体育非遗数字展示平台技术报告

目录

1. 项目介绍

1.1 项目背景与挑战

体育类非物质文化遗产（简称“体育非遗”），如传统武术、民族体育项目、龙舟竞渡等，是中华优秀传统文化的重要组成部分，蕴含着丰富的历史信息、独特的技艺体系和深厚的文化价值。然而，在现代化进程加速和社会变迁的背景下，许多体育非遗项目正面临着严峻的挑战：

- **传承断裂风险：** 依赖口传心授的传统传承模式受到冲击，传承人老龄化、后继乏人现象普遍。
- **记录与保存困难：** 动作要领、历史渊源、文化内涵等信息分散，缺乏系统化、数字化的记录与保存手段，易于流失或失真。
- **传播推广受限：** 受地域、形式等限制，体育非遗的社会认知度和影响力有限，特别是难以吸引年轻一代的关注与参与。
- **互动交流缺乏：** 爱好者、学习者、研究者和传承人之间缺少便捷有效的交流平台，不利于知识共享和社群发展。

1.2 项目目的与价值

为应对上述挑战，“体育非遗数字展示与互动平台”应运而生。本项目旨在利用现代信息技术，为体育类非物质文化遗产的**保护、传承、研究与推广**提供一个**集成化、数字化、互动化**的解决方案。

核心目的：

1. **系统性保护：** 建立一个结构化的体育非遗数字资源库，对珍贵的文字、图片、音视频资料进行有效管理和长期保存。
2. **创新性传承：** 突破时空限制，利用多媒体展示和在线互动功能，为教学、学习和实践提供新的途径，激发年轻一代的兴趣。
3. **促进学术研究：** 为研究人员提供便捷的资料检索和数据分析支持，推动体育非遗的理论研究与学科发展。
4. **扩大社会影响：** 打造一个开放的展示窗口和交流社区，提升体育非遗的社会可见度和公众参与度。

项目价值与创新点：

- **领域聚焦：** 专注于体育非遗领域，内容更具深度和专业性。

- **教、学、研、展一体化：** 平台不仅是静态展示，更融合了教学管理（教师/学生角色）、社区交流（论坛/即时通讯）和内容创作功能，形成一个动态的生态系统。
- **技术赋能：** 采用 WebSocket 实现实时互动，结合高效的文件处理（自动压缩、水印等），优化用户体验和数据管理。
- **精细化权限管理：** 基于角色的访问控制确保了信息安全和管理的有序性，满足不同用户群体的需求。

1.3 目标用户

本平台主要服务于以下用户群体：

- **非遗传承人与管理者：** 发布权威信息，管理项目资料，与学习者互动。
- **教师与学生（学校/机构）：** 在线教学、学习体育非遗知识与技能，提交作业或成果。
- **研究人员：** 查阅资料，了解项目动态，进行学术交流。
- **体育非遗爱好者与公众：** 了解体育非遗项目，参与讨论，观看学习资源。
- **平台管理员：** 负责系统维护、用户管理和内容审核。

2. 功能介绍

本平台围绕体育非遗的数字化展示与互动需求，设计并实现了以下核心功能模块，旨在为不同角色的用户提供丰富、便捷、高效的使用体验。

2.1 用户中心与权限管理

- **多角色支持：** 管理员、教师、学生。
- **基础用户功能：** 注册、登录、个人资料编辑、密码修改、内容/收藏/通知查看。
- **安全保障：** 密码哈希存储、登录认证、CSRF 防护。

2.2 体育非遗项目展示

- **结构化条目：** 名称、类别、历史、传承人等信息。
- **多媒体呈现：** 高清图片、在线视频播放、富文本详情。
- **便捷检索：** 分类浏览、关键词搜索。

2.3 在线交流与互动

- **主题论坛：** 按主题/项目发帖、回帖、管理（置顶/加精/删除）。
- **即时通讯（IM）：** 基于 WebSocket 实现一对一/群组实时文字聊天、新消息提醒。
- **系统通知与公告：** 站内通知（评论/私信/关注动态）、平台公告。

2.4 内容发布与管理

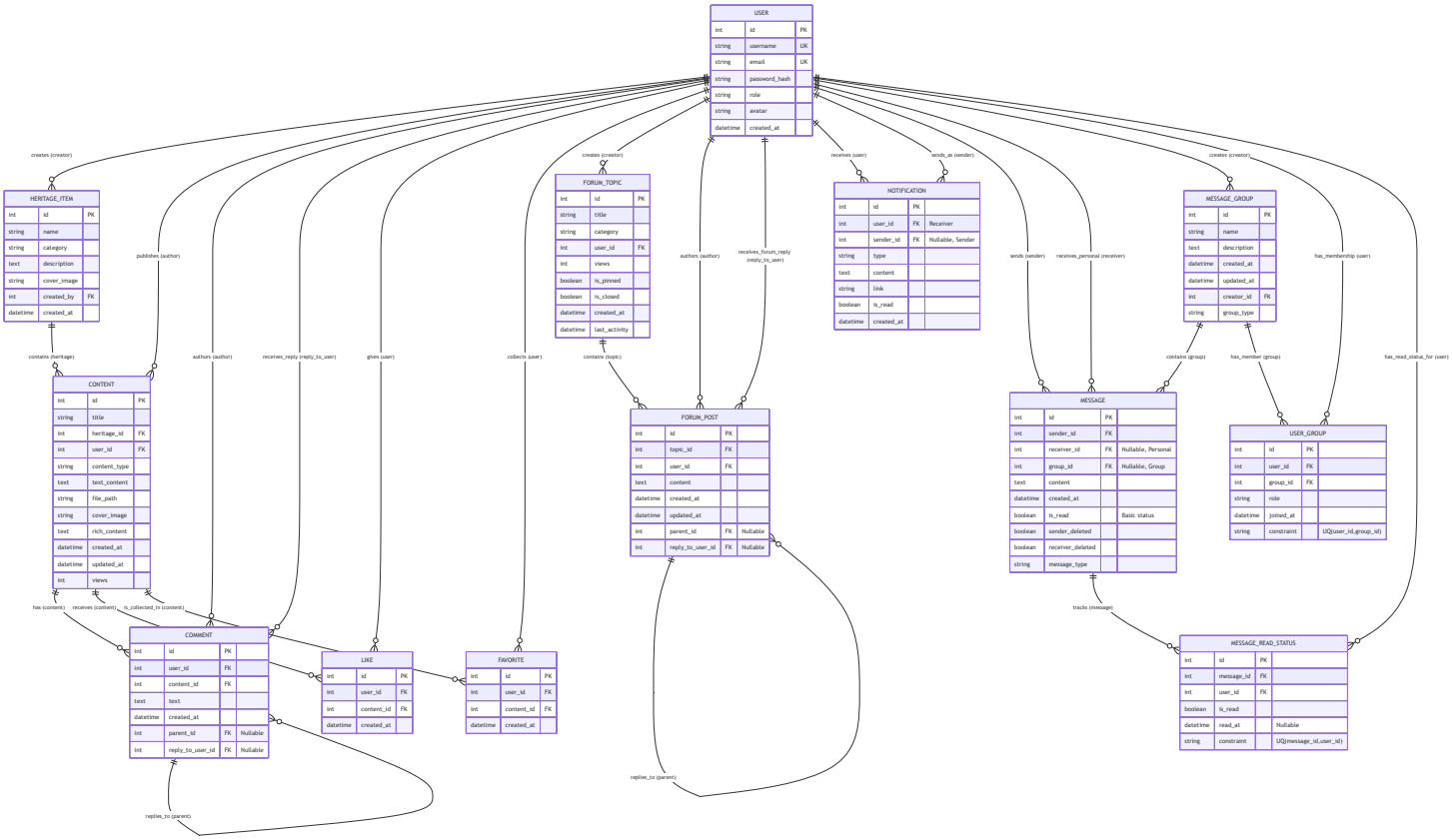
- **内容创建：** 文章资讯、教学资源（图文/视频）。
- **富文本编辑：** 集成 CKEditor，支持图文混排。
- **文件上传与处理：** 支持图片/视频，图片自动压缩与水印添加，安全存储。
- **内容管理：** 用户编辑/删除自己的内容，管理员/教师进行审核/推荐/置顶。
- **数据统计：** 包含浏览量、点赞、评论等计数。

3. 技术说明

3.1 系统架构

- **后端框架：** Flask (Python 3.11)
- **数据库：** MySQL (通过 SQLAlchemy ORM 访问)
- **前端技术：** HTML5, CSS3, JavaScript, Bootstrap (模板引擎 Jinja2)
- **实时通信：** Flask-SocketIO (基于 python-socketio 和 WebSocket)
- **文件处理库：** Pillow (用于图像处理), Werkzeug (用于安全文件名等)
- **Web 服务器（部署时）：** Gunicorn / uWSGI + Nginx (推荐)
- **其他关键库：** Flask-Login (用户认证), Flask-WTF (表单处理与 CSRF 防护), Flask-Migrate (数据库迁移), Flask-SQLAlchemy (数据库 ORM), Bleach (HTML 清理), Flask-Talisman (安全头), Flask-Limiter (速率限制)

3.2 数据模型图 (Mermaid)



(注：此图简化了部分关系细节以保持清晰)

3.3 自定义函数、类与核心方法说明

项目中的 `utils` 和 `models` 目录包含了一系列核心的辅助工具、配置函数、装饰器、数据模型和管理类，旨在提升代码质量、安全性和可维护性。

3.3.1 API 响应函数 (`app/utils/response.py`)

为确保前后端接口交互的规范性和一致性，定义了一套标准化的 API 响应函数。

- **`api_success(...)`**：生成标准成功响应 (HTTP 200)，包含 `success: True` ， `message` , 可选 `data` 。
- **`api_error(...)`**：生成标准错误响应，包含 `success: False` ， `message` , 可选 `error_code` , `details` ， 状态码可指定 (默认 400)。
- **`api_list_response(...)`**：专用于分页列表的成功响应， `data` 中含 `items` 和 `pagination` 对象。
- **`api_validation_error(...)`**：表单验证失败响应 (HTTP 422)， `error_code` 为 "VALIDATION_ERROR"。
- **`api_not_found(...)`**：资源未找到响应 (HTTP 404)。
- **`api_unauthorized(...)`**：未授权访问响应 (HTTP 401)。
- **`api_forbidden(...)`**：禁止访问响应 (HTTP 403)。

3.3.2 数据库辅助函数（`app/utils/db_helpers.py`）

提供与数据库交互的便捷工具。

- **`ensure_column_exists(...)`**：检查并按需添加数据表列。
- **`table_exists(...)`**：检查表是否存在。
- **`get_column_names(...)`**：获取表的列名。
- **`@contextmanager safe_db_operation()`**：数据库事务安全操作的上下文管理器（自动 commit 或 rollback）。
- **`execute_sql(...)`**：安全执行原生 SQL 语句（使用参数绑定防注入）。
- **`batch_insert(...)`**：高效批量插入数据。

3.3.3 权限控制与安全装饰器（`app/utils/decorators.py`）

用于简化视图函数的权限检查和安全功能。

- **`@role_required(role)`**：核心角色权限检查（依赖 User 模型的 `is_admin`，`is_teacher` 等属性）。
- **`@admin_required`**：便捷装饰器，要求管理员权限。
- **`@teacher_required`**：便捷装饰器，要求教师或管理员权限。
- **`@ratelimit(...)`**：**自定义内存实现的**访问频率限制装饰器（基于 IP 或 User ID）。
- **`@log_access`**：访问日志记录装饰器（记录请求、来源、耗时）。
- **`@cache_control(...)`**：添加 `Cache-Control` HTTP 头。

3.3.4 文件处理函数（`app/utils/file_handlers.py`）

封装了文件上传、验证、处理（图片压缩、水印）、保存和删除等一系列操作。

- **`ALLOWED_IMAGE_EXTENSIONS`，`ALLOWED_VIDEO_EXTENSIONS`**：定义了允许上传的图片和视频文件扩展名集合。
- **`allowed_file(filename, allowed_extensions)`**：检查给定文件名 `filename` 的扩展名是否在 `allowed_extensions` 集合中。
- **`validate_image_content(file_storage)`**：**健壮的图片内容验证**。不仅仅检查扩展名，还会将上传的文件（`FileStorage` 对象）临时保存，然后使用 `imghdr.what()` 识别文件头，并结合 `mimetypes.guess_type()` 检查 MIME 类型，确保文件内容确实是图片。操作完成后会删除临时文件。
- **`compress_image(image, max_size=(800, 800))`**：压缩 PIL `Image` 对象。使用 `image.thumbnail()` 方法按比例缩放图片至 `max_size` 范围内（保持宽高比），采用 `Image.Resampling.LANCZOS` 算法以获得较好的压缩质量。同时，将图片模式转换为 'RGB' 以兼容 JPEG 等格式。
- **`add_watermark(image, text)`**：给 PIL `Image` 对象添加**文字水印**。在图片右下角绘制，字体大小动态计算，带半透明背景。

- `save_file(file, file_type, watermark=None)`：**核心的文件保存函数**。处理图片（压缩、水印）和视频，包含安全检查、唯一命名、路径管理和错误处理。返回相对路径。
- `delete_file(file_path)`：根据提供的相对路径（`uploads/.../...`）构建完整的文件系统路径，检查文件是否存在，如果存在则使用 `os.remove()` 删除。包含错误处理和日志记录。

3.3.5 WebSocket 管理（`app/utils/websocket_manager.py`）

提供了 `WebSocketManager` 类来集中管理 WebSocket 连接状态和处理相关逻辑。

- `class WebSocketManager`：集中管理连接（注册、注销、活动更新、超时检查），处理连接错误（重试机制）。
- `@websocket_error_handler`：Socket.IO 事件处理器的错误处理装饰器。
- `init_websocket_manager(app)`：在 Flask 应用初始化时创建管理器实例。

3.3.6 应用安全配置（`app/utils/security_config.py`）

集中配置了应用的多项安全相关特性。

- `setup_security(app)`：主配置函数。配置 **Flask-Limiter**, **Flask-Talisman**, **Password Policy**, **Session Security**。
- `validate_password(...)`：根据策略验证密码复杂度。
- `sanitize_html(...)`：使用 `bleach` 清理 HTML，防 XSS。
- `@rate_limit(...)`：**Flask-Limiter 提供的**装饰器，用于应用特定路由的限流规则。
- `@check_content_type(...)`：检查请求 `Content-Type` 头。
- `@check_file_type(...)`：检查上传文件的扩展名。

3.3.7 日志配置（`app/utils/logging_config.py`）

配置了应用的日志记录系统。

- `setup_logging(app)`：主配置函数，设置日志目录、格式、文件轮转（应用日志、错误日志、访问日志、性能日志）。创建独立的 access 和 performance logger。
- `log_access(response)`：Flask `after_request` 处理器，记录访问日志。
- `log_performance(...)`：辅助函数，记录函数执行时间到性能日志。

3.3.8 模板上下文处理器（`app/utils/context_processors.py`）

- `common_data()`：向所有 Jinja2 模板注入通用数据（如导航分类、用户收藏数）。

3.3.9 数据模型（`app/models/`）

数据模型定义了应用的核心数据结构及其关系，使用 Flask-SQLAlchemy 进行定义。

- **初始化与辅助函数 (app/models/__init__.py)**: 定义 `beijing_time()` 辅助函数；导入并导出所有模型类。
- **用户模型 (app/models/user.py)**: 定义 `User` 类，包含基本信息、密码处理、角色属性、与各模块的关系、Flask-Login 集成 (`load_user`)。
- **非遗项目模型 (app/models/heritage.py)**: 定义 `HeritageItem` 类，包含项目信息、与创建者和内容的关系、 `to_dict()` 方法。
- **内容模型 (app/models/content.py)**: 定义 `Content` 类，包含内容信息、类型、文件路径、富文本、浏览量、与项目/作者/互动模块的关系、 `to_dict()` 方法。
- **互动模型 (app/models/interaction.py)**: 定义 `Comment` (支持嵌套回复)、 `Like` 、 `Favorite` 类及其关系、 `to_dict()` 方法。
- **论坛模型 (app/models/forum.py)**: 定义 `ForumTopic` 和 `ForumPost` (支持嵌套回复) 类及其关系、统计属性、 `to_dict()` 方法。
- **通知模型 (app/models/notification.py)**: 定义 `Notification` 类，包含接收者、发送者、类型、内容、链接、已读状态、 `to_dict()` 方法。
- **消息模型 (app/models/message.py)**: 定义 `Message` (支持私聊/群聊/广播，软删除)、 `MessageGroup` 、 `UserGroup` (多对多关联，含角色)、 `MessageReadStatus` (精确跟踪群聊已读) 类及其复杂关系。

3.4 路由与视图函数 (app/routes/)

应用的路由和视图逻辑组织在 `app/routes/` 目录下的各个 Python 文件中，每个文件通常对应一个功能模块，并使用 Flask 蓝图 (Blueprint) 进行管理。

- **核心库**: Flask, Flask-Login, Flask-WTF, SQLAlchemy。
- **组织方式**: 按功能模块划分蓝图 (`auth_bp` , `content_bp` , `forum_bp` , `heritage_bp` , `main_bp` , `message_bp` , `notification_bp` , `user_bp` , `errors_bp`)。
- **通用模式**: 使用路由装饰器、认证授权装饰器、表单处理、数据库交互、模板渲染、重定向等标准 Flask 开发模式。
- **关键模块与视图函数说明:**
  - **认证路由 (routes/auth.py)**: 处理用户注册、登录、退出。
  - **主页与静态页路由 (routes/main.py)**: 处理首页、关于我们等静态页面，首页含动态数据查询。
  - **非遗项目路由 (routes/heritage.py)**: 实现非遗项目的列表、详情、创建、编辑、删除功能，包含权限控制和文件处理。
  - **内容路由 (routes/content.py)**: 实现内容的列表（含筛选、搜索）、详情（含评论、点赞、收藏、相关推荐）、创建、编辑、删除功能，以及图片上传接口（包括 CKEditor 集成）。包含权限控制、文件处理和通知发送逻辑。
  - **用户中心路由 (routes/user.py)**: 实现用户个人资料页、编辑资料、修改密码、我的内容、我的收藏、管理员后台（控制面板、用户管理列表、用户增删改）、以及用于异步加载和操作的

API 接口。
    - **论坛路由 (`routes/forum.py`)**: 实现论坛首页、主题列表（含筛选）、主题详情页（含嵌套回复、分页）、创建主题、管理操作（置顶、关闭、删除）等。包含复杂的数据库查询（JOIN、别名）和通知发送逻辑。提供最新主题 API。
    - **消息路由 (`routes/message.py`)**: 实现复杂的消息系统，包括私信列表、发送、查看、回复、删除（软删除）；群组列表、创建、查看、发送消息、成员管理（添加、移除、提升管理员、离开）、编辑群组信息；以及群发广播功能。涉及精细的权限控制和已读状态管理。
    - **通知路由 (`routes/notification.py`)**: 实现用户通知列表、管理员发布公告功能，并提供 `send_notification` 辅助函数供其他模块调用。
    - **错误处理路由 (`routes/errors.py`)**: 定义全局的 403, 404, 500 错误处理函数，并为特定 API 路径提供 JSON 格式的错误响应。

3.5 API 接口 (`app/api/`)

项目提供了一套 RESTful API 接口，通过 `app/api/` 目录下的蓝图 (`api_bp`) 进行组织，**已禁用 CSRF 保护**，依赖 Session 或 Token 认证。

- **关键模块与接口说明:**
    - **内容 API (`api/content.py`)**: 提供内容列表（分页、筛选）、详情、创建、点赞、评论接口。
    - **论坛 API (`api/forum.py`)**: 提供最新主题、主题列表（分页、筛选）、创建主题、帖子列表（分页）、创建回复接口。
    - **非遗项目 API (`api/heritage.py`)**: 提供非遗项目列表（分页、筛选）、详情、创建（教师/管理员权限）接口。
    - **通知 API (`api/notification.py`)**: 提供获取未读通知数、标记已读、标记全部已读、获取未读私信数接口。
    - **用户 API (`api/user.py`)**: 提供获取用户资料（含统计）、用户内容列表（分页）、用户收藏列表（分页）、修改密码接口。

3.6 实时通信事件 (`app/socket_events.py`)

平台使用 Flask-SocketIO 扩展实现基于 WebSocket 的实时通信功能。

- **核心库**: Flask-SocketIO。
- **错误处理**: 使用 `@websocket_error_handler` 装饰器统一处理。
- **用户状态管理**: 调用 `WebSocketManager` 进行连接注册/注销。
- **关键事件处理函数说明:**
    - **`connect` / `disconnect`** : 处理客户端连接与断开，管理用户在线状态。
    - **`join_group` / `leave_group`** : 处理客户端加入/离开群组房间。

- `send_group_message`：核心功能，处理群组消息发送。验证权限，保存消息到数据库，为成员创建已读状态记录，并向群组房间广播新消息。
- `delete_message`：处理删除群组消息请求。验证权限（发送者或管理员），物理删除消息，并向群组房间广播删除事件。

3.7 表单定义（`app/forms/`）

项目使用 Flask-WTF 扩展来创建和管理 Web 表单。

- **核心库**: Flask-WTF (基于 WTForms)。
- **功能**: 定义字段、验证规则、CSRF 保护、模板渲染辅助。
- **关键表单类**: 包含认证、内容、评论、非遗项目、论坛、消息（私信、群组、广播）、通知、用户（资料、密码、管理）等各类表单，部分表单包含自定义验证逻辑和动态选项加载。

3.8 应用初始化与配置（`app/__init__.py`）

项目的入口和组装逻辑位于 `app/__init__.py` 文件中，其核心是 `create_app` 工厂函数。

- **工厂模式**: 使用 `create_app` 函数创建应用实例，支持不同配置环境。
- **扩展初始化**: 初始化 SQLAlchemy, LoginManager, CSRFProtect, Migrate, SocketIO 等核心扩展。
- **自定义模块初始化**: 调用 `setup_logging`，`setup_security`，`init_websocket_manager`。
- **蓝图注册**: 注册 API 蓝图（豁免 CSRF）和所有视图蓝图，设置 URL 前缀。
- **错误处理**: 注册全局 HTTP 错误处理器。
- **上下文处理器**: 注册 `common_data` 向模板注入通用数据。
- **模板过滤器**: 注册 `markdown` 和 `nl2br` 过滤器。
- **Socket.IO 事件导入**: 在应用上下文中导入 `socket_events` 模块以注册事件处理器。

4. 核心代码示例

以下选取了项目中部分关键功能的代码片段，以展示核心业务逻辑的实现方式。

4.1 内容创建与文件处理（`app/routes/content.py - create` 函数)

此函数展示了典型的 Web 表单处理流程，包括表单验证、根据用户输入处理不同类型的内容（文章、富文本、图片、视频），并调用自定义的文件处理函数 `save_file` 来保存上传的文件和封面。

```python
@content_bp.route('/create', methods=['GET', 'POST'])
@login_required # 确保用户已登录
def create():
    """创建内容页面"""
    form = ContentForm() # 实例化内容表单

    # 动态加载非遗项目选项到 SelectField
    form.heritage_id.choices = [(h.id, h.name) for h in HeritageItem.query.all()]

    if form.validate_on_submit(): # 处理 POST 请求且表单验证通过
        try:
            # 创建 Content 模型实例，填充基本信息
            content = Content(
                title=form.title.data,
                heritage_id=form.heritage_id.data,
                user_id=current_user.id,
                content_type=form.content_type.data
            )

            # 处理封面图片上传（所有类型内容都可能有封面）
            if form.cover_image.data:
                current_app.logger.info(f"处理封面图片上传: {form.cover_image.data.filename}")
                # 调用自定义的文件保存函数处理图片
                cover_path = save_file(form.cover_image.data, 'image')
                if cover_path:
                    current_app.logger.info(f"封面图片上传成功，路径: {cover_path}")
                    content.cover_image = cover_path # 保存相对路径
                else:
                    current_app.logger.error("封面图片上传失败")
                    flash('封面图片上传失败', 'danger')

            # 根据选择的内容类型，处理特定字段和文件
            if form.content_type.data == 'article':
                content.text_content = form.text_content.data # 保存纯文本
            elif form.content_type.data == 'multimedia':
                content.rich_content = form.rich_content.data # 保存富文本 HTML
            elif form.content_type.data in ['image', 'video']:
                if form.file.data: # 检查是否有文件上传
                    current_app.logger.info(f"处理文件上传: {form.file.data.filename}, 类型: {for
                    # 调用自定义的文件保存函数，根据类型保存图片或视频
                    file_path = save_file(form.file.data, form.content_type.data)
                    if file_path:
                        current_app.logger.info(f"文件上传成功，路径: {file_path}")
```

```
                    content.file_path = file_path # 保存相对路径
                else:
                    current_app.logger.error("文件上传失败")
                    flash('文件上传失败', 'danger')
                    # 上传失败则重新渲染表单
                    return render_template('content/create.html', form=form)
            else:
                # 如果选择了图片/视频类型但未上传文件，记录警告
                current_app.logger.warning(f"未检测到文件上传")
                flash('请为图片或视频类型上传文件', 'warning')
                return render_template('content/create.html', form=form)

        # 将新内容对象添加到数据库会话
        db.session.add(content)
        # 提交事务，保存到数据库
        db.session.commit()
        current_app.logger.info(f"内容创建成功：ID={content.id}, 标题={content.title}")

        flash('内容创建成功', 'success')
        # 重定向到新创建内容的详情页
        return redirect(url_for('content.detail', id=content.id))

    except Exception as e:
        # 如果发生异常，回滚事务
        db.session.rollback()
        current_app.logger.error(f"创建内容失败：{str(e)}")
        current_app.logger.error(traceback.format_exc()) # 记录详细错误堆栈
        flash('创建内容失败，请稍后重试', 'danger')

# 处理 GET 请求或表单验证失败时，渲染创建页面模板
return render_template('content/create.html', form=form)
```

4.2 论坛主题详情页与嵌套回复（app/routes/forum.py - topic 函数片段）

此函数展示了如何处理一个包含复杂数据关联和嵌套结构的页面。它查询主题信息、顶级帖子及其作者，并进一步为每个顶级帖子查询其下的所有回复（嵌套评论）及相关用户信息，最后将整理好的数据传递给模板进行渲染。同时，它还处理回复表单的提交。

```python
@forum_bp.route('/topic/<int:id>', methods=['GET', 'POST'])
def topic(id):
    """主题详情页 - 支持嵌套回复功能"""
    topic = ForumTopic.query.get_or_404(id) # 获取主题，不存在则返回404

    # 增加浏览次数 (在最后提交事务)
    topic.views += 1

    # 回复表单
    form = PostForm()

    # 处理 POST 请求 (提交回复)
    if form.validate_on_submit() and current_user.is_authenticated:
        # ... (回复处理逻辑，包括创建 Post、处理 parent_id 和 reply_to_user_id、更新 topic.last_ac
        # (此处省略回复处理代码，参见 routes/forum.py 完整代码)
        try:
            if topic.is_closed and not current_user.is_admin:
                flash('该主题已关闭，无法回复', 'warning')
                return redirect(url_for('forum.topic', id=id))

            post = ForumPost(
                topic_id=id,
                user_id=current_user.id,
                content=form.content.data
            )

            if form.parent_id.data and form.parent_id.data.isdigit():
                parent_id = int(form.parent_id.data)
                parent_post = ForumPost.query.get(parent_id)
                if parent_post and parent_post.topic_id == id:
                    post.parent_id = parent_id
                    if form.reply_to_user_id.data and form.reply_to_user_id.data.isdigit():
                        reply_to_user_id = int(form.reply_to_user_id.data)
                        user = User.query.get(reply_to_user_id)
                        if user:
                            post.reply_to_user_id = reply_to_user_id

            db.session.add(post)
            topic.last_activity = post.created_at

            if current_user.id != topic.user_id:
                from app.routes.notification import send_notification
                content_msg = f"{current_user.username} 在主题 \"{topic.title}\" 中发表了回复"
```

```
            send_notification(
                user_id=topic.user_id,
                content=content_msg,
                notification_type='reply',
                link=url_for('forum.topic', id=id),
                sender_id=current_user.id
            )

        db.session.commit() # 提交回复和浏览量更新
        flash('回复成功', 'success')
        return redirect(url_for('forum.topic', id=id))

    except Exception as e:
        db.session.rollback()
        current_app.logger.error(f"发表回复失败: {str(e)}")
        flash('发表回复失败，请稍后重试', 'danger')


# --- 处理 GET 请求，查询并组织页面数据 ---
page = request.args.get('page', 1, type=int) # 获取分页参数

# 查询顶级帖子(parent_id is None)并 JOIN 加载作者信息
posts_query = db.session.query(
    ForumPost,
    User.username.label('author_name'), # 获取作者用户名
    User.avatar.label('author_avatar')  # 获取作者头像
).outerjoin(User, ForumPost.user_id == User.id # 左连接 User 表
).filter(ForumPost.topic_id == id, ForumPost.parent_id == None # 筛选条件
).order_by(ForumPost.created_at.asc()) # 按创建时间升序

# 对顶级帖子进行分页
posts_pagination = posts_query.paginate(page=page, per_page=20, error_out=False)

# 获取主题创建者信息
creator = User.query.get(topic.user_id)
topic_data = { # 整理主题信息传递给模板
    'id': topic.id,
    'title': topic.title,
    'category': topic.category,
    'is_pinned': topic.is_pinned,
    'is_closed': topic.is_closed,
    'views': topic.views,
    'post_count': topic.post_count, # 依赖模型中的计算属性或查询
```

```
        'created_at': topic.created_at,
        'last_activity': topic.last_activity,
        'creator': creator.username if creator else "未知用户"
    }

    # 准备包含嵌套回复的帖子数据列表
    posts_with_authors = []
    for post, author_name, author_avatar in posts_pagination.items: # 遍历当前页的顶级帖子
        # --- 查询当前顶级帖子的所有回复 ---
        replies_data = []

        # 使用 SQLAlchemy 的 aliased() 函数为 User 表创建别名，以处理自连接查询中的歧义
        ReplyUser = aliased(User)       # 回复者的 User 表别名
        ReplyToUser = aliased(User)     # 被回复者的 User 表别名

        # 查询回复及其作者、被回复者用户名
        replies = db.session.query(
            ForumPost,                          # 回复帖子对象
            ReplyUser.username.label('author_name'), # 回复者用户名
            ReplyUser.avatar.label('author_avatar'), # 回复者头像
            ReplyToUser.username.label('reply_to_username') # 被回复者用户名
        ).outerjoin(ReplyUser, ForumPost.user_id == ReplyUser.id # 连接回复者
        ).outerjoin(ReplyToUser, ForumPost.reply_to_user_id == ReplyToUser.id # 连接被回复者
        ).filter(ForumPost.parent_id == post.id # 筛选条件：父帖子ID是当前顶级帖子ID
        ).order_by(ForumPost.created_at.asc()).all() # 按时间排序

        # 整理回复数据
        for reply, reply_author, reply_avatar, reply_to_username in replies:
            # 如果没有明确的被回复者，则默认为回复顶级帖子的作者
            reply_to_name = reply_to_username if reply_to_username else author_name

            replies_data.append({
                'id': reply.id,
                'content': reply.content,
                'created_at': reply.created_at,
                'updated_at': reply.updated_at,
                'author': reply_author or "未知用户",
                'author_avatar': reply_avatar,
                'author_id': reply.user_id,
                'reply_to_name': reply_to_name,
                'reply_to_user_id': reply.reply_to_user_id
            })
        # --- 回复查询结束 ---
```

```
        # 将顶级帖子及其整理好的回复列表添加到最终结果中
        posts_with_authors.append({
            'id': post.id,
            'content': post.content,
            'created_at': post.created_at,
            'updated_at': post.updated_at,
            'author': author_name or "未知用户",
            'author_id': post.user_id,
            'author_avatar': author_avatar,
            'replies': replies_data # 嵌套的回复列表
        })

    # 提交数据库更改（主要是浏览量增加）
    try:
        db.session.commit()
    except Exception as e:
        db.session.rollback()
        current_app.logger.error(f"更新主题浏览量失败: {str(e)}")

    # 渲染模板，传递整理好的数据
    return render_template('forum/topic.html',
                           topic=topic_data,
                           posts=posts_with_authors,
                           pagination=posts_pagination,
                           form=form)
```

4.3 即时消息发送（`app/socket_events.py - handle_group_message` 函数）

此函数是实时聊天功能的核心，处理客户端通过 WebSocket 发送群组消息的事件。它验证用户权限，将消息存入数据库，为每个群组成员创建已读状态记录，并使用 `socketio.emit` 将新消息实时广播给群组内的所有在线用户。

```python
@socketio.on('send_group_message') # 监听 'send_group_message' 事件
@websocket_error_handler # 应用统一的 WebSocket 错误处理装饰器
def handle_group_message(data):
    """处理群组消息发送事件"""
    if not current_user.is_authenticated: # 检查用户是否登录
        return {'status': 'error', 'message': '请先登录再发送消息'}

    group_id = data.get('group_id') # 获取客户端发送的群组 ID
    content = data.get('content')   # 获取消息内容

    if not group_id or not content:
        return {'status': 'error', 'message': '消息内容或群组ID不能为空'}

    try:
        # 查询群组是否存在
        group = MessageGroup.query.get(group_id)
        if not group:
            return {'status': 'error', 'message': '群组不存在'}

        # 检查当前用户是否为该群组成员
        is_member = UserGroup.query.filter_by(user_id=current_user.id, group_id=group_id).first(

        if not is_member:
            return {'status': 'error', 'message': '您不是该群组的成员'}

        # --- 数据库操作 ---
        # 创建新消息记录
        new_message = Message(
            sender_id=current_user.id,
            content=content,
            group_id=group_id,
            message_type='group', # 标记为群组消息
            created_at=datetime.datetime.now() # 使用数据库服务器时间可能更佳，但这里用应用时间
        )
        db.session.add(new_message)
        db.session.flush()  # 刷新会话以获取新消息的 ID (new_message.id)

        # 为群组内每个成员创建消息已读状态记录 (MessageReadStatus)
        for member_membership in group.members: # group.members 是 UserGroup 对象的集合
            # 发送者本人自动标记为已读
            is_read = member_membership.user_id == current_user.id
            read_at = datetime.datetime.now() if is_read else None
```

```
            status = MessageReadStatus(
                message_id=new_message.id, # 关联到刚创建的消息
                user_id=member_membership.user_id, # 关联到群组成员
                is_read=is_read,
                read_at=read_at
            )
            db.session.add(status)

        db.session.commit() # 提交所有数据库更改 (消息和已读状态)
        # --- 数据库操作结束 ---

        # 准备要发送给客户端的消息数据
        message_data = {
            'id': new_message.id,
            'content': new_message.content,
            'sender_id': new_message.sender_id,
            'sender_username': current_user.username,
            'sender_avatar': current_user.avatar, # 获取用户头像
            'created_at': new_message.created_at.strftime('%Y-%m-%d %H:%M:%S'), # 格式化时间
            'group_id': group_id
        }

        # --- Socket.IO 广播 ---
        # 定义目标房间名称
        room_name = f"group_{group_id}"
        # 使用 socketio.emit 向指定房间广播 'new_group_message' 事件及消息数据
        emit('new_group_message', message_data, to=room_name)
        # --- Socket.IO 广播结束 ---

        # 向发送消息的客户端返回成功状态和消息数据
        return {'status': 'success', 'message': '消息发送成功', 'data': message_data}

    except Exception as e:
        db.session.rollback() # 发生错误时回滚数据库
        current_app.logger.error(f"发送群组消息出错: {str(e)}")
        # 向发送消息的客户端返回错误信息
        return {'status': 'error', 'message': f'发送消息失败: {str(e)}'}
```

4.4 用户管理列表与 API (`app/routes/user.py - manage_users` 和 `api_users` 函数)

这两个函数协作实现了后台用户管理功能。 `manage_users` 负责渲染管理页面，包含筛选和搜索表单； `api_users` 则提供了一个 JSON API 接口，用于异步加载、筛选和搜索用户数据，这通常用于提升大型列表页面的用户体验。

```python
@user_bp.route('/manage_users')
@login_required
@admin_required # 仅限管理员访问
def manage_users():
    """用户管理页面 - 渲染基本页面结构和筛选/搜索表单"""
    # 获取可能的筛选和搜索参数，用于初始化前端控件或显示当前状态
    page = request.args.get('page', 1, type=int)
    role = request.args.get('role')
    search = request.args.get('search')
    per_page = 10 # 每页显示数量

    # 注意：此视图函数主要负责渲染页面框架
    # 实际的用户数据列表通常通过下面的 api_users 接口异步加载
    # 但为了在禁用 JS 的情况下也能工作，或作为初始加载，可以保留查询逻辑

    query = User.query
    if role:
        query = query.filter(User.role == role)
    if search:
        query = query.filter(
            (User.username.like(f'%{search}%')) |
            (User.email.like(f'%{search}%'))
        )
    pagination = query.order_by(User.created_at.desc()).paginate(
        page=page, per_page=per_page, error_out=False)
    users = pagination.items

    # 渲染管理页面模板，传递参数用于初始化筛选/搜索状态
    return render_template('user/manage_users.html',
                           users=users, # 初始加载或非JS环境下的用户列表
                           pagination=pagination, # 分页对象
                           current_role=role, # 当前筛选的角色
                           search=search) # 当前搜索关键词

@user_bp.route('/api/users')
@login_required
@admin_required # 仅限管理员访问
def api_users():
    """用户API - 返回JSON格式的用户列表，用于AJAX加载、搜索和筛选"""
    page = request.args.get('page', 1, type=int)         # 获取页码
    per_page = request.args.get('per_page', 10, type=int) # 获取每页数量
    role = request.args.get('role')                      # 获取角色筛选参数
    search = request.args.get('search')                  # 获取搜索关键词
```

```python
query = User.query # 基础查询

# 应用角色筛选
if role:
    query = query.filter(User.role == role)

# 应用搜索条件 (用户名 或 邮箱 模糊匹配)
if search:
    query = query.filter(
        (User.username.like(f'%{search}%')) |
        (User.email.like(f'%{search}%'))
    )

# 执行分页查询
pagination = query.order_by(User.created_at.desc()).paginate(
    page=page, per_page=per_page, error_out=False)

# 准备返回的 JSON 数据
users_data = []
for user in pagination.items: # 遍历当前页的用户
    users_data.append({ # 构建每个用户的数据字典
        'id': user.id,
        'username': user.username,
        'email': user.email,
        'avatar': user.avatar or url_for('static', filename='img/default-avatar.jpg'), # 提
        'role': user.role,
        'created_at': user.created_at.strftime('%Y-%m-%d'), # 格式化日期
        'is_admin': user.is_admin,   # 便于前端判断
        'is_teacher': user.is_teacher # 便于前端判断
    })

# 返回包含用户列表和分页信息的 JSON 响应
return jsonify({
    'users': users_data,          # 当前页用户数据列表
    'total': pagination.total,    # 总记录数
    'pages': pagination.pages,    # 总页数
    'current_page': pagination.page # 当前页码
})
```

5. 参考文献

1. Flask Documentation. (2024). Flask Web Development, one drop at a time. https://flask.palletsprojects.com/
2. SQLAlchemy Documentation. (2024). The Python SQL Toolkit and Object Relational Mapper. https://docs.sqlalchemy.org/
3. Flask-SocketIO Documentation. (2024). Socket.IO integration for Flask applications. https://flask-socketio.readthedocs.io/
4. Pillow Documentation. (2024). The friendly PIL fork (Python Imaging Library). https://pillow.readthedocs.io/
5. Bootstrap Documentation. (2024). Build fast, responsive sites with Bootstrap. https://getbootstrap.com/docs/
6. Flask-Login Documentation. (2024). Flask user session management. https://flask-login.readthedocs.io/
7. Flask-WTF Documentation. (2024). Simple integration of Flask and WTForms. https://flask-wtf.readthedocs.io/
8. Bleach Documentation. (2024). An easy, HTML5 sanitizing library. https://bleach.readthedocs.io/
9. Flask-Talisman Documentation. (2024). HTTP security headers for Flask. https://github.com/GoogleCloudPlatform/flask-talisman
10. Flask-Limiter Documentation. (2024). Rate limiting for Flask applications. https://flask-limiter.readthedocs.io/
11. UNESCO. (2024). What is Intangible Cultural Heritage? https://ich.unesco.org/en/what-is-intangible-heritage-00003
12. Markdown Guide. (Syntax Documentation). https://www.markdownguide.org/ (针对模板过滤器的参考)

6. 总结

“体育非遗数字展示与互动平台”项目基于 Flask 框架及其丰富的生态系统，成功构建了一个面向体育类非物质文化遗产保护、传承、研究与推广的综合性数字化平台。该平台通过**系统化的信息管理**、**多媒体的内容呈现**、**实时的在线交流**以及**精细化的权限控制**，有效应对了当前体育非遗领域面临的传承困境、资料分散和互动不足等挑战。

在技术实现上，项目采用了成熟稳健的技术栈：后端使用 Flask 结合 SQLAlchemy ORM 进行数据持久化，Flask-Login 和 Flask-WTF 分别处理用户认证与表单安全，Flask-SocketIO 实现了高效的 WebSocket 实时通信；前端利用 HTML、CSS、JavaScript 和 Bootstrap 构建了友好的用户界面。图片处理方面集成了 Pillow 库进行自动压缩和水印添加，并通过 Bleach 库保障富文本内容的安全性。同

时，项目在日志记录、错误处理、安全防护（CSRF、安全头、速率限制）等方面也进行了周全考虑，保证了系统的稳定性和可靠性。

功能层面，平台不仅提供了非遗项目和相关内容的展示、检索功能，更创新性地融合了**社区互动**（主题论坛、嵌套评论、点赞收藏）和**即时通讯**（私聊、群聊、已读状态跟踪、群发广播）功能，显著增强了用户粘性和参与度。针对不同用户角色（管理员、教师、学生）设计的权限体系，确保了平台管理的有序性和内容的专业性。后台管理功能（用户管理、内容管理、数据统计 API 等）也为平台的维护和运营提供了有力支持。

综上所述，“体育非遗数字展示与互动平台”是一个**设计完整、功能丰富、技术选型合理、具有较强实用性和创新性**的项目。它不仅为体育非遗的数字化保护和传承提供了有效的技术解决方案，也为相关领域的教学、研究和爱好者社群构建了一个富有活力的在线家园，具备良好的应用前景和进一步扩展的潜力。