软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(5):1322−1340 [doi: 10.13328/j.cnki.jos.006235]

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

基于多源特征空间的微服务可维护性评估*

晋武侠[1,2], 钟定洪[1,2], 张宇云[1,2], 杨名帆[2], 刘 烃[2]

[1](西安交通大学 软件学院,陕西 西安 710049)

[2](智能网络与网络安全教育部重点实验室(西安交通大学),陕西 西安 710049)

通讯作者: 晋武侠, E-mail: jinwuxia@mail.xjtu.edu.cn

摘 要: 软件企业实践将遗留软件系统解耦成基于微服务架构的系统,以提高软件的可维护性,达到较快市场交付.评估微服务开发阶段的代码可维护性是一个关键问题,其面临数据多源化、可维护性关注点多样化的难点.通过分析源代码、代码运行轨迹、代码修订历史,提出一种多源特征空间模型以统一表示软件多源数据,并基于该模型,从功能性、模块性、可修改性、交互复杂性等关注点度量微服务代码可维护性.据此实现了原型工具 MicroEvaluator,并在开源软件上进行了实验验证.

关键词: 微服务;可维护性;特征空间;可修改性;指标

中图法分类号: TP311

中文引用格式: 晋武侠,钟定洪,张宇云,杨名帆,刘烃.基于多源特征空间的微服务可维护性评估.软件学报,2021,32(5):1322−1340. http://www.jos.org.cn/1000-9825/6235.htm

英文引用格式: Jin WX, Zhong DH, Zhang YY, Yang MF, Liu T. Microservice maintainability measurement based on multi-sourced feature space. Ruan Jian Xue Bao/Journal of Software, 2021,32(5):1322−1340 (in Chinese). http://www.jos.org.cn/1000-9825/6235.htm

Microservice Maintainability Measurement Based on Multi-sourced Feature Space

JIN Wu-Xia[1,2], ZHONG Ding-Hong[1,2], ZHANG Yu-Yun[1,2], YANG Ming-Fan[2], LIU Ting[2]

[1](School of Software Engineering, Xi'an Jiaotong University, Xi'an 710049, China)

[2](Key Laboratory for Intelligent Networks and Network Security (Xi'an Jiaotong University), Ministry of Education, Xi'an 710049, China)

Abstract: Software industrial practices decouple legacy software systems into microservice architectures to improve software maintainability and to achieve faster market delivery. Evaluating microservice code maintainability during the development is a critical issue, facing the difficulties of multi-sourced data and diverse concerns of maintainability. By analyzing source code, code execution trace, and code revision history, a multi-sourced feature space model is proposed to unify the representation of software multi-sourced data. Based on this model, a microservice maintainability measurement system is established with comprehensive metrics, in terms of the

* 基金项目: 国家重点研发计划(2018YFB1004500); 国家自然科学基金(61632015, 61772408, U1766215, 61721002, 61833015, 62002280, 61902306, 61602369); 国网陕西省电力公司科技项目(5226SX1800FC); 教育部创新团队(IRT_17R86)和中国工程科技知识中心项目; 中国博士后科学基金(2020M683507, 2020M673439, 2019TQ0251); 西安市科协青年人才托举计划(095920201303); 西安交通大学基本科研业务费(xzy012020109)

Foundation item: National Key Research and Development Program of China (2018YFB1004500); National Natural Science Foundation of China (61632015, 61772408, U1766215, 61721002, 61833015, 62002280, 61902306, 61602369); Science and Technology Project of State Grid Shaanxi Electric Power Company (5226SX1800FC); Project of Ministry of Education Innovation Team (IRT_17R86) and China Knowledge Center for Engineering Sciences and Technology; China Postdoctoral Science Foundation (2020M683507, 2020M673439, 2019TQ0251); Youth Talent Support Program of Xi'an Association for Science and Technology (095920201303); Fundamental Research Fund of Xi'an Jiaotong University (xzy012020109)

本文由"面向持续软件工程的微服务架构技术"专题特约编辑张贺教授、王忠杰教授、陈连平研究员和彭鑫教授推荐.

收稿时间: 2020-09-16; 修改时间: 2020-10-26; 采用时间: 2020-12-15; jos 在线出版时间: 2021-02-07

concerns of functionality, modularity, modifiability, and interaction complexity. Accordingly, a tool prototype called MicroEvaluator is also implemented, and experimental analysis is carried out on open-source software systems.

Key words: microservice; maintainability; feature space; modifiability; metric

随着云计算技术的成熟和企业业务需求的增长,Netflix,Amazon 等已经实践将遗留的软件系统迁移为基于(微)服务的架构,以充分利用云基础设施,灵活进行业务扩张和性能伸缩,降低维护成本.与单体架构这种将系统各个模块统一管理打包成单个应用程序的范式相对,微服务架构由许多独立的服务组成,服务之间通过轻量级通信协议进行动态交互,每个服务应独立修改、开发、部署、维护[1].很多工作[2−7]研究遗留系统的微服务拆分(或者解耦)方法,但微服务拆分过程复杂、成本很高.获取更好的可维护性(maintainability),是企业愿意投入高成本进行微服务拆分的主要动力之一[8].

微服务可维护性描述了微服务为适应软件环境、需求等变化而相应修改时所作努力的程度,体现了微服务易于维护的程度,可包括开发阶段的代码可维护性(后续正文中出现的“可维护性”术语,如无特殊说明,则默认指代码的可维护性)和运维阶段可维护性(本文关注前者).调研发现:研究工作对可维护性的定义存在差异[9],也有一些研究工作度量可维护性,但没有显示使用“可维护性”术语.以 ISO 25010 质量模型(https://iso25000.com/index.php/en/iso-25000-standards/iso-25010)中可维护性的定义为标准,可维护性具体包括模块性、可复用性、易修改性等子属性.经典的内聚性和耦合性、自定义的实体收敛性和用例收敛性[10]、服务依赖其他服务或者被其他服务依赖的程度[2]等,都属于可维护性度量范围.

很多研究工作提出了微服务可维护性的评估方法.钟陈星等人[10]从领域驱动设计的视角,以用例图和实体关系图作为输入,评估拆分推荐的候选服务的内聚性、耦合性、用例收敛性、实体收敛性.Li 等人[11]和 Chen 等人[2]根据业务逻辑的数据流图,应用 Martin 等人[12]提出的 AfferentCoupling(*Ca*),EfferentCoupling(*Ce*)等面向对象设计评估指标来评估候选服务.Fritzsch 等人的研究工作[13]同样应用 *Ca*,*Ce* 等指标.Taibi 等人[14]以执行轨迹为输入,度量服务间的耦合度、服务的大小规模、服务外部调用频数等.Jin 等人[7]认为服务的接口暴露出该服务对外提供的功能逻辑,基于接口信息设计微服务度量指标.FoSCI[7]评估微服务的模块性、功能性、演化性.Bogner 等人[9]从复杂度、内聚度、耦合度等方面总结了微服务可维护性度量指标.

微服务代码可维护性度量受到了广泛研究,但仍然存在以下难点.(1) 数据异构多源化.微服务拆分过程使用数据流、软件结构、接口信息、运行轨迹等数据,这些数据来源于源代码、代码修订历史、代码的运行日志等不同制品.而不同源的数据存在异构特性,导致微服务代码可维护性度量很难同时考虑多源异构数据.(2) 代码可维护性关注点多样化.依据 ISO 25010 软件质量模型,可维护性体现为软件可被有效且高效修改的程度、可复用程度、模块化程度、在修改或者故障诊断时可高效且有效分析的程度等多个关注点.然而由于难以考虑多源异构数据,导致微服务代码可维护性度量也难以兼顾评估前述可维护性的各个关注点.

针对上述两个难点,本文将在前期工作[6]的基础上,提出一种基于多源特征空间的微服务代码可维护性评估方法:通过分析源代码的语法结构、代码文本信息、修订历史记录、运行轨迹等不同数据源,将这些多源异构数据统一建模为多源特征空间;基于多源特征空间模型,以 ISO 25010 软件质量模型中的可维护性定义为标准,从模块性、功能性、可修改性、交互复杂性等 4 个关注点来综合评估微服务代码可维护性.

据此实现了微服务可维护性评估的原型工具 MicroEvaluator,并在 JPetStore 和 Train-Ticket 两个开源系统上进行了初步实验.

本文第 1 节和第 2 节分别描述多源特征空间模型和可维护性评估指标.第 3 节展示实验及分析结果.第 4 节讨论相关研究工作.第 5 节总结本文工作并进行未来展望.

1 多源特征空间(MFS)建模

1.1 模型定义

本文在 Mo 等人[15]提出的软件特征空间基础上,将软件系统的特征扩展表示为多源特征空间(multisource

feature space,简称 MFS),多源特征空间包含了软件的设计元素(如实体、功能、执行行为)以及设计元素之间的二元关系.多源特性主要体现在:MFS 模型从软件系统的代码、文本、接口信息、修订历史、运行轨迹等多源数据中抽取出多样化的设计元素和二元关系.多源特征空间 MFS 表示为

$$MFS=\langle D,DSM,\sigma.(D,D)\rightarrow DSM\rangle \tag{1}$$

其中,D 表示设计元素的集合,$D=E\cup G\cup T.E$ 表示软件的实体元素的集合,实体元素包括方法/操作(*opr*)、类(*class*)、接口(*inf*)、文件(*file*),即 $E=ClassSet\cup OperationSet\cup FileSet\cup InterfaceSet$,且 $OperationSet=\{opr\}$,$ClassSet=\{class\}$,$InterfaceSet=\{inf\}$,$FileSet=\{file\}$;G 表示该软件所提供的功能集合,$G=\{g\}$;T 表示该软件的执行行为集合,$T=\{t\}$ 且 $t=\langle(opr_1,opr_2),(opr_3,opr_4),\ldots,(opr_{m-1},opr_m)\rangle$,$(opr_i,opr_j)$表示方法间的调用行为.

使用设计结构矩阵(design structure matrix,简称 DSM)表示设计元素之间的二元关系[16].DSM 以邻接矩阵的结构存储二元关系,矩阵中行和列分别对应一组设计元素,(i,j)单元格的值表示 i 行对应的设计元素与 j 列对应的设计元素之间的某种二元关系值.

映射函数σ计算每对设计元素的二元关系值,据此构建出相应的 DSM,第 1.2 节将详细介绍.

1.2 映射函数以及DSM生成

本节将介绍不同的二元关系映射函数σ及其相应的 DSM,具体包括 DSM_s、DSM_c、DSM_e、DSM_m、DSM_d、DSM_b 以及 DSM_p.

(1) DSM_s (structure DSM)

生成 DSM_s 的二元关系映射函数为 $\sigma_{structure}(class_i,class_j)$,该映射函数计算 $class_i$ 和 $class_j$ 在源代码中的结构依赖.通过分析源代码的抽象语法树来抽取类之间的继承、接口实现、调用、参数类型引用、返回值类型引用等结构依赖[10].如果从 $class_i$ 到 $class_j$ 存在上述任一种结构依赖,则 DSM 单元格(i,j)的值为 1;否则,值为空.

(2) DSM_c (concept DSM)

生成 DSM_c 的二元关系映射函数为 $\sigma_{concept}(class_i,class_j)$,该映射函数计算 $class_i$ 和 $class_j$ 在语义上的相似度.如果构成这两个类的文本标识符的功能逻辑词汇的交集不为空,则表示这两个类之间存在语义相似[8,17],在 *DSM* 中单元格(i,j)的值为 1;否则,值为空.

(3) DSM_e (evolution DSM)

生成 DSM_e 的二元映射函数为 $\sigma_{evolution}(file_i,file_j)$,该映射函数计算 $file_i$ 和 $file_j$ 在代码演化过程中的共同修改程度.参考 Mo 等人提出的方法[18],可将软件修订历史中 $file_i$ 和 $file_j$ 一起被修改的提交(commit)数可作为 *DSM* 单元格(i,j)的值.

(4) DSM_m (message DSM)

生成 DSM_m 的二元映射函数为 $\sigma_{message}(opr_i,opr_j)$,该映射函数计算 opr_i 和 opr_j 在消息粒度的相似度.参考 Mancoridis 等人的工作[19],消息粒度的相似度是 opr_i 和 opr_j 的输入消息(即参数)之间相似性和输出消息(即返回值)之间相似性的均值,计算公式如下:

$$\sigma_{message}(opr_i,opr_j)=\frac{1}{2}\left(\frac{|ret_i\cap ret_j|}{|ret_i\cup ret_j|}+\frac{|par_i\cap par_j|}{|par_i\cup par_j|}\right) \tag{2}$$

其中,ret_i 和 par_i 分别是 opr_i 的返回值集合和输入参数集合.

(5) DSM_d (domain DSM)

生成 DSM_d 的二元映射函数为 $\sigma_{domain}(opr_i,opr_j)$,该映射函数计算 opr_i 和 opr_j 在领域粒度的相似度.参考 Mancoridis 等人的工作[19],领域粒度的相似度等于 opr_i 和 opr_j 的方法签名之间的相似性,计算公式如下:

$$\sigma_{domain}(opr_i,opr_j)=\frac{|f_{term}(opr_i)\cap f_{term}(opr_j)|}{|f_{term}(opr_i)\cup f_{term}(opr_j)|} \tag{3}$$

其中,$f_{term}(opr_i)$表示操作 opr_i 的签名(method signature)所包含的领域词汇的集合.

(6) DSM_b (behavior DSM)

生成 DSM_b 的二元关系映射函数为$\sigma_{behavior}(t_i,(orp_i,opr_j))$,该映射函数计算 t_i 的运行轨迹中所包含的(opr_i,opr_j)数.t_i 是由方法间或者操作间的调用行为(opr_i,opr_j)组成的序列.使用 Kieker(http://kieker-monitoring.net/)等软件追踪工具监控软件系统运行获取到运行轨迹,追踪技术的原理是:对方法或者操作的入口和出口进行插装,执行到该方法或者操作的前后会记录下执行日志,再从执行日志中恢复出系统的执行轨迹.

(7) DSM_p (InteractionProbalistic DSM)

生成 DSM_p 的二元关系映射函数为$\sigma_{probability}(SC_i,SC_j)$,该映射函数计算在运行轨迹 T 中,服务 SC_i 参与的条件下,SC_j 也参与交互的概率.映射值越大,则两个服务同时参与执行交互的可能性越大,意味着两个服务耦合的程度越高.

通过挖掘 T 中服务调用的频繁模式和关联规则,计算$\sigma_{probability}(SC_i,SC_j)$,具体过程为:

对于每条 $t_i\in T,t_i=\langle(opr_1,opr_2),(opr_3,opr_4),\ldots,(opr_{m-1},opr_m)\rangle$,将方法层的调用行为映射到服务层,得到:$t_i'=\langle(SC_1,SC_2),(SC_3,SC_4),\ldots,(SC_{m-1},SC_m)\rangle$;$t_i'$ 作为一个事务,$T'=\{t_i'\}$,(SC_i,SC_j)是一个项,使用 FP-growth 算法挖掘 T' 中的频繁项和频繁规则,设置最小支持度为 $\min_{sup}$,最小置信度为 $\min_{cof}$;挖掘得出满足最小支持度和最小置信度的一组关联规则,记为 $Rules(\min_{sup},\min_{cof})=\{SC_i\rightarrow SC_j\}$;由关联规则集合生成 DSM,其中,DSM 的行和列对应服务 SC,$SC_i\rightarrow SC_j$ 这条关联规则的置信度$(cof_{i,j})$作为单元格(i,j)值,表示服务交互行为中 SC_i 参与的条件下,SC_j 也参与交互的概率 $cof_{i,j}$.

本文将基于包含上述 7 种 DSM 的 MFS 模型,设计微服务可维护性度量.其他应用可扩展定义 MFS 的设计元素和二元关系的映射函数σ,生成自定义的 MFS 模型.

2 基于 MFS 的可维护性评估方法

本节依 ISO 25010 软件质量模型中的可维护性定义,基于前述构建的 MFS,设计微服务可维护性评估指标.

ISO 25010 软件质量模型将可维护性定义为:“degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers”,具体体现在软件可被有效且高效修改的程度(modifiability)、可复用程度(resuability)、模块化程度(modularity)、在修改或者故障诊断时可高效且有效分析的程度(analyzability)等方面.遵循该标准,易于维护的微服务系统在理想情况下,其(物理或者候选)服务应具备良好的模块性、功能性、可修改性以及低交互复杂性.

- 模块性:服务的模块性是指服务遵循高内聚低耦合的程度.服务内部内聚程度越高且服务间的耦合程度越低,则该服务的模块性越好.这与 ISO 25010 中的“modularity”定义一致.
- 功能性:服务的功能性是指服务提供单一功能职责的程度.一个服务的功能越单一且一致,则该服务越易于应用到新系统、被新系统复用,提高新系统的开发效率.这体现了 ISO 25010 中“resuability”的目标.
- 可修改性:服务的可修改性是指服务可被独立修改的程度.一个服务内部的修改,在理想情况下,不应该影响到该系统的其他服务需随之修改.这种情况下,在一个系统的修改和演化过程中,只需要更改相关的局部服务即可.这与 ISO 25010 中“modifiability”的定义一致.
- 交互复杂性:服务的交互复杂性是指服务间动态交互行为的复杂程度.服务间的交互行为复杂,则发生故障后,很难有效在源代码中定位故障源头.低交互复杂度体现了 ISO 25010 中“analyzability”的目标.

上述模块性、可修改性、功能性、交互复杂性等 4 个方面从不同维度一致性描述微服务的代码可维护性.

本节将从上述 4 个关注点出发,评估微服务的可维护性.后续各个小节将列出不同关注点下具体的度量方法.为了便于清晰阐述,本文将一个微服务系统表示为 $System=\{SC\}$,其中,SC 表示一个微服务或者候选服务,$SC=(E_{ser},I,O)$.E_{ser} 表示服务 SC 内实体的集合;I 是 SC 的接口的集合;O 是细粒度的操作的集合,由接口 I 提供.实体是类 $class$;接口 inf 是服务对外公开发布的接口;通过发布的接口,服务提供对外部客户端可见的功能;操作 opr 是接口提供的公有方法.

2.1 模块性

模块性是指服务的内部实体应高度凝聚,跨服务边界的实体应松散耦合,即满足“高内聚、低耦合”的设计原

则.已有研究[20]发现:结构(structural)上的质量指标与概念(conceptual,也称为语义)上的质量指标存在互补关系,分别从源代码的语法结构、源代码中的文本语义信息等不同维度衡量软件质量.受此启发,我们基于 MFS 的 DSM_s 和 DSM_c,从结构和概念两个方面来扩展 Mancoridis 等人[19]定义的模块化质量 MQ,以度量服务模块化的程度.具体度量指标如下:

(1) 结构模块度(structural modularity,简称 SMQ)

SMQ 是指在结构维度上衡量服务的模块化程度.SMQ 值越大,说明服务的模块化程度越高.基于 DSM_s 的 SMQ 度量公式如下:

$$SMQ=\frac{1}{N}\sum_{i=1}^{N}scoh_i-\frac{1}{N(N-1)/2}\sum_{i\neq j}^{N}scop_{i,j} \tag{4}$$

$$scoh_i=\frac{u_i}{N_i^2},\ scop_{i,j}=\frac{v_{i,j}}{2(N_i\times N_j)} \tag{5}$$

其中,$scoh$ 度量服务的内聚程度,$scop_{i,j}$ 度量服务之间的耦合程度.这两个定义与文献[19]中定义的内部连通性和互连通性一致.基于 DSM_s 可统计出 u_i 和 $v_{i,j}$,u_i 是服务 i 内所有实体间边的数量,$v_{i,j}$ 是服务 i 内的实体与服务 j 内的实体之间的边的数量.N_i 或者 N_j 表示服务 i 或者服务 j 所包含的实体个数.$scoh$ 值越大,且 $scop$ 值越小,说明服务的结构模块性越好.

(2) 概念模块度(conceptual modularity,简称 CMQ)

CMQ 指在概念或者语义维度上衡量服务的模块化程度.CMQ 值越大,说明服务的模块化程度越高.基于 DSM_c 的 SMQ 度量公式如下:

$$CMQ=\frac{1}{N}\sum_{i=1}^{N}ccoh_i-\frac{1}{N(N-1)/2}\sum_{i\neq j}^{N}ccop_{i,j} \tag{6}$$

$$ccoh_i=\frac{u_i}{N_i^2},ccop_{i,j}=\frac{v_{i,j}}{2(N_i\times N_j)} \tag{7}$$

其中,$ccoh$、$ccop$ 的公式分别与 SMQ 中的 $scoh$,$scop$ 类似.不同的是,CMQ 是根据 DSM_c 统计出 u_i 和 $v_{i,j}$ 的值.

2.2 功能性

功能性是指服务应对外提供内聚一致的功能职责,即遵循“单一职责原则”(single responsibility principle,简称 SRP).服务发布的接口暴露出服务对外提供的职责[7],所以可根据服务的接口信息评估服务的功能性.基于 MFS 的 DSM_m 和 DSM_d,度量服务满足功能性的程度的指标如下.

(1) 接口数(InterFace number,简称 IFN)

ifn 度量服务的公开接口的个数.假设接口设计良好,那么 ifn 越小,说明该服务对外提供的接口数越少,服务对外提供的功能越有可能内敛一致,则服务粒度越小,越有可能具备单一职责.IFN 是所有服务 ifn 值的平均,形式化定义如下:

$$IFN=\frac{1}{N}\sum_{j=1}^{N}ifn_j \tag{8}$$

$$ifn_j=|I_j| \tag{9}$$

其中,I_j 是服务 j 的公开接口集合,N 是服务系统对外提供接口的服务数量.

(2) 接口消息层的内聚度(CoHesion at message level,简称 CHM)

chm 在消息层度量服务的公开接口的内聚程度.chm 值越大,说明对外提供的接口越一致、越内聚.CHM 是所有服务的内聚程度的均值.chm 是 Athanasopoulos 等人[21]提出的 LoC_{msg}(lack of message-level cohesion)的变形,即 $chm+LoC_{msg}=1$.基于 DSM_d 的 chm 计算公式如下:

$$CHM=\frac{1}{N}\sum_{j=1}^{N}chm_j \tag{10}$$

$$chm_j=\begin{cases}\sum\limits_{(k,m)}\dfrac{\sigma_{message}(opr_k,opr_m)}{\frac{1}{2}|O_j|(|O_j|-1)}, & \text{if } |O_j|\neq 1\\ 1, & \text{if } |O_j|=1\end{cases} \tag{11}$$

其中,$opr_k,opr_m\in O$,是接口 I 提供的不同操作,且 $k<m$;N 与 IFN 中 N 的含义相同;$\sigma_{message}$ 是生成 DSM_d 的二元映射函数,已在前述定义.

(3) 接口领域层的内聚度(CoHesion at domain level,简称 CHD)

chd 在领域层度量服务对外接口的内聚程度.chd 值越大,说明该服务提供的对外接口在领域上更加内聚.类似地,CHD 是微服务系统中所有服务的 chd 的均值.chd 是 Athanasopoulos 等人[21]提出的 LoC_{dom}(lack of domain-level cohesion)的变形,即 $chd+LoC_{dom}=1$.基于 DSM_d 的 chd 计算公式如下:

$$CHD=\frac{1}{N}\sum\nolimits_{j=1}^{N}chd_j \tag{12}$$

$$chd_j=\begin{cases}\sum\limits_{(k,m)}\dfrac{\sigma_{domain}(opr_k,opr_m)}{\frac{1}{2}|O_j|(|O_j|-1)}, & \text{if } |O_j|\neq 1\\ 1, & \text{if } |O_j|=1\end{cases} \tag{13}$$

其中,opr 和 O 的含义与在 chm 中的含义相同;σ_{domain} 是生成 DSM_d 的二元映射函数,已在前述定义.

2.3 可修改性

在理想情况下,一个服务可在不影响其他服务的情况下独立修改和演化,具有可修改性的服务可灵活适应未来的变化.基于 MFS 的 DSM_e,服务可修改性程度的度量指标如下.

(1) 服务内部共变频度(intra co-change frequency,简称 ICF)

icf 衡量候选服务内部的实体共同修改的频率.icf 值越高,意味着服务内的实体更有可能一起修改演化.ICF 是系统中所有 icf 的平均值:

$$ICF=\frac{1}{N}\sum\nolimits_{j=1}^{N}icf_j \tag{14}$$

$$icf_j=\frac{1}{|E_{ser_j}|}\sum_{m=1}^{|E_{ser_j}|}\frac{1}{|E_{ser_j}|}\sum_{n=1}^{|E_{ser_j}|}\sigma_{evolution}(c_m,c_n) \tag{15}$$

其中,$\sigma_{evolution}(c_m,c_n)$是实体 c_m 与实体 c_n 共同修改提交数,是 DSM_e 中的二元关系映射函数,已在前述定义;$c_m\in E_{ser_j},c_n\in E_{ser_j}$.如果 $m=n$,则 $\sigma_{evolution}(c_m,c_n)=0$.

(2) 跨服务共变频度(external co-change frequency,简称 ECF)

ecf 衡量分配在不同服务的实体却一起更改的频率.较低的 ecf 值意味着跨服务边界的实体更有可能独立修改.ECF 是系统中所有服务 ecf 的平均值:

$$ECF=\frac{1}{N}\sum\nolimits_{j=1}^{N}ecf_j \tag{16}$$

$$ecf_j=\frac{1}{|E_{ser_j}|}\sum_{m=1}^{|E_{ser_j}|}\frac{1}{|E^C_{ser_j}|}\sum_{n=1}^{|E^C_{ser_j}|}\sigma_{evolution}(c_m,c_n) \tag{17}$$

其中,ecf_j 计算服务 j 中的实体 $c_m\in E_{ser_j}$ 与其他实体 $c_n\in E^C_{ser_j}$ 的共同修改频度;$E^C_{ser_j}$ 与 E_{ser_j} 互为补集,$E^C_{ser_j}$ 是不属于 E_{ser_j} 的实体集合,即 $E^C_{ser_j}=U_{k=1,2,\ldots,N}E_{ser_k}$,$k\neq j$;$\sigma_{evolution}(c_m,c_n)$是 DSM_e 中的二元关系映射函数.如果 $|E_{ser_j}|=1$,则 $ecf_j=1$.

(3) 跨服务实体与服务内部实体共变频率的比值(ratio of ECF to ICF,简称 REI)

REI 衡量跨服务实体共同修改频度与服务内实体共同修改频度的比值.如果共同修改更频繁地发生在服务内部而不是服务间,则 REI 值会小于 1.0.值越小,说明不同服务一起修改的可能性越低,服务就更有可能会独立

演化、独立维护.理想情况下,所有共同修改都应发生在服务内部,而不应跨越服务边界:

$$REI=\frac{ECF}{ICF} \tag{18}$$

其中,ECF 和 ICF 与上述定义相同.

2.4 交互复杂性

交互复杂性描述服务之间交互行为的复杂程度,交互复杂度越高,则故障诊断、故障修复等分析活动的难度可能越高.基于 MFS 的 DSM_b 和 DSM_p,服务交互复杂度的度量指标如下.

(1) 处理用户请求的服务个数(inter-service length,简称 ISG)

ISG 是指处理用户请求时经过的不同服务个数,ISG 越大,则服务间的交互越复杂.

$$ISG=\frac{1}{|T|}\sum_{i=1}^{i=|T|}isg_i \tag{19}$$

其中,$|T|$是 MFS 中执行轨迹集合 T 的大小,isg_i 是 $t_i\in T$ 所经过的不同服务的个数.ISG 是所有执行轨迹的 isg 的均值.在 DSM_b 的第 i 行,统计单元格值不为空的所有列对应的操作调用(opr_k,opr_m),isg 值等于这些操作 opr 所属的服务集合的大小.

(2) 处理用户请求的跨服务调用链长(inter-service call chain length,简称 ICL)

处理用户请求时跨服务调用链长 ICL 越大,则服务间交互复杂度越高.ICL 的度量方式与上述 ISG 类似.不同的是,ISG 度量处理请求时所经过的不同服务的个数,ICL 度量处理请求时所经过的跨服务调用的个数.

$$ICL=\frac{1}{|T|}\sum_{i=1}^{i=|T|}icl_i \tag{20}$$

$$icl_i=|S|,S=\{(opr_m,opr_n)\} \tag{21}$$

其中,$|T|$与 ISG 中的定义相同,icl_i 是执行轨迹 $t_i\in T$ 所经过的跨服务调用的个数.在 DSM_b 的第 i 行,对于单元格值不为空的列所对应的操作调用(opr_m,opr_n),如果 opr_m 和 opr_n 是由不同服务提供的操作,则将(opr_m,opr_n)加入集合 S,最终生成的 S 大小就是 icl_i 值.ICL 是所有 icl_i 的均值.

(3) 处理用户请求的服务循环调用数(service cyclic-call number,简称 SCN)

服务循环调用是微服务设计的反模式[22,23].SCN 值越大,说明服务之间复杂交互程度越高,可分析性越差.基于 MFS 的 DSM_p,度量 SCN 的公式如下:

$$SCN=|S|,S=\{(i,j)|\sigma_{probability}(i,j)\neq None\wedge\sigma_{probability}(j,i)\neq None\wedge i<j\} \tag{22}$$

在 DSM_p 中,单元格(i,j)值$(i<j)$和其对称位置(j,i)值都不为空,则意味着服务 SC_i 和 SC_j 之间在很大概率上存在循环依赖,则将(i,j)加入集合 S.SCN 的值等于满足上述条件下生成的集合 S 的大小.DSM_p 刻画服务间频繁进行直接交互和间接交互的概率,因此,SCN 既考虑直接交互构成的循环调用,也考虑间接交互构成的循环调用.

(4) 被其他服务依赖的程度(in-degree dependence,简称 IDD)

IDD 度量一个服务被其他服务调用的程度.IDD 值越大,则被耦合的程度越高,服务之间动态交互程度高.基于 MFS 的 DSM_p,度量 IDD 的公式如下:

$$idd_j=\frac{1}{L}\sum_{i=1}^{i=L}\sigma_{probability}(i,j),i\neq j \tag{23}$$

$$IDD=\frac{1}{N}\sum_{j=1}^{N}idd_j \tag{24}$$

其中,L 表示第 j 列单元格值不为空的单元格个数.idd_j 表示一个服务 SC_j 被其他所有服务依赖的平均概率.N 是服务个数,IDD 是这些服务 idd_j 值的平均.

(5) 依赖其他服务的程度(out-degree dependence,简称 ODD)

ODD 度量一个服务调用其他服务的程度.ODD 的值越大,则该服务耦合其他服务的程度越高,服务之间动态交互程度高.与 IDD 类似,ODD 的度量也是基于 DSM_p:

$$idd_i = \frac{1}{R}\sum_{j=1}^{j=R}\sigma_{probability}(i,j), i \neq j \tag{25}$$

$$IDD = \frac{1}{N}\sum_{i=1}^{N} idd_i \tag{26}$$

其中,R 表示第 i 行单元格值不为空的单元格个数,idd_i 表示一个服务 SC_i 依赖其他所有服务的平均概率,N 是服务个数.IDD 是这些服务 idd_i 值的平均.

3 实 验

3.1 原型工具

根据本文提出的基于 MFS 的微服务可维护性评估方法,我们设计实现了相应的原型工具 MicroEvaluator.图 1 展示了 MicroEvaluator 评估微服务可维护性的处理流程,输入包括:分析对象的实现制品,即源代码、修订历史和运行轨迹;微服务解耦结果,即实体 E_{ser}、接口 I、操作 O.然后,使用第 1 节提出的 *MFS* 建模方法构建多源特征空间 MFS.接着,根据第 2 节提供的度量指标,以 *MFS* 和微服务的解耦结果 $System=\{SC\}=\{(E_{ser},I,O)\}$ 作为输入,计算微服务的可维护性度量指标,最后对度量结果进行可视化展示.MicroEvaluator 原型系统的具体实现和用户界面展示,详见 MicroEvaluator 的 Github 仓库(https://github.com/serviceassistor/MicroEvaluator/).

Fig.1 Process of microservice maintainability evaluation

图 1 微服务可维护性评估流程

3.2 实验设置

实验以 JPetStore 和 Train-Ticket 两个开源系统作为案例.JpetStore(https://github.com/mybatis/JPetStore-6)是在线宠物商店系统,提供用户管理、购物车操作、订单操作等功能,是单体架构的软件系统,常用于教学演示,也被一些研究工作[3,7]作为微服务拆分的实验对象.Train-Ticket[24]是基于微服务架构的火车票票务系统,由 41 个微服务构成,由多种编程语言开发,提供查询、预定、购买车票等功能,是微服务研究工作中常用的基准(benchmark)系统,其中,由 Java 实现的 37 个微服务将作为实验对象.

实验采用了参考文献[10]的指标合并方法,将不同的指标度量通过归一化处理以及加权求和,得出综合分数,比较不同微服务的可维护水平.即已知 n 个指标,归一化后的指标度量值为 m_1~m_n,权重设置为 w_1~w_n,并且 $\sum_{i=1}^{n} w_i = 1$,则综合评分为 $score = \sum_{i=1}^{n} m_i w_i$.*score* 值越高,则可维护性水平越高.实验设置每个指标的权重相同.

实验将依据微服务系统维护者的经验结果来评判本文方法有效性.为了尽可能减小人为因素带来的威胁和偏差,实验中,微服务维护者和参与实验评估人员为相同人员.因为系统维护者很了解微服务系统的设计目标,以他们的经验分析结果作为事实(ground-truth)来评估方法的有效性会更有意义.

3.3 数据收集

本文微服务可维护性评估方法的输入包括两类数据(如图 1 所示):一类是系统的微服务解耦结果(包括每个微服务的实体 E_{ser}、接口 I、操作 O);另一类是系统实现制品,包括源代码、修订历史、运行轨迹.下面将介绍 JPetStore 和 Train-Ticket 案例上这些数据的收集方式,并总结收集到的数据.

3.3.1 JPetstore 数据收集

(1) 获取候选微服务的 E_{ser}、I、O.

本文的两个作者各自独立拆分 JPetStore 生成相应的拆分方案 A 和 B 的 E_{ser} 结果.基于 E_{ser},采用 FoSCI[6] 中的接口识别方法生成相应的接口集合 I 和接口提供的操作集合 O,两种拆分方案生成的候选服务 SC 详情分别见表 1 和表 2.

Table 1 Service candidates from the decomposition *A* of JPetStore

表 1 JPetStore 拆分方案 *A* 下的候选服务

候选服务	E_{ser}	I	O
SC_0	org.mybatis.JPetStore.domain.Category org.mybatis.JPetStore.service.CatalogService org.mybatis.JPetStore.web.actions.CatalogActionBean org.mybatis.JPetStore.domain.Product org.mybatis.JPetStore.domain.Item org.mybatis.JPetStore.domain.Sequence	CatalogActionBean	*viewCategory*(·) *searchProducts*(·) *viewProduct*(·) *viewItem*(·)
SC_1	org.mybatis.JPetStore.domain.LineItem org.mybatis.JPetStore.web.actions.OrderActionBean org.mybatis.JPetStore.service.OrderService org.mybatis.JPetStore.domain.Order	OrderActionBean	*newOrder*(·) *isConfirmed*(·) *getOrder*(·) *newOrderForm*(·) *setOrderId*(*int*) *viewOrder*(·) *listOrders*(·)
SC_2	org.mybatis.JPetStore.domain.Cart org.mybatis.JPetStore.domain.CartItem org.mybatis.JPetStore.web.actions.CartActionBean	CartActionBean	*removeItemFromCart*(·) *updateCartQuantities*(·) *getCart*(·) *addItemToCart*(·)
SC_3	org.mybatis.JPetStore.service.AccountService org.mybatis.JPetStore.web.actions.AccountActionBean org.mybatis.JPetStore.domain.Account	AccountActionBean	*isAuthenticated*(·) *getUsername*(·) *setPassword*(·) *setUsername*(·) *newAccount*(·) *getAccount*(·) *signoff*(·)

Table 2 Service candidates from the decomposition *B* of JPetStore

表 2 JPetStore 拆分方案 *B* 下的候选服务

候选服务	E_{ser}	I	O
SC_0	org.mybatis.JPetStore.web.actions.CatalogActionBean org.mybatis.JPetStore.service.CatalogService org.mybatis.JPetStore.domain.Product org.mybatis.JPetStore.domain.Category	CatalogActionBean	*viewCategory*(·) *searchProducts*(·) *viewProduct*(·) *viewItem*(·)
SC_1	org.mybatis.JPetStore.domain.Item org.mybatis.JPetStore.domain.Cart org.mybatis.JPetStore.domain.CartItem org.mybatis.JPetStore.web.actions.OrderActionBean org.mybatis.JPetStore.service.OrderService org.mybatis.JPetStore.domain.Sequence org.mybatis.JPetStore.domain.Order org.mybatis.JPetStore.domain.LineItem org.mybatis.JPetStore.web.actions.CartActionBean	CartActionBean OrderActionBean	*removeItemFromCart*(·) *updateCartQuantities*(·) *getCart*(·) *addItemToCart*(·) *newOrder*(·) *isConfirmed*(·) *getOrder*(·) *newOrderForm*(·) *setOrderId*(*int*) *viewOrder*(·) *listOrders*(·)
SC_2	org.mybatis.JPetStore.service.AccountService org.mybatis.JPetStore.web.actions.AccountActionBean org.mybatis.JPetStore.domain.Account	AccountActionBean	*isAuthenticated*(·) *getUsername*(·) *setPassword*(·) *setUsername*(·) *newAccount*(·) *getAccount*(·) *signoff*(·)

(2) 获取源代码、修订历史、运行轨迹.

利用 Jin 等人[7]提供的功能测试案例驱动 JPetStore 运行,并使用 Kieker(http://kieker-monitoring.net/)工具监控获取执行轨迹.源代码和修订历史由原始的系统提供.基于原始系统的源代码、运行轨迹、修订历史,使用本文第 1 节提出的方法得出软件系统的多源特征空间.

3.3.2 Train-Ticket 数据收集

(1) 获取微服务的 E_{ser}、I、O.

使用 SCIToool Understand(https://scitools.com/)工具分析 Train-Ticket 的源代码,统计出每个微服务的实体 E_{ser}、接口 I 和操作 O,获取方法遵循本文第 2 节描述的微服务定义,该定义与文献[6]中的定义相同.表 3 展示了 Train-Ticket 微服务包含的实体数、接口数、操作数.可以观察到:构成 Tran-Ticket 的微服务规模比较微小,与微服务的设计理念一致.

Table 3 Statistics of microserivices in Train-Ticket system

表 3 Train-Ticket 系统的微服务统计

微服务名称	实体数$\|E_{ser}\|$	接口数$\|I\|$	操作数$\|O\|$
ts-admin-basic-info-service	10	1	21
ts-admin-order-service	8	1	5
ts-admin-route-service	7	1	4
ts-admin-travel-service	12	1	5
ts-admin-user-service	10	1	5
ts-assurance-service	11	1	9
ts-auth-service	20	2	6
ts-basic-service	14	1	3
ts-cancel-service	17	1	3
ts-notification-service	10	1	5
ts-config-service	8	1	6
ts-consign-price-service	8	1	5
ts-consign-service	10	1	6
ts-contacts-service	9	1	8
ts-execute-service	8	1	3
ts-food-map-service	12	2	7
ts-food-service	12	1	7
ts-inside-payment-service	22	1	9
ts-verification-code-service	6	1	2
ts-order-other-service	18	1	16
ts-order-service	18	1	16
ts-payment-service	10	1	4
ts-preserve-other-service	30	1	2
ts-preserve-service	31	1	2
ts-price-service	9	1	6
ts-rebook-service	21	1	3
ts-route-plan-service	15	1	4
ts-route-service	9	1	6
ts-seat-service	12	1	3
ts-security-service	12	1	6
ts-station-service	8	1	9
ts-ticketinfo-service	11	1	3
ts-train-service	8	1	6
ts-travel2-service	23	1	12
ts-travel-plan-service	17	1	5
ts-travel-service	23	1	12
ts-user-service	11	1	7

(2) 获取源代码、修订历史、运行轨迹.

Train-Ticket 项目提供了源代码和修订历史.实验部署好系统,按照系统维护者所共享的测试场景,人工操作用户界面,Train-Ticket 系统集成了 Jaeger(https://www.jaegertracing.io/)监控,从而可以获取运行轨迹.

3.3.3 数据总结

JPetStore 和 Train-Ticket 上收集到的数据汇总结果见表 4.JPetStore 有两种拆分方案:拆分方案 A 形成 4 个候选服务,拆分方案 B 形成 3 个微服务.从原始的单体 JPetStore 共收集到 47 条运行轨迹,每条运行轨迹对应一

个处理外部请求的方法执行序列.Train-Ticket 系统上共收集到 110 条运行轨迹.

Table 4 Summary of the collected data results

表 4 数据收集结果统计

	JPetStore	Train-Ticket
微服务个数	4(拆分方案 *A*),3(拆分方案 *B*)	37(Java 实现的微服务)
源代码行数 *LOC*	7 441	270 193
运行轨迹数	47	110
修订历史 *commit* 数	260	282

3.4 JPetStore实验分析

实验评估方案 *A* 和方案 *B* 中候选服务 $SC=(E_{ser},I,O)$的可维护性均值,度量结果见表 5.表中加粗单元表示方案 *B* 的相应度量值优于方案 *A* 的相应度量值.从表格数据可以看出,使用本文的可维护性评估方法得出的结论是:除了 SMQ 和 ECF 之外,方案 *A* 推荐的候选服务比方案 *B* 的结果更易于维护.

Table 5 Average Maintainability measurements of service candidates in two decompositions of JPetStore

表 5 JPetStore 两种拆分方案下候选服务的平均可维护性度量结果

拆分	*A*	*B*
Modularity:SMQ	0.427	**0.450**
Modularity:CMQ	0.411	0.338
Functionality:IFN	1.000	1.333
Functionality:CHM	0.640	0.636
Functionality:CHD	0.501	0.486
Modifiability:ICF	0.069	0.025
Modifiablility:ECF	**0.042**	0.037
Modifiability:REI	0.609	1.486
Interaction complexity:ISG	3.957	3.957
Interaction complexity:ICL	20.361	20.362
Interaction complexity:SCN	0	0
Interaction complexity:IDD	0.462	0.962
Interaction complexity:ODD	0.462	0.481

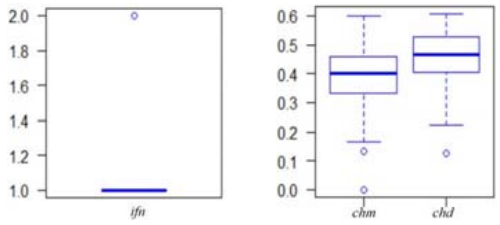
接下来分析上述结果是否与拆分人员的经验分析结果一致.我们与方案设计者进行了沟通:方案 *B* 设计者认为订单的查询、增加、删除操作与用户信息紧密绑定,因此将订单(order)逻辑与账户(account)逻辑拆分到同一个候选服务中;方案 *A* 的设计者认为订单逻辑和账户逻辑不同,因而拆分到不同的候选服务.双方讨论后一致认为:与方案 *B* 相比,方案 *A* 推荐的候选微服务更遵循单一职责原则,推荐的微服务更符合高内聚低耦合;候选服务更易于单独修改;由于方案 *B* 将两种功能逻辑合并在一起,与其他服务的动态交互都集中在该服务中,因此相应候选服务的交互复杂性相比于方案 *A* 偏高.因此,方案 *A* 比方案 *B* 推荐的候选服务更易于维护.将此人工经验分析结果与本文方法度量结果(见表 5)进行对比得出:本文方法的结论与微服务设计者的人工经验分析结果基本一致,一定程度上说明本文方法有效.

3.5 Train-Ticket实验分析

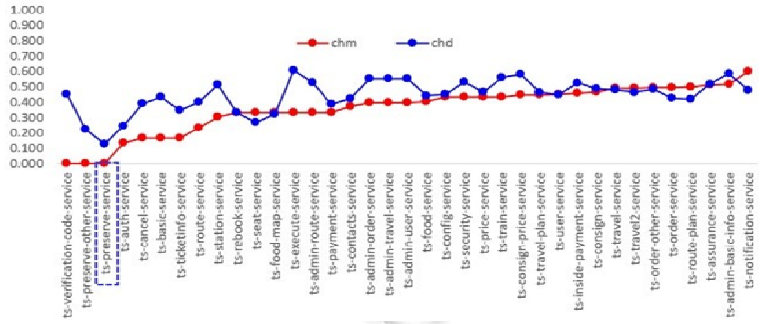
实验评估 Train-Ticket 系统中每个微服务的可维护性.图 2~图 6 展示了可维护性度量结果,其中,箱线图统计了度量值的分布,曲线图展示具体的度量结果.在曲线图中,图 2(b)、图 3(b)、图 5(b)的横坐标对应微服务,图 6(b)、图 6(c)的横坐标对应运行轨迹 trace.

图 2(a)可得:除 ts-auth-service 和 ts-food-map-service 的 *ifn*=2(即对外提供 2 个接口)之外,其他微服务的 *ifn*=2.在图 2(b)中,*chm* 和 *chd* 的中位数位于 0.4~0.5,存在小于 0.2 离散点值.结合图 2(c)得出,微服务 ts-preserve-service 的 *chm* 和 *chd* 度量值低于 0.2.

图 3(a)中 *icf* 和 *ecf* 值的分布显示:微服务总体上满足跨服务边界的共同修改频度低于服务内部的共同修改频度,即 *ecf*<*icf*.结合图 3(b)发现,ts-preserve-service 和 ts-travel2-service 的 *rei* 值大于 1,其他微服务的相应值均小于 1.这意味着:在所有微服务中,只有这两个微服务会更频繁与其他微服务发生共同修改.



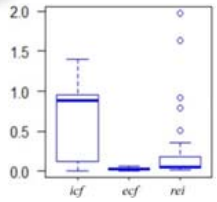
(a) *ifn* 值的箱线图 (b) *chd*,*chd* 值的箱线图



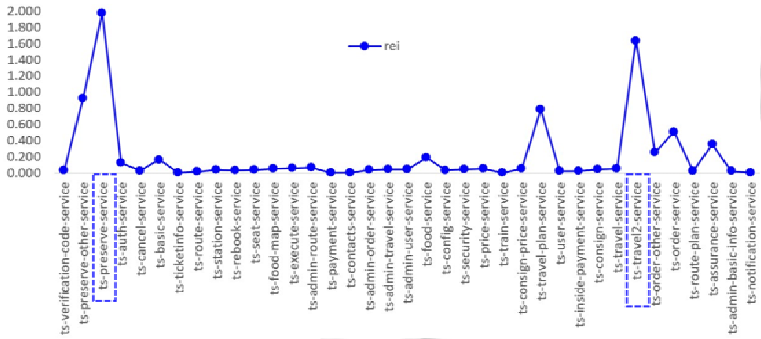
(c) *chm* 和 *chd* 值的曲线图

Fig.2 Measurements of *ifn*, *chm*, *chd* for microservices in Train-Ticket system

图 2 Train-Ticket 系统中微服务的 *ifn*、*chm*、*chd* 度量结果



(a) *icf*,*ecf*,*rei* 值的箱线图



(b) *rei* 值的曲线图

Fig.3 Measurements of *icf*, *ecf*, *rei* for microservices in Train-Ticket system

图 3 Train-Ticket 系统中微服务的 *icf*、*ecf*、*rei* 度量结果

为进一步观察上述跨服务修改,实验统计了修订历史记录中微服务对(microservice pair)的共同修改频数(co-change),该值是跨服务源代码文件共同修改数的累加,统计结果显示,*co-change* 均值为 6.65.同时,使用 Jplag (https://github.com/jplag/jplag)工具计算微服务对的代码相似度值.如图 4 所示:观察到,0.28%的微服务对的代码相似度大于 90%.分析 *co-change* 值和代码相似度得出:ts-preserve-service 与 ts-preserve-other-service 的 *co-*

change 值为 15,大于 95%的微服务对;且这对微服务的代码相似度值高达 96.4%.此外,ts-order-service 和 ts-order-other-service 的 *co-change* 值为 26,代码相似度为 94%;ts-travel-service 和 ts-travel2-service 的 *co-change* 为 10,代码相似度为 85.8%.可见,频繁发生跨服务修改的原因之一是微服务代码存在高度代码克隆:当一个微服务发生修改时,与之代码相似的微服务需同步更新,从而发生频繁跨服务修改,导致难以独立更新和演化.

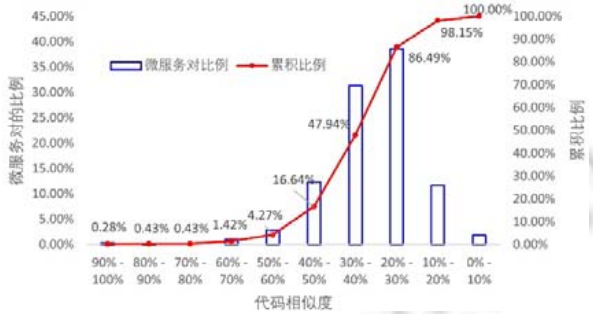
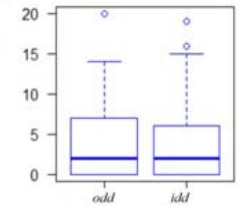


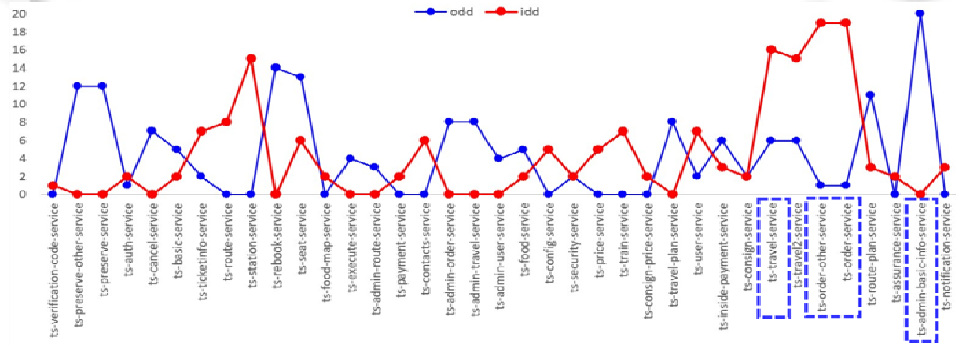
Fig.4 Statistical chart of the code similarity between microservice pairs in Train-Ticket system

图 4 Train-Ticket 系统中微服务间源代码相似度统计图

图 5 展示微服务 ts-travel-service、ts-travel2-service、ts-order-service、ts-order-other-service 的 *idd* 值相对偏高,意味着它们频繁被其他微服务调用来处理用户请求.ts-admin-basic-info-service 的 *odd* 值相对偏高,说明此微服务经常调用其他服务来完成功能.*SCN* 度量值为 2,即 2 对服务间存在循环依赖,分别是 ts-seat-service 和 ts-travel-service 间、ts-seat-service 和 ts-travel2-service 间.



(a) *odd*,*idd* 值的箱线图



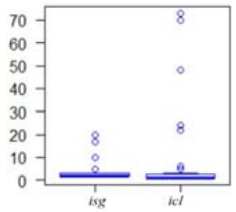
(b) *odd*,*idd* 值的曲线图

Fig.5 Measurements of *odd*, *idd* for microservices in Train-Ticket system

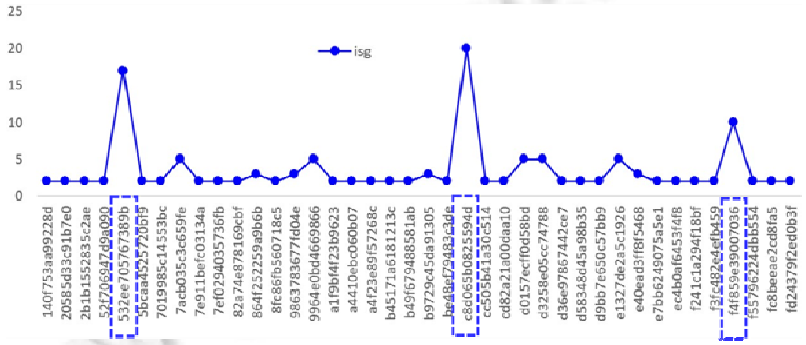
图 5 Train-Ticket 系统中微服务的 *odd*、*idd* 度量结果

图 6 展示了 *isg* 和 *icl* 度量值的分布和详情.在去冗余后的 40 条代表性的 trace 中,70%的 trace 的 *isg* 值等于 2、*icl* 值等于 1,即系统处理用户请求时常常经由 2 个微服务处理、平均涉及 1 次跨服务调用.但是,ID 为 532ee705767389b、f4f859e39007036、c8d065b0825594d 这 3 条 trace 的 *isg* 和 *icl* 值比其他 trace 明显高,说明

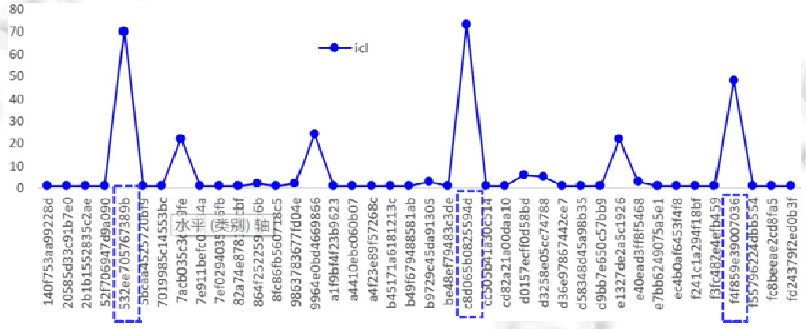
这 3 条 trace 对应的业务逻辑下的服务间的交互复杂度偏高.通过检查这 3 条 trace 数据,发现这 3 条 trace 主要处理用户的订票请求,由 ts-preserve-service 接受用户请求,检查安全性(调用 ts-security-service),查询确定行程信息(调用 ts-travel-service)、站点路径(调用 ts-station-service)、票价(调用 ts-ticketinfo-service)、座位(调用 ts-seat-service)、保险(调用 ts-assurance-service)、订餐(调用 ts-food-service)、托运行李(调用 ts-consign-service)等等.可见:ts-preservice-service 完成功能时涉及的业务逻辑比较复杂,须调用诸多其他微服务完成功能,因而与其他微服务间存在较高的交互复杂性.



(a) *isg*、*icl* 值的箱线图



(b) *isg* 值的曲线图



(c) *icl* 值的曲线图

Fig.6 Measurements of *isg*, *icl* for microservices in Train-Ticket system

图 6 Train-Ticket 系统中微服务的 *isg*、*icl* 度量结果

总结上述度量结果得出:在 Train-Ticket 系统中,与其他微服务相比,ts-preserve-service 的模块性和可修改性相对偏低(见 *chm*、*chd*、*rei* 值)、交互复杂性偏高(见 *isg*、*icl* 值);ts-travel2-service(ts-travel-service 与其类似)可修改性偏低(见 *rei* 值)、交互复杂性偏高(见 *idd*、*odd*、*SCN* 值);ts-order-other-service(ts-order-service 与其类似),ts-admin-basic-info-service 的交互复杂性偏高(见 *idd*、*odd*、*SCN*、*icl*、*isg* 值).

由于 *chm*、*chd*、*ifn*、*icf*、*ecf*、*rei*、*idd* 以及 *odd* 是针对每个微服务度量,而其他指标(如 *SCN*、*icl*、*isg*)的度量对象不是微服务,因此合并 *chm*、*chd*、*ifn*、*icf*、*ecf*、*rei*、*idd*、*odd* 的值,得出每个微服务可维护性的综合得分 *score*,结果如图 7 所示.可观察到:上述 ts-preserve-service、ts-travel2-service(ts-travel-service)、ts-order-

other-service(ts-order-service)等微服务的可维护性得分偏低,低于 0.62,这与上述指标合并之前的分析结果一致.ts-admin-basic-info-service 得分之所以偏高,是因为图 7 的 *score* 未考虑到 SCN、*icl*、*isg*.

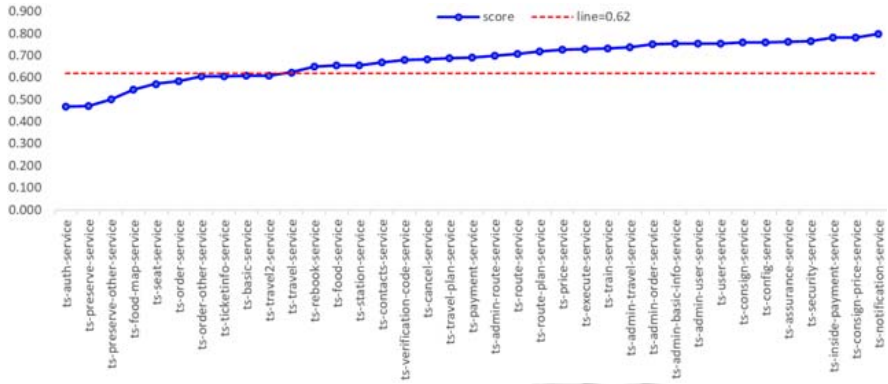


Fig.7 Comprehensive score results of microservices in Train-Ticket

图 7 Train-Ticket 微服务的综合得分(score)结果

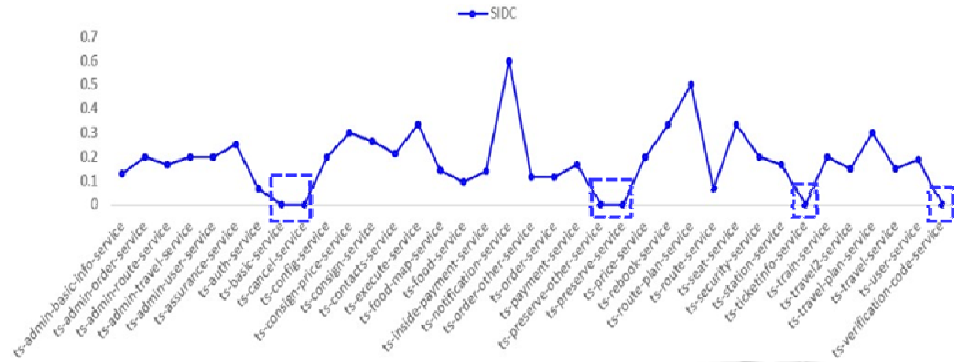
综上可得:ts-preserve-service、ts-travel2-service(ts-travel-service)、ts-order-other-service(ts-order-service)以及 ts-admin-basic-info-service 的可维护水平相对较低,所需投入的维护成本较高.实验揭示出可维护性低的一个原因是,相应微服务存在高度代码克隆.我们与 Train-Ticket 系统维护者沟通,他们确认了这一点,并阐述了存在微服务代码克隆的原因:设计时主要基于业务逻辑进行垂直拆分,且拆分粒度较细;比如,ts-travel-service 和 ts-travel2-service 的业务逻辑类似,因而代码高度相似,但两者针对的车型不同,ts-travel-service 针对高铁车型,而 ts-travel2-service 针对普快等车型.细粒度的微服务拆分确实会产生不可避免的代码克隆,导致微服务难以独立修改,此外,也会导致微服务间频繁发生交互、可维护性降低.但是,细粒度拆分会起到用户请求分流效果,提高系统的并发性能,可灵活分配硬件资源、按需伸缩不同业务的服务性能.可见,微服务设计决策难以同时满足高可维护性和高性能,因而需要根据目标折中考虑.未来我们将对此进行深入研究

3.6 方法比较

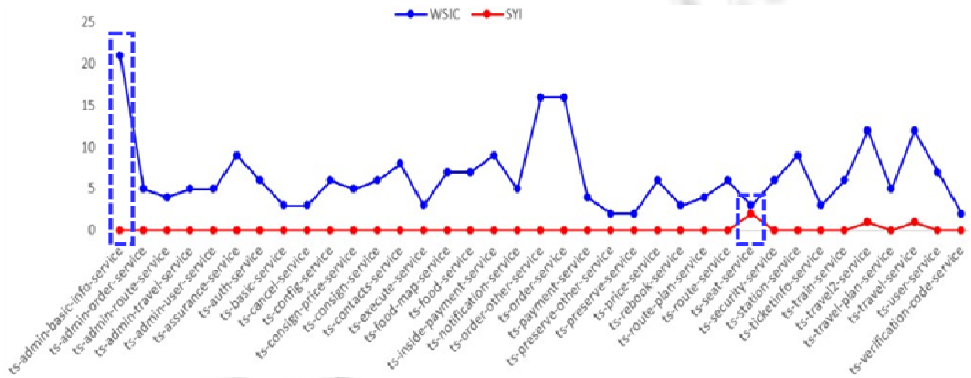
我们将本文提出的度量方法与文献[9]中的微服务可维护性度量指标进行比较.选择该文献中的指标作为本文方法的比较对象,是因为该文献明确指出:其指标是用于度量微服务的可维护性,其可维护性的定义同样遵循 ISO 25010 标准,详细解释了具有代表性的指标,包括 WSIC(weighted service interface count)、SYI(services independence in the system)、SIDC(service interface data cohesion).其中,WSIC 是服务提供的对外操作的个数的带权和,当权值默认为 1 时,WISC 值等于服务对外提供的操作个数和;SYI 是该微服务参与循环依赖的个数;SIDC 是服务提供的接口操作的参数类型的相似程度.WSIC 和 SIDC 是根据微服务提供的接口和操作进行度量;SYI 是根据微服务的源代码进行度量.WSIC 值越大、SYI 值越大、SIDC 值越小,则微服务的可维护水平越低.实验以 Train-Ticket 为对象,将本文所提方法与这 3 个可维护性度量进行对比.

图 8 绘制出 Train-Ticket 中微服务的 WSIC、SYI、SIDC 度量结果.从图中数据观察到:ts-basic-service、ts-cancel-service、ts-preserve-service(ts-preserve-other-service)、ts-ticketinfo-service、ts-verification-code-service 的 SIDC 值最小,ts-admin-basic-info-service 的 WSIC 值最大,ts-seat-service 的 SYI 值最大.也就是说,这些微服务比其他微服务的可维护性水平较低.

将上述结果与本文方法的结果(见第 3.5 节)进行对比得出:通过 WSIC、SYI、SIDC 识别出的维护性较低的微服务与本文方法识别结果存在交集,包括 ts-preserve-service(ts-preserve-other-service)、ts-admin-basic-info-service;但也存在一些差异.造成这个现象可能的原因是:WSIC、SYI、SIDC 是基于微服务的接口和代码计算而得;而本文可维护性方法不仅利用了接口和代码,也利用了运行轨迹、修订历史等其他制品,多源特征有助于更聚焦地定位出维护成本较高的微服务.我们下一步将在更多的实验对象上进行比较和验证.



(a) SIDC 值的曲线图



(b) WSIC 和 SYI 值的度量值

Fig.8 Measurements of SIDC, WSIC, SYI for microservices in Train-Ticket system

图 8 Train-Ticket 系统中微服务的 SIDC、WSIC、SYI 度量结果

4 相关研究工作

4.1 微服务拆分

微服务拆分将遗留的单体架构系统解耦成一组独立运行、以轻量级通信机制进行交互的规模相对较小的服务.根据拆分边界的选择方式分类,微服务拆分包括数据驱动的方法、结构驱动的方法、领域驱动的方法.

(1) 数据驱动拆分.这类方法倡导从软件系统的数据库或者数据流开始拆分.Levcovitz 等人[25]基于人工经验分析,将软件系统的数据库表根据领域知识划分成不同组,然后将访问同一组数据库表的代码类或者代码方法组织在一起,对应生成可选的候选服务.Chen 等人[2]认为数据流图体现了软件系统的业务逻辑,提出了基于数据流识别可能的候选服务,数据流由软件架构师或者开发人员绘制提供.丁丹等人[3]提出了一种场景驱动、自底向上的拆分方法,该方法从数据库拆分出发,向上映射到系统代码的拆分,考虑了数据关联度、数据共享度、拆分开销等不同拆分因素,在拆分迭代过程中融入用户反馈,这种半自动的方法推荐的拆分方案符合目标期望.

(2) 结构驱动拆分.还有一些研究基于软件依赖结构、运行轨迹等反应软件结构的信息来设计拆分方案.Gysel 等人[4]和 Mazlami 等人[5]抽取代码中的依赖结构信息,将软件系统建模表示成图结构,然后设计图切割算法切分图,每个子图则对应一个可选的候选服务.ServiceCutter[4]是一个服务拆分的可视化工具,用户自定义的一系列接口操作和用例作为拆分输入,基于模块化思想,根据操作间的耦合程度对接口进行聚类.Jin 等人[6]认为功能测试驱动下产生的运行轨迹能够反映系统的业务功能,提出了以单体系统的运行轨迹作为输入,设计一种基于多目标优化技术的服务拆分方法 FoSCI,该方法是对其

之前 FoME[7]方法的改进.与 FoSCI 方法类似,AMI[26]方法也是一种基于运行轨迹的多目标优化的拆分方法,不同的是,AMI 将负载均衡作为优化目标.

(3) 领域驱动拆分.领域驱动的软件设计方法提议围绕业务逻辑设计系统,通过业务分析建立领域模型.领域驱动设计将业务功能领域分解为限界上下文,每个限界上下文天然对应一个微服务.为了产生适当粒度的候选服务,通常需要对子领域进行合并或者拆分.Rademacher 等人[27]以 Cargo 领域模型为案例,探讨了领域驱动的微服务拆分面临的挑战.文献[8]调研了产业界的微服务迁移案例,发现多数参与调研的软件系统通过功能分解识别候选服务,而采用领域驱动拆分的实践相对较少.

(4) 其他方法.IBM Mon2Micro(https://www.ibm.com/cloud/blog/announcements/ibm-mono2micro) 与 FOSCI[6]方法、场景驱动[3]的方法类似,也是从单体系统的运行轨迹中抽取出系统业务逻辑.不同的是,Mon2Micro 进一步对源代码进行数据依赖分析,将数据依赖和业务逻辑同时输入到深度学习或者机器学习模型进行训练,将原始单体系统的类划分到不同的候选服务中.这种推荐结果期望尽可能减少微服务改造时的代码重写成本.

4.2 微服务评估

已有研究工作依据微服务解耦的不同目标分别展开了微服务质量评估研究,衡量产生的候选微服务或物理服务的合理程度.

(1) 可维护性评估

很多研究评估微服务的可维护性.文献[10]从领域驱动设计的视角,以用例图和实体关系图作为输入,评估候选服务的内聚性、耦合性、用例收敛性、实体收敛性.用例图描述了软件系统的业务逻辑,实体关系图描述了关键的领域实体以及实体间的关系.用例收敛性指标和实体收敛性指标的设计都遵循单一职责原则,分别度量单个服务聚焦于处理特定领域业务功能和处理特定领域实体的程度.文献[13]也将面向对象质量度量指标应用于微服务评估中.Taibi[14]提出以执行轨迹为输入,度量 Coupling BetweenMicroservice(CBM)、number of CLasses per Microservice(CLA)、number of DUPlicated classes(DUP)和 Frequency of External Calls(FEC).由于服务接口暴露出该服务对外提供的功能逻辑,FoME[7]根据接口信息来评估接口个数、接口内聚性、操作个数等,认为接口数和操作数越少且接口越内聚,服务的功能越内聚,就越有可能满足单一职责原则.FoSCI[6]从模块性、功能性、演化性这 3 个维度设计了 8 个指标评估拆分结果的合理性.功能性描述服务对外提供的功能,遵循服务的单一职责原则.模块性衡量同一服务内部的实体是否紧密一致,不同服务间的实体是否松散耦合,这是服务要满足的基本质量属性[28].可演化性[29]衡量服务独立演化的能力.一个基于服务的软件系统可能由几十甚至几百个服务构成,如果更改一个服务经常会影响到其他服务,则该服务很难进行故障分析,难以独立修改和演化.

(2) 其他质量属性评估

还有一些工作评估微服务的性能、可扩展性等其他质量属性.微服务系统是由很多规模小的服务通过轻量级通信协议进行动态交互来为用户提供服务,不同的拆分粒度会影响服务性能.文献[30]分析了微服务架构对性能带来的影响.也有一些工作遵循设计原则评估微服务,评价解耦推荐的微服务是否与之前计划的解耦设计原则保持一致.例如,文献[31]评估微服务架构是否符合服务粒度小、领域驱动设计等设计目标.Li 等人[11]和 Chen 等人[2]定性分析了候选服务的合理程度、易理解程度、客观程度.文献[32]通过挖掘应用程序访问日志,使用无监督的聚类方找出 URL 组,每个组内 URL 具有相似的性能和资源需求,因而每个 URL 组可映射生成一个微服务.该工作在真实案例上根据拆分方案实现了相应的微服务系统,分析了产生的微服务系统的性能和可扩展性.丁丹等人[3]对拆分结果使用人工评价,分析拆分方案符合期望的程度以及节省的手工拆分开销.

与上述研究工作不同,本文关注微服务的可维护性质量属性,基于软件的多源特征空间模型,从可维护性的 4 个角度综合评估微服务质量.

5 总结与展望

本文提出了一种基于多源特征空间的微服务可维护性评估方法,并据此实现了原型工具 MicroEvaluator,旨

在利用系统的源代码、修订历史、运行轨迹等多源数据,从模块性、功能性、可修改性、交互复杂性等角度评估微服务(包括候选服务)的可维护性.实验初步验证了本文方法的有效性.提供的方法和工具为微服务拆分设计决策提供一定参考,有助于识别潜在的存在可维护问题的微服务.

未来将扩展实验对象进一步验证本文方法,完善 MicroEvaluator 工具.其次,将分析同步和异步调用方式对微服务代码可维护性的影响.同步调用的实现方式相对简单,但会造成请求处理延时;异步调用实现方式相对复杂,但可减少处理延时.不同的同步调用(或异步调用)的实现范式可能也会对理解、调试、维护微服务代码带来不同影响.此外,本文方法目前只针对单体仓库(Mono-Repo,即所有微服务集中在同一个代码库上)的微服务系统,未来将研究多体仓库(Multi-Repo,即每个微服务有各自的代码库)微服务可维护性评估.

References:

[1] Ahmad A, Babar MA. A framework for architecture-driven migration of legacy systems to cloud-enabled software. In: Proc. of the WICSA 2014 Companion Volume. 2014. 1−8.

[2] Chen R, Li S, Li Z. From monolith to microservices: A dataflow-driven approach. In: Proc. of the Asia-Pacific Software Engineering Conf. (APSEC). IEEE, 2017. 466−475.

[3] Ding D, Peng X, Guo XF, Zhang J, Wu YJ. Scenario-driven and bottom-up microservice decomposition 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]

[4] Gysel M, Kölbener L, Giersche W, Zimmermann O. Service cutter: A systematic approach to service decomposition. In: Proc. of the European Conf. on Service-oriented and Cloud Computing. Springer-Verlag, 2016. 185−200.

[5] Mazlami G, Cito J, Leitner P. Extraction of microservices from monolithic software architectures. In: Proc. of the IEEE Int'l Conf. on Web Services (ICWS). 2017. 524−531.

[6] Jin W, Liu T, Cai Y, Kazman R, Mo R, Zheng Q. Service candidate identification from monolithic systems based on execution traces. IEEE Trans. on Software Engineering, 2019. Early Access. [doi: 10.1109/TSE.2019.2910531]

[7] Jin W, Liu T, Zheng Q, Cui D, Cai Y. Functionality-oriented microservice extraction based on execution trace clustering. In: Proc. of the IEEE Int'l Conf. on Web Service (ICWS). IEEE, 2018. 211−218.

[8] Fritzsch J, Bogner J, Wagner S, Zimmermann A. Microservices migration in industry: Intentions, strategies, and challenges. In: Proc. of the 2019 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). 2019. 481−490.

[9] Bogner J, Wagner S, Zimmermann A. Automatically measuring the maintainability of service-and microservice-based systems: A literature review. In: Proc. of theof the 27th Int'l Workshop on Software Measurement and 12th Int'l Conf. on Software Process and Product Measurement. 2017. 107−115.

[10] Zhong CX, Li SS, Zhang H, Zhang C. Evaluating granularity of microservices-oriented system based on bounded context. Ruan Jian Xue Bao/Journal of Software, 2019,30(10):3227−3241 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5797.htm [doi: 10.13328/j.cnki.jos.005797]

[11] Li S, Zhang H, Jia Z, Li Z, Zhang C, Li J, Gao Q, Ge J, Shan Z. A dataflow-driven approach to identifying microservices from monolithic applications. Journal of Systems and Software, 2019,157:Article No.110380. [doi: 10.1109/APSEC.2017.53]

[12] Martin RC, Martin M. Agile Principles, Patterns, and Practices in C#. Prentice Hall PTR, 2006.

[13] Jonas F, Bogner J, Zimmermann A, Wagner S. From monolith to microservices: A classification of refactoring approaches. In: Proc. of the Int'l Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment. 2018. 128−141.

[14] Taibi D, Systä K. A decomposition and metric-based evaluation framework for microservices. In: Proc. of the Int'l Conf. on Cloud Computing and Services Science. Springer-Verlag, 2019. 133−149.

[15] Mo R, Cai YF, Kazman R, Feng Q. Assessing an architecture's ability to support feature evolution. In: Proc. of the IEEE Int'l Conf. on Program Comprehension. ACM, 2018. 297−307.

[16] Baldwin CY, Clark KB, Clark KB. Design Rules: The Power of Modularity, Vol.1. MIT Press, 2000.

[17] Jin W, Cai Y, Kazman R, Zheng Q, Cui D, Liu T. ENRE: A tool framework for extensible eNtity relation extraction. In: Proc. of the IEEE/ACM 41st Int'l Conf. on Software Engineering: Companion Proceedings (ICSE-Companion). 2019. 67−70.

[18] Mo R, Cai Y, Kazman R, Xiao L, Feng Q. Decoupling level: A new metric for architectural maintenance complexity. In: Proc. of the Int'l Conf. on Software Engineering (ICSE). 2016. 499−510.

[19] Mancoridis S, Mitchell BS, Rorres C, Chen Y, Gansner ER. Using automatic clustering to produce high-level system organizations of source code. In: Proc. of the 6th Int'l Workshop on Program Comprehension (IWPC'98). IEEE, 1998. 45–52.

[20] Poshyvanyk D, Marcus A. The conceptual coupling metrics for object-oriented systems. In: Proc. of the IEEE Int'l Conf. on Software Maintenance. IEEE, 2006. 469–478.

[21] 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, 2014,8(4):550–562.

[22] Taibi D, Lenarduzzi V, Pahl C. Microservices Anti-patterns: A Taxonomy. In: Microservices. Springer-Verlag, 2020.

[23] Taibi D, Lenarduzzi V. On the definition of microservice bad smells. IEEE Software, 2018,35(3):56–62.

[24] Zhou X, *et al*. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Trans. on Software Engineering, 2021,47(2):243–260. [doi: 10.1109/TSE.2018.2887384]

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

[26] Zhang Y, Liu B, Dai L, Chen K, Cao X. Automated microservice identification in legacy systems with functional and non-functional metrics. In: Proc. of the 2020 IEEE Int'l Conf. on Software Architecture (ICSA 2020). IEEE, 2020. 135–145.

[27] Rademacher F, Sorgalla J, Sachweh S. Challenges of domain-driven microservice design: A model-driven perspective. IEEE Software, 2018,35(3):36–43.

[28] Newman S. Building Microservices: Designing Fine-grained Systems. O'Reilly Media, Inc., 2015.

[29] Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, Safina L. Microservices: Yesterday, today, and tomorrow. In: Proc. of the Present and Ulterior Software Engineering. Springer-Verlag, 2017. 195–216.

[30] Gribaudo M, Iacono M, Manini D. Performance evaluation of massively distributed microservices based applications. In: Proc. of the 31st European Conf. on Modelling and Simulation. European Council for Modelling and Simulation, 2017. 598–604.

[31] Engel T, Langermeier M, Bauer B, Hofmann A. Evaluation of microservice architectures: A metric and tool-based approach. In: Proc. of the Int'l Conf. on Advanced Information Systems Engineering. Springer-Verlag, 2018. 74–89.

[32] 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.

附中文参考文献:

[3] 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]

[10] 钟陈星,李杉杉,张贺,章程.限界上下文视角下的微服务粒度评估.软件学报,2019,30(10):3227−3241. http://www.jos.org.cn/1000-9825/5797.htm [doi: 10.13328/j.cnki.jos.005797]

晋武侠(1989－),女,博士,助理教授,CCF 专业会员,主要研究领域为软件分析,微服务,软件架构与质量.

钟定洪(1998－),男,硕士生,CCF 学生会员,主要研究领域为微服务,软件分析.

张宇云(1996－),女,硕士生,CCF 学生会员,主要研究领域为微服务.

杨名帆(1997－),男,本科生,主要研究领域为大数据处理.

刘烃(1981－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程,智能电网,AI 安全.