面向“3+X“模式的多元化实践学分管理系统

详细设计说明书

赛道名称： 2024 年全国大学生计算机系统能力赛

赛道类别： 操作系统设计赛（全国）- OS 应用赛道

项目名称： 面向"3+X"模式的多元化实践学分管理系统

参赛学校： 上海电力大学

参赛队员： 许青峰、乐天、朱凯年

指导教师： 徐曼

提交日期： 2025 年 12 月

目录

第一章 概述

1.1 项目背景与意义

随着高校教育管理信息化的深入，传统的单体教务系统逐渐暴露出数据孤岛严重、流程冗余、维护成本高昂等问题。特别是在学分管理和第二课堂活动审批中，缺乏高效、透明且安全的数字化解决方案。

本项目旨在通过数字化技术深度赋能上海电力大学"3+X"应用型人才培养模式改革。针对该模式下实践场景多元化（X）带来的管理复杂性，以及学分置换标准不统一、数据多源异构等现实挑战，团队构建了《面向"3+X"模式的多元化实践学分管理系统》。该系统旨在打破企业实习、学科竞赛与科研项目之间的数据壁垒，通过构建标准化的数据底座，实现对多模态实践教学成果的全生命周期可信管理与智能化评价支撑。

不同于常规的业务系统开发，本项目深度结合操作系统底层原理，在 Debian GNU/Linux 13 (Trixie) 实验环境下，探索并应用了 Linux Kernel 6.12 的最新特性。系统采用容器化微服务架构设计，利用云原生技术的高弹性优势解决应用交付难题，并依托 Linux 内核的 Namespaces 与 Cgroups 机制实现严格的资源隔离。同时，项目深入实践了 EEVDF 内核调度算法及系统级安全通信机制，确保在高并发场景下的调度效率与数据安全。通过该平台，管理员与师生可实现高效的学分申请、流转与审批，这不仅有效解决了教务管理痛点，更是一次在操作系统应用层面上对容器编排与内核特性应用的深度实践。

1.2 操作系统环境与选型依据

本项目在操作系统选型上采取了"开发效率优先，兼顾生产迁移"的策略，并在内核层面进行了深入考量。

1. 基础运行环境：Debian 13 (Trixie)

我们选择了 Debian 的 Testing 分支作为基础运行环境，主要基于以下依据：

内核特性支持：Debian 13 原生集成了最新的 Linux 6.12+ 内核。这使我们能够利用新内核引入的 EEVDF (Earliest Eligible Virtual Deadline First) 调度器。相比传统的 CFS 调度器，EEVDF 能显著降低微服务高并发场景下的延迟，提升响应速度。

资源管理：系统默认启用的 Cgroups v2 为容器提供了更统一、更细粒度的资源层级管理，是现代容器化应用的最佳拍档。

2. 跨系统兼容性与国产化考量

项目构建了从 Debian 开发环境到 OpenEuler 生产环境的无缝迁移路径。利用 Docker 容器技术屏蔽底层 OS 差异，既享受了 Debian 丰富的软件源生态，又保留了向国产化信创操作系统 OpenEuler 迁移的能力，符合国家对基础软件自主可控的趋势。

对比维度	Debia13 (Trixie)	OpenEuler (LTS)	本项目决策
内核策略	激进/最新 (Linux 6.12+)	保守/稳定 (Linux 5.10/6.6)	开发选 Debian：利用新内核特性调试容器性能。
软件包管理	apt (dpkg)，库极其丰富	dnf/yum (rpm)，兼容 CentOS	开发选 Debian：依赖管理更便捷，报错易排查。
容器支持	Docker 官方首选基础镜像	支持 Docker 及自研 iSula	通用：Docker 屏蔽了底层差异。
应用场景	桌面开发、前沿技术验证	服务器部署、信创国产化	生产备选 OpenEuler：满足潜在的国产化交付需求。

表 1 两种操作系统的多维度对比

3.基于容器技术的无缝迁移策略

在本项目的基础架构设计中，我们充分利用了 Docker 容器技术带来的应用层与底层操作系统解耦优势，构建了"Debian 开发，OpenEuler 落地"的灵活交付策略。由于 Linux 容器本质上是共享宿主机内核的轻量级虚拟化技术，只要目标宿主机的内核满足容器运行的最低要求且保持应用程序二进制接口兼容，在 Debian 13 环境下构建的标准容器镜像即可无缝迁移至 OpenEuler 生产环境中运行。这一策略不仅让开发团队在编码阶段充分享受了 Debian 社区丰富的软件源和便捷的调试工具，极大缩短了开发周期，同时也为系统未来在国产化信创操作系统 OpenEuler 上的生产落地预留了零成本迁移通道，完美兼顾了开发效率与交付标准。

跨系统迁移路径设计见下图（图 1）。

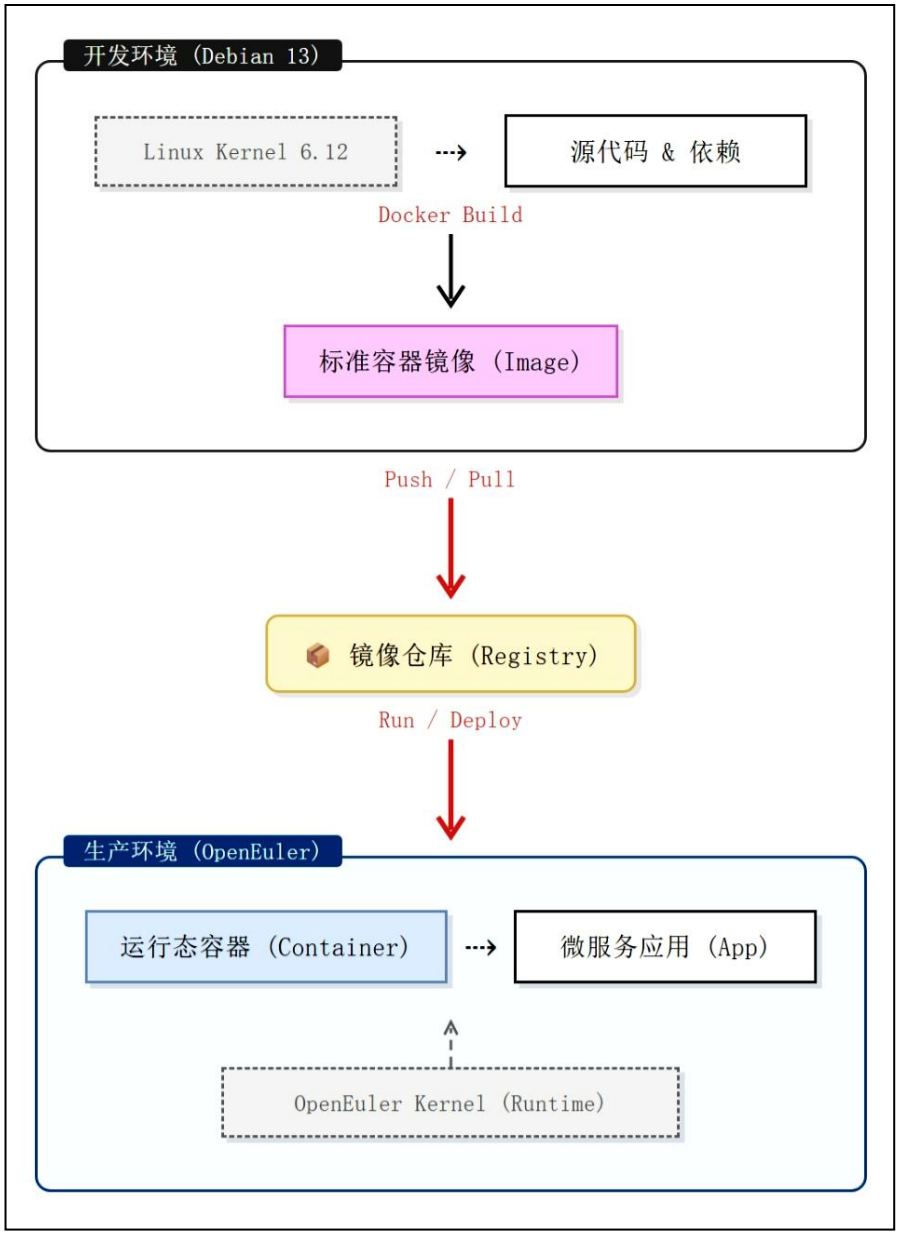

图 1　跨系统迁移路径

针对学分管理系统微服务架构高并发、低延迟的业务需求，本项目深度挖掘并应用了 Linux 6.12 新内核的关键特性：

首先，在进程调度层面，项目受益于内核引入的 EEVDF（Earliest Eligible Virtual Deadline First）调度器。与传统的 CFS（完全公平调度器）相比，EEVDF 更侧重于满足进程的延迟约束。在学分申请高峰期，当大量并发请求涌入 API Gateway 及后端微服务时，EEVDF 能够更精准地管理进程的时间片，显著降低微服务进程在就绪队列中的等待时间。这意味着系统能够以更低的延迟处理认证鉴权、数据库读写等关键任务，从而大幅提升用户端的响应速度和交互体验。

其次，在资源隔离层面，Debian 13 默认启用的 Cgroups v2 机制为多容器环境提供了坚实的保障。Cgroups v2 摒弃了 v1 版本中层级混乱、控制器独立

的弊端，提供了统一的层级管理视图。在本项目中，我们利用 Cgroups v2 对认证服务、用户服务、活动服务以及 PostgreSQL 数据库等容器进行了精细化的资源限制与审计。这种统一的资源管理机制有效防止了因某个单一服务（如复杂的报表导出任务）发生资源争抢或内存泄漏而导致整个宿主机系统雪崩，确保了学分管理系统在复杂负载环境下的高可用性与稳定性。

1.3 应用场景和需求

1.3.1 功能需求

系统覆盖了学分管理的全生命周期：

用户管理：支持学生、教师、管理员三类角色的注册、登录及鉴权。

活动管理：支持活动的发布、编辑、状态流转（草稿/待审核/已发布）。

学分审批：实现自动化的学分申请生成与原子化的审批流程。

数据可视化：提供仪表盘，展示学分统计与活动趋势。

1.3.2 非功能性需求

高可用性：通过 Docker Compose 编排，确保服务在异常退出时可自动重启。

安全性：从 SSH 密钥对连接到底层数据库加密，再到应用层 JWT 校验，构建全栈安全体系。

响应速度：利用 Go 语言的高并发特性及 Linux epoll 机制，确保高并发下的低延迟。

1.4 本文创新点

1.4.1 深度结合操作系统内核新特性的底层架构创新

本项目并未止步于应用层开发，而是深入操作系统底层，探索并应用了 Linux 内核的前沿特性，这在常规的应用开发项目中较为少见。

（1）引入 Linux 6.12 内核与 EEVDF 调度器：项目构建于 Debian GNU/Linux 13 (Trixie) 环境下，率先采用了 Linux 6.12 内核。创新性地利用了 EEVDF（Earliest Eligible Virtual Deadline First）调度器替代传统的 CFS 调度器。在微服务高并发场景下，该调度器显著降低了服务响应延迟，提升了计算密集型任务的处理效率。

（2）基于 Cgroups v2 的资源精细化管理：不同于传统的 Cgroups v1，本项目利用 Debian 13 默认启用的 Cgroups v2 机制，实现了对容器资源（CPU、内存、I/O）更统一、更层级化的隔离与限制，有效防止了单点故障导致的系统雪崩，增强了系统的健壮性。

（3）跨架构无缝迁移策略：制定了"Debian 开发，OpenEuler 落地"的技术路线。利用容器技术屏蔽底层 OS 差异，既利用了 Debian 丰富的软件生态提高开发效率，又为未来迁移至国产信创操作系统 OpenEuler 预留了可行的技术路径。

1.4.2 高内聚低耦合的云原生微服务架构设计

项目摒弃了传统的单体架构，采用符合云原生标准的微服务架构，实现了业务逻辑与基础设施的深度解耦。

（1）服务网格化的通信机制：系统将认证、用户、活动等模块解构为独立进程，运行在独立的 Namespaces 中。通过 API Gateway（Go/Gin）作为统一流量入口，结合 Nginx 实现反向代理与负载均衡。利用 Docker 内部 DNS 解析机制，实现了服务间的精准动态路由与服务屏蔽，对外仅暴露统一接口，极大提升了系统的安全性和可维护性。

（2）高性能并发处理：后端采用 Go 语言开发，充分利用其 Goroutine 协程机制与 Linux epoll I/O 多路复用技术，在处理高并发 HTTP 请求时表现出色，有效降低了上下文切换开销。

1.4.3 基于事件驱动的自动化业务流转机制

针对传统教务管理中"逐个申请、逐个审批"的低效痛点，创新设计了基于事件驱动的自动化处理流程。

（1）原子化审核与副作用触发：设计了"活动发起—审核决策—自动执行"的闭环流程。一旦教师端做出"通过"决策，后端 API 不仅更新状态，更会立即触发副作用逻辑（Side Effects）。

（2）事务性批量处理保障数据一致性：利用 PostgreSQL 的原子事务（Transaction）机制，在更新活动状态的同一上下文内，自动遍历所有参与者并批量生成学分申请记录。这种设计确保了在面对大规模数据处理时，操作要么全部成功，要么全部回滚，彻底杜绝了数据不一致问题。

1.4.4 全栈式安全防御体系与运维创新

项目构建了一套从底层网络传输到上层应用交互的纵深防御体系。

（1）底层运维安全通道：在 SSH 连接层面，摒弃了传统的 RSA 算法，采用更安全的 Ed25519 算法生成密钥对。针对服务器非标准端口（Port 33）配置了本地端口转发与 SSH 隧道，实现了在不直接暴露数据库端口的情况下进行安全的远程管理。遵循"最小权限原则"，严格限制密钥文件权限（chmod 600），杜绝了越权访问风险。

（2）双重路由守卫机制：在前端架构中，创新设计了 Authentication Guard（身份验证）与 Authorization Guard（权限鉴别）双重路由拦截算法。通过动态解析

JWT Token 载荷与路由元数据进行集合运算，实现了基于 RBAC 的细粒度权限控制，有效防止了 URL 越权访问。

（3）数据隐私保护：在数据库层面，强制实施 Bcrypt 哈希加密存储用户密码，确保即使数据库泄露也无法反推原始凭证。

1.4.5 数据驱动的前端工程化与交互体验优化

前端开发不仅仅是页面堆砌，而是引入了大量算法逻辑与工程化规范，提升了系统的可用性与扩展性。

（1）动态路由决策树与异构数据标准化：实现了多维查询参数构建算法与异构数据标准化策略（Data Normalization），通过适配器模式统一清洗后端返回的分页或扁平数据，保证了组件的鲁棒性。同时，侧边栏菜单采用线性表构建算法动态生成，实现了 UI 结构与业务权限的逻辑解耦。

（2）嵌入式 HelpMe 导航微件：针对新用户上手难的问题，开发了基于抽屉式交互的 HelpMe 导航微件。提供可视化的状态流转指引与图例字典，用户无需离开当前页面即可获取操作帮助，显著降低了学习成本。

（3）视觉反馈优化：封装 TopProgressBar 组件结合 React Suspense，在异步请求期间提供动态进度反馈，消除了网络延迟带来的用户等待焦虑，优化了交互体验。

1.5 组织结构

本项目的开发团队由三名成员组成，分工明确，协作紧密：

许青峰：负责前端 React 视图层架构设计、用户体验优化、系统测试及文档撰写。

朱凯年：负责后端 Go 微服务开发、API 网关设计、PostgreSQL 数据库建模及 SQL 调优。

乐天：负责 Docker 容器化运维、操作系统环境配置、Bug 修复及 CI/CD 流程探索。

第二章 系统整体架构及功能模块设计

2.1 系统整体技术架构

系统采用前后端分离的微服务架构，自下而上构建于 Linux 操作系统与 Docker 容器基础设施之上。利用 Linux 6.12 内核的高效调度特性，支撑上层应用的高并发运行。

接入层：Nginx 反向代理，负责 SSL 卸载与静态资源服务，监听 80/443 端口。

网关层：API Gateway (Go/Gin)，作为微服务集群的"统一门户"，负责路由分发、负载均衡与 JWT 初步校验。

服务层：

Auth Service：负责用户认证、令牌签发。

User Service：处理用户信息、角色权限。

Activity Service：核心业务服务，处理活动申请与学分计算。

数据层：PostgreSQL 15 (业务数据) + Redis (缓存)，利用 Docker Volume 实现数据持久化。

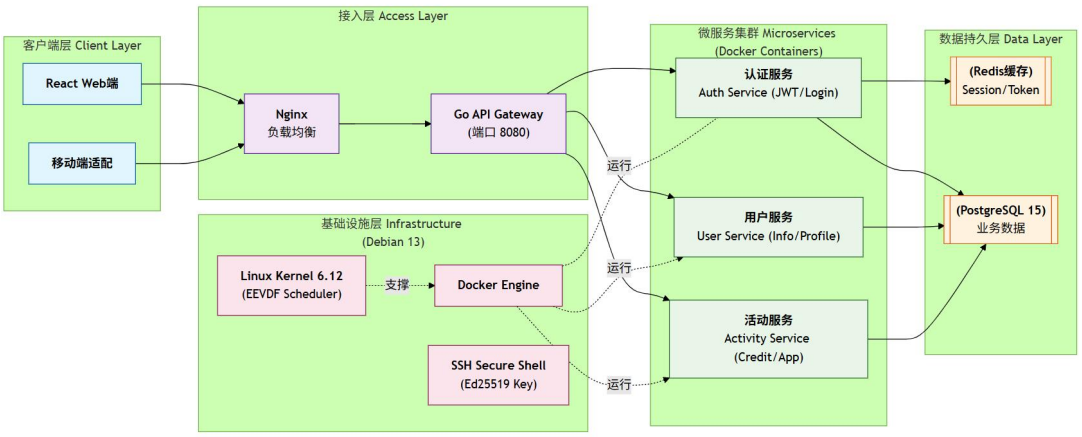

图 2 系统整体技术架构图

2.2 容器化与内核隔离机制

本项目并非单纯使用 Docker 作为打包工具，而是基于 OS 原理实现了严格的资源隔离：

1. 视图隔离 (Namespaces)

系统利用 Linux Namespaces 技术实现了环境的彻底隔离：

PID Namespace：确保每个微服务进程在容器内拥有独立的进程 ID 空间（PID 1），互不干扰。

NET Namespace：为每个服务容器提供独立的网络栈（虚拟网卡、路由表），服务间通过 Docker 内部 DNS 进行通信，对外隐藏真实 IP。

MNT Namespace：挂载独立的文件系统，配合 OverlayFS 实现镜像的分层存储与写时复制 (CoW)，提高存储效率。

2. 资源限制 (Cgroups)

为了防止单点故障导致宿主机系统雪崩，项目通过配置 docker-compose.yml，利用 Cgroups (Control Groups) 限制各微服务容器对 CPU 时间片和物理内存的使用配额。例如，限制数据库容器最大内存使用量，防止内存泄漏耗尽服务器资源。

2.3 数据库实体设计

系统核心数据存储于 PostgreSQL 15+ 中，设计遵循高度规范化与安全性，充分利用了对象关系型数据库的特性：

用户实体 (Users)：

主键采用 UUID 而非自增 ID，有效防止 ID 遍历攻击。

user_type 字段使用枚举类型严格限制角色（Student/Teacher/Admin）。

活动实体 (Credit_Activities)：

引入 JSONB 字段存储非结构化的扩展信息（如活动自定义表单），实现了 SQL 与 NoSQL 的融合。

利用数据库 CHECK 约束确保状态机逻辑的正确性。

组织架构树 (Departments)：

利用 parent_id 自引用外键，构建了"学院-专业-班级"的无限层级树形结构，支持递归查询。

2.4 微服务网关设计

API Gateway 是微服务集群的流量入口，其核心设计逻辑如下：

动态路由：解析 HTTP 请求前缀（如 /api/auth），将请求精准转发至对应的内部容器端口。

服务屏蔽：对外仅暴露统一接口，后端微服务的真实 IP 与端口对外部网络完全不可见，实现了网络层面的安全隔离。

鉴权中间件：在请求到达具体微服务前，网关先行校验 JWT Token 的签名有效性，阻断非法请求，减轻后端服务的计算压力。

2.5 关键业务流程设计

本系统改变了传统教务管理中"学生逐个申请、教师逐个审批"的低效模式，设计了基于事件驱动的自动化处理流程。核心业务流转包含"活动发起"、"审核决策"与"自动执行"三个阶段。

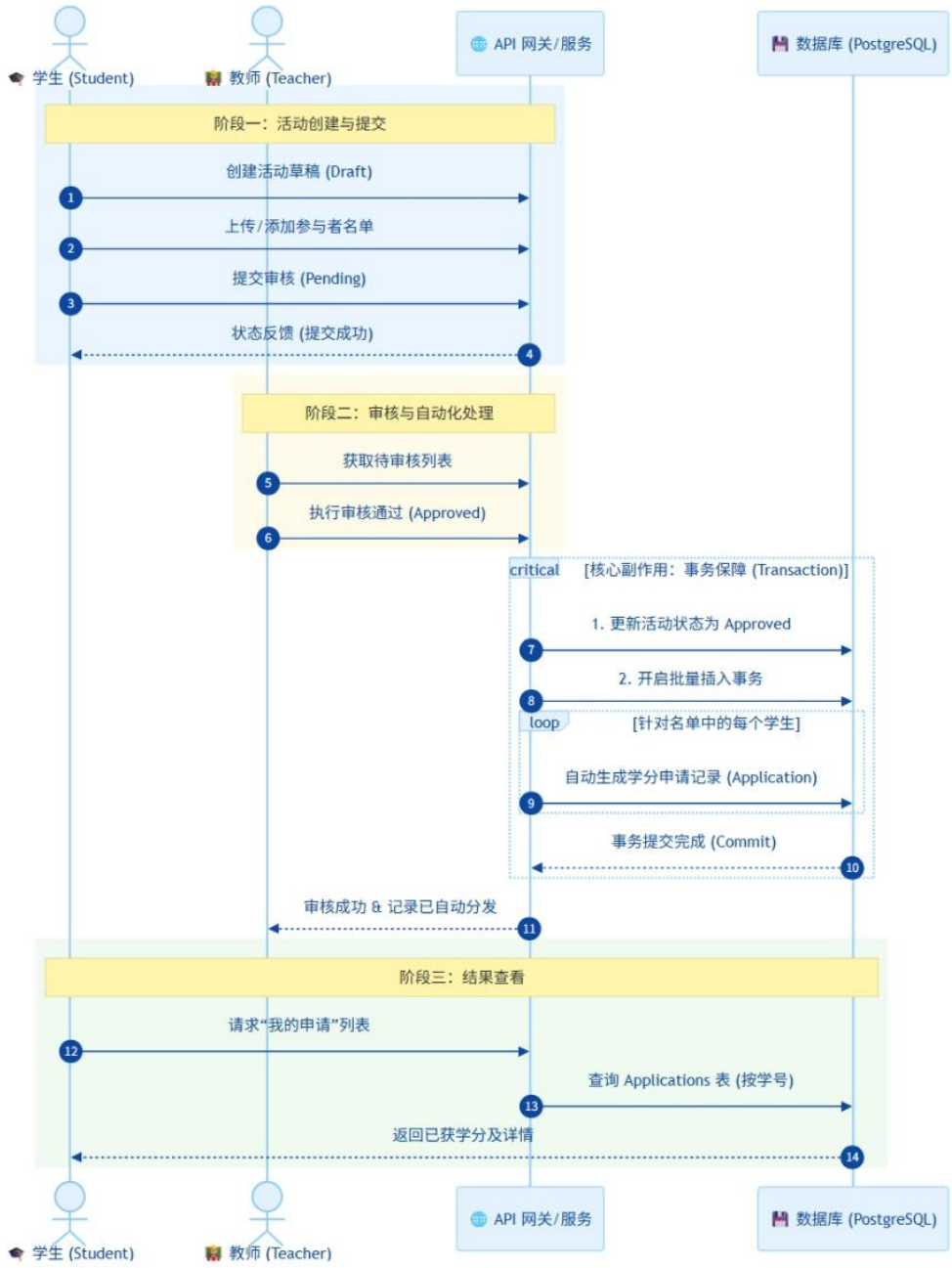

图 3 核心业务流程架构图

活动发起与提交：学生用户在前端创建活动草稿，支持批量导入参与者名单。确认信息无误后，将活动提交至系统，状态流转为 Pending（待审核）。

原子化审核决策：教师用户获取待审核列表并进行审批。一旦做出"通过"

(Approved) 决策，后端 API 不仅更新活动状态，更会立即触发副作用逻辑 (Side Effects)。

事务性自动生成：系统利用 PostgreSQL 的 原子事务 (Transaction) 机制保障数据一致性。在更新活动状态的同一事务上下文中，系统自动遍历该活动的所有参与者，批量生成对应的 Application 申请记录并授予学分。

该流程充分利用了 Go 语言的高并发处理能力与数据库的 ACID 特性，确保了在面对数百人规模的大型活动时，学分授予操作"要么全部成功，要么全部回滚"，彻底杜绝了数据不一致或部分丢失的问题。

第三章 核心算法与逻辑

3.1 基于 RBAC 的动态路由拦截算法

为了在前端实现严格的基于角色的访问控制（RBAC），系统设计了一套分层路由拦截算法见下图（图 4）。

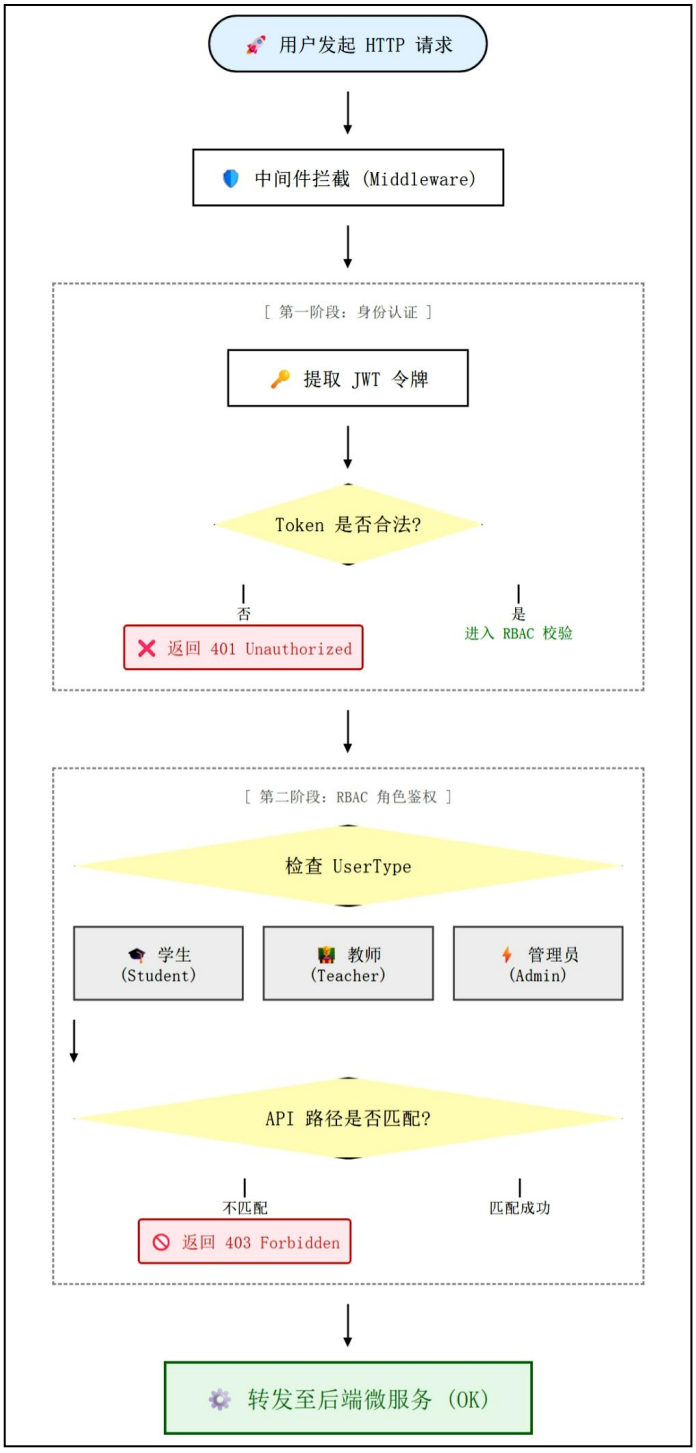

图 4 权限控制流程图

算法描述：

输入：当前用户的 JWT Token 及目标路由的元数据（allowedRoles，如 ["admin", "teacher"]）。

双重守卫机制：

Layer 1 (Authentication Guard)：校验 Token 是否存在且未过期。若失效，直接重定向至登录页。

Layer 2 (Authorization Guard)：解析 Token 载荷中的 role 字段，将其与路由元数据的白名单集合进行包含性运算。

决策执行：

若 UserRole ∈ AllowedRoles，则算法放行，渲染目标组件。

若验证失败，触发 HTTP 403 异常处理流程，跳转至"无权访问"提示页。

意义： 该算法实现了权限逻辑与业务代码的完全解耦，从视图层杜绝了越权访问。

3.2 线性表驱动的动态导航生成算法

系统的侧边栏菜单采用数据驱动视图（Data-Driven View）模式，而非硬编码。

算法逻辑：

定义函数 getMenuItems(userType)，内部维护一个基础菜单线性表。算法根据传入的 userType，以 O(1) 的时间复杂度动态向线性表中 Push 特定的菜单对象。

若角色为 Student，仅添加"我的申请"。

若角色为 Teacher，追加"学生列表"和"审批管理"。

若角色为 Admin，追加"系统设置"和"用户管理"。

该设计使得 UI 结构能随用户角色动态变化，提升了系统的灵活性和可维护性。

3.3 异构数据标准化适配策略

针对后端不同微服务可能返回的异构数据结构（例如，部分接口返回分页对象，部分返回扁平数组），前端实现了一套适配器模式（Adapter Pattern）。

实现策略：

系统拦截所有 Axios 响应，通过类型断言判断数据结构。

若数据为数组，封装为 { data: response, meta: { total: length } }。

若数据为分页对象，直接透传。

若数据为空，返回标准空结构。

这种数据清洗（Data Normalization）策略保证了前端通用表格组件（Table Component）的鲁棒性，无需针对每个接口编写独立的解析逻辑。

第四章 问题与解决方案

4.1 远程 SSH 安全运维与端口转发

问题描述：在开发过程中，本地客户端需直接连接运行在服务器容器内的数据库进行调试，但出于安全考虑，服务器防火墙未开放数据库端口。且初期配置 SSH 时频繁遭遇 Permission denied。

解决方案：

SSH 隧道技术：不直接暴露数据库端口，而是通过 ssh -L 命令建立加密隧道，将远程 localhost:5432 映射至本地端口。

code Bash

downloadcontent_copy

expand_less

```
ssh -p 33 -L 5432:localhost:5432 user@host -i ~/.ssh/id_ed25519
```

Linux 权限模型合规：针对密钥连接失败，经排查是由于私钥文件权限过于开放。遵循 Linux 安全模型的"最小权限原则"，执行 chmod 600 锁定私钥文件权限，仅允许当前用户读写，成功解决了安全校验失败的问题。

4.2 数据库连接与凭证安全加密

问题描述：项目初期测试时，用户密码曾以明文形式存储，这违反了安全规范，且无法通过安全审计。

解决方案：

在 User Service 中引入 Bcrypt 加密算法。在写入数据库前，对密码进行加盐（Salt rounds = 10）哈希处理。数据库仅存储以 $2a$10$ 开头的哈希字符串。即使数据库文件被窃取，攻击者也无法反推原始密码，极大地提升了数据安全性。

4.3 前端视图一致性与缓存控制

问题描述：用户在完成活动审批操作后，返回列表页时仍显示旧数据（状态未更新），导致用户误以为操作失败。

解决方案：

分析发现这是浏览器对 GET 请求的强缓存机制导致的。我们在前端 Axios 拦

截器中统一添加了 HTTP 头 Cache-Control: no-cache，强制浏览器每次请求都向服务器验证数据的新鲜度，确保了视图层与数据库层的强一致性。

4.4 跨环境部署的依赖冲突

问题描述：开发团队成员使用不同的操作系统（Windows/MacOS），而生产环境为 Debian Linux。由于 Node.js 和 Go 版本的差异，导致本地能运行的代码在服务器上构建失败。

解决方案：

彻底贯彻 Dockerfile 标准化交付。无论开发还是生产环境，均通过 Docker Compose 进行编排。利用 多阶段构建（Multi-stage Build） 技术，在镜像构建阶段统一编译环境，最终仅打包二进制文件到运行时镜像。这彻底解决了"在我机器上能跑"的经典问题，实现了环境的一致性。

第五章 实机测试

5.1 程序运行说明

本系统采用前后端分离架构，后端服务运行于 Debian Linux 服务器容器环境中，前端应用在本地开发环境中运行，通过 HTTP 协议与 SSH 隧道进行通信。

5.1.1 运行环境配置

1.服务端环境：

操作系统：Debian GNU/Linux 13 (Trixie)

内核版本：Linux 6.12.48+deb13-amd64

容器引擎：Docker Engine 24.0+ / Docker Compose v2.20+

数据库：PostgreSQL 15.4 (注：运行于 Docker 容器中)

2.客户端开发环境：

操作系统：Windows 10/11 或 macOS

开发工具：Visual Studio Code

运行时：Node.js 18.16+ (LTS), Go 1.21+

包管理器：NPM 9.0+

5.1.2 详细执行步骤

步骤 1：后端微服务集群启动

后端服务通过 Docker Compose 进行容器化编排。在服务器终端执行以下命令：

构建镜像与启动容器：（在项目根目录下执行）

```
docker-compose up -d --build
```

该命令会读取 docker-compose.yml，自动构建 API Gateway、Auth Service、User Service 等 Go 服务的二进制文件，并拉取 Postgres 和 Redis 镜像。

验证服务状态：

```
docker-compose ps
```

预期结果：所有 Service 状态显示为 Up，其中 API Gateway 监听 8080 端口。

步骤 2：前端应用启动 (本地客户端)
前端项目基于 Vite 构建，支持热重载（HMR）。
安装依赖：（进入 frontend 目录 cd frontend）
npm install

启动开发服务器：
npm run dev

执行后，Vite 将启动本地开发服务器，终端输出访问地址：http://localhost:5173（或 3000）。

步骤 3：数据库远程连接
由于服务器数据库端口未直接暴露到公网，需通过 SSH 隧道进行安全连接。
建立隧道：
在本地终端或 VS Code 中配置端口转发：
远程主机：9particle.top
远程端口：33 (SSH 端口)
转发规则：将远程 localhost:5432 映射到本地 127.0.0.1:5432。
命令行方式：
ssh -p 33 -L 5432:localhost:5432 emptydust@9particle.top -i ~/.ssh/id_ed25519

连接验证：
使用数据库管理工具（如 DBeaver 或 VS Code Database 插件）连接：
Host: 127.0.0.1
Port: 5432
Database: credit_management
User/Password: (根据 docker-compose.yml 环境变量配置)

5.1.3 系统访问

完成上述步骤后，打开浏览器访问前端地址。系统将自动请求本地代理或远程网关接口，用户可使用以下默认账号进行测试：
管理员: admin / admin123
学生: student / student123
教师: teacher / teacher123

5.2 测试与结果分析

5.2.1 SSH 连接与环境验证

通过 VS Code Remote SSH 插件连接服务器，验证 OS 版本。

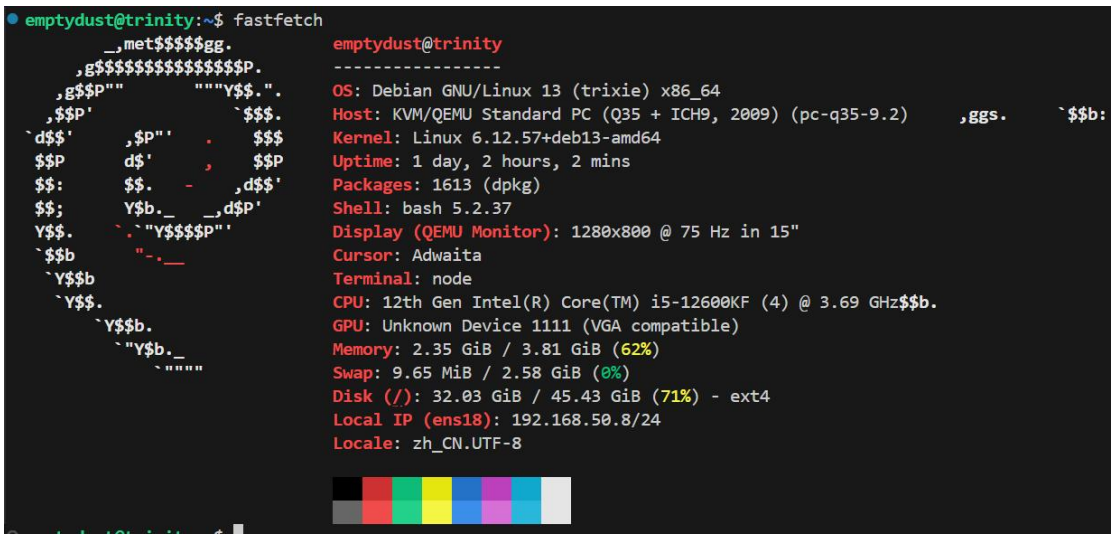

图 5 系统信息

5.2.2 数据库连接测试

配置 SSH 隧道后，使用本地数据库工具连接远程 Postgres。

```
emptydust@trinity:~/credit-management$ docker exec -it credit_management_postgres psql -U postgres
 -d credit_management
psql (15.15)
Type "help" for help.

credit_management=# \dt
                List of relations
 Schema |         Name          | Type  |  Owner
--------+-----------------------+-------+----------
 public | activity_participants | table | postgres
 public | applications          | table | postgres
 public | attachments           | table | postgres
 public | credit_activities     | table | postgres
 public | departments           | table | postgres
 public | users                 | table | postgres
(6 rows)
```

图 6 数据库表结构

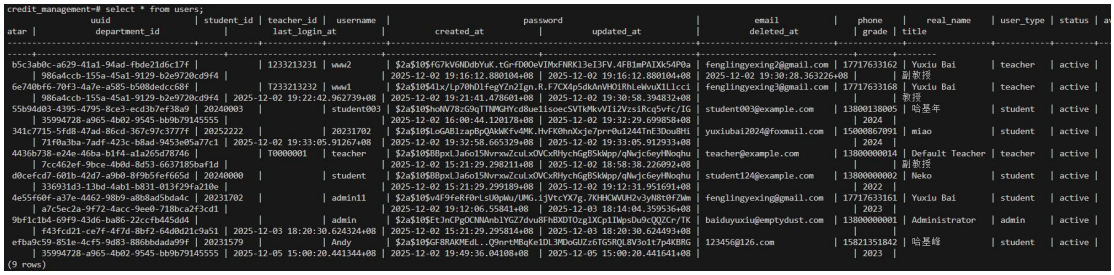

图 7 users 表中的加密密码数据

5.2.3 功能测试

用户登录：测试不同角色（学生/教师）登录，验证 JWT 生成。

图 8 登陆界面

活动创建：测试数据写入 credit_activities 表。

图 9 活动创建窗口

5.2.4 前端界面运行结果展示

基于开发过程，前端更新了多个版本，此处展示后期较完善的界面设计。

V1.0 版：

图 10 用户仪表板界面

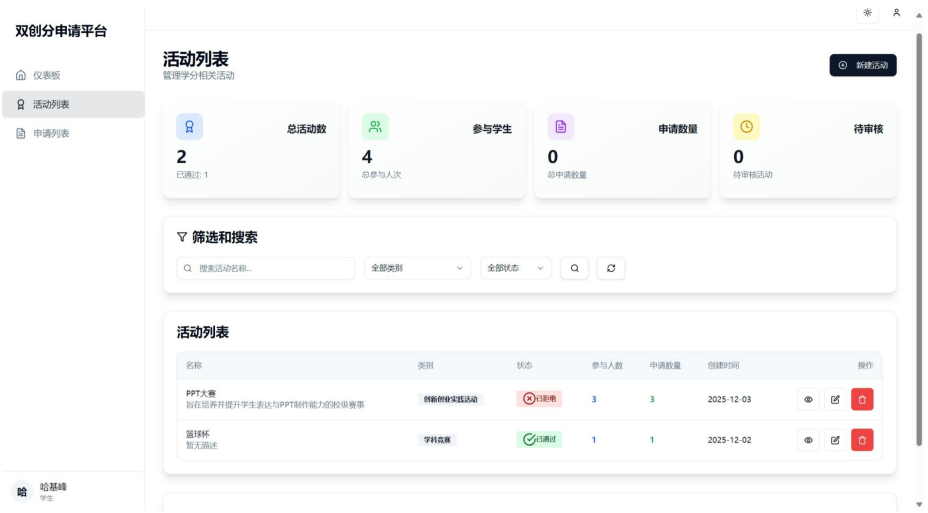

图 11 用户活动列表界面

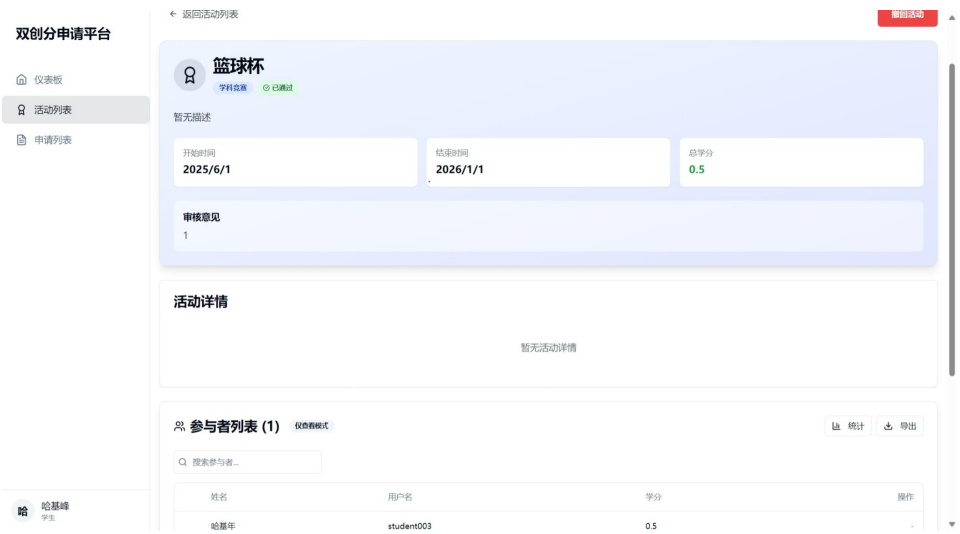

图 12 活动具体信息界面

V2.0 版：

图 13 管理员仪表板界面

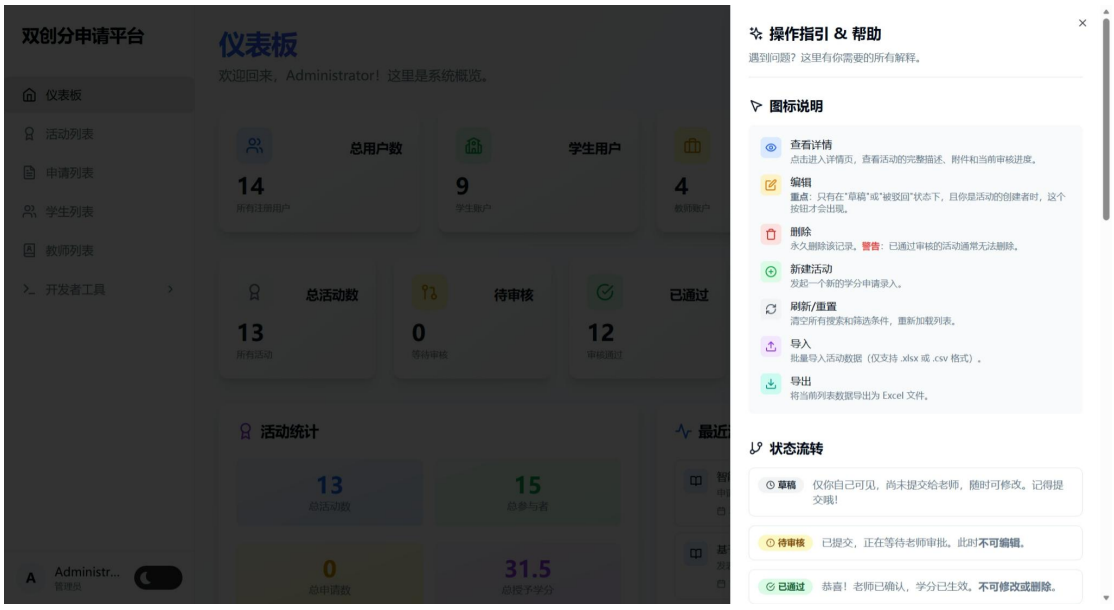

图 14 HELPME 指引界面 UI

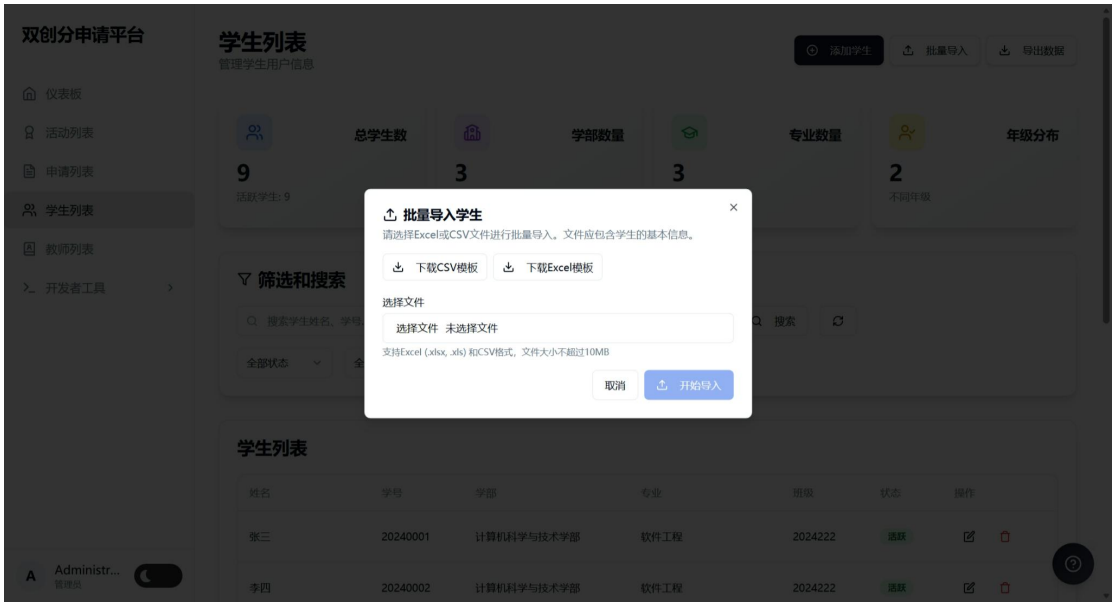

图 15 文件批量导入信息功能界面 UI

参考文献

[1] docs.docker.com. Docker Documentation[EB/OL]. (2025-12-18)[2025-12-21]. HTTPS://DOCS.DOCKER.COM/

[2] www.radix-ui.com. Radix UI[EB/OL]. (2023-06-21)[2025-12-21]. HTTPS://WWW.RADIX-UI.COM/

[3] Jinzhu. GORM The fantastic ORM library for Go[EB/OL]. (2013-01-01)[2025-12-21]. HTTPS://PKG.GO.DEV/GORM.IO/GORM

[4] 韦大善人. Go-Gin Web 框架完整教程[EB/OL]. (2024-12-25)[2025-12-21]. HTTPS://BLOG.CSDN.NET/WEIXIN_43825008/ARTICLE/DETAILS/144718647

[5] Folarin Lawal. Attention Required! | Cloudflare[EB/OL]. (2023-02-21)[2025-12-21]. HTTPS://UIVERSE.IO/MADFLOWS/FRESH-FIREANT-15