目录

1 引言

1.1 题目背景与意义

在组织的运营管理过程中，审批流程是保障业务规范、风险可控的核心环节。传统的人工审批模式存在流程不透明、审批效率低、单据易丢失、统计分析困难等问题。随着信息化技术的普及，开发一套贴合实际业务场景的审批管理信息系统（Approval-MIS），成为提升组织审批效率、优化管理流程的必然趋势。

本次课程设计以审批管理信息系统的开发为核心题目，能够帮助我们将课堂理论知识转化为实际工程能力，理解信息系统从需求分析到落地实现的完整流程，为后续从事软件开发、系统设计相关工作奠定实践基础。

1.2 本文主要工作

本文围绕审批管理信息系统的设计与实现展开，是本次课程设计实验报告的核心阐述部分，主要完成的工作如下：

- 设计并实现了一套基于 Typescript 的审批流程系统，实现了基本的类型安全。
- 基于 Next.js 构建了直观且易于维护的前后端软件架构，具有清晰的文件结构拆分和基于 react 的前端组件化设计。同时提供一套基础的 SpringBoot 后端供学习交流使用。
- 构建了一套具有现代 web 风格的前端 UI 界面，设计风格统一。
- Code: Themaoqiu/Approval-MIS

2 相关技术综述

2.1 现代 web 开发与 Typescript

随着 Web 应用复杂度的不断提升，传统的 JavaScript 开发模式逐渐暴露出类型模糊、代码可维护性差、运行时错误率高等问题，难以满足大型项目的开发需求。现代 Web 开发更注重工程化、模块化与类型安全，TypeScript 是在这一背景下成为前端开发的主流技术之一。

TypeScript 由微软开发并开源，其核心特性是为 JavaScript 添加了静态类型系统，允许开发者在编码阶段定义变量、函数、对象等的类型，编译器会在编译过程中进行类型校验，提前发现类型不匹配等错误，大幅降低运行时异常的概率。更重要的是，Typescript 的类型安全特性与 vibe coding 高度适配，比其他语言更加清晰的类型定义具有强大的表达能力，强类型语义通过接口、枚举、函数类型注解等为代码赋予结构化的语义锚点，大幅降低大模型解析代码时的认知成本，避免纯 JavaScript 弱类型导致的理解偏差；

TypeScript 的类型推导特性可与 Vibe coding 的代码注释规范互补，强化大模型对代码上下文的感知能力，使其能结合推导类型和注释中的业务规则，更贴合实际业务逻辑理解代码, 显著提升大模型开发的效率与可靠性。

2.2 React

React 是由 Facebook 开源的前端 JavaScript 库，专注于构建用户界面。组件化是 React 的核心设计理念，它将复杂的页面拆分为独立、可复用的组件，每个组件负责自身的逻辑与视图渲染，这种模式符合高内聚、低耦合的软件设计原则。不论是类似卡片和按钮这种小组件，还是用户列表这种较大型的页面都可以单独拆分成一个组件在多处复用，提供了良好的社区生态并提高了开发效率。

虚拟 DOM（Virtual DOM）是 React 提升渲染性能的关键机制。传统 DOM 操作效率低下，而 React 会先在内存中构建虚拟 DOM 树，当数据发生变化时，通过对比新旧虚拟 DOM 的差异，仅将变化的部分更新到真实 DOM 中，大幅减少 DOM 操作次数。

2.3 Next.js 全栈框架

Next.js 是基于 React 的开源全栈框架，由 Vercel 公司开发，解决了传统 React 应用在服务端渲染、路由管理、打包部署等方面的痛点，是当前 React 生态中最热门的全栈开发方案之一。

在服务端渲染（SSR）与静态站点生成（SSG）方面，Next.js 提供了开箱即用的能力。传统 React 应用采用客户端渲染，存在首屏加载慢等问题，Next.js 的 SSR 模式可在服务端完成页面初始渲染，将完整的 HTML 页面发送给客户端，既提升了首屏加载速度，也满足了审批系统中可能存在的 SEO 需求（如审批公告页面）。简化了服务端渲染的配置流程，无需手动搭建复杂的服务端环境，即可快速实现前后端一体化开发。

App router 是 Next.js 的另一特点，它采用基于文件系统的路由机制，只需在 app 目录下创建文件 / 文件夹，即可自动生成对应的路由，无需手动配置路由规则，允许开发者在前端项目中直接编写后端接口逻辑，无需单独搭建后端服务即可完成简单的接口开发，这一特性让前后端交互更便捷，符合现代 web 开发推荐的方式。

2.4 SpringBoot 框架

SpringBoot 是由 Pivotal 团队开发的基于 Spring 框架的轻量级开发框架，核心目标是简化 Spring 应用的初始搭建与开发过程，是当前 Java 后端开发的主流框架。

相较于传统 Spring 框架，SpringBoot 降低了配置成本。它内置了自动配置机制，可根据项目依赖自动配置相关组件（如引入 spring-boot-starter-web 依赖后，自动配置 Tomcat 服务器、SpringMVC 等），无需手动编写大量 XML 配置文件。属于经典的 Java 后端 web 框架。

3 需求分析

3.1 系统需求

实现一个审批流程管理系统，旨在将传统的纸质审批流程数字化、自动化，提高企业工作效率。核心目标是将请假、报销等审批事项从纸质转为线上处理以及建立标准化的审批流程，确保每个申请都经过规范的审批环节。整个系统分为普通员工、审批人、系统管理员三种角色，不同角色只能访问和操作自己权限范围内的内容。项目可以进行实时跟踪，还可以完整保存审批历史，便于追溯和统计。

3.2 用户需求

用户需求分为三部分：一个是普通员工需求，一个是审批人需求，还有一个是管理员需求，分别对应不同的用户角色。其中，普通员工的需求包括：通过邮箱和密码登录系统，填写请假、报销等申请表单，查看自己发起的所有申请及状态以及拓展功能：在审批开始前可以撤回申请。而审批人的需求包括：查看所有需要自己处理的审批任务，处理审批，填写意见，以及查看自己已处理的审批记录。最后，管理员的需求是：添加、编辑、删除用户信息，维护企业组织架构，岗位管理，为用户分配相应角色，还可以查看系统所有审批数据，创建和管理审批流程。

3.3 技术需求

- 前端：Next.js 16(React) + TailwindCSS + shadcn/ui + Recharts + zustand + zod + Framer-Motion + Next-Themes
- 后端：
  - Next.js API Routes + Better-auth + Prisma + PostgreSQL
  - Spring Boot + Spring Security + Spring Data JPA
- 运行时环境：Bun 1.3.3

3.4 预期效果

前端采用现代 web 设计风格，页面切换和交互有平滑的过渡效果。功能包括侧边栏导航等，当操作成功、失败时有明确的提示信息。在交互过程中操作响应及时，审批状态实时更新，无需刷新页面，实时验证表单输入，提示错误信息，以及重要操作（删除、拒绝）有确认对话框。

实现以下完整流程：学生/员工提交申请 → 系统自动分配审批人 → 审批人收到待办通知 → 审批人处理 → 系统自动流转 → 下一审批人或结束 → 申请人收到结果通知 → 查看审批历史。

普通员工只能看到自己的申请，不能访问其他人的数据；审批人只能看到分配给自己的待办，不能访问其他待办；管理员可以查看所有数据，但只能进行管理操作。React 组件独立拆分，便于扩展；审批流程可配置，适应不同需求；支持添加新的申请类型和审批流程；关键代码有详细注释

4 系统设计

4.1 功能模块划分

使用 Next.js 的 App Router 后端路由来实现 API 路由，其中业务功能分为管理员功能，申请处理，处理审批，审批规则管理，登录，仪表板，统计分析和用户信息修改；和一个工具路由 auth。

登录鉴权：全站使用 Better-auth 管理身份与会话，负责登录、注册、社交登录、会话维护与角色分配等功能。所有认证相关的后端接口都映射在`app/api/auth`路径下，业务路由通过 `auth.api.getSession` 等接口读取会话并做鉴权与身份判断，权限策略在 `lib/permissions` 中定义并由 Better-auth 的 admin 插件接入。

工具路由：auth 作为认证工具路由集中暴露登录、登出、会话获取与社交回调等端点，由 Better-auth 的 Next.js 适配器统一处理，项目中其他业务路由应通过该工具路由提供的会话接口来做统一鉴权与权限判断。

管理员：`api/admin/` 目录下的路由负责系统级管理功能，包含管理员查看与管理申请、审批规则的增删改查、部门与岗位管理、表单模板的创建与维护，以及审批流程模板管理等。每个管理路由均先通过 `auth.api.getSession` 校验用户身份与角色，再使用 Prisma 对数据库执行相应的增删改查操作，只有具备相应权限的管理员或角色才能执行敏感操作。

提交申请：提交与管理申请的后端接口位于`api/applications/`目录下，用前必须登录。此模块处理新申请的提交，接收表单数据、流程定义并创建申请实例以启动流程、查询当前用户的申请列表并支持“我的申请”页面的查询、以及撤回申请等操作；撤回操作仅允许申请发起人在满足撤回条件时执行，路由负责修改申请状态并记录必要的日志或审计信息。

审批任务：位于`api/approvals/`目录下，实现审批任务的生命周期管理，包括查询当前用户的待办与已办任务、查看任务详情以及对单个任务执行同意/拒绝/转办等操作。所有审批动作在执行前会校验操作者为指定审批人或具备管理员权限，并在数据库事务中更新 `approvalTask` 与关联的 `application` 状态以保证一致性。

审批人：`app/api/approvers`目录下的路由实现审批人分配和管理的功能。能够根据申请类型与当前用户，选出适用的审批规则并返回候选审批人；按关键字或部门搜索符合条件的审批人；根据部门负责人、岗位、直接上级等分配类型生成审批人建议列表

仪表板：`/api/dashboard/stats`为当前登录用户提供日常工作时需要的统计数据，

包括用户自己的待办、已处理、总申请数、系统级申请总数、若用户为审批人或管理员则还返回待审批/已处理的审批任务数、以及最近 5 条申请的摘要和用户角色；实现上通过读取 session 确认用户身份后，用 Prisma 查询 application 和 approvalTask 表并根据角色分支返回不同的统计项。

统计分析：`app/api/statistics`为仪表盘和图表供给按年/月/日聚合的趋势数据、按类型/状态的分布、平均处理时长、总量与各类指标汇总。路由通过会话鉴权获取当前用户，并根据角色区分访问权限。个人统计对所有用户开放，部门统计通常需要主管/管理员权限，用 Prisma 从数据库做聚合查询并返回 JSON 格式的 trendData、typeData、statusData、avgProcessTime 等字段，前端据此绘制折线图/饼图/指标卡。

用户信息修改：`app/api/user`负责获取与更新用户个人资料、头像、联系方式、岗位/部门等个人信息；这些路由通过 Better-auth 提供的 session 鉴权验证访问者身份，使用 Prisma 对 user 表做 CRUD 操作，返回 JSON，GET 返回用户信息，PUT/PATCH 接受更新字段并返回更新后的记录。

4.2 数据库设计与实现

整个系统的数据库以 PostgreSQL 为存储引擎，使用 Prisma 作为 ORM 层，schema 位于 schema.prisma，由 Prisma Client 编译输出到 prisma 客户端，以关系模型为主并在有限场景下使用 JSON 字段来保留可变表单结构与流程配置，在设计上保证数据规范化，用户、部门、岗位、申请、任务等各自独立建表并通过外键关联。同时采用软删除与状态字段（delFlag/status）以便审计与回溯；认证会话和社交账户通过 Session 与 Account 表持久化，审批规则与流程模板有单独表以支持版本管理和规则匹配。审批流推进、申请撤回等关键操作在代码中通过数据库事务保证一致性，统计类数据支持离线聚合并写入 Statistics 表以加速报表查询；开发与部署流程依赖 Prisma 迁移目录进行 schema 变更管理。

4.3 技术架构

项目采用全栈 TypeScript 前后端同构架构，前后端通过 Next.js App Router 统一管理路由。前端基于 React 18 与 TypeScript，利用 shadcn/ui 组件库搭建一致的用户界面，Tailwind CSS 提供响应式样式与深色模式支持，framer-motion 为交互提供平滑的入场与过渡动画。后端采用 RESTful 接口设计，所有 API 路由位于 `app/api` 目录下，通过 Next.js 的服务端函数处理请求、执行业务逻辑并与数据库交互。认证与鉴权全链路使用 Better-auth 框架统一管理，包括基于邮箱/密码与 OAuth 的多种登录方式、会话持久化与基于角色的权限控制，后端 API 在请求处理前通过 `auth.api.getSession`获取会话并校验权限，确保跨 CSRF 与未授权请求的防护。数据存储采用 PostgreSQL 关系数据库结合 Prisma ORM，Schema 定义了 13 个核心表与它们的关系，业务代码通过 Prisma 提供的类型安全查询 API 操作数据。整个系统的构建、开发与部署依赖 Bun 作

```prisma
model User {
  id          String      @id
  username    String      @unique @db.VarChar(30)
  nickname    String?     @db.VarChar(30)
  email       String      @unique @db.VarChar(50)
  role        String      @default("user") // approver, admin
  phone       String?     @db.VarChar(11)
  password    String?     @db.VarChar(100)
  avatar      String?     @db.VarChar(100)
  sex         String?     @db.Char(1) // 0男1女2未知
  status      String      @default("0") @db.Char(1) // 0正常1停用
  delFlag     String      @default("0") @db.Char(1) // 0存在2删除
  deptId      Int? // 外键（数据库层面）
  dept        Department? @relation(fields: [deptId], references: [deptId])
// 关系字段（orm层面），fields是表内，references是表外的键
  loginIp     String?     @db.VarChar(128)
  loginDate   DateTime?
  createdBy   String?     @db.VarChar(64)
  createdAt   DateTime    @default(now())
  updatedBy   String?     @db.VarChar(64)
  updatedAt   DateTime    @updatedAt
  remark      String?     @db.VarChar(500)

  applications  Application[]
  approvalTasks ApprovalTask[]
  userPosts     UserPost[]

  emailVerified Boolean   @default(false)
  image         String?
  sessions      Session[]
  accounts      Account[]

  banned     Boolean?  @default(false)
  banReason  String?
  banExpires DateTime?

  @@map("user")
}
```

图 1: user 表定义，存放系统用户基本信息与登录相关字段，包括用户名、昵称、邮箱、角色、联系方式、头像、状态与删除标记等；同时关联部门（dept）、岗位（userPosts）、用户会话（sessions）、第三方账户（accounts）以及用户创建的申请与审批任务。该表既保存认证/会话需要的数据（password、emailVerified、sessions），也保存业务侧的用户属性（role、deptId、userPosts）。

```prisma
model Department {
  deptId    Int          @id @default(autoincrement())
  parentId  Int?
  parent    Department?  @relation("DeptTree", fields: [parentId],
references: [deptId])
  children  Department[] @relation("DeptTree")
  ancestors String?      @db.VarChar(50)
  name      String       @db.VarChar(30)
  orderNum  Int          @default(0)
  leader    String?      @db.VarChar(20)
  phone     String?      @db.VarChar(11)
  email     String?      @db.VarChar(50)
  status    String       @default("0") @db.Char(1) // 0正常1停用
  delFlag   String       @default("0") @db.Char(1) // 0存在2删除
  createdBy String?      @db.VarChar(64)
  createdAt DateTime     @default(now())
  updatedBy String?      @db.VarChar(64)
  updatedAt DateTime     @updatedAt

  users User[]
}
```

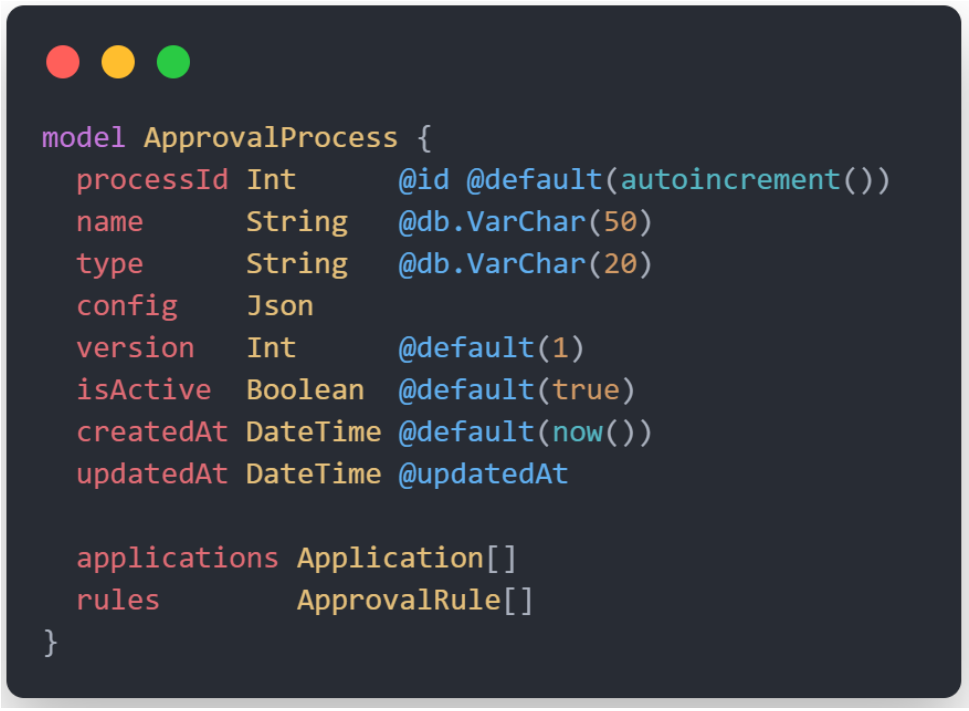
图 2: 组织架构表，表示部门树结构（parentId 与 children）、部门名称、排序、负责人（leader）、联系方式与状态字段。

```prisma
model Post {
  postId    Int      @id @default(autoincrement())
  code      String   @unique @db.VarChar(64)
  name      String   @db.VarChar(50)
  sort      Int      @default(0)
  status    String   @default("0") @db.Char(1) // 0正常1停用
  createdBy String?  @db.VarChar(64)
  createdAt DateTime @default(now())
  updatedBy String?  @db.VarChar(64)
  updatedAt DateTime @updatedAt
  remark    String?  @db.VarChar(500)

  userPosts UserPost[]
}
```

图 3: 岗位职位表，记录岗位编码、名称、排序与状态等元信息，跟 UserPost 关联，用于表征用户在组织内的岗位身份。

```
model Application {
  applyId     Int              @id @default(autoincrement())
  type        String           @db.VarChar(20)
  title       String           @db.VarChar(100)
  content     Json
  status      String           @default("pending")
  userId      String
  applicant   User             @relation(fields: [userId],
references: [id])
  currentStep Int              @default(0)
  processId   Int?
  process     ApprovalProcess? @relation(fields: [processId],
references: [processId])
  createdAt   DateTime         @default(now())
  updatedAt   DateTime         @updatedAt

  tasks ApprovalTask[]
}
```

图 4: 申请实例表，保存每条申请的类型、标题、表单内容（JSON）、当前流程状态与所属流程（processId）、当前步骤、发起人（userId）及创建/更新时间等字段。该表作为审批流程的主体记录，与 ApprovalTask、ApprovalProcess 关联，用于查询申请详情、展示申请历史与驱动流程流转。



```
model ApprovalProcess {
  processId Int      @id @default(autoincrement())
  name      String   @db.VarChar(50)
  type      String   @db.VarChar(20)
  config    Json
  version   Int      @default(1)
  isActive  Boolean  @default(true)
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  applications Application[]
  rules        ApprovalRule[]
}
```

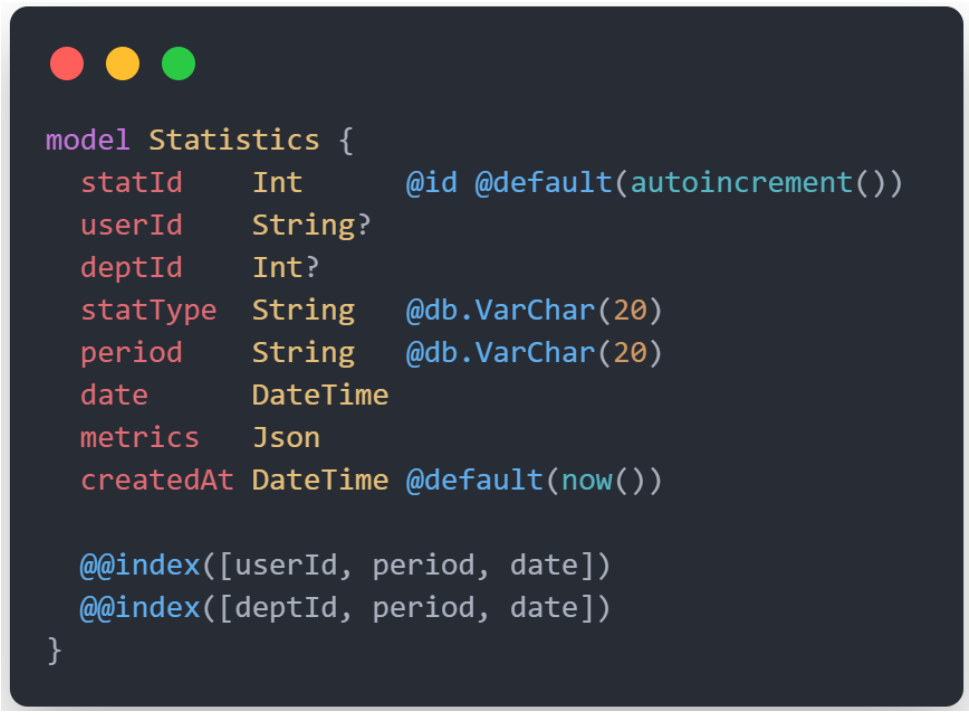
图 5: 审批流程模板表，包含流程名称、类型、流程配置（JSON）、版本与启用标志。保存流程定义与步骤配置，供创建申请时关联并在流程推进中读取 config 来决定下一步逻辑与审批人选择。

```prisma
model ApprovalRule {
  ruleId      Int             @id @default(autoincrement())
  processId   Int
  process     ApprovalProcess  @relation(fields: [processId],
references: [processId], onDelete: Cascade)
  name        String          @db.VarChar(100)
  description String?         @db.VarChar(500)

  // 适用条件
  applicantDeptId Int?         // 申请人所属部门(null表示不限)
  applicantPostId Int?         // 申请人所属岗位(null表示不限)

  // 审批人筛选条件
  approverDeptId  Int?         // 审批人必须属于的部门(null表示不限)
  approverPostId  Int?         // 审批人必须属于的岗位(null表示不限)
  specificUserIds String?      @db.Text // JSON数组,指定具体审批人ID列表

  // 审批模式
  approvalMode    String       @default("sequential") @db.VarChar(20)
// sequential(顺序), countersign(会签), or-sign(或签)

  priority        Int          @default(0)  // 优先级,数字越大优先级越高
  isActive        Boolean      @default(true)
  createdBy       String?      @db.VarChar(64)
  createdAt       DateTime     @default(now())
  updatedBy       String?      @db.VarChar(64)
  updatedAt       DateTime     @updatedAt

  @@index([processId, isActive])
  @@index([applicantDeptId, applicantPostId])
}
```

图 6: 审批规则表，描述某个审批流程下的规则项，包括适用条件（申请人部门/岗位）、审批人筛选条件（审批人部门/岗位或指定用户列表）、审批模式（顺序/会签/或签）、优先级与启用标识等，实现管理审批规则，查询并分配审批人功能。

```
model ApprovalTask {
  taskid      Int         @id @default(autoincrement())
  applyId     Int
  application Application @relation(fields: [applyId], references: [applyId])
  step        Int
  approverId  String
  approver    User        @relation(fields: [approverId], references: [id])
  status      String      @default("pending")
  comment     String?     @db.Text
  processedAt DateTime?
  createdAt   DateTime    @default(now())

  @@index([approverId, status])
}
```

图 7: 审批任务表,保存每个待办任务的引用(applyId)、步骤号、指定审批人(approverId)、任务状态、审批意见与处理时间等。审批动作对该表的更新会触发对应申请(Application)状态或下一任务的创建.



```
model Statistics {
  statId    Int      @id @default(autoincrement())
  userId    String?
  deptId    Int?
  statType  String   @db.VarChar(20)
  period    String   @db.VarChar(20)
  date      DateTime
  metrics   Json
  createdAt DateTime @default(now())

  @@index([userId, period, date])
  @@index([deptId, period, date])
}
```

图 8: 统计持久化表，实现统计分析功能，用于按用户或部门、按周期（period）存储已聚合的统计指标（metrics：JSON）与日期。

为高性能包管理器与运行时，TypeScript 编译器在开发阶段做静态检查；前端在 (auth) 与 (dashboard) 路由组中分离认证与主业务，利用 Next.js 布局嵌套为不同页面提供不同导航与样式上下文，业务组件通过自定义 Hooks 复用权限判断与响应式检测逻辑，全局状态通过 Zustand 等状态库管理。

```
approval-mis/
├── app/                                # Next.js App Router 主应用目录
│   ├── (auth)/                         # 认证相关路由（登录/注册页面）
│   ├── (dashboard)/                    # 仪表板路由组（需要登录）
│   ├── api/                            # 后端 API 路由（RESTful 接口）
│   ├── globals.css                     # 全局样式与 Tailwind CSS 配置
│   ├── layout.tsx                      # 根布局（ThemeProvider/主容器）
│   └── page.tsx                        # 首页
│
├── components/                         # React 组件库
│
├── hooks/                              # 自定义 React Hooks
│
├── lib/                                # 共享库与工具函数
│
├── prisma/                             # Prisma ORM 配置与迁移
│
├── types/                              # TypeScript 全局类型定义
```

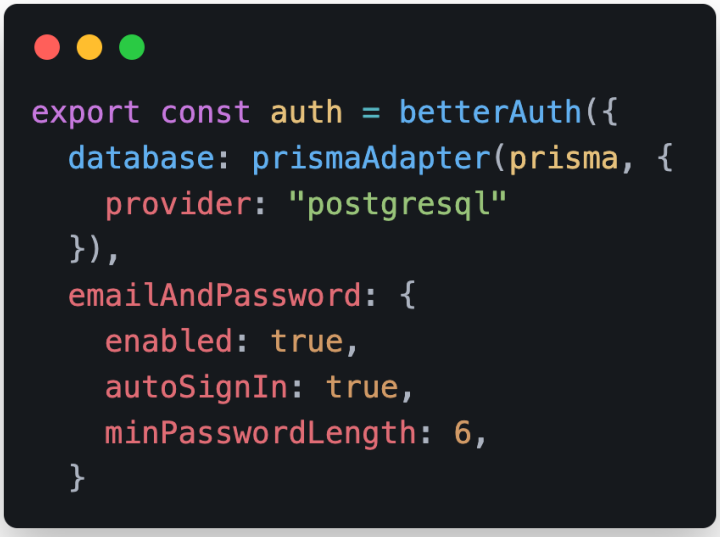
图 9: 项目文件树

5 系统核心模块设计与实现

5.1 后端鉴权系统

首先创建 Better—auth 配置文件 lib/auth.ts，使用 prismaAdapter 连接 PostgreSQL 数据库，Prisma ORM 负责 session、account、user 等认证表的 CRUD，Better-auth 会根据该配置自动创建必要的表结构。使用 emailAndPassword 字段启用基于邮箱与密码的登录方式，autoSignIn: true 表示用户注册后会自动登录，minPasswordLength: 6 设置密码最少 6 字符。同时对 session 行为，社交登录与用户字段映射进行配置。

再创建 permissions 文件定义系统的访问控制模型，使用 Better-auth 的 createAccessControl API 创建权限树，然后为三种角色分别定义允许的操作。statement 对象罗列了系统中所有可能的资源与操作，包括继承的默认权限以及业务相关的权限（application、approval、process ）。每个资源后跟一个操作数组，例如 application: [”create”,

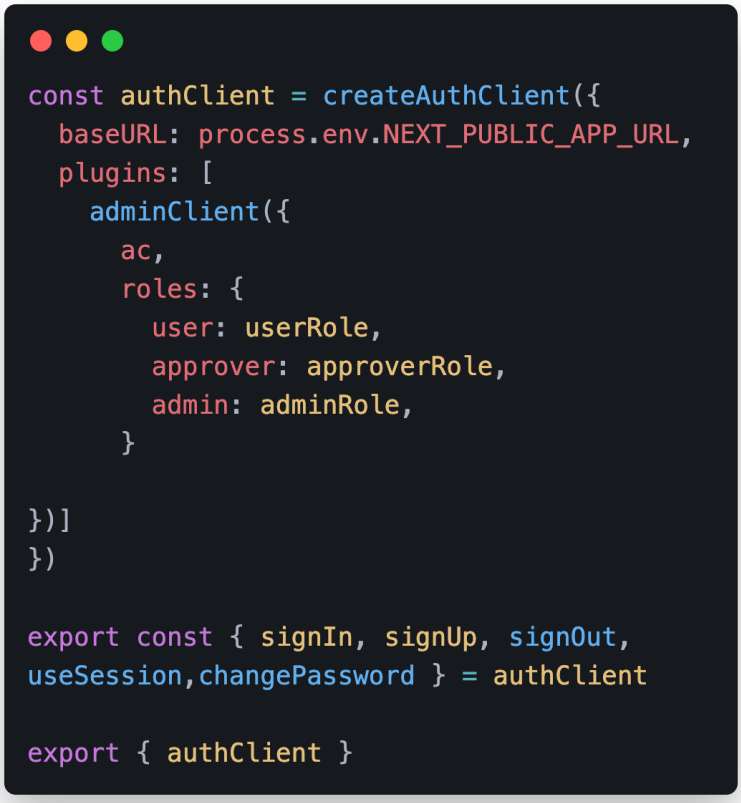


```
export const auth = betterAuth({
  database: prismaAdapter(prisma, {
    provider: "postgresql"
  }),
  emailAndPassword: {
    enabled: true,
    autoSignIn: true,
    minPasswordLength: 6,
  }
```

图 10: lib/auth.ts

```
export const statement = {
  ...defaultStatements,
  ...adminAc.statements,
  application: ["create", "view", "update", "delete", "withdraw"],
  approval: ["view", "approve", "reject", "comment"],
  process: ["create", "update", "delete", "view"],
} as const

export const ac = createAccessControl(statement)

export const userRole = ac.newRole({
  application: ["create", "view", "update", "delete", "withdraw"],
  approval: [],
  process: [],
})
```

图 11: lib/permissions.ts



```
const authClient = createAuthClient({
  baseURL: process.env.NEXT_PUBLIC_APP_URL,
  plugins: [
    adminClient({
      ac,
      roles: {
        user: userRole,
        approver: approverRole,
        admin: adminRole,
      }

})]
})

export const { signIn, signUp, signOut,
useSession,changePassword } = authClient

export { authClient }
```

图 12: auth-clients.ts

```
import { auth } from "@/lib/auth";
import prisma from "@/lib/prisma";

export async function GET(req: NextRequest) {
  const session = await auth.api.getSession({ headers: req.headers });
  if (!session?.user || session.user.role !== "admin") {
    return NextResponse.json({ error: "Forbidden" }, { status: 403 });
  }
```

图 13: route.ts

”view”, ”update”, ”delete”, ”withdraw”] 表示对申请资源可执行创建、查看、更新、删除、撤回五种操作。比如 userRole 是普通用户角色，可创建/查看/更新/删除/撤回自己的申请，不允许访问审批与流程管理。

这个文件可以在 lib/auth.ts 中传给 admin 插件；业务代码可通过 userHasPermission() 查询当前用户是否拥有某个权限。

auth-clients.ts 文件创建客户端相关功能函数，导出对应名称的函数之后可直接在客户端通过`authClient.signUp.email`和`authClient.signIn.email`实现注册和登录

定义之后即可在 route 中导入认证函数，在服务器端使用`auth.api.getSession`函数通过 api serve 方式进行用户角色校验。

特别地，Better-auth 框架提供了现成的完善的管理员的操作函数。所以对于管理员操作的实现比其他角色更加方便快捷，只需在配置文件中添加一个 admin 插件即可。在本项目中，用户管理 api 使用`auth.api.createUser`函数新建用户，Better-auth 提供鉴权和创建一整个流程的服务；查看所有用户信息使用`auth.api.listUsers`, 更改用户密码使用`auth.api.setUserPassword`等函数。开发时无需任何手动权限检查和数据库操作，避免了在手动实现过程中的权限检查错误等可能出现漏洞等问题，也避免了冗余代码的产生，对于 vibe coding 也有更高的兼容性，极大简化了开发流程。

5.2 Springboot 后端实现示例

```java
@Data
public class DepartmentRequest {
    private Integer parentId;
    private String name;
    private Integer orderNum;
    private String leader;
    private String phone;
    private String email;
    private String status;
}
```

图 14: 部门表对应的 enum 数据定义, 用到的字段和 api route 后端保持一致

```java
@PostMapping
public BaseResponse createDepartment(@RequestBody DepartmentRequest request) {
  // 假设从认证上下文中获取用户ID，这里暂时使用固定位
    String userId = "admin";
    return BaseResponse. success (departmentService. createDepartment(request, userId));
    }
@PutMapping("/{id}")
public BaseResponse updateDepartment (@PathVariable("id") Integer deptId, @RequestBody
DepartmentRequest request) {
  // 假设从认证上下文中获取用户ID，这里暂时使用固定位
  String userId = "admin";
  return BaseResponse.success(departmentService.updateDepartment(deptId, request, userId));
}
```

图 15: 部门管理功能中的 controller 一部分代码，用于接收请求转 service 层进行实际业务处理，暴露了两个 HTTP 接口：POST /departments 用于创建部门，PUT /departments/id 用于更新部门。两者都调用 departmentService，并把返回结果包在 BaseResponse.success(...) 里返回。Service 层负责把请求 DTO 的字段复制到实体上，设置审计字段（createdBy / updatedBy），并通过 departmentRepository.save(...) 持久化到数据库。

```java
@Service
public class DepartmentService {
    public Department createDepartment (DepartmentRequest request, String userId) {
        Department department = new Department ();
        BeanUtils. copyProperties (request, department);
        department.setCreatedBy(userId);
        return departmentRepository.save(department);
    }

    public Department updateDepartment(Integer deptId, DepartmentRequest request, String userId)
    {
        Department department = departmentRepository.findById(deptId)
                .orElseThrow(() -> new RuntimeException("部门不存在"));
        BeanUtils.copyProperties(request, department);
        department.setUpdatedBy(userId);
            return departmentRepository.save(department);
    }
}
```

图 16: 部门管理中的 service 层中的部分代码，进行实际的业务处理，如图所示是新建部门和更新部门信息的逻辑，通过 BeanUtils.copyProperties 将请求对象（DepartmentRequest）的属性复制到实体对象，然后设置创建者或更新者字段，最后调用 departmentRepository.save(...) 持久化到数据库。

5.3 前端交互

整个前端界面由一系列细粒度的 React 组件组成。应用程序基于 shadcn/ui 和 Radix UI 的基础组件库构建，这些基础组件包括 Button、Input、Dialog、Table、Card 等，允许通过 variant 和 size 属性灵活配置样式和行为。项目的样式使用 Tailwind CSS 框架，使用原子化 CSS 工具类进行样式构建，这种方法避免了传统 CSS 中命名冲突和样式污染的问题，同时显著提高了开发效率。Tailwind CSS 的工具类可以直接在 TSX 中使用，使用例如`className="flex items-center gap-2 rounded-md"`这样的组合方式就能够快速构建出复杂的布局和样式效果。

Sidebar 组件是管理系统中最常用的组件之一，手工构建费时费力且难以做出美观的 UI 和动画。在发现一开始手工写的 Sidebar 组件难以实现整体折叠和方便分组的功能之后，就将其重构成使用 Shadcn/ui 中的侧边栏组件。这个系统由 Sidebar、SidebarContent、SidebarHeader 等多个协作组件组成。侧边栏支持可折叠功能，通过`collapsible="icon"`属性能够让侧边栏在展开和折叠状态间平滑切换，当折叠时仅显示图标，展开时显示完整的菜单文本和标签。菜单项的导航使用 Next.js 的 Link 组件包装 SidebarMenuButton 即可实现路由的切换。根据用户的权限等级，系统会动态渲染不同的菜单项：普通用户看到工作区相关的菜单，审批员可以看到审批任务菜单，管理员则可以访问所有管理功能的菜单项。

深色模式在长时间看白色背景屏幕感觉刺眼之后选择加入，提供相对好的开发体验，深色模式的实现通过 next-themes 库与 Tailwind CSS 的 dark: globals.css 中为亮色和深色两套主题定义了 CSS 变量。亮色模式使用–background: oklch(1 0 0) 这样的高亮背景和–foreground: oklch(0.145 0 0) 这样的深色文字，确保高对比度的可读性；深色模式则反转这些值，使用深色背景和浅色文字。Sidebar、Card、Dialog 等所有主要组件都在其样式定义中使用了 dark: 前缀的 Tailwind 类来适配深色模式，即在深色模式时会使用 dark 前缀后的样式。当用户在 Settings 中选择自动或手动切换主题时，next-themes 会自动保存用户的偏好到 localStorage，并在下次访问时自动应用。

此外，项目中大量表格、对话框和删除确认组件都通过 motion.div 包装实现了进出场动画，只需把原来的 html 标签前加一个 motion. 即可变成可使用动态效果的元素。比如表格行使用 rowVariants 定义的交错动画效果，删除确认对话框在挂载时从 0.95 缩放到 1 的弹跳效果。

6 成果展示

6.1 页面效果

图 17: 主页仪表盘，提供日常工作信息显示，具有鼠标移动上去的浮动效果和多彩卡片设计，侧边栏提供折叠成 icon 功能并配有美观展开动画

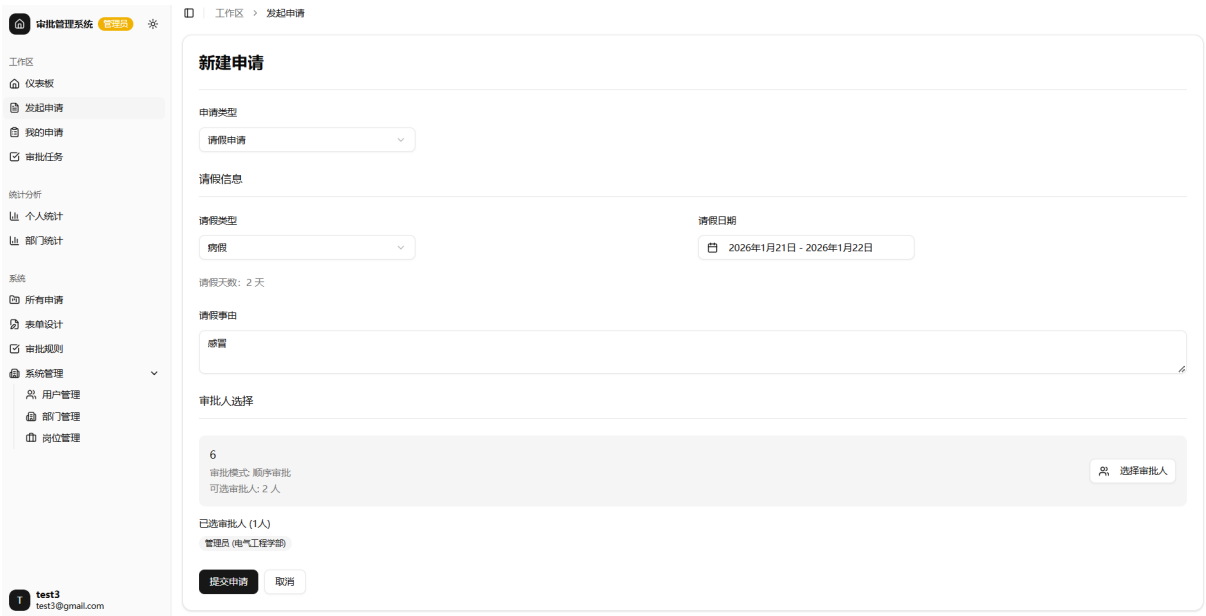


图 18: 新建申请页面，支持自动筛选审批人

图 19: 审批员进行审批任务界面

图 20: 个人统计分析图表页面

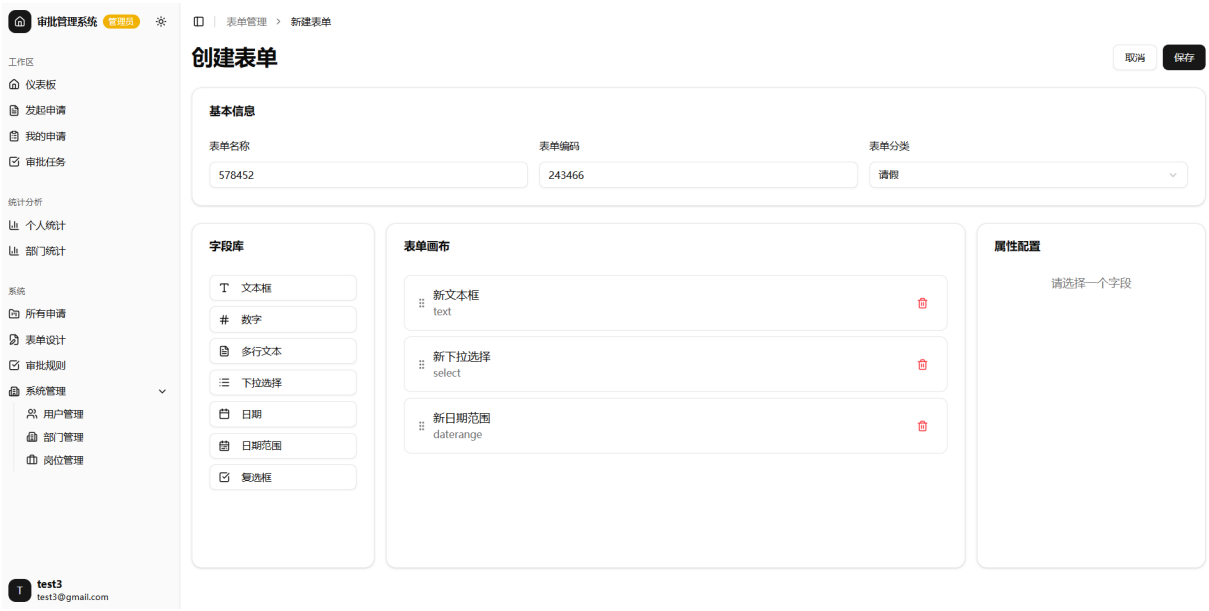


图 21: 新建审批表单原型界面

7 难点分析与解决方案

7.1 前端页面实时刷新问题

在搭建好管理员管理用户/部门等列表界面并提供对应的修改功能之后，发现在修改信息之后表格内容不能自动实时刷新，只能在手动点击浏览器刷新按钮之后刷新信息。排查问题后确定是在 AI 写相关后端代码时没有提前人工定义好接口，导致 AI 没有考虑到应该在修改信息之后实时访问后端 API 刷新页面内容。

同时 AI 修改的时候没有按照显示效果更好和性能更高的方法实现，他通过使用一个 onRefresh 变量记录组件的状态，一旦组件状态改变就把整个表格组件在页面上重新挂载。这样会导致一旦修改内容并保存，表格就会闪烁一下即重新挂载，视觉效果不佳且 AI 实现这个功能的逻辑令人匪夷所思，非常难懂。在缺乏预先设计和人工 review 的情况下 vibe coding 会经常出现此类问题。

在 review 代码之后制定了在更改信息弹窗添加 onSuccess 属性，每当修改成功之后就复用查询表格信息的 fetch 函数进行表格刷新的代码逻辑。然后在 AI 快速修改大量代码的辅助下将所有涉及修改信息时表格内容刷新的部分进行修改，实现了所有表格页面统一且美观的刷新视觉效果，并提升了前端性能。

7.2 基于 Better-auth 框架的前后端通信问题

Better-auth 在本项目中第一次采用，是本次重点学习使用登录鉴权的框架。在初期设计数据库的时候，由于需求文档中已经提供了对应的数据库字段就没有再自己进行其他设计，其中 user 表的主键字段叫做 userId，而 Better-auth 只能识别 id 字段作为主键，在一开始不想修改 schema 的情况下无法正常运行登录功能。在阅读文档之后了解到框

架可以自动生成需要的数据表,无需手动编写,使用`bunx @better-auth/cli generate`就可以实现自动构建要用的字段，之后可以手动继续添加其他业务要的字段，还可以在默认传递邮箱密码用户名的情况下自定义新建用户时传递给后端的字段。随后在文档逐步教程下成功构建登录鉴权系统，在更深入阅读文档之后发现可以通过 Better-auth 的配置文件对默认识别字段进行修改，届时已经构建成功就没有继续修改。

在新增管理员添加用户功能时使用 vibe coding，由于使用自定义的注册字段时虽然能够正常运行，但是编辑器会报错，影响了 AI 的理解。所以在用 AI 写代码时会出现使用的字段名称和实际数据库不符合的情况，而且并不会被 ide 识别，增加了 debug 的难度。所以在使用这类轮子时的最好方式依然是自己手动构建。

7.3 服务端客户端登录鉴权方法的区别

Better-auth 提供了 api 和 client 两种访问方式，分别给厚后端和前端组件使用。如判断用户角色，使用管理员插件时在 route 中就要使用服务端的 api 方法；登录注册的代码写在 tsx 组件中时就要使用 client 方法。在不熟悉框架时会搞错，在使用前应仔细阅读文档。

8 总结

本课设项目采用现代 web 开发框架构建了审批管理系统，完全实现基本功能，实现了部分高级功能。不仅提供了完整的 Next.js 架构，也实现了简单的 Springboot 后端，所有人都参与了一部分的架构设计和代码实现工作，对 web 开发都建立了基础概念。

在动手开发前，我们原本以为只要把技术栈搭好、功能实现出来加上 AI 就非常简单，但实际操作中才发现，真正的难点不在于敲代码本身，而在于如何让各个技术模块协同工作，比如 Better-auth 鉴权框架和 Prisma 数据库操作的衔接，前端组件状态与后端数据的实时同步，这些都需要一步步调试、踩坑才能理顺。尤其是在解决前端页面实时刷新的问题时，一开始照搬 AI 生成的代码导致表格闪烁，后来自己梳理逻辑，通过复用查询函数实现平滑刷新，才真正理解了前端状态管理的核心——不是简单重渲染，而是精准更新数据。

在技术应用层面，这次开发跳出了只会用框架 API 的浅层认知。比如 TypeScript 的类型安全，一开始觉得写类型注解是额外的工作量，但在多人协作和后期维护时，类型检查能够规避了很多数据格式不匹配的低级错误；Next.js 的 App Router 路由机制体现了前后端一体化开发的优势，不用再像传统开发那样单独搭建后端服务，大大提升了开发效率。也存在许多不足，比如对数据库设计的理解还停留在“能建表就行”的阶段，没有充分考虑数据关联和性能优化，后续如果要优化系统，首先要重构部分表结构，减少冗余字段，提升查询效率。

这次课程设计也让我们意识到，一个实用的系统，技术只是工具，核心还是要贴合实际业务场景。比如审批流程的设计，不能只追求功能完整，还要考虑用户的操作习惯，

像侧边栏的折叠功能、深色模式的适配，都是从实际使用体验出发的优化。未来如果继续完善这个系统会重点补充两个方向：一是增加移动端适配，实现响应式布局，现在的界面主要针对电脑端设计，手机端操作不够便捷；二是优化审批流程的灵活性，目前的流程模板还是固定的，后续可以设计可视化的流程配置界面，让管理员能自定义审批步骤和审批人规则。总的来说，这次开发不仅巩固了技术知识，更学会了从用户角度思考问题，这比单纯掌握某个框架的用法更有价值。