研 究 生 毕 业 论 文

（申请工程硕士学位）

论文题目	履约超时管理平台的设计与实现
作者姓名	
学科、专业名称	工程硕士（软件工程领域）
研究方向	软件工程
指导教师	

2020 年　　月　　日

学　　号：

论文答辩日期：　　2020 年　月　日

指 导 教 师：　　（签字）

履约超时管理平台的
设计与实现

作　　者:

指导教师:

（申请工程硕士学位）

2020 年　月

The Design and Implementation of Plateform of Performance Overtime Management

Submitted in partial fulfillment of the requirements for the degree of Master of Engineering

Supervised by

2020

摘 要

随着社会发展，外卖已经成为城市上班族解决吃饭问题的首选方式。餐饮外卖行业是一个对履约超时很敏感的行业，订单履约超时会降低用户满意度，为了提高用户满意度，外卖平台需要主动检查外卖履约是否超时，如果超时，需要主动执行超时应对策。饿了么既有超时判定逻辑分散在各个系统中，例如：赔付系统、用户触达系统、逆向系统等。随着业务发展，饿了么既有超时判定方式已经无法满足需求：由于判定逻辑分散在各个系统中，修改判定逻辑需要协调多系统，开发流程繁琐，工作量大，导致无法快速响应需求；由于无法做到快速响应需求，运营人员无法做到细粒度把控判定逻辑。为了解决上述问题，本文设计并实现了履约超时管理平台，平台提供线上制定检查规则、应对策略，规则快速生效，实时判定订单履约是否超时等功能。

本文说明了项目背景，阐述了履约超时管理平台在履约服务中的作用，介绍了国内外发展现状。综述了包括 Spring Boot 框架、RabbitMQ 消息队列、E-Scheduler 任务调度平台、Redis 缓存、异地多活技术等构建系统时采用的技术。在系统分析与设计中，本文说明了系统边界，详细分析了系统的功能需求和非功能需求，提炼了规则操作、规则预处理、审核流操作、规则应用四个功能。在系统总体设计中，设计了系统架构、规则、功能组成和系统部署。通过 ER 图设计了数据库。论文通过类图、顺序图等方法，对规则操作、规则预处理、审核流操作、规则应用四个模块进行了详细设计，给出了具体实现细节，并进行了系统测试。

该平台已经投入实际使用，较好地解决了运营需求无法得到快速响应的问题，进一步有效预警和处理履约超时情景。

关键字：履约超时，规则，可配置化，快速响应需求

Abstract

With the development of society, take out has become the first choice for urban office workers to solve the problem of eating. Food and beverage delivery industry is a very sensitive industry to the performance timeout. The order performance timeout will reduce user satisfaction. In order to improve customer satisfaction, takeout platform needs to take the initiative to check whether the takeaway is over time. Eleme's existing timeout logic is distributed among various systems, such as the compensation system, the user touch system, and the reverse system. With the development of business, the existing timeout judgment mode can no longer meet the demand: due to the judgment logic is scattered in various systems, the modification of the judgment logic needs to coordinate multiple systems, the development process is tedious, and the workload is large, resulting in the inability to respond quickly to the demand; due to the rapid response cannot be achieved, the operators cannot achieve the fine-grained control judgment logic. In order to solve the above problems, this thesis designs and implements the performance timeout management platform, which provides online inspection rules, response strategies, rules come into effect quickly, real-time determination of order performance timeout and other functions.

This thesis describes the background of the project, describes the function of the performance overtime management platform in the performance service, and introduces the development status at home and abroad. This thesis summarizes the technologies used to build the system, including Spring Boot framework, RabbitMQ message queue, E-Scheduler task scheduling platform, Redis cache, Remote multi live data center, etc. In the system analysis and design, this thesis describes the system boundary, analyzes the functional and non-functional requirements of the system in detail, and extracts four functions: rule operation, rule preprocessing, audit flow operation and rule application. In the overall design of the system, the system architecture, rules, functional composition and system deployment are designed. The database is designed by ER diagram. Through class diagram, sequence diagram and other methods, the thesis designs four modules in detail,

including rule operation, rule preprocessing, audit flow operation and rule application, gives the specific implementation details, and tests the system.

The platform has been put into practical use, which can better solve the problem that demand of operators cannot be quickly responded, and further effectively early warning and handling the performance timeout situation.

KeyWords: Performance timeout, Rule, Configurable, Fast response to demand

目录

摘　要 I
Abstract II
目录 IV
图目录 VI
表目录 VII
第一章 引言 1
1.1 项目背景 1
1.2 国内外发展现状 2
1.3 本文主要工作 3
1.4 本文组织结构 4
第二章 技术综述 5
2.1 SpringBoot 5
2.2 RabbitMQ 5
2.3 E-Scheduler 7
2.4 Redis 8
2.5 异地多活技术 9
2.6 本章小结 11
第三章 系统分析与设计 12
3.1 系统边界 12
3.2 业务描述 13
3.3 需求分析 14
3.3.1 规则管理功能 14
3.3.2 规则预处理功能 16
3.3.3 审核流程管理功能 18
3.3.4 规则应用功能 21
3.3.5 非功能需求 24
3.4 系统总体设计 24
3.4.1 规则设计 24
3.4.2 架构设计 29
3.4.3 功能组成设计 31
3.4.4 部署设计 31
3.5 数据库设计 33
3.6 本章小结 36
第四章 详细设计与实现 38
4.1 规则管理模块 38
4.1.1 规则管理模块详细设计 38
4.1.2 规则操作功能实现 39
4.1.3 日志查询功能实现 41
4.1.4 规则过期告警功能实现 41
4.2 规则预处理模块 42

4.2.1 规则预处理模块设计 42
4.2.2 构建检查规则功能实现 44
4.2.3 构建生效范围功能实现 45
4.2.4 构建 Action 功能实现 46
4.3 审核流管理模块 47
4.3.1 审核流管理模块设计 47
4.3.2 审核流操作功能实现 49
4.3.3 审核权转移功能实现 51
4.3.4 审核消息保障功能实现 52
4.4 规则应用模块 53
4.4.1 规则应用模块设计 53
4.4.2 规则判定功能实现 54
4.4.3 缓存更新功能实现 56
4.5 系统测试 57
4.5.1 测试环境 57
4.5.2 测试设计 58
4.5.3 测试结果 61
4.6 本章小结 61
第五章 总结与展望 62
5.1 总结 62
5.2 工作展望 62
参 考 文 献 64

图目录

图 1.1 外卖业务履约过程 1
图 2.1 RABBITMQ 内部结构图 6
图 2.2 饿了么多活架构 10
图 3.1 系统边界 12
图 3.2 规则管理用例图 14
图 3.3 规则预处理用例图 16
图 3.4 审核流程用例图 18
图 3.5 审核权转移流程图 19
图 3.6 业务处理用例图 21
图 3.7 规则判定流程图 22
图 3.8 系统架构图 30
图 3.9 系统功能模块图 31
图 3.10 部署设计 32
图 3.11 实体关系图 33
图 4.1 规则管理类图 38
图 4.2 规则操作时序图 39
图 4.3 新增规则实现 40
图 4.4 规则配置详情页面 40
图 4.5 日志查询功能实现 41
图 4.6 规则过期告警功能实现 41
图 4.7 规则执行表达式 42
图 4.8 规则解析树概括 42
图 4.9 PREDICTDELIVERTIMECONDITION 节点 43
图 4.10 规则预处理类图 43
图 4.11 检查规则实现图 45
图 4.12 构建生效范围实现图 46
图 4.13 构建 ACTION 功能实现图 47
图 4.14 审核流模块类图 48
图 4.15 审核流操作时序图 49
图 4.16 审核流操作功能实现图 50
图 4.17 审核流操作界面 50
图 4.18 审核权转移时序图 51
图 4.19 审核流操作实现 51
图 4.20 审核消息保障时序图 52
图 4.21 审核信息保障实现 52
图 4.22 规则应用模块类图 53
图 4.23 规则应用时序图 55
图 4.24 规则应用功能实现 56
图 4.25 缓存更新功能实现 57

表目录

表 3.1 涉众分析 13
表 3.2 规则操作用例表 15
表 3.3 规则过期告警用例表 15
表 3.4 日志查询用例表 16
表 3.5 构建检查规则用例图 17
表 3.6 构建生效范围用例图 17
表 3.7 构建 ACTION 用例表 18
表 3.8 审核流操作用例表 19
表 3.9 审核权流转用例表 20
表 3.10 审核消息保障用例表 21
表 3.11 规则判定用例表 23
表 3.12 缓存同步用例表 24
表 3.13 检查规则选项表 25
表 3.14 生效范围规则选项表 26
表 3.15 ACTION 规则选项表 27
表 3.16 函数设计表 27
表 3.17 订单消息体表 28
表 3.18 超时消息表 28
表 3.19 运单消息表 29
表 3.20 审核流消息体表 29
表 3.21 RULE 表结构 34
表 3.22 RULE_NOT_HIT 表结构 34
表 3.23 RULE_HIT 表结构 35
表 3.24 RULE_DRAFT 表结构 35
表 3.25 ACTION_EXECUTE 表结构 36
表 3.26 OPERATE_RECORD 表结构 36
表 3.27 OPERATOR 表结构 36
表 4.1 硬件和环境配置 57
表 4.2 软件配置 57
表 4.3 规则操作测试用例 58
表 4.4 日志查询测试用例 58
表 4.5 规则过期告警测试用例 59
表 4.6 审核流操作测试用例 59
表 4.7 审核权转移测试用例 59
表 4.8 审核消息保障测试用例 60
表 4.9 规则预处理功能测试用例 60
表 4.10 规则判定测试用例 60
表 4.11 系统功能测试结果表 61

第一章 引言

1.1 项目背景

论文选题源自于本人实习时参与的“饿了么履约超时管理平台”项目。

履约是指按照约定履行契约。用户在电商平台上下订单，用户就和平台签订了一个契约，平台需要按照契约进行履约，简单来说，用户下单之后平台就需要履约，下单之后的阶段就是履约阶段。

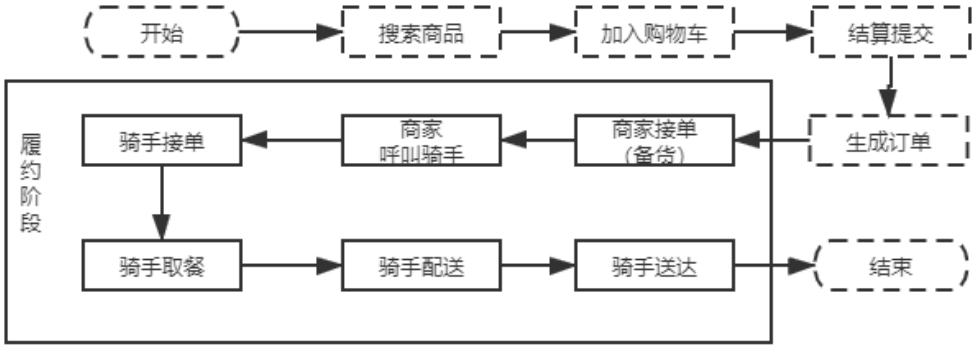

图 1.1 外卖业务履约过程

图 1.1 展示了以外卖业务为代表的履约过程。履约阶段主要业务流程包括商家接单、商家呼叫骑手、骑手接单、骑手取餐、骑手配送、骑手送达等节点。有别于传统电商履约特性，外卖行业，尤其是餐饮外卖行业，对于时间很敏感。一笔餐饮外卖订单一般需要在半小时到一小时之间完成履约，这么短的时间内需要经过六个履约节点，经常会出现因某个节点超时而导致整个订单履约超时，严重影响用户体验。为此，迫切需要采取相应的应对措施，解决或预警可能的履约问题，提升用户体验。

饿了么既有履约场景超时检查分散在各个服务中，例如逆向系统、赔付系统、用户触达系统等，由此产生了诸多问题：

一是工作量巨大，容易出错。每次修改超时规则，都需要各个系统依次同步修改，不仅增加了额外工作量，而且经常发生同步不及时，导致各种意想不到的问题。

二是涉及部门、人员多，环节繁多、流程复杂，效率低下，无法快速响应业

务需求。目前超时规则采用“硬编码”方式，嵌套于各个系统的代码。如果需要修改规则，需要产品经理提紧急需求，开发人员设计技术方案，各方评审方案，开发人员开发，测试人员测试，然后在发布窗口发布。对于开发人员来说，增加了高优先级任务，需要优先完成紧急需求，增加了开发人员工作量；对于业务方来说，需求得不到快速响应，延误了策略实施，用户满意度就会受影响。

三是规则颗粒度控制困难。上述问题，运营人员无法做到对规则的细粒度把控，例如，根据天气配置规则，根据城市配置规则，根据订单来源配置规则等，每次规则变动的需求都会造成牵一发而动全身的影响。

为了解决上述问题，饿了么需要剥离目前分散在各个系统中的履约场景超时检查逻辑，集中检查、规范超时检查逻辑，同时需要配置平台，能够对超时规则以及处理逻辑进行配置，以达到快速响应业务需求的目的，进一步提升用户满意度

1.2 国内外发展现状

在国外，Uber eats[Monty et al.,2018]是 Uber 公司旗下一款送餐应用，目前全球覆盖率最广。Uber eats 曾提出“外卖 30 分钟内送达，否则全款赔偿”口号，但实际使用情况却并非如此。对于超出预期送达时间情况，官方只给出了如下答复：餐厅及其派送员会尽全力在预计送达时间内派送餐点，但有很多因素可能导致送餐延误（例如餐厅比平时更忙、您的订单较大、意外堵车或天气状况），由此可见平台不会主动处理履约超时情景。餐厅长时间找不到配送人员时，平台甚至会自动取消订单。如果用户因配送时间太长而想申请赔偿，平台更偏向于让用户直接联系商家申请赔偿。

在国内，京东物流一直以快速、准时著称，这一切得益于京东履约引擎[周强明,2017]的加持。电商履约中最重要、也最容易发生超时的两个环节为商家发货和物流配送，《京东开发平台规则》对上述两阶段履约超时应对措施均有详细说明。在商家发货阶段，如果商家超过 48 小时未发货，系统就会判定为延时发货，第三方卖家需要赔偿 30%，最高 500 元，自营店需要赔偿 5 元，同时会扣商家积分。在物流配送阶段，如果自营物流超时，用户可以自行联系客服进行索赔，客服一般会赔偿京东豆;如果是第三方物流超，用户只能根据《快递服务行业标准》

[快递服务标准,2008]对物流公司进行索赔，京东平台在此过程中充当调节人员。京东这种履约超时处理方式也是大多数电商平台采用的方式：严控商家发货时出现的超时情景，淡化物流配送时出现的超时情景，原因在于平台能够把控商家发货时间，但无法把控物流配送时间，由于物流在运输过程中存在很多因素影响物流配送时间，即使是能够提供 211 限时、京准达、极速达服务的京东物流，也会存在配送超时情况。

美团外卖是饿了么目前最大竞争对手，也是是目前国内外卖行业领先者。美团外卖履约流程与饿了么履约流程类似，在履约过程中也会存在超时的场景。在应对履约超时方面，美团推出了“准时宝”，“准时宝”是美团外卖的增值服务，用户购买准时宝之后，如果发生订单超时，平台会进行赔付，赔付条例如下：订单实际送达时间晚于预计送达时间 15 分钟，赔偿订单实际支付金额 30%；订单实际送达时间晚于预计送达时间 30 分钟，赔偿订单实际支付金额 70%。除此之外，美团提供人工索赔申请服务，对于没有购买准时宝的用户，如果订单发生了严重超时情况，用户可以自行联系客服申请索赔。除了赔偿之外，美团还支持超时短信安抚，超时取消等应对措施，

外卖行业是一个对时间很敏感的行业，如果采用传统电商中平台被动赔偿的方式应对外卖履约超时情况，必然会导致用户满意度降低，进而影响市场份额，因此平台被动赔偿方式不适用于外卖行业。美团外卖应对履约超时的方式也是国内外卖行业主流应对方式，饿了么也有类似应对措施，“准准达”、超时安抚短信、超时取消等。在履约超时管理方面，美团并未开源相关软件，因此，为了解决 1.1 节中存在的问题，需要开发一款履约超时管理平台。

1.3 本文主要工作

本文梳理了原有系统的业务逻辑以及原有系统之间的依赖关系，分析了原有系统存在的问题，指出开发履约超时管理平台系统的必要性。

分析履约超时管理平台的需求背景、目标用户、相关功能需求及非功能需求。对于功能需求，分析各个功能用例，并通过用例描述表的形式进行表示；对于非功能性需求，给出安全性、稳定性、拓展性需求。提出系统设计方案，包括系统架构设计、部署设计以及数据库的设计。

给出系统各个模块的核心实现。对于规则配置模块，设计并实现了规则操作、日志操作、规则过期告警功能；对于流程管理模块，设计并实现了流程管理、消息保障功能；对于规则预处理模块，设计并实现了构建检查规则、构建范围、构建执行动作功能；对于业务处理模块，设计并实现了规则应用、缓存更新功能。

1.4 本文组织结构

本文组织架构如下：

第一章 引言。介绍项目背景、相关研究与应用现状以及本文主要工作。

第二章 技术综述。介绍了履约超时管理平台使用的相关技术，包括 Spring Boot 框架、RabbitMQ 消息队列、E-Scheduler 任务调度平台、Redis 缓存, 异地多活技术。

第三章 系统分析与设计。确定履约超时管理平台边界，分析功能性需求和非功能性需求。设计系统工作原理，进行了系统总体设计、系统架构设计、数据库设计。

第四章 系统实现。针对第三章中的系统分析，采用类图、顺序图等详细设计主要的功能模块，并给出关键代码，说明具体实现细节，并进行了测试。

第五章 总结与展望。对本文的工作进行了总结，阐述了当前取得的成果，并分析了待改进和研究的地方，对后续研究工作进行展望。

第二章 技术综述

2.1 SpringBoot

SpringBoot 是由 Pivotal 团队在 2013 年开始研发、2014 年 4 月发布第一个版本的全新开源的轻量级框架。它不仅继承了 Spring 框架原有的优秀特性，而且还通过简化配置来进一步简化了 Spring 应用和整个搭建和开发的过程。SpringBoot 框架中有两个非常重要的策略：开箱即用和约定优于配置[Arthur et al.,2005]。开箱即用，是指在开发过程中，通过在 Maven 项目的 pom 文件中添加相关依赖包[Agarwal,2014]，然后使用对应的注解来代替繁琐的 XML 配置文件以管理对象的生命周期，这个特点使得开发人员摆脱了复杂的配置工作以及依赖管理工作，更加专注于业务逻辑。约定优于配置，是一种由 SpringBoot 本身来配置目标结构，由开发者在结构中添加信息的软件设计范式。这一特点虽降低了部分灵活性，增加了 BUG 定位的复杂性，但减少了开发人员需要做出决定的数量，同时减少了大量 XML 配置，并且可以将代码编译、测试、打包等工作自动化。

控制反转（IOC，Inversion of Control）是 SpringBoot 继承自 Spring 的核心设计理念。IOC 将原本由开发人员创建的对象交由容器创建，容器在创建对象时会根据依赖关系将对象中需要的属性注入其中，同时，容器管理着对象的生命周期。IOC 能够最大限度降低对象之间的耦合度，当对象之间的依赖关系发生变化时，只需要修改配置即可[Ginanjar et al.2019]。

面向切面编程（AOP，Aspect Oriented Programming）是 SpringBoot 继承自 Spring 的重要特点[Laddad,2009]，它解决了业务逻辑和系统服务强耦合、代码重复的问题[Elrad et al.2001]。AOP 针对业务处理过程中的切面进行提取，以获取逻辑过程中各个部分之间低耦合的隔离效果[Kiyoshi,2006]。

2.2 RabbitMQ

RabbitMQ 是基于 AMQP 协议开发的一款开源的消息中间件[Sachs et

al.,2013]，AMQP 协议又称为高级消息队列协议（Advanced Message Queuing Protocol）[Albano et al. 2015]，是面向消息中间件提供的开放的应用层协议，其设计目标是对于消息的排序、路由（包括点对点和订阅-发布）、保持可靠性、保证安全性。AMQP 规范了消息传递方和接收方的行为，以使消息在不同的提供商之间实现互操作性，就像 SMTP，HTTP，FTP 等协议可以创建交互系统一样。与先前中间件标准（如 Java 消息服务）不同的是，JMS 在特定的 API 接口层面和实现行为上进行了统一，而高级消息队列协议则关注于各种消息如何以字节流的形式进行传递[Sachs et al.,2009]。因此，使用了符合协议实现的任意应用程序之间可以保持对消息的创建、传递。RabbitMQ 拥有完善的消息确认机制[徐震等,2018]，可以通过 AMQP、Confirm、ACK 等机制来保证消息可靠。

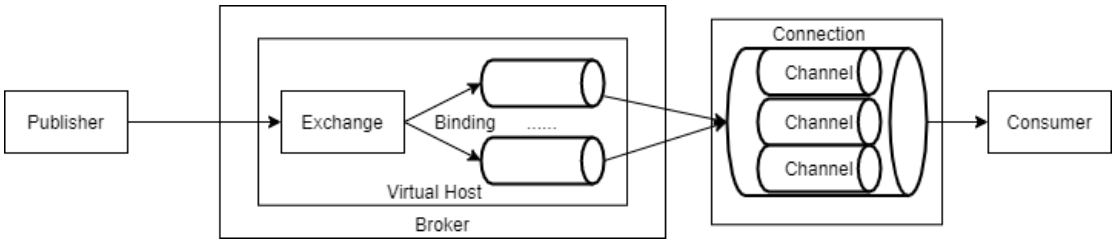

图 2.1 RabbitMQ 内部结构图

图 2.1 为 RabbitMQ 内部结构图。RabbitMQ 由以下的基本组件组成：

1. Message。具体的消息，包含消息头（即附属的配置信息）和消息体（即消息的实体内容），由发布者将消息推送到 Exchange，由消费者从 Queue 中获取。
2. Publisher。消息的生产者，负责将消息推送到 Exchange 中。
3. Exchange。交换器，用来接受生产者发送的消息，并根据消息中的属性将其路由到不同的消息队列中去。
4. Binding。绑定，描述 Exchange 和 Queue 之间的关系，当 Exchange 和 Queue 绑定之后，可以决定 Exchange 中的哪些消息发送到与之绑定的 Queue 中。
5. Queue。消息队列，用于保存消息直到发送给消费者，一个消息可以发送给多个消息队列，消息会一直存放在消息队列中，直到消费者消费该消息或者消息过期。
6. Channel。信道（通道），客户端、服务端都会通过信道和消息队列建立

连接，然后进行发布消息、订阅队列等操作。

7. Consumer。消费者，从消息队列中获取消息，消费消息。

Exchange 负责消息的分发，消息的分发主要有如下的几种策略：

1. Direct 策略。如果消息中的路由键（routing key）和 Binding 中的（绑定键）binding key 一致，交换机才会将数据发送到对应的消息队列中。
2. Topic 策略。Topic 策略是 Direct 策略的升级版本，通过 routing key 和 binding key 的模式匹配方式来分发消息。
3. Fanout 策略。广播策略，忽略 routing key 和 binding key，将消息分发到与 exchange 绑定的所有 queue 上。

RabbitMQ 支持消息、消息队列的持久化，支持消息确认机制，支持事务机制，以此来保障消息的一致性。同时 RabbitMQ 还具备高可用，可扩展，支持集群部署、异地部署等特点[Rostanski et al.,2014]。

2.3 E-Scheduler

E-Scheduler 是饿了么自研的任务调度平台，主要应用场景为：定时任务调度、大数据任务分批处理、异步执行耗时任务等。E-Scheduler 内部通过工作流方式实现，工作流本质是一个状态机[Sturm et al.,2019]，由多个状态组成。状态机可以从一个状态扭转到另一个状态。状态机从一个起始状态开始执行，到一个终结状态结束，状态之间以 JSON 传递数据。状态分为多种不同的类型，不同的类型的状态有不同的功能，类型节点如下：

1. Success：工作流的出口，表示当前工作流执行成功
2. Fail：工作流的出口，表示当前工作流执行失败
3. Pass：过度状态，可以对上个状态的输出做某些处理，并将处理结果流转到下一个状态
4. Task：工作流的远程调用状态，访问一个远程接口，获取结果
5. Wait：等待状态，是当前工作流等待一段时间或等待到某个固定的时间点
6. Choice：判断状态，提供逻辑判断操作

E-Scheduler 由三个部分组成：控制台、触发器和工作流引擎。控制台负责管理工作流的元数据、触发器以及一些配置信息；触发器是一个运行在 Docker 中的微服务，负责定时触发和监听队列触发工作流执行；工作流引擎是一个运行在 Docker 中的微服务，是整个工作流执行的核心，工作流引擎本质是通过输入数据驱动状态机执行来实现业务逻辑。

2.4 Redis

Redis 是一个高效的 key-value 存储系统[Kamal et al.,2019]，与传统关系型数据库不同的是，Redis 数据存储在内存中，具有高性能、高并发的特点。Redis 设计了一些常用的数据结构[Atzeni et al.,2020]，具体如下：

1. String 为字符串类型，可以用做 key，也可以用作 value 的类型，是一种常用的数据类型，其底层通过 Redis 自己构建的一种名叫 Simple Dynamic String(SDS)的数据结构来实现。
2. Hash 是一个 String 类型的 field 和 value 的映射表，即 hash 是由 key、field、value 这三个关键属性组成，一个 key 可以对应多个 field，一个 field 只可以对应一个 value，这样的对应关系很适合存储对象，其底层实现在数组和链表的基础上进行一些 rehash 操作进行优化。
3. List 名为链表，是 Redis 中最为重要的数据结构之一，Redis 中的 List 是一个双向链表，即可以在链表的两头进行插入、删除等操作，由于其底层设计的缘故，可以直接获链表的头节点、尾节点、链表的长度。底层在双向链表的基础上扩展了头节点、尾节点、元素个数等属性实现。
4. Set 是一种集合，不可以存放重复元素，除此之外，Set 还能提供集合的运算，包括交集、并集、差集等。
5. Sorted set，与 Set 不同的是，Sorted 增加了一个权重参数 score，使得集合中的元素能够按照 score 进行排序。其底层通过跳表的方式实现。

Redis 能够做到速度快的原因有如下几点：Redis 是一种存储在内存中的数据库[Lerner,2010]，相较于硬盘访问速度，内存访问速度更快；Redis 的数据结构更简，数据的操作也简单[Paksula,2010]；Redis 采用的是单线程 I/O 多路复用模型，I/O 多路复用是指一个线程可以处理多个 TCP 连接，具体通过调用系

统底层函数 select、poll 和 epoll 的方式实现[凌质亿,2013]；由于是单线程的缘故，并不存在并发竞争的问题。由于上述的原因，Redis 的速度很快。Redis 能持久化缓存数据，通过 AOF 和 RDB 的方式[Xu et al., 2014]，将 Redis 宕机损失降到最低。

为了实现高可用，Redis 一般使用主从结构部署[Ji et al.,2014]，即一主多从，主节点负责写数据，并将写数据同步到从节点，从节点负责读数据。当主节点宕机时，需要一种机制能够自动切换主节点，Redis 哨兵（Sentinel）模式就是一种能自动切换的机制[Xin et al.,2018]。哨兵是 Redis 中的一个独立进程，其原理是哨兵通过发送命令，等待 Redis 服务器响应，从而监控运行的多个 Redis 实例。当哨兵监测到 master 宕机，就会自动将 slave 切换成 master，然后通过发布订阅模式，通知其他的从服务器修改配置文件，让他们切换主机。哨兵一般会存在多个，目的是为了防止单一哨兵出现单点故障的问题。[许艺蓝,2018]假设主服务器宕机，哨兵 1 先检测到这个结果，系统并不会马上进行 failover 过程，仅仅是哨兵 1 主观的认为主服务器不可用，这个现象称为主观下线。当后面的哨兵也检测到主服务器不可用，并且数量达到一定值时，那么哨兵之间就会进行一次投票，投票的结果由一个哨兵发起，进行 failover 操作。切换成功后，就会通过发布订阅模式，让各个哨兵把自己监控的从服务器实现切换主机，这个过程称为客观下线。

2.5 异地多活技术

异地多活是指在不同城市建立独立的数据中心[Testa,2004]。传统数据中心采用“两地三中心”模式，即生产中心、同城备份中心、异地灾备中心，多活是指中心之间没有主备关心，每一个中心都有生产，互为灾备[Naseh et al.,2010]。

图 2.2 为饿了么多活架构，饿了么目前拥有两个机房，张北机房和万国机房，每个机房称为 Ezone，每个机房均有全量业务和全量数据。为了避免跨机房调用而产生的高延迟，用户从下单到配送的整个服务流程需要在同一个机房内进行，这就需要对服务进行分区。饿了么采用地理作为划分基础，把地理位置上接近的用户、商家、骑手划分到同一个 Ezone 中，同时采用商户 ID、订单 ID 等方式辅助修正。API Router 就负责将流量中的分取信息转换成对应的 Shard ID, 根据

Shard ID 从 GZS 中获取 Ezone，再将其路由到具体 Ezone。GZS 全称为 Global Zone Service（全局状态协调器），维护着整个多活的路由表。

用户业务流程只会在一个机房内完成，但多活要求每个机房内的数据均为全量数据，这就要求数据能够在机房之间同步。饿了么目前主要实现了 MySQL 数据同步，MySQL 数据量最大，每个 Ezone 产生的数据都要通过 DRC（数据复制中心，Data Replication Center[Kanagavelu et al.,2013]）复制到其他 Ezone。DRC 负责 MySQL 数据实时双向复制，保证跨机房延时在 1s 内。DRC 除了复制数据之外，还可以对外界提供数据变更通知，每个机房都有名为 DRC 的消息队列，张北机房监听万国机房 DRC 消息队列，万国机房监听张北机房 DRC 消息队列，机房中的服务集群可以根据 DRC 消息做相应操作，例如缓存更新。

DRC 消息同步属于弱一致性，即最终一致性，无法满足数据有强一致性[Kraska et al.,2013]需求的应用，因此饿了么多活提供了强一致性方案（Global Zone），Global Zone 是一种跨机房的读写分离机制，所有的写操作均被定向到一个 Master 机房进行，以保证强一致性，而读操作可以在多个机房进行。这一切通过数据访问层（DAL）实现。

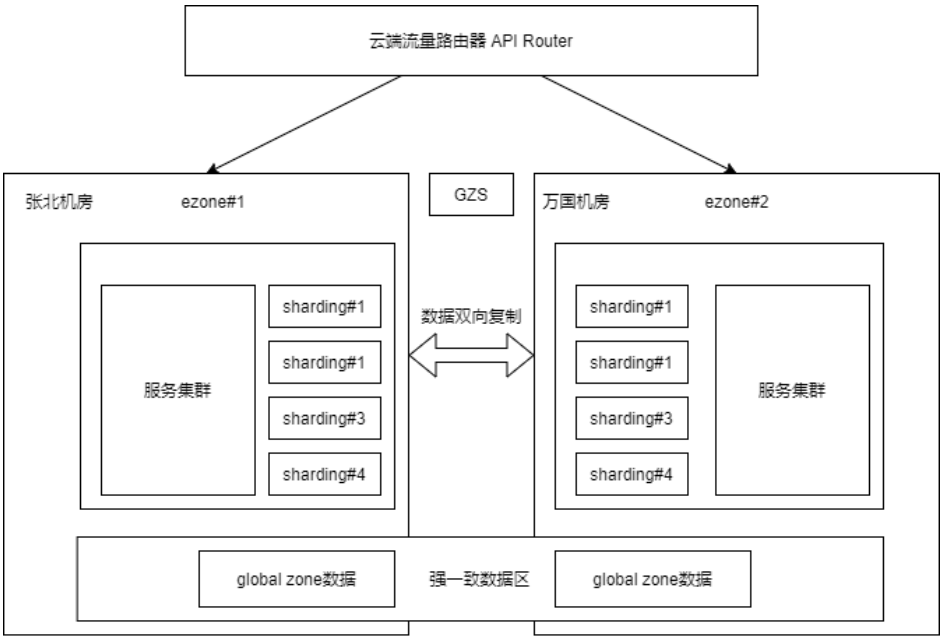

图 2.2 饿了么多活架构

2.6 本章小结

本章主要介绍了饿了么履约超时管理系统所使用到的相关技术。系统采用 SpringBoot 搭建项目的基础框架；使用 Redis 缓存来加速数据的访问，以及减轻数据库的压力；使用 RabbitMQ 降低系统之间的耦合，系统可以通过发送消息给 RabbitMQ 来实现系统之间的交互；使用 E-Scheduler 来实现定时任务的调用。

第三章 系统分析与设计

3.1 系统边界

履约超时管理平台属于履约业务中的一个相对独立的平台，负责制定履约超时规则和应用履约超时规则。

系统边界如图 3.1 所示，描述了本系统和交易模块内部系统以及非交易模块系统之间的交互以及依赖关系。订单与运单系统为消息的发送方，履约超时管理平台依赖于订单系统和运单系统发送的消息，因此需要监听这两个系统的消息队列。延时任务中心提供创建延时任务以及任务到期回调的操作，由于规则存在延时判定的情况，因此依赖延时任务中心；审核流平台提供了审核能力，履约超时管理平台有审核的需求，因此依赖审核流平台。赔付系统提供赔付支持，触达中心提供用户触达能力，履约超时管理平台对于命中规则的订单、运单，会有相应应对策略，赔付和触达就是其中的两种，因此依赖这两个系统。

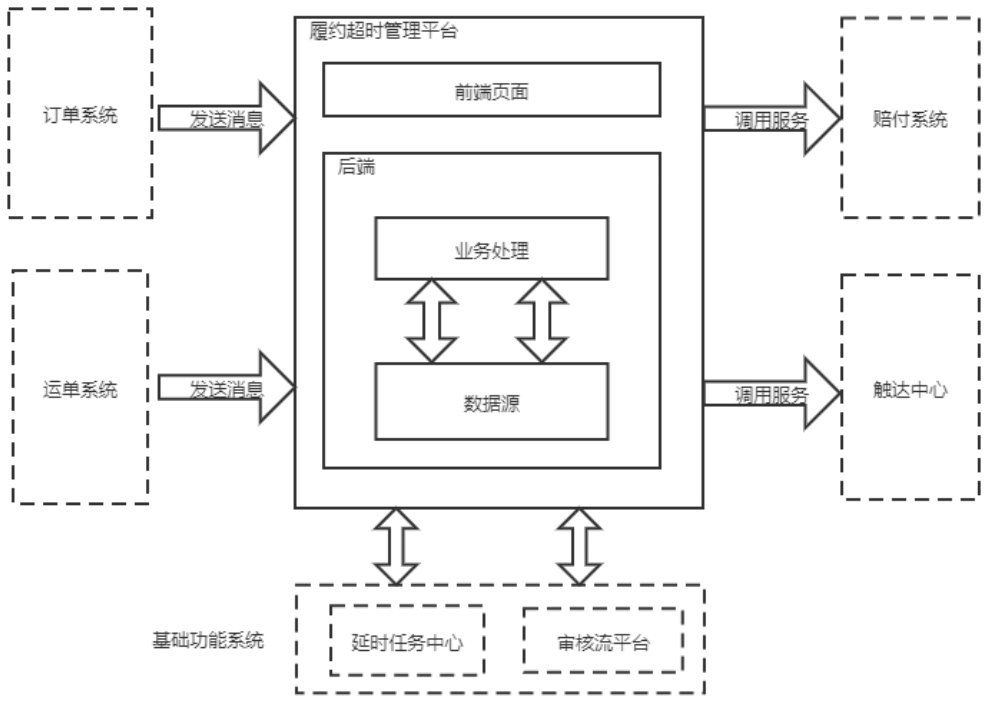

图 3.1 系统边界

3.2 业务描述

根据图 1.1 描述的外卖业务履约过程，涉及履约超时管理平台的业务描述如下：

规则操作：用户可以在履约超时管理平台上规则的编辑，并将编辑完成的规则提交审核。同时用户可以查看规则命中日志。

审核管理：新建规则需要经过层层审核，以确保规则的正确性，避免对公司造成资损。同时用户可以查看审核进展。

规则预处理：用户提交的规则数据需要进行预处理，以达到可执行标准。

规则应用：平台可以检测订单目前状态是否存在超时，对于超时订单，平台可以根据规则进行后续处理。

表 3.1 为涉众分析，涉众分为运营人员、开发人员、延时任务中心、审核平台。

表 3.1 涉众分析

序号	涉众	对系统的期望
1	运营人员	1. 能方便进行线上配置规则 2. 能线上管理审核流 3. 能查看规则命中日志
2	开发人员	1. 能查看规则命中日志
3	延时任务中心	1. 能够调用开放接口，回调指定任务
4	审核平台	1. 能够调用开放接口，转移审核权

基于 3.1 节中系统的系统边界和结合履约超时管理平台的业务场景和涉众分析，系统需要满足以下的需求：

1. 提供规则管理能力：运营人员在前端页面上能够对规则进行新增、修改、删除、查询等操作，并且能够查看规则命中和规则未命中的情况。
2. 提供规则预处理能力：将前端配置好的规则转换成系统能够执行的规则。
3. 提供审核管理能力：运营人员能够管理规则审核，包括查找、终止规则审核流，系统需要提供扭转审核流节点的能力，同时需要提供防止消息丢失的保障。
4. 提供业务处理能力：系统需要监听订单、运单消息，消费这些消息的同时需要与规则库中的规则进行匹配，并且对匹配后的结果进行下一步的处理。

3.3 需求分析

3.3.1 规则管理功能

规则管理用例图如图 3.2 所示，规则管理功能包含了规则管理、规则过期告警、日志查询三个子功能。

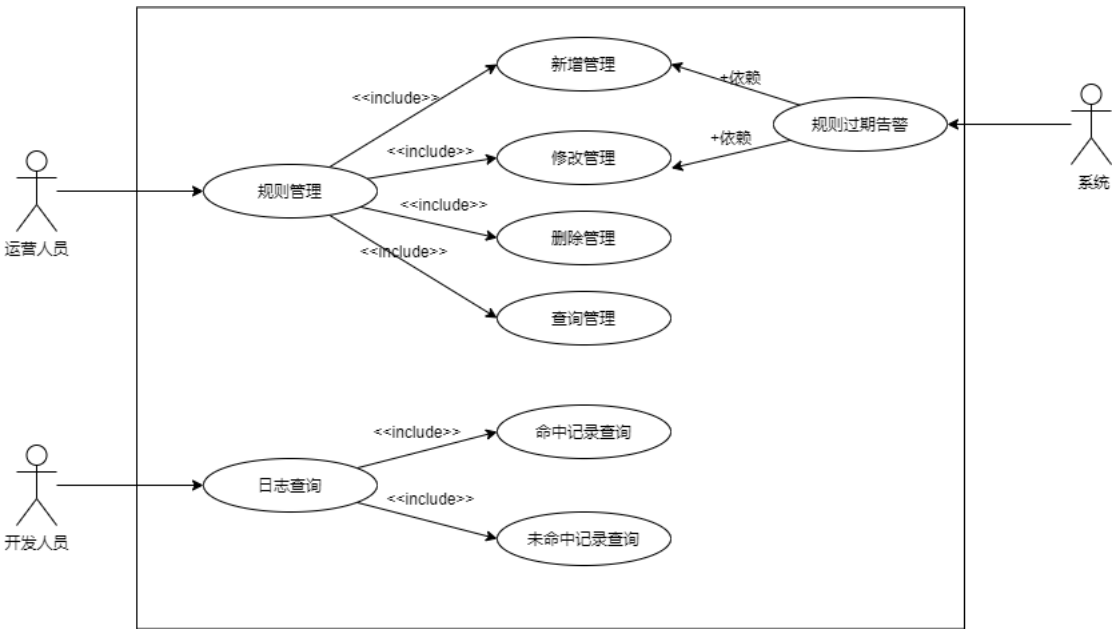

图 3.2 规则管理用例图

由于主要使用用户为运营人员，需要一个图形化界面用于进行规则的增、删、改、查，因此需要规则管理子功能。在规则管理子功能中，用户可以对规则进行新增、删除、修改、查询的操作。对于新建和修改操作，用户点击新建按钮之后，就会进入创建规则页面，页面上会填写默认选项，而用户点击已有规则旁的修改按钮之后，也会进入创建规则页面，页面会填充原有选项，之后用户可以根据需求选择相应的选项，选择结束之后需要点击提交审核来完成操作。对于删除操作，用户点击删除规则按钮之后，界面会弹出警告框，用户可以点击确认或取消，分别对应着删除和取消操作。对于查询操作，用户需要根据可选条件进行筛选。操作规则的用例表述如表 3.2 所示。

表 3.2 规则操作用例表

ID	UC1	名称	操作规则
描述	用户在前端配置页面上对规则进行新增、删除、修改、查看等操作		
参与者	运营、开发		
前置条件	用户已经通过权限认证		
后置条件	1. 新增、修改对数据库中的规则产生影响 2. 新增、修改会触发创建规则过期告警任务、触发审核流		
优先级	中		
正常流程	1.1 点击新建规则 1.2 根据页面中的选项进行配置 1.3 点击提交审核 2.1 点击修改规则 2.2 根据页面中的选项进行配置 2.3 点击提交审核 3.1 点击删除规则，点击告警框中的确认 4.2 选择查询条件，点击查询，返回查询结果		
拓展流程	4a. 提交审核的规则已经存在 1. 提醒规则已存在 2. 保留原先已配置的参数		

用户在创建或修改规则的时候，需要设置规则过期时间，当规则临近过期时，用户需要做相应的措施，因此需要规则过期告警子功能，来提醒用户规则就快过期，提前做好准备。定时任务的创建是在用户修改或新增过期时间之后，即用户在修改、新建规则时，修改了过期时间。考虑到开发成本较高，且公司有延时任务平台，因此可以借助延时任务平台完成定时发送告警邮件的操作。规则过期告警的用例描述详见表 3.3。

表 3.3 规则过期告警用例表

ID	UC2	名称	规则过期告警
描述	创建规则超时告警任务，定时发送告警邮件		
参与者	平台		
前置条件	用户修改/创建规则		
后置条件	延时任务中心会存放告警定时任务，并在指定时间后发送		
优先级	中		
正常流程	1. 用户修改/创建规则。 2. 用户点击提交审核 3. 平台提取修改/创建的规则中的过期时间 4. 平台根据过期时间在延时任务中心创建定时任务		
拓展流程	4a. 平台创建延时任务失败 1. 配置重试机制，进行重试调用 2. 重试调用失败提醒前端创建失败		

在日常的运营中，运营人员想要知道某条规则、某些订单、某个时间段规则命中情况，因此需要一个规则命中日志查询功能。除此之外，运营人员想更为直观的了解规则命中率排名、命中时间段等信息，因此需要将日志信息以大盘的方式在前端进行展示。开发人员需要查询某一笔订单对应的命中记录，来解决软件中可能存在的不足。日志查询用例描述如表 3.4 所示。

表 3. 4 日志查询用例表

ID	UC3	名称	查询日志
描述	运营和开发人员通过平台查看规则命中日志		
参与者	运营人员、开发人员		
前置条件	用户已经通过权限认证		
后置条件	无		
优先级	中		
正常流程	1. 用户进入日志查询页面 2. 点击查询条件设置 3. 根据需求设置查询条件 4. 点击查询 5. 系统返回查询结果		
拓展流程	无		

3.3.2 规则预处理功能

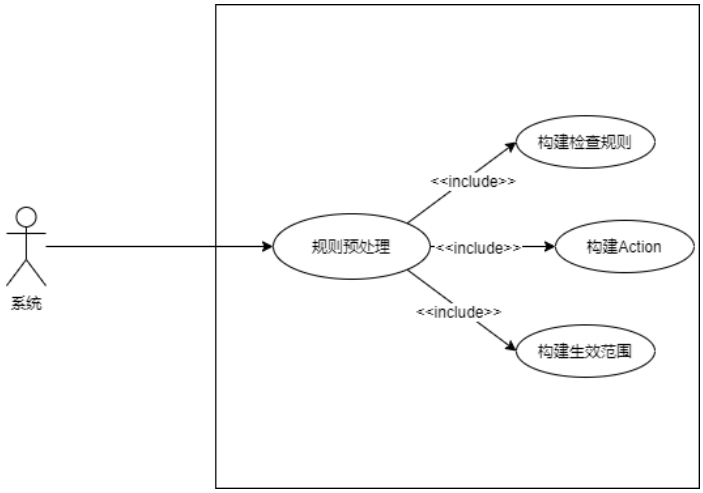

图 3.3 规则预处理用例图

图 3.3 为规则预处理用例图，规则预处理功能分为构建检查规则、构建生效范围、构建 Action。

规则预处理功能负责处理前端传送的规则数据。前端传送的规则数据保存在 VO 对象中，后端接收到数据后，需要按照指定组织关系将规则数据组织起来。

规则之间的组织关系是固定的，具体体现在前端配置页面上，运营人员无法改变这种关系，只能在固定的组织关系中配置规则。若有新判断条件、新应对策略出现，组织关系才有可能发生变化，但这种需求非常少。

构建检查规则子功能负责将用户配置的检查规则按照组织关系进行树化，检查规则包括订单类型、目前所处阶段、是否到达预定时间、是否发生指定事件等。构建检查规则用例描述如表 3.5 所示。

表 3. 5 构建检查规则用例图

ID	UC4	名称	构建检查规则
描述	根据用户配置的检查规则构建能够真实执行的规则		
参与者	系统		
前置条件	用户创建/修改规则		
后置条件	生成检查规则执行树，并存入数据库中，		
优先级	中		
正常流程	1. 用户提交规则创建/修改 2. 后端根据检查规则数据构建检查执行树 3. 将构建好的树存入数据库 4. 触发审核		
拓展流程	无		

构建生效范围子功能与构建检查子功能类似，都是对规则进行树化，不同的是该功能需要和检查规则以及触发行为一起使用，检查规则可以设置生效范围，Action 也可以设置生效范围，这样设计能够更为细粒度地把控规则。构建生效范围用例描述如表 3.6 所示。

表 3.6 构建生效范围用例图

ID	UC5	名称	构建生效范围
描述	根据用户配置的生效范围构建生效范围执行树		
参与者	系统		
前置条件	用户创建/修改规则		
后置条件	生成生效范围执行树，并作为子树传送给其他执行树		
优先级	中		
正常流程	1. 用户提交规则创建/修改 2. 检查规则或触发行为获取生效范围执行树 3. 后端根据生效范围数据构建范围执行树 4. 返回生效范围执行树		
拓展流程	无		

构建 Action 子模块负责将用户配置的 Action 以指定的方式进行树化，与其他两者不同的是，Action 并没有先后顺序，Action 甚至可以并行执行，但是每

个 Action 都有具体的执行范围，因此 Action 和执行范围组合成执行单元。构建 Action 用例描述如表 3.7 所示。

表 3.7 构建 Action 用例表

ID	UC6	名称	构建 Action
描述	根据用户配置的触发行为构建 Action 执行树		
参与者	系统		
前置条件	用户创建/修改规则		
后置条件	生成 Action 执行树，存入数据库		
优先级	中		
正常流程	1. 用户提交规则创建/修改 2. 后端获取配置的 Action 3. 后端根据获取每个 Action 的生效范围执行树 4. 构建 Action 执行树		
拓展流程	无		

3.3.3 审核流程管理功能

图 3.4 为审核流程管理功能的用例图。审核流程管理功能包括审核流操作、审核权扭转、消息保障三个子功能。

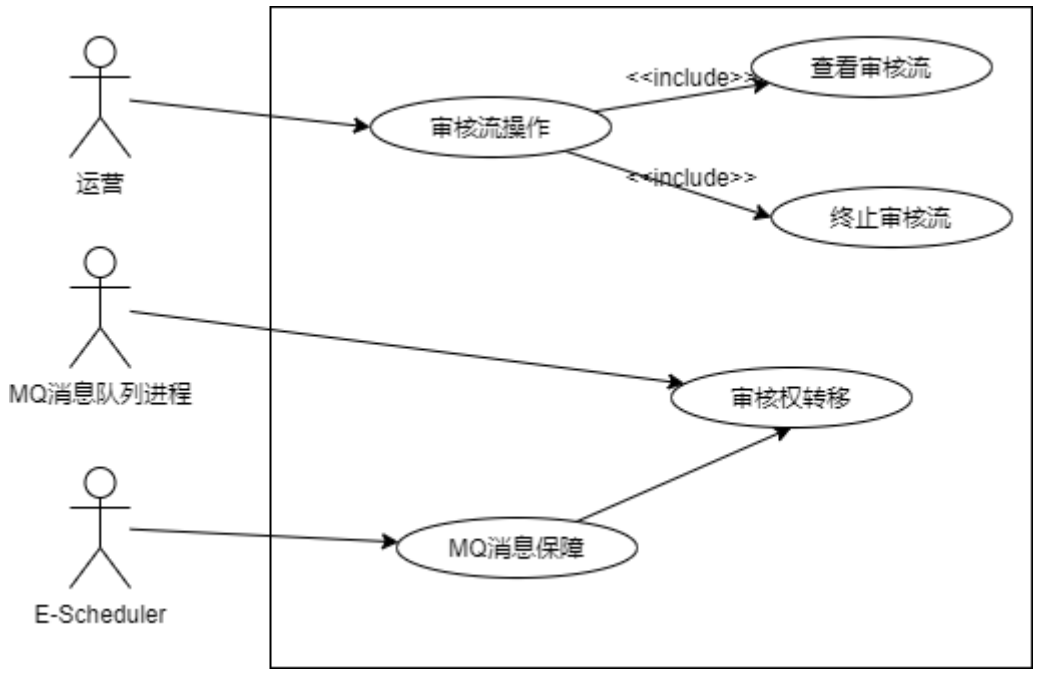

图 3.4 审核流程用例图

运营人员点击提交审核之后，需要了解规则审核的进度，同时需要终止审核配置错误的规则，因此需要一个审核流操作子功能，用来查询和终止审核流。在审核流操作子功能中，用户可以指定具体条件进行查询，条件包括：审核通过、

审核未通过、审核进行中、开始日期等。用户在配置规则的时候，可能存在配错的情况，如果放任其审核，会浪费人力资源，因此用户可以点击终止审核流程，来提前终止审核。审核流操作用例描述如表 3.8 所示。

表 3.8 审核流操作用例表

ID	UC7	名称	审核流操作
描述	对审核流进行操作，包括查看、终止审核流		
参与者	运营		
前置条件	运营配置好规则，并提交审核		
后置条件	不需要的审核流程会被终止		
优先级	中		
正常流程	1.1 用户进入审核流界面 1.2 根据需求选择查询条件 1.3 点击查询，系统返回查询结果 2.1 用户根据需求点击终止审核流 2.2 前端页面弹出警告框 2.2 选择确定或取消		
拓展流程	2.2a 在终止审核流的过程中，审核流可能正好审批通过 1.提醒用户审核流已经审核通过		

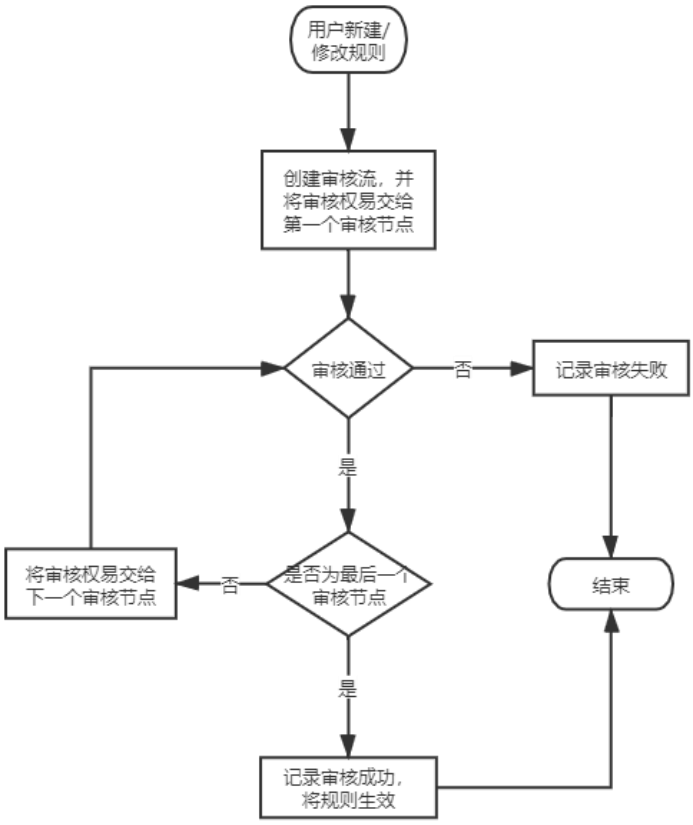

图 3.5 审核权转移流程图

审核权流转子功能负责对审核流程当中的审核权进行流转，该功能的实现需要依赖于审核流平台，审核流平台是公司已有的一个基础平台，提供流程审核的基础能力。目前审核流平台只提供审核节点的创建，但是审核节点的具体审核人员需要系统接入方自行设定，因此，系统接入审核流平台之后，需要设置每个节点的审核人，同时监听审核流平台的消息。图 3.5 审核权转移流程图，从图中可知，如果消息为审核通过，需要判断当前节点是否为最后一个节点，如果是最后一个节点，记录审核通过，否则将审核权移交给下一个审核节点；如果消息为审核不通过，记录审核失败。审核权流转用例描述如表 3.9 所示。

表 3.9 审核权流转用例表

ID	UC8	名称	审核权流转
描述	将审核权进行流转		
参与者	系统		
前置条件	系统在审核流平台创建了审核流		
后置条件	1. 规则审核通过，被纳入规则库中 2. 规则审核失败 3. 当前节点审核通过，审核权流转到下一个节点		
优先级	中		
正常流程	1. 用户点击提交审核 2. 系统在审核流平台中创建审核流 3. 审核人员在审核平台上进行审核 4. 系统接受到审核信息，根据审核信息做相应的处理		
拓展流程	无		

消息保障子功能负责解决消息丢失情况。消息丢失并非一定要处理，如果丢失消息对业务运行几乎没有影响，可以考虑不处理，但在本功能中，消息丢失如果不处理，将会导致审核流停摆，进一步影响规则投入使用的时间，因此本功能中需要对消息丢失进行处理。消息丢失有如下几种可能原因：生产者丢失数据、MQ Server 丢失数据、消费者丢失数据。如果是生产者丢失数据，那么可以通过在生产者端开启 confirm 模式解决问题；如果是 MQ 失数据，例如 MQ Server 突然宕机，可以通过 MQ 持久化方式解决问题；如果是消费方丢失数据，那么可以启用 ACK 机制解决问题，只有消息处理完之后给 MQ 发送 ACK 消息进行确认。但在公司使用场景中，作为消费者，系统并不能保证生产者和 MQ Server 不丢失数据，即使生产者和 MQ 做了消息丢失兜底，但是在消费者视角，依旧需要将其视为不可靠。因此，在消费者这端，除了要保证消费者不丢失数据外，还要

为 MQ 丢失数据情况做兜底。由于审核流平台将审核结果记录在本地数据库中，而履约超时管理平台也会记录审核状态，因此在通过 E-Scheduler 上配置定时任务，定时将履约超时管理平台的数据与审核平台同步，并将审核权移交到下一个审核节点，就能解决问题。数据同步以及审核权转移功能需要在履约超时管理平台实现。审核消息保障用例描述如表 3.10 所示。

表 3.10 审核消息保障用例表

ID	UC9	名称	审核消息保障
描述	保证消息丢失之后审核流程能够扭转下去		
参与者	履约超时管理平台、E-Scheduler 系统		
前置条件	审核消息丢失		
后置条件	1. 本地审核状态与审核平台中审核状态一致 2. 终止的审核流程能够继续进行		
优先级	中		
正常流程	1. 审核平台发送消息给 MQ 2. E-Scheduler 上执行定时任务，定时调用超时管理平台提供的接口 3. 履约超时管理平台提供纠正接口，将本地审核信息与审核平台信息进行同步，同时审核权流转到下一个节点。		
拓展流程	无		

3.3.4 规则应用功能

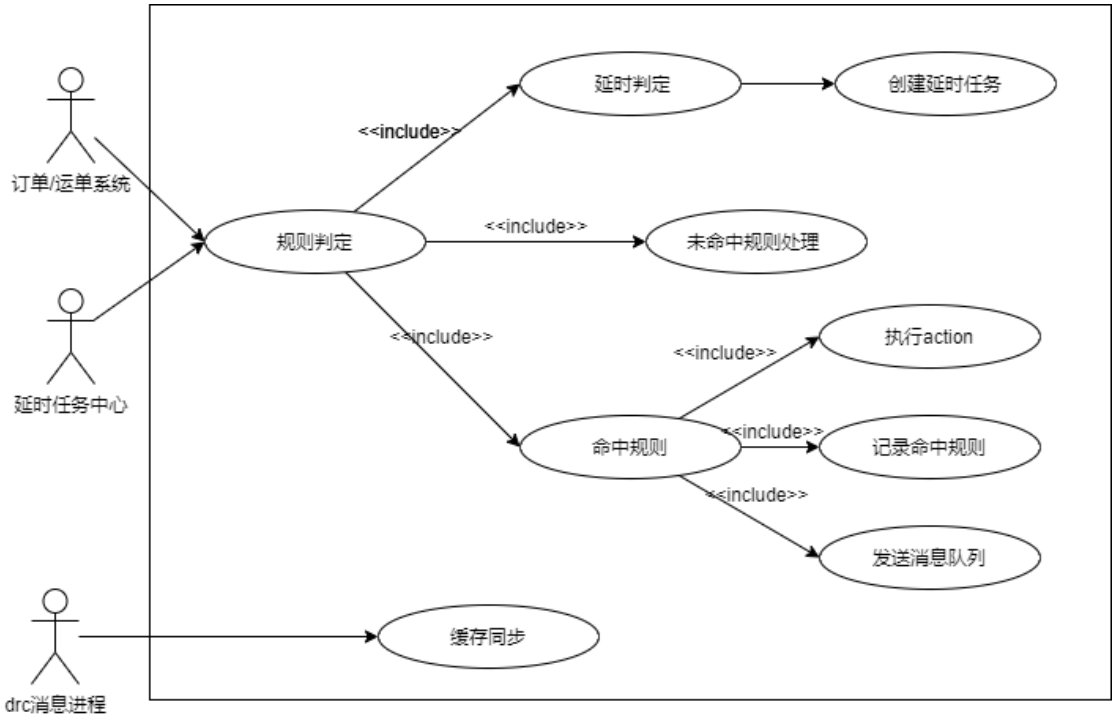

图 3.6 业务处理用例图

图 3.6 为规则应用功能的用例图，业务处理功能分为规则判定和缓存同步两个子功能。

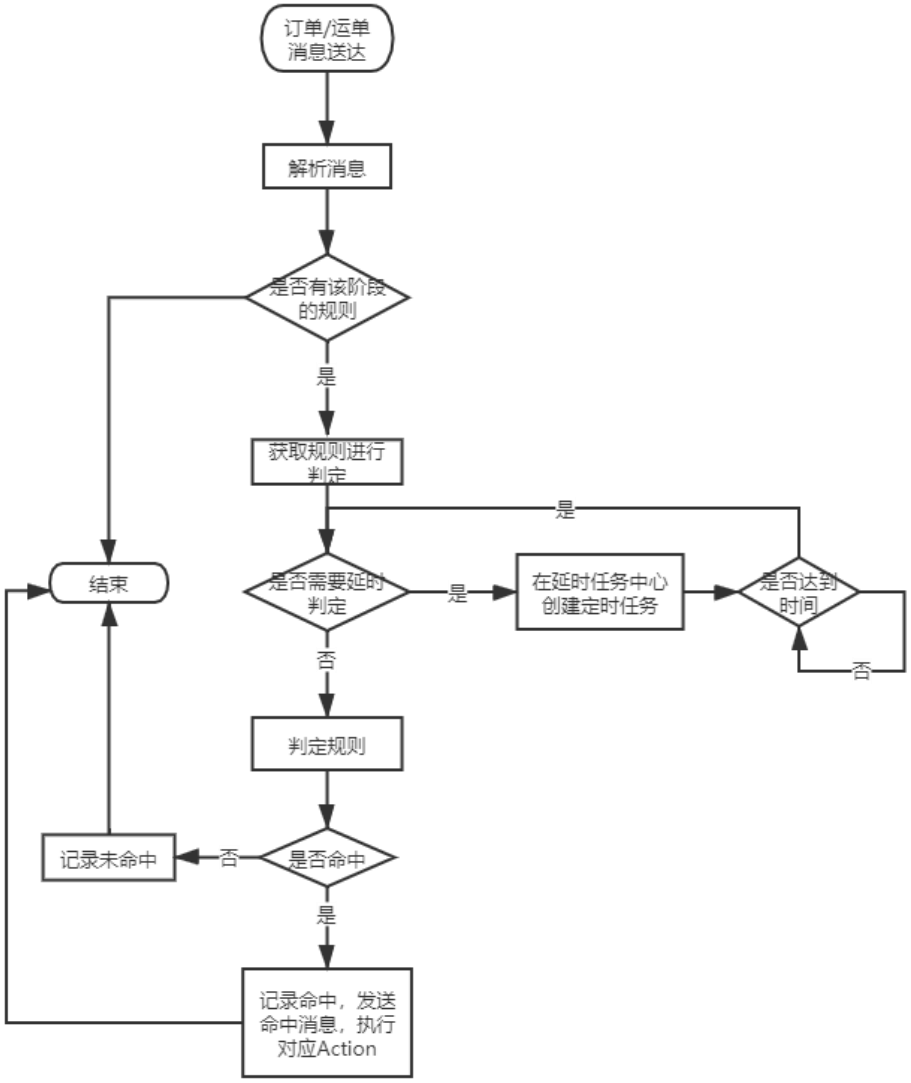

图 3.7 规则判定流程图

图 3.7 为规则判定流程图。规则判定有两个入口，消息监听入口和延时任务回调入口。每一笔订单都需要经过消息监听入口，消息监听入口流程如下：平台监听订单、运单消息队列，监听到消息之后对消息进行解析，并从消息中获取当前订单所处阶段，根据阶段获取对应规则，进而进行匹配。在匹配过程中会出现规则需要延时匹配的情况，这类规则需要走延时任务回调入口，而不需要延时判定的规则能够立即得出匹配结果，如果订单当前阶段命中超时规则，平台会记录命中日志，并根据规则中配置的 Action 执行后续操作。延时任务回调入口如下：当延时任务在延时任务中心到期时，延时任务中心会回调创建延时任务时指定的

接口，在回调接口中，平台会继续进行匹配，具体逻辑与消息监听入口逻辑相同。

规则判定的用例描述如表 3.11 所示。

表 3.11 规则判定用例表

ID	UC10	名称	规则判定
描述	判定运单/订单是否命中超时规则		
参与者	应用		
前置条件	订单/运单系统发送订单消息		
后置条件	1. 得到判定结果，并做相应的处理 2. 未得到判定结果，延时任务中心创建延时执行任务		
优先级	中		
正常流程	1. 系统监听到订单/运单系统发送的消息 2. 根据消息中的状态获取对应的规则 3. 依次执行对应的规则 4. 根据执行结果做相应的处理		
拓展流程	3a.规则不是立即判定，而是延时判定 1. 在延时任务中心创建定时任务，延时执行规则 3b.延时任务回调 1. 在已执行的基础上继续执行 4a.规则执行成功 1. 执行对应的 action 2. 记录规则执行成功 4b.规则执行失败 1. 记录规则执行失败		

缓存同步子功能负责同步部署在不同机房的缓存。由于饿了么的业务已经发展到一个数据中心支撑不住的阶段，因此饿了么使用异地多机房策略来应对业务的增长。异地机房必然存在数据同步的问题，目前饿了么多活架构通过 DRC（Data Replication Center）进行数据库之间消息的更新，以 A 机房和 B 机房为例，用户在饿了么下一笔订单，根据订单号、商户地址、用户地址等消息，多活框架将请求路由到 A 机房，A 机房会有相关数据产生，此时 DRC 就会将 A 机房产生的数据同步到 B 机房，A 机房和 B 机房数据库中的数据达到弱一致性、即最终一致性。但 A 机房中缓存和 B 机房中的缓存并未同步，DRC 提供了同步数据库的同时发送同步消息的功能，A 机房和 B 机房均有一个 DRC 消息队列，且 A 机房监听 B 机房的 DRC 消息队列，B 机房监听 A 机房的消息队列，当 A 机房数据库中数据产生变动时，B 机房会监听到 DRC 消息队列中的 A 机房同步消息，B 机房可以根据此消息更新 B 机房中的缓存数据，以保证缓存一致性。

表 3.12 缓存同步用例表

ID	UC11	名称	缓存同步
描述	同步两地机房的缓存		
参与者	系统		
前置条件	两个机房中的任意一个机房数据有变动，并向对方机房消息队列发送了 drc 消息		
后置条件	两地机房缓存达成一致		
优先级	中		
正常流程	1. 机房监听到 drc 消息 2. 提取 drc 消息中更新的数据 3. 比较缓存中是否有对应的数据		
拓展流程	3a. 缓存中存在对应的数据 1. 将缓存中的数据失效 3b. 缓存中没有对应的数据 1. 不做处理		

3.3.5 非功能需求

（1）安全性

系统运行时需要与多个外部组件进行交互，交互数据的安全性需要得到保障，因此需要对交互数据进行严格的校验和筛选。

（2）稳定性

系统需要能够稳定运行，即使发生服务器故障，也要能够正常运行。系统需要采用分布式部署，同时需要采用异地机房部署。

（3）可拓展行

系统需要能够应对潜在需求变更，在不影响当前业务情况下，增加新的 Action 和新的判断条件。

3.4 系统总体设计

3.4.1 规则设计

运营人员在进行规则配置的时候并非是随意的配置，而是需要在指定的规则选项集合下进行配置。

规则主要分为三大部分，检查规则、生效范围规则和 Action 规则。

表 3.13 为检查规则选项表，表中每一个规则选项对应着一个最小执行单元，检查规则由这些最小执行单元组成。每条判定规则都需要判断订单的类型、订单当前所处的业务阶段、在开始阶段之后或之后的指定时间内、是否发生指定事件。规则的形式相对固定，因此可以通过树的结构将上述最小执行单元进行组织。而执行单元内部是可以变动的，其变动范围如上表中所示，用户可以在前端页面上进行配置，以达到动态修改规则的目的。

表 3.13 检查规则选项表

规则名	可选项	注解
订单类型在指定范围	及时单	订单需要立即处理
	预定单	订单需要在预定时间处理
	全部	订单类型可以是即时单也可以是预订单
主场景在指定范围	提交订单	订单状态：2，订单新建
	实际用户 T	订单状态：6，订单完结
	用户付款	订单状态：3，运单状态：0
	商家接单	订单状态：4，运单状态：1
	商家呼单	订单状态：4，运单状态：2
	骑手接单	订单状态：4，运单状态：3
	骑手取餐	订单状态：4，运单状态：4
	骑手送达	订单状态：6，运单状态：6
业务阶段在指定范围	提交订单	订单状态：2，订单新建
	实际用户 T	订单状态：6，订单完结
	用户付款	订单状态：3，运单状态：0
	商家接单	订单状态：4，运单状态：1
	商家呼单	订单状态：4，运单状态：2
	骑手接单	订单状态：4，运单状态：3
	骑手取餐	订单状态：4，运单状态：4
	骑手送达	订单状态：6，运单状态：6
未发生事件	商家接单	订单状态：4，运单状态：1，使用时需要取差集
	商家呼单	订单状态：4，运单状态：2，使用时需要取差集
	骑手接单	订单状态：4，运单状态：3，使用时需要取差集
	骑手取餐	订单状态：4，运单状态：4，使用时需要取差集
	骑手送达	订单状态：6，运单状态：6，使用时需要取差集
时间比较符号	+	指定场景之后的时间
	-	指定场景之前的时间
	=	正好在指定场景所在时间

表 3.14 为生效范围规则选项表，如表所示，生效范围的可配置项有城市区域、订单来源、物流类型、业务类型、天气类型。与超时检查类似，生效范围也是通过树的方式进行组织。生效范围不和超时检查一起设计是因为，生效范围可

以与检查规则一起使用，也可以和 Action 规则一起使用，因此需要独立出来设计。

表 3.14 生效范围规则选项表

生效范围	可选值
城市区域	代理商城市（全国 2844 个三级行政区）
	直营城市（(全国 2844 个三级行政区)）
订单来源	PC 端 web
	手机端 web
	iphone App
	Android App
	openapi_platform
	微信小程序
	微信 KA
物流类型	蜂鸟专送
	众包
	择时达专送
	新零售专送
	蜂鸟快送
	新零售
	E 配送
	蜂鸟混合送
	蜂鸟质选
	全城送
	蜂鸟独家
业务类型	餐饮
	新零售
	全选
天气类型	晴天
	雨天
	雾天
	雪天
	大风
	高温

表 3.15 为 Action 规则选项表，如表所示，目前支持的 Action 为用户触达、用户取消订单、平台赔付、商家索赔操作，对于每一个可选操作，内部都会有很多细分的选项，这样的设计能够更为细粒度的制定规则。触发行为一般还需要配合生效范围使用，对于每一个 Action，用户都可以配置其生效范围，而不是沿用规则检查时使用的生效范围，这样设计的目的是为了增加灵活性，细化规则的粒度。当用户没有配置 Action 生效范围时，就沿用规则检查的生效范围。

表 3.15 Action 规则选项表

Action	可选值	
用户触达	触达类型	短信
		消息推送
	接受者	用户
		客户
	短信模板	
	备注说明	
	生效范围	
用户取消订单	是否支持	支持
		不支持
	生效范围	
平台赔付	赔付类型	红包
		优惠券
		积分
	触发时间点	订单取消或完成时
		场景触发时
	赔付规则名称	
	渠道号	
	生效范围	
商户索赔	是否支持	支持
		不支持
	生效范围	

为了能够让执行单元具有判断的能力，需要设计一些通用函数，例如逻辑运算、算数预算，同时还需要设计一些与业务相关的函数，例如判断规则匹配是否需要延时、判断当前订单、运单状态等。规则设计如表 3.16 所示。

表 3.16 函数设计表

函数名	函数介绍	参数介绍
AND	逻辑与	两个参数，参数类型为布尔值
OR	逻辑或	两个参数，参数类型为布尔值
LT	小于，左边值<右边值	两个参数，参数类型为数值类型
LE	小于等于，左边值<=右边值	两个参数，参数类型为数值类型
GT	大于，左边值>右边值	两个参数，参数类型为数值类型
GE	大于等于，左边值>=右边值	两个参数，参数类型为数值类型
ADD	加	两个参数，参数类型为数值类型
SUBTRACT	减	两个参数，参数类型为数值类型
MULTIPLICATION	乘	两个参数，参数类型为数值类型
ENV	从上下文中获取指定对象，并从对象中获取指定值	两个参数，参数类型为字符串类型
IN	关键值在指定集合中	N（N>1）个参数，第一个参数为关

		键值，后续 N-1 个参数为集合中的值，N 个参数的类型需要一致，可以为数值类型，也可以为字符串类型
TRIGGER_AT	判断第二个参数是否为 true，并且判断参数是否需要延时	两个参数，第一个参数为数值类型，表示预计触发时间，第二个为布尔值，表示是否存在未发生的事情
ORDER_AND_DELIVERY_STATUS	订单和运单状态	两个参数
DELIVERY_STATUS_DOUBLE_CHECK	运单装填双重检测	List 集合

订单消息如表 3.17 所示，系统根据订单中携带的信息进行规则的匹配，表中较为重要的信息为订单来源、是否为预订单、订单 id、发送时间。订单来源、是否为预订单均为规则中潜在判断条件；订单 id 可以作为查询参数，进而可以获取到整个订单信息。

表 3.17 订单消息体表

字段名	类型	描述
Type	String	消息代表的含义
Description	String	具体说明
Paload.order_id	Long	订单 id
Paload.role	Integer	消息发起角色
Paload.restaurant_id	Integer	餐馆 id
Paload.user_id	Integer	下单用户 id
Paload.come_from	Integer	订单来源
Paload.is_book	Integer	预订单
Paload.sent_at_millsec	Long	发送时间戳
Paload.order_mode	Integer	订单模型

超时消息表如表 3.18 所示，当订单或运单发生超时之后，系统就会发送一条超时消息到消息队列，以便监听到该消息队列的其他应用能够感知到超时。

表 3.18 超时消息表

字段名	类型	描述
Type	Integer	消息代表的含义
Event_name	String	事件名称
Payload.order_id	Long	订单 id
Payload.timeout	Long	超时时间
Payload.start_time	Long	开始时间

运单消息如表 3.19 所示，系统根据运单中携带的信息进行规则的匹配，表中较为重要的信息为运单 id、运单主状态、运单子状态。运单 id 可以作为参数

来获取订单信息；运单主状态和运单子状态可以用于判断当前运单所处阶段。

表 3.19 运单消息表

字段名	类型	描述
Type	String	消息代表的含义
Description	String	具体说明
Payload.carrier_driver_id	String	骑手 id
Payload.carrier_id	String	配送商 id
Payload.deliverer_name	String	骑手名称
Payload.deliverer_phone	String	骑手电话
Payload.order_id	Long	订单 id
Payload.send_at	Long	发送时间
Payload.state	String	运单主状态
Payload.subState	String	运单子状态
Payload.tracking_id	Long	运单 id
Payload.update_time	Long	更新时间

审批流消息如表 3.20 所示，系统与审核流平台之间通过消息队列进行交互，系统依赖于审核流平台的消息，根据审核流消息对审核权进行转移。表中较为重要的参数为上一个状态、当前状态，通过这两个状态就能够决定审核权应该如何转移。

表 3.20 审核流消息体表

字段名	类型	描述
Apply_instance_id	Long	审批实例 id
Previous_state	String	上一个状态
Current_state	String	当前状态
Aaudit_log_id	Long	审核操作 ID
Event_type	EventType	事件类型
Parameters	String	参数
Operator	Operator	操作人

3.4.2 架构设计

图 3.8 为系统架构图，如图所示，系统架构分为了展示层、Controller 层、Service 层、Data Access 层、Persistent 层和 Infrastructure 层。与传统的 MVC 架构相比，系统架构细化了系统结构，在应对复杂业务的时候，能够根据层次进行细分，同时能够提高已有能力的复用率。展示层为用户能够访问的页面，该层包含分页控件、菜单控件、布局控件等一些前端控件；Controller 层为后端控制

层，主要负责对外提供 API 接口，同时能够控制用户的访问权限；Service 层为业务层，负责处理业务相关的逻辑。

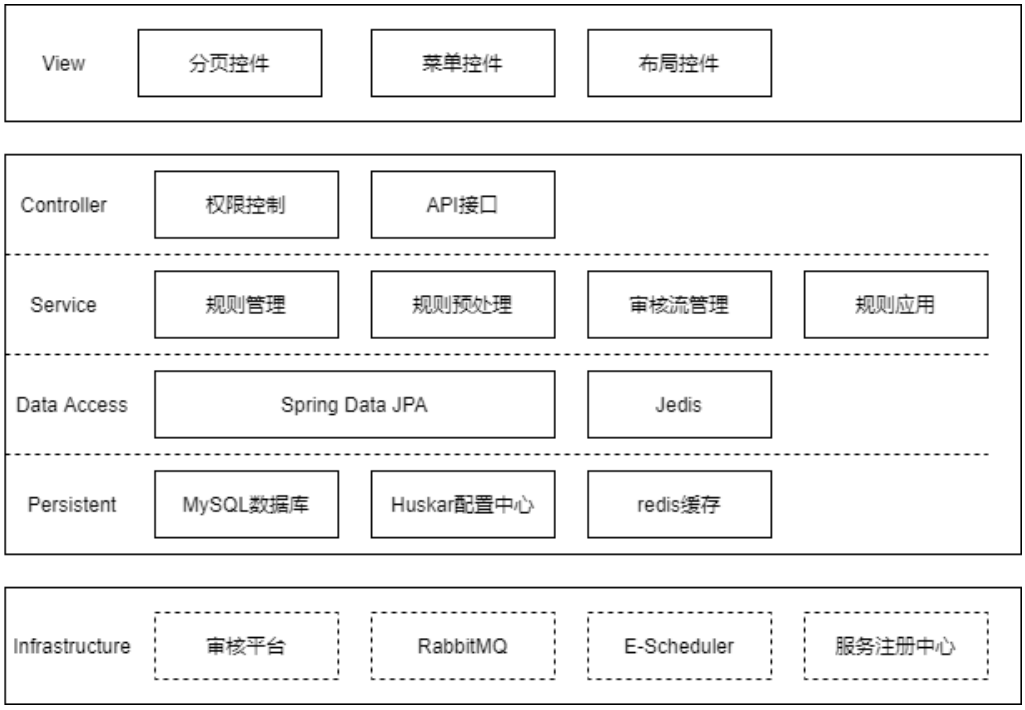

图 3.8 系统架构图

Data Access 为数据接入层，数据接入分为 MySQL 数据接入和 Redis 数据接入。JPA 是 Java Persistent API 的简称，是 JAVA EE 5.0 推出的 ORM 规范，目的是让应用能够以统一的方式访问持久层，Spring Data JPA 是 JPA 规范的一种实现，在以 Spring 为容器开发的项目中，使用 Spring Data JPA 开发项目会更轻松；Jedis 是一款开源的、轻量级 Java 客户端，实现了对 Redis 各类数据结构的基本操作，同时实现了分布式锁以及分布式原子性操作。

Persistent 为持久层，内部包含了数据库 MySQL、配置中心 Huskar、缓存 Redis。MySQL 是当前互联网中主流的关系型数据库；Redis 为非关系型数据库，负责对数据进行缓存，以加快数据访问速度，同时减轻 MySQL 的访问压力。Huskar 为配置中心，负责存储需要经常变动的配置信息，例如服务灰度开关、服务降级开关等。

Infrastructure 为基础设施层，负责支撑系统运行。系统发布到线上之后，服务注册中心会对系统进行注册，并将其暴露，以便其他服务发现并调用；审核平台和任务调度中心为系统依赖的基础服务；RabbitMQ 为消息中间件，系统通过 RabbitMQ 与其他服务进行交互。

3.4.3 功能组成设计

根据 3.3 章节需求分析，对履约超时管理平台进行数据模块划分，每个功能模块承担相应的职责，拥有不同的功能。图 3.9 为系统功能模块图,从图中可知系统分为四个功能模块：规则管理模块、审核流模块、规则预处理模块、业务处理模块。

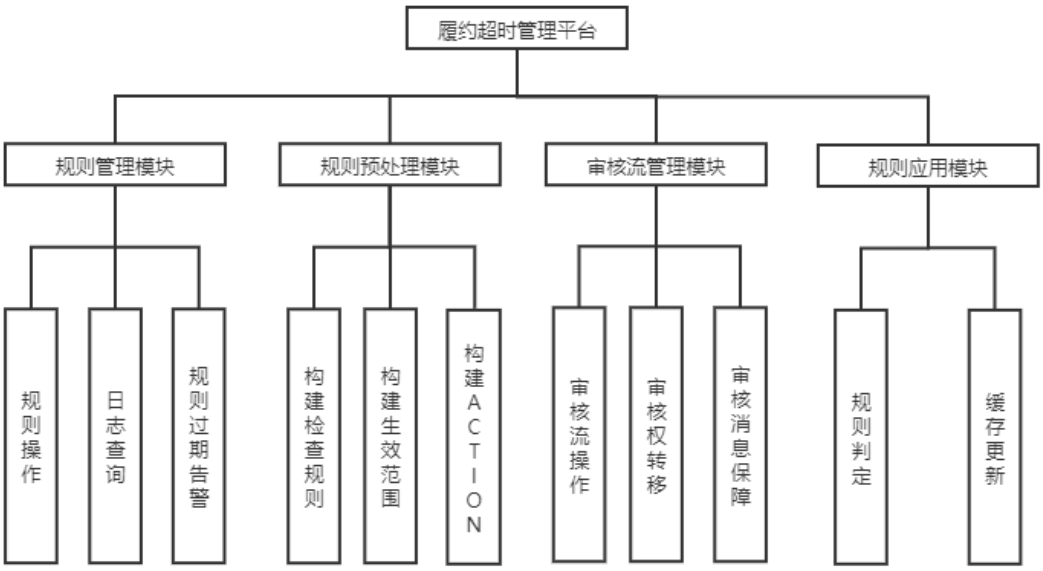

图 3.9 系统功能模块图

1. 规则管理模块包括规则操作、日志查询、规则过期告警三个子模块，负责提供规则操作、日志查询等功能。

2. 规则预处理模块包括构建检查规则、构建生效范围、构建 Action 三个子模块，负责将规则数据结构化。

3. 审核流管理模块包括审核流操作、审核权转移、审核消息保障三个子模块，负责规则审核相关功能。

4. 规则应用模块包括规则判定、缓存更新两个子模块，负责订单超时检查和执行应对策略。

3.4.4 部署设计

图 3.10 为系统部署图。应用部署时采用异地多机房技术，在张北机房和王

国机房均有部署。从单个机房来看，应用部署在 Kubernetes 集群中 worker 节点上，Pod 是 Kubernetes 中最小部署单元，Pod 可以包含一个或者多个容器，部署在 worker 节点上的应用会以 Pod 形式存在和管理。为了避免单点故障，应用需要以集群的方式进行部署。应用需要分别生产环境、预发环境、测试环境中进行部署，生产环境为真实投入使用环境，预发环境为发布前需要认证的环境，与生产环境相比，应用不真实投入使用，但是与生产环境共用一套数据库，测试环境与生产和预发环境完全隔离，测试环境的部署相对于前面两个环境较为简易。应用采用 Docker+Kubernetes 组合进行持续集成和持续交付。使用 Docker 由如下原因：Docker 技术可以将服务容器化，能够屏蔽由于底层物理机差异和操作系统差异而带来的环境差异；应用可以共享一个操作系统的内核，因此不需要为每一个应用准备一个操作系统，更加高效、解决系统资源。使用了 Docker 之后，需要对 Docker 中的容器进行管理，因此需要使用 Kubernetes，Kubernetes 可以对容器化的应用进行自动部署、扩容、管理、灰度升级等。

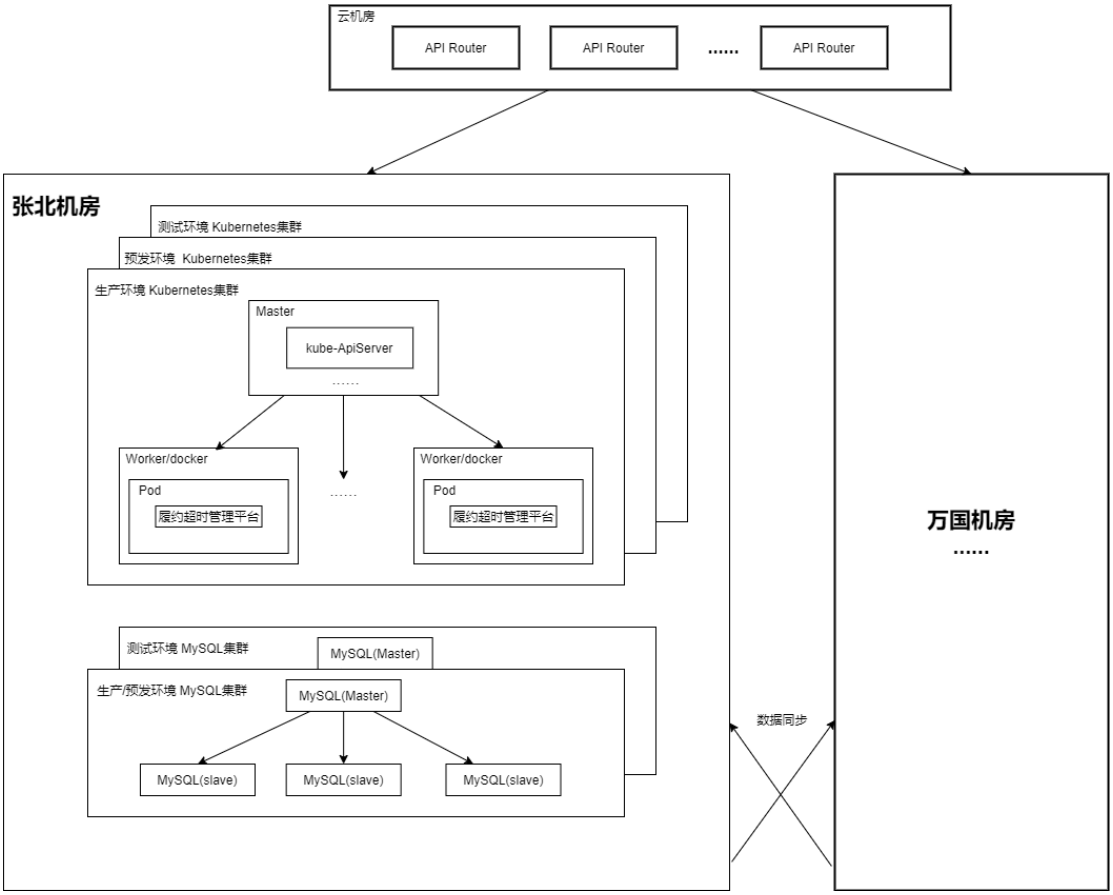

图 3.10 部署设计

3.5 数据库设计

本节对数据库系统数据库进行设计。

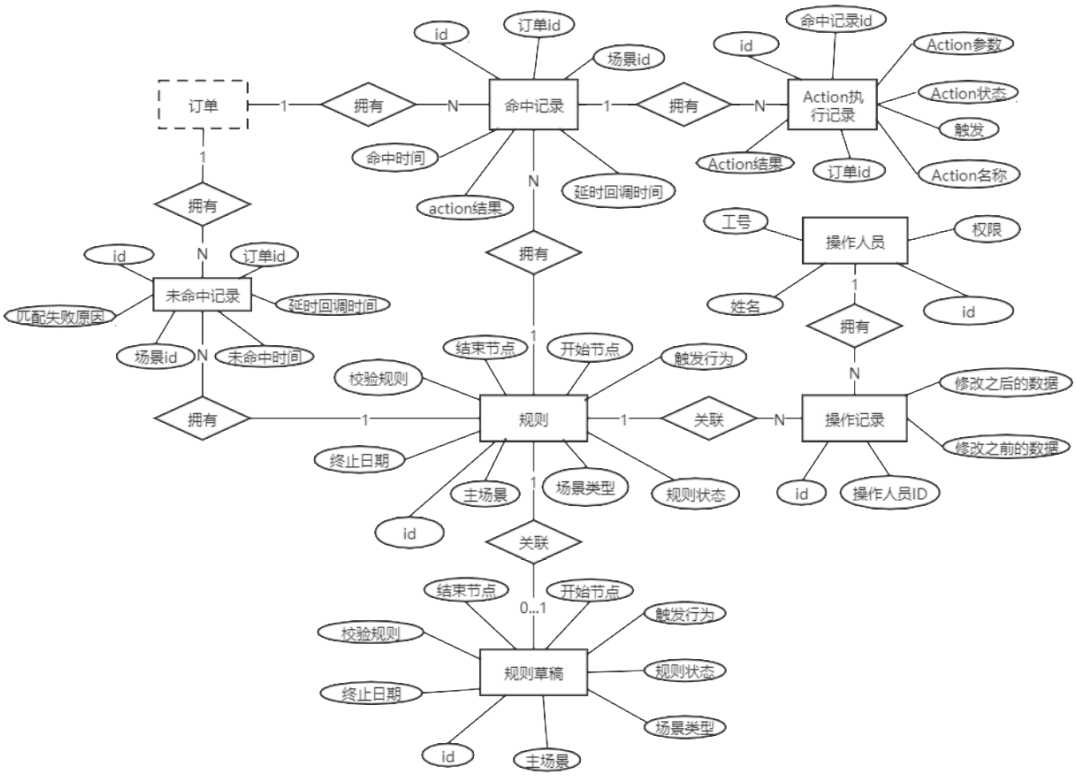

图 3.11 实体关系图

图 3.11 为实体关系图，如图所示，数据库的设计围绕着规则和对规则的操作类型展开。rule 表负责存储用户配置的规则。rule_draft 表负责存储审批通过之前的规则，审批通过之后的规则存放在 rule 表中，由于规则的审批可能会不通过，因此 rule_draft 和 rule 的对应关系为 1 对 0 或 1。rule_hit 表负责存储命中规则的信息，rule 和 rule_hit 的对应关系为 1 对 N。rule_not_hit 表负责存储未命中规则的信息，rue 和 rule_hot_hit 的对应关系为 1 对 N。action_execute 表负责存储 action 执行之后的结果，一条规则可以有多个 action，因此 rule 和 action_execute 的对应关系为 1 对 N。operate_record 表负责存储规则的修改记录，rule 和 operate_record 的对应关系为 1 对 N。operator 表负责存储操作人员信息，operator 和 operate_record 表的对应关系为 1 对 N。图中还有一个 order 实体，该实体属于订单系统，但是与本系统中的 rule_hit 和 rule_not_hit 有关联。order 和 rule_hit 的对应关系为 1 对 N；order 和 rule_not_hit 的对应关系为 1 对 N。

数据库表字段设计如下：

5. 规则表 rule 存储审批通过的规则，其中 check_rules 存储校验规则，Actions 存储执行动作，构建完成之后的规则通过转换成 JSON 的形式存储在表中，使用的时候再转换成需要的对象。 rule 表结构如 3.21 所示。

表 3.21 rule 表结构

字段名	类型	是否允许为空	描述
id	Long	否	规则 id
type	int	否	场景类型
description	int	是	场景码
Trade_node	Varchar	否	主场景
Start_trade_node	Varchar	否	业务开始节点
End_trade_node	Varchar	否	业务结束节点
Trigger_event	Varchar	否	触发事件
Status	Int	否	规则状态
Start_time	Long	否	开始时间
End_time	Long	否	结束时间
Check_rules	Text	否	校验规则
Actions	Text	否	触发行为

6. 规则未命中表 rule_not_hit 存储订单未命中规则记录。rule_not_hit，其中最主要的属性为 fail_reason，记录了匹配失败的原因，用户可以通过该属性来排查问题。与 rule_hit 表类似的是，rule_not_hit 表会存放大量的数据，且 order_id 为高频查询关键字，因此需要对 order_id 加索引。rule_not_hit 表结构如图 3.22 所示。

表 3.22 rule_not_hit 表结构

字段名	类型	是否允许为空	描述
Id	Long	否	Id
Order_id	Long	否	订单 id
Scene_id	Long	否	场景 id
Scene_code	Int	否	场景 code
Occure_time	Long	否	出现时间
Fail_reason	Tex	否	匹配失败原因
Delay	Long	是	延迟回调时间

7. 规则命中记录表 rule_hit 存储订单命中规则记录。rule_hit 表主要做查询使用，由于饿了么一天的数据量为千万量级，为了加快查询速度，需要对高频关键字做索引。结合需求分析，得出的结论为 order_id 是一个潜在的高频查询关键

字，因此，需要对 order_id 加索引。rule_hit 表结构如表 3.23 所示。

表 3.23 rule_hit 表结构

字段名	类型	是否允许为空	描述
Id	Long	否	Id
Order_id	Long	否	订单 id
Scene_id	Long	否	场景 id
Scene_code	Int	否	场景 code
Occure_time	Long	否	出现时间
Action_result	Text	否	Action 执行结果
timeout	Long	否	超时时间
Delay	Long	是	延迟回调时间

8. 规则草稿表 rule_draft 负责存储规则审核信息，只有当规则审核通过之后才能将规则存入 rule 表。rule_draft 和 rule 表的结构相同，数据上可能存在冗余，但这样设计的好处在于审核中的规则不会影响到已经投入使用的规则，在边界上有明显的区分，不易出错；同时 rule_draft 可以保存历史提交审核的规则，能够辅助用户进行决策。rule_draft 表结构如图 3.24 所示。

表 3.24 rule_draft 表结构

字段名	类型	是否允许为空	描述
id	Long	否	规则 id
type	int	否	场景类型
description	int	是	场景码
Trade_node	Varchar	否	主场景
Start_trade_node	Varchar	否	业务开始节点
End_trade_node	Varchar	否	业务结束节点
Trigger_event	Varchar	否	触发事件
Status	Int	否	规则状态
Start_time	Long	否	开始时间
End_time	Long	否	结束时间
Check_rules	Text	否	校验规则
Actions	Text	否	触发行为

9. 规则触发表 action_execute 存储规则执行结果。在 rule_hit 表中也有 action 的执行结果，action_result 为冗余设计，用户在查看命中情况的时候，更为关心 action 执行结果，如果不做冗余设计，需要做数据库连接查询，会增加数据库的负担。如果需要查询详细的信息，就可以查询 action_execute 表。action_execute 表结构如图 3.25 所示。

表 3.25 action_execute 表结构

字段名	类型	是否允许为空	描述
id	Long	否	Id
Record_id	Long	否	触发记录 id
Status	Long	否	场景动作状态
Params	Text	否	场景动作参数
Results	Text	否	场景动作结果
Type	Int	否	动作类型
Order_id	Long	否	订单号
Trigger_event	Varchar	否	触发事件
Rule_id	Long	否	规则 id
Name	Varchar	否	动作名称

10. 操作记录表 operate_record 存储了操作人员对规则的修改记录。其中包含了操作记录的 id、操作人员的 id、规则修改之前的数据、规则修改之后的数据。operate_record 表结构如表 3.26 所示。

表 3.26 operate_record 表结构

字段名	类型	是否允许为空	描述
Id	Long	否	Id
Operator_id	Long	否	操作人员 ID
Change_before	Text	否	修改之前的数据
Change_after	Text	否	修改之后的数据

11. 操作人员表 operator 存储了可以进行规则修改的操作人员。饿了么使用了阿里内部员工管理系统，因此不需要通过账号密码来登录系统，但是每个操作人员的权限需要设置，表中权限的字段为 authority，权限主要有不可查看、可查看、和可修改这三个选项。operator 表结果如表 3.27 所示。

表 3.27 operator 表结构

字段名	类型	是否允许为空	描述
Id	Long	否	操作人 id
Name	Varchar(20)	否	操作人名字
Employee_num	Long	否	操作人工号
Authority	String	否	权限

3.6 本章小结

本章介绍了系统存在的需求以及系统的总体设计，本章首先通过系统边界图界定了系统边界，之后通过结合用例图和用户的实际需求，对系统进行的需求分

析；通过对系统的架构设计、功能组成设计、部署设计等完成了对系统的总体设计。本章节的最后介绍了数据库的设计，包括 ER 图的设计以及数据表的设计。

第四章 详细设计与实现

4.1 规则管理模块

4.1.1 规则管理模块详细设计

规则管理模块主要负责对已有规则的新增、删除、修改、查询等，同时，为了方便运营人员进行命中与未命中情况的统计，方便开发人员排查线上问题，对命中规则和未命中规则的订单做了相应的统计。

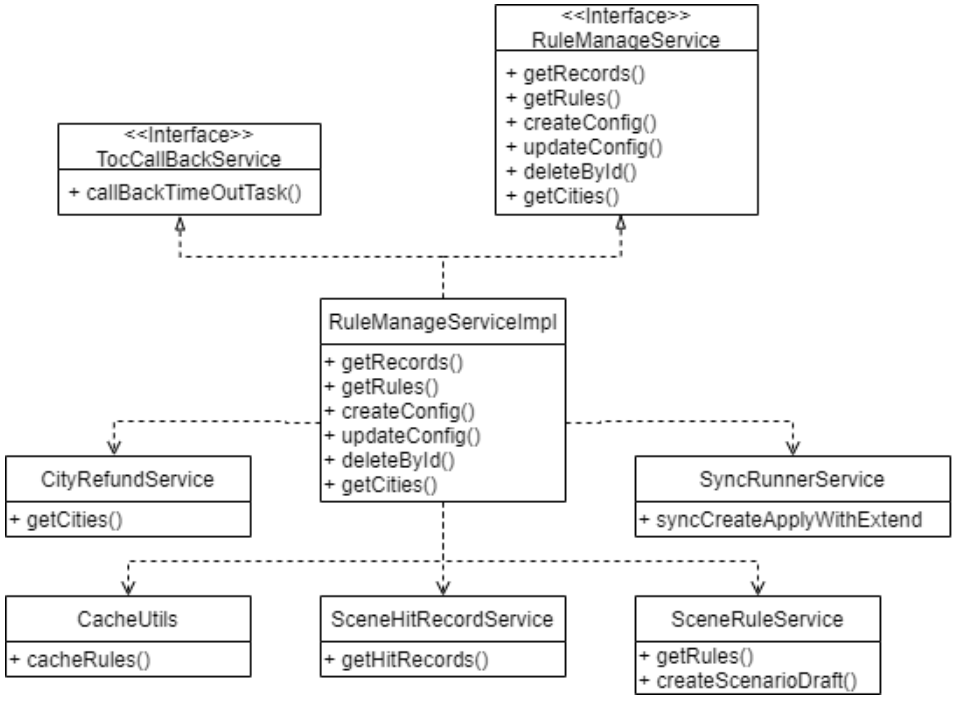

图 4.1 规则管理类图

图 4.1 为规则管理的设计类图，RuleManageServiceImpl 实现了 TocCallBackService 接口和 RuleManageService 接口，TocCallBackService 接口为延时任务中心回调接口，用以实现过期告警功能，RuleManageServiceImpl 实现了 callbackTimeOutTask 方法；RuleManageService 接口定义了规则操作和日志查询方法。SceneHitRecordService 负责处理日志查询相关操作。CacheUtils 负责处理缓存相关操作。OrderQueryService 负责获取订单信息。SecenRuleService 负责操作已有规则。SyncRunnerService 负责审核流操作。

4.1.2 规则操作功能实现

规则操作分为新增规则、删除规则、修改规则、查询规则。上述四个操作的实现有较高重合度，因此以查询规则和新增规则为例。图 4.2 为新增规则和查询规则的时序图。查询规则操作主要涉及到 SceneRuleService、SceneRuleDao、CacheUtils 这三个类，SceneRuleService 调用 SceneRuleDao 来获取已有的规则，调用 CacheUtils 来缓存查询到的规则。新增规则操作主要涉及到 SceneRuleService、SyncRunnerService、SceneRuleDraftDao 这三个类，SceneRuleService 调用 SyncRunnerService 在审核平台上创建审核流，调用 SceneRuleDraftDao 在本地创建规则草稿，本地规则草稿与审核平台中的审核流一一对应。

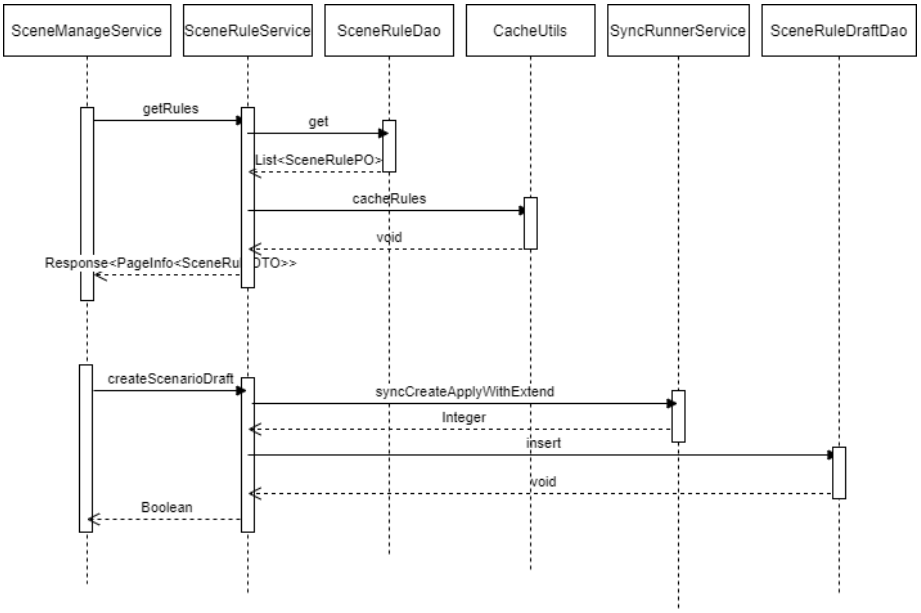

图 4.2 规则操作时序图

图 4.3 为新增规则和查询规则实现代码。方法 createScenarioRule 需要传入一个 RuleContent 对象，该对象中保存用户配置的规则信息，方法拿到 RuleContent 后，会对规则进行校验，以确保规则符合要求。校验通过之后方法调用 syncRunnerService 的 syncCreateApplyWithExtend，在审核平台中创建审核流。在创建完审核流之后，方法调用 SceneRuleDraftDao 的 insert 方法，在本地创建规则草稿。

方法 getRules 负责查询指定规则。查询操纵与新建操作基本类似，稍有区

别的是查询操作需要将查询结果进行缓存。getRules 方法中调用 CacheUtils 的 cacheRules 方法，将查询到的规则缓存在 Redis 中。

```
public Boolean createConfig(RuleContent ruleContent){
    validate(ruleContent); // 1. 校验
    ……// 2. 判断规则是否已经存在
    syncRunnerService.syncCreateApplyWithExtend(ruleContent);//3.创建审核流
    // 4. 创建本地规则草稿
    String JsonStr JSONObject.fromObject(ruleContent).toString();
    boolean result = sceneRuleDraftDao.inset(JsonStr);
    return result;
}
public Response<List<SceneRuleDTO>> getRules(RuleQuery ruleQuery){
    validate(ruleQuery); // 1. 校验参数
    List<SceneRulePO> rulePOList = sceneRuleDao.get(ruleQuery);// 2. 获取数据
    // 3. 将查询到的值在 Redis 中进行缓存
    CacheUtils.cacheRules(rulePOList);
    // 4. 将 po 转换成 DTO
    List<SceneRuleDTO> ruleDTOList = transfer(rulePOList);
    // 5. 返回结果
    return Response.SUCCESS(ruleDTOList);
}
```

图 4.3 新增规则实现

考虑到截取生产环境界面存在数据泄露风险，论文中的截图均为测试环境截图。图 4.4 为规则配置详情页面。

图 4.4 规则配置详情页面

4.1.3 日志查询功能实现

日志查询功能如图 4.5 所示。OrderHitRecordQuery 封装查询条件，包括命中时间、订单号、命中规则类型等。SceneHitRecordService 类负责规则命中和规则未命中相关操作。getHitRecord 调用 SceneHitRecordService 获得 DTO 类型返回值，之后将 DTO 类型的数据转换成给前端展示的 VO 类型数据，虽然两者在大多数的时候是完全一样的，但是做区分更有利于减少因需求变更而导致的代码变动。最后将转换好的结果通过封装至 Response 对象中，返回给前端。Response 对象中会记录返回码、返回数据等。

```
public Response getHitRecord(RecordQuery query){
    // 1. 校验规则
    validate(query);
    // 2. 调用查询服务
    OrderHitRecordDTO recordDTO = sceneHitRecordService.getHitRecord(query);
    // 3. 将 DTO 转换成 VO
    OrderHitRecordVo result = new OrderHitRecordVo();
    transfer(OrderHitRecordDTO, result);
    // 4. 将结果封装到 Response 中
    Response response = Response.SUCCESS(result);
    return result;
}
```

图 4.5 日志查询功能实现

4.1.4 规则过期告警功能实现

过期告警功能如图 4.6 所示，RuleManageServiceImpl 类实现了延时任务中心的回调接口，当任务快过期时，延时任务中心会回调 callBackTimeOutTask 方法。方法中实现了发送邮件功能。

```
@Override
// 延时任务中心回调方法
public void callBackTimeOutTask(String ruleName){
    // prepareEmail 准备邮件发送信息
    SimpleEmail mail = prepareEmail(ruleName);
    Mail.send();// 发送邮件
}
```

图 4.6 规则过期告警功能实现

4.2 规则预处理模块

4.2.1 规则预处理模块设计

规则预处理模块负责将前端配置的规则数据根据固定组织关系进行树化，运营人员配置完规则，点击提交之后，规则数据会通过 JSON 的方式传送到后端，此时的规则仅仅保存数据，无法执行，因此需要将规则数据转换成最小执行单元，并通过树状的形式将最小执行单元进行整合，以达到整条规则能够执行的目的。规则执行单元可以分为两种：运算符单元，包括逻辑运算符“AND”、“IN”和算数运算符“LT_EQ”、“ADD”、“MULTIPLY”，分别对应逻辑与，存在，小于等于，加法，乘法；参数单元，包括“CONST”、“VARIABLE”以及“ENV”，“CONST”代表常量，“VARIABLE”和“ENV”分别代表变量，需要从规则上下文中进行获取。有了规则执行单元之后，就可以通过树的形式组装成一个有业务逻辑的执行单元，例如 orderBookCondition, 判断订单是否为预订单；triggerAt, 指定规则判定时间等。表达式如图 4.7 所示：

```
Rst = (((orderBookCondtion
        AND deliveryProductIdCondition)
        AND predictDeliverTimeCondition)
        AND TRIGGERAT(triggerTimeCondition
        AND orderStatuCondition))
```

图 4.7 规则执行表达式

当表达式右边为 true 的时候，规则才算匹配上，执行树如图 4.8 所示：

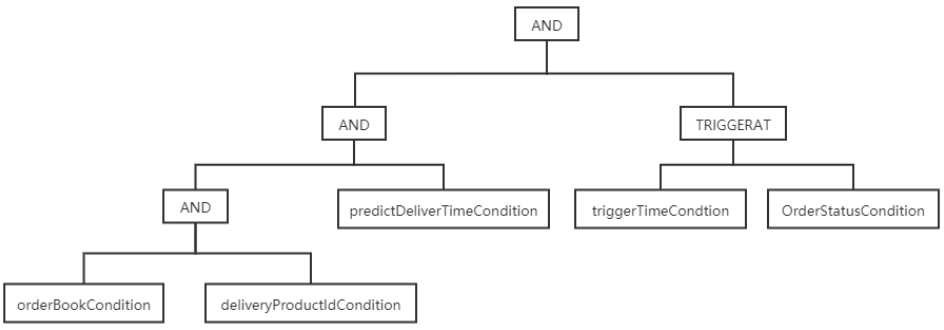

图 4.8 规则解析树概括

叶子节点为封装之后的执行单元，由上文所说的两种基本的执行单元组成，以 predictDeliverTimeCondition 为例，其树状图如图 4.9 所示：

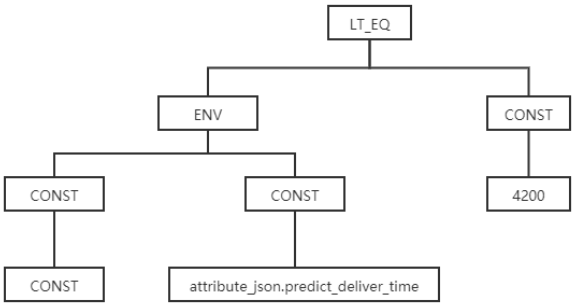

图 4.9 predictDeliverTimeCondition 节点

在实现上，模块采用的是直接生成执行树的方式，而非使用表达式的形式，理由有如下两点：

1. 表达式可以由运营人员直接写出，亦可以在后端根据运营人员配置进行生成，如果让运营人员手动写出，由于需要编写的参数过多，就会大大增加配置出错的可能性，除此之外，对于不会编程的运营人员来说，手动编写表达式的用户体验远远不如图标编辑的用户体验好。

2. 如果使用图标的前端配置方式，那么就意味着需要后端生成表达式，一方面将对象中的值拼接成字符串的形式较为繁琐，同时面临着需要解析表达式的需求，在没有使用第三方规则引擎的情况下，将表达式解析成树状结构是较好的选择。

因此，前端通过图标配置，后端将其构建成树状结构的形式较为适合。

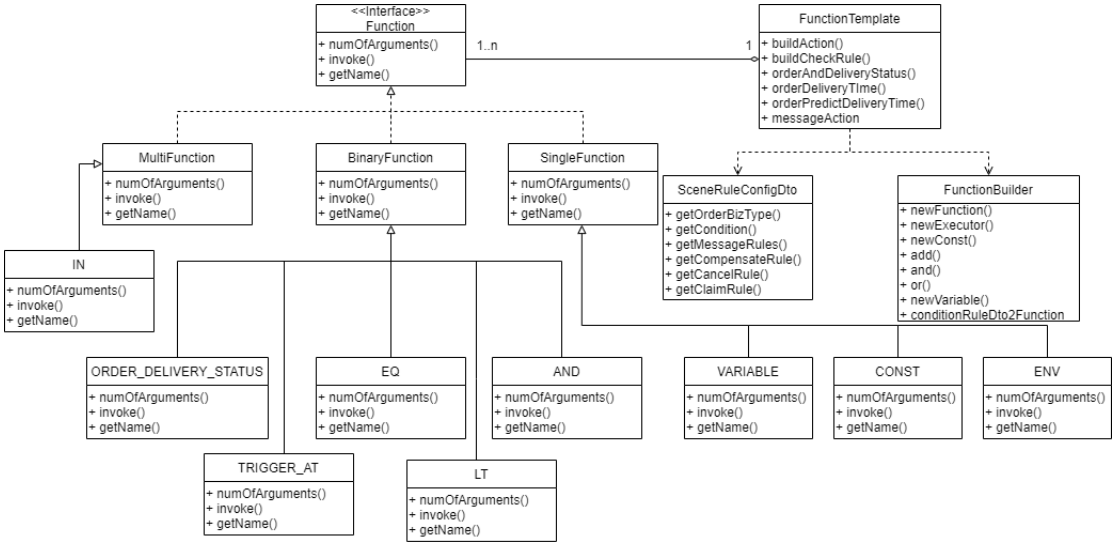

图 4.10 规则预处理类图

图 4.10 为规则预处理模块的类图, Functions 接口是所有执行单元需要实现的接口，接口中方法的含义如下：numOfArguements 函数返回的是执行单元需

要的参数个数，invoke 函数存放着执行单元的执行逻辑，getName 函数返回的执行单元的别称。根据执行单元所需要参数的个数，实现类可以分为 BinaryFunction、SingleFunction 和 MultiFunction，BinaryFunction 为需要两个参数的执行单元，拥有最多的实现类，例如 AND、EQ、LT、TARGET_AT、ORDER_DELIVERY_STATUS；SingleFunction 为需要一个参数的执行单元，实现类为常量执行单元和变量执行单元；MultiFunction 为需要两个以上的执行单元，实现类主要为自定义函数以及 IN 函数。FunctionTemplate 负责对规则的构建，按照规则的类型，可以分为条件校验规则、生效范围规则、执行动作，buildCheckRule 函数负责构建校验规则，以上文描述的树状结构进行构建；conditionRuleDto2Function 函数负责构建生效范围规则，生效范围规则不可以单独存在，需要和检查规则和 Action 规则一起使用。buildAction 函数构建的执行动作，执行动作与条件校验的一大区别在于执行动作并不需要返回值作为执行成功与否的判断条件，也就是说，待执行动作之间没有层级关系，只有并列关系，因此不需要以树状的方式进行组织。在设计 Action 组织方式时，使用 Spring Boot 当中的一些特性，Spring Boot 可以通过标签来获取对象，并执行对应的方法，因此在实现时候，可以定义标识 Action 的标签--@Action，该标签需要注解在 Action 执行类上，同时需要开发获取有@Action 注解的对象代码，并执行@Action 标签中定义的方法。

4.2.2 构建检查规则功能实现

图4.11 检查规则构建实现图, SceneRuleConfigDto 为前端传送的规则信息，后续构建规则需要的数据均从 SceneRuleConfigDto 中获取。方法构建规则时使用 builder 模式，FunctionBuilder 类 newFunction 方法创建一个函数，并返回 builder 对象，add 方法给函数添加参数，and 方法也是给函数添加参数，只不过当前函数根数超过两个时，新建一个 AND 函数，并将原有函数和新添加参数作为它的左右参数，build 方法生成检查树，并返回检查树的根节点。方法首先构建订单类型检查规则，之后构建触发时间检查规则以及是否发生指定事件的检查规则，最后再构建场景额外判断规则，例如配送事件是>70 分钟等。方法最终返回整个检查树的根节点。

```
public FunctionDto buildCheckRule(SceneRuleConfigDto sceneRuleConfigDto){
    FunctionBuilder builder = newFunction(FunctionName.AND)
                                        .add(newConst(true));
    // 构建预订单判断逻辑
    Function func = sceneRuleConfigDto.getOrderBizType() == OrderBizType
                        .COMMON_ORDER?FunctionName.EQ:FunctionName.NEQ;
    builder = builder.and(orderAttrComp(func,"is_book",0));

    TradeNodeConditionDto cond = sceneRuleConfigDto.getConditions();
    // 获取未发生节点
    HotAxeTriggerNode event = HotAxeTriggerNode
                                    .getByType(cond.getTradeNode().getType());
    // 构建触发时间逻辑
    FunctionBuilder triggerAtBuilder = newFunction(FunctionName.TRIGGER_AT)
                    .add(newFunction(cond.getOperator().getFunc())
                    .add(event.getTimeFunc())
                    .add(newFunction(FunctionName.MULTIPLY)
                        .add(newConst(cond.getTime())
                        .add(newConst(1000)).build()).build());
    HotAxeTargetNode target = HotAxeTargetNode
                                    .getByType(cond.getTargetNode().getType());
    // 构建场景额外条件，例如：（配送时间>70min）
    for(TradeNodeExtraCondition nodeCondition extraCond : cond.getExtraConditions()){
        HotAxeExtraCondition nodeCondition = extraCond.getConditionNode();
        builder = builder.and(
            newFunction(extraCond.getOperator().getFUnction())
            .add(nodeCondition.getFunc())
            .add(newConst(extraCond.getTime()))
            .build());
    }
    return bulder.and(triggerAtBuilder.add(target.getFunc()).build()).build();
}
```

图 4.11 检查规则实现图

4.2.3 构建生效范围功能实现

图 4.12 为构建生效范围功能。由于构建生效范围在构建检查规则和构建 Action 时均会被使用，因此在设计和实现时，将规则生效范围逻辑抽取成独立方法，提高代码复用率。conditionRuleDto2Function 方法入参为 Condition 类，前端所有的生效范围信息均封装在 Condition 类中。方法依次获取城市范围、订单

类型、订单来源、运单类型，通过 FunctionBuilder 构建生效范围检查树，最终返回生效范围检查树的根节点。

```
public static FunctionDto conditionRuleDto2Function(Condition condition){
    FunctionBuilder builder = newFunction(
            FunctionName.And).add(newConst(true));
    if(!CollectionUtils.isEmpty(condition.getCityIds())){// 构建城市范围检查
        builder.add(buildConditionRule("attrbute_json.district_id"));
    }
    if(!CollectioUtils.isEmpty(condition.getEleBizIds())){// 构建订单类型检查
        builder.add(buildConditionRule ("order_type"));
    }
    if(checkOrderComeFrom(condition)){ // 构建订单来源范围检查
        builder.add(buildConditionRule("come_from"));
    }
    if(!CollectionUtils.isEmpty(condition.getDeliveryIds())){// 构建运单类型范围检查
        builder.add(buildConditionRule("attribute_json.delivery_product_id"));
    }
    return builder.build();
}
public static FunctionDto buildeConditionRule(String field){
    FunctionBuilder inFunc = newFunction(FunctionName.IN)
            .add(orderAttr(field));
    condition.getCityIds().forEach(id -> inFunc.add(newConst(id)));
    return inFunc.build();
}
```

图 4.12 构建生效范围实现图

4.2.4 构建 Action 功能实现

图 4.13 为构建 Action 功能实现图。与构建检查规则不同的是，构建 Action 使用的是 Executor 函数，构建检查规则时，每个节点均是可执行节点，但是在构建 Action 时，Action 函数是非可执行节点，仅仅保留自定义注解@Action 中 name 的值，Executor 利用 Spring 容器，通过@Action 标签获取所有 Action，并通过 name 值获取到指定的执行类，最终在 Excutor 的 invoke 方法中执行对应 Action 动作。这样设计的好处在于充分利用了 Spring 容器控制反转特性，Spring 容器初始化时已经构建了所有非懒加载对象，容器中的对象可以通过指定标签获取。Executor 中获取 Action 并执行的逻辑能轻松应对未来可见的变更，例如新

增 Action，仅仅需要在实现类上添加@Action 标签，删除 Action 仅需要删除实现类上的@Action 标签。buildAction 从 sceneRuleConfigDto 中获取 Action 信息，并依次构建用户触达、用户赔付、用户取消、商家索赔 Action。buildAction 方法返回值为 Action 执行树的根节点。

```
public FunctionDto buildAction(SceneRuleConfigDto sceneRuleConfigDto){
    FunctionBuilder builder = newExecutor();
    // 触达
    if(CollectionUtils.isEmpty(sceneRuleConfigDto.getMessageRule()){
        builder.add(emptyAction());
    }else{
        sceneRuleConfigDto.getMessageRule().stream().forEach(
          item->{builder.add(buildMessageAction(item))});
    }
    // 赔付
    List<FunctionDto> compensations = buildCompensationAction(
        sceneRuleConfigDto.getCompensateRule(), sceneRuleConfigDto.getTime());
    compensations.stream().forEach(item->{builder.add(item);});
    // 用户取消
    if(Boolean.TRUE.equals(sceneRuleConfigDto.getCancelRule().getSupport())){
        FunctionDto cancelCondition = conditionRuleDto2Function(
                sceneRuleConfigDto.getCancelRUle().getCondition());
        builder.add(buildCancelAction(cancelCondition));
    }
    // 商户索赔
    builder.add(builderClaimAction(sceneRuleConfigDto.getClaimRule()).build());
    return builder.build();
}
```

图 4.13 构建 Action 功能实现图

4.3 审核流管理模块

4.3.1 审核流管理模块设计

审核流模块对新增或修改后规则合理性提供了保障，防止出现因用户操作失误而导致配置出错误的规则，从而直接导致公司的资损。审核流模块的审批是借用的公司已有的审核流平台，在开发过程中，审核流模块需要接入审核流平台，两者之间的交互通过消息队列的方式进行，用户新增或者修改规则之后，就会在审核流平台创建一个审核流，审核人员会在审核流上进行审批，审批的结果会发

送到消息队列中，系统审核流模块会监听消息，并根据消息的内容做相应的处理。除此之外，审核流模块需要保证消息丢失之后也能够进行流转，若没有消息可靠性保障，就会存在审核平台审核通过，但是审核流程没法流转到下一个节点的问题，造成规则无法及时生效，进一步影响策略的实施。

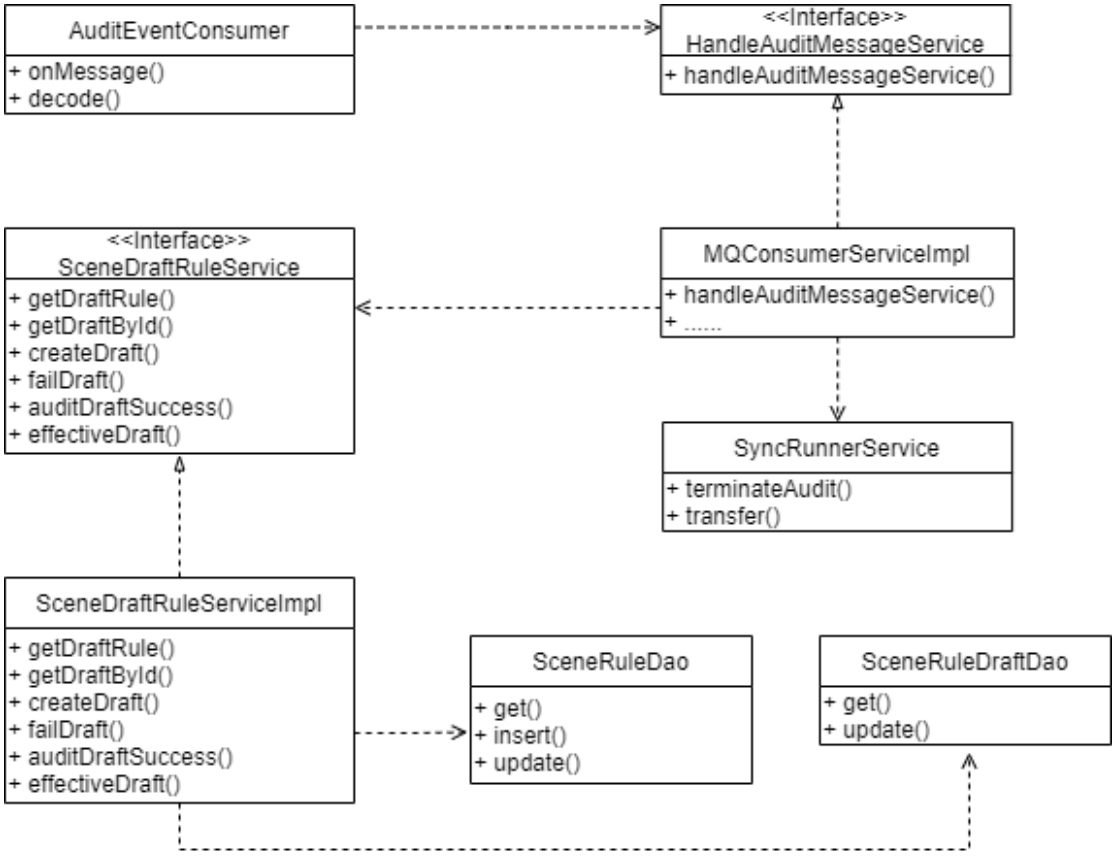

图 4.14 审核流模块类图

图 4.14 为审核流的设计类图，AuditEventConsumer 是一个消息消费者类，消费从消息队列发送过来，消息监听线程会调用 AuditEventConsumer 的 onMessage 方法。对于接受到的消息，onMessage 方法内会调用 decode()方法消息进行解码。消息的处理接口是 HandleAuditMessageService 的 handleAuditMessage()方法。HandleAuditMessageService 的实现类中引用了两个对象：RunnerService 和 SceneDraftRuleService。RunnerServic 是审核流平台提供的接口，通过 RPC 调用具体服务，负责进行流程的流转，当获取到消息队列中的消息时，消息中会携带当前审批节点和审批结果，根据这两个消息可以得出审批是否通过，以及通过之后下一个审批人员，如审批通过，此时可以调用 RunnerService 的 transfer 方法，将审核节点流转到下一个节点。

SceneDraftRuleService 为本地实现的服务，主要是对规则草稿进行操作，新建的策略并不会直接更新到 Rule 表中，而是先存在草稿表中，之后如果审核通过，再将其插入到 Rule 表中，除此之外，草稿表在解决消息丢失问题中起重要作用，审核平台会将审核流的操作记录到它本地数据库中，如果存在消息丢失的情况，那么此时审核平台数据库中的数据就是最新状态，如果将草稿表中的数据与审核平台库中的数据进行比对，那么就可以解决消息是否丢失的问题，因此需要设置定时任务，定时两个数据库中数据的差别，并以审核平台数据库为准，将草稿表中的数据更新。操作包括创建规则草稿，更新规则审核信息，获取规则草稿等操作。当审批通过的时候，需要调用 aduitDraftSuccess 方法，将草稿表中的数据设置为成功，之后需要调用 effectiveDraft 方法，该方法会将规则插入到 Rule 表中，之后就可以进行相应的匹配。SceneDraftRuleServiceImpl 中关联了三个本地对象，SceneRuleDao 负责对 Rule 表进行操作，SceneRuleDraftDao 负责对草稿表进行操作，JedisFacede 负责对缓存进行操作。

4.3.2 审核流操作功能实现

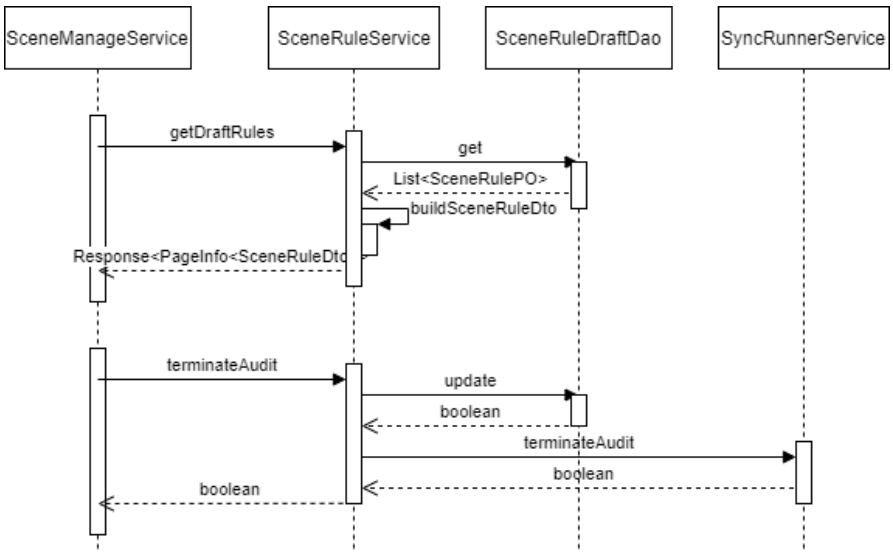

图 4.15 审核流操作时序图

图 4.15 为审核流操作时序图。审核流的操作分为查询审核流和终止审核流，SceneRuleService 通过调用 SceneRuleDraftDao 来实现查询审核流操作，终止审核流分为两步：本地终止和审核平台终止，SceneRuleService 需要分别调用

SceneRuleDraftDao 和 SyncRunnerService 来实现终止功能。

图 4.16 为审核流操作实现代码。getDraftRule 方法根据查询条件获取草稿表中的数据，草稿表中的数据与审核平台中审核流数据同步，因此查询草稿表可以获取到审核流的信息。terminateAudit 方法负责终止审核流，终止审核流需要跨系统操作，因此需要保证本地系统和审核流系统都能够执行成功，代码实现时先调用审核系统的 terminateAudit 方法，只有该方法执行成功后才可以终止本地审核流，如果审核系统的 terminateAudit 方法执行失败，本地的 terminateAudit 会返回 false，前端用户会收到终止失败提示。

```
public Response getDraftRule(DraftRuleQuery query){
    validate(query); // 1. 参数校验
    List<DraftRuleDTO> draftRuleDTOList = sceneRuleDraftService.get(query);//查询
    return Response.SUCCESS(transferToVO(draftRuleList)); // 2. 返回结果
}
public Boolean terminateAudit(DraftRuleVO draftRuleVO){ // 终止审核流
    …//参数校验，VO 转 DTO
    Boolean isSuccess = runnerService.terninateAudit();// rpc 调用。
    If( isSuccess) return false;
    sceneRuleDraftService.update();
    return true;
}
```

图 4.16 审核流操作功能实现图

图 4.17 为审核流操作页面。

图 4.17 审核流操作界面

4.3.3 审核权转移功能实现

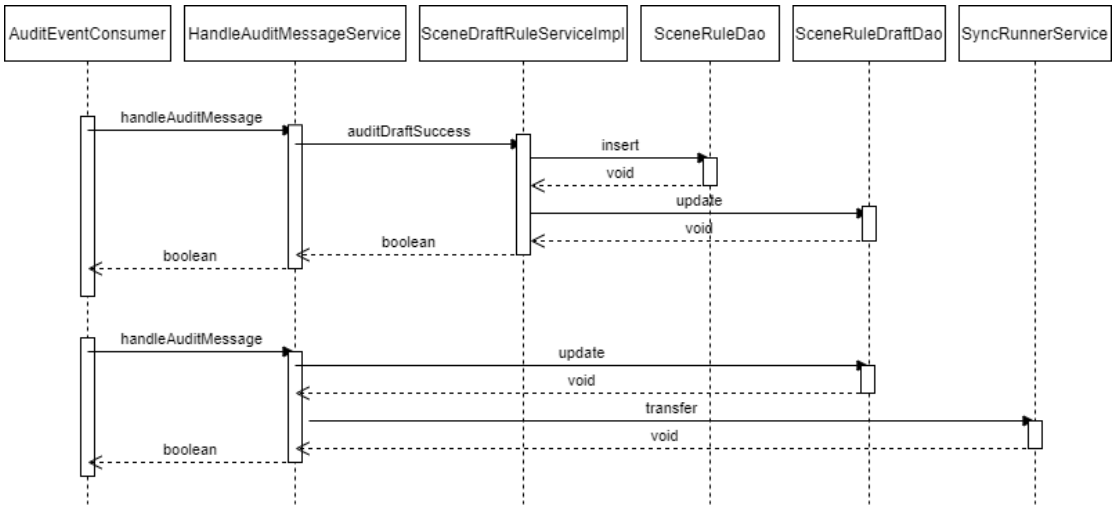

图 4.18 审核权转移时序图

图 4.18 为审核权转移时序图。以审核成功和审核权转移为例，AuditEventConsumer 负责监听审核消息队列，HandleAuditMessageService 处理 AuditEventConsumer 监听到的消息。当审核消息为最后一个节点且审核通过时，HandleAuditMessageService 调用 SceneRuleDao 向 Rule 表中插入新规则，调用 SceneRuleDraftDao 更新本地规则草稿表；当审核消息非最后一个节点但审核通过时，HanleAuditMessageService 调用 SyncRunnerService 转移审核权，同时通过 SceneRuleDraftDao 更新本地草稿表。

图 4.19 为审核权转移功能。审核流节点个数由审核平台限定，每个审核节点的审核人员由履约超时管理平台指定。用户创建规则之后，超时管理平台就会在审核流平台上创建审核流，同时将审核权转移给第一个审核人，在之后的审核权转移中，转移操作均类似，为了减少代码中 if-else 语句的使用，同时增加代码的可读性，审核权转移功能实现时使用了表驱动方法。

```
public void handleAuditMessage(AuditMessage message) {
        …// 1. 校验 message
        // 2. 判断是否到达终点
        checkSuccesOrFail(message);
        String next = NEXT_NODE.get(messsage.currentState);// 表驱动
        RunnerService.transfer(next);
}
```

图 4.19 审核流操作实现

4.3.4 审核消息保障功能实现

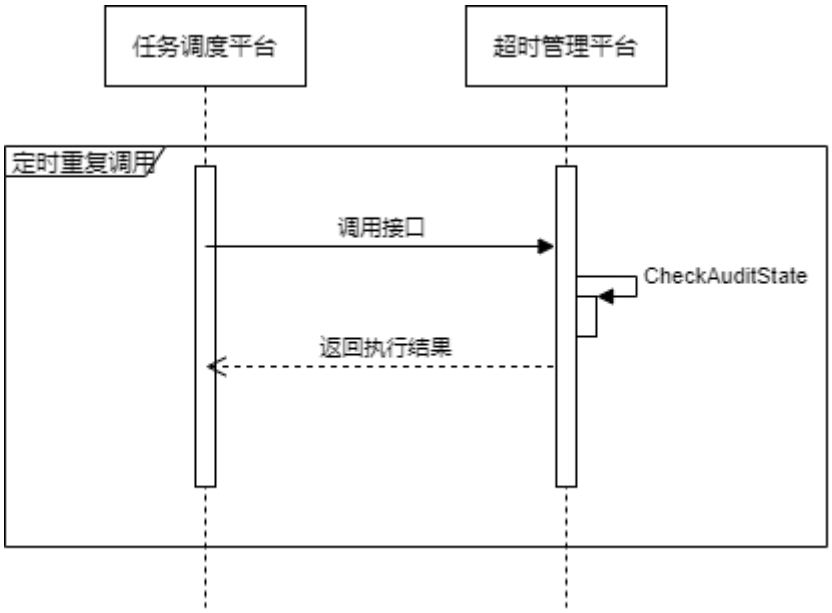

图 4.20 审核消息保障时序图

图 4.20 为审核消息保障时序图。任务调度平台会定时重复调用超时管理平台提供的接口，检测超时管理平台的审核状态和审核平台的审核状态是否一致，并更新超时管理平台中落后的审核状态。

图 4.21 为审核消息保障实现代码。draftRuleService 的 getDraftRules 方法获取本地审核流，之后遍历审核流，并根据审核规则的名称从审核平台中获取对应的审核流，本地审核流与审核平台审核流进行比对，比对有差异则以审核平台审核流为主，更新本地审核流。更新本地审核流可以复用之前审核权转移方法。

```
public void checkAuditState(){
    …// 构建查询参数，
    List<DraftRuleDTO> draftRuleList = draftRuleService.getDraftRules(query);
    for(DraftRuleDTO draftRule : draftRuleList ){
        String ruleName = draftRule.getName();
        // 获取审核平台最新结果
        AuditItem auditItem = runnerService.getByName(ruleName);
        if(auditItem.currentState != auditItem.currentState){// 不一致处理
            AuditMessage message = buildAuditMessage();// 构建扭转消息
            mqConsumerService.handleAuditMessage(message);
        }
    }
}
```

图 4.21 审核信息保障实现

4.4 规则应用模块

4.4.1 规则应用模块设计

业务处理模块负责监听订单运单的状态，在已有规则库中进行匹配，执行相应的动作。该模块将原有分散在各个服务中的判断逻辑整合在一起，达到业务逻辑内聚的效果，以此来提高系统的鲁棒性和可靠度。高内聚的业务模块能够更好的应对需求的变更，原有系统中，需求的变更会导致多个服务代码的变更，将逻辑内聚在一起之后，代码的变更数就更少了，更少的变更意味着更低的出错概率，更快的迭代速度。

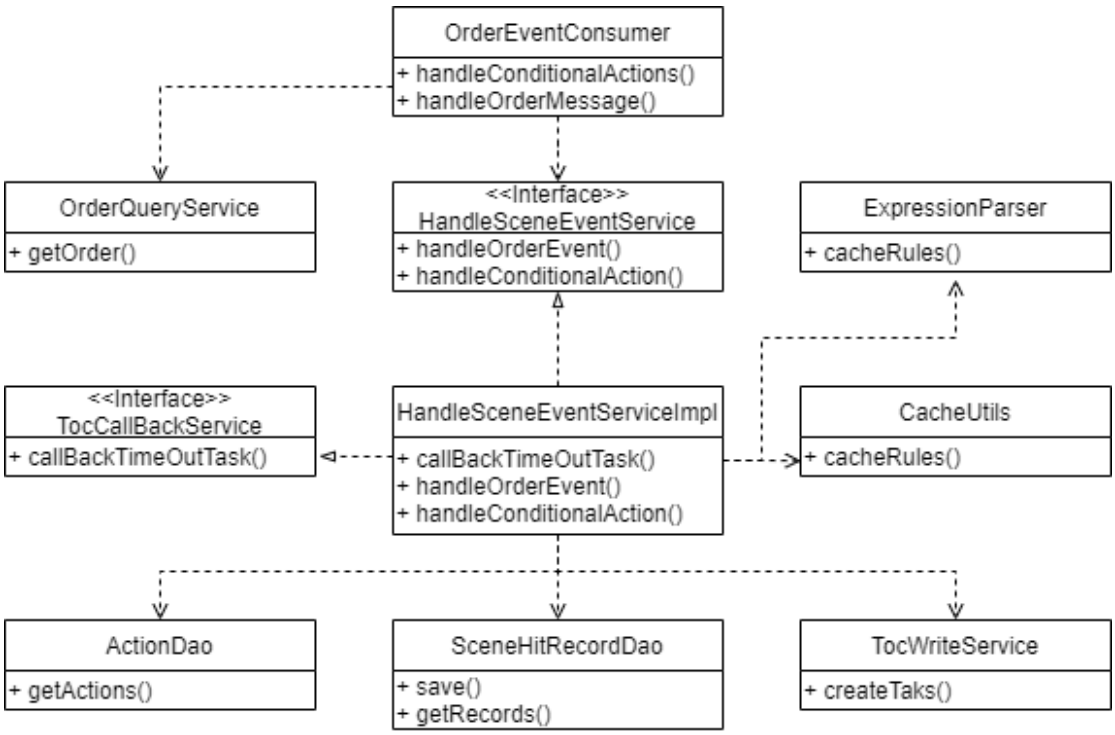

图 4.22 规则应用模块类图

图 4.22 为业务处理模块的类图，OrderEventConsumer 是消息消费类，主要消费订单消息队列和运单消息队列中的消息，当订单状态发生变化的时候，订单系统会给订单消息队列发送一条消息，业务处理模块就会监听到这条消息，并根据消息的类型调用不同的方法，OrderEventConsumer 中主要引用了两个接口，HandleSceneEventService 接口和 OrderQueryService 接口。OrderQueryService 接口是订单服务提供的接口，主要负责查询订单相关的一些

状态，规则的匹配，简单的来说就是状态的匹配，因此在匹配的时候，需要查询订单的一些状态用于判断是否符合条件。HandleSceneEventService 接口为本地接口，主要负责规则的判定与策略的执行，其实现类除了实现 HandleSceneEventService 接口外，还实现了 TocCallBackService 接口，此接口为延时任务中心提供的接口，使用方可以在延时任务中心创建任务，指定任务延时执行的时间，当时间到达之后，延时任务中心就会通过回调的方式，执行相应的任务。规则在判定的过程中，会存在延时判定的需求，此时需要在延时任务中创建延时任务，等到延时任务到达之后，通过回调接口的方式让规则能够继续匹配。

为了减轻数据库压力，同时加快访问速度，规则会加载到 Redis 缓存中，Redis 缓存中的数据来源于数据库，当数据库中的数据发生变化的时，需要及时更新缓存中的数据，Redis 缓存更新策略有很多种，根据读多写少的特点，选择“Cache Aside Pattern”，其具体逻辑如下：当从 cache 中获取数据失败，则从数据库中获取数据，成功后将其放入缓存中；当命中 cache 中的数据，直接返回；当更新数据时，先将数据存入数据库中，再将缓存失效。因此，当数据插入到数据库中的时候，需要将数据更新到缓存中，当数据库中的数据发生更新的时候，需要将缓存中的数据失效。

由于系统部署在两个不同的机房，机房之间需要做数据同步，数据库数据同步 DRC 已经实现，同时 DRC 在同步数据库数据的同时会向消息队列中发送同步消息，缓存可以通过监听消息队列的方式进行同步。A 机房会监听 B 机房是消息队列，B 机房的数据发生变化的时候，就会向 B 机房的消息队列中发送数据变更消息，A 机房监听到消息后，会将消息更新到 A 机房的数据库中，目前的数据源除了数据库之外，还有缓存，如果只更新数据库而不更新缓存，必然会造成数据过时，从而导致产生误判的情况，为了能够对缓存数据进行更新，系统需要监听数据库更新的消息队列，并根据消息队列中的数据对缓存进行更新。

4.4.2 规则判定功能实现

图 4.23 为规则应用时序图。判定规则有两个入口，一个入口为订单、运单消息队列监听入口，另一个入口为延时中心回调入口。消息入口为主要流程入口，

OrderEventConsumer 负责监听订单消息，HandleSceneEventServiceImpl 负责处理 OrderEventConsumer 监听到的消息，根据消息中订单所处阶段获取对应规则，并执行规则，在执行规则过程中，可能存在推迟执行的情况，例如，订单所处状态为商家呼叫订单，规则执行的时间点可能是商家呼叫订单之后 N 秒，针对这种情况，HandleSceneEventServiceImpl 通过 TOCWriteService 在延时任务平台创建延时任务，等到达指定时间时，延时任务平台回调指定方法继续执行。而对于不需要延时执行的规则，HandleSceneEventServiceImpl 能够立即得到执行结果，并根据执行结果选决定是否需要执行对应的操作。延时中心回调入口负责处理到期的定时任务，HandleSceneEventServiceImpl 在执行规则时产生的定时任务均通过回调入口继续处理，处理逻辑与 HandleSceneEventServiceImpl 类似。

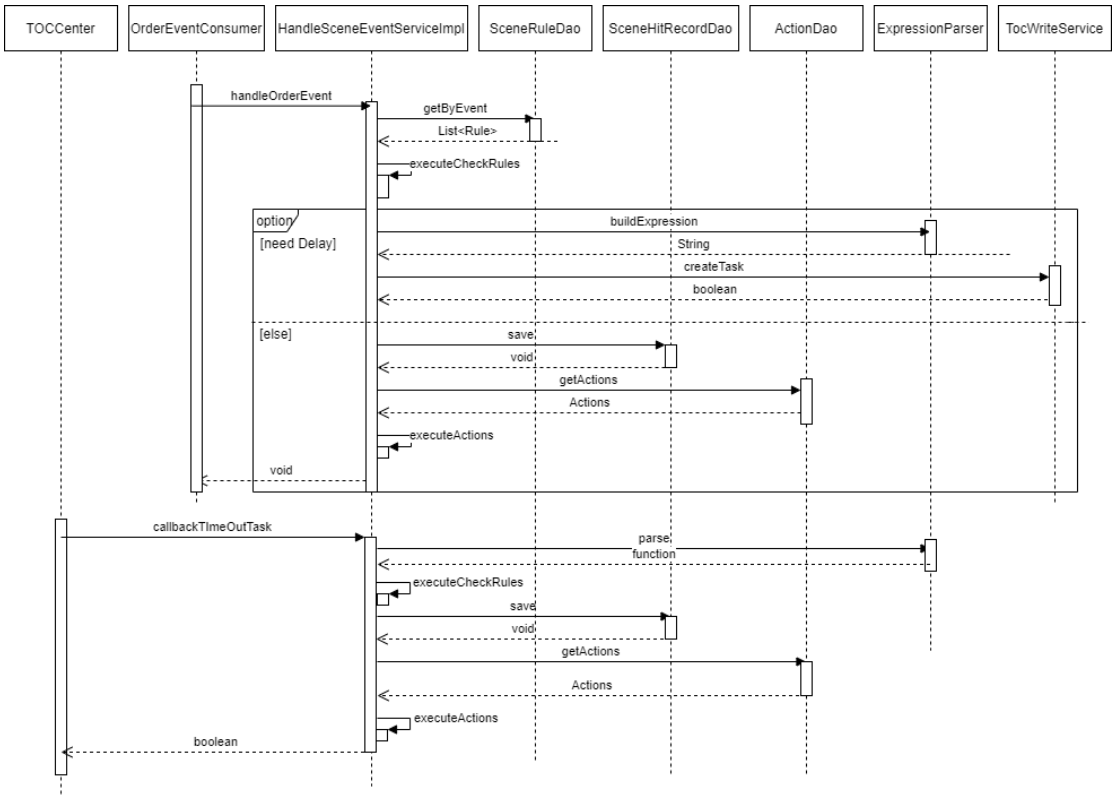

图 4.23 规则应用时序图

图 4.24 为规则应用功能实现代码，handleOrderMessage 方法负责处理订单和运单消息，方法对消息进行解码、构建上下文。handleOrderEvent 方法负责执行规则，如果规则需要延时执行，方法通过异常中断规则执行，捕获异常，并在 catch 语句中调用 tocWriteService 的 createTask 方法在延时任务中心创建

定时任务。callBackTimeOutTask 方法为延时任务中心的回调方法，负责处理定时任务到期之后的逻辑，与 handleOrderEvent 方法类似。

```
@RabbitListener(queues = "xxxxxxx") //监听消息队列
public void handleOrderMessage(OrderMessage orderMessage){
        …// 校验 解码
        // 2. 准备规则规则判定需要的参数
        RuleContext context = …// 构建参数
        // 3. 参数封装完成之后交由 handleSceneEventService 处理
        handleSceneEventService.handleOrderEvent(context);
    }
}
@Override
public void handleOrderEvent(RuleContext context, OrderMessage orderMessage){
        …// 1. 校验参数
        // 获取规则，先从缓存中获取，缓存中没有再从数据库中获取，再加入缓存。
        List<String> rules = getRules(orderMessage.getCurrentNode);
        List<RuleFunction> ruleFunctionList = new ArrayList<>();
        ruleFunctionList = rules.stream().map(buildFunction).collect(...); //json->Obj
        try{
            // 检查规则并执行 Action
            checkRuleAndExecuteAction(ruleFunctionList);
        }catch(TimeOutInterruptException e){
            //延时处理抛异常
            tocWriteService.createTask(.....);
        }
    }
}
@Override
public void callBackTimeOutTask(String ruleText){
        …// 1. 校验参数
        // 2. 将字符串构建成执行树
        RuleFunctions rootFunction = parser.parse(ruleText);
        // 3 检查规则并执行 Action
        checkRuleAndExecuteAction(ruleFunctionList);
}
```

图 4.24 规则应用功能实现

4.4.3 缓存更新功能实现

图 4.25 为缓存更新功能实现图。DrcMessageHandler 监听 DRC 消息队列，并根据监听到的消息将 Redis 中对应缓存移除。下次访问到是再重新放入缓存

中。

```
@RabbitMq(″xxxxxx″)
public void DrcMessageHandler(DrcMessage message){
    // 校验参数
    validate(message);
    Rule rule = (Rule)message.getDate();
    jedis.remove(rule.getName());// 消息更新采用删除方式。
}
```

图 4.25 缓存更新功能实现

4.5 系统测试

系统的测试有两个目标，一方面需要测试系统是否满足需求，另一方面需要找出系统中存在的问题，本章将从测试环境，测试设计以及测试结果三方面进行阐述。

4.5.1 测试环境

测试环境基于内部私有云，表 4.1 为系统测试环境硬件配置，集群使用了 2 台服务器，服务器的硬件配置为每台 8 核 CPU、32G 内存、128G 固态硬盘

表 4.1 硬件和环境配置

硬件项	备注
CPU	8 核
内存	32G
硬盘	1T
数量	两台

表 4.2 为系统测试环境软件配置，每台服务器均安装 kubernetes 用于集群的管理，使用 Docker 运行服务应用，数据库使用 Mysql。

表 4.2 软件配置

软件名称	软件类型
操作系统	Centos7.2
集群管理	Kubernetes
容器	Docker
数据库	Mysql

4.5.2 测试设计

根据第三章的需求分析，本节将对系统的主要功能进行测试，包括规则管理功能、规则预处理功能、审核流功能以及业务处理功能。通过对上述主要功能的测试，确保系统的功能能够正确正常的运行。

表 4.3 为规则操作测试用例，用于验证规则操作功能是否能够正确的运行。测试内容主要包括对规则的新增、删除、修改、查询。

表 4.3 规则操作测试用例

测试 ID	TC1
测试名称	规则操作测试
测试描述	测试规则操作功能，包括规则的新增、删除、修改、查询
测试步骤	1、 用户进入规则操作页面，按需求选择查询条件，点击查询，查看已有规则。 2、 用户点击新增按钮，填写规则详情，点击提交审核按钮 3、 用户点击修改按钮，修改已有规则详情，再点击提交审核按钮 4、 用户点击删除按钮，删除已有规则
预期结果	1、 页面显示制定条件下的规则 2、 页面显示新建规则成功，已发送审核 3、 页面显示修改规则成功，已发送审核

表 4.4 为日志查询测试用例，用于验证是否能够根据指定的条件获取正确的查询结果。

表 4.4 日志查询测试用例

测试 ID	TC2
测试名称	日志查询
测试描述	测试日志查询功能
测试步骤	1、 点击日志控制板 2、 填写日志查询条件 3、 点击查询 4、 比较查询结果与数据库中结果的差异
预期结果	1、 获取到指定条件的数据

表 4.5 为规则过期告警测试用例，用于验证当规则过期之后配置用户是否能够收到规则过期告警邮件。用户在创建规则的时候需要设置过期时间，过期时间快到的时候会收到规则过期告警邮件。

表 4.5 规则过期告警测试用例

测试 ID	TC3
测试名称	规则过期告警测试
测试描述	测试当规则过期之后配置用户是否能够收到规则过期告警邮件
测试步骤	1、 用户创建、修改规则，设置规则的过期时间 2、 用户点击提交审核
预期结果	1、 在规则快过期的前三天里，用户会陆续收到规则过期的告警邮件。

表 4.6 为审核流操作测试用例，验证审核流是否能够按照指定的条件查询，验证用户是否能够正确的终止审核流。

表 4.6 审核流操作测试用例

测试 ID	TC4
测试名称	审核流功能测试
测试描述	测试触发审核流创建、审核流查询、审核流终止
测试步骤	1、 用户新增、修改规则，点击提交审核 2、 用户点击审核流选项卡，填写审核流查询条件，点击查询 3、 点击审核流选项卡，选择已有审核流，点击终止
预期结果	1、 用户新增、修改规则，并触发审核流的创建 2、 用户获取根据指定条件查询的审核流 3、 审核流状态已由审核中状态转变为终止状态 4、 审核平台中审核流已经终止

表 4.7 为审核权转移测试用例，验证审核权是否能够正确的流转。审核人员在审核平台对审核流进行审核之后，审核的信息就会转交到履约超时管理系统中，系统会根据具体情况进行审核权的转移。

表 4.7 审核权转移测试用例

测试 ID	TC5
测试名称	审核权转移测试
测试描述	测试审核权是否能够正确转移
测试步骤	1、 用户进入审核平台 2、 用户点击审核失败 3、 用户点击审核通过
预期结果	1、 用户在审核平台查看到审核未通过 2、 用户在审核平台查看到审核进行中，系统将审核权流转到下一个审核人员 3、 用户在审核平台查看到审核通过，通过的规则会被纳入规则库中

表 4.8 为审核消息保障测试用例，用于验证系统能够在消息丢失的情况下也能保证审核流能够进行下去。

表 4.8 审核消息保障测试用例

测试 ID	TC6
测试名称	审核消息保障测试用例
测试描述	测试是否能够正确的保障审核消息
测试步骤	1、 审核平台审核当前节点 2、 审核平台不发送消息至消息队列 3、 静置三秒时间 4、 比较本系统和审核平台同一个审核流的数据是否一致
预期结果	1. 本系统中的审核流数据与审核平台中的数据能够保持一致

表 4.9 为规则预处理功能的测试用例，用于测试前端配置的规则能否在后端生成相应的执行规则。

表 4.9 规则预处理功能测试用例

测试 ID	TC8
测试名称	规则预处理测试
测试描述	测试前端规则能够转换成后端执行需要的模式，并且能够正确的执行
测试步骤	1. 用户新建规则，并点击提交审核 2. 用户修改规则，并点击提交审核
预期结果	1. 规则转换成功，并且能够正确的执行 2. 规则转换成功，并且能够正确的执行

表 4.10 为规则判定测试用例，用于校验进行中的订单和运单状态是否能够命中指定的规则。测试人员在测试的时候需要对履约过程中的每一个阶段进行测试，在测试之前需要了解每个规则可能存在的过期时间，在测试的时候需要把这个过期的时间特别的空置出来。

表 4.10 规则判定测试用例

测试 ID	TC7
测试名称	规则判定测试
测试描述	测试平台是否能够根据规则进行
测试步骤	1. 测试人员在平台上下单。 2. 测试人员间隔指定时间之后点击商家接单。 3. 测试人员间隔指定时间之后点击呼叫骑手。 4. 测试人员间隔指定时间之后点击骑手接单。 5. 测试人员间隔指定时间之后点击骑手取餐。 6. 测试人员间隔指定时间之后点击骑手配送。 7. 测试人员间隔指定时间之后点击骑手送达。
预期结果	平台能够根据已有的规则判定出每个阶段是否有超时情况的出现，并根据规则中配置的触发行为执行相应的动作。

4.5.3 测试结果

根据上一节的测试用例设计，测试人员按照步骤严格的测试了所有的测试用例，并记录了结果，测试的结果如表 4.11 所示。经过测试，系统各项功能模块均符合预期，测试用例全部通过，系统功能运行正常，能够履约超时管理的需求，达到了系统测试的目的。

表 4.11 系统功能测试结果表

测试用例/ID	测试用例名称	测试结果
TC1	规则操作	通过
TC2	日志查询	通过
TC3	规则过期告警	通过
TC4	审核流操作	通过
TC5	审核权转移	通过
TC6	审核消息保障	通过
TC7	规则判定	通过
TC8	规则预处理功能	通过

4.6 本章小结

本章介绍了系统模块的详细实现。对于每个系统模块，使用了类图和顺序图的方式进行了详细设计了规则管理模块、规则预处理模块、审核流模块、业务处理模块，通过代码实现了每个模块的主要功能。最后对系统进行了测试。

第五章 总结与展望

5.1 总结

本文阐述了饿了么履约超时管理中存在的主要问题以及构建履约超时管理平台的必要性，介绍了国内外发展现状。综述了包括 Spring Boot 框架、RabbitMQ 消息队列、E-Scheduler 任务调度平台、Redis 缓存，异地多活技术等构建系统时采用的技术。在系统分析与设计中，本文说明了系统边界，根据系统边界，详细分析了系统的功能需求和非功能需求。系统总体设计中，本文给出了规则设计，系统架构设计，功能组成设计和系统部署设计。详细分析了系统数据库，给出了系统 ER 图和关键数据表的说明。

在系统实现章节，使用类图、顺序图等方式，对规则管理模块、规则预处理模块、审核流模块、规则应用模块进行了详细的设计，对关键功能给出代码实现，并进行了系统测试。实现的主要功能如下：

- 规则管理模块：给出了规则管理、日志查询的具体实现，并通过定时任务方式给出了过期告警的具体实现。
- 规则预处理模块：根据既有的组织方式，给出了构建检查规则、生效范围、应对策略的具体实现。
- 审核流模块:给出了审核流操作、审核权转移的具体实现，并通过定时检查的方式保证审核消息不丢失。
- 规则应用模块:给出了规则应用、缓存更新的实现细节。

平台已经投入实际使用，根据运营人员在实际使用过程中的反馈，本平台已经较好的解决了运营人员快速履约超时判定规则和把控规则粒度的需求，进一步降低了客诉率，提高了用户满意度。

5.2 工作展望

履约超时管理平台投入生产环境已经有一段时间，在实际使用过程中还会存在修改源代码的需求，例如：新增执行策略、新增判断条件等，虽然这样的需求

不多，但为了尽可能减少修改源代码，平台还存在以下可以改进之处：

1. 提供前端编写、导入脚本能力，减少对源代码的修改，同时需要做好对脚本的校验。
2. 新增订单未命中规则原因查询功能。方便开发人员在遇到线上问题时进行快速定位与排查。

参 考 文 献

[Agarwal,2014] V.Agarwal. System and method for automatically resolving dependencies of Java Archive files for use with Maven. U.S. Patent No. 8,813,031. 19 Aug. 2014.

[Albano et al. 2015] M.Albano, L.L.Ferreira, L.M.Pinho, and A.R.Alkhawaja. Message-oriented middleware for smart grids[J]. *Computer Standards & Interfaces*,2015,38.

[Arthur et al.,2005] J.Arthur and S.Azadegan. Spring Framework for rapid open source J2EE Web Application Development: A case study. *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Network*. IEEE, 2005.

[Atzeni et al.,2020] P.Atzeni,F.Bugiotti,L.Cabibbo,and R.Torlone. Data modeling in the NoSQL world[J]. *Computer Standards & Interfaces*,2020,67.

[Elrad et al.2001] T.Elrad, R.E.Filman,and A.Bader. Aspect-oriented programming.Communication of the ACM 2001, 44(10):29-32.

[Ginanjar et al.2019] A.Ginanjar,and M.Hendayun. Spring Framework Reliability Investigation Against Database Bridging Layer Using Java Platform[J]. *Procedia Computer Science*,2019,161.

[Ji et al.,2014] Z.L.Ji, et al. A distributed redis framework for use in

the ucww. *2014 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*. IEEE, 2014.

[Kamal et al.,2019] S.H.Kamal,H.H.Elazhary,and E.E.Hassanein. A Qualitative Comparison of NoSQL Data Stores[J]. *International Journal of Advanced Computer Science and Applications (IJACSA)*,2019,10.

[Kanagavelu et al.,2013] R.Kanagavelu, et al. Software defined network based adaptive routing for data replication in data centers. *19th IEEE International Conference on Networks (ICON). IEEE*, 2013.

[Kiyoshi,2006] Kiyoshi.Y. An Aspect-Oriented Approach to Modular Behavioral Specification[J]. *Electronic Notes in Theoretical Computer Science*,2006,163(1).

[Kraska et al.,2013] T.Kraska et al. MDCC: Multi-data center consistency. *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013.

[Laddad,2009] R.Laddad. *Aspectj in action: enterprise AOP with spring applications*. Manning Publications Co., 2009.

[Lerner,2010] R.M.Lerner, At the forge: Redis. *Linux Journal* 2010.197 (2010): 5.

[Lonescu et al.,2015] Lonescu and V.Manuel. The analysis of the performance of RabbitMQ and ActiveMQ. *2015 14th RoEduNet International Conference-Networking in Education and Research (RoEduNet NER)*. IEEE, 2015.

[Monty et al.,2018] D.S.Monty and R.Cristina. Creative economy: how the interface of Uber eats and iFood could change

your menu. *Brazilian Journal of Operations & Production Management* 15.3 (2018): 413-419.

[Murudi et al.,2016] V.Murudi, K.M.Kumar, and D.S.Kumar. Multi data center cloud cluster federation-major challenges & emerging solutions. *2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*. IEEE, 2016.

[Naseh et al.,2010] Z.Naseh and V.Gundi. Application based active-active data center network using route health injection and IGP. U.S. Patent No. 7,769,886. 3 Aug. 2010.

[Paksula,2010] M.Paksula. Persisting objects in redis key-value database. *University of Helsinki, Department of Computer Science* (2010).

[Rostanski et al.,2014] M.Rostanski, K.Grochla, and A.Seman. Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ. *2014 federated conference on computer science and information systems*. IEEE, 2014.

[Sachs et al.,2009] K.Sachs,S.Kounev,J.Bacon,and A.Buchmann. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark[J]. *Performance Evaluation*,2009,66(8).

[Sachs et al.,2013] K.Sachs,S.Kounev,and A.Buchmann. Performance modeling and analysis of message-oriented event-driven systems[J]. *Software & Systems Modeling*,2013,12(4).

[Sturm et al.,2019] C.Sturm,J.Scalanczi,S.Schönig,and S.Jablonski. A Blockchain-based and resource-aware process

execution engine[J]. *Future Generation Computer Systems*,2019,100.

[Testa,2004] S.Testa and W.Chou. "The distributed data center: front-end solutions." *IT professional* 6.3 (2004): 26-32.

[Xin et al.,2018] G.Xin and Huang Zhen. Redis cluster application in tram onboard system. *Modern Urban Transit* 2 (2018): 7.

[Xu et al., 2014] M.Xu,X.W.Xu,J.Xu,Y.Z.Ren,H.Zhang,and N.Zheng.A Forensic Analysis Method for Redis Database based on RDB and AOF File[J]. *Journal of Computers*,2014,9(11).

[快递服务标准,2008] http://www.cssn.net.cn/cssn/front/listpage.jsp，快递服务标准，中国标准网，2008.

[凌质亿,2013] 凌质亿,刘哲星,曹蕾.高并发环境下 Apache 与 Nginx 的 I/O 性能比较.*计算机系统应用*,2013,22(06):204-208.

[吴璨等，2019] 吴璨,王小宁,肖海力,曹荣强,赵一宁,迟学斌.分布式消息系统研究综述[J].*计算机科学*,2019,46(S1):1-5+34.

[徐震等,2018] 徐震,焦文彬.RabbitMQ 小消息确认机制优化[J].*计算机系统应用*,2018,27(03):252-257.

[周强明,2017] 周强明.*京东物流:京东履约引擎——供应链管理标准化实践*.中国供应链发展报告,2017(00):248-253.