Computer Science and Application 计算机科学与应用, 2021, 11(12), 3060-3069
Published Online December 2021 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2021.1112309

Hans汉斯

云计算中基于数据流图的微服务划分方法

钱峥远[1,2]，王　顺[1,2]，曾国荪[1,2*]

[1]同济大学计算机科学与技术系，上海
[2]国家高性能计算机工程技术中心同济分中心，上海

收稿日期：2021年11月25日；录用日期：2021年12月21日；发布日期：2021年12月28日

摘　要

容器化微服务架构是当前云计算开发的主要形式，将业务功能合理地划分为多个微服务是先决条件。因此，本文开展微服务自动划分方法研究。以数据流图作为划分基础，定义图中的分支、存储交互和普通计算三类节点，增加对数据流的频度级以及长度级的描述，重点关注数据流图中“始和终”两两外部实体间的正常处理流程。通过对数据流图中的处理流程进行遍历搜索，得到高业务内聚性的潜在微服务划分，进一步对处理流程进行分割与合并，减少微服务划分的功能模块冗余和服务间通信开销，最后得到高内聚、低耦合、少冗余的微服务划分结果。ERP应用实例分析，验证了本文划分方法的正确性，微服务运行结果相较于单体服务性能上有明显的提升。

关键词

云计算，容器化微服务，数据流图，微服务划分

Data Flow Graph Based on Microservice Partitioning Method in Cloud Computing

Zhengyuan Qian[1,2], Shun Wang[1,2], Guosun Zeng[1,2*]

[1]Department of Computer Science and Technology, Tongji University, Shanghai
[2]Tongji Branch, National Engineering & Technology Center of High Performance Computer, Shanghai

Received: Nov. 25th, 2021; accepted: Dec. 21st, 2021; published: Dec. 28th, 2021

Abstract

Containerized microservice architecture is the main form of cloud computing development at present. Dividing the business functions into multiple microservices reasonably is a prerequisite. Therefore, this paper carries out a research on automatic microservice partitioning method. In this paper, based on the division of the data flow diagram (DFD), the method defines the store interac-

*通讯作者。

文章引用: 钱峥远, 王顺, 曾国荪. 云计算中基于数据流图的微服务划分方法[J]. 计算机科学与应用, 2021, 11(12): 3060-3069. DOI: 10.12677/csa.2021.1112309

tion nodes, branch nodes and common computing nodes. It also increases the description of data flows in DFD, which contains frequency level and the length level. The method uses data flow diagram and focuses on the normal processing flows between source and target external entities. By searching normal processing flows, the method can get potential microservice decomposition with high business cohesion. Then the method splits and merges the potential microservice decomposition to reduce functional module redundancy of microservices and communication overhead between microservices. Finally, the method gets the results of microservice partitioning with high cohesion, low coupling and little redundancy. Through the application verification on partitioning of basic ERP system, the correctness of the partitioning method in this paper and the high performance of microservice partitioning results are proved.

Keywords

Cloud Computing, Containerized Microservices, Data Flow Diagram, Decomposition of Microservice

1. 引言

近年来，随着容器技术、集成开发、测试、部署和运营为一体的 DevOps 等技术的兴起和发展，基于微服务的开发方法越来越受到重视[1]。传统单体架构将应用的所有功能集中在一个单独的应用软件中，然而随着功能的不断增多，软件项目越来越大，开发人员要弄清代码逻辑费时费力，功能相互耦合，使得系统难以更新和维护[2]。此外，单体应用架构系统部署不灵活，构建时间长，稳定性不高[3]，难以适应请求经常发生变化的情况。因此，软件设计人员提出了微服务结构，其基本思想是将传统的单体应用按业务功能拆分为一系列可被独立设计、开发、部署、运维的细粒度服务单元，服务间彼此配合、相互协作以实现所有功能[4]。相较于单体应用架构，微服务架构有着易于开发和维护、单个微服务启动较快、部署灵活、测试方便、技术栈不受限、便于按需伸缩，系统容错率高等优点，因此越来越多的云平台系统开始向微服务架构迁移。但是，想要使用微服务架构首先必须对系统中的软件服务进行划分，若划分不合理则微服务架构系统性能甚至比不上传统单体架构。通常，微服务划分应满足如下要求：1) 微服务的规模应该足够小，并专注于实现一个功能，微服务之间相对独立并保持松散耦合[5]。2) 微服务分解的时候应该考虑系统要求、安全性和可扩展性[6] [7]。

当前国内外围绕微服务的划分研究主要可以分为三大类：1) 对源代码静态分析后的划分方法：例如 Daniel Escobar 等[8]，通过处理 Java Enterprise Edition (JEE)应用程序的源代码，抽象出具有完整抽象语法树的应用程序模型，分析代码类间关系，对程序模型中的模块进行划分，但这种方法仅能使用在 JEE 程序上，且需要已有代码实现；再例如 Gerald Schermann 等将微服务的大小与代码行数联系起来[9]，通过计算一个微服务代码行数阈值 LOC，来指导微服务的分割，却没有考虑到不同技术栈带来的系统实现代码行数不同的问题；Baresi 等[10]则通过接口上的语义相似度聚类接口，根据聚类结果来决定适合的微服务划分，但是只适用于已经实现单体应用的情况，并不支持直接设计微服务架构应用的情况。2) 依据单体应用运行的历史数据进行划分：例如 Tugrul Asik 等人[11]提出通过计算一个微服务与其他微服务、外部服务或客户交互的资源量来判断一个微服务是否过大或过小，但是这种方法没有考虑不同的业务本身导致的交互资源量差异；Mazlami 等人[12]通过分析代码语义和历史更新记录等方面的关联，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，但该方法需要历史的代码变更信息，因此适用范围有所限制。3) 领域业务逻辑驱动的微服务划分方法，例如 Paulo Merson 等[13]使用领域驱动设计创建的域模型设计了同步(基于 rest)和异步(响应)的微服务，并探索了多种不同的微服务设计场景并针对每个场景提出了划分指导；钟陈星等[14]则通过描述候选微服务中服务与用例、实体之间的对应关系，规定了每个服务负责的领域业务逻辑和相关限界上下文，再根据服务内聚性、服务耦合性、用例收敛性等量化指标，指导开发者迭代优化微服务划分。但该类方法都需要人工操作，无法完全实现自动划分。综上，如何找到一种无需历史信息，且适用不同开发语言与功能状态的自动化微服务划分方法就显得十分重要。

数据流图(DFD)作为面向结构软件开发方法中一种功能建模工具，它通过图形的方式描绘数据在系统中流动和处理的过程，反映系统的逻辑功能，可以作为微服务划分的合理研究方向之一。因此，本文将研究一种基于数据流图的微服务划分方法，解决传统方法中，无法同时实现通用性、自动化且无需历史信息的问题。

2. 容器化微服务框架及其要求

2.1. 云计算中的容器化微服务架构

容器技术本质上是一种把操作系统资源划分成小型、独立资源空间的虚拟化技术。这种技术为在不同容器中运行的进程提供了有效的资源隔离，有利于解决资源冲突，提高资源利用效率。容器本质上是操作系统提供隔离的虚拟空间，它允许进程在主操作系统上隔离运行。微服务架构作为一种架构风格，其目的是构建一套细粒度的自包含、松耦合且业务功能自治的信息技术系统。利用容器技术实现的微服务架构，我们称为容器化微服务架构，如下图 1 所示，系统在裸机或虚拟机上运行不同微服务的容器实例，在用户请求队列中选出优先级高的业务请求进行处理，每个业务请求调用所需微服务的不同容器实例。容器化微服务框架形成了业务请求与微服务实例之间、微服务实例与裸机或虚拟机之间的两层映射关系。相较于传统分层架构，容器化微服务架构的边界垂直，微服务和业务功能间是对应一致的，因此在应用微服务架构时微服务的业务独立性至关重要，这需要对微服务的划分有所要求。

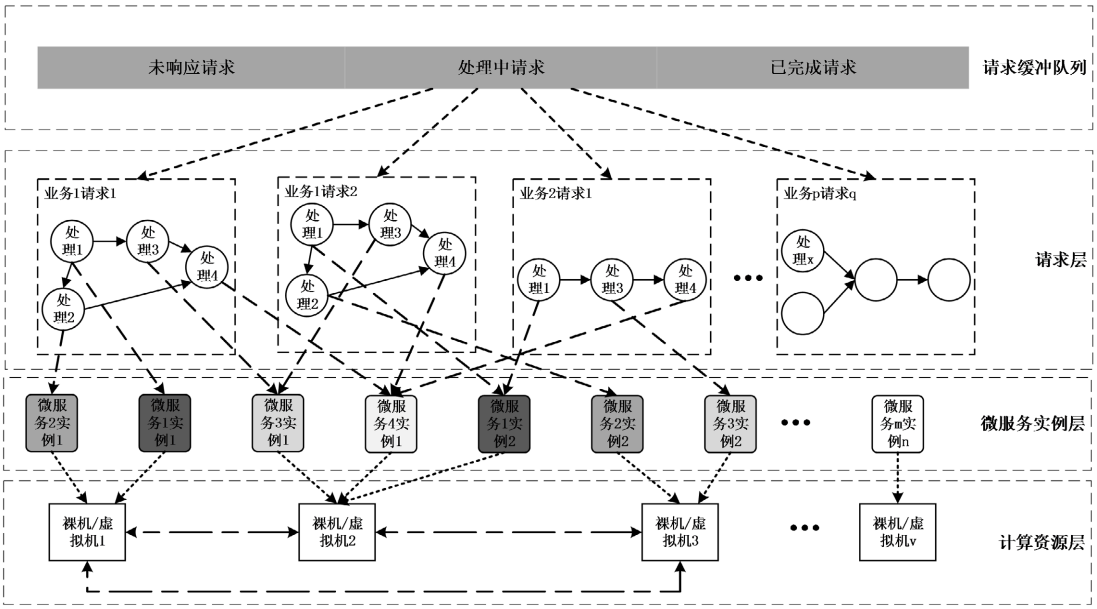

Figure 1. Containerized microservices architecture
图 1. 容器化微服务架构

2.2. 容器化微服务框架优势

如上图 1 所示，容器化微服务框架的主要特点是容器化封装、分布式部署与单业务调用多微服务(微服务间相互调用)，由此引申出单机多实例、微服务异构、轻量级通信、去中心化 4 个特点，这些特点各自支持着容器化微服务框架的一些优势，具体如下所述。1) 容器化封装：容器的隔离性便于系统实现敏捷开发和高效的弹性伸缩、增强了系统的容错性，容器便于管理的特性使得系统的部署人员工作量大大减少。容器化封装进一步给容器化微服务框架带来了微服务可异构的特点，使得微服务开发团队能够自主且松散耦合，高效的更新开发。2) 分布式部署：分布式框架下单机运行多实例的特点，支持系统对机器数量和实例数量的二级伸缩，在机器间隔离了故障。而分布式框架下结合轻量级的通信方式，使得微服务相互调用时可以不考虑技术栈的区别。3) 微服务间相互调用：微服务间相互调用的特点使得每个微服务自行其是，自我管理，系统实现去中心化，便于单个微服务的更新迭代，同时结合容器化封装的特点使得系统可以实现微服务的选择性伸缩。

2.3. 微服务划分的必要性和要求

因为 2.2 中所述容器化微服务框架的特点并不一定能转换为其对应的优势，所以必须要有一个适合的划分来实现系统到微服务的映射，才能使得系统获得微服务架构优势，划分的要求可被归类为：① 细粒度微服务快速启动② 微服务业务独立、隔离、③ 微服务间低通信开销④ 微服务内稳定性、可靠性接近。具体来看每个特点所需的要求有：容器化封装需要要求①；单机多实例的特点则需要①③；单业务调用多微服务的特点需要②③；单微服务多实例的特点需要②③；支持弹性伸缩和选择性伸缩、便于迭代开发、便于拆分数据库、稳定性和可靠性高的有点则分别对应要求：①②、①②④、②、④。

3. 软件需求模型、数据流图和微服务之间的关系

3.1. 软件需求建模与数据流图

需求分析是软件系统设计与实现的基础，需求建模质量会影响最终实现系统性能。需求建模中常使用结构化分析中的功能建模来描述软件需求，而数据流图是功能建模中常用的一种方法。数据流图包含外部实体 v_o，数据处理单元 v_p，数据存储 v_s，数据流 a 共 4 种基本元素[15]，其本质是一组有向图，重点描述系统的功能逻辑。构建数据流图时是从顶层数据流图开始，概括性的描述系统，而后逐步细化，将上层数据流图的一个功能节点细化为下层的一张局部数据流图，直至功能节点被划分的极细，此时即为底层数据流图。因为底层数据流图是通过顶层数据流图细化得来的，所以底层数据流图中各个局部的数据流图，也可以组合起来形成一张完整的底层数据流图(即一张功能模块极细化的有向连通图)，例如图 2 所示即为某 ERP 系统仓库与生产部分的局部底层数据流图合并后的状态。因为微服务划分时，要求关注业务逻辑的分解、微服务的内聚性和耦合性，使得微服务自成体系，所以如图 1 所示，一个业务请求对应多个微服务的组合调用，从而形成一张有向图，每个微服务作为有向图的一个节点涉及一部分业务逻辑，因此微服务的划分同样以逻辑功能为主要因素，故数据流图十分适合作为微服务划分建模的基础。

3.2. 分层数据流图与微服务粒度的关系

在数据流图中，通常外部实体在后续的设计与开发过程中，无法被更改或调整。数据存储节点是数据库拆分和微服务部署时需要考虑的问题，因此数据流图的划分主要关注数据处理单元。因为数据流图是逐层细化的，若以顶层数据流图的处理单元作为一个微服务，则微服务架构退化为单体架构。若以底层每个处理单元作为一个微服务，则微服务的粒度过小、调用链过长且关系复杂，系统通信开销大且响

应迟缓。若以中间层每个处理单元为一个微服务，由于需求分析建模时并未考虑微服务的特性，会导致映射出的微服务粒度有些过大，有些则过小，所以通过聚合粒度极细的处理单元，方便形成粒度不同且适合各自业务的微服务。故本文研究中，均以所有底层局部数据流图组合后形成的完整底层数据流图G_x(以下简称数据流图)作为研究对象，目标是研究一种划分方法，对G_x中的处理单元集合V_p进行划分，得到若干个粒度适中的微服务集合$M=\{m_1,m_2,m_3,\cdots\}$，且每个微服务包含多个处理单元，即$m_i=\{v_a,v_b,v_c,\cdots\}$。

3.3. 数据流图的刻画

3.3.1. 节点分类

正如3.2中所述，形成一个微服务主要是对数据流图中处理单元(节点)的进行划分或组合。为了能更精确的进行微服务的划分，需要在数据流图中包含更多处理单元的相关信息，以便分类分情况处理，为此我们定义了3种处理单元类型：

1) 分支节点V_{cho}：对于节点v，若满足判别式(1)为真则为选择节点。即节点v存在一个或以上的输入数据流，存在两个或以上指向其他处理节点的输出数据流，且存在节点v的同一输入流，对应两个及以上指向其他处理节点的可能输出流。分支节点实质是数据流的分支，正是通常发生业务变更的边界，故其在确定微服务的边界上具有重要作用，如图2中的灰色处理节点即为分支节点。

$$\begin{aligned}&\left(\exists a_1\left(v_1,v\right)\wedge\exists a_2\left(v,v_2\right)\wedge\exists a_3\left(v,v_3\right)\wedge\cdots\wedge\exists a_m\left(v,v_m\right)\right)\wedge\left(\exists a_1\to a_2\wedge\exists a_1\to a_3\wedge\cdots\wedge\exists a_1\to a_m\right)\\&\wedge\left(m\geq 2\right)\wedge\left(v\in V_p\wedge v_2\in V_p\wedge v_3\in V_p\wedge\cdots\wedge v_m\in V_p\right)\end{aligned}\tag{1}$$

其中a_1、a_2、a_3、a_m表示数据流，V_s表示数据存储节点集合，V_p表示数据处理节点集合，V为所有节点集合。

2) 存储交互节点V_{si}：对于节点v,若其满足判别式(2)为真，则节点v是一个存储交互节点。这类节点业务只是负责对系统中某类数据的维护，仅输出至存储节点。微服务架构需要拆分数据库，降低数据库的并发压力，除了部分通用数据，微服务系统更倾向于增强微服务间的数据隔离。此外，数据库也是微服务系统中故障高发的部分，故存储交互节点在微服务划分研究中具有重要影响，如图2中的蓝色处理节点即为存储交互节点。

$$\nexists a_i\left(v,v'\right)\wedge v'\notin V_s\tag{2}$$

3) 一般处理节点V_{cmp}：对于节点v，若其满足判别式(3)为真，则节点v是一个存储交互节点。由于除上述节点以外的节点性质类似，在微服务划分时影响不大，故统一考虑，如图2中的白色处理节点即为普通计算节点。

$$v\in V_p\wedge v\notin V_{cho}\ a\wedge v\notin V_{si}\tag{3}$$

3.3.2. 数据流的刻画

由于在微服务的协作过程中，微服务与微服务之间、微服务与外部实体之间都可能产生大量的数据交互，对网络信道造成极大的压力，然而划分在同一微服务中的功能模块间的数据交互只需要通过进程间数据交互即可完成，两者通信开销的差异对微服务系统的性能有重要影响。因此数据流的单位时间内发生次数，及单次使用的数据量对微服务划分的决策有着重要影响。由于准确的数据流发生频率和数据流长度在微服务系统设计阶段难以获得，但在需求分析阶段开发人员需要从用户处获取各功能的大致请求量，同时明确数据流的意义及它大致的字节量级，故本文假设数据流频率及数据流长度的等级在需求分析阶段已获得。

1) 数据流频率级 *Fre*：对于给定的数据流 *a*,其数据流频率为 $Fre=\left\lfloor \log_2 \frac{ct}{t} \right\rfloor$，其中 *t* 为计数的累积时间单位为秒，*ct* 为时间 *t* 内数据流 *a* 所发生的次数。

2) 数据流长度级 *A*：对于给定的数据流 *a*，若其数据流的长度为 *N* 个 Byte，则 $A=\left\lfloor \frac{\log_2 N}{10} \right\rfloor$

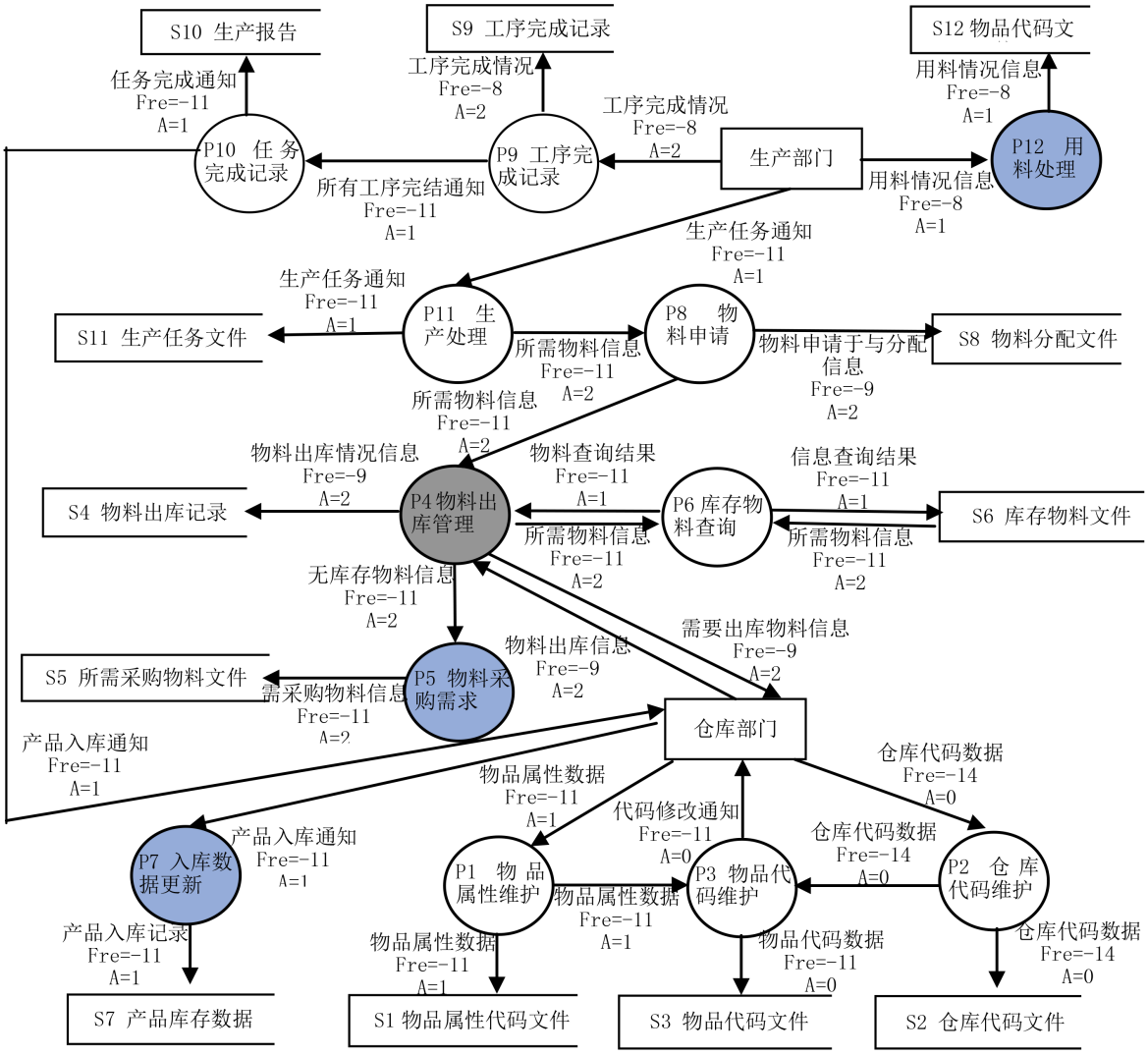

Figure 2. Data flow diagram of warehouse management and production management of an ERP system
图 2. 某 ERP 系统仓库管理与生产管理部分底层数据流图

3.4. 两个外部实体间的正常处理流程

如 2.1 中所述，微服务系统划分时需要重点考虑其业务隔离性。在数据流图中主要以文字描述和图结构两方面来体现系统的业务，文字描述难以结构化分析，故本文以图结构中包含的信息为主，重点关注外部实体间的处理流程(以下简称处理流)，其定义如下：对于数据流图 *G*，若其中有若干数据流的集合 A_p 满足公式(4)为真(其中 V_o 表示外部实体集合)，且 A_p 中的数据流图经过多个处理单元节点，这些节点经过的顺序为 $w=[v_1,v_2,v_3,\cdots,v_{k-1}]$，*w* 即为一条处理流。如图 2 中[P1, P3]即为一条处理流，以仓库部门

为起点和终点，[P7, P8, P4]也是一条处理流，以生产部门为起点，仓库部门为终点。

$$\exists\left(a_1\left(v_x,v_1\right)\wedge a_2\left(v_1,v_2\right)\wedge a_3\left(v_2,v_3\right)\wedge\cdots\wedge a_k\left(v_{k-1},v_y\right)\right),$$
$$v_x,v_y\in V_o\wedge\left(x=y\ \text{or}\ x\neq y\right)\wedge A_p=\left\{a_1,a_2,\cdots,a_{k-1}\right\} \tag{4}$$

4. 面向正常处理流程的微服务划分方法

4.1. 划分方法的基本思想

本文划分以数据流图为基础，通过将数据流图中的处理节点划分至几个不同的集合，每个集合中的节点元素即为一个微服务应包含的处理模块，所有的集合共同组成一个大集合即为系统整体的微服务划分结果。由于在实际业务中，一般一个业务涉及信息从源头经过处理再输出的过程，在数据流图中系统的输入输出均以外部实体的形式体现。又因为一个业务一般只涉及两个外部实体，若涉及多个外部实体，则可以通过增加适当的存储节点将该业务划分为多个业务，因此数据流图上两两外部实体间的处理流和数据流图描述的业务具有高度的相关性，是微服务划分的重要参照。故本文提出一种面向正常处理流程的微服务划分方法(Normal Flow Microservice Partition, NFMP)，通过获取数据流图上两两外部实体间的正常处理流来指导微服务的划分。

4.2. 划分的关键处理步骤

1) 数据流图的预划分

如 3.1 中所述，本文研究的数据流图是一张功能模块极细化的有向连通图，若直接获取外部实体间处理流，得到的处理流可能存在许多冗余，因此本文在获取处理流之前对数据流图进行预划分。因为两个外部实体间的经过存储节点的处理流可被视作两条独立的处理流，即对于每条处理流不会包含存储节点。同时，虽然数据流图中的数据流具有方向性，但是微服务划分更多的是关注功能节点间的联系是否紧密而非节点间联系的方向。因此本文暂时消除数据流图 G 中的数据流方向和存储节点，再利用广度优先遍历的方法搜索 G，得到若干个连通子图，最后将暂时除去的存储节点和数据流方向，添加回每个连通子图中，得到切分后的连通子图集合 G'，在算法 1 中记为函数 splitGraph()。

2) 获取外部实体间处理流

因为 V_{si} 节点只输出至存储节点，故其必定不在处理流中，而在图结构中搜索两个节点间的可达路径，一般可采用深度优先遍历算法，但又可能出现有向边成环的问题。故本文设计对于每一张连通子图 G_k，遍历其中的所有源外部实体节点 V_{osk} -目标外部实体节点 V_{otk} 组合，通过深度优先遍历该子图搜索源-目标节点之间的路径，得到处理流集合。而后把该子图上所有源-目标节点组合得到的处理流集合合并，得到子图 G_k 的处理流集合，在算法 1 中记为函数 searchProcessFlow()。针对有向边成环的问题，由于微服务的划分需要遵循业务内聚的原则，而成环的节点处于同一业务的概率较大，因此两节点间成环的直接加入处理流集合，去除一条成环边后继续深度搜索；对三个及以上节点成环的，则将成环部分作为一条处理流加入处理流集合以便在后续的处理流融合与分割时加以处理，并停止深度搜索。针对连通子图中的 V_{si} 节点，若只存在一个处理节点向其输入数据流，则直接将该 V_{si} 节点加入向其输入的处理节点所在的处理流中，若存在多个处理节点向 V_{si} 节点输入数据，则将该 V_{si} 节点作为一个独立的处理流加入子图的处理流集合。

3) 超集处理流分割与相似处理流合并

得到连通子图的处理流集合后，由于处理流集合中仍包含大量的类似的处理流，若将所有的处理流全部作为微服务，则代码复用率太低，不利于高效开发；同时其中的部分处理流过长，若直接作为一个

微服务，则这个微服务的粒度过大。故本文设计了超集处理流分割与相似处理流合并两个处理步骤，前者通过搜索处理流集合中每个元素在处理流集合中的超集(考虑节点的出现顺序)，若只有一个超集则将该元素合并至其超集中。若有多个超集，则尝试在每个超集中找到一个能包含这个元素的最小子集，且该子集的首尾都是V_{cho}节点或V_o节点，比较每个超集中找到最小子集，子集中所含元素数量最少的为拆分子集，拆分子集所在的超集为拆分超集。而后将拆分子集与拆分超集交接处数据流的 *Fre* 和 *A*，与拆分超集中数据流的平均 *Fre* 和 *A* 进行比较，若交接处数据流的 *Fre* 和 *A* 均较小，则执行分割。分割的具体形式为：利用拆分子集代替处理流集合中的这个元素，并把拆分超集中切出拆分子集，剩余拆分子集之前和之后的元素各自组成一个处理流程加入处理流集合，在算法 1 中记为函数 splitSupersetProcessFlow()。相似处理流合并步骤，比较处理流集合中长度相同的处理流之间的相同元素占比(不考虑节点的出现顺序)，合并占比超过阈值的处理流，在算法 1 中记为函数 mergeProcessFlow()。

4.3. 算法描述

如 4.1 中所述，面向正常处理流程的微服务划分方法大致包含数据流图划分，两两外部实体间处理流获取，处理流分割与合并三个部分，算法具体描述如下伪代码所示：

算法 1：面向正常处理流程的微服务划分方法（NFMP）

输入：数据流图 G，合并两个业务路径的相似度阈值 *sim*.

输出：得到的微服务集合 W.

```
NFMP()
{   V_cho ← getBranchNodes (G);                          //获取分支节点集合 V_cho
    V_si ← getStorageInteractiveNodes(G);                //获取存储交互节点集合 V_si
    V_cmp ← getCommonProcessingNodes (G);                //获取普通计算节点集合 V_cmp
    W ← ∅;
    G' ← splitGraph(G);                                  //按照 4.1 中所述预划分 G，得到连通子图集合 G'
    for  G_k ∈ G'
    {    W_k ← ∅;
         V_osk ← getExternalEntitySourceNodes ( G_k );   //获取子图 G_k 中的源外部实体节点集合 V_osk
         V_otk ← getExternalEntityTargetNodes ( G_k );   //获取子图 G_k 中的目标外部实体节点集合 V_otk
         for  v_i ∈ V_osk
         {    for  v_j ∈ V_odk
              {    W_ij ← searchProcessFlow ( G_k , v_i , v_j );        //获取外部实体 v_i , v_j 间处理流
                                  W_ij ← insertStorageInteractiveNodes( W_ij , V_si );
                   W'_ij ← ∅;
                   while  W'_ij ≠ W_ij
                   {    W'_ij ← W_ij ;
                   for  w_ijm ∈ W_ij
                        {    flagRemove ← False;
                        PW_ijm ← getSuperSet( W_ij , w_ijm );          //找出 W_ij 中 w_ijm 所有的超集
                        W_ij ← splitSupersetProcessFlow( W_ij , PW_ijm , w_ijm , V_cho );//超集处理流分割
```

```
                    }
                }
            }
        }
        W_ij ← mergeProcessFlow(W_ij, sim);                    // 相似处理流合并
        W_k ← W_k + W_ij;
    }
    W ← W + W_k;
    return W;
}
```

5. 微服务划分算法应用实例及效果分析

5.1. 划分实例应用背景与划分结果分析

为了检验本文方法的正确性与划分微服务的质量，我们选用通用基础云 ERP 系统的微服务划分作为验证对象，由于 ERP 系统内逻辑复杂，业务交集较多，可以有效代表同类型的众多信息管理系统，便于验证划分方法的普适性。如图 2 所示。图中处理流程对应的两个外部实体可以是生产－仓库、生产－生产、仓库－仓库、仓库－生产，设置合并相似度阈值为 0.5，得到微服务划分结果为：[[P1, P2, P3], [P4, P5, P6, P8, P11], [P9, P10], [P7], [P12]]，即节点 P1，P2，P3 和 P4，P5，P6，P8，P11 和 P9，P10 分别组合为一个微服务，节点 P7 和 P12 各自独立为一个微服务。

根据划分结果，可知图 2 中所有的数据处理节点均已分为至不同的微服务中，证明了该划分结果的正确性。其中的存储交互节点 P7，P12 被划分为独立微服务，如同应用实例 1，这样的划分降低了对单个机器的数据库吞吐压力，也提高了微服务的业务内聚性；微服务[P4, P5, P6, P8, P11]恰好代表了生产部门在生产中向仓库部门申请物料的完整业务过程，且这几个节点没有其他相邻的处理节点，说明该微服务的划分充分满足了业务内聚性的要求；微服务[P1, P2, P3]恰好对应“更新仓库代码”和“物品属性维护”业务，这样划分虽然内聚性稍差，但是减少了 P2 节点的重复实现，且仅包含 3 个处理节点，保证了微服务细粒度的要求；最后微服务[P9, P10]则对应了“任务完成处理”业务，划分结果完全内聚。综上所述，故本文算法划分的微服务结果质量较高。

5.2. 微服务划分结果容器化运行性能分析

微服务实现方面，根据 5.1 中所述的划分结果，本研究利用 SpringCloud 架构实现了 5 个微服务，将其分别打包成 jar 包的形式。此外本研究实现一个额外工程，加入所有的功能模块以模拟单体应用架构实现。运行环境方面，本研究使用 3 个同构的计算节点，配置均为单核 CPU，2 G 内存，50 G 外存，安装 CentOS-7 操作系统，JDK 版本为 1.8，docker 版本为 1.12.6，节点间统一配置 Kubernetes 版本 1.12.1。在 CPU 负载率接近的情况下测试结果表明，在各微服务均不发生故障的情况下，单体架构实现的平均服务响应时间较微服务架构少 10%，在各功能节点各有 0.01%的故障率下两种架构平均服务响应时间接近，在故障率为 0.03%和 0.05%的情况下，微服务架构实现的平均响应时间分别少 16%和 34%。测试结果证明本研究提出的面向正常处理流程的微服务划分方法划分出微服务质量较高。

6. 结束语

与单体架构相比，容器化微服务架构具有易于开发和维护、单个微服务启动较快、部署灵活、测试

方便、技术栈不受限、便于按需伸缩，系统容错率高等优点。但在应用微服务架构时，必须将一个系统的功能合理划分至若干个微服务，现有方法大多需要历史运行数据或实现代码，或者难以形式化表述。因此本文提出了一种面向正常处理流程的微服务划分方法，以数据流图作为划分输入，关注图中两两外部实体间的数据流经过路径。首先对数据流图进行分割，而后搜索其中两两外部实体间的处理流程，并通过对关键节点的分析，分割过长的处理流程，合并被包含的处理流程，得到候选处理流，最后考虑系统通信开销的同时，合并相似的候选处理流，得到最终的微服务划分。将本文划分方法在 ERP 系统上进行应用分析，结果表明本文的微服务划分算法，可以自动化、正确地划分一个系统的功能，且微服务划分的结果可以较大程度地保证微服务的业务内聚性，同时不会出现粒度过大的微服务，且能尽量减少功能在微服务中的重复实现，划分结果实现的系统性能较强。

基金项目

国家重点研发计划项目(2019YFB1704100)，国家自然基金项目(62072337)。

参考文献

[1] Balalaie, A., Heydarnoori, A. and Jamshidi, P. (2016) Microservices Architecture Enables Devops: Migration to a Cloud-Native Architecture. *IEEE Software*, **33**, 42-52. https://doi.org/10.1109/MS.2016.64

[2] 陈韶健. Spring Cloud 微服务架构实战[M]. 北京：电子工业出版社, 2020: 7-9.

[3] 尹为强. 微服务容器化开发实战[M]. 北京：电子工业出版社, 2020: 4-8.

[4] Thönes, J. (2015) Microservices. *IEEE Software*, **32**, 116. https://doi.org/10.1109/MS.2015.11

[5] 欧阳荣彬, 王倩宜, 龙新征. 基于微服务的数据服务框架设计[J]. 华中科技大学学报(自然科学版), 2016, 44(z1): 126-130.

[6] Mohsen, A. and Amjad, I. (2016) Requirements Reconciliation for Scalable and Secure Microservice (De)composition. 2016 *IEEE* 24*th International Requirements Engineering Conference Workshops*, Beijing, 12-16 September 2016, 68-73. https://doi.org/10.1109/REW.2016.026

[7] 冯志勇, 徐砚伟, 薛霄, 陈世展. 微服务技术发展的现状与展望[J]. 计算机研究与发展, 2020, 57(5): 1103-1122.

[8] Escobar, D., Cardenas, D., Amarillo, R., *et al*. (2016) Towards the Understanding and Evolution of Monolithic Applications as Microservices. 2016 *XLII Latin American Computing Conference* (*CLEI*), Valparaiso, 10-14 October 2016, 1-11. https://doi.org/10.1109/CLEI.2016.7833410

[9] Schermann, G., Cito, J. and Leitner, P. (2015) All the Services Large and Micro: Revisiting Industrial Practice in Services Computing. *International Conference on Service-Oriented Computing*, Goa, 16-19 November 2015, 36-47. https://doi.org/10.1007/978-3-662-50539-7_4

[10] Baresi, L., Garriga, M. and Renzis, A.D. (2017) Microservices Identification through Interface Analysis. 6*th European Conference on Service-Oriented and Cloud Computing*, Oslo, 27-29 September 2017, 19-33. https://doi.org/10.1007/978-3-319-67262-5_2

[11] Tugrul, A. and Yunus, E.S. (2017) Policy Enforcement upon Software Based on Microservice Architecture. 2017 *IEEE* 15*th International Conference on Software Engineering Research, Management and Applications* (*SERA*), London, 7-9 June 2017, 283-287. https://doi.org/10.1109/SERA.2017.7965739

[12] Mazlami, G., Cito, J. and Leitner, P. (2017) Extraction of Microservices from Monolithic Software. 2017 *IEEE* 24*th International Conference on Web Service*, Honolulu, 25-30 June 2017, 524-531. https://doi.org/10.1109/ICWS.2017.61

[13] Merson, P. and Yoder, J. (2020) Modeling Microservices with DDD. 2020 *IEEE International Conference on Software Architecture Companion*, Salvador, 16-20 March 2020, 7-8. https://doi.org/10.1109/ICSA-C50368.2020.00010

[14] 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[J]. 软件学报, 2019, 30(10): 3227-3241.

[15] 张海藩. 软件工程导论[M]. 北京：清华大学出版社, 1998: 40-47.