

# 党建沙盘Web系统设计文档

## 1. 项目概述

### 1.1 项目简介

党建沙盘是一个任务式、关卡式的线上学习系统，通过9个递进式关卡帮助用户深入学习党建品牌创建知识。用户需要完成当前关卡的所有任务才能解锁下一关卡，最终系统将根据用户填写的内容生成个人学习报告。

### 1.2 核心功能

- 用户注册登录系统
- 9关卡递进式学习
- 多类型任务完成（文本填写、图片上传）
- 学习进度跟踪
- 个人学习报告生成
- 团体组织验证（后期扩展）

### 1.3 技术选型

本项目选择了 **Go + Gin + GORM** 作为后端技术栈，实现高性能的 Web 服务；前端采用 **Vue 3 + Vite + TypeScript + Element Plus**；数据库使用 **PostgreSQL** 搭配 **Redis** 缓存；通过 **Docker** 容器和 **Nginx** 进行部署。（注：曾考虑 Python FastAPI 方案，但综合性能与部署便利性，最终采用 Go 技术栈。）

## 2. 技术架构

### 2.1 整体架构图

```
graph TB
    A[用户浏览器] --> B[Nginx反向代理]
    B --> C[Vue3前端应用]
    B --> D[Go Gin后端服务]
    D --> E[PostgreSQL主数据库]
    D --> F[Redis缓存/会话]
    D --> G[文件存储系统]

    subgraph "前端层"
        C --> C1[用户认证模块]
        C --> C2[关卡管理模块]
        C --> C3[任务执行模块]
        C --> C4[进度跟踪模块]
        C --> C5[报告生成模块]
    end

    end

    subgraph "后端层"
```

```
D --> D1[用户服务]
D --> D2[关卡服务]
D --> D3[任务服务]
D --> D4[文件服务]
D --> D5[报告服务]

end
```

## 2.2 前后端分离架构优势

- **独立开发**：前后端可以并行开发，提高效率
- **技术栈灵活**：前后端可以分别选择最适合的技术
- **易于扩展**：后期可以轻松添加移动端或小程序接口
- **部署灵活**：前后端可独立部署和扩容，互不影响

## 3. 前端架构设计

### 3.1 技术栈详解

#### 3.1.1 Vue 3 + Composition API

```
// 推荐使用Composition API，代码更清晰
<script setup lang="ts">
import { ref, reactive, computed } from 'vue'

// 响应式数据
const userInfo = reactive({
  username: '',
  currentLevel: 1,
  completedTasks: []
})

// 计算属性
const progressPercentage = computed(() => {
  return (userInfo.completedTasks.length / totalTasks.value) * 100
})
</script>
```

#### 3.1.2 UI框架选择 - Element Plus

选择理由：

- **官方维护**：Vue 3 生态系统的官方 UI 库
- **组件丰富**：提供表单、上传、进度条等必需组件
- **主题定制**：支持红色、黄色主题配色方案
- **文档完善**：详尽的文档说明，适合初学者学习

#### 3.1.3 状态管理 - Pinia

```
// stores/user.ts
import { defineStore } from 'pinia'
```

```
export const useUserStore = defineStore('user', {
  state: () => ({
    userInfo: null,
    currentLevel: 1,
    unlockedLevels: [1],
    taskProgress: {}
  }),

  actions: {
    async login(credentials) {
      // 登录逻辑
    },

    unlockNextLevel() {
      this.unlockedLevels.push(this.currentLevel + 1)
    }
  }
})
```

## 3.2 组件架构

### 3.2.1 组件层次结构

```
App.vue
├── Layout/
│   ├── Header.vue      (导航栏)
│   ├── Sidebar.vue     (关卡列表)
│   └── Footer.vue
├── Views/
│   ├── Login.vue       (登录页)
│   ├── Register.vue    (注册页)
│   ├── Dashboard.vue   (总览页)
│   ├── LevelDetail.vue  (关卡详情)
│   └── Report.vue       (报告页)
└── Components/
    ├── TaskCard.vue     (任务卡片)
    ├── ProgressBar.vue  (进度条)
    ├── FileUpload.vue   (文件上传)
    └── TextEditor.vue    (文本编辑器)
```

### 3.2.2 关键组件设计

#### TaskCard组件

```
<template>
  <el-card class="task-card" :class="{ completed: task.completed }">
    <template #header>
      <div class="task-header">
```

```

        <h3>{{ task.title }}</h3>
        <el-tag v-if="task.completed" type="success">已完成</el-tag>
    </div>
</template>

<div class="task-content">
    <p>{{ task.description }}</p>

    <!-- 文本任务 -->
    <el-input
        v-if="task.type === 'text'"
        v-model="task.content"
        type="textarea"
        :rows="4"
        placeholder="请输入您的答案"
        @blur="saveTask"
    />

    <!-- 图片上传任务 -->
    <FileUpload
        v-if="task.type === 'image'"
        v-model="task.files"
        @upload-success="saveTask"
    />
</div>
</el-card>
</template>

```

### 3.3 路由设计

```

// router/index.ts
import { createRouter, createWebHistory } from 'vue-router'

const routes = [
  {
    path: '/login',
    name: 'Login',
    component: () => import('@/views/Login.vue')
  },
  {
    path: '/',
    name: 'Dashboard',
    component: () => import('@/views/Dashboard.vue'),
    meta: { requiresAuth: true }
  },
  {
    path: '/level/:id',
    name: 'LevelDetail',
    component: () => import('@/views/LevelDetail.vue'),
    meta: { requiresAuth: true }
  }
]

```

```

    },
    {
      path: '/report',
      name: 'Report',
      component: () => import('@/views/Report.vue'),
      meta: { requiresAuth: true }
    }
  ]

  const router = createRouter({
    history: createWebHistory(),
    routes
  })

  // 路由守卫
  router.beforeEach((to, from, next) => {
    const isAuthenticated = localStorage.getItem('token')

    if (to.meta.requiresAuth && !isAuthenticated) {
      next('/login')
    } else {
      next()
    }
  })

```

### 3.4 主题定制

#### 3.4.1 Element Plus 主题配置

```

// styles/theme.scss
:root {
  // 主色调 - 党建红
  --el-color-primary: #d32f2f;
  --el-color-primary-light-3: #e57373;
  --el-color-primary-light-5: #ef5350;
  --el-color-primary-light-7: #ffcdd2;
  --el-color-primary-light-8: #ffebee;
  --el-color-primary-light-9: #fce4ec;

  // 辅助色 - 金黄色
  --el-color-warning: #ffc107;
  --el-color-warning-light-3: #ffeb3b;

  // 背景渐变
  --background-gradient: linear-gradient(135deg, #d32f2f 0%, #ffc107 100%);
}

// 自定义样式
.main-background {
  background: var(--background-gradient);
}

```

```

min-height: 100vh;
}

.level-card {
border-left: 4px solid var(--el-color-primary);
box-shadow: 0 4px 12px rgba(211, 47, 47, 0.15);
}

```

## 4. 后端架构设计

### 4.1 后端技术方案

本项目后端采用 **Go + Gin + GORM** 技术栈来构建 RESTful API 服务。该方案充分利用 Go 语言的高性能和原生并发支持，以及 Gin 框架的简洁高速特性和 GORM 对数据库的友好操作。与 Python FastAPI 方案相比，Go 后端在性能、部署等方面更具优势，但也需要一定的语言学习成本。主要优劣分析如下：

#### 优势：

- **极高性能**：Go 属于编译型语言，执行速度快，能够高效支撑高并发请求 <sup>①</sup>。服务编译后为二进制文件，启动和运行开销低。
- **并发能力**：Go 原生支持轻量级线程（goroutine）和通道机制，可以简单高效地实现并发处理 <sup>①</sup>。在大量用户同时完成任务提交时，后端依然能够保持良好吞吐。
- **部署简单**：Go 编译产物为独立的可执行文件，不依赖解释器环境，部署时无需复杂配置 <sup>②</sup>。容器镜像中只需包含运行文件即可，大幅减少镜像体积和安全风险。
- **内存效率**：Go 相对 Python 更加节省内存，并拥有高效的垃圾回收机制 <sup>③</sup>。对于长期运行的服务，内存占用更可控。
- **类型安全**：Go 为静态类型语言，借助强类型系统，很多错误能在编译时被发现而非运行时。配合完善的单元测试，可以显著提高代码健壮性。

#### 劣势：

- **学习曲线**：对缺乏 Go 经验的开发者来说，上手有一定门槛 <sup>④</sup>。需要适应 Go 的语法、并发模型和项目结构，相比 Python 初期开发速度略慢。
- **开发迭代速度**：Go 相对简洁但偏低级，某些高级特性需要手动实现。与 Python 丰富的库相比，Go 可能需要编写更多样板代码，开发初期效率略低。
- **生态相对新**：Go 生态经过多年发展已较为完善，但在数据科学、机器学习等领域可用的现成库仍不如 Python 丰富 <sup>④</sup>。团队需要根据项目需要评估第三方库的成熟度。

### 4.2 项目结构

后端代码遵循 Go 语言社区推荐的分层结构，将路由控制、业务逻辑、数据访问等解耦组织。典型的 Go Gin 项目目录如下：

```

backend/
├── main.go           # Go 应用入口，初始化 Gin 服务
├── config/
│   └── config.go     # 配置管理（数据库连接、配置加载等）
├── controllers/      # 控制器层（HTTP 接口处理）
│   ├── auth.go       # 认证相关接口
│   ├── user.go        # 用户相关接口
│   └── level.go       # 关卡相关接口

```

```

|   |—— task.go          # 任务相关接口
|   |—— report.go        # 报告相关接口
|—— models/              # 数据模型定义 (GORM Struct)
|   |—— user.go          # 用户模型
|   |—— level.go         # 关卡模型
|   |—— task.go          # 任务模型 (含任务进度)
|   |—— report.go        # 报告模型
|—— services/            # 服务层 (业务逻辑)
|   |—— auth_service.go  # 认证服务 (如 JWT 生成与验证)
|   |—— task_service.go  # 任务服务 (任务提交、进度检查)
|   |—— report_service.go # 报告服务 (报告生成等)
|—— middleware/          # 中间件
|   |—— jwt_auth.go      # JWT 校验中间件
|—— routes/              # 路由注册
|   |—— router.go        # 定义各接口路由和分组
|—— go.mod               # Go Modules 定义
|—— go.sum               # 依赖锁定文件
|—— Dockerfile           # Docker 镜像构建配置

```

## 4.3 数据模型设计

### 4.3.1 用户模型

```

// models/user.go
package models

import "time"

type User struct {
    ID      uint   `gorm:"primaryKey"`
    Username string `gorm:"size:50;uniqueIndex;not null"`
    Email   string `gorm:"size:100;uniqueIndex;not null"`
    HashedPassword string `gorm:"size:255;not null"`
    FullName string `gorm:"size:100"`
    IsActive bool   `gorm:"default:true"`
    CreatedAt time.Time `gorm:"autoCreateTime"`
    CurrentLevel int   `gorm:"default:1"`
    OrganizationID *uint // 可为空, 预留团体组织字段
}

```

### 4.3.2 关卡模型

```

// models/level.go
package models

type Level struct {
    ID      uint   `gorm:"primaryKey"`
    Name    string `gorm:"size:100" // 例: "中心工作"`
    Description string `gorm:"type:text"`
}

```

```

    OrderNumber int // 关卡序号, 如1-9
    IsActive bool `gorm:"default:true"`
    // 关联任务, 一对多关系
    Tasks []Task `gorm:"foreignKey:LevelID"`
}

```

#### 4.3.3 任务模型和进度

```

// models/task.go (含用户任务进度)
package models

import (
    "time"
    "gorm.io/datatypes"
)

type Task struct {
    ID      uint `gorm:"primaryKey"`
    LevelID uint // 外键, 所属关卡ID
    Title   string `gorm:"size:200"`
    Description string `gorm:"type:text"`
    TaskType string `gorm:"size:20"` // 类型: "text", "image", "choice" 等
    IsRequired bool `gorm:"default:true"`
    OrderNumber int
    // 关联关卡, 多对一关系
    Level Level `gorm:"foreignKey:LevelID"`
}

// 用户任务进度模型
type UserTaskProgress struct {
    ID      uint `gorm:"primaryKey"`
    UserID  uint // 外键, 用户ID
    TaskID  uint // 外键, 任务ID
    Content string `gorm:"type:text"` // 用户填写的内容
    FileUrls datatypes.JSON `gorm:"type:json"` // 上传的文件URL列表(JSON格式)
    IsCompleted bool `gorm:"default:false"`
    CompletedAt time.Time // 完成时间
}

```

### 4.4 API 接口设计

#### 4.4.1 认证接口

```

// controllers/auth.go
package controllers

import (
    "net/http"
    "strings"

```



```

"github.com/gin-gonic/gin"
// 假设已导入 models 包和全局 DB、JWT 工具等
)

// 登录请求体结构
type LoginRequest struct {
    Username string `json:"username"`
    Password string `json:"password"`
}

// 注册请求体结构
type RegisterRequest struct {
    Username string `json:"username"`
    Email    string `json:"email"`
    Password string `json:"password"`
    FullName string `json:"full_name"`
}

// 用户登录接口
func LoginHandler(c *gin.Context) {
    var loginReq LoginRequest
    if err := c.ShouldBindJSON(&loginReq); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": "请求参数错误"})
        return
    }
    // 验证用户身份
    var user models.User
    if err := DB.Where("username = ?", loginReq.Username).First(&user).Error; err != nil {
        c.JSON(http.StatusUnauthorized, gin.H{"error": "用户名或密码错误"})
        return
    }
    // 校验密码（假设 HashPassword 函数对明文密码进行同样哈希用于比较）
    if !checkPassword(loginReq.Password, user.HashedPassword) {
        c.JSON(http.StatusUnauthorized, gin.H{"error": "用户名或密码错误"})
        return
    }
    // 生成 JWT Token
    token, err := GenerateToken(user.Username)
    if err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": "Token 生成失败"})
        return
    }
    c.JSON(http.StatusOK, gin.H{"access_token": token, "token_type": "bearer"})
}

// 用户注册接口
func RegisterHandler(c *gin.Context) {
    var req RegisterRequest
    if err := c.ShouldBindJSON(&req); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": "请求参数错误"})
        return
    }

```

```

}
// 检查邮箱是否已存在
var existing models.User
if err := DB.Where("email = ?", req.Email).First(&existing).Error; err == nil {
    c.JSON(http.StatusBadRequest, gin.H{"error": "邮箱已被注册"})
    return
}
// 创建新用户（保存哈希后的密码）
user := models.User{
    Username: req.Username,
    Email: req.Email,
    FullName: req.FullName,
    // 假设 HashPassword 函数对明文密码进行哈希
    HashedPassword: HashPassword(req.Password),
}
if err := DB.Create(&user).Error; err != nil {
    c.JSON(http.StatusInternalServerError, gin.H{"error": "注册失败"})
    return
}
c.JSON(http.StatusOK, gin.H{"message": "注册成功", "user_id": user.ID})
}

```

#### 4.4.2 关卡接口

```

// controllers/level.go
package controllers

import (
    "net/http"
    "github.com/gin-gonic/gin"
    // 假设已导入 models 包和全局 DB 等
)

// 获取用户可访问的关卡列表
func GetUserLevels(c *gin.Context) {
    // 获取当前用户（假设中间件已将用户ID保存到上下文）
    userID := c.GetUint("userID")
    var user models.User
    if err := DB.First(&user, userID).Error; err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": "用户不存在"})
        return
    }
    // 查询用户当前解锁的关卡及之前的关卡
    var levels []models.Level
    DB.Where("order_number <= ?", user.CurrentLevel).Find(&levels)
    c.JSON(http.StatusOK, levels)
}

// 获取指定关卡详情和任务列表
func GetLevelDetail(c *gin.Context) {

```

```

userID := c.GetUint("userID")
levelID := c.Param("level_id") // 关卡ID字符串
var user models.User
if err := DB.First(&user, userID).Error; err != nil {
    c.JSON(http.StatusInternalServerError, gin.H{"error": "用户不存在"})
    return
}
// 将 levelID 转为数值类型
lvlID, _ := strconv.Atoi(levelID)
// 检查用户是否有权限访问该关卡
if user.CurrentLevel < lvlID {
    c.JSON(http.StatusForbidden, gin.H{"error": "关卡未解锁"})
    return
}
// 查询关卡信息及其任务列表
var level models.Level
if err := DB.Preload("Tasks").First(&level, lvlID).Error; err != nil {
    c.JSON(http.StatusNotFound, gin.H{"error": "关卡不存在"})
    return
}
c.JSON(http.StatusOK, level)
}

```

#### 4.4.3 任务接口

```

// controllers/task.go
package controllers

import (
    "net/http"
    "time"
    "github.com/gin-gonic/gin"
    // 假设已导入 models 包和全局 DB 等
)

// 任务提交请求体
type TaskSubmission struct {
    Content string `json:"content"`
    Files []string `json:"files"` // 文件URL列表
}

// 提交任务答案
func SubmitTask(c *gin.Context) {
    userID := c.GetUint("userID")
    taskIDStr := c.Param("task_id")
    var submission TaskSubmission
    if err := c.ShouldBindJSON(&submission); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": "请求参数错误"})
        return
    }
}

```

```

// 查找任务及当前用户
var user models.User
var task models.Task
if err := DB.First(&user, userID).Error; err != nil {
    c.JSON(http.StatusInternalServerError, gin.H{"error": "用户不存在"})
    return
}
if err := DB.Preload("Level").First(&task, taskIDStr).Error; err != nil {
    c.JSON(http.StatusNotFound, gin.H{"error": "任务不存在"})
    return
}
// 保存任务进度记录
progress := models.UserTaskProgress{
    UserID:    user.ID,
    TaskID:    task.ID,
    Content:   submission.Content,
    // 将文件列表转换为 JSON 存储
    FileUrls:  toJSON(submission.Files), // 假设 toJSON 工具函数转换切片为 datatypes.JSON
    IsCompleted: true,
    CompletedAt: time.Now(),
}
if err := DB.Create(&progress).Error; err != nil {
    c.JSON(http.StatusInternalServerError, gin.H{"error": "保存任务进度失败"})
    return
}
// 检查是否完成了当前关卡的所有任务以解锁下一关卡
// (假设 checkLevelCompleted 函数检查用户指定关卡任务完成情况)
if checkLevelCompleted(user.ID, task.LevelID) {
    if user.CurrentLevel == int(task.Level.OrderNumber) {
        user.CurrentLevel += 1 // 解锁下一关卡 (当前关卡序号+1)
        DB.Save(&user)
    }
}
c.JSON(http.StatusOK, gin.H{
    "message": "任务提交成功",
    "progress": progress,
})
}

```

## 5. 数据库设计

### 5.1 数据库选择 - PostgreSQL

- **功能强大**: 支持 JSON 等数据类型, 方便存储结构化数据 5
- **性能优秀**: 在复杂查询和高并发场景下表现出色
- **扩展性好**: 支持存储过程、自定义函数, 后期可以轻松扩展功能

## 5.2 数据库表结构

```
erDiagram
    USERS ||--o{ USER_TASK_PROGRESS : has
    USERS ||--o{ USER_REPORTS : generates
    LEVELS ||--o{ TASKS : contains
    TASKS ||--o{ USER_TASK_PROGRESS : tracks

    USERS {
        int id PK
        string username
        string email
        string hashed_password
        string full_name
        boolean is_active
        datetime created_at
        int current_level
        int organization_id
    }

    LEVELS {
        int id PK
        string name
        text description
        int order_number
        boolean is_active
    }

    TASKS {
        int id PK
        int level_id FK
        string title
        text description
        string task_type
        boolean is_required
        int order_number
    }

    USER_TASK_PROGRESS {
        int id PK
        int user_id FK
        int task_id FK
        text content
        json file_urls
        boolean is_completed
        datetime completed_at
    }

    USER_REPORTS {
        int id PK
        int user_id FK
```

```

    text report_content
    datetime generated_at
}

```

## 5.3 Redis 缓存策略

```

// rdb 为 Redis 客户端实例（已完成连接初始化）
ctx := context.Background()

// 用户会话缓存（有效期1小时）
userSessionKey := fmt.Sprintf("user_session:%d", userID)
rdb.Set(ctx, userSessionKey, userData, time.Hour)

// 关卡数据缓存（有效期30分钟）
levelCacheKey := fmt.Sprintf("level_data:%d", levelID)
rdb.Set(ctx, levelCacheKey, levelData, 30 * time.Minute)

// 用户进度缓存（有效期10分钟）
progressKey := fmt.Sprintf("user_progress:%d", userID)
rdb.Set(ctx, progressKey, progressData, 10 * time.Minute)

```

说明：以上通过 Go-Redis 客户端将关键信息写入 Redis 实现缓存，并设置了不同过期时间，以减轻数据库压力。

# 6. 九大关卡任务设计

## 6.1 关卡主题映射

关卡序号	主题名称	核心目标	主要任务类型
1	中心工作	理解党建与业务融合	文本填写、材料上传
2	问题导向	识别和分析问题	问题列举、案例分析
3	品牌设计	品牌形象设计	创意设计、图片上传
4	品牌带动	品牌影响力扩展	方案制定、效果评估
5	载体建设	载体平台构建	载体规划、资源配置
6	空间呈现	物理空间设计	空间设计、效果图
7	机制建设	制度机制完善	制度设计、流程图
8	融合机制	深度融合机制	融合方案、实施计划
9	守正创新	传承与创新	创新思路、总结报告

## 6.2 任务类型定义

### 6.2.1 文本填写类任务

```
interface TextTask {  
  type: 'text'  
  validation: {  
    minLength: number  
    maxLength: number  
    required: boolean  
  }  
  placeholder: string  
  format?: 'plain' | 'rich' // 纯文本或富文本  
}
```

### 6.2.2 图片上传类任务

```
interface ImageTask {  
  type: 'image'  
  validation: {  
    maxSize: number // MB  
    allowedFormats: string[] // ['jpg', 'png', 'pdf']  
    maxFiles: number  
    required: boolean  
  }  
  description: string  
}
```

### 6.2.3 选择题类任务

```
interface ChoiceTask {  
  type: 'choice'  
  options: Array<{  
    id: string  
    text: string  
    description?: string  
  }>  
  multiSelect: boolean  
  validation: {  
    minSelected: number  
    maxSelected: number  
  }  
}
```

## 6.3 关卡解锁机制

```
flowchart TD
    A[用户开始关卡] --> B{检查任务完成度}
    B -->|未完成| C[显示剩余任务]
    B -->|全部完成| D[关卡完成]
    D --> E{是否为最后关卡}
    E -->|否| F[解锁下一关卡]
    E -->|是| G[生成学习报告]
    F --> H[发送通知]
    G --> I[发送完成证书]
```

## 7. 文件存储系统

### 7.1 存储策略

```
// controllers/file.go
package controllers

import (
    "fmt"
    "path/filepath"
    "time"
    "github.com/gin-gonic/gin"
)

// 文件上传接口处理（保存任务相关文件）
func UploadFileHandler(c *gin.Context) {
    userID := c.GetUint("userID")
    taskIDStr := c.PostForm("task_id")
    file, err := c.FormFile("file")
    if err != nil {
        c.JSON(400, gin.H{"error": "文件上传失败"})
        return
    }
    ext := filepath.Ext(file.Filename)
    allowedExts := map[string]bool{
        ".jpg": true, ".jpeg": true, ".png": true,
        ".pdf": true, ".doc": true, ".docx": true,
    }
    // 验证文件类型和大小
    if !allowedExts[ext] {
        c.JSON(400, gin.H{"error": "文件格式不支持"})
        return
    }
    if file.Size > 10*1024*1024 {
        c.JSON(400, gin.H{"error": "文件大小超过限制"})
        return
    }
}
```



```
// 生成唯一文件名并保存
filename := fmt.Sprintf("%d_%s_%d_%s", userID, taskIDStr, time.Now().Unix(), file.Filename)
savePath := filepath.Join("/app/uploads", filename)
if err := c.SaveUploadedFile(file, savePath); err != nil {
    c.JSON(500, gin.H{"error": "保存文件失败"})
    return
}
c.JSON(200, gin.H{
    "filename": filename,
    "url": "/uploads/" + filename,
    "size": file.Size,
})
}
```

说明：上述代码通过 Gin 提供的 `c.FormFile` 和 `c.SaveUploadedFile` 方法实现文件接收与保存，并对文件类型和大小进行校验，限制在 10MB 以内。上传目录按照 `/app/uploads` 存储，可通过 Nginx 或 Gin 静态文件服务将该目录映射为 `/uploads` URL 前缀对外提供访问。

## 7.2 文件组织结构

```
uploads/
├── avatars/      # 用户头像
├── tasks/        # 任务相关文件
│   ├── level_1/  # 按关卡分类存放
│   ├── level_2/
│   └── ...
└── reports/      # 生成的报告文件
```

# 8. 个人学习报告生成

## 8.1 报告生成流程

```
sequenceDiagram
    participant U as 用户
    participant F as 前端
    participant B as 后端
    participant R as 报告服务
    participant D as 数据库

    U->>F: 完成最后关卡
    F->>B: 请求生成报告
    B->>D: 查询用户所有任务数据
    D-->>B: 返回任务完成情况
    B->>R: 调用报告生成服务
    R->>R: 分析数据、生成报告
    R-->>B: 返回报告内容
    B->>D: 保存报告
```

B-->F: 返回报告 URL  
F-->U: 显示报告预览

## 8.2 报告生成实现

```
// services/report_service.go
package services

import (
    "bytes"
    "html/template"
    "time"
)

type ReportGenerator struct { /* 可以包含模板路径配置等 */ }

func (r *ReportGenerator) GenerateUserReport(userID uint) (map[string]interface{}, error) {
    // 获取用户所有任务完成情况
    userProgress := r.getUserProgress(userID)

    reportData := map[string]interface{}{
        "user_info":    r.getUserInfo(userID),
        "completion_summary": r.analyzeCompletion(userProgress),
        "level_insights": r.analyzeByLevel(userProgress),
        "recommendations": r.generateRecommendations(userProgress),
        "generated_at":  time.Now(),
    }

    // 使用 HTML 模板生成报告内容
    tmpl, err := template.ParseFiles("templates/user_report.html")
    if err != nil {
        return nil, err
    }
    var htmlContent bytes.Buffer
    if err := tmpl.Execute(&htmlContent, reportData); err != nil {
        return nil, err
    }

    // 生成 PDF 文件（可选步骤，调用外部库）
    pdfPath, err := r.generatePDF(htmlContent.String(), userID)
    if err != nil {
        // 即使 PDF 生成失败，也可忽略错误返回 HTML 报告
        pdfPath = ""
    }

    return map[string]interface{}{
        "html_content": htmlContent.String(),
        "pdf_url":      pdfPath,
        "report_data":  reportData,
    }, nil
}
```

```
    }, nil  
}
```

说明：`ReportGenerator` 服务通过读取预定义的 HTML 模板，将用户信息及各关卡完成情况渲染生成完整的报告内容。`generatePDF` 可以利用第三方库（如 `wkhtmltopdf`、`gopdf` 等）将 HTML 转换为 PDF 文件并返回路径，用于提供下载或打印。

## 9. 用户认证与权限管理

### 9.1 JWT 认证流程

```
// core/security.go  
package core  
  
import (  
    "errors"  
    "time"  
    "github.com/golang-jwt/jwt/v4"  
)  
  
var SECRET_KEY = []byte("your-secret-key")  
const TOKEN_EXPIRE_DURATION = time.Minute * 30  
var jwtMethod = jwt.SigningMethodHS256  
  
// 生成访问令牌  
func CreateAccessToken(username string) (string, error) {  
    claims := jwt.MapClaims{  
        "sub": username,  
        "exp": time.Now().Add(TOKEN_EXPIRE_DURATION).Unix(),  
    }  
    token := jwt.NewWithClaims(jwtMethod, claims)  
    tokenString, err := token.SignedString(SECRET_KEY)  
    if err != nil {  
        return "", err  
    }  
    return tokenString, nil  
}  
  
// 验证令牌并提取用户名  
func VerifyToken(tokenStr string) (string, error) {  
    token, err := jwt.Parse(tokenStr, func(token *jwt.Token) (interface{}, error) {  
        return SECRET_KEY, nil  
    })  
    if err != nil || !token.Valid {  
        return "", errors.New("Token 无效")  
    }  
    claims, ok := token.Claims.(jwt.MapClaims)  
    if !ok {  
        return "", errors.New("Token 无效")  
    }  
}
```

```

    }
    username, ok := claims["sub"].(string)
    if !ok || username == "" {
        return "", errors.New("Token 无效")
    }
    return username, nil
}

```

说明：上面代码使用了 `golang-jwt/jwt` 包。`CreateAccessToken` 创建一个 HS256 签名的 JWT，其中 `sub` (subject) 字段保存用户名，`exp` 字段设置过期时间为30分钟<sup>6</sup><sup>7</sup>。`VerifyToken` 则在收到客户端请求时用于校验 JWT 的有效性，解析出其中的用户名。如果令牌无效或过期，则返回错误。实际项目中，可将 `VerifyToken` 封装为 Gin 中间件，在每次请求时自动检查 `Authorization` Header，验证通过则将当前用户信息写入 `context`，供后续处理使用。

## 9.2 权限控制

```

// 检查用户对关卡的访问权限
func UserCanAccessLevel(user models.User, levelID uint) bool {
    // 第一关卡对所有用户开放
    if levelID == 1 {
        return true
    }
    // 检查上一关是否已完成（假设 checkLevelCompleted 实现了关卡完成度检查）
    prevLevel := levelID - 1
    return checkLevelCompleted(user.ID, prevLevel)
}

```

说明：关卡权限控制逻辑用于防止用户跳过前置关卡直接访问后面的内容。实现中，第1关默认解锁<sup>8</sup>；对于其他关卡，需要先验证用户是否完成了前一关所有任务。可通过查询 `UserTaskProgress` 表或用户当前等级 `CurrentLevel` 等信息来判断。只有满足条件，才允许访问目标关卡内容，否则返回权限不足的错误。该检查可在获取关卡详情接口或前端路由守卫中使用。

## 10. 开发环境配置

### 10.1 前端开发环境

```

# 安装依赖
npm create vue@latest party-sandbox-frontend
cd party-sandbox-frontend
npm install

# 安装额外依赖
npm install element-plus @element-plus/icons-vue
npm install pinia vue-router@4
npm install axios
npm install @types/node --save-dev # 安装 TypeScript Node 类型定义

```

## 10.2 前端项目配置

```
// vite.config.ts
import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import { resolve } from 'path'

export default defineConfig({
  plugins: [vue()],
  resolve: {
    alias: {
      '@': resolve(__dirname, 'src')
    }
  },
  server: {
    port: 3000,
    proxy: {
      '/api': {
        target: 'http://localhost:8000',
        changeOrigin: true
      }
    }
  }
})
```

## 10.3 后端开发环境

```
# 初始化 Go 模块
go mod init party-sandbox-backend

# 获取后端所需依赖库
go get github.com/gin-gonic/gin
go get gorm.io/gorm
go get gorm.io/driver/postgres
go get github.com/go-redis/redis/v8
go get github.com/golang-jwt/jwt/v4

# (可选) 安装数据库迁移工具或测试库
go install -tags 'postgres' github.com/golang-migrate/migrate/v4/cmd/migrate@latest
go get github.com/stretchr/testify
```

说明：以上假设本地已安装 Go 环境。通过 `go mod init` 初始化模块后，使用 `go get` 引入 Gin Web 框架、GORM ORM 库、Postgres 驱动、Redis 客户端以及 JWT 库等。执行后将自动在 `go.mod` 和 `go.sum` 中记录版本信息。测试相关和迁移工具根据需要安装。

## 10.4 数据库配置

```
// config/database.go
package config

import (
    "log"
    "gorm.io/driver/postgres"
    "gorm.io/gorm"
    // 假设已导入 models 包以便使用模型 Struct
)

var DB *gorm.DB

func InitDatabase() {
    // PostgreSQL 连接字符串 (根据实际用户名、密码、数据库名称调整)
    dsn := "host=localhost user=username password=password dbname=party_sandbox port=5432"
    sslmode="disable"
    db, err := gorm.Open(postgres.Open(dsn), &gorm.Config{})
    if err != nil {
        log.Fatal("无法连接数据库:", err)
    }
    // 自动迁移模型, 创建表结构
    db.AutoMigrate(&models.User{}, &models.Level{}, &models.Task{}, &models.UserTaskProgress{})
    DB = db
}
```

说明：上述代码利用 GORM 初始化 PostgreSQL 数据库连接，并调用 `AutoMigrate` 自动创建/更新表结构（包括用户、关卡、任务、任务进度等表）。开发阶段可使用自动迁移简化操作，生产环境更推荐使用独立的迁移工具进行数据库 schema 管理。

## 10.5 数据库迁移

```
# 安装 golang-migrate CLI 工具 (包含 PostgreSQL 支持)
go install -tags 'postgres' github.com/golang-migrate/migrate/v4/cmd/migrate@latest

# 创建迁移文件 (示例)
migrate create -ext sql -dir db/migrations -seq create_users_table

# (编辑生成的迁移SQL文件, 写入创建或修改表的 SQL 语句)

# 执行迁移, 将所有待执行的变更应用到数据库
migrate -database "postgres://username:password@localhost:5432/party_sandbox?"
sslmode="disable" -path db/migrations up
```

通过上述命令，可以方便地对数据库模式变更进行版本管理。例如第一次迁移会创建 `users` 表，后续如需修改字段或增加新表，可新建迁移文件编写 SQL 并执行。**golang-migrate** 提供了回滚等功能，确保数据库演进可控。也可考虑使用更高层次的迁移工具如 **Atlas**，通过声明式的方式管理数据库 schema，但相对而言 **golang-migrate** 更简单直接，便于团队协作。

## 11. 测试策略

### 11.1 前端测试

```
// tests/components/TaskCard.test.ts
import { mount } from '@vue/test-utils'
import TaskCard from '@components/TaskCard.vue'

describe('TaskCard 组件', () => {
  test('显示任务标题', () => {
    const wrapper = mount(TaskCard, {
      props: {
        task: {
          title: '测试任务',
          description: '这是一个测试任务',
          type: 'text'
        }
      }
    })
    expect(wrapper.text()).toContain('测试任务')
  })

  test('文本输入功能', async () => {
    const wrapper = mount(TaskCard, {
      props: {
        task: { type: 'text', title: '文本任务' }
      }
    })
    const textarea = wrapper.find('textarea')
    await textarea.setValue('用户输入的内容')
    expect(wrapper.emitted('task-updated')).toBeTruthy()
  })
})
```

### 11.2 后端测试

```
package tests

import (
    "net/http"
    "net/http/httptest"
    "strings"
    "testing"
    "party_sandbox_backend/controllers"
)

func setupRouter() *gin.Engine {
    r := gin.Default()
    // (注册所有路由及中间件, 假定 controllers 中的处理器已实现)
```

```

api := r.Group("/api")
{
    api.POST("/auth/register", controllers.RegisterHandler)
    api.POST("/auth/login", controllers.LoginHandler)
    // ... 其他路由
}
return r
}

func TestUserRegistration(t *testing.T) {
    router := setupRouter()
    // 模拟发送用户注册请求
    body :=
`{"username":"testuser","email":"test@example.com","password":"testpass123","full_name":"测试用户"}`
    req, _ := http.NewRequest("POST", "/api/auth/register", strings.NewReader(body))
    req.Header.Set("Content-Type", "application/json")
    resp := httptest.NewRecorder()
    router.ServeHTTP(resp, req)

    if resp.Code != http.StatusOK {
        t.Fatalf("期望状态码 200, 实际得到 %d", resp.Code)
    }
    if !strings.Contains(resp.Body.String(), "user_id") {
        t.Errorf("响应中缺少 user_id 字段")
    }
}

func TestUserLogin(t *testing.T) {
    router := setupRouter()
    // 准备测试用户数据 (可以在此预先创建 testuser, 或者模拟数据库中的已有用户)
    form := "username=testuser&password=testpass123"
    req, _ := http.NewRequest("POST", "/api/auth/login", strings.NewReader(form))
    req.Header.Set("Content-Type", "application/x-www-form-urlencoded")
    resp := httptest.NewRecorder()
    router.ServeHTTP(resp, req)

    if resp.Code != http.StatusOK {
        t.Fatalf("期望状态码 200, 实际得到 %d", resp.Code)
    }
    if !strings.Contains(resp.Body.String(), "access_token") {
        t.Errorf("响应中缺少 access_token 字段")
    }
}

```

说明：后端测试利用 Go 的 `net/http/httptest` 包模拟 HTTP 请求发送到 Gin 路由 [9](#) [10](#)。

`TestUserRegistration` 构造了一个注册请求，提交用户信息 JSON，并断言返回状态码为 200 且响应中包含 `user_id`。`TestUserLogin` 模拟登录表单提交，断言返回的 JSON 中包含 `access_token`。在实际测试中，可使用测试数据库或 mock 对象，以保证测试可重复运行且不影响正式数据。



1 2 3 4 5 6 7 8 9 10 party-sha-san-project-setup.md  
file:///file-XJkt7hNqsTQ3DsYavwdGSX