“九如-山海”项目代码文档

本文档包含研究论文写作助手项目的所有源代码。项目由以下模块组成：

* 1. 文献综述与创意生成模块 (literature\_review\_module)
* 2. 数据收集与研究模块 (research\_module)
* 3. 论文写作模块 (writing\_module)
* 4. 论文评估模块 (evaluation\_module)

# 文件：app.py

from flask import Flask, request, jsonify  
from flask\_cors import CORS  
import os  
import logging  
from werkzeug.utils import secure\_filename  
from dashscope import Generation  
from core.coordinator import Coordinator  
from agents.professor\_advisor import ProfessorAdvisorAgent  
from agents.research\_advisor import ResearchAdvisorAgent  
import asyncio  
from functools import wraps  
  
# 配置日志  
logging.basicConfig(level=logging.DEBUG)  
logger = logging.getLogger(\_\_name\_\_)  
  
# 设置API密钥  
api\_key = os.environ.get('DASHSCOPE\_API\_KEY')  
if not api\_key:  
 api\_key = 'sk-773c0160d8f04d54a010c7075dcea6c1'  
 logger.warning("DASHSCOPE\_API\_KEY not found in environment variables. Using hardcoded key.")  
os.environ['DASHSCOPE\_API\_KEY'] = api\_key  
  
app = Flask(\_\_name\_\_)  
# 简化CORS配置  
CORS(app)  
  
# 初始化协调器和智能体  
try:  
 coordinator = Coordinator()  
 professor\_advisor = ProfessorAdvisorAgent(name="Professor Advisor", coordinator=coordinator)  
 research\_advisor = ResearchAdvisorAgent(name="Research Advisor", coordinator=coordinator)  
except Exception as e:  
 logger.error(f"Error initializing coordinator and agents: {e}")  
 coordinator = None  
 professor\_advisor = None  
 research\_advisor = None  
  
# 配置文件上传  
UPLOAD\_FOLDER = 'uploads'  
ALLOWED\_EXTENSIONS = {'doc', 'docx', 'pdf', 'mp4', 'mp3'}  
  
if not os.path.exists(UPLOAD\_FOLDER):  
 os.makedirs(UPLOAD\_FOLDER)  
  
app.config['UPLOAD\_FOLDER'] = UPLOAD\_FOLDER  
app.config['MAX\_CONTENT\_LENGTH'] = 16 \* 1024 \* 1024 # 16MB max file size  
  
def allowed\_file(filename):  
 return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED\_EXTENSIONS  
  
async def get\_agent\_responses(message: str, conversation\_id: str):  
 """获取所有智能体的响应"""  
 try:  
 if not coordinator or not professor\_advisor or not research\_advisor:  
 return "系统初始化错误", "系统初始化错误"  
  
 professor\_advisor.set\_conversation(conversation\_id)  
 research\_advisor.set\_conversation(conversation\_id)  
   
 # 并行获取响应  
 professor\_response = await professor\_advisor.process\_message(message)  
 research\_response = await research\_advisor.process\_message(message)  
   
 return professor\_response, research\_response  
 except Exception as e:  
 logger.error(f"Error getting agent responses: {e}")  
 return str(e), str(e)  
  
def async\_route(f):  
 @wraps(f)  
 def wrapped(\*args, \*\*kwargs):  
 return asyncio.run(f(\*args, \*\*kwargs))  
 return wrapped  
  
@app.route('/api/chat', methods=['POST', 'OPTIONS'])  
@async\_route  
async def chat():  
 logger.debug(f"Received request: {request.method}")  
 logger.debug(f"Request headers: {request.headers}")  
   
 if request.method == 'OPTIONS':  
 logger.debug("Handling OPTIONS request")  
 return '', 200  
   
 try:  
 # 获取消息内容  
 message = request.form.get('message', '')  
 mode = request.form.get('mode')  
 conversation\_id = request.form.get('conversationId')  
   
 if not conversation\_id and coordinator:  
 conversation\_id = coordinator.create\_conversation("New Chat")  
 elif not conversation\_id:  
 conversation\_id = "temp-" + str(hash(message))  
   
 # 将用户消息存入历史记录  
 if coordinator:  
 user\_message = {"role": "user", "content": message}  
 coordinator.store\_conversation(conversation\_id, user\_message)  
   
 logger.debug(f"Received message: {message}")  
 logger.debug(f"Mode: {mode}")  
 logger.debug(f"Conversation ID: {conversation\_id}")  
  
 # 处理文件上传  
 uploaded\_files = []  
 if 'files' in request.files:  
 files = request.files.getlist('files')  
 for file in files:  
 if file and allowed\_file(file.filename):  
 filename = secure\_filename(file.filename)  
 file\_path = os.path.join(app.config['UPLOAD\_FOLDER'], filename)  
 file.save(file\_path)  
 uploaded\_files.append({  
 'name': filename,  
 'path': file\_path,  
 'type': filename.rsplit('.', 1)[1].lower()  
 })  
 logger.debug(f"Saved file: {filename}")  
  
 # 获取智能体响应  
 professor\_feedback, research\_feedback = await get\_agent\_responses(message, conversation\_id)  
  
 # 将智能体响应存入历史记录  
 if coordinator:  
 professor\_message = {"role": "assistant", "name": "Professor Advisor", "content": professor\_feedback}  
 coordinator.store\_conversation(conversation\_id, professor\_message)  
 research\_message = {"role": "assistant", "name": "Research Advisor", "content": research\_feedback}  
 coordinator.store\_conversation(conversation\_id, research\_message)  
  
 # 返回响应  
 response = {  
 'conversationId': conversation\_id,  
 'analysis': f'收到消息: {message}',  
 'professorFeedback': professor\_feedback,  
 'researchFeedback': research\_feedback,  
 'fileAnalysis': {  
 'files': [{'filename': f['name'], 'analysis': '测试文件分析'} for f in uploaded\_files]  
 }  
 }  
 logger.debug(f"Sending response: {response}")  
 return jsonify(response)  
  
 except Exception as e:  
 logger.error(f"Error in chat endpoint: {str(e)}", exc\_info=True)  
 return jsonify({  
 'error': str(e)  
 }), 500  
  
@app.route('/api/test', methods=['GET'])  
def test():  
 logger.debug("Received test request")  
 return jsonify({'status': 'ok', 'message': 'Backend is running'})  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 logger.info("Starting server on http://localhost:5000")  
 app.run(debug=True, port=5000, host='0.0.0.0')

================================================================================

# 文件：generate\_doc.py

from docx import Document  
from docx.shared import Pt, RGBColor  
from docx.enum.text import WD\_ALIGN\_PARAGRAPH  
import os  
  
def create\_code\_document():  
 # 创建新的Word文档  
 doc = Document()  
   
 # 设置标题  
 title = doc.add\_heading('研究论文写作助手项目代码文档', 0)  
 title.alignment = WD\_ALIGN\_PARAGRAPH.CENTER  
   
 # 添加项目说明  
 doc.add\_paragraph('本文档包含研究论文写作助手项目的所有源代码。项目由以下模块组成：')  
 modules = [  
 '1. 文献综述模块 (literature\_review\_module)',  
 '2. 研究模块 (research\_module)',  
 '3. 写作模块 (writing\_module)',  
 '4. 评估模块 (evaluation\_module)'  
 ]  
 for module in modules:  
 doc.add\_paragraph(module, style='List Bullet')  
   
 # 遍历项目目录  
 for root, dirs, files in os.walk('.'):  
 # 跳过venv和.git目录  
 if 'venv' in root or '.git' in root:  
 continue  
   
 # 处理Python文件  
 for file in files:  
 if file.endswith('.py'):  
 file\_path = os.path.join(root, file)  
 relative\_path = os.path.relpath(file\_path, '.')  
   
 # 添加文件标题  
 doc.add\_heading(f'文件：{relative\_path}', level=1)  
   
 # 读取文件内容  
 with open(file\_path, 'r', encoding='utf-8') as f:  
 content = f.read()  
   
 # 添加代码内容  
 p = doc.add\_paragraph()  
 run = p.add\_run(content)  
 run.font.name = 'Courier New'  
 run.font.size = Pt(10)  
   
 # 添加分隔线  
 doc.add\_paragraph('=' \* 80)  
   
 # 保存文档  
 doc.save('project\_code\_documentation.docx')  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 create\_code\_document()

================================================================================

# 文件：get-pip.py

================================================================================

# 文件：main.py

import asyncio  
from typing import Dict, List, Any  
from agents.phd\_agent import PhDAgent  
from agents.dr\_agent import DrAgent  
from agents.writer\_agent import WriterAgent  
from agents.evaluator\_agent import EvaluatorAgent  
from modules.thesis\_evaluation import ThesisEvaluationModule  
from modules.research\_analyzer import ResearchAnalyzer  
from modules.literature\_review import LiteratureReviewModule  
from modules.research\_executor import ResearchExecutor  
from core.coordinator import Coordinator  
from modules.file\_processor import FileProcessor  
import os  
import logging  
  
# 配置日志  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
class ThesisWritingSystem:  
 def \_\_init\_\_(self):  
 # 初始化协调器  
 self.coordinator = Coordinator()  
   
 # 初始化智能体  
 self.phd\_agent = PhDAgent(coordinator=self.coordinator)  
 self.dr\_agent1 = DrAgent(name="Dr Agent 1", coordinator=self.coordinator)  
 self.dr\_agent2 = DrAgent(name="Dr Agent 2", coordinator=self.coordinator)  
 self.writer\_agent = WriterAgent(coordinator=self.coordinator)  
 self.evaluator\_agent = EvaluatorAgent(coordinator=self.coordinator)  
   
 # 初始化模块  
 self.evaluation\_module = ThesisEvaluationModule()  
 self.research\_analyzer = ResearchAnalyzer()  
 self.literature\_review = LiteratureReviewModule()  
 self.research\_executor = ResearchExecutor()  
   
 # 系统状态  
 self.current\_stage = "initialization"  
 self.research\_topic = None  
 self.conversation\_id = None  
  
 async def process\_research\_topic(self, topic: str):  
 """处理研究主题"""  
 try:  
 # 创建新的对话  
 self.conversation\_id = self.coordinator.create\_conversation(topic)  
 self.research\_topic = topic  
   
 # 设置所有智能体的对话ID  
 self.phd\_agent.set\_conversation(self.conversation\_id)  
 self.dr\_agent1.set\_conversation(self.conversation\_id)  
 self.dr\_agent2.set\_conversation(self.conversation\_id)  
 self.writer\_agent.set\_conversation(self.conversation\_id)  
 self.evaluator\_agent.set\_conversation(self.conversation\_id)  
   
 # 并行获取Dr Agents的分析  
 dr1\_analysis = await self.dr\_agent1.analyze\_research\_status(topic)  
 dr2\_analysis = await self.dr\_agent2.analyze\_research\_status(topic)  
   
 # PhD Agent基于分析更新研究框架  
 framework\_update = {  
 "research\_question": f"基于{dr1\_analysis['research\_gaps']}和{dr2\_analysis['research\_gaps']}的研究空白",  
 "theoretical\_framework": "待完善",  
 "methodology": "待确定",  
 "expected\_contributions": "待明确"  
 }  
   
 for section, content in framework\_update.items():  
 self.phd\_agent.update\_framework(section, content)  
   
 return {  
 "status": "success",  
 "framework": self.phd\_agent.get\_framework\_status(),  
 "dr1\_analysis": dr1\_analysis,  
 "dr2\_analysis": dr2\_analysis  
 }  
   
 except Exception as e:  
 logger.error(f"Error processing research topic: {e}")  
 return {  
 "status": "error",  
 "message": str(e)  
 }  
  
 async def process\_message(self, message: str) -> Dict[str, Any]:  
 """处理用户消息"""  
 try:  
 # 获取所有智能体的响应  
 responses = await asyncio.gather(  
 self.phd\_agent.process\_message(message),  
 self.dr\_agent1.process\_message(message),  
 self.dr\_agent2.process\_message(message),  
 self.writer\_agent.process\_message(message),  
 self.evaluator\_agent.process\_message(message)  
 )  
   
 return {  
 "status": "success",  
 "responses": {  
 "phd\_agent": responses[0],  
 "dr\_agent1": responses[1],  
 "dr\_agent2": responses[2],  
 "writer\_agent": responses[3],  
 "evaluator\_agent": responses[4]  
 }  
 }  
   
 except Exception as e:  
 logger.error(f"Error processing message: {e}")  
 return {  
 "status": "error",  
 "message": str(e)  
 }  
  
 def get\_system\_status(self) -> Dict[str, Any]:  
 """获取系统状态"""  
 return {  
 "current\_stage": self.current\_stage,  
 "research\_topic": self.research\_topic,  
 "conversation\_id": self.conversation\_id,  
 "phd\_agent": self.phd\_agent.get\_framework\_status(),  
 "writer\_agent": self.writer\_agent.get\_paper\_status(),  
 "evaluator\_agent": self.evaluator\_agent.evaluation\_criteria  
 }  
  
async def main():  
 # 创建系统实例  
 system = ThesisWritingSystem()  
   
 # 选择输入模式  
 print("请选择研究主题输入模式：")  
 print("1. 输入具体研究想法")  
 print("2. 输入参考文献")  
 choice = input("请输入选项（1或2）：")  
   
 if choice == "1":  
 research\_idea = input("\n请输入您的研究想法：")  
 result = await system.process\_research\_topic(research\_idea)  
 else:  
 print("\n请输入参考文献（每行一篇，输入空行结束）：")  
 references = []  
 while True:  
 ref = input()  
 if not ref:  
 break  
 references.append(ref)  
 result = await system.process\_research\_topic(references[0])  
   
 if result["status"] == "success":  
 print("\n=== 研究框架开发完成 ===")  
 print("\n研究框架：")  
 print(result["framework"])  
   
 print("\nDr Agent 1 分析：")  
 print(result["dr1\_analysis"])  
   
 print("\nDr Agent 2 分析：")  
 print(result["dr2\_analysis"])  
   
 # 进入交互式对话模式  
 print("\n=== 进入交互式对话模式 ===")  
 print("您可以与系统进行对话，输入'quit'退出")  
   
 while True:  
 message = input("\n请输入您的问题或指令：")  
 if message.lower() == 'quit':  
 break  
   
 response = await system.process\_message(message)  
 if response["status"] == "success":  
 print("\nPhD Agent 响应：")  
 print(response["responses"]["phd\_agent"])  
   
 print("\nDr Agent 1 响应：")  
 print(response["responses"]["dr\_agent1"])  
   
 print("\nDr Agent 2 响应：")  
 print(response["responses"]["dr\_agent2"])  
   
 print("\nWriter Agent 响应：")  
 print(response["responses"]["writer\_agent"])  
   
 print("\nEvaluator Agent 响应：")  
 print(response["responses"]["evaluator\_agent"])  
 else:  
 print(f"错误：{response['message']}")  
   
 else:  
 print(f"错误：{result['message']}")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main())

================================================================================

# 文件：start.py

import subprocess  
import webbrowser  
import time  
import os  
from http.server import HTTPServer, SimpleHTTPRequestHandler  
import threading  
  
def start\_backend():  
 """启动后端Flask服务器"""  
 subprocess.Popen(['python', 'app.py'])  
  
def start\_frontend\_server():  
 """启动前端静态文件服务器"""  
 os.chdir('frontend') # 切换到frontend目录  
 server = HTTPServer(('localhost', 8000), SimpleHTTPRequestHandler)  
 server.serve\_forever()  
  
def main():  
 # 启动后端服务器  
 print("正在启动后端服务器...")  
 start\_backend()  
   
 # 等待后端服务器启动  
 time.sleep(2)  
   
 # 启动前端服务器  
 print("正在启动前端服务器...")  
 frontend\_thread = threading.Thread(target=start\_frontend\_server)  
 frontend\_thread.daemon = True  
 frontend\_thread.start()  
   
 # 等待前端服务器启动  
 time.sleep(1)  
   
 # 在默认浏览器中打开前端页面  
 print("正在打开浏览器...")  
 webbrowser.open('http://localhost:8000')  
   
 print("\n系统已启动！")  
 print("前端地址: http://localhost:8000")  
 print("后端地址: http://localhost:5000")  
 print("\n按Ctrl+C可以停止服务器")  
   
 try:  
 # 保持主线程运行  
 while True:  
 time.sleep(1)  
 except KeyboardInterrupt:  
 print("\n正在关闭服务器...")  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

================================================================================

# 文件：test\_api.py

import requests  
  
def test\_chat():  
 url = 'http://localhost:5000/api/chat'  
 data = {  
 'message': '你好，我想讨论一下我的研究方向'  
 }  
 response = requests.post(url, data=data)  
 print(response.status\_code)  
 print(response.json())  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test\_chat()

================================================================================

# 文件：agents\base.py

import sys  
import traceback  
  
class BaseAgent:  
 def \_\_init\_\_(self):  
 pass  
  
 async def generate\_response(self, context: str, prompt: str) -> str:  
 try:  
 response = await Generation.acall(  
 model='qwen-turbo',  
 prompt=f"{context}\n\n{prompt}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
 # 检查API返回的响应是否成功  
 if response.status\_code == 200:  
 return response.output.text  
 else:  
 # 如果API返回错误，则将其作为异常信息处理  
 error\_message = (f"DashScope API Error: Status {response.status\_code}, "  
 f"Code: {response.code}, Message: {response.message}")  
 raise Exception(error\_message)  
 except Exception as e:  
 # 捕获所有异常，并将详细信息作为字符串返回  
 error\_info = f"Caught Exception: {str(e)}\n{traceback.format\_exc()}"  
 print(error\_info, file=sys.stderr) # 再次尝试打印，以防万一  
 return error\_info  
  
 def set\_conversation(self, conversation\_id: str):  
 """Set the current conversation ID"""  
 pass

================================================================================

# 文件：agents\base\_agent.py

from abc import ABC, abstractmethod  
from typing import Dict, List, Any  
from pydantic import BaseModel  
from dashscope import Generation  
from core.coordinator import Coordinator  
import asyncio  
import logging  
import sys  
import traceback  
  
class AgentState(BaseModel):  
 """Base state model for all agents"""  
 name: str  
 role: str  
 current\_task: str = ""  
 memory: List[Dict[str, Any]] = []  
 status: str = "idle"  
  
class BaseAgent(ABC):  
 def \_\_init\_\_(self, name: str, role: str, coordinator: Coordinator):  
 self.state = AgentState(name=name, role=role)  
 self.coordinator = coordinator  
 self.conversation\_id = None  
 self.logger = logging.getLogger(\_\_name\_\_)  
  
 @abstractmethod  
 async def process\_message(self, message: str) -> str:  
 """Process incoming messages and generate responses"""  
 pass  
  
 def update\_state(self, \*\*kwargs):  
 """Update agent state with new information"""  
 for key, value in kwargs.items():  
 if hasattr(self.state, key):  
 setattr(self.state, key, value)  
 self.coordinator.update\_agent\_state(self.state.name, self.state.dict())  
  
 def add\_to\_memory(self, item: Dict[str, Any]):  
 """Add new information to agent's memory"""  
 self.state.memory.append(item)  
 self.coordinator.update\_agent\_state(self.state.name, self.state.dict())  
  
 def get\_memory(self) -> List[Dict[str, Any]]:  
 """Retrieve agent's memory"""  
 return self.state.memory  
  
 def clear\_memory(self):  
 """Clear agent's memory"""  
 self.state.memory = []  
 self.coordinator.update\_agent\_state(self.state.name, self.state.dict())  
  
 async def generate\_response(self, prompt: str, context: str = "") -> str:  
 """Generate response using Qwen model"""  
 try:  
 # 在一个单独的线程中运行同步的SDK调用，以避免阻塞事件循环  
 response = await asyncio.to\_thread(  
 Generation.call,  
 model='qwen-turbo',  
 prompt=f"{context}\n\n{prompt}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
 # 检查API返回的响应是否成功  
 if response.status\_code == 200:  
 return response.output.text  
 else:  
 # 如果API返回错误，则将其作为异常信息处理  
 error\_message = (f"DashScope API Error: Status {response.status\_code}, "  
 f"Code: {response.code}, Message: {response.message}")  
 raise Exception(error\_message)  
 except Exception as e:  
 # 捕获所有异常，并将详细信息作为字符串返回，用于调试  
 error\_info = f"Caught Exception: {str(e)}\n{traceback.format\_exc()}"  
 print(error\_info, file=sys.stderr)  
 return "An error occurred while generating the response. Please check the server logs for details."  
  
 def set\_conversation(self, conversation\_id: str):  
 """Set the current conversation ID"""  
 self.conversation\_id = conversation\_id  
  
 def get\_conversation\_context(self, max\_messages: int = 10) -> str:  
 """Get conversation context"""  
 if not self.conversation\_id:  
 return ""  
 return self.coordinator.get\_context(self.conversation\_id, max\_messages)

================================================================================

# 文件：agents\dr\_agent.py

from typing import Dict, List, Any  
from .base\_agent import BaseAgent  
from dashscope import Generation  
  
class DrAgent(BaseAgent):  
 def \_\_init\_\_(self, name: str = "Dr Agent", coordinator=None):  
 super().\_\_init\_\_(name=name, role="Research Status Analyst", coordinator=coordinator)  
 self.expertise\_areas = [  
 "Literature Review",  
 "Research Gap Analysis",  
 "Methodology Assessment",  
 "Theoretical Contribution"  
 ]  
 self.analysis\_history: List[Dict[str, Any]] = []  
  
 async def process\_message(self, message: str) -> str:  
 """Process incoming messages and generate responses"""  
 try:  
 context = self.get\_conversation\_context()  
   
 system\_prompt = f"""你是一位专注于研究现状分析的博士研究员。  
 你的专长领域包括：{', '.join(self.expertise\_areas)}  
 你的主要职责是：  
 1. 分析当前研究主题的研究现状  
 2. 识别研究空白和机会  
 3. 评估研究方法的适当性  
 4. 提供理论贡献建议  
   
 请基于以下上下文提供专业的分析和建议：  
 {context}  
 """  
   
 response = await self.generate\_response(message, system\_prompt)  
   
 analysis = {  
 "type": "research\_analysis",  
 "content": response,  
 "timestamp": "current\_time"  
 }  
 self.analysis\_history.append(analysis)  
   
 return response  
 except Exception as e:  
 print(f"Error in Dr agent: {e}")  
 return f"Error generating response: {str(e)}"  
  
 def analyze\_research\_status(self, topic: str) -> Dict[str, Any]:  
 """分析研究主题的现状"""  
 analysis = {  
 "current\_status": "",  
 "research\_gaps": [],  
 "opportunities": [],  
 "methodology\_suggestions": [],  
 "theoretical\_contributions": []  
 }  
 self.analysis\_history.append({  
 "type": "status\_analysis",  
 "topic": topic,  
 "content": analysis,  
 "timestamp": "current\_time"  
 })  
 return analysis  
  
 def get\_analysis\_history(self) -> List[Dict[str, Any]]:  
 """获取所有分析历史"""  
 return self.analysis\_history

================================================================================

# 文件：agents\evaluator\_agent.py

from typing import Dict, List, Any  
from .base\_agent import BaseAgent  
from dashscope import Generation  
  
class EvaluatorAgent(BaseAgent):  
 def \_\_init\_\_(self, name: str = "Evaluator Agent", coordinator=None):  
 super().\_\_init\_\_(name=name, role="Paper Quality Evaluator", coordinator=coordinator)  
 self.evaluation\_criteria = {  
 "reliability": {  
 "methodology": 0.0,  
 "data\_quality": 0.0,  
 "analysis\_rigor": 0.0,  
 "reproducibility": 0.0  
 },  
 "innovation": {  
 "theoretical\_contribution": 0.0,  
 "methodological\_innovation": 0.0,  
 "practical\_implications": 0.0,  
 "future\_directions": 0.0  
 },  
 "clarity": {  
 "writing\_quality": 0.0,  
 "structure": 0.0,  
 "argumentation": 0.0,  
 "presentation": 0.0  
 }  
 }  
 self.evaluation\_history: List[Dict[str, Any]] = []  
  
 async def process\_message(self, message: str) -> str:  
 """Process incoming messages and generate responses"""  
 try:  
 context = self.get\_conversation\_context()  
   
 system\_prompt = f"""你是一位专注于学术论文评估的专家。  
 你的主要职责是：  
 1. 使用多维度指标体系评估论文质量  
 2. 确保论文的可靠性和创新性  
 3. 提供具体的改进建议  
 4. 评估论文的学术价值  
   
 当前评估标准：  
 {self.evaluation\_criteria}  
   
 请基于以下上下文提供专业的评估意见：  
 {context}  
 """  
   
 response = await self.generate\_response(message, system\_prompt)  
   
 evaluation = {  
 "type": "paper\_evaluation",  
 "content": response,  
 "timestamp": "current\_time"  
 }  
 self.evaluation\_history.append(evaluation)  
   
 return response  
 except Exception as e:  
 print(f"Error in Evaluator agent: {e}")  
 return f"Error generating response: {str(e)}"  
  
 def evaluate\_paper(self, paper\_content: Dict[str, str]) -> Dict[str, Any]:  
 """评估论文质量"""  
 evaluation = {  
 "overall\_score": 0.0,  
 "detailed\_scores": self.evaluation\_criteria.copy(),  
 "strengths": [],  
 "weaknesses": [],  
 "improvement\_suggestions": []  
 }  
   
 self.evaluation\_history.append({  
 "type": "full\_evaluation",  
 "content": evaluation,  
 "timestamp": "current\_time"  
 })  
   
 return evaluation  
  
 def update\_criteria(self, category: str, criteria: Dict[str, float]):  
 """更新评估标准"""  
 if category in self.evaluation\_criteria:  
 self.evaluation\_criteria[category].update(criteria)  
  
 def get\_evaluation\_history(self) -> List[Dict[str, Any]]:  
 """获取所有评估历史"""  
 return self.evaluation\_history

================================================================================

# 文件：agents\phd\_agent.py

from typing import Dict, List, Any  
from .base\_agent import BaseAgent  
from dashscope import Generation  
  
class PhDAgent(BaseAgent):  
 def \_\_init\_\_(self, name: str = "PhD Agent", coordinator=None):  
 super().\_\_init\_\_(name=name, role="PhD Research Framework Developer", coordinator=coordinator)  
 self.research\_framework = {  
 "research\_question": "",  
 "theoretical\_framework": "",  
 "methodology": "",  
 "expected\_contributions": "",  
 "revision\_history": []  
 }  
 self.feedback\_history: List[Dict[str, Any]] = []  
  
 async def process\_message(self, message: str) -> str:  
 """Process incoming messages and generate responses"""  
 try:  
 context = self.get\_conversation\_context()  
   
 system\_prompt = f"""你是一位专注于研究框架开发的博士研究员。  
 你的主要职责是：  
 1. 基于多轮思维链分析研究主题  
 2. 整合Dr Agents的反馈  
 3. 修正和完善研究框架  
 4. 确保研究框架的逻辑性和创新性  
   
 当前研究框架状态：  
 {self.research\_framework}  
   
 请基于以下上下文提供专业的分析和建议：  
 {context}  
 """  
   
 response = await self.generate\_response(message, system\_prompt)  
   
 feedback = {  
 "type": "framework\_feedback",  
 "content": response,  
 "timestamp": "current\_time"  
 }  
 self.feedback\_history.append(feedback)  
   
 return response  
 except Exception as e:  
 print(f"Error in PhD agent: {e}")  
 return f"Error generating response: {str(e)}"  
  
 def update\_framework(self, section: str, content: str):  
 """更新研究框架的特定部分"""  
 if section in self.research\_framework:  
 self.research\_framework[section] = content  
 self.research\_framework["revision\_history"].append({  
 "section": section,  
 "content": content,  
 "timestamp": "current\_time"  
 })  
  
 def get\_framework\_status(self) -> Dict[str, Any]:  
 """获取当前研究框架状态"""  
 return self.research\_framework  
  
 def get\_feedback\_history(self) -> List[Dict[str, Any]]:  
 """获取所有反馈历史"""  
 return self.feedback\_history

================================================================================

# 文件：agents\writer\_agent.py

from typing import Dict, List, Any  
from .base\_agent import BaseAgent  
from dashscope import Generation  
  
class WriterAgent(BaseAgent):  
 def \_\_init\_\_(self, name: str = "Writer Agent", coordinator=None):  
 super().\_\_init\_\_(name=name, role="Academic Paper Writer", coordinator=coordinator)  
 self.writing\_style = {  
 "academic\_level": "high",  
 "clarity": "high",  
 "coherence": "high",  
 "formality": "high"  
 }  
 self.paper\_sections = {  
 "introduction": "",  
 "literature\_review": "",  
 "methodology": "",  
 "results": "",  
 "discussion": "",  
 "conclusion": ""  
 }  
 self.writing\_history: List[Dict[str, Any]] = []  
  
 async def process\_message(self, message: str) -> str:  
 """Process incoming messages and generate responses"""  
 try:  
 context = self.get\_conversation\_context()  
   
 system\_prompt = f"""你是一位专注于学术论文写作的专家。  
 你的主要职责是：  
 1. 整合研究框架、数据和研究结果  
 2. 采用层次化写作方法  
 3. 确保论文的精准性和清晰度  
 4. 保持学术写作的规范性和专业性  
   
 当前论文状态：  
 {self.paper\_sections}  
   
 写作风格要求：  
 {self.writing\_style}  
   
 请基于以下上下文提供专业的写作建议：  
 {context}  
 """  
   
 response = await self.generate\_response(message, system\_prompt)  
   
 writing\_record = {  
 "type": "writing\_feedback",  
 "content": response,  
 "timestamp": "current\_time"  
 }  
 self.writing\_history.append(writing\_record)  
   
 return response  
 except Exception as e:  
 print(f"Error in Writer agent: {e}")  
 return f"Error generating response: {str(e)}"  
  
 def update\_section(self, section: str, content: str):  
 """更新论文特定部分的内容"""  
 if section in self.paper\_sections:  
 self.paper\_sections[section] = content  
 self.writing\_history.append({  
 "type": "section\_update",  
 "section": section,  
 "content": content,  
 "timestamp": "current\_time"  
 })  
  
 def get\_paper\_status(self) -> Dict[str, Any]:  
 """获取当前论文状态"""  
 return {  
 "sections": self.paper\_sections,  
 "writing\_style": self.writing\_style  
 }  
  
 def get\_writing\_history(self) -> List[Dict[str, Any]]:  
 """获取所有写作历史"""  
 return self.writing\_history

================================================================================

# 文件：agents\\_\_init\_\_.py

from .phd\_agent import PhDAgent  
from .dr\_agent import DrAgent  
from .writer\_agent import WriterAgent  
from .evaluator\_agent import EvaluatorAgent  
  
\_\_all\_\_ = ['PhDAgent', 'DrAgent', 'WriterAgent', 'EvaluatorAgent']

================================================================================

# 文件：core\coordinator.py

from typing import Dict, List, Any  
import json  
import redis  
from datetime import datetime  
  
class Coordinator:  
 def \_\_init\_\_(self):  
 # 初始化Redis连接  
 self.redis\_client = redis.Redis(host='localhost', port=6379, db=0)  
 self.conversation\_key\_prefix = "conversation:"  
 self.max\_history\_length = 1000  
  
 def store\_conversation(self, conversation\_id: str, message: Dict[str, Any]):  
 """存储对话历史"""  
 key = f"{self.conversation\_key\_prefix}{conversation\_id}"  
 message['timestamp'] = datetime.now().isoformat()  
 self.redis\_client.rpush(key, json.dumps(message))  
 # 保持历史记录在限定长度内  
 self.redis\_client.ltrim(key, -self.max\_history\_length, -1)  
  
 def get\_conversation\_history(self, conversation\_id: str) -> List[Dict[str, Any]]:  
 """获取对话历史"""  
 key = f"{self.conversation\_key\_prefix}{conversation\_id}"  
 history = self.redis\_client.lrange(key, 0, -1)  
 return [json.loads(msg) for msg in history]  
  
 def get\_context(self, conversation\_id: str, max\_messages: int = 10) -> str:  
 """获取最近的对话上下文"""  
 history = self.get\_conversation\_history(conversation\_id)  
 recent\_messages = history[-max\_messages:]  
 return "\n".join([f"{msg['role']}: {msg['content']}" for msg in recent\_messages])  
  
 def broadcast\_message(self, sender: str, message: str, conversation\_id: str):  
 """广播消息给所有相关智能体"""  
 message\_data = {  
 "sender": sender,  
 "content": message,  
 "timestamp": datetime.now().isoformat()  
 }  
 self.store\_conversation(conversation\_id, message\_data)  
 return message\_data  
  
 def get\_agent\_state(self, agent\_id: str) -> Dict[str, Any]:  
 """获取智能体状态"""  
 key = f"agent\_state:{agent\_id}"  
 state = self.redis\_client.get(key)  
 return json.loads(state) if state else {}  
  
 def update\_agent\_state(self, agent\_id: str, state: Dict[str, Any]):  
 """更新智能体状态"""  
 key = f"agent\_state:{agent\_id}"  
 self.redis\_client.set(key, json.dumps(state))  
  
 def create\_conversation(self, topic: str) -> str:  
 """创建新的对话"""  
 conversation\_id = f"conv\_{datetime.now().strftime('%Y%m%d\_%H%M%S')}"  
 initial\_message = {  
 "role": "system",  
 "content": f"New conversation started with topic: {topic}",  
 "timestamp": datetime.now().isoformat()  
 }  
 self.store\_conversation(conversation\_id, initial\_message)  
 return conversation\_id

================================================================================

# 文件：modules\coordinator.py

from typing import Dict, Any, List  
from dashscope import Generation  
from .research\_analyzer import ResearchAnalyzer  
from .file\_processor import FileProcessor  
  
# 设置API密钥  
Generation.set\_api\_key('sk-tatviaawpkyenqcmgdpjbmmsmaivxxdgnnjxtdwbnycjnunu')  
  
class Coordinator:  
 def \_\_init\_\_(self):  
 self.research\_analyzer = ResearchAnalyzer()  
 self.file\_processor = FileProcessor()  
 self.model = 'qwen-max'  
 self.temperature = 0.7  
 self.max\_tokens = 2000  
  
 async def process\_request(self, message: str, files: List[Dict[str, str]] = None) -> Dict[str, Any]:  
 """处理用户请求并协调各个模块"""  
 try:  
 # 处理文件  
 file\_results = None  
 if files:  
 file\_results = await self.file\_processor.process\_files(files)  
  
 # 分析研究想法  
 research\_analysis = await self.research\_analyzer.analyze\_research(  
 message,  
 context=file\_results  
 )  
  
 # 生成综合建议  
 final\_response = await self.\_generate\_final\_response(  
 message,  
 research\_analysis,  
 file\_results  
 )  
  
 return {  
 'response': final\_response,  
 'status': 'success',  
 'file\_analysis': file\_results,  
 'research\_analysis': research\_analysis  
 }  
  
 except Exception as e:  
 return {  
 'error': str(e),  
 'status': 'error'  
 }  
  
 async def \_generate\_final\_response(self, message: str, research\_analysis: Dict[str, Any], file\_results: Dict[str, Any] = None) -> str:  
 """生成最终的综合建议"""  
 try:  
 prompt = self.\_build\_final\_prompt(message, research\_analysis, file\_results)  
   
 response = Generation.call(  
 model=self.model,  
 prompt=prompt,  
 temperature=self.temperature,  
 max\_tokens=self.max\_tokens  
 )  
  
 if response.status\_code == 200:  
 return response.output.text  
 else:  
 return f"生成建议时出错: {response.status\_code}"  
  
 except Exception as e:  
 return f"生成建议时出错: {str(e)}"  
  
 def \_build\_final\_prompt(self, message: str, research\_analysis: Dict[str, Any], file\_results: Dict[str, Any] = None) -> str:  
 """构建最终建议的提示词"""  
 prompt = f"""基于以下信息，请提供综合性的研究建议：  
  
研究想法：  
{message}  
  
研究分析：  
{research\_analysis.get('analysis', '无分析结果')}"""  
  
 if file\_results and file\_results.get('files'):  
 prompt += "\n\n文件分析："  
 for file in file\_results['files']:  
 if file.get('analysis'):  
 prompt += f"\n{file['filename']}: {file['analysis']}"  
  
 prompt += "\n\n请提供：\n1. 研究建议总结\n2. 下一步行动建议\n3. 潜在风险提示"  
  
 return prompt

================================================================================

# 文件：modules\file\_processor.py

import os  
from typing import List, Dict, Any  
from dashscope import Generation  
import PyPDF2  
import docx  
import json  
import aiofiles  
import asyncio  
import speech\_recognition as sr  
from pydub import AudioSegment  
import logging  
from dotenv import load\_dotenv  
  
# 加载环境变量  
load\_dotenv()  
  
# 配置日志  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
# 从环境变量获取API密钥  
api\_key = os.getenv('API\_KEY')  
if api\_key:  
 os.environ['DASHSCOPE\_API\_KEY'] = api\_key  
else:  
 logger.warning("API\_KEY not found in environment variables")  
  
class FileProcessor:  
 def \_\_init\_\_(self):  
 self.supported\_types = {  
 'pdf': self.\_process\_pdf,  
 'doc': self.\_process\_doc,  
 'docx': self.\_process\_doc,  
 'mp4': self.\_process\_video,  
 'mp3': self.\_process\_audio  
 }  
  
 async def process\_files(self, files: List[Dict[str, str]]) -> Dict[str, Any]:  
 """处理上传的文件并返回分析结果"""  
 results = []  
 for file in files:  
 file\_type = file['type'].lower()  
 if file\_type in self.supported\_types:  
 try:  
 content = await self.supported\_types[file\_type](file['path'])  
 analysis = await self.\_analyze\_with\_qwen(content, file\_type)  
 results.append({  
 'filename': file['name'],  
 'type': file\_type,  
 'content': content,  
 'analysis': analysis  
 })  
 except Exception as e:  
 logger.error(f"Error processing file {file['name']}: {str(e)}")  
 results.append({  
 'filename': file['name'],  
 'type': file\_type,  
 'error': str(e)  
 })  
 else:  
 logger.warning(f"Unsupported file type: {file\_type}")  
 results.append({  
 'filename': file['name'],  
 'type': file\_type,  
 'error': 'Unsupported file type'  
 })  
 return {'files': results}  
  
 async def \_process\_pdf(self, file\_path: str) -> str:  
 """异步处理PDF文件"""  
 async with aiofiles.open(file\_path, 'rb') as file:  
 content = await file.read()  
 pdf\_reader = PyPDF2.PdfReader(content)  
 text = ""  
 for page in pdf\_reader.pages:  
 text += page.extract\_text() + "\n"  
 return text  
  
 async def \_process\_doc(self, file\_path: str) -> str:  
 """异步处理Word文档"""  
 # 由于python-docx不支持异步，使用线程池执行  
 loop = asyncio.get\_event\_loop()  
 return await loop.run\_in\_executor(None, self.\_process\_doc\_sync, file\_path)  
  
 def \_process\_doc\_sync(self, file\_path: str) -> str:  
 """同步处理Word文档"""  
 doc = docx.Document(file\_path)  
 return "\n".join(paragraph.text for paragraph in doc.paragraphs)  
  
 async def \_process\_video(self, file\_path: str) -> str:  
 """异步处理视频文件"""  
 try:  
 # 使用线程池执行视频处理  
 loop = asyncio.get\_event\_loop()  
 return await loop.run\_in\_executor(None, self.\_process\_video\_sync, file\_path)  
 except Exception as e:  
 logger.error(f"Error processing video file: {str(e)}")  
 raise  
  
 def \_process\_video\_sync(self, file\_path: str) -> str:  
 """同步处理视频文件"""  
 return "Video processing has been disabled"  
  
 async def \_process\_audio(self, file\_path: str) -> str:  
 """异步处理音频文件"""  
 try:  
 # 使用线程池执行音频处理  
 loop = asyncio.get\_event\_loop()  
 return await loop.run\_in\_executor(None, self.\_process\_audio\_sync, file\_path)  
 except Exception as e:  
 logger.error(f"Error processing audio file: {str(e)}")  
 raise  
  
 def \_process\_audio\_sync(self, file\_path: str) -> str:  
 """同步处理音频文件"""  
 # 转换音频为WAV格式  
 audio = AudioSegment.from\_file(file\_path)  
 audio.export("temp\_audio.wav", format="wav")  
   
 # 语音转文本  
 text = self.\_speech\_to\_text("temp\_audio.wav")  
   
 # 清理临时文件  
 os.remove("temp\_audio.wav")  
   
 duration = len(audio) / 1000.0 # 转换为秒  
 return f"""Audio Analysis:  
Duration: {duration} seconds  
Transcription:  
{text}"""  
  
 def \_speech\_to\_text(self, audio\_path: str) -> str:  
 """将语音转换为文本"""  
 recognizer = sr.Recognizer()  
 with sr.AudioFile(audio\_path) as source:  
 audio = recognizer.record(source)  
 try:  
 return recognizer.recognize\_google(audio, language='zh-CN')  
 except sr.UnknownValueError:  
 return "Speech recognition could not understand the audio"  
 except sr.RequestError as e:  
 return f"Could not request results from speech recognition service; {str(e)}"  
  
 async def \_analyze\_with\_qwen(self, content: str, file\_type: str) -> Dict[str, Any]:  
 """使用Qwen模型分析文件内容"""  
 try:  
 # 构建提示词  
 prompt = self.\_build\_prompt(content, file\_type)  
   
 # 调用Qwen模型  
 response = Generation.call(  
 model='qwen-max',  
 prompt=prompt,  
 temperature=0.7,  
 max\_tokens=2000  
 )  
  
 if response.status\_code == 200:  
 return {  
 'summary': response.output.text,  
 'status': 'success'  
 }  
 else:  
 logger.error(f"Qwen API error: {response.status\_code}")  
 return {  
 'error': f"Qwen API error: {response.status\_code}",  
 'status': 'error'  
 }  
  
 except Exception as e:  
 logger.error(f"Error analyzing with Qwen: {str(e)}")  
 return {  
 'error': str(e),  
 'status': 'error'  
 }  
  
 def \_build\_prompt(self, content: str, file\_type: str) -> str:  
 """构建提示词"""  
 if file\_type in ['pdf', 'doc', 'docx']:  
 return f"""请分析以下文档内容，并提供：  
1. 主要内容概述  
2. 关键观点提取  
3. 研究价值评估  
4. 相关研究建议  
  
文档内容：  
{content}"""  
 elif file\_type in ['mp4']:  
 return f"""请分析以下视频内容，并提供：  
1. 视频基本信息分析  
2. 语音内容摘要  
3. 研究价值评估  
4. 使用建议  
  
视频信息：  
{content}"""  
 else: # mp3  
 return f"""请分析以下音频内容，并提供：  
1. 音频基本信息分析  
2. 语音内容摘要  
3. 研究价值评估  
4. 使用建议  
  
音频信息：  
{content}"""

================================================================================

# 文件：modules\knowledge\_acquisition.py

from typing import Dict, List, Any  
from dashscope import Generation  
import json  
  
class KnowledgeAcquisitionModule:  
 def \_\_init\_\_(self):  
 self.paper\_database: List[Dict[str, Any]] = []  
 self.search\_history: List[Dict[str, Any]] = []  
  
 async def search\_papers(self, query: str, max\_results: int = 5) -> List[Dict[str, Any]]:  
 """Search for relevant academic papers"""  
 # In a real implementation, this would connect to academic databases  
 # For now, we'll simulate paper search results  
 system\_prompt = """你是一个研究助手，帮助查找相关的学术论文。  
 请提供与搜索查询相匹配的论文的结构化信息。"""  
  
 prompt = f"{system\_prompt}\n\n请找到{max\_results}篇与以下主题相关的学术论文：{query}"  
  
 try:  
 response = Generation.call(  
 model='qwen-max',  
 prompt=prompt,  
 temperature=0.3,  
 max\_tokens=2000  
 )  
   
 # Parse the response into structured paper data  
 # This is a simplified version - in reality, you'd want more robust parsing  
 try:  
 papers = json.loads(response.output.text)  
 except:  
 # Fallback if the response isn't valid JSON  
 papers = [{"title": "Sample Paper", "authors": ["Author 1"], "abstract": response.output.text}]  
  
 self.search\_history.append({  
 "query": query,  
 "results": papers,  
 "timestamp": "current\_time" # You might want to use actual timestamps  
 })  
  
 return papers  
 except Exception as e:  
 print(f"Error in paper search: {e}")  
 return []  
  
 async def consult\_llm(self, question: str, context: str = "") -> Dict[str, Any]:  
 """Consult with LLM about research-related questions"""  
 system\_prompt = f"""你是一个研究助手，提供专业知识。  
 背景：{context}  
 请提供详细的、学术级别的研究问题回答。"""  
  
 try:  
 response = Generation.call(  
 model='qwen-max',  
 prompt=f"{system\_prompt}\n\n{question}",  
 temperature=0.3,  
 max\_tokens=2000  
 )  
   
 return {  
 "question": question,  
 "answer": response.output.text,  
 "context": context,  
 "timestamp": "current\_time" # You might want to use actual timestamps  
 }  
 except Exception as e:  
 print(f"Error in LLM consultation: {e}")  
 return {  
 "question": question,  
 "answer": f"Error: {str(e)}",  
 "context": context,  
 "timestamp": "current\_time"  
 }  
  
 def add\_paper\_to\_database(self, paper: Dict[str, Any]):  
 """Add a paper to the local database"""  
 self.paper\_database.append(paper)  
  
 def get\_paper\_database(self) -> List[Dict[str, Any]]:  
 """Get all papers in the database"""  
 return self.paper\_database  
  
 def get\_search\_history(self) -> List[Dict[str, Any]]:  
 """Get history of all paper searches"""  
 return self.search\_history

================================================================================

# 文件：modules\literature\_review.py

from typing import Dict, List, Any  
from dashscope import Generation  
import json  
import asyncio  
from datetime import datetime  
  
class LiteratureReviewModule:  
 def \_\_init\_\_(self):  
 self.review\_history: List[Dict[str, Any]] = []  
 self.collected\_papers: List[Dict[str, Any]] = []  
 self.filtered\_papers: List[Dict[str, Any]] = []  
 self.idea\_generation\_results: List[Dict[str, Any]] = []  
  
 async def collect\_resources(self, topic: str) -> List[Dict[str, Any]]:  
 """通过MCP收集研究资源"""  
 system\_prompt = """你是一个专业的学术资源收集助手。  
 请基于给定的研究主题，从以下方面收集相关资源：  
 1. 核心学术论文  
 2. 相关研究综述  
 3. 重要理论文献  
 4. 研究方法文献  
 5. 最新研究进展  
   
 对于每篇文献，请提供：  
 - 标题  
 - 作者  
 - 发表年份  
 - 期刊/会议名称  
 - 引用次数  
 - 摘要  
 - 关键词  
 - 研究方法  
 - 主要发现  
 - 研究局限  
 - 未来研究方向"""  
  
 try:  
 response = Generation.call(  
 model='qwen-vl-plus',  
 prompt=f"{system\_prompt}\n\n研究主题：{topic}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 # 解析响应并构建论文列表  
 papers = self.\_parse\_papers\_from\_response(response.output.text)  
 self.collected\_papers.extend(papers)  
   
 collection\_record = {  
 "type": "resource\_collection",  
 "topic": topic,  
 "papers": papers,  
 "timestamp": datetime.now().isoformat()  
 }  
 self.review\_history.append(collection\_record)  
   
 return papers  
   
 except Exception as e:  
 print(f"Error collecting resources: {e}")  
 return []  
  
 def filter\_resources(self, papers: List[Dict[str, Any]],   
 min\_citations: int = 10,  
 min\_year: int = 2018) -> List[Dict[str, Any]]:  
 """基于质量指标筛选资源"""  
 filtered\_papers = []  
 for paper in papers:  
 # 计算质量分数  
 quality\_score = self.\_calculate\_quality\_score(paper, min\_citations, min\_year)  
 paper['quality\_score'] = quality\_score  
   
 # 应用筛选标准  
 if (paper.get('citations', 0) >= min\_citations and  
 paper.get('year', 0) >= min\_year and  
 quality\_score >= 0.6): # 质量分数阈值  
 filtered\_papers.append(paper)  
   
 self.filtered\_papers = filtered\_papers  
   
 filter\_record = {  
 "type": "resource\_filtering",  
 "original\_count": len(papers),  
 "filtered\_count": len(filtered\_papers),  
 "filtered\_papers": filtered\_papers,  
 "timestamp": datetime.now().isoformat()  
 }  
 self.review\_history.append(filter\_record)  
   
 return filtered\_papers  
  
 async def generate\_ideas(self, papers: List[Dict[str, Any]]) -> Dict[str, Any]:  
 """基于筛选后的资源生成研究创意"""  
 # 构建论文分析文本  
 papers\_text = "\n\n".join([  
 f"论文：{paper.get('title', '')}\n"  
 f"作者：{', '.join(paper.get('authors', []))}\n"  
 f"年份：{paper.get('year', '')}\n"  
 f"摘要：{paper.get('abstract', '')}\n"  
 f"研究方法：{paper.get('methodology', '')}\n"  
 f"主要发现：{paper.get('findings', '')}\n"  
 f"研究局限：{paper.get('limitations', '')}\n"  
 f"未来方向：{paper.get('future\_directions', '')}"  
 for paper in papers  
 ])  
  
 system\_prompt = """你是一个专业的研究创意生成专家。  
 请基于提供的论文分析，生成新的研究创意。分析应包括：  
 1. 现有研究的局限性分析  
 2. 研究方法创新建议  
 3. 研究内容扩展建议  
 4. 潜在的研究问题  
 5. 理论框架创新建议  
 6. 具体研究建议  
   
 请以结构化的方式输出分析结果。"""  
  
 try:  
 response = Generation.call(  
 model='qwen-vl-plus',  
 prompt=f"{system\_prompt}\n\n论文分析：\n{papers\_text}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 idea\_generation = {  
 "type": "idea\_generation",  
 "papers\_analyzed": len(papers),  
 "analysis": response.output.text,  
 "timestamp": datetime.now().isoformat()  
 }  
   
 self.idea\_generation\_results.append(idea\_generation)  
 return idea\_generation  
   
 except Exception as e:  
 print(f"Error generating ideas: {e}")  
 return {  
 "type": "idea\_generation",  
 "error": str(e)  
 }  
  
 def \_calculate\_quality\_score(self, paper: Dict[str, Any],   
 min\_citations: int,   
 min\_year: int) -> float:  
 """计算论文质量分数"""  
 # 引用分数 (0-0.4)  
 citation\_score = min(paper.get('citations', 0) / min\_citations, 1.0) \* 0.4  
   
 # 年份分数 (0-0.3)  
 current\_year = datetime.now().year  
 year\_score = min((paper.get('year', 0) - min\_year) / (current\_year - min\_year), 1.0) \* 0.3  
   
 # 内容完整性分数 (0-0.3)  
 content\_score = 0.0  
 if paper.get('abstract'):  
 content\_score += 0.1  
 if paper.get('methodology'):  
 content\_score += 0.1  
 if paper.get('findings'):  
 content\_score += 0.1  
   
 return citation\_score + year\_score + content\_score  
  
 def \_parse\_papers\_from\_response(self, response\_text: str) -> List[Dict[str, Any]]:  
 """解析模型响应中的论文信息"""  
 # 这里需要根据实际响应格式进行解析  
 # 示例实现  
 papers = []  
 try:  
 # 假设响应是JSON格式  
 papers = json.loads(response\_text)  
 except:  
 # 如果不是JSON格式，尝试其他解析方法  
 # 这里需要根据实际响应格式实现具体的解析逻辑  
 pass  
 return papers  
  
 def get\_review\_history(self) -> List[Dict[str, Any]]:  
 """获取文献综述历史"""  
 return self.review\_history  
  
 def get\_collected\_papers(self) -> List[Dict[str, Any]]:  
 """获取收集的论文"""  
 return self.collected\_papers  
  
 def get\_filtered\_papers(self) -> List[Dict[str, Any]]:  
 """获取筛选后的论文"""  
 return self.filtered\_papers  
  
 def get\_idea\_generation\_results(self) -> List[Dict[str, Any]]:  
 """获取创意生成结果"""  
 return self.idea\_generation\_results

================================================================================

# 文件：modules\research\_analyzer.py

from typing import Dict, List, Any  
from dashscope import Generation  
import json  
import os  
from dotenv import load\_dotenv  
  
# 加载环境变量  
load\_dotenv()  
  
# 设置API密钥  
os.environ['DASHSCOPE\_API\_KEY'] = os.getenv('API\_KEY')  
  
class ResearchAnalyzer:  
 def \_\_init\_\_(self):  
 self.analysis\_history: List[Dict[str, Any]] = []  
 self.model = 'qwen-max'  
 self.temperature = 0.7  
 self.max\_tokens = 2000  
  
 async def analyze\_research\_idea(self, idea: str) -> Dict[str, Any]:  
 """分析用户提供的研究想法"""  
 system\_prompt = """你是一位专业的研究主题分析专家。  
 请分析用户提供的研究想法，并生成详细的研究实施策略。  
 分析应包括以下方面：  
 1. 研究主题的可行性和创新性  
 2. 理论框架建议  
 3. 研究方法建议  
 4. 预期研究贡献  
 5. 潜在的研究挑战  
 6. 具体实施步骤  
 请以结构化的方式输出分析结果。"""  
  
 try:  
 response = Generation.call(  
 model='qwen-vl-plus',  
 prompt=f"{system\_prompt}\n\n研究想法：{idea}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 analysis = {  
 "type": "idea\_analysis",  
 "original\_idea": idea,  
 "analysis": response.output.text,  
 "timestamp": "current\_time" # 实际应用中应使用真实时间戳  
 }  
   
 self.analysis\_history.append(analysis)  
 return analysis  
   
 except Exception as e:  
 print(f"Error analyzing research idea: {e}")  
 return {  
 "type": "idea\_analysis",  
 "error": str(e),  
 "original\_idea": idea  
 }  
  
 async def analyze\_references(self, references: List[Dict[str, Any]]) -> Dict[str, Any]:  
 """分析用户提供的参考文献并生成研究概念"""  
 # 构建参考文献文本  
 refs\_text = "\n\n".join([  
 f"标题：{ref.get('title', '')}\n"  
 f"作者：{', '.join(ref.get('authors', []))}\n"  
 f"摘要：{ref.get('abstract', '')}"  
 for ref in references  
 ])  
  
 system\_prompt = """你是一位专业的研究主题分析专家。  
 请分析提供的参考文献，并生成新的研究概念。  
 分析应包括以下方面：  
 1. 现有研究的主题和趋势  
 2. 研究空白和机会  
 3. 潜在的研究问题  
 4. 建议的研究方向  
 5. 理论框架建议  
 6. 研究方法建议  
 请以结构化的方式输出分析结果。"""  
  
 try:  
 response = Generation.call(  
 model='qwen-vl-plus',  
 prompt=f"{system\_prompt}\n\n参考文献：\n{refs\_text}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 analysis = {  
 "type": "reference\_analysis",  
 "references": references,  
 "analysis": response.output.text,  
 "timestamp": "current\_time" # 实际应用中应使用真实时间戳  
 }  
   
 self.analysis\_history.append(analysis)  
 return analysis  
   
 except Exception as e:  
 print(f"Error analyzing references: {e}")  
 return {  
 "type": "reference\_analysis",  
 "error": str(e),  
 "references": references  
 }  
  
 def get\_analysis\_history(self) -> List[Dict[str, Any]]:  
 """获取分析历史"""  
 return self.analysis\_history  
  
 def clear\_analysis\_history(self):  
 """清除分析历史"""  
 self.analysis\_history = []  
  
 async def analyze\_research(self, research\_idea: str, context: Dict[str, Any] = None) -> Dict[str, Any]:  
 """分析研究想法并返回建议"""  
 try:  
 # 构建提示词  
 prompt = self.\_build\_prompt(research\_idea, context)  
   
 # 调用Qwen模型  
 response = Generation.call(  
 model=self.model,  
 prompt=prompt,  
 temperature=self.temperature,  
 max\_tokens=self.max\_tokens  
 )  
  
 if response.status\_code == 200:  
 return {  
 'analysis': response.output.text,  
 'status': 'success'  
 }  
 else:  
 return {  
 'error': f"Qwen API error: {response.status\_code}",  
 'status': 'error'  
 }  
  
 except Exception as e:  
 return {  
 'error': str(e),  
 'status': 'error'  
 }  
  
 def \_build\_prompt(self, research\_idea: str, context: Dict[str, Any] = None) -> str:  
 """构建分析提示词"""  
 prompt = f"""请分析以下研究想法，并提供：  
1. 研究价值评估  
2. 创新点分析  
3. 可行性分析  
4. 研究方法建议  
5. 潜在挑战和解决方案  
  
研究想法：  
{research\_idea}"""  
  
 if context and context.get('files'):  
 prompt += "\n\n相关文件分析："  
 for file in context['files']:  
 if file.get('analysis'):  
 prompt += f"\n{file['filename']}: {file['analysis']}"  
  
 return prompt

================================================================================

# 文件：modules\research\_executor.py

from typing import Dict, List, Any  
from dashscope import Generation  
import json  
from datetime import datetime  
  
class ResearchExecutor:  
 def \_\_init\_\_(self):  
 self.research\_history: List[Dict[str, Any]] = []  
 self.current\_stage = None  
 self.research\_type = None  
 self.materials = None  
 self.theory\_framework = None  
 self.research\_plan = None  
 self.analysis\_results = None  
 self.verification\_results = None  
 self.optimization\_history = []  
  
 async def start\_qualitative\_research(self, topic: str, materials: str) -> Dict[str, Any]:  
 """启动质性研究流程"""  
 self.research\_type = "qualitative"  
 self.materials = materials  
 self.current\_stage = "design"  
   
 # 设计阶段  
 design\_result = await self.\_execute\_design\_stage(topic, materials)  
 self.theory\_framework = design\_result.get("theory\_framework")  
 self.research\_plan = design\_result.get("research\_plan")  
   
 return {  
 "stage": "design",  
 "result": design\_result  
 }  
  
 async def \_execute\_design\_stage(self, topic: str, materials: str) -> Dict[str, Any]:  
 """执行设计阶段"""  
 system\_prompt = """你是一个质性研究设计专家。  
 请基于提供的研究主题和材料，完成以下任务：  
 1. 材料分析  
 - 提取关键概念  
 - 识别主要主题  
 - 发现潜在模式  
   
 2. 理论框架构建  
 - 确定理论基础  
 - 建立概念关系  
 - 形成理论框架  
   
 3. 研究方案制定  
 - 研究方法选择  
 - 分析步骤设计  
 - 数据收集计划  
 - 时间安排  
   
 请以结构化的方式输出分析结果。"""  
  
 try:  
 response = Generation.call(  
 model='qwen-vl-plus',  
 prompt=f"{system\_prompt}\n\n研究主题：{topic}\n\n研究材料：\n{materials}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 design\_result = {  
 "type": "design\_stage",  
 "analysis": response.output.text,  
 "timestamp": datetime.now().isoformat()  
 }  
   
 self.research\_history.append(design\_result)  
 return design\_result  
   
 except Exception as e:  
 print(f"Error in design stage: {e}")  
 return {  
 "type": "design\_stage",  
 "error": str(e)  
 }  
  
 async def execute\_implementation\_stage(self) -> Dict[str, Any]:  
 """执行实现阶段"""  
 self.current\_stage = "implementation"  
   
 system\_prompt = """你是一个质性研究分析专家。  
 请基于之前的设计方案和研究材料，完成以下任务：  
 1. 研究分析  
 - 按照学术论文结构进行分析  
 - 包括：引言、文献综述、研究方法、数据分析、讨论、结论  
   
 2. 论证过程  
 - 理论论证  
 - 数据支持  
 - 逻辑推理  
 - 结论推导  
   
 请以学术论文的格式输出分析结果。"""  
  
 try:  
 response = Generation.call(  
 model='qwen-vl-plus',  
 prompt=f"{system\_prompt}\n\n理论框架：\n{self.theory\_framework}\n\n研究方案：\n{self.research\_plan}\n\n研究材料：\n{self.materials}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 implementation\_result = {  
 "type": "implementation\_stage",  
 "analysis": response.output.text,  
 "timestamp": datetime.now().isoformat()  
 }  
   
 self.analysis\_results = implementation\_result  
 self.research\_history.append(implementation\_result)  
 return implementation\_result  
   
 except Exception as e:  
 print(f"Error in implementation stage: {e}")  
 return {  
 "type": "implementation\_stage",  
 "error": str(e)  
 }  
  
 async def execute\_verification\_stage(self) -> Dict[str, Any]:  
 """执行验证阶段"""  
 self.current\_stage = "verification"  
   
 system\_prompt = """你是一个质性研究验证专家。  
 请对研究结果进行验证，包括：  
 1. 结论与材料比照  
 - 检查结论是否得到材料支持  
 - 识别潜在矛盾  
 - 评估证据充分性  
   
 2. 可靠性评估  
 - 内部效度评估  
 - 外部效度评估  
 - 研究局限性分析  
   
 请以结构化的方式输出验证结果。"""  
  
 try:  
 response = Generation.call(  
 model='qwen-vl-plus',  
 prompt=f"{system\_prompt}\n\n研究材料：\n{self.materials}\n\n分析结果：\n{self.analysis\_results['analysis']}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 verification\_result = {  
 "type": "verification\_stage",  
 "analysis": response.output.text,  
 "timestamp": datetime.now().isoformat()  
 }  
   
 self.verification\_results = verification\_result  
 self.research\_history.append(verification\_result)  
 return verification\_result  
   
 except Exception as e:  
 print(f"Error in verification stage: {e}")  
 return {  
 "type": "verification\_stage",  
 "error": str(e)  
 }  
  
 async def execute\_optimization\_stage(self, content: str, professor\_feedback: str, research\_feedback: str) -> Dict[str, Any]:  
 """执行优化阶段"""  
 self.current\_stage = "optimization"  
   
 system\_prompt = """你是一个论文优化专家。  
 请基于导师反馈对论文进行优化：  
 1. 分析反馈意见  
 2. 识别需要改进的部分  
 3. 提供具体的修改建议  
 4. 生成优化后的内容  
   
 请以结构化的方式输出优化结果。"""  
  
 try:  
 response = Generation.call(  
 model='qwen-vl-plus',  
 prompt=f"{system\_prompt}\n\n当前内容：\n{content}\n\n教授反馈：\n{professor\_feedback}\n\n研究顾问反馈：\n{research\_feedback}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 optimization\_result = {  
 "type": "optimization\_stage",  
 "original\_content": content,  
 "professor\_feedback": professor\_feedback,  
 "research\_feedback": research\_feedback,  
 "optimized\_content": response.output.text,  
 "timestamp": datetime.now().isoformat()  
 }  
   
 self.optimization\_history.append(optimization\_result)  
 return optimization\_result  
   
 except Exception as e:  
 print(f"Error in optimization stage: {e}")  
 return {  
 "type": "optimization\_stage",  
 "error": str(e)  
 }  
  
 def get\_research\_history(self) -> List[Dict[str, Any]]:  
 """获取研究历史"""  
 return self.research\_history  
  
 def get\_optimization\_history(self) -> List[Dict[str, Any]]:  
 """获取优化历史"""  
 return self.optimization\_history  
  
 def get\_current\_stage(self) -> str:  
 """获取当前阶段"""  
 return self.current\_stage

================================================================================

# 文件：modules\thesis\_evaluation.py

from typing import Dict, List, Any  
from dashscope import Generation  
  
class ThesisEvaluationModule:  
 def \_\_init\_\_(self):  
 self.evaluation\_history: List[Dict[str, Any]] = []  
  
 async def evaluate\_thesis(self, thesis\_content: str) -> Dict[str, Any]:  
 """Evaluate the overall quality of the thesis"""  
 system\_prompt = """你是一位专业的论文评估专家。  
 请基于学术标准、理论贡献、研究方法和写作质量评估论文。  
 提供详细的反馈和改进建议。"""  
  
 try:  
 response = Generation.call(  
 model='qwen-max',  
 prompt=f"{system\_prompt}\n\n请评估以下论文内容：\n{thesis\_content}",  
 temperature=0.3,  
 max\_tokens=2000  
 )  
   
 evaluation = {  
 "overall\_score": 0.0, # Score from 0 to 1  
 "strengths": [],  
 "weaknesses": [],  
 "suggestions": [],  
 "detailed\_feedback": response.output.text,  
 "timestamp": "current\_time" # You might want to use actual timestamps  
 }  
  
 self.evaluation\_history.append(evaluation)  
 return evaluation  
 except Exception as e:  
 print(f"Error in thesis evaluation: {e}")  
 return {  
 "error": str(e),  
 "timestamp": "current\_time"  
 }  
  
 async def evaluate\_section(self, section\_content: str, section\_type: str) -> Dict[str, Any]:  
 """Evaluate a specific section of the thesis"""  
 system\_prompt = f"""你是一位专业的论文评估专家。  
 请基于学术标准和最佳实践评估这个{section\_type}部分。  
 提供具体的反馈和改进建议。"""  
  
 try:  
 response = Generation.call(  
 model='qwen-max',  
 prompt=f"{system\_prompt}\n\n请评估以下{section\_type}部分：\n{section\_content}",  
 temperature=0.3,  
 max\_tokens=2000  
 )  
   
 evaluation = {  
 "section\_type": section\_type,  
 "score": 0.0, # Score from 0 to 1  
 "strengths": [],  
 "weaknesses": [],  
 "suggestions": [],  
 "detailed\_feedback": response.output.text,  
 "timestamp": "current\_time" # You might want to use actual timestamps  
 }  
  
 self.evaluation\_history.append(evaluation)  
 return evaluation  
 except Exception as e:  
 print(f"Error in section evaluation: {e}")  
 return {  
 "error": str(e),  
 "section\_type": section\_type,  
 "timestamp": "current\_time"  
 }  
  
 def check\_plagiarism(self, content: str) -> Dict[str, Any]:  
 """Check for potential plagiarism issues"""  
 # In a real implementation, this would connect to plagiarism detection services  
 # For now, we'll return a simulated result  
 return {  
 "plagiarism\_score": 0.0, # Score from 0 to 1  
 "potential\_issues": [],  
 "suggestions": []  
 }  
  
 def evaluate\_citations(self, citations: List[Dict[str, Any]]) -> Dict[str, Any]:  
 """Evaluate the quality and appropriateness of citations"""  
 return {  
 "citation\_count": len(citations),  
 "quality\_score": 0.0, # Score from 0 to 1  
 "suggestions": [],  
 "missing\_citations": []  
 }  
  
 def get\_evaluation\_history(self) -> List[Dict[str, Any]]:  
 """Get history of all evaluations"""  
 return self.evaluation\_history

================================================================================

# 文件：modules\\_\_init\_\_.py

from .knowledge\_acquisition import KnowledgeAcquisitionModule  
from .thesis\_evaluation import ThesisEvaluationModule  
  
\_\_all\_\_ = ['KnowledgeAcquisitionModule', 'ThesisEvaluationModule']

================================================================================

# 文件：modules\evaluation\_module\quality\_evaluator.py

from typing import Dict, List, Any  
from dashscope import Generation  
  
class QualityEvaluator:  
 def \_\_init\_\_(self):  
 self.evaluation\_history: List[Dict[str, Any]] = []  
  
 async def evaluate\_quality(self, topic: str, content: Dict[str, Any]) -> Dict[str, Any]:  
 """评估论文质量"""  
 try:  
 system\_prompt = """你是一位论文质量评估专家。  
 请根据研究主题和论文内容，评估论文质量，包括：  
 1. 学术性  
 2. 创新性  
 3. 逻辑性  
 4. 完整性  
 5. 规范性  
 6. 改进建议  
 """  
   
 response = await Generation.call(  
 model='qwen-max',  
 prompt=f"{system\_prompt}\n\n研究主题：{topic}\n\n论文内容：{content['content']}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 evaluation = {  
 "topic": topic,  
 "content": content,  
 "evaluation": response.output.text,  
 "timestamp": "current\_time"  
 }  
 self.evaluation\_history.append(evaluation)  
   
 return evaluation  
 except Exception as e:  
 print(f"Error in quality evaluation: {e}")  
 return {"error": str(e)}  
  
 def get\_evaluation\_history(self) -> List[Dict[str, Any]]:  
 """获取评估历史"""  
 return self.evaluation\_history

================================================================================

# 文件：modules\evaluation\_module\\_\_init\_\_.py

from .quality\_evaluator import QualityEvaluator  
  
\_\_all\_\_ = ['QualityEvaluator']

================================================================================

# 文件：modules\literature\_review\_module\idea\_generator.py

from typing import Dict, List, Any  
from dashscope import Generation  
  
class IdeaGenerator:  
 def \_\_init\_\_(self):  
 self.idea\_history: List[Dict[str, Any]] = []  
  
 async def generate\_research\_ideas(self, topic: str, gap\_analysis: Dict[str, Any]) -> Dict[str, Any]:  
 """生成研究想法"""  
 try:  
 system\_prompt = """你是一位研究想法生成专家。  
 请根据研究主题和研究空白分析，生成具体的研究想法，包括：  
 1. 研究问题  
 2. 研究假设  
 3. 研究方法  
 4. 预期成果  
 5. 创新点  
 """  
   
 response = await Generation.call(  
 model='qwen-max',  
 prompt=f"{system\_prompt}\n\n研究主题：{topic}\n\n研究空白：{gap\_analysis['gaps']}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 research\_ideas = {  
 "topic": topic,  
 "gap\_analysis": gap\_analysis,  
 "ideas": response.output.text,  
 "timestamp": "current\_time"  
 }  
 self.idea\_history.append(research\_ideas)  
   
 return research\_ideas  
 except Exception as e:  
 print(f"Error in idea generation: {e}")  
 return {"error": str(e)}  
  
 def get\_idea\_history(self) -> List[Dict[str, Any]]:  
 """获取研究想法历史"""  
 return self.idea\_history

================================================================================

# 文件：modules\literature\_review\_module\literature\_analyzer.py

from typing import Dict, List, Any  
from dashscope import Generation  
  
class LiteratureAnalyzer:  
 def \_\_init\_\_(self):  
 self.analysis\_history: List[Dict[str, Any]] = []  
  
 async def analyze\_literature(self, topic: str, papers: List[Dict[str, Any]]) -> Dict[str, Any]:  
 """分析文献内容"""  
 try:  
 system\_prompt = """你是一位文献分析专家。  
 请根据研究主题和文献内容，进行深入分析，包括：  
 1. 文献综述  
 2. 主要观点  
 3. 研究方法  
 4. 研究结论  
 5. 研究贡献  
 """  
   
 papers\_text = "\n\n".join([f"论文{i+1}：{paper['content']}" for i, paper in enumerate(papers)])  
   
 response = await Generation.call(  
 model='qwen-max',  
 prompt=f"{system\_prompt}\n\n研究主题：{topic}\n\n文献内容：\n{papers\_text}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 analysis\_result = {  
 "topic": topic,  
 "papers": papers,  
 "analysis": response.output.text,  
 "timestamp": "current\_time"  
 }  
 self.analysis\_history.append(analysis\_result)  
   
 return analysis\_result  
 except Exception as e:  
 print(f"Error in literature analysis: {e}")  
 return {"error": str(e)}  
  
 def get\_analysis\_history(self) -> List[Dict[str, Any]]:  
 """获取文献分析历史"""  
 return self.analysis\_history

================================================================================

# 文件：modules\literature\_review\_module\research\_gap\_analyzer.py

from typing import Dict, List, Any  
from dashscope import Generation  
  
class ResearchGapAnalyzer:  
 def \_\_init\_\_(self):  
 self.gap\_analysis\_history: List[Dict[str, Any]] = []  
  
 async def analyze\_research\_gaps(self, topic: str, literature\_analysis: Dict[str, Any]) -> Dict[str, Any]:  
 """分析研究空白"""  
 try:  
 system\_prompt = """你是一位研究空白分析专家。  
 请根据研究主题和文献分析结果，识别研究空白，包括：  
 1. 未解决的问题  
 2. 研究机会  
 3. 潜在的研究方向  
 4. 创新点建议  
 """  
   
 response = await Generation.call(  
 model='qwen-max',  
 prompt=f"{system\_prompt}\n\n研究主题：{topic}\n\n文献分析：{literature\_analysis['analysis']}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 gap\_analysis = {  
 "topic": topic,  
 "literature\_analysis": literature\_analysis,  
 "gaps": response.output.text,  
 "timestamp": "current\_time"  
 }  
 self.gap\_analysis\_history.append(gap\_analysis)  
   
 return gap\_analysis  
 except Exception as e:  
 print(f"Error in research gap analysis: {e}")  
 return {"error": str(e)}  
  
 def get\_gap\_analysis\_history(self) -> List[Dict[str, Any]]:  
 """获取研究空白分析历史"""  
 return self.gap\_analysis\_history

================================================================================

# 文件：modules\literature\_review\_module\\_\_init\_\_.py

from .literature\_analyzer import LiteratureAnalyzer  
from .research\_gap\_analyzer import ResearchGapAnalyzer  
from .idea\_generator import IdeaGenerator  
  
\_\_all\_\_ = ['LiteratureAnalyzer', 'ResearchGapAnalyzer', 'IdeaGenerator']

================================================================================

# 文件：modules\research\_module\data\_collector.py

from typing import Dict, List, Any  
from dashscope import Generation  
  
class DataCollector:  
 def \_\_init\_\_(self):  
 self.collection\_history: List[Dict[str, Any]] = []  
  
 async def collect\_data(self, topic: str, methodology: str) -> Dict[str, Any]:  
 """收集研究数据"""  
 try:  
 system\_prompt = """你是一位数据收集专家。  
 请根据研究主题和方法论，设计数据收集方案，包括：  
 1. 数据来源  
 2. 收集方法  
 3. 样本选择  
 4. 质量控制  
 """  
   
 response = await Generation.call(  
 model='qwen-max',  
 prompt=f"{system\_prompt}\n\n研究主题：{topic}\n\n研究方法：{methodology}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 collection\_plan = {  
 "topic": topic,  
 "methodology": methodology,  
 "plan": response.output.text,  
 "timestamp": "current\_time"  
 }  
 self.collection\_history.append(collection\_plan)  
   
 return collection\_plan  
 except Exception as e:  
 print(f"Error in data collection: {e}")  
 return {"error": str(e)}  
  
 def get\_collection\_history(self) -> List[Dict[str, Any]]:  
 """获取数据收集历史"""  
 return self.collection\_history

================================================================================

# 文件：modules\research\_module\research\_executor.py

from typing import Dict, List, Any  
from dashscope import Generation  
  
class ResearchExecutor:  
 def \_\_init\_\_(self):  
 self.execution\_history: List[Dict[str, Any]] = []  
  
 async def execute\_research(self, topic: str, data: Dict[str, Any], methodology: str) -> Dict[str, Any]:  
 """执行研究分析"""  
 try:  
 system\_prompt = """你是一位研究执行专家。  
 请根据研究主题、数据和方法论，执行研究分析，包括：  
 1. 数据分析  
 2. 结果解释  
 3. 假设验证  
 4. 研究结论  
 """  
   
 response = await Generation.call(  
 model='qwen-max',  
 prompt=f"{system\_prompt}\n\n研究主题：{topic}\n\n数据：{data}\n\n研究方法：{methodology}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 execution\_result = {  
 "topic": topic,  
 "methodology": methodology,  
 "results": response.output.text,  
 "timestamp": "current\_time"  
 }  
 self.execution\_history.append(execution\_result)  
   
 return execution\_result  
 except Exception as e:  
 print(f"Error in research execution: {e}")  
 return {"error": str(e)}  
  
 def get\_execution\_history(self) -> List[Dict[str, Any]]:  
 """获取研究执行历史"""  
 return self.execution\_history

================================================================================

# 文件：modules\research\_module\research\_planner.py

from typing import Dict, List, Any  
from dashscope import Generation  
  
class ResearchPlanner:  
 def \_\_init\_\_(self):  
 self.plan\_history: List[Dict[str, Any]] = []  
  
 async def create\_research\_plan(self, topic: str, requirements: Dict[str, Any]) -> Dict[str, Any]:  
 """创建研究计划"""  
 try:  
 system\_prompt = """你是一位研究规划专家。  
 请根据研究主题和需求，制定详细的研究计划，包括：  
 1. 研究目标  
 2. 研究方法  
 3. 数据需求  
 4. 时间安排  
 5. 预期成果  
 """  
   
 response = await Generation.call(  
 model='qwen-max',  
 prompt=f"{system\_prompt}\n\n研究主题：{topic}\n\n研究需求：{requirements}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 research\_plan = {  
 "topic": topic,  
 "requirements": requirements,  
 "plan": response.output.text,  
 "timestamp": "current\_time"  
 }  
 self.plan\_history.append(research\_plan)  
   
 return research\_plan  
 except Exception as e:  
 print(f"Error in research planning: {e}")  
 return {"error": str(e)}  
  
 def get\_plan\_history(self) -> List[Dict[str, Any]]:  
 """获取研究计划历史"""  
 return self.plan\_history

================================================================================

# 文件：modules\research\_module\\_\_init\_\_.py

from .data\_collector import DataCollector  
from .research\_executor import ResearchExecutor  
from .research\_planner import ResearchPlanner  
  
\_\_all\_\_ = ['DataCollector', 'ResearchExecutor', 'ResearchPlanner']

================================================================================

# 文件：modules\writing\_module\content\_writer.py

from typing import Dict, List, Any  
from dashscope import Generation  
  
class ContentWriter:  
 def \_\_init\_\_(self):  
 self.writing\_history: List[Dict[str, Any]] = []  
  
 async def write\_content(self, topic: str, outline: Dict[str, Any], section: str) -> Dict[str, Any]:  
 """撰写论文内容"""  
 try:  
 system\_prompt = """你是一位学术写作专家。  
 请根据研究主题和大纲，撰写论文的指定部分，要求：  
 1. 学术性强  
 2. 逻辑清晰  
 3. 论证充分  
 4. 语言准确  
 5. 格式规范  
 """  
   
 response = await Generation.call(  
 model='qwen-max',  
 prompt=f"{system\_prompt}\n\n研究主题：{topic}\n\n大纲：{outline['outline']}\n\n需要撰写的部分：{section}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 content = {  
 "topic": topic,  
 "outline": outline,  
 "section": section,  
 "content": response.output.text,  
 "timestamp": "current\_time"  
 }  
 self.writing\_history.append(content)  
   
 return content  
 except Exception as e:  
 print(f"Error in content writing: {e}")  
 return {"error": str(e)}  
  
 def get\_writing\_history(self) -> List[Dict[str, Any]]:  
 """获取写作历史"""  
 return self.writing\_history

================================================================================

# 文件：modules\writing\_module\outline\_generator.py

from typing import Dict, List, Any  
from dashscope import Generation  
  
class OutlineGenerator:  
 def \_\_init\_\_(self):  
 self.outline\_history: List[Dict[str, Any]] = []  
  
 async def generate\_outline(self, topic: str, research\_ideas: Dict[str, Any]) -> Dict[str, Any]:  
 """生成论文大纲"""  
 try:  
 system\_prompt = """你是一位论文大纲生成专家。  
 请根据研究主题和研究想法，生成详细的论文大纲，包括：  
 1. 标题  
 2. 摘要  
 3. 引言  
 4. 文献综述  
 5. 研究方法  
 6. 结果分析  
 7. 讨论  
 8. 结论  
 9. 参考文献  
 """  
   
 response = await Generation.call(  
 model='qwen-max',  
 prompt=f"{system\_prompt}\n\n研究主题：{topic}\n\n研究想法：{research\_ideas['ideas']}",  
 temperature=0.7,  
 max\_tokens=2000  
 )  
   
 outline = {  
 "topic": topic,  
 "research\_ideas": research\_ideas,  
 "outline": response.output.text,  
 "timestamp": "current\_time"  
 }  
 self.outline\_history.append(outline)  
   
 return outline  
 except Exception as e:  
 print(f"Error in outline generation: {e}")  
 return {"error": str(e)}  
  
 def get\_outline\_history(self) -> List[Dict[str, Any]]:  
 """获取大纲历史"""  
 return self.outline\_history

================================================================================

# 文件：modules\writing\_module\\_\_init\_\_.py

from .outline\_generator import OutlineGenerator  
from .content\_writer import ContentWriter  
  
\_\_all\_\_ = ['OutlineGenerator', 'ContentWriter']

================================================================================