

**本科毕业设计（论文）**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **题目** | **基于TCP的远程调用RPC框架研究** | | | |
|  | | | | |
| **学生姓名** | | **郑剑** | **学 号** | **202031070390** |
| **教学院系** | | **计算机科学学院** | | |
| **专业年级** | | **软件工程2020级** | | |
| **指导教师** | | **肖斌** | **职 称** | **教授** |
| **单 位** | | **西南石油大学** | | |
| **辅导教师** | |  | **职 称** |  |
| **单 位** | |  | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **完成日期** | **2023** | **年** | **5** | **月** | **28** | **日** |

Southwest Petroleum University

Graduation Thesis



The Design And Implemention of Bank Seckill System Server-end Based on Microservices

**Grade: 2020**

**Name: Zheng jian**

**Speciality: Software Engineering**

**Instructor: Xiao Bin**

**School of Computer Science**

**2023-5**

**摘要**

互联网已然成为人们生活中不可或缺的一部分。包括中小银行在内的各行各业正如火如荼地开展数字化转型，其中“秒杀”这一促销手段逐渐成为了开辟线上市场的重要形式，通过刺激用户购买心理从而为创收与客源获取创造了有利条件。而传统的秒杀系统囿于单体式架构带来的诸多缺陷，已难以满足日益增长的业务需求。

因此，针对某银行推出存款产品上线并投放秒杀活动的需求，本文设计并实现了一套基于微服务的银行秒杀系统服务端，提供用户服务、商品服务、筛查服务、秒杀服务、订单服务和支付服务，为管理系统和客户端提供支撑。系统应用微服务架构，采用Spring Cloud作为项目框架，将Consul作为服务的注册中心，各服务之间通过Dubbo或Spring Cloud OpenFeign完成服务调用，并引入Spring Cloud Gateway服务网关统一处理来自客户端的请求。系统重点解决了秒杀业务的相关性能与安全问题，如商品超卖、链接暴露等，采用Redis缓存热点数据提高系统的响应速度，并借助RabbitMQ消息中间件完成流量削峰和应用解耦。经测试，本系统能够保障业务正常、有序地开展，并兼具可扩展性、高可用性，为客户带来良好的使用体验。

**关键词：**微服务；远程调用；计算机网络

**Abstract**

The internet has become an indispensable part of our daily existence. Various industries, including small and medium-sized banks, are enthusiastically embracing digital transformation. Among them, "flash sale" has gradually become an important form of online market expansion, creating favorable conditions for revenue generation and customer acquisition by tapping into users' buying psychology. However, traditional flash sale systems, constrained by the numerous shortcomings of monolithic architecture, have become increasingly unable to meet the growing business demands.

Therefore, in response to the demand for launching deposit products online and conducting flash sale promotions for a bank, this paper designs and implements the server-side of a microservice-based bank seckill system, which provides user service, product service, screening service, seckill service, order service and payment service, to support the front-end interface of a management system and client-side functionalities. The system employs a microservice architecture, utilizing Spring Cloud as the project framework, Consul as the service registry center, and Dubbo or Spring Cloud OpenFeign for inter-service communication among microservices. Additionally, Spring Cloud Gateway is introduced as a service gateway to handle requests from clients in a unified manner. The system focuses on addressing the performance and security issues related to the flash sale business, such as product overselling and link exposure. It utilizes Redis caching to improve system response speed by caching hot data, and leverages RabbitMQ message middleware for traffic peak shaving and application decoupling. Through testing, it has been proven that this system guarantees seamless and well-organized execution of business activities, while also providing scalability and high availability, thus providing customers with a superior user experience.

Key words: Microservices; Seckill System; Spring Cloud

**目录**

[1 绪论 1](#_Toc136274471)

[1.1 研究背景及意义 1](#_Toc136274472)

[1.2 国内外研究现状 2](#_Toc136274473)

[1.2.1 微服务研究现状 2](#_Toc136274474)

[1.2.2 秒杀系统研究现状 3](#_Toc136274475)

[1.3 本文主要工作和结构安排 3](#_Toc136274476)

[1.3.1 主要工作 3](#_Toc136274477)

[1.3.2 结构安排 4](#_Toc136274478)

[1.4 本章小结 4](#_Toc136274479)

[2 相关理论与技术 5](#_Toc136274480)

[2.1 Spring Cloud微服务框架 5](#_Toc136274481)

[2.2 微服务关键组件 5](#_Toc136274482)

[2.2.1 服务注册与发现 5](#_Toc136274483)

[2.2.2 服务网关 7](#_Toc136274484)

[2.2.3 服务调用 8](#_Toc136274485)

[2.2.4 服务容错 9](#_Toc136274486)

[2.3 RabbitMQ消息中间件 10](#_Toc136274487)

[2.4 数据存储技术 10](#_Toc136274488)

[2.4.1 MySQL关系型数据库 10](#_Toc136274489)

[2.4.2 Redis非关系型数据库 10](#_Toc136274490)

[2.5 本章小结 10](#_Toc136274491)

[3 银行秒杀系统服务端的分析 11](#_Toc136274492)

[3.1 可行性分析 11](#_Toc136274493)

[3.1.1 技术可行性 11](#_Toc136274494)

[3.1.2 操作可行性 11](#_Toc136274495)

[3.1.3 经济可行性 11](#_Toc136274496)

[3.2 功能需求分析 11](#_Toc136274497)

[3.2.1 管理员功能需求分析 11](#_Toc136274498)

[3.2.2 客户功能需求分析 12](#_Toc136274499)

[3.2.3 基于参与者的秒杀流程分析 14](#_Toc136274500)

[3.3 非功能需求分析 15](#_Toc136274501)

[3.3.1 性能效率需求 15](#_Toc136274502)

[3.3.2 可用性需求 15](#_Toc136274503)

[3.3.3 安全性需求 15](#_Toc136274504)

[3.3.4 可扩展性需求 15](#_Toc136274505)

[3.3.5 其他非功能需求 15](#_Toc136274506)

[3.4 本章小结 16](#_Toc136274507)

[4 银行秒杀系统服务端的设计 17](#_Toc136274508)

[4.1 服务端软件架构设计 17](#_Toc136274509)

[4.2 数据库设计 18](#_Toc136274510)

[4.2.1 实体设计 18](#_Toc136274511)

[4.2.2 概念模型E-R图设计 21](#_Toc136274512)

[4.2.3 物理设计 22](#_Toc136274513)

[4.3 服务端功能模块设计 26](#_Toc136274514)

[4.3.1 用户服务模块详细设计 26](#_Toc136274515)

[4.3.2 商品服务模块详细设计 29](#_Toc136274516)

[4.3.3 筛查服务模块详细设计 30](#_Toc136274517)

[4.3.4 秒杀服务模块详细设计 31](#_Toc136274518)

[4.3.5 订单服务模块详细设计 32](#_Toc136274519)

[4.3.6 支付服务模块详细设计 35](#_Toc136274520)

[4.4 服务端性能优化设计 37](#_Toc136274521)

[4.4.1 入口流量限制详细设计 37](#_Toc136274522)

[4.4.2 热点数据缓存详细设计 38](#_Toc136274523)

[4.4.3 ID生成方案的详细设计 39](#_Toc136274524)

[4.5 服务端安全性设计 40](#_Toc136274525)

[4.5.1 图形验证码的详细设计 40](#_Toc136274526)

[4.5.2 鉴权认证方法详细设计 41](#_Toc136274527)

[4.6 本章小结 41](#_Toc136274528)

[5 银行秒杀系统服务端的实现 42](#_Toc136274529)

[5.1 开发环境与工具 42](#_Toc136274530)

[5.2 服务端微服务组件的实现 42](#_Toc136274531)

[5.2.1 服务注册与发现 42](#_Toc136274532)

[5.2.2 服务网关 44](#_Toc136274533)

[5.2.3 服务调用 49](#_Toc136274534)

[5.2.4 服务容错 51](#_Toc136274535)

[5.3 用户服务模块的实现 53](#_Toc136274536)

[5.3.1 用户注册 53](#_Toc136274537)

[5.3.2 用户登录 54](#_Toc136274538)

[5.3.3 个人中心 54](#_Toc136274539)

[5.3.4 用户管理 56](#_Toc136274540)

[5.4 商品服务模块的实现 56](#_Toc136274541)

[5.4.1 商品信息 57](#_Toc136274542)

[5.4.2 产品管理 58](#_Toc136274543)

[5.4.3 活动管理 59](#_Toc136274544)

[5.5 筛查服务模块的实现 59](#_Toc136274545)

[5.5.1 申请筛查 60](#_Toc136274546)

[5.5.2 决策配置 61](#_Toc136274547)

[5.6 秒杀服务模块的实现 62](#_Toc136274548)

[5.6.1 商品库存预热 62](#_Toc136274549)

[5.6.2 秒杀图形验证 63](#_Toc136274550)

[5.6.3 秒杀地址隐藏 65](#_Toc136274551)

[5.6.4 秒杀控制逻辑 66](#_Toc136274552)

[5.7 订单服务模块的实现 70](#_Toc136274553)

[5.7.1 订单创建 71](#_Toc136274554)

[5.7.2 订单信息 72](#_Toc136274555)

[5.7.3 订单取消 73](#_Toc136274556)

[5.7.4 订单管理 75](#_Toc136274557)

[5.8 支付服务模块的实现 75](#_Toc136274558)

[5.8.1 支付 76](#_Toc136274559)

[5.8.2 资金管理 76](#_Toc136274560)

[5.9 本章小结 76](#_Toc136274561)

[6 银行秒杀系统服务端的测试 78](#_Toc136274562)

[6.1 接口测试 78](#_Toc136274563)

[6.1.1 用户服务模块接口测试 78](#_Toc136274564)

[6.1.2 商品服务模块接口测试 78](#_Toc136274565)

[6.1.3 筛查服务模块接口测试 79](#_Toc136274566)

[6.1.4 秒杀服务模块接口测试 80](#_Toc136274567)

[6.1.5 订单服务模块接口测试 80](#_Toc136274568)

[6.1.6 支付服务模块接口测试 81](#_Toc136274569)

[6.2 压力测试 82](#_Toc136274570)

[6.3 本章小结 85](#_Toc136274571)

[7 总结与展望 86](#_Toc136274572)

[7.1 本文小结 86](#_Toc136274573)

[7.2 展望 87](#_Toc136274574)

[致谢 88](#_Toc136274575)

[参考文献 89](#_Toc136274576)

# 绪论

## 研究背景及意义

背景

随着中国互联网行业的迅速发展，分布式系统在各个领域得到了广泛应用，网络节点之间的高效通信和远程方法调用变得尤为重要。在这一背景下，RPC作为一种在分布式系统中应用广泛的通信协议，发挥着至关重要的作用。本论文旨在深入研究并设计一种基于TCP的RPC框架，以更好地满足中国互联网环境下的需求和挑战。

在中国互联网生态中，我面临着不断增长的用户规模、多样化的应用场景以及对通信效率和服务可靠性的高要求。因此，本研究将专注于设计一个基于TCP的RPC系统，旨在提供高效的分布式通信、实现简单且可扩展的服务调用、确保高可靠性和容错性，并推动RPC技术在中国互联网行业的广泛应用。

通过实际实现基于TCP的RPC系统，我期望能够为中国互联网应用提供更优越的解决方案，提升分布式系统的通信效率、可靠性和可扩展性。这将有助于促进中国互联网技术的进步，推动数据驱动的科学研究、促进商业应用的创新，为社会发展提供更强有力的支持。

**选题意义：**

推动分布式系统技术的进步： 通过深入研究并实现基于TCP的RPC框架，我们将为分布式系统技术的发展做出贡献。提升通信效率、服务可靠性和可扩展性将推动分布式系统的发展，并在学术界引领相关领域的研究。

满足中国互联网环境下的需求： 中国互联网用户规模巨大，应用场景丰富多样。我们的研究旨在设计适应中国互联网环境的基于TCP的RPC系统，以满足不断增长的用户需求、提高服务质量，并促进互联网产业的创新与发展。

促进数据驱动的科学研究： 通过提高分布式系统通信效率，我们有望为数据驱动的科学研究提供更强有力的支持。高效的RPC框架将有助于更快地处理大规模数据，促进科学家们在各领域的研究成果。

支持商业应用的创新： 我们的研究成果将为商业应用领域提供技术支持，帮助企业更好地构建可靠的分布式系统，推动商业服务的创新和发展。

服务社会发展的需要： 高可靠性和容错性是分布式系统的重要特征，对于社会发展至关重要。我们的研究将有助于构建更稳定、可靠的互联网基础设施，支持社会各个领域的数字化转型。

## 国内外研究现状

### 国外研究现状

远程过程调用（Remote Procedure Call，RPC）的发展历史可以追溯到20世纪80年代。

在1984年，Bruce Jay Nelson在他的博士论文《The Design and Implementation of a High-Performance Transport Protocol》中首次提出了远程过程调用（Remote Procedure Call，RPC）的概念[1]中将远程过程调用（RPC）定义为一种协议，它允许一个程序能够导致另一台主机上的程序执行过程，而执行过程的环境看起来就像是本地过程调用。RPC抽象了底层的网络通信细节，使得开发分布式系统时，远程通信可以像本地函数调用一样直观。在论文中，Nelson详细讨论了RPC的设计目标，它旨在提供一种简单、高效、通用的远程交互机制。他强调了RPC在分布式计算中的重要性，尤其是在提高网络服务效率和简化复杂网络协议栈时的作用。Nelson还探讨了RPC协议的各种实现细节，包括参数传递、错误处理、传输协议选择等方面。

1985年：Andrew Birrell和Bruce Jay Nelson在Xerox PARC（帕克研究中心）开发了一种名为“Courier”的RPC系统。这是RPC的第一个实现，它使用了一种基于消息传递的通信协议。

1991年：Sun Microsystems发布了NFS（Network File System），这是一种基于RPC的分布式文件系统。NFS的成功推动了RPC的发展，并促使其他公司和组织开始关注RPC技术。

1995年：Microsoft发布了DCOM（Distributed Component Object Model），这是一种基于RPC的分布式对象通信技术。DCOM在Windows平台上广泛应用，为RPC的发展做出了重要贡献。

2000年：Google发布了GFS（Google File System），这是一种基于RPC的分布式文件系统。GFS的成功启发了Google开发更多基于RPC的分布式系统，如MapReduce和Bigtable。

2006年：Facebook发布了Thrift，这是一种开源的RPC框架。Thrift支持多种编程语言，并提供了强大的代码生成工具，使得开发者可以轻松地定义和使用RPC接口。

2010年：Google发布了gRPC，这是一种高性能的RPC框架。gRPC使用了基于HTTP/2的协议，并支持多种编程语言。它在性能和可扩展性方面具有显著优势，被广泛应用于云计算和微服务架构。

上述的thrift和gRPC也是现在行业内使用的最多的远程调用框架。国内的很多研发的RPC框架也是学习国外优秀开源框架的思想，结合内部业务需求进行开发的。

综上所述，自从RPC思想提出到如今RPC的身影无处不在，期间经历了20多年的发展，国外顶尖互联网公司和专家非常重视RPC这项技术， RPC技术在互联网领域的应用和发展已经取得了显著的成果，并且在未来仍有巨大的发展潜力。无论是对于构建高性能、低延迟的互联网应用，还是对于构建复杂、多样化的互联网应用，RPC都将继续发挥着重要的作用。

### 国内研究现状

随着互联网产业的快速发展，中国的互联网公司面临着越来越复杂的业务需求和技术挑战。为了解决这些问题，许多公司开始研究和应用RPC技术，以实现服务化架构和微服务架构。这种趋势推动了RPC技术的研究和应用在中国的快速发展。

2008年：阿里巴巴开始开发Dubbo框架，这是中国最早的开源RPC框架之一。Dubbo引入了服务治理的概念，并提供了丰富的服务治理功能。

2010年：Dubbo框架开源，成为中国最受欢迎的RPC框架之一。许多互联网公司开始使用Dubbo构建自己的服务化架构。

Dubbo对中国互联网的发展产生了深远的影响，推动了服务化架构和微服务架构的发展，提高了系统的性能和可靠性，并推动了开源社区的发展。未来，随着技术的不断进步和应用场景的不断扩展，Dubbo在中国的互联网产业中的影响将更加广泛。

随着这么一个现象级的产品出现，国内各个公司都开始在开源框架的思想上，开发复合自己公司业务的rpc框架，2012年：Motan框架开源，提供了更加灵活的服务治理和负载均衡功能。Motan在中国的互联网公司中得到了广泛的应用。

2014年：SofaRPC框架开源，提供了更加高效的RPC通信和服务治理功能。SofaRPC在中国的互联网公司中得到了广泛的应用。

综上所述，国内的RPC框架以阿里的dubbo为一个节点，后续各种rpc框架运行而生，有服务于自己公司业务的rpc框架，也有服务于大众开发程序员的开源框架，这些框架都为国内RPC技术做出了发展。

## 本文主要工作和结构安排

### 主要工作

本课题在基于TCP通信的网络上，旨在设计并实现一个RPC远程调用系统。本人具体工作内容如下：

1. 分析并实现RPC服务端和客户端，并在分布式的环境下可以正常进行远程调用。
2. 重点调研分析RPC框架为完成通信需要的技术，并分析各项技术的可行性和实用性。
3. 重点抽象RPC各项关键原理，包括序列化、协议、注册中心、网络连接的管理等实现。
4. 重点研究服务的健康检查、负载均衡、异常重试等机制；
5. 搭建必要环境进行开发，交付程序并通过测试验证系统的表现是否达到预期。

### 结构安排

全文共分为7章，各章安排如下：

第1章：绪论。本章首先介绍了课题的研究背景与意义，其次介绍了国内外RPC框架的研究现状，最后给出了本文的主要工作和结构安排。

第2章：相关理论与技术。本章阐述了RPC框架涉及到的理论与技术，介绍了项目采用的各种协议和总体框架。

第3章：RPC框架的分析。本章首先论证了系统的可行性，其次分析了管理员和客户各自的功能需求，描述了秒杀的具体流程，最后对非功能需求作简要分析。

第4章：RPC框架的设计。本章首先提出了基于微服务的服务端软件架构，其次叙述了系统的数据库设计，再从功能模块的划分入手描述了各功能模块设计，最后重点阐述了服务端的性能优化与安全性设计。

第5章：RPC框架的实现。本章主要是在第4章的基础上，阐述了服务端微服务组件和各个功能模块的实现，其中着重介绍了秒杀相关问题的解决思路与实施方案。

第6章：RPC框架的测试。本章的主要内容是使用软件工程的测试方法验收系统。

第7章：总结与展望。本章小结本文的主要内容，展望后续的改进方向。

## 本章小结

## 本章首先介绍了秒杀的业务场景，分析了传统单体架构应用存在的缺陷和不足，说明了应用微服务的必要性，其次介绍了微服务与秒杀系统的国内外研究现状，最后对本文的主要研究工作和组织架构进行概括。

# 相关理论与技术

2.1 Spring

2.1.1 Spring简介

2.1.2 spring ioc

2.1.3 spring aop

2.2 网络通信

2.2.1 java bio

2.2.2 java nio

2.2.3 netty

2.2.4 TCP与UDP

2.3 zookeeper

2.3.1 Zookeeper简介

2.3.2 Zookeeper节点存储

2.3.3 zookeeper 一致性原理

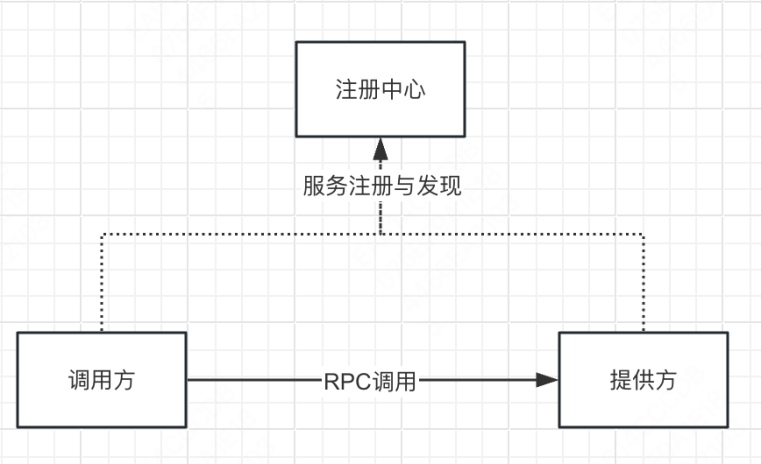
2.4 负载均衡

2.5 心跳机制

2.6 本章小结

## RPC总体结构

使用RPC进行开发时，一般开发人员调用远程接口，一般不需要关注对方的实现细节，只需要提供好接口，调用方像本地方法一样调用结果就可以获取到对应的返回值，所以对于开发者来说，只需要关注调服务调用方和服务提供方。但是另外需要注意的一点，调用方要通过何种方式知道我们服务提供方的机器ip地址，这时候，其实底层通过注册中心帮我们隐藏了具体的细节，多数框架，例如dubbo，可以通过服务名，将统一服务名对应的对应的多个ip地址，再通过负载均衡机制，找到一个具体的服务提供方进行调用。

纵观大多数RPC服务框架，例如thrift、grpc、dubbo等，基本的架构可以抽象为下图：

服务调用方：服务调用方是需要使用某个服务的客户端。当服务调用方需要使用某个服务时，它会向注册中心查询服务提供方的地址，并通过RPC调用服务提供方提供的服务。服务调用方通常会将请求数据传递给服务提供方，并接收服务提供方返回的响应数据。

服务提供方：服务提供方是提供服务的服务器。服务提供方会向注册中心注册自己提供的服务，并监听RPC调用请求。当服务提供方收到服务调用方的请求时，它会处理请求数据，并返回响应数据给服务调用方。

注册中心：注册中心是一个中心化的服务发现和注册系统。服务提供方会向注册中心注册自己提供的服务，并定期更新自己的状态信息。服务调用方在需要使用某个服务时，会向注册中心查询服务提供方的地址，并获取服务提供方的地址信息。注册中心通常会提供一个API，供服务提供方和服务调用方进行注册和查询操作。

## 调用方与提供方

调用方客户端调用方法就像处理本地方法一样，但是底层原理实际上是通过动态代理的技术，框架生成一个代理类，在代理类里面会处理对应的序列化、压缩、协议封装等逻辑，最终通过sockets连接传输到对应的服务端。

服务提供方这边会有一个监听服务监听sockets通道，当有数据到达时，socket将对应的二进制数据传输到server slub，在其中进行反序列化、协议解析，并通过反射技术调用服务端对应的方法进行执行，完成后将结果通过原路径返回给服务调用方即可。

### 动态代理

Java的动态代理[2]是Java中的一种设计模式，它允许我们在运行时创建代理对象，而不需要在编译时知道代理类的类型。动态代理可以用来实现各种功能，如日志记录、事务管理、缓存等。

在RPC框架中，服务调用方就是通过动态代理的技术，屏蔽了底层的实现细节，当程序运行时，会对调用的方法的对象生成一个代理对象，在代理中具体的逻辑就才是真正的进行网络调用的细节。

### socket

Socket是一种网络通信的抽象[3]，它提供了一种在网络上进行双向通信的接口。一个Socket由一个IP地址和一个端口号组成，它表示一个特定的网络连接。Socket可以用于建立和管理网络连接，以及发送和接收数据。

Socket的工作原理如下：

创建Socket：首先，需要创建一个Socket对象，该对象代表一个特定的网络连接。

绑定端口：然后，将Socket绑定到一个特定的端口上，以便接收来自该端口的数据。

监听连接：如果需要接收来自其他主机的连接，可以将Socket设置为监听模式，以便接收来自其他主机的连接请求。

建立连接：如果需要与其他主机建立连接，可以使用Socket的connect方法来建立连接。

发送和接收数据：一旦连接建立，就可以使用Socket的输入/输出流来发送和接收数据。

### TCP与UDP

TCP（Transmission Control Protocol）和UDP（User Datagram Protocol）是两种常见的网络传输协议[4]，它们在网络通信中扮演着不同的角色。

TCP是一种面向连接的、可靠的传输层协议。TCP提供了一种可靠的数据传输服务，它可以确保数据在网络中的正确传输，并且可以处理数据的重新传输和错误检测。TCP使用三次握手协议来建立连接，并使用滑动窗口协议来控制数据的发送和接收。TCP的主要优点是可靠性高，数据传输的顺序和完整性得到保证。

UDP是一种无连接的、不可靠的传输层协议。UDP不提供数据的可靠传输服务，它只是将数据包发送到目标主机，而不关心数据是否能够正确传输。UDP的主要优点是速度快，延迟低，适用于实时应用，如视频流和在线游戏。

本课题开发的RPC框架底层网络通信采用TCP，使用TCP协议的原因主要有以下几点：

可靠性：RPC需要确保调用的函数或方法能够正确执行，并且返回结果能够正确传输。TCP提供了可靠的数据传输服务，可以确保数据的正确传输，并处理数据的重新传输和错误检测。

顺序性：RPC需要确保调用的函数或方法的参数按照正确的顺序传输。TCP提供了数据传输的顺序保证，可以确保参数按照正确的顺序传输。

连接管理：RPC需要建立和管理网络连接，以便进行远程过程调用。TCP提供了连接管理的机制，可以建立和管理网络连接。

### 序列化

对于RPC框架来说，底层使用TCP进行网络传输，那么我们调用方法的参数、返回值等都是以对象的形式存在的，但是网络通信不能直接传输对象。所以这里我们需要使用序列化技术，将对象转化为二进制数据流后再进行数据的传输。RPC要求调用服务像调用本地方法一样，这对调用性能提出了很高的要求，对于服务调用方和服务提供方都需要使用序列化的技术，所以选择一个好的序列化工具，对rpc框架的性能有一定的影响。

### 协议

因为使用TCP进行数据的传输，传输的数据是二进制流，当服务调用方将调用的请求数据发向服务提供方时，服务提供方的监控程序接受到数据流后，不知道数据的边界在什么地方，那服务提供方怎么把二进制数据反序列化成对应的对象呢？这个时候就需要双方约定好一个“协议”。

我们可以先固定一个长度（比如 4 个字节）用来保存整个请求数据大小，这样收到数据的时候，我们先读取固定长度的位置里面的值，值的大小就代表协议体的长度，接着再根据值的大小来读取协议体的数据，整个协议可以设计成这样。

但上面这种协议，只实现了正确的断句效果，在 RPC 里面还行不通。因为对于服务提供方来说，他是不知道这个协议体里面的二进制数据是通过哪种序列化方式生成的。如果不能知道调用方用的序列化方式，即使服务提供方还原出了正确的语义，也并不能把二进制还原成对象，那服务提供方收到这个数据后也就不能完成调用了。因此我们需要把序列化方式单独拿出来，类似协议长度一样用固定的长度存放，这些需要固定长度存放的参数我们可以统称为“协议头”，这样整个协议就会拆分成两部分：协议头和协议体。

### 反射

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

反射在RPC（Remote Procedure Call）框架中起着重要的作用。RPC框架用于实现分布式系统中不同节点之间的通信。在RPC中，客户端调用远程服务器上的方法，而服务器将方法的执行结果返回给客户端。反射在RPC框架中的主要作用是动态地生成代理类和调用远程方法。

通过反射，RPC框架可以实现以下功能：

代理类生成：RPC框架可以根据接口定义动态地生成代理类。代理类可以拦截客户端的方法调用，并将方法调用转发给远程服务器。

远程方法调用：RPC框架可以使用反射来调用远程服务器上的方法。它可以通过网络传输方法的名称、参数和参数类型，并在服务器上使用反射机制找到对应的方法并执行。

参数序列化与反序列化：RPC框架需要将方法的参数从Java对象转换为字节流进行网络传输，并在远程服务器上将字节流转换回对象。反射可以帮助框架识别参数的类型，并进行相应的序列化和反序列化操作。

## 注册中心

在RPC（Remote Procedure Call）中，注册中心是一个关键组件，用于管理和维护服务提供者和服务消费者之间的关系。它充当了服务的目录和路由中心，使得服务提供者和服务消费者能够相互发现和通信。

注册中心的主要功能包括以下几个方面：

2.3.1 服务注册

服务注册：服务提供者在启动时向注册中心注册自己提供的服务，包括服务名称、网络地址、版本号等信息。

2.3.2 服务发现

服务发现：服务消费者通过向注册中心查询所需的服务，获取可用的服务提供者列表，并选择一个合适的提供者进行调用。

2.3.3 负载均衡

负载均衡：注册中心可以通过一定的负载均衡策略，将请求分发给多个服务提供者，以实现负载均衡和高可用性。

我们的系统中的某个服务的访问量特别大，我们将这个服务部署在了多台服务器上，当客户端发起请求的时候，多台服务器都可以处理这个请求。那么，如何正确选择处理该请求的服务器就很关键。假如，你就要一台服务器来处理该服务的请求，那该服务部署在多台服务器的意义就不复存在了。负载均衡就是为了避免单个服务器响应同一请求，容易造成服务器宕机、崩溃等问题。常见的负载均衡算法有：随机算法、轮训法、一致性hash环等

2.3.4 心跳检测

心跳检测是分布式系统中常用的一种机制，用于检测节点的存活状态和可用性。在RPC（Remote Procedure Call）和其他分布式系统中，心跳检测起着重要的作用，可以确保节点之间的通信正常，并及时处理节点故障。

心跳检测的基本原理是节点定期发送心跳信号（通常是网络消息或请求）给其他节点，以确认节点的存活状态。接收方节点在一定时间内收到心跳信号，则认为发送方节点是存活的；如果在指定时间内未收到心跳信号，就可以判断发送方节点可能出现了故障或不可用。基于心跳检测的结果，系统可以采取相应的措施，如重新分配任务、故障转移等。

心跳检测通常包括以下关键要素：

心跳发送：每个节点定期发送心跳信号给其他节点。心跳信号可以是一个特定的消息，也可以是简单的网络连接请求。

心跳接收：节点接收来自其他节点的心跳信号，并记录下最近接收到的心跳时间。

超时判断：每个节点都有一个超时阈值，用于判断其他节点是否超过了预设的心跳间隔时间。如果超过了超时阈值，就认为节点可能出现故障。

故障处理：一旦节点被判定为故障，系统可以采取相应的处理措施，如剔除其在注册中心的信息。

## 数据存储技术

### MySQL关系型数据库

关系型数据库（Relational Database）是一类以关系模型组织数据的数据库，由多张能互相联接的二维行列表组成，具有结构简单、易于维护等优点。MySQL因其开源、轻量级和跨平台特性，自诞生以来一直最流行的关系型数据库之一。笔者本次采用MySQL存储数据。

### Redis非关系型数据库

非关系型数据库（Non-Relational Database）泛指非关系型的数据库，具备可扩展、高性能等特性，Redis是其中的代表之一。Redis是键值（Key-Value）数据库，也可作为中间件或缓存使用。当作为缓存使用时，Redis不仅具有事务控制、持久化和主从复制等实用机制，还支持对多种数据结构的操作[24]，具备较好的读写性能，能很好地适应秒杀场景的要求。

Spring Data Redis用于在Spring项目中提供对Redis操作的支持，不仅提供自动管理连接池的能力，还由lettuce驱动高度封装了Redis底层开发包，可方便地通过RedisTemplate对Redis进行诸如对象缓存等各项操作。

### Zookeeper

ZooKeeper是一个开源的分布式协调服务，最初由雅虎公司开发，现在由Apache软件基金会进行维护。它提供了一个高度可靠的分布式环境，用于协调和管理分布式系统中的各种任务和配置。ZooKeeper的设计目标是提供简单而强大的分布式协调服务，使得开发人员可以专注于应用程序的逻辑而不必处理底层的复杂性。

本项目中的注册中心实现会使用到Zookeeper组件，

ZooKeeper中的节点（node）实际上是一个树形目录结构，用于存储和组织数据。每个节点都可以有子节点，形成一个树形结构。这种树形结构使得ZooKeeper能够有效地组织和管理数据。

在ZooKeeper中，节点可以存储数据，并且可以设置观察者（watcher）来监听节点的变化。当节点的数据发生变化时，观察者会收到通知，从而使得客户端能够及时获取最新的数据。例如当节点挂掉后，我们可以通过zookeeper的监听机制将将对应的节点数据剔除。

ZooKeeper采用了主从复制的架构，提供了高可用性的保证。在ZooKeeper集群中，有一个主节点和多个从节点。主节点负责处理客户端的请求，并将数据存储在内存中。从节点负责备份数据，并在主节点出现故障时接管。这种架构设计使得ZooKeeper能够提供高可用性的服务，即使主节点出现故障，从节点也能够快速接管，确保RPC项目的注册中心的可用性。

ZooKeeper的存储节点和高可用性还保证了数据的一致性。在ZooKeeper中，数据的更新和删除操作都是原子性的，即只有所有的从节点都成功复制了数据，才会将数据更新或删除。这种设计保证了数据的一致性，使得RPC项目的注册中心能够在分布式环境下正常运行。

## 本章小结

本章介绍了一个RPC框架的理论知识，阐述了RPC中关键的组件，包括服务提供方、服务调用方和注册中心。还包括其中的网络协议、序列化、负载均衡等技术原理，为后续的开发工作打下基础。

# RPC框架实现的分析

## 可行性分析

### 技术可行性

目前，上述技术已经由开源社区进行了维护，并且在工业界也得到了广泛的认可和应用。这表明这些技术已经相对成熟，具备了实施项目的技术基础。由于采用的技术是开源免费的，这意味着不需要支付昂贵的许可费用，从而降低了项目的成本。同时，开源技术也意味着有一个庞大的社区支持和更新，可以获得更多的技术支持和帮助。由于技术是开源免费的，我们不需要担心侵权的风险。这为项目的实施提供了法律保障，避免了潜在的法律纠纷。在限制条件下，我们需要确保必要的软件和硬件资源能够支持系统的开发工作。这包括硬件设备、操作系统、开发工具等。我们需要评估这些资源的可用性和可靠性，以确保项目的顺利实施。

### 操作可行性

从操作层面来讲，对于服务调用方，只需要引入rpc jar包就可以调用远程服务，不需要关注rpc本身的网络通信、反射、序列化等实现逻辑，对用户透明。

对于服务提供方来讲，用户只需要制定相应的接口，引入jar包，后续调用远程方法只需要像调用本地方法一样简单。

### 经济可行性

由于上述工具已经开源，我们不需要支付昂贵的许可费用。这意味着我们可以节约大量的成本，从而降低项目的总体成本。本系统能够按需部署，这意味着我们可以根据实际需求进行灵活的部署和扩展。这不仅可以降低项目的运维成本，还可以提高生产效率，从而带来更高的经济效益。若系统正式部署上线，它将为我们带来可观的经济效益。这可能包括提高生产效率、降低运维成本、增加收入等方面。因此，从长远来看，这个项目具有较好的经济价值。我们还需要考虑项目的投资回报率。通过评估项目的收益和成本，我们可以计算出项目的投资回报率。如果投资回报率较高，那么这个项目在经济上是可行的。

## 功能需求分析

本项目时基于TCP的RPC框架实现，为分布式微服务中两个服务之间的通信提供可控性服务。目前系统涉及客户端（服务调用方）和服务端（服务提供方）两个角色，以下两节将通过用例设计说明两个角色与系统各功能单元之间的关系。

### 客户端能需求分析

客户端使用RPC框架服务，涉及到方法调用、请求重试、服务发现、日志监控

和网络性能。

管理员使用秒杀系统后台管理服务（图 3.1），包括用户管理、产品管理、活动管理、决策配置、订单管理和资金管理。

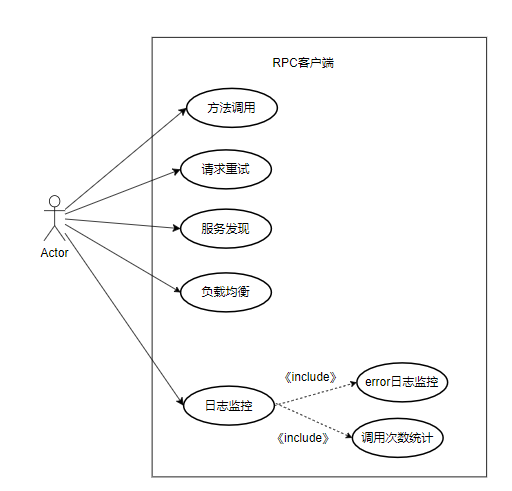


图 3.1 管理员用例图

其中，方法调用的用例包括调用远程方法并得到对应的返回值；请求重试涉及到当方法调用失败后，客户端如何进行的重试策略；服务发现用例指的是客户端需要感知到自己调用服务方的服务列表即ip地址，并需要提供负载均衡策略调用具体的一台机器服务；日志监控用例包括error日志监控和调用次数的监控。

### 服务端功能需求分析

用户使用服务端服务发布、请求处理、并发处理、错误处理、监控和日志功能。

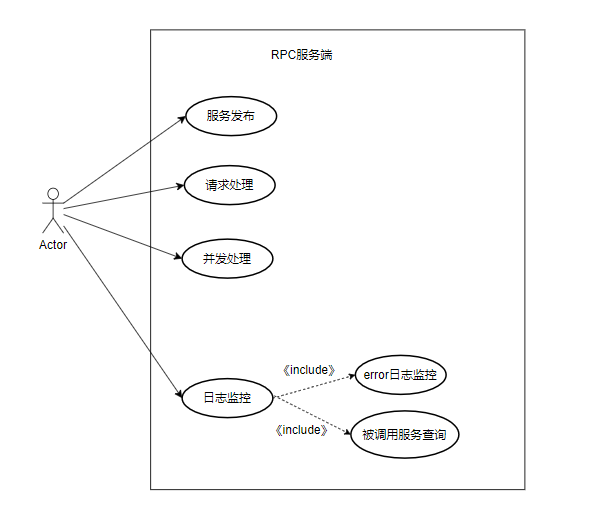


图 3.2 客户用例图

其中，服务发布包括将自己的服务方法签名和机器标识注册到注册中心，请求处理包括处理来自不同客户端发来的方法执行请求，并发处理用于处理同时接收多个请求时的并发安全处理，日志监控包括error日志的处理和使用本服务的客户端列表查询，

### 客户端调用服务端流程分析

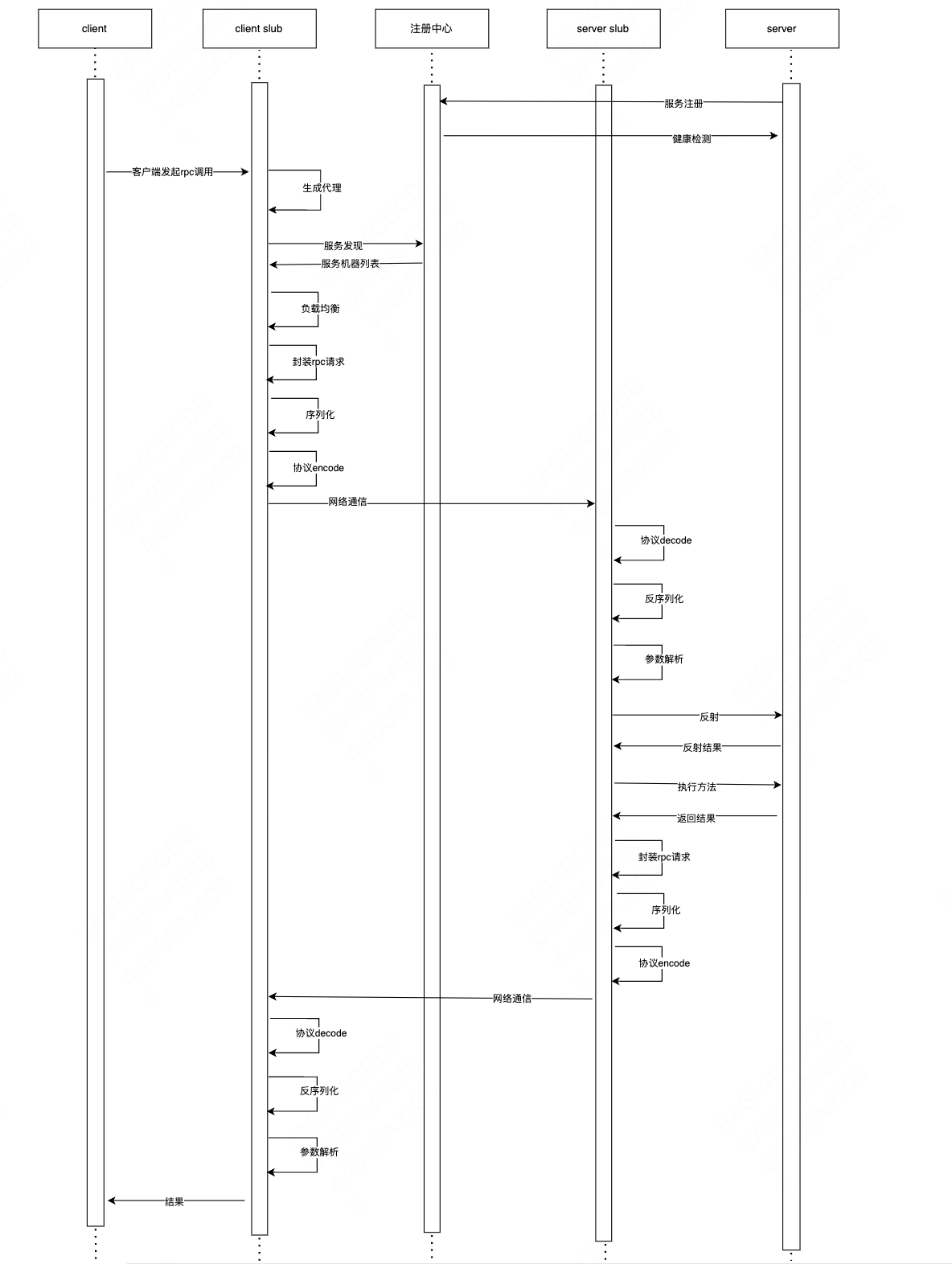


图 3.3 秒杀活动图

RPC框架的核心是客户端成功调用服务端的服务。根据UML活动图（图 3.3）的不同职责划分为server、server slub、注册中心、client和client slub五个泳道。一次rpc的调用成功，需要依次经历以下步骤：

1. client调用接口方法，client slub生成代理对象
2. client slub根据配置从注册中心获取调用目标机器列表。
3. client slub使用负载均衡功能获取到具体的机器。
4. client slub 封装请求体，并进行序列化、压缩、协议encode等后进行TCP网络传输。
5. server slub接受请求后，对请求参数进行协议decode、解压、反序列化等操作后，封装对应的rpc请求体。
6. server slub 根据rpc请求体中的请求签名，利用反射技术，拿到server的反射对象，并调用具体方法的执行，最终拿到执行结果
7. server slub根据执行结果，封装rpc响应，并进行序列化、压缩、协议encode，最后通过TCP返回给client slub。
8. client 通过协议decode、解压、反序列化等操作后，返回数据给client。

以上对于用户使用client来说，屏蔽了client slub细节，使得调用远程方法像调用本地方法一样方便。

## 非功能需求分析

### 性能效率需求

RPC框架的性能效率需求主要包括响应时间、吞吐量和资源利用率等方面。响应时间是指从客户端发起请求到服务端返回结果的时间，需要尽可能短以提供流畅的用户体验。吞吐量是指在单位时间内处理的请求数量，需要尽可能高以提供高并发处理能力。资源利用率是指RPC框架在运行过程中对系统资源的使用情况，需要尽可能低以节省资源并提高系统的稳定性和可靠性。

### 可用性需求

RPC框架的可用性需求主要包括故障恢复能力、容错能力和高可用性等方面。故障恢复能力是指在出现故障时，RPC框架能够快速恢复正常运行，减少服务中断时间。容错能力是指在出现错误或异常时，RPC框架能够正确处理并继续运行，避免系统崩溃。高可用性是指RPC框架能够在长时间内保持正常运行，提供稳定可靠的服务。

### 安全性需求

RPC框架的安全性需求主要包括数据加密、访问控制和防御攻击等方面。数据加密是指在传输过程中对数据进行加密，保护数据的安全性和隐私性。访问控制是指对服务端的访问进行权限控制，只允许授权用户访问。防御攻击是指对常见的网络攻击进行防御，如拒绝服务攻击、跨站脚本攻击等。

### 可扩展性需求

RPC框架的可扩展性需求主要包括横向扩展、纵向扩展和服务发现等方面。横向扩展是指通过增加服务器数量来提高系统的处理能力，需要RPC框架支持负载均衡和服务发现。纵向扩展是指通过增加服务器的性能来提高系统的处理能力，需要RPC框架支持服务端的性能优化和资源管理。服务发现是指在服务端有多个实例时，客户端能够自动发现并选择合适的服务端进行请求。

### 其他非功能需求

除了上述需求外，RPC框架还需要考虑其他非功能性需求，如易用性、可维护性和可测试性等。易用性是指RPC框架的使用方便，易于上手和操作。可维护性是指RPC框架的代码结构清晰，易于维护和升级。可测试性是指RPC框架支持单元测试、集成测试和性能测试等，以确保系统的质量和稳定性。

## 本章小结

本章首先从可行性分析入手叙述了系统在技术、操作和经济层面的可行性，其次分别对RPC框架涉及的几个服务端和客户端进行需求分析，再在此基础上分析了RPC方法调用的流程，最后简要分析了系统的非功能需求，包括性能效率、可用性、安全性和可扩展性等。

# 银行秒杀系统服务端的设计

## 服务端软件架构设计

图示

描述已自动生成

图 4.1 软件系统架构图

结合前期的文献调研，吸收传统单体应用的构建经验，提出服务端的软件架构设计（图 4.1）。本系统划分为客户端层、边界层、服务层和平台层[25]四个层级。其中，客户端层涵盖了所有会与服务端进行交互的客户端应用，包括PC端网页和手机、平板等移动端应用。前端请求经过反向代理分发服务到达边界层，在边界层客户端通过已定义的边界和应用进行交互。在本系统中这部分是由负责客户端和后台系统的两个微服务网关构成，这也是进入微服务内部边界的门户。通过这一边界暴露底层的各个功能，来满足外部客户的需求。服务层主要是开发的各个应用级服务，需要借助下层的平台层支持，以提供系统所需的业务功能。

## 数据库设计

### 实体设计

经3.2节的功能需求分析，抽象并得到个人账户、账户流水、申请记录、银行账户、订单、产品、规则、秒杀产品、管理员和用户实体如下。



图 4.2 个人账户实体

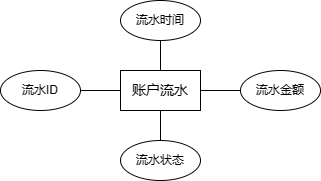


图 4.3 账户流水实体

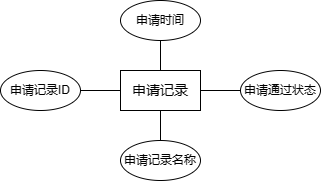


图 4.4 申请记录实体

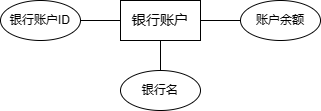


图 4.5 银行账户实体

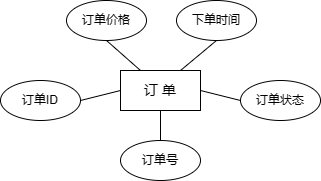


图 4.6 订单实体

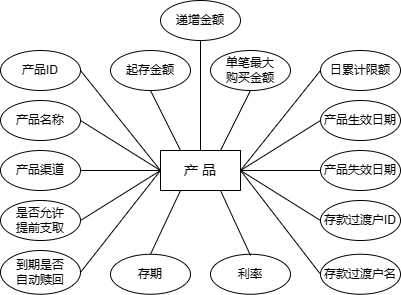


图 4.7 产品实体

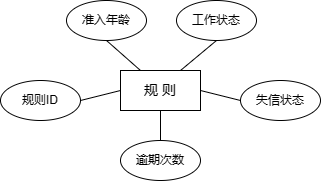


图 4.8 规则实体

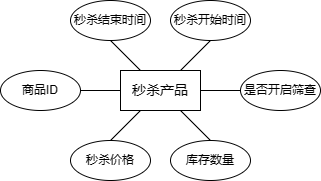


图 4.9 秒杀产品实体

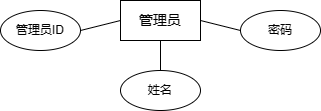


图 4.10 管理员实体

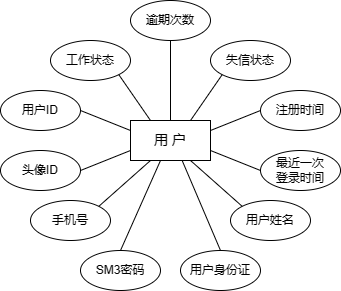


图 4.11 用户实体

### 概念模型E-R图设计

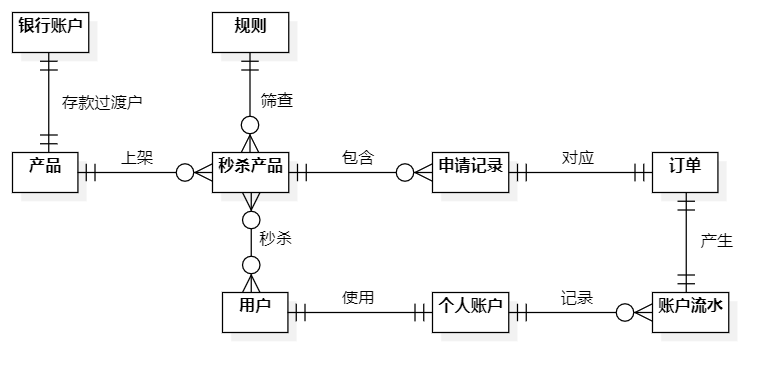


图 4.12 E-R图

如图 4.12所示，使用E-R图（Entity-Relationship Diagram，实体-联系图）描述系统中各实体之间的关系。具体情况如下：

1. 每一个用户都有自己独立的个人账户，每一个产品也都有自己在银行的存款过渡户，用户和个人账户、产品和银行账户之间均为一对一关系；
2. 每一件存款产品可独立作为不同商品上架成为秒杀产品，而每件正在秒杀的产品只能对应一件存款产品，产品和秒杀产品的关系是一对多的；
3. 一件秒杀产品对应一套准入规则，但同一套规则可以适用于多个不同的秒杀产品，因此秒杀产品和规则的关系是多对一的；
4. 一个用户在满足准入规则时，能同时参与多个秒杀产品活动，而一件秒杀产品也能够被多个用户秒杀，因此用户和秒杀产品之间的关系是多对多的；
5. 一件秒杀产品可以包含多条申请记录，而同一条申请记录只能从属于对应的秒杀产品，是为一对多关系；
6. 一条申请记录只能对应一笔订单，一笔订单也对应一条申请记录，申请记录和订单之间是一对一关系；
7. 一笔订单会产生一条账户流水，而同一条账户流水只会被一条订单产生，订单和账户流水是一对一关系；
8. 一个个人账户可以记录多条账户流水的信息，而一条账户流水只能由一个账户记录，因此个人账户与账户流水之间是一对多的关系。

### 物理设计

本系统共涉及10张数据库表，分别是个人账户表、账户流水表、申请记录表、银行账户表、订单表、产品表、规则表、秒杀产品表、管理员表和用户表。对应数据库表如下。

表 4.1 个人账户表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 1 | account\_id | bigint | 20 | N | Y | 账户ID |
| 2 | user\_id | bigint | 20 | N | N | 用户ID |
| 3 | balance | decimal | 10 | N | N | 账户余额 |
| 4 | create\_date | datetime |  | N | N | 创建日期 |
| 5 | update\_date | datetime |  | N | N | 更新日期 |

表 4.2 账户流水表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 1 | transaction\_id | bigint | 20 | N | Y | 流水ID |
| 2 | account\_id | bigint | 20 | N | N | 账户ID |
| 3 | user\_id | bigint | 20 | N | N | 用户ID |
| 4 | order\_no | varchar | 255 | N | N | 订单号 |
| 5 | transaction\_status | int | 1 | N | N | 流水状态 |
| 6 | price | decimal | 10 | N | N | 流水金额 |
| 7 | date | datetime |  | N | N | 流水时间 |
| 8 | create\_date | datetime |  | N | N | 创建日期 |
| 9 | update\_date | datetime |  | N | N | 更新日期 |

表 4.3 申请记录表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 1 | id | bigint | 20 | N | Y | 申请记录ID |
| 2 | user\_id | bigint | 20 | N | N | 用户ID |
| 3 | seckill\_product\_id | bigint | 20 | N | N | 秒杀产品ID |
| 4 | name | varchar | 255 | N | N | 申请记录名称 |
| 5 | apply\_date | datetime |  | N | N | 申请时间 |

表 4.3（续）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 6 | pass | tinyint | 1 | N | N | 申请通过状态 |
| 7 | create\_date | datetime |  | N | N | 创建日期 |
| 8 | update\_date | datetime |  | N | N | 更新日期 |

表 4.4 银行账户表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 1 | id | bigint | 20 | N | Y | 银行账户ID |
| 2 | name | varchar | 255 | N | N | 银行名 |
| 3 | balance | decimal | 10 | N | N | 账户余额 |
| 4 | create\_date | datetime |  | N | N | 创建日期 |
| 5 | update\_date | datetime |  | N | N | 更新日期 |

表 4.5 订单表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 1 | order\_id | bigint | 20 | N | Y | 订单ID |
| 2 | order\_no | bigint | 20 | N | N | 订单号 |
| 3 | order\_status | bigint | 20 | N | N | 订单交易状态 |
| 4 | user\_id | bigint | 20 | N | N | 用户ID |
| 5 | date | datetime |  | N | N | 下单时间 |
| 6 | price | decimal | 10 | N | N | 订单价格 |
| 7 | product\_id | bigint | 20 | N | N | 产品ID |
| 8 | create\_date | datetime |  | N | N | 创建日期 |
| 9 | update\_date | datetime |  | N | N | 更新日期 |

表 4.6 产品表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 1 | product\_id | bigint | 20 | N | Y | 产品ID |
| 2 | product\_name | varchar | 255 | N | N | 产品名称 |
| 3 | cumulative\_daily\_limit | bigint | 20 | Y | N | 日累计限额 |

表 4.6（续）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 4 | is\_automatic\_redemption | tinyint | 4 | N | N | 到期是否自动赎回 |
| 5 | incremental\_amount | decimal | 10 | Y | N | 递增金额 |
| 6 | transit\_account\_id | bigint | 20 | Y | N | 存款过渡户ID |
| 7 | transit\_account\_name | varchar | 255 | Y | N | 存款过渡户名 |
| 8 | effective\_date | datetime |  | N | N | 产品生效日期 |
| 9 | expiry\_date | datetime |  | N | N | 产品失效日期 |
| 10 | maximum\_single\_purchase\_amount | decimal | 10 | Y | N | 单笔最大购买金额 |
| 11 | minimum\_amount | decimal | 10 | Y | N | 起存金额 |
| 12 | is\_withdrawing\_in\_advance\_available | tinyint | 4 | Y | N | 是否允许提前支取 |
| 13 | product\_channel | varchar | 255 | Y | N | 产品渠道 |
| 14 | deposit\_term | datetime |  | Y | N | 存期 |
| 15 | interest\_rate | decimal | 10 | Y | N | 利率 |
| 16 | create\_date | datetime |  | N | N | 创建日期 |
| 17 | update\_date | datetime |  | N | N | 更新日期 |

表 4.7 规则表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 1 | id | bigint | 20 | N | Y | 规则ID |
| 2 | overdue\_count | int | 11 | N | N | 逾期次数 |
| 3 | is\_poor\_credit | tinyint | 1 | N | N | 失信状态 |
| 4 | work\_status | tinyint | 1 | N | N | 工作状态 |
| 5 | age | int | 20 | N | N | 准入年龄 |
| 6 | create\_date | datetime |  | N | N | 创建日期 |
| 7 | update\_date | datetime |  | N | N | 更新日期 |

表 4.8 秒杀产品表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 1 | seckill\_product\_id | bigint | 20 | N | Y | 商品ID |
| 2 | product\_id | bigint | 20 | N | N | 产品ID |
| 3 | seckill\_price | decimal | 10 | N | N | 秒杀价格 |
| 4 | stock\_amount | int | 20 | N | N | 库存数量 |
| 5 | start\_date | datetime |  | N | N | 秒杀开始时间 |
| 6 | end\_date | datetime |  | N | N | 秒杀结束时间 |
| 7 | screen | tinyint | 1 | Y | N | 是否开启筛查 |
| 8 | rule\_id | bigint | 20 | Y | N | 筛查规则ID |
| 9 | create\_date | datetime |  | N | N | 创建日期 |
| 10 | update\_date | datetime |  | N | N | 更新日期 |

表 4.9 管理员表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 1 | user\_id | bigint | 20 | N | Y | 管理员ID |
| 2 | name | varchar | 255 | N | N | 姓名 |
| 3 | password | varchar | 255 | N | N | 密码 |

表 4.10 用户表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 1 | user\_id | bigint | 20 | N | Y | 用户ID |
| 2 | mobile | varchar | 255 | N | Y | 手机号 |
| 3 | password | varchar | 255 | N | N | SM3密码 |
| 4 | name | varchar | 255 | N | N | 用户姓名 |
| 5 | identity\_card | varchar | 255 | Y | N | 用户身份证 |
| 6 | head | varchar | 255 | Y | N | 头像ID |
| 7 | register\_date | datetime |  | Y | N | 注册时间 |
| 8 | last\_login\_date | datetime |  | Y | N | 最近一次登录时间 |
| 9 | overdue\_count | int | 11 | Y | N | 逾期次数 |
| 10 | work\_status | tinyint | 1 | Y | N | 工作状态 |
| 11 | is\_poor\_credit | tinyint | 1 | Y | N | 失信状态 |

表 4.10（续）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **编号** | **名称** | **数据类型** | **长度** | **允许空值** | **主键** | **说明** |
| 12 | create\_date | datetime |  | N | N | 创建日期 |
| 13 | update\_date | datetime |  | N | N | 更新日期 |

## 服务端功能模块设计

依照3.2和3.3节的需求分析，将本系统的总功能（图 4.13）划分为用户服务、商品服务、筛查服务、秒杀服务、订单服务和支付服务6个功能模块。

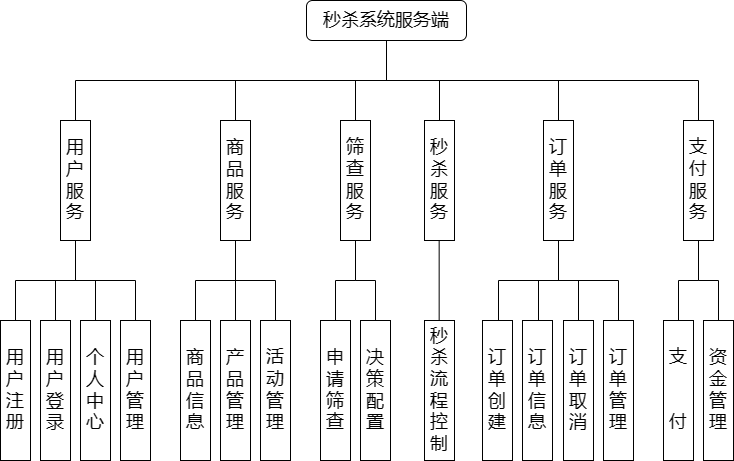


图 4.13 系统服务端功能结构图

### 用户服务模块详细设计

用户服务模块囊括了面向客户端的用户功能和面向后台系统的用户管理功能，其中用户功能包括用户注册、用户登录和个人中心。

1. **用户注册功能**

所有客户均需注册以参与秒杀活动，未注册的用户不能参与秒杀活动。用户注册流程（图 4.14）包括以下步骤：

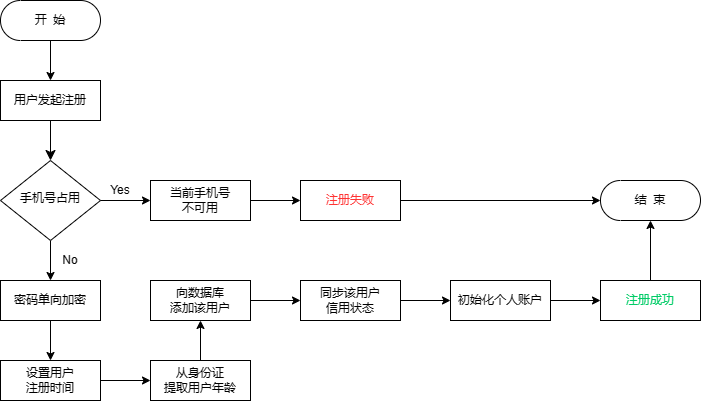


图 4.14 用户注册流程图

① 用户发起注册。用户注册时需填写的表单信息为：用户手机号、用户密码、用户姓名和用户身份证；

② 检查手机号占用情况。手机号是唯一的，若重复注册则返回“手机号已被占用”信息，视为注册失败；

③ 对密码进行单向加密；

④ 设置该用户的注册时间；

⑤ 从身份证中提取用户年龄。身份证中已包含了用户出生日期，因此无需注册时额外填写年龄信息；

⑥ 向数据库添加该用户；

⑦ 同步用户的信用状态。此部分应由相应外部系统的API提供，信用状态信息包括：用户逾期记录信息、用户是否为失信执行人及用户的工作状态；

⑧ 初始化用户的个人账户，此账户为系统创建的内部账户。上述操作均完成后向客户端返回信息，注册成功。

1. **用户登录功能**

用户登录功能包括客户登录（图 4.15）和管理员登录。其大体流程基本相同，不同之处在于管理员使用预设的账户名而非手机号登录。此处仅叙述客户登录流程，包括以下步骤：

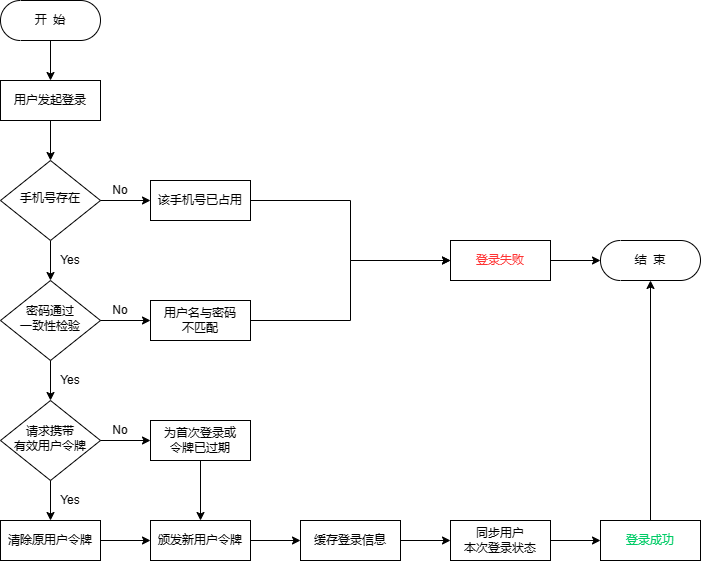


图 4.15 用户登录流程图

① 用户发起登录。用户登录时需填写的表单信息为：用户手机号和用户密码；

② 检查手机号是否存在。若无匹配的手机号，则返回“手机号不存在”信息，视为登录失败；

③ 检查密码是否通过一致性检验。若用户输入密码与系统存储密码不符，则返回“用户名与密码不匹配”信息，此情形下亦视为登录失败；

④ 检查客户端请求携带令牌的有效性。若令牌有效，则应先清除原令牌，颁发新的令牌，并重置令牌的过期时间；否则直接颁发新的令牌；

⑤ 缓存该用户的登录信息；

⑥ 同步用户本次的登录状态。上述操作均完成后向客户端返回信息，登录成功。

1. **个人中心功能**

个人中心功能分为个人信息查看与密码修改两部分。可供查看的个人信息包括但不限于下述字段：用户手机号、用户逾期记录信息、用户是否为失信执行人、用户工作状态、用户注册时间、用户上一次登录时间等。

1. **用户管理功能**

用户管理功能仅可由管理员在后台系统操作。管理员在登录后可分页查看用户信息列表，并对普通用户执行删除或更新操作。默认普通用户仅可由客户自行注册，管理员无新增用户的权限。

### 商品服务模块详细设计

商品服务模块主要为面向客户端的商品信息功能和面向后台系统的商品管理功能，其中商品管理功能包括产品管理和活动管理。

1. **商品信息功能**

商品信息功能分为商品列表查看与商品详情查看（图 4.16）两部分。服务端应提供商品信息列表，供客户端在首页展示所有已设置秒杀活动的存款产品；商品详情则是在用户进入商品详情页后，由服务端提供。此商品详情信息除应包含商品的全部信息外，还应包含活动的秒杀状态与活动结束的倒计时时间。由于商品列表查看的流程较为简单，此处叙述商品详情查看流程，包括以下步骤：

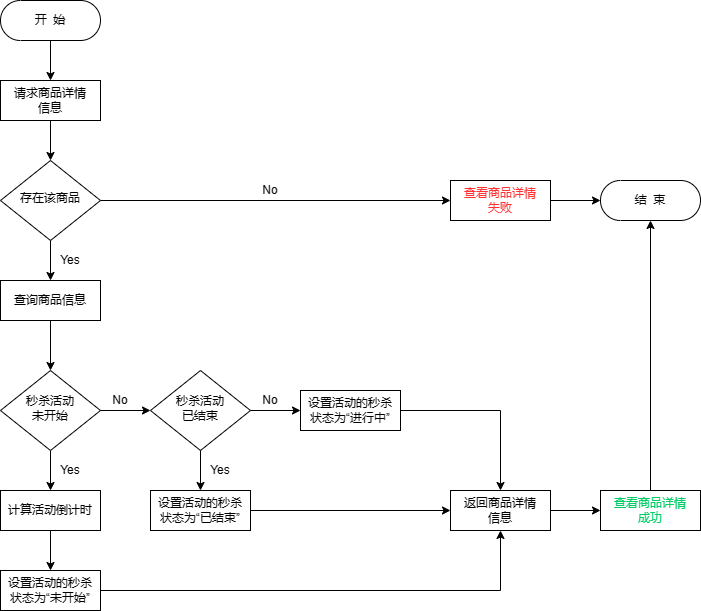


图 4.16 商品详情查看流程图

① 用户请求商品详情信息；

② 检查所查看的商品是否存在。若商品不存在，则查看商品详情失败；

③ 查询该商品信息；

④ 根据当次请求的时间，判定活动此时的秒杀状态，并设置活动倒计时；

⑤ 返回商品详情信息，操作成功。

1. **产品管理功能**

产品管理功能仅可由管理员在后台系统操作。管理员在登录后可分页查看产品信息列表，并对银行存款产品执行新增、删除和更新操作。

1. **活动管理功能**

类似地，活动管理功能仅可由管理员在后台系统操作。管理员在登录后可分页查看活动列表，并执行活动发布、更新和取消推送操作。一次秒杀活动对应一款存款产品，活动发布时需至少为产品指定以下信息：秒杀价格、秒杀库存、秒杀活动的准入规则、秒杀活动的起止时间。秒杀活动发布后，该存款产品即作为商品上架。

### 筛查服务模块详细设计

筛查服务模块主要涉及面向客户端的申请筛查功能和面向后台系统的决策配置功能。

1. **申请筛查功能**

申请筛查功能是依照预设的准入规则筛查用户是否有参与秒杀活动的条件。其中，对于未开启筛查机制的秒杀活动，仅记录用户的申请记录；对于已有筛查机制的秒杀活动，记录用户的申请记录并筛查，通过筛查的用户方可进入活动。对于后者，当筛选规则均处于开启状态时，申请筛查流程（图 4.17）包括以下步骤：

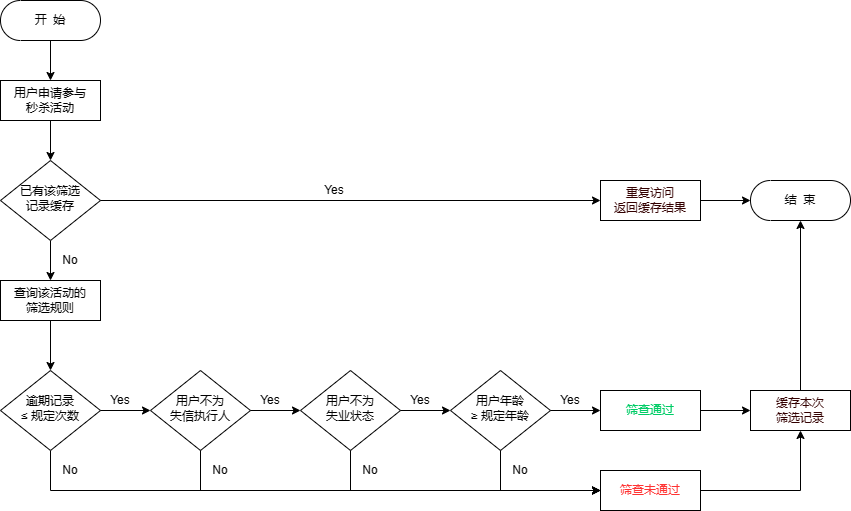


图 4.17 申请筛查流程图

① 用户申请参与秒杀活动；

② 检查该用户所参与的活动是否已存在筛选记录。若不存在，则本次为该用户首次参与该活动；反之为用户在短时间内重复操作，可直接返回上一次的缓存结果；

③ 查询该活动的筛选规则，筛选规则包括以下4项：用户的逾期次数、用户是否为失信执行人（限制失信/不限制）、用户工作状态（限制失业与无业/不限制）和用户年龄；

④ 首先检查用户的逾期次数是否不高于规定次数。高于规定逾期次数的用户不得参与本次秒杀活动；

⑤ 其次检查用户是否为失信执行人。若当前用户为失信执行人，则不得参与本次秒杀活动；

⑥ 再检查用户的工作状态，失业或无业用户也无法参与；

⑦ 最后，还应检查用户年龄是否满足要求。应拒绝年龄低于规定年龄的用户参与秒杀活动；

⑧ 仅④-⑦步骤全部通过的用户通过筛选，可参与本次秒杀活动。对于任意一项不满足者，其申请筛查均未通过；

⑨ 缓存该用户对本次秒杀活动的筛选记录。

1. **决策配置功能**

决策配置功能仅可由管理员在后台系统操作，包括准入规则管理和申请记录管理两部分，主要是对准入规则和申请记录的增删改查操作。申请筛查功能中使用的准入规则由管理员预先添加；待到新建秒杀活动时，选择适用的准入规则完成决策配置。

### 秒杀服务模块详细设计

秒杀服务模块作为聚合服务，仅处理客户端的秒杀逻辑。主要功能为秒杀流程控制，处理3.2.3节秒杀流程分析中叙述的商品秒杀步骤。秒杀控制流程（图 4.18）包括以下步骤：

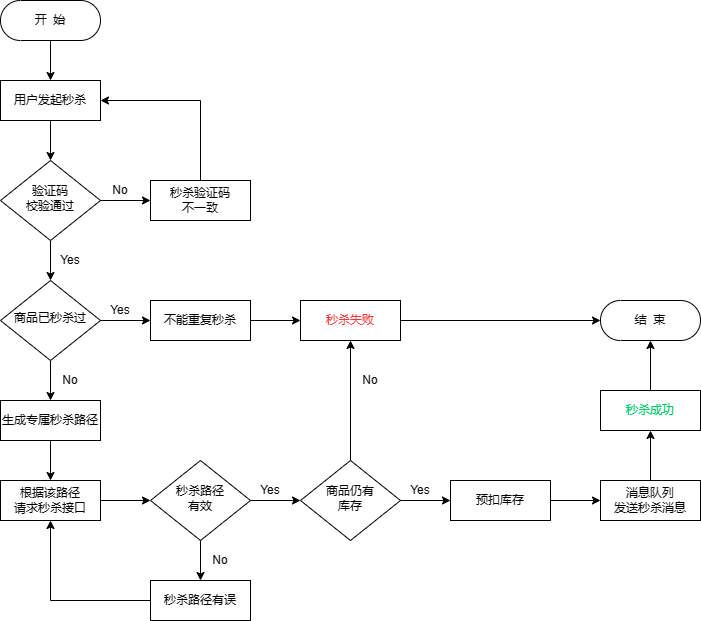


图 4.18 秒杀流程控制流程图

① 用户发起秒杀；

② 检查图形验证码校验是否通过：若验证码校验失败，应重复步骤①以获取新的图形验证码。图形验证码的设计在4.5.1节中作具体阐述；

③ 检查对于该用户本次活动的商品是否已秒杀过，重复秒杀视为秒杀失败；

④ 为用户生成一份专属的秒杀链接，该链接对于其他用户隐藏；

⑤ 根据步骤④生成的路径请求实际的秒杀接口；

⑥ 检查该秒杀路径是否有效；

⑦ 检查商品是否仍有库存。若已售罄，则秒杀失败；

⑧ 缓存预扣库存；

⑨ 向消息队列中发送秒杀消息，本次秒杀成功。

### 订单服务模块详细设计

订单服务模块涵盖了面向客户端的订单功能的和面向后台系统的订单管理功能，其中订单功能包括订单创建、订单信息和订单取消。

1. **订单创建功能**

订单在监听队列接收到秒杀服务发送的秒杀消息后异步创建。订单创建流程（图 4.19）包括以下步骤：

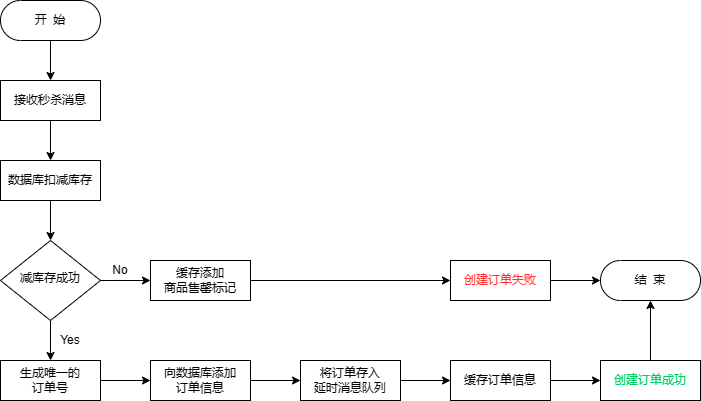


图 4.19 订单创建流程图

① 从消息队列中接收秒杀信息；

② 数据库扣减所秒杀商品库存；

③ 检查减库存是否成功。若扣减库存失败，则说明当前商品已售罄，向缓存添加标记，视为订单创建失败；

④ 扣减库存成功后，应为订单生成唯一的订单号；

⑤ 向数据库添加该订单。此时订单的交易状态为“未付款”状态；

⑥ 将订单存入延迟（死信）队列。此队列的TTL为10分钟，若用户在规定时间内未完成支付，超时订单将被自动回收；

⑦ 缓存该订单信息，操作成功。

1. **订单信息功能**

订单信息功能分为订单列表查看与订单详情查看（图 4.20）两部分。服务端应提供订单信息列表，供客户端在订单页展示用户的所有订单；订单详情则是在用户进入订单详情页后，由服务端提供。此订单详情信息除应包含订单的全部信息外，还应包含此订单所包含的商品信息。由于订单列表查看的流程较为简单，此处叙述订单详情查看流程，包括以下步骤：

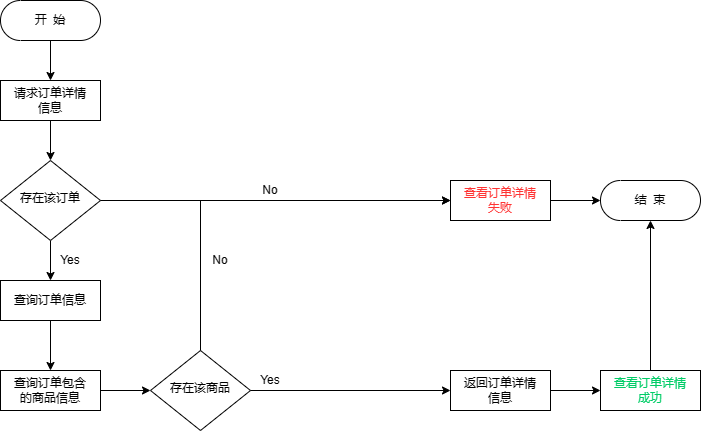


图 4.20 订单详情查看流程图

① 用户请求订单详情信息；

② 检查所查看的订单是否存在。若订单不存在，则查看订单详情失败；

③ 查询订单信息；

④ 查询该订单所包含的商品信息；

⑤ 检查是否存在该商品。若订单包含的商品不存在，亦视为查看订单详情失败；

⑥ 返回订单详情信息，操作成功。

1. **订单取消功能**

订单取消功能包括订单超时回收和手动取消订单。前者的设计基于MQ中间件，不涉及任何客户端的操作。当秒杀操作完成后秒杀消息发出，订单服务从消息队列中接收该信息创建订单，并将订单消息存入订单延迟队列。客户共计有10分钟的时间用于支付，时间耗尽后此消息将发给订单接收队列，届时将根据订单的交易状态决定是否取消订单：若用户尚未支付，则将取消此订单。

手动取消订单功能则允许用户在订单创建后，于客户端操作取消订单。手动取消订单流程（图 4.21）包括以下步骤：

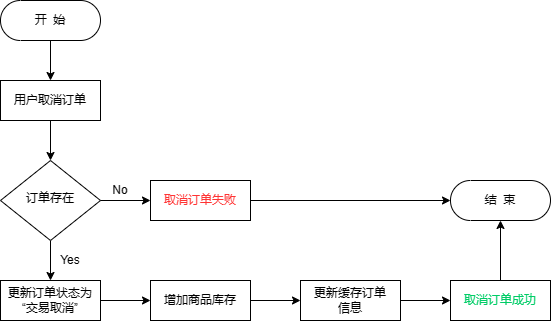


图 4.21 手动取消订单流程图

① 用户发起取消订单操作；

② 检查该订单是否存在。若订单不存在，视为取消订单失败；

③ 更新当前订单的交易状态为“取消交易”；

④ 增加数据库和缓存中的商品库存；

⑤ 更新缓存中的订单信息，操作成功。

1. **订单管理功能**

订单管理功能仅可由管理员在后台系统操作。管理员在登录后可分页查看订单列表，并执行对订单的删除和更新操作。管理员无凭空新增订单的权限。

### 支付服务模块详细设计

支付服务模块完成面向客户端的支付功能和面向后台系统的资金管理功能。

1. **支付功能**

支付功能是用户支付已创建的有效订单。前文提及订单的有效时间为10分钟，在这段时间结束后该订单将被系统自动回收，则用户无法进行支付。支付流程（图 4.22）包括以下步骤：



图 4.22支付流程图

① 用户发起支付订单操作；

② 检查该订单是否存在。若订单不存在，视为支付订单失败；

③ 检查订单的交易状态。订单的交易状态为“已支付”或“取消交易”的订单不可支付，支付操作失败；

④ 个人账户扣减订单金额；

⑤ 检查账户金额是否充足。若个人账户金额小于订单总金额，亦视为支付订单失败；

⑥ 向数据库添加其个人账户的流水信息；

⑦ 银行账户添加订单金额；

⑧ 更新当前订单的交易状态为“已付款”；

⑨ 更新缓存中的订单信息，操作成功。

1. **资金管理功能**

资金管理功能仅可由管理员在后台系统操作，包括个人账户管理、银行账户管理和账户流水管理三部分，主要涉及对应的增删改查操作。银行账户作为内部过渡户，每个存款产品均对应一个银行账户；在用户支付后，其所有金额均转账至该账户；此外，系统还会为该用户的个人账户新增一条所支付订单对应的流水记录。

## 服务端性能优化设计

### 入口流量限制详细设计

常用的限流算法包括令牌桶算法、滑动窗口算法、漏桶算法和计数器算法等。本系统采用令牌桶算法（Token Bucket Algorithm）控制进出流量的传输速率。

图 4.23是令牌桶算法的示意图。令牌以给定速率[26]恒定产生并被放入令牌桶中，过量令牌将被丢弃。处理一次请求，就从令牌桶中取出一个令牌，若令牌桶中已经没有令牌，该次请求将被拒绝。

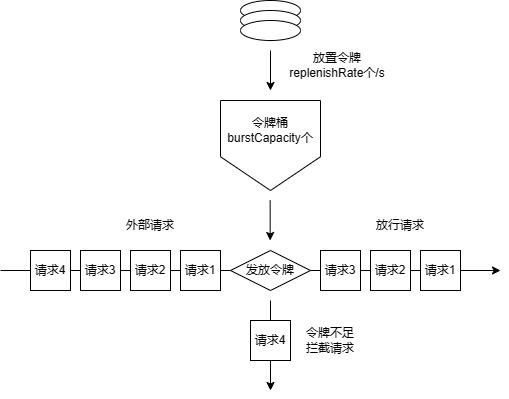


图 4.23 令牌桶算法示意图

不同于单机情况，集群系统的限流常依赖于某个中心化的组件。系统中该组件为边界层处的微服务网关，通过Spring Cloud提供的RequestRateLimiterGatewayFilterFactory过滤器工厂完成流量限制。该类使用Redis和Lua脚本实现了令牌桶算法，默认采用的速率限制器为RedisRateLimiter。

应用本算法（图 4.24）后，每当来自客户端或后台系统的请求到达服务端时，都需逐个向令牌桶请求令牌，从而限制了短时间能实际访问系统服务的请求数目，避免高额流量瞬间击穿系统。

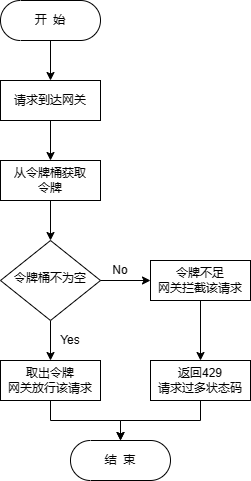


图 4.24 入口流量限制流程图

### 热点数据缓存详细设计

本系统使用Redis缓存热点对象信息，譬如用户对象、商品对象、订单对象等，利用缓存来减少对数据库的访问（图 4.25），提高系统的响应速度。

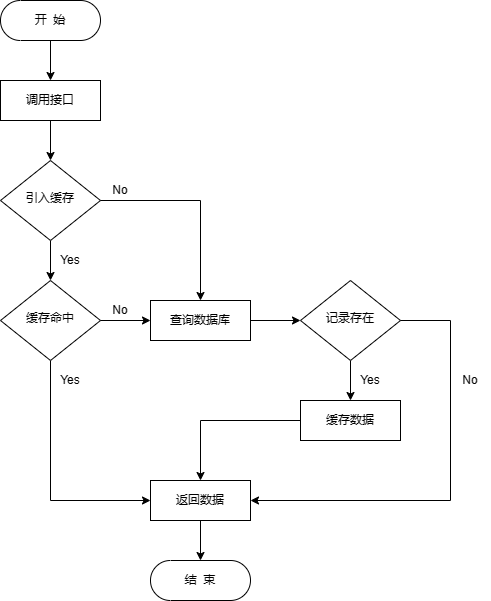


图 4.25 热点数据缓存流程图

2.4.2节叙述了NoSQL数据库Redis作为缓存的优势。为保证效率，Redis中数据均缓存在内存中[27]；不同的对象信息需先序列化后，再以JSON格式存储。对于每一个键，拆分为前缀和后缀两部分：前缀为笔者人为添加的字符串，便于区分和查找，后缀则作为查找的依据。同时，在前缀前指定对应的类名，以确定缓存信息的来源，并保证该键的唯一性。此外，基于定时过期策略为每个键指定过期时间（Time To Live，TTL）；对于热点数据，将其过期时间设置为永不过期，避免缓存失效后大量请求同时落入数据库造成缓存雪崩。项目中涉及的对象信息缓存设计如表 4.11所示：

表 4.11 对象信息

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **编号** | **键前缀（Key Prefix）** | **键后缀（Key Suffix）** | **TTL** | **说明** |
| 1 | order | userId\_productId | -1 | 订单信息 |
| 2 | orderNo | orderNo | -1 | 订单号信息 |
| 3 | product | list / productId | 60 | 活动/商品信息 |
| 4 | stock | productId | -1 | 库存信息 |
| 5 | rule | ruleId | 3600\*24 | 规则信息 |
| 6 | pass | userId\_seckillProductId | 3600\*24 | 筛查状态 |
| 7 | over | productId | -1 | 秒杀状态 |
| 8 | path | userId\_productId | 60 | 秒杀路径 |
| 9 | verifyCode | userId:productId | 300 | 秒杀验证 |
| 10 | token | token | 3600\*48 | 用户令牌 |
| 11 | uid | userId | -1 | 用户信息 |

### ID生成方案的详细设计

ID的常见分布式生成方案有UUID和雪花（Snowflake）算法等。通用唯一识别码（Universally Unique Identifier，UUID）是128位的，长度过长；其生成ID完全随机，不适用于本系统生成递增有序ID的应用场景。

相比之下，本系统基于雪花算法的ID生成方案就很好的解决了前者存在的问题。雪花算法生成的是64位ID，在Java下恰可由8字节的长整型（long）存放。



图 4.26 Snowflake算法生成的ID结构示意图

在该结构中（图 4.26），最终生成的ID由符号位、时间戳、数据中心ID（datacenterId）、机器ID（workerId）和序列号五部分组成。其中，由于生成的ID为正整数，因此符号位固定为0；时间戳是将从开始到目前这段时间以时间戳存储。这样一来，每次生成ID就不是完全随机的，而是按时间顺序升序排列。再经由数据中心ID、机器ID和序列号的区分，从而保证分布式高并发环境下生成ID不会发生重复。项目中正是采用雪花算法生成随机的秒杀接口地址路径（4.3.4节秒杀流程控制步骤④）及唯一的订单号（4.3.5节订单创建步骤④）。

## 服务端安全性设计

### 图形验证码的详细设计

验证码作为一种常用的辅助安全手段, 在Web应用具有特殊的安全保障作用[28]。图形验证码用于在用户操作前作预先验证，通过验证简易四则运算的结果，强制人机交互。以秒杀验证为例，验证通过的用户方可秒杀商品。图形验证码生成流程（图 4.27）包括以下步骤：

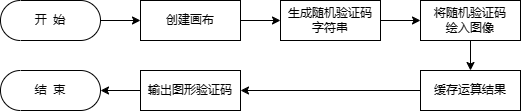


图 4.27 图形验证码生成流程图

① 创建画布，选取合适的图片大小；

② 生成随机验证码字符串。该字串为一个算式，包含三个10以内随机生成的非负整数，每两个整数间由一个四则运算符连接。

③ 将随机验证码绘入图像。

④ 缓存步骤②中算式的运算结果。

⑤ 输出图形验证码。

### 鉴权认证方法详细设计

除图形验证码措施外，本系统还基于单点登录（Single Sign On，SSO）完成请求的鉴权与认证，杜绝脚本和恶意请求。以用户登录并请求服务（图 4.28）为例：

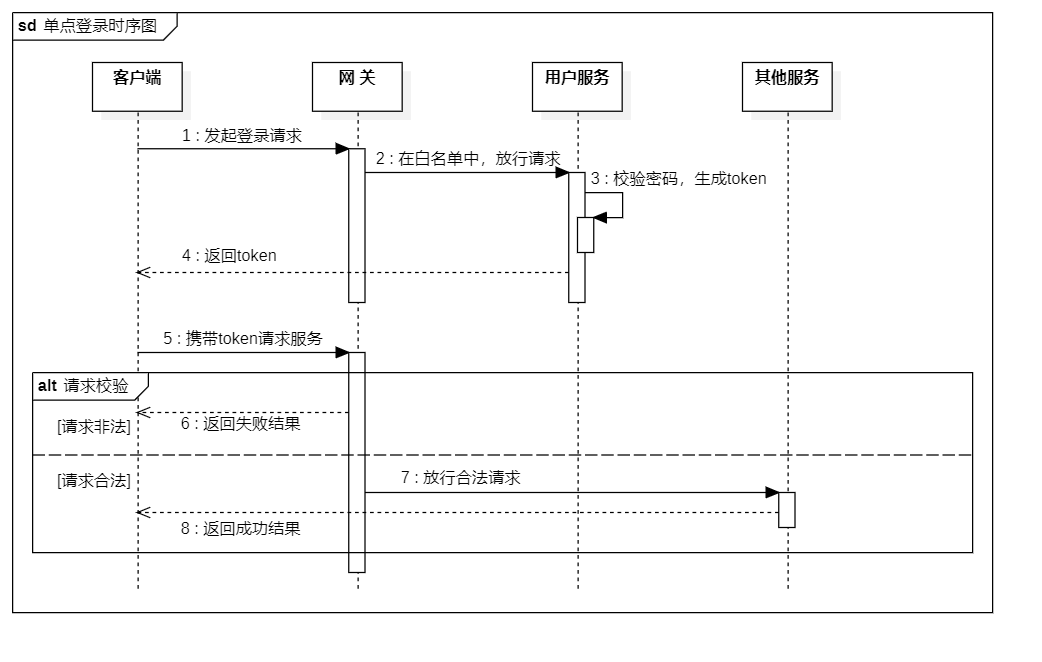


图 4.28 单点登录时序图

在该情景下，用户首次登录后将由服务端发放用户令牌，客户端保存对应的token，生成一份相同的拷贝并存入Redis；在用户后续访问中，均只由客户端携带token，并统一通过对应的路由网关进行验证。其中，在服务端对用户密码等敏感数据使用SM3国密算法加密。

目前，常用的数据加密方法分为对称加密、非对称加密和散列算法。SM3密码杂凑算法是一种散列算法，可将任意长度的消息压缩成固定长度的摘要，适用于数据的完整性校验、身份认证[29]等场景，已于2016年发布为国家密码杂凑算法标准。在用户注册后，服务端将明文密码经SM3加密后得到密文，存储在数据库中；在用户登录时，再比对验证源数据与加密数据是否一致。

## 本章小结

本章首先提出了系统的软件架构设计与数据库设计，其次明确了功能模块的划分，依次阐述了用户服务、商品服务、筛查服务、秒杀服务、订单服务和支付服务各模块的详细设计，最后重点阐述了系统的性能优化和安全策略设计，作为系统实现的指导。

# 银行秒杀系统服务端的实现

## 开发环境与工具

表 5.1 开发环境与工具

|  |  |
| --- | --- |
| **名称** | **型号/版本** |
| 处理器 | Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz |
| 内存 | 16GB 2666MHz DDR4 |
| 操作系统 | Windows 64-bit |
| 数据库 | Mysql 5.7.15，Redis 3.2.1 |
| 开发语言 | Java，JDK 1.8 |
| 集成开发环境IDE | IntelliJ IDEA 2021.1.1 |
| Web服务器 | Undertow 2.2.22.Final |

## 服务端微服务组件的实现

### 服务注册与发现

2.2.1节叙述了服务注册与发现组件Consul。在本次实践中，Consul作为注册中心集成到Spring Cloud微服务架构中。

首先，在多模块项目中使用maven管理依赖，并在父模块的pom文件中统一管理所有依赖的版本。以Spring Cloud依赖为例（图 5.1），在properties中自定义属性spring-cloud.version以指定项目采用的Spring Cloud版本，再在依赖的version中引用。

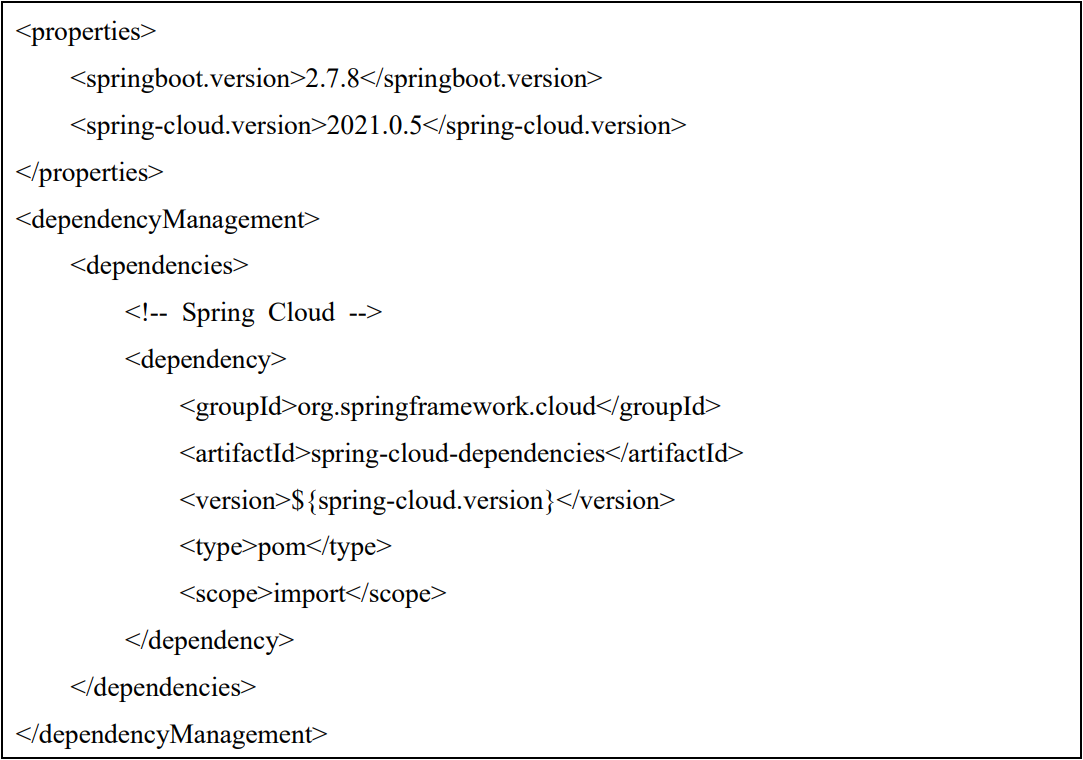


图 5.1 统一管理依赖版本

其次，在项目对应子模块的pom文件中引入spring-cloud-starter-consul-discovery依赖（图 5.2）。此处依赖的version可省，其兼容版本由spring-cloud-dependencies包指定。

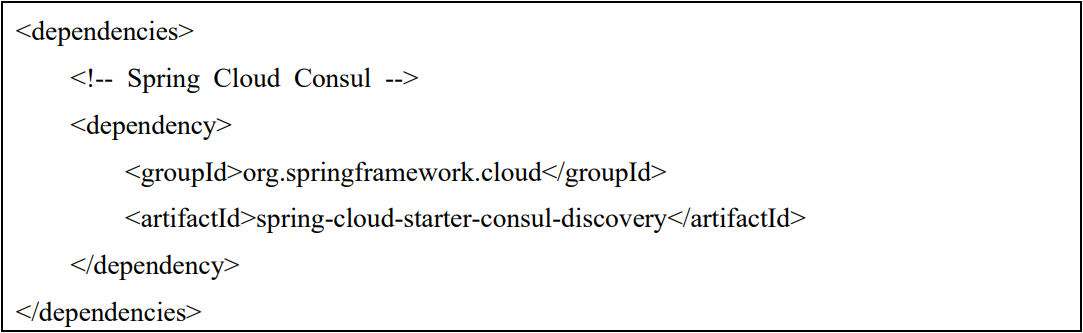


图 5.2 引入Consul依赖

最后配置Consul的有关参数（图 5.3）。为每一个实例指定唯一的instance-id，防止Consul日志中出现多个同一实例check name默认替代check id带来的冲突。开启Consul的健康检查，通过Springboot Actuator监控结点的健康信息。一般地，可通过xml或yaml进行配置。因yaml基于缩进的格式规范和更为良好的可读性采用后者，这也包括项目中的其他配置文件。

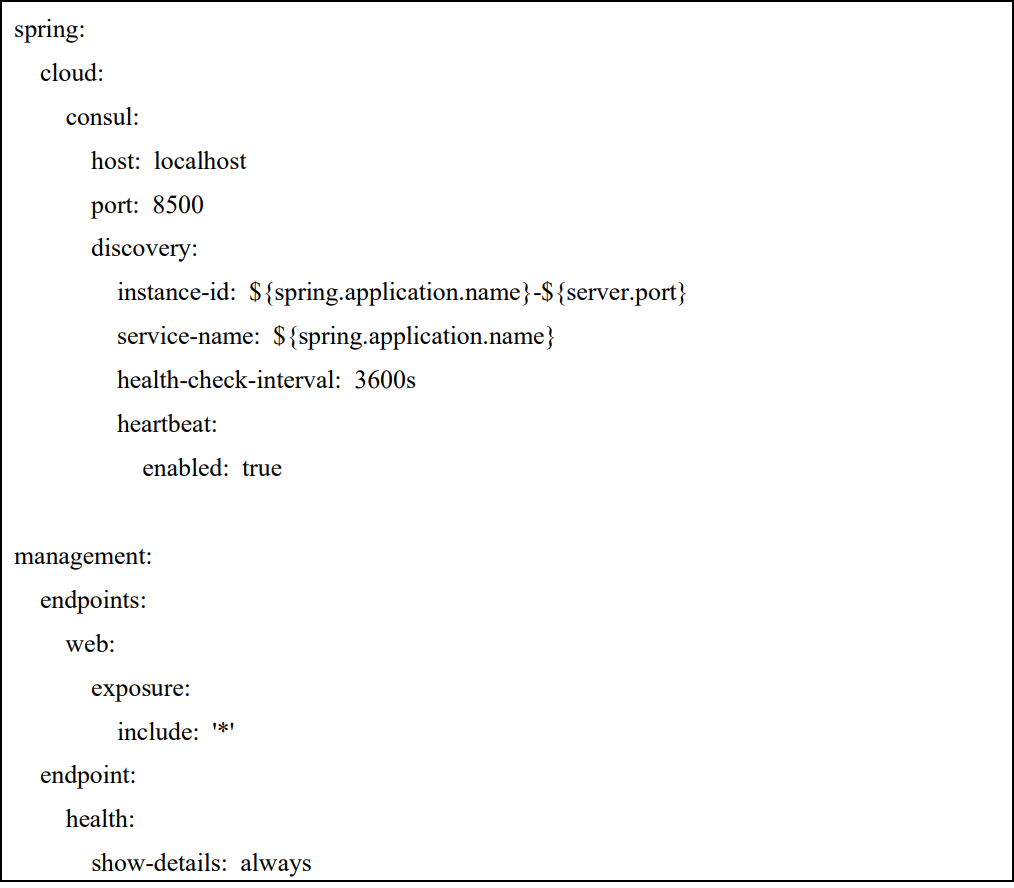


图 5.3 Consul参数配置

### 服务网关

4.1节叙述了系统的架构设计，其中边界层为Spring Cloud Gateway微服务网关，主要负责完成路由转发、白名单放行、用户鉴权和接口限流功能。在项目中的具体实现为bss-gateway与bss-gateway-admin两个Spring Boot模块，其分别处理来自客户端和后台系统的请求。以下以客户端请求秒杀服务为例：

1. **路由转发**

和注册中心结合时，Gateway的路由配置方式如图 5.4所示。先指定路由的id和路由所匹配的转发地址，再通过路由断言配置匹配所有以/api/seckill/为前缀的路径，最后由过滤器去除第一层的/api前缀。

负载均衡是将网络流量分发到多个服务器的策略，目的在于提高系统整体的响应速度，服务于高性能、高可用的目标。在请求的流量控制上，Spring Cloud Gateway内置了自动的负载均衡策略。其中，Gateway支持通过lb://形式开启负载均衡，在此情景下应配置动态路由。LoadBalancerClient是Spring Cloud提供的一种负载均衡客户端，通过LoadBalancerClient解析服务名为真实的主机地址与对应的端口号，其默认策略为轮询（Round-Robin）策略。

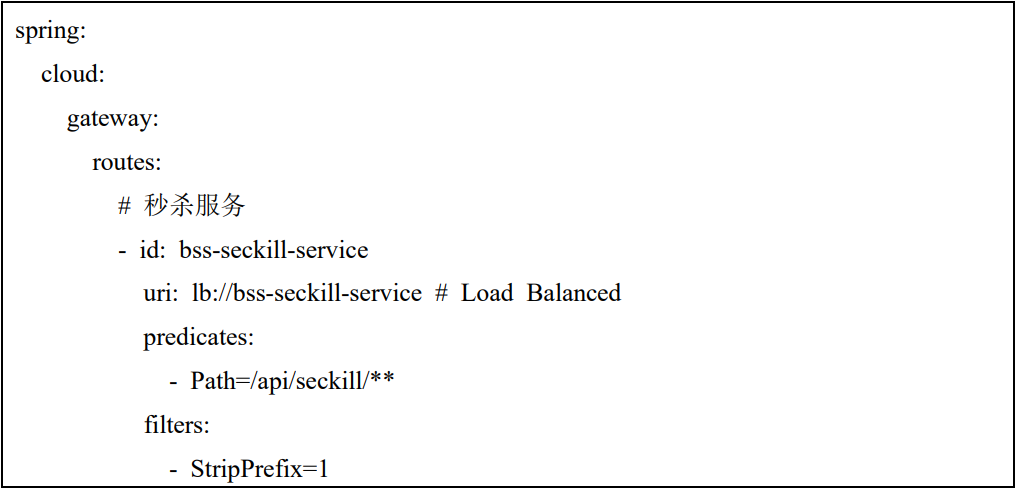


图 5.4 Spring Cloud Gateway路由配置

1. **白名单放行**

在实际业务中，诸如用户注册、登录等请求是无需鉴权的，可通过自定义白名单的形式加以放行（图 5.5）。先创建一个WhiteListConfig类，用于存放需白名单放行的接口地址列表。

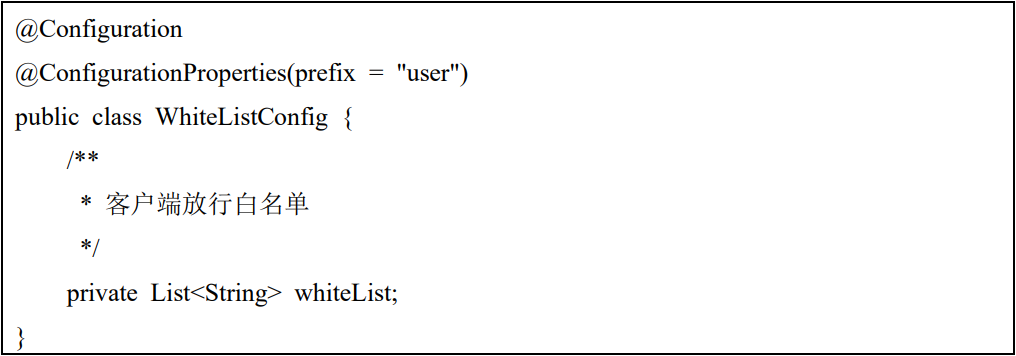


图 5.5 定义白名单

自定义白名单，为了方便起见，在yaml文件中进行配置，并通过@ConfigurationProperties注解指定需读取的自定义前缀，如图 5.6所示指定匹配“/login”、“/register”的登录、注册为白名单放行路径。



图 5.6 指定放行的接口路径

最后如图 5.7所示，在Filter中放行对应请求。

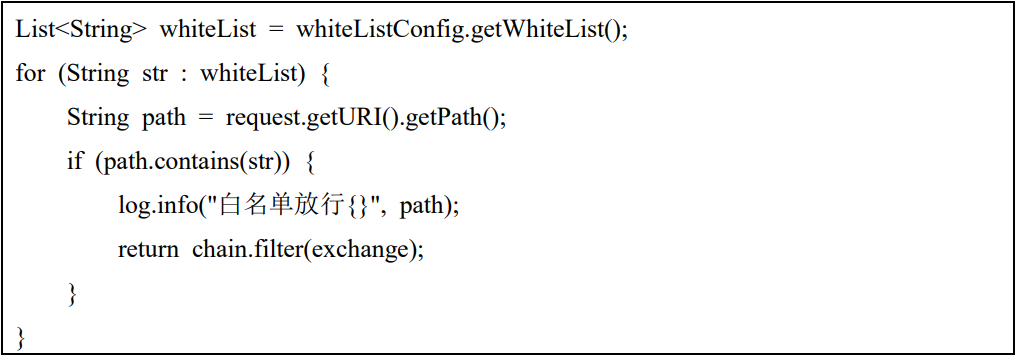


图 5.7白名单请求放行

1. **用户鉴权**

用户鉴权（图 5.8）的具体实现分为两步。一是从请求中获取token，验证令牌是否有效，非法用户将返回未授权HTTP状态码；二是重写请求头，在请求头中加入用户信息，便于后续业务处理时直接取用，避免二次调用。



图 5.8 用户鉴权的实现

1. **接口限流**

系统实现了用户IP地址限流和秒杀限流两种限流方式。首先添加spring-boot-starter-data-redis-reactive依赖（图 5.9），以在Gateway中使用速率限制器。

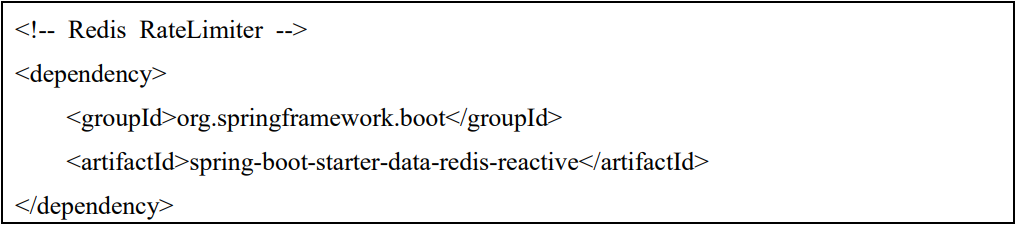


图 5.9 引入Redis RateLimiter依赖

配置用户IP地址限流（图 5.10），指定限流键解析器Bean的对象名为ipKeyResolver，将用户IP作为限流键。

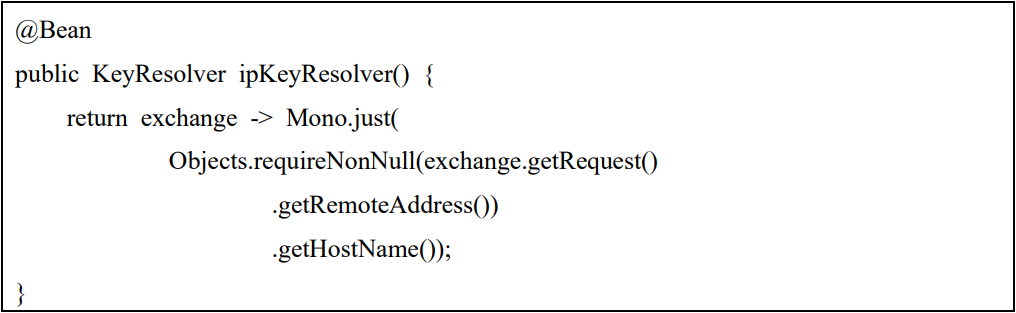


图 5.10用户IP地址限流配置

配置秒杀限流（图 5.11），指定限流键解析器Bean的对象名为seckillKeyResolver，将请求入参中的productId作为限流键。

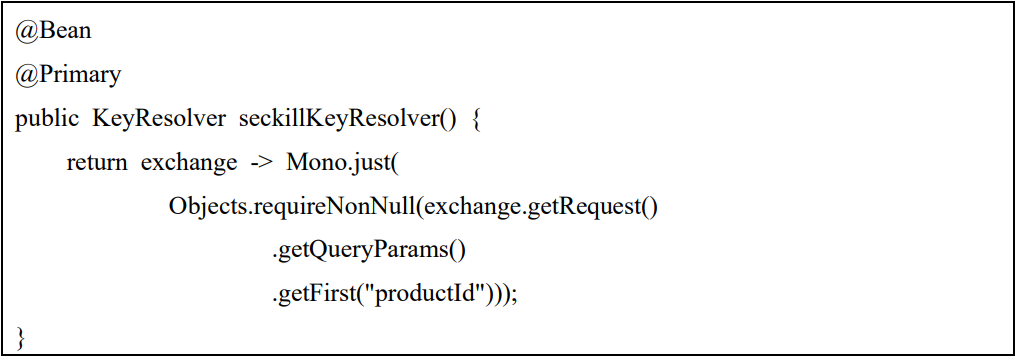


图 5.11 秒杀限流配置

通过Gateway的过滤器在yaml中配置限流（图 5.12）。通过replenishRate和burstCapacity参数可灵活地指定令牌桶每秒填充平均速率与令牌桶容量，如当前设定下每秒钟生成10个令牌，且令牌桶最大容纳200个令牌。



图 5.12 Spring Cloud Gateway过滤器配置

### 服务调用

在2.2.1节中，叙述过服务消费者调用提供者的服务经历注册服务、订阅服务、通知变更和服务调用四个步骤。本节则主要阐述服务调用的具体实现。在本系统中，针对不同场景的需求，采用了Dubbo和Spring Cloud Openfeign完成对应的服务调用过程。



图 5.13 本系统Dubbo服务调用链

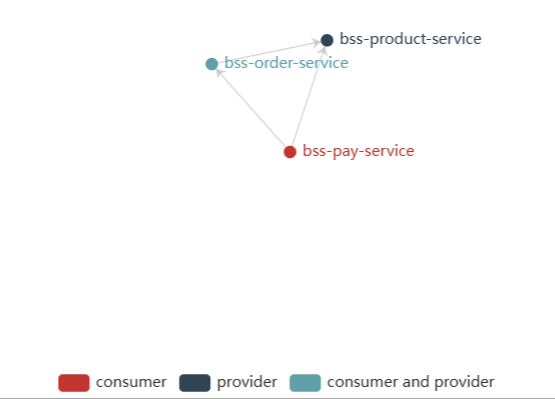


图 5.14 系统模块间的调用关系

本系统中Dubbo的服务调用链（图 5.13和图 5.14）为支付服务→订单服务→商品服务单向调用，避免了模块间出现循环依赖。其中，支付服务仅作为服务消费者；订单服务既是服务消费者又是服务提供者，为支付服务提供服务，并消费商品服务所提供的服务；商品服务是为服务提供者。

具体实现上，应先在公用模块中定义用于RPC通信的接口方法（图 5.15）。Dubbo2中各服务均是方法级的，以扣减库存方法为例，该方法经商品服务提供，由订单服务完成消费：

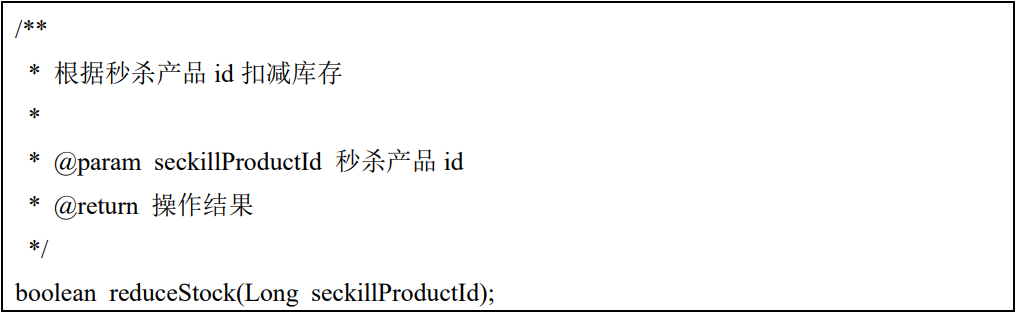


图 5.15 定义RPC接口方法

其次，应在使用到Dubbo的各模块的yaml文件中配置（图 5.16）。其中， base-packages参数仅由服务提供者指定，用于指定需要扫描的包。

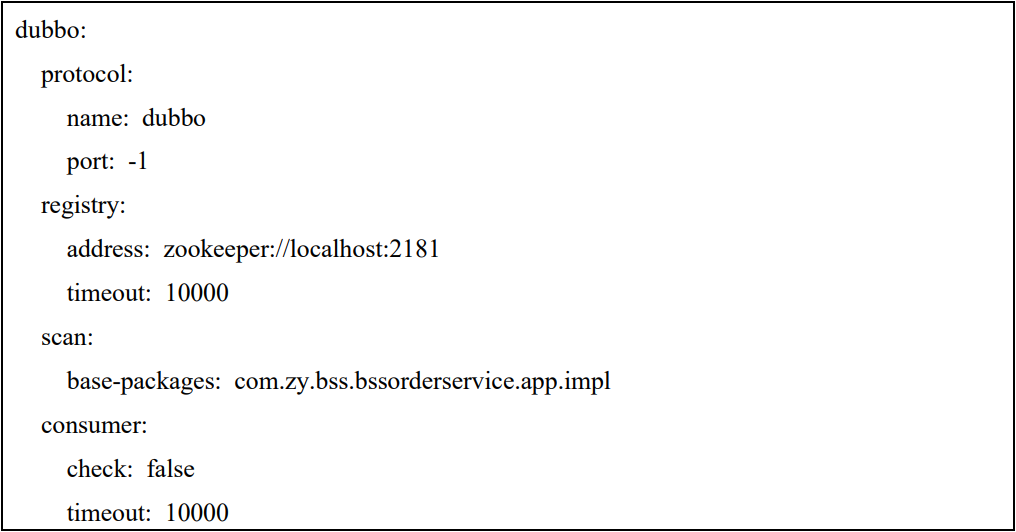


图 5.16 Dubbo参数配置

最后，在提供服务的方法类上加上@DubboService注解，并在服务消费者中需要使用的地方通过@DubboReference注解引用（图 5.17）。



图 5.17 Dubbo调用服务提供者的服务

Spring Cloud Openfeign基于Http协议通信。项目中，可在对性能无显著要求的功能中直接复用已暴露RESTful 接口，提高编码效率。

以获取商品列表方法为例（图 5.18），该方法已由商品服务向外暴露供客户端取用，恰好秒杀服务也需使用。创建一个SeckillProductFeignService 类，通过@FeignClient注解指明远程服务名。在结合注册中心使用时，将会使用该服务名向注册中心寻址并发起请求。最后在服务消费者中引用，完成调用。



图 5.18 Feign调用服务提供者的服务

### 服务容错

由于超时、重试等机制多数已封装或可直接配置，而5.2.2节中又阐述了网关中接口限流的实现，故此处主要阐述熔断器这一服务容错手段。

2.2.4节叙述了Spring Cloud Circuit Breaker组件。系统中实际采用的熔断器实现为Resilience4J，当远程服务不可达时，执行Fallback方法作为替代。此处以初始化账户方法为例（图 5.19），想要在Openfeign中使用熔断器，需先引入对应依赖：

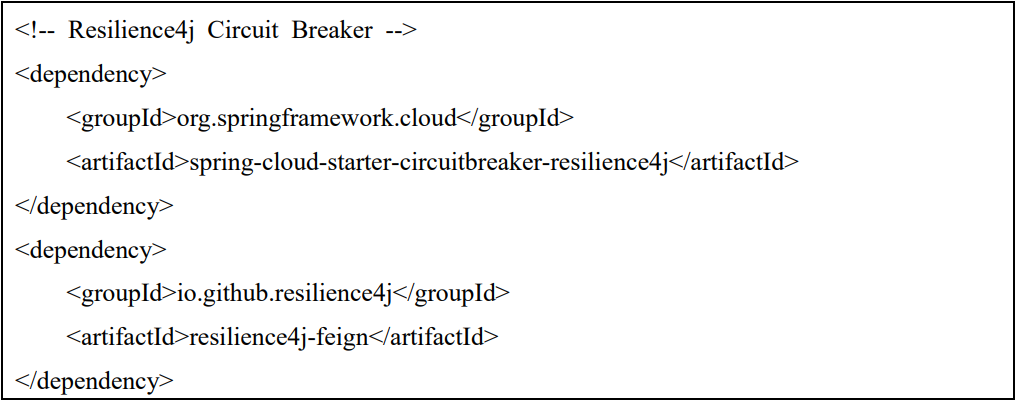


图 5.19 引入Spring Cloud Circuit Breaker依赖

配置熔断器（图 5.20），需先指定Openfeign开启熔断器功能，并配置Resilience4J熔断器。registerHealthIndicator 参数用于指定向Actuator注册结点的健康信息。在2.2.4节中，叙述了Resilience4J熔断器的工作原理。其中，对应的失败率阈值默认为50%，对应打开状态的休眠计时由waitDurationInOpenState参数配置，指定了熔断后尝试访问被熔断服务的时间间隔。

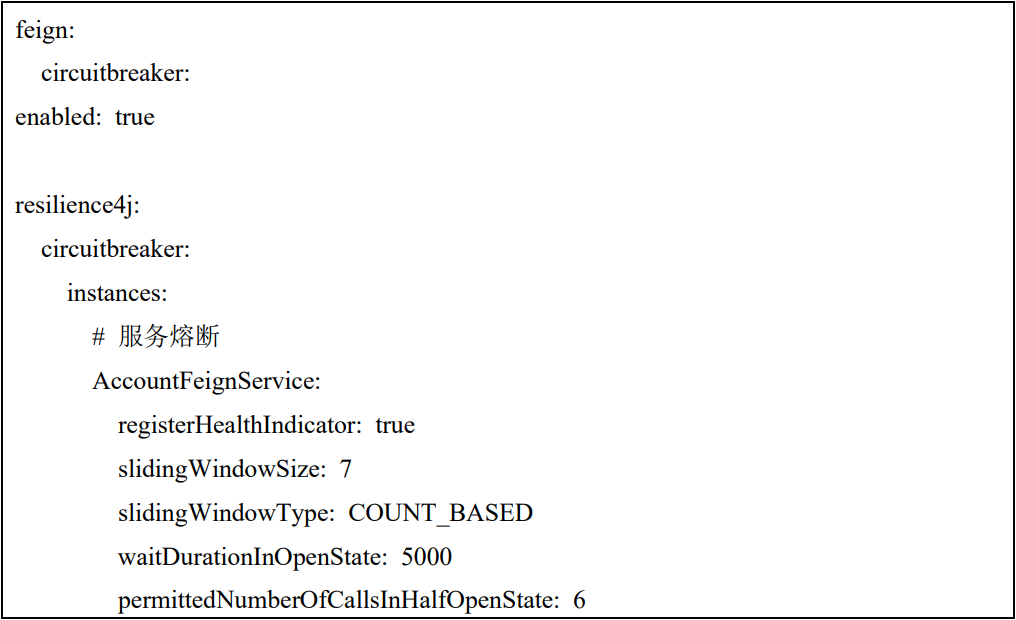


图 5.20 Resilience4J参数配置

创建fallback方法（图 5.21），并在原FeignService类中通过@FeignClient指明fallback方法类。

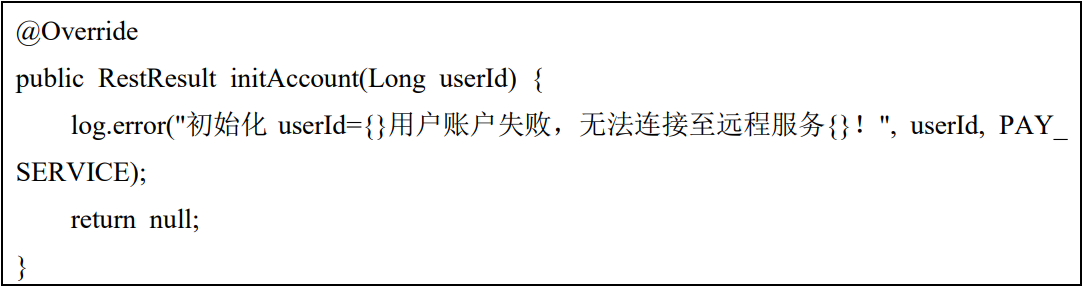


图 5.21 fallback方法的实现

## 用户服务模块的实现

用户服务主要包括用户注册、用户登录、个人中心和用户管理功能的实现。模块水平划分为Controller层、Service层和Mapper层，由UserController和AdminController类分别处理来自客户端和后台系统的请求。其实现的类图如图 5.22所示：

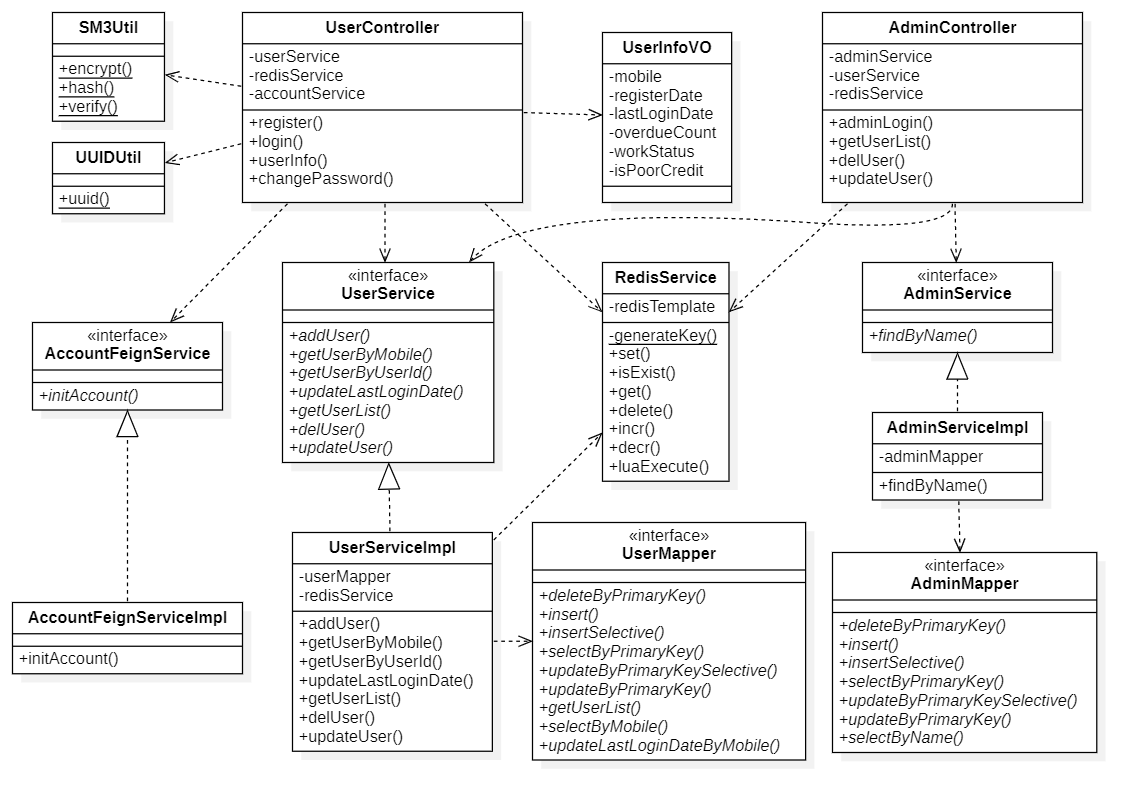


图 5.22 用户服务模块类图

### 用户注册

用户注册的UML时序图如图 5.23所示。用户在客户端填写有关信息后，信息将封装为JSON格式并以HTTP请求形式到达客户端网关。请求过滤后被发送至UserController，该类提供register()方法进行调用。此时若手机号已被占用则注册失败，否则接着进入UserService中的addUser()方法，执行UserMapper中的insertSelective()方法，在数据库中存入该用户信息。最后通过Feign远程调用支付服务模块的initAccount()方法，为该用户初始化个人账户，返回注册成功结果。

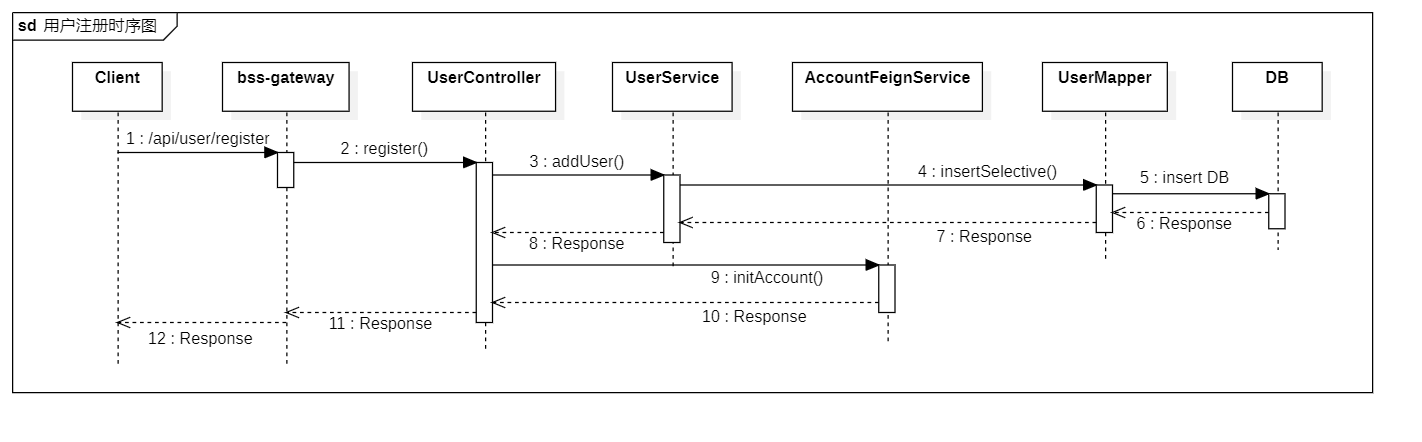


图 5.23 用户注册时序图

### 用户登录

用户登录的UML时序图如图 5.24所示。用户在客户端页面输入手机号和密码后，提交请求并经网关交由UserController的login()方法处理，调用UserService的getUserByMobile()方法查询数据库，仅用户名和密码均正确的用户方可登录，否则该次操作失败。接着依次调用RedisService的delete()和set()方法操作缓存更新用户令牌。最后进入UserService的updateLastLoginDate()方法，执行UserMapper中的updateLastLoginDateByPrimaryKey()方法，同步用户本次的登录状态，返回登录成功信息。

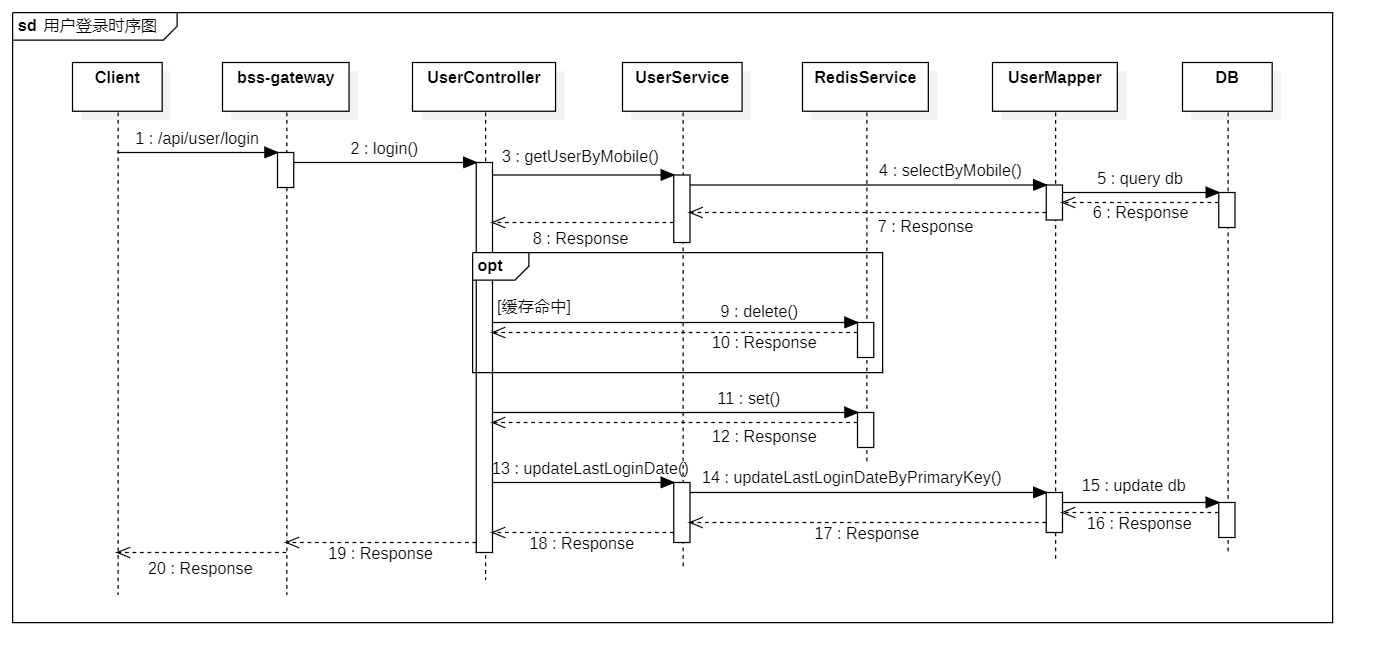


图 5.24 用户登录时序图

### 个人中心

个人中心功能包括个人信息查看与密码修改两部分。

个人信息查看的UML时序图如图 5.25所示。登录后的用户可在客户端查看本人信息，此时请求在通过网关鉴权后进入UserController的userInfo()方法，调用UserService的getUserByUserId()方法。优先从缓存中查询，调用RedisService的get()方法请求缓存，命中将直接返回结果；否则，说明此时缓存中无对应数据，将从数据库中查询，执行UserMapper中的selectByPrimaryKey()方法，返回查询结果并通过set()方法对用户信息进行缓存。

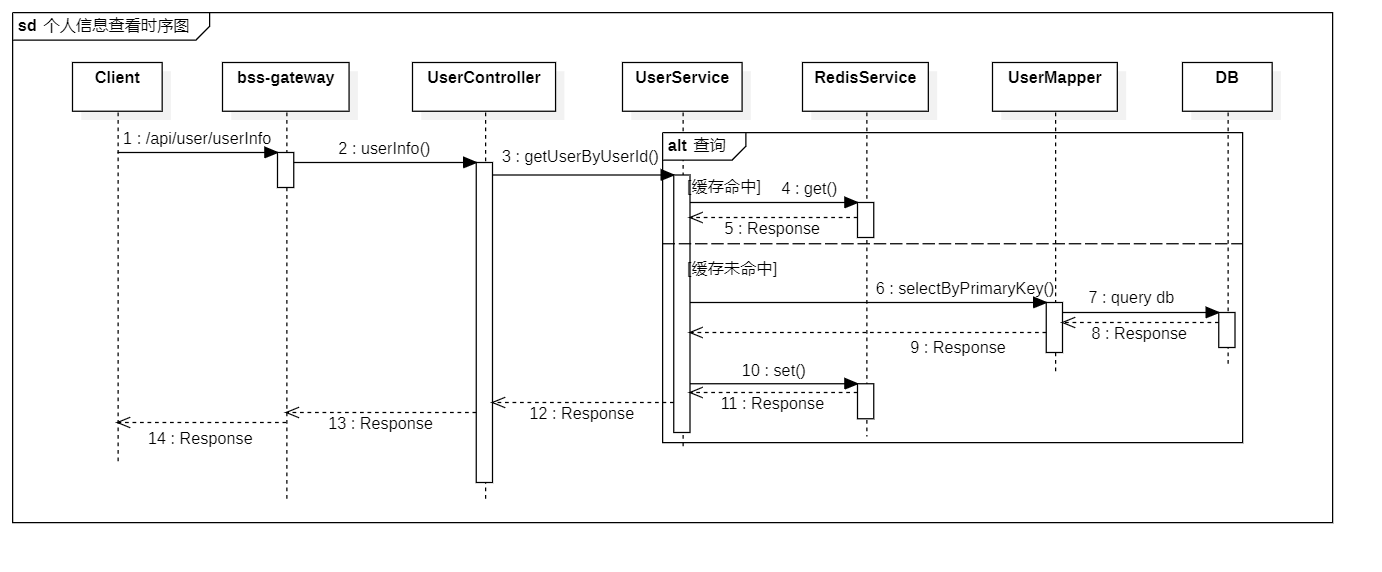


图 5.25 个人信息查看时序图

密码修改的UML时序图如图 5.26所示。登录后的用户可在客户端修改本人密码，操作成功后用户需重新登录。在用户输入原密码和新密码后请求提交，通过网关鉴权后进入UserController的changePwd()方法，调用UserService的getUserByUserId()方法，新密码校验合法后方可继续操作，否则密码修改失败。接着进入UserService的updateUser()方法，调用RedisService的delete()和set()方法更新缓存中的用户信息，最后执行UserMapper中的updateByPrimaryKey()方法更新数据库，返回结果。

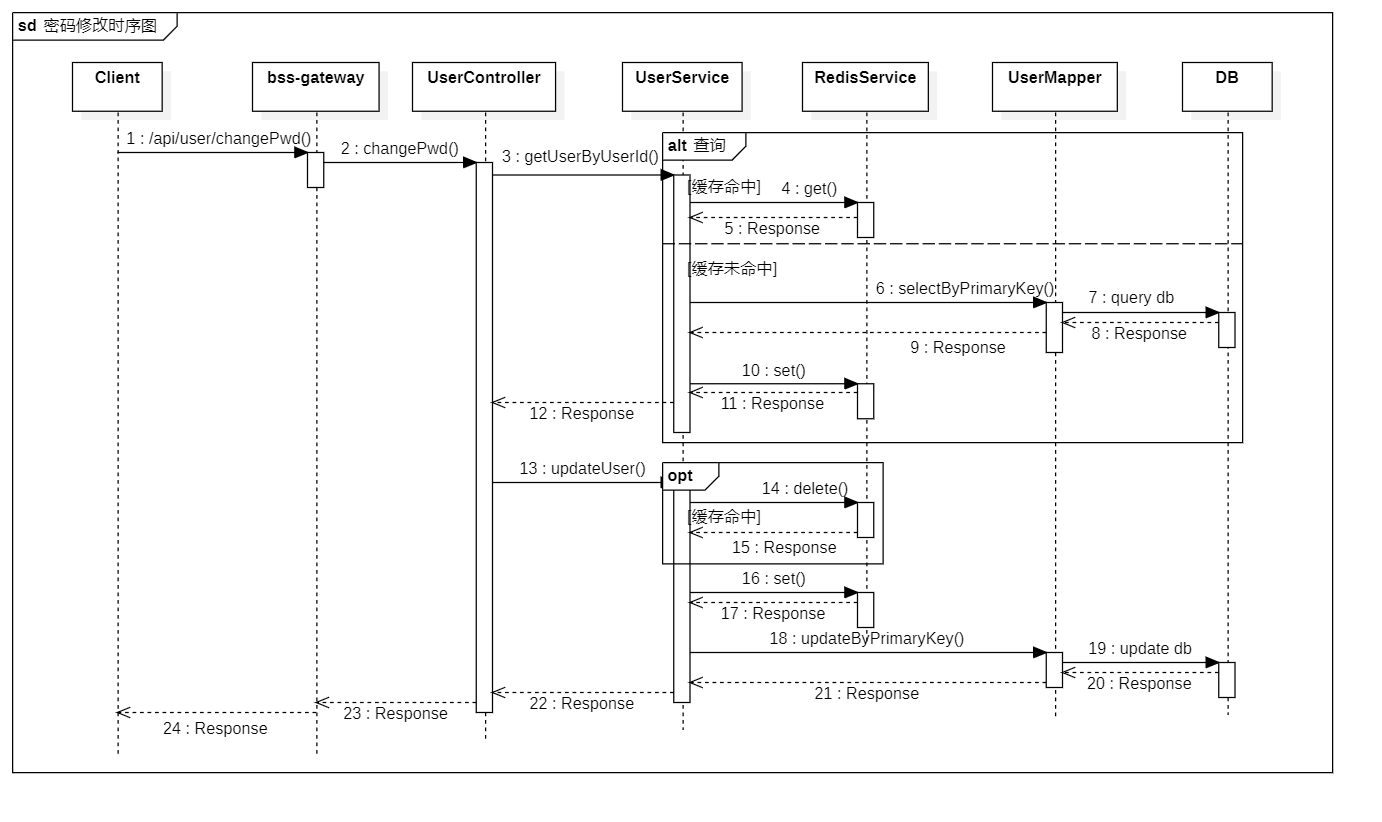


图 5.26 密码修改时序图

### 用户管理

用户管理的UML时序图如图 5.27所示。管理员登录后台系统后，可对用户信息进行管理。UserAdminController为后台系统提供API接口服务，可调用UserService的getUserList()、delUser()和updateUser()方法分别用于查询用户分页列表、删除用户和更新用户信息，并通过UserMapper与数据库交互。缓存中的用户信息设置为永不过期，因此在数据库中的用户信息发生变化后也需同步更新缓存。

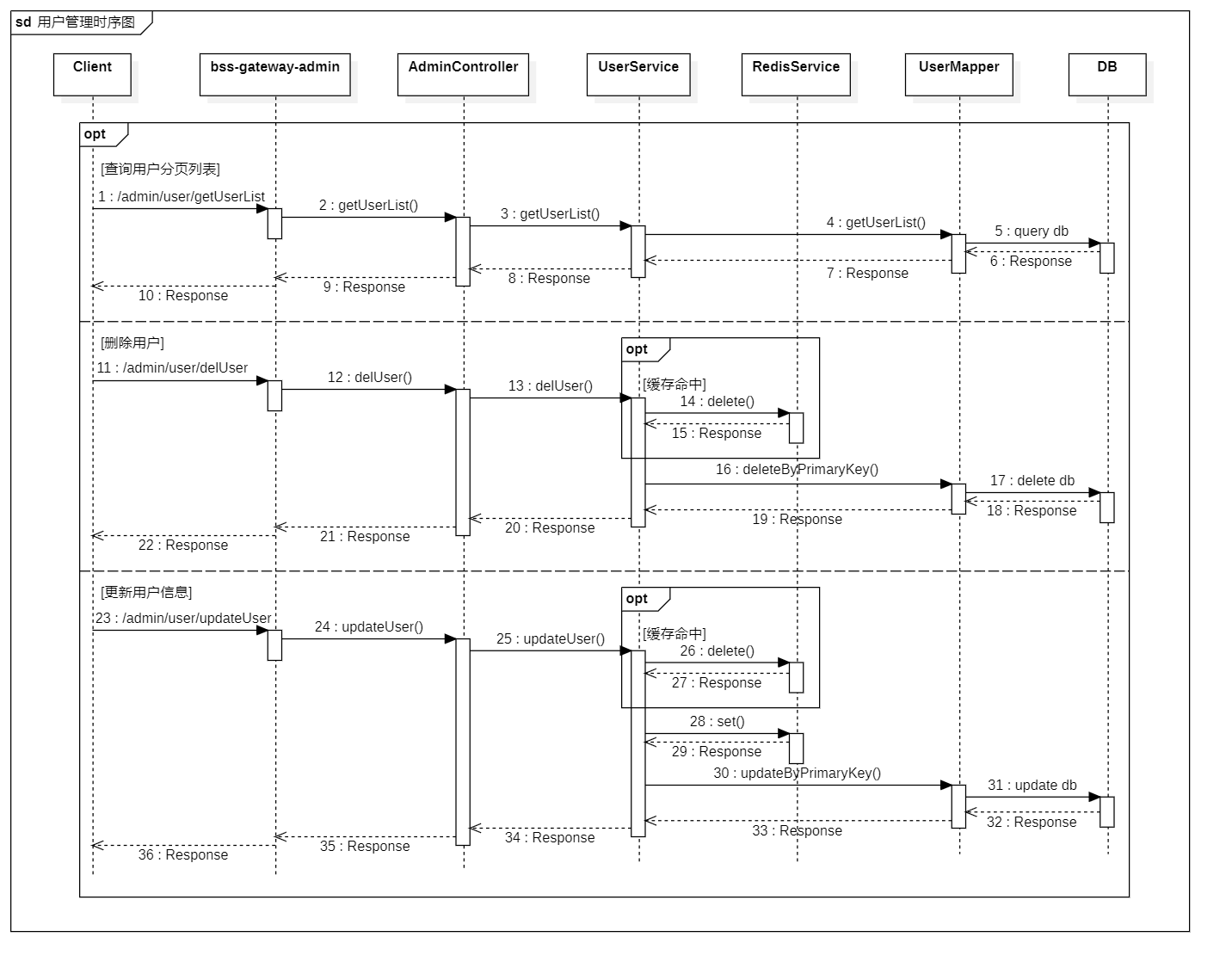


图 5.27 用户管理时序图

## 商品服务模块的实现

商品服务主要包括商品信息、产品管理和活动管理功能的实现，由ProductController和ProductAdminController类分别处理来自客户端和后台系统的请求。其实现的类图如图 5.28所示：

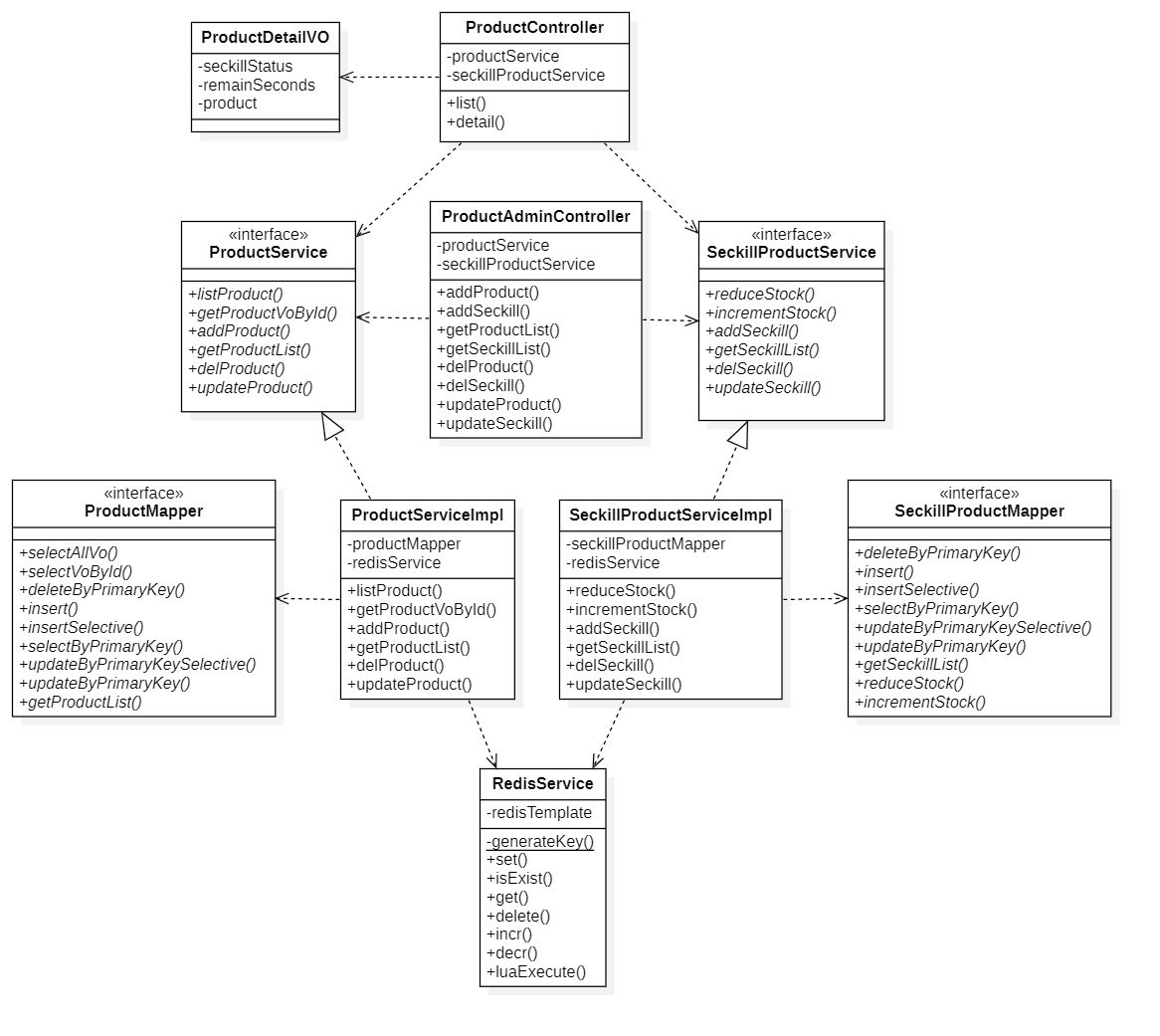


图 5.28 商品服务模块类图

### 商品信息

商品信息功能包括商品列表查看与商品详情查看两部分。

商品列表查看的UML时序图如图 5.29所示。用户可在客户端查看所有已设置秒杀活动的商品，其中包括尚未开启秒杀的商品，以方便用户提前了解产品。请求在通过网关鉴权后进入ProductController的list()方法，调用ProductService的listProduct()方法。优先从缓存中查询，调用RedisService的get()方法请求缓存，命中将直接返回结果；否则，说明此时缓存中无对应数据，将从数据库中查询，执行UserMapper中的selectAllVo()方法，该方法为对seckill.product和seckill.seckill\_product的联表查询，返回一个商品列表，包含的每条数据为产品信息与对应活动信息的组合。最后通过RedisService的set()方法对商品信息进行缓存。

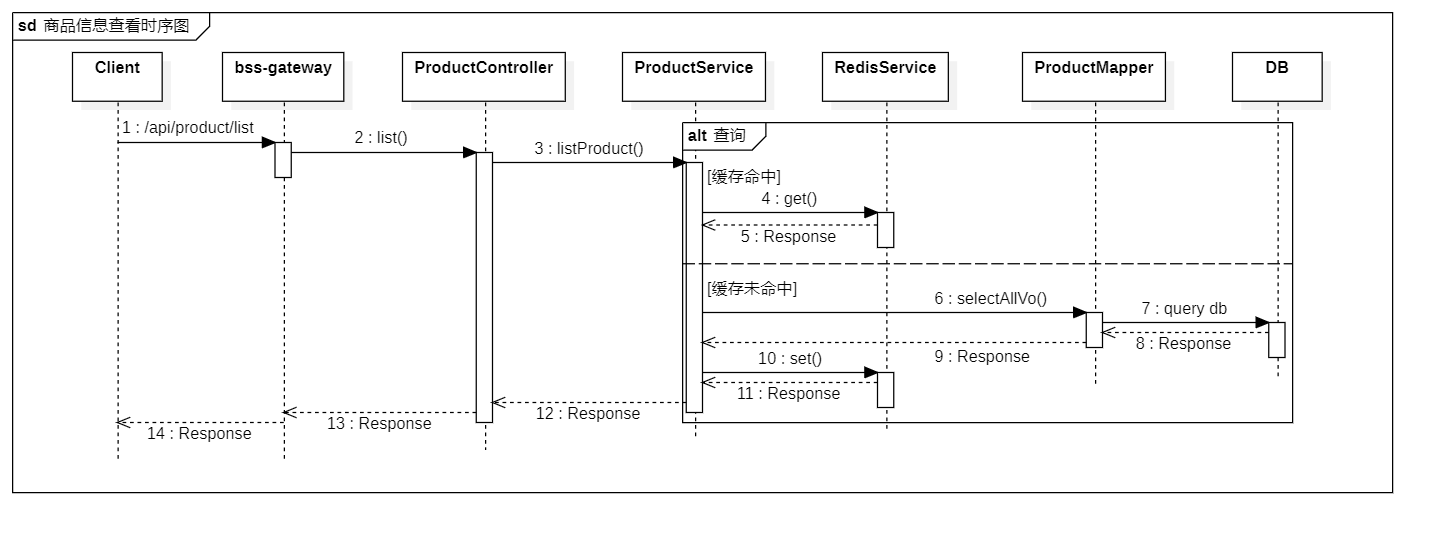


图 5.29 商品信息查看时序图

商品详情查看的UML时序图如图 5.30所示。用户可在客户端查看单个秒杀活动的商品详情，此时请求在通过网关鉴权后进入ProductController的detail()方法，调用ProductService的getProductVoById()方法。优先从缓存中查询，调用RedisService的get()方法请求缓存，命中将直接返回结果；否则，说明此时缓存中无对应数据，将从数据库中查询，执行ProductMapper中的selectVoById()方法，返回查询结果并通过set()方法对用户信息进行缓存。此后，服务端在根据当前时间戳确定该活动的秒杀状态与倒计时后，将ProductDetailVO视图对象作为结果返回。

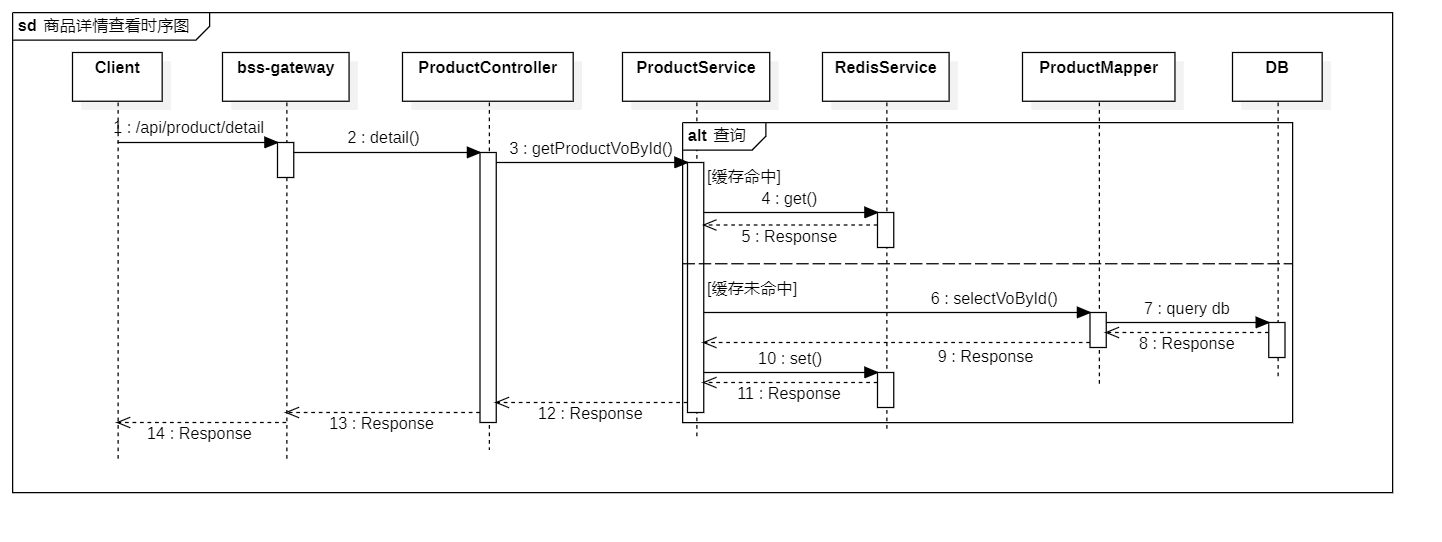


图 5.30 商品详情查看时序图

### 产品管理

产品管理的UML时序图如图 5.31所示。管理员登录后台系统后，可对产品信息进行管理。ProductAdminController为后台系统提供API接口服务，可调用ProductService的addProduct()、getProductList()、delProduct()和updateProduct()方法分别用于添加产品、查询产品分页列表、删除产品和更新产品信息，并通过ProductMapper与数据库交互。缓存中的产品信息设置了过期时间，因此在数据库中的产品信息发生变化后无需同步更新缓存。

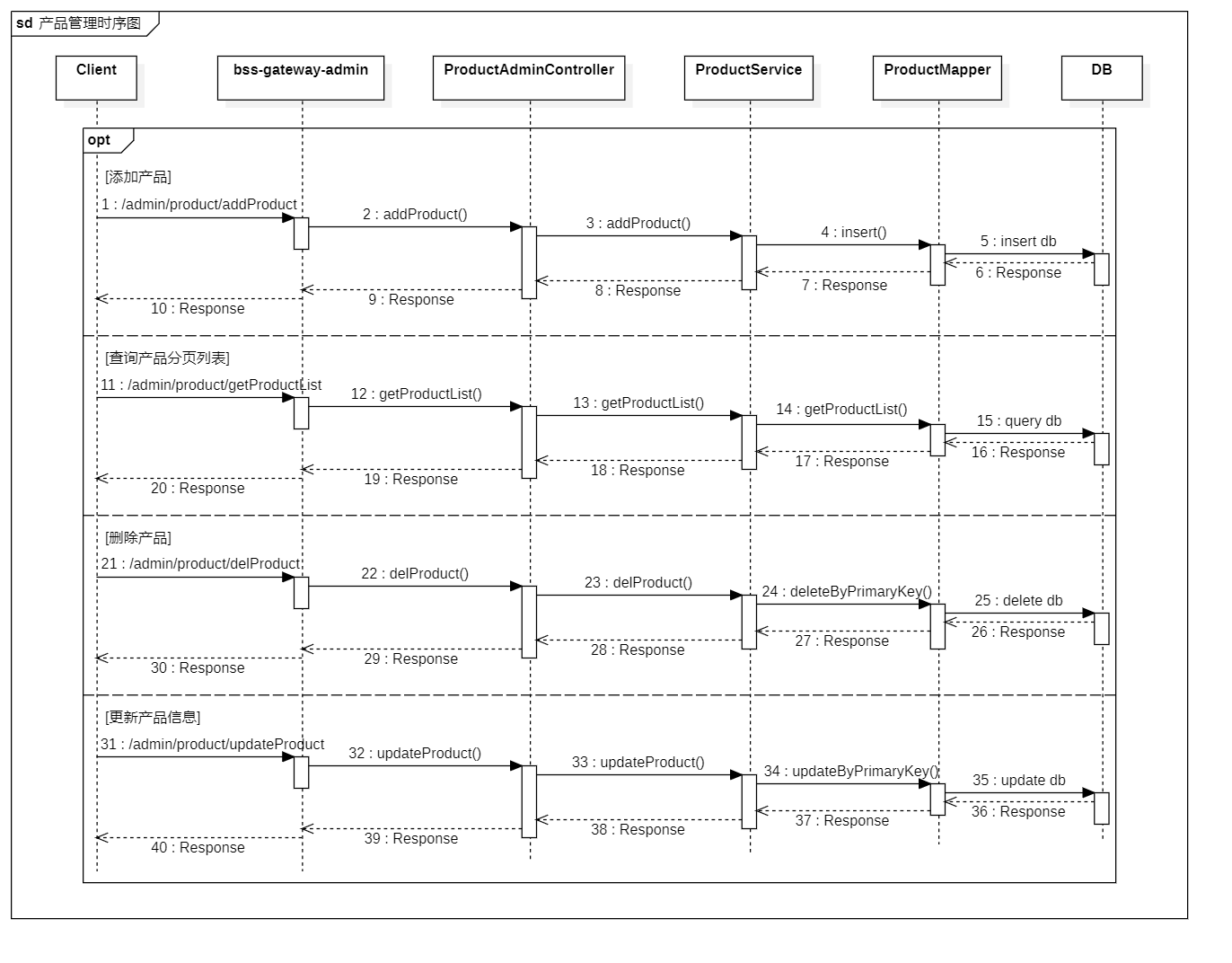


图 5.31 产品管理时序图

### 活动管理

活动管理与产品管理功能实现大致相同，均为CRUD操作，此处不再赘述。管理员可在后台系统管理秒杀活动，SeckillProductService的addSeckill()、getSeckillList()、delSeckill()和updateSeckill()方法分别用于秒杀活动发布、查询活动分页列表、取消活动推送和更新，通过SeckillProductMapper与数据库交互。由于缓存中的活动信息亦已设置过期时间，数据库中信息变化后无需更新缓存。

## 筛查服务模块的实现

筛查服务主要包括申请筛查和决策配置功能的实现，由ScreenController和ScreenAdminController类分别处理来自客户端和后台系统的请求。其实现的类图如图 5.32所示：

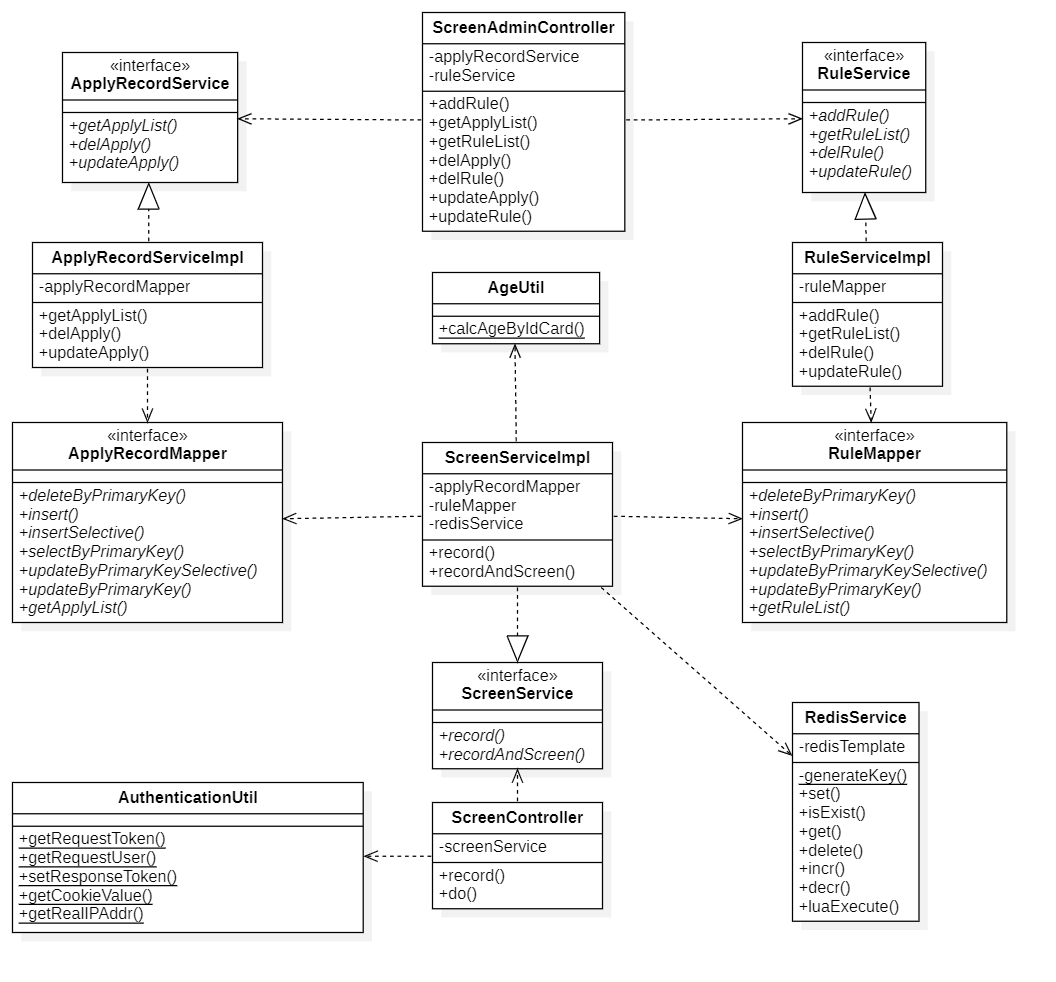


图 5.32 筛查服务模块类图

### 申请筛查

申请筛查功能的UML时序图如图 5.33和图 5.34所示。用户在客户端申请参与秒杀活动，视活动的筛查开启情况，提交不同请求并经网关交由ScreenController的record()或screen()方法处理。若无筛查机制，则仅在数据库插入本次用户的申请记录并直接放行。

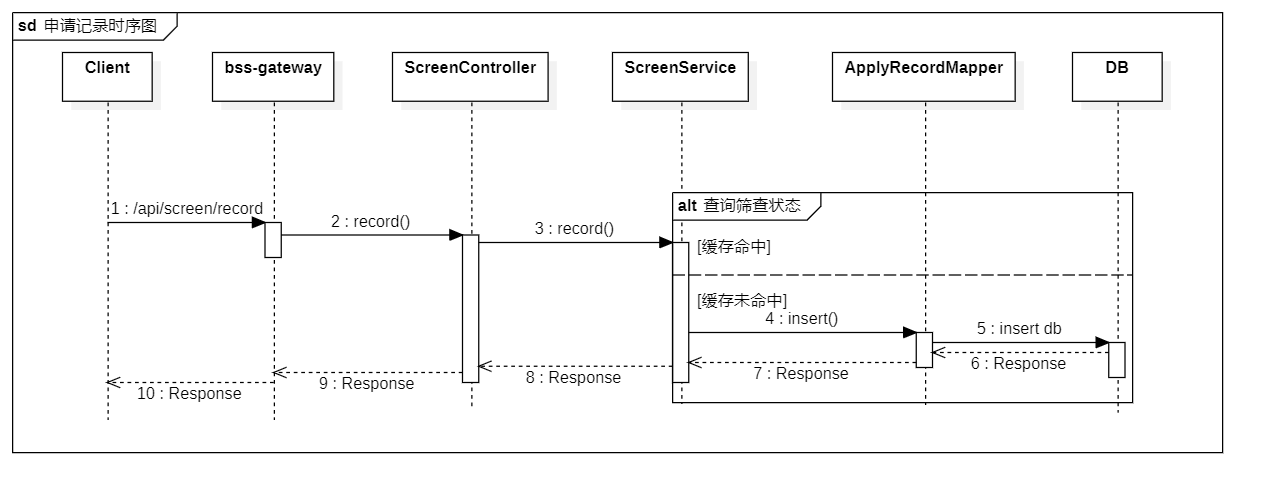


图 5.33 无筛查机制的申请筛查时序图

开启筛查则需执行筛查控制流程。首先进入ScreenService的recordAndScreen()方法，缓存中已有筛查状态信息可直接返回结果。获取该活动的有关规则信息，优先从缓存中查询，调用RedisService的get()方法请求缓存；如若未获取到将从数据库中查询，并调用set()方法缓存此规则信息。其次筛查该用户的本次申请，仅有关条件均符合的用户方可继续参与活动，否则该次操作失败。接着操作缓存更新筛查状态信息，最后执行applyRecordMapper中的insert()方法，向数据库插入本次用户的申请记录，并返回筛查状态信息。

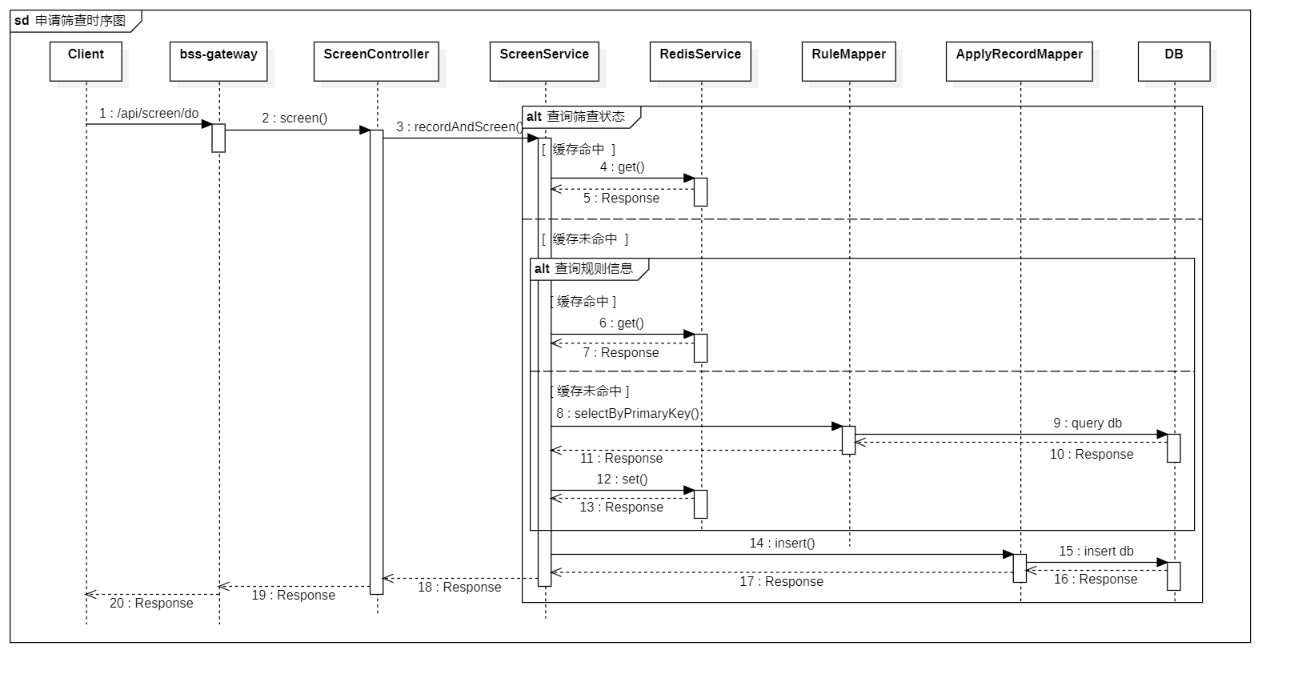


图 5.34 启用筛查机制的申请筛查时序图

### 决策配置

管理员可在后台系统配置准入规则，RuleService的addRule()、getRuleList()、delRule()和updateRule()方法分别用于添加规则、查询规则分页列表、删除规则和更新规则信息，通过SeckillProductMapper与数据库交互。由于缓存中的规则信息已设置过期时间，数据库中信息变化后无需更新缓存。

## 秒杀服务模块的实现

秒杀服务是本系统的主要服务，秒杀流程控制功能的实现类图如图 5.35所示：



图 5.35 秒杀服务模块类图

SeckillController是本服务的核心控制器，专门处理来自客户端的请求。在秒杀活动开始后，用户即可在客户端秒杀商品。SeckillController提供了秒杀流程控制功能，包含了5个API接口：afterPropertiesSet()方法用于在秒杀活动开始前预热库存，getSeckillVerifyCode()方法用于提供秒杀的图形验证码，getSeckillPath()方法用于为用户生成一份专属的秒杀链接，seckill()方法为实际的秒杀控制方法，getSeckillResult()方法则用于向客户端返回秒杀结果。以下各小节将对秒杀流程控制的子功能作详细阐述。

### 商品库存预热

早在秒杀活动开始前，就需预先缓存有关商品的库存信息，以应对来自客户端可能的高额请求。其中，SeckillController实现了InitializingBean接口，该接口由Spring提供，仅包括afterPropertiesSet()方法，为bean提供了初始化方法的方式。因此在Spring启动后初始化bean时，将自动调用并执行该方法。其实现的关键代码如图 5.36所示：

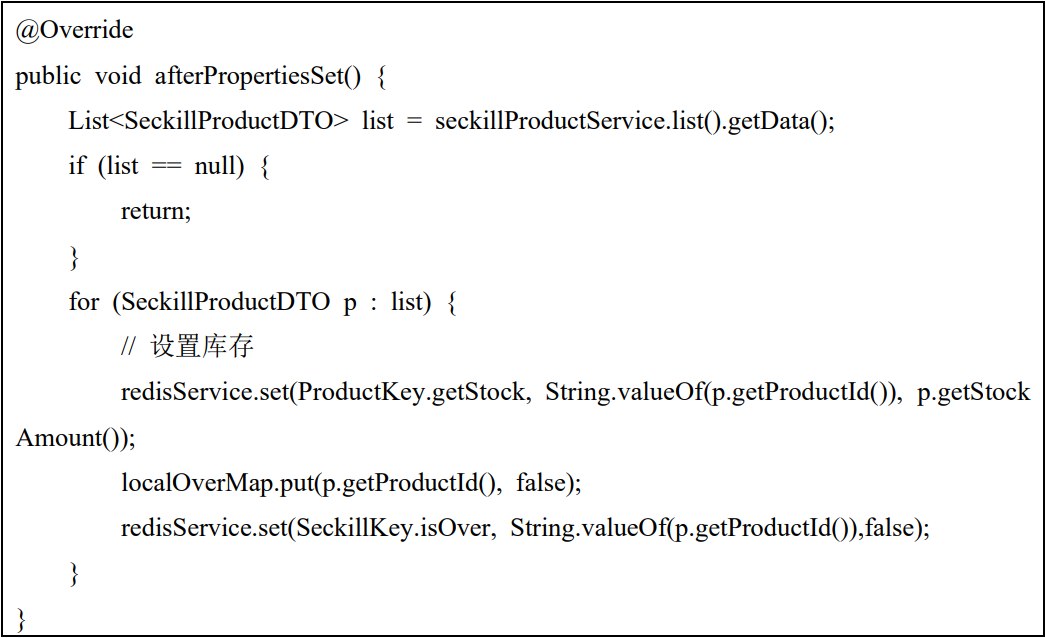


图 5.36 商品库存预热的实现

商品库存预热的UML时序图如图 5.37所示。在Spring IOC容器启动后，经历bean的实例化阶段后开始初始化bean，此时进入SeckillController的afterPropertySet()方法，再通过Feign远程调用商品服务模块的list()方法获取商品列表，最后通过RedisService的set()方法依次遍历列表设置库存信息缓存。

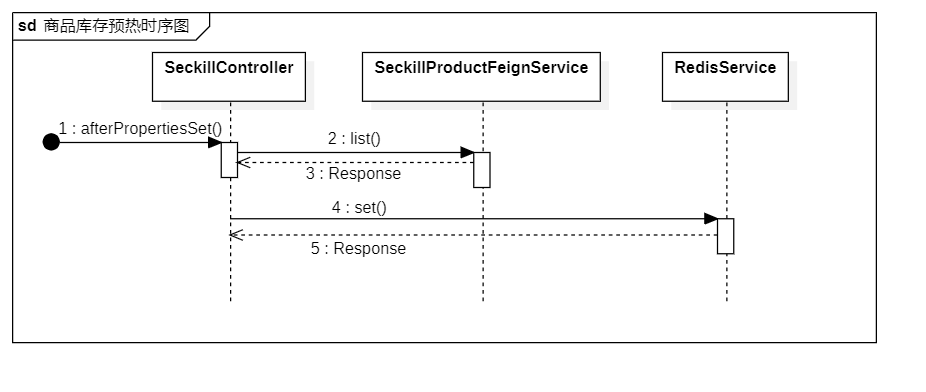


图 5.37 商品库存预热时序图

### 秒杀图形验证

在实际秒杀商品前还需通过一道图形码验证程序。该逻辑由VerifyCodeService完成，提供createSeckillVerifyCode()和checkSeckillVerifyCode()方法分别用于图形验证码的生成和验证。

图形验证码的生成分为以下五个步骤：图片初始化、随机验证码生成、随机验证码绘入图像、随机验证码结果计算和计算结果缓存，其分别对应VerifyCodeUtil的initImage()、generateCode()、generateVerifyCode()和calcVerifyCodeResult()方法。在Java中原生提供了BufferedImage和Graphics类，前者是Image的子类，可在内存中生成一个图像缓冲区；而后者则提供了基本的绘图方法，可对此图像缓冲区执行一系列操作。其实现的关键代码如图 5.38所示：



图 5.38 图形验证码生成的实现

其中，4.5.1节叙述的随机验证码字符串生成借助java.util.Random实现，创建一个随机实例以依次生成数字与操作符（图 5.39）。

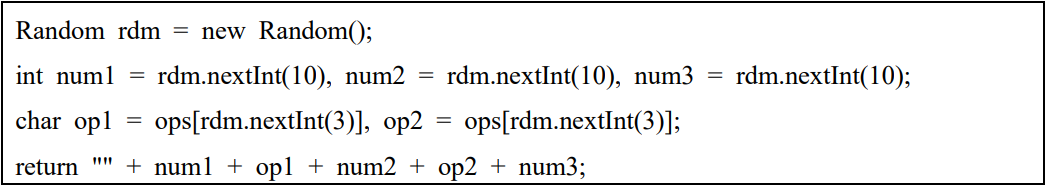


图 5.39 随机验证码字符串生成的实现

获取秒杀验证码的UML时序图如图 5.40所示。用户在客户端点击秒杀按钮后，先发起获取图形验证码请求，该请求经网关鉴权后进入SeckillController的getSeckillVerifyCode()方法。若userId或productId任意一个为空，都将返回SECKILL\_CODE\_GENERATION\_FAIL；调用VerifyCodeService的createSeckillVerifyCode()方法生成图形验证码并通过ImageIO.write()输出JPEG格式图片结果。

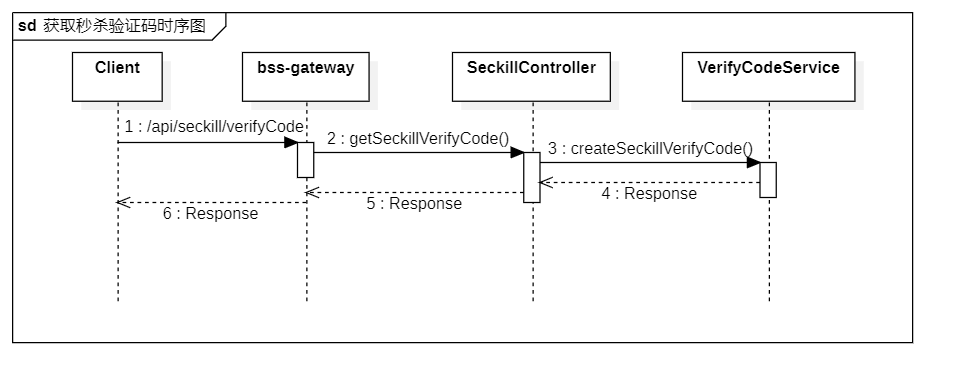


图 5.40 获取秒杀验证码时序图

图形验证码的验证（图 5.41）则是将预先计算的结果从缓存中取出并比对。该结果一经取出，无论验证正确与否均应予以清除，以保证验证的一致性；下次用户请求验证时即重新生成并比对。

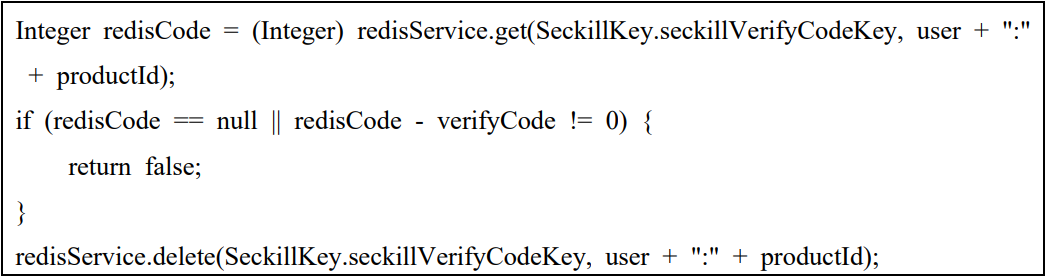


图 5.41 图形验证码验证的实现

### 秒杀地址隐藏

由于HTTP明文传输的特性，所有URL与对应参数均是可见的。为防止不法用户在活动开始前即获取到接口进行秒杀操作，有必要对其接口地址做隐藏。4.4.3节中叙述了ID生成方案雪花算法。本系统在SnowflakeIdUtil中实现了该算法，生成唯一的path作为结果返回给客户端。这样一来，客户端请求真正的秒杀接口时，只需将该path拼接至请求路径/api/seckill/{path}/do中，请求再度进入SeckillController时即可执行seckill()秒杀控制方法。秒杀地址隐藏实现的关键代码如图 5.42所示：

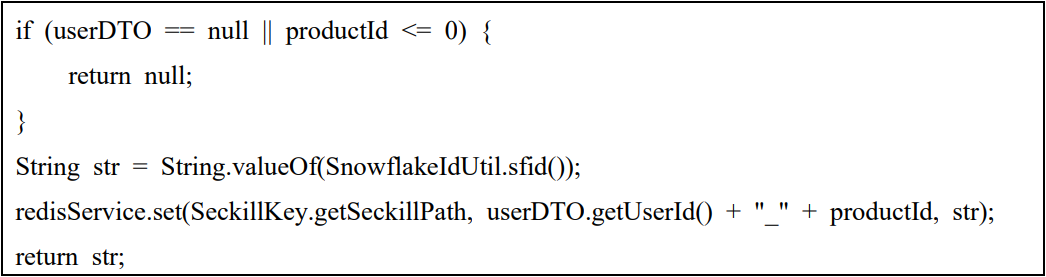


图 5.42 秒杀地址隐藏的实现

获取秒杀路径的UML时序图如图 5.43所示。客户端在收到验证码后，用户需填写该算式的计算结果并提交，此请求交由SeckillController的getSeckillPath()方法处理以获取秒杀路径。首先调用VerifyCodeService的checkSeckillVerifyCode()方法进行验证，验证码不一致将返回SECKILL\_CODE\_ERROR；其次查询缓存中是否已有该用户对此商品的订单信息，如果命中则代表用户此前已执行过秒杀操作，返回REPEAT\_SECKILL，不允许重复秒杀行为。接着进入SeckillService的createSeckillPath()方法生成秒杀路径。最后调用RedisService的set()方法将此路径path缓存在Redis中，其遵照4.4.2节热点数据缓存的设计，带有一个60秒的过期时间，并返回此结果。

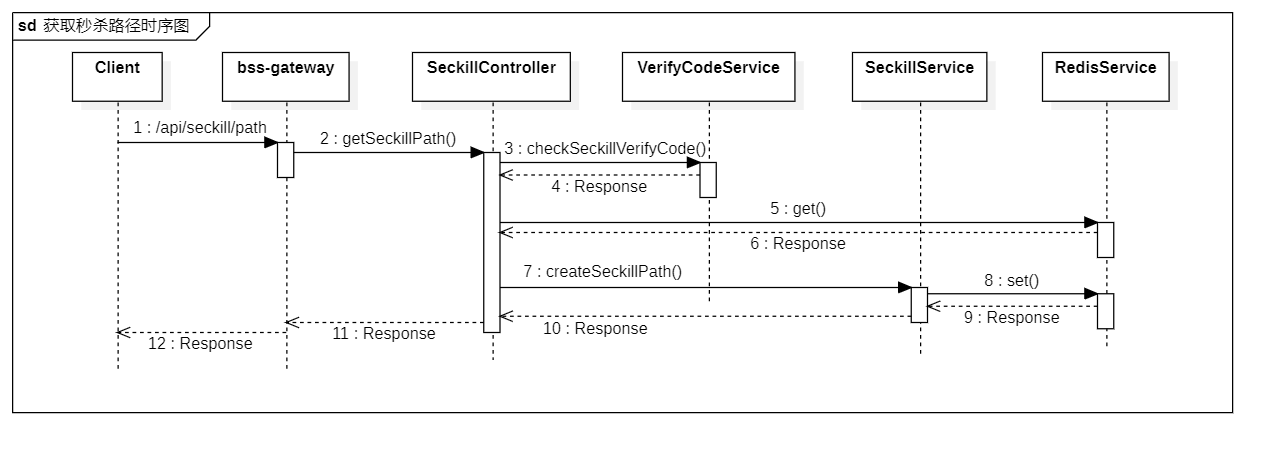


图 5.43 获取秒杀路径时序图

### 秒杀控制逻辑

客户端获取到真正的秒杀路径后发送请求，开始执行秒杀控制逻辑，对应SeckillController的seckill()接口方法。其中，商品超卖是高并发情景下的常见问题，其严重影响系统功能的正常运作。商品出现超卖的核心原因在于，多个线程同时调用该方法，计算库存并赋值给数据库，从而造成数据库中的库存出现负数。出于解决这一问题和性能优化的考量，本系统实现了本地缓存标记、Redis缓存预处理和消息队列异步下单三层缓冲保护，并最大限度地减少对数据库的访问。其实现的关键代码如图 5.44所示：



图 5.44 秒杀控制逻辑的实现

系统首先引入了本地缓存ConcurrentHashMap。ConcurrentHashMap继承自java.util包下的AbstractMap类，是支持高并发查询与更新的HashMap，引入该结构以标记商品的秒杀状态：productId商品若已售罄，则将其对应秒杀状态isOver的值置为true；这样一来，说明商品已经秒杀完毕，下次对该商品的秒杀请求会在这一层返回。

其次，系统引入Redis缓存来预处理库存。在Redis中，incr()、incrby()、decr()和decrBy()方法均为原子操作。在5.6.1节中叙述了秒杀活动开始前已将商品库存预热至Redis中，此时再预扣Redis库存，待订单服务接收秒杀消息后再减少数据库库存。若缓存中的库存已经为0，也说明商品已经秒杀完毕，则无需再减少数据库数据并将直接返回。

需要注意的是，仅通过代码中的流程控制是无法处理对Redis的CAS（check-and-set）命令的。譬如读库存和写库存操作为两个独立的操作，假设商品库存为1，100个请求同时打入seckill()方法并通过库存判断，会直接将缓存中的库存变为-99，这显然不符合业务逻辑，且进一步影响了后续功能如订单取消、订单超时回收时对库存的补偿。通过引入Lua脚本，将上述两个操作放在同一个脚本中执行，从而作为一个原子操作，每次均为单线程操作读写库存，有效保证了缓存中库存信息的正确性。对应的Lua脚本如图 5.45所示：

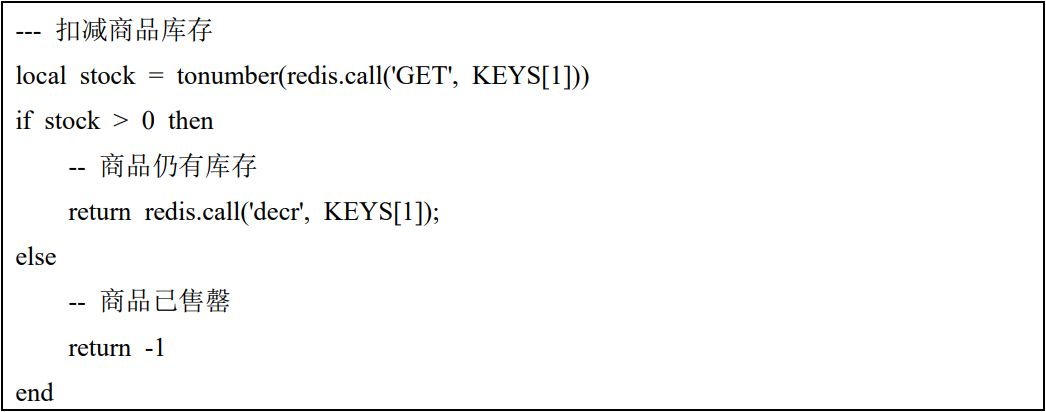


图 5.45 Lua脚本扣减商品库存

接着RabbitMQ将发送异步下单的秒杀消息至seckill.queue，既通过消息队列在秒杀服务和订单服务两个模块之间实现应用解耦，又能达到流量削峰的目的。对应关键代码如图 5.46所示：

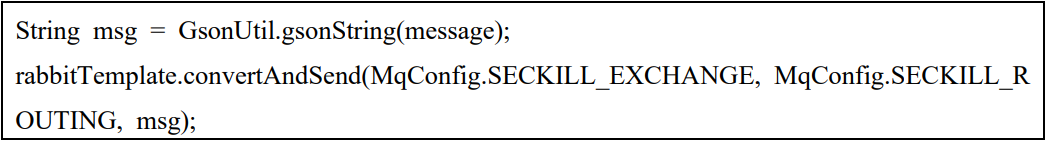


图 5.46 RabbitMQ异步下单的实现

最终在访问Mysql时，系统也在数据库层面做了限制。productId商品售出1件则对应库存减1，附加在where条件中判断stock\_amount > 0限制sql操作，直接杜绝了商品超卖的可能。操作Mysql扣减库存的sql语句如图 5.47所示：

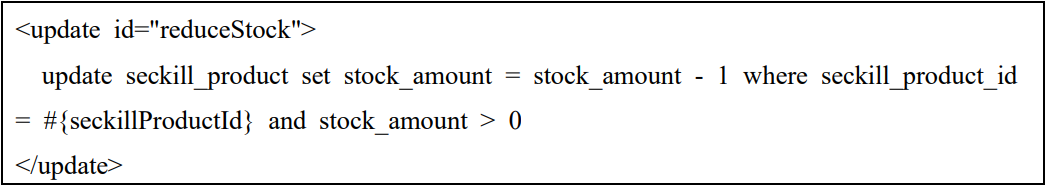


图 5.47 扣减商品库存的sql语句

此外，还需要防止重复下单问题，即单个用户多次成功秒杀到同一商品的情形。seckill()方法中对应代码已经对重复下单行为做了限制，但这是远远不够的。例如用户在极短时间内发出了两次秒杀请求且均有效，则可能出现前一个请求还未秒杀到商品，而后一个请求也执行并通过了判断用户未秒杀过语句的情况。如果数据库不做限制，则最终会为该用户创建两份订单，即该用户完成了重复秒杀行为。因此笔者在seckill.Order对user\_id和product\_id字段建立唯一索引，由于索引列的所有值只能出现一次，从而使得同一用户秒杀该商品仅能有第一个订单插入索引，后续操作事务均将回滚。

秒杀控制的UML时序图如图 5.48所示。请求首先进入SeckillController的seckill()方法，调用SeckillService的checkSeckillPath()方法对秒杀路径进行校验，若路径非法将返回INVALID\_SECKILL\_PATH。其次进入RedisService的get()方法根据userId和productId从缓存中获取订单信息，若已存在该用户对此商品的订单将返回REPEAT\_SECKILL；接着进入luaExecute()方法预扣缓存中的商品库存，最后调用MqSender的sendSeckill()方法执行异步下单操作，返回结果。

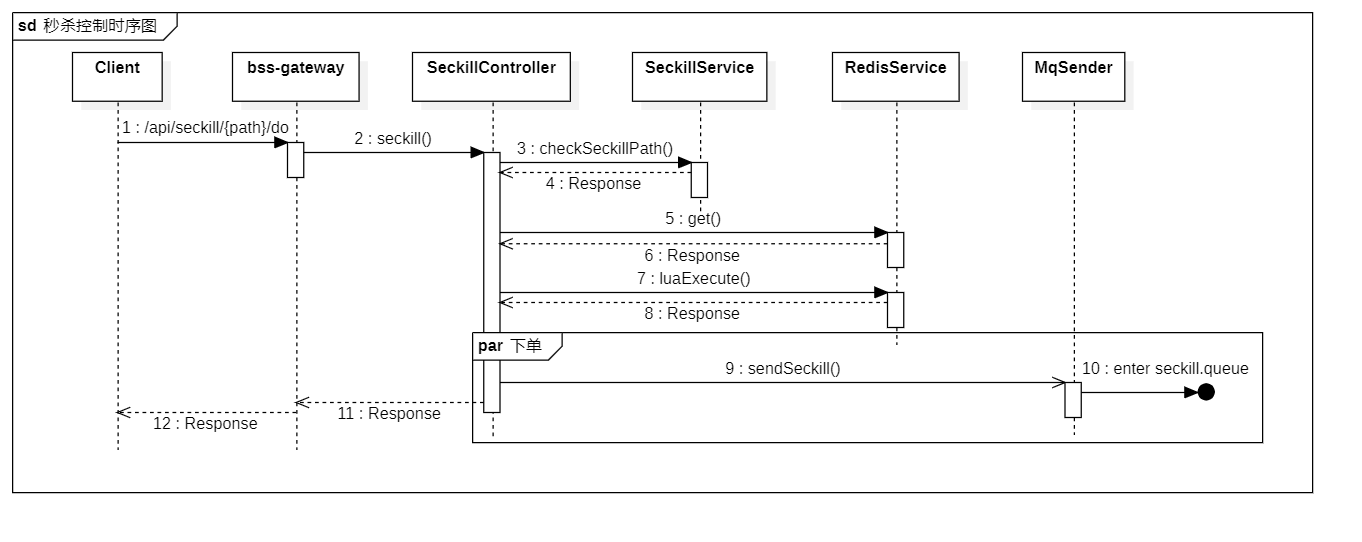


图 5.48 秒杀控制时序图

## 订单服务模块的实现

订单服务主要包括订单创建、订单信息、订单取消和订单管理功能的实现，由OrderController和OrderAdminController类分别处理来自客户端和后台系统的请求。其实现的类图如图 5.49所示：

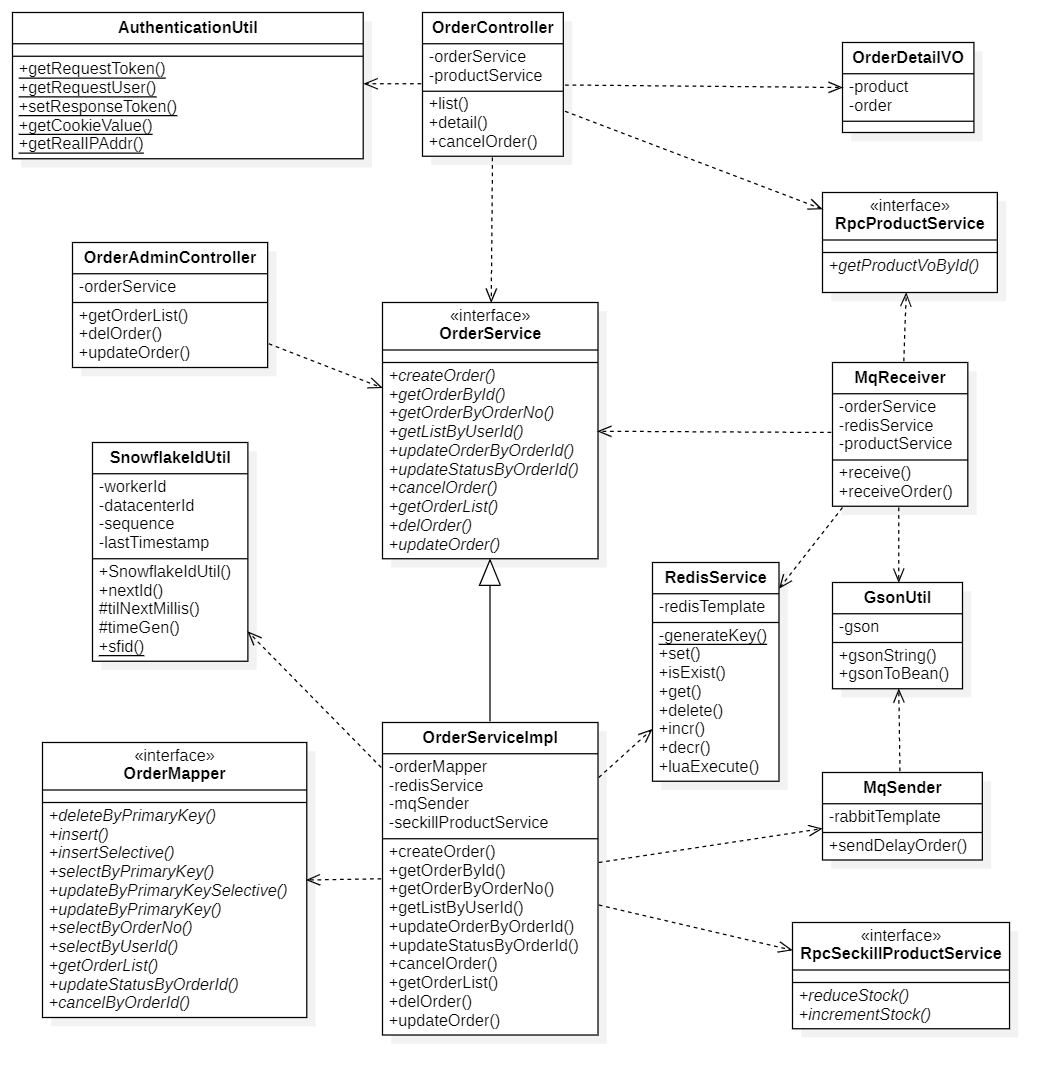


图 5.49 订单服务模块类图

### 订单创建

在5.6节中阐述了秒杀流程的控制逻辑，其中秒杀成功后秒杀服务会经消息队列向本服务发送消息，此时请求首先交给MqReceiver类的receive()方法处理，通过手动ACK确认方便可能的业务处理失败时的重试机制。接收消息后依次判断商品库存和重复下单情况，仅全部通过验证方可继续创建订单。此后由OrderService完成订单创建逻辑，实际扣减库存并写入数据库。其实现的关键代码如图 5.50所示：



图 5.50 订单创建的实现

订单创建的UML时序图如图 5.51所示。首先自seckill.queue中取出来自秒杀服务的消息，接收此消息并取出用户信息和商品id，RPC调用商品服务模块的getProductVoById()方法获取该用户所秒杀的商品信息，经验证后进入OrderService的createOrder()方法创建订单，此时订单的交易状态默认为“未付款”，待用户支付或订单取消后发生相应变化。接着执行OrderMapper的insert()方法插入此订单记录，调用MqSender的sendDelayOrder()方法将对应的订单信息存入order.delay\_queue延迟队列，最后通过RedisService的set()方法同步缓存中的订单信息。

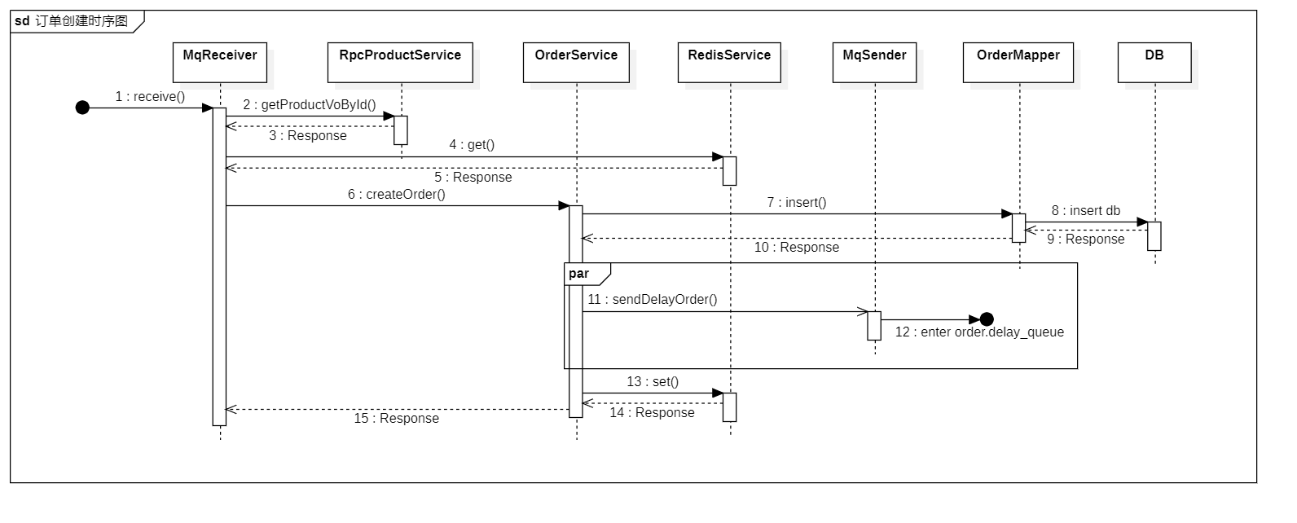


图 5.51 订单创建时序图

### 订单信息

订单信息功能包括订单列表查看与订单详情查看两部分。

订单列表查看的UML时序图如图 5.52所示。用户可在客户端查看所有已创建的订单，其中包括尚未支付或已取消的订单。请求首先在通过网关鉴权后进入OrderController的list()方法，调用OrderService的getListByUserId()方法。其次执行OrderMapper中的selectByUserId()方法，返回该用户的订单列表。最后RPC调用商品服务模块的getProductVoById()方法依次获取每条订单信息，返回列表结果。

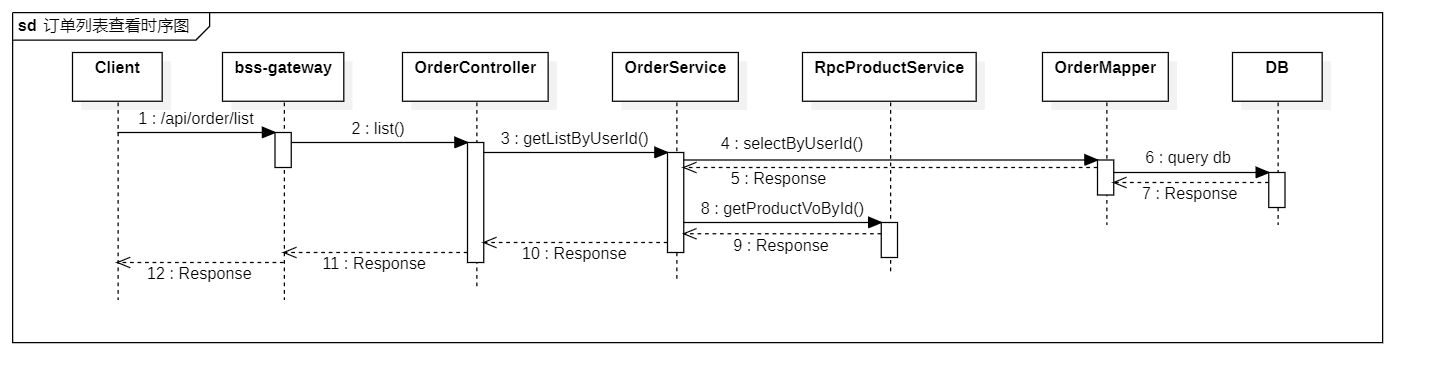


图 5.52 订单列表查看时序图

订单详情查看的UML时序图如图 5.53所示。用户可在客户端查看单个订单详情，此时请求在通过网关鉴权后首先进入OrderController的detail()方法，调用OrderService的getOrderById()方法。其次执行OrderMapper中的selectByPrimaryKey()方法获取该订单信息。最后RPC调用商品服务模块的getProductVoById()方法获取该订单所含商品信息，并将OrderDetailVO视图对象作为结果返回。

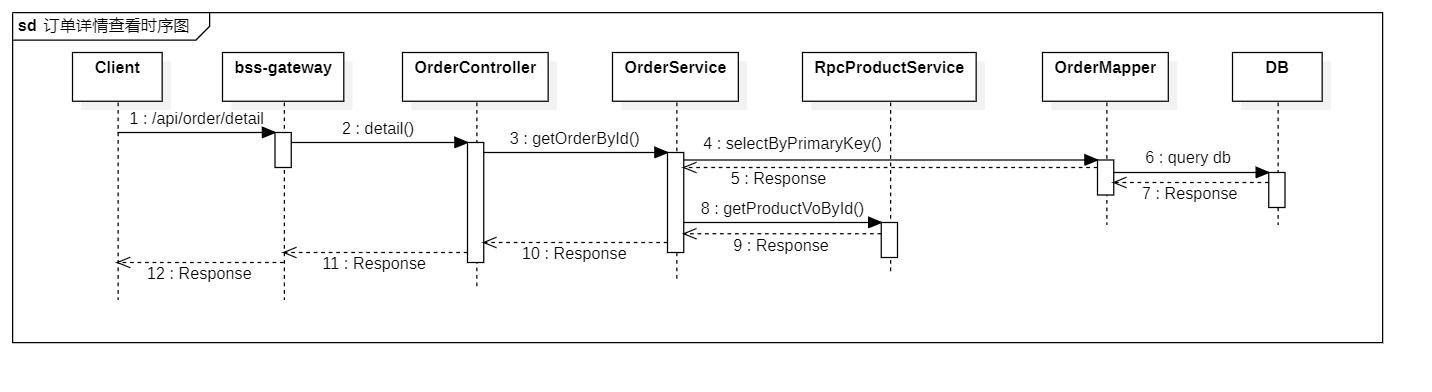


图 5.53 订单详情查看时序图

### 订单取消

订单取消功能包括订单超时回收与手动取消订单两部分。

在5.7.1节中叙述了订单创建功能的实现，其中订单创建后调用了MqSender的sendDelayOrder()方法以将订单信息存入order.delay\_queue队列，此处实际上指定了订单的有效时间ORDER\_DELAY\_TIME，即用户实际可支付订单的时间长度。其实现的关键代码如图 5.54所示：



图 5.54 订单超时回收的实现一

此后消息进入此带TTL的延迟队列order.delay\_queue，该队列没有消费者。消息到期后将被发送至order.receive\_queue，交给MqReceiver类的receiveOrder()方法处理所有的超时订单，手动ACK后接收消息。由OrderService完成订单取消逻辑，实际更改订单的交易状态并补偿库存，此逻辑的实现与手动取消订单子功能相同。其实现的关键代码如图 5.55所示：

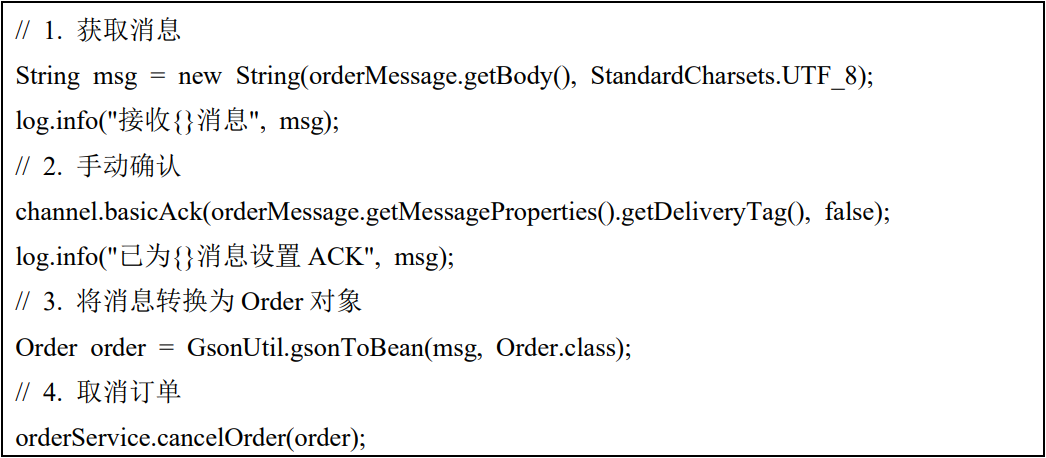


图 5.55 订单超时回收的实现二

订单超时回收的UML时序图如图 5.56所示。先自order.receive\_queue中取出延迟队列发来的消息，接收此消息并取出订单信息，再进入OrderService的cancelOrder()方法取消订单。

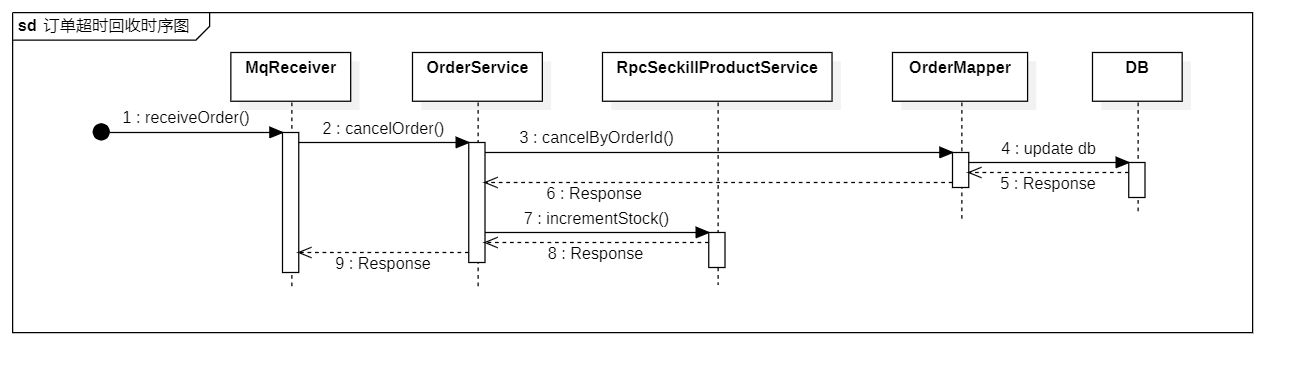


图 5.56 订单超时回收时序图

手动取消订单的UML时序图如图 5.57所示。用户可在客户端取消订单，请求首先通过网关鉴权进入OrderController的cancelOrder()方法，再调用OrderService的cancelOrder()方法，其次执行OrderMapper的cancelByOrderId()方法更新订单的交易状态为“取消交易”，最后RPC调用商品服务模块的incrementStock()方法补偿库存，返回结果。

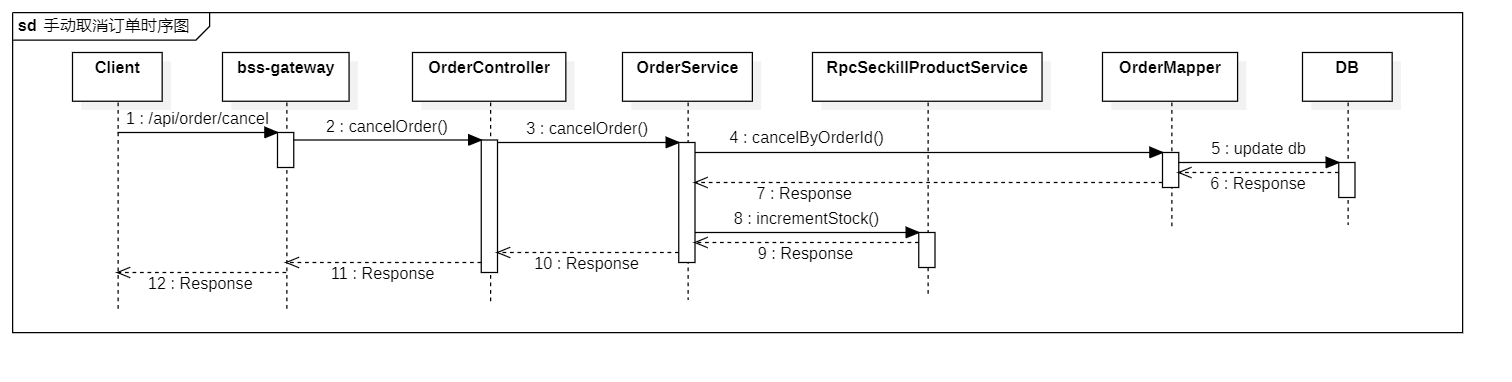


图 5.57 手动取消订单时序图

### 订单管理

管理员可在后台系统管理订单信息。OrderAdminController为后台系统提供API接口服务，可调用OrderService的getOrderList()、delOrder()和updateOrder()方法分别用于查询订单分页列表、删除订单和更新订单信息，并通过OrderMapper与数据库交互。

## 支付服务模块的实现

支付服务主要包括支付和资金管理功能的实现，由PayController和PayAdminController类分别处理来自客户端和后台系统的请求。其实现的类图如图 5.58所示：

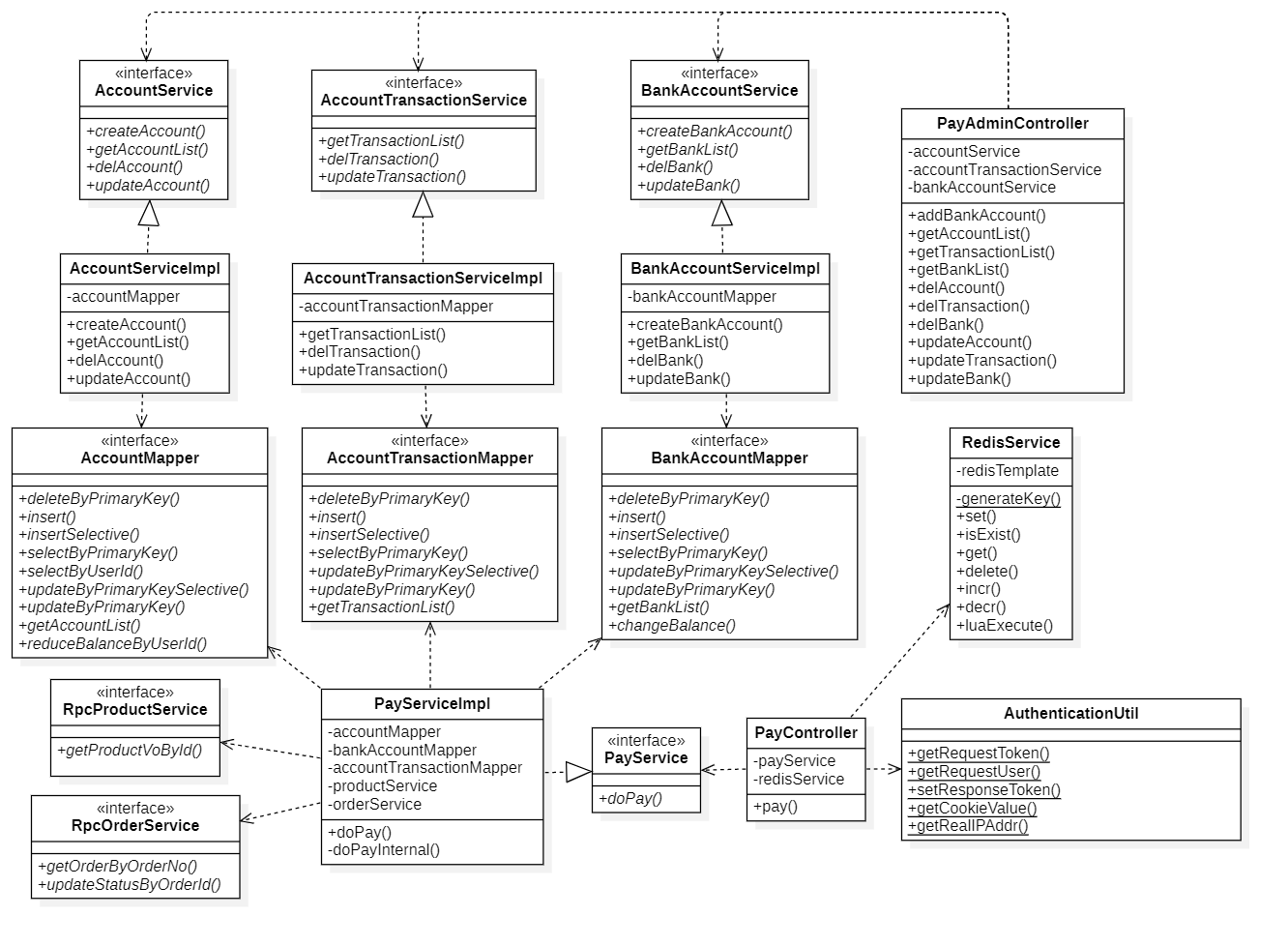


图 5.58 支付服务模块类图

### 支付

支付的UML时序图如图 5.59所示。用户在客户端支付订单，支付信息PayDTO是支付方式与订单号的组合，经客户端网关过滤后被发送至PayController，该类提供pay()方法进行调用。此时首先从缓存中获取此订单的创建此订单的用户id和订单包含的产品id，若请求用户与创建订单用户不相符将返回INCORRECT\_USER。其次RPC调用订单服务模块的getOrderByOrderNo()方法获取订单信息，仅订单的交易状态为“未付款”时方可继续支付。

接着依次执行AccountMapper中的reduceBalanceByUserId()、selectByUserId()和accountTransactionMapper的insertSelective()方法从用户的个人账户中扣款并记录流水，在数据库中存入该流水信息。这样以后RPC调用商品服务模块的getProductVoById()方法获取存款过渡户，并执行BankAccountMapper的changeBalance()方法转账。最后RPC调用订单服务模块的updateStatusByOrderId()方法更新订单的交易状态为“已付款”，返回支付成功结果。

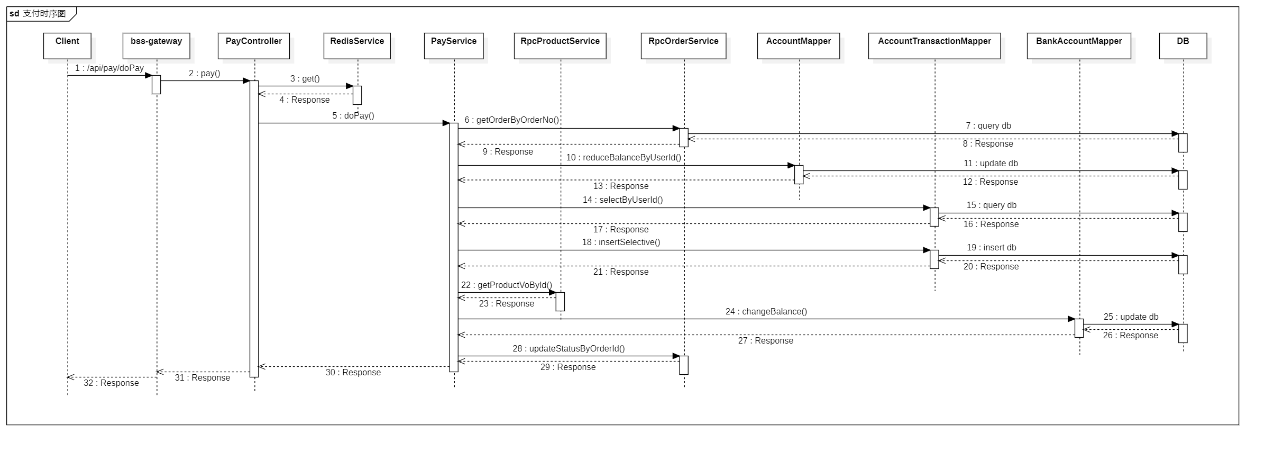


图 5.59 支付时序图

### 资金管理

管理员可在后台系统进行资金管理，由PayAdminController为后台系统提供API接口服务：可调用AccountService、AccountTransactionService和BankAccountService的CRUD方法分别用于查询账户分页列表、删除账户和更新账户信息，并通过AccountMapper、AccountTransactionMapper和BankAccountMapper与数据库交互。

## 本章小结

本章首先介绍了本系统的开发环境与工具，再阐述了服务端微服务组件的实现，包括服务注册与发现、服务网关、服务调用和服务容错，再依次叙述了各模块的实现，其中重点阐述了秒杀相关问题的解决思路与实施方案。

# 银行秒杀系统服务端的测试

## 接口测试

接口测试是对被测系统各组件的功能性、稳定性进行验证[30]，常同软件的功能测试相结合，目的在于验证系统功能是否可用，系统各模块是否正常运作。由于接口用于连接前后端，接口测试是为黑盒测试。本节使用Postman模拟HTTP的各种请求，对本系统服务端各功能进行接口测试。

### 用户服务模块接口测试

用户服务模块的接口测试结果如表 6.1所示。

表 6.1 用户服务接口测试结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **接口功能** | **接口地址** | **接口参数** | **请求方式** | **测试结果** |
| 1 | 用户注册 | /api/user/register | JSONObject | POST | 正常 |
| 2 | 用户登录 | /api/user/login | JSONObject | POST | 正常 |
| 3 | 获取个人信息 | /api/user/userInfo | token | POST | 正常 |
| 4 | 修改密码 | /api/user/changePwd | token, JSONObject | POST | 正常 |
| 5 | 管理员登录 | /admin/user/login | JSONObject | POST | 正常 |
| 6 | 查询用户分页列表 | /admin/user/getUserList?{page}&{size} | token, page, size | POST | 正常 |
| 7 | 删除用户 | /admin/user/delUser?{id} | token, id | DELETE | 正常 |
| 8 | 更新用户信息 | /admin/user/updateUser | token, JSONObject | POST | 正常 |

### 商品服务模块接口测试

商品服务模块的接口测试结果如表 6.2所示。

表 6.2 商品服务接口测试结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **接口功能** | **接口地址** | **接口参数** | **请求方式** | **测试结果** |
| 1 | 获取商品列表 | /api/product/list | token | POST | 正常 |
| 2 | 获取商品详情 | /api/product/detail?{productId} | token, productId | POST | 正常 |
| 3 | 添加产品 | /admin/product/addProduct | token, JSONObject | POST | 正常 |
| 4 | 添加商品 | /admin/product/addSeckill | token, JSONObject | POST | 正常 |
| 5 | 查询产品分页列表 | /admin/product/getProductList?{page}&{size} | token, page, size | POST | 正常 |
| 6 | 查询商品秒杀活动分页列表 | /admin/product/getSeckillList?{page}&{size} | token, page, size | POST | 正常 |
| 7 | 删除产品 | /admin/product/delProduct?{id} | token, id | DELETE | 正常 |
| 8 | 删除商品秒杀活动 | /admin/product/delSeckill?{id} | token, id | DELETE | 正常 |
| 9 | 更新产品信息 | /admin/product/updateProduct | token, JSONObject | POST | 正常 |
| 10 | 更新商品秒杀活动信息 | /admin/product/updateSeckill | token, JSONObject | POST | 正常 |

### 筛查服务模块接口测试

筛查服务模块的接口测试结果如表 6.3所示。

表 6.3 筛查服务接口测试结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **接口功能** | **接口地址** | **接口参数** | **请求方式** | **测试结果** |
| 1 | 记录 | /api/screen/record | token | POST | 正常 |
| 2 | 记录并筛查 | /api/screen/do | token, productId | POST | 正常 |
| 3 | 添加规则 | /admin/screen/addRule | token, JSONObject | POST | 正常 |

表 6.3（续）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **接口功能** | **接口地址** | **接口参数** | **请求方式** | **测试结果** |
| 4 | 查询申请记录分页列表 | /admin/screen/getApplyList?{page}&{size} | token, page, size | POST | 正常 |
| 5 | 查询规则分页列表 | /admin/screen/getRuleList?{page}&{size} | token, page, size | POST | 正常 |
| 6 | 删除申请记录 | /admin/screen/delApply?{id} | token, id | DELETE | 正常 |
| 7 | 删除规则 | /admin/screen/delRule?{id} | token, id | DELETE | 正常 |
| 8 | 更新申请记录信息 | /admin/screen/updateApply | token, JSONObject | POST | 正常 |
| 9 | 更新规则信息 | /admin/screen/updateRule | token, JSONObject | POST | 正常 |

### 秒杀服务模块接口测试

秒杀服务模块的接口测试结果如表 6.4所示。

表 6.4 秒杀服务接口测试结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **接口功能** | **接口地址** | **接口参数** | **请求方式** | **测试结果** |
| 1 | 获取秒杀图形验证码 | /api/seckill/verifyCode/{productId} | token, productId | GET | 正常 |
| 2 | 获取秒杀路径 | /api/seckill/path?{productId}&{code} | token, productId, code | POST | 正常 |
| 3 | 秒杀 | /api/seckill/{path}/do?{productId} | token, path, productId | POST | 正常 |
| 4 | 查询秒杀结果 | /api/seckill/result?{productId} | token, productId | POST | 正常 |

### 订单服务模块接口测试

订单服务模块的接口测试结果如表 6.5所示。

表 6.5 订单服务接口测试结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **接口功能** | **接口地址** | **接口参数** | **请求方式** | **测试结果** |
| 1 | 获取订单列表 | /api/order/list | token | POST | 正常 |
| 2 | 获取订单详情 | /api/order/detail?{orderId} | token, orderId | POST | 正常 |
| 3 | 取消订单 | /api/order/cancel/{orderId} | token, orderId | DELETE | 正常 |
| 4 | 查询订单分页列表 | /admin/order/getOrderList?{page}&{size} | token, page, size | POST | 正常 |
| 5 | 删除订单 | /admin/order/delOrder?{id} | token, id | DELETE | 正常 |
| 6 | 更新订单信息 | /admin/order/updateOrder | token, JSONObject | POST | 正常 |

### 支付服务模块接口测试

支付服务模块的接口测试结果如表 6.6所示。

表 6.6 支付服务接口测试结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **接口功能** | **接口地址** | **接口参数** | **请求方式** | **测试结果** |
| 1 | 支付 | /api/pay/doPay | token | POST | 正常 |
| 2 | 添加银行账户 | /admin/pay/addBankAccount | token, JSONObject | POST | 正常 |
| 3 | 查询个人账户分页列表 | /admin/pay/getAccountList?{page}&{size} | token, page, size | POST | 正常 |
| 4 | 查询账户流水分页列表 | /admin/pay/getTransactionList?{page}&{size} | token, page, size | POST | 正常 |
| 5 | 查询银行账户分页列表 | /admin/pay/getBankList?{page}&{size} | token, page, size | POST | 正常 |
| 6 | 删除个人账户 | /admin/pay/delAccount?{id} | token, id | DELETE | 正常 |

表 6.6（续）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **接口功能** | **接口地址** | **接口参数** | **请求方式** | **测试结果** |
| 7 | 删除账户流水 | /admin/pay/delBank?{id} | token, id | DELETE | 正常 |
| 8 | 删除银行账户 | /admin/pay/delTransaction?{id} | token, id | DELETE | 正常 |
| 9 | 更新个人账户信息 | /admin/pay/updateAccount | token, JSONObject | POST | 正常 |
| 10 | 更新账户流水信息 | /admin/pay/updateTransaction | token, JSONObject | POST | 正常 |
| 11 | 更新银行账户信息 | /admin/pay/updateBank | token, JSONObject | POST | 正常 |

## 压力测试

性能测试是保证Web性能的重要手段，通过测试工具模拟运行被测系统以获取系统的各项性能指标[31]，从而为系统的优化提供可靠的依据。

表 6.7 Consul集群部署

|  |  |
| --- | --- |
| **容器** | **容器IP地址** |
| consul-server1 | 172.20.0.2/16 |
| consul-server2 | 172.20.0.3/16 |
| consul-server3 | 172.20.0.4/16 |
| consul-client | 172.20.0.5/16 |

首先使用docker-compose编排并在云服务器构建Consul集群（表 6.7和图 6.1），如2.2.1节所述包含三个Server节点和一个Client节点，创建一个172.20.0.0/16子网并桥接至Docker容器网桥（docker0），容器间彼此经由对应的Container IP通信。

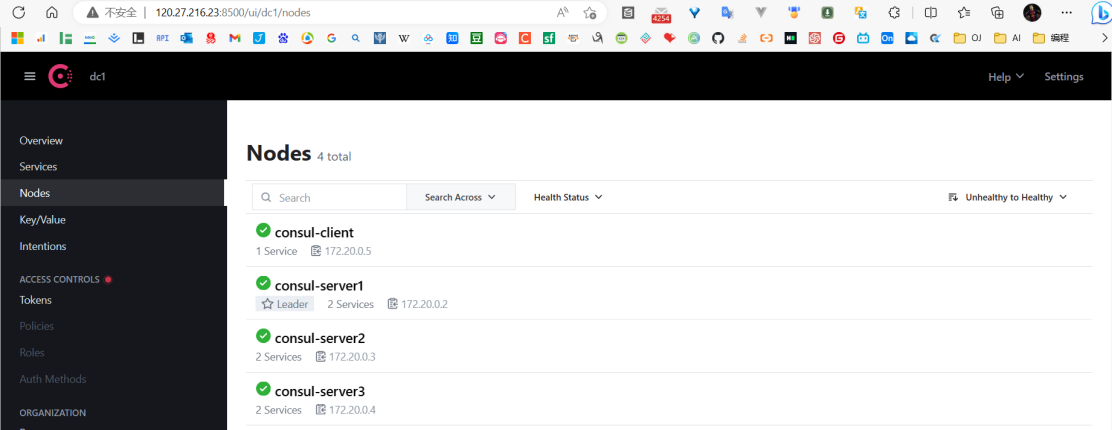


图 6.1 一个典型Consul集群的节点状况

Jmeter是常用的性能测试工具。创建一个Jmeter测试脚本（图 6.2），设定线程组的线程数、Ramp-up时间与循环次数。

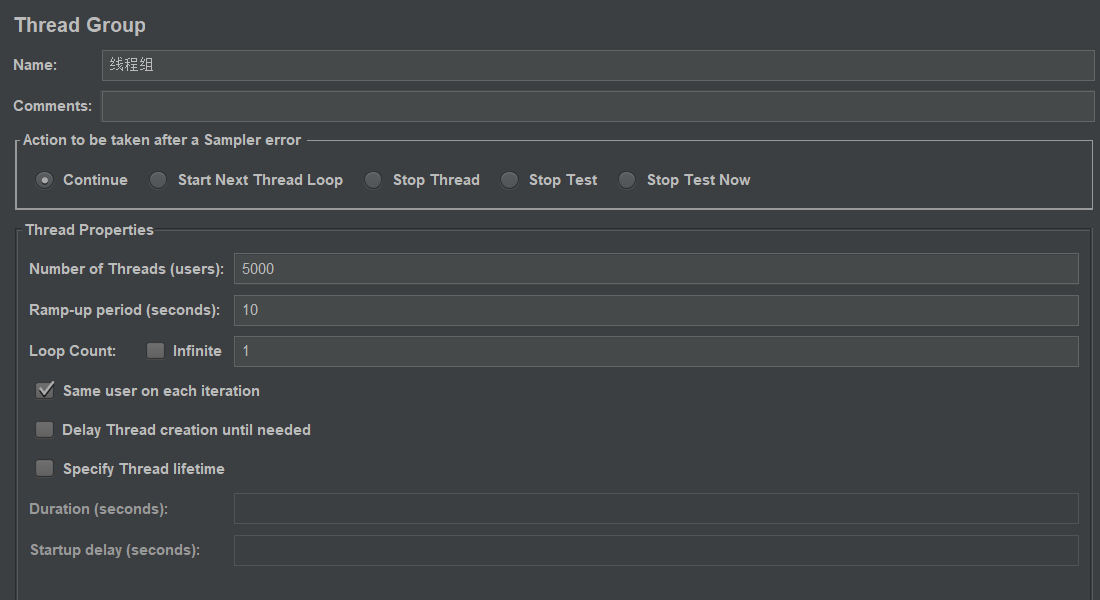


图 6.2 创建性能测试脚本，设定线程组

在用户服务模块的UserUtil中产生测试的用户数据，生成5000名用户并分批插入数据库。其关键代码如图 6.3所示：

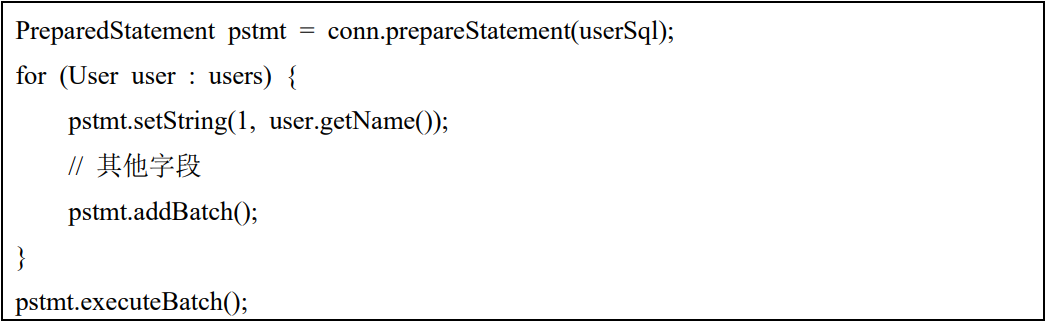


图 6.3 测试用户生成的实现

其后模拟用户登录，生成并保存对应的token信息。创建文件用于存放生成的token信息，并通过内容访问类RandomAccessFile进行操作。对每一个用户，依次创建请求并添加登录的请求参数mobile和password后发出，获取响应并将token写入txt文件（图 6.4）。

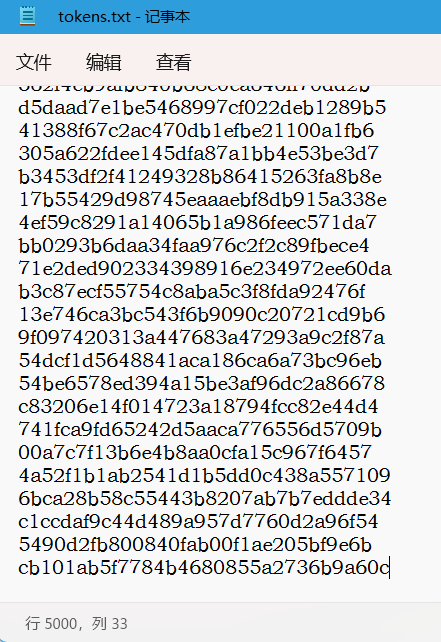


图 6.4 模拟用户的token信息

通过CSV数据文件设置，指定该txt文件的读取位置，提供token供后续接口操作处理。此后在jmeter脚本的该线程组中依次新增秒杀、秒杀结果的HTTP请求（图 6.5）。

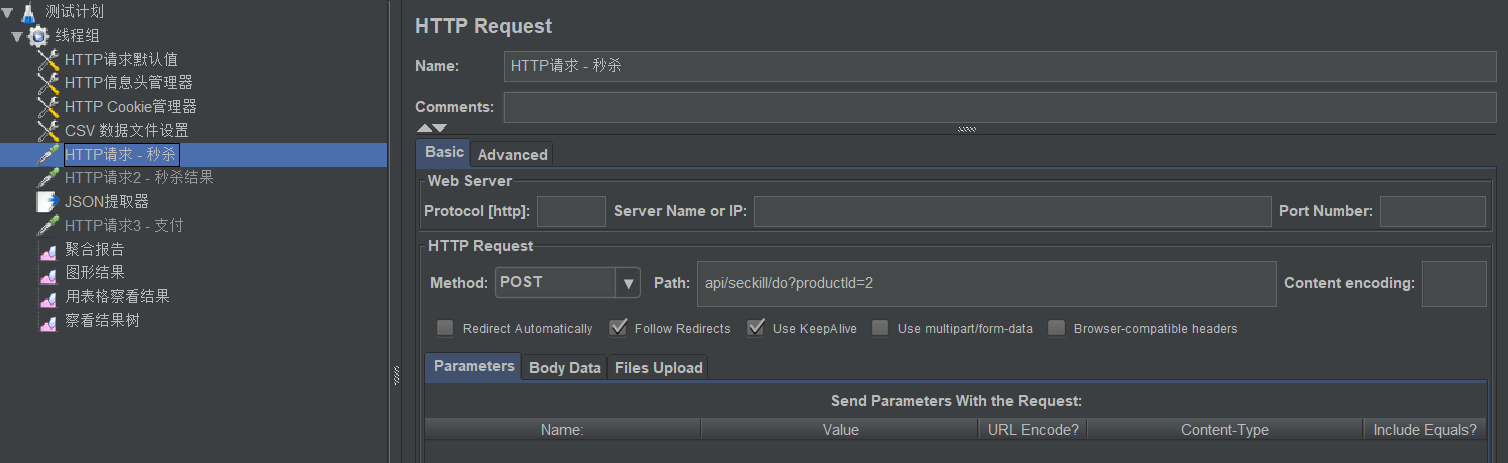


图 6.5 秒杀HTTP请求

通过JSON提取器从响应中提取秒杀成功所创建的订单号（图 6.6），并作为请求支付接口的参数（图 6.7）。



图 6.6 JSON提取器提取订单号

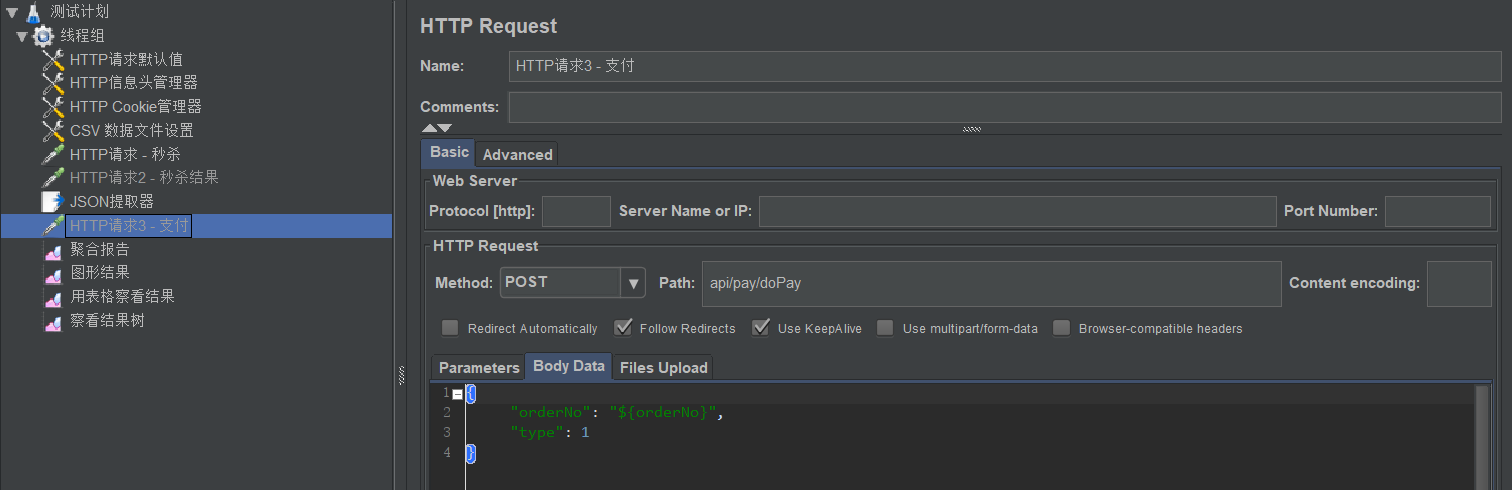


图 6.7 支付HTTP请求

此后先启用秒杀请求，关闭网关的限流功能，开始压力测试。经测试，5000名用户抢购300件商品成功，从首位用户请求执行秒杀到商品全部售完合计耗时在3s以内，商品售罄后数据库与缓存的库存均为0，没有发生超卖现象；消息队列中相应的300条消息从seckill.queue进入order.delay\_queue，状态为ready；数据库中创建了300份订单，其交易状态均为“未付款”。这表明秒杀接口工作正常，且系统秒杀服务为成功抢购的用户发送消息，订单服务接收后异步创建订单成功。

开启秒杀结果与支付请求后再次测试。经测试，订单实际交易状态为“已付款”的订单数目等于活动设置的商品库存，对应个人账户扣除了订单金额，产品存款过渡户的账户余额亦为商品价格与商品数目的乘积，账目无误。以上测试表明，系统能够满足所需的性能指标，通过了压力测试。

## 本章小结

本章先介绍了本系统的测试环境与工具，其后叙述了各模块的接口测试与系统的压力测试流程。经测试，系统表现符合预期。

# 总结与展望

## 本文小结

本文为解决目前银行推出存款产品上线并投放秒杀活动的需求，设计并实现了一套基于微服务的银行秒杀系统的服务端，其功能较为完善，秒杀场景下的性能表现良好，且基于微服务的架构有效提高了系统的可扩展性。论文的主要内容及本人完成的工作归纳如下：

首先介绍了课题的研究背景与意义，分别介绍了国内外的微服务与秒杀系统的研究现状以及本文的主要研究工作和结构安排。

第二部分介绍了Spring Cloud微服务框架和对应的组件选型，介绍了项目采用的RabbitMQ消息中间件，使用Mysql作数据存储，采用Redis作数据缓存等。这些都是目前较为热门的开源技术，有助于增强系统的稳定性与安全性。

第三部分论证了系统服务端的分析，从技术、操作和经济三个层面论证了本系统的可行性，通过UML用例图分析了管理员和客户各自的功能需求，以活动图形式描述了秒杀的具体流程，并简要分析了非功能需求。

第四部分介绍了系统服务端的设计，提出了基于微服务的四层软件架构设计和数据库设计，并依次阐述了各功能模块设计、性能优化设计和安全性设计，辅以流程图、时序图呈现设计思路。

第五部分阐述了系统服务端的实现，利用代码、类图和时序图的形式，重点阐述了微服务组件中服务注册与发现、服务网关、服务调用与服务容错的实现，秒杀服务的库存预热、图形验证、地址隐藏与整体控制逻辑的实现，以及订单服务中异步创建订单与取消的实现。

最后叙述了系统服务端的测试，得出测试结果并验收系统。

## 展望

本文完成了基于微服务的银行秒杀系统的服务端的设计与实现，通过了接口测试与压力测试，能够投入到实际的生产环节中。但一个成熟的系统还需在现有的基础上精益求精，从长远来看本系统还存在以下优化方向：

1. 应用诸如Elastic Search等搜索引擎完善商品服务，提供商品搜索功能方便用户快速地找到心仪的商品；
2. 采用AES、SM2等算法完成前后端传输数据加密，在现有服务端对敏感数据进行加密处理的基础上，进一步提升系统用户数据的安全性；
3. 进一步优化秒杀业务的有关策略，寻找更为高效的解决方案以提升系统性能；
4. 通过Jenkins实现持续集成和持续部署，提升系统的可维护性，并可经由Kubernetes动态部署以实现开发运维一体化。

# 致谢

时光飞逝。行笔至此，现如今回想大学的青葱时光，不由得思绪万千。

是不幸亦是幸运，作为19级的一分子，首个学期短暂的繁华褪去后，大部分在校时间都是在封闭管理中度过，好似时间被慢慢拉长，日子也少了些烟火气。不过，这富余反倒也给了更多时间去提升自我。少了许多不必要的琐事，生活也变得轻盈起来。

最早与肖老师谋面，是在大一下学期《程序设计基础》的课堂。转专业后课业繁重，专业课程学的迷迷糊糊，颇有分身乏术之感。在家中线上学习少有氛围，偏偏又被抽到回答问题，实是雪上加霜。好在自己一通瞎诌过后，并不曾被责备几分，甚至受到了老师的鼓励。或许正是从那个时候开始，平添了不少学习的动力。此后我也在其余数门专业课上聆听肖老师的教诲，潜心钻研专业技能；在我参加学科竞赛等一系列旅程中，老师也不吝指点迷津；本论文亦是在肖斌老师的悉心指导下完成的，在此我衷心地向您表示感谢。

由榕入蓉，是横跨两千公里的乡愁；二十余载，是父母含辛茹苦的呵护。在外求学本不是件易事，只因有了你们的陪伴、爱与支持而变得稀松平常。年少时的自己多有彷徨，而家永远是那个温暖的避风港。感激之情，无以言表。

此外，还要感谢大学的舍友和同窗。即便时至今日也没能学好四川话、完全适应美食的辣度，也和大家一同简单、欢乐且充实地度过了这四年的美好时光。有幸结伴而行，感谢相遇。

凡心所向，素履所往；生如逆旅，一苇以航。最后也要感谢这一路的自己，在无数个可能的候选项中找到了适合自己的方向。人生并无所谓标准答案，但求怡然自得。前路漫漫，来日方长。

# 参考文献

Bruce Jay Nelson. "The Design and Implementation of a High-Performance Transport Protocol". In: [会议或期刊名称]. [出版社]. [出版年份]. pp.[页码].

1. Bruce Jay Nelson. "The Design and Implementation of a High-Performance Transport Protocol". In: [会议或期刊名称]. [出版社]. [出版年份]. pp.[页码].
2. "Effective Java (Second Edition)" by Joshua Bloch. Chapter 5, Item 23: "Prefer interfaces to reflection" provides a detailed explanation of Java's dynamic proxy mechanism.
3. "TCP/IP详解 卷1：协议"，作者：Steven W. McVoy，出版社：人民邮电出版社。
4. "TCP/IP详解 卷3：TCP"，作者：W. Richard Stevens，出版社：人民邮电出版社。

中国互联网络中心. CNNIC发布第51次《中国互联网络发展状况统计报告》[EB/OL]. (2023-03-02) [2023-03-10]. https://www.cnnic.net.cn/n4/2023/0302/c199-10755.html

Martin Fowler. Microservices[EB/OL]. (2014-03-25) [2023-03-11]. https://martinfowler.com/articles/microservices.html

郭栋,王伟,曾国荪.一种基于微服务架构的新型云件PaaS平台[J].信息网络安全,2015(11):15-20.

洪华军,吴建波,冷文浩.一种基于微服务架构的业务系统设计与实现[J].计算机与数字工程,2018,46(01):149-154.

王方旭.基于Spring Cloud实现业务系统微服务化的设计与实现[J].电子技术与软件工程,2018(08):60-61.

廖俊杰,陶智勇.微服务API网关的设计及应用[J].自动化技术与应用,2019,38(08):85-88.

Krylovskiy A , Jahn M , Patti E . Designing a Smart City Internet of Things Platform with Microservice Architecture[C]// International Conference on Future Internet of Things & Cloud. IEEE, 2015.

B. Mayer and R. Weinreich, "A Dashboard for Microservice Monitoring and Management," 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 2017, pp. 66-69, doi: 10.1109/ICSAW.2017.44.

S. Kapembe and J. Quenum, "Lihonga — A Microservice-Based Virtual Learning Environment," 2018 IEEE 18th International Conference on Advanced Learning Technologies (ICALT), Mumbai, India, 2018, pp. 98-100, doi: 10.1109/ICALT.2018.00030.

李军锋,何明昕.高并发Web航空票务秒杀系统的设计与实现[J].计算机工程与设计,2013,34(03):778-782.DOI:10.16208/j.issn1000-7024.2013.03.027.

W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 2017, pp. 243-246, doi: 10.1109/ICSAW.2017.11.

董晓玮,赵月英,张一鸣.电商秒杀系统的设计与实现[J].信息技术与信息化,2020(09):40-42.

侯成程. 某订单秒杀系统的设计与实现[D].北京邮电大学,2021.DOI:10.26969/d.cnki.gbydu.2021.001384.

翟剑锟. Spring框架技术分析及应用研究[D].中国科学院大学(工程管理与信息技术学院),2013.

王永和,张劲松,邓安明等.Spring Boot研究和应用[J].信息通信,2016,No.166(10):91-94.

Ebert C, Gallardo G, Hernantes J, et al. DevOps[J].IEEE Software,2016,33(3):94-100.

Spring Team. Spring Cloud[EB/OL]. [2023-03-14]. https://spring.io/projects/spring-cloud.

王方旭.基于Spring Cloud实现业务系统微服务化的设计与实现[J].电子技术与软件工程,2018,No.130(08):60-61.

辛园园,钮俊,谢志军等.微服务体系结构实现框架综述[J].计算机工程与应用,2018,54(19):10-17.

万立超. 高并发环境下微服务网关的研究与设计[D].南昌大学,2022.DOI:10.27232/d.cnki.gnchu.2022.004563.

郭致远,魏银珍.基于Spring Cloud服务调用的设计与应用[J].信息技术与网络安全,2019,38(02):87-91.DOI:10.19358/j.issn.2096-5133.2019.02.022.

张开琦. 微服务架构负载均衡及服务容错研究[D].昆明理工大学,2021.DOI:10.27200/d.cnki.gkmlu.2021.000777.

吴璨,王小宁,肖海力等.分布式消息系统研究综述[J].计算机科学,2019,46(S1):1-5+34.

曾超宇,李金香.Redis在高速缓存系统中的应用[J].微型机与应用,2013,32(12):11-13.DOI:10.19358/j.issn.1674-7720.2013.12.004.

Morgan Bruce，Paulo A. Pereira.《微服务实战》[M]：人民邮电出版社，2020

梁润强,史伟.基于路由和QoS令牌桶的集中式限速网关[J].网络安全技术与应用,2019,No.223(07):19-20.

邱祝文.基于redis的分布式缓存系统架构研究[J].网络安全技术与应用,2014,No.166(10):52+54.

张晋源,袁丽欧.探析关于图形验证码的安全性[J].电脑编程技巧与维护,2017,No.378(12):76-77.DOI:10.16184/j.cnki.comprg.2017.12.026.

王小云,于红波.SM3密码杂凑算法[J].信息安全研究,2016,2(11):983-994.

何琼月.软件测试中接口测试概述与实践[J].电子测试,2021,No.455(02):80-81+75.DOI:10.16520/j.cnki.1000-8519.2021.02.027.

唐承玲,王虎,李光平等.基于JMeter的Web性能测试研究[J].电脑与电信,2021,No.294(06):65-68+86.DOI:10.15966/j.cnki.dnydx.2021.06.017.