传统软件系统，大部分是单体系统，即所有代码被统一构建和部署，数据集中化管理。随着需求的变更和软件的演化，单体系统的规模不断变大、复杂度越来越高、使用的技术逐渐过时，导致维护成本大幅提升、交付周期延长。在过去几年里，大部分云计算服务的基础设施都具备了弹性、自恢复和可伸缩的能力，虚拟化和容器技术发展迅速，这些因素共同促进了微服务(Microservices)，一种区别于单体架构的软件架构风格被广泛接受和应用。

微服务将一个大型复杂的软件系统分解成一组独立运行、以轻量级通信机制进行交互的相对较小的服务[1]。这些服务围绕业务功能进行构建，可以独立开发、测试、部署和更新，以此带来的好处包括系统架构的解耦、单个服务的交付周期缩短、技术选型更灵活、可扩展性更好、复用性更高等等。Neal Ford 于 2018 年 12 月发布的“微服务成熟度状态”报告[2]显示，在调查涉及到的 866 个受访者中，超过 50%的人表示，他们的组织中有超过一半的新项目使用了微服务，近 70%的人使用了容器作为微服务的部署方式，86%的人认为他们的微服务项目已经取得了部分成功。这很大程度上说明微服务已经成为一种趋势，并成功运用于生产实践中。

微服务相对于单体架构有诸多优势，但将已有的单体系统拆分成微服务并不是一件容易的事情，Netflix 公司花费了 7 年时间才完成从单体到微服务的架构迁移[3]。Eric Evans 提出的“领域驱动设计”[4]方法，由于其所强调的限界上下文(Bounded Context)的概念能够自然地映射成微服务，被广泛用作微服务设计的指导原则。但这种方法缺乏正式的建模语言和工具支持[5]，要求架构师对业务领域有深入的了解，很难在短时间内得出合适的拆分方案。除此之外，实际的单体系统往往规模庞大、年代久远、实现与文档偏差大，导致原设计文档不可信、加大了人工分析的难度。

本文提出了一种半自动化的微服务拆分方法，通过监控典型应用场景测试用例的执行，获取对应的方法调用链、SQL 语句等信息，生成系统的数据访问轨迹图，再根据数据访问轨迹图对底层的数据表图进行加权和聚合，最后综合考量数据的内聚性和代码的重构开销这两方面因素，向用户推荐合适的自底向上的微服务化拆分方案。为了验证方法的可行性，本文改造了应用性能监控工具 Kieker[6] [7]，开发了原型工具 Cutter，同时增加了用户的反馈和迭代调整策略。原型工具在多个不同的单体系统上进行了实验，验证了这种方法能够快速地推荐一个较为合适的微服务拆分方案。

本文组织结构如下：第一节概述背景及相关工作，第二节详述方法步骤，第三节介绍原型工具，第四节通过实验验证方法工具的有效性，第五节总结全文。

1 相关工作

传统软件系统的模块化重构是软件工程研究领域的一个重要分支，早在 1972 年 Parnas 就提出了对系统进行模块化解构的标准[8]，随后围绕着信息隐藏、高内聚低耦合等原则，依赖静态结构分析、性能数据分析、仓库挖掘等手段，诞生了一系列软件模块化方法和工具[9][10][11][12]。然而，微服务的模块化和重构作为微服务实践的一大挑战[13]，目前仍然缺少实用的方法和工具支持[14]，相关研究主要集中在经验总结和方法论上。

Francesco 等人[15]对 18 位在工业界真实参与了微服务化迁移改造的工程师进行了访谈，将迁移过程分成了逆向工程、架构转移和前向工程三个子过程，分别总结了这三个子过程在具体实施时遇到的主要问题和挑战。Taibi 等人[16]则将微服务化改造过程分成三种类型，包括现有功能的迁移、重新开发以及通过“绞杀模式”实现新功能，并总结提出了一个迁移过程框架。Jonas 等人[17]在文献数据库中进行关键词搜索，筛选出十篇与微服务拆分相关的论文，总结其中涉及的重构方法，并分成了静态代码分析、元数据（架构描述、

UML 图等）导向、工作流数据（通信数据、性能检测数据等）导向以及动态微服务拆分（运行时环境、资源消耗等）四种类别，对重构方法的选择给予一定的决策指导。

在以“领域驱动设计”为指导原则的微服务拆分方面，前人已做出不少探索，但在设计之初需要投入较多精力进行业务分析和领域建模。Rademacher 等人[18]针对领域驱动设计中出现的细节缺失问题，提出了一种模型驱动的开发方法，通过引入中间模型，明确了微服务接口和部署细节。其中提到的 AjiL[19]工具可以用于微服务系统的可视化建模以及中间模型的生成。Levcovitz 等人[20]通过人工识别子系统、对数据库表进行分类，从而根据代码静态依赖图自底向上进行微服务划分。Chen 等人[21]分析业务需求、绘制数据流图，将具有相同输出数据的操作和对应数据划分为一个微服务。

单体系统的微服务化拆分也可以借鉴传统软件模块化的思路和方法，已有不少这方面的研究。Service Cutter[22]是一个用于服务拆分的可视化工具，输入是用户自定义的一系列接口操作和用例，根据操作间的耦合程度对接口进行聚类划分。但是 Service Cutter 要求用户编写所有输入文件、时间成本很高，并不适用于大型系统的改造。Abdullah 等人[23]则使用系统访问日志和无监督的机器学习方法将系统自动分解为具有相似性能和资源要求的 URL 组，将每组 URL 映射为一个微服务。这种方法将请求文档的大小和日志中的 URI 请求响应时间作为机器学习的输入特征，缺点在于只将性能相似度作为度量标准，没有考虑设计上的内聚性或耦合性，拆分结果不利于服务的修改和扩展。Mazlami 等人[24]通过分析代码和变更历史，根据代码的逻辑、语义等方面的关联，生成类之间的耦合关系图，最后对图进行聚类、得到微服务拆分方案。而 Jin 等人[25]认为程序的很多行为并没有显式反映在源码中，且代码级别的关联并不完全等同于功能的内聚，他们通过监控系统、动态收集代码的执行路径，实现了一种面向功能的微服务抽取方法。以上两种方法，分别从静态和动态两个角度对系统进行分析，再寻找恰当的聚类方法对系统进行以类为最小移动单位的拆分。

然而，单体系统的微服务化拆分存在不同于单体模块化的地方。Taibi 等人[26]就指出，从单体到微服务，应该是去识别可以从单体中隔离出来的独立业务流程，而不仅仅是对不同 Web Service[27]的特征提取；微服务之间的访问，包括私有数据和共享库，都必须被小心地分析。前文提到的工作，一部分缺乏对独立业务流程的识别，另一部分没有将数据作为重要的拆分对象、给予充分的关注。本文在这两点上做了尝试和结合，以现有的代码和数据模式为出发点，将业务流程、方法调用链和数据表三者进行关联，使得业务的独立性、代码的内聚耦合性、数据间的关联度都成为最终的拆分依据。同时，本文提出方法的自动化程度和推荐方案的完整度更高，减少了人工的输入数据量和分析时间。

2 场景驱动、自底向上的拆分方法

使用本文所提出的方法需要满足以下两点假设：

第一，待拆分的单体系统可分为用户界面、服务器端应用程序和数据库三个部分，需要进行微服务化拆分的是服务器端代码和数据库这两个部分，用户界面不做拆分，因为目前没有拆分前端界面的有效方法和工具。

第二，本方法认为大型系统是在较小的子系统基础上构建的，每个子系统都有一套定义明确的业务职责和单独的数据存储。这一点假设是为了保证引入场景概念和从数据库表开始拆分的合理性。

方法共分为场景驱动的数据访问轨迹图生成、拆分方案生成、反馈调整三个部分，整个过程是迭代可控的。

2.1 场景驱动的数据访问轨迹图生成

了解一个软件系统，最直接的方法就是去使用这个系统，以此快速掌握系统的主要功能和业务流程。绝大多数正规化开发的系统都有一套端到端的测试用例，保证每次系统修改后主业务功能的正确性。本文引入场景级别的测试用例，即每个用例(Use Case)对应一个用户使用级别的场景(Scenario)，这样的场景从用户角度出发，包含用户为了某个特定的目的而进行的一系列操作。这些操作大部分为界面操作，转化为一个或多个发往服务器端的请求(Request)。例如，一个管理员发通告的场景，包含了查询办公室列表、根据办公室查询用户信息、保存通告内容、刷新页面四个请求。每个请求在服务器端产生一条方法调用链(Trace)，由处理这个请求所调用的一系列方法(Method)组成，其中有些方法会执行 SQL 语句(SQL)、对数据表(Table)中的某些字段进行增删改查。图 1 的模型描述了上述概念和它们之间的关系。收集所有测试用例执行的方法调用链，并将他们与对应的 SQL 语句、数据表相关联，就形成了一张数据访问轨迹图。

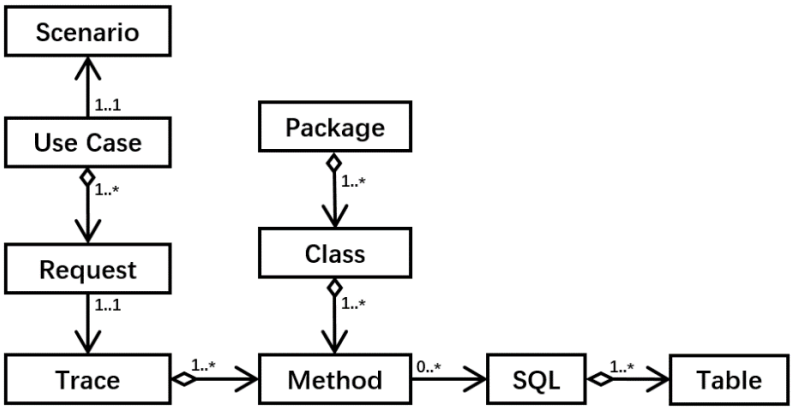

Fig. 1 Model of data access trajectory graph

图 1 数据访问轨迹图中各元素关系模型

2.1.1 输入准备

本文是通过在原单体系统运行测试用例的方法收集数据、构建数据访问轨迹图，所以需要准备以下输入信息：

(1) 测试用例：用户使用场景级别的测试用例，一个场景包含若干个界面输入或点击操作。用例数量没有限制，但至少覆盖所有主要业务流程和功能。测试既可以通过自动化的方式，也可以通过人工执行的方式实现。

(2) 用例权重：根据每个场景的重要度进行设置，默认全部为 1。对于一些比较重要或者使用频率很高的业务场景，例如火车站订票系统的订票场景、购物网站的下单流程，可以适当增加权重。

绝大多数情况下，获取以上信息所花费的时间和精力要远少于深入分析系统、构建领域模型，且一个对系统比较了解的开发、测试甚至系统管理人员就可以给出相对完整的信息，不涉及架构设计的相关知识。

2.1.2 数据访问轨迹图生成

输入测试用例和对应权重，通过监控系统的代码执行路径，生成如图 2 所示的数据访问轨迹图。其中每个场景(Scenario)由若干个请求(Request)串联而成，每个请求对应一条方法调用链，调用链的起点为服务端接收请求的第一个方法（一般情况下是控制层(Controller)方法）。调用链可能存在分岔情况，例如请求 2.1(Request 2.1)对应的调用链中，方法 5(Method 5)会先执行 SQL 3、对表 A(Table A)和表 C(Table C)进行操作，然后调用方

法 4(Method 4)、执行 SQL 2、对表 B(Table B)进行操作，所以请求 2.1 对应的调用链表示如下：

$$Trace\ 2.1{:}\ Method\ 2\ \rightarrow\ Method\ 5\ \begin{cases} \rightarrow SQL\ 3\ \rightarrow Table\ A、Table\ C \\ \rightarrow Method\ 4\ \rightarrow SQL\ 2\ \rightarrow Table\ B \end{cases}$$

每条调用链都有唯一的 ID 标识，即使同一场景下产生的两条调用链对应的代码执行路径完全相同（即所经过的方法和 SQL 语句及顺序完全相同），这两条调用链也是不同的。图中的场景和请求是抽象概念，最终以属性的形式存储在调用关系中。调用关系用箭头表示，箭头指向被调用的方法或 SQL 语句，箭头上的标号也是唯一的，包含所属调用链 ID 和调用序号。

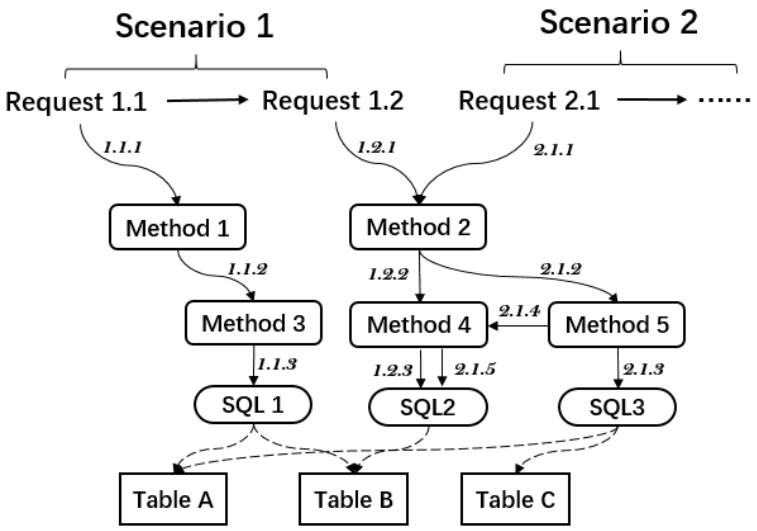

Fig. 2 Data access trajectory graph

图 2 数据访问轨迹图

2.2 拆分方案生成

依据上一步生成的数据访问轨迹图，根据微服务化拆分相关的多个维度信息，生成无向加权的数据表图，通过对数据表进行聚类划分、综合拆分开销，得到数据库拆分方案，然后在数据访问轨迹图中自底向上进行搜索、推荐与数据库拆分结果相符合的代码拆分方案。

2.2.1 考量维度

(1) 数据关联度：出现在同一个 SQL 语句、请求或者场景中的数据之间具有更加密切的关联关系，这种关联与相应 SQL 语句、请求或者场景的执行频率正相关，表示这些数据经常被一起操作。为了信息隐藏和减少数据耦合，这些数据表更应该划分到同一个微服务中。执行频率相同的情况下关联度的强弱：同 SQL>同请求>同场景。

(2) 数据共享度：被多个不同的场景、请求或 SQL 语句操作的数据，共享程度比较大。以系统的日志表为例，每当服务端进行一项数据操作后都会向日志中插入一条新的记录，这种情况下，日志和所有业务数据都有关联。但是从设计上来看，日志表不应该划分给任何一个业务服务，依据单一职责的原则，应该单独将其作为一个微服务提起出来。与关联度相反，对于共享度的影响：场景>请求>SQL。可以理解为，被多个场景共同访问的数据和代码更有可能是一个独立的业务功能。

(3) 拆分开销：由于本文关注的是已有单体系统的拆分，不仅要考虑微服务划分的合理性，也应该根据实际的拆分代价决定拆分粒度。一般情况下的开销：SQL 拆分>方法拆分>方法移动（即类的拆分）。

2.2.2 数据表图加权

从数据关联度和共享度这两个维度出发，将数据访问轨迹图转化成如图 3 所示的数据表图，图中每个顶点表示数据库中的一张表，根据以下方法向图中添加边，并计算权重：

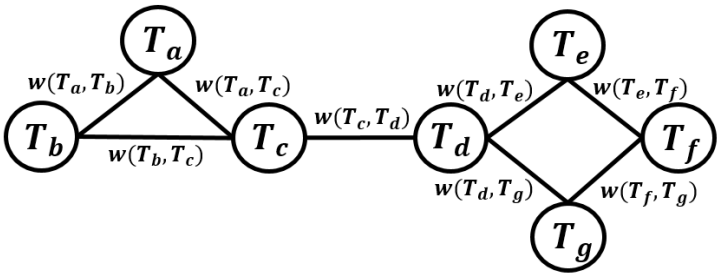

Fig. 3 Database table graph

图 3 数据表图

(1) 基础定义

a) 定义数据访问轨迹图中共包含$N_{Scenario}$个场景，N_{Trace}条方法调用链、N_{SQL}条 SQL 语句和n张表；

b) 第i个场景$Scenario_i$的权重定义为$W_{Scenario_i}$，由设计人员根据业务场景的重要程度于每个测试用例执行之前进行设置；

c) 第i条方法调用链$Trace_i$的权重定义为W_{Trace_i}，等于这条调用链所属的场景的权重；

d) 第i条 SQL 语句SQL_i的权重定义为W_{SQL_i}，等于每个场景的权重与其执行SQL_i次数的乘积的累加和，其中$T_{Scenario_k(SQL_i)}$定义为第k个场景执行SQL_i的次数：

$$W_{SQL_i} = \sum_{k=1}^{N_{Scenario}} W_{Scenario_k} \cdot T_{Scenario_k(SQL_i)} \qquad \text{(式 1)}$$

(2) 关联度计算

对于两个数据表T_a, T_b，从场景、方法调用链和 SQL 语句三个级别分别计算他们之间的关联度。

a) 定义$Scenario(T_i)$为操作了数据表T_i的场景的集合，场景级别的关联度表示为$C_{Scenario}(T_a, T_b)$，等于同时操作T_a, T_b这两张表的场景的累加权重与操作其中任意一张表的场景的累加权重之比：

$$C_{Scenario}(T_a, T_b) = \frac{\sum W_{Scenario \in Scenario(T_a) \cap Scenario(T_b)}}{\sum W_{Scenario \in Scenario(T_a) \cup Scenario(T_b)}} \qquad \text{(式 2)}$$

b) 定义$Trace(T_i)$为操作了数据表T_i的方法调用链集合，调用链级别的关联度定义为$C_{Trace}(T_a, T_b)$，等于同时操作T_a, T_b这两张表的调用链的累加权重与操作了其中任意一张表的调用链的累加权重之比：

$$C_{Trace}(T_a, T_b) = \frac{\sum W_{Trace \in Trace(T_a) \cap Trace(T_b)}}{\sum W_{Trace \in Trace(T_a) \cup Trace(T_b)}} \qquad \text{(式 3)}$$

c) 定义$SQL(T_i)$为操作了数据表T_i的 SQL 语句集合，SQL 级别的关联度定义为$C_{sql}(T_a, T_b)$，等于同时操作T_a, T_b这两张表的 SQL 语句的累加权重与操作其中任意一张表的 SQL 语句的累加权重之比：

$$C_{SQL}(T_a, T_b) = \frac{\sum W_{SQL \in SQL(T_a) \cap SQL(T_b)}}{\sum W_{SQL \in SQL(T_a) \cup SQL(T_b)}} \qquad \text{(式 4)}$$

将三个级别的关联度按下式加权累加，得到表T_a, T_b的关联度$C_{Total}(T_a, T_b)$：

$$C_{Total}(T_a, T_b) = \alpha_1 \cdot C_{SQL}(T_a, T_b) + \alpha_2 \cdot C_{Trace}(T_a, T_b) + \alpha_3 \cdot C_{Scenario}(T_a, T_b) \quad \text{(式 5)}$$

$$(\alpha_1 = 0.6,\ \alpha_2 = 0.3,\ \alpha_3 = 0.1)$$

(3) 共享群组的提取

共享度为单个数据表的特性，不能直接反映在边的权重上。本文首先识别出共享度较高的表、再根据这些表之间的依赖程度进行分类，最终得到若干个共享群组。每一个共享群组包含若干张共享表，且这些表之间存在较强的依赖关系，倾向于共同提取出来作为一个微服务。

对于第i张表T_i，定义其场景、调用链和 SQL 三个级别的共享度如下：

a) 场景级别的共享度$S_{Scenario}(T_i)$，等于操作表T_i的场景数量占总场景数量的比例：

$$S_{Scenario}(T_i) = \frac{|Scenario(T_i)|}{N_{Scenario}} \quad \text{(式 6)}$$

b) 调用链级别的共享度$S_{Trace}(T_i)$，等于操作表T_i的调用链数量占总调用链数量的比例：

$$S_{Trace}(T_i) = \frac{|Trace(T_i)|}{N_{Trace}} \quad \text{(式 7)}$$

c) SQL 级别的共享度$S_{SQL}(T_i)$，等于操作表T_i的 SQL 语句数量占总 SQL 语句数量的比例：

$$S_{SQL}(T_i) = \frac{|SQL(T_i)|}{N_{SQL}} \quad \text{(式 8)}$$

三个级别的共享度按下式加权累加，得到表T_i的共享度$S_{Total}(T_i)$：

$$S_{Total}(T_i) = \beta_1 \cdot S_{SQL}(T_i) + \beta_2 \cdot S_{Trace}(T_i) + \beta_3 \cdot S_{Scenario}(T_i) \quad \text{(式 9)}$$

$$(\beta_1 = 0.2,\ \beta_2 = 0.8,\ \beta_3 = 1)$$

计算每张表的共享度并从高到低排序，根据一定策略截取前x张表作为共享表。实际上，x的值和表总数n有一定关系，n越大，x也应该越大。假设共享表数量占总表数量的比例为y，通过定义一些特殊点（表 1）、获取x和y的拟合关系曲线（图 4）。

Table 1 Special points of shared table number - total table number's fitting curve

表 1 表总数-共享表占比拟合曲线特殊点

表的总数n	10	20	30	50	100	150 以上
共享表占比 y	0.3	0.25	0.2	0.15	0.1	0.08

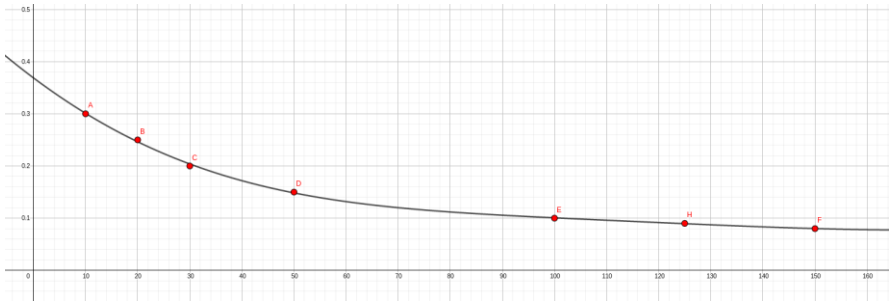

Fig. 4 shared table ratio - total table number's fitting curve

图 4 表总数-共享表占比拟合曲线

最终共享表的数量即为表总数与共享表占比之积，再取整：

$$x = \lfloor n \cdot y \rceil \qquad \text{(式 10)}$$

截取一定数量的共享表后，对这些共享表之间的依赖进行分析。共享表T_a对共享表T_b的依赖度分为 SQL 和调用链两个级别。

a) T_a对T_b的 SQL 依赖度定义为$D_{SQL}(T_a \to T_b)$，等于同时操作T_a, T_b这两张表的 SQL 语句数量占操作T_a的 SQL 语句数量的比例：

$$D_{SQL}(T_a \to T_b) = \frac{|SQL(T_a) \cap SQL(T_b)|}{|SQL(T_a)|} \qquad \text{(式 11)}$$

b) T_a对T_b的调用链依赖度定义为$D_{Trace}(T_a \to T_b)$，等于同时操作T_a, T_b这两张表的调用链数量占操作T_a的调用链数量的比例：

$$D_{Trace}(T_a \to T_b) = \frac{|Trace(T_a) \cap Trace(T_b)|}{|Trace(T_a)|} \qquad \text{(式 12)}$$

T_a对T_b的依赖度可以看作是T_a对T_b的“数据依附”关系，依赖度越高，这种依附关系越明显，T_a越倾向于和T_b在一起。但依赖度是双向的，如果T_a对T_b的依赖度很高，而T_b对T_a的依赖度很低，可以认为T_b和T_a有一种“主从”关系，而T_b与其它多个表之间也可能存在这种“主从”关系。比较典型的例子如图 5 所示，多个不同的业务数据表（如订单、购物车等）都保存有用户的 ID 外键，每次操作都会同时操作用户表、获取用户昵称或检查外键依赖，所以这些数据表会呈现出对用户表的强依赖，而用户表对这些表中的任一个依赖度很低。

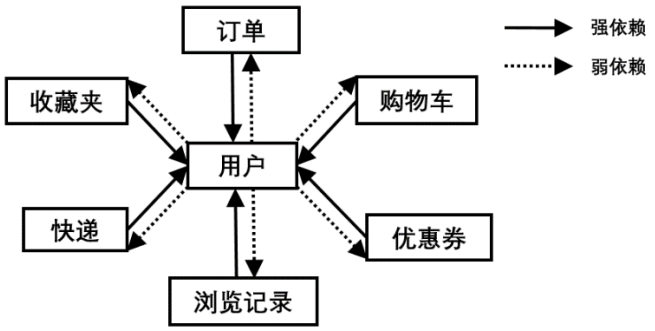

Fig. 5 Table dependency graph around user table

图 5 以用户表为中心的数据表依赖图

为了避免将以用户表为中心的多个不同子领域的表都划分到一起，只有当满足以下三个条件之一时，T_a和T_b才会被归入同一个共享群组：($\gamma_1 = 1.4,\ \gamma_2 = 1.6,\ \gamma_3 = 2.8$)

条件一：$D_{SQL}(T_a \to T_b) + D_{SQL}(T_b \to T_a) > \ \gamma_1$

条件二：$D_{Trace}(T_a \to T_b) + D_{Trace}(T_b \to T_a) > \ \gamma_2$

条件三：$D_{SQL}(T_a \to T_b) + D_{SQL}(T_b \to T_a) + D_{Trace}(T_a \to T_b) + D_{Trace}(T_b \to T_a) > \ \gamma_3$

根据以上条件，识别出的x张共享表最终分类成r个共享群组$G_1, \ldots\ldots, G_r$。

(4) 权重矩阵生成

根据式 5 计算n张数据表两两之间的关联度，得到$n \times n$的对称矩阵M，矩阵的第i行第j列元素m_{ij}等于$Table_i$与$Table_j$之间的关联度$C_{Total}(T_i, T_j)$，显然$m_{ij} = m_{ji}$。矩阵M对角线上元素值为 1，其他元素的取值区间为[0,1]。

遍历每个共享群组，针对第p个群组G_p中的共享表T_q，对矩阵M的第q行和第q列元素

做如下调整：

$$m_{qi} = m_{iq} = \begin{cases} \delta_1 & , T_i \in G_p \\ \delta_2 \cdot m_{qi}, & T_i \notin G_p \end{cases}, \ i = 1, \ldots\ldots, n \qquad \text{(式 13)}$$

$$(\delta_1 = 0.9, \ \delta_2 = 0.2)$$

式 13 的意义在于，对于连接同一个共享群组中两张表的边，赋予其较大权重δ_1，增加它们被划分到同一个微服务的概率；而与共享表相连的其它边的权重削减为原来的δ_2倍，降低该边相连的两张表被划分到同一个微服务的可能性。经过调整的矩阵M作为数据表图的邻接矩阵成为下一步图聚类算法的输入。

2.2.3 数据表图划分

图的聚类方法有很多种[28][29][30]，本文经过试验对比后选择了 Girvan-Newman 算法（下文称 G-N 算法）[31][32][33]用于数据表图的划分，最后被聚到一起的表就被划分到同一个微服务中。作为一种经典的社区发现算法，G-N 算法认为社区网络的重要特性是连接两个社团的边会有更高的权重，如果将这些边找出来并删除，剩下的网络就被很自然地划分为多个社团。为此，算法提出使用边介数(Betweenness)作为衡量一条边是否应该从图中删除的标准，一条边的边介数定义为网络中经过这条边的所有最短路径的数量。G-N 算法不断重复计算边介数与删除边介数最高的边这两个步骤，直到图中所有边都被删除。每次删边后算法都会重新计算社区结构，最终会生成一棵社区结构树（如图 6 所示），树的每一层对应一种对原网络的划分(Partition)，自根部向下，社区数量会越来越多，划分粒度越来越细，最后每个顶点都是一个独立的社区。这种连续的、粒度逐步细化的树形结构正好适应于微服务拆分过程，2.3 节所描述的反馈调整过程也是基于 G-N 算法的这种特性。

为了从多种不同粒度的划分方案中选择一个最合适的推荐方案，G-N 算法引入模块度(Modularity)[33]作为衡量标准，其定义如下式：

$$Modularity = \sum_{i=1}^{c}(e_{ii} - a_i^2) \qquad \text{(式 14)}$$

其中，e_{ii}表示社区i内所有边的权重占整个网络所有边权重的比例，a_i表示与社区i内顶点相连的所有边的权重占整个网络所有顶点所连边的权重的比例。模块度的取值范围为[-0.5, 1)，值越大，说明网络具有越强的聚类特性。遍历社区结构树的每一层，计算对应划分方案P_i的模块度，作为推荐方案的其中一个指标$Score_{Modularity}(P_i)$。整个数据表图的聚类划分过程如图 6 和算法 1 所示：

算法 1：数据表图聚类算法
输入：数据表图的邻接矩阵M **输出：**所有数据表的一个最佳划分
1: 根据邻接矩阵M构建图 G，$G.v$ 为图的点集，$G.e$ 为图的边集 2: $maxModularity$ //记录最大模块度 3: $maxBetweenness$ //记录最大边介数 3: $Map < Modularity, Partition >\ \ partitionMap$ //保存模块度和相应的划分结果 4: $while\ \ G.e \neq \emptyset$ 5: $\quad for\ \ edge\ \ in\ \ G.e$ 6: $\quad\quad$ 计算$edge$的边介数$edge.betweenness$ 7: $\quad\quad maxBetweenness = Max\ \ \{maxBetweenness,\ edge.betweenness\}$ 8: $\quad end\ \ for$ 9: $\quad for\ \ edge\ \ in\ \ G.e$ 10: $\quad\quad if\ \ edge.betweenness == maxBetweenness$

```
            remove edge from G.e //删除边介数等于最大边介数的边
        end if
    end for
    获取图G当前的划分P_current
    计算curPartition的模块度 Score_Modularity(P_current)
    maxModularity = Max{maxModularity, Score_Modularity(P_current)}
    partitionMap.put (curModularity, curPartition)
end while
```

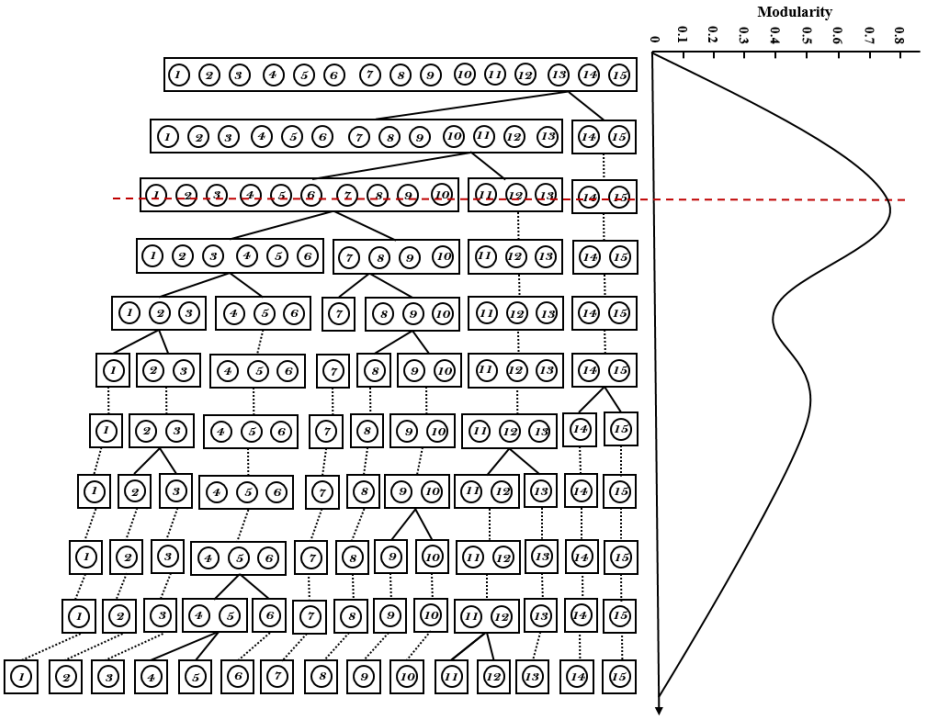

Fig. 6 Partition process of database tables

图 6 数据表划分过程

2.2.4 计算拆分开销

给定一个数据表划分方案P_i，可以依据原单体系统的静态代码结构，从 SQL 语句、方法、类三个层面自底向上进行拆分开销的计算。如果一条 SQL 语句操作的若干个数据表归属于两个及以上不同的微服务，则该条 SQL 语句需要进行拆分。一条 SQL 的拆分代价定义为μ_1，需要拆分的 SQL 总数定义为$V_{SQL}(P_i)$。同理，如果一个方法包含的 SQL 语句需要拆分或者该方法包含了两条属于不同微服务的 SQL 语句，那么它也需要进行拆分，对应的拆分代价定义为μ_2，需要拆分的方法总数定义为$V_{Method}(P_i)$。最后，除非一个类中所有方法都属于同一个微服务，否则需要进行类的拆分，拆分代价定义为μ_3，需要拆分的类的总数定义为$V_{Class}(P_i)$。拆分总开销$Cost(P_i)$通过下式计算($\mu_1 = 1,\ \mu_2 = 0.5, \mu_3 = 0.1$)：

$$Cost(P_i) = \ \mu_1 \cdot V_{SQL}(P_i) + \ \mu_2 \cdot V_{Method}(P_i) + \ \mu_3 \cdot V_{Class}(P_i) \qquad \text{(式 15)}$$

假设数据表图划分过程共产生了K种不同粒度的划分方案，给每个方案的拆分开销打分：

$$Score_{Cost}(P_i) = \frac{Max\{Cost(P_i), i=1,\ldots,K\} \ - \ Cost(P_i)}{Max\{Cost(P_i), i=1,\ldots,K\} \ - \ Min\{Cost(P_i), i=1,\ldots,K\}} \qquad \text{(式 16)}$$

最终的推荐方案倾向于减小拆分开销，所以将所有方案拆分开销的最大值与最小值之差作为分母，将最大值与当前方案的拆分开销之差作为分子，开销越大，分子越小，分数越低。

2.2.5 推荐方案

结合模块度和拆分开销计算拆分方案P_i的总分$Score_{Total}(P_i)$：

$$Score_{Total}(P_i) = \omega_1 \cdot Score_{Modularity}(P_i) + \omega_2 \cdot Score_{Cost}(P_i) \quad (式\ 17)$$
$$(\omega_1 + \omega_2 = 1)$$

按总分从大到小对所有划分方案排序，推荐分数最高的一个方案。

2.3 反馈调整

在确定最终拆分方案之前，允许用户对方法的某些参数和中间结果做出调整，使最终结果更符合用户需求：

活动图
最终方案包含

3. 系统实现

3.1 系统设计

图：架构图

3.2 原型实现

3.2.1 系统监控与数据处理

Kieker[6][7]是一个开源的应用性能监控工具，通过对目标系统进行动态代理，获取代码执行路径。监控信息被写入日志文件中，其中每一条记录表示一次方法调用，包含了被调用方法的签名、调用链 ID、调用时间、调用顺序和堆栈深度，利用这些信息可以还原一条完整的方法调用链。

为了将方法调用链和操作的数据表相关联，本文对 Kieker 进行了改造，不仅可以记录方法调用链，还可以记录数据访问层方法(DAO)执行的 SQL 语句。同时，使用 SQL 语法解析工具 JSqlParser[35]获取 SQL 语句操作的数据表名。

为了记录场景信息，Kieker 中新增页面如图 7 所示。在每个测试用例执行之前输入对应的场景名(Scenario Name)和权重(Scenario Weight)，整个用例执行期间生成的每条调用链都会带有这两项信息。

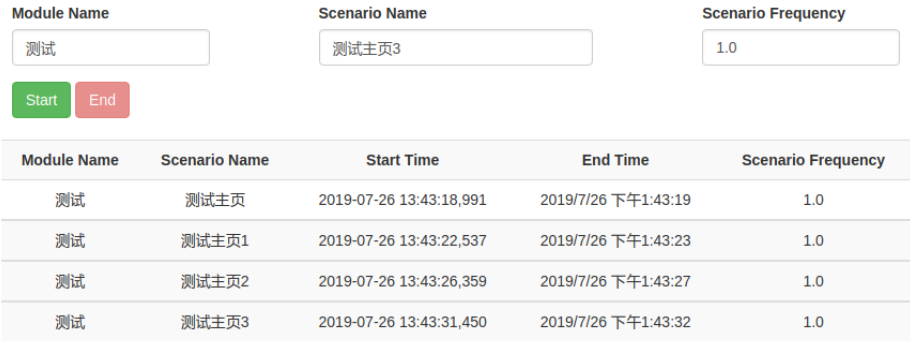

Fig. 7 Scenario input page after modifying Kieker

图 7 Kieker 改造后的场景输入界面

改造后的 Kieker 监控运行时系统并将执行日志写入日志文件中，通过对日志文件进行分析处理，构建数据访问轨迹图。为了方便之后的查看和搜索，将轨迹图连同项目的静态结构（包(Package)、类(Class)以及方法(Method)之间的包含关系）一起存储到图数据库 Neo4j[36]中，如图 8 所示。

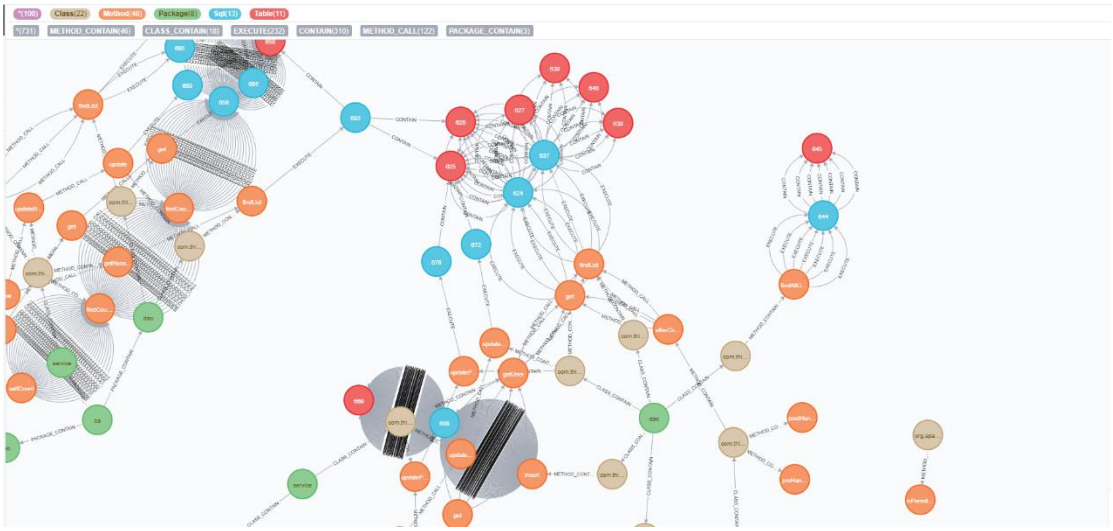

Fig. 8 Data access trajectory and project static structure in Neo4j

图 8 Neo4j 中存储的数据访问轨迹和系统静态结构

3.2.2 拆分工具实现

GN 算法实现：开源实现+模块度计算
截图、使用说明
最终方案包括…

4. 实验

本文实验的所有系统，分配给各个场景的权重取值范围为[1, 2]。
三个矩阵融合时：

$$\alpha 1 = 1, \quad \beta 1 = 0.5, \quad \gamma 1 = 0.2$$
$$\alpha 2 = 1, \quad \beta 2 = 0.8, \quad \gamma 2 = 0.2$$

5. 总结

意义、贡献

下一步工作

References:

[1] James Lewis, Martin Fowler. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html. 2014, accessed: 2019-07-21.

[2] Neal Ford. The State of Microservices Maturity. https://www.oreilly.com/programming/free/the-state-of-microservices-maturity.csp. 2018, accessed: 2019-07-23.

[3] Ruslan Meshenberg. Microservices at Netflix Scale: First Principles, Tradeoffs, Lessons Learned. https://gotocon.com/dl/goto-amsterdam-2016/slides/RuslanMeshenberg_MicroservicesAtNetflixScaleFirstPrinciplesTradeoffsLessonsLearned.pdf. accessed: 2019-07-23.

[4] Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Pearson Education, Upper Saddle River, 2003.

[5] Florian Rademacher, Sabine Sachweh, Albert Zündorf. Towards a UML Profile for Domain-Driven Design of Microservice Architectures. International Conference on Software Engineering & Formal Methods, Springer, 2017, pp. 230-245.

[6] Kieker. http://kieker-monitoring.net/. accessed: 2019-07-24.

[7] André van Hoorn, Jan Waller, Wilhelm Hasselbring. Kieker: a framework for application performance monitoring and dynamic software analysis. International Conference on Performance Engineering(ICPE), 2012, pp. 247-248.

[8] David Lorge Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. Commun. ACM, vol. 15, no. 12, pp. 1053-1058, 1972.

[9] Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison Wesley object technology series, Addison-Wesley 1999, ISBN 978-0-201-48567-7, pp. I-XXI, 1-431.

[10] Mainak Chatterjee, Sajal K. Das, Damla Turgut. WCA: A Weighted Clustering Algorithm for Mobile Ad Hoc Networks. Cluster Computing, vol. 5, no. 2, pp. 193-204, 2002.

[11] Periklis Andritsos, Vassilios Tzerpos. Information-Theoretic Software Clustering. IEEE Transactions on Software Engineering, vol. 31, no. 2, pp. 150-165, 2005.

[12] Yun Lin, Xin Peng, Yuanfang Cai, et al. Interactive and guided architectural refactoring with search-based recommendation. Acm Sigsoft International Symposium on Foundations of Software Engineering, 2016, pp. 535-546.

[13] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, et al. Microservices: The Journey So Far and Challenges Ahead. IEEE Software, vol. 35, no. 3, pp. 24-35, 2018.

[14] Claus Pahl, Pooyan Jamshidi. Microservices: A Systematic Mapping Study. International

Conference on Cloud Computing and Services Science, 2016, pp. 137-146.
[15] Paolo Di Francesco, Patricia Lago, Ivano Malavolta. Migrating Towards Microservice Architectures: An Industrial Survey. IEEE International Conference on Software Architecture. IEEE Computer Society, 2018.
[16] Davide Taibi, Valentina Lenarduzzi, Claus Pahl. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. IEEE Cloud Computing, 2017, vol. 4, no. 5, pp. 22-32.
[17] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, et al. From Monolith to Microservices: A Classification of Refactoring Approaches. CoRR abs/1807.10059, 2018.
[18] Florian Rademacher, Jonas Sorgalla, Sabine Sachweh. Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective. IEEE Software, 2018, vol. 35, no. 3, pp. 36-43.
[19] AjiL. https://github.com/SeelabFhdo/AjiL. accessed: 2017-07-24.
[20] Alessandra Levcovitz, Ricardo Terra, Marco Tulio Valente. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. CoRR abs/1605.03175, 2016.
[21] Rui Chen, Shanshan Li, Zheng Li. From Monolith to Microservices: A Dataflow-Driven Approach. IEEE, Asia-Pacific Software Engineering Conference (APSEC), 2017, pp. 466-475.
[22] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, et al. Service Cutter: A Systematic Approach to Service Decomposition. European Conference on Service-Oriented and Cloud Computing. Springer, 2016, pp. 185-200.
[23] Muhammad Abdullah, Waheed Iqbal, Abdelkarim Erradi. Unsupervised learning approach for web application auto-decomposition into microservices. Journal of Systems and Software, 2019.
[24] Genc Mazlami, Jürgen Cito, Philipp Leitner. Extraction of Microservices from Monolithic Software Architectures. IEEE International Conference on Web Services (ICWS), 2017, pp. 524-531.
[25] Wuxia Jin, Ting Liu, Qinghua Zheng, et al. Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering. IEEE International Conference on Web Services (ICWS), 2018, pp. 211-218.
[26] Davide Taibi, Valentina Lenarduzzi. On the Definition of Microservice Bad Smells. IEEE Software, vol. 35, no. 3, pp. 56-62, 2018.
[27] Web Service. https://www.w3.org/TR/ws-arch/. accessed: 2017-07-25.
[28] von Luxburg, Ulrike. A tutorial on spectral clustering. Statistics & Computing, 2007, vol. 17, no. 4, pp. 395-416. DOI: https://doi.org/10.1007/s11222-007-9033-z.
[29] Pascal Pons, Matthieu Latapy. Computing Communities in Large Networks Using Random Walks. Journal of Graph Algorithms and Applications, 2006, Vol. 10, no. 2, pp. 191-218.
[30] Vincent A. Traag. Faster unfolding of communities: speeding up the Louvain algorithm. CoRR abs/1503.01322 , 2015.
[31] M. Girvan, M. E. J. Newman. Community structure in social and biological networks. PNAS, 2001, vol. 99, no. 12, pp. 7821-7826.
[32] M. E. J. Newman. Fast algorithm for detecting community structure in networks, 2003. arXiv:cond-mat/0309508.
[33] M. E. J. Newman, M. Girvan. Finding and evaluating community structure in networks,

2004. arXiv:cond-mat/0308217.
[34]
[35] JSqlParser. http://jsqlparser.sourceforge.net/. accessed: 2017-07-28.
[36] Neo4j. https://neo4j.com/. accessed: 2017-07-28.
[37]
[38]