基于大语言模型的智能陪伴交互系统的设计与实现

课程设计报告

课程名称: 软件前沿技术

项目名称: 基于大语言模型的智能陪伴交互系统

英文名称: LLM-based Intelligent Companion Interaction System

作者

- 张梓轩 2226124805
- 刘彦汝 2226124936

目录

---

1. 项目概述

1.1 研究背景

随着人工智能技术的快速发展，大语言模型（Large Language Model, LLM）在自然语言处理领域展现出强大的能力。ChatGPT、Claude等模型的出现使得构建具有自然对话能力的智能助手成为可能。然而，现有的通用对话系统往往缺乏个性化和持续性，无法为用户提供深度的情感陪伴和生活记录功能。

本项目旨在设计并实现一个基于大语言模型的智能陪伴交互系统，通过多智能体协作、长期记忆管理、主动关怀机制等技术手段，为用户提供个性化、持续性的AI陪伴体验。

1.2 研究目的与意义

本项目的主要目的包括:

1. **探索LLM应用开发范式**: 研究如何将大语言模型与现代软件工程实践相结合，构建可维护、可扩展的AI应用系统。
2. **实现个性化交互体验**: 通过记忆系统和用户画像，实现AI助手对用户的深度理解和个性化响应。
3. **研究主动式AI交互**: 打破传统的被动问答模式，实现AI主动关心用户的能力。
4. **实践全栈开发技术**: 涵盖后端服务、数据库设计、前端开发、云部署等完整技术栈。

1.3 项目范围

本系统实现以下核心功能:

功能模块	描述
多智能体对话	支持多个具有不同人设的AI角色进行私聊和群聊
自动记忆系统	AI自动从对话中提取并存储重要信息
主动关怀服务	基于规则和上下文的主动消息推送
快捷状态记录	用户一键记录生活状态(起床、用餐等)
日记生成	自动生成每日对话总结日记
移动端推送	支持iOS/Android PWA推送通知
Token统计	API调用量和费用追踪

---

2. 需求分析

2.1 功能性需求

2.1.1 对话交互需求

- **基本对话**: 用户可与AI进行自然语言对话，AI能够理解上下文并给出连贯回复。
- **多智能体支持**: 系统支持创建多个具有不同性格和专业领域的AI角色。
- **会话管理**: 支持创建新会话、进入历史会话、导出聊天记录等操作。
- **命令系统**: 支持通过斜杠命令(如/wake、/status)执行快捷操作。

2.1.2 记忆系统需求

- **自动记忆提取**: AI能够自动识别对话中值得记忆的信息并存储。
- **记忆分类**: 支持语义记忆、情景记忆、情感记忆、预测记忆四种类型。
- **向量检索**: 基于语义相似度检索相关记忆，融入对话上下文。
- **记忆遗忘**: 支持记忆的重要性衰减和清理机制。

2.1.3 主动服务需求

- **定时问候**: 在特定时间段(如早晨、夜间)主动发送问候消息。
- **状态关注**: 检测用户长时间未起床、学习过久等情况并主动关心。
- **推送通知**: 主动消息可推送到用户移动设备。

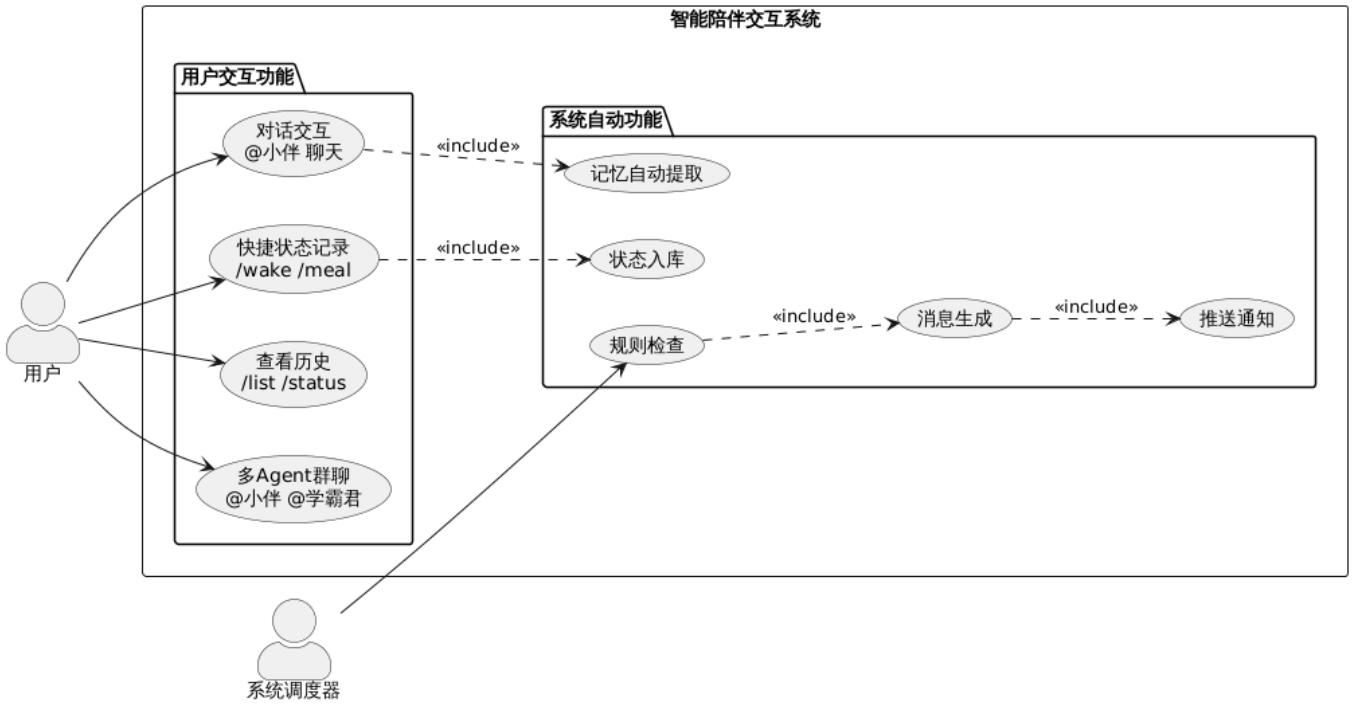
2.1.4 数据持久化需求

- **云数据库存储**: 聊天记录、用户状态、记忆等数据存储在云端。
- **数据导入导出**: 支持聊天记录的JSON格式导入导出。
- **日志记录**: 记录API调用日志用于调试和费用统计。

2.2 非功能性需求

需求类型	描述
可用性	系统通过Web部署，支持7x24小时访问
响应性	普通对话响应时间不超过5秒
可扩展性	采用模块化设计，便于添加新的Agent和功能
安全性	API密钥通过环境变量管理，不暴露在代码中
兼容性	前端适配iPhone 16 Pro等主流移动设备

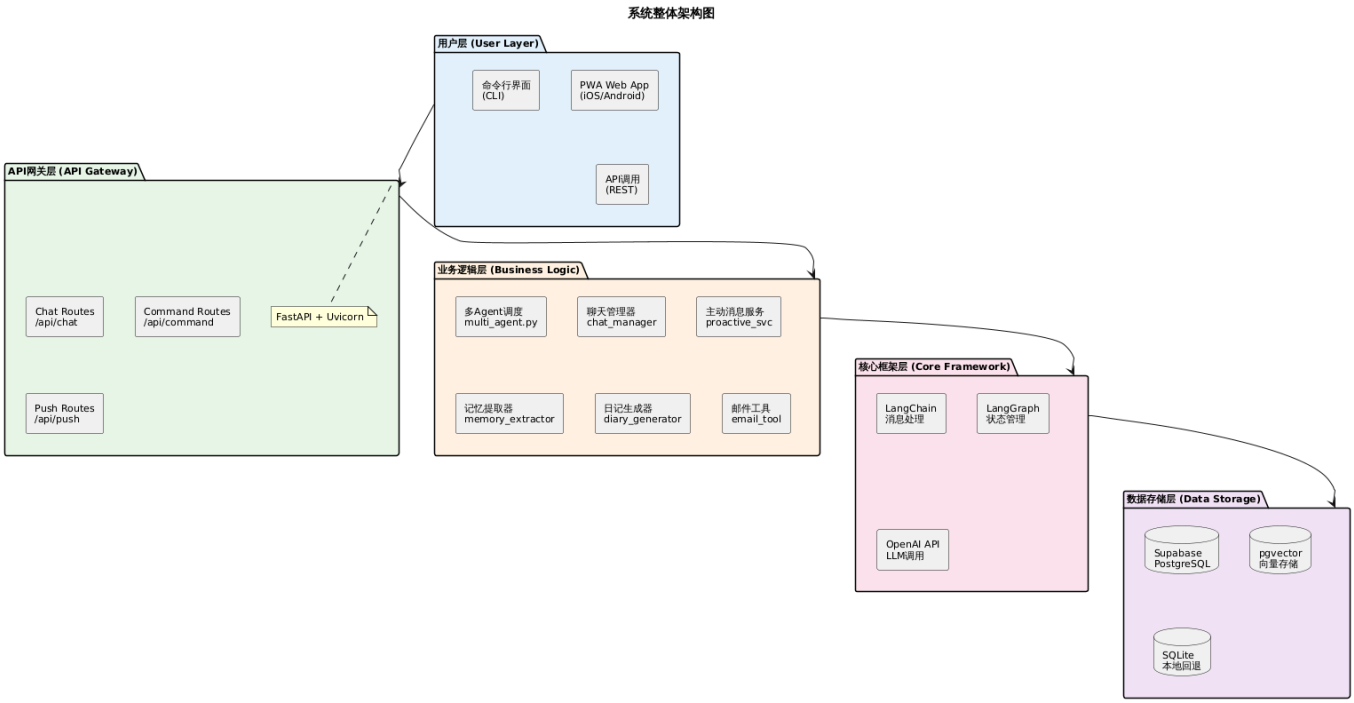
2.3 用例图



3. 系统架构设计

3.1 整体架构

本系统采用前后端分离的分层架构设计，整体架构如下:



3.2 技术选型

层级	技术	选型理由
前端	PWA (原生JavaScript)	可安装到移动设备主屏幕，支持离线和推送
后端框架	FastAPI	高性能异步框架，自动生成API文档
LLM框架	LangChain + LangGraph	成熟的LLM应用框架，支持复杂状态管理
数据库	Supabase (PostgreSQL)	免费云数据库，内置pgvector向量扩展
向量存储	pgvector	与PostgreSQL集成，支持高效向量检索
部署平台	Render	免费托管，自动HTTPS，支持Docker
包管理	uv	现代Python包管理器，速度快

3.3 目录结构

```
companion-assistant/
|-- main_v2.py              # CLI入口
|-- server.py               # Web服务器入口
|-- src/
|   |-- agents/             # Agent相关
|   |   |-- multi_agent.py      # 多Agent调度
|   |   |-- chat_manager.py     # 会话管理
|   |   |-- companion_agent.py  # LangGraph Agent
|   |   |-- diary_generator.py  # 日记生成
|   |   |-- token_callback.py   # Token统计
|   |-- api/                # API路由
|   |   |-- routes/
|   |       |-- chat.py         # 聊天接口
|   |       |-- command.py      # 命令接口
|   |       |-- push.py         # 推送接口
|   |-- database/           # 数据库层
|   |   |-- db_client.py        # 统一数据库抽象
|   |-- memory/             # 记忆系统
|   |   |-- supabase_memory.py  # 向量记忆存储
|   |   |-- memory_extractor.py # 自动记忆提取
|   |   |-- chat_store.py       # 聊天存储
|   |   |-- status_store.py     # 状态存储
|   |-- models/             # 数据模型
|   |   |-- agent_persona.py    # Agent人设
|   |   |-- chat_session.py     # 会话模型
|   |   |-- proactive_rule.py   # 主动消息规则
|   |   |-- status.py           # 状态类型
|   |-- scheduler/          # 调度服务
|   |   |-- proactive_service.py # 主动消息
|   |   |-- push_service.py     # Web Push
```

```
|   |-- tools/              # AI工具
|   |   |-- email_tool.py       # 邮件发送
|   |-- utils/              # 工具函数
|       |-- llm_factory.py      # LLM工厂
|-- frontend/               # PWA前端
|   |-- index.html
|   |-- style.css
|   |-- app.js
|   |-- sw.js                   # Service Worker
|   |-- manifest.json
|-- docs/                   # 项目文档
|-- tests/                  # 测试用例
|-- Dockerfile              # 容器配置
|-- render.yaml             # Render部署配置
```

---

4. 核心功能模块

4.1 多智能体对话系统

4.1.1 Agent人设模型

系统定义了Agent人设数据模型，每个Agent具有独立的属性配置:

```python
class AgentPersona(BaseModel):
    id: str                     # 唯一标识
    name: str                   # 显示名称
    emoji: str                  # 头像符号
    personality: str            # 人设描述(System Prompt)
    trigger_keywords: list      # 触发关键词
    trigger_probability: float # 随机触发概率
    is_default: bool            # 是否为默认Agent
    model: str                  # 使用的LLM模型
    api_base_url: str           # API地址(可选)
    api_key_env: str            # API密钥环境变量名(可选)
```

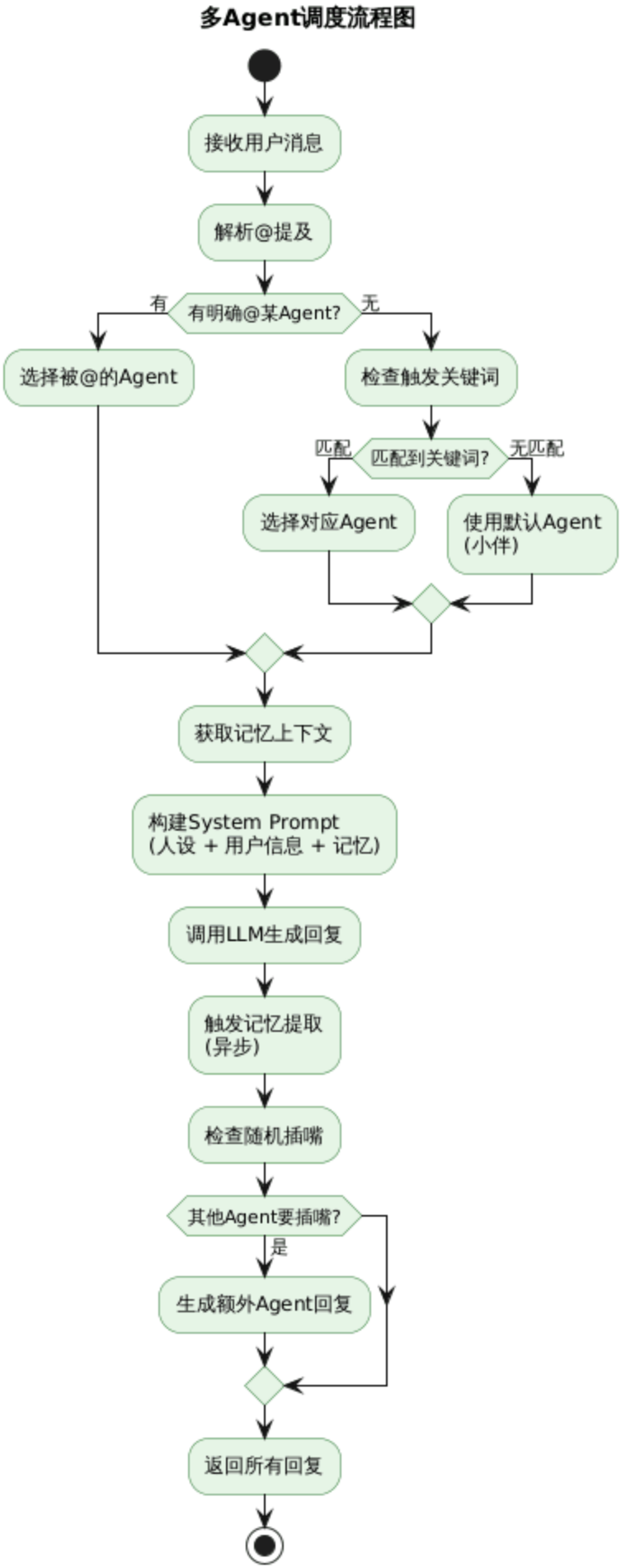
系统预置了四个Agent角色:

Agent	名称	定位	使用模型
xiaoban	小伴	默认陪伴助手，温暖体贴	gpt-4o-mini
xueba	学霸君	学习助手，专业认真	gpt-4o-mini
tiyu	运动达人	健康顾问，阳光积极	OpenRouter

Agent	名称	定位	使用模型
philosopher	哲学家	深度思考者，睿智深邃	Claude

4.1.2 Agent选择与调度

当用户发送消息时，系统按以下逻辑选择回复的Agent:


多Agent调度流程图
接收用户消息
解析@提及
有明确@某Agent?
有
无
选择被@的Agent
检查触发关键词
匹配到关键词?
匹配
无匹配
选择对应Agent
使用默认Agent
(小伴)
获取记忆上下文
构建System Prompt
(人设 + 用户信息 + 记忆)
调用LLM生成回复
触发记忆提取
(异步)
检查随机插嘴
其他Agent要插嘴?
是
生成额外Agent回复
返回所有回复

核心调度函数位于 `multi_agent.py`:

```
def multi_agent_chat(message: str) -> list[dict]:
    """多Agent聊天入口，返回所有Agent的回复"""
    # 选择主Agent
    primary_agent, cleaned_message = select_agent(message)

    # 生成主Agent回复
    responses = [generate_response(primary_agent, cleaned_message)]

    # 检查是否有Agent想要"插嘴"
    joining_agent = check_random_join(primary_agent, message)
    if joining_agent:
        responses.append(generate_response(joining_agent, cleaned_message))

    return responses
```

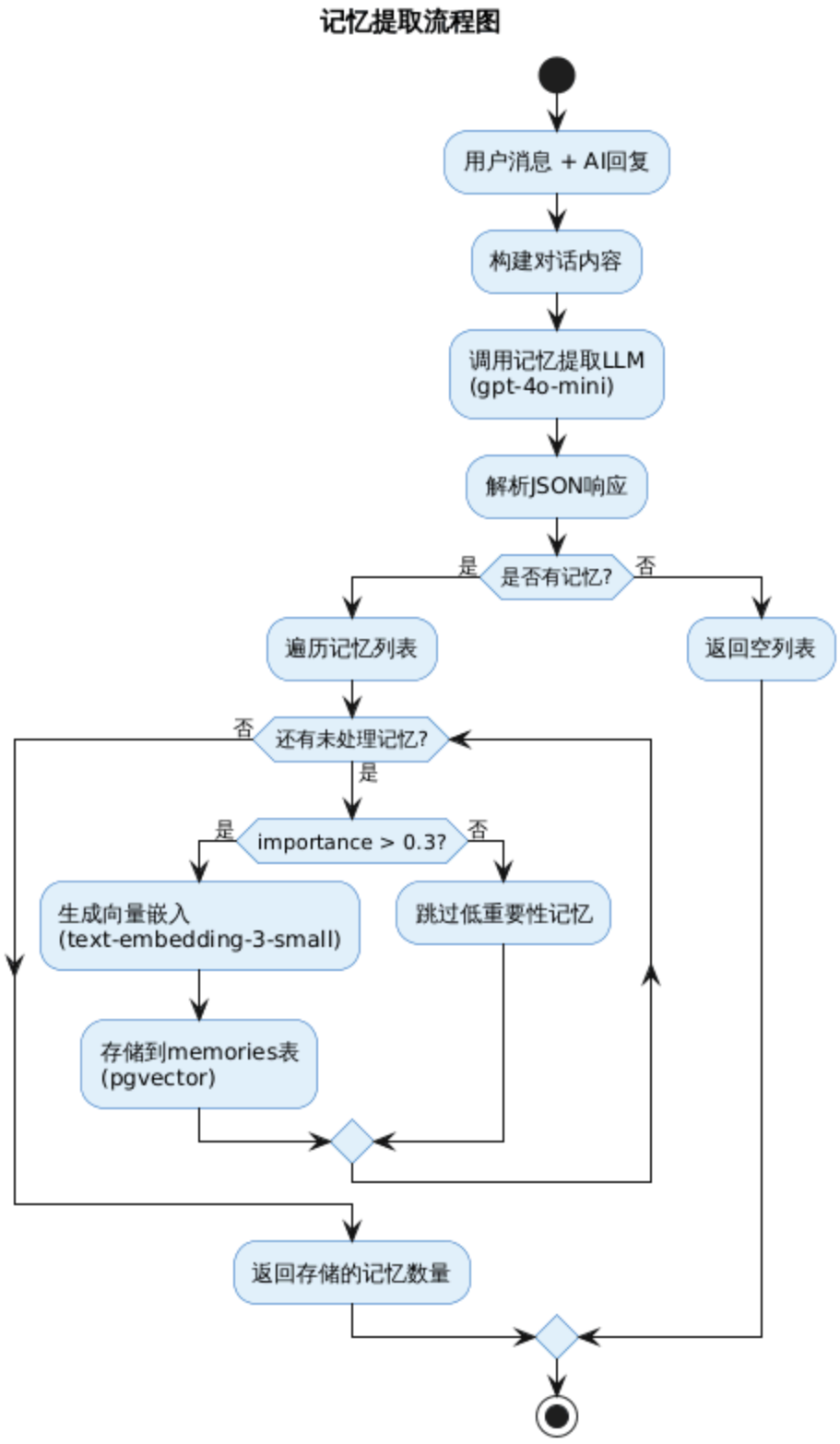
4.2 自动记忆系统

4.2.1 记忆类型定义

系统将记忆分为四种类型:

类型	英文	描述	示例
语义记忆	semantic	用户的个人信息、偏好、习惯	"用户是计算机专业研究生"
情景记忆	episodic	具体事件、经历	"用户今天考试考了90分"
情感记忆	emotional	情绪变化、心理状态	"用户提到考试时会紧张"
预测记忆	predictive	未来计划、重要日期	"用户下周三有算法考试"

4.2.2 记忆提取流程



记忆提取使用专门的System Prompt指导LLM识别有价值的信息:

```
MEMORY_EXTRACTOR_PROMPT = """
你是一个记忆提取助手。你的任务是从对话中识别值得长期记忆的信息。

## 重要性评分（0.0-1.0）
- 0.8-1.0: 极重要（生日、重大事件、核心偏好）
```

```
- 0.5-0.7: 重要（计划、情绪变化、一般偏好）
- 0.3-0.5: 一般（日常事件、小细节）
- 0.0-0.3: 不重要（不值得记忆）

## 注意事项
- 只提取明确、具体的信息，不要过度推断
- 日常寒暄（你好、再见）不需要记忆
- 用户的情感状态很重要，要注意捕捉
"""
```

4.2.3 记忆检索与注入

在生成回复前，系统会检索相关记忆并注入到上下文中:

```python
def get_context_for_chat(self, message: str) -> str:
    # 1. 向量相似度搜索
    similar = self.search_memories(message, limit=3, threshold=0.6)

    # 2. 获取最近记忆
    recent = self.get_recent_memories(days=3, limit=2)

    # 3. 合并去重并格式化
    context_parts = []
    for mem in all_memories:
        context_parts.append(f"[{mem['type']}] {mem['content']}")

    return "## 相关记忆\n" + "\n".join(context_parts)
```

4.3 主动消息服务

4.3.1 规则引擎设计

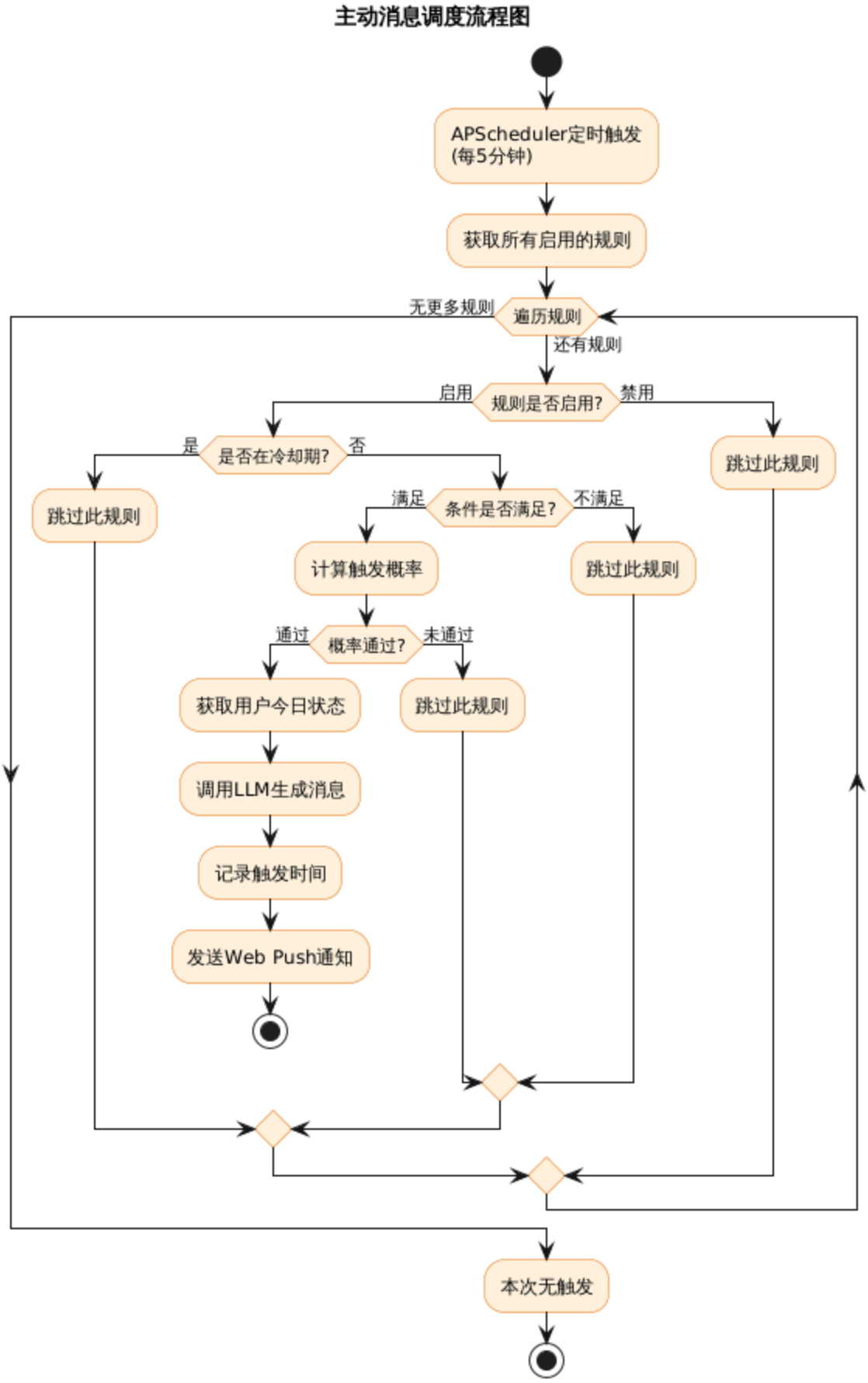
主动消息基于规则引擎触发，规则定义如下:

```python
class ProactiveRule(BaseModel):
    id: str                 # 规则ID
    name: str               # 规则名称
    rule_type: RuleType     # 规则类型
    enabled: bool           # 是否启用
    probability: float      # 触发概率
    cooldown_minutes: int   # 冷却时间(分钟)
    params: dict            # 规则参数
    prompt_hint: str        # 消息生成提示
```

支持的规则类型:

规则类型	描述	触发条件
TIME_NO_WAKE	未起床提醒	超过指定时间未记录起床状态
STATUS_STUDY_LONG	学习时长提醒	连续学习超过指定时长
STATUS_MOOD_BAD	情绪关怀	检测到负面情绪关键词

4.3.2 调度流程


主动消息调度流程图
APScheduler定时触发
(每5分钟)
获取所有启用的规则
遍历规则
无更多规则
还有规则
规则是否启用?
启用
禁用
跳过此规则
是否在冷却期?
是
否
跳过此规则
条件是否满足?
满足
不满足
跳过此规则
计算触发概率
概率通过?
通过
未通过
跳过此规则
获取用户今日状态
调用LLM生成消息
记录触发时间
发送Web Push通知
本次无触发

4.4 快捷状态记录

系统支持通过命令快速记录用户状态:

命令	功能	示例
/wake	记录起床	/wake
/meal [类型]	记录用餐	/meal breakfast
/study start	开始学习	/study start
/study end	结束学习	/study end
/mood [心情]	记录心情	/mood 有点累
/status	查看今日状态	/status

状态数据同时被记忆系统和主动消息系统使用，形成完整的用户画像。

---

5. 关键技术实现

5.1 LLM工厂模式

为支持多种LLM提供商，系统实现了工厂模式:

```
def create_llm(agent: AgentPersona) -> BaseChatModel:
    """根据Agent配置创建对应的LLM实例"""

    # 获取API密钥
    api_key = os.getenv(agent.api_key_env) if agent.api_key_env else
os.getenv("OPENAI_API_KEY")

    # 判断模型类型
    if "claude" in agent.model.lower():
        return ChatAnthropic(
            model=agent.model,
            api_key=api_key
        )
    else:
        # OpenAI兼容接口 (包括OpenRouter)
        return ChatOpenAI(
            model=agent.model,
            api_key=api_key,
            base_url=agent.api_base_url
        )
```

这种设计使得系统可以灵活支持:

- OpenAI GPT系列模型
- Anthropic Claude系列模型
- OpenRouter代理的各种模型
- 任何OpenAI API兼容的服务

5.2 数据库抽象层

为同时支持云端Supabase和本地SQLite，系统实现了数据库抽象层:

```
class DBClient(ABC):
    """数据库客户端抽象基类"""

    @abstractmethod
    def insert(self, table: str, data: dict) -> dict: pass

    @abstractmethod
    def select(self, table: str, filters: dict = None) -> list: pass

    @abstractmethod
    def update(self, table: str, data: dict, filters: dict) -> dict: pass

class SupabaseClient(DBClient):
    """Supabase实现"""
    pass

class SQLiteClient(DBClient):
    """SQLite实现"""
    pass

def get_db_client() -> DBClient:
    """自动选择数据库后端"""
    if os.getenv("SUPABASE_URL"):
        return SupabaseClient()
    return SQLiteClient()
```

5.3 向量相似度检索

记忆检索使用Supabase的pgvector扩展实现高效向量搜索:

```
-- Supabase中的向量搜索函数
CREATE OR REPLACE FUNCTION match_memories(
    query_embedding vector(1536),
```

```
    match_threshold float,
    match_count int,
    memory_type_filter text DEFAULT NULL
)
RETURNS TABLE (
    id bigint,
    content text,
    memory_type text,
    importance float,
    similarity float
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        m.id,
        m.content,
        m.memory_type,
        m.importance,
        1 - (m.embedding <=> query_embedding) as similarity
    FROM memories m
    WHERE 1 - (m.embedding <=> query_embedding) > match_threshold
        AND (memory_type_filter IS NULL OR m.memory_type = memory_type_filter)
    ORDER BY similarity DESC
    LIMIT match_count;
END;
$$;
```

5.4 Web Push推送实现

系统使用VAPID协议实现Web Push:

```
def send_push_notification(title: str, body: str) -> int:
    """发送推送通知"""
    subscriptions = load_subscriptions()

    payload = json.dumps({
        "title": title,
        "body": body,
        "icon": "/icon-192.png"
    })

    for sub in subscriptions:
        webpush(
```

```
            subscription_info={
                "endpoint": sub.endpoint,
                "keys": sub.keys
            },
            data=payload,
            vapid_private_key=VAPID_PRIVATE_KEY,
            vapid_claims={"sub": "mailto:your@email.com"}
        )
```

前端Service Worker处理推送事件:

```
self.addEventListener('push', function(event) {
    const data = event.data.json();
    event.waitUntil(
        self.registration.showNotification(data.title, {
            body: data.body,
            icon: data.icon
        })
    );
});
```

5.5 Token统计与回调

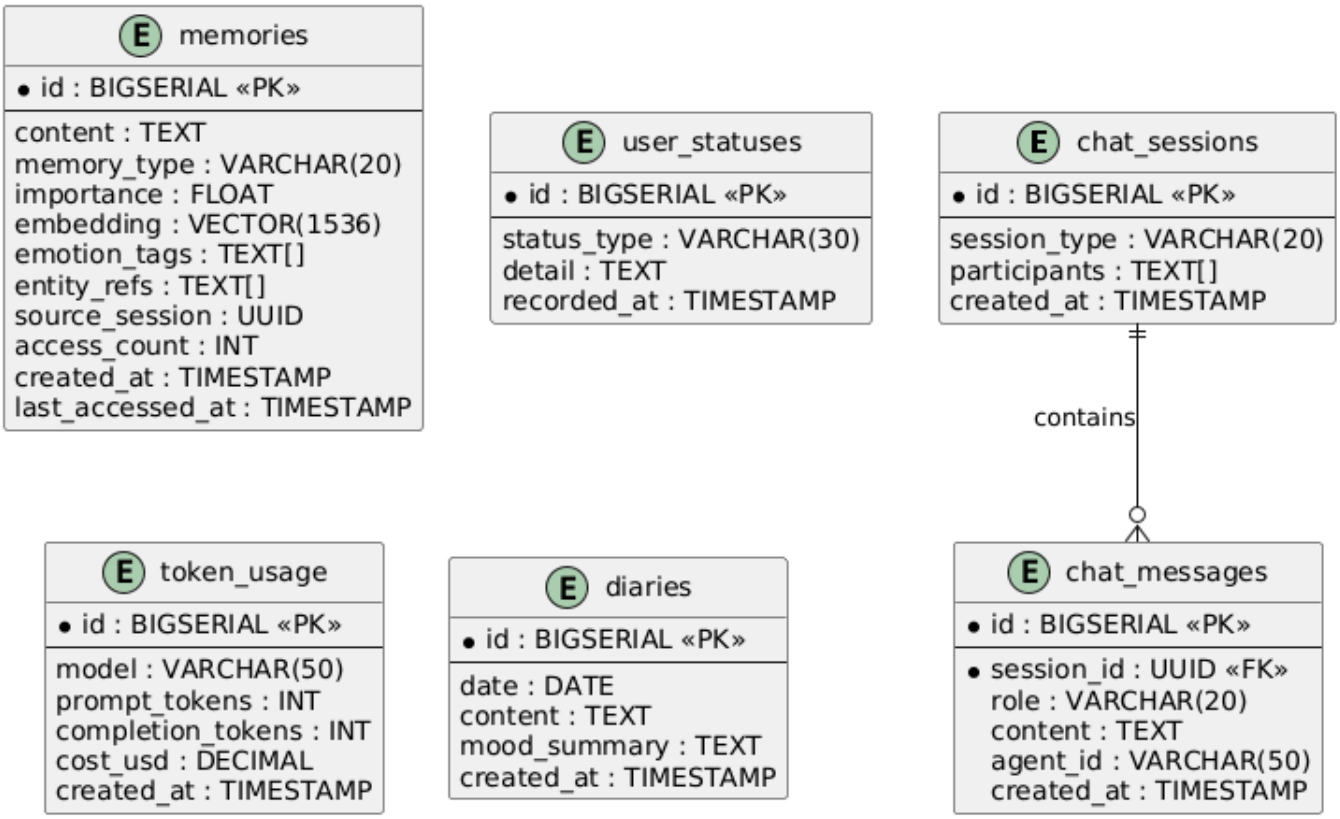
通过LangChain的回调机制统计API调用的Token用量:

```
class TokenTrackingCallback(BaseCallbackHandler):
    """Token用量追踪回调"""

    def on_llm_end(self, response: LLMResult, **kwargs):
        usage = response.llm_output.get("token_usage", {})

        # 记录到数据库
        db.insert("token_usage", {
            "prompt_tokens": usage.get("prompt_tokens", 0),
            "completion_tokens": usage.get("completion_tokens", 0),
            "model": self.model_name,
            "cost_usd": self._calculate_cost(usage)
        })
```

---

6. 数据库设计

6.1 ER图



6.2 核心表结构

memories表 (记忆存储)

字段	类型	说明
id	BIGSERIAL	主键
content	TEXT	记忆内容
memory_type	VARCHAR(20)	类型(semantic/episodic/emotional/predictive)
importance	FLOAT	重要性(0-1)
embedding	VECTOR(1536)	向量嵌入
emotion_tags	TEXT[]	情感标签数组
entity_refs	TEXT[]	关联实体
source_session	UUID	来源会话
access_count	INT	访问次数
created_at	TIMESTAMP	创建时间
last_accessed_at	TIMESTAMP	最后访问时间

chat_messages表 (聊天消息)

字段	类型	说明
id	BIGSERIAL	主键
session_id	UUID	会话ID
role	VARCHAR(20)	角色(user/assistant/system)
content	TEXT	消息内容
agent_id	VARCHAR(50)	Agent标识
created_at	TIMESTAMP	发送时间

---

7. 前端界面设计

7.1 设计理念

前端采用极简的终端风格设计，主要考量:

1. **专注对话**: 去除不必要的视觉元素，突出对话内容
2. **快捷交互**: 通过命令和快捷按钮实现高效操作
3. **PWA支持**: 可安装到移动设备主屏幕，支持离线和推送
4. **响应式布局**: 适配各种屏幕尺寸

7.2 界面布局

```
+----------------------------------+
|  AI Companion           online   |  <- 标题栏
+----------------------------------+
|                                  |
|  [小伴 14:30:15]                 |
|  你好呀，今天感觉怎么样？          |
|                                  |
|  [user 14:30:28]                 |
|  有点累，刚下课                   |
|                                  |
|  [小伴 14:30:31]                 |
|  辛苦了～要不要休息一下？          |
|                                  |  <- 消息区域
+----------------------------------+
| [wake] [breakfast] [lunch] [quit]|  <- 快捷按钮
+----------------------------------+
| > @小伴 今天天气真好               |  <- 输入区
```

```
| [->]                                |
+-------------------------------------+
```

7.3 交互设计

输入格式	功能	示例
@Agent名	与指定Agent对话	@小伴 你好
@Agent1 @Agent2	创建群聊	@小伴 @学霸君 一起讨论
/command	执行系统命令	/wake
/help	查看帮助	/help
普通文本	与默认Agent对话	今天好累

---

8. 测试与部署

8.1 测试策略

8.1.1 单元测试

针对核心模块编写单元测试:

```python
# tests/test_memory.py
def test_memory_extraction():
    """测试记忆提取功能"""
    memories = extract_memories(
        user_message="我明天有算法考试，有点紧张",
        ai_response="别紧张，你之前准备得很充分"
    )

    assert len(memories) >= 1
    assert any(m["type"] == "predictive" for m in memories)
```

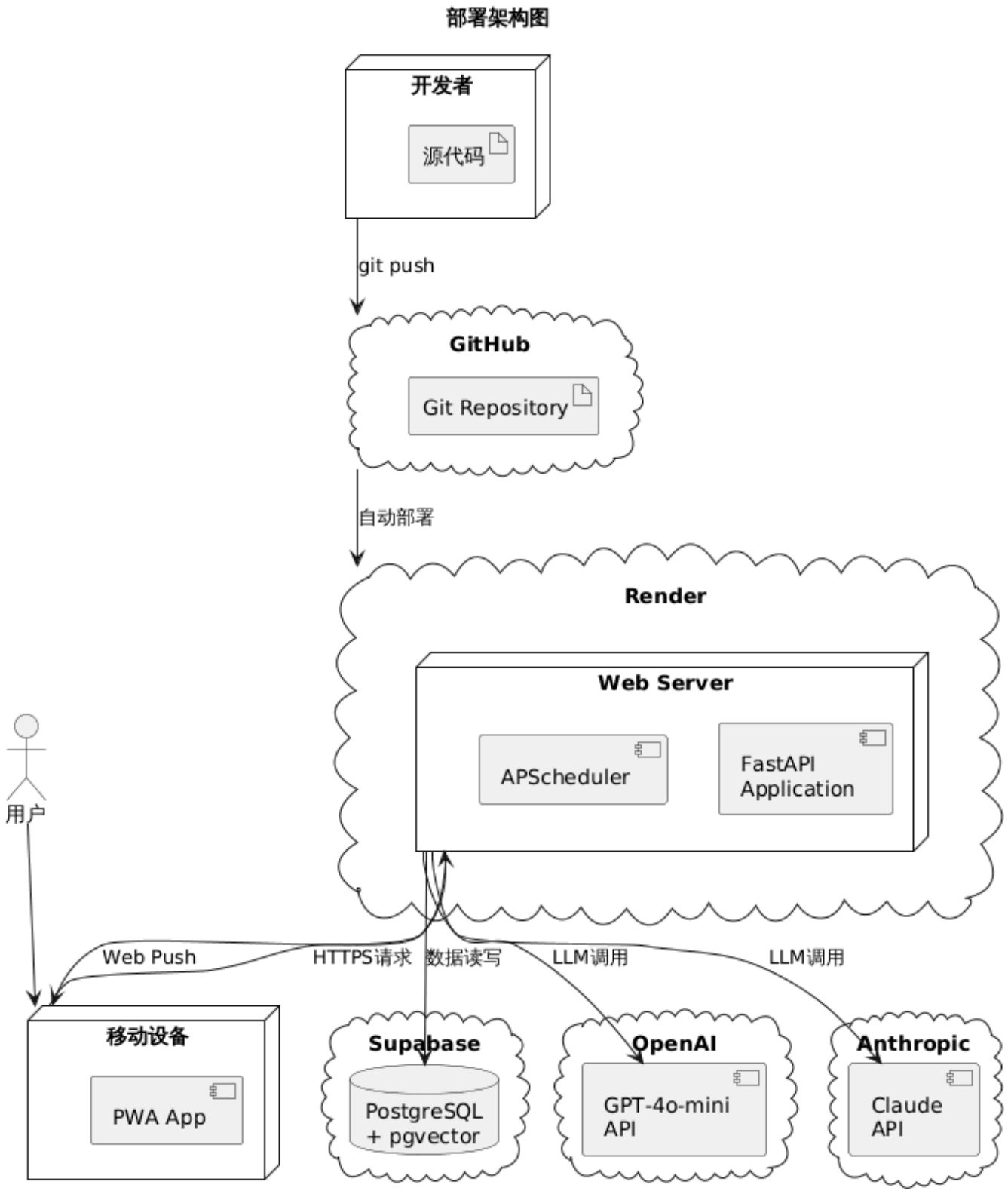
8.1.2 集成测试

测试完整的对话流程:

```python
def test_chat_flow():
    """测试完整对话流程"""
    response = client.post("/api/chat", json={
        "message": "@小伴 你好"
```

```
    })

    assert response.status_code == 200
    assert "responses" in response.json()
```

8.2 部署架构



8.3 部署配置

render.yaml

```yaml
services:
  - type: web
    name: companion-assistant
    runtime: python
    buildCommand: pip install -r requirements.txt
    startCommand: python server.py
    envVars:
      - key: OPENAI_API_KEY
        sync: false
      - key: SUPABASE_URL
        sync: false
      - key: SUPABASE_KEY
        sync: false
```

Dockerfile

```dockerfile
FROM python:3.11-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000
CMD ["python", "server.py"]
```

8.4 环境变量配置

变量名	必需	说明
OPENAI_API_KEY	是	OpenAI API密钥
SUPABASE_URL	是	Supabase项目URL
SUPABASE_KEY	是	Supabase匿名密钥
ANTHROPIC_API_KEY	否	Claude API密钥
EMAIL_SENDER	否	发件邮箱地址
EMAIL_PASSWORD	否	邮箱授权码

变量名	必需	说明
VAPID_PUBLIC_KEY	否	Web Push公钥
VAPID_PRIVATE_KEY	否	Web Push私钥

---

9. 项目亮点与创新点

9.1 技术创新

9.1.1 自动记忆提取机制

传统对话系统需要用户手动标记重要信息。本系统创新性地采用LLM自动分析对话内容，识别并提取值得记忆的信息:

- **智能分类**: 自动区分语义、情景、情感、预测四类记忆
- **重要性评估**: 自动评估信息重要性，过滤无价值内容
- **情感捕捉**: 特别关注用户情绪状态变化

9.1.2 Per-Agent API配置

突破传统多Agent系统的限制，每个Agent可独立配置:

- 不同的LLM模型 (GPT-4、Claude等)
- 不同的API提供商 (OpenAI、OpenRouter等)
- 独立的API密钥

这使得系统可以根据Agent角色选择最适合的模型，同时分散API调用风险。

9.1.3 混合检索策略

记忆检索采用向量相似度 + 时间衰减的混合策略:

```
最终相关性 = 语义相似度 * 0.7 + 时间新近度 * 0.3
```

确保既能找到语义相关的历史记忆，又能优先使用最近的上下文。

9.2 架构优势

9.2.1 数据库抽象层

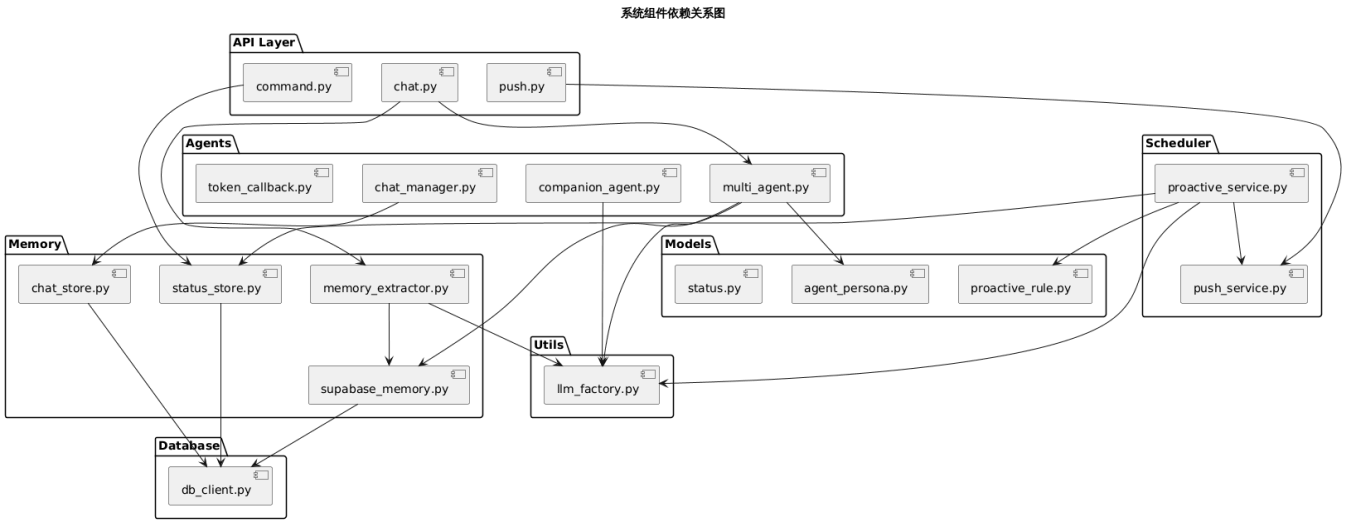
统一的DBClient接口使系统可以无缝切换存储后端:

- **开发环境**: 使用SQLite，无需外部依赖
- **生产环境**: 使用Supabase云数据库

9.2.2 模块化设计

各功能模块高度解耦:

- Agent模块只负责对话生成
- Memory模块只负责记忆管理
- Scheduler模块只负责定时任务



便于独立开发、测试和维护。

9.3 用户体验优化

9.3.1 真实的PWA推送

区别于轮询或WebSocket方案，本系统实现了真正的Web Push:

- 用户将PWA添加到主屏幕后可收到系统级推送
- 支持iOS 16.4+和Android
- 无需保持应用打开

9.3.2 命令行式交互

终端风格的设计带来独特体验:

- 快捷命令高效操作
- 无复杂菜单干扰
- 适合喜欢键盘操作的用户

10. 总结与展望

10.1 项目总结

本项目成功设计并实现了一个基于大语言模型的智能陪伴交互系统。主要成果包括:

1. **完整的多Agent对话系统**: 支持私聊、群聊，每个Agent具有独立人设和配置。
2. **创新的自动记忆机制**: LLM自动提取对话中的重要信息，构建用户画像。
3. **主动式AI交互**: 突破被动问答模式，AI可基于规则主动关心用户。
4. **跨平台移动支持**: PWA技术使系统可安装到移动设备，支持推送通知。
5. **现代化技术栈**: 采用FastAPI、LangChain、Supabase等主流技术，代码结构清晰。

10.2 遇到的挑战与解决方案

挑战	解决方案
多模型API兼容	实现LLM工厂模式，统一接口抽象
记忆检索准确性	结合向量相似度和时间衰减的混合策略
iOS推送限制	采用Web Push标准，指导用户添加到主屏幕
开发/生产环境切换	数据库抽象层自动检测和切换

10.3 未来展望

1. **多模态支持**: 增加图像理解、语音交互能力
2. **更智能的主动消息**: 基于用户行为模式学习，而非固定规则
3. **隐私保护增强**: 端侧记忆加密，支持本地部署选项
4. **社交功能**: 支持多用户，好友间共享Agent
5. **更丰富的工具集成**: 日历、待办、笔记等MCP工具

---

11. 参考文献

[1] OpenAI. ChatGPT: Optimizing Language Models for Dialogue. 2022.

[2] LangChain. LangChain: Building applications with LLMs through composability. https://langchain.com

[3] LangGraph. LangGraph: Build stateful, multi-actor applications with LLMs. https://github.com/langchain-ai/langgraph

[4] Supabase. Supabase: The Open Source Firebase Alternative. https://supabase.com

[5] pgvector. pgvector: Open-source vector similarity search for Postgres. https://github.com/pgvector/pgvector

[6] Web Push Protocol. RFC 8030: Generic Event Delivery Using HTTP Push. IETF.

[7] Progressive Web Apps. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps

---

附录

附录A: 演示视频录制

第一部分: 系统概览 (1-2分钟)

1. 展示项目GitHub仓库和README
2. 介绍技术栈和架构图
3. 展示部署后的PWA应用

第二部分: 核心功能演示 (5-7分钟)

场景1: 基本对话

- 打开PWA，发送 `@小伴 你好`
- 展示AI回复
- 发送一条包含个人信息的消息，如"我明天有算法考试"

场景2: 多Agent群聊

- 发送 `@小伴 @学霸君 一起帮我复习算法`
- 展示两个Agent的不同风格回复

场景3: 快捷状态记录

- 点击快捷按钮记录起床状态
- 输入 `/meal breakfast` 记录早餐
- 输入 `/status` 查看今日状态

场景4: 记忆系统演示

- 打开Supabase数据库面板
- 展示memories表中自动提取的记忆
- 发一条与之前相关的消息，观察AI如何调用记忆

场景5: 主动消息演示

- 展示主动消息规则配置
- 模拟触发条件(如清理wake记录后等待触发)
- 展示收到的推送通知

第三部分: 技术亮点 (2-3分钟)

1. 展示关键代码文件:
   - `multi_agent.py`: 多Agent调度
   - `memory_extractor.py`: 记忆提取
   - `db_client.py`: 数据库抽象
2. 展示模块化架构
3. 展示Per-Agent配置如何支持不同模型

第四部分: 总结 (1分钟)

- 回顾项目核心价值
- 展望未来发展方向

附录B: 开发环境搭建

```
# 1. 克隆项目
git clone https://github.com/your-repo/companion-assistant.git
cd companion-assistant

# 2. 创建环境变量文件
cp .env.example .env
# 编辑 .env 填入必要的API密钥

# 3. 安装依赖
uv sync

# 4. 启动CLI模式
uv run python main_v2.py

# 5. 启动Web服务
uv run python server.py
# 访问 http://localhost:8000
```

附录C: API接口文档

端点	方法	描述
/api/chat	POST	发送聊天消息
/api/command	POST	执行命令
/api/sessions	GET	获取会话列表
/api/push/subscribe	POST	订阅推送
/api/push/vapid-key	GET	获取VAPID公钥

报告完成日期: 2026年1月

项目版本: v1.2.5