****

**本 科 毕 业 论 文（设计）**

|  |  |
| --- | --- |
| 课题名称 | 自助发票系统后端设计与实现 |
| 学 院 | 计算机科学与网络工程学院 |
| 专 业 | 软件工程 |
| 班级名称 | 软件161班 |
| 学生姓名 | 陈灿鑫 |
| 学 号 | 1606100101 |
| 指导教师 | 金政哲 |
| 完成日期 | 2020年5月10日 |

教 务 处 制

**自助发票系统后端设计与实现**

摘要 业务线众多，财务人员人工对账累、人工开票数量多、开票人员负担重、用户迟迟收到发票导致的投诉率一直居高不下等等，是目前多数企业业务迅速发展时面临的问题。

为了释放企业财务部门的劳动力，笔者设计一套具有用户自助开票、延迟开票（可配置延迟时间）、订单退款自动冲红、退差价重开功能的发票系统，并且具有失败重传机制，最大程度避免因时延、抖动等原因导致的开票失败问题，实现几乎零人工干预，大大提高企业开票效率。

本系统实现主要采用 Golang 1.13 编写服务端程序，前端使用 Vue 编写基于微信公众号的单页面应用，负责展示页面和用户交互；通过MySQL 持久化业务数据；使用 RabbitMQ消息中间件解耦不同业务系统；使用谷歌开源的 gRPC 技术实现多业务线对接发票系统。

关键词 发票系统；自动开票；

**ABSTRACT** Numerous business lines, tired manual reconciliation of financial personnel, large number of manual invoices, heavy burden of invoicing personnel, and a high rate of complaints caused by users' delayed receipt of invoices, etc., are problems faced by the rapid development of most enterprise businesses.

In order to release the labor force of the enterprise's financial department, the author designed a set of invoice system with user self-invoicing, delayed invoicing (configurable delay time), automatic redemption of order refunds, and re-opening of difference spreads. The degree avoids the problem of invoicing failure due to delays, jitters, etc., achieves almost zero manual intervention, and greatly improves the efficiency of enterprise invoicing.

The implementation of this system mainly uses Golang 1.13 to write the server program. The front end uses Vue to write a single-page application based on WeChat official account, which is responsible for displaying pages and user interaction; persisting business data through MySQL; using RabbitMQ message middleware to decouple different business systems Use Google's open source gRPC technology to implement a multi-service line docking invoice system.

**KEYWORDS** Invoice system; automatic invoicing;

目录

[1. 前 言 4](#_Toc1643853662)

[1.1 项目意义 4](#_Toc871378579)

[1.2 发票代理商调研 5](#_Toc1568788360)

[2. 系统开发环境和主要技术 5](#_Toc1969232301)

[2.1开发环境 5](#_Toc2016798990)

[2.2主要技术 6](#_Toc458740682)

[2.2.1 Docker虚拟化技术 6](#_Toc588349644)

[2.2.2 Golang语言 6](#_Toc1377755920)

[2.2.3 数据库设计和使用 7](#_Toc1775065486)

[2.2.4 消息中间件 8](#_Toc682799078)

[3. 需求分析 9](#_Toc1798978025)

[3.1发票系统设计目标 9](#_Toc1001400062)

[3.2可行性分析 9](#_Toc701500495)

[3.3功能点分析 10](#_Toc433597435)

[4. 系统概要设计 11](#_Toc1060075774)

[4.1系统概要设计 11](#_Toc1169198106)

[4.2系统模块设计 11](#_Toc1237197492)

[4.3数据库设计 12](#_Toc1641577790)

[4.3.1数据库E-R图概览 12](#_Toc1275503521)

[4.3.2数据表 12](#_Toc1978177312)

[4.4 持续集成设计 15](#_Toc2031743577)

[5. 系统详细设计与实现 16](#_Toc376827692)

[5.1详细设计 16](#_Toc413744441)

[5.2代码结构 16](#_Toc250770901)

[5.3 WEB中间件的实现 18](#_Toc1343617693)

[5.3.1 CORS跨域 18](#_Toc1392018046)

[5.3.2 JWT鉴权 19](#_Toc960448704)

[5.3 RabbitMQ模块 21](#_Toc1774277276)

[5.3.1 RabbitMQ的消息传递过程 21](#_Toc320255490)

[5.3.2消息监听器 22](#_Toc940001048)

[5.3.3 消息分发 24](#_Toc1707906404)

[5.4功能设计 26](#_Toc1516506226)

[5.4.1订单列表 26](#_Toc1584217786)

[5.4.2退款冲红、退差价重开 28](#_Toc1446073796)

[5.4.3失败重试 29](#_Toc1089856273)

[5.4.5 健康检查 31](#_Toc1326355048)

[总结 32](#_Toc1169035876)

[参考文献 32](#_Toc658081529)

[致谢 33](#_Toc835475853)

**自助发票系统后端设计与实现**

1. 前 言

* 1. 项目意义

传统的开票的大致流程举例：一位客户在一家公司购买课程，付款1500人民币，后台会生成一笔已经支付完成的订单信息，客户支付完钱款后向客服人员表示需要购买课程的发票，客服人员问询了客户需要开票的信息后登记在表格上，财务部通过登记表在订单系统核查，并且查看开票历史记录表格中没有该订单，于是财务给用户通过开票系统开出发票后给客户进行邮寄，在历史记录表格中登记成“已开票”状态，并备注上已邮寄发票的情况。如此繁琐，并且如果某一个环节由于人为粗心（比如客户提供了错误的抬头、税号），那么整个流程涉及到的人力成本将变得十分巨大。

本项目设计初衷为实现零人工干预的客户自助开票，系统自动向企业的订单中心拉取有效订单展示给客户查看，客户填写公司名称即可搜索到公司税号，避免手动输入错误的烦恼。用户提交开票申请后默认需要延迟7 x 24小时（该时间可配置）才真正执行开票任务，避免开票成功后由于客户退款导致频繁冲红的操作发生。极大的释放了企业财务的劳动力，节约了企业用人成本。

* 1. 发票代理商调研

关于发票代理商的选择，需要根据企业具体需求具体分析。具体可从以下几个角度进行参考。

是否提供后台管理软件，如果一个代理商已经实现了后台管理界面，那么财务人员在月底导出账单时将十分方便，无需技术部门再做支持，降低开发成本，极大的缩短开发周期。

是否提供多种票种、多种开票方式。用户需求是多样的，最理想的情况是只对接一家代理商同时能开具多种票种的发票。这对企业每月的报税都是极大便利。如果提供手动开票平台，那么对于特殊情况的用户，也可以紧急处理。

其次，代理商提供的接口文档是否详尽，客服响应速度是否及时，代理商是否有足够多的成功案例，也是选择代理商时需要考虑的重要因素。

1. 系统开发环境和主要技术

2.1开发环境

本次项目开发后端使用Golang 编程语言，版本是 1.13。相比旧版本，该版本在包模块管理中第一次引入 GOPRIVATE 环境变量的配置，解决企业私有代码仓库不需要走代理的需求。

该项目实现了前后端分离，而不是传统的通过渲染模板来加载界面。前端使用Vue框架，需要安装Node环境。内部实现原理仍然是传统的 HTML、CSS、JavaScript技术，前端应用通过HTTP请求实现浏览器与服务端程序之间的数据传输。

项目中还依赖于多种中间件：Redis、RabbitMQ、MySQL,笔者推荐这些中间件特别是项目本身均使用Docker容器化。

2.2主要技术

2.2.1 Docker虚拟化技术

传统的部署流程是项目测试完成后，开发人员手动编译构建，打包成一份二进制文件。经常还需要携带很多依赖软件和环境，然后通过FTP或者SSH传输到服务器后手动运行，之后交付给运维人员。这是极其繁琐甚至痛苦的工作流程。为了一次性的软件创建或配置就能在任何机器环境正常运行这一共同目标，现代开发人员编写 Dockerfile 制作自定义的镜像，然后在持续集成工具的支撑上，当开发人员完成每一次编码工作后Docker快速创建容器，通过层的概念使整个构建过程十分清晰明了，最终使团队所有成员都能理解应用程序的部署过程。并且容器十分轻量，笔者已知最小的镜像是 Alpine，仅占用5MB的硬盘空间。容器的启动时间约为几秒钟，从而节省了大量的开发，测试和部署时间。

Docker允许用户将应用程序从一个平台直接迁移到另一个平台，甚至不丢失原有程序产生的数据（通过卷加载功能）。容器的操作不依赖于其他管理程序。因为它是内核级虚拟化的。Docker容器几乎可以在任何平台上运行，无论是个人计算机、笔记本电脑还是流行的云服务器等。

Docker并不是新技术。命名空间（NameSpaces）是Linux提供的一种用作分离资源的方法，例如进程树，网络接口以及进程间通信。Docker与主机进程和网络的隔离也是通过命名空间来实现，容器中的任何进程都不了解主机的进程。即使Docker可以借助命名空间创建隔离的网络环境，Docker中的服务仍需要连接到外部网路环境才能符合预期。在网桥模式（Bridged Adapter）下，除了分配隔离的网络名称空间，在主机上启动Docker服务进程后，还会创建一个被命名为docker0的虚拟网桥，一般情况下主机上启动的所有Docker容器都将连接到docker0网桥。

2.2.2 Golang语言

Golang是一个快速的、静态类型的、编译型的语言。谷歌公司对Golang寄予厚望。其设计初衷是让应用程序充分发挥多核心处理器并行工作的优点，同时期待它可解决面向对象程序设计随之带来的一些麻烦。它和现代的程序语言一样，具有垃圾回收等特点，帮助开发者处理重要却容易疏忽处理的内存管理问题。Golang语言编写的程序处理任务的速度几乎和C++语言编写的程序一样快，但它能够极大的减少程序员的编码工作。多核心处理器适合并行执行任务的场景，也需要软件应用的配合，而Golang的并发机制使编写高效利用多核的程序变得更便捷。

Golang语言它有很优秀的标准库，使用很少的几十行代码就可以跑一个http 服务器；同时拥有实用的内置数据结构：map字典，同时还有并发安全的sync.Map； slice切片，这是一种具有自动扩容功能的数组结构； channel通道，使得通过通信来共享内存的理念得以实现。append、range等切片方法的内置使得多数常见业务模型的代码更加简洁；代码格式规范方面，引入了go fmt工具来处理括号位置、空格、缩进等代码规范问题。值得一提的是Golang程序编译后只会生成一个可运行的（Runnable）二进制文件，十分方便。

在测试方面，Golang自带了常用的测试框架，可以很好实现单元测试（Unit Test）和压力测试（Stress Test）。Golang自带标准的性能分析工具pprof，例如CPU利用、内存分配、阻塞操作都可以检测。

Golang语言也很适合基础架构、中间件、云计算平台、PasS平台的开发，例如分布式日志平台、搜索系统、消息推送服务等。Docker便是使用Golang语言开发。

2.2.3 数据库设计和使用

本项目中遵循的原则是尽量避免长 SQL，尽可能的拆成简单 SQL，复杂运算在程序代码中处理，目的是为了避免业务高峰期拖垮数据库。特别是在企业使用微服务架构时，业务程序可以通过水平拓展缓解压力，但是数据库却很难水平拓展，即使可以水平分库分表，但带来的代价和成本总是让人望而止步的。所以，尽量只让数据库承担数据存储的功能即可。

执行SQL查询时，合适地建立正确类型的索引能大大提高查询效率。原因是数据库的数据主要存储在机械磁盘，而机械磁盘的随机读写操作有较大的时间开销。数据库索引是一种数据结构，它以额外的读写操作和存储空间为代价，达到加快数据读写速度的目的。建议SELECT、UPDATE、DELETE语句的WHERE条件列、ORDER BY、GROUP BY、DISTINCT的字段、多表join的关联列考虑是否加上索引。如果字段的值重复率很高（例如性别字段仅有男女两种值），那么不必加上索引，因为效果微乎其微反而占用空间甚至还降低了查询效率。

如果在设计阶段就能确定表的记录数会到达很大数量级（亿级或者以上），那么一开始设计就得对数据表做拆分。数据表拆分可以分水平拆分和垂直拆分。

使用多个弱耦合的、独立自治的服务替代原先强耦合的大系统，通过服务间的调用来实现业务功能。不再是直接操作不同模块的表，而是将表拆分出来后，通过服务的形式提供给其他服务使用。这即是垂直拆分，将不同列的业务数据拆分到不同表中。

相对应的，将不同行的数据拆分到不同表中的水平拆分，一般可以按时间、客户、性别、有意义的业务状态、或者通过哈希去拆分。需要具体场景具体分析。如聊天记录表、流水账之类的业务数据，十分适合按时间拆分。而对于本设计——发票系统，如果考虑到实际开票量巨大，在早期设计时也可以考虑按时间进行水平分表。

数据表的拆分会带来的问题之一就是如何选择分布式全局id。笔者推荐使用推特开源的雪花算法。雪花算法的原理其实十分简单，它产生一个64 bit long类型的值，第一位未使用。紧接着是41位的时间戳（单位是毫秒）：

2^41/1000\*60\*60\*24\*365 = 69

上面的计算结果表明这个时间戳可以使用69年不重复，这个对于大部分业务系统已经足够使用了。这个时间戳可以指定一个有意义的时间节点，例如软件上线时间。

接下来的10位是数据机器位，表示可以部署在2^10=1024个节点。最后剩下12位的序列，在毫秒的时间戳内计数。 支持每个节点每毫秒产生2^12=4096个唯一的ID序号，所以最大支持一个节点四百万左右的并发量。这是标准的划分，具体根据企业实际情况划分，如果企业集群超过1024个，那么可以调整机器位为12位或者更多，同时调小时间戳位数。

2.2.4 消息中间件

分布式环境中，服务间的调用经常使用HTTP接口或者RPC远程过程调用来实现。但这两种方式存在以下几个问题。

假设B服务依赖A服务：

1. 当A服务提供的接口发生故障，直接影响B服务的功能可用性；
2. 如果后续有新的C服务也需要依赖于A服务，需要改动代码；
3. 若后续B服务不需要依赖A服务了（如企业下线某条业务线），又需要改动代码。

使用消息中间件只要确保两个服务遵守同样的接口约束，开发者就可以独立地扩展或修改各自的业务逻辑。RPC调用、HTTP调用的强依赖性导致业务系统之间的调用存在耦合，而通过消息中间件可以有效地将业务上弱依赖、系统调用上的强依赖关系进行解耦。

消息中间件另一个常见的作用是削峰。当预期到系统在某段时间的访问量会剧增，但是这样突发的流量并不常见，如果盲目的以能处理这类峰值为标准，去投入硬件资源，会给企业带来不必要的资金浪费。在处于瓶颈的系统组件前部署消息中间件，可以挡住突发访问的压力，避免因为瞬间的超负荷请求而彻底崩溃。

异步处理是消息中间件最明显的作用。比如用户在淘宝购买商品支付成功后，会触发一系列的流程：通知商家发货、赠送买家积分、发送短信提醒用户金额变动等。主要流程是支付流程，其他流程都是可以异步处理的。异步处理可以简化业务流程，提升系统的性能。隔离主要的业务逻辑和次要的业务逻辑。

1. 需求分析

3.1发票系统设计目标

本系统的核心目标在于实现发票自助开具的线上化和自动化，用户通过企业公众号回复关键字“发票”获得开票链接，选择一个订单进入开票页后输入抬头税号以及电子邮箱等信息即可提交开票申请。程序检查用户订单是否在支付完成的 7 x 24小时之后，满足才真正调用代理商开具蓝票接口进行开票。这里延迟开票的需求意义是，尽量用户开票后由于退款或者退差价操作导致冲红或者重开发票操作频繁的。最终代理商的开票服务将发票的PDF文件发送到用户邮箱，完成开票。开票完成通过推送公众号模板消息提醒用户领取发票，支持用户将发票领取到微信卡包。

3.2可行性分析

技术上是可行的。本系统中分工为前端一人，后端开发需要两人，一人负责发票服务，一人负责微信公众号相关的服务封装。本系统设计中由于采用前后端分离、微服务拆分的架构设计，使得分工十分清晰。此外，现在有众多可选择的开票代理商，如海南高灯科技、金蝶、百望等，他们提供了符合Restful规范的HTTP API，使得企业开发内部开票系统将订单和发票一一对应成为可能。

3.3功能点分析

1. 提交开票申请

用户可以选择个人抬头或者企业抬头；

选择企业抬头时只输入企业名称的少部分关键字即可搜索企业全称，匹配正确的企业名称时自动补全企业纳税人税号；

支持更多信息的选填，包括单位地址、单位电话、银行账号、开户银行；

可查看实际开票金额，并且提供可点击的icon，点击icon显示实际开票金额的计算表达式，例如：1000.00（实际开票金额）=1400.00（原价）-200.00（优惠券差价）。

1. 延迟N天开票

用户提交开票信息后，进入“申请中”状态；

程序自动计算出执行开票时间：订单支付时间 + N天，到达开票时刻才进行开具蓝票操作；

如果到达开票时刻时监测到订单有退差价或者退款操作，重算开票金额并进行开具；

支持用户在申请中状态时重新修改开票信息，再次提交；

1. 订单退款自动冲红、退差价重开功能

当发票开具完成后，如果订单发生退差价事件，需要对原先已开具的发票做冲红操作，然后程序重新计算开票金额，重新开具蓝票。操作完毕在企业公众号推送模板消息告知用户；

若订单发生退款事件，则冲红原先已开具的发票无需重开，同样推送公众号模板消息即可。

1. 失败重传机制

当代理商提供的接口没有响应或者响应超时、或者代理商服务正在维护不可用时，需要程序积累失败任务，过后能够自动重试请求开票；

可以筛选已知的错误类型不进行重试，如参数校验出错、税局冻结等原因；

可配置重试延迟时间，可限制最大重试次数；

支持手动触发重试特定订单号的开票操作，支持手动终止特定订单号的重试任务。

3.4 用例分析

|  |  |
| --- | --- |
| 用例名称 | 提交开票 |
| 用例描述 | 1.用户在页面输入企业名称的关键词，弹出列表选择公司全称；  2.选择公司全称后自动输入税号；  3.若提交开票信息时监测到订单为退款中，则返回“订单退款中，无法开票”的反馈信息； |

|  |  |
| --- | --- |
| 用例名称 | 延迟开票 |
| 用例描述 | 1.用户提交信息后，跳转到申请中页面；  2.订单支付时间的7天后，发票成功开具，用户邮箱收到发票PDF附件；  3.用户在7天内申请退款，开票工单变为退款中状态；用户在7天内退款成功，开票工单自动取消，并且隐藏不可见。 |

|  |  |
| --- | --- |
| 用例名称 | 失败重试 |
| 用例描述 | 1.向开票代理商请求开票接口时无响应，发票系统能在8小时后自动重试；  2.重试任务发起时，如果查询到已开具成功，则终止重试流程；  3.重试次数超过30次时，取消重试机制。 |

|  |  |
| --- | --- |
| 用例名称 | 退款冲红 |
| 用例描述 | 1.已开具成功的发票，当订单退款成功，冲红已开具的发票；  2.已开具成功的发票，当订单退差价成功，冲红已开具的发票，并重新以新的发票金额进行重开。 |

1. 系统概要设计

4.1系统概要设计

本系统主要分为四层，分别为视图层，业务逻辑层，基础设施层，以及数据持久化层。

视图层： 公众号H5页面，包括订单列表页、申请开票页、修改开票信息页面、重开发票页面、发票申请中页面以及开票成功页面。

业务逻辑层：

1. 提供符合Restful 规范的API，主要有以下接口：提交开票申请接口、修改开票信息接口、重开发票接口、获取企业名称接口、获取企业税号接口、获取开票订单列表接口、开票详情接口；
2. 提供消息处理方法：退款成功消息、退差价成功消息、开具蓝票延迟消息、开具红票消息；

基础设施层：该层主要使用Redis缓存配置数据，如税率、销售方名称、销售方税号等。使用RabbitMQ作为消息分发器解耦各业务系统，如订单系统检测到用户退款事件即抛出消息给感兴趣的业务系统订阅。

数据持久化层：在本系统中，通过关系型数据库MySQL持久化数据到硬盘。

4.2系统模块设计

**获取订单列表**

为了实现对接多业务线开票，需要将获取订单数据的逻辑抽离出来，通过RPC服务或者HTTP服务获取，为避免网络请求延迟过大，可使用内网地址进行调用。

**提交开票申请**

用户从H5页面填写好开票信息后点击提交，浏览器将数据以JSON形式载入Body传输给后端。后端Web服务将数据存入工单表后，将工单ID作为消息载体，发送延迟消息即可结束流程。

**开票工单的状态转换**

当延迟消息到达后，消息处理器开始执行。将工单信息提交给发票代理商。代理商并不会直接返回结果。而是通过异步接口回调，主动请求商家配置的接口进行开票结果通知。因此，在配置的接口中实现状态判断逻辑。

当开票结果为蓝票开具成功时，返回已开具状态。当开票结果为红票开具成功时，进一步判断是否上下文场景为退差价场景，若是还需要触发开具蓝票事件，重新以新的金额开票。

4.3数据库设计

4.3.1实体关系图

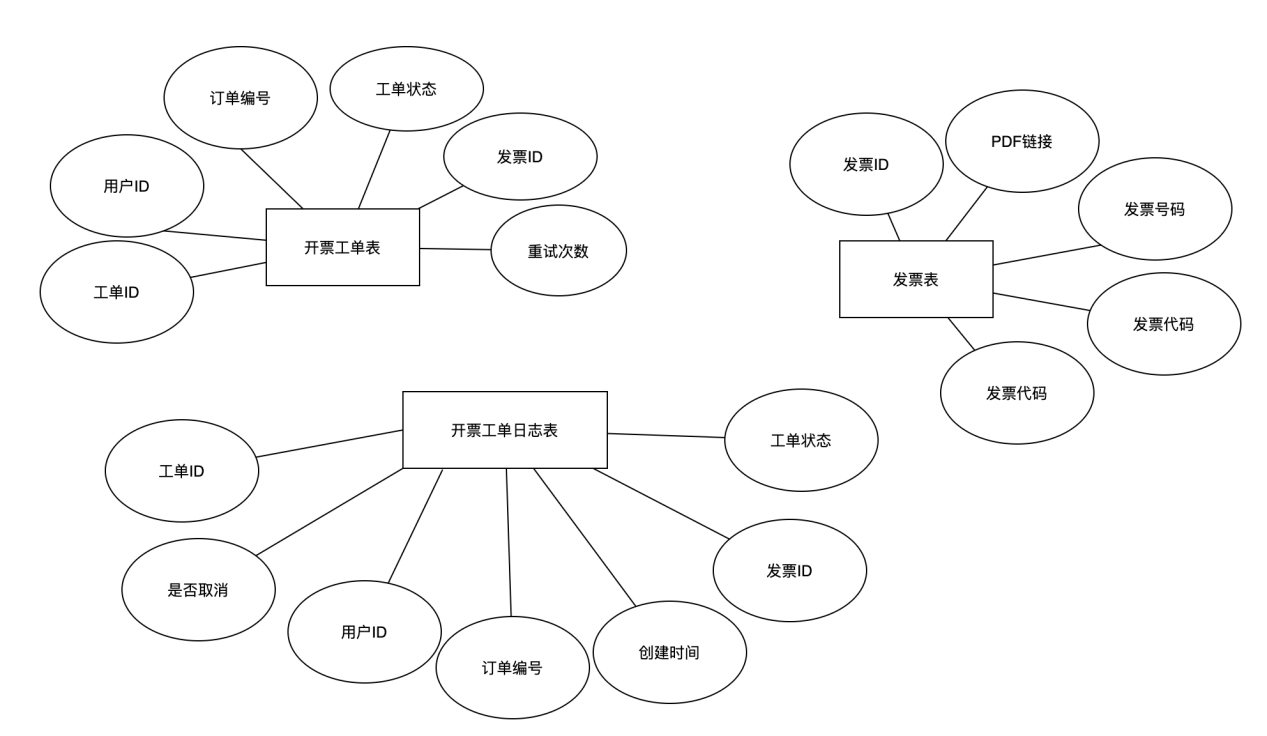


图4-3数据库设计E-R图

4.3.2数据表

order表：用于记录订单信息

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 字段名称 | 类型 | 备注 |
| 1 | id | bigint unsigned | 订单ID |
| 2 | amount | int | 订单金额/单位为分 |
| 3 | channel | varchar(32) | 支付渠道/取值有alipay支付宝支付、wechat\_pay微信支付、union\_pay银联支付 |
| 4 | production\_id | bigint unsigned | 商品ID |
| 5 | state | varchar(32) | 订单状态（created已下单、paid已支付、cancel已取消、ok已完成、pay\_fail支付失败） |
| 6 | created\_time | timestamp | 记录创建时间 |
| 7 | update\_time | timestamp | 记录更新时间 |

invoice表: 记录发票信息

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 字段名称 | 类型 | 备注 |
| 1 | id | bigint unsigned | 发票ID |
| 2 | amount | int | 发票金额/单位为分 |
| 3 | title\_type | varchar(32) | 发票抬头类型，取值有：enterprice:企业,personal:个人 |
| 4 | title\_name | varchar(256) | 抬头名称/企业抬头的话抬头名称为公司名称,个人抬头则为姓名 |
| 5 | tax\_code | varchar(64) | 买方税号 |
| 6 | bank\_account | varchar(64) | 买方银行账号 |
| 7 | organization\_address | varchar(512) | 单位地址 |
| 8 | organization\_tel | varchar(512) | 单位电话 |
| 9 | pdf\_url | varchar(256) | 电子发票的PDF下载链接 |
| 10 | ticket\_code | varchar(32) | 发票代码 |
| 11 | ticket\_sn | varchar(32) | 发票号码 |
| 12 | ticket\_date | varchar(32) | 开票时间 |
| 13 | check\_code | varchar(16) | 发票校验码 |
| 14 | created\_time | timestamp | 记录创建时间 |
| 15 | update\_time | timestamp | 记录更新时间 |

invoice\_approval：开票工单表

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 字段名称 | 类型 | 备注 |
| 1 | id | bigint unsigned | 工单ID |
| 2 | user\_id | bigint unsigned | 用户ID |
| 3 | order\_no | varchar(32) | 订单编号 |
| 4 | state | varchar(256) | 工单状态，commit已提交，blue\_request已向代理商请求开具蓝票，blue\_ok蓝票开具成功，red\_request已向代理商请求开具红票，red\_ok红票开具成功 |
| 5 | invoice\_id | bigint unsigned | 发票ID，由代理商开具完成的实际发票 |
| 6 | amount | int | 订单金额/单位为分 |
| 7 | title\_type | varchar(32) | 发票抬头类型，取值有：enterprice:企业,personal:个人 |
| 8 | title\_name | varchar(256) | 提交的抬头名称/企业抬头的话抬头名称为公司名称,个人抬头则为姓名 |
| 9 | tax\_code | varchar(64) | 工单提交的税号 |
| 10 | bank\_account | varchar(64) | 工单提交的银行账号 |
| 11 | organization\_address | varchar(512) | 工单提交的单位地址 |
| 12 | organization\_tel | varchar(512) | 工单提交的单位电话 |
| 13 | email | varchar(512) | 工单提交的电子邮箱 |
| 14 | created\_time | timestamp | 记录创建时间 |
| 15 | update\_time | timestamp | 记录更新时间 |

invoicie\_approval\_record 开票记录表

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 字段名称 | 类型 | 备注 |
| 1 | id | bigint unsigned | 工单ID |
| 2 | user\_id | bigint unsigned | 用户ID |
| 3 | order\_no | varchar(32) | 订单编号 |
| 4 | state | varchar(256) | 工单状态，commit已提交，blue\_request已向代理商请求开具蓝票，blue\_ok蓝票开具成功，red\_request已向代理商请求开具红票，red\_ok红票开具成功 |
| 5 | invoice\_id | bigint unsigned | 发票ID，由代理商开具完成的实际发票 |
| 6 | amount | int | 订单金额/单位为分 |
| 7 | title\_type | varchar(32) | 发票抬头类型，取值有：enterprice:企业,personal:个人 |
| 8 | title\_name | varchar(256) | 提交的抬头名称/企业抬头的话抬头名称为公司名称,个人抬头则为姓名 |
| 9 | tax\_code | varchar(64) | 工单提交的税号 |
| 10 | bank\_account | varchar(64) | 工单提交的银行账号 |
| 11 | organization\_address | varchar(512) | 工单提交的单位地址 |
| 12 | organization\_tel | varchar(512) | 工单提交的单位电话 |
| 13 | email | varchar(512) | 工单提交的电子邮箱 |
| 14 | cancel | tinyint | 值为1表示用户取消申请，正常默认值为0 |
| 14 | created\_time | timestamp | 记录创建时间 |
| 15 | update\_time | timestamp | 记录更新时间 |

refund\_svc\_address: 退款服务的rpc地址配置表

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 字段名称 | 类型 | 备注 |
| 1 | id | int unsigned | 记录ID |
| 2 | address | varchar(128) | 服务内网地址，需要带上端口号 |
| 3 | extra | text | 服务额外说明信息 |
| 4 | created\_time | timestamp | 记录创建时间 |
| 5 | update\_time | timestamp | 记录更新时间 |

config: 配置表

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 字段名称 | 类型 | 备注 |
| 1 | id | int unsigned | 记录ID |
| 2 | name | varchar(128) | 配置项名称 |
| 3 | value | text | 配置项的值 |
| 4 | type | varchar(12) | 配置项值类型，string、int、json |
| 5 | created\_time | timestamp | 记录创建时间 |
| 6 | update\_time | timestamp | 记录更新时间 |

4.4 持续集成设计

开发团队的每个成员需要频繁集成他们的代码。每次集成都需要自动化构建来推进，可以提前发现集成错误。持续集成可以很大程度减少问题，最终团队能够更快的上线内聚的软件。

GitLab中提供 .gitlab-ci.yml 配置文件，用来配置自动化地完成一些软件的集成工作。当仓库代码发生变动时，比如成员推代码到达仓库，GitLab服务内置的GitLab-CI触发与这个仓库相关联的Runner执行Pipeline。在构建阶段中，根据项目Dockerfile文件进行镜像的构建。然后通过镜像启动容器。

1. 系统详细设计与实现

5.1详细设计

在本章中，将会对系统的几个核心难点和重点进行详细设计，并附上后端部分具体实现代码。重点主要有Web中间件、RabbitMQ模块、延迟开票、失败重试等。

5.2代码结构

笔者将项目的web服务按照MVC分层，MVC中，Model通过调用DAO（数据访问对象）与数据库进行交互。Model层的数据发生变化，通知浏览器更新视图。而Controller更像是View的一个策略，更具体地说是专注于路由、结构化传参的一层。Controller对于View是可替换的， View和Controller的关系是一对多。此外，笔者将业务逻辑独立写在Service层，目的是为了更好的复用代码。所以，Model层通过Service层和Dao层进一步细分了。

├── Dockerfile

├── config

├── dao

│   ├── config.go

│   ├── order.go

│   ├── production.go

│   ├── refund\_rpc\_addresses.go

│   ├── refunds.go

│   ├── rpc

│   │   ├── refund

│   │   │   ├── conn\_manager.go

│   │   │   ├── refund.pb.go

│   │   │   └── refund.proto

│   ├── invoice\_approval.go

│   ├── invoice\_approval\_record.go

│   └── user.go

├── event

│   ├── delay

│   │   ├── issue\_blue\_invoice.go

│   │   └── retry\_issue\_red\_invoice.go

│   └── normal

│   ├── refund\_learning\_card.go

│   ├── refund\_or\_difference.go

│   └── wechat\_card\_authed.go

├── main.go

├── model

│   ├── config.go

│   ├── production.go

│   ├── rpc\_addresse.go

│   ├── order.go

│   ├── user.go

├── service

│   ├── handle\_invoice\_result.go

│   ├── invoice.go

│   ├── invoice\_agent

│   │   ├── blue\_invoice.go

│   │   ├── invoice\_agent.go

│   │   ├── query\_result.go

│   │   ├── red\_invoice.go

│   ├── invoice\_amount.go

│   ├── invoice\_learning\_card.go

│   ├── invoice\_retry.go

│   ├── invoice\_sent\_msg.go

│   ├── order\_learning\_card.go

│   ├── refund.go

│   ├── request\_invoice.go

├── util

├── web

│   ├── controller

│   │   ├── commit\_invoices.go

│   │   ├── edit\_or\_reissue\_invoice.go

│   │   ├── enterprise\_name.go

│   │   ├── enterprise\_ticket.go

│   │   ├── invoice\_order\_detail.go

│   │   ├── invoice\_result.go

│   │   ├── order\_list.go

│   ├── middleware

│   │   ├── cors.go

│   │   ├── jwt.go

│   │   ├── recover.go

│   ├── route

5.3 WEB中间件的实现

5.3.1 CORS跨域中间件

跨域资源共享(CORS) 是一种安全防范机制，它通过操作 HTTP Header来告诉浏览器，让运行在一个 Origin (可以是一个域名) 上的Web服务提供的资源，是否被允许来自不同源服务器上应用访问。当一个Web服务从与它所在的服务器不同的域、协议或端口请求获取资源时，服务会发起一个跨域 HTTP 请求。

Access-Control-Allow-Origin是HTTP响应报文中响应头（Response Headers）存在的字段。它可以完成最简单的访问控制。若资源可以允许被任何源访问，可在服务端返回 Access-Control-Allow-Origin: \* 。如果服务端仅允许来自 http://your-domain.com 的访问，该首部字段的内容如下：

Access-Control-Allow-Origin: http://your-domain.com

因此，CORS是用于拦截浏览器向其他域名发起请求；用于接受还是拒绝其他域名返回的响应数据。对于CORS中间件的设计目标，是将所有HTTP请求到来时，拦截下来判断是否符合企业运维的域限制，避免每添加一个新的接口便需要编写重复的代码判断。

实现代码如下：

// Cors 跨域中间件  
func Cors() gin.HandlerFunc {  
 return func(c \*gin.Context) {  
 method := c.Request.Method  
 c.Header("Access-Control-Allow-Origin", "https://your-domain.com")  
 c.Header("Access-Control-Allow-Headers", "\*")  
 c.Header("Access-Control-Allow-Methods", "GET, OPTIONS, POST, PUT, DELETE")  
 c.Set("content-type", "application/json")  
 if method == "OPTIONS" {  
 c.JSON(http.StatusOK, response.APIResponse{Errmsg: "", Errcode: 0, StatusCode: 200})  
 c.Abort()  
 }  
 c.Next()  
 }  
}

5.3.2 JWT鉴权中间件

JWT(全称JSON WEB TOKEN)定义了一种紧凑（Compact）且自包含（Self-contained）的方式，是用于各系统间以JSON格式的数据安全传输信息的开放式标准。其中数据可以通过数字签名进行验证和信任。例如通过Secrect（使用HMAC算法）或使用RSA的一组公钥/私钥对，对Token签名。一旦用户登录成功即可获得Token，前端应用需要确保后续每个请求都携带上该Token，以便用户可以正常访问该Token允许的路由。

该中间件主要职责是，从HTTP请求报文中的Authorization 这一Header提取Token，使用约定好的Secrect进行解密，判断令牌尚未过期后将用户信息如user\_id等存放在控制器的上下文中。

func Jwt() gin.HandlerFunc {  
 return func(ctx \*gin.Context) {  
 token := getUserJwtToken(ctx)  
 if len(token) == 0 {  
 ctx.AbortWithStatusJSON(http.StatusUnauthorized, response.APIResponse{  
 Errcode: http.StatusUnauthorized,  
 Errmsg: "Unauthorized",  
 Data: nil,  
 })  
 return  
 }  
  
 authUser, err := auth.ParseUserToken(token, []byte(config.Jwt.Secret))  
 if err != nil {  
 log.Info().Msgf("parse jwt failed, err: %s", err)  
 ctx.AbortWithStatus(http.StatusUnauthorized)  
 return  
 }  
  
 ctx.Set("user", authUser)  
 ctx.Next()  
 }  
}

5.3 RabbitMQ模块

RabbitMQ是使用Erlang语言编写的、遵循AMQP（Advanced Message Queuing Protocol 高级消息队列协议）实现的消息中间件。它适合在分布式环境下存储和转发消息。

为了简化业务系统中的代码结构，提高代码复用率，需要在业务系统中封装一个RabbitMQ模块（Golang Package）。该模块主要通过对 streadway/amqp （一个基于 Go语言编写的 RabbitMQ 客户端库）进行业务化封装，主要暴露出添加消息监听器、发送消息等公用方法。通过面向接口的编程规范，具有消息中间件可替换，依赖可注入等优点。更重要的是，本模块需要支持消息广播、保证消息传输可靠性。本节从RabbitMQ的消息传递过程开始分析，最终实现一个开箱即用的RabbitMQ模块。

5.3.1 RabbitMQ的消息传递过程

将投递消息的服务称为生产者，消费消息的服务定义为消费者。在RabbitMQ中Broker指的是部署完成后运行着的服务节点。生产者将消息发送到 Exchange (交换器)，由Exchange将消息路由到一个或者多个队列中。如果找不到相对应的路由，可能导致消息直接被丢弃，也可能返回给生产者。大致流程如下：

1. 生产者（一般是需要触发事件的业务应用）与 RabbitMQ Broker 建立一个 Connection（连接），为了实现多路复用节约TCP连接，还会开启一个 Channel（信道）；
2. 生产者继续声明一个交换器，配置相关属性如交换机类型（如 Topic、Direct、Fanout）。接着声明一个队列井设置相关参数，如是否自动删除。注意声明队列这一动作是幂等的，可以提前声明、创建好队列，在程序启动时再次声明，Broker发现已经创建则直接返回成功而不会执行多余的操作；
3. 交换器和队列声明好后两者通过路由键绑定，特定条件下也可以让交换器和另一个交换器通过路由器绑定；
4. 生产者将携带有路由键、交换器等信息于Lable中的消息发送至 RabbitMQ Broker

5. 途经的交换器根据消息标签里的路由键查找相匹配的队列或者交换器。若成功则将从生产者发送过来的消息投递到对应的队列中。否则根据配置丢弃该消息或回退给生产者；

6. 关闭信道和连接。

值得一提的是，交换器设置为 Fanout 类型时会忽略 Binding Key ，它直接将消息路由到所有与该Exchange绑定的队列中。然而在 Topic 交换器类型下，路由键和绑定键只有模糊匹配成功才会投递到队列。因此这两种类型的交换机对消息的处理流程并不是相同的。

在某些情形下（比如Direct类型交换器） Routing Key 和 Binding Key 可以看作同一个东西。Binding Key 其实也属于路由键中的一种（官方解释为 The routing key to use for the binding）。因此，绑定动作(channel.queueBind )需要指定Binding Key;而发送消息的行为(即channel.basicPublish)需要指定的是 Routing Key。

同一个队列被多个消费者同时订阅是被允许的。该队列中的消息会平均分摊给这些消费者去处理，并非每个消费者都能收到同一个消息。因此实现广播的功能应该使用 Fanout、Topic 之类的交换机。若消费者在信道（Channel）中正在监听某一个队列的消息，则该消费者在同一信道中继续声明其他队列是会遭到Broker拒绝的。

5.3.2消息监听器

根据上一小节的消息传递过程，可以封装消息监听器代码如下：

// ISubscriber 消息监听器

type ISubscriber interface {

// 启动

Run() error

// ExchangeConfig 获取exchange配置

ExchangeConfig() \*ExchangeConfig

// QueueConfig 获取queue配置

QueueConfig(string) \*QueueConfig

// ListenerList 获取监听者列表

ListenerList() []\*EventListener

// WithListenerList 设置监听者列表

WithListenerList([]\*EventListener)

// WorkerNum 获取worker数

WorkerNum() int

// WithFailedHandleFunc 设置错误处理方法

WithFailedHandleFunc(FailHandlerFunc)

// FailedHandleFunc 获取错误处理方法

FailedHandleFunc() FailHandlerFunc

}

// RegisterListenerList 注册监听者列表

func RegisterListenerList(subscriberManager rabbit.ISubscriber, listenerList []\*rabbit.EventListener) (map[string]\*rabbit.Subscriber, error) {

rabbitSubscriberMap := make(map[string]\*rabbit.Subscriber)

exchangeConfig := subscriberManager.ExchangeConfig()

queueHandlerMap := FormatListenerList2QueueHandlerMap(listenerList)

for queue, queueHandler := range queueHandlerMap {

subscriber, \_ := rabbit.NewSubscriber(subscriberManager.WorkerNum(), subscriberManager.ExchangeConfig(), queueHandler.QueueConfig, queueHandler.HandlerFunc)

subscriberName := fmt.Sprintf("%s:%s", subscriberManager.ExchangeConfig().Name, queue)

subscriber.Work()

subscriber.RegisterFailedHandler(subscriberManager.FailedHandleFunc())

rabbitSubscriberMap[subscriberName] = subscriber

}

return rabbitSubscriberMap, nil

}

// Worker 启动

func (subscriber \*Subscriber) Work() (err error) {

ch, \_:= subscriber.Conn.Channel()

defer ch.Close()

ch.ExchangeDeclare(

subscriber.GetExchangeName(), // name of the exchange

subscriber.exchangeConfig.Type, // type

true, // durable

false, // delete when complete

false, // internal

false, // noWait

subscriber.exchangeConfig.Args, // arguments

)

ch.QueueDeclare(

subscriber.GetQueueName(), // 队列名

true, // durable

false, // delete when usused

false, // exclusive

false, // noWait

subscriber.queueConfig.Args, // arguments

)

for event := range subscriber.HandlerFuncList {

ch.QueueBind(

subscriber.GetQueueName(), // name of the queue

event, // bindingKey

subscriber.GetExchangeName(), // sourceExchange

false, // noWait

nil, // arguments

)

subscriber.compileEventMap[event] = compileEvent(event)

}

for i := 0; i < subscriber.WorkerNum; i++ {

go subscriber.startWorker(i)

}

return nil

}

代码中，为了便于阅读去掉了错误处理和配置逻辑等次要流程。如此只需要调用对象的 RegisterListenerList方法即可完成监听者添加：

// ListenerList 监听者列表

var ListenerList = []\*rabbit.EventListener{

{

Event: model.EventIssueInvoice,

CustomQueueName: fmt.Sprintf("delay.queue.%v", model.EventIssueInvoice),

HandleFuncList: rabbit.EventListenerHandleFuncList{

NewIssueInvoiceRabbitMQHandler().Handle,

},

}

}

RegisterListenerList(ListenerList)

5.3.3 消息分发

消息分发同样也需要封装一个消息分发器接口，核心暴露出Publish方法：

// IPublisher 消息分发器

type IPublisher interface {

// Publish 发布消息

Publish(\*PublishingMessage) error

// ExchangeConfig 获取exchange配置

ExchangeConfig() \*ExchangeConfig

// WorkerNum 获取worker数量

WorkerNum() int

// WithFailedHandleFunc 设置错误处理方法

WithFailedHandleFunc(FailHandlerFunc)

// FailedHandleFunc 获取失败处理方法

FailedHandleFunc() FailHandlerFunc

}

Publish方法将待发送的消息全部转发到一个通道（Golang channel）上，再通过后台静默启动的多个Worker去循环监听通道，Worker才是真正向RabbitMQ Broker发出网路请求进行消息传送的角色：

// 启动 worker

func (publisher \*Publisher) startWorker(workerNo int) {

ch, \_ := publisher.Conn.Channel()

defer ch.Close()

for {

select {

case msg := <-publisher.messageQueue:

data, \_ := json.Marshal(msg.Body)

headers := msg.Headers

err:=ch.Publish(

publisher.GetExchangeName(),

msg.Event,

false,

false,

amqp.Publishing{

Headers: headers,

ContentType: contentJSON,

Body: data,

})

if err != nil {

log.Error().Err(err).Msgf("publish message: %s error", data)

msg.AppendError(err)

msg.IncrRetryTimes()

if msg.GetRetryTimes() >= 10 {

failedJob := FailedJob{

Type: FailedJobSend,

Exchange: publisher.GetExchangeName(),

Event: msg.Event,

BindingKey: msg.Event,

Errors: msg.errors,

RetryTimes: msg.retryTimes,

SendAt: time.Now(),

}

msg.ClearStatus()

failedJob.Message = msg

publisher.failedJobChan <- &failedJob

} else {

publisher.messageQueue <- msg

}

publisher.workerRestartChan <- workerNo

return

}

}

}

}

这样的好处是在业务方抛消息时实现了一个缓冲区，缓解 client 发送消息和RabbitMQ Broker 接收消息之间速率不匹配的问题。其二，实现了协程间的通信，解耦不同的模块，提高代码易读性，程序更加容易维护。此外，细心地读者可能已经发现，调用 ch.Publish 方法时笔者没有略去错误处理，是为了体现消息传输可靠性如何在代码中保障：分发器的Worker会累计某次发送操作的失败次数，当达到一定次数后，不再重试而是将该任务送往 failedJobChan 这一通道。在程序的初始化工作里，会常驻一个失败任务管理器，将失败任务暂时记录在硬盘持久化，等待合适的时机再次触发重试。

因此，并不是RabbitMQ具有消息确认机制就不需要开发者去操心了。因为还需要确保Broker能与应用保持正常连接。特别是RabbitMQ服务器意外宕机时，这一细节可以减轻程序员手动推送数据的负担。

在业务实体中，需要发送事件的情况下可以直接调用封装好的方法：

msg := rabbit.PublishingMessage{

Event: "invoice.result",

Body: map[string]interface{}{

"user\_id": userId,

"invoice\_time": createdAt.Unix(),

"order\_no": orderNo,

"invoice\_amount": invoiceAmount,

"notify\_type": status,

},

}

if err = event.Fire(&msg); err != nil {

return errors.Wrap(err, fmt.Sprintf("call microevent.Fire failed,msg=%+v", msg))

}

5.4功能设计

5.4.1订单列表

用户进入系统开票时，在微信客户端打开H5网页，X5内核浏览器发送HTTP请求到达发票系统的Web服务后，经由Web中间件拦截，若请求头携带的Token安全有效，则成功到达订单控制器（Order Controller）。

订单控制器的实现遵循以下接口文档描述，最终返回JSON数据给浏览器：

|  |  |
| --- | --- |
| 接口地址 | /order |
| HTTP Method | GET |
| 返回参数解释 | order\_no 订单编号  status 开票工单的状态  amount 开票金额，单位元，保留两位小数  pay\_amout 支付金额，单位元，保留两位小数  production\_title 商品标题  refund\_amout 退款金额，如果订单无退款记录则为零  difference 退差价金额  paid\_at 订单支付时间  refunding 订单是否在退款中 |
| 成功返回示例 | {  "code": 0,  "msg": "",  "data": {  "orders": [  {  "order\_no": "2020010000101",  "status": "invoicing",  "amount": "1000.00",  "pay\_amount": "1200.00",  "production\_name": "商品名称A",  "refund\_amount" : "100.00",  "difference": "100.00",  "paid\_at": "2019/01/22 14:23:34",  "refunding": false  }  ]  }  } |

控制器在实现时应考虑接口性能问题，若用户订单数量较多，并且流程涉及RPC、TCP请求（与数据库交互），那么为了接口响应时间保证在500ms以内，将每一笔订单的数据构建以并发执行的方式进行，利用Golang语言原生提供的sync.WaitGroup对象和go关键词，可以轻松实现这一点。

// 获取订单列表

func (ctrl \*OrderController) Get(ctx \*gin.Context) {

var (

result []Order

orders []dao.OrderForInvoice

)

student := ctrl.mustGetAuthUser(ctx)

orders, \_ = ctrl.OrderService.GetOrderForInvoice(student.ID)

if isErrRecordNotFound(err) {

ctrl.RespNotFound(ctx)

return

}

var mutex sync.Mutex

var wg sync.WaitGroup

for \_, v := range orders {

wg.Add(1)

go func(order dao.OrderForInvoice) {

defer wg.Done()

temp=ctrl.BuildEveryOrderInfo(order)

mutex.Lock()

result = append(result, temp)

mutex.Unlock()

}(v)

}

wg.Wait()

ctrl.RespData(ctx, result)

return

}

订单控制器中核心逻辑是调用订单服务对象（OrderService）获取到数据的。OrderService暴露方法的主要流程是，通过gRPC（gRPC是谷歌开源的可以在任何环境中运行且现代高性能RPC框架） 调用企业订单中心的获取用户订单方法，根据需求制定的规则做筛选和数据拼装等处理。对于已开票等订单数据，还需要调用发票系统本项目的InvoiceService获取已开票信息。

5.4.2退款冲红、退差价重开

订单系统在监听到用户退款或者退差价操作的请求时，执行对应主流程后，通过RabbitMQ模块抛出退款或者退差价事件消息。

发票服务作为感兴趣的一方，需要订阅该消息。在退款或者退差价的事件处理器中，首先判断是否该订单已开票，若没有开票记录直接忽略，终止流程。否则调用代理商服务提供的冲红方法进行冲红。

由于使用开箱即用的RabbitMQ模块，生产者和消费者各自所在的服务只需要约定好事件名称即可。而实际上事件名称对应RabittMQ中的Routing Key，声明的Exchange交换机为Topic 类型。

const (

EventRefundSuccess = "order.refund.success"

EventRefundDifferenceSuccess = "order.refund.difference.success"

)

var ListenerList = []\*rabbit.EventListener{

{

Event: EventRefundSuccess,

CustomQueueName: fmt.Sprintf("queue.%v", EventRefundSuccess),

HandleFuncList: rabbit.EventListenerHandleFuncList{

NewRefundOrDifferenceRabbitMQHandler().Handle,

},

},

{

Event: EventRefundDifferenceSuccess,

CustomQueueName: fmt.Sprintf("queue.%v", EventRefundDifferenceSuccess),

HandleFuncList: rabbit.EventListenerHandleFuncList{

NewRefundOrDifferenceRabbitMQHandler().Handle,

},

},

}

若事件类型为退差价，则冲红后还需抛出开具蓝票事件消息，触发开具蓝票事件处理器重新计算最新开票金额进行开票。

5.4.3失败重试

首先需要归类失败场景的种类。例如用户填写的开票信息错误，并且未被Web控制器拦截，导致已经向代理商提交开票请求。显然代理商会返回参数错误之类的错误码，这种错误只需报警或者直接反馈错误信息给到用户界面。

对于代理商服务无响应或者代理商服务底层报错、服务器错误等，才是需要失败重试的场景。

func (i invoice) ExceptionRetry(param ExceptionRetryParam) (err error) {

var (

currentTime int

queryResultResponse invoice\_agent.QueryResultResponse

maxRetryTime = 30

exceeded bool

)

// 先检查是否开票成功

bytes= i.ServiceGeoDengInvoiceAgent.QueryResult(orderNo)

json.Unmarshal(bytes, &queryResultResponse)

if i.InvoiceAgent.IsQuerySuccess(queryResultResponse.Data.Status) {

log.Warn().Msgf("查询代理商接口该订单实际已开票成功，现取消重试")

return

}

i.DaoConfig.GetCfg("invoice.max.retry.time", &maxRetryTime, 120)

exceeded,currentTime=i.ExceededLimitTimesInMinutes(fmt.Sprintf("invoice.retry.time.%v", param.Ui.OrderNo), maxRetryTime, 30\*24\*60)

if exceeded {

log.Error().Interface("param", param).Msgf("超过重试次数，现放弃重试")

return

}

delay, err := i.GetRetryDelay()

var msg = model.IssueInvoiceRabbitMQHandlerMsg{

UserInvoiceRecordId: param.Uir.Id,

}

i.SentQueueMsgToIssueInvoice(msg, delay)

return

}

5.4.5 健康检查

Liveness probes（活性探针）是Kubernetes的功能。本设计中，在Openshift Pod的后台添加活性探针，配置探针类型为HTTP GET，Path参数为"/health-check"，Port配置为8099。如此，Kubernetes将会每隔10s（可通过配置periodSeconds参数改变探测延迟）访问该Pod中8099端口提供的HTTP服务，当HTTP响应状态码为200-399之间时，Kubernetes认为该Pod是健康的，否则，连续失败次数达到failureThreshold时，Kubernetes将重启该Pod。

因此，业务系统需要额外运行一个健康检查HTTP服务并提供接口，回应容器编排工具的健康监测请求。

该健康检查控制器的判断健康逻辑可以为携带Timeout参数的Ping 数据库服务，例如5s内Ping不通数据库所在服务器，那么返回不健康。

func DbPingTimeout() bool {

dbName := "db01"

timeout := 5

ctx=context.WithTimeout(context.Background(),time.Duration(timeout)\*time.Second)

err := model.GetClient(name).DB().PingContext(ctx)

if err != nil {

log.Error().Err(err).Msgf("call PingContext fail,name=%s", name)

timeoutDbs = append(timeoutDbs, name)

return true

}

return false

}

健康规则可以是判断是否请求外网成功、是否与各个基础设施服务连接正常。还可以是检查服务所在容器的 CPU 利用率、内存使用率，检查磁盘剩余空间、IO 情况等。当不正常时，通过Error日志报警提醒相关负责人，或者重启Pod尝试解决问题。

总结

本设计与实现解决了企业自动化开票的问题。为了开发高性能的业务应用,开展了数据库设计的要点分析：索引的建立、水平和垂直分表等。阐明了RabbitMQ的消息传递过程，以及它在企业应用程序上的应用。实现一套具有失败重试、健康检查功能的稳定、健壮和可拓展的自助发票系统。

参考文献

[1] 朱忠华. RabbitMQ实战指南[M]. 电子工业出版社, 2017.

[2] 郝林 .Go并发编程实战（第2版）[M].人民邮电出版社, 2017.

[3] 艾伦 A. A. 多诺万 .Go程序设计语言[M].机械工业出版社, 2017.

[4] Sau Sheong Chang .Go Web 编程[M].人民邮电出版社, 2017.

[5] Josiah L. Carlson.Redis实战[M].人民邮电出版社, 2015.

[6]涂腾飞. Go语言中的并发问题研究[D].北京邮电大学, 2019.

[7]Andrew S. Tanenbaum, Herbert Bos.现代操作系统[M].机械工业出版社, 2017.

[8]谢希仁.计算机网络（第7版）[M].电子工业出版社, 2017.

[9]Alvaro Videla, Jason J. W. Williams. RabbitMQ in Action[M]. Manning Publications, 2012.

[10]Karl Matthias, Sean P. Kane. Docker Up and Running[M]. O'Reilly Media, 2015.

[11]Jeremy D. Zawodny, Derek J. Balling. High Performance MySQL[M]. O'Reilly Media, 2004.

[12]Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra, Elisabeth Robson. Head First Design Patterns[M]. O'Reilly Media, 2004.

[13]Leonard Richardson, Sam Ruby. RESTful Web Services

[M].O'Reilly Media, 2007.

[14]David Gourley, Brian Totty. HTTP[M]. O'Reilly Media, 2002.

[15]Kasun Indrasiri,Danesh Kuruppu. gRPC: Up and Running

[M].O'Reilly Media, 2020.

[16]Sasha Pachev. Understanding MySQL Internals[M].O'Reilly Media, 2007.

致谢

“新竹高于旧竹枝，全凭老干为扶持”。致谢金政哲老师的辛勤付出。金老师独特的教学方法，使我的学习方法得到新感悟：至少在编程这一行业，付出实践永远是获得答案最稳当的方法。金正哲老师就像我身边严厉的“大师兄”，能一针见血地指出并纠正我的表达错误。金正哲老师又像是一位生活上的朋友，多次与我们畅谈前沿的技术并且给出职业发展路线的指导和建议。

致谢15CST的师兄们。为我们的项目开发和求职道路上带来了许多指导。

致谢16CST的伙伴们，一个人走的很快，一群人走的更远。不知道多少次的技术交流会、思维碰撞，让我们不知不觉地快速成长。

致谢我的家人，你们是我最坚实的后盾，让我能在大学四年心无杂念地沉浸在编程海洋中。