中国科学技术大学

专业硕士学位论文

(工程硕士）



**基于领域驱动的第三方支付清结算系统的设计与实现**

作者姓名： 杨武

专业领域： 软件工程

校内导师： 孙广中 教授

企业导师： 杜经纬 专家工程师

完成时间： 二〇二三 年 九 月 八 日

University of Science and Technology of China

A dissertation for master’s degree

(Master of Engineering)



**Design and Implementation of Third-party Payment and Settlement Systems Based on Domain Driven Design**

Author：Yang Wu

Speciality：Software Engineering

Supervisor：Associate Professor. Sun Guangzhong

Advisor：Senior Engineer. Du jinwei

Finished time: September 8, 2023

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文,是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：\_\_\_\_\_\_\_\_\_\_\_ 签字日期：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

控阅的学位论文在解除后也遵守此规定。

□公开 □控阅（\_\_\_\_年）

作者签名：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 导师签名：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

签字日期：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 签字日期：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**摘 要**

在移动互联网技术迅速发展和全球移动支付行业持续增长的背景下，我国移动支付市场正在经历前所未有的快速增长，并取得了显著的成就。我国政府高度重视支付清结算行业的发展，支付清结算技术发展迅猛，产业规模不断扩大。第三方支付清算平台在我国支付清算行业中的地位日益重要，其在提供便捷支付服务，推动数字经济发展和普惠金融方面的贡献不可忽视。然而，随着交易规模和业务的不断扩大，第三方支付清结算系统的复杂性、稳定性和扩展性问题面临挑战，而传统的基于数据库开发模式的清结算系统，业务逻辑依赖数据模型而不是业务模型，导致系统与业务逻辑紧密耦合，难以适应市场业务需求的快速变化，系统开发和维护成本居高不下。

滴滴作为领先的出行公司，在国际化转型的过程中，其国际化支付面临着巨大的交易量和复杂的海外支付环境等挑战，对支付架构的迭代效率及扩展性要求苛刻。由于原来的清结算系统不能适应当前支付交易数据的清结算需求，为此需要对支付清结算系统进行重构设计，建设稳定高效的国际支付中台。

为了解决上述问题，本文基于对国内外支付清洁算系统架构的分析与研究，结合滴滴国际支付清结算系统的业务需求，提出了一种基于领域驱动设计的清结算系统设计与实现方案。清结算系统的工作内容和研究重点如下：（1）根据领域驱动设计思想，对清结算系统进行需求分析，将系统划分为不同层级和模块，使用统一语言（Ubiquitous Language ）分析领域事件、聚合根、实体、值对象、仓库和服务等模型元素，设计清结算领域模型。（2）根据支付中台清算业务需求，使用UML统一建模语言，分析清结算系统子模块的静态结构和动态结构，设计清结算系统核心类图、顺序图、业务活动图和状态图等图表。（3）采用主流后端技术，如Spring、SpringBoot、MyBatis、Mysql、Apollo和RocketMQ等框架技术开发实现系统。（4）使用JUnit5测试框架对系统功能函数进行单元测试；使用Jmeter压测软件对系统模块进行性能测试。该系统已经部署在实际生产环境中使用，通过配置化功能，系统能够满足不断变化的商户资金结算服务，后期配置维护十分方便，提高了系统的可扩展性；同时采用领域驱动设计的系统以领域模型为核心，能够及时响应市场需求变化，降低了后期系统的开发维护成本。

**关键词：**第三方支付 清结算系统 领域驱动设计

**ABSTRACT**

In the backdrop of rapid development of mobile internet technology and consistent growth of global mobile payment industry, China's mobile payment market is experiencing unprecedented growth, achieving notable success. The Chinese government highly values the development of the payment clearing industry, characterized by rapid technological advancements and consistent industry growth. The role of third-party payment clearing platforms in the payment clearing industry is increasingly crucial, with their contributions towards providing convenient payment services, promoting digital economy, and inclusive finance being indispensable. However, with the constant expansion of transaction volume and business scope, challenges arise in the complexity, stability, and scalability of third-party payment clearing systems. The traditional systems developed based on database models, where business logic is dependent on data models rather than business models, lead to tight coupling between systems and business logic, making them less adaptable to rapid changes in market business demands, with persistent high costs for system development and maintenance.

DiDi, as a leading transportation company, faces significant challenges such as large transaction volumes and complex overseas payment environments during its internationalization transition, thereby placing high demands on the efficiency and scalability of its payment architecture. The original clearing system was incapable of meeting current payment transaction data clearing requirements, necessitating a re-design and reconstruction of the payment clearing system to establish a stable and efficient international payment platform.

To address these challenges, this paper, based on the analysis and research of domestic and international payment clearing system architectures, and in combination with the business requirements of DiDi's international payment clearing system, proposes a design and implementation plan for a clearing system based on Domain-Driven Design (DDD). The work and research focus of the clearing system are as follows:（1）In accordance with DDD, requirement analysis is conducted on the clearing system, dividing the system into different layers and modules. The Ubiquitous Language is used to analyze domain events, aggregate roots, entities, value objects, repositories, services, and other model elements, and to design the clearing domain model.（2）As per the business requirements of the payment platform clearing, UML is used to analyze the static and dynamic structures of the sub-modules of the clearing system and to design core class diagrams, sequence diagrams, business activity diagrams, and state charts.（3）Mainstream backend technologies such as Spring, SpringBoot, MyBatis, MySQL, Apollo, and RocketMQ are employed to develop and implement the system.（4）The JUnit5 testing framework is used for unit testing of system functional methods; Jmeter performance testing software is used for performance testing of system modules.The system has been deployed in a real production environment. Through its configuration capabilities, it can meet the constantly changing merchant funds settlement services, making subsequent configuration maintenance highly convenient and enhancing system scalability. At the same time, systems designed with DDD, centered around the domain model, can respond to market demand changes in a timely manner, reducing the cost of subsequent system development and maintenance.

**Keywords:** Third-party Payment Clearing System Domain-Driven Design

**目 录**

[摘 要 I](#_Toc311882828)

[ABSTRACT II](#_Toc1954591516)

[第1章 绪论 1](#_Toc762261253)

[1.1 系统开发背景 1](#_Toc1584924816)

[1.2 国内外研究现状 2](#_Toc444225124)

[1.2.1 国外应用现状与发展趋势 2](#_Toc1438502096)

[1.2.2 国内应用现状与发展趋势 3](#_Toc533829546)

[1.3 解决的主要问题 4](#_Toc2033986103)

[1.4 本文主要工作 4](#_Toc1559740175)

[1.5 论文组织结构 5](#_Toc220242296)

[第2章 理论及相关技术 6](#_Toc1497945091)

[2.1 领域驱动设计理论研究 6](#_Toc1012350656)

[2.1.1 领域驱动产生背景 6](#_Toc64540211)

[2.1.2 领域驱动基本原理 7](#_Toc248084542)

[2.1.3 领域驱动设计技术方法 8](#_Toc1291138567)

[2.2 相关技术与框架 11](#_Toc1991126281)

[2.2.1 Spring/SpingBoot 11](#_Toc621733566)

[2.2.2 MyBatis 12](#_Toc1968101107)

[2.2.3 RocketMQ 12](#_Toc184690608)

[2.2.4 Apollo 12](#_Toc981178741)

[2.2.5 XXL-JOB 13](#_Toc144174674)

[2.3 本章小结 13](#_Toc782192102)

[第3章 需求分析 14](#_Toc1555255027)

[3.1 系统概述 14](#_Toc287505)

[3.2 需求导出 14](#_Toc537129241)

[3.2.1 领域需求导出 16](#_Toc1657385146)

[3.2.2 运营系统需求导出 17](#_Toc661763585)

[3.3 功能性需求 17](#_Toc442765282)

[3.3.1 清结算系统用例 18](#_Toc525257719)

[3.4 非功能性需求 20](#_Toc1848694063)

[3.4.1 性能需求 20](#_Toc1207712045)

[3.4.4 数据安全性 21](#_Toc908871)

[3.5 本章小结 22](#_Toc243009368)

[第4章 系统概要设计 23](#_Toc1892035029)

[4.1 系统设计目标和方法 23](#_Toc1642371274)

[4.2 系统静态结构设计 23](#_Toc1726687227)

[4.2.1 系统架构设计 23](#_Toc1485702278)

[4.2.1 系统功能模块设计 24](#_Toc1405822677)

[4.3 系统动态结构设计 25](#_Toc1046648045)

[4.3.1 清结算系统活动图 25](#_Toc975139738)

[4.3.2 清结算系统数据流图 26](#_Toc1725866309)

[4.4 本章小结 27](#_Toc573435334)

[第5章 系统详细设计 28](#_Toc1968534449)

[5.1 详细设计目标 28](#_Toc1025418661)

[5.2 清算模块设计 28](#_Toc655168252)

[5.2.1 Binlog清分详细设计 28](#_Toc1264153195)

[5.2.2 计费详细设计 30](#_Toc1567028594)

[5.2.3 清算汇总详细设计 36](#_Toc310132550)

[5.3 结算模块设计 40](#_Toc454956581)

[5.3.1 结算单创建 40](#_Toc1413473547)

[5.3.2 绑定结算详情单 43](#_Toc785801315)

[5.3.3 结算流程 45](#_Toc2085755802)

[5.4 异常模块设计 48](#_Toc1922194233)

[5.5 数据库设计 50](#_Toc1721972210)

[5.5.1 清算数据模型 50](#_Toc1697306498)

[5.5.2 结算数据模型设计 54](#_Toc1605028785)

[5.5.3 异常处理数据模型设计 56](#_Toc1176699528)

[5.6 本章小结 57](#_Toc612061873)

[第6章 系统实现与测试 58](#_Toc477230381)

[6.1 系统的部署流程 58](#_Toc2107075569)

[6.2 测试概要 58](#_Toc1613749153)

[6.2.1 测试概述 58](#_Toc1711036508)

[6.2.2 测试环境 59](#_Toc437072979)

[6.3 系统功能测试 60](#_Toc1491485313)

[6.3.1 测试需求 60](#_Toc1964527807)

[6.3.2 测试设计 61](#_Toc257779624)

[6.3.3 测试结果 64](#_Toc1027624569)

[6.3.4 优化策略 65](#_Toc1222642009)

[6.4 系统非功能测试 66](#_Toc1820710767)

[6.4.1 测试需求 66](#_Toc1191374866)

[6.4.2 测试设计 67](#_Toc299848234)

[6.4.3 测试结果 68](#_Toc1552632976)

[6.4.4 优化策略 69](#_Toc1028632935)

[6.5 本章小结 70](#_Toc990380195)

[第7章 结 论 71](#_Toc174189468)

[7.1 工作总结 71](#_Toc582177815)

[7.2 课题展望 71](#_Toc727040973)

[7.2.1 不足之处 71](#_Toc195681781)

[7.2.2 课题展望 72](#_Toc1026229710)

[参 考 文 献 74](#_Toc1401566913)

[致 谢 77](#_Toc386982848)

第1章 绪论

1.1 系统开发背景

随着互联网和信息技术高速发展，我国移动支付市场蓬勃发展，并取得显著成绩。2022年，根据中国人民银行的数据，非银行支付机构处理网络支付业务1310241.81亿笔，金额 337.87 万亿元，反映了第三方支付清算平台在我国支付清算行业中的重要地位。2015年12月，中国人民银行发布《非银行支付机构网络支付业务管理办法》，制定了第三方支付机构进行网络支付业务的相关规定。2023年4月，央行指出要强化支付清算体系顶层治理，建设稳定高效的支付清算体系。以上政策和法规都表明了中国政府对支付清算行业的重视，以及对维护其安全、稳定运行的承诺。随着全球新一轮科技和产业革命的蓬勃发展，第三方支付清算行业将迎来新的机会，支付结算规模将不断扩大。

滴滴国际化经过三年多探索，业务遍布全球十多个国家，在中国互联网公司出海中出类拔萃。滴滴国际化支付承载着出行、外卖、金融所有出海业务，面临着每天千万级交易、数亿流水、质量及效率参差不齐的海外支付环境等挑战。然而，原有的支付清算系统并不能满足当前业务的发展需求，需要重新建设一个高效稳定的支付清算系统，以支持各种交易场景下的手续费、税费计费规则和结算提现规则，给商户提供高效稳定的资金结算服务。

清结算系统是支付系统的一个核心子系统，在第三方支付平台中承担着资金清算、结算和提现等资金处理业务，对资金处理准确性、系统可扩展性和可维护性等指标有着严格要求。稳定高效的第三方支付平台清结算系统，可提高平台的资金处理精确度和结算效率，显著提升平台的支付清结算服务能力[[1]](#footnote-0)[1]。然而，传统的基于数据库建模的软件系统，将系统业务与数据库表紧耦合，在业务需求、业务场景和政策多变的支付清算领域中，无法快速响应市场变化和进行敏捷开发，导致系统模块依赖关系复杂，模块职责边界模糊，后期开发和维护成本指数级增长。领域驱动设计作为一种应对复杂系统的软件建模方法，自2003年诞生以来一直受到软件社区的追捧，因其边界上下文（Bound Context，简称 BC）概念能够有效进行微服务边界的划分而在微服务时代广受欢迎。领域驱动是一套完整的软件设计理论，提出了一系列战略设计和战术设计概念，在20多年的实践发展中逐渐形成了领域驱动设计模式。基于领域驱动的系统设计，能够有效管理系统的复杂性，应对不断变化的业务需求，保持系统的灵活性，具有较高的可扩展性和可维护性。

1.2 国内外研究现状

目前，国外PayPal[2]、Apple Pay[3] 和 Google Pay[4]等第三方支付具有广泛的影响力，特别是在欧美市场，这些支付平台被广泛使用。国内的支付宝[5]和微信支付[6]成为主导市场的两大巨头，也在全球范围内发挥着越来越大的影响力。然而，第三方支付的清结算系统架构属于公司内部保密资料，无法从外部文献和网站中获取，并且各个公司根自身业务发展和技术水平采用不同的架构设计。因此，下面只对领域驱动设计进行分析研究。

1.2.1 国外应用现状与发展趋势

领域驱动是一种软件设计方法，它通过统一语言即模型在分析和开发阶段达成共识；通过分层架构划分软件结构；利用领域模型内聚业务规则等方法来解决大型软件开发过程中面临的复杂性问题。

在技术理论方面，领域驱动设计概念（Domain Driven Design，简称 DDD）自2003 年由软件建模大师 Eric Evance [7]首次提出以来，吸引着软件开发者社区的强烈关注,一度被认为是系统建模领域的圣经。越来越多的开发组织开始实践DDD，并出版了相关技术图书介绍其最佳实践。2006年，Jimmy Nilsson[8] 出版《Applying Domain-Driven Design And Patterns》，书中基于Martin Fowler (Patterns of Enterprise Application Architecture) 和Eric Evans (Domain-Driven Design) 对每个DDD原则给出清晰且注释良好的C#代码案例，并首次将领域驱动应用于实际的.NET项目中，为开发者指明了实践道路。2013 年，Vaughn Vernon在《Implementing Domain-Driven Design》中介绍了如何将 DDD 技术落地的实践指南[9]，提出了包括六边形架构、SOA、REST、CQRS、事件驱动等架构模式适应不同的软件建模需求。2015年， [Scott Millett](https://book.douban.com/search/Scott Millett) 在 《Patterns, Principles, and Practices of Domain-Driven Design》中提出使用战略模式构建有效的领域模型，使用战术模式来维护领域模型的一致性[10]，并提供了大量的最佳实践和模式来构建可维护、可扩展的软件系统。2018年，Scott Wlaschin 在《Domain Modeling Made Functional》中首次将 Functional Programming 与DDD 相结合来对现实需求建模[11]，设计出的软件可能比采用面向对象的方式更优雅更简洁。

在技术应用方面，2006年，Harald Wesenberg[12]等人采用领域驱动设计来评估现成的第三方商业软件，发现在领域专家的帮助下，能够全面分析现有业务领域，将隐性的领域知识显示化和共享化，提高了沟通效率，节省了软件分析成本。2007年，挪威大型石油和天然气公司 Statoil ASA[13]，准备设计一个新系统来支持石油和炼油产品的交易和供应业务。该公司采用领取驱动设计技术结合敏捷软件开发方法对遗留系统进行重构，重构后的系统提高了系统的性能和代码质量。2021 年，[Alam Rahmatulloh](https://ieeexplore.ieee.org/author/37087007622)[14] 等人将DDD应用到物联网监控应用的开发中，并表示采用DDD能很好的划分软件模块、准确定义系统服务，所开发的系统具有良好的扩展性和维护性；同年，PeterOukes[15] 通过研究荷兰土地管理系统发现应用领域驱动设计思想使土地管理系统中的各个模块解耦，通过事件源（Event Sourcing）实现更好的土地登记可追溯性、功能和系统互操作性。欧洲最大的在线时尚零售商之一的Zalando[16]，基于分布式架构，围绕业务领域组织，采用领域驱动设计 (DDD) 实现“高内聚和低耦合”的微服务架构，以持续高质量的软件交付支撑业务持续增长。

以上研究表明，国外在领域驱动设计的理论研究和技术应用方面都处于前沿水平。在20年的理论到实践的发展过程中，已经形成一套完整成熟的软件设计理论和开发方法，在微服务和大型软件开发中有着广泛应用。

1.2.2 国内应用现状与发展趋势

国对领域驱动的关注比较晚，直到2008年才逐渐意识到其重要性，但很少将其应用到实际软件开发中。2008年，王忠等人在分析企业级应用系统开发现状时，提出基于数据库开发设计的软件开发方法已不能很好满足企业级系统开发，从理论层面分析了领域驱动设计软件开发方法的相关概念和开发流程[17]。2009年，郑琴琴结合领域驱动设计的思想开发实现了服装行业共享平台，结果表明领域驱动设计的思想使得平台能够快速响应需求的变化，有利于更好地维护[18]。

随着微服务的爆发，领域驱动因其界限上下文能为微服务的划分提供依据而大火起来。2015年，易立江基于领域驱动设计的思想建立了新型证券交易系统[19]。2017年，胡俊霞将领域驱动设计应用到企业信息化系统中，可解决系统的可维护性和可扩展性[20]。2021年，张帅基于领域驱动思想设计并实习了AGV调度系统，该系统已成功运用于十余个大型智能物流项目，能够满足实际应用需求，并且具有良好的扩展性、稳定性[21]。同年，黄喆基于领域驱动设计的银行存款业务核心系统，提高了银行业务处理的效率和正确性，降低了维护成本[22]。贾子甲等人通过应用系统文献综述的方法对2003年~2019年7月之间发表的领域驱动相关文献进行分析，研究得出领域驱动设计注重业务的领域知识，能够帮助开发者更好地进行软件设计[[2]](#footnote-1)[23]，并指出使用领域驱动设计能清晰地划分领域间的依赖关系、提升架构质量和促进业务专家和技术专家之间的沟通，并且以敏捷方式实现复杂架构的不断演进。

综上所述，国内关于领域驱动设计思想的认可度不断提高，相关理论研究也在不断深入，但实际企业级项目应用还不够广泛，各行各业也在此基础上不断学习和研究。支付清结算业务的特殊性意味着支付清结算核心系统使用领域驱动设计的思想进行开发和设计也是势在必行的。

1.3 解决的主要问题

本课题通过分析和梳理清结算业务的整体流程，理清清结算系统与各业务子系统的关系，将清结算业务逻辑从整个支付系统中独立出来，通过配置化手续费、结算规则和提现规则等，为商户提供多样化的资金结算服务。在系统设计上，采用领域驱动设计思想，划分清结算领域模块，定义领域模型，使重构后的系统能够聚焦业务领域，适应快速变化的市场需求。同时，所开发的软件具有可维护性和可扩展性的特点，能节省后期软件开发维护成本。

1.4 本文主要工作

（1）领域驱动设计及其他技术理论研究。

分析研究领域驱动设计的相关理论技术，包括产生背景、基本原理、技术特点和适用场景四个部分，介绍清结算系统使用到的其它技术，为后续系统设计和建模做准备。

1. 清结算系统需求分析及概要设计

根据企业支付清结算业务的发展现状，分析清结算系统的业务需求。通过对业务场景和业务用例的分析，设计系统的功能性和非功能性需求。由于清结算系统是支付系统的核心子系统，故分析了整体的支付系统结构以及清结算与其他系统的依赖关系，然后结合领域驱动设计思想，设计清结算各个子领域的模型结构。

1. 清结算系统详细设计及功能实现

详细设计清结算系统核心模块：数据获取模块、清分模块、结算模块、异常处理模块的核心功能和业务流程，设计每个模块功能的活动图和状态图。

1. 清结算系统测试分析

通过单元测试和集成测试等方式对系统进行代码质量测试，验证系统是否满足业务目标和用例需求。

1.5 论文组织结构

本文共分为七章内容，各章组织内容如下：

第一章是绪论。本章介绍清结算设计开发的社会背景，分析国内外领域驱动设计的研究现状，同时介绍论文主要的工作内容和解决的主要问题。

第二章是理论及相关技术。本章分析研究系统设计与实现阶段采用的理论与相关技术框架，通过对其原理和特性的研究，表明其理论和技术的优点和适用性。

第三章是需求分析。本章从目前企业清结算业务需求出发，通过系统化和结构化的方法详细分析系统中的参与人员角色，从不同类型用户使用系统的角度分析系统需求。

第四章是系统概要设计。本章给出整体的支付架构图和清结算系统模块图，清晰展示清结算系统和各个系统之间的交互关系，对清结算系统内部的流程结构做了详细的设计。同时，介绍了该系统中数据库的逻辑结构和物理结构。

第五章是系统详细设计。本章将从系统模块角度出发，使用UML建模语言，详细设计各模块功能的具体实现，给出各个模块的领域模型、类型结构和流程图标等。

第六章是系统实现与测试。在系统设计并编程完成之后，需要按照前期提出的标准对系统进行的测试，覆盖软件开发测试过程和软件产品测试过程，包括产品功能、质量、技术文档等多方面，由测试结果判断系统的功能与性能是否满足了前期设计要求。

第七章是结论。对本次课题研究的工作成果进行各方面的总结，分析本次设计完成后仍有待改进的地方以及产生这些不足的原因，对未来一个时期内车联网信息安全体系的发展趋势进行展望。

第2章 理论及相关技术

2.1 领域驱动设计理论研究

本章首先研究领域驱动设计的相关理论和技术，分为产生背景、基本原理、技术特点和适用场景四个部分，然后介绍了清结算系统使用到的其它技术，阐述其特点及适用性。

2.1.1 领域驱动产生背景

（1）复杂性挑战

在软件开发中，软件的复杂性包括技术实现的复杂性和软件领域的复杂性，而后者往往是软件成败的关键。2023年，[Tashtoush Yahya](https://scholar.cnki.net/home/search?sw=6&sw-input=Tashtoush Yahya" \t "/Users/yangwu/Documents\\x/_blank)等人采用机器学习算法对软件可读性和软件复杂度进行实证研究，结果表明两者之间相互影响，是软件质量的重要组成部分[24]。2017年，何磊将McCabe圈复杂度和Halstead复杂度用于评估Java软件复杂度，设计实现基于代码复杂度的软件演化评估工具，对复杂度高的模块向开发人员预警，实现对软件复杂度指标的监控预警[25]。对于绝大多数的软件，最复杂的不是技术实现而是软件所研究应用的领域本身，即用户的活动和业务逻辑。在软件设计过程中，应该首先处理领域复杂性，对领域建模，再从领域出发做技术选型和设计。成功软件设计必须系统的处理软件的核心方面，它包括：1）在软件设计中，主要的关注点应该是领域和领域逻辑，而不是技术方法。2）领域驱动设计应该基于领域模型进行设计，并通过模型来处理领域复杂性。

（2）需求分析和设计的分离

在传统的瀑布模型中，软件需求分析和系统设计是分离的。1970年，[Winston W. Royce](https://en.wikipedia.org/wiki/Winston_W._Royce" \o ") 首次在论文中提出瀑布模型，并认为该模型存在重大缺陷，因为测试在开发结束后进行，有可能导致项目失败[26]。美国国防部于94年发布MIL-STD-498（军用标准软件开发和文档），开始明确反对瀑布式方式，转而采用迭代增量式开发[27]。瀑布模型的工作流程首先是领域专家将需求告诉给产品分析师，然后产品分析师消化吸收和抽象后再将需求转述给程序员，最后程序员翻译产品需求设计实现系统。这种单方向的工作模式必定失败的原因是建模过程中缺乏反馈，也即需求和设计的分离。产品分析师完全负责模型的创建，只接受领域专家的输入，然后进行抽象建模，没有从程序员中获得对模型的反馈意见，也没有获得软件先前版本相关人员的实际经验，导致领域知识的单向流动，没有在所有参与人员中获得积累。

1. 软件缺乏抽象

即便是采用迭代的方式开发软件，也不能保证领域知识在团队中得到良好的积累。在《Code Complete》中，抽象作为一种软件设计方法，它使我们以一种简化的观点来考虑复杂的概念，是处理系统复杂度的一种重要手段[28]。然而，现实中常见的场景是，开发者采用直接翻译需求的开发方式来处理软件需求，他们刚刚从领域专家得知某个需求后，就立马进行代码开发，然后将结果展示给领域专家并等待下一个需求。这种直译式的开发虽然能满足当前阶段的软件需求，但是由于缺乏对领域的抽象，开发者无法获得对领域的深入理解，总是限于新需求的被动之中，只知道应该做什么，而忽略了软件背后的领域知识。同时，由于缺乏对软件领域概念的抽象和建模，开发出来的系统只能满足当前使用，不可能从旧的功能整合中获得对系统的新的认识，导致系统缺乏扩展性。

2.1.2 领域驱动基本原理

针对上述软件开发设计的问题，领域驱动设计提出了以下软件建模的基本原理。

（1）基于领域模型的软件建模

领域模型是领域驱动设计的核心，软件建模围绕着领域模型进行设计。领域模型是对大量领域知识的总结概括和领域概念的抽象和简化，它反映了具体领域的核心关注点，是管理领域复杂度的有效方法。同时，领域建模是一个动态活动过程，它要求领域专家和技术专家持续紧密的合作，共同不断的精细化领域模型，分析人员和编程人员对领域知识进行抽象，构建领域模型以便于代码实现，领域专家将自己对领域知识的深入理解反应到领域模型中，保证领域建模是基于真实的业务原则进行的。领域模型作为团队成员的统一交流语言，将彼此对领域知识的认识集中反馈到模型中，降低了对领域知识的误解分歧，同时将需求分析和设计过程有机结合起来。

1. 基于职责分离的软件设计

一个完整的软件系统包含许多模块，各个模块的职责和关注点不同，为了保持领域模型的单一职责和干净独立的领域内涵，需要分离系统中与领域不相关的功能特性。为此，在领域驱动设计中采用分层架构，根据系统职责划分软件层次，隔离关注点，并通过领域元素构建领域模型，使领域模型具有清晰独立的领域含义。

1. 基于重构的领域模型优化

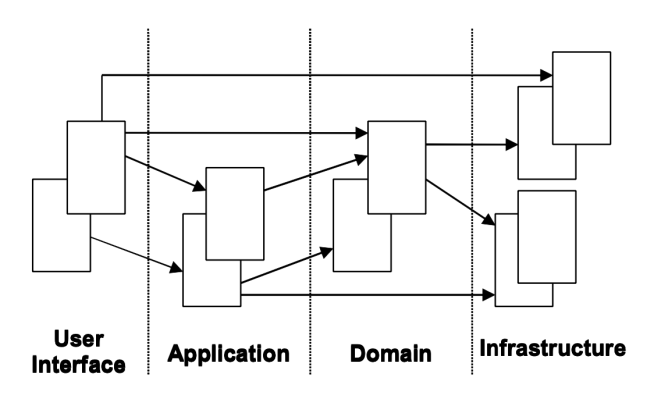
在设计的早期阶段，领域建模往往通过识别需求文档中的名词和动词来进行面向对象设计，把名词设计为对象，动词设计为对象的方法，这种过于简化的设计方法在面向对象的教学中被大家普遍接受。然而，实际上由于开始阶段缺乏对领域知识的深入理解，最初的模型往往停留在对象表面，不能反映领域专家主要的关注点和最本质的领域知识。在对领域模型不断重构优化，不断对领域知识加深理解的过程中，我们会发现新的领域概念，纠正错误的领域对象关系，或者删除多余的领域对象，使得领域模型不断接近领域知识的本质，与领域专家的理解和用户的真实需求保持一致。

2.1.3 领域驱动设计技术方法

领域驱动设计采用分层的设计方法，隔离领域对象和系统功能，同时对领域元素进行分类，明确了每类元素的职责范围，最后通过聚合、工厂和仓储对象管理领域对象的生命周期。

（1）隔离领域对象

为了处理复杂的任务，软件系统一般基于分层架构进行设计，将系统划分不不同层级，每层只关注当前层的任务，并且只依赖下层服务，这样设计的软件能够保持系统的高内聚低耦合特性，便于理解和拓展。领域驱动设计将系统划分为四个层次，分别是用户接口层、应用层、领域层和基础设施层，如图2.1所示。

图2.1 领域驱动分层结构图

下面简要介绍每层的具体内容和职责范围。

1. 用户接口层（User Interface Layer）：用户接口层负责向用户展示信息以及接受用户的输入命令。
2. 应用层（Application Layer）：应用层定义了软件应用的具体任务，是很薄的一层，不包含具体的业务逻辑和业务状态，而是通过协调领域对象来实现任务目标，负责领域对象的编排管理。
3. 领域层（Domain Layer）：领域层负责定义业务概念，展示业务状态和业务规则信息。
4. 基础设施层（Infrastructure Layer）：基础设施层为高层提供通用的基础能力，比如消息发送、领域对象的持久化、缓存等能力。同时，其他层也可以通过依赖基础设施层进行间接的交互。

领域驱动设计分层的主要目的是保持领域层的独立性，领域层将领域相关的逻辑集中在同一层，与其他层分离，不关心领域对象的持久化、展示、应用任务等内容，能够专注于领域知识的表达，使得领域模型具有清晰的领域含义。

（2）领域模型元素

为了清晰地表示模型的不同组成部分，领域驱动设计定义了如下模型元素：

1）实体（Entity）

通过唯一标识（Identity）定义的对象称为实体，不同的实体通过唯一标识而不是实体的属性进行区分，同时实体的生命周期具有连续性。在设计实体模型时，实体的类型、职责、属性和关联关系都应该围绕着实体的唯一标识进行设计。通常通过工具UUID或者外部依赖生成全局唯一编号作为实体的唯一标识。

2）值对象（Value Object）

值对象用来描述模型的某个方面，没有概念上的唯一性，不同的值对象通过属性值进行区分。值对象是不可变对象，典型的值对象如数字、字符串、颜色等对象，它们一般作为参数在对象间传递，并且通常是瞬态的（transient），使用的时候被创建，完成后就销毁。值对象可以组合其他值对象或实体形成更大的值对象，同时也可以作为实体对象的属性。

3）服务（SERVICE）

服务以接口的形式表示领域操作，它独立于实体和值对象，并且没有封装状态。当某个处理方法或转换过程不属于某个实体或值对象的职责时，通常将该操作设计为一个无状态的服务，实现实体和值对象的概念完整性。服务按照不同的用途划分到不同的分层中，比如应用层服务负责接受用户输入、向领域服务传递参数和调用基础服务发送通知等，领域层服务调用领域对象来处理领域逻辑，基础服务负责发送邮件、消息通知等。

（3）生命周期管理

为了管理复杂对象的生命周期、维护对象的一致性状态和保持领域模拟的清晰独立，领域驱动设计提出来三种模式，包括聚合、工厂和仓储模式。

1）聚合（Aggreate）

聚合将一组关联实体和值对象组合成一个单元，定义了关联对象的边界和职责。聚合通过聚合根（聚合内部的一个实体）向外界提供访问内部对象的入口，对聚合进行访问控制。在整个聚合生命周期内，聚合根负责维护聚合内部对象的状态一致性和不变性约束（Invariant），防止对关联对象的修改因缺乏同步导致系统出现不一致状态。

2）工厂（Factory）

工厂将复杂对象或聚合对象的创建组装逻辑封装在工厂中，分离了对象的创建和使用职责。工厂位于对象生命周期的开始阶段，在创建聚合对象时强制保证聚合内部关联对象的不变性约束，实现客户端代码与具体类型的解耦。在实践中，一般使用简单工厂、工厂方法和抽象工厂设计模式来创建对象。特别的，工厂的每个创建方法必须是原子性的，并且在方法内维护对象的不变性约束。同时，工厂方法的返回类型应该与具体类型无关，一般设计为父类类型或接口。

3）仓储（Repository）

仓储封装了数据库操作和元数据映射的逻辑，使设计的重点保持在领域模型上，而不是陷于底层数据的操作细节。仓储将领域层与存储层解耦，为客户端提供一个的获取数据和数据生命周期管理的接口。仓储与工厂的关系比较复杂，它们既存在显著的不同点又彼此合作。首先，仓储与工厂的职责不同，工厂负责创建新对象，而仓储负责从数据库或其他数据源中恢复旧对象。其次，仓储与工厂的所处的对象生命周期也不同，工厂位于对象生命周期的开始阶段，而仓储位于对象生命周期的中间和结束阶段。在实践中，一般通过仓储和工厂合作的方式实现对象的创建和保存，如下图2.2和2.3所示。



图2.2 仓储模型



图2.3 仓储和工厂模型

2.2 相关技术与框架

2.2.1 Spring/SpingBoot

本文采用Spring、SpringBoot开发框架，理由如下：

1. Spring[29]是当前主流的Java企业级开发框架，与同类开发框架相比，Spring提供了诸如IOC容器、AOP切面编程、事务管理和MVC模型等高级特性，能够满足清结算系统开发需要。
2. SpringBoot[30] 是Spring约定大于配置的解决方案，内嵌了Tomca或Jetty服务器而不需要单独部署WAR文件，其自动化配置功能使得开发人员可直接创建独立的、生产级别的Spring应用，大大降低系统开发成本。

Spring框架设计结构如图2.4所示：

图2.4 spring 架构图

2.2.2 MyBatis

本文采用MyBatis框架进行数据持久化，理由如下：

1. MyBatis[31] 是一个优秀的持久层框架，它具有定制化 SQL、存储过程以及高级映射等特性。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集的过程。MyBatis 可以使用简单的 XML 或注解来配置和映射原生类型、接口和 Java 的 POJO（Plain Old Java Objects，普通 Java 对象）为数据库中的记录。
2. MyBatis 相比 Hibernate[32] 等全自动的 ORM ([Object Relational Mapping](https://en.wikipedia.org/wiki/Object%E2%80%93relational_mapping" \o "))框架，开发者可以根据业务场景自己编写SQL，进行SQL优化。特别适用于需要自由控制 SQL、复杂查询或者有性能要求的场景。因此，MyBatis为开发人员编写和优化SQL提供了更高的灵活性和控制度。

2.2.3 RocketMQ

本文采用RocketMQ消息中间架进行异步通信，理由如下：

1. RocketMQ[33] 是Apache 软件顶级开源项目，具有高吞吐、高并发、顺序消息和事务消息、可靠的消息存储和传输、集群和分布式部署等特点。在实际开发中，RocketMQ适用于大规模消息处理、异步处理、应用解耦、流量削峰等场景，已经成为业内共识的金融级可靠业务消息首选方案，被广泛应用于互联网、大数据、移动互联网、物联网等领域的业务场景。在海量交易数据背景下，使用RocketMQ够提高清结算资金结算效率和系统稳定性。
2. 相比于其他消息中间件，RocketMQ拥有更丰富的功能。与 Kafka[34] 相比，RocketMQ 的功能更为丰富，例如支持顺序消息、事务消息。RocketMQ 的可用性和容错性也相对较强，支持消息的事务性提交和回滚，在处理复杂业务逻辑时比较有优势；与 RabbitMQ[35] 相比，RocketMQ 支持更高的并发和吞吐量；与 ActiveMQ[36] 相比，RocketMQ 在处理大规模消息时，性能更优。在金融支付领域和清结算领域，对消息中间件的性能和可靠性要求严苛，而RocketMQ的高并发、顺序消息和事务消息的特点能够满足清结算系统的需求。

2.2.4 Apollo

本文采用Apollo配置中心作为清结算配置管理中心，理由如下：

1. Apollo[37]开源分布式配置中心，提供了诸如分布式、实时更新、版本控制和权限管理等功能，具有使用简单、高可靠性和高可用的优点。在微服务架构中，由于服务数量增多，配置管理的复杂性增加，使用Apollo可以大大简化配置管理的工作。在清结算系统中，商户的手续费配置、交易方式配置、结算方式配置等各种配置通过统一管理在Apollo中，可以大大简化商户结算流程中的配置任务，提高了清结算系统的扩展性和灵活性。
2. 相比于其他配置中心，Apollo具有不同的优点。与 Spring Cloud Config[38] 相比，Apollo 更注重配置的实时性，配置更改后可以立即推送到所有机器。而 Spring Cloud Config 默认情况下，配置更改后需要重启应用或者调用特定的刷新接口才能更新配置。清结算系统对时效性对时效性要求很高，需要及时的获取Apollo结算配置，计算商户交易手续费等各种费用，以提高商户资金结算时效。

2.2.5 XXL-JOB

本文采用XXL-JOB作为清结算任务调度中心，理由如下：

XXL-JOB[39]是一款开源的分布式调度平台，提供了任务管理、调度器管理、日志查看、故障转移、执行器管理和通知报警等功能，具有简单易用、调度模式丰富、支持多种任务类型等优点，适用于大数据处理、游戏后台、金融电商项目等需要对任务进行定时调度的场景。在清结算项目中，需要定时清算和结算商户交易金额，需要定时对清结算流程中的异常事件进行重试，因此XXL-JOB分布式任务调度中心能很好的满足系统的定时任务需求。

2.3 本章小结

本章主要介绍了清结算系统开发中所用到的理论和技术，主要分为领域驱动设计理论和相关技术框架两个部分。通过分析对别其他相似技术和框架，剖析了选型技术的特征、优势和适用场景，并结合清结算系统的技术需求，阐述选择这些技术的理由。

第3章 需求分析

3.1 系统概述

为了解决A企业清结算业务的发展与当前支付清结算系统不匹配的问题，需要重新设计一套清结算系统来满足A企业国际化出行、外卖和钱包等上游业务的发展。本文设计了一套基于领域驱动的清结算系统，该系统能够支持商户结算业务相关的费用配置管理、结算配置管理、各种交易类型手续费的清算和结算和异常处理等需求。

本课题根据领域驱动设计思想，识别业务边界上下文，进而划分子域，然后再每个子域模块内设计核心领域构建元素，包括聚合根、领域服务、仓储等元素。同时，为了保证系统的可扩展性和可维护性，系统模块、接口和类的设计采用设计模式中的SOLID[40][41]五大原则，提高系统复用性，尽可能降低后期开发维护成本。

3.2 需求导出

本次课题设计的清结算系统是支付系统的内部子系统，和清结算系统发生交互的有交易系统、运营系统和账户系统。完整的支付系统如图3.1所示。

图3.1 支付系统架构图

交易中心把商户交易数据通过消息中间件MQ异步发送到清结算中心，清结算中心通过订阅MQ获取交易数据，然后对交易数据进行数据处理等操作，为清结算流程准备数据源。运营平台在商户入网过程中，把商户计费配置和结算配置通过清结算中心提供的API下发到清结算配置数据库，为计费流程和结算流程提供相关配置参数。账户系统为了记录商户和平台清结算相关账户余额的变动情况，需要在计费和结算流程中记录相关账户的余额变化。为了保证商户资金安全和商户主体安全，风控系统需要对包含支付相关的商户交易和商户在清算和结算流程中，对交易数据进行风控校验拦截。交易系统、运营系统、账户系统、风控系统和清结算系统的系统依赖图如图3.2所示。

图3.2 清结算系统依赖图

清结算相关子系统交互顺序图如下。从时间先后顺序来看，交易系统首先通过消息中间件异步的将待清分的交易数据发送到清结算系统，清结算系统接收数据后开始进行数据校验和解析，并将解析后的清分对象保存到清结算服务数据库中，作为数据获取和处理阶段完成的结果；运营系统把商户入驻时添加的各种配置规则通过网络保存到清结算数据库中，然后清结算在计费和结算等流程中查询配置计算手续费、税费等费项，并把各种费项更新到清分对象中，最后保存数据库清分表；清结算系统首先调用计费模块，根据计费配置计算交易各项手续费，然后再根据结算规则判断结算模式，如果是实时结算，就立即调用账户系统记账服务，变更商户待结算余额，如果是周期结算，只需要把费项累加，待结算日期到达之后触发记账服务。在结算流程中，需要对支付类型的交易数据进行风控拦截处理，防止发生资金损失，如果风控通过，进行结算表数据更新和通知账户系统更新商户现金余额等操作。同时，为了提高清结算系统的稳定性和鲁棒性，定时处理清结算流程中发生的异常事件，根据异常状态机进行状态流转。最后，根据结算配置中的提现规则，将符合条件的商户结算资金转结到商户余额账户中，清结算顺序图如图3.3所示。

图3.3 清结算顺序图

3.2.1 领域需求导出

清结算系统需要准确多样的计算手续费，核心特性需求如下：

1. 数据处理：根据交易类型对交易数据进行条件过滤，只保留满足清算条件的交易数据。比如对于支付类交易，清算条件为交易成功且是外部商户。根据交易类型对交易数据进行合法性校验，并组装清分实体落库。
2. 清分计费：根据交易数据中的交易类型和商户计费配置计算手续费、净额等所有费项，更新清分实体和清分表。
3. 账户记账：在实时结算类型中，每笔交易在清分后调用账户中心进行余额变更，在周期结算类型中，先累计各费项，在结算日期统一记账。
4. 手续费清算：根据商户交易类型将手续费累加结算明细单，生成手续费结算详情单。
5. 汇总结算：对结算配置为周期结算的交易，定期汇总已清分记账的结算详情单，生成付款结算单和收费结算单并调用账务系统记账。
6. 风控处理：在结算流程中对支付类交易进行风控审核。
7. 异常处理：对清结算流程中发生异常的单号进行自动重试或人工处理。

3.2.2 运营系统需求导出

运营系统的职责为了更好的管理维护商户信息，包含商户审批，交易查询、商户查询、配置管理和报表等功能。运营系统在和清结算系统交互过程中，主要负责清结算配置管理和清结算流水查询。

1. 配置管理：运营系统在商户入网流程中，需要配置商户计费、计税和结算规则，包括配置添加、编辑、查询和验证。当运营人员审核通过后，商户清结算业务相关配置将会下发到清结算配置数据库中。在国际化业务中，清结算系统需要根据运营主题所在时区，进行国际化处理，其中包括时区转换，语言展示等逻辑。为此，运营人员在Apollo配置中心管理商户信息、时区信息、语言等配置，定时统一下发到清结算服务器。
2. 流水查询：清结算系统在处理商户交易金额的整体业务流程中，会产生清分表流水、结算详情表流水和结算表流水。运营系统通过清结算系统提供的API服务，查询对应商户的清结算流水，为商户自身资金对账、结算报表等需求提供数据源。

3.3 功能性需求

根据上部分对需求进行初步分析与导出，整个清结算系统和核心业务在于对商户交易数据进行清算结算，包括运营系统对配置和清结算流水管理。本节根据清结算系统核心业务流程描述系统的功能性需求。系统需求列表如表3.1所示。

表3.1 清结算系统需求列表

|  |  |  |  |
| --- | --- | --- | --- |
| 需求编号 | 名称 | 参与者 | 说明 |
| R1 | 清分 | 清结算系统  交易系统 | 对交易数据进行处理 |
| R2 | 计费 | 清结算系统 | 计算各种交易手续费 |
| R3 | 记账 | 账户系统 | 调用账户系统记账 |
| R4 | 结算 | 清结算系统 | 汇总商户结算明细，发起结算流程 |
| R5 | 风控 | 风控系统 | 调用风控系统进行风控拦截 |
| R6 | 异常处理 | 清结算系统 | 处理清结算流程中的异常事件 |
| R7 | 配置管理 | 运营系统 | 管理清结算相关配置 |
| R8 | 流水管理 | 运营系统 | 查询清结算相关流水 |

3.3.1 清结算系统用例

根据上述对功能性需求的分析，可得出下面清结算系统核心用例图，如图3.4所示。

图3.4 清结算系统用例图

清结算系统核心有5个用例，分别是清分、计费、清算、结算和异常处理，下面通过用例描述表3.2-3.6简单介绍每个用例的大致流程。

表3.2 清分用例描述

|  |  |
| --- | --- |
| 用例名称 | 清分 |
| 用例描述 | 清结算系统对交易数据进行清分操作 |
| 主执行者 | 清结算系统 |
| 优先级 | 高 |
| 触发条件 | 清结算系统监听到MQ Topic有数据时 |
| 后置条件 | 创建清分对象 |
| 基本事件流 | （1）从MQ Topic 获取交易Binlog[42]数据  （2）将交易数据解析为Binlog 对象  （3）校验数据字段完整性和合法性  （4）将Binlog转换并组装成清分对象 |
| 异常事件流 | Binlog 数据校验异常 |

表3.3 计费用例描述

|  |  |
| --- | --- |
| 用例名称 | 计费 |
| 用例描述 | 清结算系统对清分对象计算各费项手续费 |
| 主执行者 | 清结算系统 |
| 优先级 | 高 |
| 触发条件 | 清分成功 |
| 后置条件 | 完成手续费、分期费、税费、净额计费 |
| 基本事件流 | （1）清分数据落库  （2）清分前置条件校验  （3）创建计费聚合根对象  （4）根据配置规则匹配计算各费项手续费  （5）计费结果后置校验  （6）更新清分对象，保存数据库 |
| 异常事件流 | 1. 配置异常 2. 重复计费异常 3. 数据库更新异常 |

表3.4 清算用例描述

|  |  |
| --- | --- |
| 用例名称 | 清算 |
| 用例描述 | 清结算系统对清分对象进行清算 |
| 主执行者 | 清结算系统 |
| 优先级 | 高 |
| 触发条件 | 清分成功 |
| 后置条件 | 记账成功，完成清分对象各费项累计 |
| 基本事件流 | （1）计费  （2）交易记账  （3）实时结算，实时对各费项记账  （4）周期结算，创建结算详情单，并累计各费项到结算详情单  （5）创建并保存各费项数据库索引 |
| 异常事件流 | 1. 计费异常 2. 记账异常 3. 数据库更新、唯一键冲突异常 |

表3.5 结算用例描述

|  |  |
| --- | --- |
| 用例名称 | 结算 |
| 用例描述 | 清结算系统对结算详情单进行汇总结算 |
| 主执行者 | 清结算系统 |
| 优先级 | 高 |
| 触发条件 | XXL-JOB 定时任务每小时触发结算流程 |
| 后置条件 | 完成商户各类费项的汇总结算 |
| 基本事件流 | 1. XXL-JOB 定时创建结算单 2. 绑定结算详情单到结算单 3. XXL-JOG 定时触发结算主流程 4. 风控处理 5. 将详情单各个费项累加到结算单上 6. 记账，结算各类费项 |
| 异常事件流 | 1. 记账异常 2. 风控异常 |

表3.6 异常处理用例描述

|  |  |
| --- | --- |
| 用例名称 | 异常处理 |
| 用例描述 | 清结算系统对异常事件进行处理 |
| 主执行者 | 清结算系统 |
| 优先级 | 高 |
| 触发条件 | XXL-JOB 定时触发 |
| 后置条件 | 完成异常事件处理 |
| 基本事件流 | 1. 扫描异常表，构造异常处理聚合根 2. 自动重试，构造清分对象 3. 重新清算 4. 自动重试超过阈值，进行人工处理 |
| 异常事件流 | 无 |

3.4 非功能性需求

3.4.1 性能需求

清结算系统整体性能要求如下：

1. 每秒请求数QPS（Queries Per Second）：清结算主要的流量入口为交易系统推送的交易数据库binlog消息，因此清结算系统QPS必须不小于交易系统binlog消息生产者QPS，当前观测到交易binlog最大QPS为96，所以清结算QPS 满足100就可以;
2. 并发度Concurrency：清结算支持多用户实时结算和周期结算，其清分效率不低于每秒1000 个binlog消息，结算效率不低于每秒500个用户。
3. 响应时间RT（Response-time）：配置查询、修改和清结算流水查询的请求平均响应时间小于50ms；
4. 系统稳定性：可承受日常使用负载量，系统运行稳定无问题。

3.4.2 准确性

清结算系统涉及商户和平台各类账户动账、费用计算和余额提现等业务，因此需要满足资金处理的准确性需求。在 Java 开发平台，使用内置的不可变、任意精度的大整数 BigDecimal 进行数值计算，避免float 和 double 精度不够造成的资金损失问题。资金处理准确性不低于99.99%

3.4.3 可扩展性和可维护性

设计阶段应充分调研和评估清结算业务在3至5年内的发展前景，充分考虑系统升级、扩展、修改的可维护性。系统总体架构应该在未来3年内保持相对稳定，同时必须能够适应不断变化的清结算业务需求。

系统可扩展和可维护性主要通过代码质量和代码复杂度判断系统是否容易扩展和维护。代码质量的主要从设计模式、注释质量、异常处理三分方面考虑。在系统设计方面，总体要求按照高内聚低耦合的软件设计思想，采用SOLID（单一职责SPR、开闭原则OCP、里式替换LSP、接口隔离ISP、依赖倒置DIP）软件设计原则，使系统保持高可扩展性；在代码注释方面，要求不做多余重复的注释，代码能够表明的就不用注释，而类似于方法和类的目的和意图之类的无法通过代码说明的，需要进行简要概括性的注释。因此，每个类、接口、方法应该对其主要内容和意图进行注释说明；在异常处理方面，系统首先根据异常层级分类，然后根据类型进行分类，系统内部异常不能暴露给外部用户，即屏蔽内部异常。同时，为了方便开发人员定位错误，内部系统异常按照类型归类，比如按照MQ异常、参数校验异常、数据库异常、文件和网络IO异常分类。

代码复杂度主要从LOC代码行数[43]、McCabe圈复杂度[44]和继承深度来评估。每个类和方法的代码函数分别不应该超过3000行和100行;代码圈复杂度不超过10；类接口继承深度数字越大意味着通过继承重用代码的可能性越大，但也意味着复杂性更高并且代码错误率更高，因此继承深度不超过6。

3.4.4 数据安全性

本系统的数据安全性主要从以下方面衡量。

1. 数据完整性：在业务数据方面，能够对交易系统发送的binlog消息进行业务字段完整性校验；在数据存储方面，能够满足数据库完整性和唯一性约束。

（3）备份和恢复：对商户配置和清结算流水数据进行备份，定时通过网络将数据传送到备用中心；提供本地数据的备份和恢复、按照每天2次进行数据的备份；采用集群部署方式，分散系统压力，避免单点故障，集群机器数不少于3台；备份硬盘至少2块，磁盘空间大小至少在1TB以上。

3.5 本章小结

本章首相介绍了清结算系统和其他支付子系统，概括描述了清结算系统和其他子系统依赖关系。然后对清结算系统的业务流程进行分析，形成最初的业务需求。接着通过用例图和用例规范表简单描述每个用例需求的具体内容和基本流程，确具体的功能性需求。最后描述系统的非功能性需求，形成完整的清结算系统需求。

第4章 系统概要设计

4.1 系统设计目标和方法

清结算系统的主要功能是为商户提供收单服务，即对商户在平台进行的交易金额进行清分结算，在平台收取服务费后把净额结算给商户。整个系统由多个低耦合高内聚的模块构成，同时各个模块相互协调以完成清结算系统任务。为了合理高效的实现该系统，本文采用了如下设计方法：：

（1）系统架构选择

系统的构建采用领域驱动设计中的四层架构和CQRS[45][46]架构结合的方式。对于清分和结算模块，由于业务领域和业务规则复杂，将业务相关逻辑抽象到领域层能够使得系统分层设计符合单一职责原则。同时，将业务变化隔离在领域层，使得其他层的功能相对稳定，不需要经常修改，各层保持高内聚低耦合的特性。对于配置管理和流水查询这些没有复杂业务规则的模块，清结算系统单独使用CQRS架构进行设计，从而简化系统整体的复杂度。这种根据模块自身业务特点，采取不同的架构设计的方式，使系统具有较高的可扩展性。

（2）系统功能设计与实现

清结算系统采用Spring+SpringBoot+MyBatis技术实现系统模块功能。根据业务需求分析，系统分为三个模块，分别是清算模块、结算模块和异常处理模块。

（3）系统测试

系统在开发中，对各个模块功能采取单元测试方法验证函数功能的正确性，在模块开发完成后，采用集成测试对各模块进行联调测试，测试是否符合预期。

4.2 系统静态结构设计

4.2.1 系统架构设计

清结算系统作为整个支付系统的核心子系统，需要与其他子系统的协调配合。清结算系统整体架构图见图4.1。

图4.1 清结算系统架构图

从图中可得，清结算系统需要和交易系统、运营系统账户系统进行交互。清结算系统内部主要分为四个模块：配置和流水管理模块，即图中的商户收费配置、商户结算管理、交易清算管理和结算单管理；清分模块，即清分计费和账户API；结算模块，包含实时清算、周期清算和周期结算子模块；异常处理单独属于一个模块。系统对外通过API层向运营系统提供配置管理和流水查询服务。

4.2.1 系统功能模块设计

清结算系统主要包含五个模块，分别是清算模块、结算模块、异常处理模块、配置模块和流水模块，每个模块下包含具体的功能子模块。清算模块包含清分模块、计费模块和记账模块；结算模块包含实时结算模块和周期结算模块；异常护理模块包含自动处理模块和人工处理模块；配置模块包含配置下发模块和配置查询模块；流水模块包含清算流水查询、结算流水查询和提现流水查询模块。系统功能模块图如图4.2所示。

图4.2 系统功能模块图

4.3 系统动态结构设计

本节通过系统活动图和数据流图描述清结算系统的动态结构。

4.3.1 清结算系统活动图

清结算系统分为清算和结算两大流程，系统整体活动图如图4.3所示。

图4.3 清结算系统活动图

从清结算活动图可以看出，清算活动包含如下步骤：

1. 清分：从交易完成开始，对binlog清分，产生清分流水
2. 计费：对清分实体计算净额和手续费。
3. 汇总：对清分实体进行交易汇总、手续费汇总和净额汇总，即汇总到结算汇总详情单上。在此步骤完成需要创建结算汇总详情单，详情单创建逻辑属于重点部分，同时保持事务完整性约束，最后将清算数据关联到结算部分。
4. 记账：清算最后需要进行逐步动账处理，周期结算模式净额记账将商户待清算记账到商户待结算账户，而手续费记账实际上只需要进行汇总；实时结算模式净额记账资金信息流从商户待清算账户转移到商户现金户，而手续费记账资金信息流从待结算户转移到平台手续费收入户，完成清算记账。
5. 结算活动包含如下步骤：
6. 定时任务：XXL-JOB定时任务启动，首先是创建结算单，然后再扫结算汇总详情单，将多个汇总详情单根据主键绑定一个结算单上。
7. 汇总：将绑定的多个汇总详情单各费项金额累计汇总到结算单中；在汇总时根据结算类型，进行实时结算和周期结算处理，实时结算模式需要进行已处理净额和已处理手续费检查，同时进行记账；周期结算模式需要进行未处理净额和未处理手续费检查，同时进行记账后置处理：最后处理税费和手续费，完成结算流程。

4.3.2 清结算系统数据流图

清结算系统数据流图如图4.4所示。

图4.4 系统数据流图

从数据流图可得，清结算系统有两个数据输入源，分别是结算失败交易topic和原始数据topic。结算失败交易会进行异常处理；原始交易数据开始进行binlog清分流程，产生清分实体对象；计费流程会拉取计费配置对清分实体进行计费，并更新请分表；如果是实时结算模式，会调用账户服务api进行记账；否则，如果是周期结算模式，创建清算单并进行费项累加，随后进行清分记账；定时器会定时创建结算单，并将结算详情单进行绑定和汇总，结算流程首先进行风控审核，然后进行周期结算，最后调用账户服务记账。

4.4 本章小结

本章从静态和动态两方面阐述清结算系统概要设计，系统静态结构方面介绍了清结算整体架构和系统功能模块，并分别以架构图和功能模块图表示；系统动态结构方面介绍了清结算活动流程和清结算数据流向，并分别以活动图和数据流图

第5章 系统详细设计

5.1 详细设计目标

本章重点描述清结算系统各模块核心业务功能实现的数据结构和算法，并通过UML类图、泳道图、时序图进行详细设计。最后，对概要设计部分中的ER数据图进行详细设计。

5.2 清算模块设计

清算模块是清结算系统核心模块之一，其核心子模块包括清分、计费和清算汇总子模块，下面将对核心子模块进行详细设计。

5.2.1 Binlog清分详细设计

在支付系统中，各个子系统相互协作完成商户支付业务，清结算数据源来自交易系统产生的商户交易信息，保存在交易数据库中。在微服务架构模式下，各个业务子系统根据业务属性进行垂直拆分，每个子系统拥有单独的数据库，彼此之间实现数据隔离。在子系统数据相互隔离的前提下，为了实现系统之间的数据共享以及保持系统之间的解耦，一般通过分布式消息组件进行数据通信和共享。

清分模块采用阿里巴巴MySQL binlog 增量订阅和消费组件Canal[46]将交易系统的binlog数据同步到消息中间件RocketMQ 中，然后清结算系统再订阅RocketMQ Topic获取交易数据。交易系统根据业务交易类型将交易数据保存到对应的数据库表中，比如支付表，退款表和拒付表等。因此，清分模块需要按照表的交易类型进行分类处理。同时，清结算接收到的是二进制日志数据，不满足自身业务数据格式需求，需要进行数据格式转换。其次，为了防止对错误数据或者缺失数据进行后续计费逻辑而导致的大量异常重试堆积，需要提前进行数据字段校验。最后，为了保持系统健壮性，需要对异常数据进行重试处理。

根据上述分析，清分模块主要的逻辑分为数据解析和字段校验和异常处理三部分。数据解析部分主要任务是定义Binlog对象，将二进制数据解析为Binlog对象，最后做一些字段缓存；字段校验部分主要内容是匹配表处理器hanler，判断数据是否满足清算条件，最后进行参数校验和构造清分对象。异常处理逻辑包含binlog 解析和校验部分的异常，对于binlog解析异常，清分模块不需要进行重试；对于字段校验异常，首先需要将异常数据落库，其次需要发送到RocketMQ延迟队列进行重试处理。清分模块交互逻辑的泳道图如图5.1所示。

图5.1清分泳道图

在Binlog处理中，需要根据交易类型匹配合适的表处理器handler，然后通过 handler 完成下面的清分条件判断和构造清分实体逻辑，使Blinog处理逻辑集中到对应的handler中，与主流程解耦。通过分析发现，根据类型匹配handler明显可以通过面向对象的多态技术实现条件匹配。因此，这里通过接口和继承实现多态行为，不同交易类型的实现类handler 继承TradeTableHander接口，实现不同的处理逻辑，避免在主流程中进行if 条件判断，违反设计模式中的开闭原则。最后，根据领域驱动设计中的充血模型思想，将Blinog解析的逻辑放在Binlog对象中，调用者BinlogParser只需要进行参数类型转换和异常处理，然后委托给Binlog处理，使得对象不仅仅是数据容器，而是包含具体的业务逻辑的领域实体。Binlog清分模块功能的类图结构如图5.2所示。



图5.2 Binlog 清分类图

5.2.2 计费详细设计

计费模块是清结算系统的核心子模块，它根据商户计费配置计算某笔交易的各种手续费。然而，作为第三方支付平台，所服务的商户数多达几十万、数百万，并提供数十种交易服务，计费服务必须能合理高效的对数量庞大的商户、丰富的交易类型以及不同商户的计费配置进行手续费计算。为此，清结算系统将商户结算配置抽离出来，单独配置和保存，整个计费流程根据商户的计费配置进行处理。为了解决不同交易类型，计费逻辑不同的问题，计费模块通过接口和抽象类和子类继承的方式，使得每种交易类型对应一种计费服务。计费模块在代码设计上，集中体现了领域驱动设计的思想，通过对计费业务的分析，设计出计费领域模型，通过统一语言使业务分析和软件设计紧密结合，有效促进了团队沟通和软件开发效率。清分计费服务ClearCalculateService 是上层清分模块调用下层计费模块的入口，负责封装计费参数TradeInfo和计费后置处理逻辑，其类图结构如图5.3所示。



图5.3 清分计费服务类图

ClearCalculateService 作为清分计费统一接口，定义了上层清肺计费操作calculate 方法签名。为了保持通用，calculate 入参为泛型，返回结果为值对象ClearResult。抽象类AbstractCearCalculateService实现该接口，并添加参数预校验方法preCheck和计费方法doCalculate，分别负责判断入参是否满足计费条件和处理具体的计费逻辑。子类继承该抽象类，覆盖doCalculate方法，添加计费逻辑。为了简化计费流程，calculate方法采用设计模式的模版方法，统一了计费的标准流程。具体流程为：先参数校验，校验成功后再计费，否则返回失败的ClearResult对象。PayCearCalculateService和RefundCearCalculateService分别为支付计费服务和退款计费服务，内部将计费服务委托给ChargeCalculatorService这个实际的收费服务，得到计费列表后再更新清分实体。这样处理的原因是划分收费服务边界上下文，使得收费服务保持独立，与清分上下文解耦，使得模块间的职责边界更加清晰，符合高内聚低耦合的编码原则。ChargeCalculatorService分别负责支付和退款交易的收费服务，对于支付交易，需要计算交易手续费、分期费、税费和净额；对于退款交易，需要计算退款手续费、退分期费、退税费和退净额。支付类型交易的清分计费整体交互时顺序如图5.4所示。



图5.4 清分计费时序图

从上图可知，PayCearCalculateService主要任务是组转计费参数，即根据清分服务传递的清分实体ClearingBillEntity构造收费计算服务ChargeCalculatorService需要的入参TradeInfo，然后调用ChargeCalculatorService进行计费；支付计费的逻辑分为两部分，前置计费和后置计费。前置计费即交易手续费计费，需要先计算，后置费项在手续费基础上计算。ChargeCalculatorService首先从仓库ChargeCalculatorRepository中获取费用计算器模型ChargeCalculator，然后把计算交易手续费的任务委托给计算器模型ChargeCalculator，最后再计算其他费项。下面首先介绍计费模型的类图和计费仓库获取计费模型的逻辑，最后再介绍计费模型中的计费逻辑。

费用计算器模型ChargeCalculator计算交易手续费是计费模块的热点代码，需要处理大批量计费请求。同时，计费模型需要根据商户配置中的费率和计算方式计算，即通过配置转换成相应的计费实体。对此，根据领域驱动设计思想，设计计费模型、计费仓库和计费服务。费用计算器ChargeCalculator类图如图5.5所示。

图5.5 费用计算器类图

费用计算器ChargeCalculator表示对某个商户的某笔交易收费的计算模型，其中包含一个分组的子项计算器SubitemComputer，表示一笔交易对应的多个费项的计算模型，即根据计费配置对商户一笔交易收取多种费用。其中的calculate方法被ChargeCalculatorService调用，接受计费参数对象TradeInfo，返回一个费项计费列表。每个费项列表值对象FeeCalItem代表某种费项计算结果，由费项配置FeeConf、该费项金额feeAount和费项币种组成。费项配置值对象FeeConf由该费项编码feeCode、商户配置ID、configId 和费项币种组成，唯一标识某个费项种类。费项计算器由三部分组成，分别是费项配置FeeConf、计费匹配器TradeMatcher和计算器IComputer。计费配置器中的match调用计费匹配策略IMatcherPolicyUtil，根据某种策略匹配当前交易计费参数，如产品类型、交易类型、交易时间、币种等是否满足商户配置数据，匹配成功才进行计算。计算器根据配置中的费基、费率、最小费用和最大费用计算费用金额，而具体的计算方式由子类决定。一元一次计算器BinaryLinarComputer 按照 a + bx 方式计费，a表示费基，b表示费率，x表示支付金额，计费结果满足最小和最大费用限制。

在领域驱动设计中，仓库给调用实体提供了一个内存对象的抽象视图，实体可以直接从仓库中获取目标对象，而不需要关心目标对象的创建逻辑和底层数据库存储和映射相关的细节，保证了领域逻辑的独立性。费用计算器仓储ChargeCalculatorRepository首先根据商户id查内存缓存，缓存在第一次查询商户所有收费配置时构建，缓存中保存商户计费模型ChargeCalculator，如果缓存命中，直接返回商户计费模型。缓存过期时间设置为1分钟，如果缓存未命中则通过查询商户计费配置，获取商户所有的配置对象ChargeConfigEntity，然后根据配置构造子项计算器SubitemComputer，再把子项计算器添加到计费模型中，最后更新缓存并返回ChargeCalculator计费模型。ChargeCalculator在添加子项计算器时根据GroupKey进行分组，把相同分组键GroupKey的子项计算器添加到计费实体map中。收费计算器仓库ChargeCalculatorRepository获取计费模型ChargeCalculator的活动图如图5.5所示。

图5.5 获取ChargeCalculator活动图

ChargeCalculatorService在通过仓库获取计费模型后，就调用计费模型的计费方法获取交易手续费，即计费模型只负责计算交易手续费，ChargeCalculatorService负责计算其他后置费项。计费模型ChargeCalculator计费流程为：

1. 根据入参获取分组Key
2. 从map中获取该笔交易的所有子项计费器，因为一个商户可能配置了多种交易手续费，每种手续费计算对应一个子项计费器SubitemComputer
3. 判断计费参数是否存在子项计费器
4. 子项计费器列表首先匹配计费参数，即如果计费参数命中多个计费器，多个计费器单独计费，SubitemComputer match方法将匹配逻辑委托给SubitemComputer中的匹配器TradeMatch，TradeMatch再根据匹配策略IMatcherPolicyUtil接口中定义的策略进行匹配，如果匹配成功，然后charge方法将计费方式委托给SubitemComputer中的IFeeComputer计算器，TradeMatch和FeeComputer在构造SubitemComputer时初始化，最后完成计费。
5. 验证计费结果，查看是否存在重复计费

计费领域模型ChargeCalculator计费的核心思想是将子项匹配和计算逻辑全部委托给子项计费器，自己只负责流程编排。其伪代码如下：

|  |  |
| --- | --- |
| 1 | List<FeeCalItem> calculate(TradeInfo info) { |
| 2 | GroupKey key ← new GroupKey(info) |
| 3 | List<SubitemComputer> computers ← map.get(key) |
| 4 | if computers == null then |
| 5 | return new ArrayList<>() |
| 6 | List<FeeCalItem> feeList ← computers.filter(e->e.match(info)).map(c -> c.charge(info)) |
| 7 | feeList.groupBy(e -> e.config().feeCode()).forEach((k,v) -> { |
| 8 | if v.size() > 1 then |
| 9 | throw RuntimeException(“重复计费: ” + k) |
| 10 | } |
| 11 | return feeList |
| 12 | } |

根据概要设计中的数据流图，计费是按逐笔进行计费的，一笔交易的计算的结果是按照费项粒度保存在清分表中。比如，商户支付了100元的交易金额，平台根据配置需要收交易手续费、分期费、税费，并计算净额。商户一笔交易的收费示意图如图5.6所示。

图5.6 收费示意图

第三方支付平台为商户提供收单服务，通过清结算系统计算各种手续费，然后将手续费收入计入平台手续费收入户，将净额计入平台商户待结算户，然后通过结算流程按结算配置结算给商户。

5.2.3 清算汇总详细设计

计费模块逐笔计费，将商户各类交易的费项结果保存在清分表中，即清分表唯一键为商户和交易类型，但是结算时间一般以天或周为级别，比如按工作日、自然日、单周、或双周结算。为了满足商户不同的业务需求和现金流管理策略，第三方支付平台为商户提供多种结算模式，在商户入网时签约某种结算模式，然后平台在运营系统的配置下发给清结算系统。所以，在清分模块需要汇总商户一天内同类型交易的费项，并保存到结算详情表中。

汇总流程主要分为两大部分，第一部分是创建用于汇总费项的结算明细单号DetailId，第二部分为费项汇总部分。清分汇总流程状态图如图5.7所示。

图5.7 清分汇总状态图

1. 首先，汇总流程开启Spring事务，保证整个操作流程满足事务ACID属性；
2. 校验清分实体ClearingBillEntity状态是否符合汇总前置状态；
3. 根据商户id、产品类型productType、交易类型tradeType和交易完成时间tradeFinishTime，调用结算服务settleService的createDetailId方法创建汇总详情单，并返回结算明细单号DetailId；
4. 为了避免商户同一笔交易重复汇总，根据商户交易号tradeId和交易类型tradeType查询清分索引表去重；
5. 如果去重校验通过，开启一个新的Spring嵌套子事务，进行费项累计；首先创建内存结算详情实体SettleDetailEntity，然后然后根据结算模式将该笔交易计费后的各类费项结果保存到实体中的对应字段中；对于实时结算模式，因为实时结算已经调用账户服务记账了，所以各费项分别累计到已处理交易手续费、已处理分期手续费、已处理税费和已处理净额字段中；同样，对于周期结算模式，因为需要延期结算，为了和实时结算模式区分，费项需要分别累积未处理交易手续费、未处理分期手续费、未处理税费和未处理净额字段；最后到通过MyBatis update进行各费项累计并更新实体到数据库结算详情表；
6. 为了保存费项累计记录，需要为各类费项创建清分索引，并更具交易类型分别创建入款交易费项索引和出款交易费项索引，以保存交易单号、交易类型、费项编码、商户号、结算详情单号等信息，保证（2）中重复交易不会重复累计费项，同时关联了结算详情单号，为后续商户对账提供数据查询支持。

汇总流程中，核心内容是创建汇总详情单，下面将逐一介绍汇总模型类图和汇总时序图，汇总类图如图5.8所示。

图5.8 汇总模型类图

SummaryModel为领域模型中的聚合根，SettleConfig为实体，SummaryId、SummaryInfo和SummaryStarted为值对象。下面分别分析每个类的主要职责。

1. SummaryModel为汇总领域模型，包含商户号userId、当前商户所有激活配置configs和最近三天内所有汇总详情单summaries。在构造SummaryModel时，通过查询数据库获取结算配置SettleConfigEntity列表和结算详情实体SettleDetailEntity列表，再转换成领域实体SettleConfig、值对象SummaryId和值对象SummaryInfo，完成成员变量configs和summaries初始化。
2. SettleConfig实体是领域层结算配置对象，反映领域专家的关注点而不是数据库实体。SettleConfig主要职责是处理汇总和结算服务所需的时区转换问题。在构造SettleConfig时，初始化其成员变量结算零点SETTLE\_ZERO为"2022-01-01T00:00:00Z"时刻；同时，初始化汇总模式SUMMARY\_MODE为ChronoUnit.DAYS；最后，保存对数据库清分对象的引用setting，方便后续获取相关配置数据。

（3）SummaryId是查询商户汇总详情单的唯一键值对象，包含用户产品类型、用户交易类型和汇总时间。其中，汇总时间是领域模型创建汇总详情单DetailId时的重难点。由于时间处理和结算配置紧密相关且封装在SettleConfig中，所以领域模型在创建DetailId时是需要在getDetail方法中匹配到某个配置，即getConfig方法，然后再委托给SettleConfig处理时间，最后在自己缓存的summaries中查找返回汇总信息。如果SummaryInfo不为空，说明交易时间在当前汇总时间范围内；如果为空，则调用领域事件处理器创建汇总详情单，并返回DetailId，创建汇总单号DetailId的时序图如图5.9所示。

图5.9 创建汇总单号时序图

清算汇总首先需要获取结算详情单号DetailId，即通过结算服务SettleService获取。结算服务根据领域驱动设计，自身不负责具体逻辑处理，而是将获取DetailId封装在汇总模型中，通过构造SummaryModel获取DetailId。

1. 结算服务通过结算配置仓库SettleConfigRepo获取汇总模型，为结算服务提供获取领域模型的内存视图，封装和屏蔽了构造汇总模型的底层细节。
2. 结算配置仓库通过配置维护服务configMaintainService查询当前商户所有生效的结算配置列表configs，并构造配置映射configMap，key为configId，value为领域值对象SettleConfig。
3. 结算配置仓库通过结算详情Mapper查询最近3天内商户所有的结算详情单，并将数据库结算详情实体SettleDetialEntity转换为领域值对象SummaryId和SummaryInfo，最后保存在结算模型中。
4. 结算服务根据产品类型productType和当前时间 currentDate获取最近一批结算详情单创建的开始时间recentStartAt。根据业务需求，最近控制时间段的开始时间为前3天当天0点。领域模型保存了当前商户所有的结算配置configs，然后根据productType和currentDate匹配某个结算config，匹配规则为产品类型相等并且配置激活时间不大于当前时间；匹配到某个结算配置config后，通过getSummaryTime方法判断汇总模式是否是以天为单位，如果通过再进行时区转换，并将开始时间设置为前三天零时。
5. 如果商户交易时间transTime小于recenttStartAt，说明该笔交易是三天前的历史数据，即人工补录的交易数据，需要调整该笔交易的时间为当前时间currentDate，进入当前汇总批次。
6. 结算模型更具唯一键（产品类型，交易类型，交易时间）查询summaries获取SummaryInfo；在匹配到某个结算配置config后，调用SettleConfig的getSummaryTime方法计算summaryTime，即将入参transTime截取为当日零点。结算模型得到截取后的交易时间后，构造唯一键SummaryId，然后查询summaries Map获取SummaryInfo，然后返回。
7. 结算服务通过领域模型的hasAttach2BillAndSettled方法判断汇总详情单已结算，即info!=null && info.state() == 1。如果汇总详情单已结算，则将交易时间设置为当前时间，汇总到最新的详情单中；然后再次调用领域模型的getDetailId方法获取最新的SummaryInfo。如此此时的汇总详情单也已经结算，则报出异常信息。
8. 如果最新的汇总信息SummaryInfo不为空且属性汇总详情单detailId不为空，则创建DetailId对象并返回。
9. 如果SummaryInfo为空，说明当前汇总详情单还没有创建。因此，通过汇总模型的summaryInitialized方法初始化汇总开始领域事件SummaryStarted；其属性汇总单号detailId通过UUID工具创建唯一单号，然后通过领域事件SettleEventHandler创建汇总详情单。
10. 在清算汇总完成后，需要以天为单位汇总清算详情单，即将商户几天内的汇总详情单累计到结算单中。结算单每个条目代表某商户过去几天所有交易类型的各费项累计之和。

5.3 结算模块设计

结算流程需要对清算流程在几天内生产的清算汇总详情单（结算详情单）进行汇总累计，对某个商户创建结算单，进入结算流程，将累计交易净额结算给商户。

5.3.1 结算单创建

XXL-JOB定时任务触发创建结算单创建流程，将某个商户结算周期内所有结算详情单绑定并汇总到该结算单中，结算单创建活动图如图5.10 所示。

图5.10 结算单创建活动图

结算单创建活动图具体内容分析如下：

1. XXL-JOB每小时触发结算单创建任务。
2. 通过结算配置查找所有激活商户。
3. XXL-JOB分片，通过商户号与集群机器数取余获取当前商户。
4. 查找当前商户所有的结算配置，通过多线程方式，为商户每个结算配置创建一个结算单。
5. 首先获取结算结算领域模型BillSettleModel，结算模型根据产品类型和交易时间匹配某个结算配置SettleConfig对象，通过结算配置对象将交易时间转换为结算时间。
6. 领域模型通过SettleKey在结算单map中查询结算单信息SettleBillInfo对象查看是否存在SettleId；如果未找到结算单info对象，构造结算单初始化事件SettleBillInit。
7. 在构造SettleBillInit中判断是否符合结算单创建规则：不能同时存在两个结算单，且上一个结算单状态必须处于资金处理状态。

账单结算模型BillSettleModel类图设计如图5.11所示。

图5.11 账单结算模型BillSettleModel类图

类似于清算汇总模型，结算单模型包含三个成员变量，分别是商户号、当前商户所有结算配置和当前商户所有未结算完的结算单。getSettleId方法调用getMatch方法匹配某个结算配置config对象，然后调用config 的getSettleTime将交易时间转换为结算时间；然后，结算模型再通过构造结算唯一键SettleKey在bills中查找SettleBillInfo判断是否已经创建了结算单；否则，构造结算单初始化事件SettleBillInit对象；在构造中，判断上一个结算的状态，查看是否满足创建规则；通过config对象的getSettleTime、getLiquidEndTime等方法计算结算时间和清算结束时间；通过UUID[48]工具类生产结算单号setttleId，然后通过结算时间处理器创建结算单。

在创建结算单流程中，核心是将交易时间转化为结算时间，即getSettleTime函数。下面将通过伪代码分析该函数内部逻辑：

输入：交易时间transTime

输出：结算时间setttleTime

|  |  |
| --- | --- |
| 1 | // 获取结算周期：实时结算按自然日汇总并延期一天结算；  周期结算按自然日汇总并按工作日延期  SettleCycleEnum cycle = getSettleCyle() |
| 2 | // 转换时区  ZonedDateTime startTime = transTime.toInstant().atZone(config.getZoneId()).turncateTo(Days) |
| 3 | // 根据汇总模式计算清算汇总结束时间  swich cycle.getSummaryMode() |
| 4 | case DAY: liquidEndTime = extendEndTime(startTime, DAYS, 1, cycle.getDelayMode()) |
| 5 | case WEEK: liquidEndTime = extendEndTime(startTime, WEEKS, 1, cycle.getDelayMode()) |
| 6 | case BIWEEK: |
| 7 | startTime = startTime.with(TemporalAdjusters.previousOrSame(DayOfWeek.MONDAY)) |
| 8 | weeks = WEEKS.between(startTime, SETTLE\_ZERO.atZone(setting.getZoneId())) |
| 9 | startTime = startTime.minusWeeks(weeks % 2) |
| 10 | liquidEndTime = extendEndTime(startTime, WEEKS, 2, cycle.getDelayCalcMode()) |
| 11 | case MONTH: |
| 12 | startTime = startTime.with(TemporalAdjusters.firstDayOfMonth()) |
| 13 | return extendEndTime(startTime, ChronoUnit.MONTHS, 1, cycle.getDelayCalcMode()) |
| 14 | default: 抛出结算周期异常 |
| 15 | // 根据结算延期模式，计算结算时间  switch cycle.getDelayCalcMode() |
| 16 | case WORKDAY: |
| 17 | /// 结算与指定延期工作日天数结束  while true then |
| 18 | if !isHoliday(liquidEndTime) then count++ |
| 19 | if count >= cycle.getCount() then 返回 Date.from(liquidEndTime.toInstant()) |
| 20 | liquidEndTime = liquidEndTime.plusDays(1) |
| 21 | case NATUAL\_DAY: 返回Date.from(liquidEndTime.plusDays(cycle.getCount() - 1).toInstant()) |
| 22 | default: 抛出延期计算模式异常 |

extendEndTime将交易时间transTime向后延期来计算清算汇总结束时间，这段延期时间就是结算单汇总的详情单的时间范围。extendEndTime伪代码如下：

输入：开始时间startTime、时间单位unit、延期天数count、延期模式delayCalMode

输出：延期后的结算时间

|  |  |
| --- | --- |
| 1 | // 正常情况向后延期 count 个 unit，但是在工作日模式下，如果count个unit全是假期，则继续向后顺延 count 个 unit日期 |
| 2 | ZonedDateTime endTime = startTime.plus(count, unit) |
| 3 | switch delayCalMode |
| 4 | case WORKDAY |
| 5 | while isAllHolidayForward(endTime, unit, count) == true |
| 6 | endTime = endTime.plus(count, unit) |
| 7 | return Date.from(endTime.toInstant()) |
| 8 | // 自然日模式  case NATUAL\_DAY: return Date.from(endTime.toInstant()); |
| 9 | default 抛出异常 |

通过上述方式给商户创建结算单，就可以实现按照商户结算配置中的结算模式和对应的延期天数汇总这段时间范围内的结算详情单。

5.3.2 绑定结算详情单

创建完结算单后，需要将汇总时间段内的所有结算详情单绑定到刚刚创建的结算单上，方便后续汇总轧差，绑定结算详情单活动图如图5.12所示。

图5.12 绑定结算详情单活动图

结算详情单绑定活动图详细步骤分析如下：

1. XXL-JOB每小时出发结算详情单绑定任务。
2. 通过ConfigMaintainService查询所有激活商户。
3. 获取任务分片索引和分片总数。
4. 通过商户号与分片总数求余确定当前商户，并通过SettleBillMapper查询商户所有未结算的结算单。
5. 通过线程池执为每个结算单执行绑定任务；首先通过SettleQueryRepo获取结算单模型BillModel，具体流程如下 ：
6. 通过SettleBillMapper查询结算单实体SettleBillEntity
7. 通过SettleDetailMapper查询已绑定的结算详情单列表details
8. 构造BillModel
9. 通过SettleQueryRepo查询未绑定的结算详情单列表（settleId为空）
10. 过滤未绑定的详情单：只保留满足详情单summaryTime < 结算单汇总结束时间条件的详情单
11. 判断过滤后的详情单列表是否不为空，如果不为空，结算单绑定模型创建结算单绑定开始事件SettleBindStarted，创建流程如下：
12. 首先判断当前结算单状态是否合法：必须为初始状态或者正在绑定状态，即SettleStatusEnum.INIT和SettleStatusEnum.BINDING
13. 判断过滤后的未绑定的详情单列表detailInfoList状态是否合法：所有详情单汇总状态必须为正在绑定状态SummaryStateEnum.BINDING，如果有任何一个结算单状态为清算状态SummaryStateEnum.CLEARING，说明存在重复绑定详情单，抛出异常；否则，构造SettleBindStarted。
14. 绑定汇总详情单：结算事件处理器SettleEventHandler通过SettleBillMapper将结算单乐观锁版本号自增1；通过SettleDetailMapper将结算详情单中的settleId设置为当前结算单单号；
15. 同步更新内存对象状态：更新模型中结算单状态为SettleStatusEnum.BINDING；结算单版本号自增1；更新模型中绑定的结算详情单状态为SummaryStateEnum.BINDING。

结算账单模型BillModel 类图如图5.13所示：

图5.13 结算账单模型BillModel类图

结算单绑定模型包含两个字段，分别是数据库结算单实体SettleBillEntity和领域层结算详情值对象DetailInfo。SettleBillEntity表示结算单，主要包含结算费项、结算时间相关字段；DetailInfo表示已经绑定到结算单的详情单，主要包含领域层关心的字段，如汇总时间、结算详情单单号，结算详情单状态等。一个结算单在清算汇总期间可能绑定多批汇总详情单。因此，每次绑定都需要从数据库获取最新的结算单和已绑定的结算详情单，并保存在BillModel模型中。BillModel主要的职责是处理结算流程相关逻辑，如结算单绑定、风控处理、累计费项、结算费项等。

5.3.3 结算流程

XXL-JOB每小时扫描结算表，对满足结算条件的结算单进行结算处理流程,结算流程活动图如图5.14所示。

图5.14 结算流程活动图

结算流程详细步骤分析如下：

1. XXL-JOB定时器每小时触发结算任务，发起结算流程。
2. 通过ConfigMaintainService查询配置获取所有激活商户。
3. 获取job分片索引和分片总数。
4. 通过商户号与分片总数求余，如果结果等于分片索引，说明当前商户被选择处理。
5. 通过SettleBillMapper获取商户待结算的结算单，查询条件为：当前时间大于汇总结束时间liquidEndTime或结算开始时间settleStartAt，并且结算单状态为未结算。
6. 配置结算任务线程池，提高并发处理能力。
7. 通过SettleQueryRepo获取结算模型BillModel，将结算实体SettleBillEntity和已绑定的汇总详情单列表details保存在BillModel中。
8. 查看结算单状态是否满足结算条件，如果满足，进入结算流程
9. 初始化结算流程，通过职责链模式将结算流程分为init、bind、riskManage、clear、tradeFee、installmentFee、taxFee、netFee和complete步骤。
10. init处理：如果结算单状态为init状态，此时刚刚创建结算单，还没有绑定汇总详情单，直接到达清算结束状态。
11. accumulate处理：首先，前置状态判断，只有结算单状态是为正在绑定详情单BINDING才能继续处理；然后，判断是否需要风控处理，如果此时采取强制结算或者结算当前时间已经超过汇总结束时间3小时（防止交易信息堆积），继续处理。否则抛出异常，终止结算流程；然后，判断结算单是否需要进行风控处理，如果结算策略不为强制结算且汇总详情单包含支付交易且商户结算配置中风控策略不为商户白名单策略，将结算单状态更新为下一个流转状态，即风控中；否则，将状态更新为清算状态，停止绑定详情单。
12. 风控拦截处理：调用风控接口对包含支付类交易的结算详情单进行处理，风控最大等待时间为10小时。风控拦截状态不可进行清算，直接返回异常，结束结算流程，等待重新发起。风控通过后，将结算单和绑定的详情单状态更新为下一个流转状态，即清算状态。
13. clearing 清算处理：首先通过SettleQueryRepo查询最新的结算单已绑定的详情单列表detailInfoList，如果该列表和BillModel中的details大小不相等，说明将要清算的详情单数据错误；在（11）中已经将结算单状态更新，不能再继续绑定详情单。判断最新detailInfoList是否包含details的所有详情单号，如果不包含，说明最新详情单列表在（11）中将状态更新到数据库这段时间内已经绑定了其他的详情单，此时抛出异常，中断结算流程；否则，累加详情单各费项，并进行交易净额轧差汇总；通过SettleQueryRepo获取最新的BillCtrModel，判断是结算单结算总额是否达商户结算配置中的最低结算金额，如果未达到，更新结算单状态为BINDING，继续绑定详情单；否则，最后通过SettleEventHandler更新结算单状态为下一个流转态，即等待结算交易手续费WAITING\_SETTLE\_TRADE\_FEE。
14. setttle处理：首先结算交易手续费，判断结算单状态和待结算手续费是否满足结算手续费前提条件；通过后调用账户接口进行记账操作完成手续费结算，并更新结算单状态为下一个流转态WAITING\_SETTLE\_INSTALLMENT\_FEE；同理，按照相同步骤依次调用账户接口完成分期费、税费和净额的结算流程；其中，净额结算有两种结算策略，要么结算到卡，要么结算到账户，需要根据结算配置处理。
15. 在（8）中，如果判断未达到结算条件，则绑定详情单并落库。

结算流程通过职责链设计模式实现，结算流程职责链类图如图5.15所示。

图5.15 结算职责链类图

结算流程接通过setNext方法把各个子类链接起来，形成处理器的单链表，每个流程依次处理，处理完后调用next后继的处理方法handle，完成结算流程。

5.4 异常模块设计

对异常进行兜底设计是保障清结算系统稳定的重要方式，及时准确的处理各类异常事件，能有有效提升平台服务水平和质量。异常模块负责处理清算模块和结算模块中发生的异常事件，通过领域驱动设计中的事件模型和状态机，对异常进行重试或人工处理。异常模块类图结构设计如图5.16所示。

图5.16 异常处理类图

异常模型StuckFlowModel包含数据库异常实体StuckFlowEntity和最大重试次数MAX\_RETRY\_COUNT；自动重试方法为retry，委托给网关对象参数实现；人工处理异常包括三个方法：手动触发异常triggerRetry、人工修复数据modifyFlowDate和人工标记忽略maskAsIgnored。异常重试通过状态机进行处理，共有7个状态，分别是Receving接受态、AutoRetrying自动重试中、RetryFailed重试失败、WaitingManualProcess等待人工处理、ManualChageData人工修复数据、ManualMarked人工标记忽略和Completed完成。异常重试状态机如图5.17所示。

图5.17 异常重试状态机

5.5 数据库设计

根据本章前面几节的详细设计内容，本节将选取与系统核心功能有关的数据关系，做出如下的数据库逻辑结构设计。

5.5.1 清算数据模型

清算数据模型：清分实体、清分索引实体、计费配置、结算详情实体E-R关系图如图5.18所示。

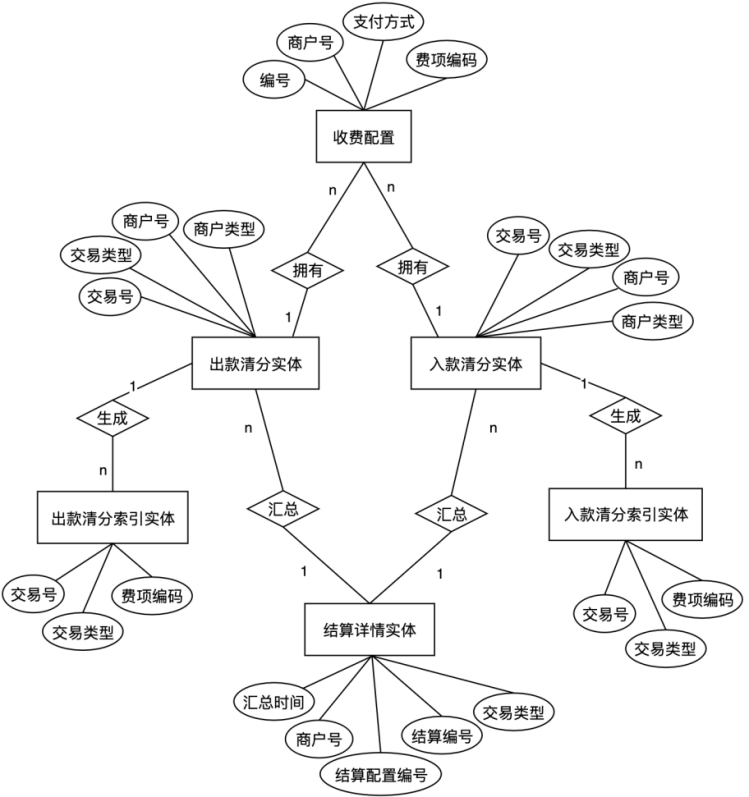


图5.18 结算详情E-R图

根据数据模型，可以对清算模块的相关数据表单进行如下设计：

（1）计费配置表表，存储商户计费相关的费项配置。其中主要属性包括ID、商户编号、支付方式、卡组、分期数、费项编码、费率、费基、最大手续费、最低手续费，详细说明如表5.1所示。

表5.1 计费配置表

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 字段意义 | 数据类型 | 说明 |
| id | id | bigint unsigned | 主键 |
| country\_code | 国家码 | varchar (10) |  |
| config\_id | 配置编号 | varchar(64) |  |
| user\_id | 商户编号mid | bigint unsigned |  |
| user\_type | 商户类别 | tinyint unsigned |  |
| user\_product | 商户产品类型 | tinyint unsigned |  |
| user\_trade\_type | 交易类型 | tinyint unsigned |  |
| pay \_type | 支付方式 | tinyint unsigned |  |
| card\_group | 卡组 | tinyint unsigned |  |
| installment\_tiers | 分期数 | tinyint unsigned |  |
| fee\_code | 费项编码 | tinyint unsigned |  |
| currency | 币种 | varchar(32) |  |
| fee\_base | 费基 | int unsigned |  |
| fee\_rate | 费率 | varchar(10) | 1.234% |
| fee\_min | 最低手续费 | int unsigned |  |
| fee\_max | 最高手续费 | int unsigned |  |
| activation\_time | 激活时间 | datatime | 北京时间 |
| expiration\_time | 失效时间 | datetime | 北京时间 |
| effective | 是否是最新生效 | tinyint unsigned | 0/1 |
| archive | 归档状态 | tinyint unsigned | 0/1 |
| version | 版本号 | int | 乐观锁 |
| create\_time | 创建时间 | timestamp |  |
| modify\_time | 修改时间 | timestamp |  |

1. 清分表，主要存储商户、交易、费项结算结构相关信息。其中主要属性包括ID、配置ID、交易号、订单号、商户号、交易类型、结算模式、支付类型、结算时间、国家码、币种、交易额、手续费、分期费、税费、净额等。详细说明如表5.2所示（费项部分只展示交易手续费，其余费项字段类型）。

表5.2 清分表

| 字段名 | 字段意义 | 数据类型 | 说明 |
| --- | --- | --- | --- |
| id | id | bigint unsigned | 主键 |
| settle\_id | 结算单号 | varchar(64) |  |
| trade\_id | 交易号 | varchar(64) |  |
| origin\_trade\_id | 原交易号 | varchar(64) | 退款类交易 |
| order\_id | 订单号 | varchar(64) |  |
| origin\_order\_id | 原订单号 | varchar(64) | 退款类交易 |
| user\_id | 商户号 | bigint unsigned |  |
| user\_type | 商户类型 | tinyint unsigned | 个人、商户、内部 |
| trade\_type | 交易类型 | tinyint unsigned | 支付、退款、拒付 |
| origin\_trade\_type | 原交易类型 | tinyint unsigned | 退款类交易 |
| settle\_mode | 结算模式 | tinyint unsigned | 周期、实时结算 |
| product\_type | 产品类型 | tinyint unsigned | 收单、代付 |
| pay\_type | 支付类型 | tinyint unsigned | 卡、余额、分期等 |
| card\_group | 卡组 | tinyint unsigned | visa、mastercard等 |
| installment\_tiers | 分期数 | tinyint unsigned | 3、6、12、24等 |
| current\_tiers | 当前分期数 | tinyint unsigned |  |
| settle\_time | 结算时间 | datetime |  |
| country\_code | 国家码 | varchar(20) | CN、MEX、BRA |
| mcc | 商户类别 | varchar(20) |  |
| currency | 币种 | varchar(8) | CNY、USD |
| amount | 交易金额 | bigint | 单位为分 |
| origin\_amount | 原交易金额 | bigint | 单位为分 |
| net\_amount | 净额 | bigint | 单位为分 |
| net\_settle\_id | 净额结算单号 | varchar(64) |  |
| net\_settle\_detail\_id | 净额结算明细单号 | varchar(64) |  |
| net\_settle\_cycle | 净额结算周期 | int unsigned |  |
| trade\_fee | 交易手续费 | bigint | 分 |
| trade\_fee\_config\_id | 手续费计费配置号 | varchar(64) |  |
| trade\_fee\_collecting\_type | 手续费收费方式 | tinyint unsigned | 内扣、外扣 |
| trade\_fee\_deduct\_type | 外扣方式 | tinyint unsigned | 同步、异步 |
| trade\_fee\_account\_type | 出金账户类型 | int unsigned | 外扣出金账户 |
| trade\_fee\_settle\_id | 手续费结算单号 | varchar(64) |  |
| trade\_fee\_settle\_detail\_id | 手续费结算明细单号 | varchar(64) |  |
| trade\_fee\_settle\_cycle | 手续费结算周期 | int unsigned |  |
| status | 清算状态 | tinyint unsigned |  |
| risk\_state | 风控状态 | tinyint unsigned |  |
| remark | 备注 | varchar(1024) |  |
| payer\_id | 付款用户号 | bigint |  |
| payer\_type | 付款用户类型 | tinyint unsigned |  |
| payee\_id | 收款用户号 | bigint |  |
| payee\_type | 收款用户类型 | tinyint unsigned |  |
| version | 版本号 | bigint unsigned | 数据库乐观锁 |
| create\_time | 创建时间 | timestamp | 自动插入 |
| modify\_time | 修改时间 | timestamp | 自动更新 |

（3）清分索引实体表，主要费项相关的实体数据。其中主要属性包括ID、交易号、交易类型、费项编码、结算单号、结算详情单号。详细说明如表5.3所示。

表5.3 清分索引实体表

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 字段意义 | 数据类型 | 说明 |
| ID | id | bigint unsigned | 主键 |
| trade\_id | 交易号 | varchar(64) |  |
| trade\_type | 交易类型 | tinyint unsigned |  |
| risk\_state | 风控状态 | tinyint unsigned |  |
| fee\_code | 费项编码 | tinyint unsigned |  |
| user\_id | 商户编号 | bigint unsigned |  |
| user\_type | 商户类型 | tinyint unsigned | 1(个人)、2(商户)、3(内部） |
| settle\_id | 结算单号 | varchar(64) |  |
| settle\_detail\_id | 结算明细单号 | varchar(64) |  |
| create\_time | 创建时间 | timestamp |  |
| modify\_time | 修改时间 | timestamp |  |

表约束为：unique key uniq\_trade\_id\_trade\_type\_fee\_code 和key idx\_create\_time

1. 结算详情表，包含商户费项结算相关数据。其中主要属性包括ID、。详细说明如表5.4所示。

表5.4 结算详情表

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 字段意义 | 数据类型 | 说明 |
| ID | id | bigint unsigned | 主键 |
| country\_code | 国家码 | varchar(10) |  |
| user\_id | 商户号 | bigint unsigned |  |
| user\_type | 商户类型 | tinyint unsigned |  |
| config\_id | 结算配置号 | varchar(64) |  |
| settle\_id | 结算单号 | varchar(64) |  |
| settle\_mode | 结算模式 | int unsigned | 实时、周期 |
| settle\_type | 结算类型 | tinyint unsigned | 净额、手续费 |
| settle\_cycle | 结算周期 | int unsigned |  |
| settle\_time | 结算时间 | datetime |  |
| total\_amount | 总交易金额 | bigint | 单位为分 |
| total\_count | 总交易笔数 | bigint unsigned |  |
| total\_processed\_trade\_fee | 总已处理交易手续费 | bigint | 单位为分 |
| total\_unprocessed\_trade\_fee | 总未处理交易手续费 | bigint | 单位为分 |
| total\_processed\_installment\_fee | 总已处理分期手续费 | bigint | 单位为分 |
| total\_unprocessed\_installment\_fee | 总未处理分期手续费 | bigint | 单位为分 |
| total\_processed\_tax\_fee | 总已处理税费 | bigint | 单位为分 |
| total\_unprocessed\_tax\_fee | 总未处理税费 | bigint | 单位为分 |
| total\_settled\_net | 总已结算交易净额 | bigint | 单位为分 |
| total\_unsettled\_net | 总未结算交易净额 | bigint | 单位为分 |
| currency | 币种 | varchar(10) | CNY、USD |
| liquid\_start\_at | 清算汇总开始时间 | datetime |  |
| liquid\_end\_at | 清算汇总结束时间 | datetime |  |
| settle\_start\_at | 结算开始时间 | datetime |  |
| settle\_end\_at | 结算结束时间 | datetime |  |
| settle\_status | 结算状态 | tinyint unsigned |  |
| withdraw\_id | 提现单号 | varchar（64） |  |
| withdraw\_start\_at | 提现开始时间 | datetime |  |
| withdraw\_end\_at | 提现结束时间 | datetime |  |
| withdraw\_status | 提现状态 | tinyint unsigned |  |
| remark | 备注 | varchar(512) |  |
| version | 版本号 | bigint unsigned | 乐观锁版本号 |
| create\_time | 创建时间 | datetime |  |
| modify\_time | 修改时间 | datetime |  |

结算详情表表约束为：

|  |
| --- |
| 约束 |
| unique key uniq\_settle\_id (settle\_id) |
| key idx\_settle\_time (settle\_time) |
| key idx\_userid\_settletime\_usertype\_settletype (user\_id, settle\_time,user\_type, settle\_type) |
| key idx\_withdraw\_id (withdraw\_id) |

5.5.2 结算数据模型设计

结算数据模型：结算配置、结算详情，结算单模型E-R模型如图5.19所示：



图5.19 结算单E-R图

根据数据模型，可以对结算模块的相关数据表单进行如下设计：

1. 结算表，主要存储结算相关信息。其中主要属性包括ID、结算单号、结算模式、结算类型、结算周期、结算时间、结算金额、结算手续费、清算时间、结算时间、提现时间等。详细说明如表5.5所示。

表5.5 结算表

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 字段意义 | 数据类型 | 说明 |
| id | id | bigint unsigned | 主键 |
| country\_code | 国家码 | varchar(10) | CN、MEX等 |
| user\_id | 商户编号 | bigint unsigned |  |
| user\_type | 商户类型 | tinyint unsigned |  |
| config\_id | 结算配置号 | varchar(64) |  |
| settle\_id | 结算单号 | varchar(64) |  |
| settle\_mode | 结算模式 | tinyint unsigned | 实时、周期 |
| settle\_type | 结算类型 | tinyint unsigned | 净额、手续费 |
| settle\_cycle | 结算周期 | int unsigned |  |
| settle\_time | 结算时间 | datetime |  |
| total\_amount | 总金额 | bigint | 分 |
| total\_count | 总交易笔数 | bigint |  |
| total\_net\_amount | 总净额 | bigint | 分 |
| total\_net\_count | 总净额笔数 | bigint |  |
| total\_fee\_amount | 总手续费 | bigint | 分 |
| total\_fee\_count | 总手续费笔数 | bigint |  |
| total\_fee1\_amount | 总手续费1 | bigint | 分 |
| total\_fee2\_amount | 总手续费2 | bigint | 分 |
| total\_fee3\_amount | 总手续费3 | bigint | 分 |
| total\_fee4\_amount | 总手续费4 | bigint | 分 |
| total\_fee5\_amount | 总手续费5 | bigint | 分 |
| total\_fee6\_amount | 总手续费6 | bigint | 分 |
| total\_fee7\_amount | 总手续费7 | bigint | 分 |
| total\_fee8\_amount | 总手续费8 | bigint | 分 |
| total\_fee9\_amount | 总手续费9 | bigint | 分 |
| currency | 币种 | varchar(32) |  |
| liquid\_start\_at | 清算开始时间 | datetime |  |
| liquid\_end\_at | 清算结束时间 | datetime |  |
| settle\_start\_at | 结算开始时间 | datetime |  |
| settle\_end\_at | 结算结束时间 | datetime |  |
| settle\_status | 结算状态 | tinyint unsigned |  |
| withdraw\_id | 提现单号 | varchar(64) |  |
| withdraw\_start\_at | 提现开始时间 | datetime |  |
| withdraw\_end\_at | 提现结束时间 | datetime |  |
| withdraw\_status | 提现状态 | tinyint unsigned |  |
| remark | 备注 | varchar(512) |  |
| version | 版本号 | bigint unsigned |  |
| create\_time | 创建时间 | timestamp |  |
| modify\_time | 修改时间 | timestamp |  |

1. 结算配置表，主要存储和结算配置相关信息。其中主要属性包括ID、。详细说明如表5.6所示。

表5.6 结算配置表

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 字段意义 | 数据类型 | 说明 |
| id | id | bigint unsigned | 主键 |
| country\_code | 国家 | varchar(10) | CN、MEX等 |
| user\_id | 商户号 | bigint unsigned |  |
| user\_type | 商户类别 | tinyint unsigned |  |
| config\_id | 结算配置号 | varchar(32) |  |
| net\_settle\_strategy | 净额结算策略 | tinyint unsigned |  |
| settle\_mode | 结算模式 | tinyint unsigned | 周期、实时 |
| risk\_strategy | 风控策略 | tinyint unsigned | 白名单、风控 |
| refund\_strategy | 退款策略 | tinyint unsigned |  |
| refund\_account\_type | 退款出金户 | int |  |
| chargeback\_strategy | 拒付策略 | tinyint unsigned |  |
| chargeback\_user\_id | 拒付转嫁承担用户 | bigint unsigned |  |
| installment\_settle\_strategy | 分期结算策略 | tinyint unsigned |  |
| settle\_cycle | 结算周期 | tinyint unsigned |  |
| min\_settle\_amount | 最低结算金额 | bigint |  |
| currency | 结算币种 | varchar(32) |  |
| disabled | 结算状态 | tinyint unsigned |  |
| activation\_time | 激活时间 | datetime |  |
| expiration\_time | 过期时间 | datetime |  |
| archive | 归档状态 | tinyint unsigned |  |
| remark | 备注 | varchar(512) |  |
| version | 版本号 | int unsigned |  |
| create\_time | 创建时间 | timestamp |  |
| modify\_time | 修改时间 | timestamp |  |

5.5.3 异常处理数据模型设计

异常处理模块E-R关系图如图5.20所示。

图5.20 异常实体E-R图

异常表，主要存储清结算流程中异常相关信息。其中主要属性包括。详细说明如表5.7所示

表5.7 异常表

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 字段意义 | 数据类型 | 说明 |
| ID | id | bigint unsigned | 主键 |
| user\_id | 商户id | bigint unsigned |  |
| stuck\_type | 导致流程卡住错误类型 | tinyint unsigned |  |
| trade\_id | 交易单号 | varchar(64) |  |
| trade\_context | 交易内容 | varchar(1024) |  |
| trade\_finish\_time | 交易完成时间 | datatime |  |
| last\_code | 最近异常码 | int unsigned |  |
| last\_message | 最近异常码内容 | varchar(512) |  |
| retry\_count | 自动重试次数 | tinyint unsigned |  |
| status | 异常处理状态 | tinyint unsigned |  |
| handler | 人工处理操作人 | varchar(64) |  |
| handler\_message | 处理意见 | varchar(100) |  |
| handler\_context | 处理内容 | varchar(500) |  |
| remark | 备注 | varchar(256) |  |
| archived | 是否归档 | tinyint unsigned |  |
| version | 版本号 | bigint unsigned |  |
| create\_time | 创建时间 | timestamp |  |
| modify\_time | 更新时间 | timestamp |  |

5.6 本章小结

本章在论文前几章的基础上，对清算、结算和异常处理核心功能模块进行了详细设计，从动态与静态两个角度出发分析实现系统的运行逻辑，设计对应的领域模型和领域事件，通过活动图和时序图描述功能交互流程，并对系统数据库进行详细设计。

第6章 系统实现与测试

6.1 系统的部署流程

在系统开发完成后，则需要将其部署到实际场景中并调试无误。系统部署到装有Linux操作系统的服务器上，部署过程如下：

1. 在 project\_env.sh 中配置项目环境参数，包括项目jar 包名和服务名。
2. 在 jvm.options.sh 中配置 JAVA 环境变量和 jvm 参数；JAVA环境变量主要是指定JAVA\_HOME和JRE\_HOME的值；jvm 参数主要配置初始堆大小 XMS、最大堆大小 XMX、年轻代大小 XMN、持久代初始值 X\_PERM\_SIZE、持久代最大值 X\_MAX\_PERM\_SIZE、Eden 区与 Survivor 区的大小比值 X\_SURVIVOR\_RATIO、gc日志输出文件位置GC\_LOG、gc算法等核心的 JVM 运行参数。
3. 在 control.sh 中编写启动、停止和重启脚本命令。
4. 通过统一CI/CD[49]流水线编译打包并发布到线上集群，实现项目部署。

6.2 测试概要

6.2.1 测试概述

在成功完成系统部署后，需要对系统进行多类型全方位的测试，通过对各项测试指标进行评估，判断系统是否满足需求分析和概要设计中规定的系统功能性和非功能性需求。对于软件测试过程中发现的软件缺陷和异常情况，分析其产生原因并给出后续解决方案，对于系统中的性能瓶颈和热点数据进行设计优化。

测试范围主要包含功能测试和性能测试两大部分。对于功能性测试，测试对象为各模块中的函数，采用TDD[50]测试方法，在软件开发过程中测试每个函数的功能逻辑是否完备正确，函数的参数、异常、边界值是否进行有效处理。对于性能测试，测试对象为系统核心业务模块，采用集成测试方法，测试整个系统的性能指标，包括系统响应时间、吞吐量、最大并发数等指标，进而判断系统是否满足支付结算业务的多用户高并发场景性能需求。

鉴于清结算系统模块代码规模较大、测试环境资源和论文篇幅有限，测试章节首先介绍测试环境资源，然后重点对系统核心模块进行测试分析，获取测试需求并设计测试用例，最后通过对测试结果指标进行分析，评估系统测试效果和软件产品质量。

6.2.2 测试环境

为了尽可能真实地模拟系统线上运行环境，测试环境选择对线上环境进行1：1的模拟，采取和线上部署时相同的系统资源，提前对系统运行情况进行测试分析，在测试阶段发现和解决系统bug和缺陷，降低系统后期维护成本。sim测试环境软硬件资源配置如下表6.1和表6.2所示。

测试PC机使用macOS Ventura 13.4操作系统，CPU为2.6GHz，内存为 16 GB，硬盘为 512GB，使用100M的局域网访问后台系统。后端服务器配置为128G内存、1TB硬盘容量等。数据库服务器配置为64GB 内存、1TB SSD等

表6.1 测试硬件环境表

|  |  |
| --- | --- |
| 硬件 | 配置 |
| 测试PC机 | CPU：2.6 GHz 6-Core Intel Core i7 |
| 内存：16 GB 2667 MHz DDR4 |
| 硬盘：512 GB SSD |
| 操作系统：macOS Ventura 13.4 |
| 网络：100M局域网 |
| 后端服务器 | CPU：2.5 GHz Intel Xeon(Skylake) Platinum 8163  内存：128 GB  硬盘：1T SSD  操作系统：CentOS 7.9 64位 |
| MySQL服务器 | CPU：2.5GHz Intel Xeon E5-2682v4  内存：64G  硬盘：1TB SSD  操作系统：64位Windows Server 2012 R2 Datacenter |

表6.2 测试软件环境表

|  |  |
| --- | --- |
| 软件 | 配置 |
| 操作系统 | macOS Ventura 13.4 |
| 数据库 | Mysql 8.0 |
| RocktMQ | [rocketmq-all-5.1.3](https://github.com/apache/rocketmq/releases/tag/rocketmq-all-5.1.3) |
| jdk | OpenJDK Version 1.8.0\_202-b08 |
| 客户端浏览器 | Chrome Version 114.0.5735.198 (Official Build) (x86\_64) |
| 测试工具 | JMeter 5.6 |

6.3 系统功能测试

6.3.1 测试需求

本节主要对清结算系统中的核心模块及其函数进行功能性测试。系统功能测试需求根据前期需求规格文档和系统设计文档获取，确定功能测试范围并描述测试重点，为后续测试用例设计打下基础。测试重点主要关注模块和函数是否与需求文档预期相符，所有组件是否正常运行，是否对异常情况进行有效处理。表6.3描述了清结算系统核心模块功能测试范围及其重点。

表6.3 功能测试范围表

|  |  |  |
| --- | --- | --- |
| 模块 | 功能 | 测试重点 |
| Binlog解析 | 接受消息 | 重点测试Binlog数据解析功能，判断是否能够正确接受数据，解析后的数据是否正确，同时对异常数据是否进入解析异常流程，测试消息队列消费状况、数据库清分表、异常表等信息是否正确。 |
| 解析Binlog |
| 校验字段 |
| 创建实体 |
| 计费 | 清分计费 | 对商户一笔交易进行收费测试，判断是否按照商户计费配置进行手续费计算，数据库清分实体中的费项是否完整、费项金额是否准确。 |
| 清算 | 创建详情单 | 测试商户多笔交易费用汇总结果是否正确，是否按照汇总规则进行汇总，包括是否为商户创汇总单，费项累计值是否准确，数据库结算实体各字段是否准确，以及每种费项对应的索引记录是否存在。 |
| 费项汇总 |
| 创建费项索引 |
| 结算 | 结算单创建 | 对结算单创建重点测试结算时间转换功能是否正确，结算单创建逻辑是否正确；对绑定详情单重点测试详情单和结算单是否正确绑定；对结算流程重点测试风控是否通过、账户服务是否记账成功和数据库结算单字段是否正确。 |
| 绑定详情单 |
| 结算处理 |
| 异常 | 自动重试 | 测试是否对清算和结算模块中的异常进行有效处理，判断异常处理状态机流转是否正确，测试异常表字段信息是否与异常处理相符。 |
| 人工处理 |

6.3.2 测试设计

测试设计部分采用黑盒测试方法对重点模块功能设计测试用例。黑盒测试在模块集成测试过程中通过模拟商户配置和交易数据，发起对系统模块的调用，然后执行业务逻辑，能够屏蔽底层实现细节，并且通过正常的测试数据和异常信息判断模块功能是否符合需求设计规定。本节对清结算系统五个模块分别设计黑盒测试用例。

Binlog清分测试用例如表6.4所示。首先模拟交易数据作为清分模块输入数据，然后将模拟数据发送到消息中间件，Binlog清分消费者获取模拟数据后开始清分处理流程。测试数据包含正常数据和异常数据两类，对于正常输入数据判断清分后的实体字段是否与正常模拟数据相符；对于异常输入数据，清分流程应该进行异常捕获和处理，判断异常表中的异常字段是否与异常输入数据相符。

表6.4 Binlog清分测试用例

|  |  |
| --- | --- |
| 用例条目 | 内容 |
| 用例名称 | Binlog清分 |
| 用例类别 | 功能测试 |
| 测试目标 | 1. 验证清分模块是否能够正常消费消息队列消息 2. 验证清分模块能否对正常交易数据进行解析和校验 3. 验证清分模块能否对异常交易数据进行异常处理 |
| 前置条件 | 部署消息中间件和清洁结算系统 |
| 测试流程 | 1. 构造正常和异常交易数据 2. 将测试数据发送到消息队列 3. 查看系统日志、数据库清分表和数据库异常表 |
| 预期结果 | 1. 清分模块能够正常消费消息队列中的测试数据 2. 对正常数据进行解析、校验和构造清分实体，并保存在清分表中 3. 对异常数据进行异常处理，将异常信息保存在数据库 |
| 实际结果 | 与预期结果相同 |
| 结论 | 功能正常，测试通过 |

计费模块在Binlog清分成功后后被调用。对于计费模块测试用例，需要测试计费主流程是否按照商户计费配置进行匹配计费，包括计费最大值、最小值、费率、费基和计算方式等，判断一笔商户交易各个费项计算结果是否符合预期，计费模块功能测试用例如表6.5所示。

表6.5 计费功能测试用例

|  |  |
| --- | --- |
| 用例条目 | 内容 |
| 用例名称 | 计费 |
| 用例类别 | 功能测试 |
| 测试目标 | 验证计费模块是否能够对清分实体进行匹配计费，并且结果正确 |
| 前置条件 | 清分后成功创建清分实体 |
| 测试流程 | 1. 为商户支付交易和退款交易配置交易手续费、税费、分期费计费配置 2. 调用计费模块接口 3. 观察系统日志、数据库清分表实体字段 |
| 预期结果 | 1. 系统日志显示计费成功，数据库清分记录更新计费字段 2. 对于支付和退款交易成功匹配对应的计费配置 3. 按照配置中的计算方式计费，计算结果正确且满足最大最小值约束 |
| 实际结果 | 和预期结果相同 |
| 结论 | 功能正常，测试通过 |

表6.6为清算模块功能的测试用例。在计费成功后，调用清算模块进行费项清算汇总，按交易类型分类汇总各费项。清算汇总时间以按天为单位进行汇总，即一天内同一个商户的所有交易汇总到一个结算详情表中，因此首先需要判断结算详情表是否创建成功。其次，系统通过数据库索引避免重复汇总费项，因此也需要测试是否过滤重复计费请求。最后，为了过滤重复计费请求，每笔交易汇总后都会创建各个费项的索引记录，即测试费项索引表是否创建对应记录。

表6.6 清算汇总测试用例

|  |  |
| --- | --- |
| 用例条目 | 内容 |
| 用例名称 | 清算汇总 |
| 用例类别 | 功能测试 |
| 测试目标 | 验证清算模块是否能正确进行费项汇总 |
| 前置条件 | 交易计费成功 |
| 测试流程 | 1. 构造商户多笔不同类型交易（包括正向和逆向交易）测试数据 2. 构造商户多笔重复交易数据，验证是否过滤重复请求 3. 调用清算汇总模块服务接口 4. 查看系统日志、结算详情表、费项索引表记录是否正确 |
| 预期结果 | 1. 能够对不同类型的交易进行费项汇总，汇总结果正确 2. 能对重复交易汇总请求进行过滤 3. 每笔交易对应一条结算详情表汇总记录和多条费项索引记录 |
| 实际结果 | 和预期结果相同 |
| 结论 | 功能正常，测试通过 |

表6.7结算模块功能的测试用例。清算汇总后，在到达结算时间时，通过创建结算单，绑定所属时间范围内的汇总详情单，进行汇总结算流程，完成商户资金转移。

表6.7 结算功能测试用例

|  |  |
| --- | --- |
| 用例条目 | 内容 |
| 用例名称 | 结算 |
| 用例类别 | 功能测试 |
| 测试目标 | 1. 验证结算模块是否能够进行汇总结算 |
| 前置条件 | 1. 清算汇总结束，生成结算详情单 |
| 测试流程 | 1. 初始化商户结算配置，写入结算数据库 2. 构造多个结算详情单测试记录 3. 模拟并发结算请求 4. 调用结算服务接口 5. 观察系统日志、结算详情单、结算单记录是否正确 |
| 预期结果 | 1. 测试数据库成功创建新的结算单 2. 结算详情单记录结算单单号，结算单成功绑定对应的结算详情单 3. 结算单汇总字段等于绑定的结算详情单字段和，结算状态为待提现 4. 系统日志记录风控处理和记账请求日志，成功对支付交易进行拦截，每个汇总费项记账请求成功 |
| 实际结果 | 和预期结果相同 |
| 结论 | 功能正常，测试通过 |

表6.8异常管理模块功能的测试用例。异常管理模拟能够对Binlog清分、计费、清算和结算流程中的异常事件进行捕获和重试处理，降低因为系统内外部异常导致清结算流程中断服务的概率。

表6.8 异常管理功能测试用例

|  |  |
| --- | --- |
| 用例条目 | 内容 |
| 用例名称 | 异常管理 |
| 用例类别 | 功能测试 |
| 测试目标 | 1. 验证异常模块是否能对清结算流程中各类异常进行捕获和处理 2. 验证异常模块是否能对异常事件进行自动重试和人工处理等流程 |
| 前置条件 | 1. 清结算系统产生异常事件 |
| 测试流程 | 1. 构造Binlog清分异常，包括binlog数据字段缺失和解析校验异常两种 2. 构造计费模块商户计费配置缺失导致的计费异常 3. 构造清算汇总模块商户结算配置缺失导致的汇总异常 4. 构造结算流程因商户结算配置缺失导致的结算异常 5. 构造商户黑名单，模拟支付风控拦截异常 6. 输入异常测试数据，调用异常模块服务接口 7. 观察系统日志异常信息、异常表记录信息是否正确 |
| 预期结果 | 1. 对于binlog数据字段缺失导致的异常，异常模版记录异常字段信息，异常不进入自动重试流程，因为数据本身有问题而自动重试会一直不成功；对于数据正确而binlog在解析过程中因解析逻辑错误导致的异常在自动重试超过阈值时会发起人工处理，此时异常表包含人工处理记录。 2. 对于计费模块因商户计费配置缺失导致构造计费领域模型失败，该异常将同样会自动重试后到达人工处理结算，异常表记录计费配置缺失异常信息。 3. 对于清算汇总模块因为缺失商户结算配置导致无法创建结算汇总模型，该异常信息经过自动重试后会进行人工处理，异常表记录结算配置缺失异常。 4. 对于结算流程因为商户结算配置缺失导致结算模型创建失败，该异常信息经过自动重试，次数达阈值后进入人工处理，异常表记录该异常信息。 5. 对于黑名单商户的交易类型数据，结算流程会进行风控拦截，拦截不通过无法进入结算流程，异常表记录风控拦截异常信息。 |
| 实际结果 | 和预期结果相同 |
| 结论 | 功能正常，测试通过 |

6.3.3 测试结果

根据设计内容，使用用例多次对整个系统进行功能测试，最终汇总得到测试结果如表6.9所示。

表6.9 功能测试结果表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 所属模块 | 功能/业务 | 用例总数 | 通过 | 失败 | 通过率 |
| binlog清分模块 | 接收 binlog | 10 | 10 | 0 | 100% |
| binlog清分模块 | 解析校验binlog | 20 | 20 | 0 | 100% |
| binlog 清分模块 | 构造清分实体 | 20 | 20 | 0 | 100% |
| 计费模块 | 解析计费配置 | 20 | 20 | 0 | 100% |
| 计费模块 | 构造计费领域模型 | 20 | 20 | 0 | 100% |
| 计费模型 | 交易各费项计费 | 50 | 50 | 0 | 100% |
| 清算汇总模块 | 创建汇总详情单 | 20 | 20 | 0 | 100% |
| 清算汇总模块 | 费项累计 | 50 | 50 | 0 | 100% |
| 清算汇总模块 | 创建费项索引记录 | 20 | 20 | 0 | 100% |
| 结算模块 | 创建结算单 | 20 | 20 | 0 | 100% |
| 结算模块 | 绑定结算详情单 | 50 | 50 | 0 | 100% |
| 结算模块 | 风控拦截 | 20 | 20 | 0 | 100% |
| 结算模块 | 汇总详情单 | 100 | 100 | 0 | 100% |
| 结算模块 | 结算记账 | 20 | 20 | 0 | 100% |
| 异常处理模块 | 自动重试 | 20 | 20 | 0 | 100% |
| 异常处理模块 | 人工处理 | 20 | 20 | 0 | 100% |

从模块功能测试结果表中可以得出，测试用例设计根据模块功能的相对重要性，选择更多的测试用例进行正确性测试。对于计费模块核心功能和结算模块核心功能，其测试用例数倍于其他模块功能，使得清结算核心模块的得到充分全面的测试，保证其功能正确运行，以提高系统整体的可靠性和软件质量。由于计费模块中的费项计费是平台手续费收入的主要来源，因此费项计费的测试用例多余其他功能测试用例；其次，结算模块中的汇总功能同样涉及到商户资金和平台资金的重大利益，其测试用例数是整个系统模块中最多的。

测试结果表中测试通过率为100%，说明清结算系统各模块功能都可以正常运行，能够按照商户配置进行手续费计费并且进行商户资金结算操作。同时也表明系统满足需求分析和系统设计阶段规定的各项功能要求，系统缺陷经过测试得到解决。测试结论总结如下：

1. binlog清分模块接受、解析和校验binog功能能够正常运行
2. 计费模块计费准确无误，平台手续费金额准确性得到可靠保证
3. 清算汇总模块汇总累计结果准确无误
4. 结算模块能够按照商户结算规则准时精确地进行商户资金结算
5. 异常模块能够对系统各类异常进行兜底处理

6.3.4 优化策略

测试结果显示系统模块功能基本完成且运行良好，没有软件明显的缺陷。但是，某些模块功能依然存在待优化的空间，进一步提高系统功能可靠性和响应速度。例如，binlog清分模块在遇到大量交易数据时可能会来不及处理导致消息积压问题，虽然一开始不影响正常清分流程，但是消息积压如果不及时处理会导致线上消息中间件物理内存用尽，进而影响中间件对外提供服务。因此需要动态的根据交易系统的QPS配置消费者组和线程数进行并发处理，保证生产者和消费者处理速度相对平衡，提高binlog清分模块的可靠性。其次，在结算流程中，在资金结算前需要同步调用支付风控进行风控拦截，在结算时需要同步调用记账服务进行费项记账，这些同步调用导致清结算系统与外部系统紧耦合，延长了结算流程处理时间。因此，在后续优化中，考虑将同步调用改成异步调用，通过消息中间件进行解耦和异步通信，提高整体结算效率。

6.4 系统非功能测试

6.4.1 测试需求

系统非功能测试主要测试系统稳定性指标。系统稳定性包含多个方面，包括外部依赖系统和组件，清结算系统稳定性主要由四个方面决定，分别是交易系统的交易QPS、消息中间件RocketMQ的稳定性和、账户系统的稳定性和系统模块本身的可靠性。非功能性测试通过系统稳定性评估和测试，验证当前系统是否满足业务非功能性需求，验证清结算系统是否具备线上环境持续稳定运行的能力，并给出故障预案、监控报警预案和后续优化方案。根据需求分析中的非功能性需求，系统应当满足以下表6.10中描述的各项非功能性指标。

表6.10 非功能性指标表

|  |  |  |
| --- | --- | --- |
| 指标 | 指标要求 | 测试重点 |
| 可扩展性和可维护性 | 扩展能力 | 清结算系统能支持更多的交易类型、支付类型、更丰富的计费规则和结算规则，不断适应业务需求变化。 |
| 维护成本 | 监控平台和日志平台实时监控和记录运行状况，提供系统运行时各项指标分析和全链路日志追踪功能，降低系统维护成本。 |
| 易用性与可读性 | 易用性 | 清结算系统提供小程序入口，能够对单笔交易进行清分结算，同时也能对异常交易进行重试处理。 |
| 可读性 | 清结算系统采取领域驱动和面向对象结合的设计方式，降低系统复杂度。同时，采取多种设计模式并遵循软件开发规范提高代码可读性。 |
| 安全性 | 应用安全 | 系统建立严格的访问控制策略，确保只有被授权的用户和系统能够访问系统进行操作。同时，系统采用集群部署，定期指定灾备计划，确保在任何情况下都能保持应用正常运行。进行身份验证和授权，所有访问系统的用户和系统都应进行严格的身份验证和授权，采用多因素认证机制提高应用安全性。 |
| 网络安全 | 网络分隔对内外网进行隔离，对于处理敏感交易数据的服务器应放在内网中，避免直接暴露在互联网上。同样，不同的系统或者功能模块也可以进一步隔离，如交易系统和清算系统等，以降低安全风险。 |
| 数据安全 | 数据加密，对所有敏感信息，包括交易数据、商户身份信息在存储和传输时采用强加密算法加密，同时进行数据备份；数据库定期安全检查，设置严格的访问控制，防止SQL注入等数据库攻击；数据权限管理，对于访问和处理敏感数据的人员，需要有严格的权限控制和管理，包括最小权限原则、定期审查权限等。数据库备份与恢复，制定详细的数据备份和恢复策略，防数据丢失或系统故障，保障业务连续性。 |

此外，性能需求是非功能性需求中最关键的内容，它决定清结算系统是否能有效支撑上层业务的流量需求。因此，非功能性测试重点测试系统性能指标，同时对其他非功能性指标进行测试。清结算系统的性能指标如下：

1. 清结算QPS 大于交易系统binlog QPS，目前交易数据库binlog生产 QPS 为 96，因此清结算 QPS 至少为100。
2. 并发处理能力：binlog清分效率不低于每秒1000个消息，清算效率不低于每秒500个商户，结算效率不低于每秒100个用户。
3. 响应时间：商户配置的查询、修改和激活请求，以及清算单、结算详情单、结算单和异常单查询请求平均响应时间小于200ms；
4. 系统稳定性：可承受日常和突发流量负载，系统可稳定运行。
5. 系统资源利用率：内存占用率 < 70%，磁盘使用率 < 60%，网络带宽使用率 < 60%。

6.4.2 测试设计

非功能新测试采用黑盒测试方法，测试对象为binlog解析、清算和结算三个核心模块的性能，测试环境为sim环境，避免对线上环境造成影响。测试接口选取清结算系统流量入口函数和性能热点函数，包括binlog解析函数、清算汇总函数和结算函数进行函数接口的性能测试。通过测试流量入口函数和性能热点函数的性能指标，判断系统整体性能是否满足前期系统性能设计需求。

清结算系统各个模块函数之间存在数据的前后依赖关系，需要统一设计测试数据来满足各个模块的测试需求。比如，清算汇总依赖binlog解析产生的清分数据，而结算依赖清算汇总产生的结算详情数据，为了同时满足binlog QPS不低于100/s、清算汇总并发度不低于500商户/s和结算并发度不低于100商户/s的性能需求，需要综合考虑测试数据的分布和数量。为此，商户测试数据选择5000个测试商户，每个商户配置4个计费配置和一个结算配置；交易测试数据选择相同的5000个测试商户，每个商户模拟10笔支付交易和10笔退款交易，总共 5000 \*(10+10) = 10w条交易数据。性能测试数据规模如表如6.11所示。

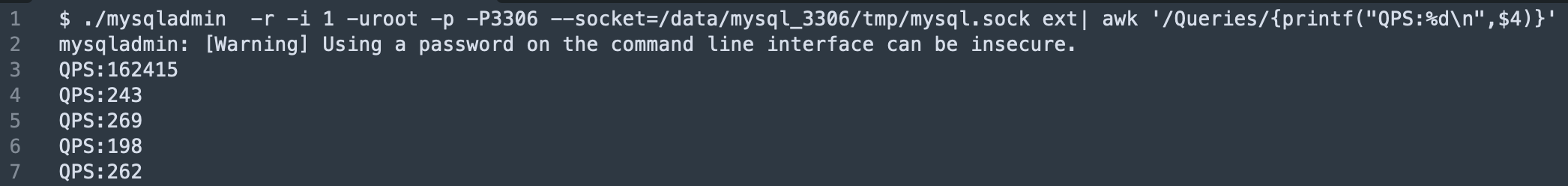
表6.11 性能测试数据规模

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 库名 | 表名 | 测试数据量 |
| Mysql | trade | pay\_trade | 50000 |
| Mysql | trade | refund\_trade | 50000 |
| Mysql | merchant | merchant | 5000 |
| Mysql | config | fee\_config | 20000 |
| Mysql | config | settle\_config | 5000 |

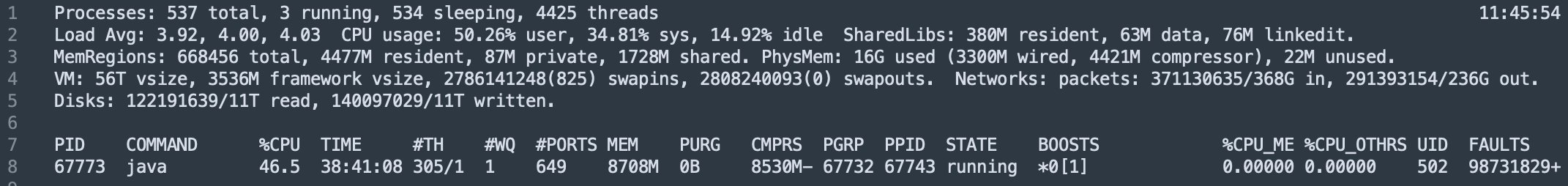
对于binlog解析性能测试，首先将10w条交易数据写入交易数据库，然后同步到消息中间件，最后观察数据库清分表写入清分实体的QPS是否大于100/s；对于清算汇总函数，观察结算详情单测试表写入QPS是否大于500/s；对于结算函数，观察结算单写入QPS是否大于100/s，以此判断系统性能指标是否满足性能设计需求。通过登录测试MySQL服务器，使用管理员命令行工具 mysqladmin 获取数据库执行情况，计算QPS。同时，登录sim后台服务器，使用top命令统计系统执行情况，计算cpu和内存使用情况。

6.4.3 测试结果

使用 mysqladmin 获取qps的结果如图6.1所示：

图6.1 mysq qps 测试结果

使用top命令获取cpu 和 内存的使用情况图6.2所示：

图6.2 top测试结果

汇总mysql 服务器QPS、后台服务器cpu和内存使用率，得到性能测试结果如表6.12所示：

表6.12 性能测试结果表

| 用例  编号 | 用例名称 | 并发数 | 结果统计 | | 后台服务器 | | DB服务器 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 实际QPS | 成功率(%） | CPU  （%） | MEM  （%） | 数据量 |
| C01 | binlog清分 | 30 | 126 | 96.43 | 42.66 | 44.23 | 100000 |
| C02 | 清算汇总 | 48 | 663 | 95.23 | 56.15 | 67.43 | 400000 |
| C03 | 结算 | 48 | 195 | 99.83 | 53.31 | 42.70 | 5000 |

分析性能测试结果，可得到如下结论：

1. binlog解析测试数据量为100000条，实际QPS大于100/s，测试通过；
2. 清算汇总测试数据量为400000条，实际QPS大于500/s，测试通过；
3. 结算测试数据量为5000条，实际QPS大于100/s，测试通过；
4. 系统资源：CPU使用率<80%，内存使用率<80%，测试通过；
5. 稳定性：测试期间系统稳定运行，未出现死锁宕机等情况，测试通过。

综上所述，本次非功能测试性能指标符合预期，同时验证了系统扩展性、稳定性等非功能性指标，系统整体非功能性指标符合设计规范，系统能够在线上稳定运行。

6.4.4 优化策略

测试结果显示各项性能指标都已经达标，但是当前binlog清分实际QPS和结算QPS不算太高，binlog解析功能吞吐量较低。在商户入驻数和交易数逐渐增大的背景下，需要进一步提高binlog清分和结算效率。为此，需要实时监控交易binlog数据量，同步增大清结算mq消息队列数和消费者组线程数，提高binlog清分并发消费能力；同时，结算模块需要将整个流程异步化，通过消息中间件解耦支付风控和记账等下游依赖服务，采用最终一致性和失败重试等补偿技术，提高整体结算效率。

6.5 本章小结

本章通过对系统进行功能性和性能测试，验证系统是否满足设计需求。通过在sim环境下模拟测试数据，执行测试用例并观察结果，分析判断系统QPS是否满足设计规定，最后对系统优化提出改进策略。

**第7章 结 论**

7.1 工作总结

随着滴滴国际支付在海外不断深入发展，原有支付清结算系统已经不能满足当前海量交易和规则复杂的支付清结算业务需求。因此，本文基于领域驱动设计和面向对象相互结合的方法，设计并实现一个专用的第三方支付清结算系统，解决了海外复杂环境下的商户收单结算问题。在实际应用中，带来灵活高效的商户资金结算能力，提高了平台收单结算效率，为国际化出行、外卖等业务提供支撑。本文主要内容为清结算系统的设计与实现，包括系统部署和测试。主要工作内容总结如下：

1. 根据领域驱动设计思想，同时结合当前清结算需求，识别出清结算领域概念，划分系统子域，包括binlog清分、计费、清算汇总、结算和异常处理子模块。通过领域驱动构建元素反应业务概念和规则，使系统各模块具有丰富的业务含义，设计开发的系统具有高内聚低耦合的软件特性。
2. 使用主流的后端服务器编程语言和技术框架开发实现系统。系统使用主流的Java语言作为面向对象编程语言，同时使用企业级后端开发框架Spring和SpringBoot提高开发效率；使用金融级别消息中间件RocketMQ实现异步通信和解耦；使用Apollo分布式配置中心统一管理维护商户配置信息；使用XXL-JOB定时任务调度平台完成定时清算和定时结算等任务。
3. 系统开发完成后，对系统进行功能性和非功能性测试。将系统部署到sim模拟环境中，观察其运行状态，然后设计测试用例和测试数据对系统功能和性能进行测试，测试系统稳定性，确保系统已经满足前期设计需求。

7.2 课题展望

7.2.1 不足之处

本课题基本完成了需求分析阶段确定的各项需求，包括功能性需求和非功能性需求。然而，由于人力、时间和机器资源等方面的限制，系统存以下问题有待改进。

系统稳定性和吞吐量受到当前有限的系统资源的限制。由于资源有限，目前系统MySQL服务器采用集群和一主多从架构的部署方式，后期由于流量增大，应该对集群进行扩容且设计为多主多从的架构；其次，消息中间件可申请的内存资源有限，后期为了解耦记账等服务，需要增加消息队列内存和cpu等资源配置；系统其他资源如redis服务器和es服务器在后期交易量扩大时，需要进行相应的扩容和数据备份；

系统监控配置有待完善。应该对系统监控指标进行分类细化，分为系统级、服务机和业务级三类指标。系统级指标应该包含CPU、内存、磁盘、网络、资源争抢等；服务级指标应该包含进程存活数、核心接口qps、核心依赖qps、数据库错误信息、mq消息积压、接口和服务的错误数和超时数据等；业务级指标应该包含各类埋点数据，包括交易总额、清算总额、清算净额、手续费清算总额、结算总额等业务数据。系统监控应该配置在企业监控平台，方便后续维护，提高系统运行时的可观测性。

最后，清结算资金安全需要开发对账系统进行严格保证。虽然清结算系统对异常交易进行重试处理，对金额计算采用BigDecimal类处理，但是还是可能存在交易漏单情况。因此，为了防止出现交易漏单，需要跟交易侧进行T+1对账，对账逻辑为交易单号、交易类型和交易金额三元组。同时，清结算和账户侧也需要对账，包括结算和提现的对账、清算和账户的对账以及结算和账户的对账，保证数据的一致性。只有进行全面的资金对账处理，才能保证清结算全流程的资金安全。

7.2.2 课题展望

目前国内许多大型企业在项目中逐渐使用 DDD 来处理自身复杂的业务问题，DDD的应用比较广泛。但是，DDD还不是主流的开发设计方法，因为 DDD的理论和实践有一定的复杂度，对业务人员和开发人员在专业知识和技能方面要求较高。此外，对 DDD 的理解和运用程度也因人而异，可能存在误解或过度设计的问题，进而导致企业系统开发成本变高以及项目无法按时交付等风险。

在一些具有复杂业务领域的行业中，比如金融、电商、物流等，DDD 的应用会更加广泛。在这些行业中，领域模型的建立和业务流程的抽象都比较复杂，DDD 的方法论可以更好地帮助开发者理解和处理这些复杂性。随着微服务架构的普及，DDD 在微服务设计中的应用也逐渐受到重视，因为 DDD 提供了有助于解决微服务中一些关键问题的思想和工具，比如如何划分微服务的边界，如何进行微服务之间的通信等。总的来说，尽管 DDD 在国内的应用现状并不是特别广泛，但是其在处理复杂业务系统设计中的优势正在被越来越多的企业和开发者认识和接受，未来有望得到更广泛的应用。

本文基于领域驱动设计实现第三方支付清结算系统有很多潜在的优势，比如可以更好地抽象复杂的业务流程，更方便地管理和维护代码，以及更容易地扩展和修改系统功能。然而，随着业务的发展也可能面临一些挑战：

首先，企业必须满足合规要求。在许多国家和地区，支付和清结算系统都需要遵守当地严格的法律法规。如何在满足这些法规要求的前提下，实现业务的灵活扩展和快速迭代，将是一个挑战；其次，随着交易量不断增大，系统将面临性能问题。支付清结算系统往往需要处理大量的交易，这就对系统性能提出了很高的要求。如何在保证系统性能的前提下，实现业务的复杂逻辑，也是一个挑战；最后，系统必须满足数据一致性要求。在分布式系统中，数据的一致性和事务管理是一大难题。而在支付清结算系统中，由于涉及到资金流转，对数据一致性的要求更高。

**参 考 文 献**

[1] 洪观琳.浅谈资金清结算对电子支付平台服务能力提升的作用[J].中国新通信,2016,18(15):121.

[2] PayPal[EB/OL].[2023-07-28].https://www.paypal.com/c2/home.

[3] Apple Pay[EB/OL].[2023-07-28].https://www.apple.com/apple-pay/.

[4] Google Pay[EB/OL].[2023-07-28].https://pay.google.com/about/.

[5] Alipay open platform[EB/OL].[2023-07-30].https://open.alipay.com.

支付宝开放平台[EB/OL].[2023-07-30].https://open.alipay.com.

[6] Weixin pay merchant platform[EB/OL].[2023-07-23].https://pay.weixin.qq.com.

微信支付开放平台[EB/OL].[2023-07-23].https://pay.weixin.qq.com.

[7] Evans, Domain- Driven Design. Addison-Wesley, August 2003.

[8] Nilsson J.Applying domain-driven design and patterns：with example in C#and .NET.NEW Jersey：Addison-Wesley professional，2006.

[9] V. Vernon, Implementing Domain-Driven Design. Addison-Wesley, February 2013.

[10] S. Millett and N. Tune, Patterns, Principles, and Practices of Domain-Driven Design. Wrox, April 2015.

[11] S Wlaschin, Domain Modeling Made Functional. Pragmatic Bookshelf, January 2018.

[12] Harald Wesenberg, Einar Landre, and Harald Rønneberg. Using domain-driven design to evaluate commercial off-the-shelf software. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA '06). 2006.824–829.

[13] Einar Landre, Harald Wesenberg, and Jorn Olmheim. 2007. Agile enterprise software development using domain-driven design and test first. In Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA '07). Association for Computing Machinery, New York, NY, USA, 983–993.

[14] A. Rahmatulloh, D. W. Sari, R. N. Shofa and I. Darmawan, "Microservices-based IoT Monitoring Application with a Domain-driven Design Approach," 2021 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS), 2021, pp. 1-8, doi: 10.1109/ICADEIS52521.2021.9701966.

[15] Oukes, Peter, et al. "Domain-Driven Design applied to land administration system development: Lessons from the Netherlands." Land use policy 104 (2021): 105379.

[16] Zalando [EB/OL].[2023-07-31]. https://qeunit.com/blog/zalando-quality-engineering-

journey-from-monolith-to-microservices/.

1. 王忠,程磊.基于领域驱动设计的软件开发[J].软件导刊,2008(02):37-39.
2. 郑琴琴. 领域模型驱动的Web服务共享平台构建[D].大连理工大学,2009.
3. 易立江. 基于领域驱动设计的证券交易系统的设计与实现[D].中国科学院大学(工程管理与信息技术学院),2015.
4. 胡俊霞.领域驱动设计在企业信息化系统中的应用研究[D].郑州大学,2017.
5. 张帅.基于领域驱动设计的AGV调度系统设计与实现[D].北京交通大学,2021.
6. 黄喆. 基于领域驱动设计的银行存款业务核心系统设计与实现[D].西安电子科技大学,2021.
7. 贾子甲,钟陈星,周世旗,荣国平,章程.领域驱动设计模式的收益与挑战:系统综述[J].软件学报,2021.
8. Tashtoush Yahya,AbuElRub Noor,Darwish Omar,AlEidi Shorouq,Darweesh Dirar,Karajeh Ola. A Notional Understanding of the Relationship between Code Readability and Software Complexity[J]. Information,2023,14(2).
9. 何磊. 基于代码复杂度的软件演化评估与分析[D].东南大学,2016.
10. Royce, Winston W. "Managing the development of large software systems: concepts and techniques." Proceedings of the 9th international conference on Software Engineering. 1987.
11. C. Larman and V. R. Basili, "Iterative and incremental developments. a brief history," in Computer, vol. 36, no. 6, pp. 47-56, June 2003.
12. ["Code Complete's 2nd edition website"](https://stevemcconnell.com/books/). Steven C. McConnell. Retrieved 2018-10-17.
13. Spring framework[EB/OL].[2023-07-31].<https://spring.io/projects/spring-framework>.
14. Spring Boot framework[EB/OL].[2023-07-31].https://spring.io/projects/spring-boot.
15. Mybatis[EB/OL].[2023-07-31].<https://mybatis.org/mybatis-3/.>
16. Hibernate[EB/OL].[2023-07-31].<https://hibernate.org/>.
17. Apache RocketMQ[EB/OL].[2023-07-31].https://rocketmq.apache.org/.
18. Apache Kafka[EB/OL].[2023-07-31].https://kafka.apache.org/.
19. RabbitMQ[EB/OL].[2023-07-31].https://www.rabbitmq.com/.
20. ActiveMQ[EB/OL].[2023-07-31].https://activemq.apache.org/.
21. Apollo[EB/OL].[2023-07-31].https://www.apolloconfig.com/#/.
22. Spring Cloud Config[EB/OL].[2023-07-31].https://docs.spring.io/spring-cloud-config/docs/ current/reference/html/.
23. XXL-JOB[EB/OL].[2023-07-31].https://www.xuxueli.com/xxl-job/.
24. Martin, Robert C. "Getting a SOLID start." Robert C. Martin-objectmentor. com (2013).
25. Martin, Robert C. "Design principles and design patterns." Object Mentor 1.34 (2000): 597.
26. Binary Log[EB/OL].[2023-07-31].https://dev.mysql.com/doc/refman/8.0/en/binary-log.html.
27. R. W. Wolverton, "The Cost of Developing Large-Scale Software," in IEEE Transactions on Computers, vol. C-23, no. 6, pp. 615-636, June 1974.
28. T. J. McCabe, "A Complexity Measure," in IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308-320, Dec. 1976.
29. Young,Greg,["CQRS Documents"](http://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf).[EB/OL].[2023-07-31].https://cqrs.files.wordpress.com/

2010 /11/cqrs\_documens.pdf.

1. Fowler, Martin."CQRS".[EB/OL].[2023-07-31].https://martinfowler.com/bliki/CQRS.html.
2. Alibaba Canal.[EB/OL].[2023-07-31].https://github.com/alibaba/canal/wiki.
3. Twitter snowflake.[EB/OL].[2023-07-31].https://github.com/twitter-archive/snowflake/tree/.
4. Rossel, Sander. Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment. Packt Publishing Ltd, 2017.
5. Beck, Kent. Test-driven development: by example. Addison-Wesley Professional, 2003.

**致 谢**

在这个特殊的时刻，我满怀感激之情，作为一名中国科学技术大学的研究生，我有幸完成了我的硕士研究生毕业论文。

本论文的完成，离不开我的校内导师孙广中教授和企业导师杜经纬专家工程师的悉心指导。首先，我要对孙教授表达我深深的感谢。在论文的写作过程中，孙教授准确地指出了我在学术研究上的缺点，并提供了深入的指导和宝贵的建议。孙教授让我明白，研究的早期阶段一定要进行深入的研究和周全的设计，以确保研究的后期阶段能够按预定的目标进行。此外，他还要求论文的写作要有条理、清晰和规范。孙教授严谨的治学态度和求真务实的科研精神深深地影响了我。

在企业实习期间，杜老师在我的职业生涯和学术研究中起到了重要的指导作用。他在系统开发设计过程中细心指导我，使我能够快速适应工作岗位；在相关技术领域，他为我确定了研究方向，提供了专业的研究学习资料，引导我掌握研究领域的专业知识和开发技能。在两位导师的指导下，我逐步熟悉并掌握了企业级软件设计和开发的相关专业技术，最终成功设计并实现了研究课题，完成论文撰写。我非常荣幸能得到他们的指导，在此，我对孙教授和杜老师致以最诚挚的敬意和最衷心的感谢。

同时，我还要对我的同学和实习公司的同事和领导表示感谢。在论文撰写期间，我和同学们经常一起讨论研究问题和开发技术，通过这些交流，我深化了对专业技术的理解，解决了一些我关心的问题。我也要感谢实习公司的同事和领导们对我工作和毕业论文的支持和理解。

最后，作为一名中国科学技术大学的研究生，我衷心祝愿中科大的未来将更加精彩与辉煌！

2023年8月

1. [↑](#footnote-ref-0)
2. [↑](#footnote-ref-1)