****



**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** | 金思晔 |
| **学科、专业名称** | 工程硕士(软件工程方向) |
| **研究方向** | 软件工程 |
| **指导教师** | 王金庆 高级工程师 |

**2019 年 4 月 日**

**学 号： MF1832073**

**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:** | **金思晔** |
| 指导教师: | **王金庆　高级工程师** |

|  |
| --- |
| **南京大学研究生毕业论文** |
| **(申请工程硕士学位)** |

|  |
| --- |
| **南京大学软件学院** |
| **2019年4月** |

**The Design and Implementation of**

**Performance Overtime Management**

**for ELEME**

**Jin, Si Ye**

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

Supervised by

Senior Engineer **Wang, Jinqing**

Software Institute

**NANJING UNIVERSITY**

Nanjing, China

April, 2019

# 摘 要

# Abstract

# 图目录

# 表目录

目录

[摘 要 V](#_Toc25947911)

[Abstract VI](#_Toc25947912)

[第一章 引言 8](#_Toc25947913)

[1.1 项目背景 8](#_Toc25947914)

[1.2 国内外发展现状 9](#_Toc25947915)

[1.3 本文主要工作 9](#_Toc25947916)

[1.4 本文的组织结构 9](#_Toc25947917)

[第二章 技术综述 9](#_Toc25947918)

[2.1 SpringBoot 9](#_Toc25947919)

[2.2 MaxQ 9](#_Toc25947920)

[2.3 Huskar 10](#_Toc25947921)

[2.4 WorkFlow 10](#_Toc25947922)

[2.5 Redis 10](#_Toc25947923)

[2.6 本章小结 10](#_Toc25947924)

[第三章 系统分析与设计 11](#_Toc25947925)

[3.1 系统总体规划 11](#_Toc25947926)

[3.2 系统需求分析 11](#_Toc25947927)

[3.2.1 规则管理模块 11](#_Toc25947928)

[3.2.2 审核流模块 12](#_Toc25947929)

[3.2.3 业务处理模块 13](#_Toc25947930)

[3.3 非功能性需求 15](#_Toc25947931)

[3.5 数据库设计 15](#_Toc25947932)

[3.6 本章小结 15](#_Toc25947933)

[第四章 系统实现 15](#_Toc25947934)

[4.1 规则配置模块 15](#_Toc25947935)

[4.2 审核流模块 16](#_Toc25947936)

[4.3 业务处理模块 17](#_Toc25947937)

[4.5 系统测试 17](#_Toc25947938)

[4.5.1 测试环境 17](#_Toc25947939)

[4.5.2 测试设计 17](#_Toc25947940)

[4.5.3 测试结果 17](#_Toc25947941)

[第五章 总结与展望 17](#_Toc25947942)

[5.1 总结 17](#_Toc25947943)

[5.2 工作展望 17](#_Toc25947944)

[参 考 文 献 18](#_Toc25947945)

[致 谢 18](#_Toc25947946)

# 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。

。和平台

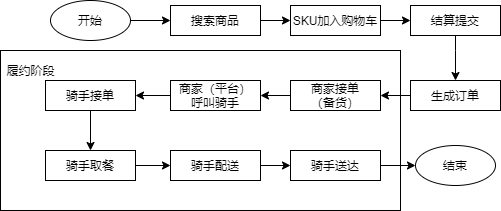


图1.1 外卖业务履约过程（建议：方框内未本文内容，线条、方框采用实线；框外是本文关联内容，虚线）

图1.1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五个履约节点，经常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，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：

一是不仅工作量巨大，而且容易出错。

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，嵌套于代码。如果需要修改规则，需要产品（？）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。

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，同时需要建立配置平台，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。

## 相关研究与应用现状这一节写的不好。（1）逻辑不太顺，（2）缺少国内外现状描述，能否补充同行、同业类似问题如何解决、解决状况，有没有什么问题？

在延时任务方面（很突兀），目前有不少实现方式，具体如下：

1. 通过redis实现。redis中有一个数据结构称为有序集合（sorted key）,集合中的数据可以根据插入值的score进行排序。将延时任务加入Sorted Set中，将延迟时间设置为score，同时启动一个线程不断的去判断Sorted Set中第一个元素的score是否大于当前时间，如果大于，从Sorted Set中移除任务并添加到执行队列中，如果小于，则休眠指定时间之后再重试。
2. 通过kafka实现，kafka通过时间轮（TimingWheel）的方式实现了消息的延时发送。时间轮由多个时间格组成，每个时间格代表当前时间轮的基本时间跨度（tickMs）。时间轮的时间格个数是固定的，可用wheelSize来表示，那么整个时间轮的总体时间跨度（interval）可以通过公式 tickMs × wheelSize计算得出。时间轮还有一个表盘指针（currentTime），用来表示时间轮当前所处的时间，currentTime是tickMs的整数倍。currentTime可以将整个时间轮划分为到期部分和未到期部分，currentTime当前指向的时间格也属于到期部分，表示刚好到期，需要处理此时间格所对应的TimerTaskList的所有任务。
3. 通过RabbitMq实现。RabbitMq可以设置消息的存活时间（Time To Live），当消息存活的时间超过这个时间时，消息就死亡了，被称为死信。RabbitMq还有一种交换器称为DLX（dead-letter-exchange）,如果一个消息在消息队列中变成死信，那么它能够被重新发送到DLX中，然后DLX再将其路由到另一个队列中，如果监听该消息队列，那么就能够获取到到期的消息，并做下一步处理。

在面对频繁的业务需求变更的时候，将业务逻辑与程序的其他代码分离是一种可靠的应对措施，具体的实现方式有如下两种：

第一种是借助规则引擎来实现，Drools是一款开源的通用规则引擎，使用的是RETE算法，RETE算法是一种前向规则快速匹配算法，其匹配速度与规则数目无关。Rete算法通过形成一个rete网络进行模式匹配，利用基于规则的系统的两个特征，即时间冗余性和结构相似性，提高系统模式匹配效率。drools有自己的语言，同时有自己的以drl结尾的文件格式，用户可以使用drools的语言，在drl文件中编写业务逻辑规则，通过修改drl文件的方式来修改业务逻辑。drools也存在如下的几个问题：规则的配置不够友好，对于不会编程的使用者来说，编写drl文件是一个较为困难的事情，上手难度较高，而且不能保证语法和逻辑的正确性；drools的处理速度相较于原生程序代码慢很多，在调研过程中，使用drools和原生代码开发同一段业务逻辑，drools需要的时间比原生代码高出一个数量级，在应对时效性强的需求的时候，drools的表现不够理想。drools更适用于逻辑判断复杂，时效性没那么高的场景。

第二种可以通过构建执行树的方式实现，阿里巴巴零售通的营销平台使用的就是这种方式。营销平台需要进行优惠计算，平台实现了一些基本计算单元，例如逻辑与、逻辑或、逻辑非、算数加、算数乘等，同时实现了一些条件权益计算单元，例如amountAtCondition（满条件）、FreePostageRights（包邮权益）等，以树的方式进行组织，对树做后序遍历，即可以判定是否满足条件。这种方式适用于整体判断逻辑变动不大，但具体某一个节点的判断逻辑存在变化的情况。

在规则的可配置化方面，实现方式有如下两种：

第一种是通过表达式或者JSON的方式进行配置。饿了么告警平台是一个用来监视系统各项指标是否正常的一个平台，由于告警判断条件和告警阈值需要经常变更，因此系统将告警条件设计成可以通过页面编写表达式的方式进行配置，例如 “${rpcTime} > threshold” 代表的含义就是rpc调用时间超过阈值的时候，就会进行告警。阿里云的Link Develop平台是阿里云针对物联网领域提供端到端的一站式开发平台，用户在平台上开发完成之后就可以推送到阿里云商城上进行销售。平台提供了规则引擎服务，能够给开发者提供规则管理、规则定义和规则运行的环境，以满足不同情况下的自动化需求，例如配置触发和联动规则，实现设备间的互联互通和消息推送等。规则采用了JSON的方式进行配置，包括规则的输入节点（如设备上报的相关数据）、处理节点（如数据的临界值）、输出节点（如触发指定设备的某个服务调用），一条完整的规则必须包含以上的三个节点。这种方式的优点在于规则的执行逻辑具有高灵活性，用户配置的执行逻辑理论上是不受限制的。而缺点也很明显，新手上手难度较大，且容易出错，配置的时候需要深入了解可编辑配置项，配置完成之后无法做到合理性的检查。

第二种是通过图形化的形式进行配置。该方式将每一个可配置的条件选项化，用户可以在界面上通过点击按钮的方式进行配置。这种配置方式对使用者来说很友好，不需要手动编写判断条件和执行逻辑，同时降低了出错了可能性，缺点就是灵活性受到了限制。

在结合系统使用人员、项目体量以及灵活性等背景，执行树与图形化配置的结合是一个较好的选择。

## 1.3 本文主要工作写的不太好，没有突出主要问题及解决方法

梳理原有系统的业务逻辑以及原有系统之间的依赖关系，分析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，给出响应时间以及高可用需求。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、实现规则配置的主要流程以及选项；对于流程管理模块，设计、实现流程管理以及通过定时任务保证不丢失流程消息；对于规则预处理模块，设计、实现规则定义，以及构建执行树；对于业务处理模块，设计、实现系统的异地多活以及缓存的更新策略。

## 本文组织结构（为什么要那么多“的”呢，这是严重口语）

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、RabbitMq、WrokFlow、Redis。

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。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，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。

## 2.2 RabbitMq

RabbitMq是基于AMQP协议开发的一款开源的消息中间件，AMQP协议又称为高级消息队列协议，是应用层协议的一个开放标准。AMQP的主要特征是面向消息、队列、路由、可靠性、安全。RabbitMq的服务端使用Erlang语言编写，客户端支持Python、Ruby、.Net、Java等语言。

RabbitMq包含以下基本组件：

1. Producer：消息生产者，负责将消息发送到消息队列中
2. Consumer：消息消费者，负责从消息队列中接受消息
3. Message：服务器和应用程序之间传递是数据，由properties和Body组成
4. Exchange：交换机，接受消息，并根据消息将其发送到对应的绑定消息队列上
5. Queue：又称为Message Queue，保存Exchange发过来的消息，并将其发送给消费者。

消息生产者连接到RabbitMQ服务器上，指定对应的Exchange名称，之后消息生产者向exchange中发送消息，exchange接收到消息后，根据routing key路由到跟当前交换机绑定的相匹配的队列里面，消费者监听接收到消息之后开始业务处理，然后发送一个ack确认告知消息已经被消费，RabbitMQ服务器收到ack之后将对应的消息从消息队列里面是删掉。

## 2.3 WorkFlow这个是比较基础的，你似乎用到了公司自有的工作流平台，如“审核流平台是公司已有的一个基础平台”，如果是，还是介绍一下公司的工作流平台，因为你是直接用这个平台

工作流本质是一个状态机，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，到一个中介状态结束，状态直接使用JSON传递数据。状态非为多种不同的类型，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

工作流系统主要有三个部分组成：控制台、触发器和工作流引擎。控制台用于管理工作流的元数据、触发器以及一些配置信息，同时也是http流量的入口；触发器用于定时触发和监听队列触发工作流的执行；工作流引擎是整个工作流执行的核心，根据预定义的工作流，使用数据驱动状态的流转。

## 2.5 Redis

Redis是一个高效的key-value存储系统，与传统关系型数据库不同的是，它以键值对的方式存储数据，支持存储的value类型相对较多，包括String类型，list类型，set类型、zset类型和hash类型，这些数据类型都支持push/pop、add/remove及取交集、并集和差集等丰富的操作，而这些操作都是原子性的。为了保证Redis的高效，Redis中的数据都是存放在内存中，同时，Redis通过nio的方式实现了单线程处理请求，因此Redis有如下的应用场景：

* 缓存：缓存是Redis最重要的一个应用场景，合理的利用缓存，不仅能够提升网站访问的速度，还能大大降低数据库的压力。
* 排行榜：Redis提供的zset数据类型能够实现各种复杂的排行榜应用
* 分布式锁：分布式技术是当前互联网常用的技术，分布式技术面临的问题是对同一个资源的并发访问，当并发量不大的时候，可以通过数据库的悲观锁、乐观锁来实现，但是在并发量高的情况下，利用数据库锁来控制资源的并发访问是不理想的，因此需要使用Redis来实现一个分布式锁，进行更高效的访问。
* 分布式会话：在集群模式下，session的管理独立于应用容器，一般会搭建以Redis为中心的session服务。

## 2.6 本章小结

##

# 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模块中的一个相对独立的系统（注意平台、模块，系统的关系逻辑），负责制定超时规则以及判定订单、运单是否命中超时规则。

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。

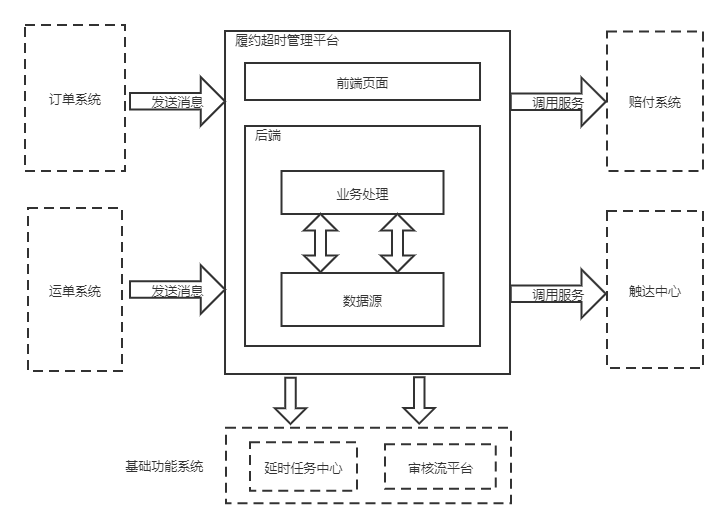


图3.1 系统边界箭头代表信息流向，注意箭头的方向，以及是否应该为双向箭头。（图中似乎有错误）

## 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：

规则创建：用户可以在履约超时管理平台上编辑进行规则的编辑，完成编辑之后，平台会将编辑结果提交审核，审核通过的规则会被纳入规则库中。

状态检查：平台在监听到订单或者运单状态变更消息之后，需要使用已有的规则进行检查，判断其是否发生超时情况。

策略执行：订单命中履约超时规则之后，平台会获取配置好的应对策略，并执行。

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，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. 提供规则审核流管理的能力，运营人员能够管理规则，包括查找、终止审核中的规则，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。

补充一下涉及的用户对象

## 3.3 需求分析

### 3.3.1 规则管理

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3.1所示。

表3.1 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. 点击新建规则   2. 根据页面中的选项进行配置   3. 点击提交审核   4. 点击修改规则   5. 根据页面中的选项进行配置   6. 点击提交审核   3.1 点击删除规则  3.2 弹出删除警告框，选择确认或取消  4.1 根据需求选择查询条件  4.2 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。用户在新增和修改规则的时候，才有可能改动规则的日期，因此过期告警需要用户修改规则日期之后才能触发。在触发之后，系统需要设置定时任务用以定时发送告警邮件，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3.2。

表3.2 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，对于查询到的数据，需要提供导出为excel的功能。除此之外，运营人员想想更为直观的了解规则命中排名、命中时间段等信息，因此需要将日志按照指定的要求以图像化的方式展示在前端。开发人员需要查询某一笔订单对应的命中记录，来应对软件中可能存在的不足。日志查询用例描述如表3.3所示。

表3.3 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 6. 点击导出结果，将结果以excel的形式导出 | | |
| 拓展流程 | 无 | | |

### 规则预处理功能

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、构建触发行为。



图3.3 规则预处理用例图

规则预处理功能的主要目的是将不可执行的规则转化成可以执行的规则（什么意思？）。运营人员配置完规则，点击提交之后，前端会将数据（规则？）封装在VO对象当中，传送到服务端，此时的规则对象仅仅保存数据，没有保存规则之间的先后依赖关系,并且不具备可执行性。规则之间的依赖关系和执行顺序是已经确定好的，其依赖关系体现在前端配置页面上，运营人员无法改变这种关系，若有新的判断条件、新的应对策略的出现，依赖关系才有可能发生变化，但这样的需求非常少。后端在接受到规则数据之后，将数据以既定的依赖关系进行组织，组织的方式是通过树来实现，规则与规则之间先后依赖关系可以通过树的层级关系来体现，规则的执行顺序可以通过树的中序遍历或后序遍历来实现。

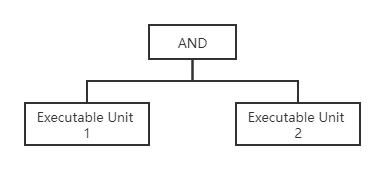


图3.4 规则树化示意图

图3.4为规则树化示意图，图中根节点为AND执行单元，需要两个参数的输入，左子树和右子树（是否需要说明一下如何构建树，以及树可能的深度？）均为可执行单元，执行结果作为参数传给AND执行单元，该执行单元的逻辑是执行两个参数的逻辑与操作。AND执行单元的执行结果可以作为最终结果进行输出，也可以作为执行单元的参数。由于同一层级的叶子节点之间没有依赖关系，因此可以选择中序遍历或者后序遍历来执行。

构建检查规则子功能负责将用户配置的检查规则按照指定的逻辑进行树化，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。检查规则构建用例描述如表3.4所示。

表 3.4 检查规则构建用例图

|  |  |  |  |
| --- | --- | --- | --- |
| 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，检查规则可以设定生效范围，触发行为也可以设置生效范围，这样设计能够更为细粒度的把控规则。生效范围构建用例描述如表3.5所示。

表3.5 生效范围构建用例图

|  |  |  |  |
| --- | --- | --- | --- |
| 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 | | |

构建触发行为子模块负责将用户配置的触发行为以指定的方式进行树化，与其他两者不同的是，触发行为并没有先后顺序，触发行为甚至可以并行执行，但是每个行为都有具体的执行范围，因此一个行为和执行范围组合成执行单元。触发行为构建用例描述如表3.6所示。

表3.6 触发行为构建用例图

|  |  |  |  |
| --- | --- | --- | --- |
| ID | UC6 | 名称 | 构建触发行为 |
| 描述 | 根据用户配置的触发行为构建触发行为执行树 | | |
| 参与者 | 系统 | | |
| 前置条件 | 用户创建/修改规则 | | |
| 后置条件 | 生成触发行为执行树，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. 后端获取配置的触发行为 3. 后端根据获取每个触发行为的生效范围执行树 4. 构建触发行为执行树 | | |
| 拓展流程 | 无 | | |

这种空行都删除！

### 审核流程管理功能

图3.5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。

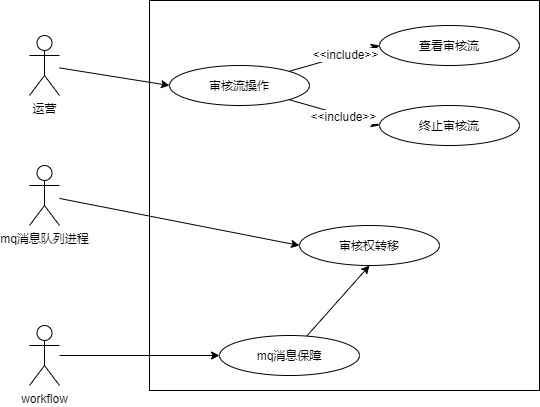


图3.5 审核流程用例图

运营人员点击提交审核之后，需要了解规则审核的进度，同时想要终止审核不需要的规则，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，用户可以根据具体条件进行查询，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3.7所示。

表3.7 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. 用户进入审核流界面   2. 根据需求选择查询条件   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 | | |

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，也就是说，审核流平台能够设置一条审核流上有几个审核节点，但是审核节点具体内容需要自行设定。因此，系统接入审核流平台之后，就存在这样的需求，系统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3.6审核流流程图，从图中可知，如果消息为审核通过，那么系统将会审核权移交到下一位审核人手上，否则，会记录审核失败。审核权流转用例描述如表3.8所示。

表3.8 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 | | |



图3.6 审核流流程图

虽然明白你以下这段意思，但写的有点绕。消息保障子功能负责解决消息丢失的情况，系统需要监听审核流平台的消息队列，来决定下一个审核节点。消息丢失的可能原有如下几种：生产者丢失数据、mq（MQ一般都大写，全文统一）丢失数据、消费者弄丢数据。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丢失数据，例如MQ Server？突然宕机，可以通过mq持久化的方式解决问题。如果是消费方丢失数据，那么可以启用ACK机制，只有消息处理完之后给mq发送ACK消息进行确认。但在公司的使用情况中，作为消费者，系统并不能保证生产者和mq不丢失数据，即使生产者和mq做了消息丢失的兜底，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丢失数据的情况做兜底。为此，需要在workflow上配置定时任务，由于审核流平台会将审核结果记录在本地数据库中，而履约超时管理平台业务记录审核状态，因此定时任务需要在固定间隔时间内比较审核流平台数据库和履约超时管理平台数据库中审核流信息，并以审核流平台的信息为标准，更新履约超时管理平台数据库。审核消息保障用例描述如表3.9所示。

表3.9 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、workflow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. mq由于种种原因丢失数据，审核流卡住 3. workflow上执行定时任务，定时调用履约超时管理平台提供的接口 4. 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 | | |

### 业务处理（叫这个名字是否合适？实际上，你想描述的是规则如何运行）功能

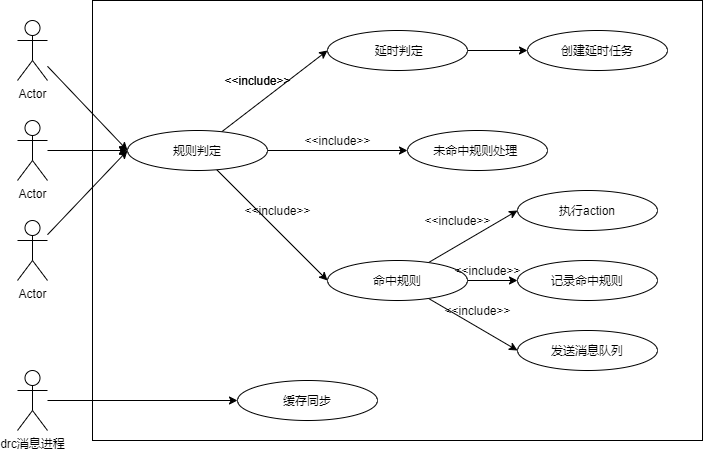


图3.7 业务处理用例图为啥要有三个Actor？

图3.7为业务处理功能的用例图，业务处理功能分为规则判定和缓存同步两个子功能。



图3.8 规则判定流程图

以下这段，好好理理逻辑顺序关系。规则判定子功能负责检查每个阶段的订单和运单是否出现超时情况，并执行相应的策略来应对超时场景。开发履约超时管理平台的一个主要目的就是将分散在各个系统中的判断逻辑整合到一个平台中，从而能够更快的应对变更，因此平台有规则判定的需求。图3.8为规则判定流程图，订单和运单系统会向消息队列中发送状态变更消息，平台监听到消息之后，需要判断规则库中是否有对应的规则，如果有对应的规则，还需要判断此条规则是否需要延时判断，对于需要延时执行的规则，平台需要在延时任务中心创建规则，对于不需要延时执行或者到达延时时间的规则，平台需要将当前订单的状态带入规则中进行判定，如果命中规则，需要接着执行后序的应对策略，如果没有命中规则，需要记录命中失败。规则判定的用例描述如表3.10所示。

表3.10 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，而drc消息就是用来进行异地机房数据同步的消息。以A机房和B机房为例，A机房有一个drc消息队列，当A机房产生数据的时候，机房就会向drc消息队列中发送消息，同时A机房还监听着B机房的drc消息队列，用于同步消息，B机房同理。由于地域和手机号码等因素的影响，用户在A机房产生了数据，A机房就会给B机房的消息队列发送一条数据库数据同步消息，B机房接收到消息后，将B机房的数据库数据同步为A机房数据库数据，从而保证数据库数据的最终一致性，B机房同理。机房之间数据库数据的同步已经由公司的基础平台实现，因此不需要过多的去关心，但是缓存的同步并没有实现，为了加快平台的访问速度，同时减轻数据库的压力，平台使用redis做数据缓存，由于服务需要异地机房进行部署，因此需要实现异地机房服务之间缓存同步功能。缓存同步用例描述如表3.11所示。

表3.11 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 | | |

## 系统总体设计

### 3.3.1 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集合下进行配置。

规则主要分为三大部分，超时判定规则、规则生效范围和规则触发动作。

表3.13为触发规则，在上述规则中，每一个规则对应着一个最小执行单元，超时判定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 超时检查

|  |  |
| --- | --- |
| 规则名 | 可选值 |
| 订单类型在指定范围 | 及时单 |
| 预定单 |
| 全部 |
| 主场景在指定范围 | 提交订单 |
| 实际用户T |
| 预计时长 |
| 用户付款 |
| 商家接单 |
| 商家呼单 |
| 骑手接单 |
| 骑手取餐 |
| 骑手送达 |
| 业务阶段在指定范围 | 提交订单 |
| 实际用户T |
| 预计时长 |
| 用户付款 |
| 商家接单 |
| 商家呼单 |
| 骑手接单 |
| 骑手取餐 |
| 骑手送达 |
| 未发生事件 | 商家接单 |
| 商家呼单 |
| 骑手接单 |
| 骑手取餐 |
| 骑手送达 |
| 时间比较符号 | + |
| - |
| > |
| < |
| = |

表3.14为规则生效范围，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，生效范围可以与超时检查一起使用，也可以和Action一起使用，因此需要独立出来设计。

表3.14 生效范围表

|  |  |
| --- | --- |
| 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为触发行为表，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 触发行为表

|  |  |  |
| --- | --- | --- |
| 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 |

运单消息如表3.18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处于那个阶段。

表3.18 运单消息体表

|  |  |  |
| --- | --- | --- |
| 字段名 | 类型 | 描述 |
| Type | String | 消息代表的含义 |
| Description | String | 具体说明 |
| 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 |

超时消息表如表3.19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3.19 超时消息体表

|  |  |  |
| --- | --- | --- |
| 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 |

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 |

### 工作原理不是非要不可

### 架构设计

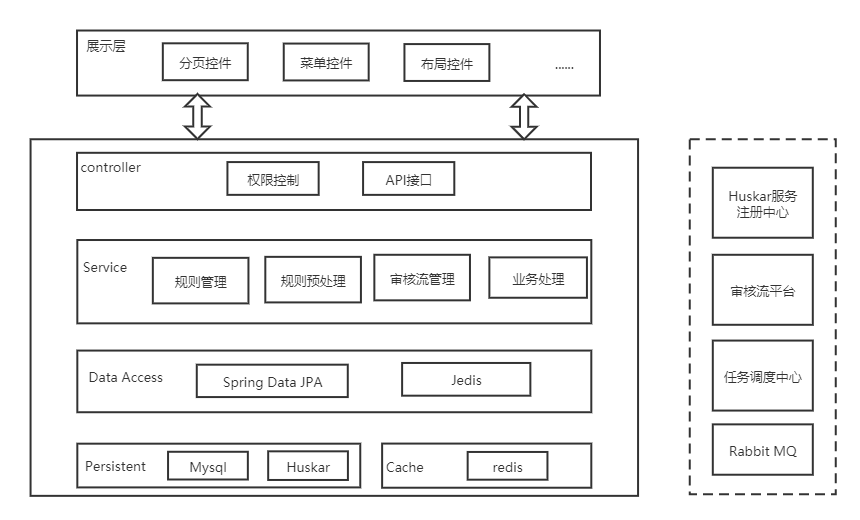


图3.9 架构设计

图3.9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层和Persistent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。

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、轻量级的redis 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，内部包含了关系型数据库MySql和非关系型数据库Redis。Mysql是主流的数据存储组件，需要长期存储的数据可以存放在Mysql中；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。

图的右边为系统需要依赖的组件，系统发布到线上之后，就会在Huskar平台上进行服务的注册，同时，平台会将服务暴露，以便其他服务引用；审核流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与其他系统进行交互。

### 功能组成设计

根据3.2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。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3.10为系统功能模块图。

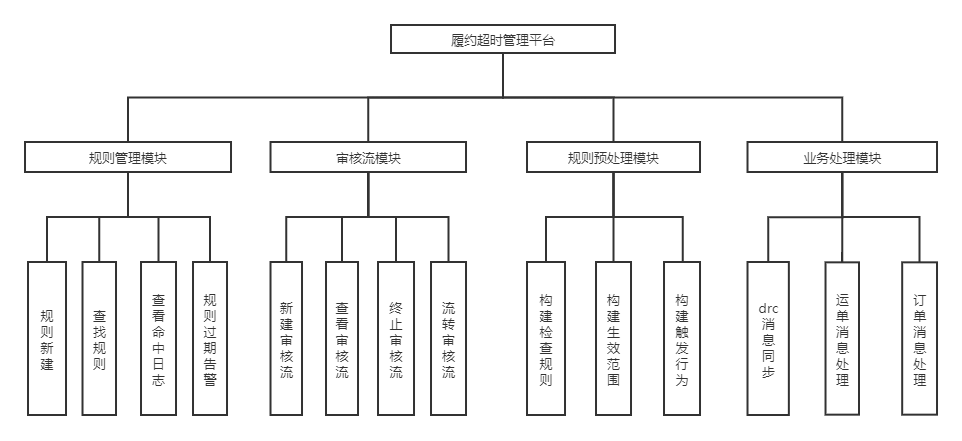


图3.10 系统功能模块图

### 部署设计

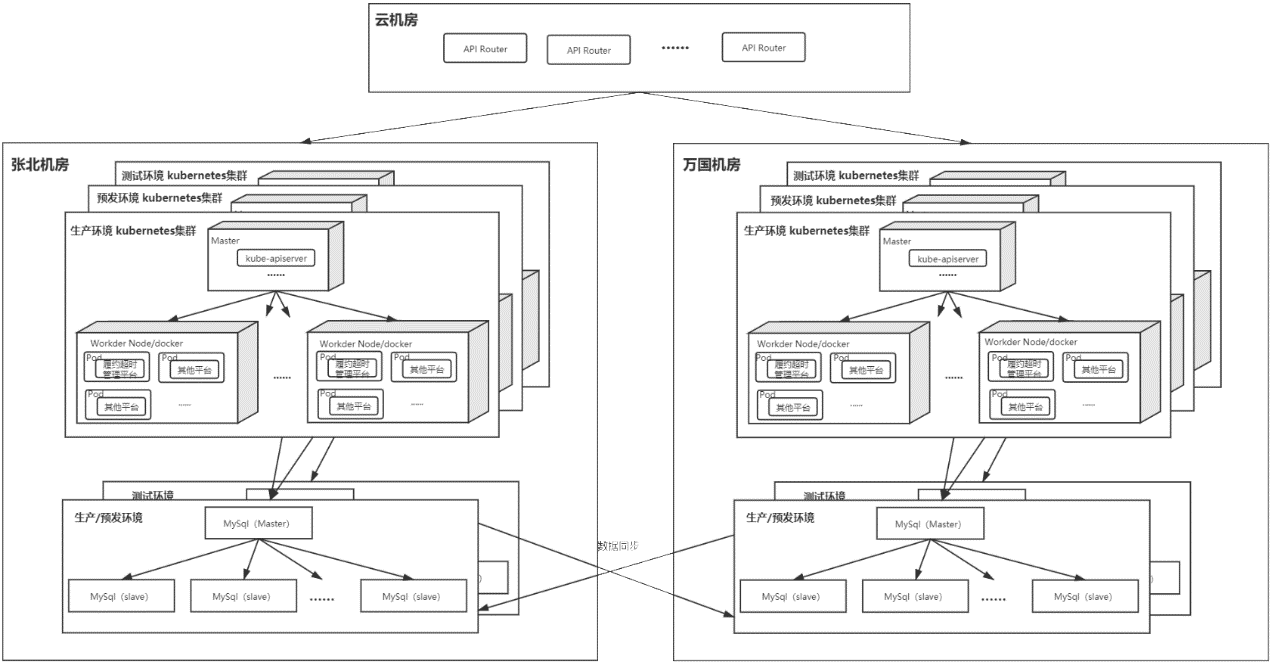


图3.11 部署设计

图3.11为系统部署图，如图所示，应用部署在kubernetes集群中的worker节点上，Pod是kubernetes中最小的部署单元，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的形式存在和管理。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。

同时，应用采用多机房异地方式进行部署，异地多机房又称为异地多活。随着业务的发展，单个数据中心已经无法支撑饿了么的体量，机房的主机数已经无法再增加，另一方面，整机房级别的事故时有发生，每次都会带来严重的后果，为了解决上述两个问题，饿了么所有应用均采用多机房的方式进行部署。多机房部署能够获得高扩展性和高可用，但在解决问题的同时也带来新的问题，最为突出的有如下两点：业务跨机房处理带来的网络高延时问题和数据同步问题。网络延时问题出现在一个请求依赖两个机房的服务，或者一个机房的服务依赖另一个机房的服务，解决方法是让业务内聚，即一个订单的旅单过程在同一个机房内完成，不允许跨机房调用，包含了所有的服务。异地机房的出现，导致了数据同步的问题，为了保证高可用，每个机房需要全量保存用户数据，因此机房之间需要数据的同步，数据的同步通过消息队列的方式，每个机房监听对应机房的消息队列，当数据的更新就会向消息队列中发送消息。

## 数据库设计

本节对数据库系统数据库进行设计。图3.12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。



图3.12 实体关系图

数据库表字段设计如下：

（1）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 |

（2）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3.22所示。

表3.22 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 |

（3）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 |

（4） 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3.24所示。

表3.24 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 |

（5）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 |

（6）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 |

（7）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 |

## 3.5 本章小结

##

# 第四章 系统实现

## 4.1 规则管理模块此节描述不太清楚，尤其是彼此关系

### 4.1.1 详细设计

规则管理模块的功能主要有对已有规则的新增、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。

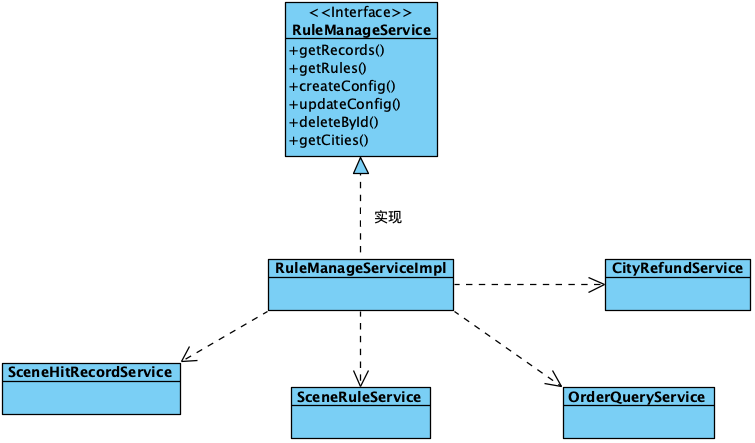


图4.1 规则管理类图除RuleManageService类外，其他类要详细一点

图4.1为规则管理的设计类图，RuleManageServiceImpl负责规则的增删改查以及命中情况的查询。为了实现规则的操纵，RuleMangeServiceImpl类主要通过调用SecneRuleService实现对象当中的方法；为了实现规则命中日志的查询，RuleMangeServiceImpl通过调用SceneHitRecordService实现类当中的方法。除此之外，前端展示的数据，例如节点的可选类型，全国城市信息等通过调用CityRefundService实现类的方法来实现。

### 4.1.2 规则操作功能实现此节和4.1.1什么关系？应该是对应需求分析3.3.1中的运营人员的规则管理吧

建议补充顺序图，说明操作执行关系，后同。

以新增规则为例，其实现代码图如下所示。方法createConfig需要传入一个RuleContent对象，该对象中保存用户创建的规则，方法拿到规则后，会对规则进行校验，以确保规则符合要求。校验通过之后调用SceneRuleService的create方法，并根据返回参数判断是否创建成功。而在SceneRuleService的create方法中，需要做如下几件事情：将规则转换成后端能够执行的执行树，创建规则审核流以及在数据库中保存规则的草稿。

|  |
| --- |
| class RuleManageServiceImpl {  private static int DEFAULT\_LIMIT = 10;  public Boolean createConfig(RuleContent ruleContent) {  // 1. 校验对象是否符合要求  validate(ruleContent);  // 2. 判断规则是否已经存在  if (sceneRuleService.cantians(ruleContent)) {  throw new Exception(" 此规则已经存在，无法添加！ ");  }  // 3. 创建规则  boolean isSuccess = sceneRuleService.create(ruleContent);  if (!isSuccess) {  throw new Exception(" 插入规则失败！");  }  return isSuccess;  }  public List<SceneHitRecord> getHitRecord(HitRecordQuery query) {  // 1. 校验查询条件，  if (query == null) {  throw new Exception(" HitRecordQuery is null!");  }  if (query.orderId == null) {  throw new Exception(" orderId is null");  }  if (query.limit == 0) {  query.limit = DEFAULT\_LIMIT;  }  // ......  // 其他校验条件  // 2. 根据查询条件获取记录  List<SceneHitRecord> results = sceneHitRecordService.getHitRecord(query);  // 3. 返回记录  return results;  }  } |

图4.2 规则操作实现

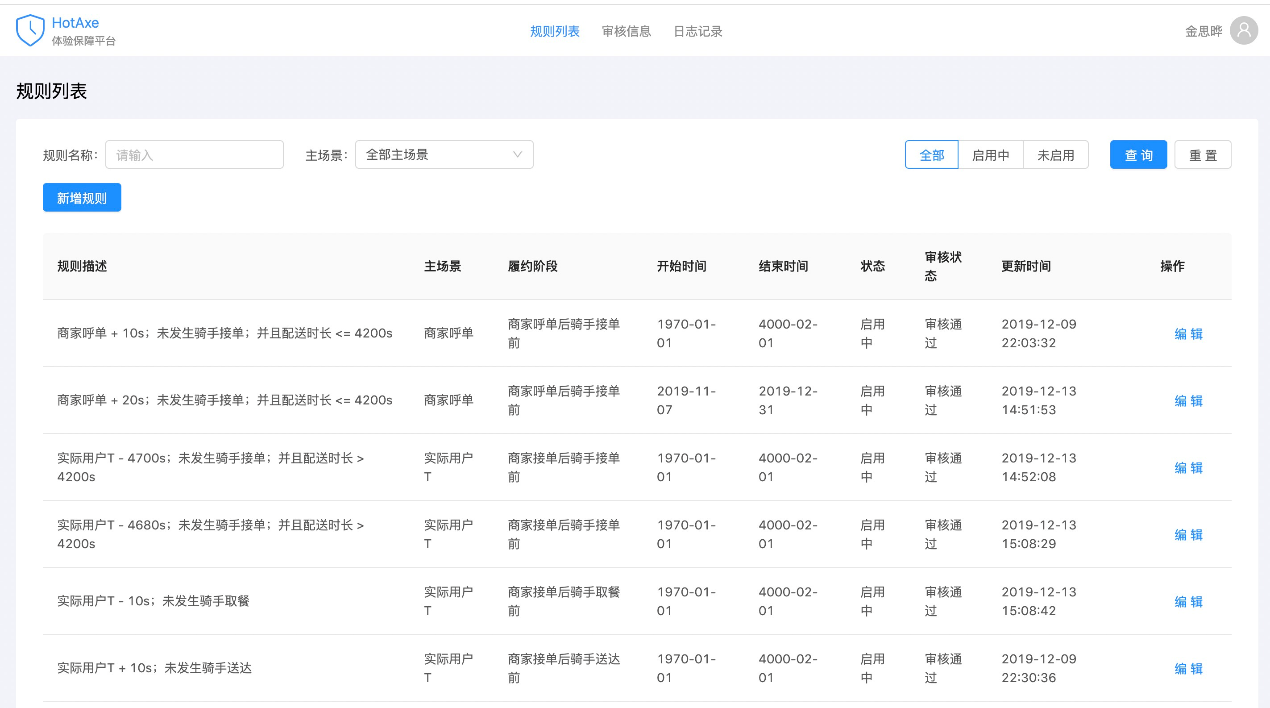


图4.3 规则操作详情页面



图4.4 规则操作详情页面

### 4.1.3 日志查询功能实现要有文字说明，不能只有图，下同

|  |
| --- |
| public Response getHitRecord(OrderHitRecordQuery 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.x 日志查询功能实现

### 规则过期告警功能实现

|  |
| --- |
| @Override  public void callBackTimeOutTask(String ruleName){  SimpleEmail mail = new SimpleEmail();  // 设置邮箱服务器信息  mail.setHostName(host);  // 设置密码验证器passwd为授权码  mail.setAuthentication(user, passwd);  // 设置邮件发送者  mail.setFrom(user);  // 设置邮件编码  mail.setCharset("UTF-8");  // 设置邮件主题  mail.setSubject(title);  //SSL方式  mail.setSSLOnConnect(true);  // 设置邮件内容  mail.setMsg(content);  // 设置邮件接收者  mail.addTo(to);  mail.addTo(tos);  mail.addCc(ccs);  // 发送邮件  MimeMultipart multipart = new MimeMultipart();  //邮件正文  BodyPart contentPart = new MimeBodyPart();  try {  String content = "xxxxxx";  contentPart.setContent(content, "text/html;charset=utf-8");  multipart.addBodyPart(contentPart);  } catch (UnsupportedEncodingException e) {  e.printStackTrace();  } catch (MessagingException e) {  e.printStackTrace();  }  } |

图4.x 规则过期告警功能实现

## 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规则由不可执行状态转换为可以执行的状态，运营人员配置完规则，点击提交之后，规则细节会通过JSON的方式传送到后端，后端以对象的方式对其进行存储，此时的规则称为原生规则，无法执行，仅仅具备了执行时需要的一部分数据，因此需要将原生规则转换成规则最小执行单元，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”、“ADD”、“MULTIPLY”，分别对应逻辑与，存在，小于，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。有了这样的规则执行单元之后，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4.5所示：

|  |
| --- |
| Rst = (((orderBookCondtion  AND deliveryProductIdCondition)  AND predictDeliverTimeCondition)  AND TRIGGERAT(triggerTimeCondition  AND orderStatuCondition)) |

图4.5 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，解析树如下所示：



图4.6 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4.7所示：



图4.7 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。

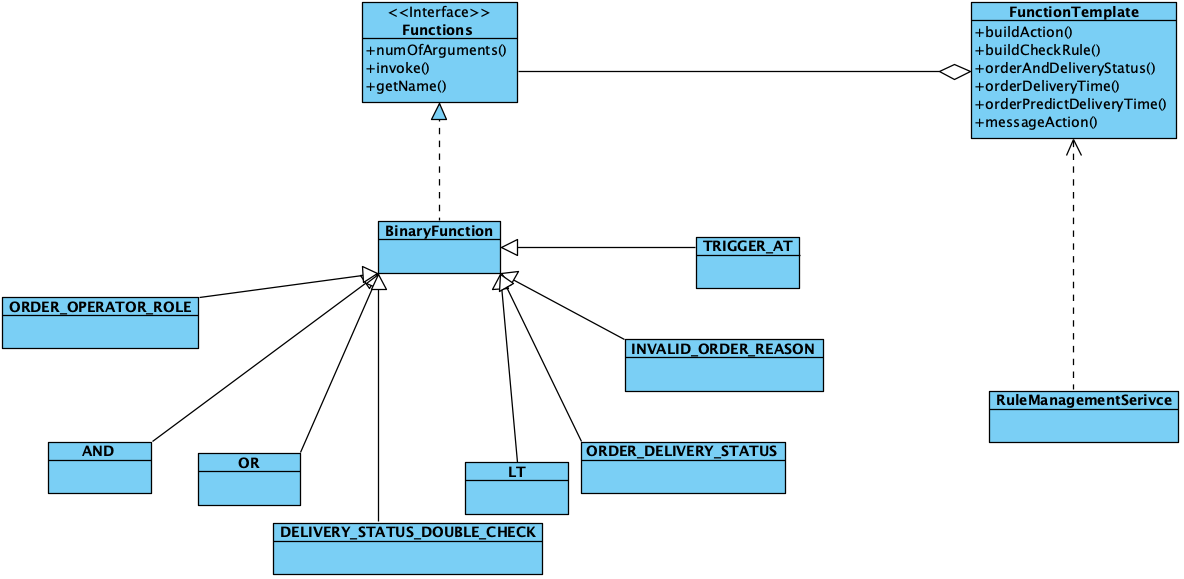


图4.8 规则预处理类图

图4.8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；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和执行动作，buildCheckRule函数构建的是条件校验规则，以上文描述的树状结构进行构建；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。对于执行动作的组织，预处理模块通过管道的方式进行实现，系统使用的是springboot框架进行开发，可以使用springboot当中的一些特性，实现的过程中定义了@Action标签，在执行动作类上加上@Action标签，spring提供按照标签获取对象的操作，因此在真正执行的时候，可以根据标签获取到对应的执行对象，然后执行即可。

### 4.2.2 规则预处理功能实现

## 审核流模块

### 4.3.1 审核流模块设计

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。

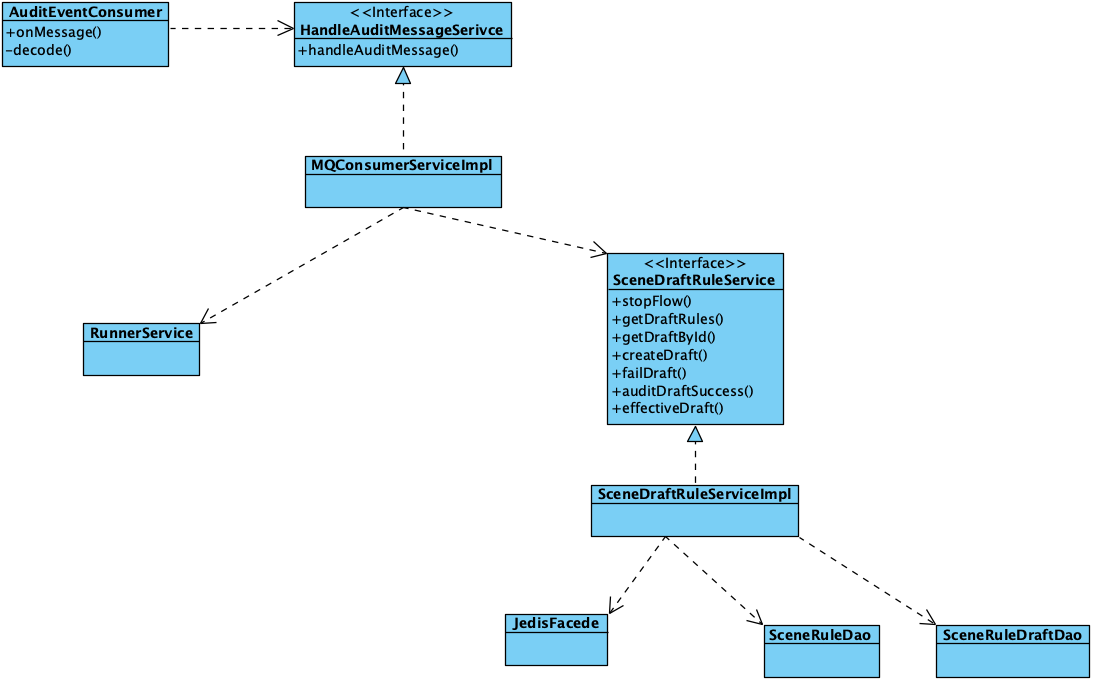


图4.9 审核流模块类图

图4.9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。

### 审核流操作功能实现

|  |
| --- |
| // 根据参数查询  public Response getDraftRule(DraftRuleQuery query){  // 1. 参数校验  validate(query);  // 2. 根据查询参数进行查询  List<DraftRuleDTO> draftRuleDTOList = sceneDraftRuleService.getDraftRule(query);  // 3. 将DTO转换成VO  List<DraftRuleVO> draftRuleVOList = draftRuleDTOList  .strem()  .filter(Object::nonNull)  .map(this::transfer)  .collect(Collectors.toList());  // 4. 返回结果  return Response.SUCCESS(draftRuleList);  }  // 终止审核流  public Boolean terminateAudit(DraftRuleVO draftRuleVO){  // 参数校验  validate(draftRuleVO);  DraftRuleDTO draftRUleDTO = transferTODTO(draftRuleVO);  try{  // rpc调用  Boolean isSuccess = runnerService.terninateAudit();  // 只有当审核平台调用成功之后，才可以本地终止。  // 因此需要通过分布式事务来保证一致性。  if(isSuccess){  sceneDraftRuleService.terminateAudit();  }  return false;  }catch(Exception e){  // rpc调用失败，返回错误。  return false;  }  } |



图4.10 审核流操作界面

### 审核权转移功能实现

|  |
| --- |
| class MQConsumerServiceImpl {  private static String FAIL = "fail";  private static String SUCCESS = "success";  private static String FIRST\_NOTE = "firstNode";  private static String SECOND\_NOTE = "secondNode";  private static String THIRD\_NODE = "thridNode";  private static String Forth\_NODE = "forthNode";  private static String FIFTH\_NODE = "fifth\_node";  private static HashMap<String, String> NEXT\_NODE = new HashMap<>();  static {  NEXT\_NODE.put(FIRST\_NOTE, SECOND\_NOTE);  NEXT\_NODE.put(SECOND\_NOTE, THIRD\_NODE);  NEXT\_NODE.put(THIRD\_NODE, Forth\_NODE);  NEXT\_NODE.put(Forth\_NODE, FIFTH\_NODE);  }  public void handleAuditMessage(AuditMessage message) {  // 1. 校验message  boolean result = validate(message);  if (!result) {  throw new Exception("校验失败！");  }  // 2. 根据message中当前节点状态进行不同的  // 如果消息体当前状态是fail，说明审核失败  if(message.currentState == FAIL){  sceneDraftRuleService.failDraft();  return;  }  if(message.currentState == SUCCESS){  sceneDraftRuleService.auditDraftSuccess();  return;  }  String next = NEXT\_NODE.get(messsage.currentState);  RunnerService.transfer(next);  }  } |

图4.11 审核流操作实现

### 审核消息保障功能实现

|  |
| --- |
| public void checkAuditState(){  // 构建查询参数，  DraftRuleQuery query = DraftRuleQuery  .builder()  .processing(true)  .build();  List<DraftRuleDTO> draftRuleList = sceneDraftRuleService.getDraftRules();  for(DraftRuleDTO draftRule : draftRuleList ){  String ruleName = draftRule.getName();  // 获取审核平台最新结果  AuditItem auditItem = runnerService.getByName(ruleName);  // 本地状态与审核平台状态不一致  if(auditItem.currentState != auditItem.currentState){  // 构建扭转消息  AuditMessage message = buildAuditMessage();  mqConsumerService.handleAuditMessage(message);  }  }  } |

图 4.12 审核信息保障实现

## 业务处理模块

### 4.4.1 业务处理模块设计

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。

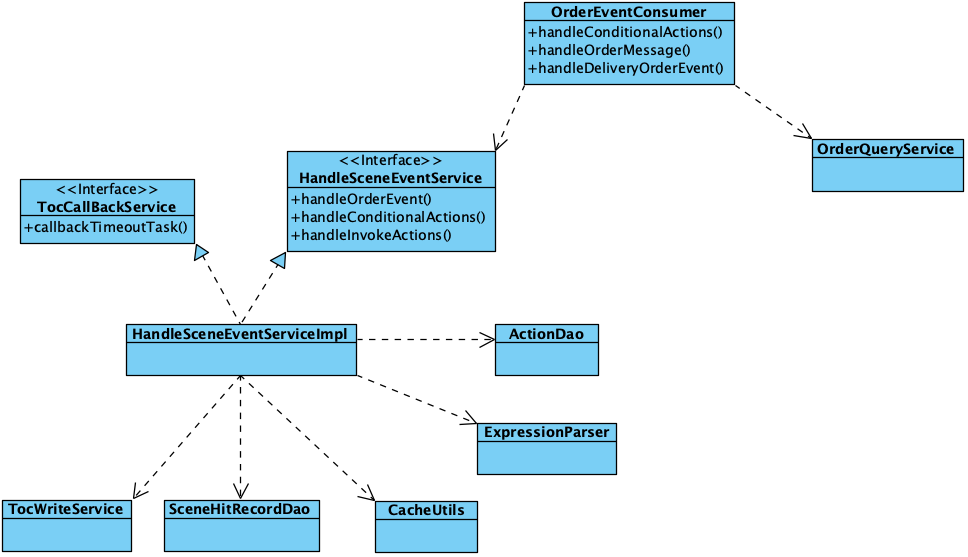


图4.13 业务处理模块类图

图4.13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，数据同步是通过监听消息队列的方式进行，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，以上的操作均是自动完成，但仅仅局限于不同机房数据库之间的更新，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

|  |
| --- |
| Class OrderEventConsumer{  @AutoWried  private HandleSceneEventService handleSceneEventService;  @AutoWried  private OrderQueryService orderQueryService;  // 监听订单消息队列  @RabbitListener(queues = "xxxxxxx")  public void handleOrderMessage(OrderMessage orderMessage){  // 1. 校验orderMessage  validate(orderMessage);  // 2. 准备规则规则判定需要的参数  RuleContext context = RuleContext  .builder(...)  .startNode(...)  .endNode(...)  .arriveTime(...)  .......  // 3. 参数封装完成之后交由handleSceneEventService处理  handleSceneEventService.handleOrderEvent(context);  }  }  Class HandleSceneEventServiceImpl implements HandleSceneEventService,  TocCallBackService {  @AutoWried  private ExpressionParser parser;  @AutoWried  private SceneHitRecordDao sceneHitRecordDao;  @AutoWried  private TocWriteService tocWriteService;  /\*\*  \* 回调函数，通过调用  \* @param ruleText  \*/  @Override  public void callBackTimeOutTask(String ruleText){  // 1. 校验参数  validate(ruleText);  // 2. 将字符串构建成执行树  RuleFunctions rootFunction = parser.parse(ruleText);  // 3. 继续执行  Boolean result= rootFunction.evl();  if(result){  sceneHitRecordDao.hitRecord();  }  }  @Override  public void handleOrderEvent(RuleContext context,OrderMessage orderMessage){  // 1. 校验参数  .......  // 2. 根据订单所在节点获取对应的规则  List<String> rules = getRules(orderMessage.getCurrentNode);  if(Collections.isEmpty(rules)){  return;  }  List<RuleFunction> ruleFunctionList = new ArrayList<>();  // 3. 将JSON数据转换成对象  for(String item : rules){  JSONObject jsonObject = JSONObject.fromObject(item);  RuleFunction rootFunction = JSONObject.toBean(jsonObject,RuleFunction.class);  ruleFunctionList.add(rootFunction);  }  try{  for(RuleFunction rootFunction : ruleFunctionList){  // 从树的根节点开始执行  Boolean result = rootFunction.evl(context);  if(result){  sceneHitRecordDao.hitRecord();  }  }  }catch(TimeOutInterruptException e){  // 对于需要延时判定的规则，会抛出TimeOutInterruptException异常  tocWriteService.createTask(......);  }  }  } |

图4.14 规则判定功能实现

### 缓存更新功能实现

|  |
| --- |
| @RabbitMq("xxxxxx")  public void DrcMessageHandler(DrcMessage message){  // 校验参数  validate(message);  Rule rule = (Rule)message.getDate();  jedis.remove(rule.getName());  } |

图4.15 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xxx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

|  |  |
| --- | --- |
| 硬件项 | 备注 |
| CPU | 8核 |
| 内存 | 32G |
| 硬盘 | 1T |
| 数量 | 两台 |

表xxx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。

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

### 4.5.2 测试设计所有表格要有文字描述

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xxx为规则管理功能的测试用例

|  |  |
| --- | --- |
| 测试ID | TC1 |
| 测试名称 | 规则管理测试 |
| 测试描述 | 测试规则管理功能，包括规则的增删改查，规则命中日志的查看 |
| 测试步骤 | 1. 用户进入规则管理页面，查看已有规则 2. 点击新增按钮，新增规则，点击提交审核按钮 3. 点击修改按钮，对已有规则进行修改，再点击提交审核按钮 4. 点击删除按钮，删除已有规则 5. 点击查看规则命中日志按钮 6. 点击查看未命中原因按钮 |
| 预期结果 | 1. 显示已有规则 2. 新建规则成功，已发送审核 3. 修改规则成功，已发送审核 4. 删除规则成功 5. 显示命中规则的订单 6. 显示订单未命中规则的原因 |

表xxx为审核流功能的测试用例。

|  |  |
| --- | --- |
| 测试ID | TC2 |
| 测试名称 | 审核流功能测试 |
| 测试描述 | 测试审核流的触发、审核流的触发、审核流节点的流转、审核流的终止，以及审核流 |
| 测试步骤 | 1. 用户修改规则 2. 用户新增规则 3. 用户点击进入审核流界面 4. 用户点击终止审核流 5. 审批人员审批审核流 |
| 预期结果 | 1. 触发创建审核流 2. 触发创建审核流 3. 查看已经存在的审核流 4. 审核流被终止 5. 审核通过/审核未通过/流转到下个节点 |

表xxx为规则预处理功能的测试用例

|  |  |
| --- | --- |
| 测试ID | TC3 |
| 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 2. 用户修改规则，并点击提交 |
| 预期结果 | 1. 规则转换成功 2. 规则转换成功 |

表xxx为业务处理功能的测试用例

|  |  |
| --- | --- |
| 测试ID | TC4 |
| 测试名称 | 业务功能测试 |
| 测试描述 | 测试业务处理功能是否达到要求 |
| 测试步骤 | 1. 用户下单 |
| 预期结果 | 1. 命中规则 2. 未命中规则 |

### 4.5.3 测试结果

##

# 第五章 总结与展望

## 5.1 总结

##

## 5.2 工作展望

##

# 参 考 文 献

##

# 致 谢

##