field(label) # 通过字典映射，将标签字符串转换为索引属性域字符串
9  _get_symbol_size(_type, cnt) # 计算结点在可视化力引导图中显示的大小
10 _get_node_category(type_list) # 根据结点的标签列表，选出主要标签
11
12 retrieve_graph_stats() # 获取知识图谱统计数据
13 retrieve_graph(limit) # 获取<limit>个漏洞结点及与之相关的结点与关系绘图数据
14 search_graph(keyword) # 根据关键字搜索知识图谱

```

5.5.1.1 知识图谱统计数据 API 实现 /api/graph/

获取知识图谱统计数据。调用持久化子系统的数据库控制器对象的 `get_session().read_transaction()` 方法，向数据库提交一个读事务。读事务 `work(tx)` 中定义查询的 CQL 语句，并调用 `transaction` 对象的 `tx.run()` 方法运行查询。查询获得漏洞数量、受影响资产类型及数量、漏洞利用代码数量及类型等信息。调用绘图数据生成方法，生成结果数据，返回给前端。

5.5.1.2 知识图谱可视化数据 API 实现 /api/graph/<limit>

获取 `<limit>` 个漏洞结点及与之相关的结点与关系绘图数据。为防止数据量过大，实际查询将取 `<limit>` 与 200 间的较小值。根据经验数据，返回结点和关系数量不超过一万个。调用绘图数据生成方法，获得各类结点及关系绘图数据，返回给前端。

需要返回的结点和关系有：漏洞结点、资产家族结点、资产结点、利用结点、漏洞 - 资产家族关系、漏洞 - 利用关系、资产 - 资产家族关系、漏洞 - 资产家族关系。由于长关系链可以拆解成三元组形式，因此对于任意查询，仅需要实现查询各种三元组的方法、将三元组融合的方法即可。

如 `_retrieve_v_af` 实现了漏洞 - 资产家族结点及关系的查询，伪代码如下：

```
1 def _retrieve_v_af(tx, cve_id):
```

```

2   cql = 'MATCH (v:Vulnerability)-[r]-(af:Asset:Family) ' \
3       'WHERE v.cve_id=$cve_id' \
4       'RETURN v,r,af' # 查询所用 CQL 语句
5   _res = tx.run(cql, cve_id=cve_id) # 使用事务运行查询，使用参数防止 CQL 注入攻击
6   v_map, r_map, af_map = {}, {}, {}
7   for entry in _res: # 对于查询结果中的每一条
8       _convert_node(entry, i, v_map) # 转换漏洞结点
9       _convert_node(entry, i, af_map) # 转换资产家族结点
10      _convert_link(entry, i, r_map) # 转换关系
11
12      for _r in r1_map.values(): # 计算受影响资产数量
13          _key = _r['name'].split('->')[0]
14          if _key.startswith('CVE'):
15              v_map[_key]['size_cnt'] += _r['asset_cnt']
16          elif _key.startswith('cpe'):
17              af_map[_key]['size_cnt'] += _r['asset_cnt']
18
19      for _v in v_map.values(): # 对于每个漏洞，根据受影响资产数量，计算绘图结点的大小
20          _v['label'] = vul_label_settings
21          _v['symbolSize'] = get_symbol_size('v', _v['size_cnt'])
22      for _af in af_map.values(): # 对于每个资产，根据子资产数量，计算绘图结点的大小
23          _af['label'] = af_label_settings
24          _af['symbolSize'] = get_symbol_size('af', _af['size_cnt'])
25      for _a in a_map.values():
26          _a['symbolSize'] = get_symbol_size('a')
27
28  return v_map, r_map, af_map # 返回存储三元组的三个字典

```

为了使可视化显示更加清晰，且能直观地从结点显示大小得知该结点的重要性，实现了计算绘图结点大小 `get_symbol_size()`。该方法接受结点类型和受影响结点数量两个参数，计算结点应显示的大小。计算公式为 $size = base_size + 5 * \sqrt{cnt}$ ，其中 `base_size` 为由一个字典预定义的数值，不同类型结点的 `base_size` 不同。`cnt` 为受该结点影响的结点的数量，最终得到的 `size` 即为结点显示大小，单位为像素。

5.5.1.3 知识图谱搜索 API 实现 /api/graph/search/<keyword>

根据关键字搜索知识图谱。根据关键字类型，判断搜索内容属于漏洞、资产、利用代码或是关系。将搜索到的结点/关系及其相关联（关系路径长度为 1）的结点或关系返回，调用绘图数据生成方法获得绘图所需数据，返回给前端。

如图 5-3 所示，查询系统采用模块化设计，将搜索服务独立为模块。支持扩展为微服务设计，在接口不变情况下可实现搜索提供服务的热更换。当前查询模块仅支持特

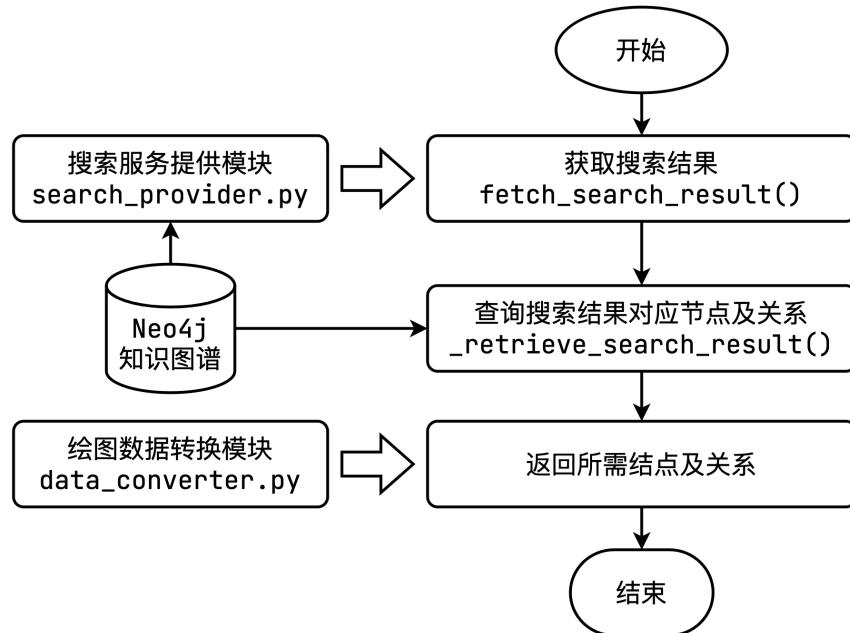
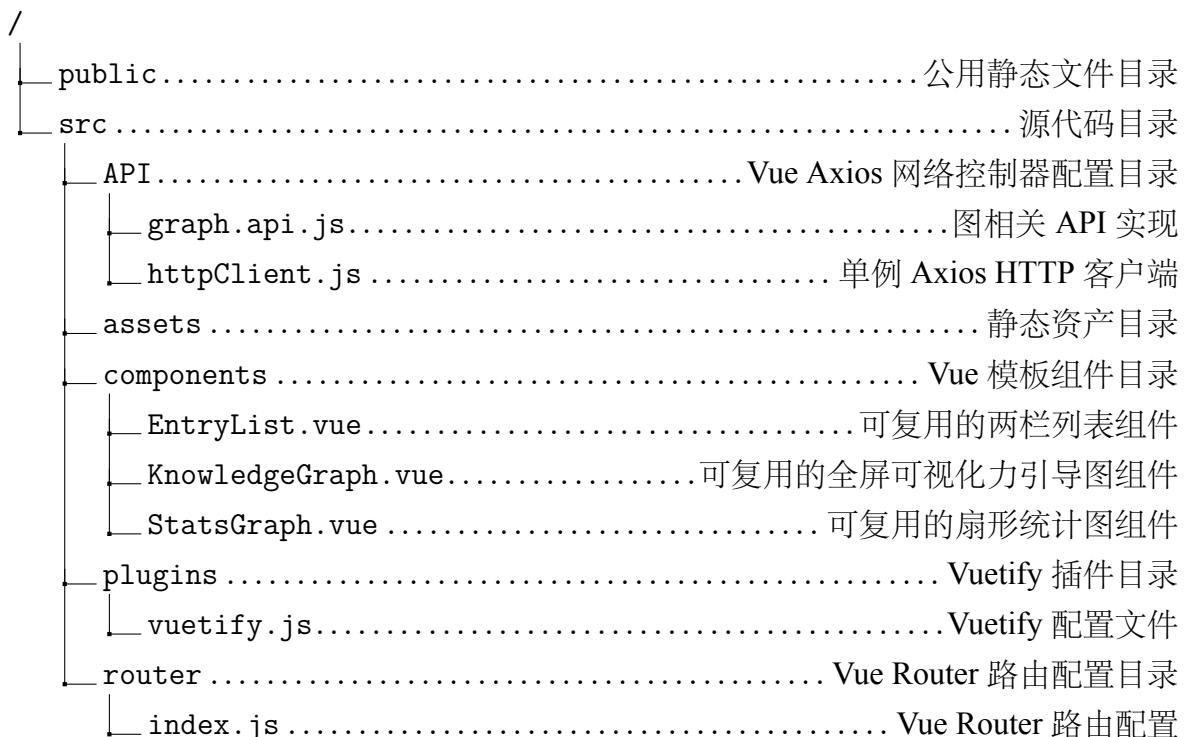


图 5-3 知识图谱搜索 API 流程图

定属性域的查询如 `cve_id`, `cpe23uri`, `edb_id` 等。后续可将系统扩展为使用正则表达式进行匹配、使用 Elastic Search 进行自然语言属性域的搜索等。获取搜索结果后，其步骤与章节 5.5.1.2 知识图谱可视化数据 API 实现 `/api/graph/<limit>` 所述过程类似。

5.6 前端子系统

5.6.1 项目结构



store	Vuex 仓库配置目录
modules	Vuex 模块配置目录
net.js	网络控制器 Vuex 模块
store.js	Vuex 仓库根模块
utils	工具目录
langZH.js	ECharts 中文本地化配置文件
tooltipConfig.js	ECharts tooltip 格式化工具
views	视图目录
Welcome.vue	登录首屏欢迎界面视图
Overview.vue	概览视图
Visualization.vue	可视化视图
Search.vue	搜索视图
About.vue	关于视图
App.vue	根组件，描述页面框架结构及基础功能
main.js	向页面引入并注册 Vue 组件
registerServiceWorker.js	注册 ServiceWorker，实现 PWA
.env	环境变量
package.json	npm 配置文件
babel.config.js	babel 配置文件
vue.config.js	vue 配置文件
eslintrc.js	eslint 配置文件
browserslistrc	支持的浏览器配置文件

5.6.2 概览视图实现

概览视图采用 Flex 流式卡片布局，通过设置 CSS 样式断点，每个卡片根据当前设备屏幕分辨率自动调节宽度。例如，宽度为 `lg` 时，界面布局为每行 3 列卡片。宽度为 `md` 时，界面布局为每行 2 列卡片。宽度为 `sm` 时，为每行 1 列卡片。

卡片中使用 `EntryList` 组件显示知识图谱统计信息，辅以 `StatsGraph` 组件显示可视化扇形图。图示见章节 6.5.1 概览视图测试。

`EntryList` 与 `StatsGraph` 使用 `Vuex` 的 `mapState()` 计算属性作为组件属性，组件内部为 `props` 添加 `watch` 倾听器，实现数据响应式动态更新。

`Dashboard.vue` 在 `mounted()` 时会通过 `Vuex` 仓库的 `this.$store.dispatch()` 发出一个异步 I/O Action 请求，向后端请求知识图谱的统计数据。此时前端界面会优先加载，由默认空值或零值填充列表和统计图。当 Axios 请求回调时，`Vuex Action` 会向 `Vuex Store` 提交数据产生的更改。这引起各个视图或组件内计算属性值变化，倾听器触发，执行回调函数，调用 `ECharts` 实例对象的 `setOption(option)` 方法，从而动态更新数据。

5.6.3 可视化视图实现

可视化视图使用 ECharts 呈现一个全屏的可视化力引导图，鼠标按住或手指拖拽可以移动视图，鼠标按住或手指拖拽图中结点可以移动结点位置。图示可见章节 6.5.2 可视化视图测试。

可视化视图同样采用来自 Vuex Store 的计算属性作为数据源，实现动态数据更新。在切换视图时，为节省内存 ECharts 实例会被销毁，但绘图数据保留在 Vuex Store 中的 `graphStore.graph` 对象中，由唯一的 `graphId` 标识，因此可同时支持保存同一视图页面的多个可视化图、保存不同视图页面的可视化图。再次切换回可视化视图时，会优先加载缓存的绘图数据，节省网络带宽、减轻后端压力、加快加载速度。

5.6.4 搜索视图实现

搜索视图包含一个悬浮搜索框，显示在容器的左上角。在搜索框输入关键词并按下回车，发出异步 I/O 请求并等待返回绘图数据。后端服务将搜索整个知识图谱查找关键字相关信息，并将绘图数据返回前端。可视化显示流程与上述章节 5.6.3 可视化视图实现相似。

5.6.5 关于视图实现

关于视图包含一些基本文字说明，使用 Flex 进行排版。

第六章 系统测试

6.1 测试环境

- CPU: AMD Ryzen ZEN3 架构 8C16T@4.8GHz
- RAM: 16GB*2 DDR4@3600MT/s
- 存储: PCIe Gen 3 固态硬盘
- 操作系统: Ubuntu 20.04

6.2 数据采集子系统测试

图 6-1 展示数据采集子系统第一轮爬虫运行完毕后, MongoDB 数据库中的 Collections 概览。

通过 Scrapy 爬虫 cve.mitre.org, 获得约 23 万条 cve id 数据, 经筛选清洗获得约 16 万条有效 cve id 信息。

爬取 cpe.mitre.org 定期发布的官方 cpe 信息文件, 使用 lxml 库解析, 经筛选清洗获得约 90 万条有效 cpe 资产信息 JSON 数据。其中包含用于识别的 cpe23uri、cpe 资产名称、参考资料链接等属性。

爬取 cvedetails.com, 获得约 16 万个 HTML 页面, 清洗整理获得超 400 万个 cve 漏洞与资产相关属性 JSON 数据。

通过 nvd.nist.gov JSON API 数据接口, 获得约 16 万条 JSON 数据, 包含漏洞名称与简介、漏洞条目发布日期、漏洞条目更新日期、CVSS 风险评级、参考资料、受影响资产的 cpe23uri 匹配正则表达式等信息。

爬取 exploit-db.com, 获得约 5 万个 HTML 页面, 包含漏洞利用代码、作者、发布时间、适用平台、利用类型、利用 cve id 等信息, 清洗处理为 JSON 格式数据。

6.3 知识图谱构建子系统测试

6.3.1 结点生成

使用 Ray 库提供的 @ray.remote 装饰器与 ray.get() 方法, 将全部漏洞数据根据并行度参数进行分割, 开辟多进程对分块漏洞数据进行生成结点操作。

开辟 32 个进程并行生成漏洞结点、资产结点、资产家族结点、漏洞利用结点, 总计约 120 万个结点。系统耗时约 5 分钟, 如图 6-2 所示。

cpe	Storage size: 56.23 MB	Documents: 872 K	Avg. document size: 539.00 B	Indexes: 2	Total index size: 41.30 MB
<hr/>					
edb_html	Storage size: 1.32 GB	Documents: 45 K	Avg. document size: 160.50 kB	Indexes: 1	Total index size: 851.97 kB
<hr/>					
edb_json	Storage size: 79.51 MB	Documents: 45 K	Avg. document size: 4.22 kB	Indexes: 3	Total index size: 1.80 MB
<hr/>					
html	Storage size: 99.12 MB	Documents: 13 K	Avg. document size: 31.58 kB	Indexes: 1	Total index size: 245.76 kB
<hr/>					
json	Storage size: 81.26 MB	Documents: 168 K	Avg. document size: 2.27 kB	Indexes: 1	Total index size: 3.01 MB
<hr/>					
nvd_json	Storage size: 122.34 MB	Documents: 168 K	Avg. document size: 3.48 kB	Indexes: 2	Total index size: 4.99 MB
<hr/>					
nvd_json_src	Storage size: 141.62 MB	Documents: 168 K	Avg. document size: 3.96 kB	Indexes: 1	Total index size: 3.01 MB

图 6-1 数据采集子系统运行测试结果

```

107 P1076673-T140451038263104- init_asset_ray - 2022-05-09 11:32:50.838 - timer - INFO: init_asset.skip(218080) with limit 54521 runtime = 0:04:14.356317
108 P1076670-T140719109216064- init_asset_ray - 2022-05-09 11:32:50.976 - timer - INFO: init_asset.skip(872320) with limit 54521 runtime = 0:00:00.382658
109 P1076667-T139979017795392- init_asset_ray - 2022-05-09 11:32:51.223 - timer - INFO: init_asset.skip(0) with limit 54521 runtime = 0:04:15.134648
110 P1076676-T139793847834432- init_asset_ray - 2022-05-09 11:32:51.372 - timer - INFO: init_asset.skip(163560) with limit 54521 runtime = 0:04:14.903336
111 P1076675-T140375827486528- init_asset_ray - 2022-05-09 11:32:51.501 - timer - INFO: init_asset.skip(327120) with limit 54521 runtime = 0:04:14.792446
112 P1076678-T140424310404928- init_asset_ray - 2022-05-09 11:32:51.565 - timer - INFO: init_asset.skip(545200) with limit 54521 runtime = 0:04:14.516161
113 P1076679-T140299165096656- init_asset_ray - 2022-05-09 11:32:51.722 - timer - INFO: init_asset.skip(381640) with limit 54521 runtime = 0:04:14.923320
114 P1076671-T140540818261824- init_asset_ray - 2022-05-09 11:32:51.834 - timer - INFO: init_asset.skip(654240) with limit 54521 runtime = 0:04:14.047892
115 P1076666-T139698529421120- init_asset_ray - 2022-05-09 11:32:51.875 - timer - INFO: init_asset.skip(490680) with limit 54521 runtime = 0:04:15.010879
116 P1076674-T139786773575488- init_asset_ray - 2022-05-09 11:32:51.937 - timer - INFO: init_asset.skip(54520) with limit 54521 runtime = 0:04:15.637990
117 P1076668-T140465220273984- init_asset_ray - 2022-05-09 11:32:51.956 - timer - INFO: init_asset.skip(708760) with limit 54521 runtime = 0:04:14.114990
118 P1076664-T140230025688896- init_asset_ray - 2022-05-09 11:32:51.965 - timer - INFO: init_asset.skip(436160) with limit 54521 runtime = 0:04:15.132061
119 P1076665-T140837714892688- init_asset_ray - 2022-05-09 11:32:52.127 - timer - INFO: init_asset.skip(763280) with limit 54521 runtime = 0:04:13.992295
120 P1076669-T139794027345728- init_asset_ray - 2022-05-09 11:32:52.989 - timer - INFO: init_asset.skip(817800) with limit 54521 runtime = 0:04:14.507559
121 P1076530-T139940318795584- <module> - 2022-05-09 11:32:53.047 - timer - INFO: Runtime = 0:05:22.802448
122

```

图 6-2 结点生成模块运行日志

6.3.2 关系生成

使用 Ray 库提供的 @ray.remote 装饰器与 ray.get() 方法，将全部漏洞数据根据并行度参数进行分割，开辟多进程对分块漏洞数据进行生成操作。

开辟 32 个进程并行执行关系生成，由于资产关系生成涉及正则匹配运算，无法有效利用数据库索引，且 CPU 性能消耗较大，因此生成时间较长。对总计约 120 万个结点共生成约 45 万条边（更多详细信息存储在边属性中），耗时约 1 小时 50 分钟，如图 6-3 所示。

```

29 P746955-T140527074762976 setup_logger - 2022-05-09 17:49:33.530 - timer : INFO: Logger "timer" logging to 0-root.log and 1-timer.log with log_file level 20 and log_stdout level 20.
30 P746955-T140527074782376 cleanup_rel_VAF_ray - 2022-05-09 17:49:33.530 - timer : INFO: create_rel_VAF_ray().skip(120329) with limit 10517 start.
31 P746953-T14004353711936 setup_logger - 2022-05-09 17:49:33.534 - timer : INFO: Logger "timer" logging to 0-root.log and 1-timer.log with log_file level 20 and log_stdout level 20.
32 P746953-T14004353711936 create_rel_VAF_ray - 2022-05-09 17:49:33.534 - timer : INFO: create_rel_VAF_ray().skip(157740) with limit 10517 start.
33 P746945-T14018342996000 setup_logger - 2022-05-09 17:49:33.558 - timer : INFO: Logger "timer" logging to 0-root.log and 1-timer.log with log_file level 20 and log_stdout level 20.
34 P746945-T14018342996000 create_rel_VAF_ray - 2022-05-09 17:49:33.558 - timer : INFO: create_rel_VAF_ray().skip(0) with limit 10517 start.
35 P746950-T139966837622592 create_rel_VAF_ray - 2022-05-09 19:11:33.543 - timer : INFO: create_rel_VAF_ray().skip(147224) with limit 10517 runtime = 1:22:00.218574
36 P746950-T139966837622592 create_rel_VAF_ray - 2022-05-09 19:11:33.560 - timer : INFO: create_rel_VAF_ray().skip(168256) with limit 10517 start.
37 P746950-T139966837622592 create_rel_VAF_ray - 2022-05-09 19:11:33.454 - timer : INFO: create_rel_VAF_ray().skip(168256) with limit 10517 runtime = 0:00:01.008214
38 P746953-T14004353711936 create_rel_VAF_ray - 2022-05-09 19:10:39.554 - timer : INFO: create_rel_VAF_ray().skip(157740) with limit 10517 runtime = 1:28:48.005284
39 P746953-T14004353711936 create_rel_VAF_ray - 2022-05-09 19:21:53.311 - timer : INFO: create_rel_VAF_ray().skip(157740) with limit 10517 runtime = 1:32:19.803467
40 P746954-T140475659360064 create_rel_VAF_ray - 2022-05-09 19:22:40.720 - timer : INFO: create_rel_VAF_ray().skip(10516) with limit 10517 runtime = 1:33:27.397257
41 P746943-T14067588637760 create_rel_VAF_ray - 2022-05-09 19:25:01.459 - timer : INFO: create_rel_VAF_ray().skip(63096) with limit 10517 runtime = 1:35:28.115858
42 P746957-T1409708081623872 create_rel_VAF_ray - 2022-05-09 19:26:19.775 - timer : INFO: create_rel_VAF_ray().skip(42064) with limit 10517 runtime = 1:36:46.457588
43 P746948-T139934798317376 create_rel_VAF_ray - 2022-05-09 19:27:59.893 - timer : INFO: create_rel_VAF_ray().skip(52580) with limit 10517 runtime = 1:38:26.572940
44 P746947-T140478029363086 create_rel_VAF_ray - 2022-05-09 19:28:11.710 - timer : INFO: create_rel_VAF_ray().skip(136708) with limit 10517 runtime = 1:38:38.391414
45 P746951-T139784795939264 create_rel_VAF_ray - 2022-05-09 19:28:25.939 - timer : INFO: create_rel_VAF_ray().skip(126192) with limit 10517 runtime = 1:38:52.597466
46 P746952-T140478029363086 create_rel_VAF_ray - 2022-05-09 19:28:29.859 - timer : INFO: create_rel_VAF_ray().skip(80108) with limit 10517 runtime = 1:38:56.510228
47 P746952-T140478029363086 create_rel_VAF_ray - 2022-05-09 19:28:30.020 - timer : INFO: create_rel_VAF_ray().skip(84644) with limit 10517 runtime = 1:38:57.000110
48 P746943-T1399812279232328 create_rel_VAF_ray - 2022-05-09 19:29:14.268 - timer : INFO: create_rel_VAF_ray().skip(73612) with limit 10517 runtime = 1:39:40.966059
49 P746943-T139919657174848 create_rel_VAF_ray - 2022-05-09 19:30:07.878 - timer : INFO: create_rel_VAF_ray().skip(115676) with limit 10517 runtime = 1:40:34.514543
50 P746955-T140522747782976 create_rel_VAF_ray - 2022-05-09 19:30:39.935 - timer : INFO: create_rel_VAF_ray().skip(21032) with limit 10517 runtime = 1:41:06.437943
51 P746945-T14018342996000 create_rel_VAF_ray - 2022-05-09 19:31:39.317 - timer : INFO: create_rel_VAF_ray().skip(0) with limit 10517 runtime = 1:42:05.792520
52 P746958-T1404287563937602 create_rel_VAF_ray - 2022-05-09 19:36:03.566 - timer : INFO: create_rel_VAF_ray().skip(105160) with limit 10517 runtime = 1:46:30.069641
53 <module> - 2022-05-09 19:36:03.580 - timer : INFO: Runtime: 1:46:33.444462

```

图 6-3 关系生成模块运行日志

6.3.3 关系融合

本系统设计上，结构化数据生成的漏洞实体具有 cve id 属性作为唯一标识；资产实体具有 cpe23uri 属性作为唯一标识；漏洞利用代码实体具有edb-id 属性作为唯一标识。且在数据库控制器对象的方法中，使用 MERGE 等 CQL 子句保证不会向数据库插入重复的实体或属性。因此当前知识图谱内并不会出现语义相同的不同实体或关系。

关系融合模块作为适配使用深度学习模型进行命名实体识别结点生成与关系抽取方案的模块，当前仅在系统数据加工管线中留有接口，以备未来扩展需要。因此暂不作测试。

6.4 持久化子系统与后端服务子系统测试

6.4.1 概览视图统计信息获取 /api/graph/

向运行在 localhost 上的后端服务子系统发送 /api/graph/ 请求，获取当前知识图谱基础统计信息。后端返回各类资产数量统计、各类漏洞数量统计、各类代码利用数量统计等信息，耗时约 200 毫秒，如图 6-4 所示。

6.4.2 可视化视图数据获取 /api/graph/<limit>

向运行在 localhost 上的后端服务子系统发送 /api/graph/100 请求，获取当前知识图谱的 100 个漏洞结点以及与其相关的资产、利用代码结点、关系等信息，用于可视化。

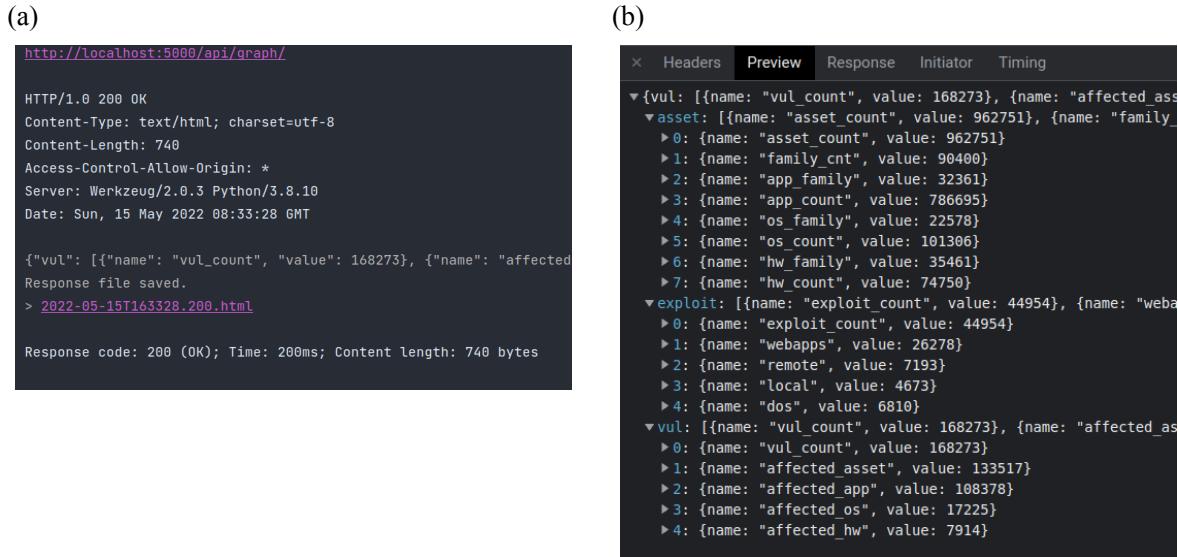


图 6-4 统计信息获取 /api/graph/ 运行测试
(a) 请求响应摘要, (b) 请求响应数据

后端返回经绘图数据生成模块处理的绘图数据, 包含分类、结点、关系三个数组, 耗时约 640 毫秒, 如图 6-5 所示。

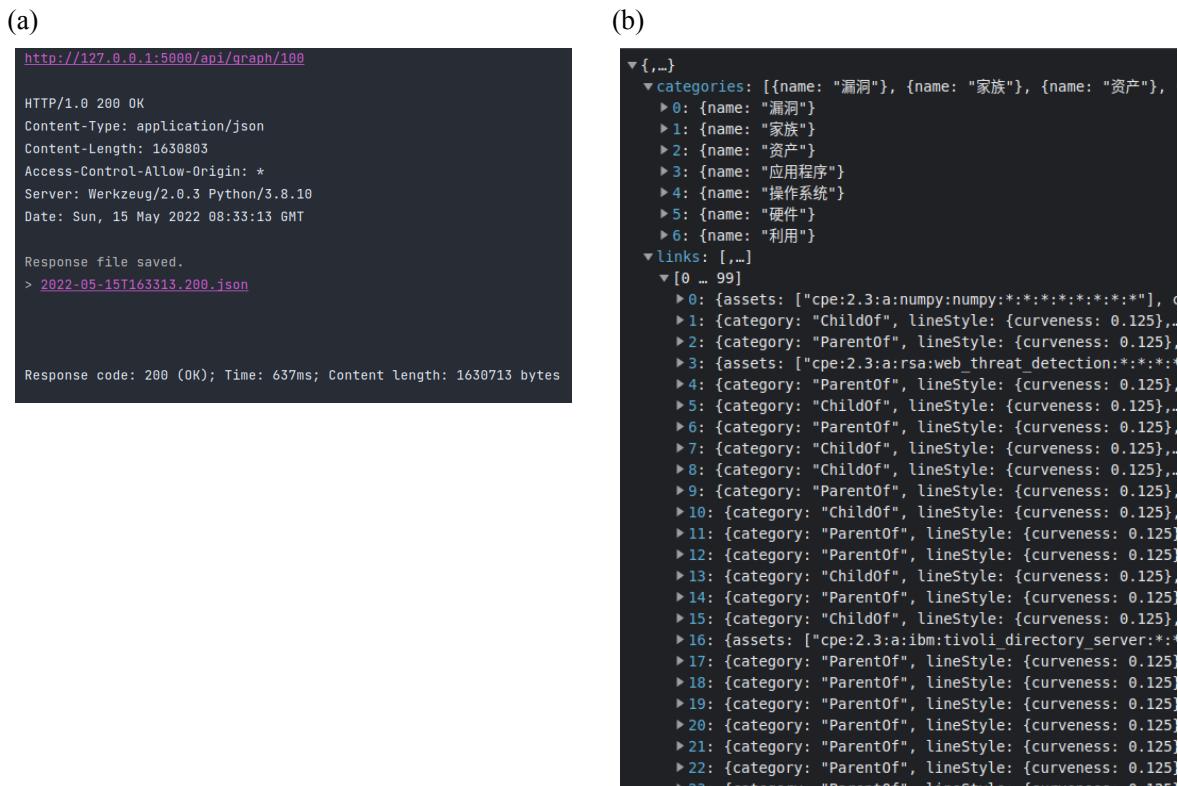


图 6-5 可视化视图数据获取 /api/graph/100 运行测试
(a) 请求响应摘要, (b) 请求响应数据

6.5 前端子系统测试

6.5.1 概览视图测试

图 6-6 所示为概览视图。从 localhost 后端 /api/graph/ 端口加载数据，实际返回，根据瀑布流图可看出，通过刷新页面冷加载概览界面及各种网络请求耗时约 850ms，且加载过程中可视化图表有加载指示器提示正在加载、数据列表有默认数据“0”填充而非显示空白，用户体验较好。如图 6-7 所示。

前端子系统全部界面均采用类 App 设计，具备响应式显示功能，卡片行列数可根据当前设备屏幕分辨率动态流式调整，保证在各种设备上均能正常显示。



图 6-6 概览视图预览

6.5.2 可视化视图测试

图 6-8 所示为知识图谱概览。从 localhost 后端 /api/graph/30 端口加载数据，实际返回 1684 个结点、3368 条关系边，根据瀑布流图可看出，通过刷新页面冷加载可视化力引导图耗时约 1800ms，且加载过程中有加载指示器提示正在加载，用户体验较好。如图 6-9 所示。

图 6-10 所示为知识图谱细节图。力引导图为有向图，因此结点间的无向边使用两条有向边表示。图中蓝色结点为漏洞结点，绿色结点为资产家族结点，红色结点为受影响资产结点。可以看到，通过将漏洞所影响的大量资产关系替换成漏洞到资产家族的一个关系，大幅简化了图的结构，在使视觉观感清晰的同时，将鼠标移至漏洞至资产家族的边上可显示受该漏洞影响的所有资产，保证图谱数据的详细精确。

当鼠标移动至或触摸图中元素，包括各类结点和各类关系，会显示 Tooltip 指示，展



图 6-7 概览视图加载请求瀑布流

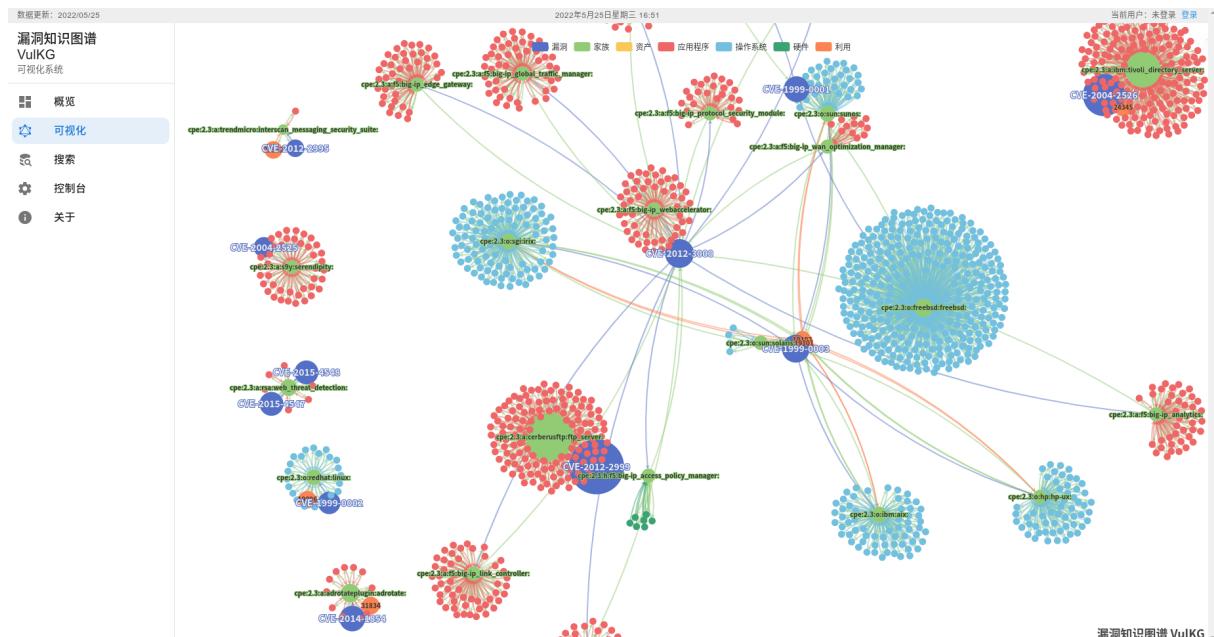


图 6-8 可视化视图概览

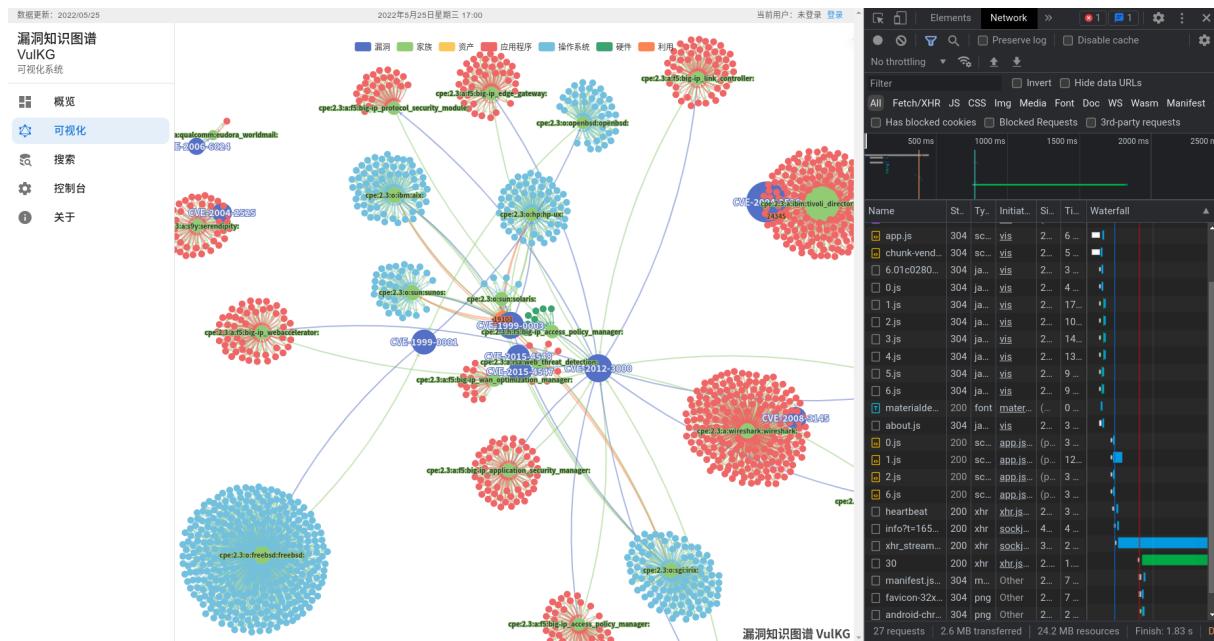


图 6-9 可视化视图加载请求瀑布流

现当前高亮元素的信息。点击结点或关系，将会弹出悬浮面板，展示结点或关系的更详细信息。

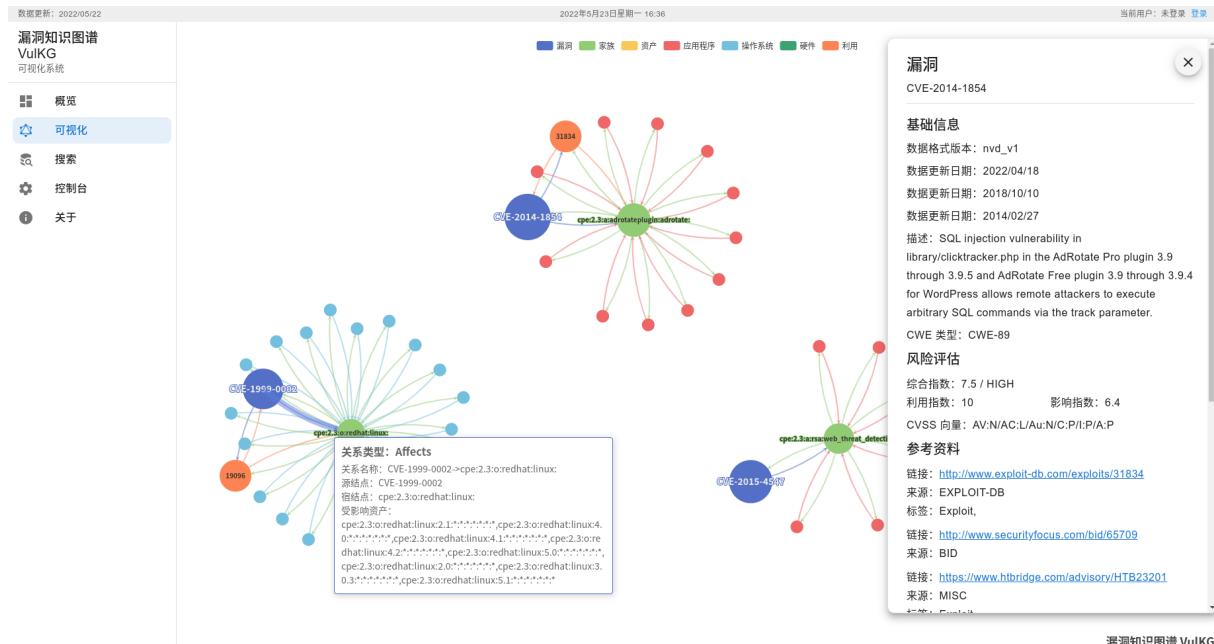


图 6-10 可视化视图 Tooltip 和弹出面板展示详细信息

6.5.3 搜索视图测试

图 6-11 所示为搜索视图。搜索视图提供一个悬浮搜索框，允许用户对知识图谱进行精确查询。输入查询关键字并按下回车将发送异步 I/O 请求，后端服务将搜索整个知

识图谱查找关键字相关信息，并将绘图数据返回前端。前端回调使用 ECharts 可视化库绘制力引导图展示数据。

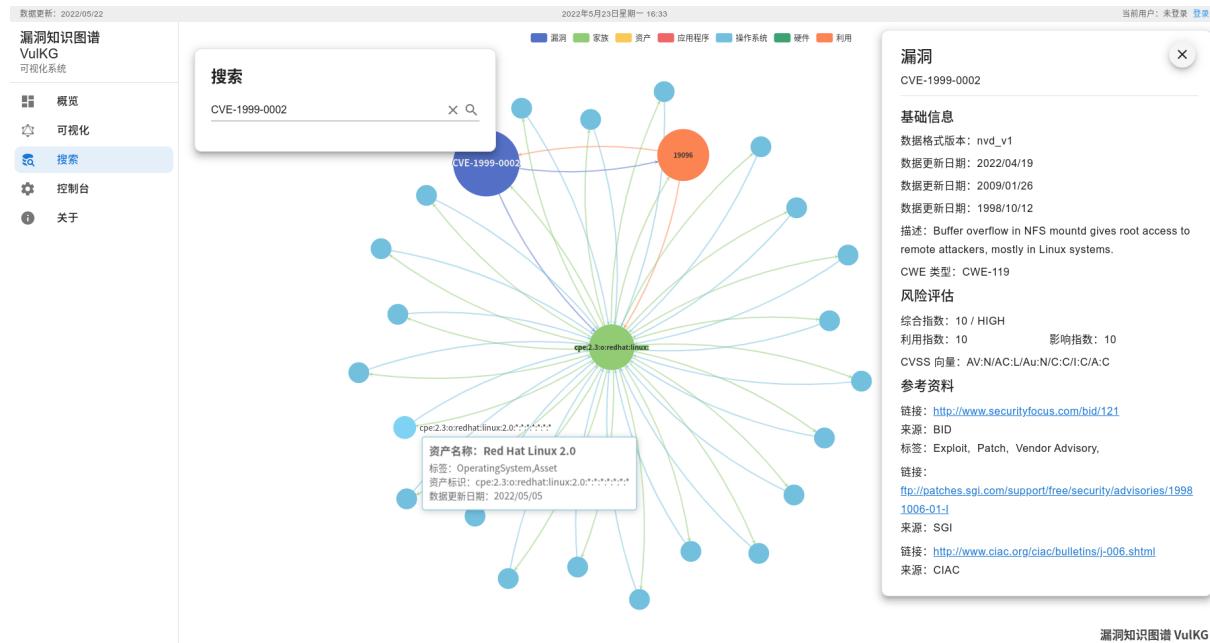


图 6-11 搜索视图使用关键字搜索，呈现结点及关系

6.5.4 关于视图测试

图 6-12 所示为关于视图。关于视图清晰地展示了本系统名称、功能、作者、使用指南等信息。



图 6-12 关于视图

第七章 结束语

7.1 项目工作总结

本文介绍了“基于漏洞知识图谱的可视化系统”的设计与实现。由于漏洞信息的收集与发布工作分散在互联网各处，且通常以纯文本的形式发布共享，这造成网络安全工作者及软硬件开发者，难以轻松直观地获知相关项目或资产的漏洞风险情况。而提供类似可视化漏洞知识图谱服务的多为商业公司、其系统不公开且收费高昂。

基于此背景，本文设计并实现了使用 GPL-3.0 自由软件许可证授权的“基于漏洞知识图谱的可视化系统”，全部代码在 GitHub¹ 开源。其由数据采集子系统、知识图谱构建子系统、持久化子系统、后端服务子系统、前端子系统五部分组成。

针对五个子系统，本文第二章分别介绍了其相关技术调研与实现理论基础，分析并确定了开发本系统采用的技术栈，主要为：Python, Scrapy, MongoDB, Neo4j, Flask, HTML, CSS, JavaScript, Vue.js, Vuetify, ECharts。

第三章针对系统数据流、功能和非功能需求进行了需求分析，细化系统开发目标。

第四章系统设计进行了系统概要设计、系统功能模块设计、系统详细设计：首先针对系统数据流分析，设计系统总体及分层结构，形成项目框架；随后划分功能模块、针对每个功能模块明确其功能与职责，并为模块绘制了类图与功能时序图。

第五章系统实现以流程图形式阐述各个子系统的设计与实现，以伪代码形式阐述关键功能模块的设计与实现。

第六章系统测试对系统运行速度与运行结果进行简单测试，以截图形式展示可视化的系统的实现效果。

本系统具备并行数据采集能力、并行知识图谱构建能力、后端 API 及 Cron 风格定时控制能力、前端数据可视化展示与交互能力。能够在无人为干预情况下定时采集数据并更新知识图谱，且每次更新花费时间可以接受（爬虫时间依网络情况而定，小于一天；知识图谱更新时间为两小时）。前端具备友好的响应式界面、清晰的图表展现知识图谱相关数据，通过力引导图实现的知识图谱可视化功能，允许用户拖拽、缩放、选中结点或关系弹出侧边栏显示详细数据。

7.2 未来工作展望

本系统借鉴微服务设计思想，将构建信息安全漏洞知识图谱任务分解成若干小任务的数据加工流。模块化设计使得系统具备强大的鲁棒性和扩展性。未来系统功能扩展主要在于智能化技术即各类机器学习及深度学习模型的引入方向。

针对数据采集子系统，未来可持续增加数据源，如 CNNVD、JVN 等；可借助搜索引擎，编写实现更智能的 Spider。例如，根据漏洞信息中提及的相关实体，使用搜索引擎

¹<https://github.com/RiddMa/KnowledgeGraph-Visualization>

擎获取搜索结果，对文本数据进行自动摘要提取信息主题，选择相关性较强的链接作为漏洞补充资料。或定期爬取技术博客、技术新闻等自然语言文本内容。

针对知识图谱构建子系统，使用深度学习模型进行命名实体识别、关系抽取、关系融合，实现实体语义去重与关系融合功能，将不同命名体系下的相同漏洞融合为同一实体。还可通过自然语言数据将部分未收录进 CPE 官方列表的资产、未被 CVE 等项目收录的漏洞作为补充加入知识图谱。

针对持久化子系统，随着知识图谱不断增大，Neo4j Community 单机版性能不足以支撑大数据量，可将知识图谱迁移至其他开源分布式图数据库如 Nebula Graph。由于本系统设计时尽量采用通用的 CQL 语句进行数据库操作而非使用 Neo4j 特定 ORM 对象，迁移工作可以较平滑进行。

针对后端服务子系统，为了应对未来可能增长的用户量，可以引入 Redis 内存数据库技术，引入 Celery 等消息队列技术，提高系统并发性能。同时，为了满足更多用户的需要，可以加入完善的用户系统，引入会话认证、权限管理等功能。为了方便管理员查看 APScheduler 服务运行状态，可在前端加入“控制台”视图，允许管理员手动控制后端微服务运行。

针对前端子系统，可以完善 UI 设计、增加更多可视化选项；优化移动端上的展示功能。