毕业论文（设计）正文

题目：微服务划分工具的设计与实现

微服务划分工具的设计与实现

摘　要：微服务架构的兴起是云计算时代的一项重要事件。在将传统架构下的代码迁移至微服务架构下时，不可避免的一个大问题就是微服务划分。当下企业主要使用的划分方案为人工划分，有较强的个体性，难以将经验直接复制使用。因此对自动化的普适划分方案有着强大的需求。现有的研究大都将该问题视作精细化的传统模块划分，而没有考虑到微服务架构带来的物理优势。同时现有方案的实验步骤也都过于理论化，没有进行实际的部署与测试。本文利用代码实际运行情况记录，自动分析系统主要特征。再抽象为图结构，设计独特的算法以获得优质的微服务划分方案。划分算法将考虑到云环境下微服务架构的逻辑层面特征和物理层面特征，尽可能使划分后系统具有高内聚低外耦合的特点，同时挖掘对高性能、高可靠有需求的模块，便于实际运行过程中的独立弹性伸缩与安全可靠易维护。最后使用将结果作为微服务部署，用实际情况来验证了优秀的效果。可以说是做到了领域内的全新突破。

关键词： 微服务；　运行路径；　基于图的划分；　物理特征识别

Design and Implementation of Microservices Extraction Tool

Abstract: The rise of microservice architecture is an important event in the era of cloud computing. When migrating the code under the traditional architecture to the microservice architecture, one of the inevitable big problems is the division of microservices. At present, the division scheme mainly used by enterprises is manual division, which has strong individuality and is difficult to directly copy and use. Therefore, there is a strong demand for automated universal division schemes. Existing studies mostly regard this problem as a refined traditional module division, without considering the physical advantages brought by the microservice architecture. At the same time, the experimental procedures of the existing schemes are too theoretical, without actual deployment and testing. This article uses the actual operation records to automatically analyze the main features of the system. It is then abstracted into a graph structure, and a unique algorithm is designed to obtain a high-quality microservice partitioning scheme. The partition algorithm will take into account the logical and physical characteristics of the microservice architecture in the cloud environment, and try to make the partitioned system have the characteristics of high cohesion and low external coupling. At the same time, it will mine the modules that require high performance and high reliability to facilitate Independent elastic scaling and safety, reliability and easy maintenance during actual operation. Finally, the result was deployed as a microservice, and the actual situation was used to verify the excellent effect. It can be said that a new breakthrough in the field has been achieved.

Keywords: Microservices; Execution Trace; Graph-based Extraction; Physical-feature Recognition

正文目录

图目录

表目录

伪代码目录

第1章 课题背景

1.1 微服务架构

科技在不断进步，5G 和新材料的发展让我们看到一幅构建在“云”上的未来图景——将占用空间的负责计算的“重”物理机器放在服务端，把只有操作功能的“轻”机器留给客户端。这样的云环境使得用户能够拥有更多的更灵活的选择，同时也能通过购买服务、资源而获得更强大的计算能力。随着“云上计算”概念的普及，如何在云上开发这个问题也走入了研究者们的视野[1]。微服务架构概念正是在解决这个问题的过程中变得非常有热度，因为它不再按传统将系统视为一体式的代码，而是视为一个个独立的微服务[2]。每个微服务都足够独立，能够部署在完全独立的机器上，使用不同的技术栈，无论是编程语言还是数据库，都能够根据每种服务的需求而独立更改。同时各个微服务间使用轻量级机制进行通信，例如使用 HTTP（超文本传输协议）资源 API（应用编程接口）。这使得各服务本身的独立调度、独立伸缩变得更加容易，各服务间的界限更为明确。其思路完全适应于完全分布式的环境，恰好能充分利用云环境的优势，因此被认为是一种最适应云生态环境的新架构。

1.2 微服务划分工具意义

微服务架构下代码结构特殊，研究人员认为从传统架构下的现有系统进行迁移会比直接构建微服务系统更加有效[3]。为充分发挥微服务架构的优势，迁移过程中就需要对代码进行合理划分。也正是因此，对一体式系统进行划分构成微服务已经被认为是一个重要需求[4]。在工业实践上，常常被选择的方案是人工划分，这对划分者的专业性要求极高，同时划分的结果、步骤也没有的普适性，据调查显示目前许多已划分的微服务都是功能小众的、适用范围狭窄的[5]，这导致了划分方案难以普及，划分过程也难以为他人所借鉴。显然，只有发现了普适化的划分方案，微服务架构才能在每个人手上发挥其作用，才能真正使云环境的优势得到展现。而普适化且自动化的工具更是直戳用户的需求，大大降低了使用微服务架构的门槛。因此，可以说普适化自动化划分工具是微服务架构推广的基石，其能够在整个微服务架构体系下发挥出重要的作用。

1.3 微服务划分问题

1.3.1 模块划分问题

在计算机历史上发挥重要作用的单体式架构仍然是一种非常有效的架构，但其一直以来都有一个巨大的问题：随着代码量的增加与功能的复杂化，单体式架构下的代码常常会变得非常臃肿。为了解决其结构混乱、复杂性高的问题，代码划分早就为人所讨论。所有划分的核心思想都是分而治之，即目的为对一个完整而庞大的系统进行拆分，使得拆分后的每个部分逻辑简单而有效，以此来实现逻辑层面的简化。

众多划分问题中，“模块划分”问题已经有长久的历史。模块划分常常出现在开发者口中，通常来说，模块划分最重要的意义在于方便开发者梳理逻辑、简化开发过程，从而对于开发者能够节约时间、提高效率，对于系统能够使其逻辑清晰，便于迭代升级或安全维护。

模块划分的主要切入点有两个：复用和解藕。研究者与开发者发现，系统的不同功能间常常会出现逻辑重复的情况，当大段代码重复出现时，系统整体自然会显得非常臃肿、混乱与复杂。这时候一个简单的处理方法就是按功能提取公共部分作为一个新模块，等价于将大段代码块更换为简短而明了的接口，同时在未来实现其他功能时，也可以通过对已有模块的组合来简单地实现，这也就是所谓的复用。而所谓的解藕则是将各功能的逻辑梳理分组，使得不同的组尽可能独立，尽可能没有逻辑上的功能依赖关系。这样就能分离各部分的责任，划清边界，便于组织管理以及维护和独立升级。

1.3.2 微服务划分与模块划分的异同

微服务划分也包含逻辑层面划分，也需要考虑如何划分能够便于使逻辑清晰、使管理方便，因此过往的划分思路对微服务划分也完全有借鉴意义。同时，微服务划分的主要思路是从业务层面出发，按业务的功能流程进行划分。另外，分布式环境带来的全新模式也给微服务划分带来了新的思路。微服务架构下服务的独立部署导致各服务间的通信代价增大，因此解藕对于微服务而言就更加有意义。同时微服务架构对提升了各服务部署、调度、运维的独立性，微服务划分时也完全可以充分利用这些特点，扬长避短。

由于分布式基础设施的特点，微服务划分也比传统划分多了一个问题：粒度，也就是在微服务大小和数量上的权衡。在传统架构下，划分的粒度对于系统的运行并无影响，因此粒度问题不需要被考虑，即便在系统中粒度有大有小也毫无问题。但对于微服务架构而言，微服务的架构对于系统的表现有着重要的影响。若粒度太大，则会使系统与传统架构下的一体式代码相似，无法发挥

出分布式基础设施的优势，也就无从说起微服务架构比传统一体式架构有什么优点。若粒度太小，则会由于微服务间的壁垒而增加数据传输代价，同时在微服务大量重用的情况下会使传输体系对系统的影响大大增加。但在一些研究者看来，微服务架构在实践中的重用远远小于预期，因此微服务的粒度应当足够小，从而能够在修改时足够迅速、对其他部分影响足够小，发挥其在快速开发，快速迭代和快速运维上的优势[6]。

微服务划分的思路还可以考虑到未来云环境的发展趋势。从 IaaS（基础架构即服务）发展到 PaaS（平台即服务）再发展到 SaaS（软件及服务），云环境向用户所提供的服务越来越高层。可以设想，在未来假若能够直接向用户提供带有接口的微服务，交由用户进行微服务的组合，将使编程成为更亲民的事情。用户不需要了解每个服务中的功能是如何实现的，只需要选择自己所需的功能，利用各个微服务向外暴露的接口进行数据传递，即可完成程序的实现。在这种情况下，用户即便不懂得编程知识，也能够通过组合微服务，按照拼图的思路将一个个简单模块构建成理想的系统。在未来，假若能够有这样的技术支持，微服务的划分则需要考虑到每个微服务的功能完整性和足够细的粒度，以此降低用户在组合时能够做出选择的时间成本。

总结而言，微服务划分与模块划分的区别有二：从逻辑层面来说，模块划分是从业务下具体功能的实现角度出发，目的是便于代码功能的实现，相较而言视角更加底层；而微服务划分是从业务的设计角度出发，目的是便于业务逻辑的切割，相较而言视角更加高层。从物理层面而言，微服务划分需要适应并利用分布式环境，而这对于模块划分而言并不需要考虑。总的来说，二者不是对立的，而是各司其职、相辅相成的。

因此，设计微服务划分工具时不能简单地照搬传统模块划分策略，而应当在其基础上有选择地吸收并发展。

第2章 算法设计和实现

2.1 概述

微服务划分问题还是一个很年轻的领域，对其的研究非常少且不完善。而目前的研究过于纸上谈兵，所提出的方案与传统的模块划分领域方案几乎没有区别。同时国内外论文的实验验证部分也都没有落实到实际运行上，只从理论上的指标进行了对算法效果的评价。

本文所提出的算法考虑了微服务架构的物理特性，充分识别系统中的高性能模块，能够给出真正适合实际使用的方案。同时，在实验部分，真正做到了将划分结果方案作为微服务部署，用实际情况来验证了效果。可以说是做到了领域内的全新突破。

最终实验结论证明，本文算法具有有效性，所得到的划分方案能够对云资源进行充分利用，实际部署的微服务系统在资源总量相同的情况下能够拥有更短的响应时间。

2.2 算法背景

分布式环境在近几年才在大众视野中崭露头角并受到各个企业的广泛应用，而面向这种环境的微服务架构更是一个新兴的冉冉升起的架构。正是因为其过于年轻，当前面向微服务架构的研究相对而言还非常少，整个领域处于一个年轻且蓬勃发展的状态。

在面向微服务的研究中，对微服务划分问题的研究更是少之又少。一方面而言，微服务划分问题较为复杂，且每个划分方案需要针对系统的特殊性来设计，方案大都不具有普适性，因此当前业界的解决方案大都是“请专家进行人工分析并划分”。另一方面而言，微服务架构解决的是超大流量情境下的问题，因此通常只有拥有大众级服务的超大公司才对其有硬性需求。而这样的大公司的服务要求方案拥有绝对的可靠性，公司又有足够的资本请专家进行设计，因此对自动化划分的需求还不急迫。

但随着 5G 技术的发展与云环境基础设施的完善，人们越来越倾向于使用“云上系统”，即将计算能力、服务能力等部署到云上设备而非部署到终端设备，最典型的例子正如微信的小程序系统。可以预见，在不远的未来就会实现“云环境”的通用化、平民化。在这样的情境下，一套自动化的微服务划分方案将会满足人们最迫切的需求。

2.3 目前研究情况

2.3.1 目前微服务划分算法研究的动态

目前，对本课题的研究中，按划分的方法的时间顺序可以分为三个步骤：确定最小划分单位；将文件结构抽象并构建成图结构；对图结构进行划分或聚类。本文将按这三个步骤，将各个算法切割开来，具体比对。

划分的最小单位有两个切入点：源代码信息，API 信息。源代码信息即传统架构下的代码最小单位，例如以文件、类、包、功能块等作为最小单位。在此基础上有的研究者还仿照业界划分模式，增加适当的人工介入，提前进行细粒度的组合，也就是对源代码进行域分析（Domain Analysis）[7]。通过这样的手段，可以只将计算各部分间关联度的功能交由自动化工具处理，能够克服自动分析不够精确的问题，不失为一种折中的第三选择。而 API 信息作为最小单位则是出于对“迪米特原则”的考虑，不同服务之间应当只通过接口进行交流，内部的具体信息不应影响结构。API 作为应用程序之间通信和交互的信息特征[8]，能够很好地反映与具体实现无关的特征。从前文“微服务划分的角度”来说，由于有选择性的忽略了具体实现相关的特征，选择 API 作为最小单位恰好符合“关注业务的设计”角度。但这种方案要求系统在设计过程中足够符合规范，或需要人工介入对不规范的信息进行修正，以保证接口足够明确并且接口名称足以显示功能，方便后续划分时有充分且正确的信息。

在构建图结构步骤中，过往工作的主要关注点集中在两个方面：静态信息，动态信息。其中，静态信息为不需要实际运行就能获得的信息，例如代码内容、（在 git 等版本控制系统中）修改记录[9]、修改人员等等。借助静态信息时，研究者们常常将各模块关联度计算问题转化为语义分析问题，例如将函数或文件中去除停用词后的单词表作为多维特征向量[9]，或直接取 API 信息中的单词作为多维特征向量[8]，以及将单词表按相似度映射到公开词典并取映射后词表作为聚类依据[6]。除此之外，也有研究者根据单一责任原则，认为可以通过文件间修改关系以及各文件修改者之间的关系，认为需要一起修改的文件应当被放在同一部分 [9]。动态信息则是在系统的运行过程中所获得的实际关联信息，例如模块间的调用关系，用户运行路径等等。部分研究者认为，由于系统随着时间需求不断变化，后期系统的实际思路与最初设计时的思路已经有了偏差，因此使用设计方面的信息为依据是不足够可靠的[10]。因此他们提出，可以使用动态信息作为划分依据。当监控用户执行路径时，我们所能获得的原始信息是各个函数之间的调用情况，根据这些信息可以构建两种图：由原始调用信息所构建的调用图，这是一种有向有边权图，可以展现各个函数间的直接关联

[10]；将动态信息按执行者分组，在同一组出现的函数认为是有关联的，也就是以一个用户的一次完整执行路径为单位，将每次执行过程中涉及到的函数都认为是有关联的，这是一种无向图，可以展现各个函数之间的传递关联[11]。

图上划分/聚类步骤则比较偏向于算法设计问题中的图结构划分/聚类问题，可以充分借鉴算法领域过往的研究成果。以往成果大多再上一步骤中构建出无向图，此时图中的边即为各点间的关联程度，边权越大表示所连接的两点关联程度越高。之后再按以下两种主要思路进行：一是作为划分问题，先将整个系统试作整体，再从连接不紧密处入手，逐步划分。此时划分问题与传统图划分问题较为类似，可以选择由边权入手，逐步按边权由小到大将边删除。在这样的删除过程中，图会被逐渐划分成多个联通块，由于每次删除的都是边权最小的边，各个联通块间的紧密程度理论上会逐渐变大。二是作为聚类问题，先将系统按每个最小单位划分成非常细碎的部分，再从连接紧密处入手，逐步聚类。此时聚类问题更接近机器学习中的聚类问题，按照前期所获得的各个特征向量进行聚类。由于聚类时选择的一定是更加有价值的边，因此各个部分在聚类过程中理应是越发紧密的。另外有研究者以有向图为依照进行划分，这种情况下算法的设计会较为复杂且麻烦。例如按照自行设计的数条划分规则进行划分[10]，此时容易造成覆盖不全面或有特殊情况导致遗漏的问题。

2.3.2 目前微服务划分算法研究的不足

目前对于微服务划分算法的相关研究很少，也就意味着研究者们都处于探索阶段。整体来看，相关问题的论文大都不具备完整的逻辑链条。对于一些很关键且很典型的问题——例如如何评判划分结果的好坏？——研究者们并无统一的看法，同时目前的各种方案在实际使用效果上的效果还较差。

若仅考虑目前研究所提出的各种算法的理论可行性，会发现其大都只关注了微服务的逻辑层面特性，而忽略了微服务的物理层面特性。换言之，只考虑如何使划分后的微服务间在逻辑上保持尽量独立，却没有考虑去利用好“云基础设施”的优势。由此导致其产出的结果更像是传统架构下的更精细的模块划分，而非云服务时代新架构下的微服务划分。

2.4 问题定义

为更明确地进行微服务划分，将面向微服务的划分问题进行形式化，如表2.1 所示：

表 2.1 动态信息示例

Sys	需要被划分的目标系统
Services	划分方案集合
$Service_i$	*Services* 中第 *i* 个微服务所包含的文件集合
n_i	*Services* 中第 *i* 个微服务所包含的文件个数
Traces	动态信息集合
$Trace_i$	第 *i* 条动态信息
$TraceId_i$	动态信息 $Trace_i$ 所属路径编号
$CallerClass_i$	动态信息 $Trace_i$ 的调用者类名
$CalleeClass_i$	动态信息 $Trace_i$ 的被调用者类名
$CallTimes_i$	动态信息 $Trace_i$ 的调用次数
$Class_x$	系统中第 *x* 个类
$Class_{i,j}$	$Service_i$ 中第 *j* 个类
Log_x	第 *x* 次用户请求所涉及的系统调用集合
$RawRelEdgeVal_{x,y}$	关联图中点 *x* 和点 *y* 间边的原始权值
$RelEdgeVal_{x,y}$	关联图中点 *x* 和点 *y* 间边的归一化后权值
$CalEdgeVal_{x,y}$	关联图中点 *x* 和点 *y* 间边的权值
CohesionVal	划分方案的内聚度
CouplingVal	划分方案的外耦合度
Metrics	逻辑层面指标，即(*CohesionVal, CouplingVal*)
$isBetter(Metrics_{old},Metrics_{new})$	$Metrics_{new}$ 是否比 $Metrics_{old}$ 更优
$relationScore(Class_x,Class_y)$	$Class_x$ 和 $Class_y$ 在关联图上的关联度

对于给定系统 *Sys*，需要根据其相关信息将其划分为多个微服务 *Services*，要求每个微服务 $Service_i$ 能够独立部署并能发挥分布式架构的优势：在逻辑层面上具有较好的独立性，降低系统的复杂程度，也要避免分割后不同微服务间的高耦合带来很大的传输代价；在物理上需要能识别出特征明显的模块，将其分割以发挥微服务架构在应对其需要高性能或高可靠特征时的优点。

2.5 算法介绍

本文提出基于物理特征识别的微服务划分算法（Physical-freatures-Recognition-Based Microservice Extraction algorithm, PRBME）。该算法通过识别目标系统中的关键模块，给出目标系统的微服务划分方案，能够做到过程自动化、结果有效化，即划分过程几乎无需人工操作且最终给出的结果能够在实际使用中拥有非常不错的效果。

算法分为两个主要部分，第一部分为建图，第二部分为划分。我们可以假设用户已经自行使用软件对系统进行了监控，获得了系统的动态信息。

2.5.1 动态信息

动态信息为本算法利用到的最重要的信息，所谓动态信息即系统运行过程中所展现出的信息，在该算法中可以简化认为动态信息即系统运行过程中各个函数或类间的调用信息。

对于 Java 系统来说，这样的信息可以通过开源组件 Kiker 进行实际监控与获取。监控过程中，需要注意的问题是：动态信息应当从被监控系统的尽可能长的运行周期中所获得，以避免因为短期内的偶然性导致动态信息不具备反映系统实际运行关系的能力；另一种方案是直接对被监控系统对外开放的接口进行测试并进行监控，当保证每个接口都被测试过相同的次数时，可认为所得到的动态信息基本可反映系统的实际运行关系。

接下来将详细介绍本文算法所用到的动态信息文件的格式。我们将系统的所有调用信息视作一组调用路径的集合 *Traces*，其中每一次调用 $Trace_i$ 包括以下几部分内容：$TraceID_i$、$CallerClass_i$、$CalleeClass_i$、$CallTimes_i$。其中 $TraceId_i$ 标记了该路径 $Trace_i$ 属于哪次用户请求，即对于任意两个路径 $Trace_j$, $Trace_k$，若其 $TraceID_j$ 和 $TraceID_k$ 相同，则证明他们是同一次用户请求 Log_x 中产生的两个调用，如公式（1）所示。$CallerClass_i$，$CalleeClass_i$ 分别表示该路径中的调用类和被调用类。$CallTimes_i$ 表示在标记为 $TraceID_i$ 的用户请求过程中，这样的调用出现了多少次。

$$Log_x = \{Trace \mid \forall Trace_i, Trace_j \in Log_x, TraceID_i = TraceID_j\} \quad (1)$$

表 2.2 所示即一份动态信息具体内容示例。

表 2.2 动态信息示例

TraceID,CallerClass,CalleeClass,CallTimes
0,org.mybatis.jpetstore.A,org.mybatis.jpetstore.B,10
0,org.mybatis.jpetstore.B,org.mybatis.jpetstore.C,10
1,org.mybatis.jpetstore.D,org.mybatis.jpetstore.E,10000
2,org.mybatis.jpetstore.D,org.mybatis.jpetstore.E,11
2,org.mybatis.jpetstore.E,org.mybatis.jpetstore.C,11

2.5.2 建图部分

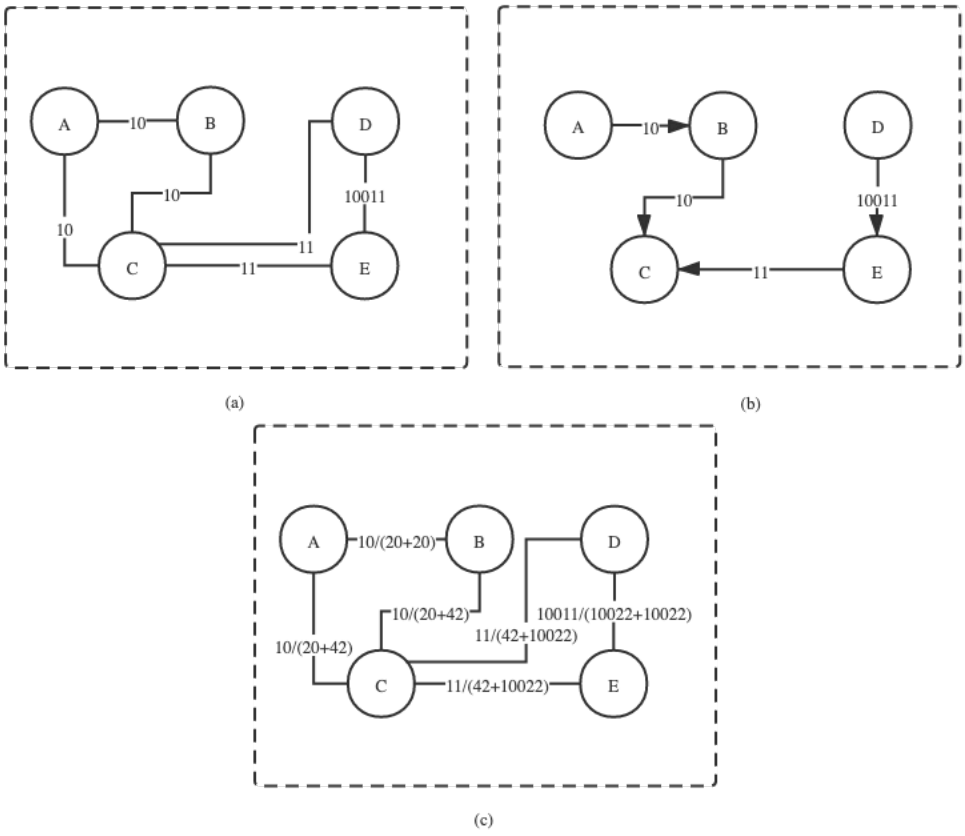

图 2.1 划分算法的流程图

建图部分目标为将复杂的系统信息抽象出关键特征，构成图结构。一方面能让杂乱的信息变为质量更高重要性更强且精简的特征，另一方面也能让复杂的经验性的微服务划分问题变为计算机领域内更为传统的图论划分问题。

根据后续算法的设计来抽象数据结构，我们所需要建图的结果有两个：关联图 *RelationGraph*，调用图 *CallingGraph*。二者所包含的信息分别对应着系统的逻辑层面特征信息和物理层面特征信息。以样例说明，当动态信息情况如表 2.2 所示时，关联图 *RelationGraph* 将如如图 2.1（c）所示，调用图 *CallingGraph* 如图 2.1（b）所示。

从逻辑层面来说，应当将用户的每个请求对应的一组类或函数视作一次服务，而每次服务可以被视为一个逻辑上紧密的整体。因此每次服务所涉及到的一组类间具有关联性，任意两个类出现在同一次服务中的次数越多，则其关联度越高。依此，我们就能从动态信息中抽取出关联图，其点为系统中的各个类，而其边为任意两个类出现在同一次服务中的个数，边权计算方法如公式（2）所示，所构建的关联图结构如图 2.1（a）所示。

$$RawRelEdgeVal_{x,y} = \left|\{ Log \mid \exists Trace_i \epsilon Log, (CallerClass_i, CalleeClass_i) = (Class_x, Class_y) \vee (CallerClass_i, CalleeClass_i) = (Class_y, Class_x) \}\right| \quad (2)$$

在此基础上，我们还会发现，当某条请求链路为关键链路时，容易对其他非关键链路的划分造成影响。例如在图 2.1（a）中，包含类 A、类 B 的链路 AB-都包含类 C，显然类 A、类 B 与类 C 的关系非常紧密；而包含类 D、类 E 的链路 DE-中只有很少的部分包含类 C，即类 D、类 E 与类 C 的关系并不紧密；但由于类 D、类 E 处在关键链路中，总调用次数较多，导致类 C 反而与类 D、类 E 的关联度更高。因此，我们还必须对逻辑层面的关联图 *RelationGraph* 进行归一化处理。归一化时，我们会用“当前边权占两端点所有边边权和比例”作为最终关联图的边权，具体归一化逻辑如公式（3）所示。最终所得到的关联图 *RelationGraph* 形式将如图 2.1（c）所示。

$$RelEdgeVal_{x,y} = \frac{RawRelEdgeVal_{x,y}}{\sum RawRelEdgeVal_x + \sum RawRelEdgeVal_y} \quad (3)$$

从物理层面来说，若一组类被其他很多类调用，同时调用量非常大，那我们就可以认为其有高性能及高可靠的需求。根据这样的逻辑，我们只需要按调用关系就能建成代表物理特征的调用图：调用图是一张有向带权图，其点为每

一个在系统中出现的类，其边表示调用次数，例如 $Class_x$ 指向 $Class_y$ 的边有边权 15，则代表 $Class_x$ 调用过 $Class_y$ 次，如图 2.1（b）所示，边权计算公式如公式（4）所示。

$$CalEdgeVal_{x,y} = \sum_{\{Trace_i | (CallerClass_i, CalleeClass_i) = (Class_x, Class_y)\}} CallTimes_i \quad (4)$$

2.5.3 划分部分

针对基于动态信息分析的微服务划分问题，我们提出带有物理特征识别的微服务划分算法，具体划分流程逻辑如图 2.2 所示：

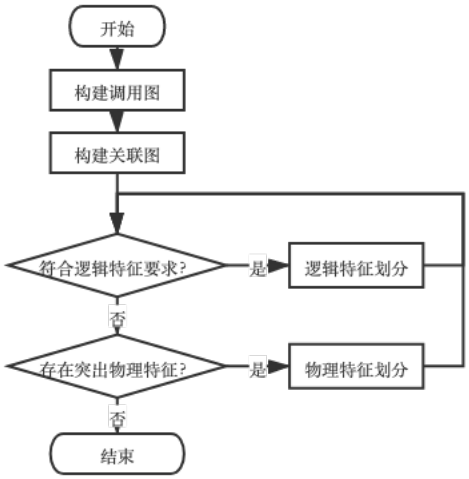

图 2.2 划分算法的流程图

划分部分主要逻辑为双重循环，循环中分别判断“能否进行逻辑划分”和“能否进行物理划分”，直到两种划分都不可行，则停止循环。

其中，逻辑划分的判断逻辑的伪代码如伪代码 2.1 所示：第 2 行使用 *Kruskal* 算法计算获得当前关联图的最小生成树的边集合；第 3 行-第 4 行将最小生成树中的最小一条边切割，具体实现为第 3 行将所获得的边集合按照边权从小到大进行排序，在第 4 行将选择其中权值最小的一条边（即排序后的第一条边）并从图中删除；第 5 行-第 10 行将剩余部分中的每一个联通块视作一个 $Service_i$ 加入 *NewServices* 集合，其中第 7 行所调用的函数将利用 *DFS* 算法（深度优先搜索算法）从剩余部分中搜索得到联通块，第 6 行及第 13 行所涉及的 *visited* 数组为一个初始化为 FALSE 的用来标记 *node* 是否被遍历过的数组，其

作用为防止重复遍历导致的死循环或重复答案；第 12 行-第 18 行为主函数所调用的获取连通分量函数，通过 *DFS* 算法（深度优先搜索算法）将一个连通分量内的相连节点都放入点集 *NowService* 中以实现目标。第 10 行-第 11 行分别计算逻辑划分前后的结果内聚度及耦合度并以 *Metrics* 表示，第 12 行判断两个 *Metrics* 的优劣关系，若逻辑划分后结果更优，则认为该次逻辑划分是可行的，返回划分后新结果 *NewServices*；否则返回原始划分结果 *Services*。计算 *Metrics* 的具体方法以及判断 *Metrics* 大小情况的方法如下：

其中内聚度的计算公式如公式（5）所示：

$$CohesionVal = \frac{\sum relationScore(Class_{i,j},Class_{i,k})}{\sum[n_i*(n_i-1)/2]} \tag{5}$$

其中耦合度的计算公式如公式（6）所示：

$$CouplingVal = \frac{\sum relationScore(Class_{i,k1},Class_{j,k2})}{\prod n_i} \tag{6}$$

其中对比新结果是否更优的函数 *newIsBetter()*所涉及的公式详情如公式（7）所示：

$$newIsBetter(Metrics_{new},Metrics_{old}) = \left[\frac{|MetricsIncrease(Metrics_{old},Metrics_{new})|}{|MetricsReduce(Metrics_{old},Metrics_{new})|} > 1\right] \tag{7}$$

其中$MetricsIncrease(Metrics_{old},Metrics_{new})$指两个 *Metrics* 之间的提升量，即 *CohesionVal* 的提高量与 *CouplingVal* 的降低量之和。因为对于内聚度 *Cohesion* 而言，微服务的内聚度越高则表示其在逻辑上的独立性越高；而对于耦合度 *Coupling* 而言则恰好相反，微服务的耦合度越低才表示其在逻辑上的独立性越高。

而$MetricsReduce(Metrics_{old},Metrics_{new})$ 则恰好与之相反，指两个 *Metrics* 之间的降低量，即 *CohesionVal* 的降低量与 *CouplingVal* 的提高量之和。与上述原理相同，反之即可。

伪代码 2.1 SplitLogical 算法伪代码

```
算法：SplitLogical算法
输入: 关联图 RelationGraph，当前划分方案 Services
输出: 新划分方案 NewServices
初始化 NewServices 为空集合
RelationMST = KruskalMinimumSpanningTree(RelationGraph, Services)
sort(RelationMST)
remove(RelationMST[0])
FOR node IN RelationGraph.Nodes:
    IF visited[node] != TRUE:
        NewServices = NewServices ∪ GetConnectedComponentByDFS(RelationGraph, node)
    ENDIF
ENDFOR
Metrics_old = evaluate(Services)
Metrics_new = evaluate(NewServices)
IF newIsBetter(Metrics_new, Metrics_old)
    RETURN NewServices
ELSE
    RETURN Services
ENDIF

FUNCTION GetConnectedComponentByDFS(graph, node):
    visited[node] = TRUE
    NowService = {node}
    FOR neighborNode IN graph.getNeighbot(node):
        NowService = NowService ∪ GetConnectedComponentByDFS(graph, neighborNode)
    ENDFOR
RETURN NowService
```

物理层面划分算法 SplitPhysical 的伪代码如伪代码 2.1 所示：

伪代码 2.2　SplitPhysical 算法伪代码

算法：SplitPhysical算法

输入: 调用图 CallingGraph，当前划分方案 Services

输出: 新划分方案 NewServices

```
1:  初始化 NewServices 为空集合
2:  FOR Service_i IN Services:
3:      FOR degreeType IN {入度, 出度}:
4:          SplitedService ← GetEssentialPart(CallingGraph,Service_i,degreeType):
5:          IF SplitedService != ∅:
6:              NewServices ← NewServices ∪ SplitedService
7:          ENDIF
8:      ENDFOR
9:      NewServices ← NewServices ∪ Service_i
10: ENDFOR
11: RETURN NewServices
12:
13: FUNCTION GetEssentialParts (graph, Service, degreeType):
14:     初始化 essentialParts, essentialNodes 为空集合
15:     degreeMean = getMeanDegree(Service, degreeType)
16:     flowMean = getMeanFlow(Service, degreeType)
17:     FOR node IN Service.Nodes():
18:         IF node.degree > degreeMean*EssentialFactor AND node.flow >
flowMean*EssentialFactor:
19:             essentialNodes ← essentialNodes ∪ {node}
20:         ENDIF
21:     ENDFOR
22:     FOR edge IN Service.Edges:
23:         IF (degreeType == 入度 AND edge.callee ∈ essentialNodes ) OR
( degreeType == 出度 AND edge.caller ∈ essentialNodes ):
24:             delete(edge)
25:         ENDIF
26:     ENDFOR
```

续伪代码 2.2　SplitPhysical 算法伪代码

```
27:     FOR node IN essentialNodes:
28:         essentialParts = essentialParts ∪
GetConnectedComponentByDFS(graph, node)
29:     ENDFOR
30: RETURN essentialPart
```

第 2 行-第 10 行针对调用图进行分析，第 3 行-第 8 行中对出入度情况分别考虑。第 4 行调用 *GetEssentialParts()* 函数寻找关键部分 *EssentialParts* 并将其从原图中分割出来，若存在这样的 *EssentialParts* 则将其放入 *NewServices* 集合中，并在搜索结束后的第 9 行将剩下部分也放入 *NewServices* 集合中。

*GetEssentialParts()*函数的具体实现如第 13 行-第 30 行所示，第 15 行-第 16 行分别按照公式（8）和公式（9）计算平均度数与平均流量，若 *degreeType* 值为入度则只考虑入边（即指向当前 *node* 的边），否则只考虑出边（从当前 *node* 指向其它 *node* 的边）。第 17 行-第 20 行将遍历每一个 *node*，通过暴力搜索的方式找到关键节点。其中，判断条件中的 *EssentialFactor* 为判断关键节点的参数，具体如公式（10）所示，考虑到当一个图中的 *node* 个数越少则 *degree* 和 *flow* 都将越集中，所以 *EssentialFactor* 应当与 *node* 个数呈负相关，当 *node* 个数越少时 *EssentialFactor* 就应当越大，也就意味着条件越苛刻；而 para 为指定参数，在物理层面的意义即“在原始调用图中，关键部分各参数至少需要达到的平均值的倍数”，当 para 越大时对关键部分的要求越苛刻，在系统中该值默认为 3。

$$getMeanDegree(Service, \text{degreeType}) = \frac{\sum_{node \in Service.Nodes} node.\text{degreeType.degree}}{|Service.Nodes|} \tag{8}$$

$$getMean\text{Flow}(Service, \text{degreeType}) = \frac{\sum_{edge \in Service.Edges} \text{edge.flow}}{|Service.Nodes|} \tag{9}$$

$$EssentialFactor = \frac{para * graph.Nodes}{|Service.Nodes|} \tag{10}$$

用如图 2.3 所示的三种情况来具像化表述物理特征识别算法的具体效果。算法分别将“入度、流入流量”和“出度、流出流量”高度集中节点与其他部分切割开。当仅存在入度高且流量集中的情况（左图）时，将会将其单独划分出为其他各部分服务，起到类似于底层公共设施（例如数据库）的作用；当仅存在出度高且流量集中的情况（中图）时，将会将其单独划分出以单独调用其他部分，起到类似于公共接口的作用；当两种情况都存在（右图）时，将会将这样的中间部分划分出，起到公共中间件的作用。

这样的部分很容易成为整个系统的性能瓶颈，即当面对大流量时，系统的速度将取决于这些部分的运行速度；因此，将其单独划分后，系统面对大流量时可以只将这样的部分进行大规模伸缩，既能充分利用资源，又能显著提高系统性能。

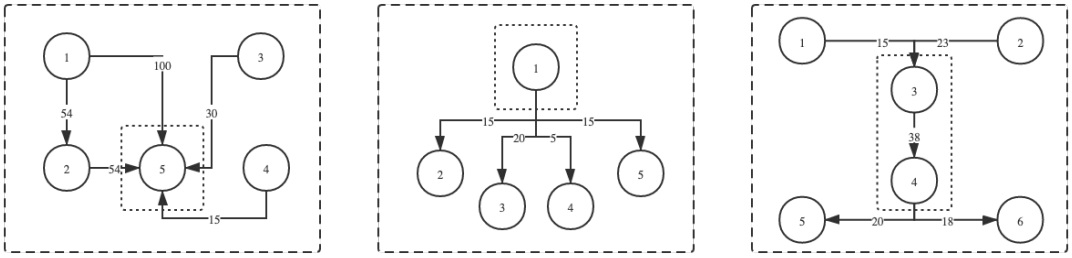

图 2.3 物理层面样例图

第3章 需求分析

3.1 用户需求

用户的核心需求只有一条：划分目标系统。我们的工具自然必须要能对目标系统给出较为合理的划分方案，这也是工具的基本功能。在此基础上还可以按时间顺序向前或向后追溯。

时间上向前追溯，用户可能对不同系统进行划分，也可能对同一个系统设定不同参数进行划分。在这个需求下，我们的工具需要给出可视化且操作简单的“系统选择”界面，“参数选择”界面以及系统信息及参数信息的前后端间传递功能。

时间上向后追溯，划分之后，用户需要直观地查看结果，同时会有反复查看数个划分结果的需求。在这种需求下，我们的工具应当有可视化界面展示所给出的划分方案，同时，每次划分所得到的结果应当被保存，工具也应给出“划分结果选择”，便于用户退出后重新查看结果。

3.2 前端业务流程

3.2.1 目标系统信息管理

该部分主要需要给用户提供系统选择功能。该部分中应当给用户提供能够传输系统相关信息的工具或接口，以及将相关信息传至服务端的传输模块。除此之外，还应当能够主动向服务端发起请求查询数据库中已保存的系统，并将所获取的信息展示给用户并提供便于其选择的表单模块。

3.2.2 划分参数管理

该部分需要给用户提供参数选择功能，主要用于补全划分算法所需要的信息。系统划分过程基于系统信息并根据划分参数执行不同细节：系统信息较为庞大与冗杂，因此在上一模块中单独保存并方便重复利用；而划分参数较为微小，通常只需要不到 1kb 信息，故只需在每次划分前传输至服务端即可。该界面的主要功能为表单提交，区别于上一模块的在于所需信息项目繁多但内容较少，需要关注于排版以便于用户进行设定。

3.2.3 结果信息管理

该部分主要需要“结果查看”界面以及“结果选择”界面。结果查看界面强调于用可视化手段将繁杂的文本信息转化为直观易懂的图像信息。结果选择界面则与之前几个部分中的选择部分相似，目的为便于用户选择并主动从服务端获取相关信息以进行展示。

3.3 服务端业务流程

3.3.1 目标系统信息存取

对应前端所传输的系统信息，该部分需要结合文件系统以及数据库将内容较多的系统信息选择合理方式保存，与此同时，还需要进行关键特征提取并以特征表示系统反馈给前端。

3.3.2 系统划分

该部分是整个系统的核心部分，需要实现所设计的算法并利用之前用户所输入的信息，最终将结果反馈至其他模块。

3.3.3 结果信息存取

该部分需要设计便于表达和存储的结果信息，要考虑到前端部分转化结果信息的难易程度，也要考虑到和数据库交互过程中的复杂程度。另外，该部分也需要提供查询接口，将结果反馈至前端。

3.4 本章小结

针对用户在系统划分问题下的不同需求，对应提出了对本工具的产品需求。按照时间顺序可表示为以下流程：

用户在前端“目标系统信息管理”模块的“系统选择”界面传入系统相关参数，同时可以选择已传入的其他系统。之后在前端“划分参数管理”模块的“参数选择”界面设定参数并将相关信息传至服务端。之后服务端根据传入的所有信息进行划分，将结果存储并返回至前端。前端的“结果信息管理”模块用可视化方式展示结果。全过程结束之后，用户还可选择查看之前的划分结果。

针对上述需求分析可设计出如图 3.1 所示系统架构：

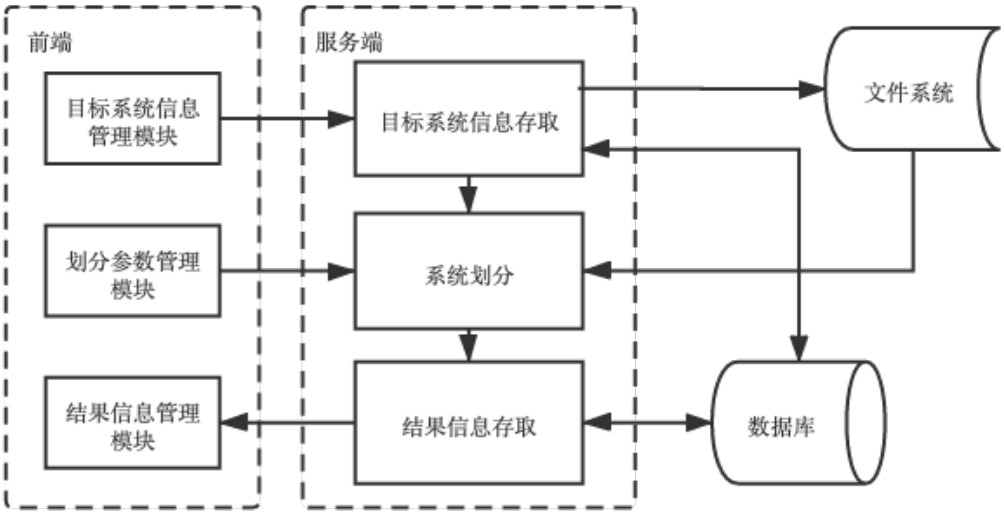

图 3.1 需求分析所得系统流程图

第4章 系统的设计和实现

4.1 整体框架

为便于用户可视化交互并降低用户使用难度，本系统按前后端分离的 Web 应用结构进行设计。前端部分基于 Angular 和 Bootstrap 框架进行设计和实现，同时基于 VisJS 可视化库完成数据可视化功能，服务端部分基于 Spring 框架、PostgreSQL 数据库进行开发。各模块间关系如图 4.1 所示。

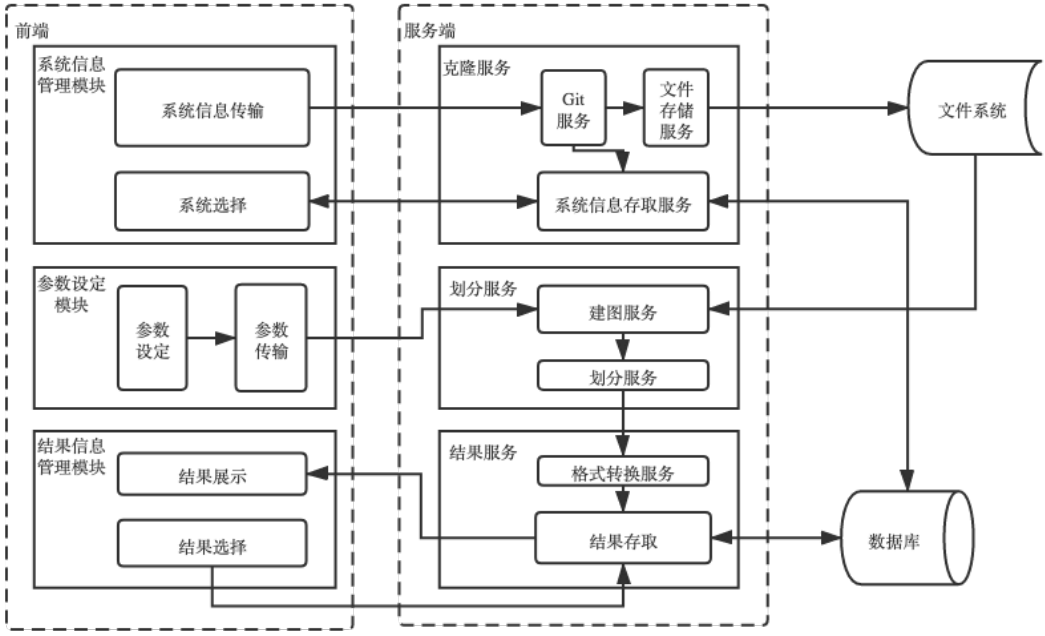

图 4.1 系统架构图

4.2 前端

前端部分主要分为三个模块：系统信息管理模块，参数设定模块，结果信息管理模块。

系统信息管理模块功能为传入系统相关信息。由于系统相关信息较为庞大与复杂，对同一个系统的多次划分过程中可以对这部分信息进行重用，以此来减少不必要的传输导致的额外时间开销。因此，在准备算法所需内容时，将系统相关信息和划分算法所需相关参数的传输分为两个部分。在这一部分中，系统相关信息的传输方案主要为：用户通过版本控制平台（Git）保存，传输项目地址给服务端，服务端通过平台接口自动克隆以获得相关信息。一方面来说，

用户正常开发过程中大多都会使用版本控制平台，因此不必给用户增加额外的负担；另一方面来说，将保存与传输任务交给第三方，能增加系统的可靠性，也能降低系统的复杂度。一个需要注意的问题是，由于运行监控过程需要用户提前执行，因此监控结果也应当放置在版本控制平台的项目目录下。除了传输之外，该部分也具有一定程度的管理功能，用户通过“系统选择”部分查看已保存信息的系统，也能直接通过该处进入特定系统的划分功能模块，这正是设计过程中所想要强调的重用功能。

其中前后端交互过程中所使用的系统信息数据结构如图 4.2 所示。

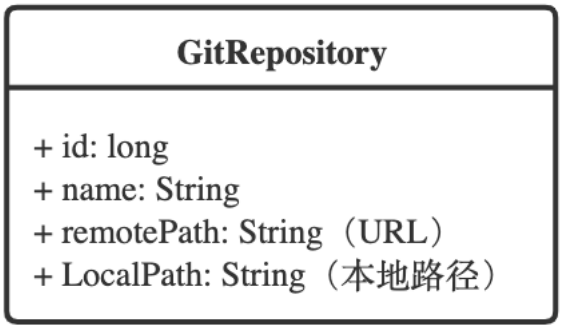

图 4.2 系统信息 UML 图

参数设定模块功能为传入划分算法所需相关参数。如上一段所述，该模块补充算法所需的另一个部分，该部分信息量较小但条目较多。该模块也是执行算法的主要入口，传入的相关参数会与之前传输保存的系统相关信息结合，交由服务端进行划分算法的执行。

前后端间所传递的参数数据结构如图 4.3 所示。

结果信息管理模块功能为接收划分结果并可视化展示。该模块主要功能为处理服务端划分算法的结果并对用户作出反馈。为方便用户对结果进行查看，该模块利用了 VisJS 可视化库将繁杂的文本数据转化为图数据，便于用户接收结果并进行下一步的处理。与系统信息管理模块类似，该模块也具有管理功能。在“结果选择”部分通过服务端获取数据库内信息并展示给用户，便于用户选择并查看历史结果。

用于可视化的结果信息的数据结构如图 4.4 中最右侧的 *GraphRepresentation* 所示。

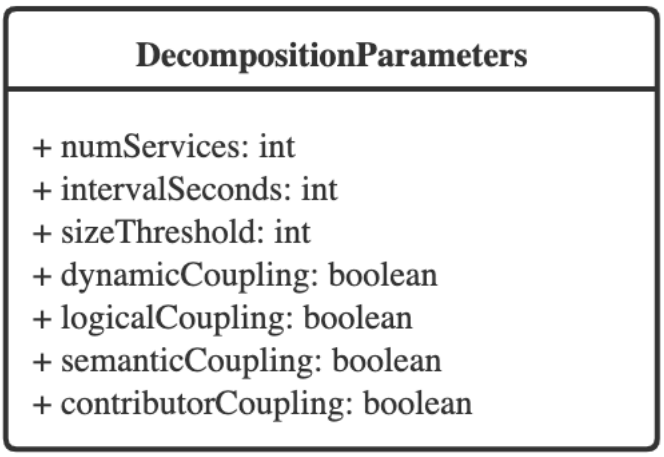

图 4.3 划分参数 UML 图

4.3 后端

后端部分主要分为三个模块：克隆服务模块，划分服务模块，结果服务模块。

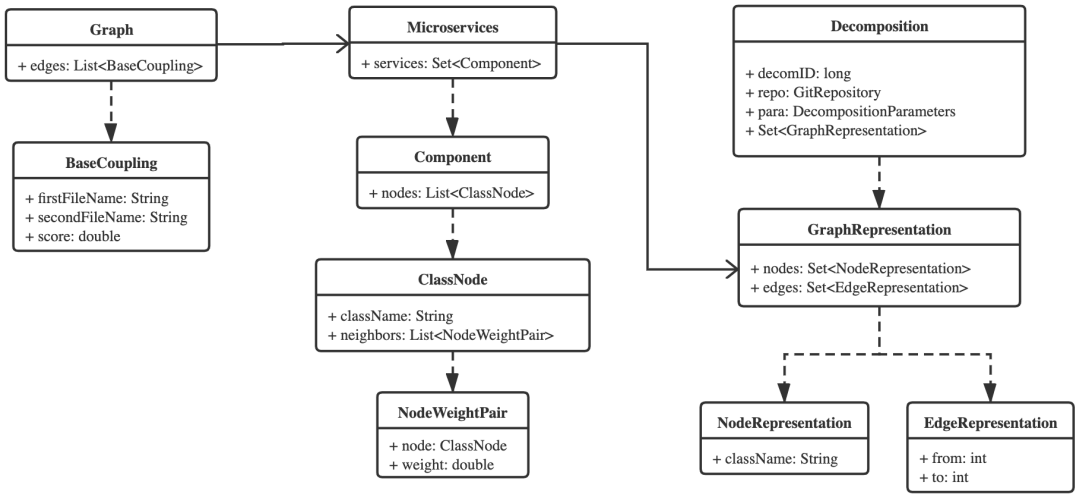

图 4.4 后端图结构 UML 图

前后端交流方案为 HTTP，前端通过 POST 或 GET 方式访问后端，Spring 框架中 Controller 层能够自动将 HTTP 请求解析至对应接口函数。各 Controller 层接口信息如图 4.5 所示。

克隆服务模块包含两方面的功能逻辑，第一方面是调用 Git 接口并与文件系统直接交流，将系统信息保存在文件系统下；另一方面是对数据库进行信息

存取，主要逻辑为提取目标系统特征并以系统名加系统编号的形式唯一表示每个系统，便于后续的数据增减操作或用户处理、浏览操作。

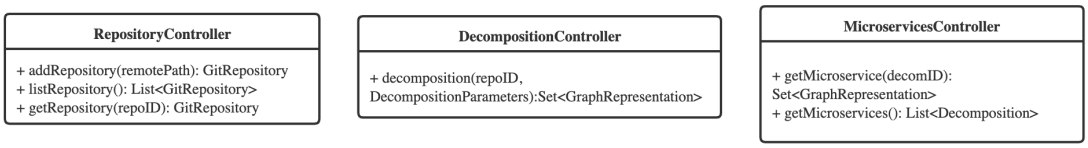

图 4.5 Controller 接口信息

划分服务模块是最单纯的算法操作模块，将数据存储功能分离至另外两个模块后，该模块专注于对微服务划分算法的实现。算法具体实现已在前文中详细描述，该处不再赘述。通过前端以及文件系统中的系统信息，该模块能够独立完成整个划分功能。在划分算法结束之后，划分服务模块会将结果传递至结果服务模块。此时划分结果还不够简单以便于前端的可视化展示，因此还需要格式转换服务对其进一步处理。

结果服务模块承担人机中介的角色以及数据管理的角色。人机中介意味着该模块利用格式转换服务简化数据，划分服务过程中的繁杂格式不必传至前端以增加不必要的逻辑功能负担。这既保证了划分服务只用关注于的复杂算法设计，便于维护人员排查错误，也保证了划分服务与其他部分在数据层面的隔离。这有助于算法设计中不用太多考虑结果的可读性，也有利于开发人员使用功能强大但专业性较强的算法或数据结构进行功能实现。另一个角色下，结果服务模块需要对数据库进行存取及维护，与划分服务模块类似，该模块也需要提取结果特征并在前端展示给用户以便浏览，结果部分的特征逻辑为系统编号与划分参数的结合，恰好对应着算法所需的两部分内容。为方便存取以及未来的复用，我们设计了数据结构 Decomposition 以存储每次划分的相关信息，其具体成员变量如图 4.4 所示。

在后端执行过程中，图数据在不同的阶段有不同的应用，不同应用中所抽象出了多种图结构的表示方式，具体 UML 图如图 4.4 所示。在划分划分服务模块中，本工具需要实现第 2 章中所提出的算法 PRBME，在算法的建图功能块中，对图结构的表示方式为使用数据结构 Graph（图 4.4 最左一列 UML），需要根据动态信息得到边权以建图，在这个过程中边为主要操作对象，故通过边表示法表示整张图结构。通过数据结构 BaseCoupling 表示边结构，既可以表示有向边也可以表示无向边。在 PRBME 算法的划分功能块中，我们需要存储 Service 信息，此时更关注的是点信息以及联通块信息，用之前的 Graph 则不太

方便，因此设计了新的数据结构 Microservice，其通过 Component 数据结构表示联通块，每个联通块中通过 ClassNode 数据结构来用点表示法表示整张图结构。最后，将算法的输出存储为结果并希望用可视化方式方便地展示时，我们则不再需要边权信息，只需要联通块信息即可。由于传输的需要，此时存储的信息越少越好，因此我们又使用了新的数据结构 GraphRepresentation 以表示联通块。此时将点数据和边数据作为同等级别信息，分别用数据结构 NodeRepresentation 和数据结构 EdgeRepresentation 来表示。

第5章 运行结果

5.1 系统展示

5.1.1 用户界面

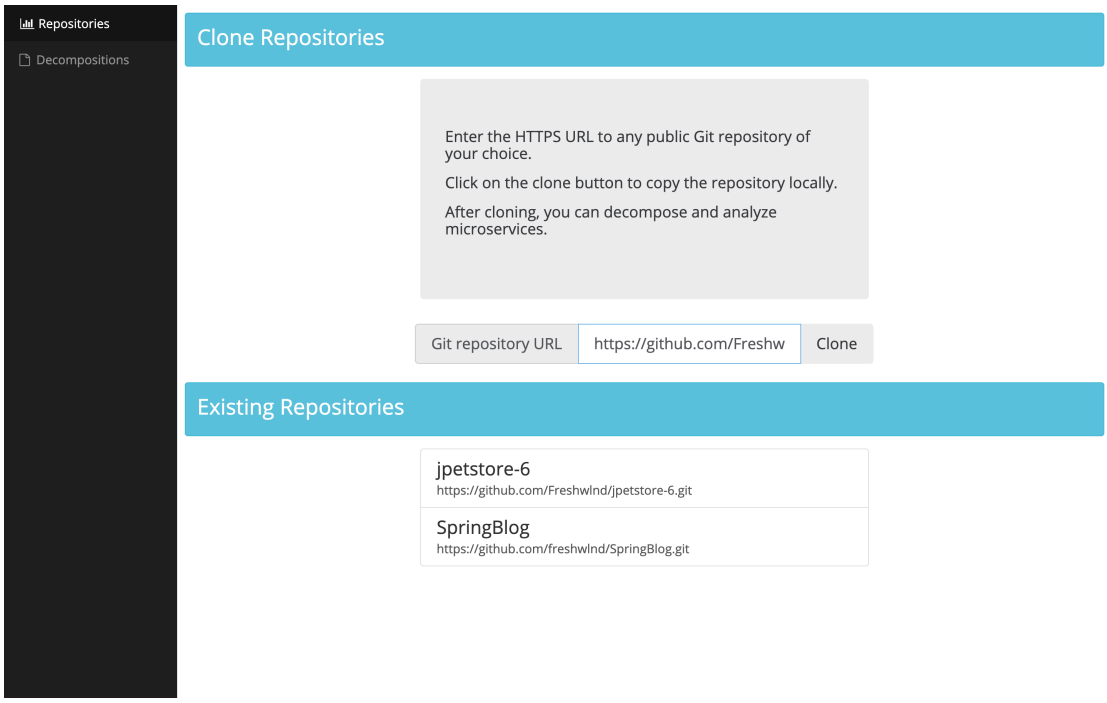

图 5.1 系统信息管理模块图

在该界面中，用户将系统对应的 Git URL 通过页面中部的文本框及按钮进行与系统的交互，通过下半部分模块进行查看已有系统并可以进行选择。选择对应系统后，将会跳转至参数设定模块。

在参数设定模块中，用户只需通过简单的选择与少量数据输入即可完成对参数的设定。之后通过 Decompose 按钮将数据传输到服务端，此时页面将会跳转至结果信息管理模块中的结果展示部分，等待服务端算法完成后便可直接查看结果。

若完成多次分解后用户想要查看历史结果，则可进入结果信息管理模块，依据所列举的每次划分选择所需的一个，便可重新进入结果展示部分，方便地查看结果。

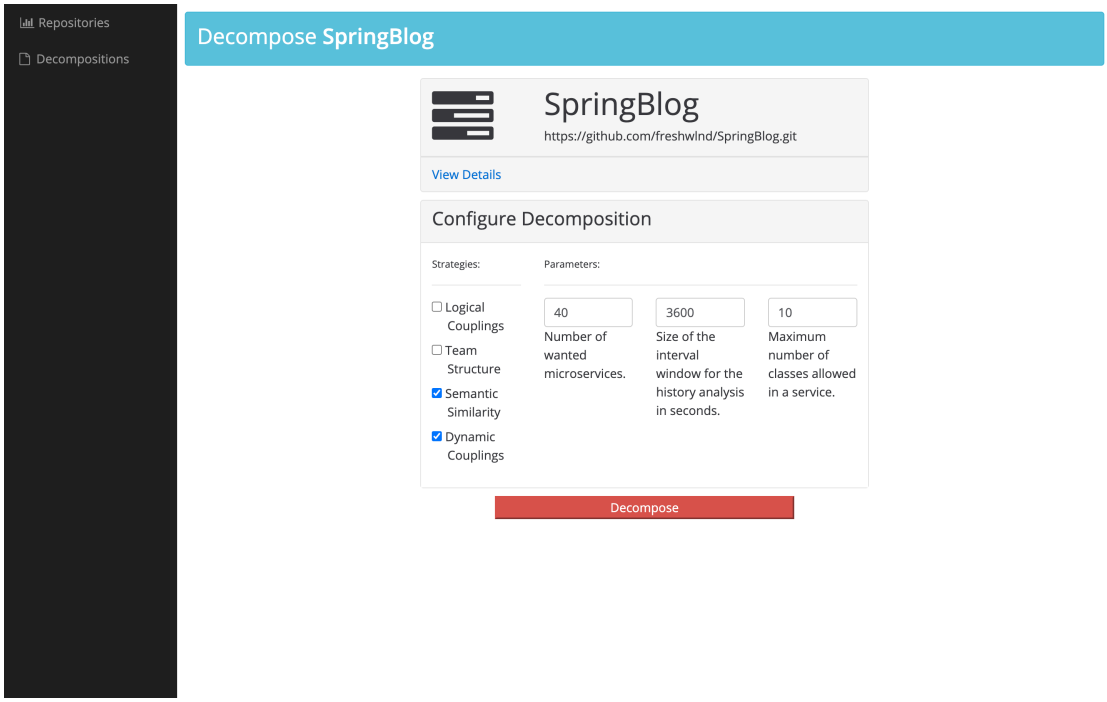

图 5.2 参数设定模块图

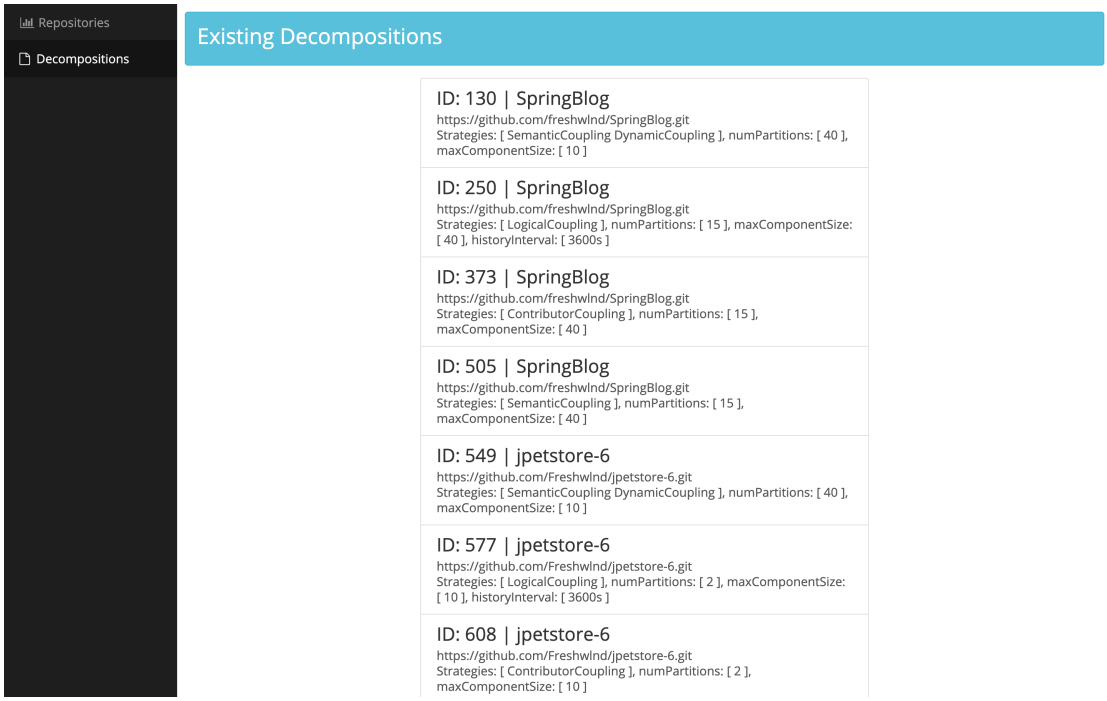

图 5.3 结果信息管理模块图

5.1.2 结果形式

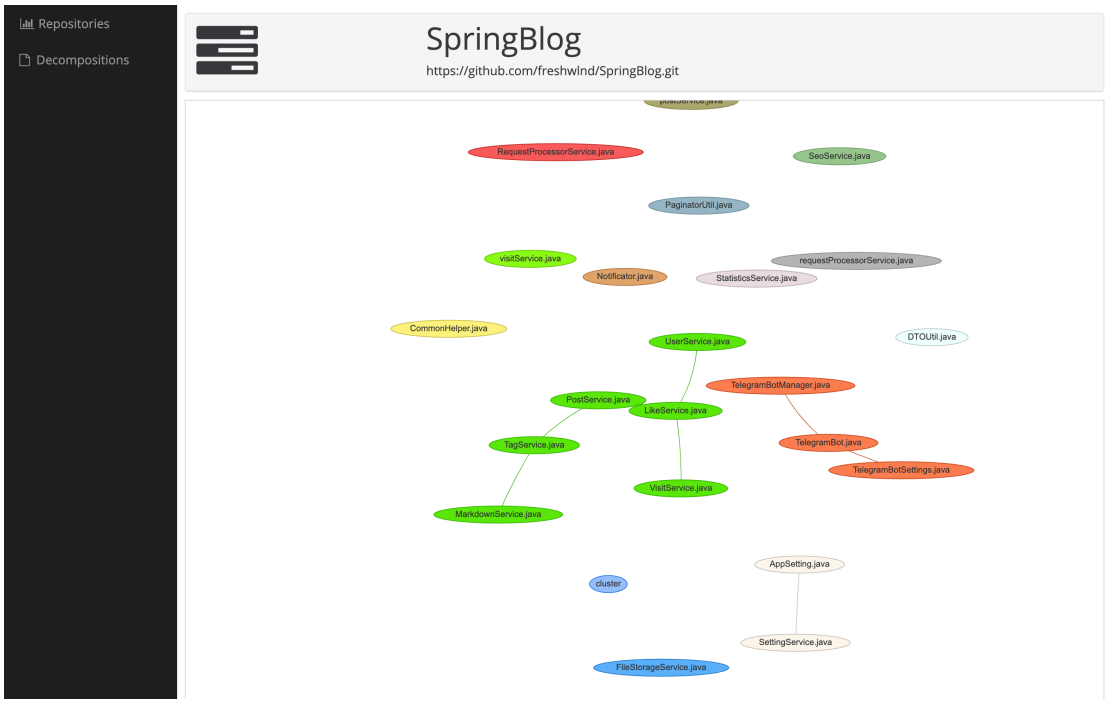

图 5.4 结果查看模块图

根据算法特性，最终结果会是一个个联通块的形式。为便于展示，不选择展现整体图结构，而是用树的形式体现各个联通块内的联通情况，忽略权值等部分繁杂内容。

5.2 实验结果

5.2.1 结果逻辑独立性评定

选择来自[9]中的三种算法 MEM-logical（下简称 MEM-l），MEM-semantic（下简称 MEM-s），MEM-contributor（下简称 MEM-c）作为对照，对两个系统 SpringBlog、JPetstore-6 进行划分。根据文本相似度及调用次数，计算微服务推荐结果中的内聚度及外耦合度，从两个维度对比显示推荐方案的质量。

（1）内聚度

从文本相似度来说：根据设计角度而言，类、函数及变量的命名应当反映其功能。从系统各部分的文本相似度进行评估，即可利用系统的设计思路衡量各部分间的功能相关度。为准确评估，需要将命名划分为单词，并去除常见的

停用词，根据 TF-IDF 计算相关度。对于每个微服务而言，用平均每对文件间文本相似度反映逻辑上的内聚度，计算平均每个微服务内部的任意两个文件间的文本相似度作为内聚度指标，如公式（11）所示。

$$SemanticCohesion = \frac{\sum_{x=1}^{m}\frac{\sum_{i=1}^{n_x}\sum_{j=1}^{n_x} SemanticSimilarity(File_i,File_j)}{n_x*(n_x-1)/2}}{m} \quad (11)$$

从调用次数方面来说，根据实际运行情况而言，调用次数可以反映任意两个文件的关联性程度，而对每个微服务而言，内部文件关联性越高则反映其逻辑独立性越高。同时内部文件个数也会影响整个微服务内的总调用次数，因此要将文件个数对调用次数的影响消去。最终，选择用平均每个微服务内部每对文件间的调用次数反映逻辑上的内聚度，计算平均每个微服务内部的任意两个文件间的调用次数作为内聚度指标，如公式（12）所示。

二者越高，内聚度越高，则微服务越独立。

$$DynamicCohesion = \frac{\sum_{x=1}^{m}\frac{\sum_{i=1}^{n_x}\sum_{j=1}^{n_x} CallingTimes(File_i,File_j)}{(n_x*(n_x-1))/2}}{m} \quad (12)$$

（2）外耦合度

和前文对文本相似度的分析一样，耦合度也可通过文本相似度来反映。一个微服务应当对应一个功能，因此可以将微服务内所有文件的关键词合并作为微服务的关键词集，之后微服务间的功能相关度就能通过这样的关键词集的文本相似度来计算得出，同样需要去除停用词并根据 TF-IDF 计算权值以及相似度。用平均每对微服务间文本相似度反映逻辑上的耦合度，如公式（13）所示。

$$SemanticCoupling = \frac{\sum_{i=1}^{m}\sum_{j=1}^{m} SemanticSimilarity(Microservice_i,Microservice_j)}{m*(m-1)/2} \quad (13)$$

和内聚度中的分析类似，不同微服务间的调用次数能反映其关联性，这种关联性越低说明微服务间耦合度越低，反映整体结构的逻辑独立性越高。若两

种方案划分次数相同，则微服务个数越多的方案耦合度越低。因此用平均每对微服务间调用次数即可反映逻辑上的耦合度，如公式（14）所示。

$$DynamicCoupling = \frac{\sum_{i=1}^{m}\sum_{j=1}^{m} CallingTimes(Microservice_i, Microservice_j)}{m*(m-1)/2} \quad (14)$$

最终考虑评估时，我们认为划分方案的内聚度越高，耦合度越低，则表示其逻辑独立性越好。

（3）实验结果

从图 5.5 和图 5.6 的实验结果可以看出，对前者的实验中，PRBME 算法结果在内聚度上与其他几种算法结果相差无几，而在耦合度上则稍微偏高。

对后者的实验中，PRBME 算法结果在调用情况耦合度指标上稍微偏高，而在其他几个指标上都与其他算法结果相差无几。

总结而言，PRBME 算法所产生的结果在逻辑独立性的内聚度上有较为可靠的保障，但由于其包含“将大流量公共模块单独划分”的划分逻辑，故使得其结果中微服务间存在更多的关联，因此耦合度会更高。但若划分出的“大流量公共模块”能够在物理层面的实验中发挥优异的表现，提高耦合度作为代价也是可以接受的。

从算法设计层面来看，这些现象合乎情理。由于有物理特征识别模块，该算法的最优化目标不只是逻辑上的独立性，而是同时追求物理上的可调控性，即便于弹性伸缩的特性。这导致在逻辑独立性最优的条件下，算法有可能会因为存在物理特征而进一步划分，使得关联紧密但流量集中的部分被划分开，导致内聚度反而降低。

虽然结果不够顶尖，但可以看出算法的效果已达到平均水平，可以认为能够符合逻辑独立性的基本要求。

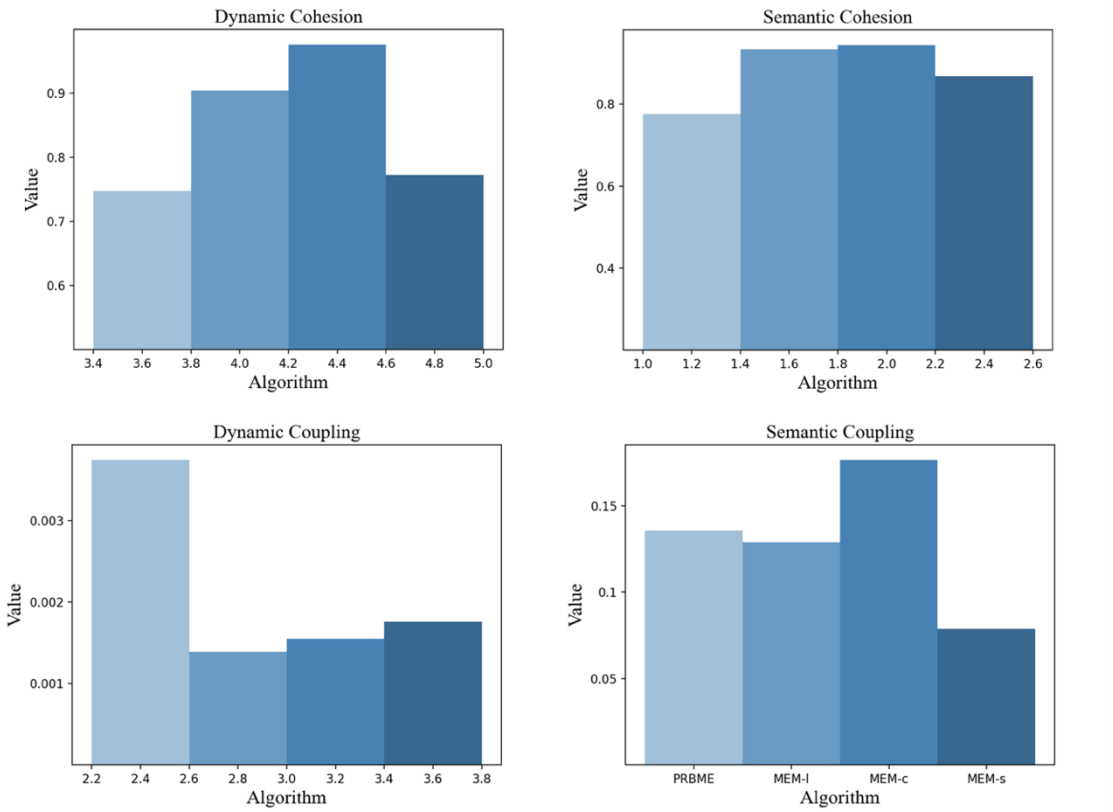

图 5.5 SpringBlog 逻辑实验结果图

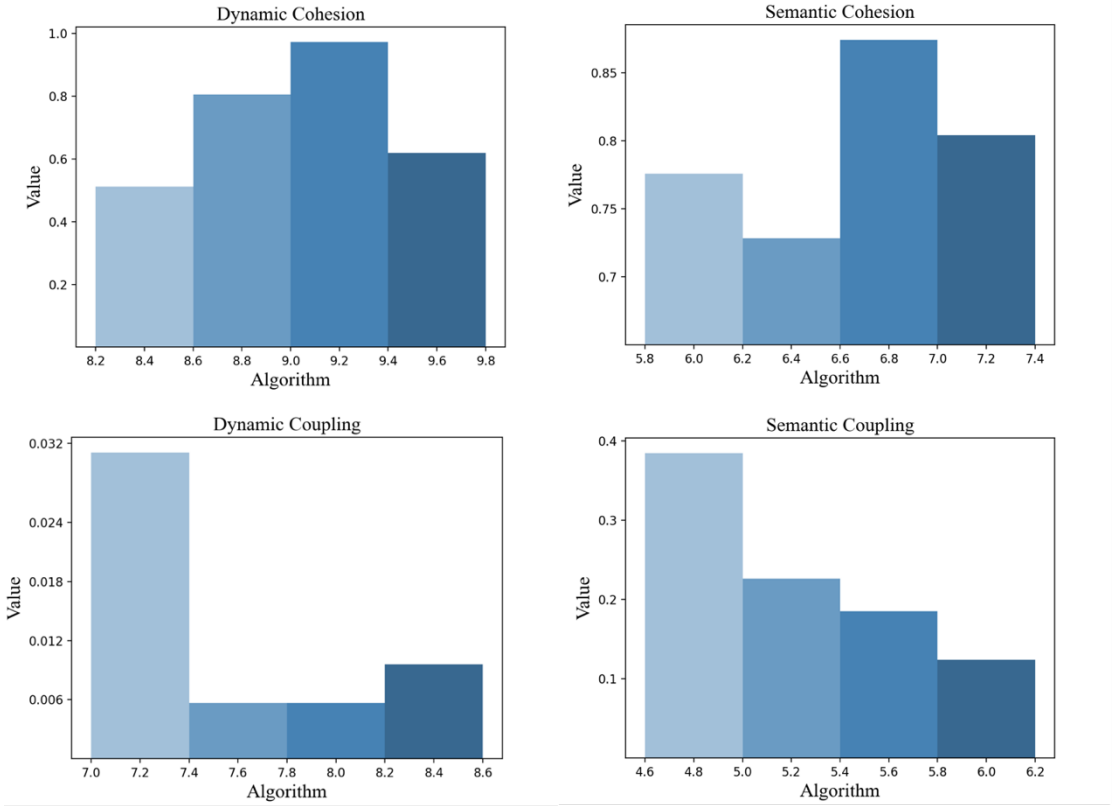

图 5.6 Jpetstore-6 逻辑实验结果图

5.2.2 实例模拟

（1）实验参数

物理层面实验选择以 JPetstore-6 项目作为样例，选择来自[9]中的算法 MEM-l 来与本文提出的 PRBME 算法作为对照。

将两种算法对 JPetsore-6 项目的划分方案部署成系统，并部署在同一个 K8S 集群中。实验所用 K8S 集群由两台 node 服务器组成，分别名为 vm-4c8g-node1 和 vm-8c16g-node12，前者为 master 节点。二者皆为 CentOS Linux 7 (Core) 系统，KERNEL 版本皆为 3.10.0-957.el7.x86_64。前者使用 4 核 2.10GHz 的 Intel(R) Xeon(R) Gold 6230 CPU，后者使用 8 核相同配置 CPU。前者内存容量为 7GB，后者内存容量为 15GB。部署 Pod 时采用 K8S 自动调控，默认策略为选择有空余容量的 node 进行部署，可认为两个 node 在逻辑上为同一台机器。

在弹性伸缩方面，采用 K8S 自带的 Horizontal Pod Autoscaler 配件进行水平自动弹性扩缩，每个微服务个数限制在 1 个- 3 个 Pod 之间，伸缩目标为 CPU 利用率低于 90%。

实验将原系统中后端面向前端所开放的接口集合作为 Requests 候选集，在该候选集中随机生成数个 Requests。之后将在不同 Requests 个数条件下反复测验所有 Request 的总响应时间，Requests 的个数分别为 1、10、100。

设定上述参数的原因如下：实验的基本思路为测试在资源无限/有限情况下，测试微服务系统对于外部请求的响应时间。Requests 个数由少到多对应着用户请求对系统资源需求的由少到多，在实验环境总资源有限的情况下，这就能够反过来使得实验环境总相对资源从多到少，通过控制变量实现目标环境。

此外每个 pod 的 Mem 资源需求量上限大约在 1500MB 左右，无请求时资源需求量约为 800MB。一个系统被划分为六种 Pod，由于两个系统同时部署且只有一个系统接受请求，可认为无请求系统占用资源量约为 4800MB，则剩下约 18000MB 空间供被测系统使用，约可部署 12 个-22 个 Pod。当六种 Pod 都达到资源请求上限时，每种 Pod 可部署 2 个。为增加系统资源竞争性但防止系统部署过多 Pod 导致的过大资源竞争对实验的影响，将每种 Pod 的个数上限设置为 3 个，即刚好可以竞争但竞争不会过于激烈的条件。

（2）实验方案

首先将系统 JPetstore-6 按两种算法所推荐的方案进行划分成两个系统并各自部署在同一个集群下。

之后随机生成 10 组个数为 1、10、100 的 Requests，对每一组相同请求 Request，分别对两个系统进行单独测试。为避免网络波动或物理设备问题等偶然性现象对实验结果产生影响，对每组 Request 重复测试 10 次，并将其中结果时间远低于平均值（result < 1e-5 s）或远高于平均值（result > 7s）的结果筛去。最后以请求个数和所用算法为变量，将筛选过滤后的结果取平均。

（3）实验结果

表 5.1 物理实验结果

请求个数 \ 算法名称	PRBME	MEM-l
1	0.0279646	0.01024734
10	0.06174104	0.09079731
100	0.39142251	0.74612471

表 5.1 中的数据为所有请求完成时间，单位为秒（s）。

根据实验结果可以看出，本算法在运行时间上有较为明显的优势。除请求个数为 1 的情况外，PRBME 算法的响应时间明显低于 MEM-l 算法的响应时间。

根据不同请求个数的数据判断，当请求个数为 1 时，总时间与传输时间基本在同一量级，可以认为此时影响总响应时间的关键在于传输时间。因此这种情况下的时间不能反映算法的效果，而应当主要分析另外两种情况。

根据算法的设计思路，在保证足够的逻辑独立性的前提下，算法识别物理层面的核心模块并将其单独切割，使得粒度更小，且核心模块被调用的几率远大于其他模块。

当资源无限时，需要被伸缩的核心模块规模更小，系统在弹性伸缩过程中的代价更小，反映在总响应时间上也就是所花费的时间更少。而在资源有限时，由于粒度更小，被划分出的核心模块与未被划分出的整体模块比所需资源量更少，相当于资源被用在了更关键的地方。例如被划分出的核心模块所需内存容量为 300MB，而未被划分的整体模块所需内存容量为 500MB；当系统环境内存容量仅有 1500MB 时，可部署被划分出的核心模块 5 个，而只能部署未被划分的整体模块 3 个；相当于在单位时间内被划分系统可处理 5 单位请求，而未被划分系统只能处理 3 单位请求，显然能够有更优秀的整体响应时间。可见被划分系统更能充分利用资源，达到更优秀的效果，即用更短时间完成所有请求。

5.2.3 实验结果总结

由实验结果可以看出，本文所提出的 PRBME 算法在传统实验（逻辑独立性实验）中的表现并不落后，且同时在实际运行实验（实例模拟实验）中的表现比传统算法优秀许多，故可证明 PRBME 算法具有充足的有效性。

第6章 总结及下一阶段研究方向

6.1 总结

微服务自动划分工具是一个当前环境下隐含的需求，对于自动划分算法的研究还不成熟，没有形成体系，受到的关注也较少。目前的研究中，大都只考虑微服务划分的逻辑层面要求，即指考虑令结果具有高内聚低耦合的逻辑独立性；且各研究的实验模块也不贴近实际，只具备理论可行性。本文提出基于物理特征识别的微服务划分算法，同时完成相对应的划分工具，并进行实际部署实验以验证效果。最终实验结果显示本算法能够在保证不低的逻辑独立性的同时，在实例运行中具有明显的优势。

6.2 下一步

从目前的方案而言，用以对比的算法还不够广泛，同时目标系统范围也较少，后续还可以继续深入实验以验证算法的有效性。另外，目前的微服务划分算法结果距离实际使用还有一定距离，后续将会结合实际划分过程中的经验，尝试对整个微服务划分问题作出更有实际效益的定义与研究。

第7章 参考文献

正文参考文献

[1] Cito J, Leitner P, Fritz T, et al. The making of cloud applications: An empirical study on software development for the cloud[J]. Joint Meeting on Foundations of Software Engineering, 2015: 393–403.

[2] Thönes J. Microservices[J]. IEEE Software, 2015, 32: 116-116.

[3] Newman S. Building microservices: designing fine-grained systems[M]. O'Reilly Media, Inc., 2015.

[4] Alessandra Levcovitz, Ricardo Terra, Marco Tulio Valente. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems[J]. 3rd Brazilian Workshop on Software Visualization, Evolution and Maintenance, 2015: 97-104.

[5] Schermann G, Cito J, Leitner P. All the services large and micro: Revisiting industrial practice in services computing[J]. International Conference on Service-Oriented Computing. 2015: 36–47.

[6] Baresi L, Garriga M, De Renzis A. Microservices Identification Through Interface Analysis[J]. European Conference on Service-Oriented and Cloud Computing, 2017: 19-33.

[7] Krause A, Zirkelbach C, Hasselbring W, et al. Microservice Decomposition via Static and Dynamic Analysis of the Monolith[J]. IEEE International Conference on Software Architecture Companion, 2020: 9-16.

[8] Al-Debagy O, Martinek P. Extracting Microservices' Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach[J]. 2020: 289-294.

[9] Mazlami G, Cito J, Leitner P. Extraction of Microservices from Monolithic Software Architectures[J]. IEEE International Conference on Web Services, 2017: 524-531.

[10] Fola-Dami Eyitemi, Stephan Reiff-Marganiec. System Decomposition to Optimize Functionality Distribution in Microservices with Rule Based Approach[J]. IEEE International Conference on Service Oriented Systems Engineering, 2020: 65-71.

[11] Jin W, Liu T, Zheng Q, et al. Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering[J]. IEEE International Conference on Web Services , 2018: 211-218.

附录

致谢

四年的紧张学习时光，我在浙江工商大学的学习生涯即将随着这篇毕业论文拉下帷幕。与完成本文前面科研或开发部分时相比，看到“致谢”二字时的我就自动抛开了理性与严谨，只觉得一股股热流从心脏涌向全身又汇聚于头顶。

很高兴也很荣幸能够在浙江工商大学学习，也很荣幸能够与各位老师相识，这里的每一位老师都有着自己鲜明的特点。无论是我的课程老师还是只闻其名的老师，在我有问题发起询问时都会耐心地对我指点与解答，在我对课堂内容发表疑惑时也不会大摆架子，只会与我理性探讨。我要感谢浙商大信息学院所有老师，是你们无私的奉献精神和爱岗敬业精神感染了我，相信这些在举手投足间散发的气质会对我的人生有着巨大的作用，尽管看上去并不是什么庄严的大事件，却会在我的心中埋下名为进步的种子。在论文完成之际，我要在这里感谢我的导师韩建伟老师，从大一起他认真负责的态度就刻在我的脑海，每每在学期初看到本学期课表上有他的课就会感到一阵兴奋。

也要感谢我的每一位同学，能拥有这一段快乐而温暖的大学时光，同学们的相互照耀自然起着无比重要的作用。如今即将分别，只期待来日江湖上再见，不知到时又会是怎样一番光景。

最后要单独感谢隋婷婷同学，在我无数次低落时给予我抚慰，在我无数次失意时助我觅见朝阳。你如朝露清新又如皎月优雅，很高兴能与你一同见识世界的多彩，也很庆幸能在最美好的年华与你相遇。未来尚布满云雾，来日方长，还请你多多指教。