软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(3):1280−1306 [doi: 10.13328/j.cnki.jos.006813]

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

MSA-Lab: 模型驱动的微服务集成设计平台*

熊靖浏[1], 任秋蓉[1], Shmuel TYSZBEROWICZ[1,2], 刘志明[1,3], 刘 波[1]

[1](西南大学 计算机与信息科学学院 软件研究与创新中心, 重庆 400715)

[2](Software Engineering Department, Afeka Academic College of Engineering, Tel-Aviv 6998812, Israel)

[3](西北工业大学 软件学院, 陕西 西安 710129)

通信作者: 刘波, E-mail: liubocq@swu.edu.cn

摘 要: 从单体系统迁移到微服务系统是当前业界对遗留系统实施再工程化的主流选项之一, 基于单体遗留系统的微服务体系架构重构则是实现该迁移的关键步骤. 目前学界多集中在微服务识别方法的研究上; 业界虽有许多面向微服务架构的遗留系统重构的实践, 但缺乏系统性的方法及高效鲁棒的工具. 鉴于此, 在微服务识别与模型驱动开发方法前期研究的基础上, 研发一种模型驱动的、可用于单体遗留系统微服务化重构的集成设计平台 MSA-Lab. 它通过分析单体遗留系统运行日志中的方法调用序列, 对其中的类和数据表进行类型识别和聚类以构造抽象微服务, 同时生成包括微服务图和微服务序列图在内的系统架构设计模型. 它包括用于微服务自动识别与设计模型自动生成的核心部件 MSA-Generator, 以及用于微服务静态结构模型与动态行为模型可视化展现、交互式建模、模型语法约束检验的核心部件 MSA-Modeller. 在 MSA-Lab 平台上, 通过对 4 个开源项目实施有效性、鲁棒性、功能转换完备性等实验以及对 3 个同类型工具实施性能对比实验, 结果表明: 所提平台拥有很好的有效性、鲁棒性及实现面向日志的功能转换完备性, 且性能更加优越.

关键词: 微服务架构; 服务识别; 设计模型生成; 交互式建模工具; 软件设计评估

中图法分类号: TP311

中文引用格式: 熊靖浏, 任秋蓉, Shmuel TYSZBEROWICZ, 刘志明, 刘波. MSA-Lab: 模型驱动的微服务集成设计平台. 软件学报, 2024, 35(3): 1280−1306. http://www.jos.org.cn/1000-9825/6813.htm

英文引用格式: Xiong JL, Ren QR, Tyszberowicz S, Liu ZM, Liu B. MSA-Lab: Integrated Design Platform for Model-driven Development of Microservices. Ruan Jian Xue Bao/Journal of Software, 2024, 35(3): 1280−1306 (in Chinese). http://www.jos.org.cn/1000-9825/6813.htm

MSA-Lab: Integrated Design Platform for Model-driven Development of Microservices

XIONG Jing-Liu[1], REN Qiu-Rong[1], Shmuel TYSZBEROWICZ[1,2], LIU Zhi-Ming[1,3], LIU Bo[1]

[1](Center for Research and Innovation in Software Engineering, College of Computer and Information Science, Southwest University, Chongqing 400715, China)

[2](Software Engineering Department, Afeka Academic College of Engineering, Tel-Aviv 6998812, Israel)

[3](School of Software, Northwestern Polytechnical University, Xi'an 710129, China)

Abstract: Migrating from monolithic systems to microservice systems is one of the mainstream options for the industry to realize the reengineering of legacy systems, and microservice architecture refactoring based on monolithic legacy systems is the key to realizing migration. Currently, academia mainly focuses on the research on microservice identification methods, and there are many industry practices of legacy systems refactored into microservices. However, systematic approaches and efficient and robust tools are insufficient.

* 基金项目: 国家自然科学基金 (62032019, 61732019, 61872051); 西南大学国家人才建设项目 (SWU116007); 重庆市自然科学基金面上项目 (CSTB2022NSCQ-MSX0437)
收稿时间: 2022-03-03; 修改时间: 2022-06-30, 2022-09-13; 采用时间: 2022-10-11; jos 在线出版时间: 2023-05-24
CNKI 网络首发时间: 2023-05-26

Therefore, based on earlier research on microservices identification and model-driven development method, this study presents MSA-Lab, an integrated design platform for microservice refactoring of monolithic legacy systems based on the model-driven development approach. MSA-Lab analyzes the method call sequence in the running log of the monolithic legacy system, identifies and clusters classes and data tables for constructing abstract microservices, and generates a system architecture design model including the microservice diagram and microservice sequence diagram. The model has two core components: MSA-Generator for automatic microservice identification and design model generation and MSA-Modeller for visualization, interactive modeling, and model syntax constraint checking of microservice static structure and dynamic behavior models. This study conducts experiments in the MSA-Lab platform for effectiveness, robustness, and function transformation completeness on four open-source projects and carries out performance comparison experiments with three same-type tools. The results show that the platform has excellent effectiveness and robustness, function transform completeness for running logs, and superior performance.

Key words: microservice architecture; service identification; design model generation; interactive modeling tool; software design evaluation

近年来, 微服务架构 (microservice architecture, MSA) 因其在模块化设计、故障隔离、技术异构性支持、独立开发部署等方面的优势, 成为替代传统单体架构 (monolithic architecture) 用以开发和运维复杂软件系统的主流选择[1]. 诸多重要的互联网企业如 Alibaba、Netflix、Amazon 等工业界大量业务服务系统的拥有者已经或正在将单体架构系统迁移到微服务架构系统. 微服务架构的核心思路是将系统构造为一系列适当粒度的、高内聚低耦合的、分布式自治的功能单元 (即微服务)[2]. 各微服务实现相对单一的业务功能; 它们边界清晰并运行在专属的进程中; 它们能被独立设计、开发、部署和运维, 并通过定义良好的接口以轻量级的通信方式 (如 RESTful style) 交互和协作[3]. 其中, 如何从单体架构迁移到微服务架构是工业界和学术界共同关心的问题[4].

在工业实践中, 除建造全新的微服务架构系统, 更多决策者采取了对遗留系统实施微服务化再工程的策略, 而基于遗留系统现有的软件制品 (如代码、日志) 实施微服务体系架构重构则尤为重要[5]. 微服务体系架构重构设计包含两个方面: (1) 系统分解与微服务识别. (2) 微服务体系架构的结构与行为特征刻画. 微服务识别的基本思想来自于传统的系统构件分解[6]与组合[7]设计, 即: 按照高内聚低耦合[8]的原则及其他非功能或服务质量 (quality of service, QoS) 属性[9], 对关联紧密的功能元素 (如类) 进行聚类. 而对微服务体系架构的静态结构和动态行为进行建模, 则是模型驱动开发方法在软件系统设计阶段的主要工作. 所谓模型驱动开发 (model-driven development, MDD) 即使用模型作为软件系统开发各阶段的关键输入和核心产出, 通过在各阶段针对不同关注点的模型的构建、转换和精化来严格地构造软件系统, 并可借助模型检验等形式化方法来确保软件系统的正确性[10,11]. MDD 被证明能成功地应用于传统的软件密集型系统开发[12]以及全新构造微服务系统的研发中[12−15]. 然而, 工业界有许多事实上正在运行的微服务系统虽是由遗留系统重构而来, 但其再工程化过程缺乏系统性方法的指导及高效工具的支持, 以致其微服务化重构设计主要依靠设计师的个人经验, 并多采用试错法完成, 其方法易错、低效且不可重复.

学术界对微服务体系架构重构设计的工作主要集中在系统分解和微服务识别方面. 相关工作尚未完全解决好以下问题: (1) 通过聚类能明确系统分解后的微服务集合及每个微服务所含类的集合, 但微服务内外部结构特征并未明确. (2) 许多基于运行日志的解析工具能根据方法调用自动构造出对象行为模型 (对象序列图), 但缺乏支持对象行为模型向微服务行为模型转换和精化的工具. 显然, 仅明确微服务内部所含类的集合, 但缺少相应结构和行为模型的微服务体系架构设计方案是不完整的. (3) 除了极少量研究工作[16], 绝大部分微服务识别的研究工作要么需要人工选择微服务数量, 要么生成微服务重构设计方案后便无法进行交互式修改, 导致任何设计变更都必须重新执行一遍其给出的重构方法. 此外, 即便是支持交互式微服务设计的工作, 也仅是在可视化外观层面提供微服务静态结构的调节, 没有系统性的设计模型自动重构机制及动态评估机制与之适配.

显然, 学术界和工业界均需要一种能够面向遗留系统的、高效且鲁棒地执行微服务体系架构重构设计的方法和支持工具. 鉴于此, 我们在前期工作[17]完成微服务体系架构设计模型相关元模型定义的基础上, 进一步研发了一个模型驱动的、可用于单体遗留系统微服务化重构的集成设计平台 MSA-Lab. 我们的主要贡献如下: (1) 研发了一个可用于微服务自动识别与设计模型自动生成的核心部件 MSA-Generator; 它可通过对单体遗留系统日志中执行轨迹 (execution traces) 的扫描, 自动完成类和数据表的类型识别和聚类以构造抽象微服务; 在此基础上根据

对象间的方法调用序列, 自动解析得到微服务内外部的静态结构模型及动态行为模型. (2) 研发了一个可支持微服务静态结构模型 (微服务图) 与动态行为模型 (微服务序列图) 可视化展现、交互式建模、模型语法约束检验的核心部件 MSA-Modeller; 它可载入 MSA-Generator 中自动生成的微服务设计模型, 在集成设计环境中可视化展现并供开发者按需进行二次建模; 并对二次建模结果, 提供基于对象约束语言 (object constraint language, OCL) 的语法检验和基于通用微服务划分合理性指标[8,16,18,19]的量化评估.

本文第 1 节综述相关工作. 第 2 节介绍 MSA-Lab 平台技术框架. 第 3 节和第 4 节分别介绍 MSA-Lab 核心部件 MSA-Generator 和 MSA-Modeller 的工作原理. 第 5 节是工具的评估实验及实验结果讨论. 第 6 节总结全文并介绍未来的工作.

1 相关研究综述

软件重构是软件工程领域的重要研究与实践方向之一[6], 将传统的单体架构重构为 MSA 已引起学术界和工业界的广泛关注. 一些技术框架和原型工具也在不断推出, 我们将其划分为: (1) 基于系统分解与聚类方法的微服务设计工具; (2) 基于模型驱动的微服务设计工具. 下文分别对相关工作进行综述.

1.1 基于系统分解与聚类方法的微服务设计工具

能否高效和自动的帮助工程师进行系统拆分和微服务识别是工具和框架的一个重要特点, 故我们对这类工具按自动化支持程度进行划分. 早期大多数工具仅支持半自动的微服务设计. 例如: Levcovitz 等人[20]根据代码静态依赖图, 实施自底向上的子系统划分与聚类, 并通过数据表的人工识别完成微服务的构造设计. Chen 等人[21]提出一种半自动框架: 即首先人工分析业务需求并绘制数据流图, 继而将具有相同输出数据的操作和数据划分为一个微服务. 类似的工作还有 Li 等人[22]提出的数据流驱动的半自动分解框架. 这些半自动的框架对大型系统改造而言, 时间成本是难以承受的.

近年来, 也有学者提出了一些全自动工具和框架. 例如: Jin 等人[8]提出的框架将在同一方法执行路径上一起出现的类认为是一组功能相关单元, 按此思想通过定义评估函数将功能相关的类聚集成微服务; 但他们所提出的框架手工配置复杂 (服务生成数量、算法迭代次数等重要参数需要手工设置), 并且没有对数据库进行划分. 李杉杉等人[23]提出了一个基于优化的数据流驱动的微服务拆分工具, 采用两阶段的聚类算法来对数据表和类实现微服务拆分; 但他们的工具目前只支持服务拆分的核心功能, 并且参数设置需要工程师的先验知识. Kalia 等人[16]根据遗留系统的静态和运行时信息提出了一个基于 AI 技术的拆分设计工具, 最后得出微服务所包含的类的集合; 但他们忽略掉了对数据表的拆分且也需事先预设服务生成数量. Abdullah 等人[24]使用系统访问日志和无监督的机器学习方法将具有相似性能和资源要求的 URL 组作为一个微服务, 但基于 URL 拆分的结果可能存在大量的代码重复, 不方便后面维护. Mazlami 等人[19]将数据库中的表当作普通的类来进行划分解决数据库的划分问题, 但没有考虑微服务数据的共享或划分问题. Santos 等人[18]提出了一系列指标来组合衡量微服务拆分成本, 以可视化的形式给出了类的拆分结果, 但他们没有对持久层的类进行划分.

不过, 大多数现有方法仅关注如何从遗留系统中获取信息以生成微服务划分结果这一过程. 其方法和工具得到的设计产出, 仅可回答每个微服务由哪些功能单元 (例如类) 组成, 却往往缺乏对可指导微服务系统后续开发工作的 MSA 静态结构及动态行为特征的刻画. Zhang 等人[9]使用功能信息和性能信息对遗留系统进行拆分, 给出的拆分结果只包括包含类名的结果. Jin 等人[8]给出的拆分结果也只包括类名, 但将类分为了接口类和业务类. Kalia 等人[16]给出了拆分结果的可视化展现, 其中刻画了类与类的调用链路, 但没有给出微服务重构后微服务之间的结构及行为特征的描述.

此外, 大多数研究中的工具和框架给出的拆分结果一旦生成, 就不允许工程师再进行手动的修改. 工业实践表明: 有经验的工程师对系统实施的交互式设计工作往往能优化和提升自动构造的微服务设计方案. 在支持交互式设计调整方面, 丁丹等人[25]提出的 MSDecomposer 可将拆分过程可视化, 并能够手动调整数据表的归属. 李杉杉等人[23]的工具可以手动将类的归属改在另一个微服务下. 但这些交互式设计调整仅限于对各微服务所含功能单

元 (类) 的归属进行修改, 尚不涉及微服务架构模型 (包括静态结构模型及动态行为模型) 的调整. 同时, 作为支持微服务架构交互式设计的工具, 还需要相应的动态语法检查与指标评估机制, 以便对调整后得到的不同划分方案给出语法正确性检验和基于通用合理性指标的评估数据对比.

1.2 基于模型驱动的微服务设计工具

与基于系统分解与聚类方法的微服务设计方法不同, 基于模型驱动的微服务设计方法从用户需求开始, 对微服务静态结构和动态行为进行严格的刻画; 并利用模型精化和模型转换构造不同层级、不同视角下的设计模型, 直至获得微服务程序 (程序代码本身也是严格的模型); 同时, 基于模型检验与分析的工具也可用于保障微服务模型的一致性、正确性等性质. 近年来, 模型驱动的 MSA 系统设计方法研究已引起广泛关注[12,15,26], 但主要用于全新微服务系统的构建. 而单体遗留系统的模型驱动 MSA 重构设计方法和工具研究仍处于初级阶段.

在面向全新微服务系统模型驱动开发的研究工作中, 基于领域驱动设计 (domain-driven design, DDD) 的模型驱动方法最具代表性[27]. Gysel 等人[13]提出了一个能基于 16 个标准, 从用户输入文件进行微服务拆分的工具 Service Cutter. 每个微服务由数据、操作和工件 (操作在数据上作用后产生的结果) 组成. 在基于 Service Cutter 的基础上, Kapferer 等人还提出了一种基于 DDD 的微服务拆分工具 Context Mapper[15], 其微服务模型由有界上下文表示, 并使用上下文关系和对称关系来描述有界上下文之间的关系. 但这两个工具要求用户编写所有的输入文件, 时间成本高, 不适用于大型系统改造. Rademacher 等人[26]使用 UML profile 来进行领域驱动设计里的概念建模. Profile 中主要包含聚合根、实体、仓库和服务等元素. Schneider 等人[28]和 Petrasch 等人[14]也同样使用 UML profile 来扩展领域驱动设计的建模元素, 并开发了手工图形化建模工具. 但他们的 profile 中没有定义元模型的约束条件, 以便检查模型语法避免模型误用.

此外, 还有许多非 DDD 系列的建模工具和框架也被提出. Sorgalla 等人[12]提出了名为 AjiL 的手工图形化微服务建模工具, 将每个微服务定义为实现业务能力的功能微服务和基础设施服务, 微服务拥有接口, 实体端口等属性. 这些模型是与 Spring Cloud 相关的, 工具可以根据这些模型和其对应的属性生成平台相关的 Spring Cloud 代码, 这种平台相关性模型难以进行迁移和复用. Terzić等人[29]提出了用于对遵循 RESTful 风格的微服务领域建模语言, 使用户通过提供图像和文字输入来生成微服务代码. 但这些模型往往只局限在微服务的结构和微服务之间的关系, 忽视了对微服务的行为的描述. Tyszberowicz 等人[30]基于模型从自然语言需求中提取系统功能来进行微服务拆分, 但这种框架对需求的要求高, 结果由系统功能对应的文字组成, 不好对应代码实现.

支持遗留系统向 MSA 系统迁移的模型驱动工具尚缺较为成熟工作. Escobar 等人[31]提出了一个以 Java 企业版应用模型为中心的流程来分析可视化业务层和数据层之间的当前结构和依赖关系, 并以图像的结果给出微服务拆分结果, 但这种图像只是一种辅助理解的可视化展现, 不能支持模型驱动的开发过程.

总之, 现有的工具和框架尚不能提供一种高效鲁棒的集成设计平台, 以便对遗留系统进行解析并自动生成包含微服务识别及微服务系统架构模型在内的设计方案; 且支持设计方案的可视化展现需通过图形用户接口 (GUI) 进行交互式调整, 而不必重新运行自动识别与模型生成算法, 也无需事先指定微服务数量等“超参数”.

2 MSA-Lab 的技术框架

MSA-Lab 的当前版本致力于提供一种模型驱动的、可用于单体遗留系统微服务化重构的集成设计平台, 其技术框架如图 1 所示. 它包含两个用于 MSA 结构与行为建模的微服务元模型, 即: 微服务图元模型 (meta-model for microservice diagram) 和微服务序列图元模型 (meta-model for microservice sequence diagram); 以及两个用于微服务及设计模型自动生成与交互式建模的核心功能部件, 即: 微服务及设计模型生成器 (MSA-Generator) 和微服务建模工具 (MSA-Modeller). 微服务元模型及核心功能部件分别从建模元素和建模工具两方面支持构建微服务体系架构的两种设计模型, 即: 刻画微服务内外部结构 (intra- and inter-structure of microservices) 特征的微服务图 (microservice diagram) 和描述微服务间交互行为的微服务序列图 (microservice sequence diagram).

微服务元模型旨在定义可用于刻画微服务体系架构两种设计模型的建模元素及模型约束. 相似地, 例如为构

建表达面向对象方法中概念模型的类图, 需定义类 (含类名, 属性, 方法定义) 和关系 (含类型, 角色, 重数等属性的定义) 等建模元素的元模型; 事实上 UML 标准规范已提供这些元模型的标准定义. 但对于微服务这类新型系统构件而言, UML 规范尚无标准元模型予以支持. 故在前期工作[17]中, 我们利用 profile 机制扩展定义了适用于刻画微服务体系架构静态结构和动态行为模型的元模型. 其本质是通过对标准 UML 类型的继承、聚合和关联等操作来定义和添加新的构造型 (stereotype); 新的构造型即自定义的建模元素, 它包含建模元素相应的图形表示、附加属性, 及利用对象约束语言 (OCL) 等方法定义的语法约束 (以避免构造型的误用). MSA-Lab 当前版本支持的两种微服务元模型所定义的新构造型如下.

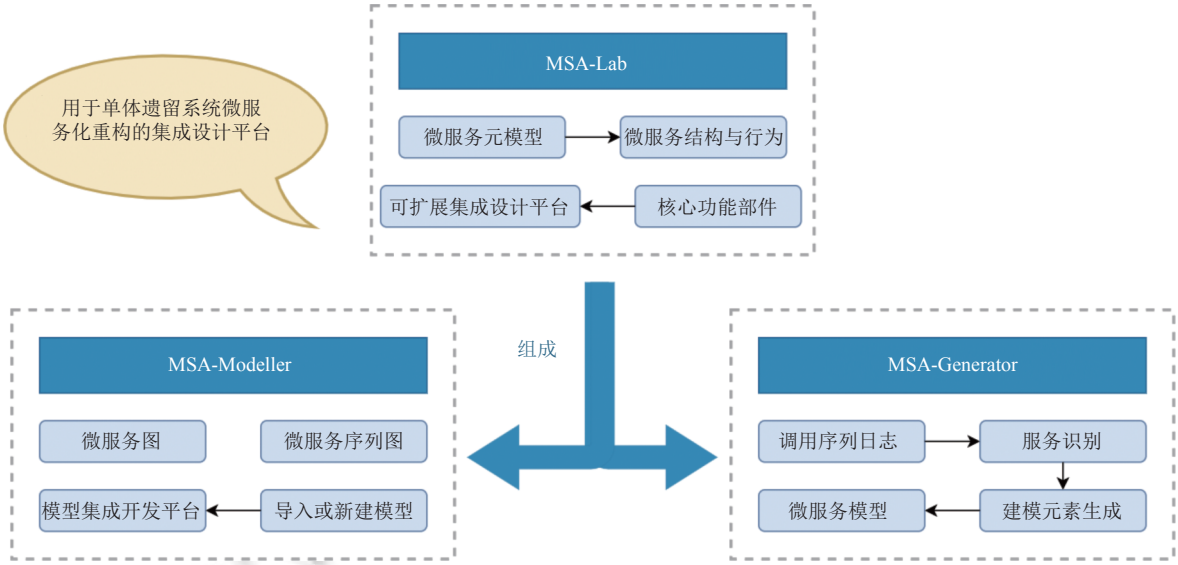


图 1 MSA-Lab 平台的技术框架

(1) 微服务图元模型 (meta-model for microservice diagram): 定义了构造微服务图 (microservice diagram) 所需的各种新构造型 (stereotypes) 及 OCL 语法约束 (如图 2 所示). 这些构造型 (stereotypes) 包括: Microservice (微服务)、Controller Class (CC, 控制类)、General Subordinate Class (GSC, 一般附属类)、Data Access Class (DAC, 数据访问类)、Data Entity Class (DEC, 数据实体类)、MSInterface (Interfaces, 微服务接口). 利用这些新构造型建立的微服务图的具体案例可见本文第 4.2 节.

(2) 微服务序列图元模型 (meta-model for microservice sequence diagram): 定义了构造微服务序列图 (microservice sequence diagram) 所需的各种新构造型 (stereotypes) 及 OCL 语法约束 (如图 3 所示). 这些构造型 (stereotypes) 包括: MSInteraction 和 MSMessage. 其中, MSInteraction 由 UML 元模型 Interaction (序列图) 继承而来, 我们添加新的 3 个标注 (MSs 代表此场景下所参与的微服务; traceNum 代表当前交互所对应的运行日志中的 Trace; actors 代表此场景下所参与的角色, 其 OCL 约束定义分别是 MSs_Constraint, TraceNum_Constraint 和 Actor_Constraint), 用来验证微服务序列图是否满足规范, 同时验证对象序列图向微服务序列图转换的过程中是否出错. MSMessage 则是由 UML 元模型 Message 继承而来; 为其新添加的标注为 Interface, 用来记录此消息对应的微服务接口; 其 OCL 约束 Interface_Constraint 限制每个 MSMessage 只有一个 MSInterface 与之对应. 利用这些新构造型建立的微服务序列图的具体案例可见本文第 4.2 节.

MSA-Lab 当前版本提供的用于微服务及设计模型自动生成与交互式建模的核心部件如下.

(1) 微服务及设计模型生成器 (MSA-Generator): 支持微服务自动识别与设计模型自动生成的核心部件. 它可从遗留系统运行日志中提取对象类型信息及交互信息, 自动完成类和数据表的识别和聚类, 以构造抽象微服务; 在此基础上根据对象间的方法调用序列, 自动解析得到包括微服务内外部静态结构模型及动态行为模型在内的微服

务设计模型.

(2) 微服务建模工具 (MSA-Modeller): 支持可视化展现与交互式建模的核心部件. 它可载入自动生成的微服务设计模型, 在集成设计环境中可视化展现并供开发者按需进行二次建模; 并对二次建模结果提供基于 OCL 的语法检验和基于通用微服务划分合理性指标的量化评估反馈.

我们将在接下来的第 3 节和第 4 节分别详述以上两个核心部件.

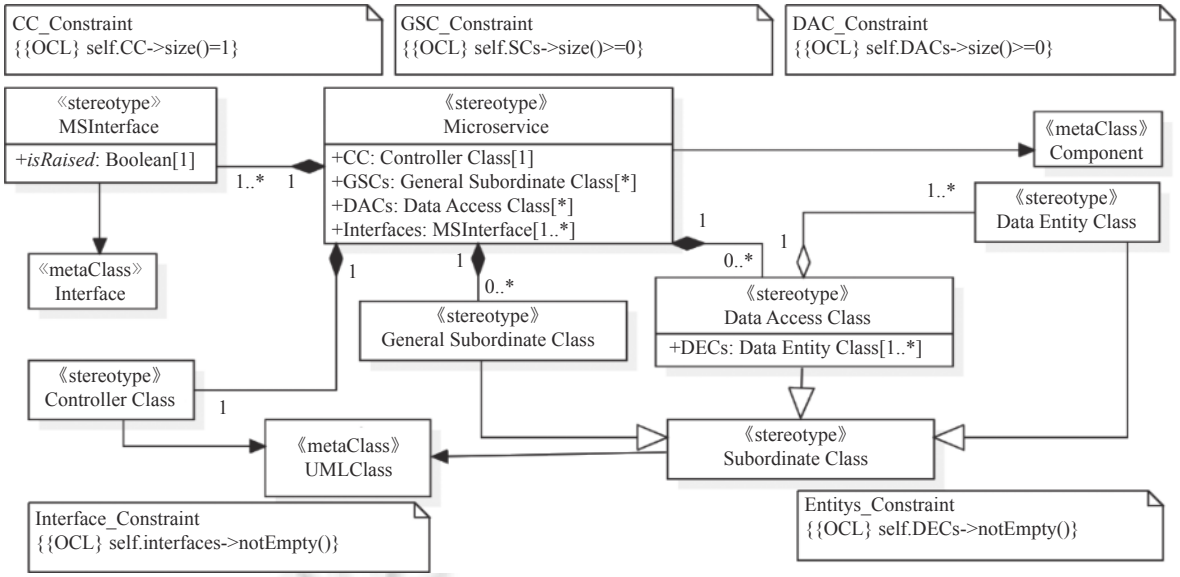


图 2 微服务图的元模型与约束

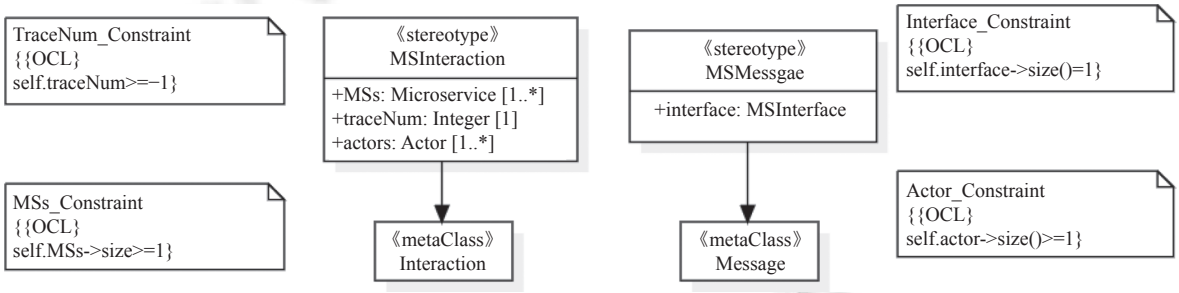


图 3 微服务序列图的元模型与约束

3 MSA-Generator

MSA-Generator 是一个 Web 工具, 它以单体遗留系统的方法调用日志为输入, 进行微服务识别与设计模型自动生成, 并提供设计方案的量化评估.

MSA-Generator 采用前后端分离的方式开发, 前端采用 Vue 开发, 后端采用 Java 和 Golang 开发. 其中 Java 开发的后端主要负责生成供 MSA-Modeller 导入的微服务模型对应文件. Golang 开发的后端主要负责服务识别、设计评估与 MSA-Modeller 前端建模相关的初级建模功能, 以及将生成模型文件所需的数据传给 Java 后端. MSA-Modeller 是一个微服务模型的集成开发环境, 在第 4 节会对其进行介绍.

图 4 是 MSA-Lab 平台进行微服务重构的工作原理和流程, 可分为 5 个部分: (1) 功能单元获取, 对收集的日志进行分析, 提取用于聚类的功能原子. (2) 调用关系分析, 对日志中的调用关系进行分析, 得到关系矩阵. (3) 抽象微

服务识别, 使用不同聚类函数对功能原子进行聚类, 并识别出抽象微服务. (4) 微服务模型与重构设计生成, 生成微服务重构设计和对应的微服务模型. 在这个过程中, 开发者可根据对设计的评价结果以及业务需求对微服务设计进行调整, 工具可根据调整结果生成新的设计与模型. (5) 微服务模型使用, 将从 MSA-Generator 中自动生成的模型文件导入进 MSA-Modeller, 在集成环境下进行模型驱动的开发.

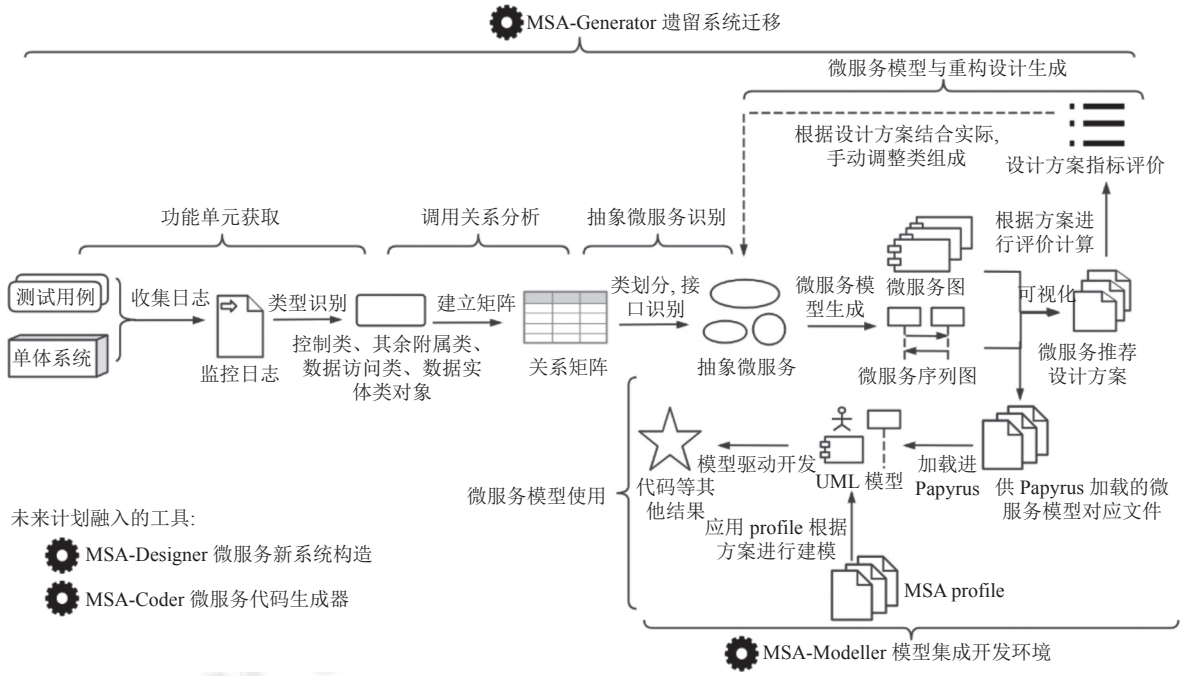


图 4 MSA-Lab 中的微服务重构流程

我们将使用 JPetStore [32]作为辅助案例来介绍工具的工作原理; JPetStore 是一个单体架构的小型宠物商店系统, 包括账户管理, 宠物品种管理, 购物车管理, 订单管理等用例. 本节接下来的部分将介绍 MSA-Generator 的主要功能: 抽象微服务识别和微服务模型生成; 并讨论微服务识别结果评估指标的设计和使用.

3.1 抽象微服务生成

在获得微服务模型之前, 我们需要获取抽象微服务, 这部分的过程对应图 4 中的前 3 步: (1) 对遗留系统进行日志的收集, 分析其中的方法调用序列, 识别对象及其对应类的所属类型, 得到组成抽象微服务的功能单元 (被识别了类型的类). (2) 通过分析对象所对应的类之间的关系, 得到代表功能单元关系的关系矩阵. (3) 根据功能单元的关系矩阵及接口复杂度、用例专注度信息, 使用改进的 NSGA-II 算法 [33]对功能单元进行聚类, 得到每个微服务的类组成. 通过类组成进一步识别微服务接口, 最后得到抽象微服务. 抽象微服务包括一个微服务的所有类和接口, 并识别了每个类的类型, 是生成微服务图和微服务序列图的关键.

3.1.1 功能单元获取

我们通过收集测试用例并在遗留系统上运行, 得到方法调用序列作为工具平台的输入, 我们选用 Kieker 来收集所需要的方法调用序列. Kieker 是一个开源的执行日志收集和分析工具[34], 它能够对 Java 系统通过动态代理的方式, 得到代码执行路径, 并将对应的方法调用序列写入日志当中. 在本文中, 我们使用 Kieker 对遗留 Java Web 系统进行日志的收集. Web 系统通常由于其业务的迭代频繁而难以维护和扩展, 具有代表性. 我们选取不同大小、不同领域的 Web 系统进行实验, 以验证我们方法的鲁棒性. 对于其他语言的遗留系统, 可以使用 SkyWalking[35], Pinpoint[36]等不同的工具进行收集. 我们的工具平台运行所需要的遗留系统信息, 是包含数据库调用信息的方法序

列, 这些方法调用序列是由开发者提供的. 根据本文以 Kieker 收集日志并进行分析的思想, 也可迁移至不同语言系统的日志进行处理.

Kieker 的标准版本 (v1.13) 不能跟踪方法调用与数据库的交互. 因此我们改进了其运行时插入到被代理方法前后的探针 (Probe), 通过提取被代理方法中直接调用的 SQL 语句来支持这一点. Kieker 生成的执行日志使用 Trace 作为基本单元, Trace 包含由 Web 请求 (例如, 单击网页中的一个按钮) 触发的方法调用序列. 方法调用序列中的方法调用分为以下 3 种类: (1) 构造方法 (创建新对象); (2) 持久层方法 (对数据实体执行 CRUD 操作); (3) 通用方法 (表示所有不同于以上两种的其他方法). 每个方法调用都由一对 Line 表示: 一个开始 Line 和一个结束 Line. Line 的属性在表 1 中描述. 特别的, 构造方法中的属性类型可以是 BeforeConstructorEvent 或 AfterConstructorEvent; 持久化方法可以是 BeforeOperationEventWithSql 或 AfterOperationEventWithSql; 通用方法可以是 BeforeOperationEvent 或 AfterOperationEvent.

表 1 日志行结构

名字	描述
type	BeforeOperationEvent, AfterOperationEvent, BeforeConstructorEvent, AfterConstructorEvent BeforeOperationEventWithSql, AfterOperationEventWithSql
generation time	该行生成的时间
trace id	用以标识唯一请求的全局唯一ID
order id	Trace中的顺序ID
method identifier	被调用方法的完备签名
class identifier	被调用类的完备签名
object id	用以标识运行时对象的唯一ID
table name	持久层方法特有属性, 标识SQL语句中所涉及的数据表名

为了提取功能单元和进一步设计模型生成所需的信息, 我们将日志中的每个方法调用重新组织为 MessageRecord, 全部的调用关系被表达为一个 MessageRecord 的集合, 记作{MessageRecords}, MessageRecord 被描述为: MessageRecord =<SenderObject, ReceiverObject, MethodSig, MethodType>, 其中,

• SenderObject =<OID, ClassSig>, 是调用关系之中, 调用发出对象的信息, 包括对象唯一 ID 和对象所对应的类签名.

• ReceiverObject =<OID, ClassSig>, 是调用关系之中, 调用接收对象的信息, 包括对象唯一 ID 和对象所对应的类签名.

• MethodSig, 是调用关系之中, 调用接收对象的被调用的方法签名, 其中构造方法的签名统一为<init> 函数.

• MethodType, 是被调用方法的类型, 分别是 CREATE, UPDATE, READ, DAOCALL 代表着创造对象的构造方法, 改变对象状态的更新方法, 不改变对象状态的读方法以及使用数据访问对象的方法. 通过 MethodSig 的关键字, 能够确定前 3 种方法的类型. 例如 CREATE 类型的方法签名是<init>, UPDATE 类型和 READ 类型的方法签名通常包含 set 和 get 关键字, DAOCALL 类型则由方法调用是否直接触发 SQL 语句执行进行识别.

基于{MessageRecords}, 我们设计了一种算法 1, 将对象自动划分为 4 种类型: 控制对象、一般附属对象、数据访问对象和数据实体对象. 这些对象所属的类与第 2 节中的控制类、一般附属类、数据访问类、数据实体类等构造型一一对应, 他们所属的类被称为功能单元. 算法 1 中自动识别的检查指标是按照类对应对象的特征来进行识别, 并能够对系统中所有类进行类型划分: 控制对象作为模块和外部通讯的唯一出入口, 是一个永久对象[37]. 意味它在执行测试用例前已经存在, 同时不在执行中被销毁; 数据访问对象下的方法会直接调用 SQL 语句; 数据实体对象对应 SQL 语句中的数据表; 一般附属对象为所有对象集合除去以上 3 种对象的补集. 通过算法 1, 能够对系统中的所有类进行类型划分. 这些识别出类型的类, 作为功能原子用于后续的微服务聚类当中.

算法 1. 对象类型识别算法.

输入: MessageRecord 的集合{MessageRecords}; 记录每个 DAO 能够访问的所有 DEO 的 Map DAO2Tables (一个 DEO 对应着数据库中的一个 Table. Tables 为数据表的集合, 在预处理 Trace 时获得. 如果日志行结构 Line 的 table name 属性不为空, 则将 Line 对应方法所属对象为键, table name 下的所有数据表为值添加或修改进 Map 中);
输出: 4 种类型的对象集合.

```
1. Function IdentifyObjectType({MessageRecords}, DAO2Tables)
2. {OBJECTs} //记录所有对象的集合
3. {COs} //控制对象集合
4. {GSOs} //一般附属对象集合
5. {DAOs} //数据访问对象集合
6. {DEOs} //数据实体对象集合
7. For MessageRecord in {MessageRecords}
8.    Add MessageRecord.SenderObject and MessageRecord.ReceiverObject to {OBJECTs}
9.    If MessageRecord.MethodType == CREATE
10.       //控制对象的特征: 此对象是单例模式; 向其他对象发送过消息; 被与系统业务无关的框架类创建;
11.       //根据控制对象特征, 检测 Object 是否符合控制对象的规范 Check_Controller (Object)
12.       If Check_Controller (MessageRecord.receiverObject)
13.          Add MessageRecord.receiverObject to {COs}
14.       End if
15.    Else MessageRecord.MethodType == DAOCALL //如果方法类型是 DAOCALL, 调用了 SQL 语句
16.       Add MessageRecord.receiverObject to {DAOs} //方法所属对象是 DAO, 加入{DAOs}
17.       //根据 DAO2Tables 查找当前 DAO 能够访问的所有 Tables, 将所有 Table 转换为对象, 添加进{DEOs}
18.       Add DAO2Tables.get(MessageRecord.receiverObject) to {DEOs}
19.    End if
20. End for
21. //{GSOs}为{OBJECTs}与{COs}、{DAOs}和{DEOs}的补集合
22. {GSOs} = {OBJECTs}∁({COs} ∪ {DAOs} ∪ {DEOs})
23. Return {COs}, {GSOs}, {DAOs}, {DEOs}
```

3.1.2 调用关系分析

这一步计算对象所对应的类之间的关系. 建立控制类与一般附属类之间的关系矩阵—业务矩阵 (控制类与一般附属类都是业务类, 执行业务逻辑但不进行数据存储), 以及数据访问类与业务类之间的关系矩阵—数据访问类使用矩阵. 在后续会使用业务矩阵对业务类进行聚类, 得到微服务的业务类. 最后根据每个微服务的业务类组成, 使用数据访问类使用矩阵得到微服务数据存储相关的类.

控制类与一般附属类决定了微服务接口以及具体的业务逻辑, 在每个微服务的业务逻辑确立后, 才能进行数据相关的类的划分. 我们使用{MessageRecords}从功能联系 (类与类之间的具体调用关系, 体现为一个类对另一个类进行了多少次什么样的操作) 的角度度量控制类和一般附属类之间的关系. 建立控制类和一般附属类的关系矩阵 (业务矩阵, 如表 2), 并称控制类和一般附属类为业务类 (Business Class).

R_{ij} 是对第 i 个控制类和第 j 个一般附属类之间函数关系的量化, 定量公式定义如下:

$$R_{ij} = IStr_c \times (IFre_c)^{kc} + IStr_u \times (IFre_u)^{ku} + IStr_r \times (IFre_r)^{kr} \tag{1}$$

$IStr_c$, $IStr_u$, $IStr_r$ 常量被用于定义不同类型方法的调用强度, 构成调用强度常量组 (InvocStren). 定义 InvocStren

的原因是不同类型的方法对对象之间的度量关系应该有不同的影响, 一个对象创建另一个对象比读取另一个对象的关系强度要强上许多, 需要区分如下 3 个常量.

- $IStr_c$, CREATE 类型方法的强度常量, 两个对象之间的最强的关联, 应该对度量关系有着最大的影响.
- $IStr_u$, UPDATE 类型方法的强度常量, 代表的关联其次.
- $IStr_r$, READ 类型方法的强度常量, 代表的关联最弱.

$IFre_c$, $IFre_u$, $IFre_r$ 常量构成调用频度常量组 (InvocFreq), 这 3 个常量用以定义不同类型方法的调用频度. 定义 InvocFreq 的原因是随着调用数量的上升, 调用数量应该越容易占据主导地位. 对象之间的频繁读操作也应该尽量放在一个微服务内, 而不是跨服务读取数据提高通信代价, 需要区分如下 3 个常量.

- $IFre_c$, CREATE 类型方法的频度常量.
- $IFre_u$, UPDATE 类型方法的频度常量.
- $IFre_r$, READ 类型方法的频度常量.

表 2 业务矩阵

控制类	一般附属类				
	GSC_1	GSC_2	GSC_3	GSC_4	…
CC_1	R_{11}	R_{12}	R_{13}	R_{14}	…
CC_2	R_{21}	R_{22}	R_{23}	R_{24}	…
CC_3	R_{31}	R_{32}	R_{33}	R_{34}	…
…	…	…	…	…	…

在 InvocStren 和 InvocFreq 中, 常数值的定义依据如下原则.

(1) 在量化过程之中, 调用强度相比调用频度应该占据主导地位. 因此, CREATE 方法的调用强度 $IStr_c$ 应该被设置为高于 UPDATE 方法的调用强度 $IStr_u$ 一个数量级的值. $IStr_u$ 和 $IStr_r$ 之间的关系也遵循该理念 (本文实验设置为 $IStr_c = 100$, $IStr_u = 10$, $IStr_r = 1$, 设置依据见第 5.1 节参数设置).

(2) 尽管调用强度常量在关系度量函数中占主导地位, 但调用数量达到一定多时, 调用频度的影响也需要有机会成为主要因素. 调用频度常量都被设置为稍微大于 1.0 的值, 并且 $IFre_c > IFre_u > IFre_r$ (本文实验设置为 1.03, 1.02 和 1.01, 设置依据见第 5.1 节参数设置).

最后, 由 kc、ku 和 kr 组成调用数量参数组, 分别是第 i 个控制类实例化的对象与第 j 个一般附属类实例化的所有对象之间存在的 CREATE 方法、UPDATE 方法的数量和 READ 方法的数量.

在业务矩阵建立后, 我们接着分析数据访问类与业务类之间的关系. 我们建立数据访问类使用矩阵, 以此统计每个业务类所实例化的所有对象对不同的数据访问类下的所有对象的使用次数, 如表 3 所示. 其中, S_{ij} 代表 {MessageRecords}中 BC_i 实例化的所有对象调用 DAC_j 实例化的所有对象的次数.

表 3 数据访问类使用矩阵

数据访问类	业务类				
	BC_1	BC_2	BC_3	BC_4	…
DAC_1	S_{11}	S_{12}	S_{13}	S_{14}	…
DAC_2	S_{21}	S_{22}	S_{23}	S_{24}	…
DAC_3	S_{31}	S_{32}	S_{33}	S_{34}	…
…	…	…	…	…	…

3.1.3 抽象微服务识别

一个抽象微服务定义为 $microservice_i =< CC_i, \{GSCs\}_i, \{DACs\}_i, \{DECs\}_i, \{Is\}_i >$, 其中, $i \in [1, |\{CC_s\}|]$, 上界$|\{CC_s\}|$表示生成的抽象微服务的数量应该等于控制类的数量.

我们首先使用基于 NSGA-II[33]改进的遗传算法将所有控制类与一般附属类聚集成$|\{CCs\}|$个业务集群. 一个业

务集群 (*BusinessCluster*) 对应着一个抽象微服务下的控制类和一般附属类, 负责完成每个抽象微服务的业务逻辑, 但不进行数据的存储. 所有业务集群称为业务集群候选集合 ({*BusinessClusters*}).

针对一个业务集群候选集合, 一组目标函数被提出以评估该集合, 它们分别是功能联系 (类与类之间的具体调用关系, 体现为一个类对另一个类进行了多少次什么样的操作) 度 (function connection objective), 接口复杂度 (interface params complexity objective), 业务专注度 (business focus objective). 同时我们也为算法预留了接口, 使工程师可以自行添加或修改目标函数.

FO 是对于一个 {*BusinessClusters*} 在功能联系视角下的度量, 其中的 n 是 {*BusinessClusters*} 中业务集群的数量, $Fc(BusinessCluster_i)$ 是 $BusinessCluster_i$ 中 CC_i 和 $\{GSCs\}_i$ 之间 R_{ij} 的累加和, 累加和可以由业务矩阵查出. *FO* 的数值越大, 意味该抽象微服务候选集合表现出更好地高内聚低耦合的特性.

$$\max FO = \sum_{i=1}^{n} Fc(BusinessCluster_i) \tag{2}$$

$$\min IO = \sum_{i=1}^{n} Ic(BusinessCluster_i) \tag{3}$$

$$\min BO = \sum_{i=1}^{n} Bc(BusinessCluster_i) \tag{4}$$

IO 是对于一个 {*BusinessClusters*} 在接口复杂度视角下的度量. *Ic* 是业务集群 $BusinessCluster_i$ 中所有接口方法的传入参数与返回参数复杂度的总和, 反映了该业务集群接口的复杂度, 其中的 k 是业务集群中接口的总数. $complexity(param_j)$ 和 $complexity(return_j)$ 用于计算业务集群中第 j 个接口的参数复杂度, 其中 $j \in [1,k]$. 为了区分不同类型的参数, 定义了两种不同的参数复杂度. 类和 List 的复杂度为 10, 基本数据类型的复杂度为 1. 当 *IO* 的值较小, 意味着最终在 {*BusinessClusters*} 上形成的抽象微服务接口的设计更简单, 符合迪米特法则. 而在服务之间进行通信时, 接口的参数越简单, 越有利于降低服务之间的通信成本.

$$Ic(BusinessCluster_i) = \sum_{j=1}^{k} complexity(param_j) + complexity(return_j) \tag{5}$$

BO 是对于一个 {*BusinessClusters*} 在用例视角下的度量, $Bc(BusinessCluster_i)$ 计算了一个业务集群在代表性 Trace 集合 (第 3.2.2 节讲述) 中参与了多少条 Trace. 如果一个业务集群涉及很多不同的 Trace, 这意味着该抽象微服务有更大的可能性参与到每个用例中, 由此产生的微服务的可重用性很低. 根据单一责任原则, 每个微服务的责任应尽可能单一, 并集中在单一业务领域. 当 *BO* 的值较小时, 意味着最终在 {*BusinessClusters*} 候选集合上形成的抽象微服务更专注于自己的业务.

在识别出 {*BusinessClusters*} 后, 我们已经知道每个抽象微服务下负责业务流程的业务类组成, 并根据业务类来确定数据访问类的归属. 在功能单元识别阶段, 我们记录了每个数据访问对象所访问的数据表 DAO2Tables (数据表 Table 对应数据实体类). 我们查询数据访问类使用矩阵, 将每个数据访问类分配给使用其次数最多的业务集群, 并将数据访问类访问的所有数据实体类也分给该业务集群. 一个数据实体类可能同时被多个抽象微服务所拥有. 我们允许这种代码重复是因为在单体架构系统下, 通常只会访问一个数据库. 而将数据库进行拆分后, 由于单体系统设计不规范, 很可能出现几个微服务同时依赖相同数据表的情况.

得到每个抽象微服务的类组成后, 我们可以通过{MessageRecords}中的消息去识别微服务接口. 如果消息的发送对象和接收对象属于不同微服务, 消息应该被识别为消息接收者所属微服务的接口. 如果消息接收者的方法不在所属微服务的控制类中, 则将消息对应的方法签名添加到控制类, 然后让控制类将消息转发到真正的消息接收类中. 这就是我们所谓的附属类的一个方法提升到控制类中, 如果发生了这种提升或接口原本属于控制类, 此接口所属的 *isRaised* 属性被置为 true. 否则, 此接口所属的 *isRaised* 属性被置为 false. 设计这种提升动作是为了满足迪米特法则, 使控制类来负责接口的请求转发, 其目的是降低类之间的耦合度, 提高模块的相对独立性.

3.2 微服务模型生成

在得到抽象微服务后, 我们可以根据抽象微服务和运行日志, 进一步生成微服务图及微服务序列图, 这些模型都是通过 MSA-Generator 自动生成并可在 MSA-Modeller 中加载. 使用微服务图及微服务序列图能够帮助工程师了解迁移后的 MSA 系统的静态结构以及动态行为.

3.2.1 微服务图生成

在得到每个抽象微服务后, 可以根据每个抽象微服务来进一步生成在模型章节中所定义的微服务图. 我们设计了相应的算法, 来为每个抽象微服务生成微服务图来描述微服务静态结构. 对于一个抽象微服务 $microservice_i = < CC_i, \{GSCs\}_i, \{DACs\}_i, \{DECs\}_i, \{Is\}_i >$, 算法根据其属性值来生成 UML 模型.

(1) UML 组件模型生成规则: 根据 $microservice_i$ 中的 CC_i 属性生成一个组件, 组件名由 CC_i 属性值确定. 该组件所适用的构造型为 Microservice.

(2) UML 类模型生成规则: 为 CC_i 属性以及 $\{GSCs\}_i, \{DACs\}_i, \{DECs\}_i$ 属性集合中的每个类生成 UML 模型中的类, 类名为类的全限定类名. 这些类所适用的构造型则由其所属的抽象微服务属性类型决定 (Controller Class, General Subordinate Class, Data Access Class, Data Entity Class).

(3) UML 接口模型生成规则: 为 $\{Is\}_i$ 数组中的每个微服务接口生成一个接口模型, 接口名由微服务接口本身名字确定, 接口参数值类型由微服务接口的参数值类型确定. 该接口所适用构造型为 MSInterface, 构造型标注 *isRaised* 的值由微服务接口的 *isRaised* 属性确定.

MSA-Generator 会为这些生成的 UML 模型自动适配上 MSA profile, 即可得到微服务构造型、控制类构造型等描述微服务静态结构的构造型. 这些构造型可被引入类图和构件图中, 按照 MSA profile 的约束对微服务的内部和外部结构进行建模, 进而得到微服务图. 在实际的实现过程中, MSA-Generator 在抽象微服务生成完后, 会将抽象微服务的数据存储在 JSON 当中 (也可使用 XML 等其他存储格式). MSA-Generator 按照上述的规则, 自动为每个抽象微服务的 JSON 数据生成对应微服务构造型的微服务图对应文件. 一个微服务图会对应 3 种文件, *.di 文件、*.uml 文件和*.notation 文件. 当把这些文件导入进 MSA-Modeller 后, 工程师可依据 UML 2.5.0 等其他标准进一步对这些微服务图进行可视化交互建模.

3.2.2 微服务序列图生成

微服务序列图根据每个微服务构造型以及代表性 Trace 集合 $\{RTs\}$(Representative Traces set) 生成. 如果任意两个 Trace 具有不同的对象调用序列, 但具有相同的类调用序列, 则将这两个 Trace 视为相同的 Trace. $\{RTs\}$是根据上述规则对 Trace 集合中的 Trace 进行过滤得到的, 这样能够完整地反映系统行为, 并去掉系统行为中的冗余信息. 对于$\{RTs\}$中的每一条 Trace, 可以根据其方法调用序列生成对象序列图. 为了获得对象调用序列, 需要迭代地访问$\{RTs\}$中的 Trace, 然后将其转换为调用树. 调用树的节点 (treenode) 表示 Trace 中的一个方法调用, 并包含被调用的方法 (treenode.Method), 类 (treenode.ClassSignature) 和对象 Id (treenode.ObjectId). 我们用 UML 中的消息 (Message) 来表达对象与对象之间的方法调用, 因为调用存在的方式是多样的 (同步、异步) 等, 因此使用消息做统一抽象. 一个调用树节点的父节点为调用者, 其孩子节点们为方法中直接调用的其他方法, 由左往右顺序为调用的先后顺序. 因此调用树的深度优先遍历即为对象的调用顺序, 并根据调用的方法将其转换为对应的消息. 为了在 MSA-Generator 上进行序列图的可视化展示, 我们使用 PlantUML 生成图片. PlantUML 是一种开源工具, 允许用户利用纯文本语言创建 UML 图. 图 5 的左下角图片和右边图片是 MSA-Generator 根据一个调用树所生成的 HTML 纯本文以及此纯文本对应的对象序列图.

在通过 Trace 得到对象序列图后, 我们设计了一个算法将对象序列图转换为微服务序列图. 微服务序列图的相关定义在第 2 节中已经给出, 每个微服务序列图对应着系统的一个功能, 描述了此功能下服务之间的交互和交互的发生顺序. 在对象序列图中, 一条生命线可以代表一个参与者 (角色), 一个 (特定类的) 对象. 在微服务序列图中, 一条生命线可以代表一个参与者 (角色), 一个特定的微服务. 在对象序列图中, 使用消息统一作为对象之间调用的一种表达. 在微服务序列图中, 使用微服务消息 (第 2 节定义的 MSMessage) 来表示服务之间的接口调用. 我们设计了对象序列图转换为微服务序列图的转换规则, 此转换规则的主要思想是按照对象序列图中的消息发送顺序, 对每一条消息进行验证. 如果消息的发送者与接收者的所属微服务不同, 则消息对应方法是微服务接口之间的调用, 应创建新的微服务生命线并生成相应的微服务消息. 对于对象序列图中的每条消息, 按照其发生顺序进行遍历并进行以下操作.

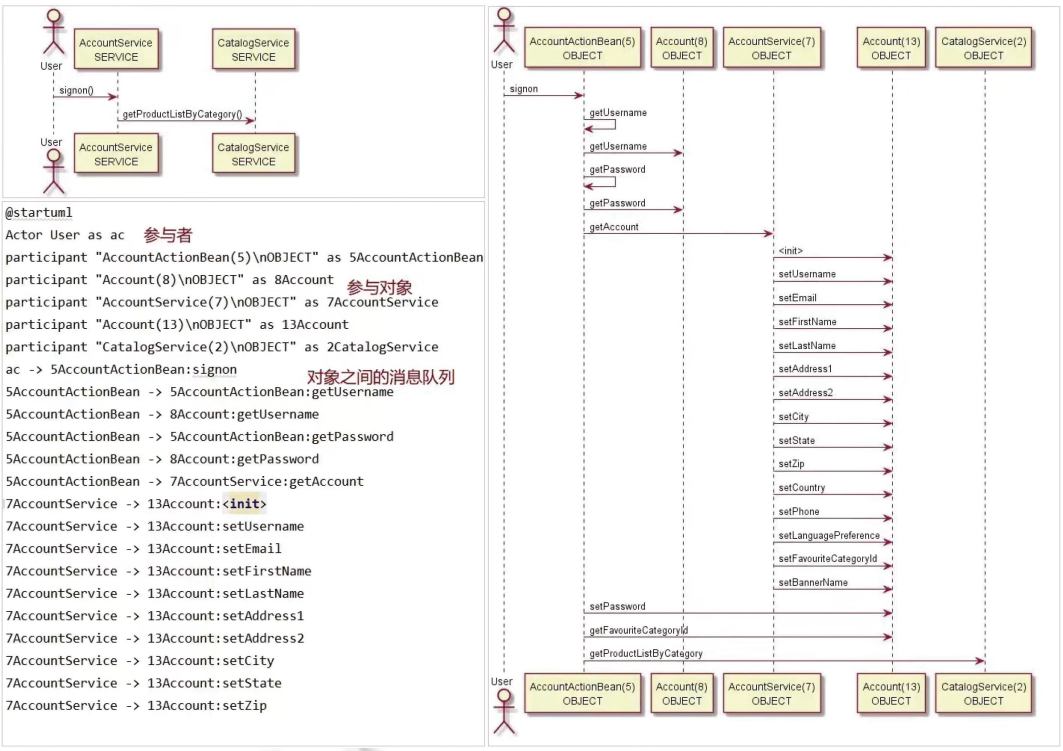


图 5 对象序列图及其对应 HTML 文本和微服务序列图

(1) 对于第 1 条消息, 这个消息是角色和微服务启动交互所产生的消息. 根据这条消息生成新的微服务消息(微服务消息名与对象消息名相同), 微服务消息的发送者为一个新生成的角色生命线 (与对象序列图中角色生命线相同). 微服务消息的接收者为一个新生成的微服务生命线, 此微服务生命线对应消息的接收者对象所属类的所属微服务构造型. 如果消息不是第 1 条消息, 则跳转至步骤 (2).

(2) 根据消息的接收对象和发送对象对应类查找分别所属的微服务构造型, 判断两者所属的微服务构造型是否相同. 如果相同, 跳过此消息, 否则跳转至步骤 (3).

(3) 根据这条消息生成新的微服务消息 (微服务消息名与对象消息名相同), 查找消息的发送者和接收者所属类的所属微服务构造型, 是否在此微服务序列图中存在对应的微服务生命线. 如果存在则将消息关联到对应的微服务生命线, 不存在则生成新的微服务生命线并关联.

待对所有消息进行验证后, 微服务序列图的生命线和微服务消息已生成完毕, 统计所有生命线以此设置微服务序列图的 Actors 及 MSs 属性, 设置 traceNum 为此微服务序列图对应的对象序列图所持有的 Trace 序号. 图 5 左上角的图片为依照上述规则所生成的微服务序列图 (PlantUML 绘制). 同样与第 2 节中生成微服务图的方法相似, MSA-Generator 也会为每个微服务序列图生成*.di 文件、*.uml 文件、*.notation 文件供加载到 MSA-Modeller 中.

3.3 微服务识别结果的评价指标

目前大多数工作仅评价针对业务类聚类的微服务识别结果, 忽视了对微服务所拥有数据相关类的划分和评价. 本文结合在工业界中常用的微服务提取指标[38]以及考虑在黑盒测试情况下所能得到的信息, 提出了 5 个对微服务设计方案的评估指标. 一共由 5 部分组成: 统计指标、服务接口内聚度、服务模块度、请求接口复杂度和数据库区分度. 基于这些评价指标, 将能够对 MSA-Generator 自动微服务识别结果的优劣进行量化评估, 还可支持 MSA-Modeller 实施设计方案动态调整后所得结果的对比评估.

3.3.1 统计指标

统计指标统计了此次拆分结果和日志输入的数量相关的信息, 包括工具运行时间、生成微服务的数量、类的数量、日志中的 Trace 数量、代表性 Trace 集合中的 Trace 数量和接口数量. 图 6 是 JPetStore 在 MSA-Generator 上首次运行后生成的设计方案的各项指标. 可以看到此次上传到 MSA-Generator 中的日志共有 2 761 条 Trace, 经过去重后获得 35 条代表性 Trace. MSA-Generator 根据这份日志将此系统下的 23 个类 (不包含数据实体类) 划分成了 3 个微服务 (一个微服务平均 7 个类), 所有的微服务一共暴露了 38 个接口.

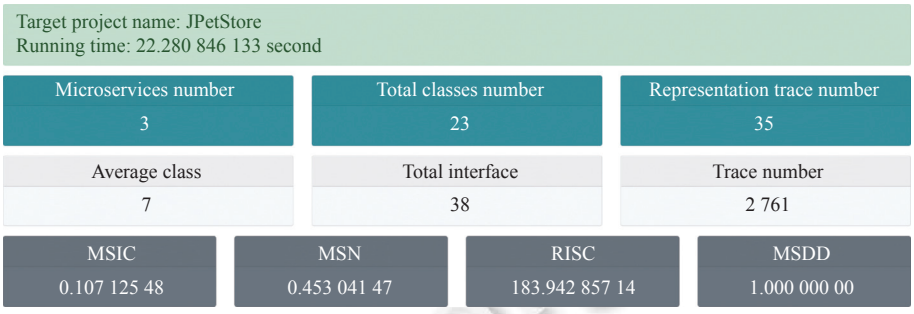


图 6 评价指标

3.3.2 服务接口内聚度

在黑盒的情况下, 无法从代码的角度度量内聚性. 因此考虑从接口的角度来衡量内聚度, 即根据接口的领域概念的重叠程度来衡量接口之间的相似性. 对于一个微服务的接口内聚度衡量, 首先根据接口所属的类对所有接口进行分类. 根据单一责任原则, 每个类下的函数应完成一个职责. 对于同一类下的接口, 我们不度量它们的域相似性. 我们应该衡量的是所属不同类的接口领域相似性, 以判断这些类是否能够合作完成一系列相关职责. 设一个微服务中有 n 个类, $I_{i\text{-}k}$ 为第 i 个类下的第 k 个接口, Num_i 为第 i 个类下的接口的数量. 使用微服务接口内聚度 (microservice interface cohesion, MIC) 衡量一个微服务的接口集合内聚程度, 它根据 Athanasopoulos 等人[39]提出的 LoC_{msg} 修改而来, 其取值范围在 [0, 1]. 从公式 (6) 可以看出, 当一个微服务中只有一个类时, 微服务的内聚性为 1 (最大值). 如果微服务下有多个类, 则两两比较不同类下的接口内聚度:

$$MIC=\begin{cases}1, & \text{if } n=1\\ \dfrac{\sum_{k=1}^{Num_i}\sum_{m=1}^{Num_j}S_I(I_{i\text{-}k},I_{j\text{-}m})}{\sum_{i=1}^{n}Num_i\times Num_j}i<j, & \text{if } n\neq 1\end{cases} \tag{6}$$

S_I 函数是一个衡量两个接口相似性的公式:

$$S_I(I_k,I_m)=\frac{|f_{\text{term}}(sig_k)\cap f_{\text{term}}(sig_m)|}{|f_{\text{term}}(sig_k)\cup f_{\text{term}}(sig_m)|} \tag{7}$$

$f_{\text{term}}(sig)$ 对每个接口中的类名, 方法名和所需参数类型名进行领域单词提取 (即有意义的业务单词提取, 注意提取过程中需要排除包名), 该函数还过滤掉了停止词 (英语中没有意义的单词, 如代词、介词、数组和单个字母).

MSIC (microservice set interface cohesion) 定义为一个微服务候选集合的平均 *MIC*, 衡量了整个系统的接口内聚度, 它的取值范围在 [0, 1] 间, 值越大, 说明内聚程度越高:

$$MSIC=\frac{\sum_{i=1}^{n}MIC_i}{n} \tag{8}$$

3.3.3 服务模块度

服务模块度 (microservice set modularity) 根据 Mitchell 等人[40]提出的模块值修改而来, 其计算基于依赖图 (DG). 依赖图是在 $\{RTs\}$ 下的每个类之间的依赖关系. $DG=(V,E)$, V 为目标系统的类集, E 为类间调用关系集. $E=\{e_i|e_i=<v_x,v_y,w_i>,v_x\in V,v_y\in V,w_i\in[1,5]\}$. 当 $\{RTs\}$ 中有一个从 v_x 到 v_y 的调用时, 依赖强度值设为 w_i, 有 $e_i=<v_x,$

$v_y, w_i > .\ w_i$ 的值是由功能场景 (考虑类所负责的功能) 中 v_x 和 v_y 的间隔层数决定的. 在功能场景中, 我们将每个微服务看作 3 层架构, 第 1 层由控制类组成, 负责公开所有外部接口和请求转发; 第 2 层由一般附属类组成, 负责完成业务流程; 第 3 层由数据访问类组成, 负责与数据库交互. 如果 v_x 和 v_y 所属微服务相同, w_i 被设置为 v_x 与 v_y 之间差值的绝对值. 如果不同, w_i 被设置为 v_x 和 v_y 所处的层数的总和相加. 提出功能场景的原因是因为服务之间正常的调用是通过微服务接口进行调用. 例如, 当微服务 A 中的控制类直接调用微服务 B 中的数据访问类和微服务 B 中的控制类相比. 两者之间的耦合关系是完全不同的. 前者的调用显然不符合设计模式, 可以认为是紧耦合, 而后者是松耦合.

微服务候选集合将依赖图划分为 n 个群 (cluster), 每个群对应一个候选微服务中的所有类. 聚类因子 (cluster factor, *CF*) 定义为群的内部边的权重和与群的内部边的权重和加上群外部边 (连接该群和其他群的边) 权重和的二分之一的比值. 使用外部边权重的一半是用来对外部边相连的两个群产生同样的负面影响. 对于第 i 个微服务的集群因子 CF_i 定义如下:

$$CF_i = \begin{cases} 0, & \mu_i = 0 \\ \dfrac{2\mu_i}{2\mu_i + \sum_{j=1, j\neq i}^{n} (\epsilon_{i,j} + \epsilon_{j,i})}, & \mu_i \neq 0 \end{cases} \tag{9}$$

其中, μ_i 为第 i 群的内部边的权重和, 而连接第 i 群和第 j 群的边的权重和被记为 $\epsilon_{i,j}$ 和 $\epsilon_{j,i}$. 从公式上来看, 一个群外部耦合关系越少, 内部聚合关系越多, 其群因子值越大, 越能满足低耦合设计原则. 对于服务模块度的计算, 即每个微服务聚类因子相加取平均值. 它的取值范围在 [0, 1] 间, 该值越大, 说明系统越能满足低耦合设计原则:

$$MSM = \frac{\sum_{i=1}^{n} CF_i}{n} \tag{10}$$

3.3.4 请求接口复杂度

通信成本是重构后的微服务之间通信时间相关的负面影响, 如网络条件和服务器之间的物理距离等. 在目前的条件下, 我们无法直接衡量微服务系统是否具有较低的通信成本. 因此, 考虑从微服务接口的参数复杂度角度出发去片面的描述通信成本. 这是因为在微服务接口调用时会通过 HTTP 等协议传输参数, 当参数涉及的内容较多时, 对一些需要低延迟的系统会产生很大的负面影响.

对于一个 Web 请求 (在日志中对应一个 Trace) 下的请求接口复杂度 (request interface complexity, *RIC*) 的计算如公式 (11) 所示. 设第 i 个 Trace ($trace_i$) 下存在 n 次接口的调用, $RIC(trace_i)$ 定义如下, 它描述了对微服务系统发出一个 Web 请求后, 需要访问的所有微服务接口参数有多复杂.

$$RIC(trace_i) = \sum_{j=1}^{n} complexity(param_j) + complexity(return_j) \tag{11}$$

其中, $param_j$ 是第 j 次接口调用所需要的参数, $return_j$ 是接口所返回的参数, $complexity()$ 在前面已经定义. 通过公式 (11), 我们可以得到一个 Trace 下的请求接口复杂度, 最后将代表微服务系统业务逻辑的 $\{RTs\}$ 中所有 Trace 的接口复杂度结果求和取平均值, 得到微服务系统的请求接口复杂度 (request interface set complexity, *RISC*):

$$RISC = \frac{\sum_{i=1}^{k} RIC(trace_i)}{k} \tag{12}$$

其中, *RISC* 的取值范围不限, 且越大说明系统的请求接口设计越复杂, 通信的代价越高.

3.3.5 数据库区分度

理想的设计是每个微服务都有自己单独的数据库, 并可以通过其余微服务提供的接口修改其余微服务的数据库. 如果两个微服务之间的数据共享十分频繁, 不断的服务调用无疑会对增加服务的负担. 这时就需要通过建立共享表, 或者主、从表等方式去减轻对系统服务的负担, 但这无疑会增加分布式事务的难度.

我们通过数据库区分度 (database discrimination, *DD*) 计算两个不同微服务数据库中的数据表集合的差别, 该公式由 Jaccard 距离计算[41]公式改进得到. $\{Tables_i\}$ 代表着第 i 个微服务数据库下的数据表集合. 公式的取值在 [0, 1] 内, 值越大, 两个数据库之间的区别就越大, 数据表重复越少:

$$DD=\begin{cases}1, & \text{if } Tables_i==\varnothing \text{ and } Tables_j==\varnothing \\ \dfrac{|Tables_i\cup Tables_j|-|Tables_i\cap Tables_j|}{|Tables_i\cup Tables_j|}, & \text{else}\end{cases} \tag{13}$$

最后, 我们将任意两个不同微服务对应的 *DD* 值相加取平均值, 即可得到整个微服务系统的数据库区分度 (microservice set database discrimination, *MSDD*), 即:

$$MSDD=\frac{\sum_{i=1}^{n}DD(Tables_i,Tables_j)}{(n-1)!},\ i<j \tag{14}$$

其中, *MSDD* 的取值范围在 [0, 1] 间, 值越大说明系统的数据库设计越不存在重复. 当值为 1 时, 说明每个微服务的数据库中的数据表没有重复.

4 MSA-Modeller

核心部件 MSA-Modeller 的当前版本包含两个建模辅助工具.

(1) MSA-Modeller 前端建模工具: 提供基于网页的服务概览、服务上下文展现, 及服务调整功能. 但需指出的是, 前端工具仅提供 MSA 建模的初级功能, 即它支持对各微服务下类的归属进行动态调整, 但不支持调整后得到的模型进行基于 OCL 的语法检验. 故需要借助 MSA-Generator 提供的模型导出功能 (model download), 导出调整后的模型文件并载入到 MSA-Modeller 桌面端建模工具中进行语法检验.

(2) MSA-Modeller 桌面建模工具: 它基于 Eclipse 的 Papyrus 插件进行开发. Papyrus 可支持 UML 2.5.0 规范, 默认支持 SysML 建模. 我们则在 Papyrus 加入了我们定义的微服务元模型, 并对相关建模元素和编辑器进行了二次开发, 使之能支持面向微服务体系架构设计模型的可视化展现、交互式建模及语法检查. 当然, 后续版本将持续开发提供如模型检测、代码生成等模型驱动开发设施.

4.1 MSA-Modeller 前端建模工具

前端工具基于 Web 开发, 与 MSA-Generator 共用前端界面, 在 MSA-Generator 运行完后即可进行使用. 旨在提供 MSA 模型查看及交互式建模的初级功能, 包含: 服务概览、服务上下文展现及服务调整. 服务概览模块能够让工程师快速对每个微服务的组成 (类, 接口) 和交互 (序列图) 有直观的了解; 服务上下文模块展示微服务化后类与类在日志中的直接调用次数; 服务调整能够交互式的供工程师按照微服务模型约束进行微服务下类的调整, 并根据调整结果生成新的微服务模型. 这种交互式功能是大部分微服务拆分工具[9,13,16]所不支持的, 其他拆分工具往往只支持对结果的可视化而不允许用户进行交互式调整与新结果生成.

4.1.1 服务概览

使用 MSA-Generator 对 JPetStore 系统进行分析后, 会得到微服务对应的模型文件, 将这些文件导入 MSA-Modeller 后会得到 3 个微服务图和 35 个微服务序列图. 这 3 个微服务分别是账号微服务, 商品目录微服务, 订单微服务. 服务概览模块负责展示每个微服务的可视化元素组成与图片形式的对象序列图和微服务序列图, 能够快速地让工程师对每个微服务的组成和交互有直观的了解. 可视化元素组成包括此微服务具体的类、接口; 图片形式的序列图包括此微服务相关联的 Trace 所转换成的对象序列图和微服务序列图, 也可进入序列图对应的 Trace 模块快速查看此 Trace 所关联的所有微服务. 后文图 7(a) 展示了商品目录微服务下的可视化元素组成, 图 7(b) 展示了商品目录微服务下的部分对象序列图和微服务序列图, 这些序列图都对应着日志中的某一个 Trace.

4.1.2 服务上下文展现

借助 MSA-Generator 统计代表性 Trace 集合中每个类与其他类之间的调用次数. 在 MSA-Modeller 前端建模工具中, 工程师可以看到每个微服务下的类与其他类的相互调用情况. 进而得知在原始业务逻辑下微服务之间以及微服务内部的类的依赖程度. 如图 8 所示, 图中的节点代表一个类, 有向边上的权值代表所连接的两个节点所对应的类在日志中存在的总调用次数. 同属于一个微服务的类被一个圆圈所包括, 每个微服务下 3 种不同颜色区分各个类的所属类别 (控制类, 一般附属类, 数据访问类).

(a) 组成

(b) 交互

图 7　商品目录微服务的组成与交互

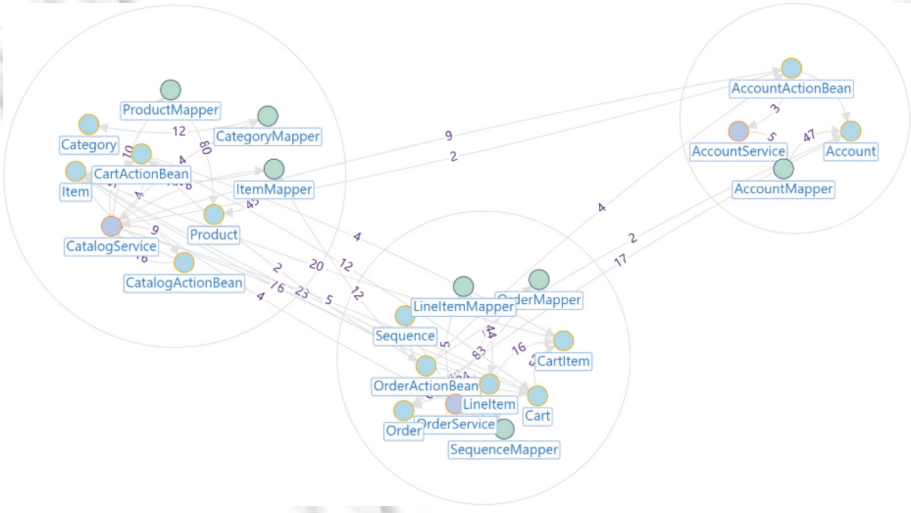


图 8　服务上下文中的各个微服务的调用关系

4.1.3　服务调整

基于 MSA-Generator 生成的微服务拆分设计, 开发者能够使用 MSA-Modeller 前端建模工具对设计方案进行修改. 图 9 为供开发者进行服务调整的页面, 一个微服务下有许多的类, 从上到下 3 种不同颜色的类对应着此微服务的控制类, 一般附属类, 数据访问类. 开发者可拖动微服务下的类到另一个微服务中, 从而改变此类的所属微服

务. 此模块允许工程师能够更改微服务中一般附属类与数据访问类的归属, 但不能修改控制类的归属. 这是因为对于每个微服务来说, 控制类暴露此微服务的业务功能, 不能对其进行更改, 否则会违背单一职责原则. 在调整完毕后, 每个微服务下的数据实体类, 接口, 微服务序列图等都会自动重新调整并生成新的设计方案, 工具会再次对新的设计方案进行评估并生成其对应的模型文件.

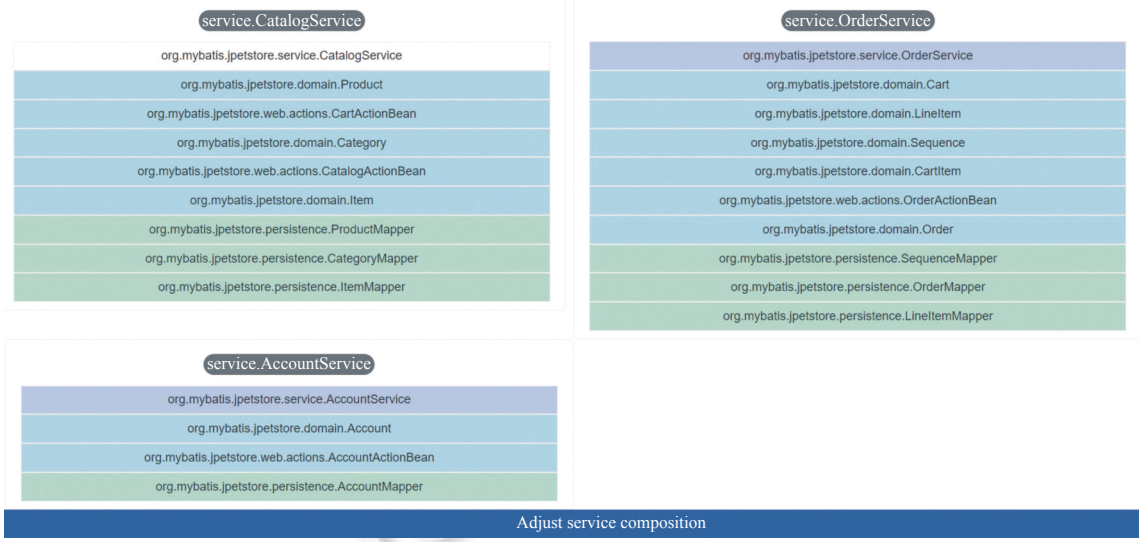


图 9 对微服务下的类归属进行调整

4.2 MSA-Modeller 桌面建模工具

MSA-Modeller 桌面建模工具提供面向微服务图和微服务序列图的可视化展现和综合编辑功能, 是一个集成的微服务模型开发环境.

下面介绍如何使用微服务图来刻画 MSA 静态结构模型.

(1) 对于一个微服务的内部结构, 我们基于类图加上前文定义的微服务元模型中的构造型, 对手动建模 (或来自 MSA-Modeller 前端建模工具生成) 的微服务进行可视化展现和交互式建模. 一个微服务主要由类构成, 类图可以描述微服务中存在的类、类的内部结构以及类与类之间的关系等. 如图 10 中, 账号微服务专门负责对账号进行相关操作, 我们对此微服务进行建模. 从图 10 中可以看到账号微服务下的具体的类, 接口以及其中各个模型的关系, 并且最下面的窗口中也显示了微服务构造型的相关标注.

(2) 对于一个微服务外部的结构, 我们基于组件图的模型表达以及微服务构造型对微服务外部结构进行模型表达; 其中的微服务构造型继承自组件标准元模型; 原有的 UML 组件和微服务共享一些公共概念, 例如 required-interface、provided-interface、端口、连接器、依赖关系和 usage 等. 在此基础上对生成的微服务构造型进行交互式建模, 建立微服务之间的接口依赖等关系. 假设订单微服务需要通过账号微服务的接口 getAccount 来获取账户数据, 则可建模如图 11 所示, 账号微服务通过 Port1 暴露此接口, OrderService 通过 Port1 端口进行依赖.

此外, 为表达微服务之间交互行为模型的微服务序列图: 我们基于对象序列图, 并引入 MSA profile 的微服务序列图元模型来构建微服务序列图. 序列图能显示对象按时间顺序排列的相互作用, 它描述了场景中涉及的对象以及执行场景功能所需的对象之间交换的消息序列. 而其扩展而成的微服务序列图则能描述微服务之间通过接口的交互, 以及与运行日志中 Trace 的对应关系. 图 12 展示了 Trace 1 中 User 用户通过账号微服务创建账户后, 根据商品的不同类别开始浏览对应类别下的商品的业务流程. MSInteraction 对应的标注值表明了此 diagram 对应的 Trace, 以及参与的微服务和 actor, 并能通过 OCL 验证来确定微服务序列图建模是否正确.

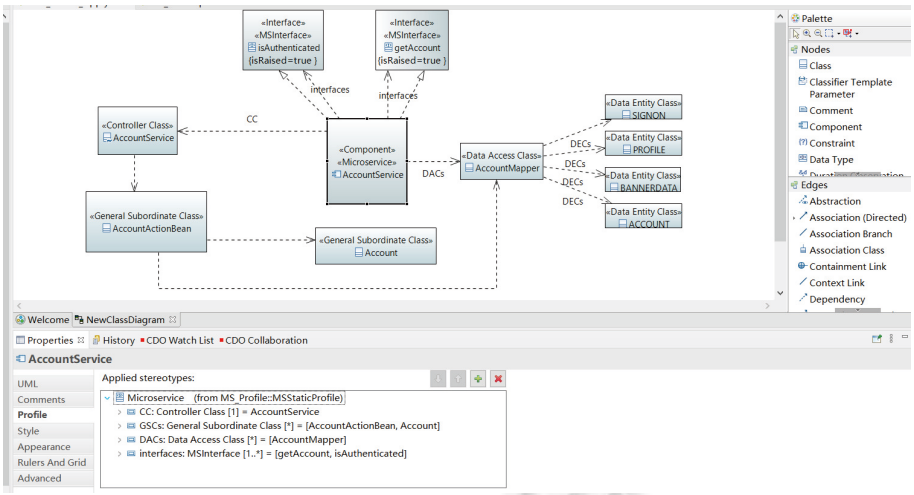


图 10 账号微服务模型

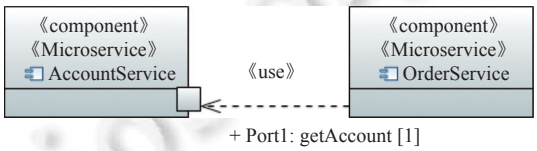


图 11 订单微服务调用账号微服务接口

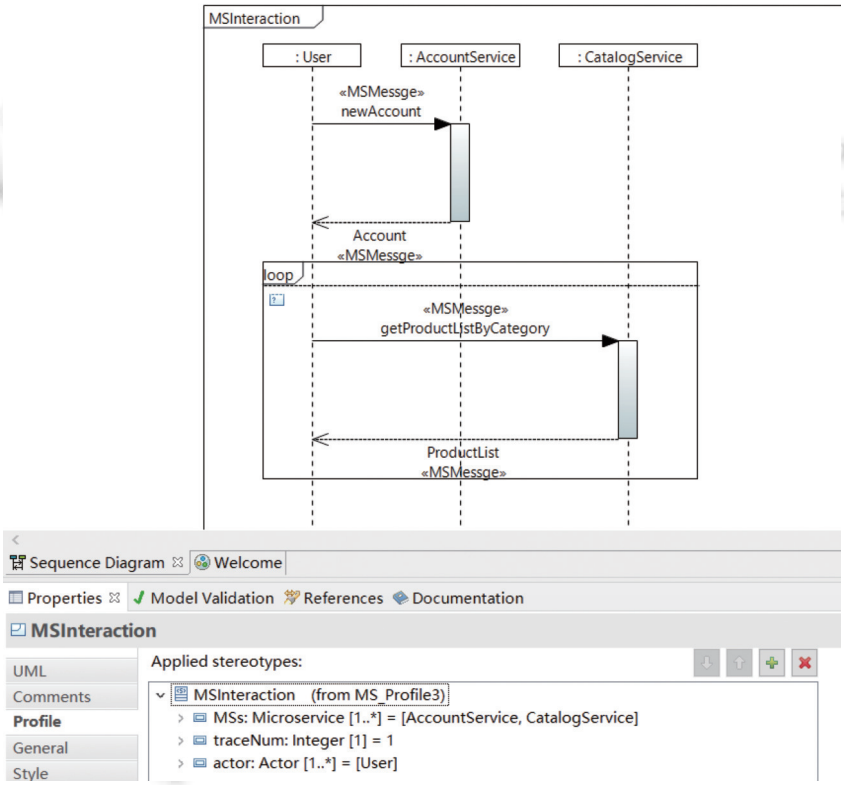


图 12 Trace 1 对应的微服务序列图

对于建模完成后的微服务图和微服务序列图, 我们可以通过 MSA-Modeller 提供的 OCL 验证来检测微服务模型是否符合 MSA profile 中所定义的约束. 这样可确保工程师按照 MSA 的规范来进行建模, 以确保后面的模型驱动开发正确. 在工程师对微服务模型修改并验证完成后, 还可根据 Eclipse 的各种第三方插件进行模型驱动开发. 例如可以使用 Papyrus Designer 为微服务模型生成平台相关性的代码 (C++, Java 等). 目前我们也正在研发代码生成工具 MSA-Coder 为微服务模型生成 Java 相关的微服务框架 Spring Cloud 代码. 当然, 如果工程师想进行其他模型驱动的开发, 也可使用 Papyrus 本身所含的基于 Moka 的模型执行插件, 集成安全技术插件等.

5 实验研究与案例分析

我们一共进行了 3 组实验. 第 1 组是工具有效性与鲁棒性实验, 通过此实验证明 MSA-Generator 在日志输入不完整的情况下, 其拆分结果仍然是有效的, 体现了工具的鲁棒性; 第 2 组是工具对比实验, 通过与其他微服务拆分工具和框架对比. 证明 MSA-Generator 在性能上更加出色, 能够提供更多的信息给工程师作为拆分参考. 在与其他工具和框架的拆分对比中, MSA-Generator 的拆分结果同样有效, 且部分指标表现更加优秀; 第 3 组是工具的功能转换完备性实验, 通过数据分析的方式证明我们的工具在转换过程中不会造成功能损失. 最后, 讨论部分指出了我们的工具平台的实用性与局限性. 我们一共选取了 4 个开源软件系统进行实验, 研究系统分属电商、社交博客、线上考试和 ERP (企业资源计划). JPetStore 是一个简单且架构清晰的小型系统, 使用 Spring, MyBatis 等常用框架开发; Zb-blog 整合了 Redis, SpringBoot, Shiro 等常用组件; Exam++是一个采用 SSM 框架开发的数据驱动系统, 提供了可灵活配置的功能用于不同考试领域; JeeSite 是一个企业信息化领域开发平台, 整合了 Spring Boot, Shiro 等大量常用组件. 它们的详细信息如表 4 所示, 展示了每个项目的类的数量, 项目的有效代码行数, 包的数量, 数据表的数量, 项目中初始化数据库的 SQL 语句的行数.

表 4 实验项目详细信息

项目名	类数量	有效代码行数	包数量	数据表数量	SQL行数
JPetStore	39	4 932	4	12	282
Zb-blog[42]	198	10 277	14	15	1 154
Exam++[43]	81	7 757	7	19	663
JeeSite[44]	292	25 366	6	55	9 162

5.1 参数设置

我们的参数设置有两方面, 一方面是对不同类型方法的调用强度和调用频度的设置, 另一方面是对 NSGA-II 中参数的设置. 我们参考了 Akoka 等人[45]对关系强度的设置, 将强度常量组中的 $IStr_c$, $IStr_u$, $IStr_r$ 设置为 100, 10, 1, 以此来遵循第 3.1.2 节中的策略以突出方法调用强度的不同影响. 同时, 为了将方法发生的频度也纳入到度量过程之中, 并使数量足够时调用频度的影响能够超越调用强度. 调用频度常量组中 $IFre_c$, $IFre_u$, $IFre_r$ 被依次设置为 1.03, 1.02 和 1.01. 我们在结构设计分明的项目 JPetStore 进行了多次实验, 确保了此参数下, 每次微服务拆分结果稳定, 能够较好地刻画类之间的功能联系度.

对于面向微服务识别的 NSGA-II 算法, 其参数设置是基于标准遗传算法的一般建议, 并综合考虑在代码量最大的项目中 JeeSite 上进行的多轮实验结果进行设置. 我们的依据是如果在大型项目下此参数对应的实验结果能表现稳定, 那么在小型项目下也一定适用, 以此避免实验结果的不稳定性. 最终, 种群被设置为 200, 交换和变异概率被设置为 0.001 和 0.2. 遗传算法最多迭代次数被设置为 300.

5.2 工具有效性及鲁棒性实验

为了证明我们的工具即使在日志输入不完整的情况下也具有鲁棒性, 且拆分结果仍然具有效性. 我们进行了敏感性实验. 我们的假设是, 根据我们的评价指标, 能够评价此次微服务拆分结果的各个性质的好坏. 我们以每个项目下的收集的一份较为完整的日志为样本, 并设此时所得到的拆分结果的评价指标是有效的. 之后取日志长度

的 2/3, 1/2 来进行实验, 将这两次实验的评价指标与完整长度日志的评价指标进行对比. 在截取日志的过程中, 我们根据 Trace 编号的倍数进行截取. 例如截取 2/3 长度的日志时, 即去掉按序编号后编号为 3 的倍数的 Trace. 这是因为在 Trace 的收集过程中, 我们按照一系列测试用例来进行收集, Trace 的顺序也对应着测试用例的顺序. 而顺序相邻的测试用例可能存在一定业务逻辑联系, 例如订单相关的用例是连续执行的, 且生成订单用例一定在删除订单用例之前. 按照倍数进行截取, 能保证业务逻辑的覆盖不会太分散 (订单相关的用例对应 Trace 仍然存在), 又减少了对相同业务逻辑方面的深入 (某个订单相关的用例被截取掉). 这种截取方法我们称为业务截取法. 我们对每份日志都按照业务截取法进行了 20 次实验, 将每次实验所获取的结果相加取平均值, 以此来避免偶然性. 结果如表 5 所示. 每一行的数据代表一份日志下的 20 次实验的平均结果. 展示了 4 个评价指标 *MSIC*, *MSM*, *RISC*, *MSDD* 下的得分, Trace 总数量, 代表性 Trace 数量. 除了业务截取法, 我们也尝试了按照 2/3, 1/2 的比例随机截取日志. 但这可能会造成日志覆盖业务场景减少过多 (表现为代表性 Trace 数量急剧减少), 漏掉微服务之间交互的关键业务场景, 导致一部分识别的微服务不符合规范 (没有接口). 在这种业务场景不足以提供自动微服务识别的情况下, 开发者可以重新规划用例设计, 或凭借业务理解通过手工调整来达到拆分.

表 5　有效性及鲁棒性实验

项目名称	*MSIC*	*MSM*	*RISC*	*MSDD*	Trace数量	代表性Trace数量
JPetStore	0.107 12	0.453 04	183.942 85	1.000 00	2 761	35
	0.1976 4	0.411 38	164.053 22	**0.666 66**	1 886	31
	0.116 90	0.366 55	206.320 00	1.000 00	1 449	25
Zb-blog	0.068 11	0.140 90	821.690 13	0.807 69	8 307	76
	0.072 28	0.146 80	831.465 51	0.807 69	5 677	58
	0.071 42	0.149 18	847.933 62	0.846 15	4 361	58
Exam++	0.075 29	0.161 51	1 524.606 71	0.900 00	3 337	67
	0.066 09	0.175 64	1 520.388 54	0.900 00	2 280	48
	0.070 34	0.175 17	1 312.912 79	0.900 00	1 751	43
JeeSite	0.031 36	0.016 06	6 466.750 00	0.973 76	40 188	272
	0.032 52	0.014 26	5 211.771 12	0.971 79	27 462	232
	0.031 41	0.013 80	**2277.616 58**	0.973 02	21 099	208

注: 项目指标*MSIC*, *MSM*, *MSDD*值越大越好, *RISC*越小越好; 每个项目的3次日志完整度从上到下分别是1, 2/3, 1/2. 加粗数据为异常数据, 其余数据波动都在一定范围内. 如Zb-blog的*RISC*指标在日志缺少情况下, 相比完整日志情况都变化不大

根据 Trace 总数量和代表性 Trace 数量可以知道此次实验所获取的日志的有效信息的大小. 从表 5 的每个项目下的 3 次实验的总体情况来看, 我们的方法在每个项目下日志不同的缺失情况下的差距不是太大, 具有一定的鲁棒性. 在对于 JPetStore 的第 2 次实验中, 其 *MSIC* 和 *MSDD* 发生了很大的变化. 这是因为日志的缺少引起了业务逻辑的减少, 本应属于 CatalogService 的一些类划分到了 OrderService 中. *MSIC* 指标较其他两次实验的急剧提高是因为 OrderService 中聚集了比以前更多的类, 虽然提高了其内聚度, 但违反了“一个微服务只解决一个问题”的原则. 由于两个微服务的业务逻辑的重合加重, 因此数据库设计也变得更差. Zb-blog 项目的第 2 次实验和第 3 次实验中的代表性 Trace 数目相同, 因此各指标的变化相差不大 (但总 Trace 数量不同也会给各指标带来影响, 动态关系分析时会考虑调用次数). Exam++项目的第 2 次实验中 *MSIC* 明显下降也是因为由于日志的缺少引起业务逻辑的变化, 导致部分微服务下类的拆分发生了很大的改变. 这些微服务的内聚程度相比其他实验会发生很大的变化, 进而影响到整个系统的 *MSIC* 指标. JeeSite 项目的每次实验的指标相对稳定, 变化较小. 第 3 次实验的 *RISC* 急剧减小是因为 JeeSite 本身就是一个比较大型的系统, 日志减少过多后, *RISC* 伴随业务逻辑减少而下降.

经过实验得知, 我们的拆分方法依赖于日志的完整情况. 但方法能在缺失日志的情况下尽力生成当前情况下各指标最好的微服务, 而不会因为日志的一些缺少, 引起微服务拆分结果的整体指标急剧下降. 即使日志不全, 在获得新的日志后可以及时投入进行弥补. 此外, MSA-Generator 能够在短时间内生成结果并允许工程师进行手工的修改. 我们的评价指标也能够对当前生成的设计方案进行一个综合的评价, 有助于使工程师对设计方案有个直

观的认识并及时进行反馈调整.

5.3 工具对比实验

为了证明本文的工具能够显著加快微服务拆分决策的速度, 并且所得到的微服务拆分结果是准确的. 我们设计了性能对比实验, 拆分结果对比实验, 来与其他工具和框架进行对比.

5.3.1 性能对比实验

实验目标是根据遗留系统来得到其重构为微服务的设计方案, 且设计方案应尽可能地包含更多的架构信息(例如评价指标, 模型信息等). 我们选用一个与我们的工具有着类似特性的传统微服务拆分工具 MSDecomposer[25], 以及一个模型驱动的微服务拆分工具 AjiL[12]来与我们的工具进行对比. MSDecomposer 工具是采用遗留系统的运行日志作为输入来得到微服务拆分结果, 其使用过程与 MSA-Generator 工具的使用过程相同: 包括 Kieker 配置, 测试用例分析, 日志收集和工具分析. 在熟练使用 Kieker 工具后, 工具分析比以前的时间花费会更少. AjiL 是一个手工图形化微服务建模工具, 需要人工分析遗留系统, 并根据人工拆分结果对新的微服务系统进行建模. 使用 AjiL 的实验是让几位有微服务研究背景和开发经验的硕士研究生, 让他们对表 6 中的 4 个实验系统一起进行人工分析并给出类和数据表的拆分方案, 并记录下每个人对每个项目得出拆分方案所得出的平均时间而来.

表 6 拆分方法过程时间记录 (min)

项目名称	Kieker配置、场景整理、日志收集	MSDecomposer分析时间	MSA-Generator分析时间	MSDecomposer分析总时间	MSA-Generator分析总时间	人工分析时间	AjiL建模时间
JPetStore	90	3	0.4	93	90.4	210.5	86.5
Zb-blog	120	5	0.9	125	120.9	266.1	312.6
Exam++	120	5	1.5	125	121.5	247.2	292.3
JeeSite	170	8	5.2	178	175.2	305.5	770.2

从表 6 中可以看出, 人工拆分的平均总耗时都是大于工具拆分的总耗时. 并且如果还需要在建模工具上进行手动建模, 其时间人力成本将会大大提高. 而 MSA-Generator 则可以自动生成模型文件, 提高工程师建模的效率. MSA-Generator 和 MSDecomposer 一样, 在上传日志后即可自动开始运行并给出拆分结果. 除了像 MSDecomposer 中的微服务拆分结果外, MSA-Generator 还给出了评价指标的计算, 直接能被载入的模型文件等, 这些结果能够带给工程师更多的设计参考. 在 MSDecomposer 与 MSA-Generator 对比中, MSA-Generator 在运行时间上相对较短. 这是因为通过分布式开发, 只要 MSA-Generator 获得抽象微服务后, 设计评估及可视化生成就能和模型生成并行执行. MSA-Generator 分析的时间主要聚集在遗传算法的迭代运行上, MSDecomposer 也同样需要 GA 算法不断迭代进行聚类. 但 MSDecomposer 的可视化与存储都需通过图数据库 Neo4j 来配合执行, 这可能也是增加 MSDecomposer 运行时间的原因. MSA-Generator 的存储直接通过 JSON 文件存储, 并通过 PlantUML 并行进行每条序列图的可视化, 降低了运行时间. 如果工程师需要对 MSA-Generator 此次生成的结果进行修改, 也可通过 MSA-Modeller 前端工具直接进行交互式修改, 并生成新的设计方案与对应模型. 以上的实验结果表明 MSA-Generator 运行效率高, 更有利于给工程师提供参考, 并且结合 MSA-Modeller 后能提升模型驱动开发的速度.

5.3.2 拆分结果对比实验

我们使用与 MSA-Generator 中的微服务识别方法相似的 FoSCI 框架中的方法和 MSDecomposer 中的方法来进行对比实验, 以证明我们的拆分结果更加合理. 为了达到方法的对比公平, 我们对 3 种方法都使用同一份日志, 将微服务生成数量设置为 MSA-Generator 识别的微服务数量, 并进行类与数据表的划分. (1) FoSCI 以从遗留系统的动态方法调用序列提取到的功能原子 (function atom), 作为遗传算法的基本单元进行聚类. 由于无法拿到源码, 我们复现了他们的微服务内部动态调用函数和微服务间动态调用函数进行聚类. (2) MSDecomposer 以遗留系统的动态方法调用序列进行微服务识别, 使用 G-N 算法对数据表进行聚类, 再由数据表根据方法调用序列搜索所关联的类得到微服务. 为了和其余方法同样只使用日志进行类与数据表的拆分, 我们根据源码并使用调用链构建数

据表权重矩阵, 使用模块度作为指标函数进行微服务聚类.

表 7 是对比实验结果, 我们将每个方法对每个项目进行 10 次实验, 并取最好的一次. 对于 *MSIC* 指标, MSDecomposer 表现较好, 特别是在 JeeSite 项目下. 一方面 MSDecomposer 采用自底向上搜索的策略能直接找到原始的接口类, 对接口识别有着积极的影响. 另一方面 MSDecomposer 加入了部分人工拆分的结果. MSDecomposer 无法对和多个数据表关联的类进行自动拆分, 需要开发者自行进行拆分, 这种类常常也是微服务识别的困难所在. 而人工拆分的类如果和微服务接口相关的话, 则会影响 *MSIC* 指标和 *RISC* 指标. 对于 *MSM* 指标, MSA-Generator 在所有项目比其余方法表现都好, MSDecomposer 每次比 MSA-Generator 稍差一点. 这可能是因为 MSA-Generator 通过对动态调用关系的分类划分和业务专注度提高了微服务的模块度. 而 FoSCI 在这一指标表现较差, 在 JPetStore 上比其他方法差了一倍左右, 只对调用次数的考虑并不能做到对模块度的很好划分. 对于 *RISC* 指标, MSA-Generator 在 Zb-blog 和 Exam++上比其他方法好. MSA-Generator 对接口的参数复杂度做出度量, 降低了 *RISC* 指标, 使微服务之间的通信代价变得更低. 对于 *MSDD* 指标, MSDecomposer 表现的最好, 因为 MSDecomposer 采用数据表聚类做到了对数据表的不重复划分. 而 FoSCI 无法对和数据操作的有关类和数据表进行监控, 因此无法进行与 *MSDD* 指标的计算. 从拆分结果对比可以看出, MSA-Generator 与其他工具同样能对微服务进行有效拆分, 虽然在 *MSIC* 指标和 *MSDD* 指标上没有达到最好, 但在 *MSM* 和 *RISC* 指标均表现比其余方法好.

表 7 拆分结果有效性实验

项目序号	*MSIC*	*MSM*	*RISC*	*MSDD*	方法种类
JPetStore	0.107 51	**0.453 04**	183.942 85	**1.000 00**	MSA-Generator
	0.130 64	0.203 97	**179.514 28**	不支持	FoSCI
	0.125 99	0.438 28	181.800 00	**1.000 00**	MSDecomposer
Zb-blog	0.082 77	**0.150 54**	**840.381 57**	0.807 69	MSA-Generator
	0.043 51	0.067 92	1 161.289 45	不支持	FoSCI
	0.104 12	0.139 00	940.934 21	**1.000 00**	MSDecomposer
Exam++	**0.103 69**	**0.191 43**	1 576.641 79	0.900 00	MSA-Generator
	0.050 57	0.036 34	2 696.641 79	不支持	FoSCI
	0.098 30	0.161 01	**1 350.731 34**	**1.000 00**	MSDecomposer
JeeSite	0.034 23	**0.015 15**	**6 273.783 08**	0.970 44	MSA-Generator
	0.030 19	0.009 92	6 532.136 02	不支持	FoSCI
	0.044 59	0.014 27	6 519.738 97	**1.000 00**	MSDecomposer

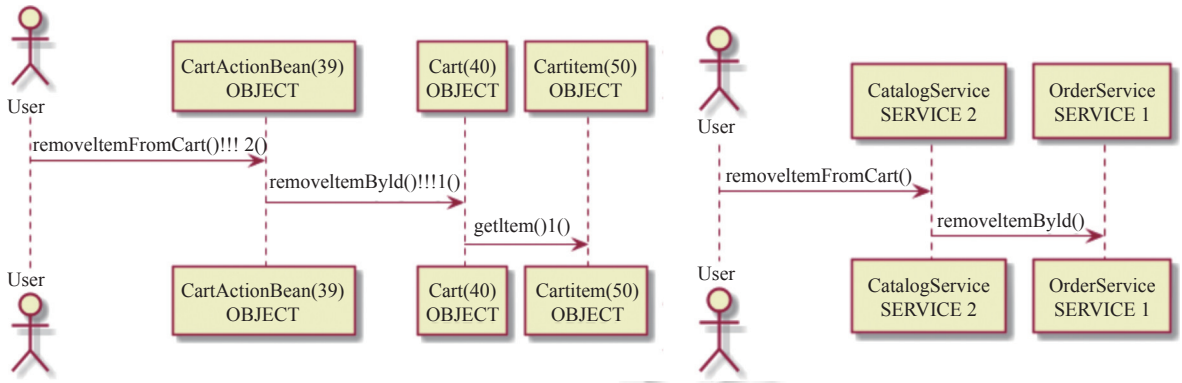
注: 项目指标*MSIC*, *MSM*, *MSDD*值越大越好, *RISC*越小越好; 加粗数据代表在此项目下表现最好的结果

5.4 工具功能转换完备性实验

我们通过设置功能转换完备性实验 (MSA-Lab 工具与实验数据相关网址: https://gitee.com/liubo-rise/msa-lab), 来证明从单体到微服务的迁移不会造成功能损失. 我们将功能转换完备性定义为: 遗留系统中对象之间的每个方法调用路径 (Trace), 总是可以在功能上找到一个对应的微服务内部, 或微服务之间的可替换调用路径. 我们使用基于数据的分析 (Trace 覆盖) 而不是理论分析来证明功能转换完备. 我们通过使用 Trace 覆盖率指标来评估功能转换完备性. 它代表了遗留系统日志中的对象之间的调用路径 (对象序列图) 在多大程度上可以在工具生成的微服务调用路径 (微服务序列图) 中找到可替换调用路径.

为了更好地进行分析, 我们标出了每个方法所属的微服务序号 (如图 13(a) 所示). 并对微服务间的接口方法采用“!!!”的标记进行标注. 我们以 Trace 25 来说明如何对一个 Trace 进行功能转换完备性验证: 在对象序列图中, 用户调用的第 1 个方法和接下来的微服务间的方法都应该作为接口在微服务序列图中出现, 并且接口调用后的同属于一个微服务的所有方法应该都包含在此接口调用内 (因为在同一个微服务下发生, 且由接口调用触发). 图 13(a) 和图 13(b) 分别是 Trace 25 的对象序列图和微服务序列图, 左边的对象序列图中 removeItemFromChart 和 removeItemByID 这两个方法符合接口定义, 应该出现在微服务序列图中, 且 getItem 方法应包含在 removeItemByID 接口

调用内. 这两个接口分别属于 2 号微服务与 1 号微服务, 那么微服务序列图应该是用户调用 2 号微服务, 2 号微服务调用 1 号微服务. 在我们的拆分结果中 2 号微服务对应 CatologService, 1 号微服务对应 OrderService. 而图 13(b)的微服务图刚好符合上述描述的序列, 因此判定功能转换完备.



(a) 对象序列图 (b) 微服务序列图

图 13 Trace 25 对应的被标注对象序列图与微服务序列图

根据上述分析方法, 我们提取并使用代表性 Trace 集合 (第 3.2.2 节讲述) 作为输入并进行分析. 如表 8 所示, 可以证明在案例项目中工具迁移的功能转换完备性, 所有 Trace 覆盖率均为 100%.

表 8 功能转换完备性实验

项目名称	Trace数量	代表性Trace数量	Trace覆盖率 (%)
JPetStore	2 761	35	100
Zb-blog	8 307	76	100
Exam++	3 337	67	100
JeeSite	40 188	272	100

5.5 讨 论

上述实验和案例证明了本文的工具平台的实用性: (1) 能够与其他工具一样做到对遗留系统的有效拆分, 在模块度和接口复杂度上表现更好, 并且在拆分过程中不会发生功能的丢失. (2) 能在更短的时间内生成更多的微服务重构设计, 以及对应的微服务模型, 减少了人工建模的时间. (3) 即使在日志不足的情况下, 也能达到一定的鲁棒性与有效性, 通过添加新日志或人工调整微服务结构来快速进行弥补. (4) MSA-Generator 能够与 MSA-Modeller 能够相互配合, 提供模型驱动开发的集成微服务设计平台.

但是, 本文所进行的实验和工具平台依然存在一些局限性: (1) MSA-Generator 的日志解析功能只支持日志收集工具 Kieker 的日志解析, 而 Kieker 目前只支持 Java 系统. 应该添加其他语言的日志解析功能, 但分析动态方法调用序列的思想是通用的. (2) 自动生成的微服务模型和重构设计信息目前只来源于方法调用序列, 应添加相关的静态代码信息进一步补全模型, 如类中具体的属性, 方法. 进一步减轻人工建模的负担. (3) 微服务拆分算法的关键步骤在于对控制类的识别和控制类与其他类之前的关系的衡量. 案例中对于 JPetStore 这种结构清晰的小型系统, 工具平台能够清晰地划分出服务边界. 但对于 JeeSite 这种规模大且结构复杂的系统, 如果原始设计中某个类作为业务核心与多个类关联, 且不符合控制类的识别指标, 那么这些类将会被拆分到关联程度稍弱的其他微服务中. 而实际生产的系统设计可能更加复杂, 业务核心类具体有什么样的特征, 类与类之间的关系是否需要从静态结构、调用顺序等多方面去考虑也有待研究. 工具平台在这种情况下支持的模型和拆分算法是否满足生产需求仍然未知. 但从模型驱动的角度去重构微服务, 屏蔽了各种语言系统以及结构的差异, 通过模型驱动也能指导后续开发, 以及带来更多的自动化操作, 是一个值得尝试的方向.

6 总 结

基于遗留系统的微服务体系架构重构是实现单体系统到微服务系统迁移的关键, 目前大多数研究都集中在微服务识别方法上. 这些研究给出的设计方案缺乏对微服务之间和其内部以及动态行为的描述, 设计方案一旦生成就难以进行二次修改. 学术界和工业界缺乏一种能够基于遗留系统高效鲁棒地执行微服务体系架构重构设计的方法和支持工具. 为了解决以上问题, 我们在前期工作中主要完成了微服务体系架构设计建模的模型元素语法与语义的定义; 在此基础上, 我们进一步研发了一种模型驱动的、可用于单体遗留系统微服务化重构的集成设计平台 MSA-Lab. 它主要提供两个用于微服务及设计模型自动生成与交互式建模的核心功能部件, 即: 微服务及设计模型生成器 (MSA-Generator) 和微服务建模工具 (MSA-Modeller), 分别用于支持微服务自动识别与设计模型自动生成, 以及支持微服务静态结构模型 (微服务图) 与动态行为模型 (微服务序列图) 可视化展现、交互式建模、模型语法约束检验等. 通过对 4 个开源项目实施有效性与鲁棒性实验; 与 3 个相同特性的工具进行性能和拆分结果对比实验; 功能转换完备性实验. 结果表明: 本平台拥有很好的有效性、鲁棒性及实现面向日志的功能转换完备性, 且性能更加优越.

未来针对 MSA-Lab 平台的研发工作, 主要集中在强化 MSA-Generator 工具以支持更多类型的项目和使用不同语言开发的系统; 以及在 MSA-Modeller 中增加更多设计模型的支持、表达和语法检验. 此外, 为微服务元模型加入更严格的形式化语义以支持模型检测; 开发新的核心部件 MSA-Coder 以支持代码生成; 以及开发新的核心部件 MSA-Designer 以支持全新 (Greenfield) 项目的模型驱动开发也在我们未来的研发计划中.

References:

[1] Zimmermann O. Microservices tenets: Agile approach to service development and deployment. Computer Science-research and Development, 2017, 32(3): 301–310. [doi: 10.1007/s00450-016-0337-0]

[2] Sampaio AR, Kadiyala H, Hu B, Steinbacher J, Erwin T, Rosa N, Beschastnikh I, Rubin J. Supporting microservice evolution. In: Proc. of the 2017 IEEE Int'l Conf on Software Maintenance and Evolution. Shanghai: IEEE, 2017. 539–543. [doi: 10.1109/ICSME.2017.63]

[3] Lewis J, Fowler M. Microservices: A definition of this new architectural term. 2014. https://martinfowler.com/articles/microservices.html

[4] Taibi D, Lenarduzzi V, Pahl C. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. IEEE Cloud Computing, 2017, 4(5): 22–32. [doi: 10.1109/MCC.2017.4250931]

[5] Zhang H, Li SS, Jia ZJ, Zhong CX, Zhang C. Microservice architecture in reality: An industrial inquiry. In: Proc. of the 2019 IEEE Int'l Conf. on Software Architecture. Hamburg: IEEE, 2019. 51–60. [doi: 10.1109/ICSA.2019.00014]

[6] Parnas DL. On the criteria to be used in decomposing systems into modules. In: Broy M, Denert E, eds. Pioneers and Their Contributions to Software Engineering. Berlin: Springer, 1972. 479–498. [doi: 10.1007/978-3-642-48354-7_20]

[7] Mancoridis S, Mitchell BS, Chen Y, Gansner ER. Bunch: A clustering tool for the recovery and maintenance of software system structures. In: Proc. of the 1999 IEEE Int'l Conf. on Software Maintenance. Oxford: IEEE, 1999. 50–59. [doi: 10.1109/ICSM.1999.792498]

[8] Jin WX, Liu T, Cai YF, Kazman R, Mo R, Zheng QH. Service candidate identification from monolithic systems based on execution traces. IEEE Trans. on Software Engineering, 2021, 47(5): 987–1007. [doi: 10.1109/TSE.2019.2910531]

[9] Zhang YK, Liu B, Dai LY, Chen K, Cao XL. Automated microservice identification in legacy systems with functional and non-functional metrics. In: Proc. of the 2020 IEEE Int'l Conf. on Software Architecture. Salvador: IEEE, 2020. 135–145. [doi: 10.1109/ICSA47634.2020.00021]

[10] Selic B. The pragmatics of model-driven development. IEEE Software, 2003, 20(5): 19–25. [doi: 10.1109/MS.2003.1231146]

[11] Liu ZM, Chen XH. Model-driven design of object and component systems. In: Engineering Trustworthy Software Systems. Cham: Springer, 2016. 152–255. [doi: 10.1007/978-3-319-29628-9_4]

[12] Sorgalla J, Wizenty P, Rademacher F, Sachweh S, Zündorf A. AjiL: Enabling model-driven microservice development. In: Proc. of the 12th European Conf. on Software Architecture. Madrid: ACM, 2018. 1. [doi: 10.1145/3241403.3241406]

[13] Gysel M, Kölbener L, Giersche W, Zimmermann O. Service Cutter: A systematic approach to service decomposition. In: Proc. of the 5th IFIP WG 2.14 European Conf. on Service-oriented and Cloud Computing. Vienna: Springer, 2016. 185–200. [doi: 10.1007/978-3-319-44482-6_12]

[14] Petrasch R. Model-based engineering for microservice architectures using enterprise integration patterns for inter-service communication.

In: Proc. of the 14th Int'l Joint Conf. on Computer Science and Software Engineering. Nakhon Si Thammarat: IEEE, 2017. 1–4. [doi: 10.1109/JCSSE.2017.8025912]

[15] Kapferer S, Zimmermann O. Domain-driven architecture modeling and rapid prototyping with Context Mapper. In: Proc. of the 8th Int'l Conf. on Model-driven Engineering and Software Development. Valletta: Springer, 2020. 250–272. [doi: 10.1007/978-3-030-67445-8_11]

[16] Kalia AK, Xiao J, Lin C, Sinha S, Rofrano J, Vukovic M, Banerjee D. Mono2Micro: An AI-based toolchain for evolving monolithic enterprise applications to a microservice architecture. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 1606–1610. [doi: 10.1145/3368089.3417933]

[17] Liu B, Xiong JL, Ren QR, Tyszberowicz S, Yang Z. Log2MS: A framework for automated refactoring monolith into microservices using execution logs. In: Proc. of the 2022 IEEE Int'l Conf. on Web Services. Barcelona: IEEE, 2022. 391–396. [doi: 10.1109/ICWS55610.2022.00065]

[18] Santos N, Silva AR. A complexity metric for microservices architecture migration. In: Proc. of the 2020 IEEE Int'l Conf. on Software Architecture. Salvador: IEEE, 2020. 169–178. [doi: 10.1109/ICSA47634.2020.00024]

[19] Mazlami G, Cito J, Leitner P. Extraction of microservices from monolithic software architectures. In: Proc. of the 2017 IEEE Int'l Conf. on Web Services. Honolulu: IEEE, 2017. 524–531. [doi: 10.1109/ICWS.2017.61]

[20] Levcovitz A, Terra R, Valente MT. Towards a technique for extracting microservices from monolithic enterprise systems. arXiv: 1605.03175, 2016.

[21] Chen R, Li SS, Li Z. From monolith to microservices: A dataflow-driven approach. In: Proc. of the 24th Asia-Pacific Software Engineering Conf. Nanjing: IEEE, 2017. 466–475. [doi: 10.1109/APSEC.2017.53]

[22] Li SS, Zhang H, Jia ZJ, Li Z, Zhang C, Li JQ, Gao QY, Ge JD, Shan ZH. A dataflow-driven approach to identifying microservices from monolithic applications. Journal of Systems and Software, 2019, 157: 110380. [doi: 10.1016/j.jss.2019.07.008]

[23] Li SS, Rong GP, Gao QY, Shao D. Optimized dataflow-driven approach for microservices-oriented decomposition. Ruan Jian Xue Bao/Journal of Software, 2021, 32(5): 1284–1301 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6233.htm [doi: 10.13328/j.cnki.jos.006233]

[24] Abdullah M, Iqbal W, Erradi A. Unsupervised learning approach for Web application auto-decomposition into microservices. Journal of Systems and Software, 2019, 151: 243–257. [doi: 10.1016/j.jss.2019.02.031]

[25] Ding D, Peng X, Guo XF, Zhang J, Wu YJ. Scenario-driven and bottom-up microservice decomposition method for monolithic systems. Ruan Jian Xue Bao/Journal of Software, 2020, 31(11): 3461–3480 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6031.htm [doi: 10.13328/j.cnki.jos.006031]

[26] Rademacher F, Sachweh S, Zündorf A. Towards a UML profile for domain-driven design of microservice architectures. In: Proc. of the 2017 SEFM Int'l Conf. on Software Engineering and Formal Methods. Trento: Springer, 2017. 230–245. [doi: 10.1007/978-3-319-74781-1_17]

[27] Evans E. Domain-driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley Professional, 2003.

[28] Schneider M, Hippchen B, Giessler P, Irrgang C, Abeck S. Microservice development based on tool-supported domain modeling. In: Proc. of the 5th Int'l Conf. on Advances and Trends in Software Engineering. Varese, 2019.

[29] Terzić B, Dimitrieski V, Kordić S, Milosavljević G, Luković I. Development and evaluation of MicroBuilder: A model-driven tool for the specification of REST microservice software architectures. Enterprise Information Systems, 2018, 12(8–9): 1034–1057. [doi: 10.1080/17517575.2018.1460766]

[30] Tyszberowicz S, Heinrich R, Liu B, Liu ZM. Identifying microservices using functional decomposition. In: Proc. of the 4th Int'l Symp. on Dependable Software Engineering: Theories, Tools, and Applications. Beijing: Springer, 2018. 50–65. [doi: 10.1007/978-3-319-99933-3_4]

[31] Escobar D, Cárdenas D, Amarillo R, Castro E, Garcés K, Parra C, Casallas R. Towards the understanding and evolution of monolithic applications as microservices. In: Proc. of the 2016 XLII Latin American Computing Conf. Valparaiso: IEEE, 2016. 1–11. [doi: 10.1109/CLEI.2016.7833410]

[32] JPetStore. 2007. https://blog.csdn.net/pcfan1/article/details/1724521

[33] Deb K, Pratap A, Agarwal S, Meyarivan T. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. on Evolutionary Computation, 2002, 6(2): 182–197. [doi: 10.1109/4235.996017]

[34] van Hoorn A, Waller J, Hasselbring W. Kieker: A framework for application performance monitoring and dynamic software analysis. In: Proc. of the 3rd ACM/SPEC Int'l Conf. on Performance Engineering. Boston: ACM, 2012. 247–248. [doi: 10.1145/2188286.2188326]

[35] SkyWalking. 2017. https://skywalking.apache.org/

[36] Pinpoint. 2023. https://github.com/pinpoint-apm/pinpoint

[37] Li D, Li XS, Liu ZM, Stolz V. Interactive transformations from object-oriented models to component-based models. In: Proc. of the 8th Int'l Workshop on Formal Aspects of Component Software. Oslo: Springer, 2011. 97–114. [doi: 10.1007/978-3-642-35743-5_7]

[38] Carvalho L, Garcia A, Assunção WKG, de Mello R, de Lima MJ. Analysis of the criteria adopted in industry to extract microservices. In: Proc. of the 7th IEEE/ACM Joint Int'l Workshop on Conducting Empirical Studies in Industry (CESI) and the 6th Int'l Workshop on Software Engineering Research and Industrial Practice. Montreal: IEEE, 2019. 22–29. [doi: 10.1109/CESSER-IP.2019.00012]

[39] Athanasopoulos D, Zarras AV, Miskos G, Issarny V, Vassiliadis P. Cohesion-driven decomposition of service interfaces without access to source code. IEEE Trans. on Services Computing, 2015, 8(4): 550–562. [doi: 10.1109/TSC.2014.2310195]

[40] Mitchell BS. A heuristic search approach to solving the software clustering problem [Ph.D. Thesis]. Philadelphia: Drexel University, 2002.

[41] Hamers L, Hemeryck Y, Herweyers G, Janssen M, Keters H, Rousseau R, Vanhoutte A. Similarity measures in scientometric research: The Jaccard index versus Salton's cosine formula. Information Processing & Management, 1989, 25(3): 315–318. [doi: 10.1016/0306-4573(89)90048-4]

[42] Zb-blog. 2009. https://gitee.com/skyblue7080/zb-blog

[43] Exam++. 2009. https://gitee.com/ocelot/examxx/

[44] JeeSite. 2023. https://github.com/thinkgem/jeesite/

[45] Akoka J, Comyn-Wattiau I. Entity-relationship and object-oriented model automatic clustering. Data & Knowledge Engineering, 1996, 20(2): 87–117. [doi: 10.1016/S0169-023X(96)00007-9]

附中文参考文献:

[23] 李杉杉, 荣国平, 高邱雅, 邵栋. 一种优化的数据流驱动的微服务化拆分方法. 软件学报, 2021, 32(5): 1284–1301. http://www.jos.org.cn/1000-9825/6233.htm [doi: 10.13328/j.cnki.jos.006233]

[25] 丁丹, 彭鑫, 郭晓峰, 张健, 吴毅坚. 场景驱动且自底向上的单体系统微服务拆分方法. 软件学报, 2020, 31(11): 3461–3480. http://www.jos.org.cn/1000-9825/6031.htm [doi: 10.13328/j.cnki.jos.006031]

熊靖浏(1998－), 男, 硕士生, 主要研究领域为微服务, 软件工程.

刘志明(1961－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为高可信软件, 形式化方法, 人机物融合, 可解释的人工智能.

任秋蓉(1997－), 女, 硕士生, 主要研究领域为智能软件工程.

刘波(1981－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为模型驱动智能化软件工程, 可信微服务, 区块链系统.

Shmuel TYSZBEROWICZ(1951－), 男, 博士, 教授, 博士生导师, 主要研究领域为软件工程, 软件验证, CPS, 可信微服务系统.