!/usr/bin/env python3

-- coding: utf-8 --

"""
Railway适用的保险方案处理流水线
====================================

基于correct_complete_pipeline.py修改，适配云环境：
1. PDF从Supabase URL下载而不是本地文件系统
2. 结果上传到Supabase而不是保存到本地文件系统
3. 异步处理支持

作者: MiniMax Agent
日期: 2025-08-07
"""

import pdfplumber
import pandas as pd
import numpy as np
import json
import re
from typing import Dict, List, Tuple, Any, Optional, Union
import os
from datetime import datetime
import logging
import numpy_financial as npf
import subprocess
import sys
import time
import tempfile
import shutil
import asyncio
import httpx

```
import mimetypes
import uuid
```

Playwright用于截图

```
try:
from playwright.sync_api import sync_playwright
from playwright.async_api import async_playwright
PLAYWRIGHT_AVAILABLE = True
except ImportError:
PLAYWRIGHT_AVAILABLE = False
print("警告: Playwright未安装，截图功能将不可用")
```

Supabase客户端

```
try:
from supabase import create_client, Client
SUPABASE_AVAILABLE = True
except ImportError:
SUPABASE_AVAILABLE = False
print("警告: Supabase客户端未安装，云存储功能将不可用")
```

设置日志

```
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger(name)
```

添加正确的IRR计算器类

class CorrectedIRRCalculator:
"""修正的IRR计算器（完全按照原脚本）"""

```
@staticmethod
def calculate_irr(cash_flows: List[float], max_iterations: int =
1000, precision: float = 1e-6) -> float:
    """计算内部收益率IRR（完全按照原脚本）"""
    try:
        # 使用numpy_financial计算IRR
        irr = npf.irr(cash_flows)

        if np.isnan(irr) or np.isinf(irr):
            # 如果numpy_financial失败，使用自定义算法
            return CorrectedIRRCalculator._custom_irr(cash_flows,
max_iterations, precision)

        return irr

    except Exception as e:
        logger.warning(f"IRR计算失败: {e}")
        return CorrectedIRRCalculator._custom_irr(cash_flows,
max_iterations, precision)

@staticmethod
def _custom_irr(cash_flows: List[float], max_iterations: int =
1000, precision: float = 1e-6) -> float:
    """自定义IRR计算算法（牛顿法）（完全按照原脚本）"""
    try:
        # 检查现金流是否有效
        if len(cash_flows) < 2:
            return 0.0

        # 检查是否有正负现金流
        has_positive = any(cf > 0 for cf in cash_flows)
        has_negative = any(cf < 0 for cf in cash_flows)

        if not (has_positive and has_negative):
            return 0.0
        # 初始猜测
        rate = 0.1

        for iteration in range(max_iterations):
            # 计算NPV和NPV导数
            npv = sum(cf / (1 + rate) ** i for i, cf in
enumerate(cash_flows))
            npv_derivative = sum(-i * cf / (1 + rate) ** (i + 1)
for i, cf in enumerate(cash_flows))

            if abs(npv) < precision:
                return rate

            if abs(npv_derivative) < precision:
                break

            # 牛顿法更新
            new_rate = rate - npv / npv_derivative

            # 防止rate变为负数或过大
            if new_rate < -0.99:
                new_rate = -0.99
            elif new_rate > 10:
                new_rate = 10

            # 检查收敛
            if abs(new_rate - rate) < precision:
                return new_rate

            rate = new_rate

        return rate

    except Exception as e:
        logger.warning(f"自定义IRR计算失败: {e}")
        return 0.0

@staticmethod
def build_cash_flows_for_surrender(annual_premium: float,
payment_period: int, policy_year: int, surrender_value: float) ->
List[float]:
    """构建一次性退保的现金流（完全按照原脚本）"""
    cash_flows = []

    # 第0年到第(payment_period-1)年：负现金流（保费）
    for year in range(payment_period):
        cash_flows.append(-annual_premium)

    # 第payment_period年到第(policy_year-1)年：0现金流
    for year in range(payment_period, policy_year):
        cash_flows.append(0.0)

    # 第policy_year年：正现金流（退保价值）
    cash_flows.append(surrender_value)

    return cash_flows

@staticmethod
def build_cash_flows_for_withdrawal(annual_premium: float,
payment_period: int, policy_year: int,
                                   withdrawal_start_year: int,
annual_withdrawal: float,
                                   final_surrender_value: float) -
> List[float]:
    """构建现金提取方案的现金流（完全按照原脚本）"""
    cash_flows = []

    # 第0年到第(payment_period-1)年：负现金流（保费）
    for year in range(payment_period):
        cash_flows.append(-annual_premium)

    # 第payment_period年到第(withdrawal_start_year-1)年：0现金流
    for year in range(payment_period, withdrawal_start_year):
        cash_flows.append(0.0)

    # 第withdrawal_start_year年到第(policy_year-1)年：正现金流（年提取金
额）
    for year in range(withdrawal_start_year, policy_year):
        cash_flows.append(annual_withdrawal)

    # 第policy_year年：正现金流（年提取金额 + 剩余退保价值）
    cash_flows.append(annual_withdrawal + final_surrender_value)

    return cash_flows
```

class FinalFilteredExtractor:
"""最终过滤版数据提取器 - 适配Railway云环境"""

```
def __init__(self, pdf_file_paths=None):
    # Railway版本：使用传入的文件路径，不进行自动检测
    if pdf_file_paths and len(pdf_file_paths) >= 2:
        self.scheme1_path = pdf_file_paths[0]
        self.scheme2_path = pdf_file_paths[1]
    else:
        raise ValueError("Railway版本需要明确指定两个PDF文件路径")

    # 页面排除规则（完全按照原脚本）
    self.exclude_rules = [
        "说明摘要",       # 第2页排除
        "最高保单贷款",  # 第15页排除
        "悲观情景"       # 第26页排除
    ]

def find_customer_info_page(self, pdf_path: str) -> int:
    """查找包含'保障摘要'的客户信息页面"""
    try:
        with pdfplumber.open(pdf_path) as pdf:
            for page_num, page in enumerate(pdf.pages, 1):
                text = page.extract_text()
                if text and any(keyword in text for keyword in ['保
障摘要', '建议书摘要', '保障项目']):
                    return page_num - 1  # 返回页面索引
    except Exception as e:
        logger.error(f"查找客户信息页面时出错: {e}")
    return 0  # 默认返回第一页

def extract_customer_info(self, pdf_path: str) -> Dict[str, Any]:
    """使用改进的通用定位规则精确提取客户基本信息"""
    customer_info = {
        'name': 'VIP 先生',
        'age': 0,
        'annual_premium': 50000,  # 默认基础保费（整数）
        'payment_period': 5,
        'total_premium': 250000,  # 默认总保费（整数）
        'currency': '美元',
        'coverage_period': '终身'
    }

    try:
        with pdfplumber.open(pdf_path) as pdf:
            # 找到包含客户信息的页面
            page_index = self.find_customer_info_page(pdf_path)
            page = pdf.pages[page_index]
            text = page.extract_text()
            tables = page.extract_tables()

            if not text:
                return customer_info

            # 方法1: 从表格中提取客户姓名和年龄
            for table in tables:
                if not table:
                    continue

                for row in table:
                    if not row:
                        continue

                    row_text = " ".join([cell.strip() if cell else
"" for cell in row])

                    # 客户姓名和年龄（通常在同一行）
                    if "受保人姓名" in row_text and "年龄" in
row_text:
                        # 提取姓名
                        import re
                        name_match = re.search(r'受保人姓名[：:]
\s*([^年]+)', row_text)
```

```
                        if name_match:
                            customer_info['name'] =
name_match.group(1).strip()

                        # 提取年龄
                        age_match = re.search(r'年龄[：:]\s*(\d+)',
row_text)
                        if age_match:
                            customer_info['age'] =
int(age_match.group(1))

            # 方法2: 从文本行中提取保单货币
            lines = text.split('\n')
            for line in lines:
                line = line.strip()

                # 查找币种信息
                if "保单货币：美元" in line:
                    customer_info['currency'] = "美元"
                elif "保单货币：港币" in line:
                    customer_info['currency'] = "港币"
                elif "保单货币：人民币" in line:
                    customer_info['currency'] = "人民币"
                elif "美元" in line and "保单货币" in line:
                    customer_info['currency'] = "美元"

            # 方法3: 从表格中精确提取基础年缴保费（不含征费）
            for i, table in enumerate(tables):
                if not table or len(table) < 2:
                    continue

                # 查找表头包含"年缴保费"的表格
                if len(table[0]) >= 4 and any("年缴保费" in
str(cell) for cell in table[0]):
                    logger.info(f"找到保费表格 {i+1}")
                    # 在数据行中查找保费信息
                    for row_idx, row in enumerate(table[1:], 1):
                        if len(row) >= 4 and row[3]:  # 第4列是年缴保
费列
                            premium_cell = str(row[3]).strip()
                            logger.info(f"  第{row_idx}行年缴保费列内
容: '{premium_cell}'")

                            # 通用保费提取逻辑 - 支持任何金额格式
                            import re
                            # 查找形如 "XX,XXX.XX" 的金额格式
                            premium_match = re.search(r'(\d{1,3}
(?:,\d{3})*(?:\.\d{2})?)', premium_cell)
                            if premium_match:
                                amount_str =
premium_match.group().replace(',', '')
                                try:
                                    extracted_value =
float(amount_str)

customer_info['annual_premium'] = round(extracted_value)  # 四舍五入
到个位数
                                    logger.info(f"从表格通用提取基础年
保费: {customer_info['annual_premium']} (原值: {amount_str})")
                                    break
                                except:
                                    pass

            # 方法4: 从表格中提取缴费期和保障期
            for table in tables:
                if not table:
                    continue

                for row in table:
                    if not row:
                        continue

                    # 查找包含"年"的列，可能是缴费期
                    for cell in row:
                        if cell and isinstance(cell, str):
                            if "5" in cell and "终身" in "
".join(row):
                                customer_info['payment_period'] = 5
                            if "终身" in cell:
                                customer_info['coverage_period'] =
"终身"

            # 设置默认值
            if customer_info['payment_period'] is None:
                customer_info['payment_period'] = 5     # 默认5年
            if customer_info['coverage_period'] is None:
                customer_info['coverage_period'] = "终身"  # 默认终身

            # 计算总保费（基础保费 × 缴费年数）
            if customer_info['annual_premium'] and
customer_info['payment_period']:
                total_premium = customer_info['annual_premium'] *
customer_info['payment_period']
                customer_info['total_premium'] =
round(total_premium)  # 四舍五入到整数
                logger.info(f"计算总保费:
{customer_info['annual_premium']} ×
{customer_info['payment_period']} =
{customer_info['total_premium']}")

            print(f"改进版提取客户信息: 姓名={customer_info['name']},
年龄={customer_info['age']}, 年缴保费=$
{customer_info['annual_premium']:,}")
            logger.info(f"客户信息: {customer_info}")
```

```
    except Exception as e:
        logger.error(f"提取客户信息时出错: {e}")

    return customer_info

def should_exclude_page(self, text: str) -> bool:
    """检查页面是否应该被排除（完全按照原脚本）"""
    for exclude_keyword in self.exclude_rules:
        if exclude_keyword in text:
            return True
    return False

def find_table_pages(self, pdf_path: str) -> Dict[str, List[int]]:
    """查找包含特定表格的页面，并应用排除规则（完全按照原脚本）"""
    table_pages = {
        'surrender_value': [],  # 退保发还金额
        'withdrawal_surrender_value': [],  # 现金提取后之退保发还金额
    }

    try:
        with pdfplumber.open(pdf_path) as pdf:
            for page_num, page in enumerate(pdf.pages, 1):
                text = page.extract_text()
                if not text:
                    continue

                # 首先检查是否应该排除此页面
                if self.should_exclude_page(text):
                    logger.info(f"第{page_num}页包含排除关键字，跳过")
                    continue

                # 然后检查是否包含目标表格
                if "退保发还金额" in text and "现金提取后之退保发还金额"
not in text:
                    table_pages['surrender_value'].append(page_num)
                    logger.info(f"第{page_num}页包含'退保发还金额'表
格")
                elif "现金提取后之退保发还金额" in text:

table_pages['withdrawal_surrender_value'].append(page_num)
                    logger.info(f"第{page_num}页包含'现金提取后之退保发
还金额'表格")

    except Exception as e:
        logger.error(f"查找表格页面时出错: {e}")

    return table_pages

def parse_newline_separated_data(self, cell_content: str) ->
List[str]:
    """解析用换行符分隔的5年数据（完全按照原脚本）"""
    if not cell_content or pd.isna(cell_content):
        return []

    parts = str(cell_content).split('\n')
    cleaned_parts = []
    for part in parts:
        part = part.strip()
        if part:
            part = part.replace(',', '')
            cleaned_parts.append(part)

    return cleaned_parts[:5]

def find_column_indices(self, table: List[List], table_type: str) -
> Dict[str, int]:
    """精确查找列索引（完全按照原脚本）"""
    column_indices = {}

    if not table or len(table) < 2:
        return column_indices

    # 显示表格结构用于调试
    logger.info(f"表格结构分析 ({table_type}):")
    for row_idx, row in enumerate(table[:4]):
        logger.info(f"  第{row_idx + 1}行: {row}")

    # 查找年龄列（第1列，固定）
    column_indices['age'] = 0

    # 查找保单年度终结列（在第1行中查找）
    if len(table) > 0:
        header_row1 = table[0]
        for col_idx, cell in enumerate(header_row1):
            if cell and '保单' in str(cell) and '年度' in str(cell):
                column_indices['policy_year'] = col_idx
                logger.info(f"找到保单年度终结列: 第{col_idx + 1}列")
                break

        # 如果没找到，使用默认位置（通常是第2列）
        if 'policy_year' not in column_indices:
            column_indices['policy_year'] = 1
            logger.info(f"使用默认保单年度终结列: 第2列")

    # 查找总额列（在第2行中查找）
    if len(table) > 1:
        header_row2 = table[1]
        for col_idx, cell in enumerate(header_row2):
            if cell and '总额' in str(cell):
                column_indices['total_amount'] = col_idx
                logger.info(f"找到总额列: 第{col_idx + 1}列")
                break

    # 查找现金提取金额列（在第1行中查找）
    if len(table) > 0:
        header_row1 = table[0]
        for col_idx, cell in enumerate(header_row1):
            if cell and '现金提取' in str(cell) and '金额' in
str(cell):
                column_indices['withdrawal_amount'] = col_idx
                logger.info(f"找到现金提取金额列: 第{col_idx + 1}列")
                break

    logger.info(f"最终列索引: {column_indices}")
    return column_indices

def extract_table_data_filtered(self, pdf_path: str, page_numbers:
List[int], table_type: str) -> List[Dict]:
    """从指定页面提取过滤后的表格数据（完全按照原脚本）"""
    all_data = []

    try:
        with pdfplumber.open(pdf_path) as pdf:
            for page_num in page_numbers:
                if page_num <= len(pdf.pages):
                    page = pdf.pages[page_num - 1]
                    tables = page.extract_tables()

                    logger.info(f"第{page_num}页包含 {len(tables)}
个表格")

                    for table_idx, table in enumerate(tables):
                        if not table or len(table) < 3:
                            continue

                        # 查找列索引
                        column_indices =
self.find_column_indices(table, table_type)

                        if not column_indices:
                            logger.warning(f"第{page_num}页表格
{table_idx + 1}无法找到有效列索引")
                            continue

                        # 从第3行开始提取数据（跳过表头）
                        for row_idx in range(2, len(table)):
                            row = table[row_idx]

                            if len(row) <=
max(column_indices.values()):
                                continue

                            # 提取各列数据
                            age_cell = row[column_indices['age']]
if 'age' in column_indices else None
                            policy_year_cell =
row[column_indices['policy_year']] if 'policy_year' in
column_indices else None
                            total_amount_cell =
row[column_indices['total_amount']] if 'total_amount' in
column_indices else None
                            withdrawal_cell =
row[column_indices['withdrawal_amount']] if 'withdrawal_amount' in
column_indices else None

                            # 解析各列的5年数据
                            age_values =
self.parse_newline_separated_data(age_cell)
                            policy_year_values =
self.parse_newline_separated_data(policy_year_cell)
                            total_amount_values =
self.parse_newline_separated_data(total_amount_cell)
                            withdrawal_values =
self.parse_newline_separated_data(withdrawal_cell) if
withdrawal_cell else []
```

```
                            logger.info(f"第{row_idx + 1}行数据:")
                            logger.info(f"  年龄: {age_values}")
                            logger.info(f"  保单年度:
{policy_year_values}")
                            logger.info(f"  总额:
{total_amount_values}")
                            logger.info(f"  现金提取:
{withdrawal_values}")

                            # 验证数据一致性
                            if not age_values or not
policy_year_values or not total_amount_values:
                                logger.warning(f"第{row_idx + 1}行数
据不完整，跳过")
                                continue

                            # 确保数据长度一致（都应该是5个）
                            min_length = min(len(age_values),
len(policy_year_values), len(total_amount_values))
                            min_length = min(min_length, 5)

                            # 生成数据记录
                            for year_offset in range(min_length):
                                try:
                                    # 年龄
                                    current_age = 0
                                    if
age_values[year_offset].isdigit():
                                        current_age =
int(age_values[year_offset])

                                    # 保单年度
                                    current_policy_year = 0
                                    if
```

```
policy_year_values[year_offset].isdigit():
                                        current_policy_year =
int(policy_year_values[year_offset])

                                    # 总额
                                    total_amount = 0
                                    if
total_amount_values[year_offset].replace('.', '').isdigit():
                                        total_amount =
float(total_amount_values[year_offset])

                                    # 现金提取金额
                                    withdrawal_amount = 0
                                    if year_offset <
len(withdrawal_values) and
withdrawal_values[year_offset].replace('.', '').isdigit():
                                        withdrawal_amount =
float(withdrawal_values[year_offset])

                                    # 验证数据合理性
                                    if current_policy_year > 100:
# 保单年度不应该超过100

logger.warning(f"异常保单年度 {current_policy_year}, 跳过此记录")
                                        continue

                                    record = {
                                        'page': page_num,
                                        'table_type': table_type,
                                        'age': current_age,
                                        'policy_year':
current_policy_year,
                                        'surrender_value':
total_amount,
                                        'withdrawal_amount':
```

```
withdrawal_amount
                                    }

                                    all_data.append(record)
                                    logger.info(f"  添加记录: 年龄
{current_age}, 第{current_policy_year}年, 退保<span class="math-
inline" style="display: inline;"><math xmlns="http://www.w3.org/
1998/Math/MathML" display="inline"><mrow><mrow><mi>t</mi><mi>o</
mi><mi>t</mi><mi>a</mi><msub><mi>l</mi><mi>a</mi></msub><mi>m</
mi><mi>o</mi><mi>u</mi><mi>n</mi><mi>t</mi><mi>:</
mi><mo>&#x0002C;</mo><mi>.0</mi><mi>f</mi></mrow><mo>&#x0002C;</
mo><mi>提</mi><mi>取</mi></mrow></math></span>{withdrawal_amount:,.
0f}")

                                except Exception as e:
                                    logger.warning(f"处理第
{year_offset + 1}年数据时出错: {e}")

    except Exception as e:
        logger.error(f"提取表格数据时出错: {e}")

    return all_data
```

class RailwayInsurancePipeline:
"""Railway适用的保险方案处理流水线"""

```
def __init__(self, pdf_urls=None, supabase_url=None,
supabase_key=None, task_id=None):
    # 初始化 Supabase 客户端
    self.supabase_url = supabase_url
    self.supabase_key = supabase_key

    if SUPABASE_AVAILABLE and supabase_url and supabase_key:
        self.supabase = create_client(supabase_url, supabase_key)
    else:
        self.supabase = None
        logger.warning("Supabase客户端未初始化，部分功能可能不可用")

    # 任务ID用于标识当前处理任务
    self.task_id = task_id or str(uuid.uuid4())
    print(f"任务ID: {self.task_id}")

    # 创建临时目录存储处理过程中的文件
    self.temp_dir =
tempfile.mkdtemp(prefix=f"railway_task_{self.task_id}_")
    print(f"临时目录: {self.temp_dir}")

    # PDF文件URL
    self.pdf_urls = pdf_urls or []
    if len(self.pdf_urls) < 2:
        raise ValueError("需要至少两个PDF文件URL")

    # 设置临时文件路径
    self.scheme1_pdf_path = os.path.join(self.temp_dir,
f"scheme1_{os.path.basename(self.pdf_urls[0])}")
    self.scheme2_pdf_path = os.path.join(self.temp_dir,
f"scheme2_{os.path.basename(self.pdf_urls[1])}")

    # 设置输出文件路径
    self.extracted_data_file = os.path.join(self.temp_dir, "计划书数
据提取结果.xlsx")
```

```
    self.irr_results_file = os.path.join(self.temp_dir, "计划书数据提
取结果_含IRR计算.xlsx")
    self.final_html_file = os.path.join(self.temp_dir,
"report.html")
    self.final_screenshot_file = os.path.join(self.temp_dir,
"screenshot.png")

    # HTML模板路径（先尝试从当前目录加载，如果不存在，使用内嵌模板）
    self.html_template_path =
os.path.join(os.path.dirname(os.path.abspath(__file__)), "两套方案对
比HTML模板_含占位符_修正7.html")

def _cleanup(self):
    """清理临时文件和目录"""
    try:
        shutil.rmtree(self.temp_dir)
        print(f"临时目录已清理: {self.temp_dir}")
    except Exception as e:
        logger.error(f"清理临时目录时出错: {e}")

async def _download_pdf(self, url: str, local_path: str) -> bool:
    """从URL下载PDF文件到本地路径"""
    try:
        # 处理不同类型的URL：Supabase URL或普通HTTP URL
        if self.supabase and url.startswith(self.supabase_url):
            # 如果是Supabase URL，从存储中获取
            # 解析存储路径
            parsed_url = url.replace(f"{self.supabase_url}/storage/
v1/object/", "")
            parts = parsed_url.split("/")
            if len(parts) < 2:
                raise ValueError(f"无效的Supabase存储URL: {url}")

            bucket_name = parts[0]
            storage_path = "/".join(parts[1:])
            # 下载文件
            try:
                res =
self.supabase.storage.from_(bucket_name).download(storage_path)
                with open(local_path, 'wb') as f:
                    f.write(res)
                print(f"从Supabase下载PDF成功: {url} ->
{local_path}")
                return True
            except Exception as e:
                logger.error(f"从Supabase下载PDF失败: {url}, 错误:
{e}")
                raise
        else:
            # 普通HTTP URL
            async with httpx.AsyncClient() as client:
                response = await client.get(url,
follow_redirects=True)
                response.raise_for_status()

                with open(local_path, 'wb') as f:
                    f.write(response.content)

                print(f"下载PDF成功: {url} -> {local_path}")
                return True

    except Exception as e:
        logger.error(f"下载PDF失败: {url}, 错误: {e}")
        return False

def _get_html_template(self) -> str:
    """获取HTML模板内容（首先尝试从文件加载，如果失败则使用内嵌模板）"""
    try:
        if os.path.exists(self.html_template_path):
                print(f"找到HTML模板文件: {self.html_template_path}")
                with open(self.html_template_path, 'r',
encoding='utf-8') as f:
                    return f.read()
            else:
                # 在这里可以添加内嵌的HTML模板，但由于模板很大，而且文件可以部署
到Railway，所以我们不内嵌模板
                logger.warning(f"HTML模板文件不存在:
{self.html_template_path}")
                return ""
        except Exception as e:
            logger.error(f"读取HTML模板时出错: {e}")
            return ""

async def upload_file_to_supabase(self, local_path: str,
storage_path: str) -> str:
    """上传本地文件到Supabase Storage"""
    try:
        # 确保文件存在
        if not os.path.exists(local_path):
            raise FileNotFoundError(f"要上传的文件不存在:
{local_path}")

        # 检查文件大小
        file_size = os.path.getsize(local_path)
        print(f"文件大小: {file_size} 字节")

        # 根据文件扩展名确定MIME类型
        import mimetypes
        mime_type, _ = mimetypes.guess_type(local_path)
        if not mime_type:
            if storage_path.endswith('.html'):
                mime_type = 'text/html'
            elif storage_path.endswith('.pdf'):
                mime_type = 'application/pdf'
            elif storage_path.endswith('.png'):
                mime_type = 'image/png'
            elif storage_path.endswith('.jpg') or
storage_path.endswith('.jpeg'):
                mime_type = 'image/jpeg'
            elif storage_path.endswith('.webp'):
                mime_type = 'image/webp'
            elif storage_path.endswith('.xlsx'):
                mime_type = 'application/vnd.openxmlformats-
officedocument.spreadsheetml.sheet'
            elif storage_path.endswith('.xls'):
                mime_type = 'application/vnd.ms-excel'
            else:
                mime_type = 'application/octet-stream'

        print(f"上传文件 {local_path} 的MIME类型: {mime_type}")

        # 添加额外的日志输出
        print(f"正在使用MIME类型 {mime_type} 上传文件 {local_path} 到
{storage_path}")

        with open(local_path, 'rb') as f:
            # 添加明确的MIME类型和缓存控制
            file_options = {
                "content-type": mime_type,
                "cache-control": "3600",
                "upsert": "true"
            }

            print(f"文件选项: {file_options}")

            self.supabase.storage.from_('results').upload(
                storage_path,
                f,
                file_options=file_options
            )

        # 生成公开访问URL
        public_url =
self.supabase.storage.from_('results').get_public_url(storage_path)
        logger.info(f"文件上传成功: {local_path} -> {storage_path}")
        return public_url

    except Exception as e:
        logger.error(f"文件上传失败: {local_path} -> {storage_path},
错误: {e}")
        raise

async def step1_extract_pdf_data(self) -> pd.DataFrame:
    """步骤1: 下载PDF并提取数据"""
    print("\n=== 步骤1: PDF数据提取 ===")

    # 下载PDF文件
    print("下载PDF文件...")
    pdf_download_tasks = [
        self._download_pdf(self.pdf_urls[0],
self.scheme1_pdf_path),
        self._download_pdf(self.pdf_urls[1], self.scheme2_pdf_path)
    ]

    results = await asyncio.gather(*pdf_download_tasks)
    if not all(results):
        print("错误: PDF下载失败")
        return pd.DataFrame()

    print(f"PDF下载成功: {self.scheme1_pdf_path},
{self.scheme2_pdf_path}")

    # 提取PDF数据
    extractor = FinalFilteredExtractor([self.scheme1_pdf_path,
self.scheme2_pdf_path])
    all_data = []

    # 处理两个PDF文件
    pdf_files = [
        (extractor.scheme1_path, "方案1"),
        (extractor.scheme2_path, "方案2")
    ]

    for pdf_path, scheme_name in pdf_files:
        print(f"\n处理PDF文件: {pdf_path} ({scheme_name})")

        if not os.path.exists(pdf_path):
            print(f"错误: PDF文件不存在 - {pdf_path}")
            continue

        # 提取客户信息
        customer_info = extractor.extract_customer_info(pdf_path)
        print(f"  提取的客户信息: {customer_info}")

        # 查找表格页面（应用排除规则）
        table_pages = extractor.find_table_pages(pdf_path)

        print(f"  退保发还金额页面:
{table_pages['surrender_value']}")
        print(f"  现金提取后退保发还金额页面:
{table_pages['withdrawal_surrender_value']}")

        # 提取退保发还金额数据
        if table_pages['surrender_value']:
            surrender_data =
extractor.extract_table_data_filtered(pdf_path,
table_pages['surrender_value'], 'surrender_value')
            for record in surrender_data:
                record['scheme'] = scheme_name
                # 添加客户信息到每条记录
                for key, value in customer_info.items():
                    record[f'customer_{key}'] = value
                all_data.append(record)

        # 提取现金提取后退保发还金额数据
        if table_pages['withdrawal_surrender_value']:
            withdrawal_data =
extractor.extract_table_data_filtered(pdf_path,
table_pages['withdrawal_surrender_value'],
'withdrawal_surrender_value')
            for record in withdrawal_data:
                record['scheme'] = scheme_name
                # 添加客户信息到每条记录
                for key, value in customer_info.items():
                    record[f'customer_{key}'] = value
                all_data.append(record)

    # 转换为DataFrame
    df = pd.DataFrame(all_data)

    if df.empty:
        print("警告: 没有提取到任何数据")
        return df

    print(f"\n数据提取完成:")
    print(f"  总记录数: {len(df)}")
    print(f"  包含方案: {df['scheme'].unique().tolist()}")

    # 保存原始提取数据
    df.to_excel(self.extracted_data_file, index=False)
    print(f"  原始数据已保存到: {self.extracted_data_file}")

    return df
```

```
def step2_calculate_irr(self, extracted_df: pd.DataFrame) ->
pd.DataFrame:
    """步骤2: 计算IRR（使用正确的IRR计算器）"""
    print("\n=== 步骤2: IRR计算（使用正确的IRR计算器） ===")

    if extracted_df.empty:
        print("错误: 没有可用的提取数据")
        return pd.DataFrame()

    # 初始化结果列表和计算器
    irr_results = []
    calculator = CorrectedIRRCalculator()

    # 动态获取客户信息（避免硬编码）
    first_row = extracted_df.iloc[0] if not extracted_df.empty
else None
    if first_row is not None:
        annual_premium = first_row.get('customer_annual_premium',
200000)  # 使用提取的年保费
        payment_period = first_row.get('customer_payment_period',
5)  # 缴费期通常为5年
    else:
        annual_premium = 200000
        payment_period = 5

    print(f"使用提取的客户信息: 年保费=${annual_premium:,}, 缴费期
={payment_period}年")

    for _, row in extracted_df.iterrows():
        try:
            scheme = row['scheme']
            table_type = row['table_type']
            policy_year = int(row['policy_year'])
            surrender_value = row['surrender_value']
            withdrawal_amount = row.get('withdrawal_amount', 0)
            if policy_year <= 0 or surrender_value <= 0:
                continue

            # 根据table_type选择不同的现金流构建方法（完全按照原脚本逻辑）
            if table_type == 'surrender_value':
                # 一次性退保方案
                cash_flows =
calculator.build_cash_flows_for_surrender(
                    annual_premium, payment_period, policy_year,
surrender_value
                )
                withdrawal_start_year = None  # 没有提取

            elif table_type == 'withdrawal_surrender_value':
                # 现金提取方案（动态确定开始年份）
                # 从当前方案的所有现金提取数据中找到第一个非零提取年份
                scheme_withdrawal_data =
extracted_df[(extracted_df['scheme'] == scheme) &

(extracted_df['table_type'] == 'withdrawal_surrender_value') &

(extracted_df['withdrawal_amount'] > 0)]

                if not scheme_withdrawal_data.empty:
                    withdrawal_start_year =
int(scheme_withdrawal_data['policy_year'].min())
                else:
                    # 如果没找到，使用硬编码的默认值
                    if scheme == '方案1':
                        withdrawal_start_year = 6
                    elif scheme == '方案2':
                        withdrawal_start_year = 10
                    else:
                        withdrawal_start_year = policy_year
                cash_flows =
calculator.build_cash_flows_for_withdrawal(
                    annual_premium, payment_period, policy_year,
                    withdrawal_start_year, withdrawal_amount,
surrender_value
                )
            else:
                continue

            # 计算IRR
            irr = calculator.calculate_irr(cash_flows)
            irr_percentage = f"{irr * 100:.2f}%" if irr and not
np.isnan(irr) else "N/A"

            irr_result = {
                'scheme': scheme,
                'page': row['page'],
                'table_type': table_type,
                'age': row['age'],
                'policy_year': policy_year,
                'surrender_value': surrender_value,
                'withdrawal_amount': withdrawal_amount,
                'annual_premium': annual_premium,
                'payment_period': payment_period,
                'withdrawal_start_year': withdrawal_start_year if
table_type == 'withdrawal_surrender_value' else None,
                'cash_flows': str(cash_flows),
                'irr': irr_percentage,
                'irr_value': irr,
                # 添加客户信息列
                'customer_name': row.get('customer_name', 'VIP 先
生'),
                'customer_age': row.get('customer_age', 0),
                'customer_annual_premium':
```

```
row.get('customer_annual_premium', annual_premium),
                'customer_payment_period':
row.get('customer_payment_period', payment_period),
                'customer_total_premium':
row.get('customer_total_premium', annual_premium * payment_period),
                'customer_currency': row.get('customer_currency',
'美元'),
                'customer_coverage_period':
row.get('customer_coverage_period', '终身')
            }

            irr_results.append(irr_result)
            print(f"IRR计算结果: {scheme}, {table_type}, 年份
={policy_year}, IRR={irr_percentage}")

        except Exception as e:
            print(f"计算IRR时出错: {e}")
            continue

    irr_df = pd.DataFrame(irr_results)

    print(f"\nIRR计算完成:")
    print(f"  总记录数: {len(irr_df)}")
    print(f"  成功计算IRR的记录:
{len(irr_df[irr_df['irr'].notna()])}")

    # 保存IRR结果
    irr_df.to_excel(self.irr_results_file, index=False)
    print(f"  IRR结果已保存到: {self.irr_results_file}")

    return irr_df

def calculate_cumulative_withdrawal(self, df: pd.DataFrame) ->
pd.DataFrame:
    """计算累计提取金额（完全基于实际数据，不使用硬编码）"""
    print("\n=== 计算累计提取金额 ===")
    df = df.copy()
    df['cumulative_withdrawal'] = 0.0

    # 按方案和表格类型分组计算累计提取
    for scheme in df['scheme'].unique():
        for table_type in df['table_type'].unique():
            if table_type != 'withdrawal_surrender_value':
                continue

            mask = (df['scheme'] == scheme) & (df['table_type'] ==
table_type)
            subset = df[mask].sort_values(['age', 'policy_year'])

            # 找到该方案实际的提取开始年份（基于数据而不是硬编码）
            withdrawal_records =
subset[subset['withdrawal_amount'] > 0]
            if withdrawal_records.empty:
                print(f"{scheme} 没有现金提取记录，跳过")
                continue

            actual_start_year =
withdrawal_records['policy_year'].min()
            print(f"{scheme} 实际提取开始年份: 第{actual_start_year}
年")

            for idx in subset.index:
                policy_year = df.loc[idx, 'policy_year']
                withdrawal_amount = df.loc[idx,
'withdrawal_amount']

                # 只有有提取金额的年份才计算累计提取
                if withdrawal_amount > 0 and policy_year >=
actual_start_year:
                    years_of_withdrawal = policy_year -
```

```
actual_start_year + 1
                    df.loc[idx, 'cumulative_withdrawal'] =
withdrawal_amount * years_of_withdrawal

    print(f"累计提取金额计算完成，处理了
{len(df[df['cumulative_withdrawal'] > 0])} 条记录")

    # 验证计算结果
    for scheme in df['scheme'].unique():
        withdrawal_data = df[(df['scheme'] == scheme) &
(df['table_type'] == 'withdrawal_surrender_value')]
        withdrawal_records =
withdrawal_data[withdrawal_data['withdrawal_amount'] >
0].sort_values('policy_year')
        if not withdrawal_records.empty:
            print(f"{scheme} 前3条累计提取验证:")
            for i, (idx, row) in
enumerate(withdrawal_records.head(3).iterrows()):
                print(f"  第{row['policy_year']}年: 年提取<span
class="math-inline" style="display: inline;"><math xmlns="http://
www.w3.org/1998/Math/MathML" display="inline"><mrow><mrow><mi>r</
mi><mi>o</mi><mi>w</mi><msup><mo stretchy="false">[</
mo><mi>&#x02032;</mi></msup><mi>w</mi><mi>i</mi><mi>t</mi><mi>h</
mi><mi>d</mi><mi>r</mi><mi>a</mi><mi>w</mi><mi>a</mi><msub><mi>l</
mi><mi>a</mi></msub><mi>m</mi><mi>o</mi><mi>u</mi><mi>n</
mi><msup><mi>t</mi><mi>&#x02032;</mi></msup><mo
stretchy="false">]</mo><mi>:</mi><mo>&#x0002C;</mo></
mrow><mo>&#x0002C;</mo><mi>累</mi><mi>计</mi></mrow></math></
span>{row['cumulative_withdrawal']:,}")

    return df

def step3_generate_html(self, irr_df: pd.DataFrame) -> str:
    """步骤3: 生成HTML（根据模板说明和用户要求重新构建）"""
    print("\n=== 步骤3: HTML生成（重新构建版本） ===")
    if irr_df.empty:
        print("错误: 没有IRR数据可供生成HTML")
        return ""

    # 读取HTML模板
    html_template = self._get_html_template()
    if not html_template:
        print("错误: 无法获取HTML模板")
        return ""

    print(f"HTML模板读取成功, 长度: {len(html_template)} 字符")

    # 分析两个方案的数据
    scheme1_data = irr_df[irr_df['scheme'] == '方案1'].copy()
    scheme2_data = irr_df[irr_df['scheme'] == '方案2'].copy()

    print(f"方案1数据点: {len(scheme1_data)}")
    print(f"方案2数据点: {len(scheme2_data)}")

    # 分离surrender_value和withdrawal_surrender_value数据
    scheme1_surrender = scheme1_data[scheme1_data['table_type'] ==
'surrender_value'].copy()
    scheme1_withdrawal = scheme1_data[scheme1_data['table_type']
== 'withdrawal_surrender_value'].copy()
    scheme2_surrender = scheme2_data[scheme2_data['table_type'] ==
'surrender_value'].copy()
    scheme2_withdrawal = scheme2_data[scheme2_data['table_type']
== 'withdrawal_surrender_value'].copy()

    print(f"方案1 一次性退保数据: {len(scheme1_surrender)}")
    print(f"方案1 现金提取数据: {len(scheme1_withdrawal)}")
    print(f"方案2 一次性退保数据: {len(scheme2_surrender)}")
    print(f"方案2 现金提取数据: {len(scheme2_withdrawal)}")
    def get_data_by_year(data, policy_year):
        """根据保单年度获取数据"""
        year_data = data[data['policy_year'] == policy_year]
        if not year_data.empty:
            row = year_data.iloc[0]
            return {
                'surrender_value': int(row['surrender_value']),
                'withdrawal_amount': int(row['withdrawal_amount'])
if pd.notna(row['withdrawal_amount']) else 0,
                'cumulative_withdrawal':
int(row['cumulative_withdrawal']) if
pd.notna(row['cumulative_withdrawal']) else 0,
                'irr': row['irr']
            }
        return None

    # 生成动态副标题
    current_time = datetime.now().strftime("%Y年%m月%d日")

    # 从原数据中获取客户信息
    customer_info = {}
    if not irr_df.empty:
        first_row = irr_df.iloc[0]
        customer_info = {
            'name': first_row.get('customer_name', 'VIP 先生'),
            'age': first_row.get('customer_age', 0),
            'annual_premium':
first_row.get('customer_annual_premium', 200000),
            'payment_period':
first_row.get('customer_payment_period', 5),
            'total_premium':
first_row.get('customer_total_premium', 1000000),
            'currency': first_row.get('customer_currency', '美元'),
            'coverage_period':
first_row.get('customer_coverage_period', '终身')
```

```
        }
    else:
        # 默认值
        customer_info = {
            'name': 'VIP 先生',
            'age': 0,
            'annual_premium': 200000,
            'payment_period': 5,
            'total_premium': 1000000,
            'currency': '美元',
            'coverage_period': '终身'
        }

    print(f"使用客户信息: 姓名={customer_info['name']}, 年龄
={customer_info['age']}, 年缴保费=$
{customer_info['annual_premium']:,}")

    # 创建替换字典
    replacements = {
        # 基础信息
        '{{DOCUMENT_TITLE}}': '高息美元储蓄方案',
        '{{HEADER_PLAN_TITLE}}': '高息美元储蓄方案',

        # 客户信息
        '{{CUSTOMER_INFO_LABEL_NAME}}': '客户姓名',
        '{{CUSTOMER_INFO_NAME}}': customer_info['name'],
        '{{CUSTOMER_INFO_LABEL_AGE}}': '投保年龄',
        '{{CUSTOMER_INFO_AGE}}': f"{customer_info['age']}岁",
        '{{CUSTOMER_INFO_LABEL_CURRENCY}}': '保单货币',
        '{{CUSTOMER_INFO_CURRENCY}}': customer_info['currency'],
        '{{CUSTOMER_INFO_LABEL_COVERAGE_PERIOD}}': '保障期间',
        '{{CUSTOMER_INFO_COVERAGE_PERIOD}}':
customer_info['coverage_period'],

        # 保费信息
        '{{PREMIUM_INFO_LABEL_ANNUAL}}': '年缴保费',
        '{{PREMIUM_INFO_ANNUAL_AMOUNT}}': f"$
{customer_info['annual_premium']:,}",
        '{{PREMIUM_INFO_LABEL_PAYMENT_PERIOD}}': '缴费期',
        '{{PREMIUM_INFO_PAYMENT_PERIOD}}':
f"{customer_info['payment_period']}年",
        '{{PREMIUM_INFO_LABEL_TOTAL}}': '总保费',
        '{{PREMIUM_INFO_TOTAL_AMOUNT}}': f"$
{customer_info['total_premium']:,}",

        # 方案标题
        '{{PLAN1_MAIN_TITLE}}': '方案1：一次性提取',
        '{{PLAN_COMPARISON_TITLE}}': '方案对比',
        '{{PLAN_FIXED_ANNUITY_TITLE}}': '固定年金',

        # 标签
        '{{LABEL_START_TIME}}': '开始提取时间:',
        '{{LABEL_DURATION}}': '提取时间:',
        '{{LABEL_CUMULATIVE_WITHDRAWAL}}': '累计提取:',
        '{{LABEL_REMAINING_VALUE}}': '剩余价值:',

        # 备注
        '{{FOOTER_NOTE_1}}': '* 以上数据如有错漏，请以计划书为准',
        '{{FOOTER_NOTE_2}}': '* 本计划书仅供参考，不构成任何法律文件'
    }

    # 获取客户年龄
    client_age = customer_info['age']

    # 方案1数据填充（一次性退保）- 根据用户要求：第10年、第20年、第30年、65
岁、80岁、90岁、100岁
    # 计算年龄对应的保单年度（确保至少为1）
    policy_year_for_age_65 = max(65 - client_age, 1)
    policy_year_for_age_80 = max(80 - client_age, 1)
    policy_year_for_age_90 = max(90 - client_age, 1)
    policy_year_for_age_100 = max(100 - client_age, 1)

    print(f"客户投保年龄: {client_age}岁")
    print(f"65岁对应第{policy_year_for_age_65}年, 80岁对应第
{policy_year_for_age_80}年, 90岁对应第{policy_year_for_age_90}年, 100
岁对应第{policy_year_for_age_100}年")

    plan1_years = [10, 20, 30, policy_year_for_age_65,
policy_year_for_age_80, policy_year_for_age_90,
policy_year_for_age_100]
    plan1_labels = ['第10年', '第20年', '第30年', '第65岁', '第80岁',
'第90岁', '第100岁']

    for i, (policy_year, label) in enumerate(zip(plan1_years,
plan1_labels), 1):
        data = get_data_by_year(scheme1_surrender, policy_year)
        if data:
            replacements[f'{{{{PLAN1_YEAR_{i}}}}}'] = label
            replacements[f'{{{{PLAN1_AMOUNT_{i}}}}}'] = f"$
{data['surrender_value']:,}"
            replacements[f'{{{{PLAN1_RATE_{i}}}}}'] = data['irr']
        else:
            replacements[f'{{{{PLAN1_YEAR_{i}}}}}'] = label
            replacements[f'{{{{PLAN1_AMOUNT_{i}}}}}'] = "N/A"
            replacements[f'{{{{PLAN1_RATE_{i}}}}}'] = "N/A"

    # 方案2数据填充（现金提取）- 修改为：20年、30年、70岁、90岁、100岁
    # 计算年龄对应的保单年度（确保至少为1）
    policy_year_for_age_70 = max(70 - client_age, 1)
    policy_year_for_age_90 = max(90 - client_age, 1)
    policy_year_for_age_100 = max(100 - client_age, 1)

    print(f"70岁对应第{policy_year_for_age_70}年, 90岁对应第
{policy_year_for_age_90}年, 100岁对应第{policy_year_for_age_100}年")
```

```
    plan2_years = [20, 30, policy_year_for_age_70,
policy_year_for_age_90, policy_year_for_age_100]
    plan2_labels = [
        '第20年',
        '第30年',
        f'第70岁',
        f'第90岁',
        f'第100岁',
    ]

    for i, (policy_year, label) in enumerate(zip(plan2_years,
plan2_labels), 1):
        withdrawal_data = get_data_by_year(scheme1_withdrawal,
policy_year)  # 使用方案1的withdrawal数据
        if withdrawal_data:
            # 使用已计算的累计提取金额
            cumulative_withdrawal =
withdrawal_data['cumulative_withdrawal']

            # 计算提取百分比 Y% (累计提取金额 / 总保费)
            total_premium = customer_info['total_premium']
            withdrawal_percentage = (cumulative_withdrawal /
total_premium * 100) if total_premium > 0 else 0

            replacements[f'{{{{PLAN2_YEAR_{i}}}}}'] = label

replacements[f'{{{{PLAN2_CUMULATIVE_WITHDRAWAL_{i}}}}}'] = f"$
{cumulative_withdrawal:,}"
            replacements[f'{{{{PLAN2_REMAINING_VALUE_{i}}}}}'] =
f"${withdrawal_data['surrender_value']:,}"
            replacements[f'{{{{PLAN2_RATE_{i}}}}}'] =
withdrawal_data['irr']
            replacements[f'{{{{PLAN2_WITHDRAWAL_PERCENT_{i}}}}}']
= f"{withdrawal_percentage:.1f}%"
        else:
            replacements[f'{{{{PLAN2_YEAR_{i}}}}}'] = label

replacements[f'{{{{PLAN2_CUMULATIVE_WITHDRAWAL_{i}}}}}'] = "N/A"
            replacements[f'{{{{PLAN2_REMAINING_VALUE_{i}}}}}'] =
"N/A"
            replacements[f'{{{{PLAN2_RATE_{i}}}}}'] = "N/A"
            replacements[f'{{{{PLAN2_WITHDRAWAL_PERCENT_{i}}}}}']
= "N/A"

    # 方案3数据填充（使用方案2的数据）
    for i, (policy_year, label) in enumerate(zip(plan2_years,
plan2_labels), 1):
        withdrawal_data = get_data_by_year(scheme2_withdrawal,
policy_year)  # 使用方案2的withdrawal数据
        if withdrawal_data:
            # 使用已计算的累计提取金额
            cumulative_withdrawal =
withdrawal_data['cumulative_withdrawal']

            # 计算提取百分比 Y% (累计提取金额 / 总保费)
            total_premium = customer_info['total_premium']
            withdrawal_percentage = (cumulative_withdrawal /
total_premium * 100) if total_premium > 0 else 0

            replacements[f'{{{{PLAN3_YEAR_{i}}}}}'] = label

replacements[f'{{{{PLAN3_CUMULATIVE_WITHDRAWAL_{i}}}}}'] = f"$
{cumulative_withdrawal:,}"
            replacements[f'{{{{PLAN3_REMAINING_VALUE_{i}}}}}'] =
f"${withdrawal_data['surrender_value']:,}"
            replacements[f'{{{{PLAN3_RATE_{i}}}}}'] =
withdrawal_data['irr']
            replacements[f'{{{{PLAN3_WITHDRAWAL_PERCENT_{i}}}}}']
= f"{withdrawal_percentage:.1f}%"
        else:
            replacements[f'{{{{PLAN3_YEAR_{i}}}}}'] = label

replacements[f'{{{{PLAN3_CUMULATIVE_WITHDRAWAL_{i}}}}}'] = "N/A"
            replacements[f'{{{{PLAN3_REMAINING_VALUE_{i}}}}}'] =
"N/A"
            replacements[f'{{{{PLAN3_RATE_{i}}}}}'] = "N/A"
            replacements[f'{{{{PLAN3_WITHDRAWAL_PERCENT_{i}}}}}']
= "N/A"

    # 固定年金数据 - 根据用户要求动态计算
    def calculate_fixed_annuity_data(withdrawal_data, scheme_name):
        """计算固定年金相关数据"""
        if withdrawal_data.empty:
            return {
                'subtitle': 'N/A',
                'start_time': 'N/A',
                'duration': 'N/A',
                'cumulative_withdrawal': 'N/A',
                'remaining_value': 'N/A'
            }

        # 找到首个有现金提取的年度
        withdrawal_records =
withdrawal_data[withdrawal_data['withdrawal_amount'] >
0].sort_values('policy_year')
        if withdrawal_records.empty:
            return {
                'subtitle': 'N/A',
                'start_time': 'N/A',
                'duration': 'N/A',
                'cumulative_withdrawal': 'N/A',
                'remaining_value': 'N/A'
            }

        start_year = withdrawal_records.iloc[0]['policy_year']
        annual_withdrawal = withdrawal_records.iloc[0]
['withdrawal_amount']
        total_premium = customer_info['total_premium']

        # 计算年提取百分比
        withdrawal_percentage = (annual_withdrawal / total_premium
* 100) if total_premium > 0 else 0

        # 计算有现金提取的年度总数
        duration = len(withdrawal_records)

        # 获取最大的累计现金提取金额
        max_cumulative =
withdrawal_records['cumulative_withdrawal'].max()

        # 获取最后一个年度的剩余价值
        last_record = withdrawal_records.iloc[-1]
        remaining_value = last_record['surrender_value']

        print(f"{scheme_name}固定年金数据: 起始年度={start_year}, 提取
比例={withdrawal_percentage:.1f}%, 持续年数={duration}, 累计提取=<span
class="math-inline" style="display: inline;"><math xmlns="http://
www.w3.org/1998/Math/MathML" display="inline"><mrow><mrow><mi>m</
mi><mi>a</mi><msub><mi>x</mi><mi>c</mi></msub><mi>u</mi><mi>m</
mi><mi>u</mi><mi>l</mi><mi>a</mi><mi>t</mi><mi>i</mi><mi>v</
mi><mi>e</mi><mi>:</mi><mo>&#x0002C;</mo><mi>.0</mi><mi>f</mi></
mrow><mo>&#x0002C;</mo><mi>剩</mi><mi>余</mi><mi>价</mi><mi>值</
mi><mo>&#x0003D;</mo></mrow></math></span>{remaining_value:,.0f}")

        return {
            'subtitle': f'第{start_year}年起每年提取
{withdrawal_percentage:.1f}%',
            'start_time': f'第{start_year}年',
            'duration': f'{duration}年',
            'cumulative_withdrawal': f'${max_cumulative:,.0f}',
            'remaining_value': f'${remaining_value:,.0f}'
        }

    # 计算方案1（PLAN2）固定年金数据
    plan2_annuity =
calculate_fixed_annuity_data(scheme1_withdrawal, "方案1")

    # 计算方案2（PLAN3）固定年金数据
    plan3_annuity =
calculate_fixed_annuity_data(scheme2_withdrawal, "方案2")

    # 固定年金数据填充
    replacements.update({
        '{{FIXED_ANNUITY_PLAN2_SUBTITLE}}':
plan2_annuity['subtitle'],
        '{{FIXED_ANNUITY_PLAN2_START_TIME}}':
plan2_annuity['start_time'],
        '{{FIXED_ANNUITY_PLAN2_DURATION}}':
plan2_annuity['duration'],
        '{{FIXED_ANNUITY_PLAN2_CUMULATIVE_WITHDRAWAL}}':
plan2_annuity['cumulative_withdrawal'],
        '{{FIXED_ANNUITY_PLAN2_REMAINING_VALUE}}':
plan2_annuity['remaining_value'],

        '{{FIXED_ANNUITY_PLAN3_SUBTITLE}}':
plan3_annuity['subtitle'],
        '{{FIXED_ANNUITY_PLAN3_START_TIME}}':
plan3_annuity['start_time'],
        '{{FIXED_ANNUITY_PLAN3_DURATION}}':
plan3_annuity['duration'],
        '{{FIXED_ANNUITY_PLAN3_CUMULATIVE_WITHDRAWAL}}':
plan3_annuity['cumulative_withdrawal'],
        '{{FIXED_ANNUITY_PLAN3_REMAINING_VALUE}}':
plan3_annuity['remaining_value'],
    })
    # 执行替换
    html_content = html_template
    replaced_count = 0
    for placeholder, value in replacements.items():
        if placeholder in html_content:
            html_content = html_content.replace(placeholder,
str(value))
            replaced_count += 1

    print(f"已替换 {replaced_count} 个占位符")

    # 处理可能遗留的占位符
    remaining_placeholders = re.findall(r'\{\{[^}]+\}\}',
html_content)
    if remaining_placeholders:
        print(f"仍有 {len(remaining_placeholders)} 个占位符未被替换")
        for placeholder in remaining_placeholders[:10]:  # 只显示前
10个
            print(f"  - {placeholder}")
            html_content = html_content.replace(placeholder, "N/A")

    # 保存HTML文件
    with open(self.final_html_file, 'w', encoding='utf-8') as f:
        f.write(html_content)

    print(f"HTML文件生成完成: {self.final_html_file}")
    print(f"HTML文件大小: {len(html_content)} 字符")

    return self.final_html_file

async def step4_generate_screenshot(self, html_file: str) -> str:
    """步骤4: 生成截图（异步版本）"""
    print("\n=== 步骤4: 截图生成 ===")
    if not PLAYWRIGHT_AVAILABLE:
        print("错误: Playwright未安装，无法生成截图")
        return ""

    if not html_file or not os.path.exists(html_file):
        print(f"错误: HTML文件不存在 - {html_file}")
        return ""

    try:
        from playwright.async_api import async_playwright

        async with async_playwright() as playwright:
            browser = await playwright.chromium.launch(
                headless=True,
                args=[
                    '--no-sandbox',
                    '--disable-setuid-sandbox',
                    '--disable-dev-shm-usage',
                    '--disable-accelerated-2d-canvas',
                    '--no-first-run',
                    '--no-zygote',
                    '--single-process',
                    '--disable-gpu'
                ]
            )
            context = await browser.new_context(
                viewport={'width': 1200, 'height': 1600},
                device_scale_factor=2
            )
            page = await context.new_page()

            # 加载HTML文件
            file_url = f"file://{os.path.abspath(html_file)}"
            print(f"加载HTML文件: {file_url}")
            await page.goto(file_url)
            # 等待页面完全加载
            await page.wait_for_load_state("networkidle")
            await page.wait_for_load_state("domcontentloaded")
            await asyncio.sleep(3)  # 额外等待确保所有样式加载完成 (异步
等待)

            # 滚动到顶部确保内容可见
            await page.evaluate("window.scrollTo(0, 0)")
            await asyncio.sleep(1)  # 异步等待

            # 尝试定位主要内容容器，使用多种选择器
            container_selectors = [
                '.container',  # 常见的容器类名
                '.content',    # 内容容器
                'main',        # HTML5语义标签
                'body > div:first-child',  # body下第一个div
                'body > div',  # body下的div
                'body'         # 最后回退到body
            ]

            container_found = False
            for selector in container_selectors:
                try:
                    # 检查元素是否存在且可见
                    element = page.locator(selector).first
                    count = await element.count()
                    if count > 0:
                        # 检查元素是否有实际内容
                        bounding_box = await element.bounding_box()
                        if bounding_box and bounding_box['width']
> 100 and bounding_box['height'] > 100:
                            print(f"找到主要容器: {selector}")
                            print(f"容器尺寸: {bounding_box}")
                            # 直接截取容器元素，不需要手动裁剪
                            print(f"正在生成超高分辨率截图...")
                            await element.screenshot(
                                path=self.final_screenshot_file,
                                type='png'
                            )

                            container_found = True
                            break
                except Exception as e:
                    print(f"尝试选择器 '{selector}' 时出错: {e}")
                    continue

            # 如果没有找到合适的容器，回退到全页面截图
            if not container_found:
                print("未找到合适的容器，使用全页面截图...")
                await page.screenshot(
                    path=self.final_screenshot_file,
                    full_page=True,
                    type='png'
                )

            await browser.close()

            print(f"超高质量截图生成完成:
{self.final_screenshot_file}")

            # 验证截图文件
            if os.path.exists(self.final_screenshot_file):
                file_size =
os.path.getsize(self.final_screenshot_file)
                print(f"截图文件大小: {file_size:,} 字节")
                return self.final_screenshot_file
            else:
                print("错误: 截图文件未生成")
                return ""

        except Exception as e:
            print(f"生成截图时出错: {e}")
            import traceback
            traceback.print_exc()
            return ""

    async def run_complete_pipeline(self) -> Dict[str, str]:
        """运行完整的端到端流水线，返回结果文件URL"""
        print("开始执行Railway版本保险数据处理流水线")
        print("=" * 60)

        start_time = datetime.now()
        result_urls = {}

        try:
            # 步骤1: 提取PDF数据
            extracted_df = await self.step1_extract_pdf_data()
            if extracted_df.empty:
                raise Exception("PDF数据提取失败")

            # 步骤2: 计算IRR
            irr_df = self.step2_calculate_irr(extracted_df)
            if irr_df.empty:
                raise Exception("IRR计算失败")

            # 步骤2.5: 计算累计提取金额
            irr_df = self.calculate_cumulative_withdrawal(irr_df)

            # 重新保存包含累计提取金额的IRR结果
            irr_df.to_excel(self.irr_results_file, index=False)
            print(f"更新的IRR结果（含累计提取金额）已保存到:
{self.irr_results_file}")
```

```
        # 步骤3: 生成HTML
        html_file = self.step3_generate_html(irr_df)
        if not html_file:
            raise Exception("HTML生成失败")

        # 步骤4: 生成截图
        screenshot_file = await
self.step4_generate_screenshot(html_file)
        if not screenshot_file:
            print("警告: 截图生成失败, 但其他步骤已完成")

        # 步骤5: 上传结果到Supabase
        print("\n=== 步骤5: 上传结果到Supabase ===")

        # 上传HTML文件
        if os.path.exists(self.final_html_file):
            html_url = await self.upload_file_to_supabase(
                self.final_html_file,
                f"results/{self.task_id}/report.html"
            )
            result_urls['html_url'] = html_url

        # 上传截图文件
        if os.path.exists(self.final_screenshot_file):
            screenshot_url = await self.upload_file_to_supabase(
                self.final_screenshot_file,
                f"results/{self.task_id}/screenshot.png"
            )
            result_urls['screenshot_url'] = screenshot_url

        # 上传Excel数据文件
        if os.path.exists(self.irr_results_file):
            excel_url = await self.upload_file_to_supabase(
                self.irr_results_file,
                f"results/{self.task_id}/data.xlsx"
            )
            result_urls['excel_url'] = excel_url

        # 计算处理时间
        end_time = datetime.now()
        duration = end_time - start_time

        print("\n" + "=" * 60)
        print("Railway流水线执行完成！")
        print(f"总耗时: {duration}")
        print("\n生成的文件:")
        for key, url in result_urls.items():
            print(f"  - {key}: {url}")

        # 保存结果到数据库（如果Supabase客户端可用）
        if self.supabase:
            try:
                self.supabase.table('tasks').update({
                    'status': 'completed',
                    'results': result_urls,
                    'completed_at': datetime.now().isoformat()
                }).eq('id', self.task_id).execute()
                print(f"任务结果已更新到数据库: {self.task_id}")
            except Exception as e:
                logger.error(f"更新任务状态时出错: {e}")

        # 清理临时文件
        self._cleanup()

        return result_urls

    except Exception as e:
        print(f"流水线执行过程中发生错误: {e}")
        import traceback
        traceback.print_exc()
        # 更新任务状态为失败（如果Supabase客户端可用）
        if self.supabase:
            try:
                self.supabase.table('tasks').update({
                    'status': 'failed',
                    'error': str(e),
                    'completed_at': datetime.now().isoformat()
                }).eq('id', self.task_id).execute()
                print(f"任务失败状态已更新到数据库: {self.task_id}")
            except Exception as db_err:
                logger.error(f"更新任务失败状态时出错: {db_err}")

        # 清理临时文件
        self._cleanup()

        raise
```

async def process_insurance_pdfs(pdf_urls, supabase_url=None, supabase_key=None, task_id=None) -> Dict[str, str]:
"""处理保险PDF文件并返回结果URL

```
Args:
    pdf_urls: PDF文件URL列表（至少2个）
    supabase_url: Supabase URL
    supabase_key: Supabase Key
    task_id: 任务ID

Returns:
    Dict[str, str]: 结果文件URL字典
"""
pipeline = RailwayInsurancePipeline(
    pdf_urls=pdf_urls,
    supabase_url=supabase_url,
    supabase_key=supabase_key,
    task_id=task_id
)

return await pipeline.run_complete_pipeline()
```