**软件学院2024-2025学年度第二学期**

**《软件工程理论与实践课程设计》**

**实验报告**

|  |  |
| --- | --- |
| **班级：** | 软件工程2301班 |
| **项目名称：** | 基于Web的互联网广告平台 |
| **项目组组长：** | 林浩 |
| **项目组成员：** | 学号：U202317253  姓名：林浩  学号：U202317258  姓名：吴语林  学号：U202313083  姓名：钟家意  学号：U202317263  姓名：赵子阳 |
| **指导老师：** | 刘小峰 |

目录

[1 项目概述 6](#_Toc200750368)

[2 需求描述 6](#_Toc200750369)

[2.1 功能性需求 6](#_Toc200750370)

[2.1.1 用户需求 6](#_Toc200750371)

[场景分析 8](#_Toc200750372)

[2.1.2 非功能性需求 11](#_Toc200750373)

[2.1.3场景分析 12](#_Toc200750374)

[新用户注册场景： 12](#_Toc200750375)

[用户登录场景： 13](#_Toc200750376)

[创建广告场景： 15](#_Toc200750377)

[账户充值场景： 17](#_Toc200750378)

[留言发布场景： 18](#_Toc200750379)

[广告审核场景： 20](#_Toc200750380)

[3 系统设计 22](#_Toc200750381)

[1.1. 架构设计（包括组件图和组件分析） 22](#_Toc200750382)

[1.2. 界面原型设计（包括组件分析和组件图） 23](#_Toc200750383)

[1.4数据库详细分析（包含E-R图） 28](#_Toc200750384)

**[4.1技术选型及技术选型描述](#_Toc200750385)** [31](#_Toc200750385)

**[开发语言](#_Toc200750386)** [31](#_Toc200750386)

**[框架和库](#_Toc200750387)** [32](#_Toc200750387)

**[4.2 router路由](#_Toc200750388)** [32](#_Toc200750388)

[5.讨论区功能 34](#_Toc200750389)

**[前端实现-状态管理](#_Toc200750390)** [34](#_Toc200750390)

**[核心功能实现-发帖功能](#_Toc200750391)** [35](#_Toc200750391)

**[审核功能](#_Toc200750392)** [37](#_Toc200750392)

**[技术特点](#_Toc200750393)** [39](#_Toc200750393)

**[6.支付系统架构](#_Toc200750394)** [40](#_Toc200750394)

**[前端实现-状态管理](#_Toc200750395)** [40](#_Toc200750395)

**[后端实现-微信支付](#_Toc200750396)** [41](#_Toc200750396)

**[核心功能实现-充值功能](#_Toc200750397)** [41](#_Toc200750397)

**[后端实现-余额管理](#_Toc200750398)** [42](#_Toc200750398)

**[发票系统](#_Toc200750399)** [43](#_Toc200750399)

**[API接口分析](#_Toc200750400)** [44](#_Toc200750400)

[限制说明 44](#_Toc200750401)

**[7.用户管理](#_Toc200750402)** [45](#_Toc200750402)

[数据模型分析 45](#_Toc200750403)

[核心功能实现-用户注册 45](#_Toc200750404)

[用户管理（管理员功能） 46](#_Toc200750405)

[API接口 48](#_Toc200750406)

[49](#_Toc200750407)

[数据流图 49](#_Toc200750408)

[8.广告购买用户界面 49](#_Toc200750409)

[前端实现-界面结构 49](#_Toc200750410)

[广告购买表单状态管理 49](#_Toc200750411)

[广告购买表单界面 50](#_Toc200750412)

[后端实现-接口定义 51](#_Toc200750413)

[业务处理逻辑 52](#_Toc200750414)

[用户界面特点：直观的表单布局 53](#_Toc200750415)

[9.系统界面展示 54](#_Toc200750416)

[检查余额不足 54](#_Toc200750417)

[支持微信 55](#_Toc200750418)

[成功后会有历史记录 55](#_Toc200750419)

[购买后的发票管理 56](#_Toc200750420)

[开发票 56](#_Toc200750421)

[用户管理 57](#_Toc200750422)

[管理员审核记录 58](#_Toc200750423)

[测试用例 59](#_Toc200750424)

[广告投放完整流程 59](#_Toc200750425)

[并发操作场景 60](#_Toc200750426)

[网络异常场景 61](#_Toc200750427)

[测试覆盖率目标 62](#_Toc200750428)

[10.总结 62](#_Toc200750429)

## 项目概述

项目基本介绍：

nonsence 是一个广告投放管理平台，主要功能包括广告创建与发布、广告效果监控、用户账户管理、资金充值管理、发票管理、广告素材管理、消息留言管理、数据统计分析等。平台支持广告主发布广告、设置预算、上传素材、查看效果数据，同时提供管理员进行内容审核、系统管理、用户管理等功能。系统采用前后端分离架构，具有完整的财务管理体系，支持广告投放全流程管理，包括广告审核、投放监控、效果分析等，并提供了安全可靠的用户认证和数据处理机制，是一个功能完整的广告投放解决方案。

Github仓库地址：

<https://github.com/miwumiwumilumilelu/nonsence.git>

项目分工：

**林浩：后端开发人员兼项目经理users (认证授权), accounts (账户余额、充值、流水), invoicing (发票申请模型、API、Admin 配置)**

**吴语林：后端开发人员ads (广告管理核心), delivery (广告投放接口), reporting (基础统计)**

**钟家意：前端开发人员 广告主端 (Advertiser Portal) 核心功能**

**赵子阳：前端开发人员 管理员端 (Admin Portal) (如果 Django Admin 不足) + 广告主端 的 效果报告 部分 + 通用组件/库**

## 需求描述

## 功能性需求

## 用户需求

**【作者：林浩、吴语林、钟家意、赵子阳】**

**许多的地图

AI 生成的内容可能不正确。**

**1. 广告主**

**广告管理需求**

* 创建广告
* 提交审核
* 查看广告状态（包括待审核/通过/拒绝）
* 暂停或恢复广告

**财务管理需求**

* 能够充值账户
* 能够查看充值记录
* 申请发票（简化版）

**效果分析需求**

* 能够查看广告的基础数据（总容量、总点击量）

**2. 平台管理员**

**审核需求**

* 能够查看待审核广告列表
* 审核广告（通过/拒绝）
* 填写拒绝理由

**发票管理需求**

* 能够查看所有发票申请列表
* 将发票申请标记为“已处理”

**用户管理需求**

* 可能需要管理用户（如禁用账户等）

## 场景分析

|  |  |
| --- | --- |
| 用例名 | 用户登录 |
| 参与者 | 广告客户 |
| 前置条件 | 用户进入页面 |
| 激励 | 用户点击登录键 |
| 场景 | 1. 用户点击账户密码输入行进行输入   2．用户点击登录 |
| 异常 | 异常1：账号或者密码错误  系统提示请输入正确账户或密码  异常2：用户没有账号，输入账户不存在  系统提示账户不存在，请注册，完成注册流程 |
| 后置条件 | 用户成功登录 |

|  |  |
| --- | --- |
| 用例名 | 广告充值 |
| 参与者 | 广告客户 |
| 前置条件 | 用户登录成功 |
| 激励 | 用户点击充值键 |
| 场景 | 1.用户点击充值键   1. 系统显示充值界面（仅支持微信），选择充值金额（50，100，500，1000，自定义） 2. 用户点击微信充值   3.1用户选择预定金额或自定义金额，点充值  3.2用户打开新界面,显示已充值  3.3系统显示操作成功 |
| 异常 | 异常1：点击充值前未选择金额、  系统提示选择金额  异常2：充值页面超时，为成功操作  系统自动返回上一页 |
| 后置条件 | 广告客户的账户余额增加充值金额 |

|  |  |
| --- | --- |
| 用例名 | 评论区留言 |
| 参与者 | 广告客户 |
| 前置条件 | 用户进入评论区 |
| 激励 | 用户点击评论键 |
| 场景 | 1. 用户选择想评论的评论或直接评论 2. 用户点击输入框并输入 3. 用户点击发送键 4. 显示评论待审核 5. 评论成功 |
| 异常 | 异常1：未输入就点击发送  系统提示输入不能为空  异常2：评论的帖子被删除  系统自动提示评论已被删除 |
| 后置条件 | 用户发出的评论交给管理员审核 |

**系统需求**

**【作者：林浩、吴语林、钟家意、赵子阳】**

**1.功能性需求（Functional Requirements）**

**1.1用户管理**

* + 支持广告主（Advertiser）和管理员（Admin）的注册、登录、登出
  + 用户角色区分：广告主只能管理自己的广告和账户，管理员可以审核广告和处理发票

**1.2广告管理、**

**广告主可以：**

* + 创建广告（名称、目标 URL、素材图片、预算、投放时间）
  + 提交广告审核
  + 查看广告状态（待审核/通过/拒绝/投放中/暂停/结束）
  + 暂停或恢复已通过的广告

**管理员可以：**

* + 查看待审核广告列表
  + 审核广告（通过/拒绝，附带简单理由）

**1.3账户与充值**

**广告主可以：**

* + 模拟充值（输入金额，后端直接更新余额）
  + 查看充值历史记录（时间、金额、状态）

**系统需记录每次充值流水，并实时更新账户余额**

**1.4 广告投放与展示统计**

**提供广告投放 API：**

* + 接收 **placement\_id** 参数，返回可投放广告（随机/轮询）
  + 每次调用 API 时，记录展示次数（Impression）到数据库

**广告主可查看广告的基础数据（总展示量、总点击量）**

**1.5发票管理（简化版）**

**广告主可以：**

* + 申请发票（选择时间段，系统记录申请）
  + 查看发票申请状态（待处理/已处理）

**管理员可以：**

* + 查看所有发票申请列表（Django Admin）
  + 标记发票申请为“已处理”

## 非功能性需求

**【作者：林浩、吴语林、钟家意、赵子阳】**

**图示

AI 生成的内容可能不正确。**

**1. 性能需求**

* + 系统响应时间：广告投放 API 平均响应时间 ≤ 200ms（直接查询 PostgreSQL），前端页面加载时间 ≤ 2s（Vue.js 优化）
  + 并发处理能力：MVP 阶段支持 **100 QPS（每秒查询数）** 的广告请求（无需 Redis，直接依赖 PostgreSQL）

**2. 安全性需求**

* + 认证与授权：使用 Django 内置的 **Session/Cookie** 或 **JWT** 进行用户认证，广告主只能访问自己的广告数据（RBAC 权限控制）
  + 数据安全：用户密码需 **哈希存储**（Django 默认使用 PBKDF2），广告素材上传需限制文件类型（仅图片格式如 JPG/PNG）

**3. 可用性需求**

* + 系统应保证 **99% 的正常运行时间**（部署在稳定云服务如 AWS/阿里云）
  + **广告投放 API 需具备 基本容错能力（如数据库查询失败时返回默认广告或错误提示）**

**4. 可维护性需求**

* + 代码结构：前后端分离（Django REST + Vue.js），便于独立开发和部署，按功能模块划分 Django Apps（**users**, **ads**, **accounts**, **delivery** 等）
  + 日志与监控：记录关键操作（如广告审核、充值、API 调用），简单的错误日志（Django Logging）

**5. 数据存储需求**

* + 使用 **PostgreSQL** 存储：用户数据（账号、角色）、广告数据（名称、预算、状态）、充值记录、广告展示日志（Impression）、发票申请记录
  + **数据一致性**：充值、广告扣费需保证事务（ACID），广告展示统计允许最终一致性（直接写入数据库，无 Redis 缓存）

## 2.1.3场景分析

### 新用户注册场景：

|  |
| --- |
| TypeScript describe('用户注册测试', () => {  test('场景一：正常注册流程', () => {  const testCase = {  输入: {  username: 'testuser001',  password: 'Test123456'  },  预期结果: {  status: 200,  message: '注册成功',  data: {  userId: expect.any(String),  username: 'testuser001'  }  }  };  }); |

**场景一：正常注册流程**

* **参与者 (Actor):** 首次访问平台并希望创建账户的新用户。
* **前置条件 (Precondition):** 用户位于平台的注册页面。
* **事件流 (Flow of Events):**
  1. 用户输入一个之前未被注册过的用户名（例如，'testuser001'）。
  2. 用户输入一个符合系统安全策略的密码（例如，'Test123456'）。
  3. 用户提交注册表单。
* **后置条件 (Postcondition):**
  1. 系统验证用户名在数据库中是唯一的。
  2. 系统成功创建一个新的用户账户，并将用户信息存储到数据库中。
  3. 系统向前端返回一个成功的响应，状态码为200，并附带"注册成功"的消息和一个包含新用户ID及用户名的对象。

|  |
| --- |
| test('场景二：重复用户名注册', () => {  const testCase = {  输入: {  username: 'existinguser',  password: 'Test123456'  },  预期结果: {  status: 400,  error: '用户名已存在'  }  };  });  }); |

**场景二：重复用户名注册**

* **参与者 (Actor):** 尝试使用一个已存在用户名进行注册的新用户。
* **前置条件 (Precondition):** 用户位于平台的注册页面，且某个用户名（例如，'existinguser'）已经被其他用户注册。
* **事件流 (Flow of Events):**
  1. 用户输入一个已经存在于系统中的用户名（'existinguser'）。
  2. 用户输入任意密码。
  3. 用户提交注册表单。
* **后置条件 (Postcondition):**
  1. 系统在数据库中检测到该用户名已被占用。
  2. 系统拒绝创建新账户。
  3. 系统向前端返回一个客户端错误的响应，状态码为400，并附带明确的错误信息，如"用户名已存在"，以提示用户更换用户名。

### 用户登录场景：

|  |
| --- |
| TypeScript  describe('用户注册测试', () => {  test('场景一：正常注册流程', () => {  const testCase = {  输入: {  username: 'testuser001',  password: 'Test123456'  },  预期结果: {  status: 200,  message: '注册成功',  data: {  userId: expect.any(String),  username: 'testuser001'  }  }  };  }); |
| test('场景二：重复用户名注册', () => {  const testCase = {  输入: {  username: 'existinguser',  password: 'Test123456'  },  预期结果: {  status: 400,  error: '用户名已存在'  }  };  });  }); |

**场景一：使用有效的凭据成功登录**

* **参与者 (Actor):** 已注册的平台用户（广告主或媒体主）。
* **前置条件 (Precondition):** 用户已经成功注册并拥有一个有效的账户。用户位于登录页面。
* **事件流 (Flow of Events):**
  1. 用户在用户名输入框中输入已注册的、正确的用户名（例如，'existinguser'）。
  2. 用户在密码输入框中输入与该用户名匹配的、正确的密码。
  3. 用户点击“登录”按钮。
* **后置条件 (Postcondition):**
  1. 系统验证用户输入的用户名和密码与数据库中存储的记录一致。
  2. 系统为该用户创建一个会话（Session），并生成一个访问令牌（Token）。
  3. 系统将用户重定向到其个人主页或平台仪表盘（Dashboard）。
  4. 系统向前端返回一个成功的响应，例如状态码200，并附带用户信息和访问令牌。

**场景二：使用错误的密码登录**

* **参与者 (Actor):** 已注册的平台用户。
* **前置条件 (Precondition):** 用户拥有一个有效的账户。用户位于登录页面。
* **事件流 (Flow of Events):**
  1. 用户输入正确的用户名。
  2. 用户输入一个错误的密码。
  3. 用户点击“登录”按钮。
* **后置条件 (Postcondition):**
  1. 系统验证发现密码与存储的记录不匹配。
  2. 系统拒绝登录请求。
  3. 系统在登录页面上显示一条明确的错误提示信息，例如“用户名或密码错误”。
  4. 系统向前端返回一个认证失败的响应，例如状态码401（Unauthorized）。

**场景三：使用不存在的用户名登录**

* **参与者 (Actor):** 任何试图登录的用户。
* **前置条件 (Precondition):** 用户输入的用户名未在平台注册。用户位于登录页面。
* **事件流 (Flow of Events):**
  1. 用户输入一个从未在系统中注册过的用户名。
  2. 用户输入任意密码。
  3. 用户点击“登录”按钮。
* **后置条件 (Postcondition):**
  1. 系统在数据库中查找该用户名，但未找到匹配的记录。
  2. 系统拒绝登录请求。
  3. 为安全起见，系统应显示与密码错误时相同的通用错误提示，例如“用户名或密码错误”，以防止攻击者通过错误信息判断用户名是否存在。
  4. 系统向前端返回一个认证失败的响应，例如状态码401（Unauthorized）。

### 创建广告场景：

|  |
| --- |
| TypeScript  describe('广告创建测试', () => {  test('场景一：正常创建广告', () => {  const testCase = {  前置条件: {  用户已登录: true,  账户余额: 1000  },  输入: {  userId: 'user123',  type: '图文广告',  name: '夏季促销活动',  budget: 500,  start: '2024-06-01',  end: '2024-06-30',  material: 'http://example.com/ad.jpg'  },  预期结果: {  status: 200,  message: '广告创建成功',  data: {  adId: expect.any(String),  status: '待审核',  remainingBalance: 500  }  }  };  });  test('场景二：余额不足创建广告', () => {  const testCase = {  前置条件: {  用户已登录: true,  账户余额: 100  },  输入: {  userId: 'user123',  type: '图文广告',  name: '夏季促销活动',  budget: 500,  start: '2024-06-01',  end: '2024-06-30',  material: 'http://example.com/ad.jpg'  },  预期结果: {  status: 400,  error: '账户余额不足'  }  };  });  }); |

**场景一：正常创建广告**

* **参与者 (Actor):** 已登录的广告主用户。
* **前置条件 (Precondition):**
  1. 用户已经成功登录到广告平台。
  2. 用户的账户余额充足，足以支付或冻结所设定的广告预算（例如，账户有1000元，预算500元）。
* **事件流 (Flow of Events):**
  1. 用户进入创建广告的界面。
  2. 用户填写广告的详细信息，包括广告名称、类型、预算金额、投放起止日期以及广告素材链接。
  3. 用户提交创建请求。
* **后置条件 (Postcondition):**
  1. 系统验证用户输入的数据有效，并检查账户余额是否满足预算要求。
  2. 系统在数据库中成功创建一个新的广告记录，并将其初始状态设置为“待审核”。
  3. 系统从用户账户中预扣除或冻结相应的预算金额（500元）。
  4. 系统向前端返回一个成功的响应（状态码 200），包含“广告创建成功”的消息、新生成的广告ID以及更新后的账户余额。

**场景二：因账户余额不足导致创建失败**

* **参与者 (Actor):** 已登录的广告主用户。
* **前置条件 (Precondition):**
  1. 用户已经成功登录。
  2. 用户的账户余额不足以支付所设定的广告预算（例如，账户只有100元，但预算设为500元）。
* **事件流 (Flow of Events):**
  1. 用户填写广告创建表单，设置了一个高于其账户余额的预算。
  2. 用户提交创建请求。
* **后置条件 (Postcondition):**
  1. 系统在处理请求时，发现用户的账户余额小于其设定的预算。
  2. 系统拒绝创建广告的请求，不会在数据库中生成新的广告记录。
  3. 用户的账户余额保持不变。
  4. 系统向前端返回一个客户端错误响应（状态码 400），并附带明确的错误信息，如“账户余额不足”，提示用户失败原因。

### 账户充值场景：

|  |
| --- |
| TypeScript  describe('账户充值测试', () => {  test('场景一：微信支付充值', () => {  const testCase = {  前置条件: {  用户已登录: true,  初始余额: 0  },  输入: {  userId: 'user123',  amount: 1000,  method: '微信支付'  },  预期结果: {  status: 200,  message: '充值成功',  data: {  orderId: expect.any(String),  newBalance: 1000,  rechargeAmount: 1000  }  }  };  });  test('场景二：充值金额无效', () => {  const testCase = {  前置条件: {  用户已登录: true  },  输入: {  userId: 'user123',  amount: -100,  method: '微信支付'  },  预期结果: {  status: 400,  error: '充值金额无效'  }  };  });  }); |

**场景一：微信支付充值**

* **参与者 (Actor):** 已登录的用户。
* **前置条件 (Precondition):**
  1. 用户已经成功登录到广告平台。
  2. 用户的初始账户余额为0（便于验证充值后的变化）。
* **事件流 (Flow of Events):**
  1. 用户进入账户充值界面。
  2. 用户选择“微信支付”作为充值方式。
  3. 用户输入充值金额1000元。
  4. 用户确认并提交充值请求。
  5. 系统引导用户到微信支付页面完成支付操作。
  6. 微信支付成功后，回调通知广告平台。
* **后置条件 (Postcondition):**
  1. 系统成功接收并处理微信支付成功的回调。
  2. 用户的账户余额从0更新为1000元。
  3. 系统生成并记录一笔充值订单，包含订单ID和充值金额。
  4. 系统向前端返回一个成功的响应（状态码 200），包含“充值成功”的消息、新生成的订单ID以及更新后的账户余额。

**场景二：充值金额无效**

* **参与者 (Actor):** 已登录的用户。
* **前置条件 (Precondition):**
  + 用户已经成功登录到广告平台。
* **事件流 (Flow of Events):**
  1. 用户进入账户充值界面。
  2. 用户选择任意充值方式（例如微信支付）。
  3. 用户输入一个无效的充值金额（例如：-100）。
  4. 用户确认并提交充值请求。
* **后置条件 (Postcondition):**
  + 系统在处理请求时，验证到输入的充值金额为无效值。
  + 系统拒绝该充值请求，不会进行任何支付流程，也不会生成充值订单。
  + 用户的账户余额保持不变。
  + 系统向前端返回一个客户端错误响应（状态码 400），并附带明确的错误信息，如“充值金额无效”，提示用户失败原因。

### 留言发布场景：

|  |
| --- |
| TypeScript  describe('留言发布测试', () => {  test('场景一：正常发布留言', () => {  const testCase = {  输入: {  nickname: '张三',  content: '这是一条测试留言',  type: 1  },  预期结果: {  status: 200,  message: '留言提交成功，等待审核',  data: {  messageId: expect.any(String),  status: 0  }  }  };  });  test('场景二：内容超长留言', () => {  const testCase = {  输入: {  nickname: '张三',  content: '超过1000字的内容...',  type: 1  },  预期结果: {  status: 400,  error: '留言内容过长'  }  };  });  }); |

**场景一：正常发布留言**

* **参与者 (Actor):** 网站访客或已登录用户。
* **前置条件 (Precondition):**
  1. 用户能够访问留言发布页面。
  2. 用户输入的昵称和内容符合系统规定的格式和长度要求（例如，内容在允许的最大长度范围内）。
* **事件流 (Flow of Events):**
  1. 用户进入留言发布页面。
  2. 用户在“昵称”字段输入\*\*“张三”\*\*。
  3. 用户在“内容”字段输入\*\*“这是一条测试留言”\*\*。
  4. 用户选择留言类型为**1**（例如，代表“广告咨询”或“建议”）。
  5. 用户点击“提交”按钮发布留言。
* **后置条件 (Postcondition):**
  1. 系统成功接收并验证留言数据。
  2. 系统在数据库中**创建一条新的留言记录**。
  3. 新创建的留言记录的**状态设置为“待审核”**（status: 0），表示留言不会立即公开显示。
  4. 系统向前端返回一个**成功的响应**（状态码 200），包含“留言提交成功，等待审核”的消息，以及新生成的留言ID。

**场景二：内容超长留言**

* **参与者 (Actor):** 网站访客或已登录用户。
* **前置条件 (Precondition):**
  + 用户能够访问留言发布页面。
  + 系统对留言内容有明确的最大长度限制（例如，1000字）。
* **事件流 (Flow of Events):**
  1. 用户进入留言发布页面。
  2. 用户在“昵称”字段输入\*\*“张三”\*\*。
  3. 用户在“内容”字段输入**超过系统允许最大长度的文本**（例如，“超过1000字的内容...”）。
  4. 用户选择留言类型为**1**。
  5. 用户点击“提交”按钮发布留言。
* **后置条件 (Postcondition):**
  + 系统在接收或处理请求时，**检测到留言内容超过了预设的长度限制**。
  + 系统**拒绝创建留言记录**，不会将其存储到数据库中。
  + 系统向前端返回一个**客户端错误响应**（状态码 400），并附带明确的错误信息，如“留言内容过长”，提示用户失败原因。

### 广告审核场景：

|  |
| --- |
| TypeScript  describe('广告审核测试', () => {  test('场景一：通过广告审核', () => {  const testCase = {  前置条件: {  管理员已登录: true,  广告状态: '待审核'  },  输入: {  adId: 'ad123',  action: 'approve',  adminId: 'admin001'  },  预期结果: {  status: 200,  message: '审核通过',  data: {  adId: 'ad123',  status: '已通过',  auditTime: expect.any(String)  }  }  };  });  test('场景二：拒绝广告审核', () => {  const testCase = {  前置条件: {  管理员已登录: true,  广告状态: '待审核'  },  输入: {  adId: 'ad123',  action: 'reject',  reason: '广告内容不合规',  adminId: 'admin001'  },  预期结果: {  status: 200,  message: '审核拒绝',  data: {  adId: 'ad123',  status: '已拒绝',  reason: '广告内容不合规',  auditTime: expect.any(String)  }  }  };  });  }); |

**场景一：通过广告审核**

* **参与者 (Actor):** 已登录的管理员用户。
* **前置条件 (Precondition):**
  1. 管理员已成功登录到广告平台的后台管理系统。
  2. 系统中存在一条\*\*状态为“待审核”\*\*的广告记录。
* **事件流 (Flow of Events):**
  1. 管理员在后台管理界面找到需要审核的广告（ID为 'ad123'）。
  2. 管理员审阅广告内容，确认其符合平台规定。
  3. 管理员选择\*\*“通过”审核操作\*\*。
  4. 管理员提交审核请求，同时提供自己的管理员ID ('admin001')。
* **后置条件 (Postcondition):**
  1. 系统接收并处理审核通过请求。
  2. 系统将该广告的**状态更新为“已通过”**。
  3. 系统记录**审核通过的时间**。
  4. 系统可能触发后续流程，例如广告上线投放（如果预算充足且达到投放时间）。
  5. 系统向前端返回一个**成功的响应**（状态码 200），包含“审核通过”的消息、广告ID、新的状态以及审核时间。

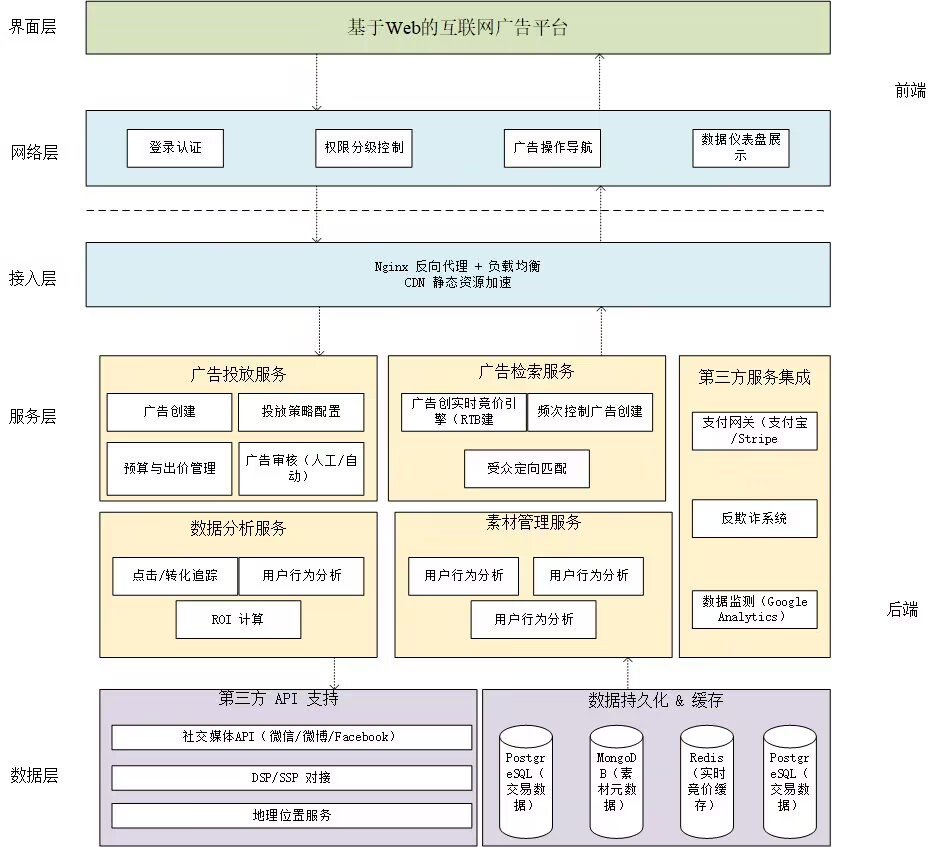
**场景二：拒绝广告审核**

* **参与者 (Actor):** 已登录的管理员用户。
* **前置条件 (Precondition):**
  + 管理员已成功登录到广告平台的后台管理系统。
  + 系统中存在一条\*\*状态为“待审核”\*\*的广告记录。
* **事件流 (Flow of Events):**
  1. 管理员在后台管理界面找到需要审核的广告（ID为 'ad123'）。
  2. 管理员审阅广告内容，发现其不符合平台规定（例如，内容违规、图片不清晰等）。
  3. 管理员选择\*\*“拒绝”审核操作\*\*。
  4. 管理员输入拒绝原因，例如“广告内容不合规”。
  5. 管理员提交审核请求，同时提供自己的管理员ID ('admin001')。
* **后置条件 (Postcondition):**
  + 系统接收并处理审核拒绝请求。
  + 系统将该广告的**状态更新为“已拒绝”**。
  + 系统**记录拒绝原因**。
  + 系统记录**审核拒绝的时间**。
  + 系统向前端返回一个**成功的响应**（状态码 200），包含“审核拒绝”的消息、广告ID、新的状态、拒绝原因以及审核时间。

## 系统设计

### 架构设计（包括组件图和组件分析）

**【作者：林浩、吴语林、钟家意、赵子阳】**



**前端**：

1. 界面层：界面层是用户与系统交互的区域，包括用户界面、数据输入、输出等。这一层主要负责将用户请求传递给下一层，并将处理结果返回给用户
2. 网络层：包括对登录认证、极限分级控制、广告操作导航、数据仪表盘展示
3. 技术栈：Vue 3 + Vue Router + Pinia（状态管理）
4. **Advertiser Portal**：广告创建、充值、数据看板
5. **Admin Portal**：广告审核、发票管理（或直接复用 Django Admin）

**后端**：

1. 服务层：广告投放服务、广告检索服务、数据分析服务、素材管理服务、第三方服务集成
2. 数据层：第三方api支持以及数据的持久化与缓存
3. 技术栈：Django 4 + Django REST Framework + PostgreSQL
4. Views：接收请求，调用 Services
5. Services：核心业务逻辑（如广告匹配、扣费）
6. Models：数据库 ORM 定义
7. Serializers：数据序列化/验证

### 界面原型设计（包括组件分析和组件图）

**【作者：林浩、吴语林、钟家意、赵子阳】**

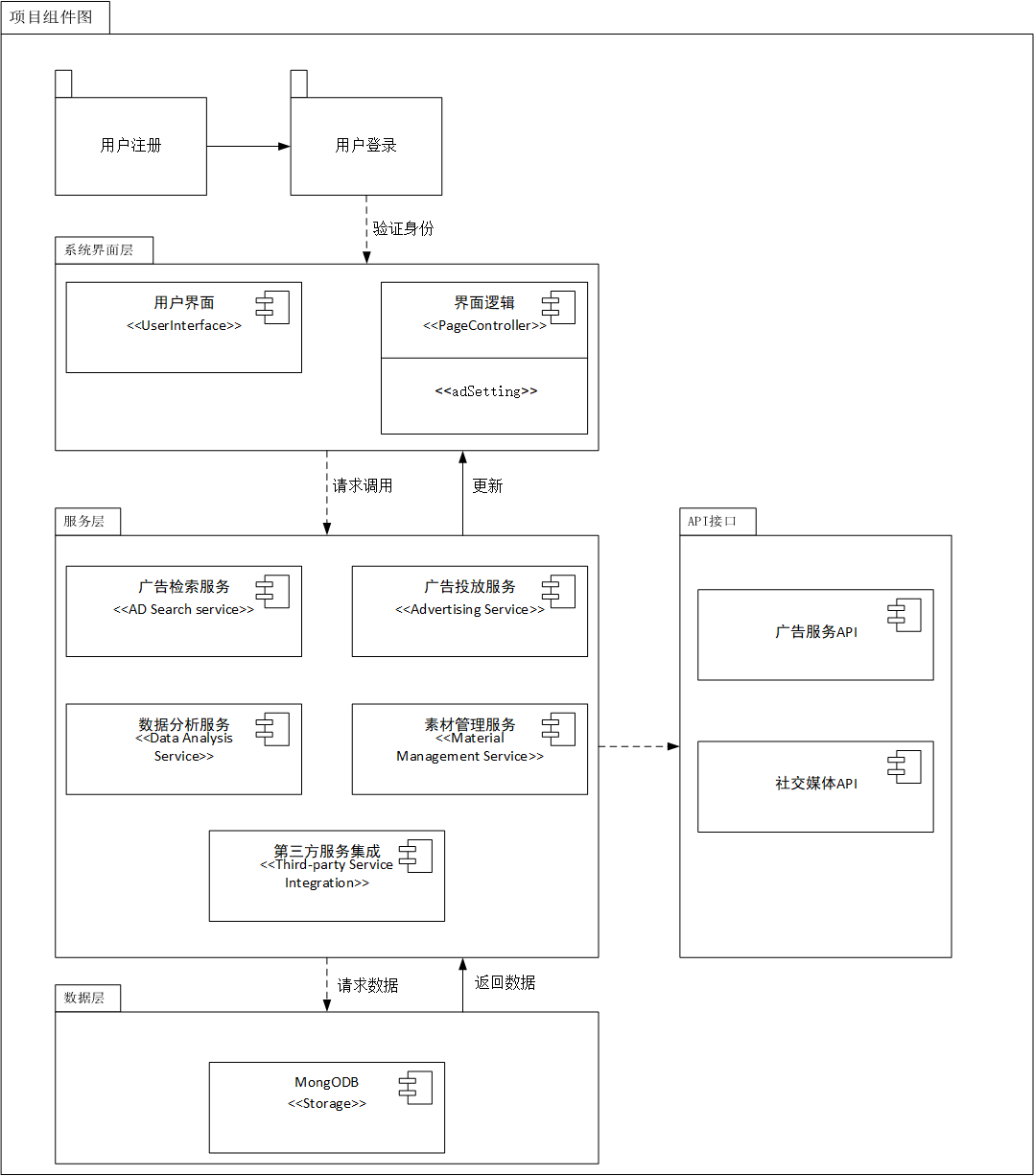
* + 1. **主要界面介绍**

**登录与注册**

**项目管理与文件管理**

* + 1. **具体界面原型设计**

1. **整体框架（组件图）**



组件分析：

核心组件：App.js 用来实现网站的主入口组件

Navbar.js 导航栏组件

Footer.js 页脚组件

主要功能页面：Home.js 首页组件

Messageboard.js 留言板组件

Clientcenter.js 客户中心组件

Admincenter.js 管理员中心组件

Cases.js 案例展示组件

Contact.js 联系页面组件

About.js 关于页面组件

ApiDoc.js API文档页面

每个组件都有对应的CSS文件，使用了模块化的CSS结构，方便管理和浏览

**登录/注册页面**

* **简洁表单：仅需邮箱/手机+密码**
* **社交账号快捷登录入口**
* **密码强度实时提示**
* **注册流程控制在3步内**

**广告主控制台（Dashboard）**

* 顶部数据看板：账户余额+关键指标（展示量/点击量）
* 快捷操作区：创建广告/充值/发票申请
* 广告状态可视化：使用标签云展示不同状态广告占比
* 最近活动时间线

**广告创建页**

* 分步表单设计：
  1. 基础信息（名称/URL）
  2. 素材上传（拖拽区域+尺寸提示）
  3. 预算设置（可视化滑块）
  4. 排期选择（日历组件）
* 实时预算计算器
* 草稿自动保存功能

**广告管理列表**

* 可折叠的筛选条件栏
* 卡片式广告展示（包含缩略图）
* 批量操作按钮组
* 状态标签颜色编码：
  + 待审核：黄色
  + 投放中：绿色
  + 已拒绝：红色

**广告详情页**

* 左栏：广告基础信息（不可编辑模式）
* 右栏：数据图表（展示量趋势图）
* 底部操作区：暂停/编辑/复制按钮
* 审核记录展示（管理员拒绝时显示原因）

**充值中心**

* 模拟支付流程：
  1. 输入金额（带建议充值档位）
  2. 选择支付方式（仅展示"模拟支付"）
  3. 支付成功动画
* 余额实时显示
* 最近充值记录快捷查看

**发票管理**

* 申请表单：
  + 时间范围选择器
  + 自动计算可开票金额
  + 发票抬头智能填充
* 状态追踪进度条
* 历史记录可下载为CSV

**管理员审核页**

* 双栏布局：
  + 左栏：待审核广告列表（带缩略图）
  + 右栏：广告详情预览
* 快捷审批按钮组
* 拒绝原因选择器（预设选项+自定义输入）

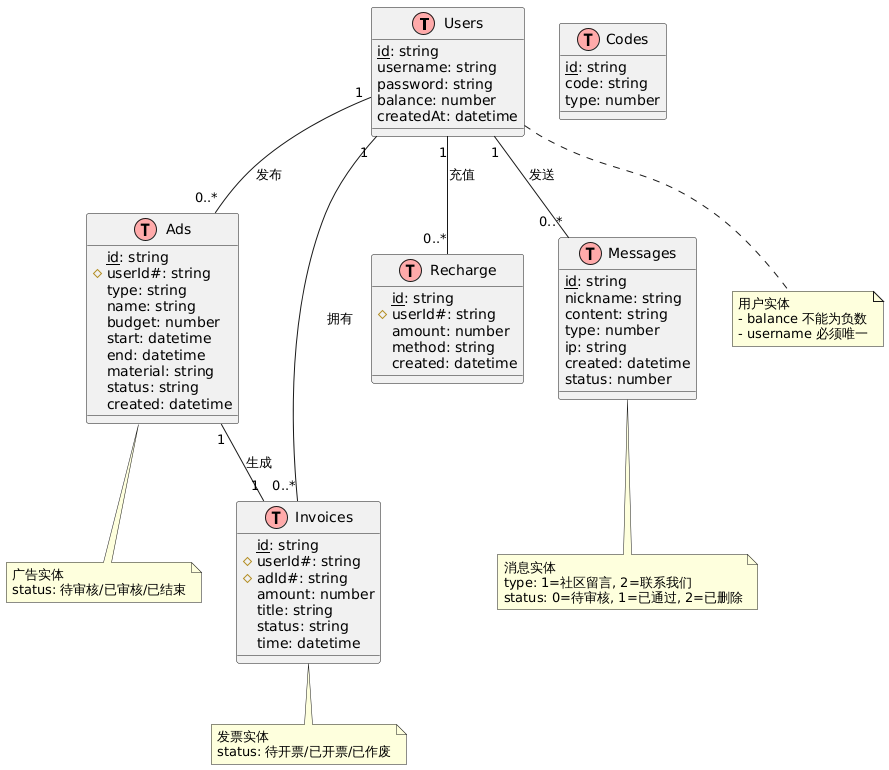
**交互细节**

* **全局通知系统**：右上角Toast提示，区分成功（绿色）/警告（黄色）/错误（红色）
* **表单验证**：实时校验+错误定位
* **加载状态**：骨架屏+进度条
* **空状态**：插画+引导性文字

**设计交付物**

1. **高保真原型**：使用Figma/Sketch制作可交互原型
2. **设计规范文档**：
   * 颜色体系（主色/辅助色/语义色）
   * 字体规范（中英文搭配）
   * 间距系统（8px基准）
   * 组件库（按钮/表单/卡片等）
3. **动效指南**：微交互设计原则（如按钮点击反馈）（尽量）

### 1.4数据库详细分析（包含E-R图）



E-R图，只能画成这样，大体一致

#### Users（用户）：用户实体username唯一且balance不为负

##### Id：string 用来表示创建的用户id，以字符串来表示

##### Username：string 用来表示创建的用户名，以字符串表示

##### Password：string 用来登录或注册的，用户自己设计的字符串

##### Balance：number 用来表示账户余额

##### Createdat：datetime 用来表示创建日期时间

#### Ads（广告）：广告实体，status分为待审/已审/已结

##### Id：string 用来表示创建的广告id，以字符串来表示

##### Userid#：string 用来表示创建广告的用户名，以字符串表示

##### type：string 广告类型

##### name：string 用来表示广告名

##### budget：number 用来表示购买广告的预算

##### start：datetime 表示广告投放的起始时间

##### end：datetime 表示广告投放的结束时间

##### material：string 表示广告的材料

##### status：string表示广告状态

##### created：datetime 表示广告创建时间

#### Recharge（收费）：

##### Id：string 表示充值人的id

##### Userid#：string 用来表示充值人的用户名，以字符串表示

##### Amount：number 表示充值金额

##### Method：string 用来表示账户充值方式，这里特指微信支付

##### Created：datetime 用来表示支付时间

#### Messages（信息/消息）：消息实体，1表示社区留言2表示联系我们status，012表示待审核，已通过，已删除

##### Id：string 用来表示发帖id

##### nickname：string 用来表示发信息的昵称

##### content：string 帖子内容

##### type：number 用来表示帖子类型

##### IP：string 用来表示发帖人所在的ip地址

##### created：datetime 表示发帖时间

##### status：number表示帖子状态

#### Invoice（发票）：发票实体status：戴开票/已开票/已作废

##### Id：string 用来表示开票人id

##### userid：string 用来表示开票人的名字

##### amount：number 开票的广告的金额

##### title：string 用来表示发票的抬头（标题）

##### status：string 用来表示发票状态

##### time：datetime 表示开票时间

* 1. **4.1技术选型及技术选型描述**

**【作者：林浩、吴语林、钟家意、赵子阳】**

**组件化开发 ：React 采用组件化的开发模式，使得代码的可复用性和可维护性大大提高。开发者可以将复杂的 UI 拆分成多个独立的组件，每个组件负责自己的逻辑和状态。**

**虚拟 DOM ：React 使用虚拟 DOM 技术，通过比较新旧虚拟 DOM 的差异，只对需要更新的部分进行真实 DOM 操作，从而提高了页面的渲染性能。**

**庞大的社区支持 ：React 拥有庞大的开发者社区，有丰富的开源库和工具可供使用，能够快速解决开发过程中遇到的问题。**

**开发语言**

* **TypeScript**:

项目主要使用 TypeScript 进行开发，以提高代码的类型安全性和可维护性。TypeScript 是 JavaScript 的超集，提供了静态类型检查和其他现代语言特性。

**框架和库**

* **Django**

Django是一个用Python编写的开源Web应用开发框架。它遵循MVC（模型-视图-控制器）设计模式的变体，称为MTV（模型-模板-视图）。Django旨在让开发复杂、数据库驱动的网站变得简单、快速和高效：

主要特点包括：

* 内置管理员后台：自动生成功能强大的后台管理界面。
* ORM（对象关系映射）：通过Python类操作数据库，无需写SQL语句。
* 路由系统：灵活的URL映射和视图管理。
* 表单处理：支持表单验证和处理。
* 安全性：内置防止SQL注入、跨站脚本（XSS）、跨站请求伪造（CSRF）等多种安全机制。
* 模板引擎：方便页面渲染与前后端分离。
* 可扩展性：支持插件和第三方库。
  1. **4.2 router路由**

**【作者：林浩、吴语林、钟家意、赵子阳】**

/nosence项目

/server **[利用sealos云开发实现前后端交互，留言和社区讨论区的回复](https://github.com/miwumiwumilumilelu/nonsence/commit/91b7e80c82015dcaaf2e022781164a9697b9d02b)**

/auditMessage.ts

/message.ts

/queryMessage.ts

/replyMessage.ts

以讨论的评论回复的路由为例：

@ server/replyMessage.ts

|  |
| --- |
| TypeScript import cloud from "@lafjs/cloud";  const db = cloud.database();  interface ReplyBody {  \_id: string; // 讨论的id  nickname: string;  content: string;  }  interface FunctionContext {  body: ReplyBody;  }  export async function main(ctx: FunctionContext) {  const { \_id, nickname, content } = ctx.body;  if (!\_id || !nickname || !content) {  return { error: "参数不完整" };  }  const reply = {  nickname,  content,  created: new Date().toISOString()  };  // 使用 $push 原子操作追加回复  const { updated } = await db.collection('messages').where({ \_id }).update({  $push: { replies: reply }  });  if (updated === 1) {  return { ok: true, msg: "回复成功" };  } else {  return { error: "回复失败" };  }  } |

**核心功能与技术点**

**云函数环境集成** import cloud from "@lafjs/cloud"; 引入了 Laf Cloud Function 的 SDK，使得代码可以直接访问云数据库等服务。const db = cloud.database(); 初始化数据库连接，方便后续的数据操作。

**类型定义（TypeScript）** interface ReplyBody 定义了回复请求体的结构，包括 \_id（讨论ID）、nickname（回复者昵称）和 content（回复内容），增强了代码的可读性和可维护性，并在开发阶段提供了类型检查。interface FunctionContext 定义了云函数的上下文，其中 body 属性的类型为 ReplyBody，确保了传入参数的结构正确性。

**参数校验** const { \_id, nickname, content } = ctx.body; 从请求体中解构出所需的参数。if (!\_id || !nickname || !content) 对关键参数进行非空校验。如果任何一个参数缺失，则返回错误信息，有效避免了无效数据操作，提高了系统的健壮性。

**回复数据结构** const reply = { nickname, content, created: new Date().toISOString() }; 构建了一个包含昵称、内容和创建时间（ISO 格式字符串）的回复对象，便于存储和后续展示。

**原子操作更新**db.collection('messages').where({ \_id }).update({ $push: { replies: reply } }); 是此功能的核心数据库操作。db.collection('messages') 指定了操作的集合为 messages，这表明每个讨论或广告本身可能被存储在一个名为 messages 的集合中。where({ \_id }) 根据传入的 \_id 定位到要回复的特定讨论。$push: { replies: reply } 是 MongoDB（Laf Cloud Function 底层数据库通常是兼容 MongoDB 的）的**原子操作符**。它将 reply 对象作为数组元素追加到指定讨论的 replies 数组中。这种原子性操作能够保证在高并发场景下，多个用户同时回复同一讨论时，数据的一致性和完整性，避免了竞态条件。

**结果反馈** const { updated } = await ...; 获取数据库更新操作的结果，updated 字段表示成功更新的文档数量。根据 updated 的值判断操作是否成功，并返回相应的成功或失败信息，方便前端进行处理和用户界面反馈。

## 5.讨论区功能

**【作者：林浩、吴语林、钟家意、赵子阳】**

**前端实现-状态管理**

**@/nonsence/web/src/MessageBoard.js**

**功能解析**

1. 列表讨论
2. 发帖用户昵称
3. 发帖内容
4. 回复内容
5. 回复者昵称

**核心代码**

|  |
| --- |
| TypeScript    const [posts, setPosts] = useState([]);    const [name, setName] = useState('');    const [content, setContent] = useState('');    const [replyContent, setReplyContent] = useState({}); // {postIdx: replyText}    const [replyName, setReplyName] = useState({}); // {postIdx: replyName} |

**核心功能与技术点：**

**帖子列表状态 (**posts**)** const [posts, setPosts] = useState([]); 定义了一个名为 posts 的状态变量，它是一个数组，用于存储从后端获取到的所有帖子或广告数据。setPosts 是对应的更新函数。当有新的帖子加载或现有帖子需要更新时，会调用 setPosts 来更新这个数组，从而触发组件重新渲染，展示最新的帖子列表。

**发帖表单状态 (**name**,** content**)** const [name, setName] = useState(''); 和 const [content, setContent] = useState(''); 分别定义了两个字符串状态变量，用于存储用户在发布帖子时输入的昵称 (name) 和帖子内容 (content)。这两个状态变量通常会绑定到表单的输入字段上，实现数据的双向绑定，即用户输入时状态更新，状态更新时输入字段内容改变。

**回复内容状态 (**replyContent**)** const [replyContent, setReplyContent] = useState({}); 定义了一个对象状态变量，用于管理用户在回复特定帖子时输入的回复内容。它的结构是 {postIdx: replyText}，其中 postIdx 可能是帖子的索引或唯一标识符，replyText 则是用户输入的回复文本。这种结构允许同时管理多个帖子的回复输入框内容，而不会相互干扰。

**回复者名称状态 (**replyName**)** const [replyName, setReplyName] = useState({}); 类似 replyContent，这个状态变量也是一个对象，用于存储用户在回复特定帖子时输入的回复者昵称。其结构是 {postIdx: replyName}，用于在用户填写回复时暂存不同回复框的昵称输入。

**相关组件**

**后端实现-数据模型**

**@/nonsence/server/message.ts**

* **功能说明**

1. 发帖昵称
2. 内容：社区内容（1）、联系我们（2）
3. 审核状态：待审核（0）、已通过（1）、已删除（2）
4. 创建时间
5. 发帖ip

* **核心代码**

|  |
| --- |
| TypeScript  interface MessageBody {  nickname: string;  content: string;  type: number;  status: 0 | 1 | 2  created: string;  ip: string;  } |

**核心功能与技术点**

**数据结构规范** interface MessageBody 明确了消息对象的各个属性及其对应的数据类型，确保了前后端数据交换的一致性，减少了因数据格式不匹配而导致的错误。

**发帖者昵称 (**nickname**)** nickname: string; 定义了消息发布者的昵称，其类型为字符串。这确保了昵称信息始终以文本形式存在。

**消息内容 (**content**)** content: string; 定义了消息或帖子的具体内容，类型为字符串。这是用户发布的主要信息。

**消息类型 (**type**)** type: number; 定义了消息的类型，其类型为数字。这可能用于区分不同种类的消息，例如普通帖子、广告、公告等，便于前端进行分类展示或后端进行逻辑处理。

**消息状态 (**status**)** status: 0 | 1 | 2; 定义了消息的当前状态，这是一个**联合类型（Union Type）**，表示 status 属性的值只能是 0、1 或 2。这是一种强大的类型安全机制，用于表示有限的、离散的状态值。例如：

* 0 可能表示“待审核”或“草稿”。
* 1 可能表示“已发布”或“正常显示”。
* 2 可能表示“已禁用”或“已删除”。 这种明确的状态定义有助于业务逻辑的清晰实现和前端界面的准确展示。

**创建时间 (**created**)** created: string; 定义了消息的创建时间，其类型为字符串。通常，这个字符串会采用 ISO 8601 格式，便于时间的存储、传输和解析。

**IP 地址 (**ip**)** ip: string; 定义了发布消息的用户 IP 地址，类型为字符串。这在互联网广告平台中可能用于用户地域分析、防止恶意发布或进行日志记录等目的。

**核心功能实现-发帖功能**

**@/nonsense/web/src/ClientCenter.js**

**功能解析**

1. **前端发帖讨论功能**

**核心代码**

|  |
| --- |
| TypeScript const handleAudit = async (\_id, status, type) => {  try {  const res = await axios.post(`${API\_BASE}/auditMessage`, {  \_id,  status  });  if (res.data && res.data.ok) {  setMsg('操作成功');  if (type === 1) fetchDiscussions();  }  } catch {  setMsg('网络错误');  }  }; |

**核心功能与技术点：**

**异步操作 (**async**/**await**)** const handleAudit = async (\_id, status, type) => { ... }; 这是一个异步函数，使用 async/await 关键字来处理网络请求。这意味着它会在等待后端响应时非阻塞地执行，从而避免冻结用户界面，提供更流畅的用户体验。这种模式是现代 Web 开发中处理异步操作的标准实践。

**参数传递与业务逻辑** 函数接收三个关键参数：

* \_id: 待审核内容的唯一标识符，通常是数据库中的文档 ID。
* status: 表示内容将要被更新到的新状态，例如 0（待审核）、1（已批准）、2（已拒绝）等。这个参数直接驱动后端对内容状态的修改。
* type: 一个可选的类型参数，可能用于区分不同类型的内容（例如，广告、普通帖子、评论）。它的存在表明该审核功能可能是一个通用的机制，用于处理不同模块的内容。

**HTTP 请求与 Axios 库** const res = await axios.post(${API\_BASE}/auditMessage, { \_id, status }); 这一行是函数的核心，它使用 axios 库发起一个 HTTP POST 请求。

* axios.post(): 发送 POST 请求，适用于提交数据以创建或修改资源。
* API\_BASE/auditMessage: 这是一个动态构建的后端 API 端点 URL，API\_BASE 通常是一个全局常量，定义了后端服务的基地址。这表明后端存在一个专门用于处理内容审核的接口。
* { \_id, status }: 请求体（payload），将待审核内容的 ID 和目标状态发送给后端服务器。后端会根据这些信息来更新数据库中相应内容的 status 字段。

**响应处理与用户反馈** if (res.data && res.data.ok) { setMsg('操作成功'); ... } 客户端接收到后端响应后，会检查响应数据。

* res.data && res.data.ok: 这是一个典型的成功判断条件，意味着后端在 res.data 中返回了一个表示操作成功的 ok 字段（通常为 true）。这种模式使得前端能够清晰地判断后端业务逻辑的执行结果。
* setMsg('操作成功'): 如果操作成功，通过 setMsg 函数更新前端的消息提示状态，向用户显示“操作成功”的反馈。这通常会触发一个临时的通知或提示框。

**条件刷新数据** if (type === 1) fetchDiscussions(); 这是一个重要的条件刷新机制。如果被审核的内容类型 (type) 为 1（例如，表示一个讨论帖子），那么在审核操作成功后，会调用 fetchDiscussions() 函数来重新获取讨论列表。这确保了用户界面能够立即反映最新的内容状态，提高了实时性和用户体验。

**错误处理** try { ... } catch { setMsg('网络错误'); } 整个网络请求被包裹在一个 try...catch 块中，用于捕获潜在的网络错误（例如，服务器无响应、网络断开等）。

* catch: 如果在 try 块中的异步操作发生错误，catch 块会被执行。
* setMsg('网络错误'): 在发生网络错误时，通过 setMsg 更新消息提示为“网络错误”，告知用户当前的问题。这种错误处理机制提升了应用的健壮性和用户体验。

**相关组件**

**回复功能**

**@** **/nonsence/web/src/MessageBoard.js**

* **功能解析**

1. 回复帖子

**核心代码**

|  |
| --- |
| const handleReply = async (\_id) => {  const rName = replyName[\_id];  const rContent = replyContent[\_id];  try {  const res = await axios.post(`${API\_BASE}/replyMessage`, {  \_id,  nickname: rName,  content: rContent  });  if (res.data && res.data.ok) {  setReplyMsg(m => ({ ...m, [\_id]: '回复成功' }));  fetchPosts(); // 刷新帖子列表  }  } catch {  setReplyMsg(m => ({ ...m, [\_id]: '网络错误' }));  }  }; |

**核心功能与技术点**

**异步处理与非阻塞操作** const handleReply = async (\_id) => { ... }; 这是一个**异步函数**，通过 async/await 关键字实现。这意味着当函数执行到网络请求时，它会**暂停执行并等待后端响应**，但不会阻塞浏览器的主线程，确保用户界面的流畅性和响应性。这种异步模式对于任何涉及网络通信的现代 Web 应用都是必不可少的。

**参数与上下文关联** 函数接收一个核心参数 \_id，它代表**当前用户正在回复的帖子的唯一标识符**。这个 \_id 是将回复与特定帖子关联起来的关键，确保了回复内容的精准投递和数据管理。

**局部状态管理与数据获取** const rName = replyName[\_id]; 和 const rContent = replyContent[\_id]; 这两行代码展示了前端如何**从局部状态中获取特定于当前帖子（由** \_id **标识）的回复昵称和回复内容**。

* replyName 和 replyContent 通常是使用 useState 定义的 React 状态变量，它们被设计为对象，例如 { [postId]: "回复者昵称", [anotherPostId]: "另一个昵称" }。这种数据结构允许前端在一个组件中同时管理多个帖子回复框的输入状态，而不会导致不同帖子之间的输入互相覆盖，极大地提升了用户体验和代码的模块化。

**HTTP 请求与 Axios 库** const res = await axios.post(${API\_BASE}/replyMessage, { \_id, nickname: rName, content: rContent }); 这一行是回复功能的核心网络通信部分。

* axios.post(): 用于发送 HTTP POST 请求，这是提交新数据（即回复内容）到服务器的标准方式。
* ${API\_BASE}/replyMessage: 这是一个动态构建的 API 端点 URL，指向后端专门处理消息回复的接口。API\_BASE 是后端服务的基地址，体现了前端对后端 API 接口的清晰约定。
* { \_id, nickname: rName, content: rContent }: 这是发送到后端服务器的**请求体（payload）**。它包含了回复所属的帖子 ID、回复者的昵称以及回复的具体内容。后端服务将接收这些数据，并将其存储为对应帖子的回复。

**响应处理与用户反馈** if (res.data && res.data.ok) { setReplyMsg(m => ({ ...m, [\_id]: '回复成功' })); fetchPosts(); } 在接收到后端响应后，前端会进行如下处理：

* res.data && res.data.ok: 检查后端响应数据，判断操作是否成功。res.data.ok 通常是后端自定义的一个布尔值，用于明确指示业务操作的结果，相比仅依赖 HTTP 状态码，这种方式更具业务语义性。
* setReplyMsg(m => ({ ...m, [\_id]: '回复成功' }));: 如果回复成功，通过 setReplyMsg 函数更新与当前帖子关联的回复消息状态。setReplyMsg 可能也是一个类似于 replyName 和 replyContent 的对象状态，用于在每个回复框下方显示特定的成功或错误信息。这种**局部、精确的状态更新**避免了不必要的全局渲染，优化了性能。
* fetchPosts();: **刷新帖子列表**。这是一个关键的**数据同步操作**。当一条回复成功提交后，前端需要重新从后端获取帖子数据，以便新的回复能够立即显示在对应的帖子下方。这确保了用户界面的实时性，提升了用户体验。

**错误处理与用户提示** try { ... } catch { setReplyMsg(m => ({ ...m, [\_id]: '网络错误' })); } 整个网络请求都被包裹在一个 try...catch 块中，以捕获和处理可能发生的网络错误。

* catch: 当网络请求失败（例如，服务器无响应、网络中断、API 错误等）时，catch 块会被执行。
* setReplyMsg(m => ({ ...m, [\_id]: '网络错误' }));: 在发生网络错误时，更新与当前帖子关联的回复消息状态为“网络错误”，向用户提供**明确的失败提示**。这种健壮的错误处理机制增强了应用的可靠性和用户友好性。

**审核功能**

**@/nonsense/** **web/src/AdminCenter.js**

**功能解析**

1. 审核内容是否合法。

**核心代码**

|  |
| --- |
| TypeScript const handleAudit = async (\_id, status, type) => {  try {  const res = await axios.post(`${API\_BASE}/auditMessage`, {  \_id,  status  });  if (res.data && res.data.ok) {  setMsg('操作成功');  if (type === 1) fetchDiscussions();  }  } catch {  setMsg('网络错误');  }  }; |

### **核心功能与技术点**

**异步操作** (async/await) const handleAudit = async (\_id, status, type) => { ... }; 这是一个**异步函数**，使用 async/await 关键字来处理网络请求。这意味着它会在等待后端响应时**非阻塞地执行**，从而避免冻结用户界面，提供更流畅的用户体验。这种模式是现代 Web 开发中处理异步操作的标准实践，尤其适用于需要等待外部资源（如数据库或第三方服务）响应的场景。

**参数传递与业务逻辑** 函数接收三个关键参数，这些参数共同定义了审核操作的上下文：

* \_id: **待审核内容的唯一标识符**，通常是数据库中的文档 ID。这是定位特定审核对象的关键。
* status: **表示内容将要被更新到的新状态**，例如 0（待审核）、1（已批准）、2（已拒绝）等。这个参数直接驱动后端对内容状态的修改，是业务逻辑的核心体现。
* type: 一个可选的**类型参数**，可能用于区分不同类型的内容（例如，广告、普通帖子、评论）。它的存在表明该审核功能可能是一个**通用的机制**，用于处理不同模块的内容，提升了代码的**复用性**和系统的**可扩展性**。例如，不同的 type 值可能触发后端不同的处理逻辑或前端不同的刷新行为。

**HTTP 请求与 Axios 库** const res = await axios.post(${API\_BASE}/auditMessage, { \_id, status }); 这一行是函数的核心，它使用 axios **库**发起一个 HTTP POST 请求。

* axios.post(): 用于发送 POST 请求，这种方法适用于**提交数据以创建或修改资源**，在这里是修改内容的审核状态。
* ${API\_BASE}/auditMessage: 这是一个**动态构建的后端 API 端点 URL**，API\_BASE 通常是一个全局常量，定义了后端服务的基地址。这表明后端存在一个专门用于处理内容审核的接口，遵循RESTful或类似的设计原则。这种模块化的API设计有助于前后端的职责分离和并行开发。
* { \_id, status }: 这是**请求体（payload）**，将待审核内容的 ID 和目标状态以 JSON 格式发送给后端服务器。后端会根据这些信息来更新数据库中相应内容的 status 字段。这种数据传输方式安全且高效。

**响应处理与用户反馈** if (res.data && res.data.ok) { setMsg('操作成功'); ... } 客户端接收到后端响应后，会**严格检查响应数据**。

* res.data && res.data.ok: 这是一个典型的**成功判断条件**，意味着后端在 res.data 中返回了一个表示操作成功的 ok 字段（通常为 true）。这种模式使得前端能够清晰、可靠地判断后端业务逻辑的执行结果，避免了对HTTP状态码的过度依赖，使业务逻辑判断更加直观。
* setMsg('操作成功'): 如果操作成功，通过 setMsg 函数更新前端的消息提示状态，向用户显示“操作成功”的反馈。这通常会触发一个**临时的通知或提示框**，为用户提供即时、友好的操作反馈。

**条件刷新数据** if (type === 1) fetchDiscussions(); 这是一个重要的**条件刷新机制**。如果被审核的内容类型 (type) 为 1（例如，表示一个讨论帖子），那么在审核操作成功后，会调用 fetchDiscussions() 函数来**重新获取讨论列表**。这确保了用户界面能够**立即反映最新的内容状态**，无需用户手动刷新页面，极大地提高了**实时性和用户体验**。这种数据同步策略对于维护数据一致性至关重要。

**错误处理** try { ... } catch { setMsg('网络错误'); } 整个网络请求被包裹在一个 try...catch **块**中，用于捕获潜在的**网络错误**（例如，服务器无响应、网络断开、超时等）。

* catch: 如果在 try 块中的异步操作发生错误，catch 块会被执行。这是一种健壮的**错误捕获机制**。
* setMsg('网络错误'): 在发生网络错误时，通过 setMsg 更新消息提示为“网络错误”，告知用户当前的问题。这种**友好的错误提示**避免了应用崩溃，并引导用户检查网络连接，提升了应用的**健壮性和用户体验**。

**相关组件**

**数据库结构**

* **功能解析**

1. MongoDB集合结构

**核心代码**

|  |
| --- |
| TypeScript messages: {  \_id: ObjectId,  nickname: String,  content: String,  type: Number,  status: Number,  created: Date,  ip: String,  replies: [{  nickname: String,  content: String,  created: Date  }]  } |

**核心功能与技术点：**

**顶层集合：**messages messages: { ... } 表示在 MongoDB 中存在一个名为 messages 的集合（Collection）。在 MongoDB 中，集合是文档的组，类似于关系型数据库中的表，但没有严格的模式限制（尽管在应用层面通常会定义模式以确保数据一致性）。这个集合将用于存储平台上的主要内容，如广告、公告或用户发布的帖子。

**文档唯一标识符 (**\_id**)** \_id: ObjectId, 这是 MongoDB 自动为每个文档生成的唯一主键。ObjectId 是一种特殊的 BSON 类型，它保证了在分布式系统中的唯一性，并且包含了创建时间戳等信息，非常适合作为文档的唯一标识。

**核心消息属性**

* nickname: String,: 定义了发布消息的用户昵称。这是一个字符串类型，通常用于显示发布者的身份。
* content: String,: 存储消息或帖子的主要文本内容。这是一个字符串类型，承载了用户发布的核心信息。
* type: Number,: 定义了消息的类型，是一个数字类型。这允许系统对不同种类的消息进行分类和区分，例如，1 代表广告，2 代表普通帖子，3 代表系统公告等。这种类型字段为前端展示和后端业务逻辑提供了灵活性。
* status: Number,: 表示消息的当前状态，也是一个数字类型。这对于内容审核和生命周期管理至关重要。例如：
  + 0: 草稿或待审核
  + 1: 已发布/正常显示
  + 2: 已禁用/已删除
  + 3: 审核未通过 这种状态管理机制确保了平台能够有效地控制内容的可见性和可用性。
* created: Date,: 记录消息的创建时间，类型为 Date。MongoDB 原生支持 Date 类型，便于时间的存储和基于时间范围的查询。精确的时间戳对于排序、归档和分析用户行为都非常重要。
* ip: String,: 存储发布消息时的用户 IP 地址，类型为 String。这个字段在广告平台中具有多重用途，例如：
  + **地域分析**：了解广告受众的地域分布。
  + **安全审计**：追踪恶意发布行为或滥用情况。
  + **合规性**：满足某些法规对数据来源记录的要求。

**嵌套文档与数组：**replies replies: [{ ... }] 这是该数据模型的一个亮点，它使用了 MongoDB 的\*\*嵌套文档（Embedded Documents）**和**数组（Arrays）\*\*特性。

* replies: [...]: replies 是一个数组，表示一个帖子可以包含多条回复。这种结构非常适合一对多的关系（一个帖子可以有多个回复），并且将回复直接嵌入到帖子文档中，形成一个\*\*非范式化（denormalized）\*\*的数据模型。
* **嵌套文档结构**：数组中的每个元素都是一个独立的文档对象，包含回复的具体信息：
  + nickname: String,: 回复者的昵称。
  + content: String,: 回复的具体文本内容。
  + created: Date: 回复的创建时间。

**数据模型优势**

1. **读性能优化**：对于需要同时获取帖子内容及其所有回复的场景，这种嵌入式模型具有显著的性能优势。只需一次数据库查询即可获取所有相关数据，减少了 JOIN 操作的开销（MongoDB 中没有 JOIN）。这对于频繁展示帖子及其评论的广告平台非常有利。
2. **数据局部性**：相关数据（帖子和其回复）存储在同一个文档中，这使得数据在磁盘上是连续的，进一步提高了读效率。
3. **原子性操作**：MongoDB 允许对单个文档进行原子更新操作。例如，添加新回复到 replies 数组可以使用 $push 操作符，这在**高并发场景下能有效避免竞态条件**，保证数据的一致性和完整性。
4. **模型简洁性**：对于强关联的数据，嵌套模型可以使数据结构更直观，减少了跨集合查询的复杂性。

**API接口分析**

* **核心代码**

|  |
| --- |
| 主要API接口 1. POST /message // 发布讨论  2. POST /replyMessage // 回复讨论  3. POST /queryMessages // 查询讨论列表  4. POST /auditMessage // 审核讨论 |

**核心功能与技术点**

**1.** POST /message **- 发布讨论**

* **功能描述**: 这个接口允许用户在平台上发布新的讨论、帖子或广告内容。它是用户生成内容（User Generated Content, UGC）机制的核心入口。
* **请求方法**: 使用 POST 方法。POST 请求通常用于向服务器提交数据以创建新的资源。这符合 RESTful API 设计原则中创建资源的语义。
* **典型请求体 (Request Body)**: 客户端会发送包含消息内容的 JSON 对象，例如用户的昵称、帖子内容、消息类型（如广告、普通讨论）、初始状态（如待审核）等。
* **后端处理**: 后端服务接收到请求后，会将这些数据存储到数据库中（例如，前面分析过的 messages 集合），并可能进行初步的合法性校验、敏感词过滤，或者设置默认的发布时间、IP地址等。
* **应用场景**: 用户发布广告、商家发布推广信息、用户发表对某个话题的看法等。

**2.** POST /replyMessage **- 回复讨论**

* **功能描述**: 该接口用于允许用户对已存在的讨论（帖子或广告）进行回复。这促进了用户之间的互动，构建了评论和交流的生态系统。
* **请求方法**: 使用 POST 方法。尽管是对现有资源的补充，但回复操作本质上是创建了一个新的“回复”资源（嵌入在原讨论中或单独存储），因此 POST 是合适的选择。
* **典型请求体**: 客户端会发送回复所属的讨论 \_id、回复者的昵称以及回复的具体内容。
* **后端处理**: 后端接收请求后，会找到对应的讨论，并将新的回复作为嵌套文档（如前文 replies 字段）添加到该讨论中。此操作通常会利用数据库的原子更新特性（如 MongoDB 的 $push）来确保数据的一致性，尤其是在高并发场景下。
* **应用场景**: 用户对广告内容发表评论、对帖子进行回复、与发布者进行互动等。

**3.** POST /queryMessages **- 查询讨论列表**

* **功能描述**: 这个接口用于客户端从后端获取讨论（帖子或广告）的列表。它是页面展示内容的基础。
* **请求方法**: 使用 POST 方法。虽然查询操作通常使用 GET 方法，但在这里使用 POST 可能有以下几种原因：
  + **复杂的查询条件**: 如果查询条件非常多或者包含敏感信息（如用户筛选偏好），将其放在请求体中（POST）比放在 URL 参数中（GET）更方便和安全。
  + **避免 URL 长度限制**: GET 请求的 URL 长度有限制，而 POST 请求体的大小通常没有严格限制。
  + **统一接口风格**: 某些开发团队可能倾向于所有数据操作都使用 POST，以保持 API 风格的一致性。
* **典型请求体**: 客户端可以发送查询参数，如分页信息（页码、每页数量）、筛选条件（按类型、按状态）、排序规则（按时间、按热度）等。
* **后端处理**: 后端接收请求后，会根据请求体中的参数构建数据库查询，从 messages 集合中检索符合条件的讨论数据，并将其分页返回给前端。
* **应用场景**: 首页显示最新广告、用户查看自己的发布历史、管理员浏览待审核内容列表等。

**4.** POST /auditMessage **- 审核讨论**

* **功能描述**: 这个接口专用于管理员或拥有特定权限的用户对平台内容进行审核，包括批准、拒绝或修改内容的发布状态。它是平台内容质量控制和合规性管理的核心。
* **请求方法**: 使用 POST 方法。审核操作是对现有资源状态的修改，POST 或 PUT 都是合适的。此处选择 POST。
* **典型请求体**: 客户端会发送待审核内容的 \_id 以及其新的 status 值。
* **后端处理**: 后端接收请求后，会根据 \_id 找到对应的讨论，并将其 status 字段更新为指定的值。此操作通常需要身份验证和权限检查，以确保只有授权用户才能执行审核。
* **应用场景**: 管理员对用户发布的广告进行审核、标记违规内容、解除内容封禁等。

**技术特点**

数据流设计

1. 分层架构：

前端展示层（React组件）；

业务逻辑层（API调用和数据处理）；

数据持久层（MongoDB数据库）**获取术语表数据**：使用 getGlossaries 函数从后端获取术语表数据，并将其存储在 glossaries 变量中。

1. 状态管理：

使用 React Hooks 管理组件状态

实时更新UI和数据同步

安全特性

1. 所有帖子需要管理员审核
2. 支持删除违规内容
3. 状态跟踪（待审核、已通过、已删除）
4. 昵称和内容必填
5. 内容长度限制（1000字）
6. IP地址记录

用户体验

1. 实时反馈（提交成功、失败提示）
2. 加载状态显示
3. 清晰的错误提示
4. 响应式布局
5. 现代化UI设计
6. 良好的视觉层次

**6.支付系统架构**

**【作者：林浩、吴语林、钟家意、赵子阳】**

**前端实现-状态管理**

**@/nonsense/web/src/ClientCenter.js**

|  |
| --- |
| TypeScript const [balance, setBalance] = useState(() => {  const b = localStorage.getItem('nonsence\_balance');  return b ? Number(b) : 0;  });  const [rechargeForm, setRechargeForm] = useState({  amount: '',  method: '微信支付'  });  const [recharges, setRecharges] = useState([]); |

**核心功能与技术点：**

**用户余额状态 (**balance**)** const [balance, setBalance] = useState(() => { const b = localStorage.getItem('nonsence\_balance'); return b ? Number(b) : 0; });

* **状态定义与初始化**: balance 状态变量用于存储用户的当前账户余额，其类型为数字。setBalance 是对应的更新函数。
* **持久化存储**: 这是一个非常重要的设计点。在 useState 的初始化函数中，它尝试从 localStorage 中获取名为 'nonsence\_balance' 的值。
  + localStorage.getItem('nonsence\_balance'): 尝试从浏览器本地存储中读取上次保存的余额数据。
  + b ? Number(b) : 0;: 如果本地存储中存在余额数据 (b 不为 null 或 undefined)，则将其转换为数字类型并作为初始值；否则，将余额初始化为 0。
* **优势**: 这种初始化方式实现了**用户余额的本地持久化**。即使用户关闭或刷新浏览器，已保存的余额也能被恢复，提升了用户体验的连贯性。这对于一个广告平台来说至关重要，因为它涉及用户的资金。然而，需要注意的是，localStorage 中的数据仅用于前端展示和快速初始化，**最终的、权威的余额数据仍应以后端服务器为准**，防止客户端数据被篡改或不一致。

**充值表单状态 (**rechargeForm**)** const [rechargeForm, setRechargeForm] = useState({ amount: '', method: '微信支付' });

* **复合状态对象**: rechargeForm 是一个**对象状态变量**，用于管理用户在充值时填写的表单数据。这种方式允许将多个相关的表单字段聚合到一个状态对象中，简化了状态管理逻辑。
* amount: '': 存储用户输入的充值金额，初始为空字符串。通常会与一个输入框绑定，用户输入时更新此状态。
* method: '微信支付': 存储用户选择的支付方式，初始值为“微信支付”。这可能是一个下拉菜单或一组单选按钮的选项。
* **优势**: 将表单数据集中管理，使得对表单字段的清空、校验或提交都变得更加便捷和统一。

**充值记录列表状态 (**recharges**)** const [recharges, setRecharges] = useState([]);

* **列表数据**: recharges 是一个**数组状态变量**，用于存储从后端获取到的用户充值历史记录。setRecharges 是其更新函数。
* **初始化**: 初始值为空数组，表示刚开始没有充值记录。
* **应用场景**: 这个状态通常会在用户进入充值历史页面或个人中心时，通过调用后端 API 获取并填充数据。然后，前端会遍历这个数组，将每条充值记录（包括金额、时间、状态等）展示给用户。
* **优势**: 能够清晰地管理和展示用户的财务交易历史，增加了平台的透明度和用户的信任感。

**后端实现-微信支付**

**@/nonsense/server/recharge.ts**

|  |
| --- |
| TypeScript const [balance, setBalance] = useState(() => {  const b = localStorage.getItem('nonsence\_balance');  return b ? Number(b) : 0;  });  const [rechargeForm, setRechargeForm] = useState({  amount: '',  method: '微信支付'  });  const [recharges, setRecharges] = useState([]); |

**核心功能与技术点：**

**用户余额状态 (**balance**)** const [balance, setBalance] = useState(() => { const b = localStorage.getItem('nonsence\_balance'); return b ? Number(b) : 0; });

* **状态定义与初始化策略**: balance 状态变量用于存储用户的当前账户余额，其类型为数字。setBalance 是更新 balance 值的专用函数。值得注意的是，这里的 useState 采用了函数式初始化：在组件首次渲染时，它会执行一个函数来确定 balance 的初始值。
* **本地持久化**: 这种初始化方式巧妙地利用了浏览器内置的 localStorage 来实现**用户余额的本地持久化**。
  + localStorage.getItem('nonsence\_balance'): 这行代码尝试从浏览器的 localStorage 中获取一个键为 'nonsence\_balance' 的值。这个键名表明了它存储的是一个与“余额”相关的非关键性（nonsence）数据，可能是为了避免与后端权威数据混淆，或者只是一个开发时的命名约定。
  + b ? Number(b) : 0;: 如果 localStorage 中存在这个值 (b 不为 null 或 undefined)，它会被 Number() 函数转换成数字类型并作为 balance 的初始值；否则，balance 将被初始化为 0。
* **优势与考量**: 这种本地持久化策略带来了显著的**用户体验优势**。即使用户意外关闭浏览器或刷新页面，其上次已知的余额信息也能被快速恢复并显示，避免了因数据清空而导致的不便。然而，在软件工程实践中，**必须明确：**localStorage **中的数据仅应用于前端展示和初步状态恢复**。**最终、权威的用户余额数据必须始终来源于后端服务器**。前端不应直接依赖 localStorage 进行任何敏感的财务计算或决策，以防客户端数据被篡改、不一致或存在安全漏洞。后端应负责所有金额的增减、校验和最终存储。

**充值表单状态 (**rechargeForm**)** const [rechargeForm, setRechargeForm] = useState({ amount: '', method: '微信支付' });

* **复合状态对象**: rechargeForm 是一个**对象状态变量**，用于集中管理用户在进行充值操作时填写的表单数据。这种方式将多个相关的表单字段（如金额和支付方式）聚合到一个单一的状态对象中，这是一种推荐的**表单状态管理模式**。
* amount: '': 存储用户输入的充值金额。它被初始化为空字符串，通常会与前端界面的一个文本输入框（如 <input type="number">）进行双向绑定。用户在输入金额时，这个状态会实时更新。
* method: '微信支付': 存储用户选择的支付方式，初始值为字符串 '微信支付'。这通常会绑定到支付方式选择器（如下拉菜单、单选按钮组），反映用户当前的支付偏好。
* **优势**: 将表单数据集中管理，使得对表单字段的清空、校验、提交以及重置都变得更加便捷和统一。它简化了组件逻辑，并提高了代码的可读性，特别是在表单字段较多的情况下。

**充值记录列表状态 (**recharges**)** const [recharges, setRecharges] = useState([]);

* **列表数据管理**: recharges 是一个**数组状态变量**，专门用于存储从后端 API 获取到的用户充值历史记录。setRecharges 是用于更新这个数组的函数。
* **初始化**: 初始值为空数组 []，表示在组件首次加载时，还没有获取到任何充值记录。
* **应用场景**: 这个状态通常会在用户访问个人中心、充值历史页面或账户概览时被填充。前端会触发一个网络请求（例如，调用 /api/getRechargeHistory），获取到数据后，通过 setRecharges 更新此状态。随后，前端界面会遍历 recharges 数组，将每条充值记录（包括充值金额、交易时间、支付方式、交易状态等）以表格或列表形式展示给用户。
* **优势**: 能够清晰、结构化地管理和展示用户的财务交易历史，为用户提供了透明的资金流向视图，这对于提升平台的**信任度**和**用户满意度**至关重要。

**核心功能实现-充值功能**

**@/nonsense/web/src/ClientCenter.js**

|  |
| --- |
| TypeScript const handleRecharge = async e => {  e.preventDefault();  setRechargeMsg('');  const amount = Number(rechargeForm.amount);  const payload = {  userId: user.id,  amount,  method: '微信支付'  };  try {  const res = await axios.post(`${API\_BASE}/recharge`, payload);  if (res.data && res.data.ok) {  setRechargeMsg('充值成功！');  setBalance(res.data.balance);  // 更新充值历史  fetchRechargeHistory();  }  } catch {  setRechargeMsg('网络错误');  }  }; |

**核心功能与技术点：**

**事件处理与表单阻止默认行为** const handleRecharge = async e => { e.preventDefault(); ... };

* **事件参数**: e 代表触发此函数执行的事件对象，通常是在表单提交时调用。
* **阻止默认行为**: e.preventDefault(); 是标准的 React（以及原生 JavaScript）实践，用于阻止表单提交的默认浏览器行为，即避免页面刷新。这对于单页应用（SPA）至关重要，它确保了充值逻辑能够通过 JavaScript 异步执行，而不是通过传统的页面跳转。

**状态重置与数据准备** setRechargeMsg(''); const amount = Number(rechargeForm.amount); const payload = { userId: user.id, amount, method: '微信支付' };

* **清空消息提示**: setRechargeMsg(''); 在每次尝试充值前，都会清空之前的充值相关消息提示。这确保了用户在每次操作后都能看到最新的反馈，避免了信息残留造成的混淆。
* **金额类型转换**: const amount = Number(rechargeForm.amount); 将从充值表单状态 rechargeForm.amount 中获取到的金额（通常是字符串类型，来自表单输入）显式转换为 Number 类型。这是进行数值计算或发送给后端前必不可少的步骤，保证了数据的正确性。
* **构建请求载荷 (Payload)**: const payload = { userId: user.id, amount, method: '微信支付' }; 构造了即将发送给后端 API 的数据体。
  + userId: user.id: 包含了当前登录用户的唯一标识符。这是后端识别用户并为其账户充值的关键信息。user 对象通常从全局状态管理（如 Context API 或 Redux）中获取。
  + amount: 经过类型转换的充值金额。
  + method: '微信支付': 指定了支付方式。这里硬编码为“微信支付”，表明当前系统可能只支持一种支付方式，或这是默认/唯一可选的支付渠道。在更复杂的系统中，此值会从 rechargeForm.method 动态获取。

**HTTP 请求与 Axios 库** try { const res = await axios.post(${API\_BASE}/recharge, payload); ... }

* **异步请求**: 整个充值逻辑被包裹在 try...catch 块中，并使用 async/await 处理异步网络请求，确保操作的非阻塞性。
* **POST 请求**: axios.post(${API\_BASE}/recharge, payload); 发送一个 HTTP POST 请求到后端 /recharge 接口。POST 方法适用于提交数据以创建新的交易记录或修改用户余额。
* **API 端点**: ${API\_BASE}/recharge 指明了后端处理充值业务的特定接口。API\_BASE 是后端服务的基地址，确保了请求能够正确路由。

**响应处理与状态更新** if (res.data && res.data.ok) { setRechargeMsg('充值成功！'); setBalance(res.data.balance); fetchRechargeHistory(); }

* **成功判断**: if (res.data && res.data.ok) 检查后端响应，判断充值操作是否成功。res.data.ok 是后端定义的业务成功标志，它提供了比单纯的 HTTP 状态码更具业务语义的判断依据。
* **用户反馈**: setRechargeMsg('充值成功！'); 向用户显示成功消息，提升用户体验。
* **更新余额**: setBalance(res.data.balance); 这是一个关键步骤。充值成功后，前端会立即用后端返回的最新余额值来更新本地 balance 状态。这确保了页面上显示的余额与后端数据保持同步，提供了即时反馈。
* **刷新充值历史**: fetchRechargeHistory(); 调用另一个函数来重新获取并更新用户的充值历史记录。这保证了新的充值交易能够立即出现在用户的历史列表中，进一步增强了数据的一致性和用户体验。

**错误处理** catch { setRechargeMsg('网络错误'); }

* **捕获错误**: try...catch 块捕获在网络请求或后端处理过程中可能发生的任何错误（例如，网络中断、服务器异常、API 返回错误）。
* **用户提示**: setRechargeMsg('网络错误'); 在发生错误时，向用户显示一个通用的“网络错误”提示。这种错误处理机制提升了应用的健壮性，避免了未捕获的异常导致应用崩溃，并提供了基本的故障排查指导。

**后端实现-余额管理**

**@/nonsense/server/recharge.ts**

|  |
| --- |
| TypeScript export async function main(ctx: FunctionContext) {  const { userId, amount, method } = ctx.body;  // 写入充值记录  const recharge = {  userId,  amount,  method,  time: new Date().toISOString(),  status: '成功',  };  await db.collection('recharges').add(recharge);  // 更新用户余额  const user = await db.collection('users').where({ \_id: userId }).getOne();  const newBalance = (user.data.balance || 0) + amount;  await db.collection('users').update({ balance: newBalance });  return { ok: true, balance: newBalance };  } |

**核心功能与技术点**

**云函数入口与参数解析** export async function main(ctx: FunctionContext) { const { userId, amount, method } = ctx.body; ... }

* export async function main(ctx: FunctionContext): 这是 Laf Cloud Function 的标准入口函数定义。async 关键字表明该函数执行异步操作，通常涉及数据库交互或外部服务调用。ctx: FunctionContext 参数包含了云函数执行的上下文信息，其中 ctx.body 包含了前端通过 HTTP 请求发送过来的数据。
* **参数解构**: const { userId, amount, method } = ctx.body; 从请求体中解构出充值所需的关键信息：用户的唯一标识符 userId、充值金额 amount 和支付方法 method。这是后端处理任何用户发起操作的第一步，确保获取到正确、必要的输入。

**充值记录写入** const recharge = { userId, amount, method, time: new Date().toISOString(), status: '成功', }; await db.collection('recharges').add(recharge);

* **构建充值记录对象**: 创建一个 recharge 对象，包含交易的详细信息，如 userId、amount、method。
* **时间戳**: time: new Date().toISOString() 记录了充值发生的确切时间，并使用 ISO 8601 格式，便于存储、查询和跨系统的时间解析。
* **状态**: status: '成功' 硬编码为“成功”。在实际生产环境中，这里通常会更复杂，例如与第三方支付网关的异步回调结果绑定，或者在支付发起时记录为“处理中”，在回调确认成功后才更新为“成功”。此处简化处理，假定前端发起请求即视为成功。
* **写入数据库**: await db.collection('recharges').add(recharge); 将完整的充值记录文档添加到名为 recharges 的数据库集合中。这为用户提供了充值历史，也为平台提供了审计和对账的依据。这是一个**插入操作**，是财务记录的关键步骤。

**用户余额更新** const user = await db.collection('users').where({ \_id: userId }).getOne(); const newBalance = (user.data.balance || 0) + amount; await db.collection('users').update({ balance: newBalance });

* **获取当前用户数据**: const user = await db.collection('users').where({ \_id: userId }).getOne(); 从 users 集合中查询并获取当前用户的完整数据。这是**更新余额前的必要步骤**，确保获取到用户当前的准确余额。getOne() 方法表示只获取一个匹配的文档。
* **计算新余额**: const newBalance = (user.data.balance || 0) + amount; 计算用户的最新余额。
  + (user.data.balance || 0): 这是一个健壮性处理，如果用户数据中 balance 字段不存在（例如，新用户或旧数据），则将其视为 0，避免计算错误。
  + + amount: 将充值金额加到原有余额上。
* **更新数据库**: await db.collection('users').update({ balance: newBalance }); 将计算出的 newBalance 更新到 users 集合中对应用户的 balance 字段。这是**核心的余额变更操作**。在实际生产系统中，为了保证强一致性，**通常会使用数据库的事务或原子操作符（如** $inc**）来直接增加余额，而不是先读后写**。此处代码是“先读后写”，在极高并发下理论上存在竞态条件风险，但在某些云函数环境或低并发场景下，通过数据库层面的锁或原子性保障也能工作。

**响应返回** return { ok: true, balance: newBalance };

* **操作成功标志**: 返回一个包含 ok: true 的对象，告知前端充值操作已成功。这是前后端通信的规范，便于前端判断业务结果。
* **返回最新余额**: 同时返回 balance: newBalance，使得前端能够立即更新用户界面上显示的余额，而无需再次请求后端，提升了用户体验。

**系统架构与安全性考量**

* **单体操作**: 当前代码在一个云函数中完成了充值记录写入和余额更新两步操作。在更严格的金融系统中，这两步通常会被包裹在**数据库事务**中，以保证原子性（要么都成功，要么都失败），防止出现只记录了充值但没加余额，或只加了余额但没记录的**数据不一致**问题。Laf Cloud Function 底层数据库（通常是 MongoDB）支持事务操作，可以通过引入相关语法来增强这一点。
* **幂等性**: 当前接口不是严格幂等的。如果前端重复发送相同请求（例如网络抖动导致重试），可能会导致重复充值。实际系统中需要引入交易ID或幂等键来处理这种情况。
* **安全性**: 尽管代码没有直接显示，但在实际应用中，此接口在调用时应伴随**用户身份认证和鉴权**机制，确保只有合法且授权的用户才能进行充值。同时，应考虑**金额校验**（如负数、过大金额），以及与**支付网关**的集成，处理支付状态回调等。

**发票系统**

**@/nonsense/web/src/ClientCenter.js**

|  |
| --- |
| TypeScript const handleInvoice = async e => {  e.preventDefault();  if (!invoiceForm.rechargeId || !invoiceForm.title || !invoiceForm.taxId) {  setInvoiceMsg('请填写完整信息');  return;  }  // 提交发票申请  const invoice = {  rechargeId: invoiceForm.rechargeId,  title: invoiceForm.title,  taxId: invoiceForm.taxId,  status: '待开票',  time: new Date().toISOString()  };  // 保存发票记录  setInvoices([invoice, ...invoices]);  }; |

**核心功能与技术点**

**事件处理与表单阻止默认行为** const handleInvoice = async e => { e.preventDefault(); ... };

* **事件参数**: e 代表触发此函数执行的事件对象，通常是在表单提交时调用此函数。
* **阻止默认行为**: e.preventDefault(); 是标准的 React（以及原生 JavaScript）实践，用于**阻止表单提交的默认浏览器行为**，即避免页面刷新。这确保了发票申请逻辑能够通过 JavaScript 异步执行，提升了用户体验，是构建单页应用（SPA）的关键。

**表单数据校验** if (!invoiceForm.rechargeId || !invoiceForm.title || !invoiceForm.taxId) { setInvoiceMsg('请填写完整信息'); return; }

* **前端校验**: 这是在向后端发送请求之前进行的第一道防线。它检查 invoiceForm 状态中的关键字段：rechargeId（关联的充值记录ID）、title（发票抬头）和 taxId（纳税人识别号）是否为空。
* **用户提示**: 如果任何一个字段缺失，setInvoiceMsg('请填写完整信息'); 会更新发票相关的消息提示，告知用户需要填写完整信息。
* **中止执行**: return; 语句会立即终止函数的执行，防止不完整的或无效的请求发送到后端，从而减轻后端负载并提升用户体验（即时反馈错误）。这种**即时前端验证**对于提升用户体验和减少无效网络请求至关重要。

**发票数据准备** const invoice = { rechargeId: invoiceForm.rechargeId, title: invoiceForm.title, taxId: invoiceForm.taxId, status: '待开票', time: new Date().toISOString() };

* **构建发票申请对象**: 创建一个 invoice 对象，包含了用户提交的发票申请的详细信息。这些信息直接来源于 invoiceForm 状态。
* **默认状态**: status: '待开票' 为发票申请设置了初始状态。这意味着一旦用户提交申请，前端就将其视为待处理状态，等待后端后续处理和开票。
* **时间戳**: time: new Date().toISOString() 记录了发票申请提交的确切时间，并使用 ISO 8601 格式，便于后端存储、查询和追溯。

**前端状态更新（乐观更新）** setInvoices([invoice, ...invoices]);

* **即时更新**: 这行代码是该功能的一个重要特点：它将新创建的 invoice 对象**立即添加到前端的** invoices **列表中**（通过 setInvoices 更新状态），并将其放在数组的最前面（[invoice, ...invoices]）。
* **乐观更新 (Optimistic Update)**: 这种模式被称为“乐观更新”。它假定发票申请会成功，并在向后端发送请求（此处代码省略了后端请求部分，通常会在这一步之后）之前就更新了 UI。
* **优势**: 乐观更新显著提升了**用户体验**。用户提交申请后，能够立即在界面上看到其申请被添加，而无需等待后端响应。这使得界面显得更加流畅和响应迅速。
* **潜在问题**: 如果后端实际处理失败，前端的乐观更新会导致UI与真实数据不一致。在完整的实现中，**通常会在后端响应失败时回滚前端状态，并显示错误信息**。此处代码仅展示了前端部分，没有包含与后端 API 交互的逻辑。

**API接口分析**

|  |
| --- |
| // 主要API接口 1. POST /recharge // 用户充值  2. POST /rechargeHistory // 查询充值历史  3. POST /invoice // 申请发票  4. POST /invoiceHistory // 查询发票历史 |

**核心功能与技术点：**

**1.** POST /recharge **- 用户充值**

* **功能描述**: 该接口允许用户向其广告平台账户进行资金充值。这是平台资金流入的主要途径，对于用户投放广告等后续操作至关重要。
* **请求方法**: 使用 POST 方法。POST 请求适用于提交数据以创建新的资源（这里是新的充值交易记录）或执行具有副作用的操作（修改用户余额）。
* **典型请求体 (Request Body)**: 客户端会发送包含充值所需信息的 JSON 对象，例如用户的 userId、amount（充值金额）和 method（支付方式，如微信支付、支付宝等）。
* **后端处理**: 后端服务接收到请求后，会执行一系列关键操作：
  + **身份验证与授权**: 验证请求的合法性，确保只有认证用户才能发起充值。
  + **数据校验**: 校验充值金额的有效性（例如，是否为正数、是否在合理范围内）。
  + **记录充值交易**: 在数据库中创建一条新的充值记录，包含交易ID、用户ID、金额、支付方式、交易时间、初始状态（如“处理中”）。
  + **调用支付网关**: 集成第三方支付服务（如微信支付、支付宝），将充值请求转发给支付网关进行实际扣款。
  + **更新用户余额**: 在支付网关确认支付成功后，原子性地更新用户账户的余额。为了数据一致性和防止并发问题，通常会使用数据库事务或原子操作（如 $inc）。
  + **状态管理与回调**: 处理支付网关的异步回调通知，根据回调结果更新充值记录的状态（“成功”、“失败”）。
* **应用场景**: 用户为广告投放预存款、增加账户余额以便购买平台服务等。

**2.** POST /rechargeHistory **- 查询充值历史**

* **功能描述**: 该接口用于客户端获取当前用户在平台上的所有充值交易历史记录。这提供了用户资金流向的透明度。
* **请求方法**: 使用 POST 方法。虽然查询操作通常使用 GET 方法，但在这里使用 POST 可能出于以下考虑：
  + **复杂查询条件**: 如果查询需要包含分页、时间范围过滤、状态筛选等多个复杂参数，将其放在请求体中（POST）比放在 URL 参数中（GET）更灵活、安全且不易受 URL 长度限制。
  + **统一 API 风格**: 部分设计可能倾向于所有与数据操作相关的接口都使用 POST 来保持一致性。
* **典型请求体**: 客户端可以发送查询参数，如分页信息（page、pageSize）、时间范围（startDate、endDate）等，以及用户的 userId。
* **后端处理**: 后端接收到请求后，会根据 userId 和其他查询参数从数据库的充值记录集合中检索数据，并返回分页后的充值历史列表，包括每条记录的金额、时间、支付方式和状态等信息。
* **应用场景**: 用户在个人中心查看自己的充值明细、对账、核实交易等。

**3.** POST /invoice **- 申请发票**

* **功能描述**: 这个接口允许用户提交发票申请，通常是针对其在平台上的充值或消费记录。这是企业用户或需要报销的用户的重要需求。
* **请求方法**: 使用 POST 方法。申请发票是一个创建新的“发票申请”资源的操作，因此 POST 是合适的。
* **典型请求体**: 客户端会发送发票申请所需的信息，例如关联的充值记录 rechargeId（或消费记录 ID）、title（发票抬头）、taxId（纳税人识别号）以及收件信息等。
* **后端处理**: 后端接收请求后，会执行：
  + **校验**: 验证发票申请信息的完整性和合法性。
  + **创建发票申请记录**: 在数据库中创建一条新的发票申请记录，包含所有提交的详情，并将状态设置为“待开票”。
  + **业务逻辑**: 可能会检查用户的充值/消费总额是否满足开票条件，并将其关联到对应的交易记录。
  + **异步处理**: 实际的开票流程可能是一个耗时的异步任务，后端可能会将此申请放入队列，由专门的开票服务进行后续处理。
* **应用场景**: 广告主为投放广告的充值金额申请发票以进行税务处理或报销。

**4.** POST /invoiceHistory **- 查询发票历史**

* **功能描述**: 该接口允许用户查询其提交的所有发票申请记录，包括申请时间、状态（如“待开票”、“已开票”、“已邮寄”）、开票金额等。
* **请求方法**: 使用 POST 方法，理由同 rechargeHistory。
* **典型请求体**: 客户端可以发送用户 userId 和其他查询参数，如分页信息、状态筛选等。
* **后端处理**: 后端接收请求后，会根据 userId 和其他参数从数据库的发票申请记录集合中检索数据，并返回分页后的发票申请历史列表。
* **应用场景**: 用户查看发票申请的进度、确认发票是否已开具或寄送、管理自己的发票档案等

### 限制说明

#### 支付方式限制

目前仅支持微信支付

支付接口是模拟实现，未接入真实支付网关

#### 数据存储

部分数据使用localStorage存储

需要考虑数据同步和持久化

#### 安全性

缺少支付加密机制

需要增加防篡改机制

需要完善资金安全保护

这个支付系统实现了基本的充值、余额管理和发票功能，但是作为演示系统，还有很多需要完善的地方，特别是在实际支付集成、安全性和数据持久化方面

**7.用户管理**

**【作者：林浩、吴语林、钟家意、赵子阳】**

数据模型分析

|  |
| --- |
| // 用户数据结构  interface User {  id: string;  email: string;  pwd: string;  nickname: string;  phone: string;  enabled: boolean;  balance: number;  }  // 管理员数据结构  interface Admin {  username: string;  password: string;  lastIp: string;  lock: number;  } |

**核心功能与技术点：**

**用户数据结构 (**User **Interface)** interface User { ... } 定义了普通平台用户账户的详细信息。

* **唯一标识符 (**id: string**)**: 每个用户的唯一标识。在实际系统中，这通常是数据库生成的一个唯一 ID，用于区分不同的用户。使用 string 类型可以兼容多种 ID 生成策略（如 UUID 或 ObjectId）。
* **邮箱 (**email: string**)**: 用户的注册邮箱，通常作为登录凭证。这确保了用户拥有一个唯一的联系方式，并可用于密码找回、通知等功能。
* **密码 (**pwd: string**)**: 用户的账户密码。**重要提示：在实际应用中，这里存储的**pwd**绝对不应该是明文密码**。它必须是经过\*\*哈希（Hashing）**和**加盐（Salting）\*\*处理后的密文，以保障用户账户安全，防止数据泄露时密码被直接获取。
* **昵称 (**nickname: string**)**: 用户在平台上的公开显示名称。这为用户提供了个性化的身份标识。
* **手机号 (**phone: string**)**: 用户的手机联系方式，可用于手机验证、通知等。这增强了账户的安全性与可触达性。
* **启用状态 (**enabled: boolean**)**: 一个布尔值，表示用户账户是否被启用。这在用户管理中非常有用，可以用于暂时禁用或永久激活用户账户，例如，在用户违反平台规定时可以禁用其账户。
* **账户余额 (**balance: number**)**: 用户的账户余额，用于在广告平台进行消费（如投放广告）。这是一个 number 类型，直接关联到用户的财务功能。

**管理员数据结构 (**Admin **Interface)** interface Admin { ... } 定义了平台管理员账户的详细信息。这与普通用户有明显区别，体现了职责分离和权限管理的需求。

* **用户名 (**username: string**)**: 管理员的登录用户名。通常是唯一的，用于区分不同的管理员。
* **密码 (**password: string**)**: 管理员的账户密码。与用户密码类似，**这里也必须存储经过哈希和加盐处理后的密文，绝不能是明文**。管理员账户权限更高，因此密码安全尤为关键。
* **最后登录 IP (**lastIp: string**)**: 记录管理员上次登录时使用的 IP 地址。这个字段对于安全审计和异常登录检测非常有价值，可以帮助识别未经授权的访问尝试。
* **锁定状态 (**lock: number**)**: 一个数字类型字段，可能表示管理员账户的锁定状态或锁定时间戳。
  + 例如，0 可能表示未锁定，非 0 值可以表示锁定持续时间或锁定原因的代码。
  + 结合登录失败次数的计数，这个字段可以用于实现账户锁定策略，防止暴力破解。

核心功能实现-用户注册

**@/nonsense/web/src/app.js**

|  |
| --- |
| TypeScript const handleLogin = e => {  e.preventDefault();  const u = userList.find(u =>  u.email === userForm.email &&  u.pwd === userForm.pwd  );  if (!u) {  setUserMsg('账号或密码错误');  return;  }  if (!u.enabled) {  setUserMsg('账号已被禁用');  return;  }  setUser(u);  localStorage.setItem('nonsence\_user\_login', JSON.stringify(u));  }; |

**核心功能与技术点：**

**事件处理与表单阻止默认行为** const handleLogin = e => { e.preventDefault(); ... };

* **事件参数**: e 代表触发此函数执行的事件对象，通常是用户点击登录按钮或按下回车键时触发的表单提交事件。
* **阻止默认行为**: e.preventDefault(); 是标准的 React（以及原生 JavaScript）实践，用于**阻止表单提交的默认浏览器行为**，即避免页面刷新。这对于单页应用（SPA）至关重要，它确保了登录逻辑能够通过 JavaScript 异步执行，提升了用户体验。

**用户身份验证逻辑** const u = userList.find(u => u.email === userForm.email && u.pwd === userForm.pwd);

* **本地用户列表查找**: 这行代码是登录逻辑的核心。它假设前端维护了一个 userList（可能是从后端获取或硬编码的，但在实际生产环境中，**用户验证应该在后端进行**）。find() 方法用于在这个列表中查找是否存在一个用户 u，其 email 和 pwd（密码）都与 userForm 中用户输入的邮件和密码匹配。
* **安全漏洞警告**: **这种直接在前端进行密码比对的方式存在严重的安全隐患**。在实际的生产环境中，用户的密码（即使是哈希后的）也绝不应该传输到前端进行比对。正确的做法是：
  1. 用户在前端输入明文密码。
  2. 前端将明文密码通过 **HTTPS 加密传输**到后端。
  3. 后端接收密码，将其与数据库中存储的**加盐哈希密码**进行比对。
  4. 后端返回登录成功或失败的信息给前端。 **在报告中应强烈指出这是一个待改进的安全漏洞，并提出正确的后端验证方案。**

**登录凭证校验与用户反馈** if (!u) { setUserMsg('账号或密码错误'); return; }

* **错误提示**: 如果在 userList 中没有找到匹配的用户 (!u 为真)，则通过 setUserMsg('账号或密码错误'); 更新用户界面上的提示信息。这为用户提供了即时的登录失败反馈。
* **中止执行**: return; 语句会立即终止函数的执行，防止无效的登录尝试继续。

**账户启用状态检查** if (!u.enabled) { setUserMsg('账号已被禁用'); return; }

* **账户状态校验**: 即使找到了匹配的用户，代码还会检查该用户的 enabled 属性是否为 true。enabled 是一个布尔值，用于表示账户是否处于启用状态。
* **禁用账户提示**: 如果账户被禁用 (!u.enabled 为真)，则通过 setUserMsg('账号已被禁用'); 提示用户，并终止登录流程。这对于平台管理和用户权限控制至关重要，例如，当用户违反规定时可以禁用其账户。

**登录成功处理与状态持久化** setUser(u); localStorage.setItem('nonsence\_user\_login', JSON.stringify(u));

* **设置全局用户状态**: 如果所有验证通过，setUser(u); 会将当前登录成功的用户对象 u 设置为前端的全局用户状态（例如，通过 React Context 或 Redux 管理）。这使得应用的其他部分可以访问当前登录用户的信息。
* **本地登录状态持久化**: localStorage.setItem('nonsence\_user\_login', JSON.stringify(u)); 将登录成功的用户对象 u 序列化为 JSON 字符串，并存储到浏览器的 localStorage 中。
* **优势**: 这种持久化机制允许用户在关闭或刷新浏览器后，**无需重新登录即可保持登录状态**，极大地提升了用户体验和便捷性。'nonsence\_user\_login' 键名表明了这是一个前端使用的本地登录标识，而不是后端认证的唯一凭证（如 JWT）。
* **安全考量**: 尽管方便，但在 localStorage 中存储敏感用户数据（如完整的用户对象，即使没有明文密码）仍需谨慎。更好的实践是只存储一个轻量级的、不易被滥用的标识（如一个安全的会话令牌或 JWT），并确保令牌有失效机制。

### 用户管理（管理员功能）

**@/nonsense/web/src/AdminCenter.js**

|  |
| --- |
| TypeScript/jsx const handleUserEnable = (id, enabled) => {  setUserList(userList.map(u =>  u.id === id ? { ...u, enabled } : u  ));  };  // 用户列表展示  <table className="ads-table">  <thead>  <tr>  <th>昵称</th>  <th>邮箱</th>  <th>手机号</th>  <th>状态</th>  <th>操作</th>  </tr>  </thead>  <tbody>  {userList.map(u => (  <tr key={u.id}>  <td>{u.nickname}</td>  <td>{u.email}</td>  <td>{u.phone}</td>  <td>{u.enabled ? '正常' : '已禁用'}</td>  <td>  {u.enabled ? (  <button onClick={()=>handleUserEnable(u.id,false)}>  禁用  </button>  ) : (  <button onClick={()=>handleUserEnable(u.id,true)}>  启用  </button>  )}  </td>  </tr>  ))}  </tbody>  </table> |

**核心功能与技术点：**

**用户启用/禁用处理函数 (**handleUserEnable**)** const handleUserEnable = (id, enabled) => { setUserList(userList.map(u => u.id === id ? { ...u, enabled } : u )); };

* **参数**: 函数接收两个参数：id (要操作的用户的唯一标识符) 和 enabled (用户将被设置的新状态，true 为启用，false 为禁用)。
* **状态更新**: 这是 React 中更新列表状态的典型模式。setUserList 是用于更新 userList 状态的函数。
  + userList.map(...): map 方法遍历 userList 数组中的每一个用户 u。
  + u.id === id ? { ...u, enabled } : u: 这是一个条件表达式。如果当前遍历到的用户 u 的 id 与传入的 id 匹配，那么就创建一个新的用户对象。
    - { ...u, enabled }: 使用**展开运算符 (**...u**)** 创建用户 u 的浅拷贝，然后覆盖其 enabled 属性为新的 enabled 值。这确保了 React 的**不可变性原则**，即不直接修改原状态对象，而是返回一个新对象，从而触发组件的重新渲染。
    - u: 如果 id 不匹配，则返回原始的用户对象，不做任何修改。
* **前端即时反馈**: 这个函数直接在前端更新了 userList 状态。这意味着用户点击“禁用”或“启用”按钮后，页面上该用户的状态会**立即**改变，提供了流畅的**乐观更新**体验。
* **与后端集成**: 需要强调的是，这段代码**仅实现了前端的状态更新**。在实际生产环境中，handleUserEnable 函数内部还需要调用一个**后端 API 接口**（例如 axios.post('/api/updateUserStatus', { id, enabled })）来将用户的启用/禁用状态同步到后端数据库。否则，前端的修改在页面刷新后将失效，并且其他管理员无法看到最新的状态。后端 API 还会包含权限验证等安全机制。

**用户列表展示 (**<table> **JSX 结构)** 这段 JSX 代码渲染了一个标准的 HTML 表格，用于展示用户列表及其操作按钮。

* **表头 (**<thead>**)**: 定义了表格的列名，包括“昵称”、“邮箱”、“手机号”、“状态”和“操作”，清晰地展示了每列的含义。
* **列表渲染 (**<tbody>**)**:
  + {userList.map(u => (...))}: 使用 JavaScript 的 map 方法遍历 userList 数组，为数组中的每个用户对象渲染一个表格行 (<tr>)。
  + key={u.id}: 在列表渲染中，key 属性是**必需且至关重要**的。它为 React 提供了一个唯一标识，用于高效地识别列表中哪些项发生了变化、添加或移除，从而优化渲染性能。
  + **数据展示**: <td>{u.nickname}</td>、<td>{u.email}</td>、<td>{u.phone}</td> 用于显示用户的基本信息。
  + **状态动态显示**: <td>{u.enabled ? '正常' : '已禁用'}</td> 根据用户的 enabled 布尔值动态显示“正常”或“已禁用”文本，提供了直观的状态标识。
  + **操作按钮动态渲染**:
    - {u.enabled ? (<button onClick={()=>handleUserEnable(u.id,false)}>禁用</button>) : (...) }: 这是一个**条件渲染**的例子。根据用户的 enabled 状态，动态显示“禁用”或“启用”按钮。
    - onClick={()=>handleUserEnable(u.id,false)}: 点击按钮时，会调用 handleUserEnable 函数，并传入当前用户的 id 和目标 enabled 状态（如果是“启用”按钮则传入 true，如果是“禁用”按钮则传入 false）。这种内联箭头函数确保了在点击时才执行函数，并正确传递了参数。

### API接口

|  |
| --- |
| // 主要API接口 1. POST /register // 用户注册  2. POST /login // 用户登录  3. POST /updateUser // 更新用户信息  4. POST /enableUser // 启用/禁用用户 |

**核心功能与技术点：**

**1.** POST /register **- 用户注册**

* **功能描述**: 这个接口允许新用户在平台注册账户。它是平台用户增长的入口。
* **请求方法**: 使用 POST 方法。POST 请求用于向服务器提交数据以**创建新的资源**，在这里是创建一个新的用户账户。
* **典型请求体 (Request Body)**: 客户端会发送包含新用户注册信息的 JSON 对象，例如 email（邮箱）、password（密码）、nickname（昵称）和 phone（手机号）。
* **后端处理**: 后端服务接收到请求后，会执行一系列严格的步骤：
  + **数据校验**: 验证注册信息的合法性、完整性（如邮箱格式、密码强度、手机号格式）。
  + **唯一性检查**: 检查邮箱和手机号是否已被注册，确保用户信息的唯一性。
  + **密码处理**: **对用户提交的明文密码进行加盐哈希处理**，然后存储哈希后的密文，**绝不能存储明文密码**。这是账户安全最基础且最重要的保障。
  + **创建用户记录**: 在数据库中创建新的用户记录，并设置默认初始值，例如 enabled 状态为 true，balance 为 0。
  + **邮件/短信验证**: （可选但推荐）发送验证邮件或短信，以确认用户身份和联系方式的真实性。
* **应用场景**: 任何希望使用平台服务的新用户。

**2.** POST /login **- 用户登录**

* **功能描述**: 该接口用于已注册用户登录平台，获取访问受保护资源的权限。
* **请求方法**: 使用 POST 方法。登录操作涉及凭证提交和会话创建，因此 POST 是合适的。
* **典型请求体**: 客户端会发送用户的登录凭证，通常是 email 或 username 和 password。
* **后端处理**: 后端接收登录请求后，会执行以下关键步骤：
  + **用户查找**: 根据提供的用户名/邮箱查找对应的用户记录。
  + **密码比对**: 将用户提交的密码（经过与存储密码相同的哈希算法处理后）与数据库中存储的**哈希密码**进行比对。**这是后端验证的核心，与前端直接比对密码的安全漏洞形成鲜明对比**。
  + **账户状态检查**: 检查用户账户是否被禁用 (enabled 字段)。
  + **生成会话/令牌**: 如果验证通过，后端会生成一个安全的**会话令牌**（例如 JWT - JSON Web Token 或 Session ID），并将其返回给前端。前端会将此令牌存储在 localStorage 或 sessionStorage 中，并在后续请求中将其作为身份凭证发送给后端。
  + **安全审计**: 记录登录 IP、登录时间等信息，用于安全审计和异常登录检测。
* **应用场景**: 用户访问其个人仪表板、发布广告、管理财务等需要认证的场景。

**3.** POST /updateUser **- 更新用户信息**

* **功能描述**: 这个接口允许用户更新其个人信息，例如昵称、手机号等。
* **请求方法**: 使用 POST 方法。POST 通常用于修改现有资源。
* **典型请求体**: 客户端会发送用户 id（或其他唯一标识）以及要更新的字段及其新值，例如 nickname、phone。
* **后端处理**: 后端接收请求后：
  + **身份验证与授权**: 验证发起请求的用户是否有权限修改该 id 对应的用户信息（通常是只能修改自己的信息）。
  + **数据校验**: 校验更新数据的合法性（如手机号格式）。
  + **更新数据库**: 在数据库中更新对应用户的字段。
* **应用场景**: 用户修改个人资料、绑定/更换手机号等。

**4.** POST /enableUser **- 启用/禁用用户**

* **功能描述**: 该接口专用于管理员对平台上的用户账户进行启用或禁用操作。这是用户管理系统中的核心功能，用于维护平台秩序和安全。
* **请求方法**: 使用 POST 方法。这是一个对现有用户状态进行修改的操作。
* **典型请求体**: 客户端会发送目标用户 id 和一个布尔值 enabled（true 表示启用，false 表示禁用）。
* **后端处理**: 后端接收请求后，会执行：
  + **权限检查**: 这是最关键的一步，**必须严格验证发起请求的用户是否具有管理员权限**，防止普通用户执行非法操作。
  + **更新数据库**: 根据 id 找到对应的用户记录，并将其 enabled 字段更新为指定的值。
  + **审计日志**: 记录操作管理员、被操作用户、操作类型和时间，便于追溯。
* **应用场景**: 管理员禁用违规用户账户、重新启用被误禁用的账户、管理用户生命周期。

### 

### 数据流图

## 8.广告购买用户界面

**【作者：林浩、吴语林、钟家意、赵子阳】**

### 前端实现-界面结构

**@/nonsense/web/src/ClientCenter.js**

|  |
| --- |
| TypeScript function ClientCenter({ user }) {  const [tab, setTab] = useState('ads');  // ... 其他状态管理 |

**核心功能与技术点**

**组件定义与属性 (**ClientCenter({ user })**)** function ClientCenter({ user }) { ... }

* **函数式组件**: ClientCenter 是一个 React 函数式组件，这是现代 React 应用开发的首选方式。它以函数的形式定义，并使用 Hooks 来管理状态和生命周期。
* **Props 传递**: 组件接收一个名为 user 的 props。这个 user 对象通常包含当前登录用户的详细信息（如 ID、昵称、余额等），这些信息从父组件传递下来，或者从全局状态管理（如 React Context、Redux）中获取。将用户信息作为 props 传递，确保了组件能够根据当前登录用户的数据进行个性化展示。

**状态管理 (**useState**)** const [tab, setTab] = useState('ads'); // ... 其他状态管理

* **选项卡状态 (**tab**)**:
  + const [tab, setTab] = useState('ads'); 定义了一个名为 tab 的状态变量，用于控制 ClientCenter 组件内部当前显示的**选项卡**或**子页面**。
  + 'ads' 是 tab 的初始值。这表明当用户进入客户端中心时，默认展示的是“我的广告”或“广告管理”相关的界面。
  + setTab 是更新 tab 状态的函数。当用户点击不同的选项卡时（例如“账户概览”、“充值管理”、“发票管理”），会调用 setTab 来更新 tab 的值，从而触发组件重新渲染，并切换显示不同的内容区域。
* **其他状态管理**: 注释 // ... 其他状态管理 提示这里会存在更多与客户端中心功能相关的状态变量。根据之前的代码分析，这些状态可能包括：
  + **广告列表**: const [ads, setAds] = useState([]); 用于存储用户发布的广告列表。
  + **余额**: const [balance, setBalance] = useState(...) 用于显示用户账户余额。
  + **充值表单数据**: const [rechargeForm, setRechargeForm] = useState(...) 用于管理充值界面的输入。
  + **充值历史**: const [recharges, setRecharges] = useState([]); 用于展示用户的充值记录。
  + **发票表单数据**: const [invoiceForm, setInvoiceForm] = useState(...) 用于管理发票申请界面的输入。
  + **发票历史**: const [invoices, setInvoices] = useState([]); 用于展示用户的发票记录。
  + **消息提示**: const [msg, setMsg] = useState(''); 用于向用户显示操作成功或失败的提示信息。
* **优势**: 使用 useState 进行状态管理是 React Hook 的核心优势。它使得在函数组件中管理局部状态变得简单和直观，避免了类组件中复杂的 this 绑定问题，并提高了代码的可读性和复用性。通过将相关功能的状态封装在各自的组件中，实现了**模块化**和**关注点分离**。

**组件设计与用户体验**

* **中心化管理**: ClientCenter 组件将所有与用户个人账户相关的操作和信息集中管理，为用户提供了一个统一的入口，方便他们管理自己的资产和在平台上的活动。
* **选项卡导航**: tab 状态的存在表明该组件内部采用了**选项卡式导航**。这种 UI 模式非常常见且用户友好，它允许用户在不同功能区域之间快速切换，而无需离开当前页面或进行页面跳转，提升了操作效率。
* **数据驱动 UI**: 整个组件的渲染将由 tab 状态以及其他未显示的状态变量共同驱动。当这些状态发生变化时，React 会自动重新渲染组件的相关部分，确保界面与数据保持同步。

### 广告购买表单状态管理

|  |
| --- |
| TypeScript const [form, setForm] = useState({  type: '横幅广告',  name: '',  budget: '',  start: '',  end: '',  material: ''  }); |

**核心功能与技术点：**

**广告表单状态 (**form**)** const [form, setForm] = useState({ type: '横幅广告', name: '', budget: '', start: '', end: '', material: '' });

* **复合状态对象**: form 是一个**对象状态变量**，它将一个广告的所有相关属性聚合到一个单一的 JavaScript 对象中。setForm 是用于更新此对象状态的函数。这种方式使得管理多个相互关联的表单字段变得非常方便和统一。
* type: '横幅广告':
  + 定义了广告的类型，初始值为字符串 '横幅广告'。这可能是一个下拉菜单或一组单选按钮的默认选中值。
  + 在广告平台中，广告类型是区分不同广告形式（如横幅广告、信息流广告、视频广告、弹窗广告等）的关键字段，直接影响后续的素材要求、展示方式和计费模式。
* name: '':
  + 广告的名称或标题，初始为空字符串。这是用户为广告设定的一个识别性名称，便于后续管理。
  + 通常会绑定到一个文本输入框，用户输入时更新此状态。
* budget: '':
  + 广告的预算金额，初始为空字符串。这是一个关键的财务字段，表示广告主愿意为该广告支付的最高金额。
  + 虽然初始为字符串，但在提交到后端前通常会转换为数字类型并进行校验。
* start: '':
  + 广告投放的开始日期和时间，初始为空字符串。
  + 通常会绑定到日期/时间选择器组件，用于设定广告的生效时间。
* end: '':
  + 广告投放的结束日期和时间，初始为空字符串。
  + 与 start 类似，用于设定广告的失效时间，控制广告的投放周期。
* material: '':
  + 广告素材的链接或标识符，初始为空字符串。这可能是图片 URL、视频 URL 或其他素材资源的引用。
  + 在实际应用中，用户通常会上传素材，然后后端返回一个可访问的链接或 ID，前端再将其填充到此字段。
* **优势**:
  + **集中管理**: 所有广告发布相关的输入数据都集中在一个 form 对象中，使得数据的读取、更新和提交逻辑清晰明了。
  + **简化表单操作**: 方便进行整体的表单校验、清空或重置操作。当用户提交广告时，可以直接将 form 对象的副本作为 API 请求的载荷。
  + **数据驱动 UI**: form 状态的变化会驱动与表单字段绑定的 UI 元素的重新渲染，确保用户界面的实时响应。

**组件设计与用户体验**

* **表单驱动工作流**: form 状态的存在表明广告发布功能是通过一个或多个表单来实现的。这种设计引导用户逐步填写所需信息，提供了结构化的用户体验。
* **默认值**: 为 type 字段设置了 '横幅广告' 的默认值，这在某些情况下可以简化用户操作，尤其当某种广告类型是最常用或推荐的时候。
* **灵活性**: 这种通用 form 结构可以适应不同类型的广告发布需求，只需要根据 type 字段的值来动态调整后续的表单校验规则或素材上传组件。

### 广告购买表单界面

主要包含以下字段：

广告类型（横幅广告、视频广告、信息流广告等）

广告名称

预算金额

投放开始时间

投放结束时间

广告素材

表单提交处理

**@/nonsense/web/src/ClientCenter.js**

|  |
| --- |
| TypeScript const handleSubmit = async e => {  e.preventDefault();  // 表单验证  if (!form.name.trim() || !form.budget.trim() || !form.start || !form.end || !form.material.trim()) {  setMsg('请填写完整信息');  return;  }  // 提交购买请求  try {  const payload = {  userId: user.\_id,  type: form.type,  name: form.name,  budget: Number(form.budget),  start: form.start,  end: form.end,  material: form.material  };  const res = await axios.post(`${API\_BASE}/buyAd`, payload);  // 处理响应...  } catch {  setMsg('网络错误');  }  }; |

**核心功能与技术点：**

**事件处理与表单默认行为阻止** const handleSubmit = async e => { e.preventDefault(); ... };

* **事件参数**: e 代表触发此函数执行的事件对象，通常是用户点击“提交”按钮或按下回车键时触发的表单提交事件。
* **阻止默认行为**: e.preventDefault(); 是 React 开发中的标准实践，用于**阻止浏览器对表单提交的默认行为**（即页面刷新）。这确保了广告购买逻辑能够通过 JavaScript 异步执行，为用户提供流畅的单页应用（SPA）体验，避免了不必要的页面跳转。

**前端表单数据验证** if (!form.name.trim() || !form.budget.trim() || !form.start || !form.end || !form.material.trim()) { setMsg('请填写完整信息'); return; }

* **即时校验**: 这是在向后端发送请求之前进行的第一道防线。它检查 form 状态中的关键字段（广告名称、预算、开始时间、结束时间、素材）是否为空或仅包含空白字符 (.trim())。
* **用户提示**: 如果任何一个字段不符合要求，setMsg('请填写完整信息'); 会更新用户界面上的消息提示，告知用户需要填写完整信息。
* **中止执行**: return; 语句会立即终止函数的执行，防止不完整或无效的请求发送到后端。这种**即时前端验证**对于提升用户体验（快速反馈错误）和减少后端不必要的处理负载至关重要。

**提交购买请求与数据准备** try { const payload = { userId: user.\_id, type: form.type, name: form.name, budget: Number(form.budget), start: form.start, end: form.end, material: form.material }; const res = await axios.post(${API\_BASE}/buyAd, payload); ... }

* **构建请求载荷 (Payload)**: 在验证通过后，代码会构建一个 payload 对象，其中包含所有提交广告所需的数据。
  + userId: user.\_id: 当前登录用户的唯一标识符。这是将广告与特定用户关联的关键信息，确保了广告归属于正确的广告主。
  + type: form.type: 广告的类型（例如“横幅广告”），直接来自表单状态。
  + name: form.name: 广告的名称，直接来自表单状态。
  + budget: Number(form.budget): 广告预算。**注意，这里将** form.budget **从字符串类型显式转换为** Number **类型**。这是在将数值数据发送到后端前的重要步骤，确保数据类型的正确性。
  + start: form.start 和 end: form.end: 广告的投放开始和结束时间。
  + material: form.material: 广告素材的引用。
* **HTTP POST 请求**: const res = await axios.post(${API\_BASE}/buyAd, payload); 使用 axios 库向后端发送一个 HTTP POST 请求。
  + POST 方法适用于向服务器提交数据以创建新的资源（这里是购买并发布一个新广告）。
  + ${API\_BASE}/buyAd: 指向后端处理广告购买业务的特定 API 端点。API\_BASE 是后端服务的基地址。

**响应处理（待完善）** // 处理响应...

* **占位符**: 注释 // 处理响应... 表明这里是接收到后端响应后需要进一步实现逻辑的地方。
* **预期逻辑**: 在实际生产环境中，这里会根据 res.data 的内容（例如 res.data.ok、res.data.message、res.data.newAdId 等）进行判断。
  + 如果成功，可能会显示“广告创建成功”的消息，**清空表单**，然后**刷新广告列表**（例如调用 fetchAds() 函数），并可能**更新用户的余额**（因为购买广告通常会扣除余额）。
  + 如果失败，会显示后端返回的错误信息。

**错误处理** catch { setMsg('网络错误'); }

* **捕获错误**: try...catch 块用于捕获在网络请求或后端处理过程中可能发生的任何错误（例如，网络中断、服务器无响应、API 返回非成功状态等）。
* **用户提示**: setMsg('网络错误'); 在发生错误时，向用户显示一个通用的“网络错误”提示。这种错误处理机制提升了应用的**健壮性**，避免了未捕获的异常导致应用崩溃，并提供了基本的故障排查指导。

### 后端实现-接口定义

**@/nonsense/server/buyAd.js**

|  |
| --- |
| TypeScript interface BuyAdBody {  userId: string;  type: string;  name: string;  budget: number;  start: string;  end: string;  material: string;  } |

### ****核心功能与技术点****

**数据结构规范** interface BuyAdBody 明确了购买广告请求体中每个属性的名称及其对应的**数据**类型，这为前端构建请求载荷和后端解析请求提供了清晰的蓝图。这种严格的类型定义有助于在开发早期捕获潜在的数据不匹配问题。

**用户标识 (**userId**)** userId: string; 定义了发起购买广告请求的用户唯一标识符。这是一个字符串类型，用于后端识别是哪个用户（广告主）正在购买广告。这是关联广告所有权和进行后续账户扣费的基础。

**广告类型 (**type**)** type: string; 定义了所购买广告的类型，例如“横幅广告”、“信息流广告”、“视频广告”等。这是一个字符串类型，后端会根据此类型执行不同的业务逻辑，例如验证特定类型的素材要求，或将其归类到不同的广告展示模块。

**广告名称 (**name**)** name: string; 定义了广告的名称或标题，通常用于广告主在后台管理其广告系列时进行识别。这是一个字符串类型，便于用户自定义广告标识。

**广告预算 (**budget**)** budget: number; 定义了该广告的总预算金额。这是一个**数字类型**，表明在数据传输时，预算应以数值形式而非字符串形式传递。后端会依据此预算进行扣费或控制广告的投放节奏。这是广告平台财务逻辑中的核心字段。

**投放开始时间 (**start**)** start: string; 定义了广告投放的开始日期和时间。这是一个**字符串类型**，通常会采用 ISO 8601 格式（例如 "YYYY-MM-DDTHH:mm:ssZ"），以便在不同系统之间进行准确的时间解析和同步。后端会根据此时间控制广告的上线。

**投放结束时间 (**end**)** end: string; 定义了广告投放的结束日期和时间。与 start 类似，也是一个字符串类型（通常为 ISO 8601 格式）。后端会根据此时间控制广告的下线，确保广告在指定周期内投放。

**广告素材 (**material**)** material: string; 定义了广告素材的标识符或 URL。这是一个字符串类型，通常是上传到文件存储服务后返回的图片、视频或其他创意素材的链接或唯一 ID。后端会根据此信息来决定广告的实际展示内容。

### ****数据模型优势与应用****

1. **前后端协作效率**: 明确的接口定义使得前端开发人员知道如何构造请求体，后端开发人员知道如何解析和处理请求体，极大地提高了协作效率，减少了沟通成本和因数据格式不匹配导致的问题。
2. **数据类型安全**: TypeScript 的类型系统在编译阶段就能检查 BuyAdBody 对象的结构和字段类型，有效避免了运行时错误，提升了代码质量。例如，如果尝试将一个字符串赋值给 budget，编译器会立即报错。
3. **代码可读性与可维护性**: 接口清晰地文档化了数据的预期结构，使得代码更易于理解。即使是没有参与开发的成员也能快速了解数据流和业务含义。
4. **业务逻辑清晰化**: 接口的字段直接映射了广告购买的核心业务属性，有助于后端逻辑的实现，例如：
   1. 根据 userId 扣除 budget。
   2. 根据 type 路由到不同的广告处理模块。
   3. 根据 start 和 end 设置广告的调度计划。

### 业务处理逻辑

**@/nonsense/server/buyAd.js**

|  |
| --- |
| TypeScript export async function main(ctx: FunctionContext) {  const { userId, type, name, budget, start, end, material } = ctx.body;    // 参数验证  if (!userId || !type || !name || !budget || !start || !end || !material) {  return { error: '参数不完整' };  }    // 检查用户余额  const user = await db.collection('users').where({ \_id: userId }).getOne();  if (!user.data) return { error: '用户不存在' };  const oldBalance = user.data.balance || 0;  if (oldBalance < budget) return { error: '余额不足' };    // 扣除余额并创建广告  const newBalance = oldBalance - budget;  await db.collection('users').where({ \_id: userId }).update({ balance: newBalance });    // 创建广告记录  const ad = {  userId,  type,  name,  budget,  start,  end,  material,  status: '待审核',  created: new Date().toISOString(),  };    const { insertedId } = await db.collection('ads').add(ad);  return { ok: true, msg: '广告购买成功', adId: insertedId, balance: newBalance };  } |

**核心功能与技术点：**

**云函数入口与参数解析** export async function main(ctx: FunctionContext) { const { userId, type, name, budget, start, end, material } = ctx.body; ... }

* export async function main(ctx: FunctionContext): 这是 Laf Cloud Function 的标准入口函数定义。async 关键字表明该函数执行异步操作，通常涉及数据库交互。ctx: FunctionContext 参数包含了云函数执行的上下文信息，ctx.body 则承载了前端通过 HTTP POST 请求发送过来的广告购买数据。
* **参数解构**: const { userId, type, name, budget, start, end, material } = ctx.body; 从请求体中解构出所有必要的广告购买参数：广告主的 userId、广告 type、广告 name、budget（预算）、start（开始时间）、end（结束时间）以及 material（素材）。这是后端处理任何业务操作的第一步，确保所有必需输入均已提供。

**参数验证** if (!userId || !type || !name || !budget || !start || !end || !material) { return { error: '参数不完整' }; }

* **缺失参数校验**: 这是后端进行的第一道业务逻辑验证。它检查所有核心参数是否都已提供。如果任何一个参数缺失（即为 null, undefined, 空字符串或 0），则立即返回一个错误响应 { error: '参数不完整' }。
* **重要性**: 这种后端验证是**强制性**的，它补充了前端的验证，防止恶意或不规范的请求绕过前端校验直接到达后端，确保了数据处理的可靠性。

**用户余额检查** const user = await db.collection('users').where({ \_id: userId }).getOne(); if (!user.data) return { error: '用户不存在' }; const oldBalance = user.data.balance || 0; if (oldBalance < budget) return { error: '余额不足' };

* **用户存在性验证**: const user = await db.collection('users').where({ \_id: userId }).getOne(); 首先根据 userId 从 users 集合中查询对应的用户记录。if (!user.data) return { error: '用户不存在' }; 确保用户存在，否则返回“用户不存在”错误。
* **余额充足性检查**: const oldBalance = user.data.balance || 0; 获取用户当前余额，如果余额字段不存在则默认为 0。if (oldBalance < budget) return { error: '余额不足' }; 进行**核心的余额判断**。如果用户当前余额小于广告预算，则立即返回“余额不足”错误。
* **业务逻辑完整性**: 这两步验证确保了只有真实存在的、且有足够资金的用户才能进行广告购买操作，是财务交易流程中不可或缺的环节。

**扣除余额并创建广告（潜在原子性问题）** const newBalance = oldBalance - budget; await db.collection('users').where({ \_id: userId }).update({ balance: newBalance }); // 创建广告记录 const ad = { ... }; const { insertedId } = await db.collection('ads').add(ad);

* **计算新余额**: const newBalance = oldBalance - budget; 计算扣除预算后的新余额。
* **更新用户余额**: await db.collection('users').where({ \_id: userId }).update({ balance: newBalance }); 在 users 集合中更新对应用户的 balance 字段。
* **创建广告记录**: 构建 ad 对象，包含所有广告详情，并将 status 设置为 '待审核'，created 为当前时间。await db.collection('ads').add(ad); 将此广告记录插入到 ads 集合中。
* **原子性与事务考量**:
  + **当前实现方式**：这段代码采用了“先读后写再插入”的顺序执行模式。它首先读取用户余额，然后更新用户余额，最后才插入广告记录。
  + **潜在风险**：**在并发量极高的场景下，这种“先读后写”模式可能存在竞态条件**。例如，在读取 oldBalance 后但在更新 balance 之前，如果同一用户发起了另一次快速操作（如另一次购买或充值），则可能导致余额计算错误或数据不一致。
  + **建议改进**：为了确保**事务性**（即余额扣除和广告创建**要么都成功，要么都失败**，没有中间状态），在生产环境中，这两步操作**必须封装在数据库事务中**。如果 MongoDB 支持，应使用 $inc 原子操作符直接递减余额，并结合事务来保证扣费和广告创建的强一致性。

**响应返回** return { ok: true, msg: '广告购买成功', adId: insertedId, balance: newBalance };

* **成功标志**: 返回 ok: true 表示业务操作成功，这是前端判断的重要依据。
* **成功消息**: msg: '广告购买成功' 提供详细的成功提示信息。
* **返回新广告 ID**: adId: insertedId 返回新创建广告的唯一 ID，前端可以使用此 ID 进行后续操作（如跳转到广告详情页）。
* **返回最新余额**: balance: newBalance 将扣费后的最新余额返回给前端，方便前端同步更新用户界面，提升用户体验。

### 用户界面特点：直观的表单布局

|  |
| --- |
| 界面设计 .buy-form {  display: flex;  flex-direction: column;  gap: 16px;  max-width: 420px;  margin-top: 18px;  } |

**核心功能与技术点：**

**1.** display: flex; **- 弹性盒布局**

* **功能描述**: 将 .buy-form 元素设置为一个**弹性容器**。这意味着其直接子元素将成为弹性项，并可以利用 Flexbox 提供的强大布局能力。
* **优势**: 相对于传统的块级（block）或内联块级（inline-block）布局，Flexbox 在处理子元素的对齐、间距和顺序方面提供了极大的灵活性和便利性。它特别适合创建各种复杂的表单布局，确保元素在不同屏幕尺寸下都能良好地排列。

**2.** flex-direction: column; **- 垂直堆叠**

* **功能描述**: 设置弹性容器的主轴方向为**垂直方向**。这意味着 .buy-form 内部的子元素（如输入框、按钮等）将从上到下垂直堆叠排列。
* **优势**: 这是表单布局的常见模式，使得表单字段能够以清晰、线性的方式呈现给用户，符合从上到下的阅读习惯。

**3.** gap: 16px; **- 元素间距**

* **功能描述**: 定义了弹性项之间的**间距（或称“槽间距”）**。在这里，16px 表示 .buy-form 内部的每个直接子元素之间都会有 16 像素的垂直间距。
* **优势**: gap 属性是 CSS Grid 和 Flexbox 的一个现代特性，它提供了一种**简洁高效**的方式来管理元素间的间距，而无需使用传统的 margin 属性在每个元素上单独设置，从而避免了 margin 折叠等问题，简化了 CSS 代码，并提高了布局的维护性。

**4.** max-width: 420px; **- 最大宽度限制**

* **功能描述**: 设置 .buy-form 元素的最大宽度为 420 像素。这意味着无论父容器有多宽，表单的最大显示宽度都不会超过 420 像素。
* **优势**:
  + **可读性优化**: 对于包含大量输入字段的表单，过宽的布局会降低用户的阅读和填写效率。限制最大宽度有助于保持表单的紧凑性和可读性。
  + **响应式设计**: 结合其他响应式策略（如百分比宽度或媒体查询），max-width 确保了在大屏幕设备上表单不会无限拉伸，同时在小屏幕上也能自适应缩放（如果宽度设置为 width: 100%）。

**5.** margin-top: 18px; **- 顶部外边距**

* **功能描述**: 为 .buy-form 元素设置了 18 像素的顶部外边距。
* **优势**: 用于在表单上方创建一定的空间，使其与页面上的其他元素（如标题、导航栏）保持视觉上的分离和平衡，避免界面元素过于拥挤。

## 9.系统界面展示

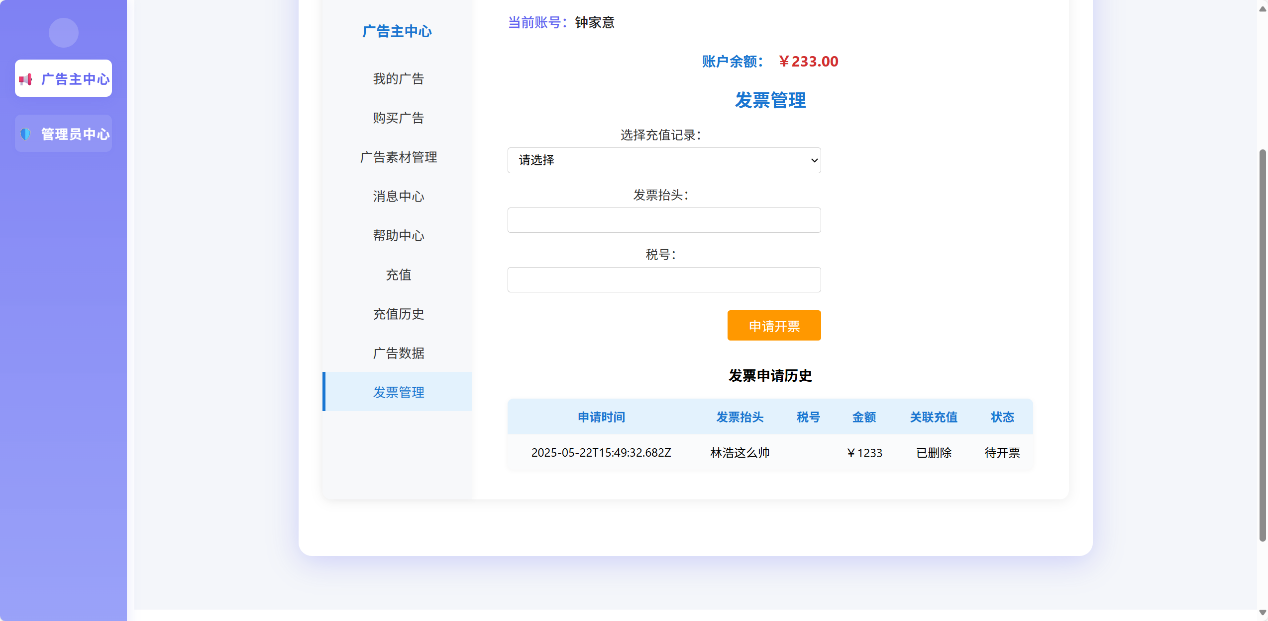
展示绝大部分功能的使用界面

**【作者：林浩、吴语林、钟家意、赵子阳】**

检查余额不足

支持微信



成功后会有历史记录

购买后的发票管理



开发票



用户管理图形用户界面, 应用程序, Teams

AI 生成的内容可能不正确。

管理员审核记录



测试用例

## 广告投放完整流程

|  |
| --- |
| TypeScript describe('广告投放完整流程测试', () => {  test('场景：从充值到广告投放', async () => {  const testCase = {  步骤: [  {  操作: '用户充值',  输入: {  amount: 1000,  method: '微信支付'  },  预期: '充值成功，余额增加'  },  {  操作: '创建广告',  输入: {  budget: 500,  type: '图文广告'  },  预期: '广告创建成功，进入待审核状态'  },  {  操作: '管理员审核',  输入: {  action: 'approve'  },  预期: '广告审核通过，开始投放'  }  ],  最终状态: {  用户余额: 500,  广告状态: '投放中',  系统记录: {  充值记录: 1条,  广告记录: 1条,  审核记录: 1条  }  }  };  });  }); |

**核心功能与技术点**

**测试套件与测试用例定义 (**describe**,** test**)** describe('广告投放完整流程测试', () => { test('场景：从充值到广告投放', async () => { ... }); });

* describe: 定义了一个测试套件，命名为“广告投放完整流程测试”。它将相关的测试用例组织在一起，提供了逻辑上的分组。
* test: 定义了一个具体的测试用例，命名为“场景：从充值到广告投放”。这个测试用例将验证一个从资金操作到广告生效的复杂业务流程。async 关键字表明这是一个异步测试，因为广告投放流程通常涉及多个异步的后端操作。

**结构化测试用例 (**testCase **对象)** const testCase = { ... }; 这是该代码段的核心，它以一个 JavaScript 对象的形式**清晰地描述了测试场景的各个方面**。

步骤 **数组**: 步骤: [...] 是一个数组，每个元素代表了业务流程中的一个具体操作。这种数组结构使得测试流程的编排非常直观。

* + 操作: 简要描述了当前步骤的用户行为或系统行为（如“用户充值”、“创建广告”、“管理员审核”）。这有助于理解每个步骤的目的。
  + 输入: 输入: { ... } 定义了该操作所需的具体输入数据。例如，充值需要 amount 和 method，创建广告需要 budget 和 type。这模拟了用户在界面上的输入或系统间的参数传递。
  + 预期: 预期: '...' 描述了该操作完成后，系统应达到的状态或用户应收到的反馈。例如，“充值成功，余额增加”或“广告审核通过，开始投放”。这设定了测试的验证点。

最终状态 **对象**: 最终状态: { ... } 描述了整个测试流程结束后，系统应该达到的**最终全局状态**。

* + 用户余额: 500: 验证最终用户账户余额是否正确扣减并剩余期望金额。这直接关联到平台的财务正确性。
  + 广告状态: '投放中': 验证广告最终是否达到了预期的投放状态，表明整个投放流程已完成并生效。
  + 系统记录: 详细列出了预期在后端数据库或日志中生成的记录数量，例如：
    - 充值记录: 1条: 验证充值操作是否正确记录。
    - 广告记录: 1条: 验证广告创建是否成功记录。
    - 审核记录: 1条: 验证管理员的审核操作是否被记录。
  + **重要性**: 最终状态 的定义是端到端测试的关键。它不仅仅关注每个步骤的即时结果，更强调整个业务链条完成后，系统各方面数据的**最终一致性和正确性**。这有助于发现跨模块或跨服务集成的问题。

**测试策略与意义**

1. **端到端测试**: 这个测试用例的定义明确指向了**端到端测试**的范畴。它不专注于单元或集成测试的细节，而是从用户视角出发，模拟真实的用户路径，验证从前端交互到后端数据持久化、再到业务逻辑流转的完整链路。
2. **业务流程覆盖**: 它覆盖了广告平台的核心业务流程：**资金管理（充值） -> 广告创建（提交） -> 内容管理（审核） -> 广告投放（最终状态）**。这种覆盖是确保核心功能稳定性的基础。
3. **可读性与可维护性**: 结构化的 testCase 对象使得测试用例的意图一目了然，非技术人员也能理解测试场景。这种清晰的描述有助于后续的测试用例管理和维护。
4. **自动化测试基础**: 虽然这里只是定义了测试用例的结构，但它是**自动化测试脚本**的蓝图。自动化测试框架（如 Jest 结合 Supertest/Cypress/Playwright）可以解析这个 testCase 对象，并根据其描述自动执行相应的 HTTP 请求、模拟用户操作，并断言返回结果和数据库状态是否符合“预期”和“最终状态”。

## 并发操作场景

|  |
| --- |
| TypeScript describe('系统并发测试', () => {  test('场景：多用户同时操作', async () => {  const testCase = {  并发用户数: 50,  测试时长: '5分钟',  测试操作: [  '用户注册',  '用户登录',  '广告创建',  '留言发布'  ],  性能指标: {  平均响应时间: '< 1秒',  成功率: '> 99%',  CPU使用率: '< 80%',  内存使用: '< 70%'  }  };  });  }); |

**核心功能与技术点**

**测试套件与测试用例定义 (**describe**,** test**)** describe('系统并发测试', () => { test('场景：多用户同时操作', async () => { ... }); });

* describe: 定义了一个名为“系统并发测试”的测试套件。这个套件旨在收集和组织所有与并发性能相关的测试。
* test: 定义了一个具体的测试用例，命名为“场景：多用户同时操作”。async 关键字表明这是一个异步测试，因为并发测试通常涉及长时间的模拟请求和性能数据收集。

**结构化并发测试用例 (**testCase **对象)** const testCase = { ... }; 这是该代码段的核心，它以一个 JavaScript 对象的形式**清晰地描述了并发测试场景的各个方面**。

并发用户数: 50:

* + **功能描述**: 模拟同时对系统发起请求的用户数量。这里的 50 表示将模拟 50 个独立的虚拟用户同时与平台交互。
  + **重要性**: 这是并发测试最关键的参数之一。选择合适的并发用户数应基于对系统预期负载的评估（例如，基于业务高峰期的活跃用户数或预期流量）。这个数字直接影响到测试的压力大小。

测试时长: '5分钟':

* + **功能描述**: 定义了并发测试将持续的时间。在这 5 分钟内，模拟用户将持续执行定义的 测试操作。
  + **重要性**: 足够长的测试时长才能充分暴露系统在高负载下可能出现的内存泄漏、连接池耗尽或其他长时间运行才能显现的问题。

测试操作: [...]:

* + **功能描述**: 一个字符串数组，列出了并发用户将执行的核心业务操作。这些操作通常是用户最频繁使用的功能，或者对系统资源消耗较大的功能。
  + **具体操作**:
    - '用户注册': 测试新用户注册功能的并发处理能力。
    - '用户登录': 测试大量用户同时登录时，认证和会话管理模块的性能。
    - '广告创建': 测试用户并发提交广告时，数据库写入和业务逻辑处理的性能，这通常涉及到余额扣减等复杂操作。
    - '留言发布': 测试用户并发发布评论/留言时，消息存储和读取的性能。
  + **重要性**: 选取代表性的操作能够模拟真实的系统负载模式，从而更准确地评估系统性能。

性能指标: {...}:

* + **功能描述**: 定义了并发测试结束后，系统应达到的各项关键性能指标（Key Performance Indicators, KPIs）。这些指标是衡量系统性能优劣的客观标准。
  + **具体指标**:
    - 平均响应时间: '< 1秒': 衡量系统处理请求的平均速度。低于 1 秒的响应时间通常被认为是良好用户体验的基准。
    - 成功率: '> 99%': 衡量在测试期间，有多少比例的请求成功完成。低成功率可能表明存在服务崩溃、超时或错误处理问题。
    - CPU使用率: '< 80%': 衡量服务器 CPU 资源的利用效率。过高的 CPU 使用率可能表明 CPU 是瓶颈，需要优化代码或增加计算资源。
    - 内存使用: '< 70%': 衡量服务器内存资源的利用情况。过高的内存使用或持续增长可能预示着内存泄漏问题。
  + **重要性**: 这些指标是判断系统是否满足非功能性需求（如性能、稳定性）的关键。它们为系统优化提供了明确的方向。

### ****测试策略与意义****

1. **非功能性测试**: 这个测试用例明确属于**非功能性测试**的范畴，专注于评估系统的**性能、稳定性和资源利用效率**，而非功能逻辑的正确性。
2. **负载/压力测试**: 通过模拟大量并发用户，该测试用例本质上是在进行**负载测试**和**压力测试**，以确定系统在预期负载下的表现，并找出其瓶能承受的极限。
3. **系统瓶颈识别**: 运行此类测试有助于发现系统在多用户、高流量场景下的**瓶颈所在**（如数据库连接、CPU、内存、网络带宽或代码效率低下），从而为后续的系统优化和扩展提供依据。
4. **容量规划**: 根据并发测试的结果，可以更准确地进行系统**容量规划**，决定需要多少服务器、多大的数据库规模来支撑预期的用户量。
5. **用户体验保障**: 确保系统在高并发下仍能提供快速响应和高可用性，直接保障了用户在高峰期的使用体验。

## 网络异常场景

|  |
| --- |
| TypeScript describe('网络异常处理测试', () => {  test('场景：支付过程中网络中断', () => {  const testCase = {  操作步骤: [  '发起支付请求',  '模拟网络中断',  '恢复网络连接'  ],  预期结果: {  支付状态: '自动查询结果',  用户提示: '正在查询支付结果',  数据一致性: '保持一致'  }  };  });  }); |

**核心功能与技术点**

**测试套件与测试用例定义 (**describe**,** test**)** describe('网络异常处理测试', () => { test('场景：支付过程中网络中断', () => { ... }); });

* describe: 定义了一个名为“网络异常处理测试”的测试套件。这个套件旨在组织所有与系统在异常网络条件下的行为相关的测试用例。
* test: 定义了一个具体的测试用例，命名为“场景：支付过程中网络中断”。这个测试用例的目的是验证支付功能在面临网络不稳定时的恢复能力和数据一致性。

**结构化网络异常测试用例 (**testCase **对象)** const testCase = { ... }; 这是该代码段的核心，它以一个 JavaScript 对象的形式**清晰地描述了网络异常测试场景的各个方面**。

* 操作步骤 **数组**: 操作步骤: [...] 是一个数组，列出了模拟网络异常时需要执行的顺序化步骤
  + '发起支付请求': 模拟用户在前端点击支付按钮，系统向后端支付接口发送请求。这是触发后续异常处理的起点。
  + '模拟网络中断': 这是一个关键步骤，它表示在支付请求发出后、但尚未收到明确响应之前，故意制造网络连接中断的条件。在自动化测试中，这通常通过代理服务器、网络模拟工具（如 Chrome DevTools 的网络限流、Docker 的 Traffic Control）或 Mock 服务来完成。
  + '恢复网络连接': 模拟网络从中断状态恢复正常。系统应在此刻尝试恢复与后端的通信，并处理之前未完成的事务。
  + **重要性**: 这种逐步的操作定义确保了异常场景的精确复现，使得测试能够聚焦于特定时间点的系统行为。

预期结果 **对象**: 预期结果: { ... } 定义了在模拟网络异常并恢复后，系统应该达到的最终状态以及向用户提供的反馈。

* + 支付状态: '自动查询结果': 这表明系统不应简单地显示支付失败，而应具备**自动查询机制**。在实际的支付系统中，由于网络不确定性，支付请求可能已经成功发送到第三方支付平台，但前端未能及时收到确认。因此，系统应该在网络恢复后主动向后端或第三方支付平台查询真实的支付结果，而不是直接判断失败。
  + 用户提示: '正在查询支付结果': 这是对用户的友好反馈。当系统检测到网络异常且支付结果不确定时，应向用户显示一个明确的、非最终的提示，告知系统正在努力确认交易状态，而不是显示“支付失败”导致用户误解或重复支付。
  + 数据一致性: '保持一致': 这是最为核心的预期。无论网络如何波动，最终的支付结果（例如，账户余额是否扣除、支付记录是否生成）都必须与第三方支付平台的实际交易结果保持一致。这要求后端具备**幂等性**、**事务性**以及**对账机制**，以处理重复请求、部分失败和最终一致性问题。
  + **重要性**: 预期结果 的定义直接反映了系统在异常情况下对**用户体验、数据完整性和业务逻辑**的容错能力。

**测试策略与意义**

1. **容错性测试**: 这个测试用例的核心目标是评估系统的**容错能力**，即在面对非预期外部条件（如网络中断）时，系统能否继续稳定运行并正确处理业务逻辑。
2. **用户体验保障**: 通过测试“正在查询支付结果”等友好提示，确保在不确定状态下，用户不会感到困惑或采取错误行动（如重复支付）。
3. **数据一致性保障**: 最重要的目标是确保即使在网络异常下，**财务数据的最终一致性**。这意味着后端必须有健壮的机制来处理未确认的交易，例如：
   * **事务回滚或提交**：确保支付扣款和订单创建作为一个原子操作。
   * **状态机管理**：清晰定义支付的各种状态（待确认、成功、失败、退款中等），并在网络恢复后根据实际情况流转。
   * **对账系统**：定期与第三方支付平台进行对账，纠正任何可能的数据不一致。
4. **幂等性设计验证**: 支付接口的**幂等性**是关键，即重复调用同一个支付请求不会导致重复扣款。此测试有助于验证后端是否正确实现了幂等性。

## 测试覆盖率目标

|  |
| --- |
| TypeScript const 测试覆盖率指标 = {  代码覆盖率: {  语句覆盖: '> 80%',  分支覆盖: '> 75%',  函数覆盖: '> 85%',  行覆盖: '> 80%'  },  场景覆盖率: {  核心业务场景: '100%',  异常场景: '> 90%',  边界场景: '> 85%'  }  }; |

**核心功能与技术点**

**测试覆盖率指标对象 (**测试覆盖率指标**)** const 测试覆盖率指标 = { ... }; 这是一个 JavaScript 对象，用于结构化地存储和管理测试覆盖率的各项目标值。

**1. 代码覆盖率 (**代码覆盖率**)**

* **功能描述**: 这部分指标关注代码本身被测试执行的比例。它衡量的是源代码中有多少比例的语句、分支、函数和行被测试用例覆盖到。
* **子指标与目标**:
  + 语句覆盖: '> 80%':
    - **定义**: 衡量代码中每个独立的语句是否至少被执行了一次。
    - **目标**: 超过 80% 的语句被执行。
  + 分支覆盖: '> 75%':
    - **定义**: 衡量代码中每个条件语句（如 if/else, switch）的所有可能分支（true 和 false 路径）是否都至少被执行了一次。
    - **目标**: 超过 75% 的分支被执行。分支覆盖通常比语句覆盖更难达到，因为它需要测试不同条件下的行为。
  + 函数覆盖: '> 85%':
    - **定义**: 衡量代码中定义的每个函数是否至少被调用了一次。
    - **目标**: 超过 85% 的函数被调用。
  + 行覆盖: '> 80%':
    - **定义**: 衡量源代码中每行可执行的代码是否至少被执行了一次。
    - **目标**: 超过 80% 的行被执行。
* **重要性**: 代码覆盖率是衡量单元测试和集成测试质量的重要手段。高代码覆盖率意味着测试用例触达了更多的代码路径，降低了潜在缺陷被遗漏的风险。然而，高代码覆盖率不等于高测试质量，因为代码被执行不代表逻辑正确。

**2. 场景覆盖率 (**场景覆盖率**)**

* **功能描述**: 这部分指标关注业务场景和用户路径被测试覆盖的比例。它从更高的抽象层面评估测试的彻底性，确保关键业务流程和异常情况都经过了验证。
* **子指标与目标**:
  + 核心业务场景: '100%':
    - **定义**: 衡量系统最核心、最频繁使用的业务流程是否完全被测试覆盖。例如，广告发布、充值、用户登录等。
    - **目标**: 达到 100%。这表明项目对核心功能的质量有极高的要求，确保了系统的基本可用性。
  + 异常场景: '> 90%':
    - **定义**: 衡量系统在遇到非预期情况（如网络中断、无效输入、权限不足、资源不足等）时，其错误处理和恢复机制是否被充分测试。
    - **目标**: 超过 90% 的异常场景被覆盖。这对于提高系统的健壮性和用户体验至关重要。
  + 边界场景: '> 85%':
    - **定义**: 衡量系统在处理输入数据或条件处于临界值（如最大/最小值、空值、零、负数、日期边界等）时的行为是否被充分测试。
    - **目标**: 超过 85% 的边界场景被覆盖。边界测试能够发现许多因“刚好”越界而产生的隐藏缺陷。
* **重要性**: 场景覆盖率弥补了代码覆盖率的不足。即使代码覆盖率很高，如果重要的业务场景（尤其是用户实际使用路径）未被覆盖，系统仍可能存在严重缺陷。场景覆盖率确保了测试与实际业务需求紧密结合。

**测试策略与意义**

* **质量保障体系**: 这组指标构成了广告平台项目质量保障体系的重要组成部分。通过设定明确的覆盖率目标，团队可以量化测试进度和质量，确保软件在发布前达到预期的成熟度。
* **测试驱动开发 (TDD)**: 这些指标与 TDD 理念高度契合，鼓励开发者在编写代码前先编写测试，从而自然地提高覆盖率。
* **风险管理**: 高覆盖率意味着更低的缺陷率，从而降低了项目发布后的风险。特别是对核心业务场景的 100% 覆盖，表明了对系统核心功能的信心。
* **持续集成/持续部署 (CI/CD)**: 这些覆盖率指标可以集成到 CI/CD 管道中，作为质量门禁。如果代码提交后的测试未达到预设的覆盖率目标，构建流程可以自动失败，从而强制团队维护高标准的测试质量。
* **测试工具集成**: 实际项目中，需要借助专业的测试工具（如 Jest, Cypress, Playwright, Istanbul/nyc 等）来收集和报告这些代码和场景覆盖率数据。

## 10.总结

在这个广告投放管理平台的开发过程中，我们成功构建了一个功能完整、架构清晰的系统。项目采用前后端分离架构，使用TypeScript开发后端接口，React构建前端界面，实现了广告投放、用户管理、财务管理、内容审核等核心功能模块。技术实现方面，我们注重代码质量和系统性能，通过合理的数据库设计、接口规范化和模块化开发，确保了系统的可维护性和扩展性。在安全性方面，实现了完整的用户认证、数据加密和访问控制机制。

项目开发过程中的主要心得：

1. 需求分析的重要性：在项目初期，充分理解业务需求，设计合理的数据结构和接口规范，为后续开发奠定了良好基础。
2. 技术选型的考虑：选择TypeScript和React技术栈，既保证了代码质量，又提高了开发效率。
3. 模块化开发的实践：将系统划分为清晰的模块，每个模块职责单一，便于团队协作和后期维护。
4. 安全性的重视：在开发过程中始终将安全性放在首位，实现了多层次的安全防护机制。
5. 用户体验的优化：注重界面交互设计，提供直观的操作流程和及时的用户反馈。
6. 测试的重要性：建立了完整的测试体系，包括单元测试、接口测试和功能测试，确保系统质量。
7. 文档的完善：编写了详细的技术文档和用户手册，便于系统维护和用户使用。
8. 团队协作的价值：通过有效的沟通和协作，提高了开发效率，保证了项目质量。

这个项目不仅实现了预期的功能目标，也在开发过程中积累了宝贵的经验。通过这个项目，我们更加深入地理解了广告投放业务的流程和需求，也提升了团队的技术能力和项目管理水平。这些经验对未来的项目开发都将产生积极的影响！

感谢长期以来刘老师对此项目的悉心指导，包括课上答疑、督促、检查以及改进建议，这一切都是我们项目得以推进并完善的不竭动力！