# **基于情绪识别与记忆构建的旅游社交互动系统及方法**

## **文档信息**

|  |  |
| --- | --- |
| 信息项 | 内容 |
| 文档版本 | v2.1.0 |
| 创建日期 | 2024-12-19 |
| 最后更新 | 2024-12-20 |
| 文档状态 | 正式版 |
| 作者 | 约旅平台技术团队 |
| 审核状态 | 已审核 |
| 审核人 | 约旅平台技术团队 |
| 文档类型 | 专利申请书 |
| 适用范围 | 国内专利申请 |

## **版本历史**

### **v2.1.0 版本更新记录**

|  |  |  |
| --- | --- | --- |
| 更新时间 | 更新类型 | 变更详情 |
| 2024-12-20 | 文档优化 | **完善文档信息结构**: 对文档的整体结构与信息组织方式进行了优化，提升可读性。 |
| 2024-12-20 | 新增 | **审核人信息字段**: 新增审核人字段，旨在加强文档审核流程的规范化管理。 |
| 2024-12-20 | 新增 | **文档类型和适用范围说明**: 补充了文档类型和适用范围的说明，以更清晰地界定文档用途。 |
| 2024-12-20 | 修复 | **版本历史格式标准化**: 对版本历史的记录格式进行了统一和标准化，确保版本信息的一致性。 |
| 2024-12-20 | 变更 | **审核状态更新为已审核**: 文档已通过内部审核流程，状态更新为“已审核”。 |
| 2024-12-20 | 变更 | **遵循最新文档管理规范**: 确保文档内容与格式均符合项目最新的文档管理规范与标准。 |

### **v2.0.0 版本更新记录**

|  |  |  |
| --- | --- | --- |
| 更新时间 | 更新类型 | 变更详情 |
| 2024-12-19 | 重大更新 | **完全重构专利申请书结构**: 基于专利申请规范要求，对文档架构进行全面重新设计，确保结构清晰、层次分明。 |
| 2024-12-19 | 新增 | **多模态数据采集模块详细技术方案**: 详细阐述了图像、音频、文本等多模态数据的采集、预处理及存储方案。 |
| 2024-12-19 | 新增 | **多模态情绪识别引擎完整实现方案**: 深入描述了基于深度学习的多模态融合情绪识别算法及其实现细节。 |
| 2024-12-19 | 新增 | **社交互动推荐引擎核心算法**: 详细说明了基于用户画像和情境感知的个性化社交推荐算法实现方案。 |
| 2024-12-19 | 新增 | **记忆构建引擎技术架构**: 阐述了基于知识图谱的用户互动记忆构建机制及其技术实现方案。 |
| 2024-12-19 | 新增 | **社交图谱构建与演化引擎**: 详细描述了动态社交关系图谱的构建算法与实时更新机制。 |
| 2024-12-19 | 新增 | **5个详细实施例说明**: 增加了5个完整的技术实施案例，涵盖核心功能模块的具体实现过程。 |
| 2024-12-19 | 新增 | **10项完整权利要求**: 系统性地提出了10项专利权利要求，全面保护核心技术创新点。 |
| 2024-12-19 | 优化 | **技术创新点突出表达**: 重点突出了系统的技术创新性，强化了专利保护的关键技术特征。 |
| 2024-12-19 | 修复 | **文档结构标准化**: 统一规范了文档格式，确保符合专利申请的标准化要求。 |
| 2024-12-19 | 变更 | **遵循专利申请书标准格式**: 严格按照国家知识产权局的专利申请规范要求进行文档编排。 |

### **v1.0.0 版本记录（历史版本）**

|  |  |  |
| --- | --- | --- |
| 更新时间 | 更新类型 | 变更详情 |
| 2024-12-10 | 初始创建 | **创建文档初始版本**: 初步建立文档框架。 |
| (历史问题) | 问题记录 | **内容丢失、结构不完整、核心方案缺失 (已废弃)**: 该历史版本存在严重问题：内容丢失（仅保留标题）、文档结构不完整、缺少核心技术方案。因此该版本已被废弃。 |

## **专利名称**

基于情绪识别与记忆构建的旅游社交互动系统及方法

## **技术领域**

本发明属于信息技术领域，具体涉及人工智能（Artificial Intelligence, AI）、社交网络服务（Social Networking Services, SNS）、情绪计算（Affective Computing）以及认知科学中的记忆建模（Memory Modeling）等多个交叉学科。更具体地说，本发明提出了一种创新的、基于用户情绪状态的实时识别与长期行为记忆的智能构建，从而实现更精准、更具情感共鸣的旅游社交互动系统及其配套的实现方法。该技术旨在通过深度理解用户在旅行过程中的情感变化和体验沉淀，优化社交匹配的质量与互动体验的深度，为用户在旅途中创造更富有意义的连接。

## **背景技术**

近年来，随着移动互联网的普及和社交媒体的兴盛，在线旅游行业得到了飞速发展。网红经济的崛起进一步推动了旅游与社交的深度融合，用户不再仅仅满足于传统的观光游览，而是更加追求个性化、深度化、情感化的旅行体验，并乐于在社交平台分享自己的旅途见闻与感受。旅游社交互动已成为提升用户粘性、丰富旅行体验、乃至驱动旅游消费的关键因素。

然而，现有的旅游社交产品或传统社交平台的推荐机制在满足用户深层次情感与社交需求方面仍存在诸多不足，具体表现在：

1. **情绪感知的缺失与肤浅**：传统的社交推荐系统大多依赖于用户的历史行为数据（如点击、浏览、购买记录）、显式标签（如兴趣爱好）或简单的地理位置信息进行匹配。这些系统普遍缺乏对用户在特定情境下（尤其是旅行这种高度情绪化的场景中）的实时情绪状态的深度感知和理解。因此，它们难以提供真正“懂你”的、能够触动情感共鸣的社交匹配和互动建议，例如，无法准确识别用户是处于兴奋、疲惫、孤独还是渴望分享的状态，并据此推荐合适的同伴或活动。

2. **共享记忆构建的系统性缺乏与体验碎片化**：旅行往往是与他人共同创造回忆的过程。然而，当前的旅游社交平台大多侧重于即时性的信息分享，缺乏对用户间共同经历的系统性记录、组织和再现机制。用户的宝贵记忆往往散落在各自的设备或不同的应用中，难以形成连贯、生动、可供回味的共享记忆。这种体验的碎片化不仅削弱了社交连接的深度，也使得基于共同回忆的长期关系维系变得困难。

3. **社交互动推荐算法的单一性与场景适应性不足**：许多平台的社交推荐算法模型较为单一，或者泛化能力不足，难以适应旅游场景高度动态、情境多变的特点。例如，在不同的旅行阶段（规划期、旅途中、结束后）、不同的旅行主题（探险、休闲、文化体验）、不同的结伴形式（独自旅行、情侣出游、家庭旅行）下，用户的社交需求和偏好可能截然不同。现有算法往往无法进行精细化的场景感知和个性化匹配。

4. **动态社交图谱构建与演化机制的滞后**：人际关系是动态变化的，尤其是在旅行这种短期高频互动的环境中，新的社交连接可能迅速建立，旧的关系也可能得到加强或减弱。传统的社交图谱构建往往是静态的或更新缓慢的，难以实时捕捉和反映基于用户情绪变化、互动行为和共同经历所引发的社交关系演化，从而错失了促进关系深化或拓展新连接的机会。

5. **旅游社交数据的多模态融合技术不成熟**：用户在旅行中的表达是多模态的，包括文字（游记、评论、聊天）、图像（照片、视频）、语音（语音消息、实时通话）、行为（点赞、分享、打卡）乃至生理信号（通过可穿戴设备获取的心率、步数等）。如何有效地融合这些来自不同渠道、不同结构的多模态数据，以更全面、更准确地理解用户状态和社交意图，是提升互动质量的关键。现有技术在多模态数据融合的深度和精度上仍有较大提升空间，简单的数据拼接难以发挥协同效应。

综上所述，现有技术中，尚未出现一种能够将实时情绪识别、系统性记忆构建以及动态社交图谱演化三者有机结合，并专门针对旅游场景进行优化的社交互动系统。特别是在情绪驱动的社交互动技术方面，仍处于探索的初级阶段，远未能满足用户对高质量、深层次旅游社交体验的期待。因此，研发一种能够克服上述缺陷的新型旅游社交互动系统具有重要的理论意义和广阔的应用前景。

## **发明内容**

**发明目的：**

本发明的主要目的在于克服现有技术的不足，提供一种全新的基于情绪识别与记忆构建的旅游社交互动系统及其实现方法。该系统旨在通过深度感知用户在旅行过程中的实时情绪状态，并结合对用户个体及群体共享记忆的智能构建与管理，实现更加精准、个性化、富有情感共鸣的社交匹配与互动推荐，从而显著提升用户的旅游社交体验，促进更有意义的人际连接，并最终增强用户对平台的粘性与满意度。

具体而言，本发明致力于解决以下核心技术问题：

1. 如何准确、实时地识别用户在多变的旅游场景下的多模态情绪状态？

2. 如何系统性地为用户构建个性化的旅行记忆，并促进同行者之间共享记忆的形成与沉淀？

3. 如何将情绪识别结果与记忆模型应用于社交推荐，实现情绪驱动的、与用户当下心境高度契合的社交匹配？

4. 如何动态构建和演化社交图谱，以反映旅行中人际关系的变化和发展趋势？

5. 如何设计一个集成上述功能的、高效、可扩展的系统架构？

**技术方案概述：**

为实现上述目的，本发明提出了一种社交互动引擎，该引擎是整个旅游社交互动系统的核心。它通过精密集成先进的情绪识别技术和创新的记忆构建机制，驱动一个高度智能化的旅游社交互动系统。该系统主要由以下相互协作的核心技术组件构成，这些组件共同工作，以实现本发明的目标：

（后续将详细阐述各个模块的具体构成和工作原理）

### **技术方案**

#### **1. 多模态数据采集模块**

多模态数据采集模块是本系统的基础数据获取层，负责全方位、多维度地采集用户在旅行过程中产生的各类数据信息。该模块采用分布式采集架构，支持实时数据流处理和批量数据处理，并实现了数据的预处理、清洗、标准化等基础功能。具体包含以下核心采集单元：

##### **1.1 文本数据采集单元**

• **功能描述**：负责采集、解析和预处理各类文本形式的用户生成内容

• **数据来源**：

• 用户发布的旅游体验、游记、评论

• 实时聊天记录和私信内容

• 社交媒体平台的文字分享和互动

• 用户对景点、活动、其他用户的评价

• **技术特点**：

• 支持多语言文本识别和编码转换

• 实现敏感信息自动过滤

• 基于NLP的文本分词和语义标注

• 文本质量评估和垃圾内容过滤

##### **1.2 图像数据采集单元**

• **功能描述**：采集和处理用户在旅行过程中产生的各类图像数据

• **数据类型**：

• 旅游照片和视频片段

• 用户表情和肢体语言图像

• 场景和环境图片

• AR/VR场景中的虚拟形象表情

• **技术特点**：

• 支持多种图像格式的实时压缩和转换

• 图像质量评估和优化

• 基于深度学习的图像预处理

• 隐私内容智能识别和保护

##### **1.3 音频数据采集单元**

• **功能描述**：获取和处理用户的语音数据及环境音频信息

• **数据来源**：

• 语音消息和语音通话

• 环境背景音

• 用户情绪语调

• 群组语音互动

• **技术特点**：

• 实时音频流采集和编码

• 噪声消除和音质增强

• 语音活动检测（VAD）

• 多声道音频处理

##### **1.4 行为数据采集单元**

• **功能描述**：记录和分析用户在平台上的各类交互行为

• **采集维度**：

• 用户操作行为（点击、滑动、停留）

• 社交互动行为（关注、点赞、评论、分享）

• 旅行轨迹数据（位置变化、停留时间）

• 消费行为数据（预订、购买、支付）

• **技术特点**：

• 行为序列实时捕获

• 用户意图识别

• 异常行为检测

• 行为模式挖掘

##### **1.5 生理信号采集单元**

• **功能描述**：通过可穿戴设备采集用户的生理状态数据

• **信号类型**：

• 心率和心率变异性（HRV）

• 皮肤电导反应（GSR）

• 体温变化

• 运动状态数据

• **技术特点**：

• 多设备协议适配

• 生理信号实时同步

• 数据可靠性验证

• 隐私数据加密存储

##### **1.6 数据质量保障机制**

• **数据验证**：

• 实时数据质量检测

• 异常值识别与处理

• 数据完整性校验

• **隐私保护**：

• 数据脱敏处理

• 用户授权管理

• 数据访问控制

• **存储优化**：

• 分布式存储架构

• 数据压缩策略

• 冷热数据分层

##### **1.7 系统集成接口**

• **标准化数据输出**：

• 统一数据格式规范

• 跨模态数据对齐

• 实时/批量数据接口

• **监控与告警**：

• 数据采集状态监控

• 系统性能指标跟踪

• 异常情况实时告警

#### **2. 多模态情绪识别引擎**

本发明的核心技术之一是多模态情绪识别引擎，其部分代码实现参见 /Users/yingyang/Documents/project/mdworld/TripWithNetCelebs/专利申请文件/国内专利申请/code/EmotionRecognitionEngine.py。该引擎能够综合处理来自面部表情、文本和语音等多种模态的数据，以实现对用户情绪状态的精准识别。

多模态情绪识别引擎是本系统的核心技术模块之一，负责对来自不同模态的用户数据进行深度分析，实现对用户情绪状态的精准识别和量化表达。该引擎采用模块化设计，集成了多种先进的深度学习算法，并创新性地实现了跨模态的情绪特征融合与推理。

##### **2.1 面部表情分析单元**

**python**

class FaceEmotionModel(nn.Module):

    def \_\_init\_\_(self):

        super(FaceEmotionModel, self).\_\_init\_\_()

        # 使用预训练的ResNet作为特征提取器

        self.resnet = models.resnet50(pretrained=True)

        # 修改最后的全连接层以适应情绪分类

        self.resnet.fc = nn.Linear(2048, 7) # 7种基本情绪

        # 添加注意力机制

        self.attention = SpatialAttention()

    def forward(self, x):

        # 提取特征

        features = self.resnet.conv1(x)

        features = self.resnet.bn1(features)

        features = self.resnet.relu(features)

        features = self.resnet.maxpool(features)

        # 应用注意力机制

        attention\_weights = self.attention(features)

        features = features \* attention\_weights

        # 继续前向传播

        features = self.resnet.layer1(features)

        features = self.resnet.layer2(features)

        features = self.resnet.layer3(features)

        features = self.resnet.layer4(features)

        # 全局平均池化

        features = F.adaptive\_avg\_pool2d(features, (1, 1))

        features = torch.flatten(features, 1)

        # 情绪分类

        emotions = self.resnet.fc(features)

        return emotions

class SpatialAttention(nn.Module):

    def \_\_init\_\_(self):

        super(SpatialAttention, self).\_\_init\_\_()

        self.conv = nn.Conv2d(2, 1, kernel\_size=7, padding=3, bias=False) # Modified to take 2 input channels

        self.sigmoid = nn.Sigmoid()

    def forward(self, x):

        # 计算空间注意力权重

        avg\_out = torch.mean(x, dim=1, keepdim=True)

        max\_out, \_ = torch.max(x, dim=1, keepdim=True)

        attention = torch.cat([avg\_out, max\_out], dim=1)

        attention = self.conv(attention)

        return self.sigmoid(attention)

##### **2.1 面部表情分析单元**

• **功能描述**：  
实现对用户面部表情的实时捕捉、分析和情绪映射，支持静态图像和动态视频流处理。

• **核心技术实现**：

1. **面部检测与关键点定位**

• 采用改进的MTCNN（Multi-task Cascaded Convolutional Networks）实现快速准确的人脸检测

• 使用Dense U-Net架构进行68个面部关键点的精确定位

• 支持多角度、弱光环境下的稳定检测

2. **表情特征提取**

• 基于ResNet-50骨干网络的深度特征提取

• 引入空间注意力机制增强关键区域特征表达

• 设计多尺度特征金字塔实现细粒度表情变化捕捉

3. **情绪分类与量化**

• 采用FER2013+扩展数据集训练的深度分类器

• 支持7种基本情绪（快乐、悲伤、愤怒、惊讶、恐惧、厌恶、中性）的识别

• 创新性地引入情绪强度量化机制

4. **时序动态分析**

• 使用3D-CNN捕捉表情的时序变化特征

• 实现微表情检测与分析

• 表情变化趋势预测

##### **2.2 语音情绪分析单元**

**python**

class AudioEmotionModel(nn.Module):

    def \_\_init\_\_(self, num\_classes=7, input\_features=40, hidden\_dim=256, num\_layers=2, dropout\_rate=0.5):

        super(AudioEmotionModel, self).\_\_init\_\_()

        self.conv1 = nn.Conv2d(1, 32, kernel\_size=(3,3), stride=(1,1), padding=(1,1))

        self.relu1 = nn.ReLU()

        self.bn1 = nn.BatchNorm2d(32)

        self.pool1 = nn.MaxPool2d(kernel\_size=(2,2), stride=(2,2))

        self.conv2 = nn.Conv2d(32, 64, kernel\_size=(3,3), stride=(1,1), padding=(1,1))

        self.relu2 = nn.ReLU()

        self.bn2 = nn.BatchNorm2d(64)

        self.pool2 = nn.MaxPool2d(kernel\_size=(2,2), stride=(2,2))

        # Calculate the flattened size after conv and pooling layers

        # Assuming input\_features is the number of mel-spectrogram bins (e.g., 40 for MFCCs)

        # and a certain number of time frames (e.g., T)

        # After pool1: H\_out = input\_features / 2, W\_out = T / 2

        # After pool2: H\_out = input\_features / 4, W\_out = T / 4

        # lstm\_input\_size = 64 \* (input\_features // 4)

        # For simplicity, let's assume a fixed flattened size after conv layers, or calculate dynamically in forward

        # This needs to be adjusted based on the actual input dimensions of the spectrogram

        self.lstm\_input\_size = 64 \* (input\_features // 4)

        self.lstm = nn.LSTM(input\_size=self.lstm\_input\_size,

                              hidden\_size=hidden\_dim,

                              num\_layers=num\_layers,

                              batch\_first=True,

                              bidirectional=True,

                              dropout=dropout\_rate if num\_layers > 1 else 0)

        self.dropout = nn.Dropout(dropout\_rate)

        self.fc = nn.Linear(hidden\_dim \* 2, num\_classes) # \*2 for bidirectional

    def forward(self, x):

        # x shape: (batch\_size, 1, num\_mels, num\_frames)

        x = self.pool1(self.bn1(self.relu1(self.conv1(x))))

        x = self.pool2(self.bn2(self.relu2(self.conv2(x))))

        # Prepare for LSTM: (batch\_size, num\_frames\_reduced, features\_per\_frame)

        batch\_size, channels, height, width = x.size()

        x = x.permute(0, 3, 1, 2) # (batch\_size, num\_frames\_reduced, channels, height\_reduced)

        x = x.reshape(batch\_size, width, channels \* height) # (batch\_size, num\_frames\_reduced, lstm\_input\_size)

        lstm\_out, (h\_n, c\_n) = self.lstm(x)

        # Use the concatenation of the last hidden states from both directions

        # h\_n is (num\_layers \* num\_directions, batch, hidden\_dim)

        # We want the last layer's hidden states: h\_n[-2,:,:] (forward) and h\_n[-1,:,:] (backward)

        hidden = torch.cat((h\_n[-2,:,:], h\_n[-1,:,:]), dim=1)

        out = self.dropout(hidden)

        emotions = self.fc(out)

        return emotions

##### **2.2 语音情绪分析单元**

• **功能描述**：  
分析用户语音中蕴含的情绪信息，包括语调、语速、音量等声学特征，实现多语言环境下的情绪识别。

• **技术实现细节**：

1. **音频特征提取**

• 多维度声学特征计算（MFCC、色谱图、基频等）

• 语音活动检测（VAD）优化

• 环境噪声自适应消除

2. **语音情绪建模**

• 基于BiLSTM-Attention的时序特征建模

• 多语言声学模型适配

• 说话人情绪特征归一化

3. **上下文感知分析**

• 对话场景理解

• 情绪状态迁移追踪

• 多轮对话情绪演变分析

##### **2.3 文本情感分析单元**

**python**

class TextEmotionModel(nn.Module):

    def \_\_init\_\_(self, model\_name='bert-base-uncased', num\_classes=7, dropout\_rate=0.1):

        super(TextEmotionModel, self).\_\_init\_\_()

        self.bert = BertModel.from\_pretrained(model\_name)

        self.dropout = nn.Dropout(dropout\_rate)

        # BERT's default hidden size is 768 for 'bert-base-uncased'

        self.classifier = nn.Linear(self.bert.config.hidden\_size, num\_classes)

    def forward(self, input\_ids, attention\_mask, token\_type\_ids=None):

        outputs = self.bert(input\_ids=input\_ids,

                            attention\_mask=attention\_mask,

                            token\_type\_ids=token\_type\_ids)

        # Use the [CLS] token's output for classification

        pooled\_output = outputs.pooler\_output

        pooled\_output = self.dropout(pooled\_output)

        emotions = self.classifier(pooled\_output)

        return emotions

##### **2.3 文本情感分析单元**

• **功能描述**：  
对用户产生的文本内容进行深度语义分析，提取文本中明示和暗示的情感信息。

• **核心技术方案**：

1. **文本预处理**

• 多语言文本标准化

• 网络用语识别与转换

• 表情符号语义映射

2. **语义理解模型**

• 基于BERT-wwm-ext的中文预训练模型

• 融合RoBERTa和ALBERT的集成模型

• 领域适应性微调机制

3. **情感计算引擎**

• 多维度情感词典构建

• 上下文语义依存分析

• 讽刺与反语识别

4. **细粒度情感分析**

• 方面级情感分析

• 情感强度量化

• 情感倾向演变跟踪

##### **2.4 生理信号情绪分析单元**

• **功能描述**：  
通过分析用户的生理信号数据，实现对情绪状态的客观量化评估。

• **技术实现方案**：

1. **信号预处理**

• 生理信号降噪与平滑

• 异常值检测与修正

• 信号质量评估

2. **特征工程**

• 心率变异性（HRV）指标计算

• 皮肤电导特征提取

• 多源生理特征融合

3. **情绪状态建模**

• 基于深度学习的生理-情绪映射

• 个体差异自适应

• 情绪状态动态跟踪

##### **2.5 多模态融合策略**

**python**

class ModalityFusionModel(nn.Module):

    def \_\_init\_\_(self, num\_classes=7, feature\_dim=256, num\_heads=8, dropout\_rate=0.1):

        super(ModalityFusionModel, self).\_\_init\_\_()

        self.num\_modalities = 0 # Will be set dynamically

        self.feature\_dim = feature\_dim

        # Placeholder for alignment layers, to be created in align\_features

        self.alignment\_layers = nn.ModuleDict()

        # Multi-Head Attention for fusion

        self.attention = MultiHeadAttention(d\_model=feature\_dim, num\_heads=num\_heads, dropout=dropout\_rate)

        # Classifier for the fused features

        self.fusion\_classifier = nn.Linear(feature\_dim, num\_classes)

        self.dropout = nn.Dropout(dropout\_rate)

    def align\_features(self, modal\_features\_list):

        """Aligns features from different modalities to a common dimension."""

        aligned\_features = []

        self.num\_modalities = len(modal\_features\_list)

        for i, feature\_tensor in enumerate(modal\_features\_list):

            original\_dim = feature\_tensor.size(-1)

            modality\_name = f'modality\_{i}'

            if original\_dim != self.feature\_dim:

                if modality\_name not in self.alignment\_layers:

                    # Create and store alignment layer if it doesn't exist

                    self.alignment\_layers[modality\_name] = nn.Linear(original\_dim, self.feature\_dim).to(feature\_tensor.device)

                aligned\_feature = self.alignment\_layers[modality\_name](feature\_tensor)

            else:

                aligned\_feature = feature\_tensor

            aligned\_features.append(aligned\_feature.unsqueeze(1)) # Add sequence dimension for attention

        # Concatenate along the sequence dimension (dim=1)

        # Resulting shape: (batch\_size, num\_modalities, feature\_dim)

        return torch.cat(aligned\_features, dim=1)

    def forward(self, modal\_features\_list):

        # modal\_features\_list: a list of tensors, each from one modality

        # Each tensor shape: (batch\_size, modality\_specific\_feature\_dim)

        aligned\_modal\_features = self.align\_features(modal\_features\_list)

        # aligned\_modal\_features shape: (batch\_size, num\_modalities, feature\_dim)

        # Apply Multi-Head Attention

        # Q, K, V are all the aligned\_modal\_features for self-attention across modalities

        # The attention mechanism will weigh the importance of each modality

        context\_vector, attention\_weights = self.attention(aligned\_modal\_features, aligned\_modal\_features, aligned\_modal\_features)

        # context\_vector shape: (batch\_size, num\_modalities, feature\_dim)

        # We can average or take the [CLS] equivalent if we had one, or sum, or use another layer.

        # Here, let's average the features from different modalities after attention.

        fused\_representation = torch.mean(context\_vector, dim=1) # (batch\_size, feature\_dim)

        fused\_representation = self.dropout(fused\_representation)

        final\_emotions = self.fusion\_classifier(fused\_representation)

        return final\_emotions

class MultiHeadAttention(nn.Module):

    def \_\_init\_\_(self, d\_model, num\_heads, dropout=0.1):

        super(MultiHeadAttention, self).\_\_init\_\_()

        assert d\_model % num\_heads == 0, "d\_model must be divisible by num\_heads"

        self.d\_model = d\_model

        self.num\_heads = num\_heads

        self.d\_k = d\_model // num\_heads

        self.W\_q = nn.Linear(d\_model, d\_model)

        self.W\_k = nn.Linear(d\_model, d\_model)

        self.W\_v = nn.Linear(d\_model, d\_model)

        self.W\_o = nn.Linear(d\_model, d\_model)

        self.dropout = nn.Dropout(dropout)

        self.softmax = nn.Softmax(dim=-1)

    def forward(self, query, key, value, mask=None):

        batch\_size = query.size(0)

        # Linear projections

        Q = self.W\_q(query) # (batch\_size, seq\_len\_q, d\_model)

        K = self.W\_k(key) # (batch\_size, seq\_len\_k, d\_model)

        V = self.W\_v(value) # (batch\_size, seq\_len\_v, d\_model)

        # Split into multiple heads

        Q = Q.view(batch\_size, -1, self.num\_heads, self.d\_k).transpose(1, 2) # (batch\_size, num\_heads, seq\_len\_q, d\_k)

        K = K.view(batch\_size, -1, self.num\_heads, self.d\_k).transpose(1, 2) # (batch\_size, num\_heads, seq\_len\_k, d\_k)

        V = V.view(batch\_size, -1, self.num\_heads, self.d\_k).transpose(1, 2) # (batch\_size, num\_heads, seq\_len\_v, d\_k)

        # Scaled Dot-Product Attention

        scores = torch.matmul(Q, K.transpose(-2, -1)) / np.sqrt(self.d\_k) # (batch\_size, num\_heads, seq\_len\_q, seq\_len\_k)

        if mask is not None:

            scores = scores.masked\_fill(mask == 0, -1e9) # Apply mask

        attention\_weights = self.softmax(scores) # (batch\_size, num\_heads, seq\_len\_q, seq\_len\_k)

        attention\_weights = self.dropout(attention\_weights)

        context = torch.matmul(attention\_weights, V) # (batch\_size, num\_heads, seq\_len\_q, d\_k)

        # Concatenate heads and apply final linear layer

        context = context.transpose(1, 2).contiguous().view(batch\_size, -1, self.d\_model) # (batch\_size, seq\_len\_q, d\_model)

        output = self.W\_o(context) # (batch\_size, seq\_len\_q, d\_model)

        return output, attention\_weights

##### **2.5 多模态融合策略**

• **功能描述**：  
实现多个模态数据的智能融合，提供更准确、稳定的情绪识别结果。

• **创新技术方案**：

1. **特征级融合**

• 跨模态特征对齐与归一化

• 基于Transformer的特征交互

• 模态间关联性建模

2. **决策级融合**

• 动态权重分配机制

• 基于不确定性的可信度评估

• 多模态一致性验证

3. **时序融合优化**

• 多模态时序同步

• 长短期依赖建模

• 情绪状态平滑过渡

##### **2.6 系统优化与创新特点**

**python**

class MultiModalEmotionRecognitionEngine:

    def \_\_init\_\_(self, device=None):

        self.device = device if device else torch.device("cuda" if torch.cuda.is\_available() else "cpu")

        # Initialize individual modality models and move them to the specified device

        self.face\_model = FaceEmotionModel().to(self.device)

        self.audio\_model = AudioEmotionModel().to(self.device)

        self.text\_model = TextEmotionModel().to(self.device)

        self.fusion\_model = ModalityFusionModel().to(self.device)

        # It's good practice to set models to evaluation mode if they are pre-trained and used for inference

        self.face\_model.eval()

        self.audio\_model.eval()

        self.text\_model.eval()

        self.fusion\_model.eval()

    def preprocess\_face\_data(self, face\_image\_path):

        # Example preprocessing for face data (e.g., loading an image, transforming)

        # This needs to be consistent with how FaceEmotionModel expects its input

        # For ResNet, typical input is a 3-channel image, normalized

        # Placeholder: actual implementation would use libraries like PIL, OpenCV, torchvision.transforms

        # transform = transforms.Compose([

        # transforms.Resize((224, 224)),

        # transforms.ToTensor(),

        # transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

        # ])

        # image = Image.open(face\_image\_path).convert('RGB')

        # return transform(image).unsqueeze(0).to(self.device) # Add batch dimension

        print(f"Preprocessing face data from: {face\_image\_path}")

        # Dummy tensor for now, replace with actual preprocessing

        return torch.randn(1, 3, 224, 224).to(self.device)

    def preprocess\_audio\_data(self, audio\_file\_path):

        # Example preprocessing for audio data (e.g., loading audio, extracting MFCCs)

        # This needs to be consistent with how AudioEmotionModel expects its input

        # Placeholder: actual implementation would use libraries like librosa

        # y, sr = librosa.load(audio\_file\_path, sr=16000)

        # mfccs = librosa.feature.mfcc(y=y, sr=sr, n\_mfcc=40)

        # mfccs\_processed = torch.tensor(mfccs).unsqueeze(0).unsqueeze(0).to(self.device) # (batch, channel, n\_mfcc, time)

        print(f"Preprocessing audio data from: {audio\_file\_path}")

        # Dummy tensor for AudioEmotionModel (batch, 1, num\_mels, num\_frames)

        # Assuming num\_mels=40, num\_frames could vary, e.g., 100

        return torch.randn(1, 1, 40, 100).to(self.device)

    def preprocess\_text\_data(self, text\_input, tokenizer\_name='bert-base-uncased', max\_length=128):

        # Example preprocessing for text data (tokenizing)

        # This needs to be consistent with how TextEmotionModel expects its input

        tokenizer = BertTokenizer.from\_pretrained(tokenizer\_name)

        inputs = tokenizer.encode\_plus(

            text\_input,

            add\_special\_tokens=True,

            max\_length=max\_length,

            padding='max\_length',

            truncation=True,

            return\_attention\_mask=True,

            return\_tensors='pt' # Return PyTorch tensors

        )

        return inputs['input\_ids'].to(self.device), inputs['attention\_mask'].to(self.device)

    def process(self, face\_data\_path=None, audio\_data\_path=None, text\_input\_str=None):

        """Process multi-modal input data and return fused emotion recognition results."""

        modal\_features\_for\_fusion = []

        with torch.no\_grad(): # Ensure no gradients are computed during inference

            if face\_data\_path:

                processed\_face\_data = self.preprocess\_face\_data(face\_data\_path)

                face\_emotion\_logits = self.face\_model(processed\_face\_data)

                # Assuming the model output before softmax is used as features for fusion

                modal\_features\_for\_fusion.append(face\_emotion\_logits)

            if audio\_data\_path:

                processed\_audio\_data = self.preprocess\_audio\_data(audio\_data\_path)

                audio\_emotion\_logits = self.audio\_model(processed\_audio\_data)

                modal\_features\_for\_fusion.append(audio\_emotion\_logits)

            if text\_input\_str:

                input\_ids, attention\_mask = self.preprocess\_text\_data(text\_input\_str)

                text\_emotion\_logits = self.text\_model(input\_ids, attention\_mask)

                modal\_features\_for\_fusion.append(text\_emotion\_logits)

        if not modal\_features\_for\_fusion:

            raise ValueError("No input data provided for any modality.")

        # Perform fusion if there are features from any modality

        # The fusion model will handle alignment internally

        fused\_emotion\_logits = self.fusion\_model(modal\_features\_for\_fusion)

        # Convert logits to probabilities or class predictions as needed

        # emotion\_probabilities = torch.softmax(fused\_emotion\_logits, dim=-1)

        # predicted\_class = torch.argmax(emotion\_probabilities, dim=-1)

        return fused\_emotion\_logits # Or probabilities, or class

# Example Usage (Illustrative - requires actual data and model weights)

if \_\_name\_\_ == '\_\_main\_\_':

    # This example won't run without actual model weights and data paths

    # And also assumes necessary imports like torch, nn, models, BertModel, BertTokenizer, etc.

    # --- Mocking necessary imports for the sake of this example structure ---

    import torch

    import torch.nn as nn

    import torch.nn.functional as F

    from torchvision import models # Assuming torchvision is available

    from transformers import BertModel, BertTokenizer # Assuming transformers is available

    import numpy as np

    # --- End Mocking ---

    print("Initializing MultiModalEmotionRecognitionEngine...")

    engine = MultiModalEmotionRecognitionEngine()

    print("Engine initialized.")

    # Create dummy data paths/inputs for testing

    # In a real scenario, these would be paths to actual files or actual text.

    dummy\_face\_path = "path/to/your/face\_image.jpg" # Replace with a real path or mock data loading

    dummy\_audio\_path = "path/to/your/audio\_file.wav" # Replace with a real path or mock data loading

    dummy\_text = "This is a wonderfully exciting experience!"

    print(f"\nProcessing with all modalities...")

    try:

        # Simulate providing all inputs

        # Note: Preprocessing functions currently return random tensors or require actual data/tokenizers

        # For a true test, these preprocess methods need to be fully implemented.

        # To make this runnable for structure demonstration, let's mock the preprocessed data directly

        # instead of calling preprocess methods that depend on external files/real tokenizers.

        mock\_face\_input = torch.randn(1, 3, 224, 224).to(engine.device)

        mock\_audio\_input = torch.randn(1, 1, 40, 100).to(engine.device)

        mock\_tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

        mock\_text\_tokens = mock\_tokenizer.encode\_plus(dummy\_text, return\_tensors='pt', max\_length=128, padding='max\_length', truncation=True)

        mock\_input\_ids = mock\_text\_tokens['input\_ids'].to(engine.device)

        mock\_attention\_mask = mock\_text\_tokens['attention\_mask'].to(engine.device)

        with torch.no\_grad():

            face\_logits = engine.face\_model(mock\_face\_input)

            audio\_logits = engine.audio\_model(mock\_audio\_input)

            text\_logits = engine.text\_model(mock\_input\_ids, mock\_attention\_mask)

            results = engine.fusion\_model([face\_logits, audio\_logits, text\_logits])

        print(f"Fusion model output (logits): {results}")

        print(f"Predicted emotion class: {torch.argmax(results, dim=-1).item()}")

    except Exception as e:

        print(f"Error during processing: {e}")

        print("This example requires actual model weights, data, and complete preprocessing steps to run fully.")

    print("\n--- Example with only text ---")

    try:

        with torch.no\_grad():

            text\_logits\_only = engine.text\_model(mock\_input\_ids, mock\_attention\_mask)

            # When using fusion model with single modality, it should still work if align\_features handles it.

            # Or, you might directly use the output of the single modality model.

            results\_text\_only\_direct = text\_logits\_only

            results\_text\_only\_fused = engine.fusion\_model([text\_logits\_only])

        print(f"Direct text model output (logits): {results\_text\_only\_direct}")

        print(f"Predicted emotion class (direct): {torch.argmax(results\_text\_only\_direct, dim=-1).item()}")

        print(f"Fusion model with text only (logits): {results\_text\_only\_fused}")

        print(f"Predicted emotion class (fused text): {torch.argmax(results\_text\_only\_fused, dim=-1).item()}")

    except Exception as e:

        print(f"Error during text-only processing: {e}")

##### **2.6 系统优化与创新特点**

详细的代码实现请参考 /Users/yingyang/Documents/project/mdworld/TripWithNetCelebs/专利申请文件/国内专利申请/code/EmotionRecognitionEngine.py 文件。

1. **自适应性增强**

• 用户个体差异自适应

• 场景上下文感知

• 动态阈值调整

2. **鲁棒性提升**

• 模态缺失补偿机制

• 噪声干扰抑制

• 异常状态检测

3. **实时性保障**

• 计算流水线优化

• 模型量化与加速

• 边缘计算部署支持

4. **可解释性设计**

• 情绪识别结果可视化

• 决策依据追踪

• 置信度评估机制

#### **3. 社交互动推荐引擎**

社交互动推荐引擎是实现个性化、智能化社交体验的核心驱动力。该引擎综合运用多种推荐算法和策略，结合用户实时情绪状态、兴趣偏好、地理位置以及活动情境，为用户提供精准、及时的社交对象和互动内容推荐。

##### **3.1 情绪匹配推荐单元**

• **功能描述**：  
基于用户当前及历史情绪状态，进行社交匹配推荐，旨在促进具有情感共鸣的互动。

• **核心算法与策略**：

1. **情绪相似度匹配**

• 采用欧氏距离或余弦相似度计算用户间情绪向量的相似性。

• 引入情绪强度和稳定性作为匹配权重。

• 优先推荐当前情绪状态相近或高度一致的用户。

**python**

class EmotionSimilarityMatcher:

    def \_\_init\_\_(self):

        self.emotion\_dim = 8 # 情绪向量维度：喜、怒、哀、乐、惊、恐、厌、中性

    def calculate\_emotion\_similarity(self, user1\_emotion, user2\_emotion):

        """计算两个用户的情绪向量相似度

        Args:

            user1\_emotion: 用户1的情绪向量 [float] \* emotion\_dim

            user2\_emotion: 用户2的情绪向量 [float] \* emotion\_dim

        Returns:

            float: 情绪相似度分数 (0-1)

        """

        # 计算欧氏距离

        euclidean\_dist = np.sqrt(np.sum((np.array(user1\_emotion) - np.array(user2\_emotion)) \*\* 2))

        # 计算余弦相似度

        cosine\_sim = np.dot(user1\_emotion, user2\_emotion) / \

                    (np.linalg.norm(user1\_emotion) \* np.linalg.norm(user2\_emotion))

        # 综合考虑两种相似度指标

        similarity = 0.4 \* (1 / (1 + euclidean\_dist)) + 0.6 \* cosine\_sim

        return similarity

    def calculate\_emotion\_stability(self, emotion\_history):

        """计算用户情绪的稳定性

        Args:

            emotion\_history: 用户历史情绪记录 [[float] \* emotion\_dim]

        Returns:

            float: 情绪稳定性分数 (0-1)

        """

        if len(emotion\_history) < 2:

            return 1.0

        variations = []

        for i in range(1, len(emotion\_history)):

            variation = np.linalg.norm(

                np.array(emotion\_history[i]) - np.array(emotion\_history[i-1]))

            variations.append(variation)

        stability = 1 / (1 + np.mean(variations))

        return stability

    def find\_matching\_users(self, target\_user, candidate\_users, top\_k=5):

        """为目标用户找到情绪最匹配的用户

        Args:

            target\_user: 目标用户对象，包含current\_emotion和emotion\_history

            candidate\_users: 候选用户列表

            top\_k: 返回的匹配用户数量

        Returns:

            list: 匹配度最高的top\_k个用户及其匹配分数

        """

        matching\_scores = []

        target\_stability = self.calculate\_emotion\_stability(target\_user.emotion\_history)

        for candidate in candidate\_users:

            # 计算情绪相似度

            emotion\_sim = self.calculate\_emotion\_similarity(

                target\_user.current\_emotion, candidate.current\_emotion)

            # 计算候选用户情绪稳定性

            candidate\_stability = self.calculate\_emotion\_stability(candidate.emotion\_history)

            # 综合评分：情绪相似度 \* (目标用户稳定性 + 候选用户稳定性) / 2

            final\_score = emotion\_sim \* (target\_stability + candidate\_stability) / 2

            matching\_scores.append((candidate, final\_score))

        # 按匹配分数降序排序并返回top\_k

        matching\_scores.sort(key=lambda x: x[1], reverse=True)

        return matching\_scores[:top\_k]

1. **情绪互补性分析与推荐**

• 识别具有情绪互补潜力的用户对（例如，一个寻求安慰，一个乐于倾听）。

• 基于心理学理论构建情绪互补模型。

• 推荐能够提供情感支持或积极影响的互动对象。

2. **情绪状态转移预测与干预**

• 基于马尔可夫链或RNN预测用户未来情绪状态的转变概率。

• 在预测到负面情绪趋势时，主动推荐能够改善情绪的社交互动或内容。

• 结合用户历史情绪模式进行个性化预测。

3. **群体情绪感知与引导**

• 分析特定区域或活动中群体的整体情绪氛围。

• 推荐能够融入当前群体情绪的社交活动。

• 避免推荐与当前群体情绪冲突的互动。

##### **3.2 兴趣相似度计算单元**

• **功能描述**：  
深度挖掘和理解用户的旅游兴趣偏好，计算用户间的兴趣相似度，为推荐提供重要依据。

• **技术实现细节**：

1. **用户兴趣画像构建**

• 基于用户历史行为（浏览、收藏、评论、分享、参与活动等）构建多维度兴趣标签体系。

• 采用TF-IDF、Word2Vec等技术生成用户兴趣向量。

• 结合用户显式填写的兴趣标签和隐式行为推断。

2. **旅游偏好多维度相似度计算**

• 不仅考虑兴趣点的重合度，还分析兴趣的深度和广度。

• 引入旅行方式（如穷游、奢华游）、旅行主题（如探险、美食、文化）、目的地类型等多维度偏好。

• 采用加权相似度计算方法，权重可根据场景动态调整。

3. **动态兴趣演化追踪机制**

• 用户的兴趣会随时间、经历和外部影响而变化。

• 采用时间衰减模型和增量学习算法，实时更新用户兴趣画像。

• 捕捉用户短期兴趣热点和长期稳定偏好。

4. **冷启动用户兴趣探索**

• 对于新用户或行为稀疏用户，采用基于内容的推荐、热门推荐或引导式问答等方式进行兴趣探索。

• 结合人口统计学特征进行初步兴趣预估。

**python**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

import numpy as np

import time

class AdvancedInterestSimilarityCalculator:

    def \_\_init\_\_(self, interest\_corpus=None, time\_decay\_factor=0.01):

        self.vectorizer = TfidfVectorizer(stop\_words='english')

        self.user\_profiles = {} # 存储用户ID -> {'interests': [], 'vector': None, 'last\_updated': 0, 'preferences': {}}

        self.time\_decay\_factor = time\_decay\_factor # 时间衰减因子，用于动态兴趣追踪

        if interest\_corpus:

            # 预处理语料库，每个用户一个文档字符串

            corpus\_docs = [' '.join(interests) for interests in interest\_corpus]

            if corpus\_docs:

                self.vectorizer.fit(corpus\_docs)

    def \_get\_interest\_text(self, interests\_list):

        return ' '.join(sorted(list(set(interests\_list))))

    def \_update\_vectorizer\_if\_needed(self, new\_interest\_text=None):

        # 如果词汇表为空或需要基于新数据更新

        # 实际应用中，大规模更新词汇表可能需要更复杂的策略

        if not hasattr(self.vectorizer, 'vocabulary\_') or not self.vectorizer.vocabulary\_:

            all\_texts = [self.\_get\_interest\_text(profile['interests'])

                         for profile in self.user\_profiles.values() if profile['interests']]

            if new\_interest\_text:

                all\_texts.append(new\_interest\_text)

            if all\_texts:

                self.vectorizer.fit(all\_texts)

            else:

                return False # 没有文本可供训练

        return True

    def add\_or\_update\_user(self, user\_id, interests\_list, preferences=None, timestamp=None):

        """ 添加或更新用户信息，包括兴趣、偏好和时间戳

        Args:

            user\_id (str): 用户ID

            interests\_list (list of str): 用户的兴趣标签列表

            preferences (dict, optional): 用户的旅行偏好，例如 {'style': '穷游', 'theme': '探险'}

            timestamp (float, optional): 更新时间戳，默认为当前时间

        """

        interest\_text = self.\_get\_interest\_text(interests\_list)

        current\_time = timestamp if timestamp is not None else time.time()

        if not self.\_update\_vectorizer\_if\_needed(interest\_text):

            # 如果向量化器无法训练（例如没有兴趣数据），则不生成向量

            self.user\_profiles[user\_id] = {

                'interests': interests\_list,

                'vector': np.array([]),

                'last\_updated': current\_time,

                'preferences': preferences or {}

            }

            return

        vector = self.vectorizer.transform([interest\_text])

        self.user\_profiles[user\_id] = {

            'interests': interests\_list,

            'vector': vector,

            'last\_updated': current\_time,

            'preferences': preferences or {}

        }

    def \_calculate\_preference\_similarity(self, prefs1, prefs2):

        """计算旅行偏好相似度"""

        if not prefs1 or not prefs2:

            return 0.0

        common\_keys = set(prefs1.keys()) & set(prefs2.keys())

        if not common\_keys:

            return 0.0

        matches = sum(1 for k in common\_keys if prefs1[k] == prefs2[k])

        return matches / len(common\_keys)

    def calculate\_similarity(self, user1\_id, user2\_id, interest\_weight=0.7, preference\_weight=0.3, time\_weight\_factor=0.1):

        """ 计算两个用户之间的综合相似度，考虑兴趣、偏好和时间衰减

        Args:

            user1\_id (str): 用户1的ID

            user2\_id (str): 用户2的ID

            interest\_weight (float): 兴趣相似度的权重

            preference\_weight (float): 偏好相似度的权重

            time\_weight\_factor (float): 时间衰减对兴趣相似度的影响因子

        Returns:

            float: 综合相似度分数 (0-1)。如果任一用户不存在或无有效数据，返回0。

        """

        if user1\_id not in self.user\_profiles or user2\_id not in self.user\_profiles:

            return 0.0

        profile1 = self.user\_profiles[user1\_id]

        profile2 = self.user\_profiles[user2\_id]

        if profile1['vector'].shape[1] == 0 or profile2['vector'].shape[1] == 0:

            # 如果任一用户的兴趣向量为空（例如，只有新用户且没有语料库训练vectorizer）

            interest\_sim = 0.0

        else:

            interest\_sim = cosine\_similarity(profile1['vector'], profile2['vector'])[0][0]

        # 时间衰减: 最近更新的兴趣更重要

        time\_diff = abs(profile1['last\_updated'] - profile2['last\_updated'])

        # 简单的时间衰减模型，可以根据需求调整

        time\_decay\_penalty = np.exp(-self.time\_decay\_factor \* (time\_diff / (3600\*24))) # 按天计算衰减

        adjusted\_interest\_sim = interest\_sim \* time\_decay\_penalty

        preference\_sim = self.\_calculate\_preference\_similarity(profile1['preferences'], profile2['preferences'])

        total\_similarity = (interest\_weight \* adjusted\_interest\_sim) + (preference\_weight \* preference\_sim)

        return max(0, min(total\_similarity, 1.0)) # 确保在0-1之间

    def find\_similar\_users(self, target\_user\_id, top\_k=5, \*\*kwargs):

        """ 为目标用户找到最相似的用户

        Args:

            target\_user\_id (str): 目标用户ID

            top\_k (int): 返回数量

            \*\*kwargs: 传递给 calculate\_similarity 的参数 (interest\_weight, etc.)

        Returns:

            list: (user\_id, similarity\_score) 列表

        """

        if target\_user\_id not in self.user\_profiles or \

           self.user\_profiles[target\_user\_id]['vector'].shape[1] == 0:

            # 处理冷启动：如果目标用户兴趣未知，可以返回热门用户或基于其他策略

            # 此处简化为返回空列表

            print(f"Warning: Target user {target\_user\_id} has no interest vector. Cannot find similar users based on interests.")

            return []

        scores = []

        for user\_id in self.user\_profiles:

            if user\_id == target\_user\_id:

                continue

            if self.user\_profiles[user\_id]['vector'].shape[1] == 0:

                # 跳过没有兴趣向量的用户

                continue

            similarity = self.calculate\_similarity(target\_user\_id, user\_id, \*\*kwargs)

            scores.append((user\_id, similarity))

        scores.sort(key=lambda x: x[1], reverse=True)

        return scores[:top\_k]

# 示例用法

# adv\_calculator = AdvancedInterestSimilarityCalculator(time\_decay\_factor=0.005)

# # 添加用户时可以指定偏好和时间戳

# adv\_calculator.add\_or\_update\_user('userA', ['徒步', '摄影', '历史'],

# preferences={'style': '探险', 'theme': '文化', 'duration': '长途'},

# timestamp=time.time() - 86400 \* 10) # 10天前

# adv\_calculator.add\_or\_update\_user('userB', ['摄影', '美食', '艺术'],

# preferences={'style': '休闲', 'theme': '艺术', 'duration': '短途'},

# timestamp=time.time() - 86400 \* 2) # 2天前

# adv\_calculator.add\_or\_update\_user('userC', ['徒步', '历史', '自然'],

# preferences={'style': '探险', 'theme': '自然', 'duration': '长途'},

# timestamp=time.time())

# adv\_calculator.add\_or\_update\_user('userD', ['美食', '购物'],

# preferences={'style': '奢华', 'theme': '购物'},

# timestamp=time.time() - 86400 \* 30) # 30天前

# # 计算相似度时可以调整权重

# sim\_ab = adv\_calculator.calculate\_similarity('userA', 'userB', interest\_weight=0.6, preference\_weight=0.4)

# print(f"Advanced Similarity UserA and UserB: {sim\_ab}")

# # 查找相似用户

# similar\_to\_a = adv\_calculator.find\_similar\_users('userA', top\_k=2)

# print(f"Most similar to UserA: {similar\_to\_a}")

# # 冷启动用户示例 (没有兴趣标签)

# adv\_calculator.add\_or\_update\_user('userE', [], preferences={'style': '经济型'})

# similar\_to\_e = adv\_calculator.find\_similar\_users('userE')

# print(f"Similar to UserE (cold start): {similar\_to\_e}")

# # 只有单个用户时，vectorizer可能未训练

# single\_user\_calc = AdvancedInterestSimilarityCalculator()

# single\_user\_calc.add\_or\_update\_user('userS', ['滑雪'])

# print(f"Similar to UserS: {single\_user\_calc.find\_similar\_users('userS')}")

##### **3.3 地理位置匹配单元**

• **功能描述**：  
利用LBS（Location-Based Service）技术，实现基于用户实时地理位置的社交匹配与推荐。

• **核心技术与策略**：

1. **实时位置感知与匹配**

• 高精度获取用户实时GPS坐标。

• 计算用户间的地理距离，筛选附近用户。

• 支持自定义距离范围的匹配。

2. **旅游路线相似度分析**

• 记录并分析用户的历史和计划旅行轨迹。

• 推荐具有相似旅行路线或目的地计划的用户。

• 促进途中结伴或经验分享。

3. **地理围栏与场景感知推荐**

• 针对特定景点、区域或活动设置地理围栏。

• 当用户进入或离开围栏区域时，触发场景相关的社交推荐（如推荐该景点的热门同游者、附近特色活动等）。

• 结合时间因素（如白天、夜晚）进行场景细分。

**python**

import numpy as np

from math import radians, sin, cos, sqrt, atan2

class GeoLocationMatcher:

    def \_\_init\_\_(self, earth\_radius\_km=6371.0):

        self.earth\_radius\_km = earth\_radius\_km

        self.user\_locations = {} # user\_id -> {'lat': float, 'lon': float, 'timestamp': float}

        self.user\_travel\_history = {} # user\_id -> [{'lat': float, 'lon': float, 'timestamp': float}, ...]

        self.user\_planned\_routes = {} # user\_id -> [{'lat': float, 'lon': float, 'order': int}, ...]

        self.geo\_fences = {} # fence\_id -> {'center\_lat': float, 'center\_lon': float, 'radius\_m': float, 'name': str}

    def \_haversine\_distance(self, lat1, lon1, lat2, lon2):

        """计算两个GPS坐标点之间的距离（公里）"""

        dlat = radians(lat2 - lat1)

        dlon = radians(lon2 - lon1)

        a = sin(dlat / 2)\*\*2 + cos(radians(lat1)) \* cos(radians(lat2)) \* sin(dlon / 2)\*\*2

        c = 2 \* atan2(sqrt(a), sqrt(1 - a))

        return self.earth\_radius\_km \* c

    def update\_user\_location(self, user\_id, lat, lon, timestamp):

        self.user\_locations[user\_id] = {'lat': lat, 'lon': lon, 'timestamp': timestamp}

        # 同时更新旅行历史

        if user\_id not in self.user\_travel\_history:

            self.user\_travel\_history[user\_id] = []

        self.user\_travel\_history[user\_id].append({'lat': lat, 'lon': lon, 'timestamp': timestamp})

        # 可以考虑历史记录的长度限制

        self.user\_travel\_history[user\_id].sort(key=lambda x: x['timestamp'])

    def find\_nearby\_users(self, target\_user\_id, max\_distance\_km, max\_time\_diff\_sec=3600):

        """查找附近的用户"""

        if target\_user\_id not in self.user\_locations:

            return []

        target\_loc = self.user\_locations[target\_user\_id]

        nearby\_users = []

        current\_time = time.time() # 或者使用传入的时间戳

        for user\_id, loc\_data in self.user\_locations.items():

            if user\_id == target\_user\_id:

                continue

            # 检查位置信息的新鲜度

            if current\_time - loc\_data['timestamp'] > max\_time\_diff\_sec:

                continue

            distance = self.\_haversine\_distance(target\_loc['lat'], target\_loc['lon'],

                                                loc\_data['lat'], loc\_data['lon'])

            if distance <= max\_distance\_km:

                nearby\_users.append({'user\_id': user\_id, 'distance\_km': distance, 'location': loc\_data})

        nearby\_users.sort(key=lambda x: x['distance\_km'])

        return nearby\_users

    def add\_planned\_route(self, user\_id, route\_points):

        """添加用户的计划路线，route\_points是点的列表，每个点是{'lat': ..., 'lon': ..., 'order': ...}"""

        self.user\_planned\_routes[user\_id] = sorted(route\_points, key=lambda x: x.get('order', 0))

    def \_calculate\_route\_similarity\_dtw(self, route1, route2):

        """使用动态时间规整(DTW)计算两条路线的相似度（简化版，仅考虑点序列）"""

        # 此处DTW实现较为复杂，实际应用中可使用库如 fastdtw

        # 简化：计算对应点之间的平均距离，如果长度不同则惩罚

        if not route1 or not route2:

            return float('inf') # 表示不相似

        len1, len2 = len(route1), len(route2)

        # 为了简化，我们只比较等长部分，或者对齐后比较

        # 真实DTW会找到最佳对齐路径

        # 这里使用一个非常简化的版本：如果长度差异过大，则认为不相似

        if abs(len1 - len2) > min(len1, len2) \* 0.5: # 长度差异超过50%

             return float('inf')

        dist\_sum = 0

        min\_len = min(len1, len2)

        for i in range(min\_len):

            dist\_sum += self.\_haversine\_distance(route1[i]['lat'], route1[i]['lon'],

                                                 route2[i]['lat'], route2[i]['lon'])

        avg\_dist = dist\_sum / min\_len if min\_len > 0 else float('inf')

        # 相似度可以定义为 1 / (1 + avg\_dist)

        return 1 / (1 + avg\_dist) if avg\_dist != float('inf') else 0

    def find\_similar\_route\_users(self, target\_user\_id, top\_k=3):

        if target\_user\_id not in self.user\_planned\_routes:

            return []

        target\_route = self.user\_planned\_routes[target\_user\_id]

        similar\_routes = []

        for user\_id, route in self.user\_planned\_routes.items():

            if user\_id == target\_user\_id:

                continue

            similarity = self.\_calculate\_route\_similarity\_dtw(target\_route, route)

            if similarity > 0: # 假设DTW返回的是距离，相似度是其倒数或转换

                similar\_routes.append({'user\_id': user\_id, 'similarity': similarity})

        similar\_routes.sort(key=lambda x: x['similarity'], reverse=True)

        return similar\_routes[:top\_k]

    def add\_geo\_fence(self, fence\_id, center\_lat, center\_lon, radius\_m, name):

        self.geo\_fences[fence\_id] = {

            'center\_lat': center\_lat,

            'center\_lon': center\_lon,

            'radius\_m': radius\_m,

            'name': name

        }

    def check\_user\_in\_geo\_fences(self, user\_id):

        """检查用户当前位置是否在任何地理围栏内"""

        if user\_id not in self.user\_locations:

            return []

        user\_loc = self.user\_locations[user\_id]

        fences\_entered = []

        for fence\_id, fence\_data in self.geo\_fences.items():

            distance\_km = self.\_haversine\_distance(user\_loc['lat'], user\_loc['lon'],

                                                 fence\_data['center\_lat'], fence\_data['center\_lon'])

            if distance\_km \* 1000 <= fence\_data['radius\_m']:

                fences\_entered.append({'fence\_id': fence\_id, 'fence\_name': fence\_data['name']})

        return fences\_entered

# 示例用法

# geo\_matcher = GeoLocationMatcher()

# current\_ts = time.time()

# geo\_matcher.update\_user\_location('userX', 34.0522, -118.2437, current\_ts) # Los Angeles

# geo\_matcher.update\_user\_location('userY', 34.0520, -118.2430, current\_ts - 600) # Nearby, 10 mins ago

# geo\_matcher.update\_user\_location('userZ', 40.7128, -74.0060, current\_ts) # New York

# nearby\_to\_x = geo\_matcher.find\_nearby\_users('userX', max\_distance\_km=1.0)

# print(f"Nearby users to UserX: {nearby\_to\_x}")

# # 路线示例

# route\_x = [{'lat': 34.05, 'lon': -118.24, 'order': 1}, {'lat': 34.06, 'lon': -118.25, 'order': 2}]

# route\_y = [{'lat': 34.055, 'lon': -118.245, 'order': 1}, {'lat': 34.065, 'lon': -118.255, 'order': 2}]

# geo\_matcher.add\_planned\_route('userX', route\_x)

# geo\_matcher.add\_planned\_route('userY', route\_y)

# similar\_routes\_to\_x = geo\_matcher.find\_similar\_route\_users('userX')

# print(f"Users with similar routes to UserX: {similar\_routes\_to\_x}")

# # 地理围栏示例

# geo\_matcher.add\_geo\_fence('disneyland', 33.8121, -117.9190, 1000, "Disneyland Park")

# in\_fences\_x = geo\_matcher.check\_user\_in\_geo\_fences('userX') # UserX is not in Disneyland

# print(f"UserX is in fences: {in\_fences\_x}")

# # 假设UserX移动到Disneyland内

# geo\_matcher.update\_user\_location('userX', 33.8121, -117.9190, time.time())

# in\_fences\_x\_updated = geo\_matcher.check\_user\_in\_geo\_fences('userX')

# print(f"UserX (updated) is in fences: {in\_fences\_x\_updated}")

1. **位置隐私保护**

• 提供位置模糊化、匿名化等隐私设置选项。

• 严格遵守用户隐私协议，确保位置数据安全。

##### **3.4 活动推荐单元**

• **功能描述**：  
根据用户情绪、兴趣、位置以及活动本身的特性，推荐合适的旅游活动和社交场景。

• **推荐算法与机制**：

1. **基于群体情绪的活动推荐**

• 分析特定活动的参与者群体情绪状态。

• 推荐与用户当前情绪相契合的、具有良好氛围的活动。

• 例如，为情绪高涨的用户推荐热闹的派对，为寻求宁静的用户推荐茶艺体验。

2. **个性化活动偏好学习**

• 基于用户历史参与活动、评分、反馈等数据，学习其活动偏好。

• 采用协同过滤、矩阵分解等算法进行个性化活动推荐。

• 考虑活动类型、规模、时间、费用等多种因素。

3. **实时活动热度与推荐排序**

• 实时监测各类活动的参与人数、讨论热度、用户评价等数据。

• 结合活动热度和个性化偏好进行综合排序推荐。

• 优先推荐当前热门且符合用户兴趣的活动。

4. **活动组织者与参与者匹配**

• 不仅推荐活动给参与者，也为活动组织者推荐潜在的感兴趣用户。

• 促进活动供需双方的有效对接。

##### **3.5 推荐策略优化与评估**

• **功能描述**：  
持续优化推荐算法和策略，提升推荐效果和用户满意度。

• **核心机制**：

1. **多臂老虎机（Multi-Armed Bandit）算法的推荐策略动态选择**

• 将不同的推荐算法或参数组合视为老虎机的臂。

• 根据实时用户反馈（点击率、转化率、满意度等）动态调整各策略的探索（Exploration）与利用（Exploitation）平衡。

• 自动选择当前最优的推荐策略组合。

2. **用户反馈驱动的推荐模型持续更新**

• 收集用户对推荐结果的显式反馈（如评分、喜欢/不喜欢）和隐式反馈（如点击、忽略、停留时间）。

• 利用在线学习或增量学习算法，实时更新推荐模型参数。

• 确保模型能够快速适应用户偏好和外部环境的变化。

3. **A/B测试框架的推荐效果评估**

• 建立完善的A/B测试平台，支持并行运行多种推荐策略。

• 通过小流量实验，科学评估不同策略在关键指标（如CTR、CVR、用户留存率、社交互动频率等）上的表现。

• 为算法迭代和策略调整提供数据支持。

4. **推荐多样性与新颖性保障**

• 避免过度推荐相似内容导致用户审美疲劳。

• 引入随机性或探索性机制，增加推荐结果的多样性和新颖性。

• 鼓励用户发现新的兴趣点和社交圈。

5. **可解释性推荐**

• 向用户展示推荐理由（如“因为你喜欢XX”、“与你情绪相似的人也喜欢这个”）。

• 增强用户对推荐结果的信任度和接受度。

#### **4. 记忆构建引擎**

记忆构建引擎致力于将用户在旅途中产生的碎片化数据（照片、视频、文字、位置信息、情绪状态等）智能地组织、串联并升华为富有情感和故事性的共享旅游记忆。该引擎不仅记录事实，更注重捕捉和再现旅途中的情感体验和互动瞬间。

##### **4.1 共享旅游记忆生成单元**

• **功能描述**：  
自动整合来自同一旅行活动中多个用户的数据，构建统一的、多视角的共享旅游记忆时间轴，并从中提炼关键事件和情感高潮。

• **技术实现与算法**：

1. **多用户异构数据融合与时间轴精确对齐**

• **数据源**：整合来自不同用户的照片（含EXIF元数据）、视频片段、文字记录（日记、评论）、地理位置轨迹、生理信号数据（如心率、皮电）、以及系统记录的互动行为数据。

• **时间同步**：利用NTP（网络时间协议）或GPS时间戳对各用户设备采集的数据进行精确的时间校准，解决设备间时间差异问题。

• **空间对齐**：结合GPS数据和图像内容分析（如地标识别），将不同用户在同一时间、同一地点产生的数据进行空间关联。

• **事件聚合**：通过聚类算法（如DBSCAN、Mean Shift）将时间上相近、空间上重叠、内容上相关的多用户数据点聚合成候选事件。

2. **关键事件与情感节点智能识别**

• **事件显著性评估**：

• **内容分析**：利用计算机视觉技术（如物体识别、场景分类、美学评分）和NLP技术（如关键词提取、主题建模、情感分析）评估各数据片段的重要性。

• **互动强度**：分析用户间的互动频率和深度（如共同评论、点赞、合影）作为事件重要性的指标。

• **情绪波动**：结合多模态情绪识别引擎的结果，识别出情绪强度较高或情绪发生显著变化的时刻作为潜在情感节点。

• **代表性片段选取**：从每个关键事件或情感节点中，自动选取最具代表性的照片、视频片段或文字描述。

3. **自动化故事线生成与叙事结构优化**

• **叙事单元构建**：将识别出的关键事件和情感节点作为叙事的基本单元。

• **时序与逻辑串联**：默认按照时间顺序组织故事线，同时允许基于特定主题（如美食之旅、探险时刻）或情感脉络（如从期待到惊喜）进行重组。

• **叙事节奏控制**：通过调整事件密度、引入过渡元素（如地图轨迹、风景空镜）来控制故事的节奏和流畅性。

• **多视角叙事**：支持从不同参与者的视角生成个性化的故事线分支，或融合多个视角形成更全面的主故事线。

• **算法**：可采用基于规则的系统、时序模式挖掘或借鉴叙事理论（如三幕式结构）的机器学习模型。

**python**

import datetime

from collections import defaultdict

from sklearn.cluster import DBSCAN

import numpy as np

class SharedMemoryGenerator:

    def \_\_init\_\_(self, emotion\_engine, vision\_model, nlp\_model):

        """

        初始化共享旅游记忆生成单元。

        :param emotion\_engine: 多模态情绪识别引擎实例

        :param vision\_model: 计算机视觉模型实例 (用于图像分析)

        :param nlp\_model: 自然语言处理模型实例 (用于文本分析)

        """

        self.emotion\_engine = emotion\_engine

        self.vision\_model = vision\_model

        self.nlp\_model = nlp\_model

        self.user\_data = defaultdict(list) # {user\_id: [data\_points]}

        self.events = []

        self.storylines = {}

    def \_normalize\_timestamp(self, data\_point):

        """将不同来源的时间戳统一处理 (示例)"""

        # 实际应用中需要更复杂的NTP或GPS同步逻辑

        if isinstance(data\_point.get('timestamp'), str):

            try:

                return datetime.datetime.fromisoformat(data\_point['timestamp'])

            except ValueError:

                return datetime.datetime.now() # 降级处理

        elif isinstance(data\_point.get('timestamp'), (int, float)):

            return datetime.datetime.fromtimestamp(data\_point['timestamp'])

        return data\_point.get('timestamp', datetime.datetime.now())

    def add\_user\_data(self, user\_id, data\_points):

        """

        添加单个用户在旅行中的数据点。

        :param user\_id: 用户ID

        :param data\_points: 数据点列表，每个数据点是一个字典，包含如：

                          {'type': 'photo', 'content': 'path/to/image.jpg', 'timestamp': '2023-10-26T10:00:00', 'location': (lat, lon), ...}

                          {'type': 'text', 'content': '今天天气真好！', 'timestamp': 1672502400, ...}

                          {'type': 'video', 'content': 'path/to/video.mp4', ...}

                          {'type': 'interaction', 'target\_user\_id': 'user\_B', 'interaction\_type': 'comment', ...}

        """

        for dp in data\_points:

            dp['timestamp\_dt'] = self.\_normalize\_timestamp(dp)

            dp['user\_id'] = user\_id

            self.user\_data[user\_id].append(dp)

    def fuse\_and\_align\_data(self):

        """整合所有用户数据并按时间排序，准备进行事件聚合"""

        all\_data = []

        for user\_id in self.user\_data:

            all\_data.extend(self.user\_data[user\_id])

        # 按时间戳排序

        all\_data.sort(key=lambda x: x['timestamp\_dt'])

        return all\_data

    def \_calculate\_spatial\_temporal\_features(self, data\_points):

        """提取时空特征用于聚类 (示例)"""

        features = []

        for dp in data\_points:

            # 时间特征：转换为自某个参考点以来的秒数

            time\_feature = (dp['timestamp\_dt'] - data\_points[0]['timestamp\_dt']).total\_seconds()

            # 空间特征：如果存在位置信息

            loc\_feature\_lat, loc\_feature\_lon = 0, 0

            if 'location' in dp and isinstance(dp['location'], tuple) and len(dp['location']) == 2:

                loc\_feature\_lat, loc\_feature\_lon = dp['location']

            features.append([time\_feature, loc\_feature\_lat, loc\_feature\_lon])

        return np.array(features)

    def aggregate\_events(self, all\_data, eps=3600, min\_samples=2, location\_weight=0.1):

        """

        使用DBSCAN等聚类算法聚合事件。

        eps: 定义邻域大小（例如，时间上相差1小时，空间上相差一定距离）

        min\_samples: 形成一个核心对象的最小样本数

        location\_weight: 地理位置在距离计算中的权重

        """

        if not all\_data:

            return

        features = []

        for dp in all\_data:

            time\_feature = dp['timestamp\_dt'].timestamp() # 秒级时间戳

            lat, lon = (0,0)

            if 'location' in dp and dp['location']:

                lat, lon = dp['location']

            features.append([time\_feature, lat \* location\_weight, lon \* location\_weight])

        features\_np = np.array(features)

        # DBSCAN聚类

        # eps 需要根据实际数据的时间和空间尺度进行调整

        # 例如，如果时间以秒为单位，eps=3600 表示1小时内

        # 如果地理坐标经过加权，也需要相应调整

        db = DBSCAN(eps=eps, min\_samples=min\_samples).fit(features\_np)

        labels = db.labels\_

        # 将聚类结果组织成事件

        num\_events = len(set(labels)) - (1 if -1 in labels else 0)

        self.events = [[] for \_ in range(num\_events)]

        for i, label in enumerate(labels):

            if label != -1: # -1表示噪声点

                self.events[label].append(all\_data[i])

        print(f"Aggregated {len(self.events)} events.")

    def identify\_key\_moments(self, event\_data):

        """

        在每个事件中识别关键时刻和情感节点。

        :param event\_data: 单个事件包含的数据点列表

        :return: 关键时刻的数据点列表

        """

        key\_moments = []

        if not event\_data:

            return key\_moments

        for dp in event\_data:

            significance\_score = 0

            # 1. 内容分析 (示例)

            if dp['type'] == 'photo' and self.vision\_model:

                # 假设 vision\_model 有一个评估美学/重要性的方法

                significance\_score += self.vision\_model.assess\_photo\_significance(dp['content'])

            elif dp['type'] == 'text' and self.nlp\_model:

                # 假设 nlp\_model 有一个提取主题和情感的方法

                analysis = self.nlp\_model.analyze\_text(dp['content'])

                significance\_score += analysis.get('importance\_score', 0)

                dp['text\_emotion'] = analysis.get('emotion')

            # 2. 互动强度 (示例)

            if dp['type'] == 'interaction':

                significance\_score += 0.5 # 简单示例，可根据互动类型加权

            # 3. 情绪波动 (示例)

            if self.emotion\_engine:

                # 假设数据点可以送入情绪引擎分析

                # 这里简化为如果文本本身有情绪，则增加显著性

                if dp.get('text\_emotion'):

                    emotion\_intensity = self.emotion\_engine.get\_emotion\_intensity(dp['text\_emotion'])

                    significance\_score += emotion\_intensity \* 0.3

            dp['significance\_score'] = significance\_score

        # 按显著性排序并选取top-N或高于阈值的作为关键时刻

        event\_data.sort(key=lambda x: x.get('significance\_score', 0), reverse=True)

        key\_moments = event\_data[:max(1, len(event\_data) // 5)] # 取前20%或至少1个

        return key\_moments

    def generate\_storyline(self, user\_id=None, theme=None):

        """

        生成故事线。

        :param user\_id: 如果为特定用户生成个性化故事线，否则为共享故事线

        :param theme: 可选的主题，如 'food', 'adventure'

        :return: 故事线 (一系列有序的事件/关键时刻)

        """

        story = []

        target\_events = self.events # 默认使用所有聚合事件

        # TODO: 如果是特定用户，可能需要筛选与其相关性更高的事件

        # TODO: 如果有主题，筛选与主题相关的事件

        for event\_cluster in target\_events:

            key\_moments\_in\_event = self.identify\_key\_moments(event\_cluster)

            if key\_moments\_in\_event:

                # 简单地将关键时刻按时间排序加入故事线

                key\_moments\_in\_event.sort(key=lambda x: x['timestamp\_dt'])

                story.extend(key\_moments\_in\_event)

        # 叙事结构优化 (简单示例：按时间排序)

        story.sort(key=lambda x: x['timestamp\_dt'])

        story\_id = f"shared\_story\_{datetime.datetime.now().strftime('%Y%m%d%H%M%S')}"

        if user\_id:

            story\_id = f"user\_{user\_id}\_story\_{datetime.datetime.now().strftime('%Y%m%d%H%M%S')}"

        self.storylines[story\_id] = story

        print(f"Generated storyline '{story\_id}' with {len(story)} moments.")

        return story

    def process\_shared\_memories(self):

        """完整处理流程：融合数据 -> 聚合事件 -> 生成故事线"""

        all\_fused\_data = self.fuse\_and\_align\_data()

        if not all\_fused\_data:

            print("No data to process.")

            return

        self.aggregate\_events(all\_fused\_data)

        shared\_story = self.generate\_storyline()

        return shared\_story

# --- 示例辅助类 (需要用户根据实际情况实现) ---

class MockEmotionEngine:

    def get\_emotion\_intensity(self, emotion\_label):

        # 示例：简单返回一个基于标签的强度值

        intensities = {'happy': 0.8, 'sad': 0.6, 'excited': 0.9, 'neutral': 0.2}

        return intensities.get(emotion\_label, 0.5)

class MockVisionModel:

    def assess\_photo\_significance(self, photo\_path):

        # 示例：随机返回一个显著性评分

        return np.random.rand()

class MockNLPModel:

    def analyze\_text(self, text\_content):

        # 示例：简单分析文本

        analysis\_result = {'importance\_score': len(text\_content) / 100.0, 'emotion': None}

        if '开心' in text\_content or '高兴' in text\_content:

            analysis\_result['emotion'] = 'happy'

        elif '难过' in text\_content or '伤心' in text\_content:

            analysis\_result['emotion'] = 'sad'

        return analysis\_result

# --- 示例用法 ---

if \_\_name\_\_ == '\_\_main\_\_':

    # 初始化依赖的引擎/模型 (使用Mock对象代替真实实现)

    mock\_emotion\_engine = MockEmotionEngine()

    mock\_vision\_model = MockVisionModel()

    mock\_nlp\_model = MockNLPModel()

    # 创建共享记忆生成器实例

    memory\_generator = SharedMemoryGenerator(

        emotion\_engine=mock\_emotion\_engine,

        vision\_model=mock\_vision\_model,

        nlp\_model=mock\_nlp\_model

    )

    # 添加用户数据 (示例)

    user1\_data = [

        {'type': 'photo', 'content': 'beach.jpg', 'timestamp': '2024-07-01T10:00:00', 'location': (34.0522, -118.2437)},

        {'type': 'text', 'content': '在洛杉矶海滩，非常开心！', 'timestamp': '2024-07-01T10:05:00', 'location': (34.0522, -118.2437)},

        {'type': 'video', 'content': 'sunset.mp4', 'timestamp': '2024-07-01T18:30:00', 'location': (34.0522, -118.2437)}

    ]

    user2\_data = [

        {'type': 'photo', 'content': 'food\_market.jpg', 'timestamp': '2024-07-01T12:30:00', 'location': (34.0588, -118.2488)},

        {'type': 'text', 'content': '午餐太美味了，当地的市场很有特色。', 'timestamp': '2024-07-01T12:35:00', 'location': (34.0588, -118.2488)},

        {'type': 'interaction', 'target\_user\_id': 'user1', 'interaction\_type': 'comment', 'content': '你的海滩照片很棒！', 'timestamp': '2024-07-01T14:00:00'}

    ]

    user3\_data = [

        {'type': 'photo', 'content': 'group\_photo\_beach.jpg', 'timestamp': '2024-07-01T10:15:00', 'location': (34.0522, -118.2437), 'tags': ['group']},

        {'type': 'text', 'content': '和大家一起在海边合影，难忘的时刻。', 'timestamp': '2024-07-01T10:20:00'}

    ]

    memory\_generator.add\_user\_data('user1', user1\_data)

    memory\_generator.add\_user\_data('user2', user2\_data)

    memory\_generator.add\_user\_data('user3', user3\_data)

    # 处理并生成共享记忆

    shared\_storyline = memory\_generator.process\_shared\_memories()

    if shared\_storyline:

        print("\n--- Shared Storyline ---")

        for moment in shared\_storyline:

            print(f"- Time: {moment['timestamp\_dt']}, User: {moment['user\_id']}, Type: {moment['type']}, Content: {moment.get('content', 'N/A')}, Significance: {moment.get('significance\_score', 0):.2f}")

    else:

        print("\nNo shared storyline generated.")

    # 示例：为特定用户生成故事线 (假设已处理过数据)

    # user1\_story = memory\_generator.generate\_storyline(user\_id='user1')

    # if user1\_story:

    # print("\n--- User1's Storyline ---")

    # for moment in user1\_story:

    # print(f"- Time: {moment['timestamp\_dt']}, Type: {moment['type']}, Content: {moment.get('content', 'N/A')}")

##### **4.2 个性化记忆筛选与定制单元**

• **功能描述**：  
允许用户根据个人偏好筛选记忆片段，进行自定义编辑和标注，并支持多种格式的记忆输出，满足个性化的回顾与分享需求。

• **核心功能与技术**：

1. **基于用户偏好的记忆片段智能筛选**

• **偏好学习**：分析用户历史互动行为（如点赞、收藏、分享特定类型的记忆）、明确设置的偏好标签，以及与特定人物或地点的关联度。

• **个性化过滤**：提供基于人物、地点、时间、事件类型、情绪标签等多维度筛选器。

• **推荐算法**：为用户推荐可能感兴趣但未主动关注的记忆片段。

2. **用户自定义编辑与丰富标注功能**

• **内容编辑**：支持对照片进行裁剪、旋转、滤镜调整；对视频进行剪辑、合并、添加背景音乐；对文字进行修改、润色。

• **交互式标注**：允许用户为记忆片段添加自定义标签、文字描述、语音评论、手绘涂鸦等。

• **隐私控制**：用户可以设置特定记忆片段的可见性（公开、好友可见、仅自己可见）。

3. **多种记忆输出格式与分享渠道支持**

• **格式支持**：支持生成图文并茂的电子游记（如Markdown、PDF）、动态影集（如MP4视频）、交互式时间轴网页等。

• **模板选择**：提供多种风格的游记和影集模板。

• **一键分享**：支持将生成的记忆作品快速分享到主流社交平台（如微信、微博、Instagram）或通过邮件、链接分享。

**python**

from enum import Enum

import uuid

class OutputFormat(Enum):

    MARKDOWN = "md"

    PDF = "pdf"

    MP4\_VIDEO = "mp4"

    HTML\_TIMELINE = "html"

class PrivacySetting(Enum):

    PUBLIC = "public"

    FRIENDS\_ONLY = "friends\_only"

    PRIVATE = "private"

class PersonalizedMemoryCustomizer:

    def \_\_init\_\_(self, user\_profile\_manager, media\_editor\_service, output\_generator\_service):

        """

        初始化个性化记忆筛选与定制单元。

        :param user\_profile\_manager: 用户画像与偏好管理模块实例

        :param media\_editor\_service: 媒体编辑服务实例 (用于图片/视频编辑)

        :param output\_generator\_service: 记忆输出生成服务实例 (如PDF, 视频生成)

        """

        self.user\_profile\_manager = user\_profile\_manager

        self.media\_editor\_service = media\_editor\_service

        self.output\_generator\_service = output\_generator\_service

    def \_get\_user\_preferences\_for\_memory(self, user\_id):

        """获取用户关于记忆筛选和定制的偏好"""

        # 示例偏好结构，实际应从 user\_profile\_manager 获取

        # preferences = {

        # 'filter\_by\_people': ['user\_B', 'user\_D'],

        # 'filter\_by\_locations': ['Paris', 'Kyoto'],

        # 'filter\_by\_event\_types': ['food\_experience', 'sightseeing'],

        # 'filter\_by\_emotion\_tags': ['happy', 'excited'],

        # 'preferred\_output\_format': OutputFormat.MP4\_VIDEO,

        # 'default\_privacy': PrivacySetting.FRIENDS\_ONLY

        # }

        return self.user\_profile\_manager.get\_memory\_customization\_preferences(user\_id)

    def filter\_shared\_moments(self, user\_id, shared\_storyline, custom\_filters=None):

        """

        根据用户偏好和自定义筛选条件筛选共享记忆片段。

        :param user\_id: 用户ID

        :param shared\_storyline: 共享记忆故事线 (由SharedMemoryGenerator生成)

        :param custom\_filters: 用户本次操作临时的筛选条件 (可选)

        :return: 筛选后的记忆时刻列表

        """

        preferences = self.\_get\_user\_preferences\_for\_memory(user\_id)

        active\_filters = {\*\*preferences, \*\*(custom\_filters or {})} # 合并并优先使用临时过滤器

        filtered\_moments = []

        for moment in shared\_storyline:

            passes\_filter = True

            # 示例筛选逻辑 (实际应用中会更复杂和灵活)

            if active\_filters.get('filter\_by\_people'):

                # 假设moment有 'participants' 或 'user\_id' 字段

                moment\_participants = set(moment.get('participants', [moment.get('user\_id')]))

                if not moment\_participants.intersection(set(active\_filters['filter\_by\_people'])):

                    passes\_filter = False

            if passes\_filter and active\_filters.get('filter\_by\_locations') and moment.get('location\_name'):

                if moment['location\_name'] not in active\_filters['filter\_by\_locations']:

                    passes\_filter = False

            if passes\_filter and active\_filters.get('filter\_by\_event\_types') and moment.get('event\_type'):

                if moment['event\_type'] not in active\_filters['filter\_by\_event\_types']:

                    passes\_filter = False

            if passes\_filter and active\_filters.get('filter\_by\_emotion\_tags') and moment.get('emotion\_tags'):

                if not set(moment['emotion\_tags']).intersection(set(active\_filters['filter\_by\_emotion\_tags'])):

                    passes\_filter = False

            if passes\_filter:

                filtered\_moments.append(moment)

        print(f"User {user\_id}: Filtered {len(shared\_storyline)} moments down to {len(filtered\_moments)}.")

        return filtered\_moments

    def edit\_moment\_content(self, user\_id, moment\_data, edits):

        """

        用户对单个记忆片段的内容进行编辑。

        :param user\_id: 用户ID

        :param moment\_data: 原始记忆片段数据

        :param edits: 编辑操作指令，例如：

                      {'photo\_edit': {'crop': [x,y,w,h], 'filter': 'sepia'},

                       'video\_edit': {'trim': [start\_sec, end\_sec], 'music\_id': 'song123'},

                       'text\_edit': '新的文字描述。'}

        :return: 编辑后的记忆片段数据 (可能包含指向新媒体资源的链接)

        """

        edited\_moment = moment\_data.copy()

        if moment\_data['type'] == 'photo' and edits.get('photo\_edit') and self.media\_editor\_service:

            edited\_moment['content\_url'] = self.media\_editor\_service.edit\_photo(

                original\_url=moment\_data['content\_url'],

                edits=edits['photo\_edit']

            )

        elif moment\_data['type'] == 'video' and edits.get('video\_edit') and self.media\_editor\_service:

            edited\_moment['content\_url'] = self.media\_editor\_service.edit\_video(

                original\_url=moment\_data['content\_url'],

                edits=edits['video\_edit']

            )

        elif moment\_data['type'] == 'text' and edits.get('text\_edit'):

            edited\_moment['content'] = edits['text\_edit']

        print(f"User {user\_id} edited moment {moment\_data.get('id', 'N/A')}.")

        return edited\_moment

    def add\_moment\_annotations(self, user\_id, moment\_data, annotations):

        """

        用户为记忆片段添加标注。

        :param user\_id: 用户ID

        :param moment\_data: 记忆片段数据

        :param annotations: 标注内容，例如：

                          {'tags': ['family', 'fun'], 'description': '美好的回忆',

                           'voice\_comment\_url': 'path/to/voice.mp3', 'drawing\_url': 'path/to/drawing.png'}

        :return: 更新标注后的记忆片段数据

        """

        annotated\_moment = moment\_data.copy()

        if 'annotations' not in annotated\_moment:

            annotated\_moment['annotations'] = {}

        for key, value in annotations.items():

            if key == 'tags' and isinstance(value, list):

                existing\_tags = set(annotated\_moment['annotations'].get('tags', []))

                existing\_tags.update(value)

                annotated\_moment['annotations']['tags'] = list(existing\_tags)

            else:

                annotated\_moment['annotations'][key] = value

        print(f"User {user\_id} added annotations to moment {moment\_data.get('id', 'N/A')}.")

        return annotated\_moment

    def set\_moment\_privacy(self, user\_id, moment\_data, privacy\_setting: PrivacySetting):

        """

        设置特定记忆片段的隐私等级。

        :param user\_id: 用户ID (操作者，应与记忆片段所有者匹配或有权限)

        :param moment\_data: 记忆片段数据

        :param privacy\_setting: PrivacySetting枚举值

        :return: 更新隐私设置后的记忆片段数据

        """

        # 实际应用中需要权限校验

        # if moment\_data.get('owner\_id') != user\_id and not self.user\_profile\_manager.has\_edit\_permission(user\_id, moment\_data):

        # raise PermissionError("User does not have permission to change privacy.")

        updated\_moment = moment\_data.copy()

        updated\_moment['privacy'] = privacy\_setting.value

        print(f"User {user\_id} set privacy of moment {moment\_data.get('id', 'N/A')} to {privacy\_setting.value}.")

        return updated\_moment

    def generate\_customized\_output(self, user\_id, personalized\_storyline, output\_format: OutputFormat, title="My Travel Memory", template\_id=None):

        """

        根据用户的选择生成特定格式的个性化记忆作品。

        :param user\_id: 用户ID

        :param personalized\_storyline: 经过筛选、编辑、标注的个性化故事线

        :param output\_format: OutputFormat枚举值

        :param title: 作品标题

        :param template\_id: 可选的模板ID

        :return: 生成的作品的URL或文件路径，或者生成任务的ID

        """

        if not self.output\_generator\_service:

            raise NotImplementedError("Output generator service is not configured.")

        output\_details = self.output\_generator\_service.generate(

            user\_id=user\_id,

            storyline\_data=personalized\_storyline,

            output\_format=output\_format.value,

            title=title,

            template\_id=template\_id

        )

        print(f"User {user\_id} initiated generation of '{title}' in {output\_format.value} format. Details: {output\_details}")

        return output\_details

# --- 示例辅助类 (需要用户根据实际情况实现) ---

class MockUserProfileManagerForCustomizer:

    def get\_memory\_customization\_preferences(self, user\_id):

        # 示例偏好

        if user\_id == 'userX':

            return {

                'filter\_by\_people': ['userY'],

                'filter\_by\_event\_types': ['adventure'],

                'preferred\_output\_format': OutputFormat.MARKDOWN,

                'default\_privacy': PrivacySetting.PRIVATE

            }

        return {}

class MockMediaEditorService:

    def edit\_photo(self, original\_url, edits):

        print(f"Editing photo {original\_url} with edits: {edits}")

        return f"edited\_{original\_url.split('/')[-1]}" # 返回模拟的编辑后文件名

    def edit\_video(self, original\_url, edits):

        print(f"Editing video {original\_url} with edits: {edits}")

        return f"edited\_{original\_url.split('/')[-1]}"

class MockOutputGeneratorService:

    def generate(self, user\_id, storyline\_data, output\_format, title, template\_id):

        output\_id = str(uuid.uuid4())

        print(f"Generating output '{title}' (ID: {output\_id}) for user {user\_id} in {output\_format} format. Template: {template\_id}. Moments: {len(storyline\_data)}")

        return {'job\_id': output\_id, 'status\_url': f"/api/outputs/{output\_id}/status"}

# --- 示例用法 ---

if \_\_name\_\_ == '\_\_main\_\_':

    # 假设 shared\_storyline\_example 来自 SharedMemoryGenerator 的输出

    # (包含 'user\_id', 'type', 'content\_url', 'location\_name', 'event\_type', 'emotion\_tags'等字段)

    shared\_storyline\_example = [

        {'id': 'moment1', 'user\_id': 'userX', 'type': 'photo', 'content\_url': 'photo1.jpg', 'location\_name': 'Beach', 'event\_type': 'leisure', 'emotion\_tags': ['happy', 'relaxed'], 'participants': ['userX', 'userY']},

        {'id': 'moment2', 'user\_id': 'userY', 'type': 'text', 'content': 'Amazing sunset!', 'location\_name': 'Beach', 'event\_type': 'sightseeing', 'emotion\_tags': ['awed']},

        {'id': 'moment3', 'user\_id': 'userX', 'type': 'video', 'content\_url': 'adventure\_video.mp4', 'event\_type': 'adventure', 'emotion\_tags': ['excited', 'thrilled'], 'participants': ['userX']}

    ]

    profile\_manager = MockUserProfileManagerForCustomizer()

    media\_editor = MockMediaEditorService()

    output\_generator = MockOutputGeneratorService()

    customizer = PersonalizedMemoryCustomizer(profile\_manager, media\_editor, output\_generator)

    test\_user\_id = 'userX'

    # 1. 筛选记忆片段

    print(f"\n--- Filtering for {test\_user\_id} ---")

    filtered\_moments = customizer.filter\_shared\_moments(test\_user\_id, shared\_storyline\_example)

    if not filtered\_moments:

        print("No moments after filtering.")

    else:

        print(f"Filtered moments for {test\_user\_id}: {len(filtered\_moments)}")

        for m in filtered\_moments: print(f" - {m['id']}: {m.get('event\_type')}")

        # 2. 编辑一个片段 (假设是第一个筛选出的片段)

        if filtered\_moments and filtered\_moments[0]['type'] == 'photo':

            print(f"\n--- Editing moment {filtered\_moments[0]['id']} ---")

            photo\_edits = {'crop': [10,10,100,100], 'filter': 'vintage'}

            edited\_photo\_moment = customizer.edit\_moment\_content(test\_user\_id, filtered\_moments[0], {'photo\_edit': photo\_edits})

            filtered\_moments[0] = edited\_photo\_moment # 更新列表中的片段

            print(f"Edited moment new content\_url: {edited\_photo\_moment['content\_url']}")

        # 3. 为一个片段添加标注

        if filtered\_moments:

            print(f"\n--- Annotating moment {filtered\_moments[0]['id']} ---")

            annotations\_to\_add = {'tags': ['favorite', 'summer2024'], 'description': 'Best day ever!'}

            annotated\_moment = customizer.add\_moment\_annotations(test\_user\_id, filtered\_moments[0], annotations\_to\_add)

            filtered\_moments[0] = annotated\_moment # 更新列表中的片段

            print(f"Annotations for moment: {annotated\_moment['annotations']}")

        # 4. 设置片段隐私

        if filtered\_moments:

            print(f"\n--- Setting privacy for moment {filtered\_moments[0]['id']} ---")

            private\_moment = customizer.set\_moment\_privacy(test\_user\_id, filtered\_moments[0], PrivacySetting.PRIVATE)

            filtered\_moments[0] = private\_moment

            print(f"Privacy for moment: {private\_moment['privacy']}")

        # 5. 生成个性化输出

        print(f"\n--- Generating output for {test\_user\_id} ---")

        user\_prefs\_for\_output = profile\_manager.get\_memory\_customization\_preferences(test\_user\_id)

        output\_result = customizer.generate\_customized\_output(

            test\_user\_id,

            filtered\_moments,

            user\_prefs\_for\_output.get('preferred\_output\_format', OutputFormat.MARKDOWN),

            title=f"{test\_user\_id}'s Awesome Trip"

        )

        print(f"Output generation result: {output\_result}")

##### **4.3 记忆情感化渲染与沉浸式呈现单元**

• **功能描述**：  
通过视觉、听觉等多感官元素的增强，以及结合VR/AR等新兴技术，提升旅游记忆的情感表达力和沉浸感。

• **技术手段与创新点**：

1. **结合情绪识别结果的视觉与听觉动态增强**

• **视觉渲染**：根据记忆片段关联的情绪状态（通过多模态情绪识别引擎获取），自动推荐或应用合适的色彩滤镜、光影效果、动态转场特效。

• 例如，快乐的记忆片段采用明亮、温暖的色调；宁静的记忆片段采用柔和、自然的色调。

• **听觉渲染**：

• **情感化背景音乐推荐**：基于记忆的情绪主题和内容，从曲库中智能匹配并推荐合适的背景音乐或音效。

• **语音情感增强**：对用户录制的语音评论，可进行音调、语速的微调，以更贴合其表达的情感。

2. **个性化与情境感知的主题元素推荐**

• **主题贴纸与装饰**：根据记忆内容（如美食、景点、活动）和情感氛围，推荐相关的主题贴纸、虚拟道具或装饰元素。

• **动态天气与环境模拟**：在呈现记忆时，可根据原始数据中的天气信息，模拟当时的天气效果（如阳光、雨滴、雪花），增强情境代入感。

3. **虚拟现实（VR）/增强现实（AR）的沉浸式记忆呈现**

• **VR场景再现**：将360度照片/视频或三维重建的场景，转化为VR体验，让用户仿佛身临其境地重温旅途。

• 支持在VR场景中进行交互，如点选特定物体查看相关信息或触发其他用户的评论。

• **AR记忆叠加**：

• **基于地理位置的AR记忆**：当用户回到曾经的旅行地点时，通过手机AR功能，可以在现实场景中看到当时拍摄的照片、视频或留下的虚拟标记。

• **基于图像识别的AR记忆**：扫描特定的实体照片或纪念品，可以触发相关的数字记忆内容（如播放一段视频、展示一段故事）。

• **技术挑战**：需要高质量的3D内容捕获与重建技术，以及低延迟的VR/AR渲染与交互技术。

4. **多感官融合与未来展望**

• 探索结合嗅觉、触觉等其他感官信息的记忆呈现方式（如通过特定气味装置或触觉反馈设备）。

• 利用脑机接口技术，探索更深层次的情感读取与记忆交互的可能性。

**python**

import random

class EmotionalRenderer:

    def \_\_init\_\_(self, emotion\_engine, asset\_library, vr\_ar\_service=None):

        """

        初始化记忆情感化渲染与沉浸式呈现单元。

        :param emotion\_engine: 多模态情绪识别引擎实例

        :param asset\_library: 包含滤镜、音乐、贴纸等资源的库

        :param vr\_ar\_service: VR/AR服务接口 (可选)

        """

        self.emotion\_engine = emotion\_engine

        self.asset\_library = asset\_library

        self.vr\_ar\_service = vr\_ar\_service

    def \_get\_emotion\_for\_moment(self, moment\_data):

        """从记忆片段中提取或推断情绪信息"""

        if 'emotion\_tags' in moment\_data and moment\_data['emotion\_tags']:

            # 优先使用已有的情绪标签

            return moment\_data['emotion\_tags']

        elif self.emotion\_engine:

            # 如果没有标签，尝试用情绪引擎分析 (假设moment\_data包含可分析内容)

            # 这是一个简化示例，实际可能需要更复杂的数据传递

            # emotion\_result = self.emotion\_engine.analyze\_moment(moment\_data)

            # return emotion\_result.get('dominant\_emotion')

            return [random.choice(['happy', 'calm', 'excited', 'nostalgic'])] # 模拟情绪

        return ['neutral']

    def apply\_visual\_enhancements(self, moment\_media\_url, moment\_emotions):

        """

        根据情绪应用视觉增强效果 (如滤镜、光影)。

        :param moment\_media\_url: 记忆片段的媒体URL (图片/视频)

        :param moment\_emotions: 记忆片段的情绪列表

        :return: 增强后的媒体URL或处理指令

        """

        primary\_emotion = moment\_emotions[0] if moment\_emotions else 'neutral'

        filter\_name = self.asset\_library.get\_filter\_for\_emotion(primary\_emotion)

        if filter\_name:

            print(f"Applying visual filter '{filter\_name}' for emotion '{primary\_emotion}' to {moment\_media\_url}")

            # 实际应用中会调用图像/视频处理服务

            return f"{moment\_media\_url}?filter={filter\_name}"

        return moment\_media\_url

    def suggest\_audio\_enhancements(self, moment\_content, moment\_emotions):

        """

        根据情绪和内容推荐背景音乐或音效。

        :param moment\_content: 记忆片段的内容 (文本、标签等)

        :param moment\_emotions: 记忆片段的情绪列表

        :return: 推荐的音频资源列表 (包含ID、名称、情感匹配度等)

        """

        primary\_emotion = moment\_emotions[0] if moment\_emotions else 'neutral'

        music\_suggestions = self.asset\_library.get\_music\_for\_emotion\_and\_content(

            primary\_emotion,

            moment\_content

        )

        print(f"Suggested music for emotion '{primary\_emotion}': {music\_suggestions}")

        return music\_suggestions

    def enhance\_voice\_comment(self, voice\_url, target\_emotion):

        """

        对语音评论进行情感化处理 (调整音调、语速等)。

        :param voice\_url: 原始语音评论的URL

        :param target\_emotion: 目标情感

        :return: 处理后的语音URL或处理指令

        """

        print(f"Enhancing voice comment {voice\_url} to reflect emotion '{target\_emotion}'")

        # 实际应用中会调用语音处理服务

        return f"{voice\_url}?emotion\_enhance={target\_emotion}"

    def recommend\_theme\_elements(self, moment\_data, moment\_emotions):

        """

        推荐主题贴纸、装饰等。

        :param moment\_data: 记忆片段数据 (包含内容、地点、活动类型等)

        :param moment\_emotions: 记忆片段的情绪列表

        :return: 推荐的贴纸/装饰列表

        """

        primary\_emotion = moment\_emotions[0] if moment\_emotions else 'neutral'

        content\_keywords = moment\_data.get('tags', []) + [moment\_data.get('event\_type', '')]

        stickers = self.asset\_library.get\_stickers(primary\_emotion, content\_keywords)

        print(f"Recommended stickers for moment (emotion: {primary\_emotion}, content: {content\_keywords}): {stickers}")

        return stickers

    def simulate\_environment\_effects(self, moment\_data):

        """

        根据原始天气信息模拟环境效果。

        :param moment\_data: 记忆片段数据 (应包含天气、时间等信息)

        :return: 环境效果指令 (如 'show\_rain\_effect', 'apply\_sunset\_lighting')

        """

        weather = moment\_data.get('weather\_condition')

        effects = []

        if weather == 'rainy':

            effects.append('show\_rain\_effect')

        elif weather == 'sunny' and moment\_data.get('time\_of\_day') == 'evening':

            effects.append('apply\_sunset\_lighting')

        print(f"Environmental effects for moment: {effects}")

        return effects

    def present\_in\_vr(self, user\_id, moment\_data\_or\_storyline):

        """

        将记忆片段或故事线在VR中呈现。

        :param user\_id: 用户ID

        :param moment\_data\_or\_storyline: 单个记忆片段或完整故事线数据

        :return: VR场景ID或访问链接

        """

        if not self.vr\_ar\_service:

            raise NotImplementedError("VR/AR service is not configured.")

        # 假设 moment\_data\_or\_storyline 包含360度媒体或3D模型信息

        if isinstance(moment\_data\_or\_storyline, dict) and moment\_data\_or\_storyline.get('media\_type\_360'):

            vr\_scene = self.vr\_ar\_service.create\_vr\_scene\_from\_360media(

                user\_id,

                moment\_data\_or\_storyline['media\_url\_360']

            )

            print(f"Presenting moment in VR: {vr\_scene}")

            return vr\_scene

        # 更多逻辑处理故事线等

        return None

    def overlay\_ar\_memory(self, user\_id, current\_location\_or\_image\_marker):

        """

        在现实场景中叠加AR记忆。

        :param user\_id: 用户ID

        :param current\_location\_or\_image\_marker: 当前地理位置信息或图像识别标记

        :return: AR叠加内容或指令

        """

        if not self.vr\_ar\_service:

            raise NotImplementedError("VR/AR service is not configured.")

        ar\_content = self.vr\_ar\_service.get\_ar\_memories\_for\_context(

            user\_id,

            current\_location\_or\_image\_marker

        )

        print(f"AR memories for context '{current\_location\_or\_image\_marker}': {ar\_content}")

        return ar\_content

# --- 示例辅助类 ---

class MockEmotionEngineForRenderer:

    def analyze\_moment(self, moment\_data):

        # 模拟情绪分析

        return {'dominant\_emotion': random.choice(['happy', 'sad', 'excited', 'calm']), 'scores': {}}

class MockAssetLibrary:

    def \_\_init\_\_(self):

        self.filters = {'happy': 'brighten\_warm', 'sad': 'blue\_tint\_subtle', 'calm': 'soft\_focus\_nature', 'excited': 'vibrant\_contrast', 'neutral': 'none'}

        self.music = {

            'happy': [{'id': 'm001', 'title': 'Upbeat Sunshine Pop'}, {'id': 'm002', 'title': 'Joyful Acoustic Strum'}],

            'calm': [{'id': 'm003', 'title': 'Peaceful Piano Melody'}],

            'excited': [{'id': 'm004', 'title': 'Energetic Rock Anthem'}]

        }

        self.stickers\_db = {

            ('happy', 'beach'): ['sun\_sticker.png', 'cocktail\_sticker.png'],

            ('excited', 'adventure'): ['mountain\_peak.png', 'compass\_rose.png']

        }

    def get\_filter\_for\_emotion(self, emotion):

        return self.filters.get(emotion, self.filters['neutral'])

    def get\_music\_for\_emotion\_and\_content(self, emotion, content\_keywords):

        # 简单匹配，实际会更复杂

        return self.music.get(emotion, [])

    def get\_stickers(self, emotion, content\_keywords):

        results = []

        for (e, kw), stickers\_list in self.stickers\_db.items():

            if e == emotion and any(k in content\_keywords for k in kw.split()): # 简单关键词匹配

                results.extend(stickers\_list)

        return list(set(results)) #去重

class MockVrArService:

    def create\_vr\_scene\_from\_360media(self, user\_id, media\_url\_360):

        scene\_id = f"vr\_scene\_{random.randint(1000,9999)}"

        print(f"VR Service: Created scene {scene\_id} for user {user\_id} with media {media\_url\_360}")

        return {'scene\_id': scene\_id, 'access\_url': f"https://vr.example.com/{scene\_id}"}

    def get\_ar\_memories\_for\_context(self, user\_id, context):

        # 模拟基于地理位置或图像标记的AR内容获取

        if isinstance(context, dict) and 'latitude' in context: # Geo-based

            print(f"AR Service: Fetching memories for user {user\_id} at geo: {context}")

            return [{'type': 'photo', 'url': 'ar\_photo\_nearby.jpg', 'caption': 'Remember this spot?'}]

        elif isinstance(context, str) and context.startswith('marker\_'): # Image-marker based

            print(f"AR Service: Fetching memories for user {user\_id} for marker: {context}")

            return [{'type': 'video', 'url': 'ar\_video\_marker.mp4', 'title': 'Marker Activated Memory'}]

        return []

# --- 示例用法 ---

if \_\_name\_\_ == '\_\_main\_\_':

    mock\_emotion\_engine = MockEmotionEngineForRenderer()

    mock\_asset\_library = MockAssetLibrary()

    mock\_vr\_ar\_service = MockVrArService()

    renderer = EmotionalRenderer(mock\_emotion\_engine, mock\_asset\_library, mock\_vr\_ar\_service)

    # 示例记忆片段数据 (通常来自 PersonalizedMemoryCustomizer 的输出)

    example\_moment = {

        'id': 'moment\_abc\_123',

        'user\_id': 'userZ',

        'type': 'photo',

        'content\_url': 'original\_photo.jpg',

        'media\_url\_360': '360\_photo\_experience.jpg', # 用于VR

        'emotion\_tags': ['happy', 'excited'],

        'tags': ['birthday', 'party'],

        'event\_type': 'celebration',

        'weather\_condition': 'sunny',

        'time\_of\_day': 'afternoon',

        'location': {'latitude': 34.0522, 'longitude': -118.2437} # 用于AR地理位置

    }

    print("\n--- Applying Visual Enhancements ---")

    moment\_emotions = renderer.\_get\_emotion\_for\_moment(example\_moment)

    enhanced\_media\_url = renderer.apply\_visual\_enhancements(example\_moment['content\_url'], moment\_emotions)

    print(f"Enhanced media URL: {enhanced\_media\_url}")

    print("\n--- Suggesting Audio Enhancements ---")

    audio\_suggestions = renderer.suggest\_audio\_enhancements(example\_moment, moment\_emotions)

    print(f"Audio suggestions: {audio\_suggestions}")

    # 假设有语音评论

    # voice\_comment\_url = "user\_voice\_happy.mp3"

    # enhanced\_voice = renderer.enhance\_voice\_comment(voice\_comment\_url, 'very\_happy')

    # print(f"Enhanced voice URL: {enhanced\_voice}")

    print("\n--- Recommending Theme Elements ---")

    theme\_elements = renderer.recommend\_theme\_elements(example\_moment, moment\_emotions)

    print(f"Recommended theme elements: {theme\_elements}")

    print("\n--- Simulating Environment Effects ---")

    env\_effects = renderer.simulate\_environment\_effects(example\_moment)

    print(f"Environment effects: {env\_effects}")

    if renderer.vr\_ar\_service:

        print("\n--- Presenting in VR ---")

        vr\_info = renderer.present\_in\_vr(example\_moment['user\_id'], example\_moment)

        if vr\_info: print(f"VR Presentation Info: {vr\_info}")

        print("\n--- Overlaying AR Memory (Geo-based) ---")

        ar\_geo\_content = renderer.overlay\_ar\_memory(example\_moment['user\_id'], example\_moment['location'])

        if ar\_geo\_content: print(f"AR Geo Content: {ar\_geo\_content}")

        print("\n--- Overlaying AR Memory (Marker-based) ---")

        ar\_marker\_content = renderer.overlay\_ar\_memory(example\_moment['user\_id'], 'marker\_001')

        if ar\_marker\_content: print(f"AR Marker Content: {ar\_marker\_content}")

#### **5. 社交图谱构建与演化引擎**

社交图谱构建与演化引擎负责描绘用户之间以及用户与兴趣点之间的复杂关系网络，并动态追踪这些关系的演变。该引擎为理解用户社交偏好、发现潜在社交连接、以及优化推荐策略提供基础。

##### **5.1 用户节点定义与丰富属性**

• **功能描述**：  
定义图谱中的核心元素——用户节点，并为其赋予多维度、动态更新的属性，全面刻画用户特征。

• **节点属性构成**：

1. **基础信息**：

• 用户ID（唯一标识）、昵称、头像、注册时间等。

• 人口统计学信息（可选，用户授权）：年龄、性别、地理区域等。

2. **情绪特征**：

• **长期情绪基线**：通过对用户历史情绪数据的统计分析，得到其常见的情绪状态和稳定性。

• **近期情绪状态**：由多模态情绪识别引擎实时或准实时更新的当前情绪向量。

• **情绪模式**：用户在特定情境下（如特定活动、与特定人互动）的情绪反应模式。

3. **兴趣标签与画像**：

• **显式兴趣**：用户主动填写的兴趣爱好、关注的主题、想去的目的地等。

• **隐式兴趣**：通过分析用户行为数据（浏览、搜索、收藏、参与活动、互动内容）推断出的潜在兴趣点，形成用户兴趣向量或主题分布。

• **兴趣强度与时效性**：每个兴趣标签都附带强度评分和最近活跃时间，以反映兴趣的持久性和当前热度。

4. **社交偏好**：

• **互动风格**：主动/被动、开放/内向等。

• **偏好关系类型**：倾向于建立短期玩伴关系还是长期深度连接。

• **信任用户列表/黑名单**。

5. **行为数据摘要**：

• 活动参与频率、分享频率、内容产出量等。

• 在平台上的活跃度指标。

• **数据存储与管理**：

• 采用图数据库（如Neo4j、JanusGraph）或关系型数据库结合JSON字段存储用户节点及其属性。

• 建立高效的索引机制，支持快速查询和属性更新。

**python**

import time

from typing import Dict, Any, List, Optional, Tuple

import uuid

# 辅助类和枚举

class EmotionProfile:

    """用户情绪特征"""

    def \_\_init\_\_(self, baseline: Dict[str, float], current\_vector: List[float], patterns: Dict[str, Any]):

        self.baseline = baseline # 如: {'happy': 0.6, 'sad': 0.1, ...}

        self.current\_vector = current\_vector # 如: [0.8, 0.1, 0.05, ...] (效价, 唤醒度, 优势度)

        self.patterns = patterns # 如: {'activity\_type\_A': 'calm', 'with\_user\_X': 'excited'}

class InterestProfile:

    """用户兴趣画像"""

    def \_\_init\_\_(self, explicit\_interests: Dict[str, float], implicit\_interests: Dict[str, float]):

        # 兴趣: 强度评分 (0-1)

        self.explicit\_interests = explicit\_interests # 如: {'hiking': 0.9, 'photography': 0.7}

        self.implicit\_interests = implicit\_interests # 如: {'museums': 0.6, 'street\_food': 0.8}

        self.last\_active\_time: Dict[str, float] = {}

    def update\_interest\_activity(self, interest\_tag: str):

        self.last\_active\_time[interest\_tag] = time.time()

    def get\_interest\_vector(self, all\_tags: List[str], decay\_factor: float = 0.01) -> List[float]:

        """生成考虑时效性的兴趣向量"""

        vector = []

        current\_time = time.time()

        for tag in all\_tags:

            score = 0.0

            base\_score = self.explicit\_interests.get(tag, 0.0) + self.implicit\_interests.get(tag, 0.0)

            if base\_score > 0:

                last\_active = self.last\_active\_time.get(tag, current\_time)

                time\_decay = max(0, 1 - decay\_factor \* (current\_time - last\_active) / (3600 \* 24)) # 按天衰减

                score = base\_score \* time\_decay

            vector.append(score)

        return vector

class SocialGraphNode:

    """社交图谱中的用户节点"""

    def \_\_init\_\_(self, user\_id: str, nickname: str, avatar\_url: str, registration\_time: float,

                 demographics: Optional[Dict[str, Any]] = None):

        self.user\_id: str = user\_id

        self.nickname: str = nickname

        self.avatar\_url: str = avatar\_url

        self.registration\_time: float = registration\_time

        self.demographics: Dict[str, Any] = demographics if demographics else {}

        self.emotion\_profile: Optional[EmotionProfile] = None

        self.interest\_profile: Optional[InterestProfile] = None

        self.social\_preferences: Dict[str, Any] = {

            'interaction\_style': 'unknown', # 'active', 'passive', 'open', 'introverted'

            'preferred\_relation\_type': 'unknown', # 'short\_term\_playmate', 'long\_term\_connection'

            'trusted\_users': [],

            'blocked\_users': []

        }

        self.behavior\_summary: Dict[str, Any] = {

            'activity\_participation\_frequency': 0,

            'sharing\_frequency': 0,

            'content\_production\_volume': 0,

            'platform\_activity\_score': 0.0

        }

    def update\_emotion\_profile(self, baseline: Dict[str, float], current\_vector: List[float], patterns: Dict[str, Any]):

        self.emotion\_profile = EmotionProfile(baseline, current\_vector, patterns)

    def update\_interest\_profile(self, explicit\_interests: Dict[str, float], implicit\_interests: Dict[str, float]):

        self.interest\_profile = InterestProfile(explicit\_interests, implicit\_interests)

    def update\_social\_preferences(self, preferences: Dict[str, Any]):

        self.social\_preferences.update(preferences)

    def update\_behavior\_summary(self, summary: Dict[str, Any]):

        self.behavior\_summary.update(summary)

    def \_\_repr\_\_(self):

        return f"SocialGraphNode(user\_id='{self.user\_id}', nickname='{self.nickname}')"

# 示例：用户节点定义与属性丰富

if \_\_name\_\_ == "\_\_main\_\_":

    # 创建用户节点

    user\_node\_A = SocialGraphNode(

        user\_id=str(uuid.uuid4()),

        nickname="旅行家小明",

        avatar\_url="http://example.com/avatars/xiaoming.png",

        registration\_time=time.time() - 86400 \* 30, # 30天前注册

        demographics={'age': 28, 'gender': 'male', 'city': '北京'}

    )

    # 更新情绪特征

    user\_node\_A.update\_emotion\_profile(

        baseline={'happy': 0.7, 'neutral': 0.2, 'adventurous': 0.6},

        current\_vector=[0.85, 0.7, 0.9], # 假设为 (积极, 兴奋, 主导)

        patterns={'with\_friends': 'joyful', 'solo\_travel': 'contemplative'}

    )

    # 更新兴趣画像

    user\_node\_A.update\_interest\_profile(

        explicit\_interests={'摄影': 0.9, '徒步': 0.8, '历史文化': 0.7},

        implicit\_interests={'当地美食': 0.75, '独立音乐': 0.6}

    )

    if user\_node\_A.interest\_profile:

      user\_node\_A.interest\_profile.update\_interest\_activity('摄影') # 模拟最近活跃于摄影

    # 更新社交偏好

    user\_node\_A.update\_social\_preferences(

        {'interaction\_style': 'active', 'preferred\_relation\_type': 'long\_term\_connection'}

    )

    # 更新行为数据摘要

    user\_node\_A.update\_behavior\_summary(

        {'activity\_participation\_frequency': 5, 'sharing\_frequency': 10, 'platform\_activity\_score': 0.85}

    )

    print(f"创建的用户节点: {user\_node\_A}")

    if user\_node\_A.emotion\_profile:

        print(f" 情绪基线: {user\_node\_A.emotion\_profile.baseline}")

    if user\_node\_A.interest\_profile:

        all\_interest\_tags = list(user\_node\_A.interest\_profile.explicit\_interests.keys()) + \

                            list(user\_node\_A.interest\_profile.implicit\_interests.keys())

        print(f" 兴趣向量 (考虑时效性): {user\_node\_A.interest\_profile.get\_interest\_vector(list(set(all\_interest\_tags)))}")

    print(f" 社交偏好: {user\_node\_A.social\_preferences}")

##### **5.2 关系边定义与动态权重计算**

• **功能描述**：  
定义图谱中连接用户节点之间的"边"，用以表示不同类型的社交关系，并通过动态计算的权重来量化关系的强度和性质。

• **关系类型定义**：

1. **共同经历关系**：

• **共同参与活动**：用户共同参与了某次旅行、聚会或线上活动。

• **共同到访地点**：用户在相近时间段内到访过同一地理位置或景点。

2. **互动关系**：

• **直接互动**：点赞、评论、私信、分享、@提及等。

• **间接互动**：共同关注同一话题、对同一内容产生相似情绪反应等。

3. **情感连接关系**：

• **情绪共鸣**：在特定事件或互动中表现出相似或互补的情绪状态。

• **情感支持**：一方为另一方提供了情感上的安慰、鼓励或理解。

4. **兴趣相似关系**：

• 基于用户兴趣画像计算出的兴趣相似度达到一定阈值。

5. **关注/好友关系**（若平台提供此类功能）：

• 用户之间明确建立的单向关注或双向好友连接。

• **关系权重计算模型**：

• **频率与新近度**：互动频率越高、最近互动时间越近，权重越大（可采用时间衰减函数）。

• **互动深度**：评论比点赞权重高，深度交流比浅层互动权重高。

• **情感强度**：共同经历强烈正面或负面情绪的事件，关系权重相应调整。

• **共同点数量**：共同参与的活动越多、共同兴趣越多，关系权重越高。

• **关系持续时间**：长期稳定的互动关系权重更高。

• **权重归一化**：将不同因素计算得到的权重进行归一化处理，确保可比性。

• **动态更新**：每当新的互动行为发生或用户属性变化时，实时或定期重新计算相关边的权重。

**python**

from enum import Enum

import time

from typing import Dict, Any, List, Optional, Tuple

import math

# (复用之前定义的 SocialGraphNode, EmotionProfile, InterestProfile)

class RelationType(Enum):

    COMMON\_ACTIVITY = "common\_activity"

    COMMON\_LOCATION = "common\_location"

    DIRECT\_INTERACTION\_LIKE = "direct\_interaction\_like"

    DIRECT\_INTERACTION\_COMMENT = "direct\_interaction\_comment"

    DIRECT\_INTERACTION\_MESSAGE = "direct\_interaction\_message"

    EMOTIONAL\_RESONANCE = "emotional\_resonance"

    EMOTIONAL\_SUPPORT = "emotional\_support"

    INTEREST\_SIMILARITY = "interest\_similarity"

    FOLLOW = "follow"

    FRIEND = "friend"

class SocialGraphEdge:

    """社交图谱中的关系边"""

    def \_\_init\_\_(self, from\_node: SocialGraphNode, to\_node: SocialGraphNode, relation\_type: RelationType,

                 timestamp: float, properties: Optional[Dict[str, Any]] = None):

        self.edge\_id: str = str(uuid.uuid4())

        self.from\_node: SocialGraphNode = from\_node

        self.to\_node: SocialGraphNode = to\_node

        self.relation\_type: RelationType = relation\_type

        self.timestamp: float = timestamp # 关系建立或最近一次互动的时间戳

        self.properties: Dict[str, Any] = properties if properties else {}

        self.weight: float = 0.0

        self.update\_weight() # 初始化时计算一次权重

    def update\_weight(self, new\_interaction\_properties: Optional[Dict[str, Any]] = None):

        """根据关系类型、频率、新近度、深度、情感强度等计算权重"""

        base\_weight = 0.1 # 基础权重

        current\_time = time.time()

        # 1. 新近度 (Recency)

        time\_diff\_days = (current\_time - self.timestamp) / (3600 \* 24)

        recency\_score = math.exp(-0.1 \* time\_diff\_days) # 指数衰减，0.1为衰减因子

        # 2. 频率 (Frequency) - 假设在properties中记录了互动次数

        interaction\_count = self.properties.get('interaction\_count', 1)

        frequency\_score = math.log1p(interaction\_count) # log(1+x) 避免count为0或1时差异过大

        # 3. 互动深度 (Depth) - 不同类型互动赋予不同基础权重

        depth\_score = 1.0

        if self.relation\_type == RelationType.DIRECT\_INTERACTION\_COMMENT:

            depth\_score = 1.5

        elif self.relation\_type == RelationType.DIRECT\_INTERACTION\_MESSAGE:

            depth\_score = 2.0

        elif self.relation\_type == RelationType.EMOTIONAL\_SUPPORT:

            depth\_score = 2.5

        # 4. 情感强度 (Emotional Intensity) - 假设在properties中记录

        emotional\_intensity = self.properties.get('emotional\_intensity', 0.0) # 0-1范围

        emotional\_score = 1.0 + emotional\_intensity # 基础1，加上强度

        # 5. 共同点数量 (Commonality) - 比如共同参与的活动数量

        commonality\_count = self.properties.get('common\_activities\_count', 0)

        commonality\_score = math.log1p(commonality\_count)

        # 综合计算权重

        # 这里的权重计算模型可以非常复杂，具体取决于业务需求

        # 示例：加权平均或更复杂的模型

        calculated\_weight = (base\_weight +

                             recency\_score \* 0.3 +

                             frequency\_score \* 0.2 +

                             depth\_score \* 0.2 +

                             emotional\_score \* 0.15 +

                             commonality\_score \* 0.15)

        # 权重归一化 (可选, 例如限制在0-1之间)

        self.weight = max(0, min(1, calculated\_weight / 5.0)) # 假设最大可能权重和为5

        if new\_interaction\_properties:

            self.properties.update(new\_interaction\_properties)

            self.timestamp = time.time() # 更新互动时间

            # 互动后重新计算权重

            # (这里可以简化，直接调用self.update\_weight()，但为了清晰，先更新属性再计算)

            # 假设 interaction\_count 等属性已在 new\_interaction\_properties 中更新

            self.update\_weight() # 递归调用，但注意避免无限递归，实际应拆分

    def \_\_repr\_\_(self):

        return f"SocialGraphEdge(from='{self.from\_node.nickname}', to='{self.to\_node.nickname}', type='{self.relation\_type.value}', weight={self.weight:.2f})"

# 示例：关系边定义与动态权重计算

if \_\_name\_\_ == "\_\_main\_\_":

    # (假设 user\_node\_A 和 user\_node\_B 已经创建并填充了属性)

    user\_node\_A = SocialGraphNode(user\_id="user\_A", nickname="小明", avatar\_url="", registration\_time=time.time())

    user\_node\_B = SocialGraphNode(user\_id="user\_B", nickname="小红", avatar\_url="", registration\_time=time.time())

    # 1. 建立一个“共同参与活动”的关系

    common\_activity\_edge = SocialGraphEdge(

        from\_node=user\_node\_A,

        to\_node=user\_node\_B,

        relation\_type=RelationType.COMMON\_ACTIVITY,

        timestamp=time.time() - 86400 \* 5, # 5天前

        properties={'activity\_name': '西藏自驾游', 'interaction\_count': 1, 'common\_activities\_count': 1}

    )

    print(common\_activity\_edge)

    # 2. 模拟一次新的互动：小明给小红的游记评论 (DIRECT\_INTERACTION\_COMMENT)

    # 假设这条边已存在或新创建

    comment\_edge = SocialGraphEdge(

        from\_node=user\_node\_A,

        to\_node=user\_node\_B,

        relation\_type=RelationType.DIRECT\_INTERACTION\_COMMENT,

        timestamp=time.time() - 86400 \* 1, # 1天前

        properties={'interaction\_count': 1, 'comment\_length': 50}

    )

    print(f"评论关系 (初始): {comment\_edge}")

    # 模拟小明又评论了一次，更新关系权重

    comment\_edge.properties['interaction\_count'] += 1

    comment\_edge.timestamp = time.time() # 更新为当前时间

    comment\_edge.update\_weight() # 手动触发权重更新

    print(f"评论关系 (更新后): {comment\_edge}")

    # 3. 建立一个“兴趣相似”的关系

    # 假设已通过兴趣画像计算出相似度

    interest\_similarity\_edge = SocialGraphEdge(

        from\_node=user\_node\_A,

        to\_node=user\_node\_B,

        relation\_type=RelationType.INTEREST\_SIMILARITY,

        timestamp=time.time(),

        properties={'similarity\_score': 0.85} # 假设预先计算的相似度

    )

    # 兴趣相似的权重计算可以特殊处理，例如直接使用similarity\_score

    interest\_similarity\_edge.weight = interest\_similarity\_edge.properties.get('similarity\_score', 0.0)

    print(interest\_similarity\_edge)

    # 4. 情感支持关系

    support\_edge = SocialGraphEdge(

        from\_node=user\_node\_B, # 小红支持小明

        to\_node=user\_node\_A,

        relation\_type=RelationType.EMOTIONAL\_SUPPORT,

        timestamp=time.time() - 86400 \* 2, # 2天前

        properties={'interaction\_count': 1, 'emotional\_intensity': 0.9, 'support\_type': 'encouragement'}

    )

    print(support\_edge)

##### **5.3 图谱动态更新与智能分析应用**

• **功能描述**：  
确保社交图谱能够实时反映用户关系网络的最新状态，并基于图谱进行深度分析，挖掘社区结构、识别关键节点、预测关系趋势，为上层应用提供支持。

• **核心机制与应用**：

1. **基于新互动数据的图谱实时更新**

• **事件驱动更新**：当系统捕获到新的用户行为（如新的互动、新的活动参与、情绪状态变化）时，触发图谱更新流程。

• **增量更新**：仅更新受影响的节点属性和关系边权重，避免全量重算，提高效率。

• **数据一致性保障**：确保图谱数据与用户行为日志、情绪数据等源数据的一致性。

2. **社区发现与用户群体画像**

• **算法应用**：采用Louvain、LPA（标签传播算法）、Girvan-Newman等社区发现算法，识别图谱中紧密连接的用户子群（社区）。

• **社区特征分析**：对每个社区内的用户属性进行统计分析，形成社区画像（如"美食爱好者社群"、“周末徒步小队”）。

• **应用场景**：为用户推荐合适的社群、辅助活动组织者定位目标人群。

3. **关键节点识别与影响力分析**

• **中心性算法**：计算节点的度中心性（Degree Centrality）、介数中心性（Betweenness Centrality）、特征向量中心性（Eigenvector Centrality）等指标，识别网络中的核心人物或意见领袖。

• **影响力传播模型**：模拟信息或情绪在图谱中的传播路径和影响范围。

• **应用场景**：识别网红潜力用户、优化信息推送策略、助力活动推广。

4. **关系强度变化追踪与预测**

• **时序分析**：追踪特定用户对之间关系权重的历史变化趋势。

• **预测模型**：基于历史数据和用户行为模式，预测未来关系强化的可能性或减弱的风险（如流失预警）。

• **应用场景**：个性化关系维护提醒、潜在冲突预警。

5. **链接预测与潜在关系挖掘**

• **算法应用**：基于图谱结构特征（如共同邻居数量、路径长度）和节点属性相似度，预测未来可能建立连接的用户对。

• **应用场景**：为用户推荐可能认识的人、发现新的社交机会。

**python**

import time

from typing import Dict, Any, List, Optional, Tuple, Set

import networkx as nx # 使用networkx库进行图分析

from collections import defaultdict, Counter

import uuid

# (复用之前定义的 SocialGraphNode, SocialGraphEdge, RelationType, EmotionProfile, InterestProfile)

class SocialGraph:

    """社交图谱核心类，管理节点和边，并提供分析功能"""

    def \_\_init\_\_(self):

        self.nodes: Dict[str, SocialGraphNode] = {}

        self.edges: Dict[str, SocialGraphEdge] = {}

        self.graph = nx.MultiDiGraph() # networkx图对象，支持有向多重图

    def add\_node(self, node: SocialGraphNode):

        if node.user\_id not in self.nodes:

            self.nodes[node.user\_id] = node

            self.graph.add\_node(node.user\_id, \*\*node.\_\_dict\_\_) # 将节点属性也加入

            print(f"节点添加: {node.nickname}")

        else:

            # 更新节点属性 (如果需要)

            self.nodes[node.user\_id] = node

            self.graph.nodes[node.user\_id].update(node.\_\_dict\_\_)

            print(f"节点更新: {node.nickname}")

    def add\_edge(self, edge: SocialGraphEdge):

        if edge.edge\_id not in self.edges:

            # 确保节点存在于图中

            if edge.from\_node.user\_id not in self.nodes:

                self.add\_node(edge.from\_node)

            if edge.to\_node.user\_id not in self.nodes:

                self.add\_node(edge.to\_node)

            self.edges[edge.edge\_id] = edge

            self.graph.add\_edge(edge.from\_node.user\_id, edge.to\_node.user\_id,

                                key=edge.edge\_id, # 多重图需要key

                                type=edge.relation\_type.value,

                                weight=edge.weight,

                                timestamp=edge.timestamp,

                                \*\*edge.properties)

            print(f"边添加: {edge.from\_node.nickname} -> {edge.to\_node.nickname} ({edge.relation\_type.value}), 权重: {edge.weight:.2f}")

        else:

            # 更新边属性 (如果需要，例如权重变化)

            self.edges[edge.edge\_id] = edge

            self.graph.edges[edge.from\_node.user\_id, edge.to\_node.user\_id, edge.edge\_id].update({

                'weight': edge.weight,

                'timestamp': edge.timestamp,

                \*\*edge.properties

            })

            print(f"边更新: {edge.from\_node.nickname} -> {edge.to\_node.nickname} ({edge.relation\_type.value}), 新权重: {edge.weight:.2f}")

    def handle\_new\_interaction(self, from\_user\_id: str, to\_user\_id: str, interaction\_type: RelationType, properties: Dict[str, Any]):

        """处理新的用户互动，触发图谱更新"""

        if from\_user\_id not in self.nodes or to\_user\_id not in self.nodes:

            print(f"错误: 互动节点 {from\_user\_id} 或 {to\_user\_id} 不存在.")

            return

        from\_node = self.nodes[from\_user\_id]

        to\_node = self.nodes[to\_user\_id]

        # 查找是否已存在此类关系边，如果存在则更新，否则创建新边

        existing\_edge\_id = None

        for edge\_id, edge\_data in self.graph.get\_edge\_data(from\_user\_id, to\_user\_id, default={}).items():

            if edge\_data.get('type') == interaction\_type.value:

                existing\_edge\_id = edge\_id

                break

        if existing\_edge\_id and existing\_edge\_id in self.edges:

            edge = self.edges[existing\_edge\_id]

            # 更新互动计数等属性

            edge.properties['interaction\_count'] = edge.properties.get('interaction\_count', 0) + 1

            edge.properties.update(properties) # 更新其他相关属性

            edge.timestamp = time.time()

            edge.update\_weight() # 重新计算权重

            self.add\_edge(edge) # 调用add\_edge进行更新

        else:

            new\_edge = SocialGraphEdge(from\_node, to\_node, interaction\_type, time.time(), properties)

            new\_edge.properties['interaction\_count'] = 1 # 首次互动

            new\_edge.update\_weight()

            self.add\_edge(new\_edge)

        print(f"图谱因新互动更新: {from\_node.nickname} 对 {to\_node.nickname} 进行 {interaction\_type.value}")

    def find\_communities(self, algorithm: str = 'louvain') -> List[Set[str]]:

        """社区发现"""

        if not self.graph.nodes:

            return []

        # 为了社区发现，通常在无向图上进行，或者对有向图进行转换

        # 这里简单地使用 MultiDiGraph 的 to\_undirected()，实际应用中可能需要更复杂的处理

        undirected\_graph = self.graph.to\_undirected()

        if algorithm == 'louvain':

            # networkx 的 louvain\_communities 需要一个非多重图

            # 我们先简化图，权重可以取平均或最大等

            simple\_undirected\_graph = nx.Graph()

            for u, v, data in undirected\_graph.edges(data=True):

                if simple\_undirected\_graph.has\_edge(u,v):

                    simple\_undirected\_graph[u][v]['weight'] = max(simple\_undirected\_graph[u][v]['weight'], data.get('weight',0))

                else:

                    simple\_undirected\_graph.add\_edge(u,v, weight=data.get('weight',0))

            if not simple\_undirected\_graph.edges(): return []

            communities = list(nx.community.louvain\_communities(simple\_undirected\_graph, weight='weight'))

            return communities

        # 可以扩展支持其他算法如LPA等

        return []

    def get\_community\_profiles(self, communities: List[Set[str]]) -> List[Dict[str, Any]]:

        """分析社区特征，形成社区画像"""

        profiles = []

        for i, community\_nodes in enumerate(communities):

            profile = {'community\_id': i, 'size': len(community\_nodes), 'members': list(community\_nodes)}

            # 示例：统计社区内最常见的兴趣 (需要节点有InterestProfile)

            all\_interests = []

            for node\_id in community\_nodes:

                node = self.nodes.get(node\_id)

                if node and node.interest\_profile:

                    all\_interests.extend(node.interest\_profile.explicit\_interests.keys())

                    all\_interests.extend(node.interest\_profile.implicit\_interests.keys())

            if all\_interests:

                profile['top\_interests'] = [item[0] for item in Counter(all\_interests).most\_common(3)]

            profiles.append(profile)

        return profiles

    def get\_key\_nodes(self, top\_n: int = 5) -> Dict[str, List[Tuple[str, float]]]:

        """识别关键节点 (基于度中心性、介数中心性等)"""

        if not self.graph.nodes: return {}

        key\_nodes\_summary = {}

        try:

            key\_nodes\_summary['degree\_centrality'] = sorted(

                nx.degree\_centrality(self.graph).items(), key=lambda item: item[1], reverse=True)[:top\_n]

            # Betweenness centrality 计算成本较高，对于大图可能需要采样或近似算法

            if self.graph.number\_of\_nodes() < 1000: # 只对小图计算

                 key\_nodes\_summary['betweenness\_centrality'] = sorted(

                    nx.betweenness\_centrality(self.graph, weight='weight').items(), key=lambda item: item[1], reverse=True)[:top\_n]

            # Eigenvector centrality 要求图是连通的，且不能是MultiDiGraph，需要转换

            # simple\_graph = nx.Graph(self.graph) # 转换为简单图

            # if nx.is\_connected(simple\_graph.to\_undirected()):

            # key\_nodes\_summary['eigenvector\_centrality'] = sorted(

            # nx.eigenvector\_centrality\_numpy(simple\_graph, weight='weight').items(), key=lambda item: item[1], reverse=True)[:top\_n]

        except Exception as e:

            print(f"计算中心性时出错: {e}")

        return key\_nodes\_summary

    def predict\_links(self, user\_id: str, top\_n: int = 5) -> List[Tuple[str, float]]:

        """链接预测：为指定用户推荐可能认识的人"""

        if user\_id not in self.graph or not self.graph.nodes:

            return []

        # 使用 Jaccard 系数作为示例 (基于共同邻居)

        # networkx.jaccard\_coefficient 返回一个迭代器，包含 (u, v, p) 元组

        # 我们需要为特定用户 user\_id 找到与其他所有非邻居节点的 Jaccard 系数

        potential\_links = []

        all\_other\_nodes = [n for n in self.graph.nodes() if n != user\_id and not self.graph.has\_edge(user\_id, n)]

        # nx.jaccard\_coefficient(self.graph.to\_undirected(), [(user\_id, target\_node) for target\_node in all\_other\_nodes])

        # 上述方法对于多重图和有向图可能不直接适用，需要转换为适合的图结构

        # 简化：我们手动计算共同好友

        user\_neighbors = set(self.graph.successors(user\_id)) | set(self.graph.predecessors(user\_id))

        scores = {}

        for target\_node\_id in all\_other\_nodes:

            if target\_node\_id == user\_id or target\_node\_id in user\_neighbors: # 跳过自己和已连接的

                continue

            target\_neighbors = set(self.graph.successors(target\_node\_id)) | set(self.graph.predecessors(target\_node\_id))

            common\_neighbors = user\_neighbors.intersection(target\_neighbors)

            union\_size = len(user\_neighbors.union(target\_neighbors))

            if union\_size == 0: continue

            jaccard\_score = len(common\_neighbors) / union\_size

            if jaccard\_score > 0:

                scores[target\_node\_id] = jaccard\_score

        sorted\_links = sorted(scores.items(), key=lambda item: item[1], reverse=True)

        return [(self.nodes[uid].nickname if uid in self.nodes else uid, score) for uid, score in sorted\_links[:top\_n]]

# 示例：图谱动态更新与智能分析应用

if \_\_name\_\_ == "\_\_main\_\_":

    social\_graph = SocialGraph()

    # 1. 创建并添加用户节点 (假设 SocialGraphNode, EmotionProfile, InterestProfile 已定义并可用)

    user\_data = [

        ("user1", "探险家阿强", {'摄影': 0.8, '登山': 0.9}, {'徒步':0.7}),

        ("user2", "美食家小丽", {'烹饪': 0.9, '餐厅打卡': 0.85}, {'甜点':0.7}),

        ("user3", "文艺青年小文", {'电影': 0.9, '阅读': 0.8}, {'展览':0.75}),

        ("user4", "运动达人小刚", {'健身': 0.95, '跑步': 0.8}, {'篮球':0.8}),

        ("user5", "摄影爱好者小赵", {'摄影': 0.9, '后期': 0.7}, {'旅游':0.8})

    ]

    for uid, name, explicit\_interests, implicit\_interests in user\_data:

        node = SocialGraphNode(user\_id=uid, nickname=name, avatar\_url="", registration\_time=time.time())

        node.update\_interest\_profile(explicit\_interests, implicit\_interests)

        social\_graph.add\_node(node)

    # 2. 添加关系边 (模拟一些初始关系)

    social\_graph.handle\_new\_interaction("user1", "user5", RelationType.INTEREST\_SIMILARITY, {'similarity\_score': 0.9, 'common\_interest': '摄影'})

    social\_graph.handle\_new\_interaction("user1", "user4", RelationType.COMMON\_ACTIVITY, {'activity\_name': '黄山徒步'})

    social\_graph.handle\_new\_interaction("user2", "user3", RelationType.DIRECT\_INTERACTION\_COMMENT, {'comment\_on': '电影影评'})

    social\_graph.handle\_new\_interaction("user4", "user1", RelationType.FOLLOW, {})

    social\_graph.handle\_new\_interaction("user5", "user1", RelationType.FRIEND, {})

    print("\n--- 初始图谱构建完成 ---")

    # 3. 模拟新的互动，触发图谱更新

    print("\n--- 模拟新互动 --- ")

    social\_graph.handle\_new\_interaction("user2", "user3", RelationType.DIRECT\_INTERACTION\_COMMENT, {'comment\_on': '新书推荐', 'length': 120})

    social\_graph.handle\_new\_interaction("user1", "user2", RelationType.COMMON\_LOCATION, {'location': '网红咖啡店', 'duration\_hours': 2})

    # 4. 社区发现

    print("\n--- 社区发现 --- ")

    communities = social\_graph.find\_communities(algorithm='louvain')

    if communities:

        print(f"发现 {len(communities)} 个社区:")

        for i, community in enumerate(communities):

            member\_nicknames = [social\_graph.nodes[uid].nickname for uid in community if uid in social\_graph.nodes]

            print(f" 社区 {i+1}: {member\_nicknames}")

        community\_profiles = social\_graph.get\_community\_profiles(communities)

        for profile in community\_profiles:

            print(f" 社区画像 {profile['community\_id']}: 大小={profile['size']}, 主要兴趣={profile.get('top\_interests', 'N/A')}")

    else:

        print("未能发现明显社区结构或图过小/不连通。")

    # 5. 关键节点识别

    print("\n--- 关键节点识别 --- ")

    key\_nodes = social\_graph.get\_key\_nodes()

    for centrality\_type, nodes in key\_nodes.items():

        print(f" {centrality\_type}:")

        for node\_id, score in nodes:

            nickname = social\_graph.nodes[node\_id].nickname if node\_id in social\_graph.nodes else node\_id

            print(f" - {nickname}: {score:.3f}")

    # 6. 链接预测 (为阿强推荐)

    print("\n--- 链接预测 (为探险家阿强推荐) --- ")

    potential\_friends\_for\_aqiang = social\_graph.predict\_links("user1")

    if potential\_friends\_for\_aqiang:

        print(" 可能认识的人:")

        for name, score in potential\_friends\_for\_aqiang:

            print(f" - {name} (Jaccard得分: {score:.3f})")

    else:

        print(" 没有发现新的潜在连接。")

    # 7. 关系强度变化追踪 (示例：假设我们有一个函数来获取特定边的历史权重)

    # def track\_relation\_strength(graph: SocialGraph, user\_id1: str, user\_id2: str, relation\_type: RelationType):

    # # 此处应从历史记录或边属性中获取权重变化

    # print(f"追踪 {user\_id1} 和 {user\_id2} 之间 {relation\_type.value} 关系强度变化 (功能待实现)")

    # track\_relation\_strength(social\_graph, "user2", "user3", RelationType.DIRECT\_INTERACTION\_COMMENT)

### **技术创新性**

本发明提出的社交互动引擎，在现有技术基础上进行了多方面的创新与突破，主要体现在以下几个方面：

1. **多模态情绪识别与情感计算的深度融合与应用创新**：

• **创新点详述**：

• **多模态数据源的协同感知**：本发明创新性地整合了多种异构数据源用于情绪识别，包括但不限于：

• **生理信号**：实时采集并分析心率变异性（HRV）、皮肤电导率（GSR）、体温等客观生理指标，这些指标能直接反映自主神经系统的活动，是情绪状态的可靠生物标记。

• **视觉信息**：通过摄像头捕捉面部微表情、头部姿态、肢体动作等，利用先进的计算机视觉算法进行精确识别和量化。

• **听觉信息**：分析用户语音的音调、语速、能量、韵律以及语音内容中的情感词汇，进行语音情绪识别。

• **文本信息**：对用户输入的文字、评论、聊天记录等进行自然语言处理，提取情感极性、情感强度和具体情感类别。

• **行为模式**：分析用户在应用内的操作行为，如点击速度、滑动模式、内容偏好变化等，作为情绪状态的辅助判断依据。

• **深度融合与上下文感知模型**：并非简单地叠加各模态的识别结果，而是构建了一个多层级的深度融合模型（如基于注意力机制的循环神经网络、Transformer等）。该模型能够：

• **动态加权**：根据不同情境下各模态数据的可靠性和相关性，动态调整其在最终情绪判断中的权重。

• **时序依赖建模**：有效捕捉情绪状态在时间序列上的动态演变过程，而非孤立地分析瞬时数据。

• **上下文情境理解**：结合用户当前所处的环境（如地理位置、活动类型）、社交互动对象等上下文信息，提升情绪识别的准确性和情境适应性。

• **细粒度与连续维度情绪表示**：超越传统离散情感分类（如喜怒哀乐），本发明致力于在连续的情感维度（如效价-唤醒度-优势度模型，Valence-Arousal-Dominance）上对情绪进行更精细的刻画和量化，能够识别更复杂和混合的情绪状态。

• **优势分析**：

• **显著提升准确性与鲁棒性**：多模态信息互补，克服了单一模态易受环境噪声、伪装等因素干扰的缺陷，使得情绪识别结果更稳定、更可靠。

• **更全面的情感洞察**：能够捕捉到用户内隐的、不自觉流露的真实情感，以及细微的情感波动，提供远超传统方法的情感洞察深度。

• **个性化情感模型**：通过持续学习用户个体的情感表达模式，可以构建个性化的情绪识别模型，进一步提升针对特定用户的识别精度。

• **为下游应用提供坚实基础**：高精度的实时情绪数据是实现真正情感化、智能化社交互动的前提。

2. **情绪驱动的个性化、情境化社交互动智能推荐**：

• **创新点详述**：

• **以实时情绪为核心驱动**：将多模态情绪识别引擎输出的实时、细粒度情绪状态作为社交推荐的首要和核心驱动因素，而非仅仅是辅助参考。

• **多维度情境感知与动态匹配**：推荐算法不仅考虑用户当前情绪，还深度融合了：

• **用户兴趣画像**：确保推荐的社交对象或活动符合用户的长期兴趣偏好。

• **地理位置信息**：优先推荐附近或同城的、具有线下互动可能性的社交机会。

• **动态社交图谱**：分析用户现有的社交关系强度、社区归属，推荐能够增强现有连接或拓展新连接的互动。

• **时间与活动背景**：根据当前时间（如白天/夜晚、工作日/周末）和用户正在参与或计划参与的活动类型，进行场景化推荐。

• **情绪匹配与引导策略**：

• **情绪共鸣推荐**：为用户推荐当前情绪状态相似或相近的社交伙伴或内容，以期产生情感共鸣。

• **情绪互补推荐**：在某些情境下（如用户感到失落时），推荐能够提供安慰、支持或积极引导的社交互动。

• **情绪调节推荐**：当识别到用户处于负面情绪状态时，主动推荐有助于改善情绪的活动、内容或社交对象。

• **可解释性与用户控制**：推荐结果会向用户提供一定的解释（如"因为你现在心情愉悦，推荐你参加这个派对"），并允许用户对推荐偏好进行调整。

• **优势分析**：

• **超越传统推荐范式**：突破了传统社交推荐主要依赖静态兴趣标签或历史行为的局限，更能满足用户当下、即时的情感和社交需求。

• **提升社交匹配精准度与用户满意度**：基于情绪的匹配更容易促成高质量的社交互动，从而显著提升用户体验和平台粘性。

• **促进积极情感体验**：通过智能的情绪引导和调节推荐，帮助用户建立更积极的社交连接，提升整体幸福感。

• **动态适应性强**：能够快速响应用户情绪和情境的变化，提供真正"懂你"的推荐。

3. **共享旅游记忆的自动化构建、情感化渲染与沉浸式再现**：

• **创新点详述**：

• **多用户异构数据智能融合与故事化重构**：

• 自动整合来自同一旅行活动中多个参与者的照片、视频、文字、位置轨迹、情绪记录等碎片化数据。

• 通过时间轴对齐、空间关联、语义理解和情感分析，智能识别旅途中的关键事件、精彩瞬间和情感高潮。

• 运用叙事学原理和算法（如基于模板、基于规则或机器学习模型），自动生成具有连贯故事情节和多重视角的共享旅游记忆。

• **深度情感化渲染与个性化定制**：

• 将记忆片段与对应时刻的情绪数据（来自多模态情绪识别引擎）紧密关联。

• 根据记忆的情感基调，自动推荐或应用合适的视觉滤镜、转场特效、背景音乐、音效等，增强记忆的情感表达力。

• 允许用户对自动生成的记忆故事进行个性化编辑、筛选、标注和补充，赋予用户对记忆的最终控制权。

• **VR/AR沉浸式记忆再现与互动**：

• 利用360度影像、三维场景重建等技术，将共享旅游记忆转化为可交互的VR场景，让用户能够"故地重游"，身临其境地回顾旅途点滴。

• 通过AR技术，在现实的旅行地点叠加展示相关的数字记忆（如照片、故事、其他用户的留言），或通过扫描实体纪念品触发AR记忆内容，实现虚实融合的记忆体验。

• **优势分析**：

• **从数据到故事的价值升华**：将原本散乱的个人数据升华为富有情感、可共享、可传承的集体记忆，极大地提升了旅游数据的价值。

• **增强情感连接与社交深度**：共同创造和回顾充满情感的共享记忆，能够显著增强参与者之间的情感纽带和社交认同感。

• **前所未有的回忆体验**：情感化渲染和VR/AR技术的应用，提供了远超传统照片、视频的回忆深度和沉浸感，让回忆"活起来"。

• **激发新的分享与互动模式**：创新的记忆呈现方式将激发用户更积极地记录、分享和与他人互动，形成良性循环。

4. **融合情绪特征的动态社交图谱构建、演化分析与深度应用**：

• **创新点详述**：

• **图谱节点与边的丰富语义表达**：

• **用户节点（Node）**：不仅包含用户的基本信息、兴趣标签，更创新性地融入了用户的长期情绪特征（如情绪基线、情绪模式）和短期实时情绪状态作为核心属性。

• **关系边（Edge）**：定义了多样化的社交关系类型（如共同经历、直接互动、情感共鸣等），并引入了基于互动频率、互动深度、情感强度、共同点等多维度因素动态计算的关系权重。

• **图谱的实时构建与动态演化追踪**：

• 基于用户在平台上的实时行为数据（包括情绪变化、社交互动、活动参与等），对社交图谱进行事件驱动的增量式更新，确保图谱的鲜活性和准确性。

• 记录和分析关系边的权重变化、新节点的加入、社区结构的变迁等图谱演化过程。

• **基于情绪的图谱分析与应用**：

• **情绪影响力分析**：识别在社交网络中具有较强情绪感染力或引导力的关键用户。

• **情绪传播路径模拟**：分析特定情绪在图谱中的传播方式和影响范围。

• **社区情绪氛围感知**：评估不同社群或用户群体的整体情绪状态及其稳定性。

• **关系质量评估**：结合情绪互动数据，更准确地评估用户间关系的积极性与健康度。

• **高级图谱挖掘技术应用**：运用社区发现算法（如Louvain）、链接预测模型（如基于图神经网络GNN）、影响力最大化算法等，从图谱中挖掘深层价值。

• **优势分析**：

• **更深刻的用户与关系理解**：融合情绪特征的社交图谱能够更全面、更动态、更深刻地揭示用户个性和用户间关系的本质。

• **驱动更智能的上层应用**：为个性化推荐（如推荐情绪匹配的伙伴）、社群运营（如识别社群情绪风险）、用户行为预测（如预测关系破裂）等提供前所未有的精准数据支持和决策依据。

• **促进健康的社交生态**：通过对社交图谱中情绪动态的洞察，有助于平台引导积极互动，预警和干预不良社交行为，构建更健康的社交生态系统。

• **发掘潜在商业价值**：基于深度图谱分析，可以衍生出更精准的用户分层、营销推广和增值服务设计。

## **附图说明**

图1：系统整体架构图  
图2：多模态情绪识别流程图  
图3：社交互动推荐算法流程图  
图4：记忆构建与检索机制图  
图5：社交图谱演化示意图

## **具体实施方式**

### **实施例1：系统架构与模块协作**

本发明提出的基于情绪识别与记忆构建的旅游社交互动系统，其核心为一高效协同的社交互动引擎。该引擎及系统整体采用模块化、分层化的架构设计，如图1所示（附图说明中图1：系统整体架构图），确保了系统的高内聚、低耦合、易扩展和可维护性。主要包括以下核心层面与关键模块：

**一、 数据采集与预处理层 (Data Acquisition and Preprocessing Layer)**

此层面是系统感知用户与环境的基础，负责从多样化的输入源实时或准实时地采集与用户旅游行为及状态相关的多模态数据，并进行初步的清洗、格式化和结构化处理，为上层分析模块提供高质量的数据输入。

• **核心模块**：

• **多模态数据采集模块**（详见技术方案1）：

• **文本数据采集单元**：通过API接口、爬虫或SDK集成，收集用户在平台内外的文本信息，如游记、评论、聊天记录、社交媒体帖子等。

• **图像数据采集单元**：通过移动端摄像头、用户上传接口，获取照片、短视频，并提取EXIF元数据。

• **音频数据采集单元**：通过麦克风输入，采集语音消息、环境声音、用户交谈片段。

• **行为数据采集单元**：通过前端埋点、后端日志，记录用户在App/Web端的交互行为，如点击、浏览、搜索、购买、停留时长、分享、点赞、关注等。

• **生理信号采集单元**：通过与可穿戴设备（如智能手环、手表）的蓝牙或API连接，获取心率、心率变异性（HRV）、皮肤电导率（GSR）、体温、运动数据等生理指标。

• **地理位置与环境数据采集单元**：通过GPS、基站定位、Wi-Fi指纹等方式获取用户实时地理位置、移动轨迹；通过第三方API获取天气、POI（Point of Interest）信息等环境数据。

• **数据预处理与标准化模块**：

• **数据清洗**：去除噪声数据、重复数据、无效数据、处理缺失值。

• **格式转换**：将不同来源、不同格式的数据（如图片转为统一编码、音频转为标准采样率）统一化。

• **数据标注与对齐**：对部分数据进行初步的语义标注（如图像内容标签、文本关键词），并进行多模态数据间的时间戳对齐和用户ID关联。

• **隐私保护处理**：对敏感数据进行脱敏、匿名化处理，确保符合数据安全与隐私法规要求。

**二、 智能分析与决策层 (Intelligent Analysis and Decision Layer)**

此层面是系统的“大脑”，负责对采集到的多模态数据进行深度分析和智能决策，核心在于情绪的精准识别、记忆的有效构建以及社交关系的动态理解。

• **核心模块**：

• **多模态情绪识别引擎**（详见技术方案2，附图说明中图2：多模态情绪识别流程图）：

• **单模态特征提取单元**：分别从文本、图像、音频、生理信号中提取与情绪相关的特征。

• **多模态特征融合单元**：采用早期融合（特征级）、晚期融合（决策级）或混合融合策略，结合注意力机制、深度神经网络等模型，综合判断用户情绪状态（如离散情绪类别、连续情绪维度值）。

• **情绪时序分析单元**：追踪用户情绪随时间的变化趋势和模式。

• **记忆构建引擎**（详见技术方案4，附图说明中图4：记忆构建与检索机制图）：

• **个人记忆生成与管理单元**：为单个用户构建个性化的旅游记忆时间轴，记录关键事件、情感节点和个人感悟。

• **共享旅游记忆生成与协同编辑单元**：整合同一旅行活动中多个用户的数据，构建统一的、多视角的共享记忆，并支持用户协同编辑和丰富记忆内容。

• **记忆情感化渲染与呈现单元**：结合情绪数据，对记忆内容进行视觉、听觉等方面的增强，并通过VR/AR等技术提供沉浸式回顾体验。

• **社交图谱构建与演化引擎**（详见技术方案5，附图说明中图5：社交图谱演化示意图）：

• **用户节点与关系边定义单元**：定义用户画像（含情绪特征、兴趣偏好）和多类型社交关系（如共同经历、情感连接）。

• **图谱动态更新与权重计算单元**：根据用户互动和行为实时更新图谱结构和关系强度。

• **图谱分析与挖掘单元**：进行社区发现、影响力分析、链接预测等。

**三、 智能推荐与应用服务层 (Intelligent Recommendation and Application Service Layer)**

此层面基于分析决策层的输出，为用户提供具体、智能化的社交互动服务和个性化内容推荐。

• **核心模块**：

• **社交互动推荐引擎**（详见技术方案3，附图说明中图3：社交互动推荐算法流程图）：

• **情绪匹配推荐单元**：基于用户实时情绪状态进行社交对象或活动的匹配。

• **兴趣与偏好匹配单元**：结合用户长期兴趣和短期意图进行推荐。

• **地理位置与场景感知推荐单元**：根据用户当前位置和所处场景提供LBS推荐。

• **活动与内容推荐单元**：推荐符合用户情绪和兴趣的旅游活动、内容资讯等。

• **推荐策略优化与评估单元**：通过A/B测试、多臂老虎机等算法持续优化推荐效果。

• **个性化记忆服务模块**：

• **记忆智能检索单元**：支持用户通过关键词、时间、地点、人物、情绪等多维度检索个人和共享记忆。

• **记忆智能推荐与回顾提醒单元**：在特定时间点（如周年纪念日）或当用户情绪与某段记忆关联时，主动推送相关记忆回顾。

**四、 接口与交互层 (Interface and Interaction Layer)**

此层面是系统与用户及外部系统交互的门户。

• **核心模块**：

• **用户交互界面（UI/UX）**：通过移动应用（App）、Web网站等形式，为用户提供直观、易用的操作界面，展示情绪状态、社交推荐、记忆内容等。

• **API服务接口层**：提供标准化的RESTful API或GraphQL接口，供前端应用调用，并支持与第三方服务（如支付、地图、社交平台）的集成。

• **通知与消息推送模块**：负责向用户发送系统通知、推荐提醒、社交互动消息等。

**模块协作流程概述：**

1. **数据驱动**：用户通过交互界面产生数据，数据采集层捕获并预处理这些多模态数据。

2. **智能分析**：预处理后的数据流向智能分析与决策层。多模态情绪识别引擎分析用户情绪；记忆构建引擎组织和生成旅游记忆；社交图谱引擎更新用户关系网络。

3. **服务生成**：分析结果传递给智能推荐与应用服务层。社交互动推荐引擎根据用户情绪、兴趣、位置以及图谱信息，生成个性化的社交对象、活动或内容推荐；记忆服务模块提供记忆的检索与智能回顾。

4. **结果呈现与反馈循环**：推荐结果和记忆内容通过API接口层，在用户交互界面上呈现给用户。用户的进一步交互行为又会产生新的数据，形成一个持续优化、动态演进的闭环系统。

该系统架构通过各层模块的紧密协作与信息共享，实现了从数据感知到智能决策再到个性化服务的完整流程，确保了基于情绪识别与记忆构建的旅游社交互动功能的有效实现。

### **实施例2：多模态情绪识别过程**

本实施例详细描述了本发明中多模态情绪识别引擎（详见技术方案2）的具体工作流程和技术实现细节，如图2（附图说明中图2：多模态情绪识别流程图）所示。假设用户小明正在参与一次城市探索旅游活动，并使用本发明所述的社交互动App。

**1. 数据采集与同步：**

• **面部表情数据**：App通过手机前置摄像头，在用户授权前提下，以特定频率（例如，每5秒或在用户进行特定操作如拍照、录制视频时）捕捉小明的面部图像帧。采集到的图像帧会附带时间戳。

• **语音数据**：当小明使用App内的语音输入功能（如发表语音游记、与同行者语音聊天）或在特定场景下（如系统检测到周边环境音量变化较大或用户主动开启录音），App通过手机麦克风录制小明的语音片段。语音片段同样附带时间戳，并进行降噪预处理。

• **文本数据**：小明在App内发布的文字游记、评论、聊天信息等文本内容，以及通过API接入的其在关联社交平台（需用户授权）上发布的与本次旅行相关的公开文本，都会被收集并关联时间戳。

• **生理信号数据**：若小明佩戴了与App兼容的智能手环，App通过蓝牙低功耗（BLE）技术，实时或准实时（例如，每分钟）采集其心率（HR）、心率变异性（HRV）、皮肤电导率（GSR）等生理数据，并同步时间戳。

• **行为数据**：App记录小明在应用内的操作行为，如浏览特定景点的时长、对某个推荐活动的点击率、分享内容的频率等，这些行为数据也带有时间戳。

• **数据同步**：所有采集到的多模态数据通过时间戳进行初步对齐，确保后续分析的是同一时间窗口或相近时间段内的用户状态。

**2. 单模态特征提取：**

• **面部表情特征提取单元**：

• **人脸检测与对齐**：对采集到的图像帧，首先采用高效的人脸检测算法（如MTCNN、RetinaFace）定位人脸区域，并进行关键点检测（如眼角、鼻尖、嘴角）和人脸对齐，消除姿态、光照变化的部分影响。

• **表情特征提取**：从对齐后的人脸图像中提取表情特征。可以采用基于深度学习的方法，如使用预训练的卷积神经网络（CNN，例如ResNet、VGG-Face）提取深层特征；或者采用传统方法，如提取面部动作单元（Action Units, AUs）的激活强度、LBP（Local Binary Patterns）、HOG（Histogram of Oriented Gradients）等特征。

• **语音情绪特征提取单元**：

• **语音分段与端点检测**：对语音信号进行有效语音段的识别和分割。

• **声学特征提取**：提取多种声学特征，包括韵律特征（如基频F0、共振峰、音高、语速、停顿）、音质特征（如MFCC、LPCC、声门参数、谐波噪声比HNR）、谱特征（如频谱图、梅尔频谱）等。

• **文本情感特征提取单元**：

• **文本预处理**：进行分词、去除停用词、词性标注、命名实体识别等。

• **情感特征提取**：采用词袋模型（BoW）、TF-IDF、Word2Vec、GloVe或BERT等预训练语言模型提取文本的语义和情感特征。构建情感词典，计算文本的情感极性（正面、负面、中性）和情感强度。

• **生理信号特征提取单元**：

• **时域特征**：如心率的平均值、标准差、最大最小值；GSR的平均幅值、峰值数量等。

• **频域特征**：如HRV的LF（低频功率）、HF（高频功率）、LF/HF比值等，这些与自主神经系统活动相关，间接反映情绪状态。

• **非线性特征**：如样本熵、近似熵等，用于分析生理信号的复杂性。

• **行为特征提取单元**：

• 将用户的点击、浏览、停留等行为序列化，或提取统计特征，如特定行为的发生频率、持续时间等，这些行为模式可能与用户的情绪状态相关联（例如，快速滑动可能表示烦躁，长时间停留可能表示感兴趣或愉悦）。

**3. 多模态特征融合与情绪分类：**

系统采用一种基于注意力机制的混合融合策略。

• **特征级早期融合（部分模态）**：例如，可以将面部表情的视觉特征和语音的声学特征在特征层进行拼接或通过更复杂的融合网络（如跨模态注意力网络）进行初步融合，生成视听融合特征。

• **决策级晚期融合（所有模态）**：

• 各个单模态（或初步融合后的多模态）特征分别输入到各自的分类器中（如SVM、LSTM、Transformer或专门设计的深度网络），得到每个模态对情绪的初步判断结果（如情绪类别概率分布）。

• **注意力加权融合**：引入注意力机制，根据不同模态在特定情境下对情绪表达的可靠性和贡献度，动态分配权重。例如，在光线较暗时，面部表情的权重可能会降低，而语音和文本的权重可能会提升。注意力权重可以通过一个小型神经网络根据上下文信息（如环境光照度、是否有语音输入）和各模态特征的置信度学习得到。

• **最终决策**：将加权后的各模态决策结果进行融合（如加权平均、投票、或再通过一个小型决策网络），输出最终的综合情绪状态。情绪状态可以表示为离散类别（如高兴、悲伤、愤怒、惊喜、平静、厌恶、恐惧）或连续维度（如效价Valence、唤醒度Arousal、支配度Dominance）。

**4. 情绪时序动态分析：**

• 不仅仅识别瞬时情绪，系统还会利用循环神经网络（RNN）、长短期记忆网络（LSTM）或Transformer等模型，对一段时间内的情绪序列进行建模，分析情绪的动态变化趋势、持续时间、转换模式等。例如，可以识别出小明的情绪从“平静”逐渐转变为“惊喜”，或者在某个景点停留期间，其“愉悦”情绪的平均强度。

**5. 输出与应用：**

• 识别出的情绪状态（如“小明当前情绪：高兴，置信度0.85；愉悦度：0.7，唤醒度：0.6”）会实时更新到小明的用户画像中。

• 该情绪信息将作为后续社交互动推荐引擎、记忆构建引擎等模块的重要输入，用于个性化推荐、情感化记忆生成等。

通过上述流程，本发明能够更准确、鲁棒地识别用户在旅游过程中的多模态情绪，为后续的个性化服务提供坚实基础。

### **实施例3：情绪驱动的互动推荐**

本实施例详细描述了本发明中社交互动推荐引擎（详见技术方案3）的具体工作流程和技术实现细节，如图3（附图说明中图3：社交互动推荐算法流程图）所示。以下以一个具体场景为例进行说明：

假设用户小红（用户ID：U2023120501）正在西湖景区游览，当前位于断桥残雪景点（经纬度：30.2595° N, 120.1485° E）。系统通过多模态情绪识别引擎实时分析其情绪状态。

**1. 用户状态分析与特征提取：**

• **实时情绪状态**：

• 当前主导情绪：兴奋（置信度0.85）

• 情绪向量：[愉悦度(Valence): 0.8, 唤醒度(Arousal): 0.7, 支配度(Dominance): 0.6]

• 情绪持续时间：近30分钟内持续处于积极情绪状态

• **用户画像特征**：

• **基础属性**：25岁，女性，来自上海

• **兴趣偏好**：

• 主要兴趣标签：[“摄影”:0.9, “美食”:0.8, “文化古迹”:0.7, “网红打卡”:0.8]

• 近期浏览内容：西湖十景攻略、杭州特色小吃、旅拍构图技巧

• **社交特征**：

• 活跃度：0.8（近7天日均社交互动次数>10）

• 社交风格：外向型（根据历史互动数据分析）

**2. 多维度匹配计算：**

• **情绪匹配计算**：

• **情绪相似度计算**：采用余弦相似度计算情绪向量的匹配程度def emotion\_similarity(user1\_emotion, user2\_emotion):
return cosine\_similarity(user1\_emotion, user2\_emotion)

• **情绪互补性分析**：考虑情绪的互补效应，如兴奋-兴奋（共鸣型）或兴奋-平静（平衡型）的匹配

• **情绪动态趋势分析**：比较情绪变化曲线的相似性

• **兴趣相似度计算**：

• **标签匹配**：使用Jaccard相似系数计算兴趣标签重叠度

• **内容协同过滤**：基于用户历史浏览、收藏、评论的内容计算兴趣相似度

• **深度兴趣建模**：使用深度学习模型（如DeepFM）对用户长短期兴趣进行建模

• **地理位置相关性计算**：

• **实时距离计算**：使用Haversine公式计算用户间的球面距离

• **轨迹相似度**：比较用户历史轨迹和预期轨迹的重叠度

• **场景相关性**：分析用户所处场景（如景点、餐厅、商场）的匹配度

**3. 候选用户筛选与排序：**

• **初筛条件**：

• 地理范围：当前位置500米范围内

• 时间窗口：预计在该区域停留时间重叠>30分钟

• 基础过滤：排除已拒绝/拉黑用户、超出年龄差异阈值用户等

• **多目标排序模型**：def ranking\_score(user\_pair):
return w1 \* emotion\_sim + w2 \* interest\_sim + w3 \* location\_sim + w4 \* social\_compatibility
# w1-w4为动态学习的权重系数

• **排序优化**：

• 应用Learning to Rank（LTR）模型，综合考虑多个特征

• 引入多样性优化，避免推荐结果过于单一

• 考虑历史互动反馈，动态调整特征权重

**4. 推荐结果生成与展示：**

• **推荐用户示例**：

• 用户小明（U2023120502）：

• 情绪相似度：0.85（同为兴奋状态）

• 兴趣相似度：0.78（共同兴趣：摄影、网红打卡）

• 地理相关性：0.95（当前距离：120米）

• 综合评分：0.86

• **互动建议生成**：

• **场景化建议**：“在断桥残雪取景最佳位置一起拍摄雪景”

• **活动推荐**：“参与#西湖冬日随手拍#摄影活动”

• **话题建议**：基于共同兴趣生成破冰话题

• **推荐展示优化**：

• **个性化展示**：根据用户偏好选择展示风格（简约/详细）

• **实时更新**：每3分钟自动刷新推荐列表

• **渐进式展示**：先展示核心信息，用户感兴趣后再展示更多详情

**5. 互动效果反馈与优化：**

• **直接反馈**：记录用户对推荐的接受/拒绝/忽略行为

• **间接反馈**：追踪后续互动质量（聊天持续时间、互动深度等）

• **A/B测试**：持续测试不同的匹配策略和权重配置

• **模型更新**：基于反馈数据，定期更新模型参数，优化推荐效果

通过上述流程，本发明能够基于用户的实时情绪状态、兴趣偏好和地理位置，智能推荐合适的社交对象和互动方式，提升用户的旅游社交体验。系统通过持续学习和优化，不断提高推荐的准确性和用户满意度。

### **实施例4：共享旅游记忆构建**

本实施例详细描述了本发明中记忆构建引擎（详见技术方案4）在构建共享旅游记忆方面的具体工作流程和技术实现细节，如图4（附图说明中图4：记忆构建与检索机制图）所示。假设用户小李（U001）、小王（U002）和小张（U003）三人组成一个临时旅行团，共同参与了一次为期一天的“古镇文化探索”活动。

**1. 共享旅游事件定义与数据采集授权：**

• **事件创建**：用户小李在App内创建一个名为“古镇一日游”的共享旅游事件，并邀请小王和小张加入。事件设定时间为2023年12月6日，地点为乌镇。

• **数据授权**：小李、小王、小张分别在App内授权系统在事件持续期间（2023年12月6日08:00 - 20:00）收集其与该事件相关的多模态数据，用于构建共享记忆。授权范围可精细化控制，如仅限特定类型数据（照片、位置）或特定地理围栏内的数据。

• **数据源**：

• **照片/视频**：用户通过App拍摄或从手机相册导入的与乌镇相关的照片和短视频，包含EXIF元数据（时间、地点、设备信息）。

• **文字记录**：用户在App内发布的关于乌镇的游记片段、心情随笔、评论。

• **位置轨迹**：App通过GPS记录的用户在乌镇内的移动轨迹点序列（经纬度、时间戳）。

• **情绪数据**：系统通过多模态情绪识别引擎（实施例2）分析用户在游览过程中的情绪状态序列。

• **互动数据**：三人在App内围绕该事件的聊天记录、共同点赞/评论的内容。

**2. 多源异构数据聚合与预处理：**

• **数据汇聚**：系统将所有授权用户在事件时间范围和地理围栏（乌镇景区）内产生的相关数据，自动汇聚到该共享旅游事件的数据池中。

• **时间戳对齐与校准**：对所有数据进行严格的时间戳对齐，对于部分设备时间不准的情况，可尝试通过网络时间协议（NTP）或与其他可信时间源对比进行校准。

• **地理位置编码与POI关联**：将GPS坐标转换为具体的地理位置名称（如“乌镇西栅景区入口”、“茅盾故居”），并关联到POI数据库中的兴趣点信息。

• **内容去重与筛选**：去除重复上传的照片、相似度极高的文本内容，过滤掉与主题明显无关的数据（如用户误操作上传的非乌镇照片）。

• **隐私保护**：对于共享记忆中可能涉及的非同行人员（路人）的人脸进行模糊化处理，或提示用户手动处理。

**3. 共享记忆核心要素提取与构建：**

• **时空主线构建**：

• **共同轨迹融合**：分析三人位置轨迹的重叠部分，识别共同游览的路线和在各景点的停留时长，构建一条共同的游览时空主线。

• **关键节点识别**：在时空主线上，根据停留时长、照片拍摄密度、情绪高点（如集体表现出“惊喜”、“愉悦”情绪的时刻）、以及POI的重要性，自动识别出关键的共同经历节点（如“一起在廊桥合影”、“共同品尝特色小吃”、“在某展览馆长时间驻足”）。

• **多视角内容组织**：

• **场景识别与聚类**：利用计算机视觉技术（如图像场景分类、图像聚类算法DBSCAN、K-Means）对所有照片/视频进行场景识别（如“古桥”、“河道”、“老街”、“特色店铺”），并将同一场景下不同用户拍摄的内容聚合在一起。

• **人物识别与关联**：通过人脸识别技术，识别照片/视频中的小李、小王、小张，以及他们共同遇到的人物（需用户预先标记或授权识别）。

• **事件/故事片段提取**：结合文本内容中的描述、照片/视频内容以及情绪变化，尝试自动或半自动地提取有意义的事件片段或小故事（如“小王在石桥上喂鱼，大家都很开心”）。

• **情感脉络梳理**：

• **共享情绪分析**：分析在同一时间、同一地点，三人情绪状态的相似性与差异性，识别共同的情绪高潮点和情绪转折点。

• **情感标签生成**：为共享记忆的各个片段或整体打上情感标签（如“欢乐的午后”、“宁静的黄昏”、“探索的乐趣”）。

**4. 共享记忆生成与个性化呈现：**

• **自动化故事线生成**：

• 系统根据提取的关键节点、多视角内容和情感脉络，自动生成一个初步的共享旅游记忆故事线。故事线可以是一个图文并茂的时间轴，或一个短视频剪辑的初稿。

• **模板选择**：提供多种记忆呈现模板（如“Vlog风格”、“游记风格”、“相册风格”），用户可以选择喜欢的模板。

• **协同编辑与丰富**：

• **邀请编辑**：小李、小王、小张均可对自动生成的共享记忆进行协同编辑，如调整顺序、增删内容、修改文字描述、添加背景音乐、选择滤镜等。

• **补充视角**：鼓励用户补充各自独特的观察和感受，丰富共享记忆的细节。

• **多版本与隐私控制**：

• 系统支持保存共享记忆的多个编辑版本。

• 用户可以设置共享记忆的查看权限（如仅团队成员可见、可分享给特定好友、或公开）。

• **情感化渲染与沉浸式呈现**：

• 根据记忆片段的情感基调，自动匹配合适的背景音乐、转场特效。

• 对于包含360度照片或视频的内容，支持在VR设备中进行沉浸式回顾。

**5. 记忆的存储、检索与再激活：**

• **结构化存储**：共享记忆以结构化的方式存储，包括元数据（时间、地点、参与者、主题）、媒体内容、文本描述、情感标签、用户评论等。

• **智能检索**：用户可以通过关键词、时间、地点、人物、情绪标签等多维度检索共享记忆。

• **智能推荐与回顾**：系统可以在特定纪念日（如“古镇一日游”一周年）或当用户再次访问相似地点时，主动推送相关的共享记忆，唤醒美好回忆。

通过上述流程，本发明能够高效、智能地将多个用户在共同旅游活动中产生的零散数据，聚合成一段生动、完整、富有情感的共享旅游记忆，并支持用户协同编辑和个性化呈现，极大地丰富了旅游社交的深度和乐趣。

### **实施例5：社交图谱演化**

本实施例详细描述了本发明中社交图谱构建与演化引擎（详见技术方案5）在动态构建、更新和应用社交图谱方面的具体工作流程和技术实现细节，如图5（附图说明中图5：社交图谱构建与演化流程图）所示。假设平台已有存量用户小明（U101）和小红（U102），新用户小强（U103）刚刚注册加入。

**1. 新用户节点创建与初始化：**

• **节点创建**：当小强（U103）成功注册账户后，系统立即为其在社交图谱中创建一个新的用户节点，节点ID为U103。

• **基础属性填充**：

• **静态属性**：从注册信息中提取，如昵称“小强”、年龄段（可选）、性别（可选）、注册时间。

• **动态属性初始化**：

• **兴趣标签**：引导小强选择或输入初始兴趣标签，如“旅游”、“摄影”、“美食”、“历史文化”。这些标签将作为其兴趣画像的初始种子。

• **情绪基线**：通过简短的引导式问卷或结合其在平台上的初步浏览行为（如果允许），初步设定一个情绪基线（如“乐观”、“平和”），此基线会随着后续行为数据动态调整。

• **活跃度**：初始活跃度设为较低值，随其后续在平台上的操作（登录、发帖、互动等）动态更新。

• **影响力**：初始影响力设为较低值，随其发布内容的受欢迎程度（点赞、评论、分享）和关注者数量动态更新。

**2. 互动行为捕捉与关系边建立/更新：**

• **场景一：小强关注小明，小明回关小强**

• **行为捕捉**：系统监测到小强（U103）对小明（U101）执行了“关注”操作，随后小明（U101）对小强（U103）执行了“关注”操作。

• **关系建立**：

• 创建一条从U103指向U101的“关注”类型的有向边，初始权重根据平台策略设定（如0.5）。

• 创建一条从U101指向U103的“关注”类型的有向边，初始权重设为0.5。

• **关系强化**：由于是双向关注，系统可以定义一种“好友”或“强连接”的无向关系边，其权重可以设置为较高值（如0.8），或者通过规则引擎将两条单向关注边的权重提升。

• **属性更新**：U103的“关注数”+1，U101的“粉丝数”+1；U101的“关注数”+1，U103的“粉丝数”+1。

• **场景二：小强点赞小红发布的关于“古镇摄影技巧”的动态**

• **行为捕捉**：系统监测到小强（U103）对小红（U102）发布的ID为P001的动态执行了“点赞”操作。

• **关系建立/强化**：

• 如果U103与U102之间尚无直接关系边，则创建一条从U103指向U102的“互动”类型的有向边，互动子类型为“点赞”，初始权重较低（如0.1）。

• 如果已有“互动”边，则根据点赞行为增加该边的权重（如增加0.05，权重上限为1）。

• **属性更新**：

• U103的兴趣画像中，“摄影”标签的权重可能因此行为而略微提升。

• U102发布的动态P001的“点赞数”+1，其创作者U102的“影响力”可能因此略微提升。

• U103的“活跃度”提升。

• **场景三：小强与小明共同参与了“周末登山”活动，并共享了活动记忆**

• **行为捕捉**：系统监测到小强（U103）和小明（U101）均报名并实际参与了平台组织的“周末登山”活动（A001），并在活动结束后共同确认了一份共享活动记忆（M001）。

• **关系强化**：

• 大幅提升U103与U101之间已存在的“好友”或“强连接”关系的权重（如增加0.2）。

• 如果之前仅为单向关注或弱连接，则可升级为强连接或显著增加现有连接权重。

• 可以创建或强化一条U103与U101之间基于“共同活动A001”的特定上下文关系边。

• **属性更新**：

• U103和小明U101的兴趣画像中，“户外运动”、“登山”等标签权重提升。

• U103和小明U101的“活跃度”提升。

• U103和小明U101的个人情绪状态可能因为愉快的共同活动而变得更加积极，影响其短期情绪画像。

**3. 图谱动态更新与维护：**

• **实时/准实时更新**：上述所有用户行为（注册、关注、点赞、评论、分享、参与活动、发布内容、情绪变化等）都会触发图谱数据库的实时或准实时更新，包括节点属性的修改和关系边的创建、删除或权重调整。

• **权重衰减机制**：对于某些类型的关系（如基于单次点赞的弱互动），如果长时间没有新的互动发生，其权重会随时间推移而适当衰减，以反映关系的时效性。

• **社群发现与演化**：定期运行社群检测算法（如Louvain、Girvan-Newman）分析图谱结构，识别兴趣社群、意见领袖，并追踪社群的形成、发展和解散过程。

**4. 基于演化图谱的智能应用：**

• **个性化推荐**：

• **好友推荐**：基于“共同好友数量”、“共同兴趣标签”、“相似地理位置”、“共同参与活动”等图谱特征，为小强推荐可能认识的人，如小红（因为小强点赞过她的摄影动态，且小红与小明也是好友）。

• **内容推荐**：根据小强的兴趣画像、其好友（如小明）的兴趣偏好以及当前情绪状态，向其推荐相关的动态、文章、活动或旅游产品。

• **群组推荐**：根据小强的兴趣标签和已加入的群组，推荐相似的兴趣群组或活动群组。

• **用户影响力评估**：通过分析用户在图谱中的中心度（如度中心性、介数中心性、特征向量中心性）、粉丝数量、内容传播范围等，评估用户的影响力，用于KOL识别和精准营销。

• **舆情监控与趋势分析**：通过分析图谱中特定话题相关的节点和边的连接模式、情绪传播路径，进行舆情监控和热点趋势预测。

• **风险控制**：识别异常社交行为模式（如虚假账户、刷单行为、恶意传播），提前预警并采取相应措施。

通过上述流程，本发明能够动态构建和演化一个富含用户属性、情绪特征和多元关系的社交图谱，并基于此图谱为用户提供精准的个性化服务，提升用户体验和平台价值。

## **权利要求书**

1. 一种基于情绪识别与记忆构建的旅游社交互动系统，其特征在于，包括：

• 多模态数据采集模块，用于采集用户的文本、图像、音频、行为和生理信号数据；

• 多模态情绪识别引擎，包括面部表情分析单元、语音情绪分析单元、文本情感分析单元、生理信号情绪分析单元和多模态融合策略；

• 社交互动推荐引擎，包括情绪匹配推荐单元、兴趣相似度计算单元、地理位置匹配单元、活动推荐单元和推荐策略优化；

• 记忆构建引擎，包括个人记忆构建单元、共享记忆构建单元、记忆检索单元和记忆推荐单元；

• 社交图谱构建与演化引擎，包括社交关系建模单元、社交影响力分析单元、社群发现单元和图谱演化预测单元。

2. 根据权利要求1所述的系统，其特征在于，所述多模态情绪识别引擎采用基于注意力机制的模态权重动态调整，实现时序一致性约束的多模态特征对齐。

3. 根据权利要求1所述的系统，其特征在于，所述社交互动推荐引擎基于情绪相似度、兴趣相似度和地理位置相关性进行综合匹配，并采用多臂老虎机算法优化推荐策略。

4. 根据权利要求1所述的系统，其特征在于，所述记忆构建引擎通过时空对齐和内容融合技术，构建包含多视角内容的共享记忆体系。

5. 根据权利要求1所述的系统，其特征在于，所述社交图谱构建与演化引擎基于互动频率和情绪同步性量化社交关系强度，并采用图神经网络预测关系演化。

6. 一种基于情绪识别与记忆构建的旅游社交互动方法，其特征在于，包括以下步骤：

• 数据采集步骤：实时采集用户的多模态数据；

• 情绪识别步骤：通过多模态情绪识别引擎分析用户情绪状态；

• 互动推荐步骤：基于情绪状态和用户特征生成社交互动推荐；

• 记忆构建步骤：构建个人和共享旅游记忆；

• 图谱更新步骤：动态更新社交图谱结构。

7. 根据权利要求6所述的方法，其特征在于，所述情绪识别步骤采用深度学习模型进行单模态分析，并通过注意力机制实现多模态融合。

8. 根据权利要求6所述的方法，其特征在于，所述互动推荐步骤综合考虑情绪匹配度、兴趣相似度和地理位置因素，生成个性化推荐结果。

9. 根据权利要求6所述的方法，其特征在于，所述记忆构建步骤通过时空对齐技术整合多用户的旅游数据，生成具有情感价值的共享记忆。

10. 根据权利要求6所述的方法，其特征在于，所述图谱更新步骤基于用户互动行为和情绪变化，动态调整社交关系权重并预测演化趋势。

## **摘要**

本发明提供一种基于情绪识别与记忆构建的旅游社交互动系统及方法。该系统包括多模态数据采集模块、多模态情绪识别引擎、社交互动推荐引擎、记忆构建引擎和社交图谱构建与演化引擎。本发明首创情绪驱动的旅游社交匹配技术，创新性地开发了共享记忆构建机制，突破性地实现了动态社交图谱演化，并高度重视多模态情绪融合算法的创新。通过情绪识别与记忆构建技术，系统显著提高了旅游社交互动的精准度与情感满足度，实现了基于情绪状态的个性化社交匹配，构建了具有情感价值的共享记忆体系，并通过动态社交图谱演化预测用户社交趋势，为旅游者提供深度的社交互动体验。

## **申请人**

[香港美思未来科技有限公司]

## **发明人**

[陈永璇，杨英，谢嘉文，洪文成，聂建豪]

## **申请日期**

[申请日期]