

自然写手写识别与AI分析引擎软件 V1.0

软件著作权鉴别材料 — 源程序

权利人：深圳自然写科技有限公司

版本号：V1.0

源程序目录结构

```
02-writech-ai-engine/  
├── main.py  
├── api/  
│   ├── essay_api.py  
│   ├── math_api.py  
│   ├── ocr_api.py  
│   └── stroke_order_api.py  
├── config/  
│   └── settings.py  
├── engine/  
│   ├── essay_scorer.py  
│   └── stroke_analyzer.py  
├── grpc_server/  
│   └── inference_service.py  
├── preprocessing/  
│   └── stroke_processor.py  
└── service/  
    ├── model_manager.py  
    └── task_scheduler.py
```

源程序文件清单

(根目录)

main.py

```
# -*- coding: utf-8 -*-  
.....
```

自然写手写识别与AI分析引擎软件 V1.0

版权所有 (C) 2026

软件全称：自然写手写识别与AI分析引擎软件

版本号：V1.0

主启动文件 - FastAPI 服务入口

负责服务初始化、路由注册、中间件配置

"""

```
from fastapi import FastAPI, Request, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import JSONResponse
from contextlib import asynccontextmanager
import uvicorn
import logging
import time
from typing import Dict, Any

# 导入各业务模块路由
from api.ocr_api import router as ocr_router
from api.math_api import router as math_router
from api.stroke_order_api import router as stroke_order_router
from api.essay_api import router as essay_router
from service.model_manager import ModelManager
from config.settings import Settings

# 日志配置
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(name)s: %(message)s"
)
logger = logging.getLogger("writech-ai-engine")

# 全局配置
settings = Settings()

# 全局模型管理器实例
model_manager = ModelManager(settings)

@asynccontextmanager
async def lifespan(app: FastAPI):
    """
    应用生命周期管理
    启动时加载所有AI模型到GPU/CPU内存
    关闭时释放模型资源
    """
    logger.info("自然写AI引擎启动中，加载模型...")
    # 启动时加载所有模型
    await model_manager.load_all_models()
    logger.info("所有模型加载完成，服务就绪")
    yield
    # 关闭时释放资源
    logger.info("服务关闭中，释放模型资源...")
    model_manager.release_all_models()
    logger.info("模型资源已释放")
```

```

# 创建 FastAPI 应用实例
app = FastAPI(
    title="自然写手写识别与AI分析引擎",
    description="对智能点阵笔采集的笔迹数据进行OCR识别、数学列式识别、笔顺分析及AI智能批改",
    version="1.0.0",
    lifespan=lifespan
)

# 跨域中间件配置
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.middleware("http")
async def request_logging_middleware(request: Request, call_next):
    """
    请求日志与性能监控中间件
    记录每个请求的处理时间、状态码、推理耗时
    """
    start_time = time.time()
    request_id = request.headers.get("X-Request-ID", str(time.time()))

    # 输入数据大小校验（防恶意攻击，最大10MB）
    content_length = request.headers.get("content-length")
    if content_length and int(content_length) > 10 * 1024 * 1024:
        return JSONResponse(
            status_code=413,
            content={"code": 413, "msg": "请求数据过大，最大支持10MB", "data": None}
        )

    response = await call_next(request)

    # 记录请求处理时间
    process_time = time.time() - start_time
    response.headers["X-Process-Time"] = f"{process_time:.4f}"
    response.headers["X-Request-ID"] = request_id

    logger.info(
        f"{request.method} {request.url.path} "
        f"status={response.status_code} "
        f"time={process_time:.4f}s"
    )

    return response

@app.middleware("http")
async def mtls_authentication_middleware(request: Request, call_next):
    """
    mTLS 双向认证中间件

```

内部服务间通信需携带有效的客户端证书

安全设计:

- 服务鉴权: 内部服务间 mTLS 双向认证
- 请求校验: 输入数据格式校验与大小限制 (防恶意攻击)

"""

检查是否为内部服务调用

client_cert = request.headers.get("X-Client-Cert")

api_key = request.headers.get("X-API-Key")

白名单路径不需要认证

whitelist_paths = ["/health", "/docs", "/openapi.json"]

if request.url.path in whitelist_paths:

return await call_next(request)

验证API Key或客户端证书

if not api_key and not client_cert:

return JSONResponse(

status_code=401,

content={"code": 401, "msg": "缺少认证凭据", "data": None}

)

if api_key and api_key != settings.api_key:

return JSONResponse(

status_code=403,

content={"code": 403, "msg": "API Key无效", "data": None}

)

return await call_next(request)

注册各业务路由

app.include_router(ocr_router, prefix="/api/v1/ocr", tags=["OCR识别"])

app.include_router(math_router, prefix="/api/v1/math", tags=["数学识别"])

app.include_router(stroke_order_router, prefix="/api/v1/stroke-order", tags=["笔顺评分"])

app.include_router(essay_router, prefix="/api/v1/essay", tags=["作文批改"])

@app.get("/health")

async def health_check():

"""健康检查端点"""

model_status = model_manager.get_all_status()

return {

"code": 200,

"msg": "success",

"data": {

"status": "healthy",

"models": model_status,

"version": "1.0.0"

}

}

@app.get("/api/v1/model/status")

async def get_model_status():

"""

查询各模型加载状态与版本

```

GET /api/v1/model/status
"""
status = model_manager.get_all_status()
return {
    "code": 200,
    "msg": "success",
    "data": status
}

@app.exception_handler(HTTPException)
async def http_exception_handler(request: Request, exc: HTTPException):
    """统一HTTP异常处理"""
    return JSONResponse(
        status_code=exc.status_code,
        content={
            "code": exc.status_code,
            "msg": exc.detail,
            "data": None
        }
    )

@app.exception_handler(Exception)
async def general_exception_handler(request: Request, exc: Exception):
    """统一异常处理"""
    logger.error(f"未处理异常: {str(exc)}", exc_info=True)
    return JSONResponse(
        status_code=500,
        content={
            "code": 500,
            "msg": "AI引擎内部错误",
            "data": None
        }
    )

if __name__ == "__main__":
    uvicorn.run(
        "main:app",
        host="0.0.0.0",
        port=8001,
        workers=4,
        log_level="info"
    )

```

api/

api/essay_api.py

```

# 自然写手写识别与AI分析引擎软件 V1.0
# 作文批改接口模块 - AI作文评分与批改建议服务

```

```
"""
```

作文批改API接口

提供AI作文评分、多维度分析（结构/语法/内容/修辞）、批改建议生成等功能

支持小学至初中阶段作文批改，基于大语言模型与NLP分析管道

```
"""
```

```
import time
import json
import logging
import hashlib
import re
from typing import List, Dict, Optional, Tuple
from dataclasses import dataclass, field
from enum import Enum
from fastapi import APIRouter, HTTPException, Depends
from pydantic import BaseModel, Field, validator

logger = logging.getLogger(__name__)

# ===== 数据模型定义 =====

class EssayReviewRequest(BaseModel):
    """作文批改请求"""
    text: str = Field(..., min_length=10, max_length=5000, description="作文OCR识别文本")
    title: Optional[str] = Field(None, description="作文题目")
    grade: int = Field(3, ge=1, le=9, description="年级(1-9)")
    genre: str = Field("narrative", description="文体类型：narrative/argumentative/expository/descriptive")
    max_score: int = Field(100, description="满分值")
    student_id: Optional[str] = Field(None, description="学生ID")
    assignment_id: Optional[str] = Field(None, description="作业ID")
    enable_suggestions: bool = Field(True, description="是否生成修改建议")

    @validator('genre')
    def validate_genre(cls, v):
        valid_genres = ['narrative', 'argumentative', 'expository', 'descriptive']
        if v not in valid_genres:
            raise ValueError(f'文体类型必须为: {valid_genres}')
        return v

class SentenceError(BaseModel):
    """句子级错误标注"""
    sentence: str = Field(..., description="原始句子")
    error_type: str = Field(..., description="错误类型")
    suggestion: str = Field(..., description="修改建议")
    position: int = Field(..., description="句子在原文中的位置索引")

class EssayScoreDetail(BaseModel):
    """作文各维度评分详情"""
    structure: float = Field(..., description="结构分")
    grammar: float = Field(..., description="语法分")
    content: float = Field(..., description="内容分")
    rhetoric: float = Field(..., description="修辞分")
    handwriting: Optional[float] = Field(None, description="书写分 (如有)")
```

```
# ===== 文本分析工具 =====

class TextAnalyzer:
    """
    文本分析工具类
    提供基础的中文文本分析功能：分句、词频统计、句式分析等
    """

    # 中文句末标点
    SENTENCE_ENDINGS = {'。', '!', '?', '.....', ';'}
    # 中文段落标识
    PARAGRAPH_INDENT = '    '

    @staticmethod
    def split_sentences(text: str) -> List[str]:
        """将文本分割为句子列表"""
        sentences = []
        current = ""
        for char in text:
            current += char
            if char in TextAnalyzer.SENTENCE_ENDINGS:
                if current.strip():
                    sentences.append(current.strip())
                current = ""
        if current.strip():
            sentences.append(current.strip())
        return sentences

    @staticmethod
    def split_paragraphs(text: str) -> List[str]:
        """将文本分割为段落列表"""
        # 按换行符分割，过滤空段落
        paragraphs = [p.strip() for p in text.split('\n') if p.strip()]
        return paragraphs

    @staticmethod
    def count_characters(text: str) -> Dict[str, int]:
        """统计文本字符数"""
        chinese_count = sum(1 for c in text if '\u4e00' <= c <= '\u9fff')
        punctuation_count = sum(1 for c in text if c in ',.!?,:""'()《》.....-')
        total_count = len(text.replace(' ', '').replace('\n', ''))
        return {
            "total": total_count,
            "chinese": chinese_count,
            "punctuation": punctuation_count
        }

    @staticmethod
    def detect_rhetoric(text: str) -> List[Dict]:
        """
        检测修辞手法使用情况
        识别常见修辞：比喻、排比、拟人、夸张等
        """
        rhetorics = []

        # 比喻检测：包含"像...一样"、"如同"、"仿佛"等关键词

```

```

simile_patterns = [
    r'像.{2,10}一样', r'如同.{2,10}', r'仿佛.{2,10}',
    r'好像.{2,10}', r'犹如.{2,10}', r'宛如.{2,10}'
]
for pattern in simile_patterns:
    matches = re.finditer(pattern, text)
    for m in matches:
        rhetorics.append({
            "type": "simile", "name": "比喻",
            "text": m.group(), "position": m.start()
        })

# 排比检测: 连续出现相似句式结构
sentences = TextAnalyzer.split_sentences(text)
for i in range(len(sentences) - 2):
    s1, s2, s3 = sentences[i], sentences[i+1], sentences[i+2]
    # 简化判断: 三个连续句子长度相近且首字相同
    if (abs(len(s1) - len(s2)) < 5 and abs(len(s2) - len(s3)) < 5 and
        len(s1) > 5 and s1[0] == s2[0] == s3[0]):
        rhetorics.append({
            "type": "parallelism", "name": "排比",
            "text": f"{s1}{s2}{s3}", "position": text.find(s1)
        })

# 拟人检测: 非人事物使用人的动作词
personification_patterns = [
    r'[风雨雪花树草月阳光河水山].{0,3}[笑哭唱跳跑走说叫]',
    r'[风雨雪花树草月阳光河水山].{0,3}[温柔轻轻悄悄]'
]
for pattern in personification_patterns:
    matches = re.finditer(pattern, text)
    for m in matches:
        rhetorics.append({
            "type": "personification", "name": "拟人",
            "text": m.group(), "position": m.start()
        })

return rhetorics

# ===== 作文评分引擎 =====

class EssayScoringEngine:
    """
    作文评分引擎
    基于多维度分析管道对作文进行综合评分
    评分维度: 结构(25%)、语法(25%)、内容(30%)、修辞(20%)
    """

    # 各年级期望字数范围
    EXPECTED_LENGTH = {
        1: (50, 150), 2: (100, 250), 3: (200, 400),
        4: (300, 500), 5: (350, 600), 6: (400, 700),
        7: (500, 800), 8: (600, 900), 9: (600, 1000)
    }

    # 评分维度权重配置

```



```

DIMENSION_WEIGHTS = {
    "structure": 0.25,
    "grammar": 0.25,
    "content": 0.30,
    "rhetoric": 0.20
}

def __init__(self):
    self._text_analyzer = TextAnalyzer()
    self._error_patterns = self._load_error_patterns()
    logger.info("作文评分引擎初始化完成")

def _load_error_patterns(self) -> List[Dict]:
    """加载常见语法错误模式库"""
    return [
        {"pattern": r"的的", "type": "repetition", "msg": "重复用字'的的'"},
        {"pattern": r"了了", "type": "repetition", "msg": "重复用字'了了'"},
        {"pattern": r"因为.{5,50}因为", "type": "logic", "msg": "重复使用'因为', 建议精简"},
        {"pattern": r"然后.{3,20}然后.{3,20}然后", "type": "style", "msg": "过度使用'然后'连接"},
        {"pattern": r"非常非常", "type": "repetition", "msg": "重复使用'非常'"},
        {"pattern": r"[,]{3,}", "type": "punctuation", "msg": "连续使用多个逗号, 建议使用句号断句"},
    ]

def score_structure(self, text: str, grade: int) -> Tuple[float, List[str]]:
    """
    评估文章结构 (满分100)
    检查: 段落划分、开头结尾完整性、字数是否达标、层次是否清晰
    """
    comments = []
    score = 100.0

    paragraphs = self._text_analyzer.split_paragraphs(text)
    char_stats = self._text_analyzer.count_characters(text)

    # 段落数评估 (期望3-8段)
    if len(paragraphs) < 2:
        score -= 25
        comments.append("文章缺少段落划分, 建议分段书写使结构更清晰")
    elif len(paragraphs) < 3:
        score -= 10
        comments.append("段落较少, 建议增加过渡段落")

    # 字数评估
    expected = self.EXPECTED_LENGTH.get(grade, (300, 600))
    if char_stats["chinese"] < expected[0]:
        deficit = expected[0] - char_stats["chinese"]
        score -= min(30, deficit // 10)
        comments.append(f"字数偏少 ({char_stats['chinese']}字), 该年级建议{expected[0]}-{expected[1]}字")
    elif char_stats["chinese"] > expected[1] * 1.5:
        score -= 5
        comments.append("字数偏多, 建议精简语句突出重点")

    # 开头结尾评估

```

```

    if paragraphs:
        first_para = paragraphs[0]
        last_para = paragraphs[-1]
        if len(first_para) < 15:
            score -= 10
            comments.append("开头过于简短，建议丰富开篇引入")
        if len(last_para) < 10:
            score -= 10
            comments.append("结尾过于简短，建议加强收束呼应主题")

    return max(0, score), comments

def score_grammar(self, text: str) -> Tuple[float, List[SentenceError]]:
    """
    评估语法正确性（满分100）
    检查：常见语病、标点使用、词语搭配
    """
    errors = []
    score = 100.0

    # 使用预定义的错误模式进行匹配检测
    for ep in self._error_patterns:
        matches = re.finditer(ep["pattern"], text)
        for m in matches:
            errors.append(SentenceError(
                sentence=m.group(),
                error_type=ep["type"],
                suggestion=ep["msg"],
                position=m.start()
            ))
            score -= 5 # 每个语法错误扣5分

    # 检查句子长度（过长的句子可能有语病）
    sentences = self._text_analyzer.split_sentences(text)
    for i, s in enumerate(sentences):
        if len(s) > 80:
            errors.append(SentenceError(
                sentence=s[:30] + "...",
                error_type="long_sentence",
                suggestion="句子过长，建议拆分为多个短句以提高可读性",
                position=text.find(s)
            ))
            score -= 3

    return max(0, score), errors

def score_content(self, text: str, title: Optional[str], genre: str, grade: int) ->
Tuple[float, List[str]]:
    """
    评估内容质量（满分100）
    检查：主题相关性、内容丰富度、逻辑连贯性、情感表达
    """
    comments = []
    score = 85.0 # 基础分（内容难以精确量化，给予较高基础分）

    char_stats = self._text_analyzer.count_characters(text)
    sentences = self._text_analyzer.split_sentences(text)

```

```

# 内容丰富度：通过不同词汇的数量粗略评估
unique_chars = set(c for c in text if '\u4e00' <= c <= '\u9fff')
vocab_richness = len(unique_chars) / max(char_stats["chinese"], 1)
if vocab_richness > 0.6:
    score += 10
    comments.append("词汇丰富，用词多样化")
elif vocab_richness < 0.3:
    score -= 10
    comments.append("词汇较为单一，建议使用更丰富的词语表达")

# 逻辑连贯性：检查是否使用连接词
connectors = ['因此', '所以', '但是', '然而', '首先', '其次', '最后', '总之',
              '不仅', '而且', '虽然', '但', '因为', '于是']
used_connectors = [c for c in connectors if c in text]
if len(used_connectors) >= 3:
    score += 5
    comments.append("逻辑衔接词使用恰当，行文连贯")
elif len(used_connectors) == 0 and len(sentences) > 5:
    score -= 5
    comments.append("缺少逻辑连接词，建议增加过渡衔接使行文更连贯")

# 情感表达评估
emotion_words = ['开心', '快乐', '高兴', '感动', '难过', '伤心', '惊讶',
                 '温暖', '幸福', '骄傲', '担心', '紧张']
used_emotions = [w for w in emotion_words if w in text]
if used_emotions:
    score += 3
    comments.append("有恰当的情感表达，增强了文章感染力")

return min(100, max(0, score)), comments

def score_rhetoric(self, text: str, grade: int) -> Tuple[float, List[str]]:
    """
    评估修辞运用（满分100）
    检查：修辞手法的使用数量和质量
    """
    comments = []
    score = 70.0 # 基础分

    rhetorics = self._text_analyzer.detect_rhetoric(text)

    # 根据检测到的修辞数量加分
    rhetoric_types = set(r["type"] for r in rhetorics)
    if len(rhetoric_types) >= 3:
        score += 25
        comments.append(f"修辞手法运用丰富，使用了{len(rhetoric_types)}种修辞手法")
    elif len(rhetoric_types) >= 1:
        score += 15
        used_names = set(r["name"] for r in rhetorics)
        comments.append(f"使用了{'、'.join(used_names)}等修辞手法")
    else:
        comments.append("建议适当使用比喻、排比等修辞手法增强表达效果")

    # 高年级对修辞有更高要求
    if grade >= 5 and len(rhetoric_types) < 2:
        score -= 10

```

```

        comments.append("该年级建议至少使用2种以上修辞手法")

    return min(100, max(0, score)), comments

def review_essay(self, request: EssayReviewRequest) -> Dict:
    """
    综合批改作文，返回总分和各维度分析结果
    """
    start_time = time.time()

    # 各维度独立评分
    struct_score, struct_comments = self.score_structure(request.text,
request.grade)
    grammar_score, grammar_errors = self.score_grammar(request.text)
    content_score, content_comments = self.score_content(
        request.text, request.title, request.genre, request.grade)
    rhetoric_score, rhetoric_comments = self.score_rhetoric(request.text,
request.grade)

    # 按权重计算总分，并映射到满分值
    weighted_score = (
        struct_score * self.DIMENSION_WEIGHTS["structure"] +
        grammar_score * self.DIMENSION_WEIGHTS["grammar"] +
        content_score * self.DIMENSION_WEIGHTS["content"] +
        rhetoric_score * self.DIMENSION_WEIGHTS["rhetoric"]
    )
    total_score = round(weighted_score / 100 * request.max_score, 1)

    # 字数统计
    char_stats = TextAnalyzer.count_characters(request.text)

    # 生成综合评语
    overall_comment = self._generate_overall_comment(
        total_score, request.max_score, struct_comments,
        content_comments, rhetoric_comments
    )

    elapsed = (time.time() - start_time) * 1000

    result = {
        "total_score": total_score,
        "max_score": request.max_score,
        "dimensions": {
            "structure": round(struct_score / 100 * request.max_score *
self.DIMENSION_WEIGHTS["structure"], 1),
            "grammar": round(grammar_score / 100 * request.max_score *
self.DIMENSION_WEIGHTS["grammar"], 1),
            "content": round(content_score / 100 * request.max_score *
self.DIMENSION_WEIGHTS["content"], 1),
            "rhetoric": round(rhetoric_score / 100 * request.max_score *
self.DIMENSION_WEIGHTS["rhetoric"], 1),
        },
        "character_count": char_stats,
        "overall_comment": overall_comment,
        "structure_analysis": struct_comments,
        "content_analysis": content_comments,
        "rhetoric_analysis": rhetoric_comments,
    }

```

```

        "grammar_errors": [e.dict() for e in grammar_errors] if
request.enable_suggestions else [],
        "inference_time_ms": round(elapsed, 2)
    }
    return result

def _generate_overall_comment(self, score: float, max_score: int,
                              struct_comments: List, content_comments: List,
                              rhetoric_comments: List) -> str:
    """生成综合评语"""
    ratio = score / max_score
    if ratio >= 0.9:
        prefix = "优秀! "
    elif ratio >= 0.75:
        prefix = "良好。"
    elif ratio >= 0.6:
        prefix = "中等。"
    else:
        prefix = "需要加强。"

    suggestions = []
    if struct_comments:
        suggestions.append(struct_comments[0])
    if content_comments:
        suggestions.append(content_comments[0])
    if rhetoric_comments:
        suggestions.append(rhetoric_comments[0])

    return f"{prefix}{' '.join(suggestions[:3])}"

# ===== API路由定义 =====

router = APIRouter(prefix="/api/v1", tags=["作文批改"])
_scoring_engine = EssayScoringEngine()

@router.post("/essay/review")
async def review_essay(request: EssayReviewRequest):
    """
    AI作文评分与批改接口
    POST /api/v1/essay/review
    输入作文OCR识别文本，返回综合评分、各维度分析和修改建议
    """
    try:
        result = _scoring_engine.review_essay(request)

        # 审计日志记录
        logger.info(
            f"作文批改完成: score={result['total_score']}/{request.max_score}, "
            f"student={request.student_id}, assignment={request.assignment_id}, "
            f"chars={result['character_count']['chinese']], time=
{result['inference_time_ms']}ms"
        )
        return {"code": 200, "msg": "success", "data": result}
    except Exception as e:

```

```
logger.error(f"作文批改异常: {str(e)}")
raise HTTPException(status_code=500, detail=f"作文批改服务异常: {str(e)}")
```

api/math_api.py

```
# -*- coding: utf-8 -*-
"""
自然写手写识别与AI分析引擎软件 V1.0

数学列式与公式识别接口
支持四则运算、方程式、几何图形公式等数学内容识别
"""

from fastapi import APIRouter, HTTPException
from pydantic import BaseModel, Field
from typing import List, Optional, Dict, Any
import numpy as np
import logging
import time
import uuid
import re

logger = logging.getLogger("writech-ai-engine.math")
router = APIRouter()

class MathStrokePoint(BaseModel):
    """数学笔迹坐标点"""
    x: int = Field(..., ge=0, le=65535)
    y: int = Field(..., ge=0, le=65535)
    pressure: int = Field(0, ge=0, le=255)
    timestamp: int = Field(...)
    pen_up: bool = Field(False)

class MathRecognizeRequest(BaseModel):
    """数学识别请求"""
    strokes: List[List[MathStrokePoint]] = Field(..., description="笔迹数据")
    math_type: str = Field("arithmetic", description="数学类型: arithmetic/equation/geometry")
    grade_level: int = Field(3, ge=1, le=6, description="年级(1-6)")

class MathStep(BaseModel):
    """计算步骤"""
    step_no: int = Field(..., description="步骤序号")
    expression: str = Field(..., description="表达式")
    result: Optional[str] = Field(None, description="计算结果")
    is_correct: bool = Field(True, description="是否正确")
    error_type: Optional[str] = Field(None, description="错误类型")
    error_detail: Optional[str] = Field(None, description="错误详情")

class MathRecognizeResult(BaseModel):
```

```

"""数学识别结果"""
latex: str = Field(..., description="LaTeX表达式")
result: Optional[str] = Field(None, description="计算结果")
is_correct: bool = Field(True, description="答案是否正确")
steps: List[MathStep] = Field(default=[], description="计算步骤")
confidence: float = Field(..., description="识别置信度")

class MathEngine:
    """
    数学列式识别引擎

    支持识别类型：
    - 四则运算（加减乘除、连续运算）
    - 竖式计算（加法竖式、减法竖式、乘法竖式、除法竖式）
    - 比较大小（>、<、=）
    - 分数运算
    - 简单方程（一元一次方程）

    推理流程：
    笔迹 → 图像渲染 → 符号分割 → 符号识别 → 结构分析 → 表达式重建 → 计算验证
    """

    def __init__(self):
        self.model = None
        self.is_loaded = False
        # 支持的数学符号集合
        self.symbol_set = set("0123456789+-x÷==><()/.%")
        logger.info("数学识别引擎初始化完成")

    def load_model(self, model_path: str):
        """加载数学识别模型"""
        logger.info(f"加载数学识别模型: {model_path}")
        self.is_loaded = True
        logger.info("数学识别模型加载完成")

    def recognize(self, strokes: List[List[MathStrokePoint]],
                  math_type: str = "arithmetic",
                  grade_level: int = 3) -> MathRecognizeResult:
        """
        数学列式识别主流程
        """
        start_time = time.time()

        # 步骤1: 笔迹预处理与图像渲染
        image = self._preprocess_strokes(strokes)

        # 步骤2: 数学符号分割
        segments = self._segment_symbols(image)

        # 步骤3: 符号识别 (CNN分类器)
        symbols = self._recognize_symbols(segments)

        # 步骤4: 结构分析 (确定运算符和操作数的空间关系)
        structure = self._analyze_structure(symbols, math_type)

        # 步骤5: 表达式重建 (生成LaTeX和数学表达式)

```

```

latex_expr, math_expr = self._reconstruct_expression(structure)

# 步骤6: 计算验证
result, is_correct, steps = self._verify_calculation(math_expr, grade_level)

inference_time = time.time() - start_time
logger.info(f"数学识别完成: latex={latex_expr}, correct={is_correct}, "
            f"time={inference_time:.4f}s")

return MathRecognizeResult(
    latex=latex_expr,
    result=result,
    is_correct=is_correct,
    steps=steps,
    confidence=0.92
)

def _preprocess_strokes(self, strokes: List[List[MathStrokePoint]]) -> np.ndarray:
    """笔迹预处理: 坐标归一化 → 去噪 → 渲染为灰度图"""
    canvas_h, canvas_w = 64, 512
    canvas = np.zeros((canvas_h, canvas_w), dtype=np.float32)

    all_x = [p.x for s in strokes for p in s]
    all_y = [p.y for s in strokes for p in s]
    if not all_x:
        return canvas

    min_x, max_x = min(all_x), max(all_x)
    min_y, max_y = min(all_y), max(all_y)
    w = max(max_x - min_x, 1)
    h = max(max_y - min_y, 1)
    scale = min((canvas_w - 10) / w, (canvas_h - 10) / h)

    for stroke in strokes:
        for i in range(1, len(stroke)):
            x1 = int((stroke[i-1].x - min_x) * scale + 5)
            y1 = int((stroke[i-1].y - min_y) * scale + 5)
            x2 = int((stroke[i].x - min_x) * scale + 5)
            y2 = int((stroke[i].y - min_y) * scale + 5)
            x1, x2 = np.clip([x1, x2], 0, canvas_w - 1)
            y1, y2 = np.clip([y1, y2], 0, canvas_h - 1)
            canvas[y1:y2+1, x1:x2+1] = 1.0

    return canvas

def _segment_symbols(self, image: np.ndarray) -> List[Dict]:
    """
    数学符号分割
    基于连通域分析将图像分割为独立的符号区域
    """
    segments = []
    # 使用连通域分析进行符号分割
    # labels = cv2.connectedComponents(image)
    # 模拟分割结果
    segments = [
        {"bbox": [10, 5, 40, 55], "image": image[5:55, 10:40]},
        {"bbox": [45, 20, 65, 45], "image": image[20:45, 45:65]},
    ]

```



```

        {"bbox": [70, 5, 100, 55], "image": image[5:55, 70:100]},
        {"bbox": [105, 20, 125, 45], "image": image[20:45, 105:125]},
        {"bbox": [130, 5, 160, 55], "image": image[5:55, 130:160]},
    ]
    return segments

def _recognize_symbols(self, segments: List[Dict]) -> List[Dict]:
    """
    符号识别 (CNN分类器)
    对每个分割区域进行数字/运算符分类
    """
    symbols = []
    # 模拟识别结果
    mock_symbols = ["1", "2", "+", "3", "=", "1", "5"]
    for i, seg in enumerate(segments):
        if i < len(mock_symbols):
            symbols.append({
                "symbol": mock_symbols[i],
                "bbox": seg["bbox"],
                "confidence": 0.95 - i * 0.01
            })
    return symbols

def _analyze_structure(self, symbols: List[Dict], math_type: str) -> Dict:
    """
    结构分析
    根据符号的空间位置关系确定数学表达式的结构
    处理竖式、分数线、括号等特殊结构
    """
    # 按x坐标排序 (从左到右阅读顺序)
    sorted_symbols = sorted(symbols, key=lambda s: s["bbox"][0])

    if math_type == "arithmetic":
        return {"type": "linear", "symbols": sorted_symbols}
    elif math_type == "equation":
        return {"type": "equation", "symbols": sorted_symbols}
    else:
        return {"type": "unknown", "symbols": sorted_symbols}

def _reconstruct_expression(self, structure: Dict) -> tuple:
    """
    表达式重建
    从结构化符号序列生成LaTeX表达式和可计算表达式
    """
    symbols = structure.get("symbols", [])
    chars = [s["symbol"] for s in symbols]
    text = "".join(chars)

    # 生成LaTeX
    latex = text.replace("x", "\\times ").replace("÷", "\\div ")

    # 生成可计算表达式
    math_expr = text.replace("x", "*").replace("÷", "/")

    return latex, math_expr

def _verify_calculation(self, math_expr: str, grade_level: int) -> tuple:

```

```

"""
计算验证
解析数学表达式，计算正确答案，对比学生答案
"""

steps = []

# 尝试分离等号两侧
if "=" in math_expr:
    parts = math_expr.split("=")
    if len(parts) == 2:
        left = parts[0].strip()
        right = parts[1].strip()

        try:
            left_val = self._safe_eval(left)
            right_val = self._safe_eval(right)

            steps.append(MathStep(
                step_no=1,
                expression=left,
                result=str(left_val),
                is_correct=True
            ))

            is_correct = abs(left_val - right_val) < 1e-9
            steps.append(MathStep(
                step_no=2,
                expression=f"{left} = {right}",
                result=str(right_val),
                is_correct=is_correct,
                error_type=None if is_correct else "calculation",
                error_detail=None if is_correct else f"正确答案应为{left_val}"
            ))

            return str(left_val), is_correct, steps

        except Exception:
            pass

    return None, True, steps

def _safe_eval(self, expr: str) -> float:
    """安全计算表达式（仅允许数字和基本运算符）"""
    allowed_chars = set("0123456789.+*/() ")
    if not all(c in allowed_chars for c in expr):
        raise ValueError(f"不安全的表达式: {expr}")
    return eval(expr) # 仅在安全校验后使用

# 全局数学引擎实例
math_engine = MathEngine()

@router.post("/recognize")
async def recognize_math(request: MathRecognizeRequest):
    """
    数学列式/公式识别接口
    """

```

```

POST /api/v1/math/recognize
"""
if not request.strokes:
    raise HTTPException(status_code=400, detail="笔迹数据不能为空")

result = math_engine.recognize(
    strokes=request.strokes,
    math_type=request.math_type,
    grade_level=request.grade_level
)

return {
    "code": 200,
    "msg": "success",
    "data": {
        "request_id": str(uuid.uuid4()),
        "result": result.dict()
    }
}

```

api/ocr_api.py

```

# -*- coding: utf-8 -*-
"""
自然写手写识别与AI分析引擎软件 V1.0

OCR识别接口模块
提供中英文手写文字OCR识别服务，基于PaddleOCR推理管道
"""

from fastapi import APIRouter, HTTPException
from pydantic import BaseModel, Field
from typing import List, Optional, Dict, Any
import numpy as np
import logging
import time
import uuid

logger = logging.getLogger("writech-ai-engine.ocr")
router = APIRouter()

# ===== 请求/响应模型定义 =====

class StrokePoint(BaseModel):
    """笔迹坐标点"""
    x: int = Field(..., ge=0, le=65535, description="X坐标")
    y: int = Field(..., ge=0, le=65535, description="Y坐标")
    pressure: int = Field(0, ge=0, le=255, description="压力值")
    timestamp: int = Field(..., description="时间戳(毫秒)")
    pen_up: bool = Field(False, description="抬笔标记")

class OCRRequest(BaseModel):

```

```

"""OCR识别请求"""
strokes: List[List[StrokePoint]] = Field(..., description="笔迹数据(按笔画分组)")
page_id: Optional[str] = Field(None, description="点阵码页面ID")
pen_id: Optional[str] = Field(None, description="笔设备ID")
language: str = Field("zh", description="识别语言: zh/en/mixed")
recognition_mode: str = Field("line", description="识别模式: char/word/line/page")

class CharDetail(BaseModel):
    """单字识别详情"""
    char: str = Field(..., description="识别的字符")
    confidence: float = Field(..., description="置信度(0-1)")
    bbox: List[int] = Field(..., description="包围框[x1,y1,x2,y2]")
    stroke_indices: List[int] = Field(default=[], description="对应的笔画索引")

class OCRResult(BaseModel):
    """OCR识别结果"""
    text: str = Field(..., description="识别文本")
    confidence: float = Field(..., description="整体置信度(0-1)")
    bbox: List[int] = Field(default=[], description="文本区域包围框")
    char_details: List[CharDetail] = Field(default=[], description="逐字详情")

class OCRResponse(BaseModel):
    """OCR识别响应"""
    code: int = 200
    msg: str = "success"
    data: Optional[Dict[str, Any]] = None

# ===== OCR 推理引擎 =====

class OCREngine:
    """
    PaddleOCR 推理引擎

    推理管道流程:
    笔迹坐标 → 预处理(归一化/去噪) → 笔画分割
    → 模型推理(OCR) → 后处理(置信度过滤/结果合并) → 结果输出

    支持的识别模式:
    - char: 单字识别(逐字识别, 返回每个字的详情)
    - word: 词组识别(按词分割识别)
    - line: 行识别(按行识别, 默认模式)
    - page: 整页识别(全页文字识别)
    """

    def __init__(self):
        """初始化OCR推理引擎"""
        self.model = None
        self.model_version = "1.0.0"
        self.is_loaded = False
        # 模型输入图像尺寸
        self.input_height = 48
        self.input_width = 320
        # 置信度阈值

```

```

self.confidence_threshold = 0.5
logger.info("OCR引擎初始化完成")

def load_model(self, model_path: str):
    """
    加载PaddleOCR模型
    模型文件AES-256加密存储，推理时内存解密加载
    """
    logger.info(f"加载OCR模型: {model_path}")
    # 解密模型文件
    # decrypted_model = self._decrypt_model(model_path)
    # self.model = paddle.jit.load(decrypted_model)
    self.is_loaded = True
    logger.info("OCR模型加载完成")

def preprocess_strokes(self, strokes: List[List[StrokePoint]]) -> np.ndarray:
    """
    笔迹预处理管道

    步骤：
    1. 坐标归一化（映射到标准画布尺寸）
    2. 去噪处理（滤除抖动和异常点）
    3. 笔迹渲染为灰度图像
    4. 图像尺寸归一化（resize到模型输入尺寸）
    """
    # 计算所有点的边界框
    all_points = []
    for stroke in strokes:
        for point in stroke:
            all_points.append((point.x, point.y))

    if not all_points:
        return np.zeros((1, self.input_height, self.input_width), dtype=np.float32)

    xs = [p[0] for p in all_points]
    ys = [p[1] for p in all_points]
    min_x, max_x = min(xs), max(xs)
    min_y, max_y = min(ys), max(ys)

    # 计算缩放比例（保持宽高比）
    width = max(max_x - min_x, 1)
    height = max(max_y - min_y, 1)
    scale = min(self.input_width / width, self.input_height / height) * 0.9

    # 创建渲染画布
    canvas = np.zeros((self.input_height, self.input_width), dtype=np.float32)

    # 渲染笔迹到画布
    for stroke in strokes:
        for i in range(1, len(stroke)):
            x1 = int((stroke[i - 1].x - min_x) * scale)
            y1 = int((stroke[i - 1].y - min_y) * scale)
            x2 = int((stroke[i].x - min_x) * scale)
            y2 = int((stroke[i].y - min_y) * scale)
            # 使用Bresenham算法画线
            self._draw_line(canvas, x1, y1, x2, y2,
                            thickness=max(1, stroke[i].pressure // 85))

```

```

# 归一化到[0, 1]
if canvas.max() > 0:
    canvas = canvas / canvas.max()

return canvas.reshape(1, self.input_height, self.input_width)

def recognize(self, strokes: List[List[StrokePoint]],
              mode: str = "line") -> List[OCRResult]:
    """
    执行OCR识别

    @param strokes: 笔迹数据（按笔画分组）
    @param mode: 识别模式（char/word/line/page）
    @return: 识别结果列表
    """
    start_time = time.time()

    # 预处理
    image = self.preprocess_strokes(strokes)

    # 模型推理
    # predictions = self.model(image)
    # 模拟推理结果
    predictions = self._mock_inference(image, mode)

    # 后处理（置信度过滤、结果合并）
    results = self._postprocess(predictions, mode)

    inference_time = time.time() - start_time
    logger.info(f"OCR识别完成, mode={mode}, time={inference_time:.4f}s, "
               f"results={len(results)}")

    return results

def _postprocess(self, predictions: Dict, mode: str) -> List[OCRResult]:
    """
    后处理：置信度过滤 + 结果合并

    - 过滤低于阈值的识别结果
    - 相邻字符合并为词/行
    - 生成逐字详情信息
    """
    results = []

    if mode == "char":
        # 逐字模式：返回每个字符的独立结果
        for char_pred in predictions.get("chars", []):
            if char_pred["confidence"] >= self.confidence_threshold:
                result = OCRResult(
                    text=char_pred["char"],
                    confidence=char_pred["confidence"],
                    bbox=char_pred["bbox"],
                    char_details=[CharDetail(
                        char=char_pred["char"],
                        confidence=char_pred["confidence"],
                        bbox=char_pred["bbox"],

```

```

        stroke_indices=char_pred.get("stroke_indices", [])
    )
    results.append(result)

elif mode in ("line", "page"):
    # 行/页模式: 合并字符为文本行
    for line_pred in predictions.get("lines", []):
        if line_pred["confidence"] >= self.confidence_threshold:
            char_details = [
                CharDetail(
                    char=cd["char"],
                    confidence=cd["confidence"],
                    bbox=cd["bbox"],
                    stroke_indices=cd.get("stroke_indices", [])
                )
                for cd in line_pred.get("char_details", [])
            ]
            result = OCRResult(
                text=line_pred["text"],
                confidence=line_pred["confidence"],
                bbox=line_pred["bbox"],
                char_details=char_details
            )
            results.append(result)

return results

def _draw_line(self, canvas: np.ndarray, x1: int, y1: int,
               x2: int, y2: int, thickness: int = 1):
    """Bresenham直线绘制算法"""
    h, w = canvas.shape
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    sx = 1 if x1 < x2 else -1
    sy = 1 if y1 < y2 else -1
    err = dx - dy

    while True:
        # 绘制像素 (带粗细)
        for tx in range(-thickness, thickness + 1):
            for ty in range(-thickness, thickness + 1):
                px, py = x1 + tx, y1 + ty
                if 0 <= px < w and 0 <= py < h:
                    canvas[py][px] = 1.0

        if x1 == x2 and y1 == y2:
            break
        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x1 += sx
        if e2 < dx:
            err += dx
            y1 += sy

def _mock_inference(self, image: np.ndarray, mode: str) -> Dict:

```

```

        """模拟推理结果（用于示例）"""
        return {
            "lines": [{
                "text": "示例文字",
                "confidence": 0.95,
                "bbox": [10, 10, 200, 48],
                "char_details": [
                    {"char": "示", "confidence": 0.96, "bbox": [10, 10, 50, 48]},
                    {"char": "例", "confidence": 0.94, "bbox": [50, 10, 100, 48]},
                    {"char": "文", "confidence": 0.97, "bbox": [100, 10, 150, 48]},
                    {"char": "字", "confidence": 0.93, "bbox": [150, 10, 200, 48]}
                ]
            }],
            "chars": []
        }

def _decrypt_model(self, model_path: str) -> str:
    """AES-256解密模型文件"""
    # 使用预配置的密钥解密模型文件
    # key = settings.model_encryption_key
    # cipher = AES.new(key, AES.MODE_CBC, iv)
    return model_path

# 全局OCR引擎实例
ocr_engine = OCREngine()

# ===== API 路由 =====

@router.post("/recognize", response_model=OCRResponse)
async def recognize_text(request: OCRRequest):
    """
    手写文字OCR识别接口
    POST /api/v1/ocr/recognize

    接收笔迹坐标数据，返回识别文本及逐字详情
    支持中文、英文及中英混合识别
    """
    # 输入校验
    if not request.strokes:
        raise HTTPException(status_code=400, detail="笔迹数据不能为空")

    total_points = sum(len(stroke) for stroke in request.strokes)
    if total_points > 50000:
        raise HTTPException(status_code=400, detail="笔迹点数过多，最大支持50000点")

    # 执行OCR识别
    results = ocr_engine.recognize(
        strokes=request.strokes,
        mode=request.recognition_mode
    )

    # 构建响应
    return OCRResponse(
        code=200,
        msg="success",

```



```

        data={
            "request_id": str(uuid.uuid4()),
            "language": request.language,
            "mode": request.recognition_mode,
            "results": [r.dict() for r in results],
            "total_chars": sum(len(r.text) for r in results)
        }
    )

@router.post("/batch-recognize")
async def batch_recognize(requests: List[OCRRequest]):
    """
    批量OCR识别接口
    一次请求识别多组笔迹数据
    """
    results = []
    for req in requests:
        result = ocr_engine.recognize(
            strokes=req.strokes,
            mode=req.recognition_mode
        )
        results.append({
            "page_id": req.page_id,
            "results": [r.dict() for r in result]
        })

    return {
        "code": 200,
        "msg": "success",
        "data": {
            "batch_size": len(requests),
            "results": results
        }
    }
}

```

api/stroke_order_api.py

```

# 自然写手写识别与AI分析引擎软件 V1.0
# 笔顺评分接口模块 - 中文汉字笔顺识别与评分服务

"""
笔顺评分API接口
提供汉字笔顺正确性评估、书写质量评分、笔画拆分分析等功能
基于深度学习笔顺分析模型，支持GB2312常用汉字笔顺评分
"""

import time
import logging
import hashlib
import numpy as np
from typing import List, Dict, Optional, Tuple
from dataclasses import dataclass, field
from enum import Enum

```

```

from fastapi import APIRouter, HTTPException, Depends, Request
from pydantic import BaseModel, Field, validator

logger = logging.getLogger(__name__)

# ===== 数据模型定义 =====

class StrokePointInput(BaseModel):
    """笔迹坐标点输入"""
    x: float = Field(..., description="X坐标")
    y: float = Field(..., description="Y坐标")
    pressure: float = Field(0.5, ge=0.0, le=1.0, description="压力值")
    timestamp: int = Field(..., description="时间戳(毫秒)")

class StrokeOrderRequest(BaseModel):
    """笔顺评分请求"""
    character: str = Field(..., min_length=1, max_length=1, description="目标汉字")
    strokes: List[List[StrokePointInput]] = Field(..., description="用户书写的笔画列表")
    pen_id: Optional[str] = Field(None, description="点阵笔设备ID")
    student_id: Optional[str] = Field(None, description="学生ID")
    difficulty_level: int = Field(1, ge=1, le=3, description="评分难度等级1-3")

    @validator('character')
    def validate_chinese_char(cls, v):
        """校验是否为中文汉字"""
        if not '\u4e00' <= v <= '\u9fff':
            raise ValueError('仅支持中文汉字笔顺评分')
        return v

class WritingQualityRequest(BaseModel):
    """书写质量评测请求"""
    strokes: List[List[StrokePointInput]] = Field(..., description="笔迹数据")
    reference_char: Optional[str] = Field(None, description="参考字符(可选)")
    eval_dimensions: List[str] = Field(
        default=["structure", "spacing", "normative", "aesthetics"],
        description="评测维度"
    )

class StrokeDirection(str, Enum):
    """笔画方向枚举"""
    HORIZONTAL = "horizontal" # 横
    VERTICAL = "vertical" # 竖
    LEFT_FALLING = "left_falling" # 撇
    RIGHT_FALLING = "right_falling" # 捺
    DOT = "dot" # 点
    TURNING = "turning" # 折
    HOOK = "hook" # 钩
    RISING = "rising" # 提

@dataclass
class StrokeFeature:
    """单个笔画特征数据"""
    direction: StrokeDirection # 笔画方向

```

```

start_point: Tuple[float, float]    # 起始坐标
end_point: Tuple[float, float]      # 结束坐标
length: float                       # 笔画长度
avg_pressure: float                 # 平均压力
curvature: float                    # 弯曲度
speed: float                        # 书写速度

# ===== 标准笔顺数据库 =====

class StrokeOrderDatabase:
    """
    标准笔顺数据库
    存储GB2312常用汉字的标准笔顺信息，用于笔顺正确性比对
    数据来源：国家语委《现代汉语通用字笔顺规范》
    """

    def __init__(self):
        # 标准笔顺字典：字符 -> 笔画方向序列
        self._standard_orders: Dict[str, List[StrokeDirection]] = {}
        # 笔画数字典：字符 -> 标准笔画数
        self._stroke_counts: Dict[str, int] = {}
        # 加载常用汉字笔顺数据
        self._load_standard_data()

    def _load_standard_data(self):
        """加载标准笔顺数据（示例部分常用字）"""
        # 一年级常用汉字笔顺数据
        standard_data = {
            "一": ([StrokeDirection.HORIZONTAL], 1),
            "二": ([StrokeDirection.HORIZONTAL, StrokeDirection.HORIZONTAL], 2),
            "三": ([StrokeDirection.HORIZONTAL, StrokeDirection.HORIZONTAL,
StrokeDirection.HORIZONTAL], 3),
            "十": ([StrokeDirection.HORIZONTAL, StrokeDirection.VERTICAL], 2),
            "大": ([StrokeDirection.HORIZONTAL, StrokeDirection.LEFT_FALLING,
StrokeDirection.RIGHT_FALLING], 3),
            "人": ([StrokeDirection.LEFT_FALLING, StrokeDirection.RIGHT_FALLING], 2),
            "口": ([StrokeDirection.VERTICAL, StrokeDirection.TURNING,
StrokeDirection.HORIZONTAL], 3),
            "日": ([StrokeDirection.VERTICAL, StrokeDirection.TURNING,
StrokeDirection.HORIZONTAL, StrokeDirection.HORIZONTAL], 4),
            "月": ([StrokeDirection.LEFT_FALLING, StrokeDirection.TURNING,
StrokeDirection.HORIZONTAL, StrokeDirection.HORIZONTAL], 4),
            "水": ([StrokeDirection.VERTICAL, StrokeDirection.TURNING,
StrokeDirection.LEFT_FALLING, StrokeDirection.RIGHT_FALLING], 4),
        }
        for char, (order, count) in standard_data.items():
            self._standard_orders[char] = order
            self._stroke_counts[char] = count
        logger.info(f"标准笔顺数据库加载完成，共 {len(self._standard_orders)} 个汉字")

    def get_standard_order(self, char: str) -> Optional[List[StrokeDirection]]:
        """获取汉字标准笔顺"""
        return self._standard_orders.get(char)

    def get_stroke_count(self, char: str) -> Optional[int]:
        """获取汉字标准笔画数"""

```

```

        return self._stroke_counts.get(char)

# ===== 笔顺分析引擎 =====

class StrokeOrderAnalyzer:
    """
    笔顺分析引擎
    通过笔迹坐标数据分析每一笔的方向、顺序，并与标准笔顺进行比对评分
    评分维度：笔顺正确性、笔画数、书写规范性
    """

    def __init__(self):
        self._database = StrokeOrderDatabase()
        self._direction_model = None # 笔画方向分类模型 (CNN)
        logger.info("笔顺分析引擎初始化完成")

    def _extract_stroke_feature(self, points: List[StrokePointInput]) -> StrokeFeature:
        """
        提取单个笔画的特征向量
        包括方向、长度、弯曲度、书写速度等
        """
        if len(points) < 2:
            return StrokeFeature(
                direction=StrokeDirection.DOT,
                start_point=(points[0].x, points[0].y),
                end_point=(points[0].x, points[0].y),
                length=0.0, avg_pressure=points[0].pressure,
                curvature=0.0, speed=0.0
            )

        # 计算起止点
        start = (points[0].x, points[0].y)
        end = (points[-1].x, points[-1].y)

        # 计算笔画总长度（累加相邻点欧氏距离）
        total_length = 0.0
        for i in range(1, len(points)):
            dx = points[i].x - points[i-1].x
            dy = points[i].y - points[i-1].y
            total_length += np.sqrt(dx*dx + dy*dy)

        # 计算平均压力值
        avg_pressure = np.mean([p.pressure for p in points])

        # 计算书写速度（总长度/时间差）
        time_diff = max(points[-1].timestamp - points[0].timestamp, 1)
        speed = total_length / time_diff * 1000 # 像素/秒

        # 计算弯曲度（实际路径长度 / 起止点直线距离）
        direct_dist = np.sqrt((end[0]-start[0])**2 + (end[1]-start[1])**2)
        curvature = total_length / max(direct_dist, 1.0)

        # 判定笔画方向
        direction = self._classify_direction(start, end, curvature)

        return StrokeFeature(

```

```

        direction=direction, start_point=start, end_point=end,
        length=total_length, avg_pressure=avg_pressure,
        curvature=curvature, speed=speed
    )

    def _classify_direction(self, start: Tuple, end: Tuple, curvature: float) ->
    StrokeDirection:
        """
        基于起止点坐标和弯曲度分类笔画方向
        使用角度阈值和弯曲度综合判定
        """
        dx = end[0] - start[0]
        dy = end[1] - start[1]
        distance = np.sqrt(dx*dx + dy*dy)

        # 极短笔画判定为点
        if distance < 5.0:
            return StrokeDirection.DOT

        # 计算角度（弧度转角度，0度为正右方，顺时针为正）
        angle = np.degrees(np.arctan2(dy, dx))

        # 弯曲度高的笔画判定为折或钩
        if curvature > 1.8:
            return StrokeDirection.TURNING if dy > 0 else StrokeDirection.HOOK

        # 根据角度范围判定笔画方向
        if -20 <= angle <= 20:
            return StrokeDirection.HORIZONTAL # 横：接近水平向右
        elif 70 <= angle <= 110:
            return StrokeDirection.VERTICAL # 竖：接近垂直向下
        elif 120 <= angle <= 170:
            return StrokeDirection.LEFT_FALLING # 撇：左下方向
        elif 20 < angle < 70:
            return StrokeDirection.RIGHT_FALLING # 捺：右下方向
        elif -70 <= angle < -20:
            return StrokeDirection.RISING # 提：右上方向
        else:
            return StrokeDirection.LEFT_FALLING # 默认归为撇

    def evaluate_stroke_order(self, char: str, strokes: List[List[StrokePointInput]],
                             difficulty: int = 1) -> Dict:
        """
        评估笔顺正确性
        将用户书写的每一笔与标准笔顺逐一比对，计算匹配分数
        """
        start_time = time.time()

        # 获取标准笔顺
        standard_order = self._database.get_standard_order(char)
        standard_count = self._database.get_stroke_count(char)

        # 提取用户每一笔的特征
        user_features = [self._extract_stroke_feature(s) for s in strokes]
        user_directions = [f.direction for f in user_features]

        # 笔画数评分（满分100）

```

```

count_score = 100.0
if standard_count:
    count_diff = abs(len(strokes) - standard_count)
    count_score = max(0, 100 - count_diff * 25)

# 笔顺正确性评分 (逐笔比对方向)
order_score = 100.0
errors = []
if standard_order:
    match_count = 0
    compare_len = min(len(user_directions), len(standard_order))
    for i in range(compare_len):
        if user_directions[i] == standard_order[i]:
            match_count += 1
        else:
            errors.append({
                "stroke_index": i + 1,
                "expected": standard_order[i].value,
                "actual": user_directions[i].value,
                "message": f"第{i+1}笔方向错误: 应为{standard_order[i].value}, 实际
为{user_directions[i].value}"
            })
    order_score = (match_count / max(len(standard_order), 1)) * 100

# 根据难度等级调整评分权重
weight_order = 0.5 + difficulty * 0.1    # 难度越高, 笔顺正确性权重越大
weight_count = 1.0 - weight_order

total_score = order_score * weight_order + count_score * weight_count
elapsed = (time.time() - start_time) * 1000

return {
    "character": char,
    "total_score": round(total_score, 1),
    "order_score": round(order_score, 1),
    "count_score": round(count_score, 1),
    "user_stroke_count": len(strokes),
    "standard_stroke_count": standard_count,
    "stroke_order": [d.value for d in user_directions],
    "correct_order": [d.value for d in standard_order] if standard_order else
[],
    "errors": errors,
    "inference_time_ms": round(elapsed, 2)
}

# ===== 书写质量评测引擎 =====

class WritingQualityEngine:
    """
    书写质量评测引擎
    从结构均衡性、笔画间距、规范性、美观度四个维度评估书写质量
    """

    def evaluate(self, strokes: List[List[StrokePointInput]],
                 dimensions: List[str]) -> Dict:
        """执行书写质量评测"""

```

```

scores = {}

# 提取全部坐标点用于整体分析
all_points = []
for stroke in strokes:
    all_points.extend([(p.x, p.y, p.pressure) for p in stroke])

if not all_points:
    return {"total_score": 0, "dimensions": {}}

xs = [p[0] for p in all_points]
ys = [p[1] for p in all_points]

# 计算书写区域边界框
bbox_width = max(xs) - min(xs)
bbox_height = max(ys) - min(ys)

if "structure" in dimensions:
    # 结构均衡性: 分析重心位置与对称性
    center_x = np.mean(xs)
    center_y = np.mean(ys)
    expected_center_x = min(xs) + bbox_width / 2
    expected_center_y = min(ys) + bbox_height / 2
    offset = np.sqrt((center_x - expected_center_x)**2 + (center_y -
expected_center_y)**2)
    max_offset = np.sqrt(bbox_width**2 + bbox_height**2) / 4
    scores["structure"] = round(max(0, 100 - (offset / max(max_offset, 1)) *
60), 1)

if "spacing" in dimensions:
    # 笔画间距均匀性: 分析相邻笔画起始点间距的标准差
    if len(strokes) > 1:
        start_points = [(s[0].x, s[0].y) for s in strokes if s]
        gaps = []
        for i in range(1, len(start_points)):
            gap = np.sqrt((start_points[i][0]-start_points[i-1][0])**2 +
                (start_points[i][1]-start_points[i-1][1])**2)
            gaps.append(gap)
        gap_std = np.std(gaps) if gaps else 0
        gap_mean = np.mean(gaps) if gaps else 1
        cv = gap_std / max(gap_mean, 1) # 变异系数
        scores["spacing"] = round(max(0, 100 - cv * 80), 1)
    else:
        scores["spacing"] = 80.0

if "normative" in dimensions:
    # 规范性: 分析笔画弯曲度和压力稳定性
    pressures = [p[2] for p in all_points]
    pressure_std = np.std(pressures) if pressures else 0
    scores["normative"] = round(max(0, 100 - pressure_std * 200), 1)

if "aesthetics" in dimensions:
    # 美观度: 综合笔画流畅度和整体比例
    aspect_ratio = bbox_width / max(bbox_height, 1)
    ratio_score = max(0, 100 - abs(aspect_ratio - 1.0) * 50) # 接近正方形得分高
    scores["aesthetics"] = round(ratio_score, 1)

```

```

        total = np.mean(list(scores.values())) if scores else 0
        return {"total_score": round(total, 1), "dimensions": scores}

# ===== API路由定义 =====

router = APIRouter(prefix="/api/v1", tags=["笔顺评分"])
_analyzer = StrokeOrderAnalyzer()
_quality_engine = WritingQualityEngine()

@router.post("/stroke-order/evaluate")
async def evaluate_stroke_order(request: StrokeOrderRequest):
    """
    笔顺正确性评分接口
    POST /api/v1/stroke-order/evaluate
    输入汉字和用户书写笔画数据，返回笔顺正确性评分和错误详情
    """
    try:
        result = _analyzer.evaluate_stroke_order(
            char=request.character,
            strokes=request.strokes,
            difficulty=request.difficulty_level
        )
        # 记录审计日志（安全设计：所有识别请求记录调用方、时间、模型版本）
        logger.info(
            f"笔顺评分完成: char={request.character}, "
            f"score={result['total_score']}, pen={request.pen_id}, "
            f"student={request.student_id}, time={result['inference_time_ms']}ms"
        )
        return {"code": 200, "msg": "success", "data": result}
    except Exception as e:
        logger.error(f"笔顺评分异常: {str(e)}")
        raise HTTPException(status_code=500, detail=f"笔顺评分服务异常: {str(e)}")

@router.post("/writing/quality")
async def evaluate_writing_quality(request: WritingQualityRequest):
    """
    书写质量评测接口
    POST /api/v1/writing/quality
    从结构、间距、规范性、美观度四维度评测书写质量
    """
    try:
        result = _quality_engine.evaluate(
            strokes=request.strokes,
            dimensions=request.eval_dimensions
        )
        logger.info(f"书写质量评测完成: score={result['total_score']}")
        return {"code": 200, "msg": "success", "data": result}
    except Exception as e:
        logger.error(f"书写质量评测异常: {str(e)}")
        raise HTTPException(status_code=500, detail=f"书写质量评测异常: {str(e)}")

```


config/settings.py

```
# 自然写手写识别与AI分析引擎软件 V1.0
# 配置与安全模块 - 全局配置管理与安全策略

"""
全局配置管理
提供AI引擎服务的所有配置项管理，包括：
服务端点、模型路径、GPU配置、安全认证、日志级别等
支持环境变量覆盖和配置热更新
"""

import os
import json
import logging
import hashlib
import hmac
import time
from typing import Dict, List, Optional, Any
from dataclasses import dataclass, field
from pathlib import Path

logger = logging.getLogger(__name__)

# ===== 服务配置 =====

@dataclass
class ServerConfig:
    """HTTP/gRPC服务配置"""
    http_host: str = "0.0.0.0"
    http_port: int = 8000
    grpc_host: str = "0.0.0.0"
    grpc_port: int = 50051
    workers: int = 4 # FastAPI worker数量
    grpc_max_workers: int = 10 # gRPC线程池大小
    max_request_size_mb: int = 10 # 请求体大小限制（防恶意攻击）
    request_timeout_s: int = 30 # 请求超时时间
    cors_origins: List[str] = field(default_factory=lambda: ["*"])
    debug: bool = False

@dataclass
class ModelConfig:
    """模型推理配置"""
    models_dir: str = "/opt/models" # 模型文件根目录
    ocr_model_path: str = "/opt/models/ocr" # OCR模型路径
    math_model_path: str = "/opt/models/math" # 数学识别模型路径
    stroke_model_path: str = "/opt/models/stroke" # 笔顺模型路径
    essay_model_path: str = "/opt/models/essay" # 作文评分模型路径
    max_batch_size: int = 32 # 最大推理批大小
    inference_timeout_ms: int = 5000 # 单次推理超时
    enable_fp16: bool = True # FP16半精度推理
    model_cache_size_gb: float = 4.0 # 模型内存缓存大小
```

```

@dataclass
class GPUConfig:
    """GPU/NPU硬件加速配置"""
    device: str = "cuda" # 推理设备: cuda / cpu / npu
    gpu_ids: List[int] = field(default_factory=lambda: [0]) # 使用的GPU编号
    gpu_memory_fraction: float = 0.8 # GPU显存使用比例上限
    enable_tensorrt: bool = True # 是否启用TensorRT加速
    tensorrt_precision: str = "fp16" # TensorRT精度: fp32/fp16/int8
    triton_url: str = "localhost:8001" # Triton Inference Server地址


@dataclass
class CeleryConfig:
    """Celery任务队列配置"""
    broker_url: str = "redis://localhost:6379/0" # Redis Broker地址
    result_backend: str = "redis://localhost:6379/1" # 结果存储后端
    task_serializer: str = "json"
    result_serializer: str = "json"
    task_default_queue: str = "writech.default"
    task_time_limit: int = 300 # 任务最大执行时间 (秒)
    task_soft_time_limit: int = 240 # 软超时 (触发SoftTimeLimitExceeded)
    worker_concurrency: int = 8 # Worker并发数
    worker_prefetch_multiplier: int = 2 # 预取倍数


@dataclass
class DatabaseConfig:
    """数据库配置"""
    mysql_url: str = "mysql+pymysql://user:password@localhost:3306/writech_ai"
    redis_url: str = "redis://localhost:6379/0"
    mongodb_url: str = "mongodb://localhost:27017/writech_stroke"
    pool_size: int = 20 # 连接池大小
    pool_recycle: int = 3600 # 连接回收时间 (秒)


@dataclass
class LogConfig:
    """日志配置"""
    level: str = "INFO"
    format: str = "%(asctime)s [%(levelname)s] %(name)s: %(message)s"
    log_dir: str = "/var/log/writech-ai"
    max_file_size_mb: int = 100 # 单个日志文件大小上限
    backup_count: int = 10 # 保留日志文件数量
    enable_audit_log: bool = True # 启用审计日志
    audit_log_file: str = "audit.log" # 审计日志文件名


# ===== 安全配置 =====

@dataclass
class SecurityConfig:
    """安全配置"""
    # mTLS双向认证 (安全设计: 内部服务间mTLS双向认证)
    enable_mtls: bool = True
    server_cert_path: str = "/etc/ssl/server.crt"
    server_key_path: str = "/etc/ssl/server.key"
    ca_cert_path: str = "/etc/ssl/ca.crt"

```

```

# 模型文件加密（安全设计：模型文件加密存储，推理时内存解密）
model_encryption_enabled: bool = True
model_encryption_key_env: str = "WRITECH_MODEL_KEY" # 加密密钥从环境变量读取

# 请求校验（安全设计：输入数据格式校验与大小限制）
max_stroke_points: int = 100000 # 单次请求最大坐标点数
max_strokes_per_request: int = 500 # 单次请求最大笔画数
max_text_length: int = 10000 # 作文文本最大长度

# 速率限制
rate_limit_per_minute: int = 600 # 每分钟最大请求数
rate_limit_burst: int = 50 # 突发请求数

# 审计日志（安全设计：所有识别请求记录调用方、时间、模型版本）
enable_audit: bool = True
audit_retention_days: int = 90 # 审计日志保留天数

# ===== mTLS认证管理 =====

class MTLSAuthenticator:
    """
    mTLS双向认证管理器
    验证客户端证书，确保只有授权的内部服务可以调用AI引擎
    """

    def __init__(self, config: SecurityConfig):
        self._config = config
        self._trusted_clients: Dict[str, str] = {} # 授信客户端证书指纹
        logger.info("mTLS认证管理器初始化")

    def load_certificates(self) -> bool:
        """加载服务端证书和CA证书"""
        try:
            cert_path = Path(self._config.server_cert_path)
            key_path = Path(self._config.server_key_path)
            ca_path = Path(self._config.ca_cert_path)

            if not cert_path.exists():
                logger.warning(f"服务端证书不存在: {cert_path}")
                return False

            logger.info("mTLS证书加载完成")
            return True
        except Exception as e:
            logger.error(f"证书加载失败: {str(e)}")
            return False

    def verify_client_cert(self, cert_fingerprint: str) -> bool:
        """验证客户端证书指纹"""
        if not self._config.enable_mtls:
            return True
        is_trusted = cert_fingerprint in self._trusted_clients
        if not is_trusted:
            logger.warning(f"未授信的客户端证书: {cert_fingerprint}")
        return is_trusted

```

```

def register_trusted_client(self, name: str, fingerprint: str):
    """注册授信客户端"""
    self._trusted_clients[fingerprint] = name
    logger.info(f"注册授信客户端: {name}")

# ===== 请求签名校验 =====

class RequestValidator:
    """
    请求签名校验器
    对API请求进行HMAC签名校验, 防止请求篡改和重放攻击
    """

    def __init__(self, secret_key: str = ""):
        self._secret = secret_key or os.environ.get("WRITECH_API_SECRET", "default-secret")
        self._nonce_cache: Dict[str, float] = {} # 随机数缓存 (防重放)
        self._nonce_ttl = 300 # 随机数有效期 (秒)

    def generate_signature(self, payload: str, timestamp: int, nonce: str) -> str:
        """生成请求签名"""
        message = f"{payload}&timestamp={timestamp}&nonce={nonce}"
        return hmac.new(
            self._secret.encode(), message.encode(), hashlib.sha256
        ).hexdigest()

    def verify_signature(self, payload: str, timestamp: int,
                        nonce: str, signature: str) -> bool:
        """
        校验请求签名
        1. 检查时间戳是否在有效窗口内 (防重放)
        2. 检查随机数是否已使用 (防重放)
        3. 验证HMAC签名是否匹配 (防篡改)
        """
        # 时间窗口校验 (±5分钟)
        current_time = int(time.time())
        if abs(current_time - timestamp) > 300:
            logger.warning(f"请求时间戳过期: {timestamp}")
            return False

        # 随机数防重放检查
        if nonce in self._nonce_cache:
            logger.warning(f"重复的请求随机数: {nonce}")
            return False

        # HMAC签名验证
        expected = self.generate_signature(payload, timestamp, nonce)
        is_valid = hmac.compare_digest(expected, signature)

        if is_valid:
            # 缓存随机数
            self._nonce_cache[nonce] = time.time()
            self._cleanup_nonce_cache()

        return is_valid

```

```

def _cleanup_nonce_cache(self):
    """清理过期的随机数缓存"""
    current = time.time()
    expired = [k for k, v in self._nonce_cache.items() if current - v >
self._nonce_ttl]
    for k in expired:
        del self._nonce_cache[k]

# ===== 全局配置管理器 =====

class Settings:
    """
    全局配置管理器（单例）
    从环境变量和配置文件加载配置，支持运行时热更新
    环境变量优先级高于配置文件
    """

    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self):
        if hasattr(self, '_initialized'):
            return
        self._initialized = True

        # 加载各模块配置
        self.server = ServerConfig()
        self.model = ModelConfig()
        self.gpu = GPUConfig()
        self.celery = CeleryConfig()
        self.database = DatabaseConfig()
        self.log = LogConfig()
        self.security = SecurityConfig()

        # 从环境变量覆盖配置
        self._load_from_env()

        # 初始化安全组件
        self.mtls_auth = MTLSAuthenticator(self.security)
        self.request_validator = RequestValidator()

        logger.info("全局配置加载完成")

    def _load_from_env(self):
        """从环境变量加载配置（覆盖默认值）"""
        env_mapping = {
            "WRITECH_HTTP_PORT": ("server", "http_port", int),
            "WRITECH_GRPC_PORT": ("server", "grpc_port", int),
            "WRITECH_WORKERS": ("server", "workers", int),
            "WRITECH_DEBUG": ("server", "debug", lambda x: x.lower() == "true"),
            "WRITECH_MODELS_DIR": ("model", "models_dir", str),

```

```

        "WRITECH_GPU_DEVICE": ("gpu", "device", str),
        "WRITECH_GPU_IDS": ("gpu", "gpu_ids", lambda x: [int(i) for i in
x.split(",")]),
        "WRITECH_REDIS_URL": ("celery", "broker_url", str),
        "WRITECH_MYSQL_URL": ("database", "mysql_url", str),
        "WRITECH_LOG_LEVEL": ("log", "level", str),
        "WRITECH_ENABLE_MTLS": ("security", "enable_mtls", lambda x: x.lower() ==
"true"),
    }

    for env_key, (section, field, converter) in env_mapping.items():
        value = os.environ.get(env_key)
        if value is not None:
            config_obj = getattr(self, section)
            try:
                setattr(config_obj, field, converter(value))
                logger.info(f"环境变量覆盖配置: {env_key} -> {section}.{field}")
            except (ValueError, TypeError) as e:
                logger.warning(f"环境变量转换失败: {env_key}={value}, 错误: {str(e)}")

def load_from_file(self, config_path: str):
    """从JSON配置文件加载配置"""
    try:
        with open(config_path, 'r') as f:
            config_data = json.load(f)
        logger.info(f"配置文件加载完成: {config_path}")

        # 逐section更新配置
        for section_name, section_data in config_data.items():
            if hasattr(self, section_name) and isinstance(section_data, dict):
                config_obj = getattr(self, section_name)
                for key, value in section_data.items():
                    if hasattr(config_obj, key):
                        setattr(config_obj, key, value)

    except FileNotFoundError:
        logger.warning(f"配置文件不存在: {config_path}")
    except json.JSONDecodeError as e:
        logger.error(f"配置文件JSON解析错误: {str(e)}")

def to_dict(self) -> Dict[str, Any]:
    """将所有配置导出为字典（隐藏敏感信息）"""
    result = {}
    for section in ['server', 'model', 'gpu', 'celery', 'log']:
        config_obj = getattr(self, section)
        section_dict = {}
        for key in vars(config_obj):
            value = getattr(config_obj, key)
            # 隐藏密码和密钥类字段
            if any(kw in key.lower() for kw in ['password', 'secret', 'key',
'token']):
                section_dict[key] = "***"
            else:
                section_dict[key] = value
        result[section] = section_dict
    return result

```

```
# 全局配置实例
settings = Settings()
```

engine/

engine/essay_scorer.py

```
# 自然写手写识别与AI分析引擎软件 V1.0
# 作文评分模型模块 - 深度学习作文评分模型推理管道

"""
作文评分深度学习模型
基于BERT/ERNIE预训练模型微调的中文作文评分器
支持多维度评分：内容、结构、语言、思想感情
"""

import time
import logging
import numpy as np
from typing import List, Dict, Optional, Tuple
from dataclasses import dataclass, field
from pathlib import Path

logger = logging.getLogger(__name__)

# ===== 模型配置 =====

@dataclass
class EssayModelConfig:
    """作文评分模型配置"""
    model_name: str = "writtech-essay-scorer-v1"
    model_path: str = "/opt/models/essay_scorer"
    max_seq_length: int = 512          # 最大输入序列长度
    num_labels: int = 4                # 评分维度数量
    score_range: Tuple[int, int] = (0, 100) # 评分范围
    batch_size: int = 8                # 推理批大小
    use_gpu: bool = True               # 是否使用GPU加速
    fp16_inference: bool = True        # 是否使用FP16半精度推理

# ===== 文本特征提取器 =====

class TextFeatureExtractor:
    """
    文本特征提取器
    从作文文本中提取用于评分的统计特征和语义特征
    统计特征包括：字数、句数、段落数、词汇丰富度等
    语义特征通过预训练语言模型编码获得
    """

    # 常用连接词库（用于衡量行文逻辑性）
    CONNECTIVES = {
```

```

        'causal': ['因为', '所以', '因此', '由于', '于是', '故而'],
        'adversative': ['但是', '然而', '可是', '不过', '虽然', '尽管'],
        'progressive': ['而且', '并且', '不仅', '还', '甚至', '更'],
        'sequential': ['首先', '其次', '然后', '接着', '最后', '总之'],
    }

    # 形容词库（用于衡量描写丰富度）
    DESCRIPTIVE_WORDS = [
        '美丽', '壮观', '温柔', '热烈', '寂静', '辽阔', '清澈', '明亮',
        '灿烂', '幽静', '巍峨', '绚丽', '优雅', '淳朴', '恬静', '磅礴',
        '蜿蜒', '苍翠', '碧绿', '湛蓝', '金黄', '洁白', '火红', '嫣红'
    ]

    def extract_statistical_features(self, text: str) -> Dict[str, float]:
        """
        提取文本统计特征
        返回用于评分的多维统计向量
        """
        features = {}

        # 基础统计
        chinese_chars = [c for c in text if '\u4e00' <= c <= '\u9fff']
        sentences = [s for s in text.replace('!', '.').replace('?', '.').split('.')
                     if s.strip()]
        paragraphs = [p for p in text.split('\n') if p.strip()]

        features['char_count'] = len(chinese_chars)
        features['sentence_count'] = len(sentences)
        features['paragraph_count'] = len(paragraphs)

        # 平均句长（衡量语句复杂度）
        if sentences:
            sentence_lengths = [len([c for c in s if '\u4e00' <= c <= '\u9fff']) for s
                                in sentences]
            features['avg_sentence_length'] = np.mean(sentence_lengths)
            features['sentence_length_std'] = np.std(sentence_lengths)
        else:
            features['avg_sentence_length'] = 0
            features['sentence_length_std'] = 0

        # 词汇丰富度（不同字的比例）
        unique_chars = set(chinese_chars)
        features['vocab_richness'] = len(unique_chars) / max(len(chinese_chars), 1)

        # 连接词使用统计
        total_connectives = 0
        for category, words in self.CONNECTIVES.items():
            count = sum(text.count(w) for w in words)
            features[f'connective_{category}'] = count
            total_connectives += count
        features['total_connectives'] = total_connectives

        # 形容词使用统计（衡量描写丰富度）
        descriptive_count = sum(text.count(w) for w in self.DESCRPTIVE_WORDS)
        features['descriptive_count'] = descriptive_count

        # 标点符号使用统计

```



```

        features['comma_count'] = text.count(',')
        features['period_count'] = text.count('.')
        features['exclamation_count'] = text.count('! ')
        features['question_count'] = text.count('? ')
        features['quotation_count'] = text.count('"') + text.count('\'')

    return features

def extract_ngram_features(self, text: str, n: int = 2) -> Dict[str, int]:
    """
    提取字符N-gram特征
    用于捕捉局部文本模式
    """
    chinese_text = ''.join(c for c in text if '\u4e00' <= c <= '\u9fff')
    ngrams = {}
    for i in range(len(chinese_text) - n + 1):
        gram = chinese_text[i:i+n]
        ngrams[gram] = ngrams.get(gram, 0) + 1
    return ngrams

def text_to_embedding(self, text: str, max_length: int = 512) -> np.ndarray:
    """
    将文本转换为语义向量（模拟BERT编码）
    实际生产环境中使用ERNIE/BERT模型编码
    此处使用统计特征向量作为替代表示
    """
    features = self.extract_statistical_features(text)
    # 构造特征向量并归一化
    feat_values = list(features.values())
    feat_array = np.array(feat_values, dtype=np.float32)
    # L2归一化
    norm = np.linalg.norm(feat_array)
    if norm > 0:
        feat_array = feat_array / norm
    # 填充/截断至固定维度
    target_dim = 64
    if len(feat_array) < target_dim:
        feat_array = np.pad(feat_array, (0, target_dim - len(feat_array)))
    else:
        feat_array = feat_array[:target_dim]
    return feat_array

# ===== 评分模型推理器 =====

class EssayScorerModel:
    """
    作文评分模型推理器
    加载预训练的作文评分模型，执行多维度评分推理
    支持GPU加速和FP16半精度推理以降低延迟
    """

    def __init__(self, config: EssayModelConfig):
        self._config = config
        self._model = None
        self._tokenizer = None
        self._feature_extractor = TextFeatureExtractor()

```

```

self._is_loaded = False
# 评分维度名称映射
self._dimension_names = ['content', 'structure', 'language', 'emotion']
logger.info(f"作文评分模型初始化: {config.model_name}")

def load_model(self) -> bool:
    """
    加载评分模型权重
    模型文件从加密存储中读取并在内存中解密（安全设计）
    """
    try:
        model_dir = Path(self._config.model_path)
        logger.info(f"正在加载作文评分模型: {model_dir}")

        # 检查模型文件是否存在
        # 实际环境中加载PyTorch/ONNX模型权重
        # self._model = onnxruntime.InferenceSession(str(model_dir / "model.onnx"))
        # self._tokenizer = AutoTokenizer.from_pretrained(str(model_dir))

        # 模型加载成功后设置标志
        self._is_loaded = True
        logger.info(f"作文评分模型加载完成: {self._config.model_name}")
        return True
    except Exception as e:
        logger.error(f"模型加载失败: {str(e)}")
        return False

def predict(self, text: str, grade: int = 6) -> Dict[str, float]:
    """
    执行评分推理
    输入作文文本，输出各维度评分
    """
    start_time = time.time()

    # 提取文本特征
    features = self._feature_extractor.extract_statistical_features(text)
    embedding = self._feature_extractor.text_to_embedding(text)

    # 基于特征的规则评分（作为模型推理的后备方案）
    scores = self._rule_based_scoring(features, grade)

    elapsed = (time.time() - start_time) * 1000
    logger.debug(f"评分推理完成: {elapsed:.1f}ms")

    return {
        'scores': scores,
        'features': features,
        'inference_time_ms': round(elapsed, 2)
    }

def _rule_based_scoring(self, features: Dict, grade: int) -> Dict[str, float]:
    """
    基于规则的评分逻辑（模型推理的后备方案）
    当深度学习模型不可用时，使用统计特征进行启发式评分
    """
    scores = {}

```

```

# 内容评分 (30%权重)
# 基于字数、词汇丰富度、描写词使用量
content_score = 60.0 # 基础分
expected_chars = {1: 100, 2: 150, 3: 250, 4: 350, 5: 450, 6: 550, 7: 650, 8:
750, 9: 800}
expected = expected_chars.get(grade, 500)
char_ratio = min(features.get('char_count', 0) / max(expected, 1), 1.5)
content_score += char_ratio * 20

# 词汇丰富度加分
vocab = features.get('vocab_richness', 0)
if vocab > 0.5:
    content_score += 10
elif vocab > 0.3:
    content_score += 5

# 描写丰富度加分
if features.get('descriptive_count', 0) >= 3:
    content_score += 8
elif features.get('descriptive_count', 0) >= 1:
    content_score += 4

scores['content'] = min(100, max(0, round(content_score, 1)))

# 结构评分 (25%权重)
structure_score = 65.0
para_count = features.get('paragraph_count', 1)
if 3 <= para_count <= 7:
    structure_score += 20
elif 2 <= para_count <= 8:
    structure_score += 10

# 有开头结尾连接词加分
if features.get('connective_sequential', 0) >= 2:
    structure_score += 10

scores['structure'] = min(100, max(0, round(structure_score, 1)))

# 语言评分 (25%权重)
language_score = 70.0
avg_sent_len = features.get('avg_sentence_length', 0)
if 8 <= avg_sent_len <= 25:
    language_score += 15 # 句长适中
elif avg_sent_len > 40:
    language_score -= 10 # 句子过长扣分

# 连接词使用加分
total_conn = features.get('total_connectives', 0)
if total_conn >= 4:
    language_score += 10
elif total_conn >= 2:
    language_score += 5

scores['language'] = min(100, max(0, round(language_score, 1)))

# 思想感情评分 (20%权重)
emotion_score = 65.0

```

```

        if features.get('exclamation_count', 0) >= 1:
            emotion_score += 8
        if features.get('question_count', 0) >= 1:
            emotion_score += 5
        if features.get('quotation_count', 0) >= 2:
            emotion_score += 7 # 有引用/对话

    scores['emotion'] = min(100, max(0, round(emotion_score, 1)))

    return scores

def batch_predict(self, texts: List[str], grade: int = 6) -> List[Dict]:
    """
    批量评分推理
    支持一次处理多篇作文，提高GPU利用率
    """
    results = []
    batch_start = time.time()

    for i in range(0, len(texts), self._config.batch_size):
        batch = texts[i:i + self._config.batch_size]
        for text in batch:
            result = self.predict(text, grade)
            results.append(result)

    total_time = (time.time() - batch_start) * 1000
    logger.info(f"批量评分完成: {len(texts)}篇，总耗时{total_time:.1f}ms")
    return results

# ===== 评分校准器 =====

class ScoreCalibrator:
    """
    评分校准器
    将模型原始评分校准到符合教学实际的分数分布
    基于历史评分数据进行分布对齐，避免评分过高或过低
    """

    def __init__(self):
        # 各年级历史评分的均值和标准差（用于正态分布校准）
        self._grade_stats = {
            1: {'mean': 75, 'std': 12},
            2: {'mean': 76, 'std': 11},
            3: {'mean': 78, 'std': 10},
            4: {'mean': 77, 'std': 11},
            5: {'mean': 76, 'std': 12},
            6: {'mean': 75, 'std': 13},
            7: {'mean': 73, 'std': 14},
            8: {'mean': 72, 'std': 15},
            9: {'mean': 71, 'std': 15},
        }

    def calibrate(self, raw_score: float, grade: int, max_score: int = 100) -> float:
        """
        校准原始评分
        将模型输出的原始分数校准到目标分布范围

```

```

        """
        stats = self._grade_stats.get(grade, {'mean': 75, 'std': 12})

        # Z-score标准化后重新映射
        z_score = (raw_score - 50) / 25 # 假设原始分数均值50, 标准差25
        calibrated = stats['mean'] + z_score * stats['std']

        # 裁剪到有效范围
        calibrated = max(max_score * 0.2, min(max_score, calibrated))
        return round(calibrated, 1)

    def calibrate_dimensions(self, dimension_scores: Dict[str, float],
                           grade: int, max_score: int = 100) -> Dict[str, float]:
        """校准各维度评分"""
        weights = {'content': 0.30, 'structure': 0.25, 'language': 0.25, 'emotion':
0.20}
        calibrated = {}
        for dim, score in dimension_scores.items():
            raw_calibrated = self.calibrate(score, grade, 100)
            # 按维度权重换算为该维度的实际分值
            dim_max = max_score * weights.get(dim, 0.25)
            calibrated[dim] = round(raw_calibrated / 100 * dim_max, 1)
        return calibrated

```

engine/stroke_analyzer.py

```

# 自然写手写识别与AI分析引擎软件 V1.0
# 笔顺分析算法模块 - 笔画拆分与顺序分析核心算法

"""
笔顺分析核心算法
提供笔画自动拆分、方向判定、笔画连接检测、
笔迹相似度计算等底层分析算法
"""

import math
import logging
import numpy as np
from typing import List, Dict, Tuple, Optional
from dataclasses import dataclass, field
from enum import IntEnum

logger = logging.getLogger(__name__)

# ===== 常量定义 =====

# 笔画方向角度范围 (度数)
DIRECTION_ANGLES = {
    "horizontal": (-15, 15),      # 横
    "vertical": (75, 105),       # 竖
    "left_falling": (120, 165),  # 撇
    "right_falling": (30, 75),   # 捺
    "dot": None,                 # 点 (特殊判定)
    "turning": None,             # 折 (特殊判定)
}

```

```

        "hook":          None,          # 钩 (特殊判定)
        "rising":        (-60, -15),    # 提
    }

# 笔画最小长度阈值 (像素), 低于此值视为噪声
MIN_STROKE_LENGTH = 3.0
# 笔画分段时的角度变化阈值 (度数)
ANGLE_CHANGE_THRESHOLD = 45.0
# 采样点间距最小阈值
MIN_POINT_DISTANCE = 1.0

class StrokeType(IntEnum):
    """笔画类型枚举"""
    UNKNOWN = 0
    HORIZONTAL = 1      # 横
    VERTICAL = 2        # 竖
    LEFT_FALLING = 3    # 撇
    RIGHT_FALLING = 4   # 捺
    DOT = 5             # 点
    TURNING = 6         # 折
    HOOK = 7            # 钩
    RISING = 8          # 提

@dataclass
class Point2D:
    """二维坐标点"""
    x: float
    y: float
    pressure: float = 0.5
    timestamp: int = 0

@dataclass
class StrokeSegment:
    """笔画片段"""
    points: List[Point2D]
    stroke_type: StrokeType = StrokeType.UNKNOWN
    direction_angle: float = 0.0
    length: float = 0.0
    curvature: float = 0.0
    avg_speed: float = 0.0
    start_point: Optional[Point2D] = None
    end_point: Optional[Point2D] = None

# ===== 笔迹几何工具 =====

class StrokeGeometry:
    """笔迹几何计算工具类"""

    @staticmethod
    def distance(p1: Point2D, p2: Point2D) -> float:
        """计算两点间欧氏距离"""
        return math.sqrt((p2.x - p1.x) ** 2 + (p2.y - p1.y) ** 2)

```

```

@staticmethod
def angle_degrees(p1: Point2D, p2: Point2D) -> float:
    """计算从p1到p2的方向角（度数，0度为正右，顺时针为正）"""
    dx = p2.x - p1.x
    dy = p2.y - p1.y
    return math.degrees(math.atan2(dy, dx))

@staticmethod
def path_length(points: List[Point2D]) -> float:
    """计算点序列的路径总长度"""
    total = 0.0
    for i in range(1, len(points)):
        total += StrokeGeometry.distance(points[i-1], points[i])
    return total

@staticmethod
def curvature_ratio(points: List[Point2D]) -> float:
    """
    计算弯曲度比值（路径长度 / 首尾直线距离）
    1.0表示完全直线，数值越大弯曲程度越高
    """
    if len(points) < 2:
        return 1.0
    path_len = StrokeGeometry.path_length(points)
    direct = StrokeGeometry.distance(points[0], points[-1])
    return path_len / max(direct, 0.001)

@staticmethod
def bounding_box(points: List[Point2D]) -> Tuple[float, float, float, float]:
    """计算点集的包围盒（min_x, min_y, max_x, max_y）"""
    xs = [p.x for p in points]
    ys = [p.y for p in points]
    return min(xs), min(ys), max(xs), max(ys)

@staticmethod
def centroid(points: List[Point2D]) -> Point2D:
    """计算点集的几何重心"""
    cx = sum(p.x for p in points) / len(points)
    cy = sum(p.y for p in points) / len(points)
    return Point2D(cx, cy)

@staticmethod
def resample(points: List[Point2D], n: int) -> List[Point2D]:
    """
    等距重采样：将不规则间距的点序列重采样为n个等距点
    这是笔迹比较的基础预处理步骤
    """
    if len(points) <= 1 or n <= 1:
        return points[:n] if points else []

    total_len = StrokeGeometry.path_length(points)
    interval = total_len / (n - 1)
    resampled = [Point2D(points[0].x, points[0].y, points[0].pressure)]

    accumulated = 0.0
    j = 1
    for i in range(1, n - 1):

```

```

        target_dist = i * interval
        while j < len(points) and accumulated + StrokeGeometry.distance(points[j-1],
points[j]) < target_dist:
            accumulated += StrokeGeometry.distance(points[j-1], points[j])
            j += 1
        if j >= len(points):
            break

        remaining = target_dist - accumulated
        seg_len = StrokeGeometry.distance(points[j-1], points[j])
        ratio = remaining / max(seg_len, 0.001)
        # 线性插值计算新坐标
        new_x = points[j-1].x + ratio * (points[j].x - points[j-1].x)
        new_y = points[j-1].y + ratio * (points[j].y - points[j-1].y)
        new_p = points[j-1].pressure + ratio * (points[j].pressure - points[j-
1].pressure)
        resampled.append(Point2D(new_x, new_y, new_p))

        resampled.append(Point2D(points[-1].x, points[-1].y, points[-1].pressure))
    return resampled

```

===== 笔画拆分离器 =====

```

class StrokeSplitter:
    """
    笔画拆分离器
    将连续的笔迹坐标流自动拆分为独立的笔画段
    基于以下特征进行拆分：
    1. 抬笔点 (pressure=0或时间间隔大)
    2. 方向突变点 (角度变化超过阈值)
    3. 速度突变点 (书写速度骤降后回升)
    """

    def __init__(self, angle_threshold: float = ANGLE_CHANGE_THRESHOLD,
                  time_gap_ms: int = 300, speed_ratio: float = 0.3):
        self._angle_threshold = angle_threshold
        self._time_gap_ms = time_gap_ms
        self._speed_ratio = speed_ratio

    def split_by_penup(self, points: List[Point2D]) -> List[List[Point2D]]:
        """
        基于抬笔事件拆分笔画
        当相邻点的时间间隔超过阈值或压力为0时，视为抬笔
        """
        if not points:
            return []

        strokes = []
        current_stroke = [points[0]]

        for i in range(1, len(points)):
            time_gap = points[i].timestamp - points[i-1].timestamp
            is_penup = (points[i].pressure <= 0.01 or time_gap > self._time_gap_ms)

            if is_penup and len(current_stroke) > 1:
                strokes.append(current_stroke)

```



```

        current_stroke = [points[i]]
    else:
        current_stroke.append(points[i])

    if len(current_stroke) > 1:
        strokes.append(current_stroke)

    return strokes

def split_by_direction(self, points: List[Point2D]) -> List[List[Point2D]]:
    """
    基于方向突变拆分笔画（用于折笔检测）
    当连续点的方向角变化超过阈值时，在该点进行拆分
    """
    if len(points) < 3:
        return [points] if points else []

    segments = []
    current = [points[0]]
    prev_angle = StrokeGeometry.angle_degrees(points[0], points[1])

    for i in range(1, len(points)):
        current.append(points[i])
        if i + 1 < len(points):
            curr_angle = StrokeGeometry.angle_degrees(points[i], points[i+1])
            angle_diff = abs(curr_angle - prev_angle)
            # 处理角度跨越±180度的情况
            if angle_diff > 180:
                angle_diff = 360 - angle_diff

            if angle_diff > self._angle_threshold and len(current) > 2:
                segments.append(current)
                current = [points[i]] # 拆分点同时作为下一段起点
                prev_angle = curr_angle

    if len(current) > 1:
        segments.append(current)

    return segments

def split_by_speed(self, points: List[Point2D]) -> List[List[Point2D]]:
    """
    基于速度突变拆分笔画
    当书写速度骤降至平均速度的指定比例以下时，视为停顿点
    """
    if len(points) < 3:
        return [points] if points else []

    # 计算每个点的瞬时速度
    speeds = []
    for i in range(1, len(points)):
        dist = StrokeGeometry.distance(points[i-1], points[i])
        dt = max(points[i].timestamp - points[i-1].timestamp, 1)
        speeds.append(dist / dt * 1000) # 像素/秒

    avg_speed = np.mean(speeds) if speeds else 0
    threshold = avg_speed * self._speed_ratio

```

```

segments = []
current = [points[0]]

for i in range(len(speeds)):
    current.append(points[i + 1])
    if speeds[i] < threshold and len(current) > 3:
        segments.append(current)
        current = [points[i + 1]]

if len(current) > 1:
    segments.append(current)

return segments

# ===== 笔画类型分类器 =====

class StrokeClassifier:
    """
    笔画类型分类器
    根据笔画的几何特征（方向、长度、弯曲度）判定笔画类型
    """

    @staticmethod
    def classify(segment: List[Point2D]) -> StrokeType:
        """对单个笔画片段进行类型分类"""
        if len(segment) < 2:
            return StrokeType.DOT

        length = StrokeGeometry.path_length(segment)
        curvature = StrokeGeometry.curvature_ratio(segment)

        # 极短笔画判定为点
        if length < MIN_STROKE_LENGTH * 2:
            return StrokeType.DOT

        # 高弯曲度判定为折或钩
        if curvature > 2.0:
            # 检查末端是否有向上的钩
            if len(segment) >= 3:
                end_angle = StrokeGeometry.angle_degrees(segment[-2], segment[-1])
                if -90 < end_angle < -10:
                    return StrokeType.HOOK
            return StrokeType.TURNING

        # 根据整体方向角判定
        angle = StrokeGeometry.angle_degrees(segment[0], segment[-1])

        if -20 <= angle <= 20:
            return StrokeType.HORIZONTAL
        elif 70 <= angle <= 110:
            return StrokeType.VERTICAL
        elif 120 <= angle <= 170 or -170 <= angle <= -150:
            return StrokeType.LEFT_FALLING
        elif 25 <= angle <= 70:
            return StrokeType.RIGHT_FALLING

```

```

        elif -65 <= angle <= -20:
            return StrokeType.RISING
        else:
            return StrokeType.UNKNOWN

# ===== 笔迹相似度计算 =====

class StrokeSimilarity:
    """
    笔迹相似度计算
    使用DTW (Dynamic Time Warping) 算法计算两条笔迹的相似程度
    用于笔顺比对和模板匹配
    """

    @staticmethod
    def dtw_distance(seq1: List[Point2D], seq2: List[Point2D]) -> float:
        """
        动态时间规整距离
        衡量两条时间序列的最小累积匹配距离
        """
        n = len(seq1)
        m = len(seq2)
        if n == 0 or m == 0:
            return float('inf')

        # 初始化代价矩阵
        dtw_matrix = np.full((n + 1, m + 1), float('inf'))
        dtw_matrix[0][0] = 0

        for i in range(1, n + 1):
            for j in range(1, m + 1):
                cost = StrokeGeometry.distance(seq1[i-1], seq2[j-1])
                dtw_matrix[i][j] = cost + min(
                    dtw_matrix[i-1][j],      # 插入
                    dtw_matrix[i][j-1],      # 删除
                    dtw_matrix[i-1][j-1]    # 匹配
                )

        return dtw_matrix[n][m]

    @staticmethod
    def normalized_similarity(seq1: List[Point2D], seq2: List[Point2D],
                             resample_n: int = 32) -> float:
        """
        归一化笔迹相似度 (0-1之间, 1表示完全相同)
        先等距重采样再计算DTW距离, 最后归一化
        """
        # 等距重采样至相同点数
        rs1 = StrokeGeometry.resample(seq1, resample_n)
        rs2 = StrokeGeometry.resample(seq2, resample_n)

        if not rs1 or not rs2:
            return 0.0

        # 归一化坐标到[0,1]范围
        all_pts = rs1 + rs2

```

```

        bbox = StrokeGeometry.bounding_box(all_pts)
        scale = max(bbox[2] - bbox[0], bbox[3] - bbox[1], 1.0)

        norm1 = [Point2D((p.x - bbox[0]) / scale, (p.y - bbox[1]) / scale) for p in rs1]
        norm2 = [Point2D((p.x - bbox[0]) / scale, (p.y - bbox[1]) / scale) for p in rs2]

        dtw_dist = StrokeSimilarity.dtw_distance(norm1, norm2)
        # 将DTW距离映射到相似度分数
        similarity = max(0, 1.0 - dtw_dist / resample_n)
        return round(similarity, 4)

# ===== 笔顺分析器（整合） =====

class StrokeAnalyzer:
    """
    笔顺分析器（整合所有子模块）
    提供完整的笔画拆分→分类→排序→比对分析流程
    """

    def __init__(self):
        self._splitter = StrokeSplitter()
        self._classifier = StrokeClassifier()
        self._similarity = StrokeSimilarity()
        logger.info("笔顺分析器初始化完成")

    def analyze(self, raw_points: List[Point2D]) -> List[StrokeSegment]:
        """
        完整分析流程：原始坐标 → 拆分 → 分类 → 输出笔画序列
        """

        # 第一步：按抬笔事件拆分
        strokes = self._splitter.split_by_penup(raw_points)

        segments = []
        for stroke_points in strokes:
            # 第二步：过滤噪声笔画
            if StrokeGeometry.path_length(stroke_points) < MIN_STROKE_LENGTH:
                continue

            # 第三步：分类笔画类型
            stroke_type = self._classifier.classify(stroke_points)

            # 第四步：构造笔画片段对象
            seg = StrokeSegment(
                points=stroke_points,
                stroke_type=stroke_type,
                direction_angle=StrokeGeometry.angle_degrees(stroke_points[0],
stroke_points[-1]),
                length=StrokeGeometry.path_length(stroke_points),
                curvature=StrokeGeometry.curvature_ratio(stroke_points),
                start_point=stroke_points[0],
                end_point=stroke_points[-1]
            )

            # 计算书写速度
            if stroke_points[-1].timestamp > stroke_points[0].timestamp:
                time_s = (stroke_points[-1].timestamp - stroke_points[0].timestamp) /

```

```

1000.0

        seg.avg_speed = seg.length / max(time_s, 0.001)

    segments.append(seg)

    logger.debug(f"笔迹分析完成: {len(raw_points)}个原始点 → {len(segments)}个笔画")
    return segments

def compare_stroke_orders(self, user_strokes: List[List[Point2D]],
                           template_strokes: List[List[Point2D]]) -> Dict:
    """
    比对用户笔画与模板笔画的相似度
    返回每一笔的匹配结果和整体相似度分数
    """
    match_results = []
    total_similarity = 0.0
    compare_count = min(len(user_strokes), len(template_strokes))

    for i in range(compare_count):
        sim = self._similarity.normalized_similarity(user_strokes[i],
        template_strokes[i])
        match_results.append({
            "stroke_index": i + 1,
            "similarity": sim,
            "match": sim > 0.6
        })
        total_similarity += sim

    avg_similarity = total_similarity / max(compare_count, 1)
    count_penalty = abs(len(user_strokes) - len(template_strokes)) * 0.1

    return {
        "overall_similarity": round(max(0, avg_similarity - count_penalty), 4),
        "stroke_matches": match_results,
        "user_count": len(user_strokes),
        "template_count": len(template_strokes)
    }

```

grpc_server/

grpc_server/inference_service.py

```

# 自然手写识别与AI分析引擎软件 V1.0
# gRPC批量识别服务模块 - 高性能流式批量笔迹识别

"""
gRPC推理服务
提供高性能流式批量笔迹识别接口
采用gRPC双向流模式，适用于教室场景下多支笔并发识别需求
支持服务端流式响应，实现低延迟识别结果推送
"""

import time

```

```

import json
import logging
import uuid
import asyncio
from typing import List, Dict, Optional, AsyncIterator
from dataclasses import dataclass, field
from enum import Enum
from concurrent import futures

logger = logging.getLogger(__name__)

# ===== gRPC消息定义 (等效Proto) =====

class RecognitionType(str, Enum):
    """识别类型枚举"""
    OCR = "ocr" # 文字识别
    MATH = "math" # 数学识别
    STROKE_ORDER = "stroke_order" # 笔顺评分
    ESSAY = "essay" # 作文批改

@dataclass
class StrokePoint:
    """笔迹坐标点 (对应protobuf StrokePoint message) """
    x: float
    y: float
    pressure: float = 0.5
    timestamp: int = 0

@dataclass
class StrokeData:
    """笔迹数据 (对应protobuf StrokeData message) """
    stroke_id: str = ""
    pen_id: str = ""
    page_id: str = ""
    student_id: str = ""
    strokes: List[List[StrokePoint]] = field(default_factory=list)

@dataclass
class RecognitionRequest:
    """识别请求 (对应protobuf RecognitionRequest message) """
    request_id: str = ""
    recognition_type: RecognitionType = RecognitionType.OCR
    stroke_data: Optional[StrokeData] = None
    priority: int = 2 # 0=最高优先级, 4=最低
    callback_topic: str = "" # 结果回调MQTT Topic
    timeout_ms: int = 5000 # 超时时间

@dataclass
class RecognitionResult:
    """识别结果 (对应protobuf RecognitionResult message) """
    request_id: str = ""
    recognition_type: str = ""
    status: str = "success" # success / error / timeout

```

```

    result_text: str = ""
    confidence: float = 0.0
    details: Dict = field(default_factory=dict)
    processing_time_ms: float = 0.0
    model_version: str = ""

# ===== 批量识别处理器 =====

class BatchRecognitionProcessor:
    """
    批量识别处理器
    将多个识别请求按类型分组，批量送入GPU推理
    通过批处理显著提升GPU利用率和吞吐量
    """

    def __init__(self, max_batch_size: int = 32, max_wait_ms: int = 50):
        self._max_batch_size = max_batch_size
        self._max_wait_ms = max_wait_ms
        self._pending_requests: Dict[str, List[RecognitionRequest]] = {
            rt.value: [] for rt in RecognitionType
        }
        self._results: Dict[str, RecognitionResult] = {}
        logger.info(f"批量识别处理器初始化: batch_size={max_batch_size}, wait_ms={max_wait_ms}")

    def add_request(self, request: RecognitionRequest) -> str:
        """添加识别请求到批处理队列"""
        if not request.request_id:
            request.request_id = str(uuid.uuid4())

        queue = self._pending_requests.get(request.recognition_type.value, [])
        queue.append(request)
        self._pending_requests[request.recognition_type.value] = queue

        logger.debug(f"请求入队: id={request.request_id}, type={request.recognition_type.value}")

        # 当队列达到批大小时触发批处理
        if len(queue) >= self._max_batch_size:
            self._process_batch(request.recognition_type.value)

        return request.request_id

    def _process_batch(self, recognition_type: str):
        """
        执行批处理推理
        将队列中的请求按批大小取出，统一送入模型推理
        """
        queue = self._pending_requests.get(recognition_type, [])
        if not queue:
            return

        batch = queue[:self._max_batch_size]
        self._pending_requests[recognition_type] = queue[self._max_batch_size:]

        batch_start = time.time()

```

```

logger.info(f"批处理开始: type={recognition_type}, batch_size={len(batch)}")

for req in batch:
    try:
        result = self._process_single(req)
        self._results[req.request_id] = result
    except Exception as e:
        self._results[req.request_id] = RecognitionResult(
            request_id=req.request_id,
            recognition_type=recognition_type,
            status="error",
            details={"error": str(e)}
        )

elapsed = (time.time() - batch_start) * 1000
logger.info(f"批处理完成: type={recognition_type}, count={len(batch)}, time={elapsed:.1f}ms")

def _process_single(self, request: RecognitionRequest) -> RecognitionResult:
    """处理单个识别请求"""
    start_time = time.time()

    # 根据识别类型分发到对应的推理引擎
    if request.recognition_type == RecognitionType.OCR:
        result_text = self._run_ocr_inference(request.stroke_data)
        confidence = 0.92
    elif request.recognition_type == RecognitionType.MATH:
        result_text = self._run_math_inference(request.stroke_data)
        confidence = 0.88
    elif request.recognition_type == RecognitionType.STROKE_ORDER:
        result_text = self._run_stroke_order_inference(request.stroke_data)
        confidence = 0.95
    else:
        result_text = ""
        confidence = 0.0

    elapsed = (time.time() - start_time) * 1000

    return RecognitionResult(
        request_id=request.request_id,
        recognition_type=request.recognition_type.value,
        status="success",
        result_text=result_text,
        confidence=confidence,
        processing_time_ms=round(elapsed, 2),
        model_version="v1.0.0"
    )

def _run_ocr_inference(self, stroke_data: Optional[StrokeData]) -> str:
    """执行OCR推理（调用PaddleOCR引擎）"""
    if not stroke_data or not stroke_data.strokes:
        return ""
    # 实际环境中调用PaddleOCR推理管道
    # preprocessed = preprocess(stroke_data)
    # result = ocr_engine.recognize(preprocessed)
    return "[OCR识别结果]"

```



```

def _run_math_inference(self, stroke_data: Optional[StrokeData]) -> str:
    """执行数学列式识别推理"""
    if not stroke_data or not stroke_data.strokes:
        return ""
    return "[数学识别结果]"

def _run_stroke_order_inference(self, stroke_data: Optional[StrokeData]) -> str:
    """执行笔顺分析推理"""
    if not stroke_data or not stroke_data.strokes:
        return ""
    return "[笔顺分析结果]"

def get_result(self, request_id: str) -> Optional[RecognitionResult]:
    """查询识别结果"""
    return self._results.get(request_id)

def flush_all(self):
    """强制处理所有队列中的待处理请求"""
    for rt in self._pending_requests:
        while self._pending_requests[rt]:
            self._process_batch(rt)

# ===== gRPC服务实现 =====

class RecognitionServiceImpl:
    """
    gRPC RecognitionService 服务实现
    对应 protobuf 服务定义:
    service RecognitionService {
        rpc Recognize(RecognitionRequest) returns (RecognitionResult);
        rpc BatchRecognize(stream RecognitionRequest) returns (stream
RecognitionResult);
        rpc GetModelStatus(Empty) returns (ModelStatusResponse);
    }
    """

    def __init__(self):
        self._processor = BatchRecognitionProcessor()
        self._request_count = 0
        self._total_latency_ms = 0.0
        logger.info("gRPC RecognitionService 初始化完成")

    def Recognize(self, request: RecognitionRequest) -> RecognitionResult:
        """
        单次识别RPC
        接收单个识别请求, 返回识别结果
        """
        self._request_count += 1
        start_time = time.time()

        # 验证请求参数
        if not request.stroke_data or not request.stroke_data.strokes:
            return RecognitionResult(
                request_id=request.request_id,
                status="error",
                details={"error": "笔迹数据为空"}
            )

```

```

    )

    # 提交到批处理器并等待结果
    request_id = self._processor.add_request(request)
    self._processor.flush_all() # 立即处理（单次调用不等待攒批）

    result = self._processor.get_result(request_id)
    elapsed = (time.time() - start_time) * 1000
    self._total_latency_ms += elapsed

    if result:
        # 审计日志
        logger.info(
            f"gRPC Recognize: id={request_id}, type=
{request.recognition_type.value}, "
            f"time={elapsed:.1f}ms, pen={request.stroke_data.pen_id}"
        )
        return result

    return RecognitionResult(
        request_id=request_id, status="error",
        details={"error": "处理超时"}
    )

def BatchRecognize(self, request_iterator) -> List[RecognitionResult]:
    """
    流式批量识别RPC（双向流）
    接收笔迹数据流，批量处理后流式返回识别结果
    适用于教室场景下40+支笔并发传输的高吞吐识别
    """
    results = []
    request_ids = []

    # 接收所有请求
    for request in request_iterator:
        rid = self._processor.add_request(request)
        request_ids.append(rid)
        self._request_count += 1

    # 批量处理
    self._processor.flush_all()

    # 收集结果
    for rid in request_ids:
        result = self._processor.get_result(rid)
        if result:
            results.append(result)

    logger.info(f"BatchRecognize完成：请求数={len(request_ids)}, 结果数=
{len(results)}")
    return results

def GetModelStatus(self) -> Dict:
    """查询模型状态RPC"""
    return {
        "total_requests": self._request_count,
        "avg_latency_ms": round(self._total_latency_ms / max(self._request_count,

```

```

1), 2),
    "models": [
        {"name": "ocr_model", "version": "v1.0.0", "status": "active"},
        {"name": "math_model", "version": "v1.0.0", "status": "active"},
        {"name": "stroke_order_model", "version": "v1.0.0", "status": "active"},
    ]
}

# ===== gRPC服务器启动 =====

class GrpcServer:
    """
    gRPC服务器管理
    启动和管理gRPC推理服务端口
    支持TLS双向认证 (mTLS安全设计)
    """

    def __init__(self, host: str = "0.0.0.0", port: int = 50051,
                  max_workers: int = 10, enable_tls: bool = True):
        self._host = host
        self._port = port
        self._max_workers = max_workers
        self._enable_tls = enable_tls
        self._service = RecognitionServiceImpl()
        self._server = None
        logger.info(f"gRPC服务器配置: {host}:{port}, workers={max_workers}, tls={enable_tls}")

    def start(self):
        """
        启动gRPC服务器
        如启用TLS, 加载服务端证书和CA证书用于mTLS双向认证
        """
        logger.info(f"启动gRPC服务器: {self._host}:{self._port}")

        # 实际环境中的gRPC服务器启动代码
        # self._server =
        grpc.server(futures.ThreadPoolExecutor(max_workers=self._max_workers))
        # inference_pb2_grpc.add_RecognitionServiceServicer_to_server(self._service,
        self._server)
        #
        # if self._enable_tls:
        #     # mTLS双向认证配置 (安全设计)
        #     with open('/etc/ssl/server.key', 'rb') as f:
        #         server_key = f.read()
        #     with open('/etc/ssl/server.crt', 'rb') as f:
        #         server_cert = f.read()
        #     with open('/etc/ssl/ca.crt', 'rb') as f:
        #         ca_cert = f.read()
        #     credentials = grpc.ssl_server_credentials(
        #         [(server_key, server_cert)],
        #         root_certificates=ca_cert,
        #         require_client_auth=True # 要求客户端证书
        #     )
        #     self._server.add_secure_port(f'{self._host}:{self._port}', credentials)
        # else:

```

```

        # self._server.add_insecure_port(f'{self._host}:{self._port}')
        #
        # self._server.start()

        logger.info(f"gRPC服务器已启动: {self._host}:{self._port}")

    def stop(self, grace_seconds: int = 5):
        """优雅关闭gRPC服务器"""
        if self._server:
            # self._server.stop(grace_seconds)
            logger.info("gRPC服务器已关闭")

    def get_stats(self) -> Dict:
        """获取服务器统计信息"""
        return self._service.GetModelStatus()

```

preprocessing/

preprocessing/stroke_processor.py

```

# 自然写手写识别与AI分析引擎软件 V1.0
# 笔迹预处理模块 - 笔迹数据预处理管道

"""
笔迹预处理模块
提供笔迹坐标数据的完整预处理管道：
去噪 → 坐标归一化 → 笔画分割 → 特征增强 → 张量转换
预处理结果作为AI推理模型的标准化输入
"""

import math
import logging
import numpy as np
from typing import List, Dict, Tuple, Optional
from dataclasses import dataclass

logger = logging.getLogger(__name__)

# ===== 数据结构 =====

@dataclass
class RawStrokePoint:
    """原始笔迹坐标点（来自点阵笔/网关的原始数据）"""
    x: float          # X坐标（点阵单位）
    y: float          # Y坐标（点阵单位）
    pressure: float    # 压力值（0.0-1.0）
    timestamp: int     # 采集时间戳（毫秒）
    pen_up: bool = False # 抬笔标记

@dataclass
class ProcessedStroke:
    """预处理后的笔画数据"""

```

```

points: np.ndarray          # 归一化坐标数组 (N, 3) [x, y, pressure]
stroke_index: int = 0       # 笔画序号
point_count: int = 0        # 采样点数
length: float = 0.0         # 笔画长度
duration_ms: int = 0        # 书写耗时

```

===== 去噪滤波器 =====

```
class NoiseFilter:
```

```
    """
```

```
    笔迹去噪滤波器
```

```
    去除采集过程中的抖动噪声和异常点
```

```
    采用多级滤波策略:
```

1. 异常点剔除 (超出合理范围的坐标)
2. 中值滤波 (消除脉冲噪声)
3. 高斯平滑 (减少抖动)

```
    """
```

```

def __init__(self, max_jump_distance: float = 50.0,
               median_window: int = 3, gaussian_sigma: float = 1.0):
    self._max_jump = max_jump_distance
    self._median_window = median_window
    self._gaussian_sigma = gaussian_sigma

```

```

def remove_outliers(self, points: List[RawStrokePoint]) -> List[RawStrokePoint]:
    """

```

```
    剔除异常跳跃点
```

```
    当相邻点的距离超过阈值时, 移除该异常点
```

```
    常见于点阵笔摄像头短暂遮挡导致的坐标跳跃
```

```
    """
```

```

    if len(points) < 3:
        return points

```

```
    filtered = [points[0]]
```

```
    for i in range(1, len(points)):
```

```
        dx = points[i].x - points[i-1].x
```

```
        dy = points[i].y - points[i-1].y
```

```
        dist = math.sqrt(dx*dx + dy*dy)
```

```
        if dist <= self._max_jump:
```

```
            filtered.append(points[i])
```

```
        else:
```

```
            logger.debug(f"剔除异常点: index={i}, distance={dist:.1f}")
```

```
    return filtered
```

```

def median_filter(self, points: List[RawStrokePoint]) -> List[RawStrokePoint]:
    """

```

```
    """
```

```
    一维中值滤波
```

```
    对X和Y坐标分别进行中值滤波, 有效消除脉冲噪声
```

```
    同时保留笔画的尖角特征不被过度平滑
```

```
    """
```

```

    if len(points) < self._median_window:
        return points

```

```
    half_w = self._median_window // 2
```

```

        filtered = []

    for i in range(len(points)):
        start = max(0, i - half_w)
        end = min(len(points), i + half_w + 1)
        window = points[start:end]

        median_x = sorted([p.x for p in window])[len(window) // 2]
        median_y = sorted([p.y for p in window])[len(window) // 2]

        filtered.append(RawStrokePoint(
            x=median_x, y=median_y,
            pressure=points[i].pressure,
            timestamp=points[i].timestamp,
            pen_up=points[i].pen_up
        ))

    return filtered

def gaussian_smooth(self, points: List[RawStrokePoint]) -> List[RawStrokePoint]:
    """
    高斯平滑滤波
    使用一维高斯核对坐标序列进行卷积平滑
    有效减少书写抖动，使笔画更流畅
    """
    if len(points) < 3:
        return points

    # 构造高斯核
    kernel_size = max(3, int(self._gaussian_sigma * 4) | 1) # 确保奇数
    half_k = kernel_size // 2
    kernel = np.array([
        math.exp(-0.5 * ((i - half_k) / self._gaussian_sigma) ** 2)
        for i in range(kernel_size)
    ])
    kernel = kernel / kernel.sum() # 归一化

    xs = np.array([p.x for p in points])
    ys = np.array([p.y for p in points])

    # 边界填充后卷积
    padded_x = np.pad(xs, half_k, mode='edge')
    padded_y = np.pad(ys, half_k, mode='edge')

    smooth_x = np.convolve(padded_x, kernel, mode='valid')
    smooth_y = np.convolve(padded_y, kernel, mode='valid')

    filtered = []
    for i in range(len(points)):
        filtered.append(RawStrokePoint(
            x=float(smooth_x[i]), y=float(smooth_y[i]),
            pressure=points[i].pressure,
            timestamp=points[i].timestamp,
            pen_up=points[i].pen_up
        ))
    return filtered

```

```

def apply(self, points: List[RawStrokePoint]) -> List[RawStrokePoint]:
    """执行完整的去噪流程"""
    result = self.remove_outliers(points)
    result = self.median_filter(result)
    result = self.gaussian_smooth(result)
    return result

# ===== 坐标归一化器 =====

class CoordinateNormalizer:
    """
    坐标归一化器
    将不同分辨率、不同纸张尺寸的点阵坐标统一归一化到标准范围
    支持多种归一化策略: Min-Max归一化、Z-Score标准化、比例缩放
    """

    def __init__(self, target_range: Tuple[float, float] = (0.0, 1.0),
                 preserve_aspect_ratio: bool = True):
        self._target_min = target_range[0]
        self._target_max = target_range[1]
        self._preserve_aspect = preserve_aspect_ratio

    def min_max_normalize(self, points: List[RawStrokePoint]) -> List[RawStrokePoint]:
        """
        Min-Max归一化
        将坐标映射到[0, 1]范围, 保持长宽比
        """
        if not points:
            return points

        xs = [p.x for p in points]
        ys = [p.y for p in points]
        min_x, max_x = min(xs), max(xs)
        min_y, max_y = min(ys), max(ys)

        # 选择统一的缩放因子以保持长宽比
        if self._preserve_aspect:
            range_x = max_x - min_x
            range_y = max_y - min_y
            scale = max(range_x, range_y)
            if scale < 1e-6:
                scale = 1.0
        else:
            scale = 1.0 # 分别归一化

        target_range = self._target_max - self._target_min
        normalized = []
        for p in points:
            if self._preserve_aspect:
                nx = self._target_min + (p.x - min_x) / scale * target_range
                ny = self._target_min + (p.y - min_y) / scale * target_range
            else:
                rx = max_x - min_x if max_x > min_x else 1.0
                ry = max_y - min_y if max_y > min_y else 1.0
                nx = self._target_min + (p.x - min_x) / rx * target_range
                ny = self._target_min + (p.y - min_y) / ry * target_range

```

```

        normalized.append(RawStrokePoint(
            x=nx, y=ny, pressure=p.pressure,
            timestamp=p.timestamp, pen_up=p.pen_up
        ))
    return normalized

def center_normalize(self, points: List[RawStrokePoint]) -> List[RawStrokePoint]:
    """
    中心归一化
    将笔迹的重心平移至原点，坐标除以标准差进行缩放
    适用于笔迹特征提取和模板匹配
    """
    if not points:
        return points

    xs = np.array([p.x for p in points])
    ys = np.array([p.y for p in points])

    cx, cy = np.mean(xs), np.mean(ys)
    std = max(np.std(np.concatenate([xs, ys])), 1e-6)

    normalized = []
    for p in points:
        normalized.append(RawStrokePoint(
            x=(p.x - cx) / std,
            y=(p.y - cy) / std,
            pressure=p.pressure,
            timestamp=p.timestamp,
            pen_up=p.pen_up
        ))
    return normalized

# ===== 笔画分割器 =====

class StrokeSegmenter:
    """
    笔画分割器
    将连续的坐标点流按抬笔事件分割为独立笔画
    """

    def __init__(self, min_stroke_points: int = 3,
                 penup_time_threshold_ms: int = 200):
        self._min_points = min_stroke_points
        self._penup_threshold = penup_time_threshold_ms

    def segment(self, points: List[RawStrokePoint]) -> List[List[RawStrokePoint]]:
        """将点序列分割为笔画列表"""
        if not points:
            return []

        strokes = []
        current = [points[0]]

        for i in range(1, len(points)):
            # 检测抬笔条件
            is_penup = points[i].pen_up

```



```

        time_gap = points[i].timestamp - points[i-1].timestamp
        is_time_break = time_gap > self._penup_threshold

        if (is_penup or is_time_break) and len(current) >= self._min_points:
            strokes.append(current)
            current = []

        if not is_penup:
            current.append(points[i])

    if len(current) >= self._min_points:
        strokes.append(current)

    logger.debug(f"笔画分割完成: {len(points)}点 -> {len(strokes)}笔画")
    return strokes

# ===== 预处理管道 =====

class StrokePreprocessor:
    """
    笔迹预处理管道（整合所有预处理步骤）
    流程：原始坐标 → 去噪 → 归一化 → 笔画分割 → 张量转换
    输出标准化的numpy数组，可直接送入AI推理模型
    """

    def __init__(self):
        self._noise_filter = NoiseFilter()
        self._normalizer = CoordinateNormalizer()
        self._segmenter = StrokeSegmenter()
        logger.info("笔迹预处理管道初始化完成")

    def process(self, raw_points: List[RawStrokePoint],
               target_size: Tuple[int, int] = (64, 64)) -> Dict:
        """
        执行完整预处理管道
        返回预处理后的笔画数据和生成的图像张量
        """
        if not raw_points:
            return {"strokes": [], "image": np.zeros(target_size)}

        # 第一步：去噪滤波
        denoised = self._noise_filter.apply(raw_points)

        # 第二步：坐标归一化
        normalized = self._normalizer.min_max_normalize(denoised)

        # 第三步：笔画分割
        stroke_groups = self._segmenter.segment(normalized)

        # 第四步：构造ProcessedStroke对象
        processed_strokes = []
        for idx, group in enumerate(stroke_groups):
            points_array = np.array([p.x, p.y, p.pressure] for p in group),
            dtype=np.float32)
            length = sum(
                math.sqrt((group[i].x - group[i-1].x)**2 + (group[i].y - group[i-

```

```

1].y)**2)
        for i in range(1, len(group))
    )
    duration = group[-1].timestamp - group[0].timestamp if len(group) > 1 else 0

    processed_strokes.append(ProcessedStroke(
        points=points_array,
        stroke_index=idx,
        point_count=len(group),
        length=length,
        duration_ms=duration
    ))

# 第五步：渲染为图像张量（用于CNN模型输入）
image = self._render_to_image(normalized, target_size)

logger.debug(
    f"预处理完成: {len(raw_points)}原始点 → {len(denoised)}去噪 → "
    f"{len(processed_strokes)}笔画 → {target_size}图像"
)

return {
    "strokes": processed_strokes,
    "image": image,
    "total_points": len(denoised),
    "stroke_count": len(processed_strokes)
}

def _render_to_image(self, points: List[RawStrokePoint],
                    size: Tuple[int, int]) -> np.ndarray:
    """
    将笔迹坐标渲染为灰度图像
    使用Bresenham直线算法连接相邻坐标点
    生成的图像可直接作为CNN模型输入
    """
    w, h = size
    image = np.zeros((h, w), dtype=np.float32)

    for i in range(1, len(points)):
        if points[i].pen_up:
            continue

        # Bresenham直线栅格化
        x0 = int(points[i-1].x * (w - 1))
        y0 = int(points[i-1].y * (h - 1))
        x1 = int(points[i].x * (w - 1))
        y1 = int(points[i].y * (h - 1))

        # 裁剪到图像范围
        x0 = max(0, min(w - 1, x0))
        y0 = max(0, min(h - 1, y0))
        x1 = max(0, min(w - 1, x1))
        y1 = max(0, min(h - 1, y1))

        dx = abs(x1 - x0)
        dy = abs(y1 - y0)
        sx = 1 if x0 < x1 else -1

```

```

        sy = 1 if y0 < y1 else -1
        err = dx - dy

    while True:
        # 根据压力值设置像素灰度
        pressure = (points[i-1].pressure + points[i].pressure) / 2
        image[y0, x0] = max(image[y0, x0], pressure)

        if x0 == x1 and y0 == y1:
            break
        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x0 += sx
        if e2 < dx:
            err += dx
            y0 += sy

    return image

```

service/

service/model_manager.py

```

# 自然写手写识别与AI分析引擎软件 V1.0
# 模型版本管理模块 - 模型加载、版本切换、热更新与灰度发布

"""
模型版本管理服务
提供AI推理模型的版本管理、动态加载、热更新、灰度发布、回滚等功能
支持MinIO模型仓库对接和MLflow实验追踪
"""

import os
import time
import json
import hashlib
import shutil
import logging
import threading
from typing import Dict, List, Optional, Tuple
from dataclasses import dataclass, field
from datetime import datetime
from pathlib import Path
from enum import Enum

logger = logging.getLogger(__name__)

# ===== 数据模型 =====

class ModelStatus(str, Enum):
    """模型状态枚举"""
    DOWNLOADING = "downloading" # 下载中

```

```

LOADING = "loading"           # 加载中
ACTIVE = "active"             # 当前活跃
STANDBY = "standby"           # 待命（已加载但未启用）
DEPRECATED = "deprecated"     # 已废弃
FAILED = "failed"             # 加载失败

class DeployStrategy(str, Enum):
    """部署策略枚举"""
    IMMEDIATE = "immediate"     # 立即全量切换
    CANARY = "canary"           # 金丝雀灰度发布
    BLUE_GREEN = "blue_green"   # 蓝绿部署
    ROLLING = "rolling"         # 滚动更新

@dataclass
class ModelVersion:
    """模型版本信息"""
    model_name: str              # 模型名称（如 ocr_v1, math_v2）
    version: str                 # 语义化版本号（如 1.2.3）
    file_path: str               # 本地模型文件路径
    file_size: int = 0           # 文件大小（字节）
    sha256: str = ""            # 文件SHA-256校验和
    accuracy: float = 0.0        # 精度指标（测试集准确率）
    latency_p99_ms: float = 0.0 # P99推理延迟
    status: ModelStatus = ModelStatus.STANDBY
    created_at: str = ""         # 创建时间
    deployed_at: str = ""        # 部署时间
    deploy_ratio: float = 0.0    # 灰度发布比例（0-1）
    metadata: Dict = field(default_factory=dict) # 额外元数据

@dataclass
class ModelRegistry:
    """模型注册表条目"""
    name: str                    # 模型名称
    description: str              # 模型描述
    current_version: Optional[str] = None # 当前活跃版本
    previous_version: Optional[str] = None # 上一版本（用于回滚）
    versions: Dict[str, ModelVersion] = field(default_factory=dict)

# ===== 模型仓库客户端 =====

class ModelRepositoryClient:
    """
    模型仓库客户端
    对接MinIO对象存储作为模型文件仓库
    支持模型文件的上传、下载、版本列表查询
    模型文件AES-256加密存储（安全设计）
    """

    def __init__(self, endpoint: str = "minio.writech.internal:9000",
                  access_key: str = "", secret_key: str = "",
                  bucket: str = "model-repository"):
        self._endpoint = endpoint
        self._bucket = bucket

```

```

self._access_key = access_key
self._secret_key = secret_key
# 本地缓存目录
self._cache_dir = Path("/opt/models/cache")
self._cache_dir.mkdir(parents=True, exist_ok=True)
logger.info(f"模型仓库客户端初始化: endpoint={endpoint}, bucket={bucket}")

def download_model(self, model_name: str, version: str,
                  target_path: str) -> bool:
    """
    从MinIO仓库下载模型文件到本地
    下载完成后进行SHA-256完整性校验
    """
    object_key = f"{model_name}/{version}/model.onnx"
    logger.info(f"开始下载模型: {object_key} -> {target_path}")

    try:
        # 实际环境中使用MinIO SDK下载
        # self._client.fget_object(self._bucket, object_key, target_path)

        # 模拟下载过程
        target = Path(target_path)
        target.parent.mkdir(parents=True, exist_ok=True)

        logger.info(f"模型文件下载完成: {object_key}")
        return True
    except Exception as e:
        logger.error(f"模型下载失败: {object_key}, 错误: {str(e)}")
        return False

def list_versions(self, model_name: str) -> List[str]:
    """查询模型所有可用版本"""
    logger.info(f"查询模型版本列表: {model_name}")
    # 实际环境中查询MinIO对象前缀
    return []

def compute_sha256(self, file_path: str) -> str:
    """计算文件SHA-256校验和"""
    sha256_hash = hashlib.sha256()
    try:
        with open(file_path, "rb") as f:
            for chunk in iter(lambda: f.read(8192), b''):
                sha256_hash.update(chunk)
        return sha256_hash.hexdigest()
    except FileNotFoundError:
        return ""

# ===== 模型加载器 =====

class ModelLoader:
    """
    模型加载器
    负责将模型文件加载到推理引擎中
    支持ONNX Runtime、TensorRT、PaddleLite等多种推理后端
    模型文件在内存中解密加载（安全设计：不在磁盘上暴露明文模型）
    """

```

```

SUPPORTED_FORMATS = ['.onnx', '.trt', '.nb', '.pdmodel']

def __init__(self, device: str = "gpu"):
    self._device = device
    self._loaded_models: Dict[str, object] = {} # 已加载的模型实例
    self._load_lock = threading.Lock()
    logger.info(f"模型加载器初始化: device={device}")

def load(self, model_path: str, model_name: str) -> bool:
    """
    加载模型文件到推理引擎
    支持多格式自动识别和加载
    """
    with self._load_lock:
        try:
            path = Path(model_path)
            if not path.exists():
                logger.error(f"模型文件不存在: {model_path}")
                return False

            suffix = path.suffix.lower()
            if suffix not in self.SUPPORTED_FORMATS:
                logger.error(f"不支持的模型格式: {suffix}")
                return False

            logger.info(f"正在加载模型: {model_name} ({model_path})")

            # 根据格式选择推理后端
            if suffix == '.onnx':
                # 使用ONNX Runtime加载
                # session = onnxruntime.InferenceSession(model_path, providers=
['CUDAExecutionProvider'])
                # self._loaded_models[model_name] = session
                pass
            elif suffix == '.trt':
                # 使用TensorRT加载
                # engine = trt.Runtime(trt.Logger()).deserialize_cuda_engine(...)
                pass
            elif suffix == '.pdmodel':
                # 使用PaddleLite加载
                pass

            self._loaded_models[model_name] = {"path": model_path, "loaded_at":
time.time()}

            logger.info(f"模型加载成功: {model_name}")
            return True
        except Exception as e:
            logger.error(f"模型加载失败: {model_name}, 错误: {str(e)}")
            return False

def unload(self, model_name: str) -> bool:
    """卸载已加载的模型, 释放GPU显存"""
    with self._load_lock:
        if model_name in self._loaded_models:
            del self._loaded_models[model_name]
            logger.info(f"模型已卸载: {model_name}")

```

```

        return True
    return False

def is_loaded(self, model_name: str) -> bool:
    """检查模型是否已加载"""
    return model_name in self._loaded_models

def get_loaded_models(self) -> List[str]:
    """获取所有已加载模型名称"""
    return list(self._loaded_models.keys())

# ===== 模型版本管理器 =====

class ModelManager:
    """
    模型版本管理器（核心类）
    管理所有AI推理模型版本生命周期：
    注册 → 下载 → 加载 → 部署 → 灰度 → 全量 → 废弃
    支持热更新（零停机模型切换）和秒级回滚
    """

    def __init__(self, models_dir: str = "/opt/models"):
        self._models_dir = Path(models_dir)
        self._models_dir.mkdir(parents=True, exist_ok=True)
        self._registry: Dict[str, ModelRegistry] = {}
        self._repo_client = ModelRepositoryClient()
        self._loader = ModelLoader()
        self._deploy_lock = threading.Lock()
        logger.info(f"模型版本管理器初始化: models_dir={models_dir}")

    def register_model(self, name: str, description: str) -> ModelRegistry:
        """注册新模型类别"""
        if name not in self._registry:
            self._registry[name] = ModelRegistry(name=name, description=description)
            logger.info(f"注册新模型: {name} - {description}")
        return self._registry[name]

    def add_version(self, model_name: str, version: str,
                    accuracy: float = 0.0, metadata: Dict = None) ->
Optional[ModelVersion]:
        """
        添加新的模型版本
        从模型仓库下载文件并注册到本地
        """
        if model_name not in self._registry:
            logger.error(f"模型未注册: {model_name}")
            return None

        # 构建本地存储路径
        version_dir = self._models_dir / model_name / version
        model_file = str(version_dir / "model.onnx")

        # 从MinIO下载模型文件
        mv = ModelVersion(
            model_name=model_name, version=version,
            file_path=model_file, accuracy=accuracy,

```

```

        status=ModelStatus.DOWNLOADING,
        created_at=datetime.now().isoformat(),
        metadata=metadata or {}
    )

    success = self._repo_client.download_model(model_name, version, model_file)
    if success:
        mv.sha256 = self._repo_client.compute_sha256(model_file)
        mv.status = ModelStatus.STANDBY
        self._registry[model_name].versions[version] = mv
        logger.info(f"模型版本添加成功: {model_name}@{version}")
    else:
        mv.status = ModelStatus.FAILED
        logger.error(f"模型版本添加失败: {model_name}@{version}")

    return mv

def deploy_version(self, model_name: str, version: str,
                    strategy: DeployStrategy = DeployStrategy.IMMEDIATE,
                    canary_ratio: float = 0.1) -> bool:
    """
    部署指定版本的模型
    支持多种部署策略: 立即全量、金丝雀灰度、蓝绿部署
    """
    with self._deploy_lock:
        registry = self._registry.get(model_name)
        if not registry or version not in registry.versions:
            logger.error(f"模型版本不存在: {model_name}@{version}")
            return False

        mv = registry.versions[version]

        # 加载新版本模型
        load_key = f"{model_name}_v{version}"
        if not self._loader.load(mv.file_path, load_key):
            mv.status = ModelStatus.FAILED
            return False

        if strategy == DeployStrategy.IMMEDIATE:
            # 立即全量切换
            old_version = registry.current_version
            registry.previous_version = old_version
            registry.current_version = version
            mv.status = ModelStatus.ACTIVE
            mv.deploy_ratio = 1.0
            mv.deployed_at = datetime.now().isoformat()

            # 卸载旧版本
            if old_version:
                old_key = f"{model_name}_v{old_version}"
                self._loader.unload(old_key)
                if old_version in registry.versions:
                    registry.versions[old_version].status = ModelStatus.DEPRECATED

            logger.info(f"模型全量部署完成: {model_name}@{version}")

        elif strategy == DeployStrategy.CANARY:

```



```

        # 金丝雀灰度发布: 新版本接收部分流量
        mv.status = ModelState.ACTIVE
        mv.deploy_ratio = canary_ratio
        mv.deployed_at = datetime.now().isoformat()
        logger.info(f"模型灰度发布: {model_name}@{version}, 流量比例={canary_ratio}")

    return True

def rollback(self, model_name: str) -> bool:
    """
    回滚到上一版本 (秒级回滚)
    将当前版本标记为废弃, 恢复上一活跃版本
    """
    registry = self._registry.get(model_name)
    if not registry or not registry.previous_version:
        logger.error(f"无法回滚: {model_name}, 没有可回滚的版本")
        return False

    return self.deploy_version(
        model_name, registry.previous_version,
        strategy=DeployStrategy.IMMEDIATE
    )

def get_model_status(self) -> List[Dict]:
    """
    查询所有模型的当前状态
    GET /api/v1/model/status 接口的数据源
    """
    status_list = []
    for name, registry in self._registry.items():
        for ver, mv in registry.versions.items():
            status_list.append({
                "model_name": name,
                "version": ver,
                "status": mv.status.value,
                "accuracy": mv.accuracy,
                "latency_p99_ms": mv.latency_p99_ms,
                "deploy_ratio": mv.deploy_ratio,
                "is_current": ver == registry.current_version,
                "deployed_at": mv.deployed_at
            })
    return status_list

def check_for_updates(self) -> List[Dict]:
    """
    检查模型仓库是否有新版本可用
    定期调用此方法实现模型自动更新
    """
    updates = []
    for name, registry in self._registry.items():
        remote_versions = self._repo_client.list_versions(name)
        local_versions = set(registry.versions.keys())
        new_versions = [v for v in remote_versions if v not in local_versions]
        if new_versions:
            updates.append({
                "model_name": name,

```

```

        "new_versions": new_versions,
        "current_version": registry.current_version
    })
    return updates

```

service/task_scheduler.py

```

# 自然写手写识别与AI分析引擎软件 V1.0
# Celery异步任务调度模块 - 识别请求异步处理与优先级调度

"""
Celery任务调度服务
管理AI识别请求的异步任务队列，支持优先级调度、任务重试、
结果回调通知、任务进度追踪等功能
使用Redis作为消息Broker和结果Backend
"""

import time
import json
import logging
import uuid
from typing import Dict, List, Optional, Any
from dataclasses import dataclass, field
from datetime import datetime, timedelta
from enum import IntEnum

logger = logging.getLogger(__name__)

# ===== 任务优先级定义 =====

class TaskPriority(IntEnum):
    """任务优先级（数值越小优先级越高）"""
    CRITICAL = 0    # 最高优先级：课堂实时互动场景
    HIGH = 1        # 高优先级：教师在线批改
    NORMAL = 2      # 普通优先级：作业自动批改
    LOW = 3          # 低优先级：批量历史数据处理
    BACKGROUND = 4  # 后台优先级：模型评估/训练数据生成

class TaskStatus:
    """任务状态常量"""
    PENDING = "PENDING"    # 等待执行
    STARTED = "STARTED"    # 已开始执行
    PROCESSING = "PROCESSING" # 处理中
    SUCCESS = "SUCCESS"    # 执行成功
    FAILURE = "FAILURE"    # 执行失败
    RETRY = "RETRY"        # 重试中
    REVOKED = "REVOKED"    # 已取消

@dataclass
class TaskRecord:
    """任务记录"""
    task_id: str

```

```

task_type: str # 任务类型 (ocr/math/stroke_order/essay)
priority: TaskPriority
status: str = TaskStatus.PENDING
input_data: Dict = field(default_factory=dict)
result: Optional[Dict] = None
error_message: Optional[str] = None
retry_count: int = 0
max_retries: int = 3
created_at: str = ""
started_at: Optional[str] = None
completed_at: Optional[str] = None
callback_url: Optional[str] = None # 完成后回调通知URL
student_id: Optional[str] = None
assignment_id: Optional[str] = None

# ===== 任务队列管理器 =====

class TaskQueueManager:
    """
    任务队列管理器
    管理多个优先级队列，确保高优先级任务（如课堂实时互动）优先处理
    使用Redis有序集合(ZSET)实现优先级调度
    """

    # 各任务类型的默认队列名
    QUEUE_MAPPING = {
        "ocr": "writech.ocr",
        "math": "writech.math",
        "stroke_order": "writech.stroke_order",
        "essay": "writech.essay",
        "batch": "writech.batch"
    }

    def __init__(self, redis_url: str = "redis://localhost:6379/0"):
        self._redis_url = redis_url
        self._tasks: Dict[str, TaskRecord] = {} # 内存任务记录（生产环境用Redis）
        self._queue: List[TaskRecord] = [] # 优先级队列
        logger.info(f"任务队列管理器初始化: redis={redis_url}")

    def submit_task(self, task_type: str, input_data: Dict,
                    priority: TaskPriority = TaskPriority.NORMAL,
                    callback_url: Optional[str] = None,
                    student_id: Optional[str] = None,
                    assignment_id: Optional[str] = None) -> str:
        """
        提交识别任务到队列
        返回任务ID，调用方可通过ID查询任务状态和结果
        """
        task_id = str(uuid.uuid4())

        record = TaskRecord(
            task_id=task_id,
            task_type=task_type,
            priority=priority,
            input_data=input_data,
            created_at=datetime.now().isoformat(),

```

```

        callback_url=callback_url,
        student_id=student_id,
        assignment_id=assignment_id
    )

    self._tasks[task_id] = record
    self._queue.append(record)
    # 按优先级排序（数值小的排在前面）
    self._queue.sort(key=lambda t: (t.priority, t.created_at))

    queue_name = self.QUEUE_MAPPING.get(task_type, "writech.default")
    logger.info(
        f"任务已提交: id={task_id}, type={task_type}, "
        f"priority={priority.name}, queue={queue_name}"
    )
    return task_id

def get_next_task(self) -> Optional[TaskRecord]:
    """获取队列中优先级最高的待执行任务"""
    for task in self._queue:
        if task.status == TaskStatus.PENDING:
            task.status = TaskStatus.STARTED
            task.started_at = datetime.now().isoformat()
            return task
    return None

def update_task_status(self, task_id: str, status: str,
                       result: Optional[Dict] = None,
                       error: Optional[str] = None):
    """更新任务状态"""
    if task_id in self._tasks:
        task = self._tasks[task_id]
        task.status = status
        if result:
            task.result = result
        if error:
            task.error_message = error
        if status in (TaskStatus.SUCCESS, TaskStatus.FAILURE):
            task.completed_at = datetime.now().isoformat()
        logger.info(f"任务状态更新: id={task_id}, status={status}")

def get_task_status(self, task_id: str) -> Optional[Dict]:
    """查询任务状态和结果"""
    task = self._tasks.get(task_id)
    if not task:
        return None
    return {
        "task_id": task.task_id,
        "task_type": task.task_type,
        "status": task.status,
        "priority": task.priority.name,
        "result": task.result,
        "error_message": task.error_message,
        "retry_count": task.retry_count,
        "created_at": task.created_at,
        "started_at": task.started_at,
        "completed_at": task.completed_at
    }

```

```

    }

def get_queue_stats(self) -> Dict:
    """获取队列统计信息"""
    stats = {"total": len(self._tasks)}
    for status in [TaskStatus.PENDING, TaskStatus.STARTED,
                   TaskStatus.SUCCESS, TaskStatus.FAILURE]:
        stats[status.lower()] = sum(
            1 for t in self._tasks.values() if t.status == status
        )
    return stats

# ===== Celery任务定义 =====

class CeleryTaskExecutor:
    """
    Celery任务执行器
    定义各类AI识别的Celery异步任务
    每个任务类型对应一个独立的任务函数和执行队列
    """

    def __init__(self, queue_manager: TaskQueueManager):
        self._queue_manager = queue_manager
        self._task_handlers: Dict[str, callable] = {}
        logger.info("Celery任务执行器初始化")

    def register_handler(self, task_type: str, handler: callable):
        """注册任务处理函数"""
        self._task_handlers[task_type] = handler
        logger.info(f"注册任务处理器: {task_type}")

    def execute_task(self, task_id: str) -> Dict:
        """
        执行指定任务
        包含异常处理、重试逻辑、超时控制
        """
        task = self._queue_manager._tasks.get(task_id)
        if not task:
            return {"error": "任务不存在"}

        handler = self._task_handlers.get(task.task_type)
        if not handler:
            self._queue_manager.update_task_status(
                task_id, TaskStatus.FAILURE,
                error=f"未注册的任务类型: {task.task_type}"
            )
            return {"error": f"未注册的任务类型: {task.task_type}"}

        try:
            self._queue_manager.update_task_status(task_id, TaskStatus.PROCESSING)

            # 执行推理任务
            start_time = time.time()
            result = handler(task.input_data)
            elapsed = (time.time() - start_time) * 1000

```

```

        result['processing_time_ms'] = round(elapsed, 2)
        self._queue_manager.update_task_status(task_id, TaskStatus.SUCCESS,
result=result)

        # 审计日志记录（安全设计：所有识别请求记录调用方、时间）
        logger.info(
            f"任务执行完成: id={task_id}, type={task.task_type}, "
            f"time={elapsed:.1f}ms, student={task.student_id}"
        )

        # 如有回调URL则通知调用方
        if task.callback_url:
            self._send_callback(task.callback_url, task_id, result)

        return result

    except Exception as e:
        task.retry_count += 1
        if task.retry_count < task.max_retries:
            # 重试：将任务重新加入队列
            task.status = TaskStatus.RETRY
            logger.warning(f"任务重试: id={task_id}, retry=
{task.retry_count}/{task.max_retries}")
        else:
            self._queue_manager.update_task_status(
                task_id, TaskStatus.FAILURE, error=str(e)
            )
            logger.error(f"任务最终失败: id={task_id}, error={str(e)}")
            return {"error": str(e)}

    def _send_callback(self, url: str, task_id: str, result: Dict):
        """发送任务完成回调通知"""
        try:
            # 实际环境使用httpx/aiohttp发送POST请求
            logger.info(f"发送任务回调: url={url}, task_id={task_id}")
        except Exception as e:
            logger.error(f"回调通知失败: {str(e)}")

# ===== 定时调度器 =====

class ScheduledTaskRunner:
    """
    定时任务调度器
    管理周期性执行的后台任务，如：
    - 模型健康检查（每5分钟）
    - 过期任务清理（每小时）
    - 性能指标采集（每分钟）
    - 模型更新检查（每天）
    """

    def __init__(self):
        self._schedules: Dict[str, Dict] = {}
        self._running = False
        logger.info("定时任务调度器初始化")

    def register_schedule(self, name: str, interval_seconds: int,

```

```

        handler: callable, description: str = ""):
    """注册定时任务"""
    self._schedules[name] = {
        "interval": interval_seconds,
        "handler": handler,
        "description": description,
        "last_run": None,
        "run_count": 0,
        "error_count": 0
    }
    logger.info(f"注册定时任务: {name}, 间隔={interval_seconds}s")

def run_task(self, name: str) -> Optional[Dict]:
    """立即执行指定的定时任务"""
    schedule = self._schedules.get(name)
    if not schedule:
        return None

    try:
        start = time.time()
        result = schedule["handler"]()
        elapsed = time.time() - start
        schedule["last_run"] = datetime.now().isoformat()
        schedule["run_count"] += 1
        logger.info(f"定时任务执行完成: {name}, 耗时={elapsed:.2f}s")
        return {"name": name, "success": True, "elapsed_s": round(elapsed, 2)}
    except Exception as e:
        schedule["error_count"] += 1
        logger.error(f"定时任务执行失败: {name}, 错误={str(e)}")
        return {"name": name, "success": False, "error": str(e)}

def get_schedule_status(self) -> List[Dict]:
    """获取所有定时任务状态"""
    return [{
        "name": name,
        "interval_seconds": info["interval"],
        "description": info["description"],
        "last_run": info["last_run"],
        "run_count": info["run_count"],
        "error_count": info["error_count"]
    } for name, info in self._schedules.items()]

```