南京大學

研 究 生 毕 业 论 文
（申 请 硕 士 学 位）

论　文　题　目　面向版本一致性的微服务动态更新技术研究

作　者　姓　名　王东宇

学科、专业方向　计算机科学与技术

研　究　方　向　软件方法学

指　导　教　师　曹春 教授

2021 年 **5** 月 **29** 日

学　　号: **MG1833071**

论文答辩日期: **2021** 年 **5** 月 **25** 日

指 导 教 师:　　（签字）

Research on Microservice Dynamic Update Technology With Version Consistency

by

Dongyu Wang

Supervised by

Professor Chun Cao

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfilment of the requirements for the degree of
MASTER
in
Computer Science and Technology

Department of Computer Science and Technology
Nanjing University

May 24, 2021

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：面向版本一致性的微服务动态更新技术研究

计算机科学与技术 专业 2018 级硕士生姓名：王东宇

指导教师（姓名、职称）：曹春 教授

摘　要

基于微服务架构 (Microservices Architecture) 的开发可以有效地缩短开发周期，满足了软件快速持续集成和持续交付的需求，在工业界得到广泛的应用。已部署的软件通常需要不断地进行更新迭代，以达到漏洞修复、增强功能以及适应外部需求等多方面的目的。然而传统的更新方式通常需要停止系统、部署新版本后再重新启动系统。很多关键领域的系统并不能接受停机更新所导致的系统中断，因此软件动态更新技术显得尤为关键。

当前相关的微服务框架与架构对服务的动态更新支撑不足，没有考虑动态更新过程中系统的一致性。基于此，本文给出了一种面向版本一致性的微服务动态更新方法，并在具体的微服务架构上进行模型的扩展，完成了系统的设计与实现。此外，将本文的方法与实际的应用案例进行整合，完成了此方法和相关更新算法的实验评估对比，验证了其在更新过程中的安全性、及时性和低干扰性，以及对多服务同时进行更新的支持。

具体而言，本文的工作主要包括：

- 提出一种面向版本一致性的微服务动态更新的实现方法。此方法针对当前主流微服务框架技术在动态更新支撑方面的不足，在事务模型、动态依赖管理和服务状态模型三个方面进行了相应的扩展，为具体的动态更新过程提供运行时支持。
- 基于上述的实现方法，在开源实现框架 Istio 之上完成了模型的扩展，添加系统对动态更新的支持。该系统不仅保证动态更新过程的安全性、及时性和低干扰性，而且实现的扩展模块对用户保持透明，与具体的服务业务逻辑解耦开。
- 利用实际的应用案例，整合上述实现方法并进行实验评估。通过使用不同的更新算法，主要对动态更新过程中各个算法的安全性 (Safety) 、及时性 (Timeliness) 、干扰性 (Disruption) 进行了性能比较，具体实验中同时向多个

服务发起动态更新请求，验证了本文实现方法对于多服务同时更新的可行性和有效性。

关键词：微服务架构；版本一致性；动态更新

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Research on Microservice Dynamic Update Technology With Version Consistency

SPECIALIZATION: Computer Science and Technology

POSTGRADUATE: Dongyu Wang

MENTOR: Professor Chun Cao

Abstract

Development based on the microservices architecture can effectively shorten the development cycle, meet the needs of rapid continuous integration and continuous delivery of software, and is widely used in the industry. The deployed software often need to be updated continuously to achieve various purposes such as bug fixes, enhancements, and adaptation to external requirements. However, the traditional way to update a software system is to stop the system, replace it with the new version, then restart it. This is not acceptable for some mission critical systems for the interruption caused by downtime updates, so in such situations we need dynamic software update (DSU) .

Existing microservice frameworks and architectures provide limited supports for DSU, and do not take in account the system consistency during the dynamic update process. Based on this, this thesis presents a method for microservices that supports dynamic update with version consistency, extends the models on specific microservices architecture, and completes the design and implementation of the system. We have also integrated the method with the actual application case, completed the experimental evaluation and comparison with the related update algorithm, and the safety, update timeliness and low disruption during the update process , as well as support for simultaneous updates of multiple services are verified.

The work of this paper can be mainly summarized as follows:

- This thesis presents a method for microservices that supports dynamic update with version consistency. Aiming at the deficiencies of the existing microservice frameworks in dynamic update support, this method extends the transaction model, dy-

namic dependency management and service state model to provide runtime support for specific dynamic update process.

- Based on the implementation method, the model is extended on the open source implementation framework Istio and support for dynamic update is added. The system not only ensure the safety, update timeliness and low disruption of the dynamic update process, but also the extension modules remain transparent to users and decoupled from the specific service business logic.
- We have chosen a practical application case to integrate the implementation method and conduct experimental evaluations. Through the use of various update algorithms, the focus is put on the safety, update timeliness and disruption. In the specific experiments, dynamic update requests are initiated to multiple services at the same time, which verified the feasibility and effectiveness of the method implemented in this paper for multi services simultaneous update.

keywords: Microservices Architecture, Version Consistency, Dynamic Update

目　　次

目　　次 ····· **v**

插图清单 ····· **vii**

附表清单 ····· **ix**

1　绪言 ····· **1**

1.1　研究背景 ····· 1

1.1.1　微服务架构 ····· 1

1.1.2　软件动态更新技术 ····· 2

1.2　研究现状 ····· 2

1.3　本文工作 ····· 4

1.4　本文组织 ····· 4

2　相关工作 ····· **7**

2.1　软件动态更新技术 ····· 7

2.1.1　软件系统实例 ····· 8

2.1.2　Quiescence 算法 ····· 9

2.1.3　Tranquility 算法 ····· 11

2.1.4　Version Consistency 算法 ····· 12

2.2　微服务架构与动态更新 ····· 14

2.3　本章小结 ····· 15

3　面向版本一致性的微服务动态更新方法 ····· **17**

3.1　问题分析 ····· 17

3.2　方法概述 ····· 19

3.2.1　请求状态支持 ····· 19

3.2.2　安全更新点定义 ····· 20

3.2.3　服务状态模型 ····· 21

3.3 本章小结 ····· 28

4 支持动态更新的微服务系统实现 ····· 29

4.1 系统设计概述 ····· 29

4.2 系统扩展实现 ····· 30

4.2.1 事务依赖管理器 ····· 31

4.2.2 服务状态管理器 ····· 33

4.2.3 更新控制模块 ····· 38

4.3 系统模块交互 ····· 40

4.4 本章小结 ····· 41

5 实验评估与应用展示 ····· 43

5.1 应用介绍 ····· 43

5.2 实验环境与设计 ····· 45

5.2.1 实验环境 ····· 45

5.2.2 实验设计 ····· 45

5.3 实验分析 ····· 46

5.3.1 安全性 (Safety) ····· 47

5.3.2 及时性 (Timeliness) ····· 47

5.3.3 干扰性 (Disruption) ····· 48

5.4 实验结论 ····· 50

5.5 应用展示 ····· 51

5.6 本章小结 ····· 54

6 总结与展望 ····· 55

6.1 工作总结 ····· 55

6.2 研究展望 ····· 56

致 谢 ····· 57

参考文献 ····· 59

简历与科研成果 ····· 65

学位论文出版授权书 ····· 67

插图清单

2-1 示例系统静态依赖图 ····· 9
2-2 示例系统时序图 ····· 9
2-3 Quiescence 节点状态转移图 ····· 10
2-4 系统运行时动态依赖关系示例 ····· 13

3-1 主流动态更新流程示例 ····· 18
3-2 事务状态转移图 ····· 20
3-3 服务状态模型 ····· 23
3-4 初始状态下的系统 ····· 24
3-5 更新准备前的系统 ····· 24
3-6 版本更新中的系统 ····· 25
3-7 维护动态依赖信息 ····· 25
3-8 阻塞访问 ····· 26
3-9 同步状态 ····· 27
3-10 等待撤销 ····· 27
3-11 更新完成后系统状态 ····· 28

4-1 Service Mesh 架构 ····· 30
4-2 Istio 架构图 ····· 31
4-3 注入模块配置 ····· 32
4-4 微服务实例模型 ····· 32
4-5 控制器模式 ····· 34
4-6 Normal 状态下的路由规则 ····· 36
4-7 Prepared 状态下的路由规则 ····· 37
4-8 Shifted 状态下的路由规则 ····· 37
4-9 系统整体架构图 ····· 41
4-10 版本更新时序图 ····· 42
4-11 版本撤销时序图 ····· 42

5-1 台风系统静态依赖关系图 ····· 44
5-2 台风系统时序图 ····· 44
5-3 及时性实验结果 ····· 48
5-4 总干扰性实验结果 ····· 49
5-5 平均干扰性实验结果 ····· 50
5-6 系统运行状态图 ····· 51
5-7 错误路径状态图 ····· 52
5-8 前端模块展示 ····· 52
5-9 动态更新结束 ····· 53
5-10 正确路径状态图 ····· 53

附表清单

2-1 动态更新算法比较 ····· 13

5-1 安全性实验结果 ····· 47

第一章　绪言

1.1　研究背景

1.1.1　微服务架构

在软件技术的发展过程中，诞生了多样化的软件架构和技术。随着软件系统规模的扩大、功能的追加扩展，传统的单体应用 (Monolithic) 将应用程序的所有功能打包成一个独立的单元，最终将成为一个庞然大物，变得越来越复杂，逻辑耦合严重，难以理解。因此为了满足用户对于一定规模的软件的快速开发集成、提高可扩展性等要求，必须将其迁移到微服务架构中来[1–4]。微服务架构 (Microservices Architecture) 是一种基于一组独立部署运行的小型服务来构建应用的架构方法，服务间使用与语言无关的轻量级的通信协议进行通信[5,6]。与传统的单体应用相比，微服务架构可以帮助我们对应用进行有效地拆分，缩短软件开发周期，降低软件维护成本，满足了软件程序对于快速持续集成和持续交付的需求[7,8]。由于前述的相关特性，微服务架构开始在诸多公司内部得到广泛的应用，包括 Amazon[1]、Netflix[2]、Spotify[3]等，成熟的微服务框架有 Spring[4]、Dubbo[5]。

微服务架构的基本思想是将应用软件拆分为若干个较小的服务，不同的服务可由不同的团队使用不同的技术栈进行开发，从而达到解耦和降低复杂度的目的[5]。微服务架构的开发流程中的关键步骤在于对服务的拆分。微服务在进行拆分时需要遵循服务独立、数据一致、高内聚低耦合等原则，并且随着需求的迭代，服务的拆分需要持续的进行优化。

从开发过程来看，微服务架构尽量采用去中心化的管理机制，服务主要围绕应用系统的业务能力来进行开发。从软件架构来看，一个微服务架构的系统需要满足一系列的条件和原则，而不仅仅是使用了某个微服务框架即可。因此

[1]https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/
[2]https://www.nginx.com/blog/Microservices-at-netflix-architectural-best-practices/
[3]https://www.infoq.com/presentations/linkedin-Microservices-urn/
[4]https://spring.io/projects
[5]https://dubbo.apache.org/zh/

微服务架构更多地被看成一种架构风格，而不是一种具体的架构。

现有的关于微服务架构的研究较多关注于服务敏捷开发、服务治理等技术，而对于如何保证微服务在运行时正确地进行更新，相关的技术研究较少。

1.1.2 软件动态更新技术

Internet 的发展推动了软件技术的快速发展，而随着软件外部环境和用户需求的不断变更，已部署的软件同样需要及时且不时地进行更新迭代，来达到修复 bug、增加功能、适应需求变化等多方面的目的。传统的软件更新通常需要停止旧版本软件的运行，然后完成软件的更新，再重新进行新版本软件的部署。显然，这种更新方法会引起一定时间内的系统中断，但某些服务提供商需要为用户提供全天候不间断的服务，对于企业服务，系统的中断将会导致经济效益的损失；而在航空和医疗领域，系统的中断将可能威胁到人的生命安全[9–11]。因此，此类软件系统不能接受传统的更新方式，需要在不停止软件正常处理用户请求的情况下对其进行更新，保证系统服务的持续可用，即所谓的软件动态更新[12]。然而，实现软件的动态更新并不简单，它在安全性[1]、及时性、低干扰性等多方面面临着较大的挑战。

从软件动态更新的粒度来看，基于软件的动态更新可大致分为三个层面：

- **业务过程层面** 借助可变过程模型建模和过程迁移技术，可以将过程模型的变化迅速反应到过程的动态执行上。
- **程序代码层面** 在程序代码层面，对新旧版本程序进行差别分析，将旧版本程序的状态动态迁移到新版本中，并按照新版本的程序继续执行。
- **软件服务层面** 在构成软件服务的实例层面，研究面向动态更新的服务容器等，实现软件系统的动态更新和重配置。

本文主要关注于从软件服务层面对软件进行动态更新。

1.2 研究现状

目前在工业界较为常见的微服务动态更新部署方式有：蓝绿部署[13]、金丝雀部署[14,15] 和滚动发布。其中蓝绿部署利用冗余的硬件设备来加载新版本的软件服务，属于基于硬件的动态更新技术，这种方式时间和价格成本较高；另

[1]本文中的安全性和一致性均指系统在动态更新过程的前后都能正确运行

外两种则属于基于软件的动态更新技术，在旧版本实例运行时同时运行新版本服务实例，利用划分用户流量的方式，将用户请求逐渐地转发往新版本服务实例，直至将系统中所有服务实例变为新版本，完成更新。

对于基于微服务的动态更新，文献[16] 中实现的 JRO 基于指定的目标体系结构来完成微服务的自动更新部署，但局限于特定的编程语言 Jolie[17]。文献[18] 描述了一个支持微服务演化的模型，该模型涵盖了微服务架构与功能相关内容，确保微服务完成重构的一致性。文献[19] 提出了一种描述相关 PaaS 站点信息及其内部微服务部署信息的架构模型，运维人员通过指定更新策略来对特定的架构模型进行更新操作，最终反映到具体的服务中。这些方法实现了微服务的更新替换，但都缺乏对更新过程中运行时事务的考虑，无法保证动态更新前后系统运行的安全性。

如何选择正确的更新点来保证动态更新前后系统运行的安全性，是动态更新中至关重要的问题，一致性要求系统中的运行时事务在动态更新前后都不会出现正确性问题[20]。在这方面的研究已经存在较多的研究成果。文献[12,21] 中的 CONIC 系统，提出了 Quiescence 更新算法，该算法给出了动态更新过程中对构件的管理包括增加、删除和替换等操作，而且定义了 Quiescence 状态并将其作为可以执行动态更新的安全时机。但为使系统到达 Quiescence 状态，算法要求系统中断所有外来的业务处理直至目标更新完成，对系统的干扰极大。因此在文献[22] 中放松了对安全点的定义，提出了 Tranquility 动态更新算法。但对于某些具体应用，Tranquility 算法虽然大大降低了更新对系统造成的干扰，它仅仅保证了事务的局部而非全局一致性，因此导致它所定义的安全点可能并不安全。综合前述两种算法在动态更新过程中的问题，文献[23,24] 提出了 Version Consistency 算法，该算法利用运行时动态依赖关系来对安全点进行定义，在保证动态更新过程的安全性的同时较大程度地减少了对系统带来的干扰，在安全性和干扰性等方面取得了较好的平衡。

想要将动态更新应用于实际的微服务系统中，尚存在诸多问题。主要原因在于当前的动态更新算法通常都建立在特定的系统假设和约束条件之上，当系统的具体运行环境泛化时，相关的框架系统缺乏对动态更新的支撑。现有的微服务系统对动态更新支持的不足主要体现为：

- **安全性不足** 无法保证在系统中某个目标服务在动态更新过程中，正在运行时事务和将要发生的事务的一致性。
- **及时性不足** 当收到动态更新请求后，系统需要长时间的操作才能完成，无

法快速及时地完成目标服务的动态更新。

- **干扰性过大** 系统在进行动态更新的过程中，需要长时间地中断目标服务，对目标服务造成的干扰性过大。

1.3 本文工作

随着微服务开发架构的流行，微服务系统变得多元且复杂，而现有的微服务架构并未对服务的动态更新提供相关的支持。本文针对上述动态更新技术中存在的问题，提出了一种面向版本一致性的微服务动态更新实现方法。该方法针对性在相关模型上完成了扩展与实现，为系统中多个服务的动态更新提供支持，对实际场景中不同类型的微服务系统均具有透明性。整体来看，本文的工作主要包括：

- 提出一种面向版本一致性的微服务动态更新的实现方法。通过分析当前主流微服务系统框架的基本功能和原理，得出现有框架技术在动态更新支撑方面的不足。针对此问题，本文提出了一种面向版本一致性的微服务动态更新的实现方法，在事务模型、动态依赖管理和服务状态模型三个方面进行了相应的扩展，为具体的动态更新过程提供运行时支持。
- 基于上述的实现方法，在 Service Mesh 的开源实现框架 Istio[1]之上完成了模型的扩展，添加系统对动态更新的支持。该系统不仅保证动态更新过程的安全性、及时性和低干扰性，而且实现的扩展模块对用户保持透明，与具体的服务业务逻辑解耦开。
- 利用实际的应用案例，整合上述实现方法并进行实验评估，使用不同的更新算法，主要对动态更新过程中各个算法的安全性 (Safety) 、及时性 (Timeliness) 、干扰性 (Disruption) 进行了性能比较，具体实验中同时向多个服务发起动态更新请求，验证了本文实现方法对于多服务同时更新的可行性和有效性。

1.4 本文组织

本文内容组织如下：

第二章从软件动态更新和已有的微服务框架对动态更新的支持两个方面对

[1]Istio home page: https://istio.io/

相关工作进行介绍。在软件动态更新方面，主要介绍已有的动态更新算法，包括 Quiescence、Tranquility 和 Version Consistency；然后重点介绍相关微服务框架在动态更新方面的支持，以及相关的云平台对微服务架构在动态更新方面的支持。

第三章给出一种面向版本一致性的微服务动态更新实现方法。首先针对相关微服务架构对动态更新支持方面的不足与需求进行了探讨，然后基于此讨论，在事务模型、动态依赖管理和服务状态模型三个方面的扩展进行了详细的阐述，完成模型的扩展。

第四章介绍面向版本一致性的微服务动态更新系统实现。首先，给出系统实现的设计目标和准则。然后结合上述的扩展方案，在具体的开源实现框架 Istio 之上对相关的模型进行了模块化开发，分别详细介绍了各个扩展模块的内部结构设计与实现，以及系统运行时的模块交互关系。

第五章案例分析与性能评估。结合具体的应用案例，通过实验来对不同的更新算法在安全性、及时性和干扰性三个方面进行性能的比较评估，此外针对实际应用的需求，添加了同时对多个服务进行更新的相关实验，并对实验结果进行分析。最后对应用进行了展示。

第六章对本文的工作进行总结，并对未来的研究工作进行展望。

第二章　相关工作

本文主要研究针对分布式微服务系统的一致性动态更新，与本文的相关工作包括: 软件动态更新的相关算法，以及相关微服务架构在动态更新方面的支持。下面将具体从这两个方面展开介绍。

2.1　软件动态更新技术

随着网络环境和用户需求的不断变化，运行时的软件系统无法在设计时就考虑到所有的功能，因此软件系统总是需要不断地进行动态更新演化，以满足多元的功能和性能需求[25]。相比较与传统的离线更新方式，动态软件更新(DSU)[26] 要求在软件系统不停止服务的状态下，使用新版本的软件来取代目标软件的旧版本，从而使软件系统达到增加功能或修复错误等目的。在动态更新的过程中，软件系统必须要确保程序始终处于运行状态，保证用户的请求不会被阻塞，而且需要满足一定的约束条件。因此软件系统的动态更新变得十分困难，面临诸多挑战[27]:

- **安全性 (Safety)** 动态更新需要选择正确安全的更新时机，保证更新发生的过程中以及更新完成后，系统中运行的请求都不会出现不一致的情况。
- **及时性 (Timeliness)** 系统从收到动态更新请求到更新结束所耗费的时间，要求动态更新操作应尽快完成。
- **干扰性 (Disruption)** 系统在处理动态更新请求的状态下，相较于没有动态更新请求的状态下，对正常处理业务请求所额外造成的干扰程度，即响应时间的增加。要求动态更新算法所额外造成的干扰性应尽量减小。

动态更新的基础是保证安全性，关键在于如何确定安全的更新时间点。最基本的要求是，当目标更新服务没有处理请求，同时也不存在于任何一个用户请求的分布式调用中，那么此服务处于一个安全的更新时间点。为了更好地对相关算法进行描述，我们在此引入事务的相关概念。

定义 2–1 (事务)　一个事务表示在某一服务上执行且在有限时间内结束的一系

列操作。在某一服务上运行的事务 T 可以向其它服务发起调用，从而在目标服务生成一个新的事务 T'，前后两个事务 T 和 T' 我们分别称为父事务和子事务，记为 $sub(T, T')$。由系统外部调用生成的事务称为根事务。事务 T 运行所在的节点记为 h_T。

定义 2-2 (分布式事务) 根事务及其所有的子事务统称为一个分布式事务，使用对应的根事务符号 T 进行标识。分布式事务 T 包括的所有扩展事务集 $ext(T)$ 表示为：$ext(T) = \{x | x = T \vee sub(T, x)\}$

定义 2-3 (动态更新的安全性) 一次动态更新满足安全性，当且仅当不存在一个分布式事务的某两个子事务在同一个微服务的不同版本上运行，即一个分布式事务只能全由一个微服务的旧版本进行处理或对应的新版本进行处理。

为了解决上述动态更新所面临的挑战，现有工作已有不少的研究。本节主要介绍三种：Krammer 和 Magee 提出了 Quiescence[12] 算法。但 Quiescence 算法对系统造成的干扰极大，Vandewoude 等则在其基础上提出了 Tranquility[22] 算法。针对 Tranquility 只能保证局部一致性而不是全局一致性的问题，马晓星等提出了 Version Consistency[23] 算法，此算法通过维护服务运行时的动态依赖关系，对安全点提出了新的定义，从而较好地满足了动态更新过程中的对于安全性的要求。

2.1.1 软件系统实例

为了更好地阐述说明这些动态更新算法在实际更新时的原理，在具体介绍三种算法之前，本节先引入一个简单但不失一般性的图像拍摄转换系统，静态依赖图如 2-1。该系统由四个服务组成，分别是 Portal、Camera、Viewer 和 Convertor。用户可通过 Portal 发起一次图像拍摄请求，收到请求的 Portal 转而调用 Camera 进行具体的拍摄，拍摄所得结果经 Convertor 转换后返回。然后 Portal 请求 Viewer 服务，Viewer 服务则进一步请求 Convertor，Convertor 服务负责将转换后的数据进行解码并返回。为了加快传输转换速度，Convertor 负责将来自 Camera 的数据编码，相应的，将来自 Viewer 的数据解码。

系统运行的时序图如 2-2。由用户调用在 Portal 服务上首先开始根事务 T_0，T_0 在 Camera 上发起子事务并得到拍摄数据，为了能够快速传输数据，Camera 请求 Convertor 对数据进行转换，即子事务 T_2。Camera 将得到的内部表

图 2-1: 示例系统静态依赖图

示形式的数据返回给 Portal。Portal 则在 Viewer 上发起子事务 T_3，并向其传输数据。Viewer 在 Convertor 上发起新的子事务 T_4，并将数据进一步传输，最终将解码后的数据结果返回，整个根事务 T_0 结束。

图 2-2: 示例系统时序图

在该软件系统实例中，假设希望提高数据的传输转换速度，系统需要对 Convertor 服务进行动态更新。若动态更新操作发生在时刻④和⑥之间，那么动态更新前的事务由旧版本的 Convertor 进行编码，而更新完成之后的事务由新版本的 Convertor 进行解码。若新旧版本的编解码算法不可兼容，便会导致系统出现不一致的情形。因此为满足安全性的需求，动态更新算法需要保证，不存在一个分布式事务的某两个子事务在同一个微服务的不同版本上运行。

2.1.2 Quiescence 算法

对于一个分布式系统，通常由若干个节点组成，它们之间相互请求调用从而形成相应的依赖关系。此处节点为泛化代称，在具体的应用场景中，节点有

具体的表现形态。当我们需要对此分布式系统中的某个节点执行动态更新操作时，包括节点的创建、删除等，便有可能导致系统的静态依赖关系或目标节点的内部状态发生变化，从而进一步地影响运行时事务的正确性。因此，为了保证动态更新过程中系统的安全性，Kramer 和 Magee[12] 提出了 Quiescence 动态更新算法，算法中重点为节点定义了 Active 和 Passive 状态：

- **Active** 处于该状态的节点可以主动地发起事务、接受事务并为其他事务服务。
- **Passive** 处于该状态的节点不能主动地发起新事务，可以继续接受并为其他事务提供服务，但需要保证：(1) 该节点当前不处于自身发起的事务中；(2) 该节点不会发起新的事务。

当某个节点需要执行动态更新操作时，节点的状态可以在 Active 和 Passive 之间进行转换，具体的转换关系如图 2-3。由前述的 Passive 状态定义可知，即使节点处于 Passive 状态下，执行动态更新也不能保证系统的安全性，因为此时节点有可能正在处理由其它节点所发起的事务。因此作者提出了更严格且保证安全性的约束，即 Quiescence 状态：

- 该节点当前没有参与到自身发起的事务之中。
- 该节点不会发起新的事务。
- 该节点当前未为其他事务提供服务。
- 该节点无需为其他节点已经发起或者是将来会发起的事务服务。

图 2-3: Quiescence 节点状态转移图

前述对于 Quiescence 状态的定义中，前两点与 Passive 状态定义一致，要求当前节点处于 Passive 状态。而后两点要求本节点不能为其他节点已经发起或者是将来会发起的事务服务，即表示依赖于本节点的其他节点也应处于 Passive 状态。若某一节点 Q 想要达到 Quiescence 状态，那么对于集合 Passive

Set $PS(Q) = \{Q\} \cup \{$ 直接依赖于 Q 的节点集合 $\}$，只有 $PS(Q)$ 中的所有节点均达到 Passive 状态，节点 Q 才能达到 Quiescence 状态。

对于 $PS(Q)$ 的定义仅仅考虑了一层依赖关系，更为全面的定义为集合 Enlarged Passive Set $EPS(Q)$：

- $PS(Q)$ 中的所有节点均属于 $EPS(Q)$。
- 所有依赖于 $PS(Q)$ 的节点。

综上，若某一节点 Q 想要达到 Quiescence 状态，只有当 $EPS(Q)$ 中所有的节点均达到 Passive 状态时，节点 Q 才能达到 Quiescence 状态。

在图2-1 的示例中，对应的 Convertor 服务需要等到 Portal、Camera、Viewer 服务上运行的事务全部结束后才可达到 Quiescence 状态。可以发现，此时的 Portal、Camera、Viewer 服务均处于 Passive 状态，不能发起事务对外提供服务。因此 Quiescence 算法虽然可以确保动态更新过程中的安全性，但同时给系统带来了极大的干扰[28]。

2.1.3 Tranquility 算法

Quiescence 算法对于安全点的定义过于严格，给运行中的系统带来了极大的干扰。针对此问题，Vandewoude 等在 Quiescence 的基础上提出了 Tranquility 算法[22]，并利用该算法在 DRACO[29] 中间件上实现了一个动态更新系统。Tranquility 算法中继续沿用了 Quiescence 中事务、Active 状态和 Passive 状态的相关定义，重新为节点定义了 Tranquility 状态。节点想要达到 Tranquility 状态需要满足条件：

- 该节点当前没有参与到自身发起的事务中。
- 该节点不会发起新的事务。
- 该节点当前没有处理任何事务。
- 该节点的相邻节点不存在过去使用过此节点并且将来还将使用此节点的事务。

在 Tranquility 的定义中，前两点同样与 Passive 状态定义一致。而后两点定义均弱于 Quiescence 的定义，因此满足 Quiescence 状态的节点必然满足 Tranquility 状态，但反之则未必。不同点在于，Tranquility 算法只要求目标节点处于 Passive 状态，且与其相邻的节点也处于 Passive 状态即可。它不需要考虑运行于其它更多节点的事务运行情况，只要相邻的节点不存在使用过目标节点

且将来还会使用目标节点的事务，目标节点就可以达到 Tranquility 状态。

在图 2-2 中，当 Camera 将处理结果返回给 Portal 后，系统运行到时刻⑤，此时 Camera、Viewer 和 Convertor 均处于 Passive 状态，因此 Convertor 满足 Tranquility 状态的定义，系统判断可进行更新。但如果此时对 Convertor 进行版本更新，便有可能因为新版本的 Convertor 与旧版本存在不可兼容性而出现系统不一致的问题。因此 Tranquility 算法相比较于 Quiescence 算法，虽然可以减少对系统造成的干扰，但是无法保证动态更新的过程中系统的安全性问题。

2.1.4 Version Consistency 算法

已有的一些动态更新算法或依据静态依赖信息[12]，或依据动态依赖信息[22,30,31] 来保证本地一致性。而马晓星等提出了 Version Consistency 算法[23] 则对版本一致性的概念进行定义，用于解决了分布式系统下分布式事务的全局一致性问题。

Version Consistency 算法定义的版本一致性，要求系统中运行的每一个事务都在旧版本节点上运行或者都在新版本节点上运行。以图 2-2 为示例，系统在时刻②之前和时刻⑦之后，对应的 $ext(T_0)$ 中的事务均满足版本一致性的定义，因此都可以进行动态更新。但若系统选择在时刻⑤对 Convertor 进行动态更新，此时 $h_{T_2} = Convertor, h_{T_4} = Convertor'$，且 $T_2, T_4 \in ext(T_0)$，其中 $Convertor$ 和 $Convertor'$ 分别表示更新前后的 Convertor，显然不满足前述版本一致性的定义，此次动态更新将使得系统出现版本不一致的情况。

版本一致性虽然定义了动态更新的约束条件，但该条件在实际应用中需要全局中心化的信息，很难进行操作和检测判断。因此 Version Consistency 算法使用节点运行时的动态依赖关系，很好地解决了更新状态条件难以判断的问题。在 Version Consistency 算法中，动态依赖关系由具体的动态依赖边进行定义表示：其中 $C \xrightarrow[T]{future} C'$ 表示在根事务 T 中，节点 C 将来可能会用到节点 C'；$C \xrightarrow[T]{past} C'$ 表示在根事务 T 中，节点 C 过去曾经使用过节点 C'。

系统在运行时，将依据每一个根事务 T 的运行状态，动态地在节点间执行依赖边的添加和删除操作。基于动态依赖边，Version Consistency 算法提出了 Freeness 更新判断条件，该状态要求，不存在同时指向目标更新节点且被标识为同一个根事务的 $future$ 边和 $past$ 边，表示不存在当前已经调用过目标更新节点且将来还有可能会调用目标更新节点的根事务。

图 2-4 展示了系统中某个根事务运行时的动态依赖关系，其中 $(a), (b), (c), (d)$

表 2-1: 动态更新算法比较

更新算法	安全性	及时性	干扰性
Quiescence	安全	慢	高
Tranquility	不安全	快	低
Version Consistency	安全	快	低

分别对应图 2-2 中的时刻①、时刻③、时刻⑤、时刻⑦。可以发现，Convertor 在时刻①和时刻 178 满足 Freeness，不存在标记为 T_0 的一对 *future* 和 *past* 边指向 Convertor，可进行动态更新；而在时刻③和时刻⑤则同时存在标记为 T_0 的 *future* 和 *past* 边指向 Convertor，因此 Convertor 不可以进行动态更新。

图 2-4: 系统运行时动态依赖关系示例

上面介绍的三种具体软件动态更新算法分别对安全更新点给出了自己的定义，即节点分别需要满足 Quiescence 状态、Tranquility 状态和 Freeness 状态。判断的条件和使用的方式也略有差别，例如 Quiescence 算法主要利用的是事务和节点间的静态依赖信息，而 Version Consistency 在其基础上进一步地使用了节点间运行时刻的动态依赖信息，使得对于安全点的判断更为精准。对于前述的动态更新所面临的诸多挑战，如安全性 (Safety) 、及时性 (Timeliness) 和干扰性 (Disruption) ，三种算法的表现也不尽相同。具体对比如表 2-1 所示。

2.2 微服务架构与动态更新

基于微服务理念进行分布式应用程序的开发，已经成为多数平台的通用指导模式[32,33]。应用微服务架构的关键之一在于对复杂的业务系统进行拆分，使得拆分出来的服务能够相互保持逻辑上的独立，从而保证服务可由不同的团队进行负责，快速地完成开发、部署和更新等操作。拆分步骤必然导致系统中的模块数量变多，实际上是将原先业务系统内部维护的复杂度，转换成了模块与模块之间的通信与管理监控的复杂度。

随着微服务架构的流行，相关动态更新的工作也有了较多的关注，目前已经出现了不少的研究成果。

当前工业界中与微服务动态更新相关的部署技术包括蓝绿部署[13]、金丝雀部署[14,15] 和滚动发布。其中，蓝绿部署不停止老版本的运行，使用冗余的设备部署一套新版本的应用来完成测试等相关工作，完成后将流量直接切换到新版本中。蓝绿部署需要额外的资源部署一套新版本的应用，成本较高。金丝雀部署部署新版本时对用户流量在新旧版本间进行分流，然后依据用户的反馈，逐步扩大新版本的应用范围，最终完成更新。滚动发布更新时取出一个或者多个服务器停止服务执行更新，完成后重新将其投入使用。周而复始，直至集群中的所有实例均更新为最新版本，在金丝雀发布基础上完成优化改进，是一种自动化程度较高的发布方式。后两种更新方式均未对更新过程中事务的运行状态进行考虑，不可避免地会导致系统出现不一致的行为。

在 Spring 框架[34] 中，支持两种热部署 (Hot Deploy) 的方式，热部署主要希望在修改应用代码时，不重启停止应用即可完成代码的更新。此功能类似于动态更新 (动态配置) ，但是在具体的实现上，热部署主要基于 Java 的动态类加载机制，不能很好地定义安全点，无法保证应用更新过程中的正确性和一致性。例如在 Spring 框架中，Spring Load 是一个用于在 JVM 运行时重新加载类文件更改的 JVM 代理，允许用户动态对某个方法字段等进行添加、修改、删除。当使用 Spring Load 启动程序后，系统会自动监视对应的文件，当目标文件发生改动时，系统便会重新加载类文件，无需重启服务。Devtools 的功能类似，原理在于使用两个类加载器：不改变的类包括第三方 jar，由 base 类加载器加载，而目标正在开发的类由 restart 类加载器加载。当检查到类发现变化时，应用进行重启，restart 类加载器直接丢弃重建，而 base 类加载器则继续留用。由于 base 类加载器已经可用且已填充，因此本次应用的重启会比普通的重启速度

要快得多。这两种热部署的实现方式均考虑的是目标文件的相关变化，但并不关心更新过程中运行时事务的相关状态，因此无法保证更新的正确性。

微服务架构具有耦合度低、独立部署扩展等特性，因此它通常在相关的分布式云服务平台上进行部署运行并使用其底层框架的能力[18]。例如 Spinnaker[1], AWS CodeDeploy[2], IBM UrbanCode[3]都支持对部署于异构 Paas 平台上的微服务进行更新部署。更新的执行主要依靠运维人员对底层操作进行编写，并形成一系列的流水线操作，最后以脚本的形式来运行。这种更新方法具有较大的局限性，一方面它要求运维人员撰写出正确且有效的运行脚本，这通常需要耗费较长的调试时间；另一方面，脚本的执行通常不具有幂等性，如果基于脚本的更新过程出现异常的话，往往需要回滚并重启整个更新过程，并小心翼翼地保存相关的状态。push2cloud[4]则运行对更新的目标系统结构进行描述，并支持部署在其上的微服务进行更新。但目前仅支持单站点的微服务部署，更新策略同样需要定义成对应平台的底层流水线操作。

2.3 本章小结

本章主要介绍了软件动态更新的相关算法，以及现有主流的微服务框架、相关微服务架构对动态更新方面的支持。在软件动态更新的相关算法中，本章重点介绍了 Quiescence、Tranquility 和 Version Consistency 三种算法，它们分别在各自的算法中对动态更新的安全点进行定义，以及安全点的相关判断条件。同时还对这三种算法在安全性 (Safety) 、及时性 (Timeliness) 和干扰性 (Disruption) 进行了具体的比较。而在现有工业界主流的微服务动态部署技术中，对蓝绿部署、金丝雀部署和滚动发布进行了相关说明。最后介绍了 Spring 框架对于热部署机制的两种支持方式，以及相关分布式云服务平台对微服务架构在动态更新方面的支持。

[1]Spinnaker. https://www.spinnaker.io/
[2]AWS CodeDeploy. https://aws.amazon.com/codedeploy/
[3]IBM UrbanCode. https://developer.ibm.com/urbancode/
[4]Push2Cloud website. https://push2.cloud/

第三章　面向版本一致性的微服务动态更新方法

微服务动态更新在实际应用场景中有着较大的需求，但当前的微服务架构对更新过程中的安全性考虑不足，缺乏对服务正确且快速更新的支持。本章首先对微服务动态更新进行讨论，分析当前的微服务架构在动态更新支撑方面未解决的关键问题。然后为解决上述问题，给出一种面向版本一致性的微服务动态更新实现方法 VCIM (Version Consistency In Microservices)，针对性地在事务模型、动态依赖管理和服务状态模型三个方面提出了相应的扩展和实现方案。

3.1　问题分析

事务的概念具有多义性，在进行具体问题的分析之前，我们需要进行区分。当前相关的微服务框架所支持的事务主要指传统数据库中的 ACID (Atomicity, Consistency, Isolation, Durability) 事务，如 Spring 框架中的 @Transactional 注解，显然此类事务无法满足对于用户请求状态的相关需求，也无法在更新时提供有效的语义支持。而本文所涉及的事务概念，不同于前述的数据库中的事务概念，指的是系统接受外来请求时将在服务上执行的一系列有意义的动作，依据前述定义 2-1，这些动作称之为事务，一次用户请求对应于一个特定的分布式事务。本文后续所提及的事务，均遵循此定义。

在微服务系统场景下，服务通过多副本来达到负载均衡、容错等目的，因此微服务系统中的动态更新将主要考虑多版本实例共存的服务更新。为方便说明，这里仍以 2.1.1 节中引入的系统为例进行说明。当系统中的 Convertor 服务需要进行更新时，应用 2.2 节中所介绍的相关部署技术，用户首先上线相应的新版本服务，然后进行流量的划分，此时的系统状态如图 3-1 所示。

由前述的讨论可知，版本一致性指的是微服务系统中的事务版本一致性，它与动态更新的安全性定义 2-3 保持一致，要求在一次用户请求所对应的分布式事务中，Camera 服务和 Viewer 服务应调用同一版本的 Convertor 服务，但当

图 3-1: 主流动态更新流程示例

前系统并不能为此提供保证。因此当前主流的微服务架构和相关的动态部署技术无法支持微服务正确地进行动态更新，即无法保证动态更新过程中的版本一致性等关键特征，未解决的问题主要包括：

- 未对请求的相关状态进行支持：动态更新安全性的定义如前述定义 2-3，系统更新前后的一致性和用户请求在系统中所调用的服务版本密切相关。在当前的微服务架构中，用户向特定的服务发起请求，微服务架构随即将请求转发往特定的目标服务中，而选择哪一版本的目标服务实例将由相关的负载均衡策略来决定。请求的无标识性使得调用过程具有无状态性，因此无法保证正确地在不同版本间进行请求的转发，从而有可能导致更新过程中系统的不一致性。
- 未对动态更新的安全点进行考虑：前述的相关动态更新算法对具体的安全更新点进行定义，在对应的时间点可执行相关的更新及撤销操作，并且保证动态更新的安全性。而主流的微服务架构在进行服务的更新时，目标在于服务的快速迭代部署，要求新版本服务尽快完成上线，同时允许多版本服务实例共存并处理外来请求，没有对动态更新的时间点进行严格的定义，无法保证动态更新过程的安全性和及时性等特征。

支持动态更新的微服务架构不仅需要有效地解决上述问题，同时还需尽量保证：

- 动态更新开始前，用户可以对目标更新服务进行指定，且更新前后需保证系统的安全性 (定义 2-3) ，不会出现因更新而导致的不一致现象。
- 能够尽快地完成动态更新，有较好的及时性，且更新过程不会对整个系统，包括其它非更新服务带来较大的干扰，保持良好的用户体验。
- 内部的相关模块具有较低的耦合度，从而保证不会对基于此架构开发的应

用造成较大的影响，可以适配实际不同应用场景下的微服务系统，具有较好的兼容性。

基于上述的讨论和分析，为使当前的微服务系统支撑具体的动态更新算法，我们将为上述问题提出相关的解决方法，并将其整合到相应的微服务架构中。

3.2 方法概述

实际应用场景下，不同的微服务系统将由不同的团队来进行开发，在通信方式、编程语言、运行环境等方面都存在较大的差异，而扩展模块方式 (如流量代理) 的实现，通过将相关的非业务功能抽取出来，与具体的微服务业务逻辑无关，保证了具体的业务逻辑与扩展模块之间的低耦合。因此，我们将基于流量代理的方式，给出一种面向版本一致性的微服务动态更新实现方法 VCIM (Version Consistency In Microservices) 。此方法针对前述微服务架构在动态更新支持方面的问题，进行相关模块的扩展实现和整合，在保证版本一致性的基础上，快速有效地完成微服务的动态更新。

3.2.1 请求状态支持

由于事务的相关处理逻辑与微服务实际的业务逻辑无关，因此 VCIM 为每一个微服务实例插入相应的流量代理，当收到请求时，流量代理负责拦截请求并提取其中的事务上下文信息，若其中不存在相关的分布式事务标识，流量代理将负责生成；否则，沿用指定的分布式事务标识。后续发生远程服务调用时，前述的事务上下文信息将随请求一同发送，被调用方的流量代理将执行相同的逻辑，从而保证了请求在系统中运行时的唯一标识性。

此外 VCIM 为微服务系统定义合适的事务模型，对请求状态进行相关支持。事务状态的定义如图 3-2 所示，具体来说，相关的状态包括：初始化、运行、依赖变更、结束。

- **初始化** 当服务接收到外来请求时，首先被流量代理所拦截，生成相应的事务，事务进入初始化状态。此时的事务还未被真正的服务实例处理，请求将被进一步地转发。
- **运行** 当服务实例开始执行由流量代理转发而来的事务时，事务进入运行状态，表示事务开始执行具体的业务逻辑。

- **依赖变更** 事务在运行时有可能对一个或多个依赖服务发起远程调用，并在其之上发起相应的子事务。当远程调用返回时，服务间的动态依赖关系便可能发生变化，记录当前服务已经使用过该依赖服务，以及当前服务在将来是否还会调用该依赖服务。处于依赖变更状态的事务仍处于运行状态，若后续还存在相关的远程调用，则此事务状态保持不变，否则，等待事务执行完成，将转入结束状态。
- **结束** 当事务执行完成时，将结果经流量代理返回给具体的调用者。此时事务进入结束状态，表示一次事务完整的执行。

图 3-2: 事务状态转移图

分布式事务状态的定义与事务状态的定义保持一致，不同之处在于它由其所有的子事务的状态来共同表示。这些子事务的状态在分布式服务实例的本地进行维护，不存在中心化的存储。当系统在需要相关信息时，每个服务可借助于分布式事务标识信息，向依赖于自己的相关服务发起请求，获得全局的分布式事务信息状态。

虽然 VCIM 从服务启动后就在不断地维护相关的事务信息，但可以看到 VCIM 对事务模型的定义较为精简，每个服务只需知晓和维护其本地事务的详细信息和对应的分布式事务标识，而无需在服务间进行事务上下文的同步工作，尽可能地减少对全局事务信息的维护，从而避免了服务间通信所带来的额外开销，并且达到了对相关模块减负的目的。

3.2.2 安全更新点定义

由于在微服务系统场景下，多版本服务实例可共同对外提供服务，因此微服务动态更新所定义的安全更新点主要针对的是动态更新过程中对旧版本服务实例的撤销时机。

安全更新点的判断标准是旧版本服务实例何时不再需要处理请求。服务间的静态依赖关系可以保守地对系统运行时进行描述，但是无法保证动态更新的及时性。而基于运行时服务间的动态依赖信息，动态更新算法可以有效地对安全更新点进行判断，在保证动态更新一致性的前提下，及时地完成服务的更新。因此，VCIM 通过扩展的动态依赖管理器，将维护服务间的动态依赖信息，为更新过程提供基础保障。这里的动态依赖信息将沿用文献[23] 中提出的动态依赖边概念。

事务运行时状态影响服务间的动态依赖关系，服务间的动态依赖关系也需要具体的事务来进行标识。VCIM 扩展的依赖管理器通过对具体的程序进行分析，得到相对应的事务状态自动机，并利用此事务状态机来进行动态依赖信息的获取[35]。当本地事务运行时，事务的依赖会发生相应的变化，从而反映出服务间依赖关系的变化。该事务状态自动机将分析出在当前状态下，事务曾经调用过的服务集合 PAST_SET，以及将来有可能需要调用的服务集合 FUTURE_SET，依据此信息形成动态依赖关系，并在本地进行管理。

当系统中的目标服务需要进行旧版本的撤销时，收集相关服务的动态依赖信息，同时结合本地维护的局部信息，对安全更新点进行判断：若不存在同时指向目标服务的 *past* 边和 *future* 边时，即集合 PAST_SET 和集合 FUTURE_SET 不存在交集时，可执行旧版本服务实例的撤销操作。下一章中将给出支持动态更新的微服务系统实现，对动态依赖管理器的实现进行详细说明。

3.2.3 服务状态模型

VCIM 通过前述扩展的事务模型和动态依赖管理器，为微服务系统的动态更新提供了基础信息。此外，动态更新过程中版本切换算法和旧版本撤销算法的具体实现均需要存在相应的载体，且应该保证具有较好的抽象性，对服务的业务逻辑保持透明。因此 VCIM 对服务状态模型进行了扩展，一方面将相关的算法在服务状态模型中的相关扩展状态进行具体体现，另一方面很好地满足了动态更新过程中目标服务在不同状态下对于更新的需求。

由于普通的微服务通常具有无状态、即插即用等特性，因此其状态模型较为简单，无法对动态更新中的相关语义进行理解。现有的微服务系统通常仅考虑服务的基本状态，包括初始状态、运行状态、终止状态等。显然单一的运行状态并不能很好地刻画目标服务在整个更新过程中的状态。具体来说，除了相

关的基本状态，运行状态的服务从收到更新请求到最后更新结束，还需包括更多拥有特定意义的状态，如更新开始状态、版本切换状态、结束撤销状态等。因此，VCIM 需要对服务的运行状态进行更细粒度的扩展，同时定义相关的状态跳转关系，保证更新前后的一致性和持续性。

VCIM 允许用户在执行动态更新前指定需要更新的服务，称为目标更新服务，该服务可能对应于多个具体的服务实例。此处服务状态模型的扩展主要是针对服务运行时 (Running) 状态的扩展，这里服务的状态将代表系统对外暴露的服务运行情况，是所有服务实例运行状态的总称，而不是特指某一个服务实例的运行状态。扩展后的服务状态如图 3–3 所示。可以看出，VCIM 定义的服务扩展状态及其之间的跳转关系形成了一次完整的更新闭环流程，可为系统提供连续多次更新的能力。

在前述动态更新的过程中，VCIM 仅关注目标更新服务的状态变化情况。在特定状态下，目标更新服务需要向依赖于自身的相关服务发起请求，以获得动态依赖信息，但并不对相关服务的状态进行限定。这意味着使用 VCIM 方法扩展的服务状态模型具有较好的隔离性，可以将不同服务间的状态解耦开，有效地提高了更新的效率。

下面将结合扩展的服务状态，对实现的版本更新算法和版本撤销算法进行介绍。

3.2.3.1 版本更新

为了保证动态更新过程中的安全性，即满足定义 2–3 中的要求，版本更新算法除了对运行时事务进行标识，还需要对运行中的事务所涉及的服务版本进行标识。因此，我们在头信息中引入自定义字段 x-version 来进行具体的标识。当调用方向服务发起请求并返回时，利用流量代理模块的拦截功能，在其头信息中添加上带有 x-version 的服务版本信息，作为事务上下文信息进行返回。随后调用方在接收处理结果时，从头信息提取出事务上下文信息并更新本地信息，后续发起远程调用时将利用更新后的信息进行传递，从而完成了运行时事务针对服务版本的标识工作。

当服务初始启动就绪，或者某次更新完成后，服务将进入 Normal 状态。处于 Normal 状态下的服务和未扩展前正常运行状态下的服务一致，系统仅存在 (使用) 一个版本的服务实例来处理外来请求。如图 3–4 所示，表示 Convertor 服务在初始版本为 v1 时，系统的正常运行状态。所有向 Convertor 服务发起

图 3-3: 服务状态模型

调用的请求，针对于目标更新服务 Convertor，都将被标识为 v1 版本。这里将 Convertor 服务作为目标更新服务，因此仅关注与 Convertor 服务相关的字段信息，其它非目标更新服务可类推。

基于运行时事务针对服务版本的正确标识，我们可进一步实现服务版本的正确更新。当用户想要对某个服务进行动态更新时，首先需要上线相应的新版本服务实例并等待其全部就绪，此时服务进入 Deployed 状态。此时系统中对应的服务包括新旧多个版本实例，但请求仅由旧版本的服务实例来进行处理。若此时未能确保所有新版本服务实例就绪，则后续随着服务状态的转变，请求有

图 3-4: 初始状态下的系统

可能被发往未就绪的实例，进而导致异常，返回错误结果。

想要对多版本共存的微服务系统进行版本的切换，VCIM 添加基于头信息的相关路由规则，进行更新的准备工作，服务进入 Prepared 状态。假定对应的新旧版本标识分别为 v2、v1，对应系统的运行状态如图 3-5 所示，虽然系统中的 v2 版本服务实例已经就绪，但请求全部由 v1 版本的服务实例进行处理。

图 3-5: 更新准备前的系统

当前述动态更新的准备工作完成后，VCIM 进行流量的切换，将原先路由规则中默认转发往旧版本修改为默认转发往新版本，服务进入 Shifted 状态。当存在外来请求向目标更新服务发起调用时，若该请求对应的分布式事务曾经在目标更新服务上发起过子事务，那么请求将被转发往旧版本的服务实例；否则，请求将被转发往新版本的服务实例。显然，处于此状态中的服务对于用户请求的处理满足定义 2-3 中对于动态更新的安全性要求，对应系统的运行状态如图 3-6 所示。

至此，版本更新算法很好地完成了目标更新服务新旧版本的切换，同时保

图 3-6: 版本更新中的系统

证了动态更新过程中系统的一致性。

3.2.3.2 版本撤销

在版本更新算法的基础上，为了避免旧版本服务实例对于系统资源持续占有所造成的浪费，用户向目标更新服务发起撤销旧版本的请求，服务进入 Freeing 状态。

在正常状态下，当服务收到请求生成事务时，动态依赖管理器将负责对本地的动态依赖信息进行更新，保存已在本地完成调用的分布式事务标识集合 PAST_SET，以及将来有可能向其它服务发起调用的分布式事务标识集合 FUTURE_SET。如图 3-7 显示了某一运行时刻，系统中各个服务针对 Convertor 服务的本地动态依赖信息。

图 3-7: 维护动态依赖信息

版本撤销算法的关键在于判断：

- 旧版本服务实例不会再为未结束的事务提供服务。
- 旧版本服务实例当前未执行任何事务。

应用文献[23]中所提出的 Freeness 更新判断条件，利用前述的事务依赖管理器维护的 PAST_SET 和 FUTURE_SET 集合，实现了对应的版本撤销算法，具体包括如下步骤：

1. **阻塞访问**：当收到旧版本撤销请求时，系统首先阻塞所有向旧版本服务实例发起调用的新生成的分布式事务。而对于曾经在旧版本服务实例上发起过的子事务的分布式事务，则允许通过。被阻塞的请求将在调用方执行重试，依据前述版本更新算法中路由规则的定义，本次请求将会被转发往新版本的服务实例中，保证了用户请求依然能够被正常地处理，而且不会再有新的分布式事务标识添加到相关的事务依赖管理器所维护的 PAST_SET 集合中，对应的系统状态如图 3-8 所示。

图 3-8: 阻塞访问

2. **同步状态**：由前述可知，分布式事务将来是否会调用目标更新服务的状态信息，保存于系统内的各个服务实例中。目标更新服务可利用本地维护的 PAST_SET 集合，向依赖于自身的相关服务发起请求，同步相关服务的事务状态信息，便可得到相应的 FUTURE_SET 集合。此集合代表所有将来会向目标更新服务发起请求的分布式事务标识集合，同步过程如图 3-9 所示，其中的红色虚线段表示具体请求过程。
3. **等待撤销**：基于前一步骤所得到的 FUTURE_SET 集合，并结合实例本地事务依赖管理器所维护的相关信息，撤销算法如图 3-10 所示，可执行进一步

图 3-9: 同步状态

的判断：若接收到的 FUTURE_SET 集合为空，表示针对当前所有调用过旧版本服务实例的分布式事务，均不会再向目标更新服务发起请求，因此只需等待当前旧版本实例上的事务执行结束，即可执行撤销操作，算法执行结束；否则，即表示当前系统中的分布式事务在将来仍会使用到旧版本的服务实例，此时撤销算法流程将进入等待状态，待所有 FUTURE_SET 集合中的分布式事务均完成调用后，对 PAST_SET 集合进行更新，并重新执行前一步骤，以重新同步相关的分布式事务状态。

图 3-10: 等待撤销

处于 Freeing 状态的服务在执行完前述的版本撤销算法后，便可执行旧版本实例的撤销操作，服务进入到 Updated 状态。在此状态下，VCIM 将对相关

服务的流量代理模块进行路由规则的重配置和相关事务的标记清理工作。目标更新服务完成更新，回到 Normal 状态，系统状态如图 3-11 所示，完成一次完整的动态更新闭环流程。

图 3-11: 更新完成后系统状态

版本撤销算法利用本地维护的相关事务依赖信息，通过相关的同步请求来收集到具体的分布式事务状态信息，正确判断服务能否被撤销，保证撤销操作的及时性，而且不会阻塞正常的业务请求，对系统的干扰性较小。

3.3 本章小结

本章主要对面向版本一致性的微服务动态更新方法 VCIM 进行了介绍。首先，对当前微服务架构支持动态更新所面临的关键问题进行了探讨，对其不足进行了具体的分析。然后针对相应的问题，给出对应的实现方法 VCIM，有效地在事务模型、动态依赖管理和服务状态模型三个方面提出了具体的扩展方案，有效地为微服务系统提供动态更新。下一章将结合具体的框架技术，对 VCIM 的系统实现进行详细的阐述和说明。

第四章　支持动态更新的微服务系统实现

在上一章中，VCIM 具体从事务模型、动态依赖管理和服务状态模型三个方面，对支持动态更新的微服务架构进行了扩展。本章将根据第三章中提出的 VCIM 方法，介绍如何在开源框架 Istio 之上进行具体的设计，实现支持动态更新的微服务系统。

4.1　系统设计概述

虽然当前主流的微服务框架本身对用户屏蔽了绝大多数通用功能的实现细节，但开发人员同样需要耗费大量时间与精力去学习掌握框架本身。同时，开发框架通常局限于一种或几种特定的编程语言，业务服务往往得随着框架服务的升级而升级，对支持动态更新的微服务系统实现有较大的局限性。

Mogan 于 2017 年在其文章中提出 Service Mesh 的概念[36]。Service Mesh 考虑使用边车模式 (Sidecar) ，为每一个服务实例部署一个对应的 Sidecar 实例，开发人员只需关注核心的业务逻辑代码，而由 Sidecar 实例来负责实现服务发现、负载均衡、认证授权等通用性功能[37,38]。Service Mesh 的架构图如图 4–1 所示。

使用 Service Mesh 作为微服务的底层实现架构的优势主要体现在：

- Service Mesh 使用流量代理的方式来对具体的微服务实例进行管理，因此相关的扩展模块可以使用相同的方式来进行实现，在保证透明性的同时，对开发人员的业务代码具有较小的侵入性。
- Service Mesh 允许开发人员选择自身偏好的编码语言，只需保证微服务间的网络可达性即可，可适配于实际应用场景下不同类型的微服务系统。
- Service Mesh 架构可保证微服务与动态更新的管理模块解耦开，两者都可独立地完成升级。

图 4-1: Service Mesh 架构

当前主流的 Service Mesh 实现方案是 Google、IBM 及 Lyft 合作开发的 Istio 开源框架[39,40]。Service Mesh 架构为每一个微服务实例均部署特定的网络代理，整个系统引入了大量独立运行的单机代理服务。为了更好地保障系统的整体稳定性，Istio 使用集中式的控制面板，提供了统一的管理运维入口，具体架构图如图 4-2 所示，其中：

- **控制平面 (Control Plane)** 负责监听底层平台数据源，作为配置中心，将实时的配置信息动态地发往数据平面，控制整个系统的通信等各方面行为。
- **数据平面 (Data Plane)** 自动注入的所有服务实例的流量代理总称，负责连接到控制平面并接收相关配置信息，然后对请求按配置进行处理，是配置规则的实际执行者。

因此对 VCIM 的实现将基于具体的开源实现框架 Istio，考虑使用与流量代理类似的方式，对事务和依赖管理进行扩展。同时，为了使用户可以方便地对系统执行动态更新等操作，在具体的应用系统之外，定义了额外的服务状态管理器和更新控制模块，为应用系统提供相应的管理功能，实现了模块间的解耦合。

4.2 系统扩展实现

本节将基于上一章提出的扩展方案，对相关的扩展模块进行实现，并且为用户提供相关的动态更新管理接口。下面将详细阐述事务依赖管理器模块 *TxDepManager*、服务状态管理器模块 *SvcStateManager*，以及更新控制模块 *UpdateController* 的内部结构设计。

图 4-2: Istio 架构图

4.2.1 事务依赖管理器

运行时事务信息和服务间的动态依赖信息是更新算法判断服务能否被更新的基础依据。动态依赖信息需要同特定的事务相绑定才有具体的表征意义，因此系统实现时定义了事务依赖管理器模块，统一负责对运行时事务和动态依赖进行管理。

为实现 VCIM 中对于事务与动态依赖管理的扩展模型，我们选择在数据平面对相关模型提供支持。实现时将扩展模型打包成相关的镜像服务，在相关的文件中添加相应的配置，如图 4-3 所示，利用 Istio 框架提供的自动注入功能，在对微服务实例进行部署时，除了插入流量代理模块，还添加了对应的事务依赖管理器。

扩展后的某个具体的微服务实例如图 4-4 所示，其中包括了：

- **流量代理** 拦截并接管所有的外来请求，生成相应的本地事务并添加或继承特定的分布式事务标识。流量代理模块一方面负责将请求分发至相应的业务逻辑模块，另一方面负责对前述的事务上下文信息进行传递。
- **事务依赖管理器** 负责事务与依赖的管理，保证在事务依赖发生变更时，

```yaml
containers:
- command:
  - /usr/local/bin/meshTraceManager
  image: registry.cn-hangzhou.aliyuncs.com/wdongyu/trace-manager:sa-debugv1.1
  imagePullPolicy: Always
  name: tx-dep-manager
  ports:
  - containerPort: 8091
    protocol: TCP
  resources:
    limits:
      cpu: "1"
      memory: 1Gi
    requests:
      cpu: 100m
      memory: 128Mi
```

图 4-3: 注入模块配置

对相应的服务集合 PAST_SET 和 FUTURE_SET 等信息进行更新保存，为 VCIM 提供运行时事务信息和动态依赖信息。

- **业务逻辑** 由服务的开发人员完成编写，从流量代理模块接收请求，执行真正的业务逻辑并返回结果，再由流量代理模块将结果进一步地返回给调用者。

图 4-4: 微服务实例模型

其中，流量代理模块和事务依赖管理器通过 UDS[1] (Unix Domain Socket) 进行通信，UDS 通过指定同一主机上的 socket 文件，不需要经过网络协议栈即可完成数据的传输，效率较高。片段 4.1 展示了为流量代理模块添加的 UDS 客户端代码片段，片段 4.2 则展示了事务依赖管理器作为 UDS 服务端的代码片段，相关的数据总是以单向的形式发送，因此不会对流量代理模块中的请求处理分发流程造成阻塞，保证了对系统具有较低的干扰性。与此同时，流量代理模块对业务逻辑保持透明，为其屏蔽了调用方的相关细节，开发人员只需关注于业务代码，可以很好地与其它模块解耦开。

```
int sock_ = socket (AF_UNIX, SOCK_STREAM, 0) ;
if  (sock_ < 0)  {
    perror ("opening stream socket") ;
    exit (1) ;
}
struct sockaddr_un server_;
server_.sun_family = AF_UNIX;
std::strcpy (server_.sun_path, socket_path) ;
int len = std::strlenn (server_.sun_path)  + sizeof (server_.sun_family) ;
if  (connect (sock_, reinterpret_cast<struct socketaddr *>* (&server_) , len
    )  < 0)  {
    close (sock_) ;
    perror ("connecting stream socket") ;
}
std::string message = "ADD " + traceid + " " + spanid;
if  (write (sock_, message.c_str () , std::strlen (message.c_str () ) )  <
    0)  {
    perror ("writinng on stream socket") ;
}
close (sock_) ;
```

代码 4.1: UDS 客户端

4.2.2 服务状态管理器

想要对运行于 Istio 框架之上的服务进行动态更新，需要对原先的服务状态模型进行了扩展，定义了对应的服务状态管理器，提供监听管理服务状态的基本功能。

[1]https://en.wikipedia.org/wiki/Unix_domain_socket

```
listen, err := net.Listen ("unix", SockAddr)
if err != nil {
    log.Fatal ("listen error : ", err)
}
if err = os.Chmod (SockAddr, 0777) ; err != nil {
    log.Fatal (err)
}
defer listen.Close ()

for {
    con, err := listen.Accept ()
    if err != nil {
        log.Fatal ("accept error : ", err)
    }

    go handleServer (con)
}
```

代码 4.2: UDS 服务端

当前 Istio 框架的运行环境与容器编排系统 Kubernetess[1]相耦合，服务以容器的形式运行在其之上。因此，服务状态管理器的实现主要利用 Kubernetes 中提供的自定义资源 CRD (Custom Resource Definition) 功能，其实现原理称为控制器模式，如图 4-5 所示。用户通过声明式命令对期望状态进行描述，控制器对比期望状态与系统实际的运行状态，如果有出入则通过自定义的调谐逻辑 (Reconcile) 驱使实际状态调整为期望状态。

图 4-5: 控制器模式

通过对系统中服务的概念进行抽象，定义出与服务状态相匹配的自定义资

[1]Kubernetes home page: https://kubernetes.io/

源，称之为托管服务 MS (Managed Service) 。托管服务是一个抽象资源，其结构如 4.3 所示，每个托管服务包含了服务名称、服务的路由版本和当前状态等信息，负责对系统中的某一服务进行管理，对应服务在系统中可能存在多个版本的运行实例。当用户主动发起对托管服务的状态变更请求，如切换新版本、撤销旧版本等请求时，托管服务的控制器将监听到相应的变化，然后将变化反映到系统具体的资源中，驱使服务向目标状态进行跳转；若出现跳转过程出现异常，调谐逻辑将重复执行，直至服务的状态跳转成功。

```
// ManagedServiceSpec defines the desired state of ManagedService
type ManagedServiceSpec struct {
    // Name of project
    ProjectName string `json:"projectName"`

    // Name of service
    ServiceName string `json:"serviceName"`

    // Destination cluster subset for the traffic to route
    RouteSubset string `json:"routeSubset"`

    // Ports to be exposed
    ExposePorts []int32 `json:"exposePorts,omitempty"`

    // Ready to revoke obsolete subset
    Phase UpdatePhase `json:"updatePhase,omitempty"`

    // Whether the service is served as gateway entrance service
    Gateway GatewaySpec `json:"gateway"`
}
```

代码 4.3: 托管服务结构

用户通过声明式命令对托管服务进行描述，托管服务的控制器对相关的资源进行更新，包括路由规则、版本信息等，而更新后的资源将由控制面板进一步分发至对应服务的流量代理中，从而达到管理服务状态的目的。托管服务资源的定义很好地与服务状态的概念完成了匹配，对外屏蔽了内部服务的细节信息，抽象出对应的服务运行状态，对内又可以很好地对各个服务实例的状态进行管理。

由前一章节对服务状态模型的讨论可知，用户对目标更新服务的动态更新操作与服务当前所处的状态存在着密不可分的关系。因此，我们将动态更新过

程中的相关更新与撤销算法整合到托管服务的调谐逻辑之中，在完成动态更新中关键的版本更新和撤销操作的同时，驱使服务状态进行相应的跳转。

4.2.2.1 版本更新实现

由前述可知，当作为目标更新服务的 Convertor 服务处于 Normal 状态下时，所有向 Convertor 服务发起调用的请求，针对于目标更新服务 Convertor 都被标识为 v1 版本。相关流量被全部转发往 v1 版本，同时在其返回头信息中添加 x-version: convertor-v1 字段，规则示例如图 4-6 所示。

```
- route:
  - destination:
      host: convertor
      subset: v1
    headers:
      response:
        add:
          x-version: convertor-v1
```

图 4-6: Normal 状态下的路由规则

用户对新版本进行上线部署工作，服务进入 Deployed 状态。此时，对流量代理模块中的路由规则进行修改，如图 4-7 添加基于头信息的匹配规则，规则将从上之下逐一匹配：若本次请求头信息中的 x-version 字段包含特定的版本信息，则将该请求转发往特定版本，对应于规则条目①与②；否则，默认转发往 v1 版本，同时在返回头信息中添加对应字段，对应规则条目③。

基于前述的准备工作，只需将路由规则中的第③条规则对应修改，将请求默认发往 v2 版本即可完成版本更新工作，服务进入 Shifted 状态，修改后的规则如图 4-8 所示。

考虑对于任意一个分布式事务 T，在规则生效之后，向 Convertor 服务发起请求，将包括如下几种情况：

①若分布式事务 T 在运行的过程中还未在 Convertor 服务上发起过子事务，则其请求头信息中将不包含与 Convertor 服务相关的 x-version 字段。依据生效后的路由规则，此分布式事务 T 对于 Convertor 服务的请求，将被转发往新版本；

②若分布式事务 T 在运行的过程中曾经在 Convertor 服务上发起过子事务，则由前述规则可知，其请求头信息中必然包含了与 Convertor 服务相关的

```
- match:
  - headers:
      x-version:
        exact: convertor-v1
  route:
  - destination:
      host: convertor
      subset: v1
- match:
  - headers:
      x-version:
        exact: convertor-v2
  route:
  - destination:
      host: convertor
      subset: v2
- route:
  - destination:
      host: convertor
      subset: v1
    headers:
      response:
        add:
          x-version: convertor-v1
```

图 4-7: Prepared 状态下的路由规则

```
- match:
  - headers:
      x-version:
        exact: convertor-v1
  route:
  - destination:
      host: convertor
      subset: v1
- match:
  - headers:
      x-version:
        exact: convertor-v2
  route:
  - destination:
      host: convertor
      subset: v2
- route:
  - destination:
      host: convertor
      subset: v2
    headers:
      response:
        add:
          x-version: convertor-v2
```

图 4-8: Shifted 状态下的路由规则

x-version 字段。依据生效后的路由规则，此分布式事务 T 对于 Convertor 服务的请求，将被转发往与 x-version 字段具有相同标识的版本实例中；

综上，保证了任一分布式事务 T 在运行过程中，不会在目标更新服务 Convertor 的不同版本实例上发起子事务，既满足了版本更新过程对于安全性的要求，同时也使得新生成的分布式事务可以较快地使用到新版本的服务。

4.2.2.2 版本撤销实现

依据版本更新算法的最新路由规则，新生成的分布式事务将被转发往新版本的服务实例，而已经调用过 Convertor 服务的事务会在有限时间内完成，因此旧版本服务实例会在有限时间内达到安全更新点，即不再需要执行事务，因此可被正确地执行撤销操作。具体实现包括：

1. **阻塞访问**：当收到旧版本撤销请求时，为旧版本服务实例的流量代理模块添加相应的过滤器，过滤器使用特定的 Lua 语言进行编写，相应的代码片段展示于 4.4 中。此过滤器将放行曾经在旧版本服务实例上发起过的子事务的请求，其头信息中的 x-version 字段存在特定标识信息；其余向旧版本服务实例发起调用的请求将直接返回错误。调用方将依据返回的错误码执行重

试操作，依据最新的路由规则，请求便会被转发往新版本的服务实例中。

```
function envoy_on_request (request_handle)
  local found = false
  local headers = request_handle:headers ()
  for key, value in pairs (headers)  do
    if  (key == "x-version" and value == "convertor-v1")  then
      found = true
      break
    end
  end
  if not (found)  then
    request_handle:respond ({[":status"] = "503"}, nil)
  end
end
```

代码 4.4: Lua 过滤器

2. **同步状态**：为同步相关服务的事务状态信息，目标更新服务将以协程的方式，同时向依赖于自身的相关服务实例发起请求。当返回结果不为空时，利用互斥锁更新相应的 FUTURE_SET 集合，确保多个请求同时操作同一集合的正确性，发送请求与锁机制的实现代码片段如 4.5 所示。此集合代表所有将来会向目标更新服务发起请求的分布式事务标识集合。
3. **等待撤销**：利用前一步骤所得到的 FUTURE_SET 集合，执行相关的判断：若接收到的 FUTURE_SET 集合为空，释放占用的锁资源，等待本地事务全部结束的唤醒信号，退出循环，算法结束，如表 4.6 ；若非空，则计算此集合与请求此集合的过程中已发起的事务标识集合之差，为空则继续循环，执行前一步骤，不然等待差集中的事务向本地发起调用，如表 4.7 。

4.2.3 更新控制模块

前述的相关模块为系统提供了支持动态更新的核心能力，为使系统具有更好的操作性，本节定义了更新控制模块。

更新控制模块作为系统与用户的交互工具，屏蔽系统内部服务的详细运行信息的同时，对外暴露相关的管理接口。用户可通过接口来进行服务状态的查询，以及发起相关的动态更新请求，系统可自动完成动态更新过程，无需额外的人工操作。片段 4.8 展示了更新控制模块对于托管服务的部分管理接口。

```
// Send request
client := &http.Client{}
resp, _ := client.Do (req)
defer resp.Body.Close ()

futureList := []string{}
if err = json.NewDecoder (resp.Body) .Decode (&futrueList) ; err != nil {
  log.Printf ("Fail to decode response body : %v\n", err)
  errChan<-1
  wg.Done ()
  return
}
log.Printf ("Receive future list from %s : %v\n", remoteUrl, futureList)
if len (futureList)  != 0 {
  // Lock
  traceIdSet.RWLock.Lock ()
  defer traceIdSet.RWLock.Unlock ()

  _ = traceIdSet.ReceiveIncomingSet (futureList)
}
wg.Done ()
```

代码 4.5: 同步状态实现

```
// Lock
traceIdSet.RWLock.Lock ()
if len (traceIdSet.Incoming)  == 0 {
  // Judge if there exists local transactions
  if len (traceIdSet.SpanidSet)  != 0 {
    traceIdSet.DeleteFuture = true
    traceIdSet.RWLock.Unlock ()

    // Wait local transactions to be end
    <-traceIdSet.EmptyFuture
  } else {
    // Unlock and break the loop
    traceIdSet.DeleteFuture = true
    traceIdSet.RWLock.Unlock ()
  }
  break
}
```

代码 4.6: 接收集合为空代码片段

```
// Lock
traceIdSet.RWLock.Lock ()
ids := []string{}
for id, _ := range traceIdSet.Incoming {
  ids = append (ids, id)
}
// Compute diff
for _, id := range ids {
  if _, ok := traceIdSet.FutureList[id]; !ok {
    _ = traceIdSet.AddOneFuture (id)
  } else {
    _ = traceIdSet.RemoveOneFuture (id)
  }
}
log.Printf ("Current future list : %v\n", traceIdSet.FutureList)
if len (traceIdSet.FutureList)  != 0 {
  traceIdSet.DeleteFuture = true
  traceIdSet.RWLock.Unlock ()
} else {
  // Diff is empty and continue next loop
  traceIdSet.RWLock.Unlock ()
  traceIdSet.EmptyFuture<-1
}
```

代码 4.7: 等待撤销代码片段

```
func (h *APIHandler) ListManagedServices(w ResponseWriter, r *Request) {}

func (h *APIHandler) CreateManagedService(w ResponseWriter, r *Request) {}

func (h *APIHandler) DeleteManagedService(w ResponseWriter, r *Request) {}

func (h *APIHandler) UpdateManagedService(w ResponseWriter, r *Request) {}
```

代码 4.8: 部分接口示例

4.3 系统模块交互

上一节具体对扩展模块的内部结构及每个模块的功能进行了介绍，系统的整体架构图如4-9 所示。本节将重点描述系统运行时各个模块之间的交互关系。

图4-10 展示了版本更新过程的时序图。用户在完成更新前的相关准备工

图 4-9: 系统整体架构图

作之后，向更新控制模块发起更新请求。更新控制模块向运行平台 Kubernetes 中的 *API Server* 发送相关资源的修改请求，而服务状态管理器将监听到相应的资源变化，进入相应的调谐逻辑。调谐逻辑中通过对路由规则进行更新，利用 Istio 框架前述的配置分发功能，对所有相关服务实例的流量代理模块进行行为配置，以驱使服务的状态完成跳转。

图 4-11 则展示了版本撤销过程的时序图。同样由用户向更新控制模块发起撤销请求，再由服务状态管理器进行相应的处理。此时服务状态管理器将进一步与目标更新服务的旧版本实例中的事务依赖管理模块进行交互，执行前述的版本撤销算法，满足条件后完成相应的版本撤销操作。

4.4 本章小结

本章基于前一章节所提出的 VCIM 方法，实现支持动态更新的微服务系统。首先，我们对系统的设计实现进行了概述。然后本章对三个扩展模块的实现及其模块内部结构进行了详细的介绍，其中包括事务依赖管理器如何对事务进行拦截和管理，为系统支持动态更新提供基础信息；服务状态管理器如何对

图 4-10: 版本更新时序图

图 4-11: 版本撤销时序图

服务的状态进行监听管理，版本更新和版本撤销算法如何完成整合，保证动态更新过程的安全性和及时性；更新控制模块如何完成用户与系统间的解耦，为用户提供相关的更新管理功能。最后，重点介绍了系统运行时各个模块间的交互关系。

第五章　实验评估与应用展示

为了验证前述的微服务动态更新系统实现方案是否满足动态更新过程的相关准则，本章将利用一个实际场景下的分布式微服务系统进行实验。在本章的实验中，VCIM 算法将对应于本文所提出的 VCIM 方法，作为一种特定的动态更新算法，与其它的动态更新算法一起作为具体的实验对比算法。同时在特定的实验中对多个服务进行动态更新，具体在安全性、及时性和干扰性三个方面进行不同算法性能的评估和比较。最后，利用台风演示系统的前端界面对应用的动态更新过程进行了具体的展示。

5.1　应用介绍

这里使用的分布式微服务系统，是一个台风演示系统。该系统模拟 2018 年的第 18 号台风 Rumbia，将台风从生成发展到最后结束的运行轨迹进行展示，同时系统通过对雨情和风情的相关服务进行查询，整合了台风及其途径城市的详细信息，为用户提供展示功能。具体来说，用户可通过该演示系统，直观地看到当前台风的运行轨迹。当台风的影响范围波及某个具体的城市时，系统将对所在城市的相关雨情和风情服务进行查询，并进行展示预警。该系统的静态依赖关系如图 5-1 所示：

台风演示系统中，部分主要服务提供的功能如下：

- **Backend** 服务负责整合包括台风强度、位置等详细信息，以及途径影响城市的具体信息。
- **RainController** 服务负责整合所有台风波及城市具体的雨情信息。
- **WindController** 服务负责整合所有台风波及城市具体的风情信息。
- **Typhoon** 服务负责提供台风当前的详细信息，包括经纬度位置、强度和速度等。
- **xx-Rain (Wind)** 服务负责具体 xx 城市的雨情 (风情) 信息。

服务间的交互时序图如图 5-2 所示 (已省略去部分非关键服务)：

图 5-1: 台风系统静态依赖关系图

图 5-2: 台风系统时序图

5.2 实验环境与设计

5.2.1 实验环境

我们将使用上述的台风演示系统来对本文实现的动态更新技术进行评估，为了更好地模拟真实场景下的分布式微服务系统，本次实验将台风演示系统中的 16 个服务部署于 Kubernetes 系统之上，由其进行容器运行时的编排，具体的环境配置为：

物理节点机器硬件配置：24 核 2.5GHz CPU，64GB 内存，1T 硬盘，千兆以太网卡。

软件环境配置：Ubuntu 20.04.1 LTS，Kubernetes v1.17.14。

5.2.2 实验设计

在台风演示系统的实际运行中，多个服务均有相应的更新需求：

1. Typhoon 服务可能需要通过更新来更新数据或修复错误。
2. RainController、WindController 服务可能需要通过更新来整合更多的城市服务信息。
3. xx-Rain (Wind) 服务可能需要通过更新来引入新的信息提供方式。

对于上述需求，系统开发人员希望能够在完成代码的改动之后，不停止系统运行的前提下，完成目标服务的动态更新。

服务间的静态依赖关系如 5-1 所示，其中 Typhoon 服务在实验案例中处于整个调用层次的第三层，可在其之上发起子事务的服务包括 RainController 和 WindController，整个分布式事务的根事务在 Backend 服务上发起，因此四个服务的调用关系近似形成一个“菱形”的关系。对于 Quiescence 算法来说，当 Typhoon 服务需要进行动态更新时，需要保证 EPS (Typhoon) 集合中的所有服务均属于 Passive 状态，即 Backend、RainController、WindController 都必须达到 Passive 状态，显然这将对系统造成很大的干扰，并且更新所耗费的时间也会很长，对应于动态更新的干扰性和及时性。而对于 VCIM 算法来说，只需依据 Typhoon 服务上事务的相关信息以及同步相关服务上事务的运行状态，即可完成动态更新。因此选择 Typhoon 服务进行动态更新可以有效地评估不同更新算法的及时性和干扰性。另一方面，对于常见的微服务部署方案来说，其考虑新版本服务的快速部署更新，对应的算法为直接切换版本算法 (Direct) 。

若 Typhoon 服务需要进行动态更新，系统会在新版本服务就绪后，直接进行版本切换更新，显然，若更新的时机发生在 RainController 发生之后，且在开始调用 WindController 之前，那么本次更新将导致系统出现不一致的情况，而 Quiescence 和 VCIM 则不会出现不一致的情况，对 Typhoon 服务进行更新对验证各个算法的安全性具有实际意义。因此在进行单个服务更新实验中，将选择 Typhoon 服务作为目标更新服务。

在具体的应用场景下，系统内不同的微服务可能由独立的团队来开发管理，因此多个服务可能同时存在更新需求。为了验证本文实现方法的有效性以及服务状态模型的独立性，将同时对系统中多个服务进行更新，多服务更新实验中选择 WindController 服务和 Typhoon 服务作为目标更新服务。

在具体的实验设置方面，将 < 算法，请求间隔，实验目标 > 称为一组实验，其中算法包括 Quiescence、VCIM、VCIM-MUL 和相应的基准算法，其中 Quiescence 算法与具体更新的服务数量无关，而 VCIM 和 VCIM-MUL 则分别表示应用 VCIM 对单个服务和多个服务进行更新；请求间隔包括 1500ms、750ms、500ms、300ms、150ms、100ms，请求间隔表示系统收到请求的时间间隔，用来表征系统的负载状况，请求间隔越小说明系统负载越大；实验目标则包括了安全性、及时性和干扰性三个方面。

针对每一组实验，将分别独立地进行 10 次。每次实验将依据对应的请求间隔，不断地向系统发起请求，一共发送 80 次，同时在第 20 个请求处，向系统中的目标更新服务发起动态更新请求。在单服务更新实验中，仅在第 20 个请求处，额外向系统发起一次动态更新请求；而在多服务更新实验中，在第 20 个请求处，针对不同的目标更新服务，额外向系统发起两次动态更新请求。此时可以保证系统中已经存在多个运行时分布式事务，可以更好地模拟真实状态下的分布式微服务系统，确保实验的合理性和公平性。

5.3 实验分析

在实际的应用系统中，当运维人员尝试对其中的服务进行动态更新时，动态更新的前后系统的运行时事务是否安全、更新所需要耗费的时间以及整个更新过程对系统会造成多大的干扰，这些都是其评价动态更新算法的重要标准。因此本章的实验将从安全性 (Safety) 、及时性 (Timeliness) 和干扰性 (Disruption) 三个方面，进行具体的评测。

表 5-1: 安全性实验结果

请求间隔 (ms)	1500	750	500	300	150	100
Direct	5	8	12	16	33	50
Quiescence	0	0	0	0	0	0
VCIM	0	0	0	0	0	0
VCIM-MUL	0	0	0	0	0	0

5.3.1 安全性 (Safety)

安全性表示在服务进行动态更新的过程前后，系统处理正在运行时的事务和后续将要发生的事务都应满足一致性，即逻辑正确性。不同的动态更新算法对于安全更新点的定义各不相同，因此对于安全性的满足也并不一致。

安全性实验针对前述的三种动态更新算法，分别按照对应的实验设置来进行。例如实验 <Direct，1500ms，安全性 > 表示使用直接切换版本算法进行安全性的实验，模拟的用户请求间隔为 1500ms。实验中记录的结果为动态更新过程中出现不一致的用户请求数量，用来表示该算法是否满足安全性。实验结果如表 5-1 所示。

从表中数据可以看出，直接切换版本算法并未对更新的安全点进行考虑，不能保证动态更新的安全性，且不一致的数量随系统负载的增加而增加。如在请求间隔为 500ms 时，系统中共出现 12 次不一致的用户请求数量。而 Quiescence 算法和本文中实现的 VCIM、VCIM-MUL 算法在实验中都没有出现不一致的请求，可以保证动态更新的安全性，实验结果和算法的相关理论分析相吻合。

由于直接切换版本算法无法保证动态更新的安全性，因此在后续及时性和干扰性的实验中，我们将仅对 Quiescence 和实现的 VCIM、VCIM-MUL 进行实验。

5.3.2 及时性 (Timeliness)

及时性表示服务从收到动态更新请求到真正完成动态更新所需要的时间。及时性用于反映动态更新算法的效率，及时性越高对应的服务到达安全更新点所耗费的时间越短，即可以更快地完成动态更新。

在单服务更新的及时性实验中，对于每一组实验都分别记录下服务收到动

态更新请求的时间，记为开始时间 $time_0$，以及对应完成动态更新的时间，记为结束时间 $time_1$，则该组及时性实验的计算结果为 $time_1 - time_0$。而在多服务更新的及时性实验中，开始时间 $time_0$ 为两个服务中较早收到动态更新请求的时间，结束时间 $time_1$ 为两个服务中较晚完成动态更新的时间，该组及时性实验的计算结果同样为 $time_1 - time_0$。每组实验将独立重复多次进行，通过对多次实验结果进行求和平均，便可得到该组实验的及时性实验结果。

实验结果如图 5-3 所示。从图中可以看出，两种算法更新目标服务所需要耗费的时间都随系统负载的增加而增加。对于同一系统负载的情况下，Quiescence 算法要求的安全点较为严格，更新所耗费的时间较长。而本文中实现的 VCIM 和 VCIM-MUL 所耗费的更新时间则要少一些，优于 Quiescence 算法。同时比较 VCIM 和 VCIM-MUL，可以看出同时更新多个服务并未对更新的及时性造成较大的影响。

图 5-3: 及时性实验结果

5.3.3 干扰性 (Disruption)

干扰性表示应用动态更新算法给系统正常处理用户请求所导致的中断程度，即相较于没有动态更新请求下，动态更新过程对系统处理用户请求额外增加的时间。某种动态更新算法的干扰性越小，表明该算法更为透明高效。

干扰性实验的结果可包括：总干扰性 (Total Disruption) ，平均干扰性 (Average Disruption) 。

由于需要计算动态更新算法导致额外增加的时间，因此干扰性实验只对受更新影响的分布式事务进行计算。对于某一个分布式事务，其受更新影响当前仅当对应的事务生命周期与服务进行更新的时间段相互重叠。在进行干扰性实验时，首先进行基准组实验，即在没有更新请求的状态下，系统正常处理用户请求的响应时间，重复多次后计算得到平均的响应时间，记为 $time_0$。然后对于使用特定更新算法的第 j 次实验中，统计所有受更新影响的用户请求 (分布式事务) 的数量，记为 $count_j$，以及这些请求对应的响应时间，记为 $time_i$。该组实验共进行多次，对应的总干扰性计算结果为：$TotalDisruption = \sum_{j=1}^{10} \sum_{i=1}^{count_j} (time_i - count_j \cdot time_0)$，对应的平均干扰性计算结果为：$AverageDisruption = TotalDisruption/(\sum_{j=1}^{10} count_j)$，其中 $count_j$ 表示在第 j 次实验中，受更新影响的事务的总数量。

总干扰性和平均干扰性的实验结果分别如图 5-4 、图 5-5 所示。从干扰性实验结果可以明显看出，由于 Quiescence 算法在更新过程中需要阻塞外来请求，因此对系统造成的干扰程度较大。而 VCIM 和 VCIM-MUL 可以显著降低更新过程对系统的影响，明显优于 Quiescence 算法。比较 VCIM 和 VCIM-MUL，两者都未对系统造成显著的干扰，表示更新服务的数量对于系统的干扰性影响较小。

图 5-4: 总干扰性实验结果

图 5-5: 平均干扰性实验结果

5.4 实验结论

通过上述的实验，我们对各个动态更新算法在安全性、及时性、干扰性三方面进行了性能对比：

安全性： Quiescence 算法和 VCIM 算法都保证了动态更新过程中的安全性，不会导致系统出现不一致的情况。而直接切换版本算法对应于常见的蓝绿部署和金丝雀部署方案，它们的主要目标是版本的快速更新迭代，并未对安全更新点进行考虑，显然将有可能导致更新过程中系统出现不一致的情况。因此，在安全性方面，Quiescence 算法和 VCIM 算法要优于直接切换版本算法。

及时性： Quiescence 算法利用静态依赖信息，更新需要等待相关事务全部结束，而 VCIM 算法则利用动态依赖信息，只需保证相关事务不会再次调用到旧版本的目标更新服务即可。因此在及时性方面，VCIM 算法要优于 Quiescence 算法。

干扰性： 由于 Quiescence 算法更新时对相关服务的状态具有较强的约束性，更新期间需要阻塞相关的用户请求，导致对系统造成较大的干扰性。而 VCIM 算法使用流量代理接管用户请求的方式，不需要阻塞事务的运行，可以降低对系统造成的干扰。因此在干扰性方面，VCIM 算法要优于 Quiescence 算法。

与此同时，在 VCIM 算法的实验中，分别进行了单服务更新和多服务更新实验，验证了 VCIM 算法对于多服务同时更新的支持以及算法的有效性。

5.5 应用展示

前述台风演示系统通过具体的前端界面，展示从后端所得到的整合信息，图 5-6 展示了某一时刻的系统运行状态。从图中用户可以直观地看到台风的运行轨迹，当前位置的详细信息以及波及城市的相关信息。

图 5-6: 系统运行状态图

由于开发人员在实现编码时考虑不周，对台风索引信息处理不当，前端台风信息无法形成正确的闭环路径，导致了路径无法被正确地清除。图 5-7 展示了因旧版本 Typhoon 服务所导致的错误路径。

为了修复旧版本 Typhoon 服务的缺陷，开发人员希望能够在保证系统安全性的前提下，对 Typhoon 服务进行动态更新。因此，开发人员首先需要完成代码版本的修复并推送到远端。然后，通过系统的前端控制器查看系统的运行状态与相关信息，如图 5-8 展示了前端控制器的相关模块，其中左侧是当前台风系统所包含的所有微服务仓库列表，中间对应当前系统内服务的调用拓扑图，而通过点击左侧的某一服务，便可在右侧查看其对应的所有代码版本以及当前正在使用的版本。

图 5-7: 错误路径状态图

图 5-8: 前端模块展示

图 5-8 中，开发人员可在右侧的版本树中点击选择刚刚提交的代码版本标识，向更新控制器发起动态更新请求。系统在收到动态更新请求后，将在后台拉取 Typhoon 服务对应版本的代码，执行代码的编译操作，打包并在系统中完成部署，最后应用本文所提出的 VCIM 方法，将流量正确地切换到新版本的 Typhoon 服务中，并执行旧版本实例的撤销操作。图 5-9 展示了 Typhoon 服务

动态更新成功，而图 5-10 则展示了运行新版本 Typhoon 服务的系统运行状态图，台风路径可以被正确消除。

可以看出，系统依据服务的相关状态自动完成服务的动态更新过程，无需额外的人工操作，Typhoon 服务的缺陷被修复的同时保证了系统的持续可用。

图 5-9: 动态更新结束

图 5-10: 正确路径状态图

5.6 本章小结

本章首先对一个实际场景下的分布式微服务系统进行介绍和分析，然后基于此案例进行了相应的动态更新实验。实验通过分析并选取了系统中合适的目标更新服务，重点对各个动态更新算法在安全性、及时性和干扰性三个方面进行了比较，实验中还针对实际应用的需求，添加了同时对多个服务进行更新的相关实验，并对实验结果进行了解释说明，得出了相应的实验结论。最后，利用台风演示系统的前端界面对应用的动态更新过程进行了具体的展示。

第六章　总结与展望

6.1　工作总结

本文首先探讨了基于微服务架构的软件开发技术在工业界应用中的重要性。但随着互联网和软件技术的发展，软件外部运行环境和对应的需求也在不断地变化，微服务系统变得多元复杂且难以维护，因此微服务系统对于动态更新的支持显得尤为重要。本文首先对现有的软件动态更新算法以及它们在安全性、及时性和干扰性方面的表现进行了介绍，并讨论了当前微服务框架和架构在动态更新方面的支持。接着，针对当前微服务架构在动态更新支撑方面的不足，本文对支持动态更新的分布式微服务系统所面临的问题进行了分析，给出了一种面向版本一致性的微服务动态更新方法，对应在事务模型、动态依赖管理以及服务状态三个方面进行了扩展与实现。然后依据此方法，在具体的 Istio 框架上完成了系统的设计与实现。最后，通过实际的应用案例对此动态更新方法和相关更新算法进行了实验对比，验证了此方法在更新过程中的安全性、及时性和低干扰性，以及对多个服务同时进行更新的可行性，并对应用进行了展示。

本文的主要贡献总结如下：

- 提出了一种面向版本一致性的微服务动态更新实现方法，有针对性地弥补了当前主流微服务系统框架在动态更新支撑方面的不足。该方法在事务模型、动态依赖管理和服务状态模型三个方面进行了具体的扩展，为具体的动态更新过程提供运行时支持。
- 基于上述的实现方法，在 Service Mesh 的开源实现框架 Istio 之上完成了系统的设计与实现，并且提供了相关的动态更新管理接口，支持服务运行时的动态更新。同时系统保证动态更新过程的安全性、及时性和低干扰性，模块之间具有较低的耦合度。
- 利用实际的应用案例，应用上述实现方法并进行实验评估，使用不同的更新算法，主要对动态更新过程中各个算法的安全性 (Safety) 、及时性 (Timeliness) 、干扰性 (Disruption) 进行了性能比较，实验中还包括了对多个

服务同时进行更新，验证了上述实现方法对多服务同时更新的支持。

6.2 研究展望

本文探讨了微服务架构在动态更新支持方面的不足，给出了一种面向版本一致性的微服务动态更新实现方法，并且在具体的开源框架 Istio 上进行了模型的扩展和实现，并通过具体的实验验证了此方法的可行性和有效性。然而，还有很多具体的工作需要进一步的研究：

1. 对于真实网络环境下的分布式微服务系统，消息的发送与处理都有可能出现异常，但本文所定义的相关模型均基于如下假设：消息能够正确无异常地完成传输。在未来的工作中，我们将对相关的异常情况进行考虑，对模型进行完善。
2. 在真实运行环境中，节点可能在任意时刻出现崩溃，若目标更新服务运行于之上则更新过程将出现错误。在后续工作中，我们将考虑更新过程发生错误的回滚机制，使得系统在动态更新方面具有更好的健壮性。

致　　谢

时光荏苒，研究生三年的学习时光已经接近尾声，在此我想对我的父母，我的老师和同学们表达由衷的谢意。

感谢我的导师曹春教授。在我读研期间，曹老师一直关心我的科研和生活。本文从选题、成文到修改都离不开曹老师的耐心指导。曹老师对待学术严谨执着，待人和善，平时非常关心学生的生活状况和身体健康。他的谆谆教诲使我受益匪浅。

感谢吕建教授、马晓星教授、陶先平教授、徐锋教授、许畅教授、黄宇教授、胡昊副教授、余萍副教授、姚远副教授、李樾副教授、徐经纬老师、张建莹老师等所有关心和帮助过我的老师。

感谢实验室的所有同学，良好的学习氛围是我顺利完成科研项目的保障，能够和你们一起学习和工作让我感到非常荣幸。

感谢同寝的邱圣广和张文明同学，你们的陪伴是我宝贵的精神财富。

最后向我的家人和女朋友致以最诚挚的感谢。感谢父母对我的养育之恩和无条件的支持，使我可以全身心的投入到科研中；感谢我的女朋友对我的关心和鼓励，让我一路上都有人同行，充满喜悦。

参考文献

[1] BUCCHIARONE A, DRAGONI N, DUSTDAR S, et al. From monolithic to microservices: An experience report from the banking domain[J]. Ieee Software, 2018, 35(3): 50 – 55.

[2] TAIBI D, LENARDUZZI V, PAHL C. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation[J]. IEEE Cloud Computing, 2017, 4(5): 22 – 32.

[3] AL-DEBAGY O, MARTINEK P. A comparative review of microservices and monolithic architectures[C] // 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI). 2018: 000149 – 000154.

[4] 马晓星, 刘譞哲, 谢冰, et al. 软件开发方法发展回顾与展望 [J/OL]. 软件学报, 2019, 1: 3 – 21.
http://www.jos.org.cn/1000-9825/5650.htm.

[5] LEWI J, FOWLER M. Microservices[EB/OL]. 2014 [2021-04-15].
https://www.martinfowler.com/articles/microservices.html.

[6] DRAGONI N, GIALLORENZO S, LAFUENTE A L, et al. Microservices: yesterday, today, and tomorrow[J]. Present and ulterior software engineering, 2017: 195 – 216.

[7] BALALAIE A, HEYDARNOORI A, JAMSHIDI P. Microservices architecture enables devops: Migration to a cloud-native architecture[J]. Ieee Software, 2016, 33(3): 42 – 52.

[8] TAIBI D, LENARDUZZI V, PAHL C. Architectural Patterns for Microservices: A Systematic Mapping Study.[C] // CLOSER. 2018: 221 – 232.

[9] NEAMTIU I, DUMITRAŞ T. Cloud software upgrades: Challenges and opportunities[C] // 2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems. 2011 : 1 – 10.

[10] BARESI L, GUINEA S, LA MANNA V P. Consistent runtime evolution of service-based business processes[C] // 2014 IEEE/IFIP Conference on Software Architecture. 2014 : 77 – 86.

[11] ELLIOT S. DevOps and the cost of downtime: Fortune 1000 best practice metrics quantified[J]. International Data Corporation (IDC), 2014.

[12] KRAMER J, MAGEE J. The evolving philosophers problem: Dynamic change management[J]. IEEE Transactions on software engineering, 1990, 16(11) : 1293 – 1306.

[13] FOWLER M. BlueGreenDeployment[J]. WWW, Available: http://martinfowler.com/bliki/BlueGreenDeployment.html, 2010.

[14] SATO D. Canary update strategies[J]. WWW, Available: https://martinfowler.com/bliki/CanaryRelease.html, 2014.

[15] TARVO A, SWEENEY P F, MITCHELL N, et al. CanaryAdvisor: a statistical-based tool for canary testing[C] // Proceedings of the 2015 International Symposium on Software Testing and Analysis. 2015 : 418 – 422.

[16] GABBRIELLI M, GIALLORENZO S, GUIDI C, et al. Self-reconfiguring microservices[G] // Theory and Practice of Formal Methods. [S.l.] : Springer, 2016 : 194 – 210.

[17] Jolie. Official Web Site.[EB/OL]. [2021-04-15]. https://www.jolie-lang.org/.

[18] SAMPAIO A R, KADIYALA H, HU B, et al. Supporting microservice evolution[C] // 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2017 : 539 – 543.

[19] BOYER F, ETCHEVERS X, DE PALMA N, et al. Architecture-based automated updates of distributed microservices[C] // International Conference on Service-Oriented Computing. 2018 : 21 – 36.

[20] SEGAL M E, FRIEDER O. On-the-fly program modification: Systems for dynamic updating[J]. IEEE software, 1993, 10(2) : 53 – 65.

[21] MAGEE J, KRAMER J. Dynamic structure in software architectures[J]. ACM SIGSOFT Software Engineering Notes, 1996, 21(6) : 3 – 14.

[22] VANDEWOUDE Y, EBRAERT P, BERBERS Y, et al. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates[J]. IEEE Transactions on Software Engineering, 2007, 33(12) : 856 – 868.

[23] MA X, BARESI L, GHEZZI C, et al. Version-consistent dynamic reconfiguration of component-based distributed systems[C] // Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011 : 245 – 255.

[24] BARESI L, GHEZZI C, MA X, et al. Efficient dynamic updates of distributed components through version consistency[J]. IEEE Transactions on Software Engineering, 2016, 43(4) : 340 – 358.

[25] 王怀民, 史佩昌, 丁博, et al. 软件服务的在线演化 [D]. [S.l.] : [s.n.], 2011.

[26] HICKS M, NETTLES S. Dynamic software updating[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2005, 27(6) : 1049 – 1096.

[27] MENS T, WERMELINGER M, DUCASSE S, et al. Challenges in software evolution[C] // Eighth International Workshop on Principles of Software Evolution (IWPSE'05). 2005 : 13 – 22.

[28] ARNOLD R S. Software change impact analysis[M]. [S.l.] : IEEE Computer Society Press, 1996.

[29] VANDEWOUDE Y, RIGOLE P, URTING D, et al. Draco: An adaptive runtime environment for components[J]. Appendix of the EMPRESS deliverable for Runtime Evolution and Dynamic (Re) configuration of Components, 2003.

[30] BIDAN C, ISSARNY V, SARIDAKIS T, et al. A dynamic reconfiguration service for CORBA[C] // Proceedings. Fourth International Conference on Configurable Distributed Systems (Cat. No. 98EX159). 1998 : 35 – 42.

[31] CHEN X, SIMONS M. A component framework for dynamic reconfiguration of distributed systems[C] // International Working Conference on Component Deployment. 2002 : 82 – 96.

[32] NEWMAN S. Building microservices: designing fine-grained systems[M]. [S.l.] : " O'Reilly Media, Inc.", 2015.

[33] NADAREISHVILI I, MITRA R, MCLARTY M, et al. Microservice architecture: aligning principles, practices, and culture[M]. [S.l.] : " O'Reilly Media, Inc.", 2016.

[34] JOHNSON R, HOELLER J, DONALD K, et al. The spring framework–reference documentation[J]. interface, 2004, 21 : 27.

[35] SU P, CAO C, MA X, et al. Automated management of dynamic component dependency for runtime system reconfiguration[C] // 2013 20th Asia-Pacific Software Engineering Conference (APSEC) : Vol 1. 2013 : 450 – 458.

[36] MORGAN W. What'sa service mesh? And why do I need one[J]. Tarkistettu, 2017, 27 : 2019.

[37] LI W, LEMIEUX Y, GAO J, et al. Service mesh: Challenges, state of the art, and future research opportunities[C] // 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE). 2019 : 122 – 1225.

[38] EL MALKI A, ZDUN U. Guiding architectural decision making on service mesh based microservice architectures[C] // European Conference on Software Architecture. 2019 : 3 – 19.

[39] SHEIKH O, DIKALEH S, MISTRY D, et al. Modernize digital applications with microservices management using the istio service mesh[C] // Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering. 2018 : 359 – 360.

[40] WANG Y, MA D. Developing a process in architecting microservice infrastructure with Docker, Kubernetes, and Istio[J]. arXiv preprint arXiv:1911.02275, 2019.

[41] Netflix Open Source Software.[EB/OL]. [2021-04-15]. http://netflix.github.io/.

[42] REN R, MA J, SUI X, et al. D 2 P: a distributed deadline propagation approach to tolerate long-tail latency in datacenters[C] // Proceedings of 5th Asia-Pacific Workshop on Systems. 2014 : 1 – 6.

简历与科研成果

基本信息

王东宇，男，汉族，1996 年 3 月出生，海南省临高人。

教育背景

2018 年 9 月 — 2021 年 6 月 南京大学计算机科学与技术系 硕士
2014 年 9 月 — 2018 年 6 月 南京大学计算机科学与技术系 本科

攻读硕士学位期间完成的学术成果

1. 曹春, 马晓星, 王东宇.“一种支持事务一致性的微服务动态更新方法”, 专利号：202110263875.0
2. 软件著作权：微服务动态演化支撑平台 V1.0, 登记号：2020SR1259091

攻读硕士学位期间参与的科研课题

1. 国家重点研发项目：软件定义的人机物融合云计算支撑技术与平台 (2018YFB1004805)，2018 年 5 月-2021 年 4 月
2. 国家重点研发项目：可持续演化的智能化软件理论、方法和技术 (2017YFB1001800)，2017 年 9 月-2021 年 8 月

学位论文出版授权书

本人完全同意《中国优秀博硕士学位论文全文数据库出版章程》（以下简称“章程”），愿意将本人的学位论文提交“中国学术期刊（光盘版）电子杂志社”在《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》中全文发表。《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》可以以电子、网络及其他数字媒体形式公开出版，并同意编入《中国知识资源总库》，在《中国博硕士学位论文评价数据库》中使用和在互联网上传播，同意按“章程”规定享受相关权益。

作者签名：_____

2021 年 5 月 30 日

论文题名	面向版本一致性的微服务动态更新技术研究				
研究生学号	MG1833071	所在院系	计算机科学与技术系	学位年度	2021
论文级别	☑ 硕士 ☐ 硕士专业学位 ☐ 博士 ☐ 博士专业学位 （请在方框内画勾）				
作者电话	18362926576		作者 Email	wdongyu@outlook.com	
第一导师姓名	曹春 教授		导师电话	18951679203	

论文涉密情况：

☑ 不保密

☐ 保密，保密期：_____年_____月_____日 至 _____年_____月_____日

注：请将该授权书填写后装订在学位论文最后一页（南大封面）。