

自然写教学数据分析与学情诊断系统软件 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

```
03-writech-learning-analytics/  
├── main.py  
├── analytics/  
│   ├── knowledge_graph.py  
│   ├── student_profiler.py  
│   └── writing_growth.py  
├── api/  
│   ├── profile_api.py  
│   └── report_api.py  
├── etl/  
│   └── flink_processor.py  
└── report/  
    └── report_generator.py
```

源程序文件清单

(根目录)

main.py

```
# 自然写教学数据分析与学情诊断系统软件 V1.0  
# main.py - 服务启动入口 (FastAPI + 定时任务调度)  
  
import os  
import sys  
import logging  
import asyncio  
from typing import Optional
```

```

from datetime import datetime
from contextlib import asynccontextmanager

from fastapi import FastAPI, Request, Response
from fastapi.middleware.cors import CORSMiddleware
from fastapi.middleware.trustedhost import TrustedHostMiddleware
from fastapi.responses import JSONResponse
import uvicorn

# =====
# 日志配置
# =====

LOG_FORMAT = (
    "%(asctime)s | %(levelname)-8s | %(name)s:%(lineno)d | %(message)s"
)

def setup_logging(log_level: str = "INFO") -> None:
    """初始化日志系统，同时输出到控制台和文件"""
    logging.basicConfig(
        level=getattr(logging, log_level.upper(), logging.INFO),
        format=LOG_FORMAT,
        handlers=[
            logging.StreamHandler(sys.stdout),
            logging.FileHandler(
                "logs/analytics.log", encoding="utf-8", mode="a"
            ),
        ],
    )

logger = logging.getLogger("writech.analytics")

# =====
# 全局配置
# =====

class AnalyticsConfig:
    """学情系统全局配置"""

    # 服务基本配置
    SERVICE_NAME: str = "writech-learning-analytics"
    SERVICE_VERSION: str = "1.0.0"
    HOST: str = os.getenv("ANALYTICS_HOST", "0.0.0.0")
    PORT: int = int(os.getenv("ANALYTICS_PORT", "8300"))
    DEBUG: bool = os.getenv("ANALYTICS_DEBUG", "false").lower() == "true"

    # 数据库连接配置
    CLICKHOUSE_HOST: str = os.getenv("CH_HOST", "localhost")
    CLICKHOUSE_PORT: int = int(os.getenv("CH_PORT", "9000"))
    CLICKHOUSE_DB: str = os.getenv("CH_DB", "writech_analytics")
    CLICKHOUSE_USER: str = os.getenv("CH_USER", "default")
    CLICKHOUSE_PASSWORD: str = os.getenv("CH_PASSWORD", "")

    MYSQL_HOST: str = os.getenv("MYSQL_HOST", "localhost")
    MYSQL_PORT: int = int(os.getenv("MYSQL_PORT", "3306"))
    MYSQL_DB: str = os.getenv("MYSQL_DB", "writech_analytics")

```

```

MYSQL_USER: str = os.getenv("MYSQL_USER", "root")
MYSQL_PASSWORD: str = os.getenv("MYSQL_PASSWORD", "")

# Neo4j知识图谱连接
NEO4J_URI: str = os.getenv("NEO4J_URI", "bolt://localhost:7687")
NEO4J_USER: str = os.getenv("NEO4J_USER", "neo4j")
NEO4J_PASSWORD: str = os.getenv("NEO4J_PASSWORD", "")

# Kafka配置
KAFKA_BROKERS: str = os.getenv("KAFKA_BROKERS", "localhost:9092")
KAFKA_TOPIC_STROKE: str = "writech.stroke.raw"
KAFKA_TOPIC_GRADE: str = "writech.grade.result"
KAFKA_GROUP_ID: str = "analytics-consumer-group"

# 报告生成配置
REPORT_OUTPUT_DIR: str = os.getenv("REPORT_DIR", "/data/reports")
REPORT_TEMPLATE_DIR: str = os.getenv(
    "TEMPLATE_DIR", "/data/templates"
)

# JWT鉴权密钥（与云平台共享）
JWT_SECRET: str = os.getenv("JWT_SECRET", "writech-jwt-secret-key")
JWT_ALGORITHM: str = "HS256"

# 定时任务配置
DAILY_REPORT_CRON: str = "0 2 * * *" # 每天凌晨2点
WEEKLY_REPORT_CRON: str = "0 3 * * 1" # 每周一凌晨3点

# =====
# 应用生命周期管理
# =====

@asynccontextmanager
async def lifespan(app: FastAPI):
    """应用启动和关闭时的资源管理"""
    logger.info(
        "正在启动 %s v%s ...",
        AnalyticsConfig.SERVICE_NAME,
        AnalyticsConfig.SERVICE_VERSION,
    )

# 启动时初始化各服务组件
try:
    # 初始化ClickHouse连接池
    logger.info("初始化ClickHouse连接: %s:%d",
                AnalyticsConfig.CLICKHOUSE_HOST,
                AnalyticsConfig.CLICKHOUSE_PORT)
    # await init_clickhouse_pool()

    # 初始化MySQL连接池
    logger.info("初始化MySQL连接: %s:%d",
                AnalyticsConfig.MYSQL_HOST,
                AnalyticsConfig.MYSQL_PORT)
    # await init_mysql_pool()

    # 初始化Neo4j驱动

```

```

        logger.info("初始化Neo4j连接: %s", AnalyticsConfig.NE04J_URI)
        # await init_neo4j_driver()

        # 启动Kafka消费者线程
        logger.info("启动Kafka消费者: %s", AnalyticsConfig.KAFKA_BROKERS)
        # start_kafka_consumers()

        # 注册定时任务调度
        logger.info("注册定时报告生成任务")
        # register_cron_jobs()

        logger.info("所有服务组件初始化完成")
    except Exception as e:
        logger.error("服务初始化失败: %s", str(e))
        raise

    yield

    # 关闭时释放资源
    logger.info("正在关闭服务...")
    # await close_clickhouse_pool()
    # await close_mysql_pool()
    # await close_neo4j_driver()
    # stop_kafka_consumers()
    logger.info("服务已安全关闭")

# =====
# FastAPI应用创建
# =====

app = FastAPI(
    title="自然写教学数据分析与学情诊断系统",
    description="对学生书写及答题数据进行大数据分析, 生成学情诊断报告",
    version=AnalyticsConfig.SERVICE_VERSION,
    lifespan=lifespan,
)

# CORS中间件 (允许管理前端跨域访问)
app.add_middleware(
    CORSMiddleware,
    allow_origins=[
        "https://admin.writech.com",
        "https://teacher.writech.com",
    ],
    allow_credentials=True,
    allow_methods=["GET", "POST", "PUT"],
    allow_headers=["*"],
)

# 可信主机校验
app.add_middleware(
    TrustedHostMiddleware,
    allowed_hosts=["*.writech.com", "localhost"],
)

```

```

# =====
# 全局中间件
# =====

@app.middleware("http")
async def audit_logging_middleware(request: Request, call_next):
    """审计日志中间件：记录所有数据查询与导出操作"""
    start_time = datetime.now()
    request_id = request.headers.get("X-Request-ID", "")

    # 执行请求
    response: Response = await call_next(request)

    # 计算耗时
    duration_ms = (datetime.now() - start_time).total_seconds() * 1000

    # 记录审计日志（数据查询和导出类接口）
    if request.url.path.startswith("/api/v1/"):
        logger.info(
            "AUDIT | %s | %s %s | status=%d | %.1fms | user=%s",
            request_id,
            request.method,
            request.url.path,
            response.status_code,
            duration_ms,
            request.headers.get("X-User-ID", "anonymous"),
        )

    return response

@app.middleware("http")
async def data_permission_middleware(request: Request, call_next):
    """数据权限中间件：教师仅查看本班数据，家长仅查看子女数据"""
    # 从JWT中提取用户角色和权限范围
    # token = request.headers.get("Authorization", "").replace("Bearer ", "")
    # user_info = decode_jwt(token)
    # role = user_info.get("role", "")
    #
    # 数据权限过滤规则：
    # - teacher： 仅可访问 class_ids 范围内的数据
    # - parent： 仅可访问 student_ids 范围内的数据
    # - admin： 可访问本校全部数据
    # - super_admin： 无限制

    response = await call_next(request)
    return response

# =====
# 路由注册
# =====

# 导入并注册各API路由模块
# from api.profile_api import router as profile_router
# from api.report_api import router as report_router
# from api.growth_api import router as growth_router

```

```

#
# app.include_router(profile_router, prefix="/api/v1/profile")
# app.include_router(report_router, prefix="/api/v1/report")
# app.include_router(growth_router, prefix="/api/v1/growth")

# =====
# 健康检查接口
# =====

@app.get("/health")
async def health_check():
    """健康检查端点, Kubernetes存活探针使用"""
    return {
        "status": "healthy",
        "service": AnalyticsConfig.SERVICE_NAME,
        "version": AnalyticsConfig.SERVICE_VERSION,
        "timestamp": datetime.now().isoformat(),
    }

@app.get("/ready")
async def readiness_check():
    """就绪检查端点, 确认所有依赖服务可用"""
    checks = {
        "clickhouse": False,
        "mysql": False,
        "neo4j": False,
        "kafka": False,
    }

    # 检查ClickHouse连接
    # try:
    #     await clickhouse_ping()
    #     checks["clickhouse"] = True
    # except Exception:
    #     pass

    all_ready = all(checks.values())
    return JSONResponse(
        status_code=200 if all_ready else 503,
        content={
            "ready": all_ready,
            "checks": checks,
        },
    )

# =====
# 全局异常处理
# =====

@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    """全局异常捕获, 返回统一错误格式"""
    logger.error(
        "未处理异常 | %s %s | %s: %s",
    )

```

```

        request.method,
        request.url.path,
        type(exc).__name__,
        str(exc),
    )
    return JSONResponse(
        status_code=500,
        content={
            "code": 500,
            "message": "服务内部错误",
            "detail": str(exc) if AnalyticsConfig.DEBUG else None,
        },
    )

# =====
# 启动入口
# =====

if __name__ == "__main__":
    # 确保日志目录存在
    os.makedirs("logs", exist_ok=True)
    os.makedirs(AnalyticsConfig.REPORT_OUTPUT_DIR, exist_ok=True)

    setup_logging("DEBUG" if AnalyticsConfig.DEBUG else "INFO")
    logger.info("启动学情诊断系统服务...")

    uvicorn.run(
        "main:app",
        host=AnalyticsConfig.HOST,
        port=AnalyticsConfig.PORT,
        reload=AnalyticsConfig.DEBUG,
        workers=4 if not AnalyticsConfig.DEBUG else 1,
        log_level="info",
    )

```

analytics/

analytics/knowledge_graph.py

```

# 自然写教学数据分析与学情诊断系统软件 V1.0
# analytics/knowledge_graph.py - Neo4j知识图谱查询与推理引擎

import logging
from typing import Any, Dict, List, Optional, Tuple
from dataclasses import dataclass, field

logger = logging.getLogger("writech.analytics.knowledge_graph")

# =====
# 知识图谱数据模型
# =====

```

```

@dataclass
class KnowledgeNode:
    """知识点节点"""
    node_id: str
    name: str
    subject: str
    grade: str
    chapter: str = ""
    section: str = ""
    difficulty: float = 0.5 # 难度系数 0-1
    importance: float = 0.5 # 重要程度 0-1
    description: str = ""

@dataclass
class KnowledgeEdge:
    """知识点关系边"""
    source_id: str
    target_id: str
    relation_type: str # prerequisite/includes/related
    weight: float = 1.0

@dataclass
class StudentMastery:
    """学生对某知识点的掌握度"""
    student_id: str
    knowledge_id: str
    mastery_level: float = 0.0 # 掌握度 0-1
    practice_count: int = 0
    correct_count: int = 0
    error_count: int = 0
    last_practice: str = ""

@dataclass
class ErrorAttribution:
    """错题归因结果"""
    question_id: str
    error_knowledge_ids: List[str] # 直接关联知识点
    root_cause_ids: List[str] # 根因知识点（前驱未掌握）
    suggestion: str = ""

# =====
# 知识图谱引擎
# =====

class KnowledgeGraphEngine:
    """
    Neo4j知识图谱引擎

    负责：
    1. 知识点图谱的查询与遍历
    2. 错题归因推理（追溯前驱知识点）
    3. 学习路径推荐
    """

```

4. 知识点掌握度聚合计算

```
"""
def __init__(self, uri: str, user: str, password: str):
    """初始化Neo4j连接"""
    self.uri = uri
    self.user = user
    self.password = password
    # self._driver = GraphDatabase.driver(uri, auth=(user, password))
    logger.info("知识图谱引擎初始化: %s", uri)

async def query_subject_graph(
    self, subject: str, grade: Optional[str] = None
) -> Tuple[List[KnowledgeNode], List[KnowledgeEdge]]:
    """
    查询某科目的完整知识图谱结构

    Args:
        subject: 科目名称
        grade: 可选年级过滤

    Returns:
        (节点列表, 边列表)
    """
    logger.info("查询知识图谱: subject=%s, grade=%s", subject, grade)

    # Cypher查询: 获取所有知识点节点
    node_query = """
    MATCH (k:KnowledgePoint {subject: $subject})
    WHERE ($grade IS NULL OR k.grade = $grade)
    RETURN k.id AS id, k.name AS name, k.subject AS subject,
           k.grade AS grade, k.chapter AS chapter, k.section AS section,
           k.difficulty AS difficulty, k.importance AS importance,
           k.description AS description
    ORDER BY k.chapter, k.section
    """

    # Cypher查询: 获取所有关系边
    edge_query = """
    MATCH (a:KnowledgePoint {subject: $subject})-[r]->(b:KnowledgePoint)
    WHERE ($grade IS NULL OR a.grade = $grade)
    RETURN a.id AS source, b.id AS target, type(r) AS relation,
           r.weight AS weight
    """

    nodes: List[KnowledgeNode] = []
    edges: List[KnowledgeEdge] = []

    # async with self._driver.async_session() as session:
    #     # 查询节点
    #     result = await session.run(node_query, subject=subject, grade=grade)
    #     async for record in result:
    #         nodes.append(KnowledgeNode(
    #             node_id=record["id"],
    #             name=record["name"],
    #             ...
    #         ))
```

```

#
# # 查询边
# result = await session.run(edge_query, subject=subject, grade=grade)
# async for record in result:
#     edges.append(KnowledgeEdge(
#         source_id=record["source"],
#         target_id=record["target"],
#         relation_type=record["relation"],
#         weight=record["weight"] or 1.0,
#     ))

logger.info(
    "图谱查询完成: %d节点, %d边", len(nodes), len(edges)
)
return nodes, edges

async def query_prerequisites(
    self, knowledge_id: str, max_depth: int = 3
) -> List[KnowledgeNode]:
    """
    查询知识点的前驱依赖链（递归向上追溯）

    用于错题归因：当某知识点未掌握时，追溯其前驱
    知识点是否也未掌握，找到根本原因。

    Args:
        knowledge_id: 目标知识点ID
        max_depth: 最大追溯深度

    Returns:
        前驱知识点列表（按依赖顺序排列）
    """
    query = """
    MATCH path = (target:KnowledgePoint {id: $kid})
    <-[:PREREQUISITE*1..$depth]-(prereq:KnowledgePoint)
    RETURN prereq.id AS id, prereq.name AS name,
           prereq.subject AS subject, prereq.grade AS grade,
           prereq.chapter AS chapter, prereq.difficulty AS difficulty,
           length(path) AS distance
    ORDER BY distance ASC
    """

    prerequisites: List[KnowledgeNode] = []
    # async with self._driver.async_session() as session:
    #     result = await session.run(
    #         query, kid=knowledge_id, depth=max_depth
    #     )
    #     async for record in result:
    #         prerequisites.append(KnowledgeNode(
    #             node_id=record["id"],
    #             name=record["name"],
    #             ...
    #         ))

    logger.debug(
        "知识点 %s 的前驱链: %d个",
        knowledge_id,
    
```

```

        len(prerequisites),
    )
    return prerequisites

async def attribute_errors(
    self,
    student_id: str,
    error_question_ids: List[str],
    mastery_map: Dict[str, float],
) -> List[ErrorAttribution]:
    """
    错题归因分析

    对每道错题：
    1. 查找该题关联的知识点
    2. 查找这些知识点的前驱知识点
    3. 检查前驱知识点的掌握度
    4. 如果前驱也未掌握，则认为是根因

    Args:
        student_id: 学生ID
        error_question_ids: 错题ID列表
        mastery_map: {knowledge_id: mastery_level} 掌握度映射

    Returns:
        错题归因结果列表
    """
    logger.info(
        "错题归因: student=%s, 错题数=%d",
        student_id,
        len(error_question_ids),
    )

    attributions: List[ErrorAttribution] = []
    mastery_threshold = 0.6 # 掌握度阈值（低于此视为未掌握）

    for question_id in error_question_ids:
        # 查询错题关联的知识点
        # question_kps = await self._query_question_knowledge(question_id)
        question_kps: List[str] = []

        root_causes: List[str] = []

        for kp_id in question_kps:
            mastery = mastery_map.get(kp_id, 0.0)

            if mastery < mastery_threshold:
                # 该知识点未掌握，追溯前驱
                prereqs = await self.query_prerequisites(kp_id)

                for prereq in prereqs:
                    prereq_mastery = mastery_map.get(
                        prereq.node_id, 0.0
                    )
                    if prereq_mastery < mastery_threshold:
                        # 前驱也未掌握，记为根因
                        if prereq.node_id not in root_causes:

```

```

        root_causes.append(prereq.node_id)

    # 生成归因建议
    suggestion = self._generate_suggestion(
        question_kps, root_causes, mastery_map
    )

    attributions.append(ErrorAttribution(
        question_id=question_id,
        error_knowledge_ids=question_kps,
        root_cause_ids=root_causes,
        suggestion=suggestion,
    ))

    return attributions

def _generate_suggestion(
    self,
    knowledge_ids: List[str],
    root_cause_ids: List[str],
    mastery_map: Dict[str, float],
) -> str:
    """根据归因结果生成学习建议"""
    if root_cause_ids:
        return (
            f"建议先复习前驱知识点 (共{len(root_cause_ids)}个), "
            f"夯实基础后再针对性练习当前知识点"
        )
    elif knowledge_ids:
        avg_mastery = sum(
            mastery_map.get(k, 0) for k in knowledge_ids
        ) / max(len(knowledge_ids), 1)
        if avg_mastery < 0.3:
            return "该知识点掌握度较低, 建议从基础概念开始系统学习"
        elif avg_mastery < 0.6:
            return "该知识点已有一定基础, 建议加强专项练习巩固提升"
        else:
            return "知识点掌握较好, 本次错误可能是粗心或审题不清"
    return "暂无具体建议"

async def recommend_learning_path(
    self,
    student_id: str,
    target_knowledge_id: str,
    mastery_map: Dict[str, float],
) -> List[KnowledgeNode]:
    """
    学习路径推荐

    基于知识图谱拓扑排序, 为学生推荐从当前水平到
    目标知识点的最优学习路径。

    原则:
    1. 先补足未掌握的前驱知识点
    2. 按难度从低到高排序
    3. 已掌握的知识点可跳过
    """

```

```

# 获取目标知识点的所有前驱
all_prereqs = await self.query_prerequisites(
    target_knowledge_id, max_depth=5
)

# 过滤出未掌握的前驱知识点
unmastered = [
    node for node in all_prereqs
    if mastery_map.get(node.node_id, 0.0) < 0.6
]

# 按难度从低到高排序
unmastered.sort(key=lambda n: n.difficulty)

# 添加目标知识点本身
# target_node = await self._get_knowledge_node(target_knowledge_id)
# if target_node:
#     unmastered.append(target_node)

logger.info(
    "学习路径推荐: student=%s, target=%s, 路径长度=%d",
    student_id,
    target_knowledge_id,
    len(unmastered),
)

return unmastered

async def aggregate_chapter_mastery(
    self,
    student_id: str,
    subject: str,
    mastery_map: Dict[str, float],
) -> List[Dict[str, Any]]:
    """
    按章节聚合知识点掌握度

    将知识图谱按章节分组，计算每章的综合掌握度，
    用于生成章节维度的学情雷达图。
    """
    nodes, _ = await self.query_subject_graph(subject)

    # 按章节分组
    chapter_map: Dict[str, List[float]] = {}
    for node in nodes:
        chapter = node.chapter or "其他"
        mastery = mastery_map.get(node.node_id, 0.0)
        chapter_map.setdefault(chapter, []).append(mastery)

    # 计算各章节平均掌握度
    result = []
    for chapter, masteries in chapter_map.items():
        avg_mastery = sum(masteries) / max(len(masteries), 1)
        result.append({
            "chapter": chapter,
            "avg_mastery": round(avg_mastery, 3),
            "knowledge_count": len(masteries),
        })

```

```

        "mastered_count": sum(1 for m in masteries if m >= 0.6),
    })

    result.sort(key=lambda x: x["chapter"])
    return result

async def close(self) -> None:
    """关闭Neo4j连接"""
    # await self._driver.close()
    logger.info("知识图谱引擎已关闭")

```

analytics/student_profiler.py

```

# 自然写教学数据分析与学情诊断系统软件 V1.0
# analytics/student_profiler.py - 学生画像分析引擎

import logging
import math
from typing import Any, Dict, List, Optional, Tuple
from datetime import datetime, date, timedelta
from dataclasses import dataclass, field

logger = logging.getLogger("writech.analytics.profiler")

# =====
# 画像分析数据模型
# =====

@dataclass
class ScoreTrend:
    """成绩趋势数据点"""
    date: str
    score: float
    subject: str
    exam_type: str = "" # homework/exam/practice

@dataclass
class SubjectAbility:
    """科目能力评估"""
    subject: str
    overall_score: float = 0.0
    knowledge_coverage: float = 0.0 # 知识点覆盖率
    practice_frequency: float = 0.0 # 练习频率 (次/周)
    improvement_rate: float = 0.0 # 进步速率
    stability: float = 0.0 # 稳定性 (分数方差的倒数)

@dataclass
class LearningHabit:
    """学习习惯画像"""
    avg_daily_minutes: float = 0.0
    peak_study_hour: int = 0 # 学习高峰时段 (小时)

```

```

weekly_pattern: List[float] = field(default_factory=list) # 周一~日时长
consistency_score: float = 0.0 # 学习规律性评分
homework_timeliness: float = 0.0 # 作业及时提交率

@dataclass
class WritingAbility:
    """书写能力评估"""
    stroke_order_accuracy: float = 0.0 # 笔顺正确率
    writing_quality: float = 0.0 # 书写规范性
    writing_speed: float = 0.0 # 书写速度(字/分)
    char_structure_score: float = 0.0 # 字形结构评分
    improvement_trend: str = "stable" # 进步趋势

@dataclass
class ComprehensiveProfile:
    """综合学情画像"""
    student_id: str
    student_name: str
    class_id: str
    grade: str
    school_id: str

    # 综合评分
    overall_score: float = 0.0
    rank_in_class: int = 0
    rank_in_grade: int = 0
    percentile: float = 0.0

    # 各科能力
    subject_abilities: List[SubjectAbility] = field(default_factory=list)

    # 学习习惯
    learning_habit: Optional[LearningHabit] = None

    # 书写能力
    writing_ability: Optional[WritingAbility] = None

    # 成绩趋势
    score_trends: List[ScoreTrend] = field(default_factory=list)

    # 分析时间
    analyzed_at: str = ""

# =====
# 画像分析引擎
# =====

class StudentProfiler:
    """
    学生画像分析引擎

    功能：
    1. 综合学情评分计算
    2. 各科目能力多维评估

```

3. 学习习惯分析
 4. 书写能力评估
 5. 成绩趋势分析与预测
 6. 班级/年级排名计算
- """

各维度权重（用于综合评分计算）

```
WEIGHT_HOMEWORK_SCORE = 0.30    # 作业成绩权重
WEIGHT_EXAM_SCORE = 0.35         # 考试成绩权重
WEIGHT_PRACTICE = 0.15           # 练习表现权重
WEIGHT_WRITING = 0.10            # 书写能力权重
WEIGHT_HABIT = 0.10              # 学习习惯权重
```

评分标准

```
EXCELLENT_THRESHOLD = 90.0
GOOD_THRESHOLD = 75.0
PASS_THRESHOLD = 60.0
```

def __init__(self):

"""初始化画像分析引擎"""

logger.info("学生画像分析引擎初始化")

async def build_profile(

```
    self,
    student_id: str,
    student_info: Dict[str, Any],
    period_days: int = 30,
```

) -> ComprehensiveProfile:

"""

构建学生综合画像

Args:

```
    student_id: 学生ID
    student_info: 学生基本信息
    period_days: 分析周期（天）
```

Returns:

综合学情画像

"""

logger.info(

"构建学生画像: %s, 分析周期=%d天", student_id, period_days

)

end_date = date.today()

start_date = end_date - timedelta(days=period_days)

1. 获取原始数据

```
homework_data = await self._fetch_homework_data(
    student_id, start_date, end_date
)
```

```
exam_data = await self._fetch_exam_data(
    student_id, start_date, end_date
)
```

```
practice_data = await self._fetch_practice_data(
    student_id, start_date, end_date
)
```

```
writing_data = await self._fetch_writing_data(
```

```

        student_id, start_date, end_date
    )
    usage_data = await self._fetch_usage_data(
        student_id, start_date, end_date
    )

# 2. 分析各维度
subject_abilities = self._analyze_subject_abilities(
    homework_data, exam_data, practice_data
)
learning_habit = self._analyze_learning_habit(usage_data)
writing_ability = self._analyze_writing_ability(writing_data)
score_trends = self._analyze_score_trends(
    homework_data, exam_data
)

# 3. 计算综合评分
overall_score = self._calculate_overall_score(
    subject_abilities, learning_habit, writing_ability
)

# 4. 计算排名
rank_in_class, rank_in_grade, percentile = (
    await self._calculate_rankings(
        student_id,
        student_info.get("class_id", ""),
        student_info.get("grade", ""),
        overall_score,
    )
)

profile = ComprehensiveProfile(
    student_id=student_id,
    student_name=student_info.get("name", ""),
    class_id=student_info.get("class_id", ""),
    grade=student_info.get("grade", ""),
    school_id=student_info.get("school_id", ""),
    overall_score=round(overall_score, 1),
    rank_in_class=rank_in_class,
    rank_in_grade=rank_in_grade,
    percentile=round(percentile, 1),
    subject_abilities=subject_abilities,
    learning_habit=learning_habit,
    writing_ability=writing_ability,
    score_trends=score_trends,
    analyzed_at=datetime.now().isoformat(),
)

# 5. 写入ClickHouse画像宽表
await self._save_profile(profile)

logger.info(
    "画像构建完成: %s, 综合评分=%.1f, 班级排名=%d",
    student_id, overall_score, rank_in_class,
)

return profile

```

```

async def _fetch_homework_data(
    self, student_id: str, start: date, end: date
) -> List[Dict[str, Any]]:
    """从ClickHouse获取作业成绩数据"""
    # query = """
    #     SELECT subject, score, total_score, submitted_at, is_on_time
    #     FROM homework_submissions
    #     WHERE student_id = %(sid)s
    #     AND submitted_at BETWEEN %(start)s AND %(end)s
    #     ORDER BY submitted_at
    # """
    # return await clickhouse_query(query, {
    #     "sid": student_id, "start": str(start), "end": str(end)
    # })
    return []

async def _fetch_exam_data(
    self, student_id: str, start: date, end: date
) -> List[Dict[str, Any]]:
    """从ClickHouse获取考试成绩数据"""
    return []

async def _fetch_practice_data(
    self, student_id: str, start: date, end: date
) -> List[Dict[str, Any]]:
    """获取练习（字帖/笔顺）数据"""
    return []

async def _fetch_writing_data(
    self, student_id: str, start: date, end: date
) -> List[Dict[str, Any]]:
    """获取书写质量评分数据"""
    return []

async def _fetch_usage_data(
    self, student_id: str, start: date, end: date
) -> List[Dict[str, Any]]:
    """获取应用使用时长数据"""
    return []

def _analyze_subject_abilities(
    self,
    homework_data: List[Dict[str, Any]],
    exam_data: List[Dict[str, Any]],
    practice_data: List[Dict[str, Any]],
) -> List[SubjectAbility]:
    """
    各科目能力多维评估

    评估维度：
    - 作业/考试平均分
    - 知识点覆盖率（已接触/总知识点数）
    - 练习频率（次/周）
    - 进步速率（最近30天vs前30天分数差）
    - 稳定性（分数标准差的倒数归一化）
    """

```

```

subject_map: Dict[str, Dict[str, List[float]]] = {}

# 按科目聚合作业分数
for hw in homework_data:
    subject = hw.get("subject", "unknown")
    subject_map.setdefault(subject, {"scores": [], "dates": []})
    total = hw.get("total_score", 100)
    score = hw.get("score", 0)
    normalized = (score / max(total, 1)) * 100
    subject_map[subject]["scores"].append(normalized)

# 按科目聚合考试分数
for exam in exam_data:
    subject = exam.get("subject", "unknown")
    subject_map.setdefault(subject, {"scores": [], "dates": []})
    total = exam.get("total_score", 100)
    score = exam.get("score", 0)
    normalized = (score / max(total, 1)) * 100
    subject_map[subject]["scores"].append(normalized)

abilities: List[SubjectAbility] = []
for subject, data in subject_map.items():
    scores = data["scores"]
    if not scores:
        continue

    avg_score = sum(scores) / len(scores)

    # 稳定性:  $1 / (1 + \text{std\_dev})$  归一化到0-1
    variance = sum((s - avg_score) ** 2 for s in scores) / max(
        len(scores), 1
    )
    std_dev = math.sqrt(variance)
    stability = 1.0 / (1.0 + std_dev / 10) # 归一化

    # 进步速率: 后半段均分 - 前半段均分
    mid = len(scores) // 2
    if mid > 0:
        first_half_avg = sum(scores[:mid]) / mid
        second_half_avg = sum(scores[mid:]) / max(
            len(scores) - mid, 1
        )
        improvement = second_half_avg - first_half_avg
    else:
        improvement = 0.0

    abilities.append(SubjectAbility(
        subject=subject,
        overall_score=round(avg_score, 1),
        stability=round(stability, 3),
        improvement_rate=round(improvement, 1),
    ))

return abilities

def _analyze_learning_habit(
    self, usage_data: List[Dict[str, Any]]

```

```

) -> LearningHabit:
    """
    学习习惯分析

    分析维度：
    - 日均学习时长
    - 学习高峰时段
    - 周学习模式（周一到周日）
    - 学习规律性评分
    """
    if not usage_data:
        return LearningHabit()

    # 按日期聚合使用时长
    daily_minutes: Dict[str, float] = {}
    hourly_counts: Dict[int, int] = {}
    weekday_minutes: Dict[int, List[float]] = {
        i: [] for i in range(7)
    }

    for record in usage_data:
        date_str = record.get("date", "")
        minutes = record.get("duration_minutes", 0)
        hour = record.get("start_hour", 0)

        daily_minutes[date_str] = (
            daily_minutes.get(date_str, 0) + minutes
        )
        hourly_counts[hour] = hourly_counts.get(hour, 0) + 1

    # 日均时长
    total_days = max(len(daily_minutes), 1)
    avg_daily = sum(daily_minutes.values()) / total_days

    # 学习高峰时段
    peak_hour = max(
        hourly_counts, key=hourly_counts.get, default=0
    )

    # 学习规律性：日均时长的变异系数越小越规律
    if daily_minutes:
        values = list(daily_minutes.values())
        mean_val = sum(values) / len(values)
        variance = sum((v - mean_val) ** 2 for v in values) / len(
            values
        )
        std_val = math.sqrt(variance)
        cv = std_val / max(mean_val, 1)
        consistency = max(0.0, 1.0 - cv) # 变异系数越小规律性越高
    else:
        consistency = 0.0

    return LearningHabit(
        avg_daily_minutes=round(avg_daily, 1),
        peak_study_hour=peak_hour,
        consistency_score=round(consistency, 3),
    )

```

```

def _analyze_writing_ability(
    self, writing_data: List[Dict[str, Any]]
) -> WritingAbility:
    """
    书写能力评估

    基于笔顺准确率、书写规范性评分、书写速度等维度综合评估。
    通过对比最近和较早的数据判断进步趋势。
    """
    if not writing_data:
        return WritingAbility()

    # 计算各维度平均值
    stroke_scores = [
        d.get("stroke_order_score", 0) for d in writing_data
    ]
    quality_scores = [
        d.get("quality_score", 0) for d in writing_data
    ]
    speeds = [d.get("speed", 0) for d in writing_data]
    structure_scores = [
        d.get("structure_score", 0) for d in writing_data
    ]

    avg_stroke = sum(stroke_scores) / max(len(stroke_scores), 1)
    avg_quality = sum(quality_scores) / max(len(quality_scores), 1)
    avg_speed = sum(speeds) / max(len(speeds), 1)
    avg_structure = sum(structure_scores) / max(
        len(structure_scores), 1
    )

    # 判断趋势：后半段 vs 前半段
    mid = len(quality_scores) // 2
    if mid > 0:
        early_avg = sum(quality_scores[:mid]) / mid
        recent_avg = sum(quality_scores[mid:]) / max(
            len(quality_scores) - mid, 1
        )
        if recent_avg - early_avg > 3:
            trend = "improving"
        elif early_avg - recent_avg > 3:
            trend = "declining"
        else:
            trend = "stable"
    else:
        trend = "stable"

    return WritingAbility(
        stroke_order_accuracy=round(avg_stroke, 1),
        writing_quality=round(avg_quality, 1),
        writing_speed=round(avg_speed, 1),
        char_structure_score=round(avg_structure, 1),
        improvement_trend=trend,
    )

def _analyze_score_trends(

```

```

        self,
        homework_data: List[Dict[str, Any]],
        exam_data: List[Dict[str, Any]],
    ) -> List[ScoreTrend]:
        """生成成绩趋势数据"""
        trends: List[ScoreTrend] = []

        for hw in homework_data:
            total = hw.get("total_score", 100)
            score = hw.get("score", 0)
            normalized = (score / max(total, 1)) * 100
            trends.append(ScoreTrend(
                date=hw.get("submitted_at", "")[:10],
                score=round(normalized, 1),
                subject=hw.get("subject", ""),
                exam_type="homework",
            ))

        for exam in exam_data:
            total = exam.get("total_score", 100)
            score = exam.get("score", 0)
            normalized = (score / max(total, 1)) * 100
            trends.append(ScoreTrend(
                date=exam.get("exam_date", "")[:10],
                score=round(normalized, 1),
                subject=exam.get("subject", ""),
                exam_type="exam",
            ))

        # 按日期排序
        trends.sort(key=lambda t: t.date)
        return trends

    def _calculate_overall_score(
        self,
        subject_abilities: List[SubjectAbility],
        learning_habit: LearningHabit,
        writing_ability: WritingAbility,
    ) -> float:
        """
        计算综合评分（百分制）

        加权公式：
        综合分 = 作业成绩×0.30 + 考试成绩×0.35 + 练习×0.15
                + 书写×0.10 + 学习习惯×0.10
        """
        # 作业/考试平均分
        if subject_abilities:
            academic_avg = sum(
                a.overall_score for a in subject_abilities
            ) / len(subject_abilities)
        else:
            academic_avg = 0.0

        # 书写能力评分（归一化到百分制）
        writing_score = writing_ability.writing_quality

```

```

# 学习习惯评分（规律性×100）
habit_score = learning_habit.consistency_score * 100

# 加权综合
overall = (
    academic_avg * (self.WEIGHT_HOMEWORK_SCORE + self.WEIGHT_EXAM_SCORE)
    + academic_avg * self.WEIGHT_PRACTICE
    + writing_score * self.WEIGHT_WRITING
    + habit_score * self.WEIGHT_HABIT
)

return min(100.0, max(0.0, overall))

async def _calculate_rankings(
    self,
    student_id: str,
    class_id: str,
    grade: str,
    score: float,
) -> Tuple[int, int, float]:
    """
    计算班级排名和年级百分位排名

    从ClickHouse查询同班和同年级学生的综合评分，
    计算当前学生的排名位置。
    """
    # 查询同班学生评分
    # class_scores = await query_class_scores(class_id)
    # class_rank = sum(1 for s in class_scores if s > score) + 1

    # 查询同年级学生评分
    # grade_scores = await query_grade_scores(grade)
    # grade_rank = sum(1 for s in grade_scores if s > score) + 1
    # percentile = (1 - grade_rank / max(len(grade_scores), 1)) * 100

    return 0, 0, 0.0

async def _save_profile(self, profile: ComprehensiveProfile) -> None:
    """将画像数据写入ClickHouse画像宽表"""
    # clickhouse_client.execute(
    #     "INSERT INTO student_profile VALUES",
    #     [profile_to_row(profile)],
    # )
    pass

```

analytics/writing_growth.py

```

# 自然写教学数据分析与学情诊断系统软件 V1.0
# analytics/writing_growth.py - 书写能力成长评测引擎

import logging
import math
from typing import Any, Dict, List, Optional, Tuple
from datetime import datetime, date, timedelta

```

```

from dataclasses import dataclass, field

logger = logging.getLogger("writech.analytics.writing_growth")

# =====
# 书写成长数据模型
# =====

@dataclass
class WritingSnapshot:
    """书写能力时间切片"""
    date: str
    stroke_order_accuracy: float = 0.0
    writing_quality: float = 0.0
    writing_speed: float = 0.0
    char_structure: float = 0.0
    practice_count: int = 0
    total_chars: int = 0

@dataclass
class CharacterProgress:
    """单字书写进步记录"""
    character: str
    first_score: float
    latest_score: float
    best_score: float
    practice_count: int
    improvement: float # latest - first
    mastery_level: str # beginner/intermediate/advanced/master

@dataclass
class WritingGrowthReport:
    """书写成长评测报告"""
    student_id: str
    period_start: str
    period_end: str

    # 总体评级
    overall_level: str = "" # 初学/入门/进阶/优秀/精通
    overall_score: float = 0.0
    overall_trend: str = "stable"

    # 各维度评分与趋势
    stroke_order_score: float = 0.0
    stroke_order_trend: str = "stable"
    quality_score: float = 0.0
    quality_trend: str = "stable"
    speed_score: float = 0.0
    speed_trend: str = "stable"
    structure_score: float = 0.0
    structure_trend: str = "stable"

    # 时序数据
    snapshots: List[WritingSnapshot] = field(default_factory=list)

```

```

# 单字进步排行
most_improved_chars: List[CharacterProgress] = field(
    default_factory=list
)
needs_practice_chars: List[CharacterProgress] = field(
    default_factory=list
)

# 练习统计
total_practice_sessions: int = 0
total_characters_written: int = 0
avg_daily_practice_minutes: float = 0.0

# 生成时间
analyzed_at: str = ""

# =====
# 书写成长评测引擎
# =====

class WritingGrowthAnalyzer:
    """
    书写能力成长评测引擎

    功能:
    1. 多维度书写能力评分 (笔顺、规范性、速度、结构)
    2. 成长趋势分析 (移动平均法平滑噪声)
    3. 单字进步追踪
    4. 书写等级评定
    5. 书写问题诊断
    """

    # 书写等级评定标准
    LEVEL_THRESHOLDS = {
        "精通": 95.0,
        "优秀": 85.0,
        "进阶": 70.0,
        "入门": 50.0,
        "初学": 0.0,
    }

    # 各维度权重
    WEIGHTS = {
        "stroke_order": 0.25,
        "quality": 0.35,
        "speed": 0.15,
        "structure": 0.25,
    }

    def __init__(self):
        logger.info("书写成长评测引擎初始化")

    async def analyze_growth(
        self,
        student_id: str,

```

```

        start_date: str,
        end_date: str,
        granularity: str = "weekly",
    ) -> WritingGrowthReport:
        """
        分析学生书写能力成长情况

        Args:
            student_id: 学生ID
            start_date: 分析起始日期
            end_date: 分析结束日期
            granularity: 时间粒度 (daily/weekly/monthly)

        Returns:
            书写成长评测报告
        """
        logger.info(
            "书写成长分析: student=%s, %s~%s, 粒度=%s",
            student_id, start_date, end_date, granularity,
        )

        # 1. 获取原始书写评分数据
        raw_data = await self._fetch_writing_scores(
            student_id, start_date, end_date
        )

        # 2. 按时间粒度聚合
        snapshots = self._aggregate_by_period(raw_data, granularity)

        # 3. 计算各维度评分和趋势
        stroke_score, stroke_trend = self._calc_dimension_trend(
            [s.stroke_order_accuracy for s in snapshots]
        )
        quality_score, quality_trend = self._calc_dimension_trend(
            [s.writing_quality for s in snapshots]
        )
        speed_score, speed_trend = self._calc_dimension_trend(
            [s.writing_speed for s in snapshots]
        )
        structure_score, structure_trend = self._calc_dimension_trend(
            [s.char_structure for s in snapshots]
        )

        # 4. 计算综合评分
        overall_score = self._calc_overall_score(
            stroke_score, quality_score, speed_score, structure_score
        )
        overall_level = self._determine_level(overall_score)
        overall_trend = self._determine_overall_trend(snapshots)

        # 5. 分析单字进步
        char_data = await self._fetch_character_scores(
            student_id, start_date, end_date
        )
        most_improved, needs_practice = self._analyze_char_progress(
            char_data
        )

```

```

# 6. 练习统计
total_sessions = sum(s.practice_count for s in snapshots)
total_chars = sum(s.total_chars for s in snapshots)
days = max(
    (
        datetime.fromisoformat(end_date)
        - datetime.fromisoformat(start_date)
    ).days,
    1,
)
avg_daily = total_chars / days * 0.5 # 估算每日练习分钟

report = WritingGrowthReport(
    student_id=student_id,
    period_start=start_date,
    period_end=end_date,
    overall_level=overall_level,
    overall_score=round(overall_score, 1),
    overall_trend=overall_trend,
    stroke_order_score=round(stroke_score, 1),
    stroke_order_trend=stroke_trend,
    quality_score=round(quality_score, 1),
    quality_trend=quality_trend,
    speed_score=round(speed_score, 1),
    speed_trend=speed_trend,
    structure_score=round(structure_score, 1),
    structure_trend=structure_trend,
    snapshots=snapshots,
    most_improved_chars=most_improved[:10],
    needs_practice_chars=needs_practice[:10],
    total_practice_sessions=total_sessions,
    total_characters_written=total_chars,
    avg_daily_practice_minutes=round(avg_daily, 1),
    analyzed_at=datetime.now().isoformat(),
)

return report

async def _fetch_writing_scores(
    self, student_id: str, start: str, end: str
) -> List[Dict[str, Any]]:
    """从ClickHouse获取书写评分原始数据"""
    # query = """
    #     SELECT date, stroke_order_accuracy, writing_quality,
    #            writing_speed, char_structure, practice_count, total_chars
    #     FROM writing_growth
    #     WHERE student_id = %(sid)s
    #            AND date BETWEEN %(start)s AND %(end)s
    #     ORDER BY date
    # """
    return []

async def _fetch_character_scores(
    self, student_id: str, start: str, end: str
) -> List[Dict[str, Any]]:
    """获取单字练习评分数据"""

```

```

# query = """
#     SELECT character, score, practice_at
#     FROM practice_records
#     WHERE student_id = %(sid)s
#     AND practice_at BETWEEN %(start)s AND %(end)s
#     ORDER BY character, practice_at
# """
return []

def _aggregate_by_period(
    self,
    raw_data: List[Dict[str, Any]],
    granularity: str,
) -> List[WritingSnapshot]:
    """按时间粒度聚合书写评分"""
    if not raw_data:
        return []

    # 按日期分组
    period_map: Dict[str, List[Dict[str, Any]]] = {}
    for record in raw_data:
        date_str = record.get("date", "")
        if granularity == "weekly":
            # 按周分组（取周一日期）
            dt = datetime.fromisoformat(date_str)
            week_start = dt - timedelta(days=dt.weekday())
            period_key = week_start.date().isoformat()
        elif granularity == "monthly":
            period_key = date_str[:7] # YYYY-MM
        else:
            period_key = date_str

        period_map.setdefault(period_key, []).append(record)

    # 聚合每个周期
    snapshots: List[WritingSnapshot] = []
    for period, records in sorted(period_map.items()):
        n = len(records)
        snapshot = WritingSnapshot(
            date=period,
            stroke_order_accuracy=sum(
                r.get("stroke_order_accuracy", 0) for r in records
            ) / n,
            writing_quality=sum(
                r.get("writing_quality", 0) for r in records
            ) / n,
            writing_speed=sum(
                r.get("writing_speed", 0) for r in records
            ) / n,
            char_structure=sum(
                r.get("char_structure", 0) for r in records
            ) / n,
            practice_count=sum(
                r.get("practice_count", 0) for r in records
            ),
            total_chars=sum(
                r.get("total_chars", 0) for r in records
            )
        )
        snapshots.append(snapshot)
    return snapshots

```

```

        ),
    )
    snapshots.append(snapshot)

    return snapshots

def _calc_dimension_trend(
    self, values: List[float]
) -> Tuple[float, str]:
    """
    计算某维度的当前评分和趋势

    使用指数移动平均(EMA)平滑数据噪声,
    对比最近EMA与早期EMA判断趋势。
    """
    if not values:
        return 0.0, "stable"

    # 指数移动平均 (衰减因子0.3)
    alpha = 0.3
    ema_values = [values[0]]
    for i in range(1, len(values)):
        ema = alpha * values[i] + (1 - alpha) * ema_values[-1]
        ema_values.append(ema)

    current_score = ema_values[-1]

    # 趋势判断: 对比前半段和后半段的EMA均值
    if len(ema_values) >= 4:
        mid = len(ema_values) // 2
        early_avg = sum(ema_values[:mid]) / mid
        recent_avg = sum(ema_values[mid:]) / (len(ema_values) - mid)
        diff = recent_avg - early_avg

        if diff > 3:
            trend = "improving"
        elif diff < -3:
            trend = "declining"
        else:
            trend = "stable"
    else:
        trend = "stable"

    return current_score, trend

def _calc_overall_score(
    self,
    stroke: float,
    quality: float,
    speed: float,
    structure: float,
) -> float:
    """加权计算综合书写评分"""
    return (
        stroke * self.WEIGHTS["stroke_order"]
        + quality * self.WEIGHTS["quality"]
        + speed * self.WEIGHTS["speed"]

```

```

        + structure * self.WEIGHTS["structure"]
    )

def _determine_level(self, score: float) -> str:
    """根据综合评分确定书写等级"""
    for level, threshold in self.LEVEL_THRESHOLDS.items():
        if score >= threshold:
            return level
    return "初学"

def _determine_overall_trend(
    self, snapshots: List[WritingSnapshot]
) -> str:
    """判断总体趋势"""
    if len(snapshots) < 2:
        return "stable"

    # 计算每个快照的综合分
    scores = []
    for s in snapshots:
        overall = self._calc_overall_score(
            s.stroke_order_accuracy,
            s.writing_quality,
            s.writing_speed,
            s.char_structure,
        )
        scores.append(overall)

    # 简单线性回归斜率判断趋势
    n = len(scores)
    x_mean = (n - 1) / 2
    y_mean = sum(scores) / n
    numerator = sum(
        (i - x_mean) * (scores[i] - y_mean) for i in range(n)
    )
    denominator = sum((i - x_mean) ** 2 for i in range(n))

    if denominator == 0:
        return "stable"

    slope = numerator / denominator

    if slope > 0.5:
        return "improving"
    elif slope < -0.5:
        return "declining"
    return "stable"

def _analyze_char_progress(
    self, char_data: List[Dict[str, Any]]
) -> Tuple[List[CharacterProgress], List[CharacterProgress]]:
    """
    分析单字进步情况

    对每个练习过的汉字，比较首次评分和最近评分，
    找出进步最大的字和仍需练习的字。
    """

```

```

char_map: Dict[str, List[Tuple[float, str]]] = {}

for record in char_data:
    char = record.get("character", "")
    score = record.get("score", 0.0)
    practice_at = record.get("practice_at", "")
    char_map.setdefault(char, []).append((score, practice_at))

progress_list: List[CharacterProgress] = []

for char, entries in char_map.items():
    # 按时间排序
    entries.sort(key=lambda e: e[1])

    first_score = entries[0][0]
    latest_score = entries[-1][0]
    best_score = max(e[0] for e in entries)
    improvement = latest_score - first_score

    # 掌握等级判定
    if latest_score >= 90:
        level = "master"
    elif latest_score >= 75:
        level = "advanced"
    elif latest_score >= 60:
        level = "intermediate"
    else:
        level = "beginner"

    progress_list.append(CharacterProgress(
        character=char,
        first_score=first_score,
        latest_score=latest_score,
        best_score=best_score,
        practice_count=len(entries),
        improvement=round(improvement, 1),
        mastery_level=level,
    ))

# 按进步幅度降序排列（进步最大的）
most_improved = sorted(
    progress_list, key=lambda p: p.improvement, reverse=True
)

# 仍需练习的（最新分低于70且练习次数>3）
needs_practice = sorted(
    [
        p for p in progress_list
        if p.latest_score < 70 and p.practice_count > 3
    ],
    key=lambda p: p.latest_score,
)

return most_improved, needs_practice

```

api/

api/profile_api.py

```
# 自然写教学数据分析与学情诊断系统软件 V1.0
# api/profile_api.py - 学情画像API接口

import logging
from typing import Optional, List, Dict, Any
from datetime import datetime, date, timedelta
from enum import Enum

from fastapi import APIRouter, Query, Path, Depends, HTTPException
from pydantic import BaseModel, Field

logger = logging.getLogger("writech.analytics.profile")

router = APIRouter(tags=["学情画像"])

# =====
# 数据模型定义
# =====

class SubjectEnum(str, Enum):
    """学科枚举"""
    CHINESE = "chinese"
    MATH = "math"
    ENGLISH = "english"
    PHYSICS = "physics"
    CHEMISTRY = "chemistry"
    BIOLOGY = "biology"

class KnowledgeMastery(BaseModel):
    """知识点掌握度模型"""
    knowledge_id: str = Field(..., description="知识点ID")
    knowledge_name: str = Field(..., description="知识点名称")
    chapter: str = Field("", description="所属章节")
    mastery_level: float = Field(0.0, ge=0.0, le=1.0, description="掌握度(0-1)")
    practice_count: int = Field(0, description="练习次数")
    correct_rate: float = Field(0.0, description="正确率")
    last_practice_at: Optional[str] = Field(None, description="最近练习时间")
    trend: str = Field("stable", description="趋势: improving/declining/stable")

class WeakPoint(BaseModel):
    """薄弱知识点模型"""
    knowledge_id: str
    knowledge_name: str
    mastery_level: float
    error_count: int = Field(0, description="错误次数")
    suggested_exercises: List[str] = Field([], description="推荐练习题ID")
    related_knowledge: List[str] = Field([], description="关联知识点")
```

```

class StudentProfile(BaseModel):
    """学生学情画像完整模型"""
    student_id: str
    student_name: str
    class_id: str
    grade: str
    school_id: str

    # 总体学业水平
    overall_score: float = Field(0.0, description="综合评分(百分制)")
    overall_rank: int = Field(0, description="班级排名")
    overall_trend: str = Field("stable", description="总体趋势")

    # 各科目掌握度
    subject_scores: Dict[str, float] = Field({}, description="各科目评分")

    # 知识点掌握度矩阵
    knowledge_mastery: List[KnowledgeMastery] = Field([])

    # 薄弱环节
    weak_points: List[WeakPoint] = Field([])

    # 书写能力评估
    writing_quality_score: float = Field(0.0, description="书写规范性评分")
    stroke_order_accuracy: float = Field(0.0, description="笔顺正确率")
    writing_speed: float = Field(0.0, description="书写速度(字/分)")

    # 学习习惯统计
    avg_daily_study_minutes: float = Field(0.0, description="日均学习时长(分)")
    homework_completion_rate: float = Field(0.0, description="作业完成率")
    homework_on_time_rate: float = Field(0.0, description="按时提交率")

    # 更新时间
    updated_at: str = Field("", description="画像更新时间")


class ClassProfile(BaseModel):
    """班级学情统计模型"""
    class_id: str
    class_name: str
    grade: str
    student_count: int

    # 班级整体指标
    avg_score: float = Field(0.0, description="班级平均分")
    median_score: float = Field(0.0, description="班级中位分")
    max_score: float = Field(0.0, description="最高分")
    min_score: float = Field(0.0, description="最低分")
    std_deviation: float = Field(0.0, description="标准差")

    # 成绩分布 (分数段人数)
    score_distribution: Dict[str, int] = Field(
        {}, description="分数段分布: {'90-100': 5, '80-89': 10, ...}"
    )

```

```

# 知识点班级掌握度
knowledge_avg_mastery: List[Dict[str, Any]] = Field([])

# 薄弱知识点 (班级维度)
class_weak_points: List[Dict[str, Any]] = Field([])

# 作业统计
homework_avg_completion: float = Field(0.0)
homework_avg_score: float = Field(0.0)

class ProfileCompareResponse(BaseModel):
    """学情对比响应"""
    student_profile: StudentProfile
    class_avg: Dict[str, float]
    grade_avg: Dict[str, float]
    percentile: float = Field(0.0, description="年级百分位排名")

# =====
# API接口实现
# =====

@router.get("/student/{student_id}", response_model=StudentProfile)
async def get_student_profile(
    student_id: str = Path(..., description="学生ID"),
    subject: Optional[SubjectEnum] = Query(None, description="筛选科目"),
):
    """
    获取学生个人学情画像

    返回学生的知识掌握度、薄弱环节、书写能力、学习习惯等全面画像数据。
    教师可查看本班学生，家长可查看自己子女。
    """
    logger.info("查询学生画像: student_id=%s, subject=%s", student_id, subject)

    try:
        # 从ClickHouse查询学生画像宽表数据
        # profile_data = await query_student_profile(student_id)

        # 从Neo4j查询知识点掌握度和薄弱环节
        # mastery = await query_knowledge_mastery(student_id, subject)
        # weak = await query_weak_points(student_id, subject)

        # 组装画像数据
        profile = StudentProfile(
            student_id=student_id,
            student_name="",
            class_id="",
            grade="",
            school_id="",
            updated_at=datetime.now().isoformat(),
        )

        return profile

    except Exception as e:

```

```

        logger.error("查询学生画像失败: %s", str(e))
        raise HTTPException(status_code=500, detail=f"查询学生画像失败: {str(e)}")

@router.get("/class/{class_id}", response_model=ClassProfile)
async def get_class_profile(
    class_id: str = Path(..., description="班级ID"),
    subject: Optional[SubjectEnum] = Query(None, description="筛选科目"),
    start_date: Optional[str] = Query(None, description="起始日期"),
    end_date: Optional[str] = Query(None, description="结束日期"),
):
    """
    获取班级学情统计

    返回班级平均分、分数分布、薄弱知识点等班级维度的统计数据。
    仅班级教师 and 校管理员可查看。
    """
    logger.info("查询班级学情: class_id=%s, subject=%s", class_id, subject)

    try:
        # 从ClickHouse聚合查询班级统计数据
        # class_stats = await aggregate_class_stats(class_id, subject, ...)

        class_profile = ClassProfile(
            class_id=class_id,
            class_name="",
            grade="",
            student_count=0,
        )

        return class_profile

    except Exception as e:
        logger.error("查询班级学情失败: %s", str(e))
        raise HTTPException(status_code=500, detail=f"查询班级学情失败: {str(e)}")

@router.get("/compare/{student_id}", response_model=ProfileCompareResponse)
async def compare_student_with_class(
    student_id: str = Path(..., description="学生ID"),
    subject: Optional[SubjectEnum] = Query(None),
):
    """
    学生与班级/年级对比分析

    将学生各项指标与班级平均和年级平均对比, 计算百分位排名。
    """
    logger.info("学情对比分析: student_id=%s", student_id)

    try:
        # 查询学生个人画像
        # student = await query_student_profile(student_id)

        # 查询班级和年级平均值
        # class_avg = await query_class_avg(student.class_id, subject)
        # grade_avg = await query_grade_avg(student.grade, subject)

```

```

# 计算百分位排名
# percentile = await calc_percentile(student_id, student.grade)

return ProfileCompareResponse(
    student_profile=StudentProfile(
        student_id=student_id,
        student_name="",
        class_id="",
        grade="",
        school_id="",
    ),
    class_avg={},
    grade_avg={},
    percentile=0.0,
)

except Exception as e:
    logger.error("学情对比失败: %s", str(e))
    raise HTTPException(status_code=500, detail=str(e))

@router.get("/knowledge-map/{student_id}")
async def get_knowledge_map(
    student_id: str = Path(..., description="学生ID"),
    subject: SubjectEnum = Query(..., description="科目"),
):
    """
    获取知识图谱掌握度可视化数据

    从Neo4j查询该科目知识图谱结构，叠加学生个人掌握度，
    生成可供前端ECharts渲染的图谱JSON数据。
    """
    logger.info(
        "查询知识图谱: student_id=%s, subject=%s", student_id, subject
    )

    try:
        # 从Neo4j查询知识点节点和边
        # nodes = await neo4j_query_knowledge_nodes(subject)
        # edges = await neo4j_query_knowledge_edges(subject)

        # 查询学生对各知识点的掌握度
        # mastery_map = await query_mastery_map(student_id, subject)

        # 组装ECharts图谱数据格式
        graph_data = {
            "nodes": [], # [{id, name, mastery, category, ...}]
            "edges": [], # [{source, target, relation_type}]
            "categories": [
                {"name": "已掌握"},
                {"name": "部分掌握"},
                {"name": "未掌握"},
                {"name": "未学习"},
            ],
        }

    return {

```

```

        "code": 0,
        "message": "success",
        "data": graph_data,
    }

except Exception as e:
    logger.error("查询知识图谱失败: %s", str(e))
    raise HTTPException(status_code=500, detail=str(e))

@router.get("/weak-analysis/{student_id}")
async def analyze_weak_points(
    student_id: str = Path(..., description="学生ID"),
    subject: Optional[SubjectEnum] = Query(None),
    top_n: int = Query(10, ge=1, le=50, description="返回前N个薄弱点"),
):
    """
    薄弱知识点深度分析

    结合错题归因和知识图谱前驱关系，分析薄弱根因并给出学习建议。
    """
    logger.info(
        "薄弱分析: student_id=%s, subject=%s, top=%d",
        student_id, subject, top_n,
    )

    try:
        # 查询错题记录及关联知识点
        # errors = await query_error_records(student_id, subject)

        # 利用Neo4j知识图谱进行根因分析
        # 如果某知识点正确率低，检查其前驱知识点是否也未掌握
        # root_causes = await trace_knowledge_prerequisites(errors)

        # 生成学习建议
        weak_analysis = {
            "weak_points": [], # 薄弱知识点列表
            "root_causes": [], # 根因知识点
            "suggestions": [], # 学习建议
            "recommended_exercises": [], # 推荐练习
        }

        return {
            "code": 0,
            "message": "success",
            "data": weak_analysis,
        }

    except Exception as e:
        logger.error("薄弱分析失败: %s", str(e))
        raise HTTPException(status_code=500, detail=str(e))

```

api/report_api.py

```

# 自然写教学数据分析与学情诊断系统软件 V1.0
# api/report_api.py - 报告导出与查询API
# api/growth_api.py - 成长轨迹API
# model/data_models.py - 核心数据模型定义

import logging
from typing import Optional, List, Dict, Any
from datetime import datetime, date
from enum import Enum

from fastapi import APIRouter, Query, Path, HTTPException, BackgroundTasks
from pydantic import BaseModel, Field

logger = logging.getLogger("writech.analytics.api")

# =====
# 报告导出API路由
# =====

report_router = APIRouter(tags=["报告导出"])

class ExportRequest(BaseModel):
    """报告导出请求"""
    report_type: str = Field(..., description="报告类型")
    target_id: str = Field(..., description="目标ID(学生/班级)")
    start_date: str = Field(..., description="开始日期")
    end_date: str = Field(..., description="结束日期")
    format: str = Field("pdf", description="输出格式: json/pdf/html")
    include_charts: bool = Field(True, description="是否包含图表")

class ExportResponse(BaseModel):
    """报告导出响应"""
    task_id: str
    status: str
    download_url: Optional[str] = None
    estimated_seconds: int = 0

@report_router.post("/export", response_model=ExportResponse)
async def export_report(
    request: ExportRequest,
    background_tasks: BackgroundTasks,
):
    """
    生成并导出学情报告

    异步生成报告，返回任务ID。
    客户端可通过任务ID轮询状态或等待WebSocket通知。
    """
    logger.info(
        "报告导出请求: type=%s, target=%s, format=%s",
        request.report_type,
        request.target_id,
    )

```

```

        request.format,
    )

    # 生成任务ID
    task_id = f"rpt_{datetime.now().strftime('%Y%m%d%H%M%S')}-{request.target_id[:8]}"

    # 将报告生成任务加入后台队列
    # background_tasks.add_task(
    #     generate_report_task,
    #     task_id=task_id,
    #     config=request,
    # )

    return ExportResponse(
        task_id=task_id,
        status="processing",
        estimated_seconds=30,
    )

@report_router.get("/status/{task_id}")
async def get_export_status(task_id: str = Path(...)):
    """查询报告导出任务状态"""
    # status = await query_task_status(task_id)
    return {
        "task_id": task_id,
        "status": "completed",
        "download_url": None,
    }

@report_router.get("/class/{class_id}")
async def get_class_report(
    class_id: str = Path(..., description="班级ID"),
    subject: Optional[str] = Query(None),
    start_date: Optional[str] = Query(None),
    end_date: Optional[str] = Query(None),
):
    """
    获取班级学情统计报告

    返回班级平均分、分数分布、薄弱知识点等统计数据。
    仅班级教师 and 校管理员有权限查看。
    """
    logger.info("班级报告查询: class=%s, subject=%s", class_id, subject)

    # 权限校验: 教师仅可查看本班数据
    # verify_class_permission(current_user, class_id)

    # 从ClickHouse查询班级统计数据
    # stats = await aggregate_class_report(class_id, subject, ...)

    return {
        "code": 0,
        "message": "success",
        "data": {
            "class_id": class_id,

```

```

        "student_count": 0,
        "avg_score": 0,
        "score_distribution": {},
        "weak_points": [],
        "top_students": [],
    },
}

@report_router.get("/history")
async def list_report_history(
    target_id: str = Query(..., description="目标ID"),
    report_type: Optional[str] = Query(None),
    page: int = Query(1, ge=1),
    page_size: int = Query(20, ge=1, le=100),
):
    """查询历史报告列表"""
    # reports = await query_report_history(target_id, report_type, ...)
    return {
        "code": 0,
        "data": {
            "total": 0,
            "page": page,
            "items": [],
        },
    }
}

# =====
# 成长轨迹API路由
# =====

growth_router = APIRouter(tags=["成长轨迹"])

@growth_router.get("/{student_id}")
async def get_growth_trajectory(
    student_id: str = Path(..., description="学生ID"),
    subject: Optional[str] = Query(None, description="科目"),
    start_date: Optional[str] = Query(None),
    end_date: Optional[str] = Query(None),
    granularity: str = Query("weekly", description="粒度: daily/weekly/monthly"),
):
    """
    获取学生成长轨迹

    返回学生在指定时间范围内的各项指标时序数据,
    包括成绩趋势、书写能力变化、学习习惯变化等。
    家长仅可查看自己子女的数据。
    """
    logger.info(
        "成长轨迹查询: student=%s, subject=%s, granularity=%s",
        student_id, subject, granularity,
    )

    # 权限校验
    # verify_student_access(current_user, student_id)

```

```

# 从ClickHouse查询时序数据
# trend_data = await query_growth_trend(student_id, subject, ...)

return {
    "code": 0,
    "message": "success",
    "data": {
        "student_id": student_id,
        "period": f"{start_date} ~ {end_date}",
        "score_trend": [],      # 成绩趋势
        "writing_trend": [],    # 书写能力趋势
        "habit_trend": [],      # 学习习惯趋势
        "milestones": [],       # 里程碑事件
    },
}

@growth_router.get("/writing/{student_id}")
async def get_writing_growth(
    student_id: str = Path(..., description="学生ID"),
    start_date: str = Query(..., description="开始日期"),
    end_date: str = Query(..., description="结束日期"),
):
    """
    获取书写能力成长报告

    返回笔顺准确率、书写规范性、书写速度等维度的成长趋势。
    """
    logger.info(
        "书写成长查询: student=%s, %s~%s",
        student_id, start_date, end_date,
    )

    # 调用书写成长分析引擎
    # from analytics.writing_growth import WritingGrowthAnalyzer
    # analyzer = WritingGrowthAnalyzer()
    # report = await analyzer.analyze_growth(
    #     student_id, start_date, end_date
    # )

    return {
        "code": 0,
        "message": "success",
        "data": {
            "student_id": student_id,
            "overall_level": "",
            "overall_score": 0,
            "dimensions": {
                "stroke_order": {"score": 0, "trend": "stable"},
                "quality": {"score": 0, "trend": "stable"},
                "speed": {"score": 0, "trend": "stable"},
                "structure": {"score": 0, "trend": "stable"},
            },
            "snapshots": [],
            "most_improved_chars": [],
            "needs_practice_chars": [],
        }
    }

```

```

    },
}

@growth_router.get("/error/analysis/{student_id}")
async def get_error_analysis(
    student_id: str = Path(..., description="学生ID"),
    subject: Optional[str] = Query(None),
    top_n: int = Query(20, ge=1, le=100),
):
    """
    错题归因分析

    返回学生的错题统计、知识点薄弱分析、错因归类。
    结合知识图谱进行根因分析。
    """
    logger.info(
        "错题分析: student=%s, subject=%s", student_id, subject
    )

    return {
        "code": 0,
        "message": "success",
        "data": {
            "student_id": student_id,
            "total_errors": 0,
            "by_subject": {},          # 按科目分组
            "by_knowledge": [],       # 按知识点排序
            "error_types": {},        # 错因分类
            "root_causes": [],        # 根因分析（知识图谱）
            "recommendations": [],    # 学习建议
        },
    }

@growth_router.post("/push/parent")
async def push_to_parent(
    student_id: str = Query(..., description="学生ID"),
    report_type: str = Query("weekly", description="推送报告类型"),
    background_tasks: BackgroundTasks = None,
):
    """
    触发学情报告推送至家长端

    通过WebSocket或APP推送通知家长查看学情报告。
    家长端展示简化版本的学情摘要。
    """
    logger.info("家长推送: student=%s, type=%s", student_id, report_type)

    # 生成家长版报告
    # background_tasks.add_task(
    #     generate_and_push_parent_report,
    #     student_id=student_id,
    #     report_type=report_type,
    # )

    return {

```

```

        "code": 0,
        "message": "推送任务已提交",
        "data": {"student_id": student_id},
    }

# =====
# 核心数据模型定义 (model/data_models.py)
# =====

class GradeLevel(str, Enum):
    """年级枚举"""
    GRADE_1 = "grade_1"
    GRADE_2 = "grade_2"
    GRADE_3 = "grade_3"
    GRADE_4 = "grade_4"
    GRADE_5 = "grade_5"
    GRADE_6 = "grade_6"
    GRADE_7 = "grade_7"
    GRADE_8 = "grade_8"
    GRADE_9 = "grade_9"

class StudentInfo(BaseModel):
    """学生基本信息"""
    student_id: str
    name: str
    class_id: str
    grade: GradeLevel
    school_id: str
    gender: Optional[str] = None
    created_at: Optional[str] = None

class ClassInfo(BaseModel):
    """班级基本信息"""
    class_id: str
    class_name: str
    grade: GradeLevel
    school_id: str
    teacher_id: str
    student_count: int = 0

class SchoolInfo(BaseModel):
    """学校信息"""
    school_id: str
    school_name: str
    region: str
    district: str

class ErrorRecord(BaseModel):
    """错题记录模型 (MySQL) """
    id: Optional[int] = None
    student_id: str
    homework_id: str

```

```
question_id: str
subject: str
knowledge_point: str = ""
error_type: str = ""          # 计算错误/概念混淆/审题不清/粗心
student_answer: str = ""
correct_answer: str = ""
created_at: str = ""
```

```
class ExamAnalysis(BaseModel):
    """考试分析结果模型 (ClickHouse) """
    exam_id: str
    class_id: str
    subject: str
    exam_date: str
    avg_score: float = 0.0
    median_score: float = 0.0
    max_score: float = 0.0
    min_score: float = 0.0
    std_deviation: float = 0.0
    pass_rate: float = 0.0
    excellent_rate: float = 0.0
    score_distribution: Dict[str, int] = {}
    difficulty_coefficient: float = 0.0
    discrimination_index: float = 0.0
```

```
class KafkaEventSchema(BaseModel):
    """Kafka事件消息Schema"""
    event_id: str
    event_type: str
    student_id: str
    class_id: str = ""
    school_id: str = ""
    timestamp: str
    source: str = ""
    payload: Dict[str, Any] = {}
```

```
class Config:
    json_schema_extra = {
        "example": {
            "event_id": "evt_20240101_001",
            "event_type": "grade_result",
            "student_id": "stu_001",
            "class_id": "cls_001",
            "school_id": "sch_001",
            "timestamp": "2024-01-01T10:00:00+08:00",
            "source": "pad",
            "payload": {
                "homework_id": "hw_001",
                "subject": "chinese",
                "score": 85,
                "total_score": 100,
            },
        },
    }
}
```

etl/

etl/flink_processor.py

```
# 自然写教学数据分析与学情诊断系统软件 V1.0
# etl/flink_processor.py - Flink ETL实时数据处理管道

import logging
import json
import hashlib
from typing import Any, Dict, List, Optional, Tuple
from datetime import datetime, timedelta
from dataclasses import dataclass, field, asdict
from enum import Enum

logger = logging.getLogger("writech.analytics.etl")

# =====
# ETL数据模型
# =====

class EventType(str, Enum):
    """数据事件类型"""
    STROKE_RAW = "stroke_raw" # 原始笔迹数据
    GRADE_RESULT = "grade_result" # 批改结果
    HOMEWORK_SUBMIT = "homework_submit" # 作业提交
    OCR_RESULT = "ocr_result" # OCR识别结果
    STROKE_ORDER = "stroke_order" # 笔顺评分结果
    WRITING_QUALITY = "writing_quality" # 书写质量评分
    EXAM_SCORE = "exam_score" # 考试成绩
    LOGIN_EVENT = "login_event" # 登录事件

@dataclass
class RawEvent:
    """原始事件数据"""
    event_id: str
    event_type: EventType
    student_id: str
    class_id: str
    school_id: str
    timestamp: str
    payload: Dict[str, Any]
    source: str = "" # 事件来源 (pad/mobile/pc/board)

@dataclass
class AggregatedMetric:
    """聚合指标数据 (写入ClickHouse) """
    metric_id: str
    student_id: str
    class_id: str
    school_id: str
```

```

subject: str
metric_type: str # 指标类型
metric_value: float
dimension: str = "" # 维度 (如knowledge_id)
period: str = "daily" # 聚合周期
period_start: str = ""
period_end: str = ""
created_at: str = ""

@dataclass
class StudentDailyStats:
    """学生每日统计汇总"""
    student_id: str
    date: str
    subject: str
    # 作业维度
    homework_count: int = 0
    homework_completed: int = 0
    homework_avg_score: float = 0.0
    # 练习维度
    practice_count: int = 0
    practice_total_chars: int = 0
    practice_avg_score: float = 0.0
    # 书写维度
    writing_quality_avg: float = 0.0
    stroke_order_accuracy: float = 0.0
    writing_speed_avg: float = 0.0
    # 错题维度
    error_count: int = 0
    error_knowledge_points: List[str] = field(default_factory=list)
    # 时间维度
    study_duration_minutes: int = 0

# =====
# Flink ETL处理管道
# =====

class FlinkETLProcessor:
    """
    Flink实时ETL处理器

    数据流：
    原始笔迹/批改数据 → Kafka → Flink实时计算 →
    聚合指标写入ClickHouse → 定时生成诊断报告

    处理阶段：
    1. 数据采集 (Kafka Source)
    2. 数据清洗与标准化
    3. 实时聚合计算
    4. 窗口统计
    5. 写入ClickHouse (Sink)
    """

    def __init__(self, config: Dict[str, Any]):
        """初始化ETL处理器"""

```

```

self.kafka_brokers = config.get("kafka_brokers", "localhost:9092")
self.kafka_topics = config.get("kafka_topics", [])
self.clickhouse_config = config.get("clickhouse", {})
self.batch_size = config.get("batch_size", 100)
self.window_size_seconds = config.get("window_size", 60)

# 内存中的聚合缓冲区
self._daily_stats_buffer: Dict[str, StudentDailyStats] = {}
self._metric_buffer: List[AggregatedMetric] = []
self._error_records_buffer: List[Dict[str, Any]] = []

# 数据质量统计
self._processed_count = 0
self._error_count = 0
self._dropped_count = 0

logger.info(
    "FlinkETL初始化: brokers=%s, topics=%s, batch=%d",
    self.kafka_brokers,
    self.kafka_topics,
    self.batch_size,
)

def start_pipeline(self) -> None:
    """启动ETL处理管道"""
    logger.info("启动Flink ETL处理管道...")

    # 配置Flink执行环境
    # env = StreamExecutionEnvironment.get_execution_environment()
    # env.set_parallelism(4)
    # env.enable_checkpointing(60000) # 60秒checkpoint

    # 定义Kafka数据源
    # kafka_source = KafkaSource.builder() \
    #     .set_bootstrap_servers(self.kafka_brokers) \
    #     .set_topics(self.kafka_topics) \
    #     .set_group_id("analytics-etl") \
    #     .set_starting_offsets(KafkaOffsetsInitializer.latest()) \
    #     .set_value_only_deserializer(SimpleStringSchema()) \
    #     .build()

    # 创建数据流
    # stream = env.from_source(kafka_source, ...)

    # 数据处理链
    # stream \
    #     .map(self._parse_event) \
    #     .filter(self._validate_event) \
    #     .key_by(lambda e: e.student_id) \
    #     .window(TumblingEventTimeWindows.of(Time.minutes(1))) \
    #     .process(self._aggregate_window) \
    #     .add_sink(clickhouse_sink)

    # env.execute("WriteAnalyticsETL")

    logger.info("ETL管道已启动")

```

```

def _parse_event(self, raw_json: str) -> Optional[RawEvent]:
    """
    解析原始JSON消息为RawEvent对象

    数据清洗规则：
    - 必须包含event_type, student_id, timestamp字段
    - timestamp格式校验 (ISO 8601)
    - 过滤空payload
    """
    try:
        data = json.loads(raw_json)

        # 字段完整性校验
        required_fields = ["event_type", "student_id", "timestamp"]
        for field_name in required_fields:
            if field_name not in data or not data[field_name]:
                self._dropped_count += 1
                logger.debug("丢弃不完整事件: 缺少%s", field_name)
                return None

        # 事件类型校验
        try:
            event_type = EventType(data["event_type"])
        except ValueError:
            self._dropped_count += 1
            logger.debug("丢弃未知事件类型: %s", data["event_type"])
            return None

        # 时间戳校验
        try:
            datetime.fromisoformat(
                data["timestamp"].replace("Z", "+00:00")
            )
        except (ValueError, AttributeError):
            self._dropped_count += 1
            return None

        # 生成唯一事件ID (去重用)
        event_id = hashlib.md5(
            f"{data['student_id']}_{data['timestamp']}_{raw_json[:50]}"
            .encode()
        ).hexdigest()

        event = RawEvent(
            event_id=event_id,
            event_type=event_type,
            student_id=data["student_id"],
            class_id=data.get("class_id", ""),
            school_id=data.get("school_id", ""),
            timestamp=data["timestamp"],
            payload=data.get("payload", {}),
            source=data.get("source", ""),
        )

        self._processed_count += 1
        return event

```

```

except json.JSONDecodeError as e:
    self._error_count += 1
    logger.warning("JSON解析失败: %s", str(e))
    return None
except Exception as e:
    self._error_count += 1
    logger.error("事件解析异常: %s", str(e))
    return None

def _validate_event(self, event: Optional[RawEvent]) -> bool:
    """事件有效性过滤"""
    if event is None:
        return False

    # 过滤过旧的数据 (超过7天不处理)
    try:
        event_time = datetime.fromisoformat(
            event.timestamp.replace("Z", "+00:00")
        )
        if datetime.now(event_time.tzinfo) - event_time > timedelta(days=7):
            self._dropped_count += 1
            return False
    except Exception:
        return False

    return True

def process_event(self, event: RawEvent) -> None:
    """
    根据事件类型分发处理

    不同事件类型有不同的聚合逻辑：
    - stroke_raw: 累计书写笔迹量
    - grade_result: 更新作业得分统计
    - stroke_order: 更新笔顺准确率
    - writing_quality: 更新书写质量评分
    """
    handler_map = {
        EventType.STROKE_RAW: self._process_stroke,
        EventType.GRADE_RESULT: self._process_grade,
        EventType.HOMEWORK_SUBMIT: self._process_homework,
        EventType.OCR_RESULT: self._process_ocr,
        EventType.STROKE_ORDER: self._process_stroke_order,
        EventType.WRITING_QUALITY: self._process_writing_quality,
        EventType.EXAM_SCORE: self._process_exam_score,
    }

    handler = handler_map.get(event.event_type)
    if handler:
        handler(event)
    else:
        logger.debug("未处理的事件类型: %s", event.event_type)

def _get_daily_stats(
    self, student_id: str, date_str: str, subject: str
) -> StudentDailyStats:
    """获取或创建学生每日统计缓冲"""

```

```

        key = f"{student_id}_{date_str}_{subject}"
        if key not in self._daily_stats_buffer:
            self._daily_stats_buffer[key] = StudentDailyStats(
                student_id=student_id,
                date=date_str,
                subject=subject,
            )
        return self._daily_stats_buffer[key]

def _process_stroke(self, event: RawEvent) -> None:
    """处理原始笔迹数据事件"""
    payload = event.payload
    stroke_count = payload.get("stroke_count", 0)
    page_id = payload.get("page_id", "")

    # 累计笔迹量到每日统计
    date_str = event.timestamp[:10]
    subject = payload.get("subject", "unknown")
    stats = self._get_daily_stats(event.student_id, date_str, subject)
    stats.practice_total_chars += stroke_count

    # 生成笔迹量聚合指标
    metric = AggregatedMetric(
        metric_id=event.event_id,
        student_id=event.student_id,
        class_id=event.class_id,
        school_id=event.school_id,
        subject=subject,
        metric_type="stroke_count",
        metric_value=float(stroke_count),
        dimension=page_id,
        period_start=date_str,
        created_at=event.timestamp,
    )
    self._metric_buffer.append(metric)

def _process_grade(self, event: RawEvent) -> None:
    """处理批改结果事件"""
    payload = event.payload
    score = payload.get("score", 0)
    total_score = payload.get("total_score", 100)
    subject = payload.get("subject", "unknown")
    homework_id = payload.get("homework_id", "")

    date_str = event.timestamp[:10]
    stats = self._get_daily_stats(event.student_id, date_str, subject)
    stats.homework_count += 1
    stats.homework_completed += 1

    # 增量更新平均分
    n = stats.homework_completed
    stats.homework_avg_score = (
        stats.homework_avg_score * (n - 1) + score
    ) / n

    # 处理错题记录
    errors = payload.get("errors", [])

```

```

for error in errors:
    knowledge_point = error.get("knowledge_point", "")
    if knowledge_point:
        stats.error_count += 1
        if knowledge_point not in stats.error_knowledge_points:
            stats.error_knowledge_points.append(knowledge_point)

    # 错题写入MySQL
    self._error_records_buffer.append({
        "student_id": event.student_id,
        "homework_id": homework_id,
        "question_id": error.get("question_id", ""),
        "subject": subject,
        "knowledge_point": knowledge_point,
        "error_type": error.get("error_type", ""),
        "created_at": event.timestamp,
    })

def _process_homework(self, event: RawEvent) -> None:
    """处理作业提交事件"""
    payload = event.payload
    subject = payload.get("subject", "unknown")
    time_cost = payload.get("time_cost_minutes", 0)

    date_str = event.timestamp[:10]
    stats = self._get_daily_stats(event.student_id, date_str, subject)
    stats.study_duration_minutes += time_cost

def _process_ocr(self, event: RawEvent) -> None:
    """处理OCR识别结果事件"""
    payload = event.payload
    confidence = payload.get("confidence", 0.0)
    char_count = payload.get("char_count", 0)

    # OCR识别结果用于辅助计算书写清晰度指标
    metric = AggregatedMetric(
        metric_id=event.event_id,
        student_id=event.student_id,
        class_id=event.class_id,
        school_id=event.school_id,
        subject="chinese",
        metric_type="ocr_confidence",
        metric_value=confidence,
        created_at=event.timestamp,
    )
    self._metric_buffer.append(metric)

def _process_stroke_order(self, event: RawEvent) -> None:
    """处理笔顺评分结果事件"""
    payload = event.payload
    score = payload.get("score", 0.0)
    character = payload.get("character", "")

    date_str = event.timestamp[:10]
    stats = self._get_daily_stats(event.student_id, date_str, "chinese")

    # 增量更新笔顺准确率

```

```

stats.practice_count += 1
n = stats.practice_count
stats.stroke_order_accuracy = (
    stats.stroke_order_accuracy * (n - 1) + score
) / n

def _process_writing_quality(self, event: RawEvent) -> None:
    """处理书写质量评分事件"""
    payload = event.payload
    quality_score = payload.get("quality_score", 0.0)
    speed = payload.get("speed", 0.0)

    date_str = event.timestamp[:10]
    stats = self._get_daily_stats(event.student_id, date_str, "chinese")

    # 更新书写质量指标
    count = max(stats.practice_count, 1)
    stats.writing_quality_avg = (
        stats.writing_quality_avg * (count - 1) + quality_score
    ) / count
    stats.writing_speed_avg = (
        stats.writing_speed_avg * (count - 1) + speed
    ) / count

def _process_exam_score(self, event: RawEvent) -> None:
    """处理考试成绩事件"""
    payload = event.payload
    subject = payload.get("subject", "unknown")
    score = payload.get("score", 0)
    total = payload.get("total_score", 100)

    metric = AggregatedMetric(
        metric_id=event.event_id,
        student_id=event.student_id,
        class_id=event.class_id,
        school_id=event.school_id,
        subject=subject,
        metric_type="exam_score",
        metric_value=float(score),
        dimension=payload.get("exam_id", ""),
        created_at=event.timestamp,
    )
    self._metric_buffer.append(metric)

def flush_to_clickhouse(self) -> int:
    """
    将缓冲区的聚合指标批量写入ClickHouse

    使用ClickHouse的INSERT批量写入提高性能。
    写入后清空缓冲区。
    返回写入的记录数。
    """
    if not self._metric_buffer and not self._daily_stats_buffer:
        return 0

    total_written = 0

```

```

# 写入聚合指标
if self._metric_buffer:
    metrics = [asdict(m) for m in self._metric_buffer]
    # clickhouse_client.execute(
    #     "INSERT INTO analytics_metrics VALUES",
    #     metrics,
    # )
    total_written += len(metrics)
    logger.info("写入%d条聚合指标到ClickHouse", len(metrics))
    self._metric_buffer.clear()

# 写入每日统计
if self._daily_stats_buffer:
    daily_stats = [
        asdict(s) for s in self._daily_stats_buffer.values()
    ]
    # clickhouse_client.execute(
    #     "INSERT INTO student_daily_stats VALUES",
    #     daily_stats,
    # )
    total_written += len(daily_stats)
    logger.info("写入%d条每日统计到ClickHouse", len(daily_stats))
    self._daily_stats_buffer.clear()

# 写入错题记录到MySQL
if self._error_records_buffer:
    # mysql_batch_insert("error_record", self._error_records_buffer)
    total_written += len(self._error_records_buffer)
    logger.info(
        "写入%d条错题记录到MySQL", len(self._error_records_buffer)
    )
    self._error_records_buffer.clear()

return total_written

def get_pipeline_stats(self) -> Dict[str, int]:
    """获取管道处理统计"""
    return {
        "processed": self._processed_count,
        "errors": self._error_count,
        "dropped": self._dropped_count,
        "buffer_metrics": len(self._metric_buffer),
        "buffer_daily": len(self._daily_stats_buffer),
        "buffer_errors": len(self._error_records_buffer),
    }

def stop_pipeline(self) -> None:
    """停止ETL管道，刷新所有缓冲区"""
    logger.info("正在停止ETL管道...")
    self.flush_to_clickhouse()
    logger.info(
        "ETL管道已停止。统计: %s", self.get_pipeline_stats()
    )

```

report/report_generator.py

```
# 自然写教学数据分析与学情诊断系统软件 V1.0
# report/report_generator.py - 学情报告生成引擎

import logging
import json
import hashlib
from typing import Any, Dict, List, Optional
from datetime import datetime, date, timedelta
from dataclasses import dataclass, field
from enum import Enum

logger = logging.getLogger("writech.analytics.report")

# =====
# 报告类型与模型
# =====

class ReportType(str, Enum):
    """报告类型枚举"""
    STUDENT_WEEKLY = "student_weekly" # 学生周报
    STUDENT_MONTHLY = "student_monthly" # 学生月报
    CLASS_WEEKLY = "class_weekly" # 班级周报
    CLASS_MONTHLY = "class_monthly" # 班级月报
    EXAM_ANALYSIS = "exam_analysis" # 考试分析报告
    WRITING_GROWTH = "writing_growth" # 书写成长报告
    PARENT_PUSH = "parent_push" # 家长推送报告

class ReportFormat(str, Enum):
    """报告输出格式"""
    JSON = "json"
    PDF = "pdf"
    HTML = "html"

@dataclass
class ReportSection:
    """报告章节"""
    title: str
    section_type: str # summary/chart/table/text/recommendation
    content: Dict[str, Any] = field(default_factory=dict)
    order: int = 0

@dataclass
class ReportConfig:
    """报告生成配置"""
    report_type: ReportType
    target_id: str # 学生ID或班级ID
    start_date: str
    end_date: str
    output_format: ReportFormat = ReportFormat.JSON
```

```

include_charts: bool = True
include_recommendations: bool = True
language: str = "zh-CN"

@dataclass
class GeneratedReport:
    """生成的报告"""
    report_id: str
    report_type: ReportType
    target_id: str
    title: str
    period: str
    sections: List[ReportSection]
    summary: str = ""
    generated_at: str = ""
    file_path: Optional[str] = None

    def to_json(self) -> Dict[str, Any]:
        """序列化为JSON"""
        return {
            "report_id": self.report_id,
            "report_type": self.report_type.value,
            "target_id": self.target_id,
            "title": self.title,
            "period": self.period,
            "summary": self.summary,
            "sections": [
                {
                    "title": s.title,
                    "type": s.section_type,
                    "content": s.content,
                    "order": s.order,
                }
                for s in self.sections
            ],
            "generated_at": self.generated_at,
            "file_path": self.file_path,
        }

# =====
# 报告生成引擎
# =====

class ReportGenerator:
    """
    学情报告生成引擎

    支持生成：
    1. 学生周报/月报（个人学情概览+各科分析+书写能力+建议）
    2. 班级周报/月报（班级统计+分数分布+薄弱知识点）
    3. 考试分析报告（成绩分析+区分度+难度系数）
    4. 书写成长报告（书写质量趋势+笔顺进步+对比）
    5. 家长推送报告（简化版个人学情+学习建议）

    输出格式：JSON / PDF / HTML

```

```

"""

def __init__(self, output_dir: str, template_dir: str):
    """初始化报告引擎"""
    self.output_dir = output_dir
    self.template_dir = template_dir
    logger.info("报告引擎初始化: output=%s", output_dir)

async def generate_report(
    self, config: ReportConfig
) -> GeneratedReport:
    """
    根据配置生成报告

    流程:
    1. 从ClickHouse/MySQL查询原始数据
    2. 调用对应报告类型的分析逻辑
    3. 组装报告章节
    4. 输出为指定格式
    """
    logger.info(
        "开始生成报告: type=%s, target=%s, period=%s~%s",
        config.report_type.value,
        config.target_id,
        config.start_date,
        config.end_date,
    )

    # 根据报告类型分发
    generator_map = {
        ReportType.STUDENT_WEEKLY: self._gen_student_report,
        ReportType.STUDENT_MONTHLY: self._gen_student_report,
        ReportType.CLASS_WEEKLY: self._gen_class_report,
        ReportType.CLASS_MONTHLY: self._gen_class_report,
        ReportType.EXAM_ANALYSIS: self._gen_exam_report,
        ReportType.WRITING_GROWTH: self._gen_writing_report,
        ReportType.PARENT_PUSH: self._gen_parent_report,
    }

    gen_func = generator_map.get(config.report_type)
    if not gen_func:
        raise ValueError(f"不支持的报告类型: {config.report_type}")

    report = await gen_func(config)

    # 输出为指定格式
    if config.output_format == ReportFormat.PDF:
        await self._export_pdf(report)
    elif config.output_format == ReportFormat.HTML:
        await self._export_html(report)

    logger.info(
        "报告生成完成: id=%s, title=%s",
        report.report_id, report.title,
    )

    return report

```

```

async def _gen_student_report(
    self, config: ReportConfig
) -> GeneratedReport:
    """
    生成学生个人学情报告（周报/月报）

    章节结构：
    1. 总体概览（综合评分、排名、趋势）
    2. 各科目分析（分数、掌握知识点、薄弱点）
    3. 作业完成情况
    4. 书写能力评估
    5. 学习习惯分析
    6. 个性化建议
    """
    report_id = self._gen_report_id(config)
    period_label = f"{config.start_date} ~ {config.end_date}"
    is_weekly = config.report_type == ReportType.STUDENT_WEEKLY

    sections: List[ReportSection] = []

    # 第1节：总体概览
    # overview_data = await self._query_student_overview(
    #     config.target_id, config.start_date, config.end_date
    # )
    sections.append(ReportSection(
        title="总体学情概览",
        section_type="summary",
        content={
            "overall_score": 0,
            "rank_in_class": 0,
            "rank_change": 0, # 与上期对比排名变化
            "trend": "stable",
            "highlight": "", # 亮点描述
        },
        order=1,
    ))

    # 第2节：各科目分析
    sections.append(ReportSection(
        title="各科学情分析",
        section_type="chart",
        content={
            "chart_type": "radar", # 雷达图
            "subjects": [], # [{name, score, class_avg, grade_avg}]
            "detail": [], # 各科详细分析
        },
        order=2,
    ))

    # 第3节：作业完成情况
    sections.append(ReportSection(
        title="作业完成统计",
        section_type="table",
        content={
            "total_homework": 0,
            "completed": 0,

```

```

        "on_time": 0,
        "avg_score": 0,
        "completion_rate": 0,
        "detail_list": [], # 各科作业明细
    },
    order=3,
))

# 第4节：书写能力评估
sections.append(ReportSection(
    title="书写能力评估",
    section_type="chart",
    content={
        "chart_type": "line", # 折线图展示趋势
        "stroke_order_accuracy": 0,
        "writing_quality": 0,
        "writing_speed": 0,
        "trend_data": [], # 时序数据点
        "improvement": "",
    },
    order=4,
))

# 第5节：学习习惯
sections.append(ReportSection(
    title="学习习惯分析",
    section_type="chart",
    content={
        "chart_type": "bar", # 柱状图展示每日时长
        "avg_daily_minutes": 0,
        "peak_hour": 0,
        "weekly_pattern": [], # 周一~日时长
        "consistency": 0,
    },
    order=5,
))

# 第6节：个性化建议
if config.include_recommendations:
    recommendations = self._generate_recommendations(
        student_id=config.target_id,
        sections=sections,
    )
    sections.append(ReportSection(
        title="个性化学习建议",
        section_type="recommendation",
        content={
            "recommendations": recommendations,
        },
        order=6,
    ))

# 生成摘要
summary = self._generate_summary(sections, "student")

return GeneratedReport(
    report_id=report_id,

```

```

        report_type=config.report_type,
        target_id=config.target_id,
        title=f"学生{'周' if is_weekly else '月'}学情报告",
        period=period_label,
        sections=sections,
        summary=summary,
        generated_at=datetime.now().isoformat(),
    )

async def _gen_class_report(
    self, config: ReportConfig
) -> GeneratedReport:
    """
    生成班级学情报告

    章节： 班级概览、成绩分布、薄弱知识点、优秀/进步学生、教学建议
    """
    report_id = self._gen_report_id(config)
    sections: List[ReportSection] = []

    # 班级概览
    sections.append(ReportSection(
        title="班级学情概览",
        section_type="summary",
        content={
            "student_count": 0,
            "avg_score": 0,
            "median_score": 0,
            "pass_rate": 0,
            "excellent_rate": 0,
        },
        order=1,
    ))

    # 成绩分布
    sections.append(ReportSection(
        title="成绩分布分析",
        section_type="chart",
        content={
            "chart_type": "histogram",
            "distribution": {}, # 分数段人数分布
            "comparison": {}, # 与上期对比
        },
        order=2,
    ))

    # 薄弱知识点
    sections.append(ReportSection(
        title="班级薄弱知识点",
        section_type="table",
        content={
            "weak_points": [], # [{知识点, 正确率, 涉及人数}]
        },
        order=3,
    ))

    # 优秀/进步学生

```

```

sections.append(ReportSection(
    title="优秀与进步学生",
    section_type="table",
    content={
        "top_students": [],      # 前10名
        "most_improved": [],     # 进步最大的学生
        "need_attention": [],    # 需关注的学生
    },
    order=4,
))

# 教学建议
sections.append(ReportSection(
    title="教学改进建议",
    section_type="recommendation",
    content={
        "recommendations": [
            "针对薄弱知识点加强集中讲解和专项练习",
            "关注成绩下滑学生，及时进行个别辅导",
            "利用分层作业满足不同水平学生需求",
        ],
    },
    order=5,
))

return GeneratedReport(
    report_id=report_id,
    report_type=config.report_type,
    target_id=config.target_id,
    title="班级学情分析报告",
    period=f"{config.start_date} ~ {config.end_date}",
    sections=sections,
    generated_at=datetime.now().isoformat(),
)

async def _gen_exam_report(
    self, config: ReportConfig
) -> GeneratedReport:
    """生成考试分析报告（成绩分布+题目区分度+难度系数）"""
    report_id = self._gen_report_id(config)

    sections = [
        ReportSection(
            title="考试基本信息",
            section_type="summary",
            content={"exam_name": "", "subject": "", "total_score": 100},
            order=1,
        ),
        ReportSection(
            title="成绩统计",
            section_type="chart",
            content={
                "avg": 0, "median": 0, "max": 0, "min": 0,
                "std_dev": 0, "pass_rate": 0,
                "distribution": {},
            },
            order=2,

```

```

    ),
    ReportSection(
        title="题目分析",
        section_type="table",
        content={
            "questions": [], # 每题的得分率、区分度、难度系数
        },
        order=3,
    ),
]

return GeneratedReport(
    report_id=report_id,
    report_type=config.report_type,
    target_id=config.target_id,
    title="考试分析报告",
    period=config.start_date,
    sections=sections,
    generated_at=datetime.now().isoformat(),
)

async def _gen_writing_report(
    self, config: ReportConfig
) -> GeneratedReport:
    """生成书写成长报告"""
    report_id = self._gen_report_id(config)

    sections = [
        ReportSection(
            title="书写能力总评",
            section_type="summary",
            content={
                "overall_level": "",
                "stroke_accuracy": 0,
                "quality_score": 0,
                "speed": 0,
            },
            order=1,
        ),
        ReportSection(
            title="成长趋势",
            section_type="chart",
            content={
                "chart_type": "line",
                "data_points": [], # 按周/月的评分趋势
            },
            order=2,
        ),
        ReportSection(
            title="常见书写问题",
            section_type="table",
            content={
                "issues": [], # 笔顺错误、结构问题等
            },
            order=3,
        ),
    ]

```

```

return GeneratedReport(
    report_id=report_id,
    report_type=config.report_type,
    target_id=config.target_id,
    title="书写成长报告",
    period=f"{config.start_date} ~ {config.end_date}",
    sections=sections,
    generated_at=datetime.now().isoformat(),
)

async def _gen_parent_report(
    self, config: ReportConfig
) -> GeneratedReport:
    """
    生成家长推送报告（简化版）

    家长端报告简洁明了：
    - 本周学习概况（评分、排名变化）
    - 学习时长统计
    - 需要关注的科目
    - 家长配合建议
    """
    report_id = self._gen_report_id(config)

    sections = [
        ReportSection(
            title="本周学习概况",
            section_type="summary",
            content={
                "overall_score": 0,
                "rank_change": 0,
                "homework_completed": 0,
                "total_homework": 0,
                "study_minutes": 0,
            },
            order=1,
        ),
        ReportSection(
            title="需要关注",
            section_type="text",
            content={
                "attention_subjects": [],
                "weak_points": [],
            },
            order=2,
        ),
        ReportSection(
            title="家长建议",
            section_type="recommendation",
            content={
                "recommendations": [
                    "建议督促孩子按时完成作业",
                    "建议每天安排15-20分钟练字时间",
                    "多鼓励孩子在薄弱科目上的进步",
                ],
            },
        ),
    ]

```

```

        order=3,
    ),
]

return GeneratedReport(
    report_id=report_id,
    report_type=config.report_type,
    target_id=config.target_id,
    title="孩子本周学情报告",
    period=f"{config.start_date} ~ {config.end_date}",
    sections=sections,
    generated_at=datetime.now().isoformat(),
)

def _generate_recommendations(
    self,
    student_id: str,
    sections: List[ReportSection],
) -> List[str]:
    """基于各章节数据生成个性化学习建议"""
    recommendations: List[str] = []

    # 根据作业完成情况生成建议
    for section in sections:
        if section.title == "作业完成统计":
            rate = section.content.get("completion_rate", 0)
            if rate < 80:
                recommendations.append(
                    "作业完成率偏低，建议养成当天作业当天完成的习惯"
                )

        if section.title == "书写能力评估":
            quality = section.content.get("writing_quality", 0)
            if quality < 60:
                recommendations.append(
                    "书写规范性有待提高，建议每天坚持15分钟字帖练习"
                )

        if section.title == "学习习惯分析":
            consistency = section.content.get("consistency", 0)
            if consistency < 0.5:
                recommendations.append(
                    "学习时间不够规律，建议制定固定的学习作息计划"
                )

    if not recommendations:
        recommendations.append("继续保持良好的学习习惯，争取更大进步！")

    return recommendations

def _generate_summary(
    self,
    sections: List[ReportSection],
    report_target: str,
) -> str:
    """根据报告章节自动生成文字摘要"""
    if report_target == "student":

```

```

        return "本报告汇总了该学生在报告周期内的学业表现、书写能力和学习习惯分析。"
    elif report_target == "class":
        return "本报告汇总了班级在报告周期内的整体学情、成绩分布和教学建议。"
    return ""

def _gen_report_id(self, config: ReportConfig) -> str:
    """生成唯一报告ID"""
    raw = (
        f"{config.report_type.value}_{config.target_id}_"
        f"{config.start_date}_{config.end_date}"
    )
    return hashlib.md5(raw.encode()).hexdigest()[:16]

async def _export_pdf(self, report: GeneratedReport) -> None:
    """
    将报告导出为PDF文件

    使用ReportLab/WeasyPrint渲染PDF:
    - 页眉: 自然写logo + 报告标题
    - 正文: 各章节内容 (图表使用ECharts渲染为图片)
    - 页脚: 页码 + 生成时间
    """
    # from weasyprint import HTML
    # html_content = self._render_html_template(report)
    # pdf_path = f"{self.output_dir}/{report.report_id}.pdf"
    # HTML(string=html_content).write_pdf(pdf_path)
    # report.file_path = pdf_path
    logger.info("PDF导出: %s", report.report_id)

async def _export_html(self, report: GeneratedReport) -> None:
    """将报告导出为HTML文件"""
    # html_path = f"{self.output_dir}/{report.report_id}.html"
    # with open(html_path, "w", encoding="utf-8") as f:
    #     f.write(self._render_html_template(report))
    # report.file_path = html_path
    logger.info("HTML导出: %s", report.report_id)

# =====
# 定时报告生成调度
# =====

class ReportScheduler:
    """
    报告定时生成调度器

    支持:
    - 每日凌晨生成前一天的学生日报
    - 每周一生成上周的学生周报和班级周报
    - 每月1日生成上月的月报
    """

    def __init__(self, generator: ReportGenerator):
        self.generator = generator
        logger.info("报告调度器初始化")

    async def run_daily_reports(self) -> int:

```

```

        """执行每日报告生成任务"""
        yesterday = (date.today() - timedelta(days=1)).isoformat()
        logger.info("执行每日报告生成: date=%s", yesterday)

        generated_count = 0
        # 查询所有活跃学生ID
        # student_ids = await get_active_student_ids()
        # for sid in student_ids:
        #     config = ReportConfig(
        #         report_type=ReportType.PARENT_PUSH,
        #         target_id=sid,
        #         start_date=yesterday,
        #         end_date=yesterday,
        #     )
        #     await self.generator.generate_report(config)
        #     generated_count += 1

        logger.info("每日报告生成完成: 共%d份", generated_count)
        return generated_count

    async def run_weekly_reports(self) -> int:
        """执行每周报告生成任务"""
        end_date = date.today() - timedelta(days=1)
        start_date = end_date - timedelta(days=6)
        logger.info(
            "执行每周报告: %s ~ %s",
            start_date.isoformat(),
            end_date.isoformat(),
        )

        generated_count = 0
        # 生成学生周报和班级周报
        # ...

        logger.info("每周报告生成完成: 共%d份", generated_count)
        return generated_count

    async def run_monthly_reports(self) -> int:
        """执行月度报告生成任务"""
        today = date.today()
        end_date = today.replace(day=1) - timedelta(days=1)
        start_date = end_date.replace(day=1)
        logger.info(
            "执行月度报告: %s ~ %s",
            start_date.isoformat(),
            end_date.isoformat(),
        )

        generated_count = 0
        # 生成学生月报、班级月报、书写成长报告
        # ...

        logger.info("月度报告生成完成: 共%d份", generated_count)
        return generated_count

```