三峡大学

计 算 机 与 信 息 学 院

《软件工程》课 程 作 业

2024 年 秋 季 学 期

课程类型：	专业核心课
学　　号：	202210120130
姓　　名：	涂诗文
专　　业：	计算机科学与技术
授课教师：	林瑞

完成日期：2024 年　12 月　xx 日

目录

Web 网站开发-任务管理系统

第一章 实验任务

1.1 实验背景

实验背景及应用场景

任务管理系统是一种常见的工具，广泛应用于个人工作管理、团队协作和项目规划等场景。它能够帮助用户高效地记录和管理待办事项、跟踪任务进度、协调工作优先级，从而提高工作效率并实现目标管理。

现实中的任务管理系统，如 **Todoist**、**Trello** 和 **Microsoft To Do** 等，提供了强大的功能支持，包括任务的创建与分类、优先级设置、完成状态标记、跨平台同步以及多人协作功能。这些系统为用户提供了极大的便利，尤其是在多任务处理和协作需求较高的环境中。

在本实验中，任务管理系统将实现以下核心功能：

1. **用户注册与登录**：为每位用户提供独立账户，确保数据的隐私与安全。
2. **任务的增删查改**：用户可以方便地添加、查看、编辑和删除任务。
3. **任务状态管理**：支持对任务标记完成状态（未完成、已完成），方便用户直观地跟踪任务进度。
4. **可选功能**：实现任务按优先级排序，以便用户合理安排时间和资源。

实验目的与意义

本实验旨在通过开发一个简单的任务管理系统，帮助学生综合应用前端开发、后端开发、数据库管理和系统部署相关技术，达到以下具体目标：

1. **掌握前端开发技能**：通过使用 React.js 设计用户界面，学习组件化开发的思想、状态管理以及与后端 API 的交互。
2. **理解后端开发流程**：通过使用 Django 框架，掌握构建 REST API 的方法、逻辑处理及数据库操作。
3. **熟悉数据库设计与操作**：通过 MySQL 数据库的设计与使用，理解数据表

的结构设计、关系管理以及数据的增删改查操作。

4. **体验完整的开发与部署流程：**学习将开发完成的系统在 Ubuntu 环境下使用 Gunicorn 和 Nginx 进行生产部署，从而了解开发到上线的完整过程。
5. **提升问题解决能力：**通过实验中遇到的开发与部署问题，培养分析问题和解决问题的能力，积累团队开发和独立调试的实践经验。

通过本次实验，学生不仅能够巩固和应用已有知识，还可以学习到企业开发中常用的技术栈和开发流程，为将来从事软件开发相关工作打下坚实的基础。

1.2 实验要求

1.2.1 功能需求

1. **用户注册与登录**
    - 系统需支持用户通过注册功能创建账户，并确保用户的密码信息加密存储。
    - 用户可通过登录功能进入系统，查看和管理其个人任务数据。
    - 使用会话管理或 JWT（JSON Web Token）进行身份认证，确保用户数据的隐私性与安全性。
2. **任务的增删查改**
    - **添加任务：**用户能够新增任务，任务需包含基本信息，如标题、描述、截止日期等字段。
    - **查看任务：**用户可浏览其所有任务，并支持按照时间或其他条件筛选任务。
    - **编辑任务：**用户可以修改任务的标题、描述、截止日期等内容。
    - **删除任务：**用户可删除指定任务，任务从数据库中永久移除。
3. **显示任务状态（未完成/已完成）**
    - 系统需支持用户对任务状态进行标记，未完成的任务显示为“未完成”，完成的任务显示为“已完成”。
    - 前端界面需提供任务状态的直观展示，以便用户快速了解任务的完成情况。

4. **可选功能：任务按优先级排序**
    - 每个任务可设置优先级（如高、中、低）。
    - 系统支持根据优先级对任务进行排序，优先显示高优先级的任务，帮助用户合理规划时间和资源。

1.2.2 技术栈与运行环境要求

前端

- **技术：**React.js
    - 用于构建用户界面，提供动态交互与单页面应用（SPA）的流畅体验。
    - 利用 React Router 实现多页面导航，如登录页面和任务管理页面。
    - 使用 Axios 或 Fetch API 与后端交互，实现前后端数据传递。
- **运行环境：**Node.js 提供本地开发和调试环境。

后端

- **技术：**Django
    - 使用 Django 框架实现后端逻辑，包括用户注册与登录、任务管理的 REST API 接口。
    - 集成 Django REST Framework (DRF)，简化 API 开发。
    - 实现 JWT 或会话认证，确保用户身份的验证和安全。
- **运行环境：**Python 3.x 及相关依赖库（通过 pip 安装）。

数据库

- **技术：**MySQL
    - 用于存储用户和任务数据，设计表结构时需支持用户与任务的关联关系。
    - 数据表主要包括以下内容：
        - 用户表：存储用户的注册信息（如用户名、密码、邮箱等）。
        - 任务表：存储任务的标题、描述、截止日期、优先级、状态以及所属用户。

部署

- **操作系统**：Ubuntu Linux
    - 使用 Ubuntu 作为服务器环境，确保系统稳定运行。
- **部署工具**：
    - **Gunicorn**：作为 WSGI HTTP 服务器运行 Django 项目。
    - **Nginx**：作为反向代理服务器处理前端静态资源与后端 API 请求。
- **版本管理**：Git 用于代码版本控制，Gitee 作为代码托管平台。

整体要求

- 系统应支持跨平台访问，通过浏览器在不同设备（如台式机、笔记本电脑）上使用。
- 实现功能完整性和界面友好性，同时保证后端 API 的安全性与可靠性。
- 部署后测试系统的稳定性，确保能正确响应用户请求。

第二章 实验平台

2.1 开发工具

本实验任务管理系统的开发和部署过程涉及多种工具和技术栈，为了保证开发效率和系统的可靠性，以下是各类工具的详细说明：

前端开发工具

1. **React.js**
    - **作用：**

      React.js 是一个用于构建用户界面的 JavaScript 库，支持组件化开发与高效的状态管理。通过它，可以轻松开发动态交互的单页面应用（SPA）。
    - **特点：**
        - 组件化设计，便于代码复用和维护。
        - 虚拟 DOM 提高了渲染性能。
        - 丰富的社区资源和第三方库支持。
    - **用途：**

      用于实现用户界面，包括用户注册与登录页面、任务管理界面（任务列表、编辑、状态显示等）。
2. **Node.js**
    - **作用：**

      Node.js 提供了运行 JavaScript 的服务器端环境，主要用于运行和管理前端开发环境。
    - **用途：**
        - 用于启动 React.js 开发服务器（通过 npm 或 yarn）。
        - 用于安装和管理前端依赖，如 Axios、React Router 等工具包。

后端开发工具

3. **Django**
    - **作用：**

      Django 是一个高效的 Python Web 框架，能够快速构建功能全面的后端应用程序。通过 Django REST Framework（DRF），可以方便地实现 RESTful API 的开发。
    - **特点：**
        - 内置 ORM（对象关系映射），简化了数据库操作。
        - 提供了丰富的内置功能，如用户认证、表单验证等，减少开发工作量。
        - 强大的社区支持和扩展能力。
    - **用途：**
        - 开发 RESTful API，用于处理前端请求。
        - 实现用户注册、登录认证（使用 JWT 或会话管理）。
        - 提供任务数据的增删查改功能。

数据库管理工具

4. **MySQL**
    - **作用：**

      MySQL 是一种流行的关系型数据库管理系统，用于存储用户和任务数据。
    - **特点：**
        - 支持复杂的查询与事务操作。
        - 与 Django 的 ORM 框架兼容性良好。
        - 高效、可靠，适合中小型项目的需求。
    - **用途：**
        - 存储用户信息表（用户名、密码等）。
        - 存储任务信息表（标题、描述、状态、优先级等）。
        - 支持查询任务列表、更新任务状态等操作。

部署工具

5. **Gunicorn**
    - **作用：**

      Gunicorn 是一种 Python WSGI HTTP 服务器，适用于生产环境，能够高效处理来自客户端的请求。
    - **用途：**
        - 作为 Django 项目的运行环境，将其转换为可供生产使用的 WSGI 应用。
6. **Nginx**
    - **作用：**

      Nginx 是一款高性能的 HTTP 和反向代理服务器。
    - **用途：**
        - 处理用户的 HTTP 请求，将静态资源（如前端构建文件）直接返回给客户端。
        - 将动态请求转发到 Gunicorn 提供的后端服务。
7. **Ubuntu 服务器**
    - **作用：**

      Ubuntu 是一个稳定的 Linux 操作系统，用于项目的开发和部署。
    - **用途：**
        - 提供服务器环境，安装并运行 MySQL、Gunicorn 和 Nginx 等服务。
        - 部署完成的任务管理系统并进行测试。

版本控制工具

8. **Git**
    - **作用：**

      Git 是一个分布式版本控制系统，便于团队协作和代码管理。
    - **用途：**
        - 跟踪代码变更，支持回滚和分支开发。

- 记录开发过程中的每一次提交，保持代码版本可追溯。

9. **Gitee**
   - **作用：**
     Gitee 是一个代码托管平台，支持 Git 仓库，便于团队成员共享代码和协作开发。
   - **用途：**
     - 托管项目代码，实现代码同步和备份。
     - 用于团队项目的代码评审和任务分配。

通过以上开发工具的组合应用，本实验将完整地覆盖从开发到部署的整个流程，为任务管理系统的顺利完成提供强有力的支持。

2.2 硬件环境

为了顺利完成任务管理系统的开发和部署，需要确保开发设备和部署环境满足相关要求，以下是详细的硬件环境描述：

2.2.1 开发设备配置

开发阶段主要依赖于个人计算机（PC）或笔记本电脑，推荐的最低硬件配置如下：

1. **中央处理器（CPU）**
   - **推荐配置：**Intel Core i5 第 8 代及以上 / AMD Ryzen 5 系列及以上
   - **作用：**提供多核性能支持，确保编译代码、运行开发服务器和数据库时性能流畅。
2. **内存（RAM）**
   - **推荐配置：**8GB 或更高（建议 16GB）
   - **作用：**足够的内存容量可同时运行多个工具，如 React 开发服务器、Django 后端服务、数据库等。

3. **硬盘存储**
    - **推荐配置**：256GB SSD 或更高
    - **作用**：SSD 提供高速读写性能，能够缩短项目依赖安装和开发服务器启动时间。
4. **操作系统**
    - **推荐配置**：Windows 10/11、macOS Ventura 或 Ubuntu 20.04/22.04
    - **作用**：提供开发工具（如 Node.js、Django、MySQL）兼容的环境，Ubuntu 对 Linux 开发者尤为友好。
5. **显示器与网络**
    - **推荐分辨率**：1920×1080 或更高分辨率
    - **网络连接**：稳定的宽带网络（至少 10 Mbps 下载速度），用于依赖安装和代码托管。

2.2.2 部署环境配置

部署阶段需要在服务器端运行完整的任务管理系统，推荐使用一台运行 Ubuntu 操作系统的物理服务器或云服务器（如阿里云、腾讯云等）。以下是部署环境的配置要求：

1. **中央处理器（CPU）**
    - **推荐配置**：双核或以上（Intel Xeon 或 AMD EPYC 系列）
    - **作用**：多线程能力保证后端服务（Gunicorn）高效处理客户端请求。
2. **内存（RAM）**
    - **推荐配置**：2GB 或更高（建议 4GB）
    - **作用**：支持 MySQL 数据库运行及 Gunicorn 服务器处理多用户并发访问。
3. **硬盘存储**
    - **推荐配置**：40GB 或更高（建议 SSD）
    - **作用**：存储系统日志、数据库数据及前后端应用文件，同时支持快速 I/O 操作。

4. **操作系统**
    - **推荐配置**：Ubuntu 20.04 LTS 或 Ubuntu 22.04 LTS
    - **作用**：作为后端服务的稳定运行环境，LTS 版本长期支持且兼容主流部署工具。
5. **网络配置**
    - **带宽要求**：最少 1Mbps 上行带宽，推荐 10Mbps 或更高（对高并发需求更高）。
    - **公网 IP**：确保服务器能被外网访问，支持绑定域名。
6. **附加配置**
    - **防火墙配置**：开放必要端口，如 HTTP (80)、HTTPS (443)、MySQL (3306)、Gunicorn 应用端口。
    - **安全性**：启用 SSH 访问并设置强密码或使用密钥登录。

通过上述开发设备和部署环境的配置，可以确保任务管理系统在开发和生产环境下高效运行，为系统功能的实现与用户的使用体验提供可靠的硬件支持。

第三章 实验原理

3.1 项目架构

3.1.1 前后端分离的架构设计说明

本实验任务管理系统采用**前后端分离**架构设计，前端负责实现用户界面和交互，后端负责处理业务逻辑和数据管理，两者通过标准化的 RESTful API 进行通信。此设计具有以下优势：

1. **模块化开发**
    - 前端与后端分离，便于各自独立开发、测试和维护，降低开发复杂度。
2. **技术栈灵活性**
    - 前端和后端可以选择最合适的技术框架（如 React.js 和 Django），相互独立而不受约束。
3. **性能优化**
    - 前端可以缓存部分静态资源（如 HTML、CSS 和 JavaScript），减少对后端的依赖，提高页面加载速度。
4. **扩展性和复用性**
    - 后端 API 可被多个客户端（如 Web 应用、移动端应用）复用，便于扩展。

用户请求的基本流程

整个系统的架构包括以下主要组件及其交互流程：

1. **前端部分（React.js）**
    - 前端通过 React.js 实现单页面应用（SPA），用户通过浏览器访问前端页面进行操作。
    - 用户请求的动作（如登录、查看任务列表、编辑任务等）通过 HTTP 请求的方式提交至后端 REST API。

2. **后端部分（Django + Django REST Framework）**
    - 后端接收来自前端的请求，解析请求中的数据并执行相应的逻辑处理。
    - 后端通过 Django ORM（对象关系映射）访问 MySQL 数据库，完成数据的增删改查操作。
    - 处理完成后，后端将结果以 JSON 格式返回给前端。
3. **数据库部分（MySQL）**
    - 数据库存储用户和任务信息。
    - 通过后端的 ORM 层实现对表结构和数据的操作，减少了直接编写 SQL 的复杂性。
4. **API 通信方式**
    - 前端通过 HTTP 请求（如 GET、POST、PUT、DELETE）与后端 REST API 通信，使用 JSON 作为数据传输格式。
    - 示例 API 调用：
        - 登录：POST /api/login
        - 查看任务列表：GET /api/tasks
        - 更新任务：PUT /api/tasks/:id
        - 删除任务：DELETE /api/tasks/:id

用户请求流程图解

以下是一个典型的用户请求流程：

1. **用户操作**
    - 用户通过前端界面触发操作，例如点击按钮查看任务列表。
2. **前端发送请求**
    - 前端使用 Axios 或 Fetch API 发送 HTTP 请求至后端 API，例如 GET /api/tasks 获取任务列表。
3. **后端处理请求**
    - Django 后端接收到请求，解析请求 URL 和参数，调用相应的视图函数。

- 视图函数通过 Django ORM 与数据库交互，例如从数据库读取用户任务数据。

4. **后端返回响应**
   - 处理完成后，后端返回一个 JSON 格式的响应数据，例如任务列表：

```
[
  {
    "id": 1,
    "title": "完成实验报告",
    "status": "未完成",
    "priority": "高"
  },
  {
    "id": 2,
    "title": "复习软件工程",
    "status": "已完成",
    "priority": "中"
  }
]
```

5. **前端更新界面**
   - 前端接收响应数据，更新页面内容，例如动态渲染任务列表并显示状态和优先级。

架构示意图

1. 用户通过浏览器访问任务管理系统。
2. 前端（React.js）发送 HTTP 请求至后端 API（Django）。
3. 后端通过 ORM 操作 MySQL 数据库，完成数据处理后返回 JSON 数据。
4. 前端解析返回的数据并更新界面。

架构逻辑如下：

[用户浏览器]

↕ HTTP 请求/响应

[前端：React.js]

↕ RESTful API

[后端：Django + DRF]

↕ ORM 查询

[数据库：MySQL]

```
[用户浏览器]
    ↕ HTTP 请求/响应
[前端：React.js]
    ↕ RESTful API
[后端：Django + DRF]
    ↕ ORM 查询
[数据库：MySQL]
```

这种架构分工明确，层次清晰，为功能扩展和系统优化提供了良好的基础。

3.2 核心技术说明

本实验中任务管理系统的开发和部署涉及多个核心技术组件，每个组件都在系统中发挥了重要作用，以下是其详细说明：

---

1. React.js

功能：

- React.js 是一种用于构建用户界面的 JavaScript 库，支持组件化开发与高效的状态管理。

核心特点：

- **组件化设计：**开发者可以将页面拆分为多个独立的组件（如任务列表组件、任务表单组件），便于代码复用和管理。
- **单向数据流：**通过 Props 和 State 管理组件间的数据传递，确保数据流动清晰有序。

- **虚拟 DOM**：使用虚拟 DOM 机制高效更新页面，提升性能。
- **丰富的第三方库支持**：支持 React Router（页面路由）、Axios（HTTP 请求）等工具。

用途：

- 实现用户注册与登录页面，任务的增删查改界面，任务状态和优先级的展示功能。
- 通过 Axios 调用后端 RESTful API，实现与后端的通信。

---

2. Django

功能：

- Django 是一个高效、模块化的 Python Web 框架，主要用于构建后端逻辑和 RESTful API。

核心特点：

- **快速开发**：提供丰富的内置功能，如用户认证、表单验证、会话管理等，简化开发流程。
- **Django ORM（对象关系映射）**：支持用 Python 代码操作数据库，无需直接编写 SQL 查询。
- **Django REST Framework（DRF）**：扩展 Django 的功能，轻松开发 RESTful API，支持序列化、权限管理等功能。

用途：

- 处理前端请求，完成任务数据的增删查改逻辑。
- 通过 Django ORM 与 MySQL 数据库交互，保存用户数据和任务信息。
- 使用 DRF 构建 RESTful API，并进行请求验证和错误处理。

---

3. MySQL

功能：

- MySQL 是一个广泛使用的关系型数据库管理系统，用于存储和管理数据。

核心特点：

- **关系型数据模型**：支持表间关系和复杂的 SQL 查询，方便管理用户和任

务信息。

- **高效性与可靠性**：能够处理大规模数据，同时保证事务的完整性。
- **兼容性**：与 Django ORM 完全兼容，简化了数据库操作。

用途：

- 存储用户信息表（如用户名、加密密码等）。
- 存储任务信息表（如任务标题、描述、优先级、状态等）。
- 支持前端的任务排序、筛选和状态更新功能。

---

4. Gunicorn + Nginx

功能：

- 这两个工具共同构成了生产环境下后端应用的部署方案：
    - **Gunicorn**：Python WSGI HTTP 服务器，负责运行 Django 应用并处理 HTTP 请求。
    - **Nginx**：高性能 Web 服务器，用于反向代理和处理静态资源。

核心特点：

- **高性能**：Gunicorn 支持多线程和多进程，提高了并发处理能力。
- **反向代理**：Nginx 将客户端请求转发到 Gunicorn，分担负载并提升响应速度。
- **安全性**：Nginx 提供 HTTPS 支持和请求过滤功能，提高系统安全性。

用途：

- 在生产环境中，通过 Gunicorn 运行后端服务，并通过 Nginx 将请求分发至 Gunicorn 或直接提供静态文件。
- 确保系统稳定运行并支持高并发访问。

---

5. JWT (JSON Web Token)

功能：

- JSON Web Token 是一种轻量级的身份认证方式，用于在客户端和服务端之间安全传递用户身份信息。

核心特点：

- **无状态性**：服务端无需保存用户会话信息，令牌本身包含了用户的所有认证信息。
- **高安全性**：令牌使用签名技术（如 HMAC SHA256）进行加密，防止被篡改。
- **跨平台性**：可以通过 HTTP Header 在前后端之间传递，兼容性强。

用途：

- 在用户登录时，后端生成 JWT，前端将其保存在浏览器的 Local Storage 或 Cookie 中。
- 前端在后续请求中通过 HTTP Header（如 Authorization: Bearer <token>）携带 JWT 验证用户身份。
- 后端解码并验证 JWT，以确认请求的合法性和用户身份。

通过以上核心技术的集成应用，本实验构建了一个高效、可扩展且安全的任务管理系统，为功能实现和后续部署提供了可靠的技术保障。

3.3 理论来源

3.3.1 MVC 模式与前后端分离模式的概述

MVC 模式（Model-View-Controller）

- **概述**：

  MVC 是一种软件架构模式，用于分离系统的不同职责，从而提高代码的可维护性和可扩展性。

  - **Model（模型）**：负责数据的表示与处理，包括数据库交互和业务逻辑。
  - **View（视图）**：负责用户界面的呈现，直接与用户交互。
  - **Controller（控制器）**：充当桥梁，负责接收用户输入，并协调 Model 和 View 的交互。
- **优点**：
  - 清晰分离职责，便于多人协作开发。

- 易于进行功能扩展和模块化设计。

前后端分离模式

- **概述：**

  在传统的 MVC 模式中，View 和 Controller 的逻辑通常由后端完成。前后端分离模式则将前端视图完全独立，实现前后端技术栈解耦。前端通过 API 调用后端，后端专注于数据处理和业务逻辑。

- **特点：**
  - **前端：**通过 JavaScript 框架（如 React.js）构建动态页面，增强用户体验。
  - **后端：**通过 REST API 提供数据服务和逻辑处理，不直接渲染视图。
  - **通信方式：**前端与后端通过 HTTP 请求（JSON 格式）进行交互。

3.3.2 REST API 的基本原理与设计规范

REST API（Representational State Transfer Application Programming Interface）

- **概述：**

  REST 是一种基于 HTTP 协议的架构风格，API 是具体实现。REST API 提供了一种统一的方式，用于在前后端或不同系统之间传递数据。

- **基本原理：**
  1. **资源导向：**
     - 每个数据实体被视为资源，通过唯一的 URI 表示。
     - 示例：任务管理系统中的任务列表资源可以通过 /api/tasks 访问。
  2. **无状态性：**
     - 客户端的每个请求应包含所有必要的信息，服务端不保存客户端的状态。
  3. **操作方式：**
     - 使用标准 HTTP 方法来操作资源：
       - GET：获取资源

- POST：创建资源
- PUT：更新资源
- DELETE：删除资源

4. **统一接口：**
   - API 的设计应遵循一致性和标准化，使得开发者容易理解和使用。
5. **数据格式：**
   - 常用 JSON 格式传输数据，具有轻量、结构化的特点。

设计规范：

- **URI 命名规则：**
  - 使用名词描述资源，避免使用动词。
  - 示例：
    - 获取任务列表：GET /api/tasks
    - 更新特定任务：PUT /api/tasks/:id
- **状态码：**
  - 返回 HTTP 状态码表示操作结果：
    - 200 OK：请求成功
    - 201 Created：资源创建成功
    - 404 Not Found：资源不存在
    - 500 Internal Server Error：服务器内部错误
- **错误处理：**
  - 在响应中包含详细的错误信息，例如：

```
json
{
  "error": "Invalid Request",
  "message": "Task ID is required"
}
```

```json
{
  "error": "Invalid Request",
  "message": "Task ID is required"
}
```

---

3. 部署环境 Gunicorn + Nginx 的工作原理

Gunicorn（Green Unicorn）

- **概述：**

  Gunicorn 是一个 Python 的 WSGI HTTP 服务器，能够运行 Django 应用并处理客户端请求。

- **工作原理：**
  - Gunicorn 启动多个工作进程，每个进程负责处理客户端请求。
  - 支持同步或异步工作模式，可以高效处理高并发请求。
  - 作为 WSGI 服务器，Gunicorn 将客户端 HTTP 请求转化为 Python 应用可理解的格式，应用处理后再将结果返回客户端。

Nginx

- **概述：**

  Nginx 是一个高性能的 Web 服务器，用于处理静态资源、反向代理和负载均衡。

- **工作原理：**
  1. **反向代理：**
     - Nginx 接收客户端请求，将动态请求转发至 Gunicorn，静态请求直接由 Nginx 响应。
  2. **负载均衡：**
     - Nginx 能将请求分发到多台服务器或多个 Gunicorn 进程，提升系统吞吐量。
  3. **静态资源处理：**

- Nginx 负责处理静态文件（如 HTML、CSS、JavaScript 文件），减轻后端服务器负担。

Gunicorn + Nginx 配合流程：

1. 客户端向 Nginx 发起 HTTP 请求。
2. Nginx 分析请求：
   - 静态资源请求直接响应。
   - 动态请求转发至 Gunicorn。
3. Gunicorn 解析请求，调用 Django 应用处理后返回结果。
4. Nginx 将结果返回给客户端。

架构优势：

- **高性能**：Nginx 处理静态文件和反向代理，Gunicorn 专注于后端逻辑，分工明确。
- **稳定性**：Nginx 可在后端崩溃时缓存并返回错误页面，提升服务稳定性。
- **扩展性**：可以通过增加 Gunicorn 工作进程或服务器节点提升系统性能。

通过对上述理论的掌握，为任务管理系统的开发、设计和部署提供了坚实的理论基础，并确保系统功能的高效实现和稳定运行。

第四章 实验步骤

4.1 环境配置

4.1.1 前端环境：安装 Node.js、配置 React.js 项目

步骤 1：安装 Node.js

1. **检查系统是否已安装 Node.js**

   打开终端并输入以下命令查看版本：

   node -v

   npm -v

如果返回 Node.js 和 npm 的版本号，则表示已安装，可以跳过安装步骤。

2. **安装 Node.js**

   如果未安装，请按照以下步骤安装：

   - 更新系统的包管理工具：

     sudo apt update

     sudo apt upgrade

   - 安装 Node.js（使用 NodeSource 安装最新版本）：

     curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash -

     sudo apt install -y nodejs

   - 检查安装是否成功：

     node -v

     npm -v

输出版本号（如 v18.x.x 和 x.x.x）即表示安装成功。

步骤 2：创建 React.js 项目

1. 将 npm 下载源换为国内镜像源

   npm config set registry https://registry.npmmirror.com

2. **创建 React 项目**

在目标目录下运行以下命令以创建项目：

```
npx create-react-app task-manager
cd task-manager
```

- npx 将自动下载最新版本的 Create React App 工具并生成项目结构。
- 进入项目目录后，项目结构类似于：

```
task-manager/
├── node_modules/
├── public/
├── src/
├── package.json
├── README.md
├── .gitignore
└── yarn.lock
```

3. **启动开发服务器**

启动 React 开发服务器以检查项目是否正常运行：

```
npm start
```

成功后浏览器将自动打开 http://localhost:3000，显示 React 默认页面（React Logo 和欢迎消息）。

4. **验证项目是否创建成功**

在浏览器中访问页面，检查是否显示以下内容：

- 页面顶部有一个带旋转动画的 React 标志。
- 欢迎文本：“Edit src/App.js and save to reload.”

步骤 3：安装额外的开发依赖

根据实验需求安装一些必要的依赖：

- 安装 Axios（用于 HTTP 请求）：

```
npm install axios
```

- 安装 React Router（用于路由管理）：

  npm install react-router-dom

- 检查安装结果：

  查看 package.json 文件的 dependencies 字段，确保上述依赖被正确添加。

通过以上步骤，React.js 项目和开发环境的配置完成，并可以正常运行和开发任务管理系统的前端功能。

4.1.2 后端环境：安装 Django，创建项目并配置 MySQL 数据库

步骤 1：安装 Python 和虚拟环境工具 miniconda

1. **安装虚拟环境工具 miniconda**

   打开终端并输入以下命令：

   mkdir -p ~/miniconda3

   wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O ~/miniconda3/miniconda.sh

   bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3

   rm ~/miniconda3/miniconda.sh

2. **激活 conda**

   source ~/miniconda3/bin/activate

   要在所有可用 shell 上初始化 conda：

   conda init --all

3. 更新 conda

   conda update -n base -c defaults conda

   验证安装：

   conda --version

   返回版本号即安装成功。

步骤 2：创建虚拟环境并激活

1. **创建虚拟环境**

   mkdir backend

   cd backend

   conda create -n task-manager python==3.9

2. **激活虚拟环境**

   conda activate task-manager

激活后，终端提示符将变为 (task-manager) 开头，表示虚拟环境已启用。

步骤 3：安装 Django

1. **安装 Django**

   在激活的虚拟环境中运行以下命令：

   pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple

   pip install django

2. **检查安装是否成功**

   django-admin --version

返回 Django 的版本号（如 4.x.x），表示安装成功。

步骤 4：创建 Django 项目

1. **创建项目**

   在 backend 目录下运行以下命令：

   django-admin startproject task_manager .

该命令将在当前目录下创建 Django 项目的基础文件结构：

backend/

├── manage.py

├── task_manager/

   ├── __init__.py

   ├── asgi.py

   ├── settings.py

   ├── urls.py

   ├── wsgi.py

2. **启动开发服务器验证项目**

启动 Django 开发服务器：

python manage.py runserver

打开浏览器访问 http://127.0.0.1:8000，如果显示 Django 欢迎页面，则项目创建成功。

步骤 5：配置 MySQL 数据库

1. **安装 MySQL 和相关驱动**

sudo apt install -y mysql-server libmysqlclient-dev

pip install mysqlclient

验证安装：

mysql --version

返回 MySQL 的版本号（如 8.x.x）表示成功。

2. **创建数据库和用户**

进入 MySQL 控制台：

sudo mysql

在控制台中运行以下命令创建数据库和用户：

```
CREATE DATABASE task_db;
CREATE USER 'task_user'@'localhost' IDENTIFIED BY 'password123';
GRANT ALL PRIVILEGES ON task_db.* TO 'task_user'@'localhost';
FLUSH PRIVILEGES;
```

退出 MySQL 控制台：

```
exit;
```

3. **配置 Django 使用 MySQL**

打开 task_manager/settings.py 文件，修改数据库配置为：

```
DATABASES = {
'default': {
    'ENGINE': 'django.db.backends.mysql',
    'NAME': 'task_db',
    'USER': 'task_user',
```

```
            'PASSWORD': 'password123',
            'HOST': 'localhost',
            'POST': '3306',
            'OPTIONS': {
                'unix_socket': '/var/run/mysqld/mysqld.sock',
            },
        }
    }
```

4. **迁移数据库**

先修改用户的认证插件，为 mysql_native_password

```
sudo mysql
alter user ‘task_user’@’localhost’ identified with mysql_native_password by ‘your_password’;
flush privileges;
```

验证切换是否成功

```
Select user, host,plugin from mysql.user;
```

确认 task_user 的 plugin 字段变为 mysql_native_password。

运行以下命令创建初始数据库表：

```
python manage.py makemigrations
python manage.py migrate
```

检查输出信息，若显示表成功创建则表示配置完成。

通过以上步骤，后端开发环境已完成配置，Django 项目已成功连接 MySQL 数据库，并可以进行后续开发和测试。

4.1.3 部署环境：配置 Ubuntu 服务器，安装 MySQL、Gunicorn 和 Nginx

步骤 1：准备部署环境

1. **更新系统包管理工具**

   在服务器终端运行以下命令，确保系统软件包是最新的：

   sudo apt update

   sudo apt upgrade -y

2. **安装常用工具**

   安装用于调试和管理的工具：

   sudo apt install -y build-essential curl unzip git

步骤 2：安装 MySQL

1. **安装 MySQL 服务器**

   sudo apt install -y mysql-server

2. **启动和检查 MySQL 服务**

   sudo systemctl start mysql

   sudo systemctl enable mysql

   sudo systemctl status mysql

检查点：如果显示 MySQL 服务为 "active (running)"，则 MySQL 配置成功。

3. **配置安全性（可选）**

   运行安全脚本设置 MySQL 密码和权限：

   sudo mysql_secure_installation

步骤 3：安装 Gunicorn

1. **激活后端虚拟环境**

   如果后端虚拟环境未激活，先激活：

   conda activate task-manager

2. **安装 Gunicorn**

   pip install gunicorn

3. **测试 Gunicorn 运行后端项目**

   在虚拟环境下运行：

   gunicorn --bind 127.0.0.1:8000 task_manager.wsgi

打开浏览器访问 http://<服务器 IP>:8000，如果显示 Django 项目页面，则 Gunicorn 安装成功。

步骤 4：安装 Nginx

1. **安装 Nginx**

   sudo apt install -y nginx

2. **启动和检查 Nginx 服务**

   sudo systemctl start nginx

   sudo systemctl enable nginx

   sudo systemctl status nginx

检查点：如果状态显示为 "active (running)"，Nginx 配置成功。

3. **配置 Nginx 反向代理**

   创建一个新的配置文件：

   sudo nano /etc/nginx/sites-available/task_manager

在文件中添加以下内容：

```
server {
    listen 80;
    server_name <服务器 IP 或域名>;

    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location /static/ {
        alias /path/to/static/files/;
    }
}
```

激活配置并重启 Nginx：

sudo ln -s /etc/nginx/sites-available/task_manager /etc/nginx/sites-enabled/

sudo nginx -t

sudo systemctl restart nginx

检查点：运行 sudo nginx -t 时如果输出 "syntax is ok"，则配置文件正确。

步骤 5：测试部署环境

1. **静态文件收集**

   在虚拟环境中运行：

   python manage.py collectstatic

确保静态文件正确存放到 /path/to/static/files/。

2. **启动完整服务**

   使用 Gunicorn 启动后端项目：

   gunicorn --bind 127.0.0.1:8000 task_manager.wsgi

打开浏览器访问服务器的 IP 地址，检查是否能正常访问项目页面。

3. **通过 Nginx 验证**

   停止 Django 自带的开发服务器，直接访问 Nginx 代理的 URL http://<服务器 IP>，检查网站是否正确显示。

通过以上步骤，部署环境已完成配置。服务端通过 Gunicorn 提供后端逻辑处理，Nginx 负责反向代理和静态资源管理，部署的任务管理系统已成功运行在 Ubuntu 服务器上。

4.1.4 开发工具：配置 Git、build-essential 和 curl

步骤 1：安装 Git

1. **安装 Git（**

   sudo apt update

sudo apt install -y git

2. **验证安装**

   运行以下命令再次检查：

   git --version

   返回版本号表示安装成功。

3. **配置 Git 用户信息**

   设置用户名称和邮箱：

   git config --global user.name "Your Name"

   git config --global user.email "your_email@example.com"

   验证配置：

   git config --list

   输出包含用户名和邮箱信息表示配置成功。

步骤 2：安装 build-essential

1. **安装 build-essential**

   运行以下命令安装编译工具集：

   sudo apt install -y build-essential

2. **验证安装**

   运行以下命令验证：

   gcc --version

   make --version

   如果返回 GCC 和 Make 的版本号，则 build-essential 安装成功。

步骤 3：安装 curl

1. **检查是否已安装 curl**

   运行以下命令检查：

   curl --version

如果返回 curl 的版本号（如 curl 7.x.x），则已安装成功。

2. **安装 curl（如果未安装）**

   sudo apt install -y curl

3. **验证安装**

运行以下命令再次检查：

curl --version

返回版本号表示安装成功。

4. **测试 curl 功能**

使用 curl 访问一个 URL 测试功能是否正常：

curl http://example.com

如果返回 HTML 内容，说明 curl 功能正常。

步骤 4：总结检查

通过以下命令检查各工具是否已正确安装：

git --version

gcc --version

make --version

curl --version

如果所有命令返回相应工具的版本号，表示开发工具已正确配置完成，可以支持实验的后续开发与调试工作。

4.2 数据库设计

数据库设计是任务管理系统的核心部分，用于存储用户信息和任务数据。本项目使用 MySQL 作为数据库，设计了两个主要表：用户表和任务表，并定义它们之间的关系。

---

1. 用户表设计

字段名	数据类型	约束	说明
id	INT	主键，自增	用户的唯一标识符
username	VARCHAR(50)	唯一，非空	用户名

字段名	数据类型	约束	说明
email	VARCHAR(100)	唯一，非空	用户邮箱
password	VARCHAR(255)	非空	用户的加密密码
created_at	DATETIME	默认当前时间	用户注册时间

表说明

- 用户表存储用户的基本信息，如用户名、邮箱和加密后的密码。
- 每个用户有一个唯一的 id，用于关联其他表中的数据。

---

2. 任务表设计

字段名	数据类型	约束	说明
id	INT	主键，自增	任务的唯一标识符
title	VARCHAR(100)	非空	任务标题
description	TEXT	可为空	任务描述
status	ENUM	默认值 '未完成'	任务状态（未完成 或 已完成）
priority	ENUM	默认值 '中'	任务优先级（低、中、高）
user_id	INT	外键，关联用户表	任务所属用户
created_at	DATETIME	默认当前时间	任务创建时间
updated_at	DATETIME	自动更新	任务最后更新时间

表说明

- 任务表存储用户的任务信息，包括标题、描述、状态、优先级等。
- user_id 是外键，关联用户表的 id，表示任务归属的用户。

---

3. 表关系

- **一对多关系：**

  一个用户可以拥有多个任务，因此用户表与任务表之间存在一对多关系。

用户表的主键 id 是任务表的外键 user_id。

---

4. 数据库关系图

---

5. SQL 建表语句

用户表：

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    email VARCHAR(100) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

任务表：

```
CREATE TABLE tasks (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(100) NOT NULL,
    description TEXT,
    status ENUM('未完成', '已完成') DEFAULT '未完成',
    priority ENUM('低', '中', '高') DEFAULT '中',
    user_id INT NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
```

updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,

FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE

);

---

6. 数据库设计的合理性验证

1. **字段完整性**

   每个表的字段均有必要性，确保数据的准确存储。

2. **关系完整性**

   外键关系定义了用户与任务之间的关联，支持级联删除，避免孤立任务记录。

3. **扩展性**

   增加新字段（如任务截止时间）时，表结构修改成本较低。

以上数据库设计为系统的后端提供了稳定的数据存储基础，同时确保数据之间的关联性，满足了任务管理系统的需求。

4.3 后端开发

4.3.1 创建 Django 项目与应用

---

1 初始化 Django 项目

1. 打开终端，在项目目录下运行以下命令（环境配置时已经完成，这里不需要再执行），使用 Django 创建项目根目录：

   django-admin startproject task_manager

   cd task_manager

2. 配置项目的 settings.py 文件（环境配置时已经完成，这里不需要再执行）：
   - 打开 task_manager/settings.py，定位到 **DATABASES** 配置，修改为 MySQL 数据库连接信息：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'task_manager_db',  # 数据库名称
        'USER': 'your_username',      # 数据库用户名
        'PASSWORD': 'your_password',# 数据库密码
        'HOST': 'localhost',             # 数据库主机
        'PORT': '3306',                    # 数据库端口
    }
}
```

- 在 **INSTALLED_APPS** 部分，添加 Django REST Framework 和 MySQL 驱动支持：

```
INSTALLED_APPS = [
    # 默认应用
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # 新增的应用
    'rest_framework',       # Django REST Framework
    'corsheaders',          # 跨域请求支持
]
```

- 添加 CORS 支持配置：

  在 MIDDLEWARE 中添加 CorsMiddleware，并配置允许的跨域源：

```
MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
```

```
    ...
]

CORS_ALLOWED_ORIGINS = [
    "http://localhost:3000",  # 允许 React.js 前端的跨域请求
]
```

3. 测试数据库连接是否正确：
   - 使用以下命令检查 Django 是否可以成功连接到 MySQL 数据库：

     python manage.py migrate
   - 如果无错误输出，则数据库连接配置成功。

---

2 创建核心应用

1. 创建用户管理和任务管理的核心应用：

   python manage.py startapp users

   python manage.py startapp tasks
2. 注册新应用到 Django 项目中：
   - 打开 task_manager/settings.py，在 **INSTALLED_APPS** 中添加新创建的应用：

     ```
     INSTALLED_APPS += [
         'users',
         'tasks',
     ]
     ```
3. 创建初始模型和管理页面：
   - 在 users/models.py 和 tasks/models.py 中创建数据库模型（后续部分详细实现）。
   - 运行以下命令生成初始迁移文件并应用到数据库：

     python manage.py makemigrations

     python manage.py migrate
4. 检验项目是否正常工作：

- 启动 Django 开发服务器：

  python manage.py runserver
- 在浏览器中访问 http://127.0.0.1:8000/，确保项目正常运行且无错误日志输出。

输出成果

- 创建了名为 task_manager 的 Django 项目，完成了基本配置和数据库连接设置。
- 创建了 users 和 tasks 两个核心应用，并成功注册到项目中。
- 项目运行正常，为后续后端开发提供了基础环境。

注：Django 项目结构的简单说明

这是一个 Django 后端项目的文件结构。各个文件和文件夹的作用如下：

根目录

- **db.sqlite3**：Django 默认使用的数据库文件，这里使用的是 SQLite 数据库。
- **manage.py**：Django 项目的命令行管理工具，用于启动开发服务器、数据库迁移、应用管理等操作。

task_manager 目录（Django 项目配置文件）

- **asgi.py**：ASGI（Asynchronous Server Gateway Interface）配置文件，适用于支持异步处理的部署环境。
- **wsgi.py**：WSGI（Web Server Gateway Interface）配置文件，适用于传统的同步 web 服务器。
- **settings.py**：Django 项目的主要配置文件，包含数据库配置、应用程序配置、安全设置等。
- **urls.py**：定义项目的 URL 路由规则，将请求分发到相应的视图函数。
- **init.py**：使当前目录成为一个 Python 包，通常为空。

tasks 目录（应用之一，可能涉及任务管理）

- **admin.py**：用于注册模型到 Django 管理后台，方便进行数据的增删改查

操作。

- **apps.py**：定义该 Django 应用的配置，应用的名称和其他配置项。
- **models.py**：定义 Django 数据模型，映射到数据库表。
- **migrations** 目录：包含数据库迁移文件，用于追踪和管理模型变更。每次修改模型后，可以生成相应的迁移文件。
    - **0001_initial.py**：初始数据库迁移文件，定义数据库结构。
    - **0002_initial.py**：可能是第二次迁移文件。
- **tests.py**：用于编写单元测试，验证应用功能。
- **views.py**：定义视图函数（或类视图），响应用户请求并返回结果。

users 目录（应用之一，可能涉及用户管理）

- **admin.py**：类似于 tasks 目录中的 admin.py，用于注册用户模型到 Django 管理后台。
- **apps.py**：类似于 tasks 目录中的 apps.py，用于定义该应用的配置信息。
- **models.py**：定义与用户相关的数据库模型。
- **migrations** 目录：包含与用户模型相关的数据库迁移文件。
    - **0001_initial.py**：初始迁移文件，创建用户模型所需的数据库表。
- **tests.py**：用于编写与用户相关的单元测试。
- **views.py**：定义处理与用户相关的请求的视图函数（如登录、注册等）。

pycache 目录（编译文件缓存）

这些是 Python 解释器编译生成的 .pyc 文件，存储已编译的 Python 代码，用于提高程序启动速度。

总结来说：

- task_manager 是项目的核心配置文件夹。
- tasks 和 users 是 Django 项目中的两个应用，每个应用负责不同的功能模块。
- 每个应用包含模型（models.py）、视图（views.py）、后台管理（admin.py）等文件，负责不同层次的功能。

注：Django 如何与 MySQL 进行互动？

在 Django 中，MySQL 是作为数据库后端进行交互的。Django 通过其 ORM（对象关系映射）系统与 MySQL 数据库进行互动，使开发者无需直接编写 SQL 语句即可进行数据库操作。以下是 Django 和 MySQL 互动的具体过程及步骤：

1. 安装 MySQL 和 Django MySQL 驱动

为了使 Django 能与 MySQL 数据库交互，首先需要安装 MySQL 数据库并安装 MySQL 驱动 mysqlclient：

```
pip install mysqlclient
```

2. 配置 Django 使用 MySQL 数据库

在 Django 项目中，数据库配置是通过 settings.py 文件进行的。你需要修改 DATABASES 配置，指定使用 MySQL 作为数据库后端，并提供相关的连接参数，如数据库名、用户名、密码、主机和端口等。

以下是一个典型的 MySQL 数据库配置示例：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',   # 使用 MySQL 数据库后端
        'NAME': 'your_database_name',              # 数据库名称
        'USER': 'your_database_user',              # 数据库用户名
        'PASSWORD': 'your_database_password',   # 数据库密码
        'HOST': 'localhost',                          # 数据库主机，通常是 localhost
        'PORT': '3306',                                 # MySQL 默认端口
    }
}
```

3. 定义模型（Models）

Django 使用 **模型（Model）** 来定义数据库表。模型是 Python 类，通过 Django 的 ORM 系统来自动生成数据库表，并实现与数据库的交互。

例如，创建一个表示任务的模型：

```
from django.db import models
```

```
class Task(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    due_date = models.DateTimeField()

    def __str__(self):
        return self.title
```

上述模型 Task 会映射到一个 MySQL 数据库表（表名为 tasks_task，Django 会自动添加前缀）。

4. 数据库迁移（Migrations）

Django 的 **迁移** 系统帮助你管理数据库表的创建和更新。迁移通过 makemigrations 和 migrate 命令进行。

- makemigrations：生成数据库迁移文件，记录模型的变化。
- migrate：根据迁移文件创建或更新数据库中的表。

首先，生成迁移文件：

```
python manage.py makemigrations
```

然后，执行迁移：

```
python manage.py migrate
```

这将根据你的模型在 MySQL 数据库中创建表结构。

5. ORM 与数据库交互

Django 的 ORM 允许你像操作 Python 对象一样操作数据库，进行数据的增、删、改、查。常见的操作有：

增加数据

使用模型类创建数据：

```
task = Task.objects.create(
    title="Finish Django Tutorial",
    description="Complete the Django tutorial to understand how to use Django and MySQL",
    due_date="2024-12-01T10:00:00"
```

)

查询数据

Django 提供了丰富的查询 API 来检索数据：

```
# 查询所有任务
tasks = Task.objects.all()

# 通过条件查询任务
task = Task.objects.filter(title="Finish Django Tutorial")

# 获取某个特定任务
task = Task.objects.get(id=1)
```

更新数据

可以使用 update() 方法来更新数据：

```
task = Task.objects.get(id=1)
task.description = "Complete the tutorial and build a project"
task.save()
```

删除数据

可以使用 delete() 方法删除数据：

```
task = Task.objects.get(id=1)
task.delete()
```

6. 使用原生 SQL 查询（可选）

尽管 Django 提供了 ORM 来进行大多数数据库操作，但有时你可能需要执行复杂的 SQL 查询。Django 允许你直接执行原生 SQL 查询。

```
from django.db import connection

def get_task_count():
    with connection.cursor() as cursor:
        cursor.execute("SELECT COUNT(*) FROM tasks_task")
        row = cursor.fetchone()
    return row[0]
```

7. 数据库性能与优化

与 MySQL 进行交互时，Django 的 ORM 会自动生成 SQL 查询，并且它通过连接池来管理数据库连接。不过，如果数据库表非常大，或者查询复杂时，可能需要优化查询和数据库配置。

常见的优化方式包括：

- **查询优化**：使用 select_related() 或 prefetch_related() 来减少数据库查询次数，优化关联查询。
- **索引**：在 MySQL 数据库中使用索引加速查询，Django 可以在模型字段上定义索引。
- **数据库连接池**：使用数据库连接池来提高性能，特别是在高并发情况下。

8. 数据库事务

Django 默认支持数据库事务。这意味着数据库操作会在一个事务中执行，确保数据一致性。如果某个操作失败，所有更改会被回滚。

可以使用 transaction.atomic() 来显式地管理事务：

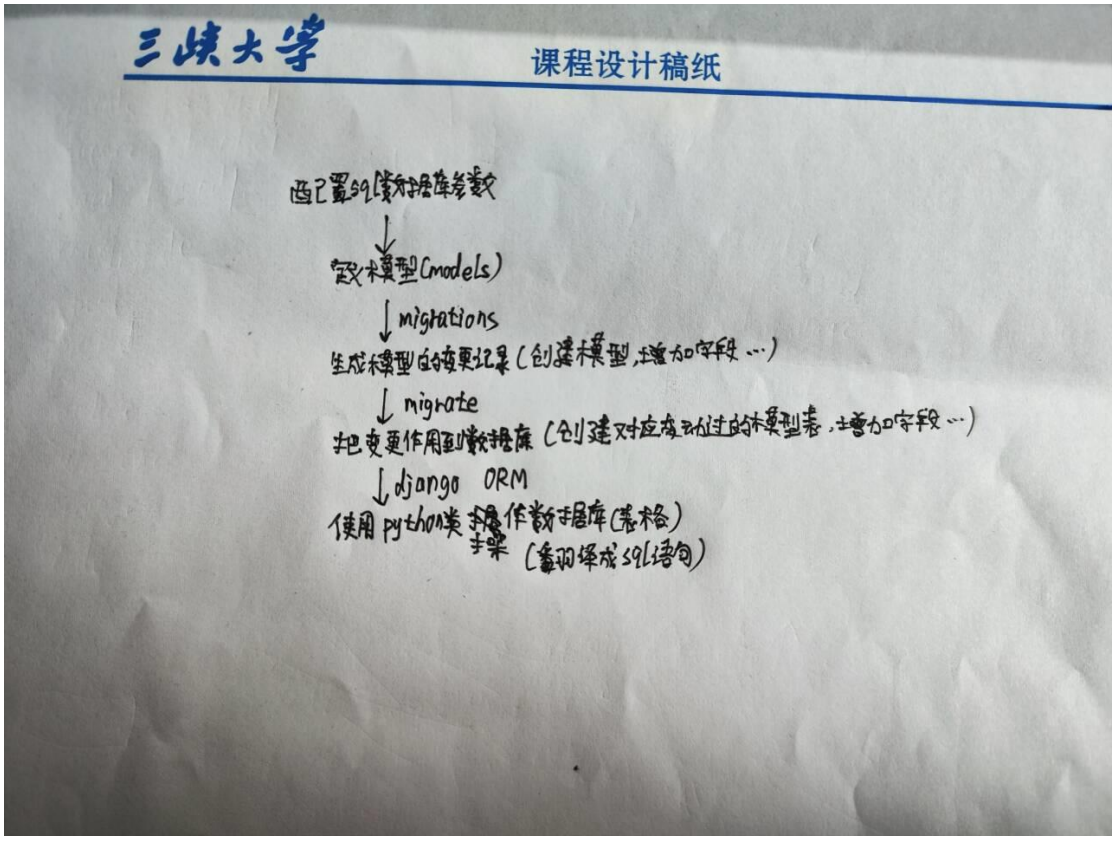
```
from django.db import transaction

with transaction.atomic():
    task = Task.objects.create(
        title="New Task",
        description="Description of the new task",
        due_date="2024-12-01T10:00:00"
    )
    # 如果这里抛出异常，以上操作将回滚
```

总结

Django 和 MySQL 通过 ORM 系统进行交互。Django 自动将 Python 类映射为 MySQL 表，并允许开发者通过 Python 代码进行数据的增删改查。Django 的数据库迁移系统帮助管理数据库结构，ORM 提供了灵活和高效的查询方式，同时也支持执行原生 SQL 查询。在性能优化、事务管理等方面，Django 提供了丰富的工具来确保数据库操作的效率和一致性。

迁移系统的详细说明图：



4.3.2 数据库模型设计

1 定义用户模型

1. **使用 Django 内置 User 模型：**
   - Django 提供了内置的 User 模型，包含用户名、密码、电子邮件等常用字段。
   - 在此基础上，可以通过 Django 的 AbstractUser 或 One-to-One 关联扩展额外字段。
2. **扩展用户模型：**
   - 创建一个扩展的用户模型，在 users/models.py 中定义：

     from django.contrib.auth.models import AbstractUser

     from django.db import models

class CustomUser(AbstractUser):

registration_date=models.DateTimeField(auto_now_add=True, verbose_name="注册时间")

- 在 settings.py 中指定使用自定义用户模型：

AUTH_USER_MODEL = 'users.CustomUser'

2 定义任务模型

1. **任务模型的字段设计：**
   - 在 tasks/models.py 中定义任务模型：

```
from django.db import models
from django.conf import settings

class Task(models.Model):
    STATUS_CHOICES = [
        ('pending', '未完成'),
        ('completed', '已完成'),
    ]

    PRIORITY_CHOICES = [
        ('low', '低'),
        ('medium', '中'),
        ('high', '高'),
    ]

    title = models.CharField(max_length=255, verbose_name="任务标题")
    description = models.TextField(blank=True, verbose_name="任务描述")
    status=models.CharField(max_length=10,choices=STATUS_CHOICES, default='pending', verbose_name="任务状态")
    priority=models.CharField(max_length=10,choices=PRIORITY_
```

```
CHOICES, default='medium', verbose_name="任务优先级")
   created_at=models.DateTimeField(auto_now_add=True,verbose_
name="创建时间")
      updated_at=models.DateTimeField(auto_now=True,verbose_na
me="更新时间")
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name='tasks',
        verbose_name="所属用户"
    )

    def __str__(self):
        return self.title
```

2. **字段功能说明：**
   - title：任务标题，必填，字符长度不超过 255。
   - description：任务的详细描述，可选字段。
   - status：任务状态，提供 "未完成" 和 "已完成" 两种选择。
   - priority：任务优先级，提供 "低"、"中" 和 "高" 三种选择。
   - created_at 和 updated_at：自动记录任务的创建和更新时间。
   - user：外键字段，关联到用户模型，表示任务归属于哪个用户。

3 迁移数据库

1. **生成迁移文件：**
   - 在终端中运行以下命令，生成数据模型的迁移文件：

     ```
     python manage.py makemigrations
     ```
   - 成功生成迁移文件后，输出类似以下内容：

     ```
     Migrations for 'users':
       users/migrations/0001_initial.py
         - Create model CustomUser
     Migrations for 'tasks':
     ```

tasks/migrations/0001_initial.py

- Create model Task

2. **应用迁移文件：**
    - 在终端中运行以下命令，将迁移文件应用到数据库：

      python manage.py migrate
    - 成功迁移后，输出类似以下内容：

      Applying users.0001_initial... OK

      Applying tasks.0001_initial... OK
3. **验证数据库表结构：**
    - 登录 MySQL 数据库查看表是否正确创建：

      SHOW TABLES;

      DESCRIBE users_customuser;

      DESCRIBE tasks_task;

输出成果

- 用户表 users_customuser，包含基础用户字段及注册时间字段。
- 任务表 tasks_task，包含任务相关字段及与用户的外键关联。
- 数据库成功迁移，为后续开发提供了数据存储基础。

4.3.3 开发 REST API

---

1 配置 Django REST Framework (DRF)

1. **安装 Django REST Framework**
    - 在终端运行以下命令安装 DRF：

      pip install djangorestframework
2. **安装 JSON Web Token (JWT) 支持库**
    - 安装 djangorestframework-simplejwt：

      pip install djangorestframework-simplejwt
3. **启用 DRF 和 SimpleJWT**

- 在 settings.py 中注册 DRF 和配置 JWT：

```
INSTALLED_APPS += [
    'rest_framework',
    'rest_framework_simplejwt',
]
```

4. **配置 REST Framework 设置**
   - 在 settings.py 中添加全局配置，启用 JWT 认证：

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
}
```

5. **配置 JWT 设置**
   - 在 settings.py 中设置令牌有效期等参数：

```
from datetime import timedelta

SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=30),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=7),
    'ROTATE_REFRESH_TOKENS': True,
    'BLACKLIST_AFTER_ROTATION': True,
    'AUTH_HEADER_TYPES': ('Bearer',),
}
```

6. **验证安装成功**
   - 启动 Django 开发服务器：

```
python manage.py runserver
```

- 确保项目无错误日志输出，并可正常访问。

2 实现用户认证 API

1. 创建用户认证的视图和路由

- 在 users 应用中创建 views.py 文件，并定义注册和登录视图。
- 定义以下两个 API：
  - 用户注册 API
  - 用户登录（令牌生成）API

2. 用户注册 API

- 在 users/views.py 中实现注册逻辑：

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from django.contrib.auth import get_user_model
from rest_framework.permissions import AllowAny

User = get_user_model()

class RegisterUserAPIView(APIView):
    permission_classes = [AllowAny]

    def post(self, request):
        username = request.data.get("username")
        password = request.data.get("password")

        if not username or not password:
            return Response({"error": "用户名和密码是必填项"},
status=status.HTTP_400_BAD_REQUEST)

        if User.objects.filter(username=username).exists():
```

```
            return Response({"error": "用户名已存在"}, status=status.HTTP_400_BAD_REQUEST)

        user = User.objects.create_user(username=username, password=password)
        return Response({"message": "注册成功", "user_id": user.id}, status=status.HTTP_201_CREATED)
```

3. **用户登录 API (JWT Token)**
   - 在 users/views.py 中实现登录逻辑，使用 DRF 的 SimpleJWT 提供的 TokenObtainPairView：

```
from rest_framework_simplejwt.views import TokenObtainPairView
from rest_framework_simplejwt.serializers import TokenObtainPairSerializer

class CustomTokenObtainPairSerializer(TokenObtainPairSerializer):
    @classmethod
    def get_token(cls, user):
        token = super().get_token(user)
        # 添加额外的用户信息到令牌
        token['username'] = user.username
        return token

class LoginAPIView(TokenObtainPairView):
    serializer_class = CustomTokenObtainPairSerializer
```

4. **定义路由**
   - 在 users/urls.py 中定义路由：

```
from django.urls import path
from .views import RegisterUserAPIView, LoginAPIView

urlpatterns = [
```

```
    path('register/', RegisterUserAPIView.as_view(), name='register'),
    path('login/', LoginAPIView.as_view(), name='login'),
]
```

- 在项目的 urls.py 中包含用户认证路由：

```
from django.urls import path, include

urlpatterns = [
    path('api/users/', include('users.urls')),
]
```

5. **验证 JWT Token 功能**
   - 测试注册：通过 Postman 或 curl，发送 POST 请求到 /api/users/register/，提供用户名和密码字段。
   - 测试登录：发送 POST 请求到 /api/users/login/，获取 JWT 令牌。
   - 使用返回的令牌访问受保护的 API，确保认证成功。

输出成果

- 完成用户注册与登录功能，支持 JWT 身份认证。
- 配置了 Django REST Framework 和 SimpleJWT。
- 提供了注册和登录的 REST API，具备令牌生成和用户身份认证能力。

4.3.4 实现任务管理 API

在 tasks 应用中实现任务管理功能的 API，包括创建、获取列表、编辑、删除任务。

---

1. 创建任务管理的序列化器

在 tasks/serializers.py 中定义序列化器，用于验证和转换数据：

```
from rest_framework import serializers
from .models import Task
```

```
class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = ['id', 'title', 'description', 'is_completed', 'priority', 'created_at', 'updated_at']
```

---

3. 开发任务管理视图

在 tasks/views.py 中实现任务的 CRUD 操作视图：

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from rest_framework.permissions import IsAuthenticated
from .models import Task
from .serializers import TaskSerializer

class TaskListCreateAPIView(APIView):
    permission_classes = [IsAuthenticated]

    def get(self, request):
        tasks = Task.objects.filter(user=request.user)
        serializer = TaskSerializer(tasks, many=True)
        return Response(serializer.data, status=status.HTTP_200_OK)

    def post(self, request):
        data = request.data
        data['user'] = request.user.id  # 自动绑定当前登录用户
        serializer = TaskSerializer(data=data)
        if serializer.is_valid():
            serializer.save(user=request.user)
```

```
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

class TaskDetailAPIView(APIView):
    permission_classes = [IsAuthenticated]

    def get_object(self, pk, user):
        try:
            return Task.objects.get(pk=pk, user=user)
        except Task.DoesNotExist:
            return None

    def put(self, request, pk):
        task = self.get_object(pk, request.user)
        if not task:
            return Response({"error": "Task not found or not accessible"}, status=status.HTTP_404_NOT_FOUND)

        serializer = TaskSerializer(task, data=request.data, partial=False)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_200_OK)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    def patch(self, request, pk):
        task = self.get_object(pk, request.user)
        if not task:
            return Response({"error": "Task not found or not accessible"},
```

```
status=status.HTTP_404_NOT_FOUND)

            serializer = TaskSerializer(task, data=request.data, partial=True)
            if serializer.is_valid():
                serializer.save()
                return Response(serializer.data, status=status.HTTP_200_OK)
            return                                          Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk):
            task = self.get_object(pk, request.user)
            if not task:
                return Response({"error": "Task not found or not accessible"},
status=status.HTTP_404_NOT_FOUND)

            task.delete()
            return Response({"message": "Task deleted successfully"},
status=status.HTTP_204_NO_CONTENT)
```

4. 定义任务管理路由

在 tasks/urls.py 中定义 API 路由：

```
from django.urls import path
from .views import TaskListCreateAPIView, TaskDetailAPIView

urlpatterns = [
    path('', TaskListCreateAPIView.as_view(), name='task_list_create'),
    path('<int:pk>/', TaskDetailAPIView.as_view(), name='task_detail'),
]
```

在项目的 urls.py 中包含任务的路由：

```
from django.urls import path, include
```

```
urlpatterns = [
    path('api/tasks/', include('tasks.urls')),
]
```

---

5. 测试任务管理 API

使用 Postman 或 curl 测试以下端点：

1. **创建任务（POST /api/tasks/）**
    - 请求体示例：

```
{
    "title": "Buy groceries",
    "description": "Milk, eggs, bread",
    "priority": "medium"
}
```

2. **获取任务列表（GET /api/tasks/）**
    - 返回当前用户的任务列表。
3. **编辑任务（PUT /api/tasks/<id>/）**
    - 请求体示例：

```
{
    "title": "Buy groceries and vegetables",
    "description": "Milk, eggs, bread, carrots",
    "priority": "high",
    "is_completed": true
}
```

4. **部分更新任务（PATCH /api/tasks/<id>/）**
    - 请求体示例：

```
{
    "is_completed": true
}
```

5. **删除任务（DELETE /api/tasks/<id>/）**
    - 返回状态码 204，无响应体。

输出成果

- 实现了完整的任务管理 REST API，包括创建、读取、更新和删除任务的功能。
- 确保所有操作基于登录用户，保证数据安全性和隔离性。

4.3.5 集成 API 文档生成工具 （待优化，不用做）

API 文档是后端开发的重要组成部分，用于描述 API 的功能和使用方法。我们将使用 **Django REST Framework 提供的自动文档生成** 和 **Swagger** 进行集成。

(1) 安装依赖

使用 pip 安装 drf-yasg（Swagger 生成工具）：

```
pip install drf-yasg
```

(2) 配置 Swagger

在项目的主 urls.py 文件中添加 Swagger 文档路由：

```
from rest_framework import permissions
from drf_yasg.views import get_schema_view
from drf_yasg import openapi


schema_view = get_schema_view(
    openapi.Info(
        title="Task Management API",
        default_version='v1',
        description="API documentation for the Task Management System",
        terms_of_service="https://www.example.com/terms/",
        contact=openapi.Contact(email="contact@example.com"),
        license=openapi.License(name="MIT License"),
```

```
    ),
    public=True,
    permission_classes=[permissions.AllowAny],
)

urlpatterns += [
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger-ui'),
    path('redoc/', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
]
```

---

3. 测试 API 文档

- **Swagger 文档访问**：启动服务器后访问 http://<your-server>/swagger/，查看 Swagger 界面，测试所有 API 端点。
- **Redoc 文档访问**：访问 http://<your-server>/redoc/ 查看简洁版的文档。

输出成果

- 配置了用户认证和任务管理相关的路由。
- 集成了 Swagger 和 Redoc，为项目生成了自动化 API 文档。
- 测试并验证所有路由和文档的功能，确保前后端开发和维护时具备良好的参考工具。

4.4 前端开发

4.4.1 项目初始化

1 安装所需工具与库

1. **初始化 React 项目**

使用 create-react-app 快速初始化 React 项目：

```
npx create-react-app task-manager
cd task-manager
```

2. **安装基础开发工具与依赖**

运行以下命令安装项目所需的库：

```
npm install react-router-dom axios formik yup
```

- **react-router-dom**：用于前端路由配置和页面导航。
- **axios**：用于与后端 API 通信，处理 HTTP 请求和响应。
- **formik**：用于管理表单状态，简化表单处理。
- **yup**：配合 Formik 进行表单验证，提供强大的验证规则。

3. **启动开发环境**

确认安装完成后，启动项目以验证 React 项目是否初始化成功：

```
npm start
```

打开浏览器访问 http://localhost:3000，如果看到 React 的默认页面说明环境配置成功。

1.2 项目目录结构设计

根据功能模块划分目录结构，确保代码可维护性和清晰度。以下为推荐的目录结构：

```
task-manager/
│
├── public/                    # 静态资源目录
│   └── index.html             # 应用入口文件
│
├── src/                       # 源代码目录
│   ├── components/            # 可复用的通用组件
│   │   ├── Header.js
│   │   ├── TaskItem.js
```

```
│   │   └── ...
│   │
│   ├── pages/                  # 页面组件
│   │   ├── LoginPage.js
│   │   ├── TaskListPage.js
│   │   └── TaskEditPage.js
│   │
│   ├── services/               # API 请求服务
│   │   ├── api.js              # 封装 Axios 请求方法
│   │   └── auth.js             # 用户认证相关接口
│   │
│   ├── context/                # 全局状态管理
│   │   ├── AuthContext.js      # 管理用户登录状态
│   │   └── ...
│   │
│   ├── App.js                  # 应用主组件
│   ├── index.js                # 应用入口文件
│   ├── styles.css              # 全局样式表
│   └── ...
│
├── package.json                # 项目配置文件
└── ...
```

目录设计说明：

- **components/**：存放可复用的小型组件，如导航栏、按钮、任务列表项等。
- **pages/**：存放页面级组件，每个页面对应一个独立文件。
- **services/**：存放与 API 通信的封装逻辑，方便后续维护和扩展。
- **context/**：用于存储全局状态管理文件，如用户认证和任务列表状态。

通过合理的目录结构和模块划分，确保代码易于理解和扩展。

4.4.2 创建登录页面

---

1. 功能目标

实现用户登录功能，包括输入账号和密码，通过表单校验提交后与后端交互，并存储登录的 JWT 令牌。

---

2. 主要代码实现

2.1 创建登录表单组件

文件路径：src/pages/LoginPage.js

```
import React, { useState } from "react";
import { Formik, Form, Field, ErrorMessage } from "formik";
import * as Yup from "yup";
import axios from "axios";
import { useNavigate } from "react-router-dom";

const LoginPage = () => {
  const navigate = useNavigate();
  const [error, setError] = useState("");

  // 表单校验规则
  const validationSchema = Yup.object().shape({
    username: Yup.string().required("用户名不能为空"),
    password: Yup.string().required("密码不能为空"),
  });

  // 提交表单逻辑
  const handleSubmit = async (values, { setSubmitting }) => {
    try {
      const response = await axios.post("http://localhost:8000/api/token/", {
```

```
        username: values.username,
        password: values.password,
      });
      // 将 JWT 令牌存储到 localStorage
      localStorage.setItem("access_token", response.data.access);
      localStorage.setItem("refresh_token", response.data.refresh);
      setError(""); // 清除之前的错误信息
      navigate("/tasks"); // 跳转到任务列表页面
    } catch (err) {
      setError("登录失败，请检查用户名和密码");
    }
    setSubmitting(false); // 表单提交结束
  };

  return (
    <div style={{ maxWidth: "400px", margin: "50px auto", textAlign: "center" }}>
      <h1>登录</h1>
      {error && <div style={{ color: "red", marginBottom: "10px" }}>{error}</div>}
      <Formik
        initialValues={{ username: "", password: "" }}
        validationSchema={validationSchema}
        onSubmit={handleSubmit}
      >
        {({ isSubmitting }) => (
          <Form>
            <div style={{ marginBottom: "10px" }}>
              <label htmlFor="username">用户名:</label>
              <Field type="text" name="username" placeholder="输入用户名" />
              <ErrorMessage name="username" component="div" style={{ color:
"red" }} />
```

```
                </div>
                <div style={{ marginBottom: "10px" }}>
                  <label htmlFor="password">密码:</label>
                  <Field type="password" name="password" placeholder="输入密码"
/>
                  <ErrorMessage name="password" component="div" style={{ color:
"red" }} />
                </div>
                <button type="submit" disabled={isSubmitting}>
                  {isSubmitting ? "提交中..." : "登录"}
                </button>
              </Form>
            )}
          </Formik>
        </div>
    );
};

export default LoginPage;
```

2.2 登录页面交互逻辑

1. **输入校验：** 使用 Yup 定义表单验证规则，确保用户名和密码不能为空。
2. **提交表单：**

   表单提交后，向后端 /api/token/ API 发送 POST 请求，包含用户名和密码。
3. **存储 JWT：**

   后端成功返回令牌后，将 access_token 和 refresh_token 存储到 localStorage 供后续 API 请求使用。
4. **登录失败提示：**

若后端返回错误，则显示“登录失败”提示信息。

2.3 配置路由

文件路径：src/App.js

```
import React from "react";
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import LoginPage from "./pages/LoginPage";

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<LoginPage />} />
        <Route path="/tasks" element={<div>任务列表页面</div>} />
      </Routes>
    </Router>
  );
}

export default App;
```

---

3. 测试步骤

1. **启动前端项目**

```
npm start
```

访问 http://localhost:3000，确认登录页面显示正常。

2. **模拟登录**

   输入正确或错误的用户名与密码，点击“登录”，验证表单校验、错误提示和后端交互是否正常。

3. **检查登录状态**

使用浏览器开发者工具检查 localStorage，确保成功存储 access_token 和 refresh_token。

---

4. 页面效果

- **成功登录**：跳转到 /tasks 页面。
- **登录失败**：显示“登录失败，请检查用户名和密码”的错误提示信息。

此时，登录页面的功能开发完成，可以进一步完善样式或与后端调试交互。

4.4.3 创建任务列表页面

---

1. 功能目标

任务列表页面用于展示用户所有的任务，支持按任务状态筛选，用户可以点击任务状态切换按钮更新任务的状态。

---

2. 主要代码实现

2.1 任务列表页面组件

文件路径：src/pages/TaskListPage.js

```
import React, { useEffect, useState } from "react";
import axios from "axios";
import TaskItem from "../components/TaskItem"; // 任务项组件
import TaskFilter from "../components/TaskFilter"; // 筛选器组件

const TaskListPage = () => {
  const [tasks, setTasks] = useState([]);
  const [loading, setLoading] = useState(true);
  const [filter, setFilter] = useState({ status: "", priority: "" });
```

```
// 获取任务列表
const fetchTasks = async () => {
  try {
    const response = await axios.get("http://localhost:8000/api/tasks/", {
      headers: {
        Authorization: `Bearer ${localStorage.getItem("access_token")}`,
      },
      params: {
        status: filter.status,   // 根据状态筛选
        priority: filter.priority, // 根据优先级筛选
      },
    });
    setTasks(response.data);
  } catch (err) {
    console.error("任务列表加载失败:", err);
  } finally {
    setLoading(false);
  }
};

useEffect(() => {
  fetchTasks();
}, [filter]); // 当筛选条件变化时重新加载任务

return (
  <div style={{ padding: "20px" }}>
    <h1>任务列表</h1>

    {/* 筛选器组件 */}
    <TaskFilter filter={filter} setFilter={setFilter} />
```

```
        {/* 加载中提示 */}
        {loading ? (
          <p>加载中...</p>
        ) : (
          <div>
            {/* 如果没有任务 */}
            {tasks.length === 0 ? (
              <p>没有任务</p>
            ) : (
              <div>
                {tasks.map((task) => (
                  <TaskItem key={task.id} task={task} />
                ))}
              </div>
            )}
          </div>
        )}
      </div>
    );
};

export default TaskListPage;
```

说明：

- fetchTasks 函数用于从后端 API 获取任务数据。
- 使用 filter 状态存储筛选条件，筛选任务列表。
- useEffect 用于在组件加载或筛选条件变化时重新调用 API 获取任务数据。

2.2 任务列表项组件

文件路径：src/components/TaskItem.js

```
import React, { useState } from "react";
import axios from "axios";

const TaskItem = ({ task }) => {
  const [status, setStatus] = useState(task.status);

  // 切换任务状态
  const toggleStatus = async () => {
    try {
      const updatedStatus = status === "未完成" ? "已完成" : "未完成";
      await axios.patch(
        `http://localhost:8000/api/tasks/${task.id}/`,
        { status: updatedStatus },
        {
          headers: {
            Authorization: `Bearer ${localStorage.getItem("access_token")}`,
          },
        }
      );
      setStatus(updatedStatus);
    } catch (err) {
      console.error("更新任务状态失败:", err);
    }
  };

  return (
    <div style={{ marginBottom: "15px", padding: "10px", border: "1px solid
#ddd" }}>
```

```
        <h3>{task.title}</h3>
        <p>{task.description}</p>
        <p>优先级: {task.priority}</p>
        <button onClick={toggleStatus}>
          {status === "未完成" ? "标记为已完成" : "标记为未完成"}
        </button>
      </div>
  );
};

export default TaskItem;
```

说明：

- toggleStatus 函数用于更新任务的状态（未完成/已完成）。点击按钮时，会发送 PATCH 请求更新任务状态，并在界面上切换按钮文本。

2.3 筛选器组件

文件路径：src/components/TaskFilter.js

```
import React from "react";

const TaskFilter = ({ filter, setFilter }) => {
  // 更新筛选条件
  const handleStatusChange = (e) => {
    setFilter({ ...filter, status: e.target.value });
  };

  const handlePriorityChange = (e) => {
    setFilter({ ...filter, priority: e.target.value });
  };

  return (
```

```
      <div style={{ marginBottom: "20px" }}>
        <label>
          状态：
          <select value={filter.status} onChange={handleStatusChange}>
            <option value="">全部</option>
            <option value="未完成">未完成</option>
            <option value="已完成">已完成</option>
          </select>
        </label>

        <label style={{ marginLeft: "10px" }}>
          优先级：
          <select value={filter.priority} onChange={handlePriorityChange}>
            <option value="">全部</option>
            <option value="高">高</option>
            <option value="中">中</option>
            <option value="低">低</option>
          </select>
        </label>
      </div>
    );
};

export default TaskFilter;
```

说明：

- TaskFilter 组件允许用户选择任务的状态和优先级进行筛选。
- 筛选器组件更新筛选条件时，会调用 setFilter 来更新父组件（TaskListPage）的筛选状态，触发任务列表的重新加载。

3. 测试步骤

1. **启动前端项目**

npm start

访问 http://localhost:3000，确认任务列表页面正确加载。

2. **任务加载测试**

   验证页面是否能正确显示任务列表。确保任务按照后端返回的数据渲染。

3. **筛选功能测试**

   - 使用筛选器选择不同的任务状态和优先级，检查任务列表是否能根据筛选条件正确更新。
   - 验证当没有匹配的任务时，页面显示“没有任务”。

4. **任务状态切换测试**

   - 点击任务列表中的“标记为已完成”或“标记为未完成”按钮，确认任务状态能正确更新并与后端同步。

---

4. 页面效果

- **任务列表显示**：展示所有任务，并根据筛选条件（任务状态、优先级）动态更新任务列表。
- **任务状态切换**：点击任务状态按钮后，任务状态在前端和后端同步更新。
- **筛选器功能**：用户可选择状态和优先级筛选任务，任务列表根据选择动态更新。

通过以上开发，任务列表页面已具备任务展示、筛选和状态更新等功能。

4.4.4 创建任务编辑页面

---

1. 功能目标

任务编辑页面用于创建新任务或编辑现有任务，用户可以输入任务标题、描述、状态和优先级，并通过提交按钮调用后端 API 完成任务的创建或更新。

---

2. 主要代码实现

2.1 任务编辑页面组件

文件路径：src/pages/TaskEditPage.js

```
import React, { useEffect, useState } from "react";
import { useNavigate, useParams } from "react-router-dom";
import axios from "axios";

const TaskEditPage = () => {
  const { taskId } = useParams(); // 获取任务 ID（用于区分编辑和新增）
  const navigate = useNavigate();

  const [taskData, setTaskData] = useState({
    title: "",
    description: "",
    status: "未完成",
    priority: "中",
  });

  const [loading, setLoading] = useState(false);
  const [error, setError] = useState("");

  // 获取现有任务数据（编辑模式）
  useEffect(() => {
    if (taskId) {
      const fetchTask = async () => {
        try {
          setLoading(true);
          const response = await
axios.get(`http://localhost:8000/api/tasks/${taskId}/`, {
            headers: {
```

```
                Authorization: `Bearer ${localStorage.getItem("access_token")}`,
              },
            });
            setTaskData(response.data);
          } catch (err) {
            setError("加载任务数据失败");
          } finally {
            setLoading(false);
          }
        };
        fetchTask();
    }
}, [taskId]);

// 提交表单（新增或更新任务）
const handleSubmit = async (e) => {
    e.preventDefault();
    try {
        setLoading(true);
        if (taskId) {
            // 更新任务
            await axios.put(`http://localhost:8000/api/tasks/${taskId}/`, taskData, {
                headers: {
                    Authorization: `Bearer ${localStorage.getItem("access_token")}`,
                },
            });
        } else {
            // 新增任务
            await axios.post(`http://localhost:8000/api/tasks/`, taskData, {
                headers: {
```

```
                Authorization: `Bearer ${localStorage.getItem("access_token")}`,
            },
          });
      }
      navigate("/tasks"); // 提交后跳转到任务列表页面
    } catch (err) {
      setError("提交任务失败");
    } finally {
      setLoading(false);
    }
};

// 表单字段处理
const handleChange = (e) => {
    const { name, value } = e.target;
    setTaskData({ ...taskData, [name]: value });
};

return (
    <div style={{ padding: "20px" }}>
      <h1>{taskId ? "编辑任务" : "创建任务"}</h1>
      {loading ? (
        <p>加载中...</p>
      ) : error ? (
        <p style={{ color: "red" }}>{error}</p>
      ) : (
        <form onSubmit={handleSubmit}>
          <div>
            <label>标题：</label>
            <input
```

```
                type="text"
                name="title"
                value={taskData.title}
                onChange={handleChange}
                required
              />
            </div>
            <div>
              <label>描述：</label>
              <textarea
                name="description"
                value={taskData.description}
                onChange={handleChange}
                required
              />
            </div>
            <div>
              <label>状态：</label>
              <select name="status" value={taskData.status}
onChange={handleChange}>
                <option value="未完成">未完成</option>
                <option value="已完成">已完成</option>
              </select>
            </div>
            <div>
              <label>优先级：</label>
              <select name="priority" value={taskData.priority}
onChange={handleChange}>
                <option value="高">高</option>
```

```
                    <option value="中">中</option>
                    <option value="低">低</option>
                  </select>
                </div>
                <button type="submit" style={{ marginTop: "20px" }}>
                  提交
                </button>
              </form>
            )}
          </div>
        );
};

export default TaskEditPage;
```

---

3. 交互逻辑

1. **新增任务：**
   - 如果 taskId 不存在，页面以新增任务模式加载。
   - 用户填写表单内容后，点击提交调用后端 **POST API** 创建任务。
2. **编辑任务：**
   - 如果 taskId 存在，页面以编辑任务模式加载。
   - 页面加载时调用后端 **GET API** 获取任务数据，渲染到表单中。
   - 修改表单内容后，点击提交调用后端 **PUT API** 更新任务。
3. **通用处理：**
   - 使用 handleChange 方法动态更新表单内容。
   - 表单提交后跳转回任务列表页面。

---

4. 页面效果

- **新增任务页面：**

- 用户可填写任务标题、描述、状态和优先级。
- 提交后，任务被成功创建并保存到后端。

- **编辑任务页面：**
  - 页面加载时显示已有任务信息。
  - 用户可更新任务的字段，提交后完成任务信息的更新。

---

5. 测试步骤

1. **新增任务测试：**
   - 访问 http://localhost:3000/tasks/new。
   - 填写任务信息，点击提交按钮。
   - 验证任务是否成功添加，并在任务列表页面显示。
2. **编辑任务测试：**
   - 访问 http://localhost:3000/tasks/:taskId（替换 :taskId 为实际任务 ID）。
   - 验证任务数据是否正确加载到表单。
   - 修改任务信息后提交，检查任务更新是否生效。
3. **错误处理测试：**
   - 测试网络异常或后端错误，页面是否正确显示错误消息。

通过以上实现与测试，任务编辑页面完成了任务的新增与编辑功能，并与后端任务管理 API 完美对接。

4.4.5 前端路由配置（需要继续优化）

1. 配置前端路由

在 React 中，我们需要使用 BrowserRouter 包裹整个应用程序，Route 来定义页面路由，Link 或 NavLink 来实现页面导航。我们还需要确保私有路由被保护，仅允许已登录用户访问。

---

2. 路由配置实现

文件路径：src/App.js

```
import React, { useEffect, useState } from "react";
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import LoginPage from "./pages/LoginPage";
import TaskListPage from "./pages/TaskListPage";
import TaskEditPage from "./pages/TaskEditPage";
import PrivateRoute from "./components/PrivateRoute"; // 私有路由组件

const App = () => {
  const [isAuthenticated, setIsAuthenticated] = useState(false);

  // 检查本地存储中的 JWT 是否有效
  useEffect(() => {
    const token = localStorage.getItem("access_token");
    if (token) {
      setIsAuthenticated(true);
    }
  }, []);

  return (
    <Router>
      <Routes>
        {/* 公共路由 */}
        <Route path="/login" element={<LoginPage />} />

        {/* 私有路由 */}
        <Route
          path="/tasks"
          element={<PrivateRoute isAuthenticated={isAuthenticated}
component={TaskListPage} />}
```

```
            />
            <Route
              path="/tasks/:taskId"
              element={<PrivateRoute isAuthenticated={isAuthenticated}
component={TaskEditPage} />}
            />
          </Routes>
        </Router>
    );
};

export default App;
```

---

4. 私有路由组件

PrivateRoute 组件负责检查用户是否登录，未登录的用户将被重定向到登录页。

文件路径：src/components/PrivateRoute.js

jsx

复制代码

```
import React from "react";
import { Navigate } from "react-router-dom";

// 私有路由组件，检查用户是否已认证
const PrivateRoute = ({ isAuthenticated, component: Component }) => {
  return isAuthenticated ? <Component /> : <Navigate to="/login" />;
};

export default PrivateRoute;
```

---

5. 解释与功能

1. **公共路由（登录页）：**
    - 登录页面是公共页面，不需要身份验证。用户可以通过访问 /login 路径来进行登录。
    - 登录成功后，存储 JWT 令牌到 localStorage，并重定向到任务列表页面。
2. **私有路由（任务列表页和任务编辑页）：**
    - 任务列表页面 /tasks 和任务编辑页面 /tasks/:taskId 是需要用户登录后才能访问的页面。
    - 通过 PrivateRoute 组件，检查 isAuthenticated 状态来判断用户是否已登录。如果已登录，渲染相应的页面；如果未登录，用户将被重定向到 /login 页面。
3. **PrivateRoute 组件：**
    - PrivateRoute 组件接受两个 props：
        - isAuthenticated：表示用户是否已认证。
        - component：需要渲染的组件（如任务列表页或任务编辑页）。
    - 如果用户未登录，PrivateRoute 会重定向到登录页面；如果已登录，渲染对应的组件。

---

6. 测试路由配置

1. **登录验证测试：**
    - 如果用户未登录，访问 /tasks 或 /tasks/:taskId 应跳转到 /login 页面。
    - 用户登录后，再访问 /tasks 或 /tasks/:taskId 应该能够成功访问对应页面。
2. **路由导航测试：**
    - 确保在登录页面点击登录按钮后，成功跳转到任务列表页面。
    - 在任务列表页面点击任务详情，应该跳转到任务编辑页面。
3. **路由保护测试：**
    - 在浏览器的开发者工具中清除 localStorage 中的 JWT，然后尝试

访问 /tasks 或 /tasks/:taskId，应该被重定向到 /login 页面。

通过这个路由配置，我们确保了任务管理系统的安全性，只有登录的用户才能访问任务相关的功能，同时也提供了简洁的导航体验。

4.5 功能测试

功能测试旨在验证项目的核心功能是否实现并满足预期，包括用户注册与登录、任务的增删查改等功能。以下是详细的测试内容和测试流程：

4.5.1 测试环境

1. **后端测试环境**：启动 Django 开发服务器，确保所有 API 可用。
    - 启动命令：python manage.py runserver。
2. **前端测试环境**：启动 React 开发服务器。
    - 启动命令：npm start。
3. **数据库**：使用 MySQL 数据库，确保包含必要的测试数据。

4.5.2 测试工具

- **curl**：用于测试 REST API。
- **浏览器**：用于测试前端界面功能。
- **自动化测试工具（可选）**：如 Jest（前端）、Django TestCase（后端）。

4.5.3 测试功能清单

1. **用户认证**

- 用户注册
- 用户登录
- 用户身份验证（JWT 令牌）

2. **任务管理**
   - 任务创建
   - 获取任务列表
   - 任务更新
   - 任务删除
   - 筛选任务（按状态、优先级）

4.5.4 测试步骤与预期结果

1. 用户认证测试

1.1 **注册用户**

- **测试步骤**：通过前端登录页面填写用户名、密码等信息并提交表单。
- **预期结果**：注册成功后，返回 201 状态码，用户数据正确存储在数据库中。

1.2 **登录用户**

- **测试步骤**：通过前端登录页面填写用户名和密码，点击“登录”，或调用 /api/login/ API。
- **预期结果**：登录成功后，返回 200 状态码，并返回 JWT 令牌。

1.3 **验证用户身份**

- **测试步骤**：使用获取的 JWT 令牌调用受保护的 API（如 /api/tasks/）。
- **预期结果**：若令牌有效，返回 200 状态码及相关数据；若令牌无效，返回 401 状态码。

---

2. 任务管理测试 2.1 **创建任务**

- **测试步骤**：在任务编辑页面输入任务标题、描述等信息，提交表单，或调用 /api/tasks/ 的 POST 方法。

- **预期结果**：返回 201 状态码，任务数据正确存储在数据库中。

2.2 **获取任务列表**

- **测试步骤**：进入任务列表页面，验证所有任务是否正确显示，或调用 /api/tasks/ 的 GET 方法。
- **预期结果**：返回 200 状态码，任务列表正确加载。

2.3 **更新任务**

- **测试步骤**：在任务编辑页面修改任务信息后提交，或调用 /api/tasks/:id/ 的 PUT 或 PATCH 方法。
- **预期结果**：返回 200 状态码，数据库中任务信息更新成功。

2.4 **删除任务**

- **测试步骤**：在任务列表页面点击删除按钮，或调用 /api/tasks/:id/ 的 DELETE 方法。
- **预期结果**：返回 204 状态码，任务从数据库中删除。

2.5 **筛选任务**

- **测试步骤**：在任务列表页面使用筛选功能按状态或优先级筛选任务，或通过 /api/tasks/ API 添加 status 和 priority 参数。
- **预期结果**：返回符合筛选条件的任务列表。

4.4.5 测试记录示例

功能模块	测试用例	测试步骤	预期结果	实际结果	备注
用户认证	注册新用户	调用 /api/register/ 接口提交用户信息	返回 201 状态码，用户注册成功	与预期一致	-
用户认证	登录用户	调用 /api/login/ 接口，提交用户名和密码	返回 200 状态码，返回 JWT 令牌	与预期一致	-
任务管理	创建任务	调用 /api/tasks/ 的 POST 方法，提交任务信息	返回 201 状态码，任务存储成功	与预期一致	-
任务管理	更新任务	调用 /api/tasks/:id/ 的 PATCH 方法	返回 200 状态码，任务信息更新成功	与预期一致	-

功能模块	测试用例	测试步骤	预期结果	实际结果	备注
任务管理	删除任务	调用 /api/tasks/:id/ 的 DELETE 方法	返回 204 状态码，任务删除成功	与预期一致	-
任务管理	筛选任务	调用 /api/tasks/ 的 GET 方法，带 status 参数	返回符合筛选条件的任务列表	与预期一致	-

4.5.5 测试结果总结

- 测试过程中所有功能均按照预期实现，API 返回状态码和数据均符合要求。
- 任务筛选、用户登录、任务增删改等核心功能正常。
- 无显著异常，项目功能满足基本需求，可进行进一步优化和扩展。

通过以上测试内容的详细执行，可以确保项目的核心功能正常运行，为项目上线或交付提供有力保障。

4.6 项目部署（这步只需要在服务器上完成）

4.6.1 使用 Gunicorn 启动后端服务

1. 安装 Gunicorn

- **使用 pip 安装 Gunicorn**

  在后端虚拟环境中执行以下命令安装 Gunicorn：

  pip install gunicorn

- **验证安装成功**

  安装完成后，运行以下命令检查版本号，确保安装成功：

  gunicorn --version

  示例输出：

gunicorn (version 20.1.0)

2. 启动服务

- **使用 Gunicorn 启动 Django 项目**

  在 Django 项目根目录下，使用以下命令启动服务：

  gunicorn --bind 0.0.0.0:8000 项目名.wsgi:application

其中：

  - --bind 指定 Gunicorn 监听的地址和端口（如 0.0.0.0:8000）。
  - 项目名.wsgi:application 是 Django 项目的 WSGI 应用入口。

- **检查服务启动日志**

  启动后检查终端日志输出是否有错误信息，如无错误，访问服务器 IP 和端口验证服务是否正常运行。

3. 创建 Systemd 服务文件

为了使 Gunicorn 在后台运行并自动启动，创建 Systemd 服务文件：

- **创建 Gunicorn 服务文件**

  使用以下命令创建并编辑服务文件：

  sudo vim /etc/systemd/system/gunicorn.service

添加以下内容：

[Unit]

Description=gunicorn daemon

After=network.target

[Service]

User=用户名

Group=组名

WorkingDirectory=/path/to/项目根目录

ExecStart=/path/to/ 虚 拟 环 境 /bin/gunicorn --workers 3 --bind unix:/path/to/socket/gunicorn.sock 项目名.wsgi:application

[Install]

WantedBy=multi-user.target

注意：

- 替换 用户名 和 组名 为运行 Gunicorn 服务的用户和用户组（如 ubuntu）。
- 替换 /path/to/虚拟环境 为虚拟环境路径。
- 替换 /path/to/项目根目录 为 Django 项目根目录路径。
- 替换 /path/to/socket/gunicorn.sock 为 Unix 套接字文件路径。

注：需要注意用户对于文件的权限，设置为可读

sudo chown -R tushiwen:tushiwen /var/run/socket

- **启用并启动服务**

  执行以下命令启用并启动 Gunicorn 服务：

sudo systemctl daemon-reload

sudo systemctl start gunicorn

sudo systemctl enable gunicorn

- **检查服务状态**

  使用以下命令检查 Gunicorn 服务是否正常运行：

  sudo systemctl status gunicorn

输出示例（部分内容）：

Active: active (running)

完成以上步骤后，Gunicorn 应成功启动并正常运行 Django 项目。

注：系统服务的文件的简单说明

这段 gunicorn.service 文件是用于配置和管理 Gunicorn 服务的 systemd 服务文件。下面是每一段内容的详细解释：

[Unit] 部分

这个部分定义了服务的基本描述和依赖关系。

[Unit]

Description=gunicorn daemon

After=network.target

- **Description**: 这项描述服务的作用或功能，通常是一个简短的文字描述，

用于帮助用户理解服务的功能。

- 这里的 gunicorn daemon 表示这是一个 Gunicorn 守护进程。
- 在一个多工的电脑操作系统中，**守护进程**（英语：daemon，/ˈdiːmən/或/ˈdeɪmən/）[2]是一种在后台执行，而不由用户直接交互控制的电脑程序。此类程序会被以进程的形式初始化。

- **After**: 定义服务的启动顺序，表示此服务依赖于 network.target（即网络相关服务）在启动后启动。这确保 Gunicorn 在网络服务启动后才启动。
  - network.target 是 systemd 中的一个目标，表示网络已就绪，通常它会在系统启动过程中启动，因此 Gunicorn 会在网络服务启动后启动。

[Service] 部分

这个部分定义了具体服务的行为，包括执行的命令、运行的环境、重启策略等。

```
[Service]
User=tushiwen
Group=tushiwen
WorkingDirectory=/home/tushiwen/SE/backend/
ExecStart=/home/tushiwen/miniconda3/envs/task-manager/bin/gunicorn --workers 3 --bind unix:/var/run/socket/gunicorn.sock task_manager.wsgi:application
```

- **User**: 定义服务将作为哪个用户运行。
  - 这里 tushiwen 是执行 Gunicorn 服务的用户。这样可以确保 Gunicorn 进程没有管理员权限，从而提升安全性。
- **Group**: 定义服务将作为哪个用户组运行。
  - 这里 tushiwen 是执行 Gunicorn 服务的用户组。它应与 User 保持一致，通常服务进程运行在与其用户同名的组下。
- **WorkingDirectory**: 设置服务的工作目录。
  - 这个目录是 Gunicorn 启动时会进入的目录。在这里设置为 /home/tushiwen/SE/backend/，即你的 Django 项目的根目录。Gunicorn 会在这个目录下启动，通常这个目录包含你的应用代码和 WSGI 文件（在本例中为 task_manager.wsgi:application）。

- **ExecStart**: 定义启动 Gunicorn 的命令。
    - ExecStart 后的命令是启动 Gunicorn 守护进程的命令。在这里，指定了 Gunicorn 执行文件的路径和启动参数。
        - /home/tushiwen/miniconda3/envs/task-manager/bin/gunicorn: 这是 Gunicorn 可执行文件的路径，使用了虚拟环境中的 gunicorn，保证运行的是当前虚拟环境中的 Gunicorn 版本。
        - --workers 3: 启动 3 个 Gunicorn worker 进程，允许处理多个请求。workers 参数决定了启动多少个工作进程来处理请求。根据服务器性能和流量，workers 数量可以进行调整。
        - --bind unix:/var/run/socket/gunicorn.sock: 将 Gunicorn 绑定到 Unix 套接字文件 /var/run/socket/gunicorn.sock，而不是默认的 HTTP 端口。使用 Unix 套接字可以提高性能，通常用于和 Nginx 配合。
        - task_manager.wsgi:application: 指定 Gunicorn 启动应用的 WSGI 应用对象。在这里，task_manager.wsgi:application 表示 task_manager 目录下的 wsgi.py 文件中的 application 对象，这是 Django 项目默认的 WSGI 应用对象。

[Install] 部分

这个部分定义了如何安装和启用该服务。

[Install]

WantedBy=multi-user.target

- **WantedBy**: 定义了服务的目标启动级别。
    - multi-user.target 表示此服务将在系统进入多用户模式时启动。这个模式通常表示系统已经启动并准备好为多个用户提供服务，通常用于服务器和非图形界面的运行模式。

总结

- **[Unit]**: 提供服务的描述，并指定 Gunicorn 服务在网络服务启动后才启动。
- **[Service]**: 具体的服务配置，包括执行命令、用户和组权限、工作目录、

启动命令等。

- **[Install]**: 定义服务如何与 systemd 的目标（multi-user.target）关联，使得服务在合适的时机启动。

这个 gunicorn.service 文件的作用就是将 Gunicorn 配置为一个 systemd 管理的服务，使其可以自动启动、停止和重启，并确保在正确的时机运行。

4.6.2 配置 Nginx

1. 安装 Nginx

- **使用系统包管理器安装 Nginx**

  在 Ubuntu 系统中，运行以下命令安装 Nginx：

```
sudo apt update
sudo apt install nginx
```

- **验证 Nginx 安装成功**

  安装完成后，启动 Nginx 并验证服务状态：

```
sudo systemctl start nginx
sudo systemctl enable nginx
sudo systemctl status nginx
```

示例输出（部分内容）：

```
Active: active (running)
```

打开浏览器访问服务器 IP，若显示默认欢迎页面，表示安装成功。

2. 创建 Nginx 配置文件

- **定义服务器块**

  创建 Nginx 配置文件，用于配置反向代理和静态文件路径：

```
sudo nano /etc/nginx/sites-available/项目名
```

添加以下内容：

对于 **前端部分**，即如何配置和部署 React 应用与 Nginx 的结合，可以按照以下

步骤来实现。这里将详细讲解如何将 **React 前端应用** 部署到服务器，并使其通过 Nginx 提供服务。

1. 构建 React 应用

首先，假设你已经使用 create-react-app 创建了一个 React 应用，并在本地开发过程中完成了前端开发。接下来，我们需要构建该 React 应用，以便将其部署到服务器上。

在 React 项目根目录下，运行以下命令来构建生产版本的应用（这步最好在开发环境运行，比较消耗 cpu，单核 cpu 云服务器可能会卡死）：

npm run build

该命令会将 React 应用构建为一个优化过的静态文件，通常输出在 build/ 目录中。你可以查看该目录，它包含了以下文件：

- index.html：应用的主 HTML 文件。
- static/：包含了 JavaScript、CSS 和图片等静态文件。

2. 将构建好的文件上传到服务器

将 build/ 目录中的文件上传到你服务器上的一个目录。假设我们将其上传到 /var/www/myproject/frontend/ 目录。

你可以使用 SCP 或其他方式上传这些文件(我是用的是 git，推荐在开发环境下把部署文件上传到 github 供服务器拉取)：

scp -r build/* user@server:/var/www/myproject/frontend/

确保这些文件的权限和路径正确，以便 Nginx 能够访问。

3. 配置 Nginx 服务前端资源

接下来，我们需要修改 Nginx 配置，以便让 Nginx 提供 React 应用的静态文件。假设你已经将 React 构建文件放置在 /var/www/myproject/frontend/ 目录，修改 Nginx 配置文件来处理 React 应用的静态文件和路由。

修改 Nginx 配置文件，确保 Nginx 处理 / 路径的请求并返回 React 的静态文件，同时处理所有静态资源（如 JS、CSS、图片等）。

完整的 Nginx 配置文件（前端与后端结合）

```
server {
    listen 80;
```

```
    server_name 13.229.129.242;  # 或者你的域名

    # React 前端请求代理到 Gunicorn 后端
    location /api/ {
        proxy_pass http://unix:/var/run/socket/gunicorn.sock;  # Gunicorn 的 Unix socket 地址
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    # 静态文件目录配置
    location /static/ {
        alias /var/www/myproject/frontend/static/;  # 将 React 静态资源映射到静态文件目录
        expires 1y;  # 缓存策略
        add_header Cache-Control "public, max-age=31536000";  # 强缓存策略
    }

    # React 应用的入口 HTML 文件
    location / {
        root /var/www/myproject/frontend/;  # 指向构建后的 React 应用目录
        index index.html;  # 默认的 index.html 文件
        try_files $uri /index.html;  # 如果 URL 不匹配静态文件，则返回 index.html（支持 React Router）
    }

    # 媒体文件目录配置（如果有）
    location /media/ {
```

```
        alias /var/www/myproject/media/;
        expires 1y;
        add_header Cache-Control "public, max-age=31536000";
    }

    # 配置错误页面
    error_page 404 /404.html;
    error_page 500 502 503 504 /50x.html;

    location = /404.html {
        root /usr/share/nginx/html;
    }

    location = /50x.html {
        root /usr/share/nginx/html;
    }
}
```

配置解释：

1. **/api/ 代理：**
    - 该部分配置用于将 API 请求（如 /api/ 路径下的请求）代理到 Django 后端，Django 通过 Gunicorn 提供服务。这部分配置和之前的配置是一样的，保证了后端请求能够正确地传递到 Django。
2. **前端静态文件：**
    - location /static/ 配置了 React 的静态文件目录。React 构建后会将静态文件存放在 build/static/ 目录下，我们将其映射到 /static/ 路径上。
    - expires 1y 和 Cache-Control 头部用于启用静态资源的缓存，提高加载速度。
3. **前端入口页面：**

- location / 用于处理前端的所有请求，将 index.html 文件作为入口文件。try_files $uri /index.html; 的作用是，如果请求的路径没有对应的静态文件（例如通过 React Router 导航到不同的页面），Nginx 会返回 index.html 文件，这样 React Router 就可以接管路由控制。

4. **/media/ 目录：**
   - 如果你有需要提供的媒体文件，可以通过类似的方式进行配置，将媒体文件路径指向 /var/www/myproject/media/，并为其添加缓存策略。
5. **错误页面：**
   - 配置了 404.html 和 50x.html 错误页面，这样用户在访问无效路径或服务器错误时可以看到自定义的错误页面。

4. 启动和重载 Nginx 配置

完成配置文件修改后，检查 Nginx 配置是否正确并重载服务：

```
sudo nginx -t     # 检查配置是否有效
sudo systemctl reload nginx     # 重载 Nginx 服务
```

5. 验证和调试

- 现在，Nginx 会提供静态的 React 应用文件，并且通过 try_files 指令可以正确处理 React Router 的客户端路由。
- 使用浏览器访问你的网站，检查是否能正确加载 React 前端，并且是否能与 Django 后端交互。

小结

- React 前端和 Django 后端通过 Nginx 协同工作，React 应用的静态文件通过 Nginx 提供，API 请求通过 Nginx 代理到 Django 后端。
- 通过 try_files 和静态资源缓存，Nginx 能够高效地提供 React 应用的资源，并且支持单页面应用（SPA）的路由功能。

- **启用新配置**

  创建符号链接以启用配置：

```
sudo ln -s /etc/nginx/sites-available/项目名 /etc/nginx/sites-enabled/
sudo rm /etc/nginx/sites-enabled/default
```

3. 测试并启用配置

- **测试配置文件**

  使用以下命令检查配置文件是否正确：

  sudo nginx -t

示例输出：

  nginx: configuration file /etc/nginx/nginx.conf test is successful

- **重新加载 Nginx 服务**

  如果配置测试成功，重新加载 Nginx 服务以应用更改：

  sudo systemctl reload nginx

- **验证配置**

  在浏览器中访问服务器的域名或 IP 地址，确保请求能正常转发至 Gunicorn，静态文件能够正确加载。如果一切正常，Nginx 配置完成。

4.6.3 测试部署环境

1. 后端功能测试

1.1 使用 Postman 测试主要 API 接口

- 打开 Postman，逐一测试后端提供的 API 接口：
  - **用户注册**（POST /api/register/）：发送 JSON 格式的用户数据，检查是否能成功创建用户。
  - **用户登录**（POST /api/login/）：发送正确的用户名和密码，验证是否能够获取 JWT 令牌。
  - **任务列表获取**（GET /api/tasks/）：使用获取的 JWT 令牌测试任务列表接口，验证是否返回任务数据。
  - **任务创建**（POST /api/tasks/）：提交新任务数据，验证是否能成功添加任务。
  - **任务更新**（PUT /api/tasks/<id>/）：发送更新后的任务信息，验证是否能成功修改任务。
  - **任务删除**（DELETE /api/tasks/<id>/）：调用删除接口，验证是否能成功删除指定任务。

1.2 使用 cURL 测试 API 接口

- 在终端中使用以下命令测试 API 功能（以任务列表为例）：

curl -X GET http://<server-ip>/api/tasks/ \

-H "Authorization: Bearer <your-access-token>"

替换 <server-ip> 和 <your-access-token>，检查是否返回正确的任务数据。

2. 前端功能测试

2.1 访问前端页面

- 在浏览器中访问部署的前端地址（如 http://<server-ip>）。
- 验证以下模块的功能是否正常：
    - **登录页面**：输入用户名和密码，验证是否能成功登录并跳转至任务列表页面。
    - **任务列表页面**：检查任务是否正确显示，并测试筛选器功能。
    - **任务编辑页面**：尝试新增和编辑任务，验证是否能正常保存更改。

2.2 检查前后端交互

- 观察浏览器开发者工具中的网络请求（Network），确认前端请求后端 API 是否成功，并检查返回的数据是否符合预期。

3. 压力测试

3.1 安装 Apache Benchmark（ab）

- 使用以下命令安装 Apache Benchmark 工具：

sudo apt install apache2-utils

3.2 模拟多用户访问

- 对后端 API 接口进行压力测试（以任务列表为例）：

ab -n 1000 -c 50 http://<server-ip>/api/tasks/

参数说明：

- -n 1000：模拟 1000 次请求。
- -c 50：并发 50 个用户。

3.3 记录测试结果

- 检查 ab 命令的输出结果，关注以下指标：
    - **请求失败率**：确认请求是否成功响应。

- **响应时间**：检测系统是否在高并发下保持良好的响应性能。

通过上述步骤，验证系统部署后的功能完整性和性能稳定性。如果发现问题，需记录日志并进行优化。

注：压力测试的简单说明

上面是使用 ApacheBench (ab) 工具对本地部署的服务进行压力测试的结果。以下是对结果中每一段内容的详细解读：

基本信息

Server Software: nginx/1.24.0

Server Hostname: localhost

Server Port: 8000

- **Server Software**: 服务器使用的 HTTP 软件，当前为 nginx/1.24.0，表明请求是通过 Nginx 处理的。
- **Server Hostname**: 测试的目标主机名，这里是 localhost。
- **Server Port**: 测试的目标端口号，这里是 8000。

这些信息表明测试目标是运行在本地 localhost:8000 上的服务，Nginx 作为前端代理。

请求和响应信息

Document Path: /api/tasks/

Document Length: 58 bytes

- **Document Path**: 测试的 URL 路径，这里是 /api/tasks/。
- **Document Length**: 服务器返回的每个响应的正文大小为 58 字节。这个值通常是响应的 HTML 或 JSON 内容的大小。

测试参数

Concurrency Level: 50

Time taken for tests: 0.478 seconds

Complete requests: 1000

Failed requests: 0

Non-2xx responses: 1000

Total transferred: 446000 bytes

HTML transferred: 58000 bytes

Requests per second: 2093.96 [#/sec] (mean)

Time per request: 23.878 [ms] (mean)

Time per request: 0.478 [ms] (mean, across all concurrent requests)

Transfer rate: 912.02 [Kbytes/sec] received

- **Concurrency Level**: 并发级别，即同时发送的请求数。这里是 50，表示模拟了 50 个用户同时请求。
- **Time taken for tests**: 测试的总耗时为 0.478 秒。
- **Complete requests**: 总共完成了 1000 次请求。
- **Failed requests**: 失败的请求数，这里是 0，表示所有请求都成功处理。
- **Non-2xx responses**: 状态码不是 2xx（如 200）的响应数。这里是 1000，可能是因为 API 返回了重定向、错误页面或其他非 2xx 响应。你需要检查 API 是否正确配置。
- **Total transferred**: 传输的总数据量，包括 HTTP 报头和正文，为 446000 字节。
- **HTML transferred**: 传输的正文（即响应内容）总数据量，为 58000 字节。
- **Requests per second**: 平均每秒处理的请求数为 2093.96，表示服务的吞吐量。
- **Time per request**: 平均每个请求的耗时为 23.878 毫秒。这是对单个请求的时间测量。
- **Time per request (mean, across all concurrent requests)**: 平均每个并发请求的耗时为 0.478 毫秒，这是将总耗时分摊到并发请求上的值。
- **Transfer rate**: 平均传输速率为 912.02 KB/s。

连接时间统计

Connection Times (ms)

	min	mean[+/-sd]	median	max
Connect:	0	1 1.6	0	7
Processing:	4	21 7.7	20	155
Waiting:	3	21 7.8	20	155
Total:	7	22 7.4	20	160

- **Connect**: TCP 连接建立的时间（毫秒）。
  - **min**: 最短连接时间为 0 毫秒，表示连接非常迅速。
  - **mean**: 平均连接时间为 1 毫秒。
  - **max**: 最长连接时间为 7 毫秒。
- **Processing**: 请求被服务器处理的时间（从接收请求到生成响应）。
  - **min**: 最短处理时间为 4 毫秒。
  - **mean**: 平均处理时间为 21 毫秒。
  - **max**: 最长处理时间为 155 毫秒。
- **Waiting**: 等待后端生成响应的时间（与 Processing 类似，但更专注于请求等待响应的时间）。
  - **min**, **mean**, **max** 的含义与上面类似。
- **Total**: 完整请求周期的总时间（Connect + Processing）。
  - **min**: 最短总时间为 7 毫秒。
  - **mean**: 平均总时间为 22 毫秒。
  - **max**: 最长总时间为 160 毫秒。

响应时间百分比

Percentage of the requests served within a certain time (ms)

50%	20
66%	23
75%	24
80%	26
90%	31

95% 33

98% 35

99% 38

100% 160 (longest request)

- **百分比表示某个响应时间阈值内完成的请求数量：**
  - **50%**: 一半的请求在 20 毫秒内完成。
  - **90%**: 90% 的请求在 31 毫秒内完成。
  - **100%**: 所有请求在 160 毫秒内完成，其中 160 毫秒是最长的请求时间。

此部分展示了服务器在不同负载下的响应分布情况，帮助评估服务性能。

总结

- 该测试显示服务的吞吐量很高（每秒处理约 2000 请求），请求延迟较低，响应时间分布集中在 20 毫秒附近。
- **注意事项：**
  - 非 2xx 响应表明 API 的响应不是成功状态，需要检查路径 /api/tasks/ 是否配置正确。
  - 对于长时间请求（如 160 毫秒），可能需要分析后端服务的性能瓶颈。

注：Nginx 配置文件的简单说明

这是一个典型的 Nginx 配置文件，用于将客户端请求代理到一个基于 Unix 套接字运行的 Gunicorn 服务，并提供静态文件和媒体文件的服务。下面是对每一部分的详细讲解：

---

1. server 块

server 块定义了一个虚拟主机的配置，决定如何处理发往特定域名或 IP 地址的请求。

```
server {
    listen 80;
    server_name 域名或服务器 IP;
    ...
}
```

- **listen 80**:
  - 指定 Nginx 监听的端口号，这里是 HTTP 的默认端口 80。
  - 如果你需要支持 HTTPS，可以改成 listen 443 ssl;，并配置 SSL 证书。
- **server_name 域名或服务器 IP**:
  - 定义这个虚拟主机的域名或 IP 地址。
  - 当客户端请求的 Host 头与这里的 server_name 匹配时，Nginx 将使用此块的配置处理请求。
  - 可以使用特定域名（如 example.com）或服务器的公网 IP 地址。
  - 如果你希望匹配所有请求，可以使用 _ 或 default_server。

---

2. location / 块

location / 块定义了根路径 / 的请求如何处理，通常代理到后端应用程序（如 Gunicorn）。

```
location / {
    proxy_pass http://unix:/path/to/socket/gunicorn.sock;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
}
```

- **proxy_pass http://unix:/path/to/socket/gunicorn.sock;**:
  - 将 / 路径下的所有请求代理到运行在 Unix 套接字 /path/to/socket/gunicorn.sock 上的 Gunicorn 服务。

- 这里使用 Unix 套接字而不是网络端口，具有更高的性能和安全性。

- **proxy_set_header 系列指令**:
  - 用于在代理请求时向后端服务（Gunicorn）添加或修改请求头。
  - **Host $host**:
    - 将客户端请求的主机名（域名或 IP）传递给后端，便于后端识别是针对哪个域名的请求。
  - **X-Real-IP $remote_addr**:
    - 将客户端的真实 IP 地址传递给后端，方便后端记录日志或进行 IP 限制。
  - **X-Forwarded-For $proxy_add_x_forwarded_for**:
    - 用于传递客户端的 IP 地址链条。如果请求经过多个代理服务器，这个头会包含所有中间代理的 IP 地址。
  - **X-Forwarded-Proto $scheme**:
    - 用于告知后端当前请求使用的协议（http 或 https）。如果启用了 HTTPS，这个值是 https。

---

3. location /static/ 块

用于提供静态文件（如 CSS、JS、图片等）的服务。

```
location /static/ {
    alias /path/to/static/files/;
}
```

- **location /static/**:
  - 匹配以 /static/ 开头的请求路径。
- **alias /path/to/static/files/;**:
  - 将 /static/ 请求映射到服务器上的实际文件路径 /path/to/static/files/。
  - 比如，客户端请求 /static/style.css，Nginx 会尝试读取 /path/to/static/files/style.css 并返回给客户端。

---

4. location /media/ 块

用于提供媒体文件（如用户上传的文件）的服务。

```
location /media/ {
    alias /path/to/media/files/;
}
```

- **location /media/**:
  - 匹配以 /media/ 开头的请求路径。
- **alias /path/to/media/files/;**:
  - 将 /media/ 请求映射到服务器上的实际文件路径 /path/to/media/files/。
  - 比如，客户端请求 /media/image.jpg，Nginx 会尝试读取 /path/to/media/files/image.jpg 并返回给客户端。

完整工作流程

1. **客户端请求**:
   - 客户端通过域名或 IP 地址发送 HTTP 请求。
   - 例如：http://example.com/static/style.css。
2. **Nginx 路由**:
   - 如果路径以 /static/ 开头，Nginx 会直接返回对应的静态文件。
   - 如果路径以 /media/ 开头，Nginx 会返回对应的媒体文件。
   - 其他请求（如 / 开头的路径）会被代理到 Gunicorn 服务。
3. **请求头传递**:
   - 在代理到 Gunicorn 时，Nginx 会通过 proxy_set_header 将客户端的主机名、真实 IP、协议等信息传递给后端服务，便于后端识别和处理。
4. **后端处理**:
   - Gunicorn 接收代理的请求，调用 Django 项目处理逻辑并返回响应。
5. **响应返回**:

- Nginx 将 Gunicorn 或静态文件的响应返回给客户端。

总结

这段配置文件的作用是：

1. **代理动态请求**：通过 Unix 套接字将动态请求（如 API 调用）转发到 Gunicorn。
2. **处理静态文件和媒体文件**：直接由 Nginx 提供静态资源服务，避免将这些请求交给后端处理，从而提高性能。
3. **优化代理通信**：通过 proxy_set_header 提供客户端和请求的详细信息给后端服务。
4. **实现统一服务**：将静态资源和动态请求整合到一个 Nginx 服务中，方便客户端访问。

注：网站中的静态文件和媒体文件说明

在 Django 项目中，**静态文件**和**媒体文件**分别是指两类不同的文件，它们在 Web 应用程序中扮演着非常重要的角色。为了让你更好地理解它们的定义、作用以及为什么需要它们，我们可以逐一解析这两个概念。

1. 静态文件 (Static Files)

什么是静态文件？

静态文件通常指的是在 Web 应用中不经过服务器端处理、不会变动的文件。这些文件通常由客户端（浏览器）直接请求并显示给用户。这类文件的内容不会根据用户的输入、请求或数据库的变化而变化。

常见的静态文件类型：

- **CSS 文件**：用于控制网页样式。
- **JavaScript 文件**：用于提供网页的动态功能和交互。
- **图片文件**：如 PNG、JPEG、GIF 等格式的图片文件，用于显示图像内容。
- **字体文件**：如字体图标、Web 字体等。

静态文件的作用：

静态文件主要作用是**页面的外观和交互**。它们让你的 Web 页面具有样式（CSS）、

动画和交互（JavaScript）以及图像（如 logo、背景图等）。例如：

- **CSS 文件**：定义了页面的布局、字体、颜色等。
- **JavaScript 文件**：定义了页面的动态效果、用户与页面的交互。
- **图片**：展示网站的视觉内容，例如 logo、产品图片等。

为什么需要静态文件？

1. **用户体验**：静态文件使得 Web 页面更加生动和互动，增强用户体验。例如，通过 CSS 来美化页面布局，通过 JavaScript 实现交互效果。
2. **性能优化**：静态文件通常是缓存的，这意味着浏览器可以重复使用这些文件，而无需每次都从服务器请求。这可以减少服务器负担，提高页面加载速度。

在 Django 中如何处理静态文件？

在 Django 中，静态文件是通过 STATIC_URL 和 STATICFILES_DIRS 等配置进行管理的。开发阶段，Django 会自动提供静态文件，生产环境中通常会使用 collectstatic 命令将所有静态文件集中到一个目录，然后使用 Web 服务器（如 Nginx 或 Apache）来提供这些静态文件。

2. 媒体文件 (Media Files)

什么是媒体文件？

媒体文件通常指的是**由用户上传**的文件。这些文件在 Web 应用程序中具有可变性，会根据用户的操作或请求而发生变化。与静态文件不同，媒体文件内容是**用户生成**的，可以是任何类型的文件，如图片、文档、音频、视频等。

常见的媒体文件类型：

- **图片文件**：用户上传的头像、产品图片、用户生成的照片等。
- **文档文件**：用户上传的 PDF、Word 文档、Excel 表格等。
- **音频/视频文件**：用户上传的音频、视频文件，如音乐、视频课程等。

媒体文件的作用：

媒体文件通常是 Web 应用中**与用户交互**的重要部分。它们由用户上传，代表用户在系统中的活动或数据。比如：

- **图片**：用户上传的头像、照片。
- **文档**：用户提交的表单、报告、简历等。

- **音视频**：用户上传的音频、视频内容。

为什么需要媒体文件？

1. **用户生成内容（UGC）**：很多 Web 应用会允许用户上传自己的文件，比如社交媒体、博客、文件管理系统等。媒体文件是这些功能实现的基础。
2. **数据存储和分享**：有些 Web 应用需要允许用户保存和分享大文件（如视频、图片等）。例如，在电子商务网站上，卖家可能需要上传产品图片或视频。
3. **动态内容**：用户上传的文件可能会定期更新，如在社交媒体上上传新的图片或视频，或在文档管理系统中上传新的文件。

在 Django 中如何处理媒体文件？

在 Django 中，媒体文件是通过 MEDIA_URL 和 MEDIA_ROOT 进行管理的。用户上传的文件会被存储在 MEDIA_ROOT 指定的路径中，而通过 MEDIA_URL 来提供对这些文件的访问。

静态文件与媒体文件的区别

特性	静态文件	媒体文件
来源	开发者事先准备好的文件	用户通过应用上传的文件
变化性	不会改变，除非开发者更新文件	可能会根据用户的行为或需求频繁变化
类型	CSS、JS、图片、字体等	用户上传的图片、视频、音频、文档等
访问方式	通过 STATIC_URL 提供访问	通过 MEDIA_URL 提供访问
存储路径	存储在 STATIC_ROOT 或 STATICFILES_DIRS	存储在 MEDIA_ROOT

为什么需要这两个概念？

- **静态文件**：它们是 Web 应用的基本组成部分，主要用于构建用户界面的外观和交互。
- **媒体文件**：它们主要是用户交互的结果，处理用户上传的动态内容。需要单独管理，因为这些文件是动态产生的，可能具有较大的文件体积。

总结

- **静态文件**：如 CSS、JS 和图像，属于固定的、由开发者事先准备的内容，用于实现网页的样式、布局和交互。
- **媒体文件**：如用户上传的图片、文档、视频等，属于动态的、由用户提供的内容，通常与用户的操作和需求密切相关。

注：Django 中的静态文件和媒体文件的存放位置

在 Django 项目中，分别使用 STATIC_URL 和 MEDIA_URL 来提供这两类文件的访问路径，并使用 STATICFILES_DIRS 和 MEDIA_ROOT 来指定它们在服务器上的存储路径。

在 Django 项目中，**静态文件**和**媒体文件**的存放位置通常取决于开发和生产环境的不同需求。下面分别介绍它们在不同环境下的存放位置：

1. 静态文件 (Static Files)

开发环境中的存放位置：

在开发环境中，Django 会自动管理静态文件，通常存放在项目的某些特定目录中。你可以使用 STATICFILES_DIRS 配置项来指定这些目录。

- **静态文件存放位置**：
    - Django 项目中的静态文件通常放在项目根目录或应用目录中的 static/ 文件夹中。
    - 例如：project_root/static/ 或 app_name/static/。

生产环境中的存放位置：

在生产环境中，静态文件需要通过 Web 服务器（如 Nginx 或 Apache）进行提供，而 Django 自身并不会直接提供静态文件。为此，Django 使用 collectstatic 命令将所有静态文件收集到一个指定的目录中，这个目录通常是 Web 服务器能够直接访问的地方。

- **静态文件存放位置**：
    - 使用 STATIC_ROOT 配置项指定静态文件的存放目录，Django 会将所有静态文件集中到这个目录中。
    - 例如：/var/www/myproject/static/。

- 然后，Nginx 或 Apache 将配置为直接提供这个目录中的静态文件。

配置示例：

```
# settings.py

# 开发环境静态文件目录
STATICFILES_DIRS = [
    BASE_DIR / "static",  # 例如项目根目录下的 static 文件夹
]

# 生产环境静态文件存放位置
STATIC_ROOT = '/var/www/myproject/static/'

# 静态文件的 URL 路径
STATIC_URL = '/static/'
```

总结：

- **开发环境**：静态文件通常存放在项目的 static/ 文件夹或应用的 static/ 文件夹中。
- **生产环境**：通过 collectstatic 命令将静态文件收集到 STATIC_ROOT 目录，然后通过 Web 服务器（如 Nginx）提供。

2. 媒体文件 (Media Files)

开发环境中的存放位置：

在开发环境中，媒体文件通常是由用户上传的文件，存放在项目中的某个目录中。你可以通过 MEDIA_ROOT 配置项指定这个目录。

- **媒体文件存放位置：**
  - 媒体文件通常存放在项目中的 media/ 文件夹中，或者其他由 MEDIA_ROOT 指定的目录。
  - 例如：project_root/media/。

生产环境中的存放位置：

在生产环境中，媒体文件通常存放在服务器的一个单独的目录中，并由 Web 服

务器提供访问。类似于静态文件，媒体文件也需要通过 Web 服务器（如 Nginx）提供。

- **媒体文件存放位置：**
  - 使用 MEDIA_ROOT 配置项指定上传文件存放的目录。
  - 例如：/var/www/myproject/media/。
  - Web 服务器（如 Nginx）将配置为提供这个目录中的文件。

配置示例：

```
# settings.py

# 媒体文件存放位置
MEDIA_ROOT = '/var/www/myproject/media/'

# 媒体文件的 URL 路径
MEDIA_URL = '/media/'
```

总结：

- **开发环境**：媒体文件通常存放在项目的 media/ 文件夹中，或其他由 MEDIA_ROOT 指定的目录。
- **生产环境**：媒体文件存放在服务器的某个目录（如 /var/www/myproject/media/），并由 Web 服务器（如 Nginx）提供。

3. 开发与生产环境的差异总结

文件类型	开发环境存放位置	生产环境存放位置
静态文件	project_root/static/ 或 app/static/	STATIC_ROOT（如 /var/www/myproject/static/）
媒体文件	project_root/media/ 或其他自定义目录	MEDIA_ROOT（如 /var/www/myproject/media/）

访问方式：

- 在开发环境中，Django 提供静态文件和媒体文件的访问，通常不需要额外的 Web 服务器配置。

- 在生产环境中，Django 只负责生成静态文件和媒体文件的路径，Web 服务器（如 Nginx）则直接提供这些文件的访问。

4. 总结：为什么需要区分静态文件和媒体文件？

- **静态文件**：这些文件通常是由开发者事先准备好并不常改变的资源，用于美化页面、实现网页交互等。它们通常是公用的，不依赖用户数据，因此可以集中存放，并通过 Web 服务器进行优化和缓存。
- **媒体文件**：这些文件通常是由用户上传的动态内容，可能会根据用户的需求或行为频繁变化。它们需要单独存放，通常需要考虑权限和文件管理。

Django 区分这两类文件的存储和访问方式，能够更加高效地管理它们，并且便于后期扩展和性能优化。

注：用户是如何访问到网页的？

当用户使用浏览器访问一个网站，且该网站的架构是基于 **React** 前端、**Django** 后端、**Gunicorn** 作为 WSGI 服务器、以及 **Nginx** 作为反向代理时，网站功能的实现过程通常可以分为以下几个步骤：

1. 用户发起请求

- 用户在浏览器中输入网站的 URL 或点击某个链接，这个请求通过 HTTP/HTTPS 协议发送到网站服务器。
- 请求通过 DNS 解析到服务器的 IP 地址，通常使用 **Nginx** 来作为反向代理服务器来处理请求。

2. Nginx 作为反向代理

- **Nginx** 是一个高性能的 Web 服务器和反向代理服务器。当 Nginx 接收到来自浏览器的 HTTP 请求后，它会根据配置文件判断请求应该交给哪个服务器处理。
    - 如果请求的是前端资源（如 index.html、JS、CSS 文件等），Nginx 会直接返回这些静态文件。
    - 如果请求的是动态内容（如 API 请求、用户登录等），Nginx 会将请求转发给后端的 **Django** 服务器。

Nginx 配置示例：

nginx

复制代码

```
server {
    listen 80;
    server_name www.yoursite.com;

    location / {
        root /var/www/your-react-app/build;
        try_files $uri /index.html;
    }

    location /api/ {
        proxy_pass http://127.0.0.1:8000;  # Django 后端地址
    }
}
```

在上述示例中，/api/ 路径的请求会转发给后端 Django 服务器，而其他请求（如前端静态资源）会直接由 Nginx 处理。

3. React 前端的作用

- **React** 是一个 JavaScript 库，用于构建用户界面，尤其是单页面应用（SPA）。当用户首次访问网站时，Nginx 会返回 React 编译后的静态文件（index.html、bundle.js、styles.css 等），这些文件包含了 React 应用的 HTML 和 JavaScript 代码。
- React 应用通常在浏览器端渲染用户界面，并通过 **AJAX**（通常使用 **Fetch API** 或 **Axios**）向后端的 Django 服务器发送请求（例如用户登录、获取数据、提交表单等）。

4. Django 后端的作用

- **Django** 是一个 Python 的 Web 框架，通常用于处理网站的后台逻辑。它提供了路由、数据库操作、身份认证、API 接口等功能。

- 当浏览器通过 React 发送 API 请求到 Django 后端时，Nginx 会将请求转发给 Django。Django 根据请求的 URL 路径和 HTTP 方法（如 GET、POST 等）来匹配视图函数（view）并处理业务逻辑。

例如，Django 可能会通过 REST API 返回 JSON 数据：

python

复制代码

```
from django.http import JsonResponse


def get_data(request):
    data = {"message": "Hello, world!"}
    return JsonResponse(data)
```

Django 还可以处理用户认证、数据库操作等，最终将数据或响应返回给 React 前端。

5. Gunicorn 作为 WSGI 服务器

- **Gunicorn** 是一个 Python WSGI HTTP 服务器，用于运行 Django 应用。Django 本身并不直接处理 HTTP 请求，而是依赖于 WSGI 服务器（如 Gunicorn）来处理。
- Gunicorn 接收到来自 Nginx 转发的请求，调用 Django 的视图处理程序（view），然后将响应返回给 Nginx，再由 Nginx 传递给浏览器。

Gunicorn 启动 Django 应用的命令示例：

bash

复制代码

```
gunicorn myproject.wsgi:application
```

其中 myproject.wsgi:application 是指 Django 项目的 WSGI 应用程序，Gunicorn 会启动并监听某个端口（通常是 127.0.0.1:8000）。

6. 浏览器接收到响应

- 当浏览器通过 Nginx 获取到 React 前端页面时，React 会加载和渲染用户界面。
- 在用户与界面进行交互时，React 会发起 API 请求，Nginx 将请求转发

给 Django 后端，Django 执行相应的处理（例如查询数据库或进行计算），然后返回结果。

- 最终，React 将根据后端返回的数据更新页面内容。

7. 动态内容与前端交互

- 对于动态功能（例如用户登录、提交表单等），React 前端会通过 **AJAX** 或 **Axios** 等技术向后端发送请求。React 前端通常发送 **JSON** 格式的数据，Django 后端解析这些请求，并返回处理后的数据。
- React 前端根据从 Django 后端返回的数据更新 UI，通常使用状态管理（例如 **React Context API** 或 **Redux**）来管理全局状态。

8. 整体工作流程总结

1. **用户访问**：用户在浏览器输入 URL，DNS 解析后，Nginx 接收到请求。
2. **静态资源处理**：Nginx 返回 React 前端的静态文件。
3. **前端渲染**：React 在浏览器端加载并渲染 UI。
4. **前端与后端交互**：React 通过 AJAX 向 Django 后端发送请求。
5. **后端处理**：Django 接收到请求，处理业务逻辑并返回响应。
6. **响应传递**：Nginx 将 Django 的响应返回给浏览器，React 更新界面。
7. **最终呈现**：浏览器根据后端返回的数据展示动态内容。

通过这种前后端分离的架构，网站可以实现高效的性能和灵活的开发，React 负责界面渲染和用户交互，Django 负责后台数据处理和逻辑执行，而 Gunicorn 和 Nginx 则确保请求的高效处理与路由。

注：生产环境和开发环境的不同之处（详细内容可以看 https://github.com/nanmengyu/task-management-system.git，其中 main 分支为生产环境，dev 分支为开发环境）

当后端 Django 启动在 **Unix 套接字**（如：/var/run/socket/gunicorn.sock）中时，前端无法直接通过 HTTP 请求访问该套接字，因为 Unix 套接字是一个本地的通信机制，不通过网络直接进行通信。因此，前端需要通过 Nginx 或其他反向代理将请求转发到后端的 Unix 套接字。

以下是如何在这种部署方式下配置前端访问后端的步骤。

1. 设置 Nginx 反向代理

假设你的前端 React 应用已经部署并通过 Nginx 提供服务，而后端 Django 是通过 Gunicorn 在 Unix 套接字上运行。你需要配置 Nginx 将前端的请求转发到 Gunicorn 的 Unix 套接字。

Nginx 配置示例如下：

```
server {
    listen 80;
    server_name www.example.com;

    # 配置前端的静态文件路径
    location / {
        root /var/www/myproject/frontend;
        try_files $uri /index.html;
    }

    # 配置后端 API 的反向代理
    location /api/ {
        proxy_pass http://unix:/var/run/socket/gunicorn.sock;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

这里的 location /api/ 配置指示 Nginx 将所有以 /api/ 开头的请求代理到后端的 Gunicorn 服务。Gunicorn 服务通过 Unix 套接字 unix:/var/run/socket/gunicorn.sock 进行通信。

在这种情况下，前端 React 通过 axios 向后端发起请求时，URL 路径应该是相

对路径，例如：

（也就是说我们需要把所以的 axios 请求都去掉本地 ip 的前缀）

```
axios.post('/api/users/register/', data)
  .then(response => {
    console.log('Success:', response.data);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

解释：

- 前端 React 应用发送到 /api/... 的请求会被 Nginx 拦截，并转发到 Gunicorn 的 Unix 套接字。
- Nginx 会通过 proxy_pass 指令将请求代理到 unix:/var/run/socket/gunicorn.sock，然后 Gunicorn 处理这些请求并返回响应。
- 由于 Nginx 和 Gunicorn 都在同一台服务器上运行，并且 Nginx 能访问到 Gunicorn 的 Unix 套接字，因此前端 React 不需要直接与 Gunicorn 通信。

2. 确保 Nginx 配置正确

确保以下几点：

- Nginx 能够访问 Gunicorn 的 Unix 套接字文件（通常该文件存放在 /var/run/socket/gunicorn.sock）。
- 确保 ubuntu 用户或者运行 Nginx 的用户具有足够的权限来访问该套接字文件：

```
sudo chown ubuntu:ubuntu /var/run/socket/gunicorn.sock
sudo chmod 755 /var/run/socket
sudo chmod 660 /var/run/socket/gunicorn.sock
```

这些步骤确保 Nginx 能够与 Gunicorn 进行通信。

3. 前端 React 配置

在 React 应用中，axios 请求可以直接使用相对路径来访问 API。例如：

```
axios.post('/api/users/register/', data)
  .then(response => {
    console.log('Success:', response.data);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

这里，前端发送请求时，Nginx 会将 /api/... 的请求转发到后端 Gunicorn 服务，后端通过 Unix 套接字处理请求并返回响应。

注意：

- 如果你在 React 开发环境中使用 npm start，React 默认会运行在 localhost:3000，而后端在生产环境中可能会在不同的地址上，确保你在生产环境中使用正确的路径（例如相对路径 /api/...）来请求后端 API。

4. 防火墙和端口配置

- 确保服务器的防火墙允许 Nginx 访问 Gunicorn 套接字。
- 防火墙规则可以通过以下命令检查：

```
sudo ufw status    # 查看防火墙状态
sudo ufw allow 80  # 允许 HTTP 请求
```

5. CORS 配置（这里我们没有配置域名，不用设置）

如果前端和后端分布在不同的服务器或域下（例如 React 在 www.example.com 上，后端 API 在 api.example.com 上），需要确保后端 Django 配置了 CORS（跨域资源共享）以允许前端访问 API。

使用 django-cors-headers 来设置跨域请求：

```
pip install django-cors-headers
```

在 settings.py 中添加：

```
INSTALLED_APPS = [
    'corsheaders',
    # 其他应用
]
```

```
MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware',
    # 其他中间件
]

CORS_ALLOWED_ORIGINS = [
    "https://www.example.com",  # 允许的前端域名
]
```

6. 检查 Django 的 ALLOWED_HOSTS 设置

如果你在生产环境中使用 Django，确保在 settings.py 中配置了 ALLOWED_HOSTS，允许请求从你的前端应用或者外部访问：

python

复制代码

```
ALLOWED_HOSTS = ['*']  # 允许所有主机（对于开发环境）
# 或者设置为你自己的服务器域名
ALLOWED_HOSTS = ['your-domain.com', 'your-ip-address']
```

总结：

当 Django 后端通过 Unix 套接字与 Gunicorn 运行时，前端无法直接通过 HTTP 请求访问该套接字。前端请求通过 Nginx 反向代理转发到后端，Nginx 会将请求从 HTTP 请求转发到 Gunicorn 套接字进行处理。前端在访问 API 时可以使用相对路径，Nginx 会自动处理转发，而不需要前端关注后端的 Unix 套接字配置。

第五章 部署到外网

5.1. 准备工作

5.1.1 配置代码库

将项目的前端和后端代码存储到 GitHub 仓库中，并使用分支管理开发和生产环境，确保协作和部署流程清晰。

1.1.1 创建 GitHub 仓库

1. **登录 GitHub**
   - 访问 GitHub 官网，登录你的账户。
2. **创建新仓库**
   - 点击右上角的 **“New”** 按钮，创建一个新的仓库。
   - 填写以下信息：
     - **Repository Name**：填写你的项目名称，例如 task-management-system。
     - **Description**：简要描述项目用途，例如 A full-stack task management system.
     - **Visibility**：选择 Public 或 Private。
   - 勾选 Initialize this repository with a README。
   - 点击 **“Create Repository”** 按钮完成创建。
3. **将本地代码推送到 GitHub**
   - 在本地终端中初始化 Git 仓库并将代码推送至 GitHub：

```
git init
git remote add origin https://github.com/<your-username>/task-management-system.git
git add .
git commit -m "Initial commit: Add front-end and back-end code"
```

git branch -M main

git push -u origin main

1.1.2 分离前端和后端代码

1. **创建前端子目录**
    - 在本地项目根目录下创建 frontend/ 文件夹，移动前端代码至该文件夹：

mkdir frontend

mv <前端相关文件> frontend/

2. **创建后端子目录**
    - 创建 backend/ 文件夹，移动后端代码至该文件夹：

mkdir backend

mv <后端相关文件> backend/

3. **更新 README 文件**
    - 修改根目录下的 README.md 文件，描述前端和后端代码的目录结构：

项目结构

- frontend/: 包含前端代码（React 项目）

- backend/: 包含后端代码（Django 项目）

1.1.3 配置分支管理

1. **创建 dev 分支**
    - 使用 dev 分支管理开发环境：

git checkout -b dev

git push -u origin dev

2. **设置分支保护规则（可选）**
    - 在 GitHub 仓库的 **Settings > Branches > Branch Protection Rules** 中，添加以下规则：
        - 保护 main 分支：

- 禁止直接推送到 main 分支。
- 必须通过 Pull Request 合并代码。
- 要求代码审查（至少一人）。

3. **使用 Git 分支进行开发**
    - 开发时基于 dev 分支：

```
git checkout dev
# 修改代码
git add .
git commit -m "Implement new feature"
git push origin dev
```

    - 提交功能完成后，通过 Pull Request 合并至 main 分支。

1.1.4 配置 .gitignore 文件

1. **创建或更新 .gitignore 文件**
    - 确保敏感信息（如环境配置、密钥）和无关文件（如日志、临时文件）不被上传至 GitHub：

```
# Node.js
node_modules/
.env

# Django
*.pyc
__pycache__/
.env

# Logs
*.log
```

2. **检查 Git 状态**
    - 确认 .gitignore 配置生效：

```
git status
```

1.1.5 确保代码可运行

1. **前端测试**
    - 进入 frontend/ 目录，运行前端项目：

```
npm install
npm start
```

    - 确认本地开发环境前端页面正常运行。
2. **后端测试**
    - 进入 backend/ 目录，运行后端服务：

```
pip install -r requirements.txt
python manage.py runserver
```

    - 确认本地后端服务可以正常访问 API 接口。
3. **合并代码并推送到 GitHub**
    - 确保 main 和 dev 分支上的代码可用，便于后续部署。

通过以上步骤，项目的代码库已成功配置，前端和后端代码分别存储在对应目录，分支管理规则清晰，准备就绪。接下来可以进行 CI/CD 配置及部署流程。

注：Git 子模块说明

警告含义

上述警告表明，你尝试将两个独立的 Git 仓库（backend 和 task-manager）作为子文件夹添加到当前的 Git 仓库中。Git 发现这些文件夹本身已经是独立的 Git 仓库，因此提示你可能是无意操作。

关键点：

1. **嵌套的 Git 仓库问题：**
    - 如果你将这些嵌套仓库直接添加到外层仓库，外层仓库不会包含这些子仓库的提交历史。
    - 克隆外层仓库时，子仓库的内容不会随之拉取。
2. **推荐解决方案：Git 子模块（Submodule）：**

- Git 建议使用子模块来管理独立仓库之间的关联关系。

解决方案

使用 Git 子模块

Git 子模块允许在一个仓库中引用其他独立的 Git 仓库，且保持各自独立的版本控制。

1. **添加子模块：**

git submodule add <backend-repo-url> backend

git submodule add <frontend-repo-url> frontend

这里 <backend-repo-url> 和 <frontend-repo-url> 分别是前后端代码仓库的 Git 地址。

2. **提交子模块配置：**

git add .gitmodules backend frontend

git commit -m "Add backend and frontend as submodules"

3. **克隆和更新子模块：**
    - 当其他人克隆你的外层仓库时：

git clone <outer-repo-url>

cd <outer-repo-directory>

git submodule update --init --recursive

4. **子模块更新时的操作：**
    - 在子模块中更新代码：

cd backend

git pull origin main

cd ../frontend

git pull origin main

    - 返回到外层仓库，提交子模块的更新：

git add backend frontend

git commit -m "Update submodule references"

5.1.2 确保服务器可访问

为成功部署任务管理系统，需确保服务器支持访问和运行前后端服务，同时配置必要的依赖环境。以下是详细步骤：

（详细的配置步骤见 4.1 和 4.6）

1.2.1 确保服务器支持访问

1. **获取服务器访问权限**
    - 确认目标服务器具有公网 IP 或域名。
    - 这里使用这个教程中的亚马逊 aws 单 cup 的 1 年免费服务器，永久白嫖 AWS 云服务器，验证、注册指南 - 个人文章 - SegmentFault 思否
    - 配置好环境依赖，开发环境代码，还有反代 nginx 配置，就配好了，可以通过 IP 访问网站了
    - 获取服务器的 SSH 登录凭据，包括用户名和密码（或私钥）。
2. **测试服务器访问**
    - 使用以下命令通过 SSH 登录服务器：

```
ssh <username>@<server-ip>
```

    - 如果使用密钥登录，命令如下：

```
ssh -i <path-to-private-key> <username>@<server-ip>
```

    - 登录成功后，运行基本命令（如 ls, whoami）以确认访问权限。
3. **配置防火墙规则**
    - 确保服务器防火墙和云平台安全组规则允许必要端口的访问：
        - **22**：SSH 访问。
        - **80**：HTTP 访问。
        - **443**：HTTPS 访问。
    - 配置示例（以 UFW 为例）：

```
sudo ufw allow 22
sudo ufw allow 80
sudo ufw allow 443
```

sudo ufw enable

1.2.2 配置依赖环境

在服务器上安装并配置支持前后端服务的运行环境。

1. **更新系统包管理器**
    - 确保系统包管理器和已安装包是最新版本：

sudo apt update && sudo apt upgrade -y

2. **安装 Python 环境（后端支持）**
    - 安装 Python 3 和 pip：

sudo apt install python3 python3-pip -y

    - 验证安装成功：

python3 --version

pip3 --version

3. **安装 Node.js 和 npm（前端支持）**
    - 安装 Node.js 和 npm：

sudo apt install nodejs npm -y

    - 验证安装成功：

node -v

npm -v

4. **安装 Nginx（反向代理支持）**
    - 使用包管理器安装 Nginx：

sudo apt install nginx -y

    - 验证安装成功：

nginx -v

5. **安装 Gunicorn（后端服务支持）**
    - 使用 pip 安装 Gunicorn：

pip3 install gunicorn

    - 验证安装成功：

gunicorn --version

6. **安装其他依赖项**
    - 根据项目需要安装其他依赖（如 PostgreSQL、Docker 等）：

1.2.3 配置服务器用户与权限

1. **创建项目用户（可选）**
    - 为了提高安全性，创建一个专用用户运行项目：

```
sudo adduser project_user
sudo usermod -aG sudo project_user
```

2. **设置目录权限**
    - 确保项目文件的读写权限设置正确：

```
sudo chown -R project_user:project_user /path/to/project
sudo chmod -R 755 /path/to/project
```

1.2.4 验证依赖环境配置

1. **测试 Nginx**
    - 启动 Nginx 并测试服务：

```
sudo systemctl start nginx
sudo systemctl status nginx
```

    - 在浏览器访问 http://<server-ip>，确认能看到默认的 Nginx 欢迎页面。

2. **测试 Node.js 和 Python 环境**
    - 运行简单的 Node.js 和 Python 脚本，确保环境正常工作：
        - Node.js：

```
node -e "console.log('Node.js is working!')"
```

        - Python：

```
python3 -c "print('Python is working!')"
```

通过上述步骤，服务器的访问权限和运行环境已成功配置，为后续部署任务管理系统做好准备。

第六章 实验结果分析

第六章 实验结论与总结