ChatGPT

OCR 파이프라인 입력 구조 및 멀티스레딩 리팩토링

기존 **OCR 파이프라인** 코드를 새 **API 기반 입력 형식(**`dUserInput` **구조)**에 맞게 전면적으로 리팩토링했습니다. 주요 변경사항은 다음과 같습니다:

- `in_params` **기반 입력 제거:** 함수 시그니처 및 내부 로직에서 `in_params` 사용을 삭제하고 `dUserInput`을 활용하도록 수정했습니다. 특히 날짜 입력은 `dUserInput['CcrParams']['TargetDate']`에서 가져옵니다.
- **Azure 설정 분리:** Azure Form Recognizer 엔드포인트와 키 등의 설정은 소스코드에서 제거하고 별도의 JSON 설정 파일(`azure_config.json` 등)로 관리합니다. OCR 모듈이 해당 파일을 로드하여 사용하도록 변경했습니다.
- **디렉토리 경로 상수화:** 파일 저장에 사용되는 경로들은 인자를 통해 받지 않고, 코드 내에 **고정 상수**(`./downloads`, `./converted` 등)로 정의했습니다. 예를 들어 다운로드 디렉토리와 OCR 결과 저장 디렉토리 등을 상수로 지정하고 사용합니다.
- **멀티스레딩 로직 개선:** 기존의 `ThreadPoolExecutor` 기반 멀티스레딩 코드를 제거하고, **회사 공통 유틸리티**의 `idp_utils.wrapped_func` 및 `idp_utils.run_in_multi_thread` 함수를 사용하도록 변경했습니다. 각 레코드 처리를 `wrapped_func`로 감싼 작업 목록을 만들고 `run_in_multi_thread`로 실행합니다.
- **변경된 부분 주석 표시:** 아래 코드 블록들에 변경된 줄에는 요구된 대로 `# 수정됨` 주석을 추가하여 표시하였습니다.

각 파일별로 주요 수정 내용을 정리하고, 변경된 코드 부분을 함께 제시합니다.

1. wrapper.py 수정

주요 변경점:
- `run_wrapper` 함수의 입력을 `in_params` 대신 `dUserInput: dict`로 받고, 내부에서 `TargetDate`를 추출합니다.
- 데이터베이스 접속 정보는 더 이상 `in_params`로 받지 않으며, 함수 내에서 별도의 설정 파일(예: `Module_config_dex.toml`)을 로드하여 DB 연결을 설정합니다.
- 조회된 레코드 목록을 처리할 때 기존 `ThreadPoolExecutor` 사용 코드를 삭제하고, `idp_utils.wrapped_func`와 `run_in_multi_thread`로 멀티스레딩을 수행합니다.
- 파이프라인 처리 후에는 열었던 DB 연결을 적절히 `close()`하여 자원을 반납합니다.

```
import tomllib
from sqlalchemy import create_engine
from idp_utils import wrapped_func, run_in_multi_thread  #  수정됨: 공통 유틸 임포트 추가

def run_wrapper(dUserInput: dict) -> None:  #  수정됨: dUserInput 기반 입력
    """
    지정한 날짜의 모든 DB 레코드를 조회하여 OCR 파이프라인을 실행합니다.
    """  # (함수 설명의 in_params 언급 제거)  수정됨
    logger.info("[시작] run_wrapper")
    # 1️⃣ 입력 날짜 추출
    target_date = dUserInput['CcrParams']['TargetDate']  #  수정됨: 사용자 입력에서 날짜 가져오기
```

```
    # 2️⃣ DB 연결 설정 (설정 파일 로드)
    with open("Module_config_dex.toml", "rb") as f:  #  수정됨: TOML 설정 로드
        config = tomllib.load(f)
    hana_conf = config.get("SAP HANA DB") or config.get("database") or config.get("hana", {})
    conn_str = f"hdbcli://{hana_conf['User']}:{hana_conf['Password']}@{hana_conf['Host']}:
{hana_conf['Port']}"
    engine = create_engine(conn_str)                  #  수정됨: DB 엔진 생성
    conn = engine.connect()                           #  수정됨: DB 연결 객체 생성

    # 3️⃣ 대상 날짜의 레코드 조회
    data_records = query_data_by_date(conn, target_date)  #  수정됨: conn과 target_date 전달
    if not data_records:
        logger.info("📭 처리할 데이터가 없습니다.")
        conn.close(); return  #  수정됨: 처리할 레코드 없으면 DB 연결 닫고 종료

    logger.info(f"총 {len(data_records)}건 처리 시작")
    # 4️⃣ 멀티스레드로 레코드 처리
    max_workers = 4  # 쓰레드 개수 (고정 또는 설정값)  수정됨
    tasks = [wrapped_func(process_single_record, record, conn) for record in data_records]
    #  수정됨: 각 레코드 처리 함수를 래핑하여 작업 리스트 생성
    run_in_multi_thread(tasks, max_workers=max_workers)
    #  수정됨: 공통 유틸 함수로 멀티스레드 실행

    conn.close()  #  수정됨: 모든 처리 완료 후 DB 연결 닫기
    logger.info("  전체 파이프라인 완료")
    logger.info("[종료] run_wrapper")
```

`process_single_record` **함수 수정:**
`process_single_record` 함수는 각 레코드를 처리하는 파이프라인 단계 함수입니다. `in_params`를 제거하면서 함수 정의를 `def process_single_record(record: dict, conn)`으로 변경하여, DB 연결을 직접 전달받도록 했습니다. 내부에서 각 단계 함수 호출 시에도 수정된 인터페이스에 맞춰 호출합니다. Azure OCR 오류 시 실패 결과 JSON을 저장할 때 경로를 고정된 디렉토리를 사용하고, 후처리 및 DB 저장 호출 부분도 새 구조에 맞게 수정했습니다:

```
def process_single_record(record: dict, conn) -> None:  #  수정됨: DB 연결 conn 추가
  """
   하나의 DB 레코드에 대해 전처리 → Azure OCR → 후처리 → DB 저장 수행
  """
    logger.info(f"[시작] process_single_record - FIID={record.get('FIID')},
LINE_INDEX={record.get('LINE_INDEX')}")
    try:
        # 전처리 단계: 다운로드 및 YOLO 크롭
        cropped_list = run_pre_pre_process(record)  #  수정됨: in_params 없이 record만 전달

        for cropped in cropped_list:
            if "RESULT_CODE" in cropped:
                logger.warning(f"[SKIP] YOLO 오류 발생: {cropped}")
                continue
```

```
            # Azure OCR 단계
            ocr_result = run_azure_ocr(cropped)  #  수정됨: in_params 없이 cropped 데이터만 전달
            if ocr_result.get("RESULT_CODE") == "AZURE_ERR":
                logger.warning(f"[ERROR] Azure OCR 실패 → 오류 summary 저장 시도")
                # Azure 실패 시 후처리용 요약 JSON 생성
                now_str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                error_summary = {
                    "FIID": cropped.get("FIID"),
                    "LINE_INDEX": cropped.get("LINE_INDEX"),
                    "RECEIPT_INDEX": cropped.get("RECEIPT_INDEX"),
                    "COMMON_YN": cropped.get("COMMON_YN"),
                    "GUBUN": cropped.get("GUBUN"),
                    "ATTACH_FILE": record.get("ATTACH_FILE"),
                    "COUNTRY": None, "RECEIPT_TYPE": None, "MERCHANT_NAME": None, "MERCHANT_PHONE_NO": None,
                    "DELIVERY_ADDR": None, "TRANSACTION_DATE": None, "TRANSACTION_TIME": None,
                    "TOTAL_AMOUNT": None, "SUMTOTAL_AMOUNT": None, "TAX_AMOUNT": None, "BIZ_NO": None,
                    "RESULT_CODE": ocr_result.get("RESULT_CODE"),
                    "RESULT_MESSAGE": ocr_result.get("RESULT_MESSAGE"),
                    "CREATE_DATE": now_str, "UPDATE_DATE": now_str
                }
                # 실패 요약 JSON 저장 (post 결과 디렉토리 사용)
                os.makedirs(POST_JSON_DIR, exist_ok=True)  #  수정됨: 고정 경로 사용
                error_result_path = os.path.join(
                    POST_JSON_DIR,
                    f"fail_{error_summary['FIID']}_{error_summary['LINE_INDEX']}_{error_summary['RECEIPT_INDEX']}_post.json"
                )  #  수정됨
                with open(error_result_path, "w", encoding="utf-8") as f:
                    json.dump({"summary": error_summary, "items": []}, f, ensure_ascii=False, indent=2)
                # 실패 결과를 DB에 저장 (요약 정보만)
                insert_postprocessed_result(conn, error_result_path)  #  수정됨: conn과 경로 전달
                continue

            # Azure OCR 성공 시: OCR 결과 JSON 경로 생성
            base_name = os.path.splitext(os.path.basename(cropped['file_path']))[0]
            json_path = os.path.join(OCR_JSON_DIR, f"{base_name}.ocr.json")  #  수정됨: OCR JSON 고정 경로 사용

            # 후처리 단계 수행 및 결과 JSON 저장
            # (cropped 딕셔너리에 원본 ATTACH_FILE 추가하여 후처리에 전달)
            cropped_record = {**cropped, "json_path": json_path, "ATTACH_FILE": record.get("ATTACH_FILE")}  #  수정됨
            post_json_path = post_process_and_save(cropped_record)  #  수정됨: 설정 인자 없이 호출
```

```
            # 후처리 결과 DB 저장
            insert_postprocessed_result(conn, post_json_path)  #  수정됨
    except Exception as e:
        logger.error(f"[FATAL] 처리 중 오류 발생 - FIID={record.get('FIID')}: {e}", exc_info=True)
    logger.info(f"[종료] process_single_record - FIID={record.get('FIID')},
LINE_INDEX={record.get('LINE_INDEX')}")
```

참고: 위 코드에서 `POST_JSON_DIR`과 `OCR_JSON_DIR`은 고정 경로 상수로 정의된 전역 변수입니다(아래 섹션 참고). Azure OCR 오류 발생 시, 최종 후처리 결과 저장 디렉토리(`POST_JSON_DIR`)에 `fail_..._post.json` 형식의 빈 결과 파일을 저장하고, 곧바로 DB에 기록합니다.

2. pre_process.py 수정 (전처리 단계)

주요 변경점:
- 전처리 함수 `run_pre_pre_process`의 시그니처에서 `in_params`를 제거하고 `db_record`만 받도록 수정했습니다.
- 함수 내에서 사용하던 경로 (`download_dir`, `cropped_dir` 등)와 YOLO 모델 경로는 상수로 고정하였습니다. `DOWNLOAD_DIR`를 기준으로 필요한 하위 폴더(`document_merged`, `cropped` 등)를 생성하여 사용합니다.
- Azure와 무관한 부분이므로 Azure 관련 변경은 없습니다.

먼저, **고정 경로 상수와 YOLO 모델 경로**를 파일 상단에 정의했습니다:

```
# 고정 디렉토리 경로 정의
DOWNLOAD_DIR = "./downloads"                    #  수정됨: 원본 및 처리 파일 저장 기본 경로
MERGED_DOC_DIR = os.path.join(DOWNLOAD_DIR, "document_merged")  #  수정됨
CROPPED_DIR = os.path.join(DOWNLOAD_DIR, "cropped")             #  수정됨

YOLO_MODEL_PATH = "./yolo/best.pt"          #  수정됨: YOLO 모델 파일 경로
```

이제 `run_pre_pre_process` 함수의 변경 내용을 살펴보겠습니다:

```
def run_pre_pre_process(db_record: dict) -> list:  #  수정됨: in_params 제거, DB 레코드만 입력
    """
    전처리: 파일 다운로드 → PNG 변환/병합 → YOLO 크롭 실행
    """
    logger.info("[시작] run_pre_pre_process")
    try:
        download_dir = DOWNLOAD_DIR                 #  수정됨: 고정 경로 사용
        model_path = YOLO_MODEL_PATH                #  수정됨
        merged_doc_dir = MERGED_DOC_DIR             #  수정됨: 문서 병합 디렉토리
        model = YOLO(model_path)

        fiid = db_record["FIID"]
        line_index = db_record["LINE_INDEX"]
        gubun = db_record["GUBUN"]
        results = []
```

```
        # ATTACH_FILE과 FILE_PATH 두 종류에 대해 처리
        for file_type in ["ATTACH_FILE", "FILE_PATH"]:
            url = db_record.get(file_type)
            if not url:
                continue

            common_yn = 0 if file_type == "ATTACH_FILE" else 1
            receipt_index = 1 if file_type == "ATTACH_FILE" else None

            # 1) 파일 다운로드
            orig_path = download_file_from_url(url, download_dir, is_file_path=(file_type ==
"FILE_PATH"))
            if not orig_path:
                logger.info(f"[{file_type}] URL 다운로드 스킵됨")
                continue

            # 2) 문서 파일일 경우 PDF/DOCX 등 → 이미지 병합
            ext = os.path.splitext(orig_path)[1].lower()
            if ext in [".pdf", ".docx", ".pptx", ".xlsx"]:
                merged_path = process_document_file(orig_path, merged_doc_dir)
                if not merged_path:
                    logger.warning(f"[{file_type}] 문서 처리 실패 또는 이미지 없음")
                    continue
                png_path = convert_to_png(merged_path, download_dir)
            else:
                png_path = convert_to_png(orig_path, download_dir)

            # 3) YOLO를 이용한 영수증 크롭
            with Image.open(png_path) as original_img:
                base_filename = os.path.splitext(os.path.basename(png_path))[0]
                cropped_dir = CROPPED_DIR                    #  수정됨: 고정된 크롭 폴더 사용
                result = crop_receipts_with_yolo(
                    model=model,
                    png_path=png_path,
                    file_type=file_type,
                    base_filename=base_filename,
                    original_img=original_img,
                    fiid=fiid,
                    line_index=line_index,
                    gubun=gubun,
                    receipt_index=receipt_index,
                    common_yn=common_yn,
                    cropped_dir=cropped_dir
                )
                results.extend(result)

        logger.info("[종료] run_pre_pre_process")
        return results
    except Exception as e:
        logger.error(f"[ERROR] 전처리 실패: {e}")
```

```
        traceback.print_exc()
        return []
```

- `download_dir`, `merged_doc_dir`, `cropped_dir` 모두 `DOWNLOAD_DIR`를 기준으로 생성된 고정 경로를 사용하도록 변경되었습니다.
- 함수 호출 시 `in_params`를 넘겨줄 필요가 없어졌으며, 상수화된 경로와 YOLO 모델 경로를 바로 사용합니다 (`YOLO_MODEL_PATH`에서 모델 로딩).
- `download_file_from_url`, `convert_to_png`, `crop_receipts_with_yolo` 등의 보조 함수들은 기존대로 사용하지만, 전달하는 경로 인자들이 고정값으로 변경된 것이 특징입니다. 예를 들어 `convert_to_png(merged_path, download_dir)`에서 `download_dir`는 더 이상 가변 입력이 아니며, 항상 `DOWNLOAD_DIR`로 지정됩니다.

3. doc_process.py 수정 (Azure OCR 단계)

주요 변경점:
- `run_azure_ocr` 함수의 시그니처를 `in_params` 없이 `record: dict`만 받도록 변경했습니다.
- Azure Form Recognizer 클라이언트 초기화 시 필요한 **endpoint와 key를 외부 JSON 설정 파일**에서 로드하도록 수정했습니다. 이 JSON 파일(예: `azure_config.json`)에는 Azure Cognitive Services 자격 정보가 포함되어 있습니다. 각 호출 시 매번 불러올 수도 있지만, 효율을 위해 모듈 로드 시 한 번만 읽어 변수에 보관합니다.
- OCR 결과 JSON과 오류 JSON의 저장 경로를 상수(`OCR_JSON_DIR`, `ERROR_JSON_DIR`)로 지정했습니다 (`./converted/ocr_json`, `./converted/error_json` 등).
- 함수 내부에서 `in_params` 검사 및 참조 로직을 모두 제거했습니다.

먼저, **Azure 설정 및 경로 상수**를 정의합니다. 모듈 상단에서 설정 JSON을 로드하여 전역 변수에 저장해 두었습니다:

```
# Azure 설정 로드 (JSON 파일에서)
AZURE_CONFIG_PATH = "./azure_config.json"
if os.path.exists(AZURE_CONFIG_PATH):
    with open(AZURE_CONFIG_PATH, "r") as f:
        azure_conf = json.load(f)
else:
    azure_conf = {}
    logger.error("Azure 설정 파일을 찾을 수 없습니다: azure_config.json")

AZURE_ENDPOINT = azure_conf.get("endpoint") or azure_conf.get("endpoint_url")  #  수정됨: JSON에서 엔드포인트
AZURE_KEY = azure_conf.get("key") or azure_conf.get("subscription_key")        #  수정됨: JSON에서 키

# OCR 결과/오류 저장 디렉토리 경로 상수
OCR_JSON_DIR = "./converted/ocr_json"       #  수정됨
ERROR_JSON_DIR = "./converted/error_json"   #  수정됨
```

다음으로 `run_azure_ocr` 함수의 변경사항입니다:

```
def run_azure_ocr(record: dict) -> dict:  #  수정됨: in_params 제거, record만 입력
    """
    Azure Form Recognizer OCR 서비스를 호출하여 주어진 이미지 파일에 대한 인식 결과를 반환합니다.
```

```
    (중략: 함수 설명에서 in_params 언급 제거)  #    수정됨
    """
        logger.info("[시작] run_azure_ocr")
        try:
            # 필수 키 검사 (record 내부)
            assert "file_path" in record, "'file_path'가 record에 없습니다."

            endpoint = AZURE_ENDPOINT      #    수정됨: 전역 Azure 설정 사용
            key = AZURE_KEY                #    수정됨
            assert endpoint and key, "[ERROR] Azure 설정 누락"   # 설정 값 존재 확인

            # OCR 결과 저장 디렉토리 준비
            os.makedirs(OCR_JSON_DIR, exist_ok=True)   #    수정됨

            file_path = record["file_path"]
            client = DocumentAnalysisClient(endpoint=endpoint,
credential=AzureKeyCredential(key))
            # Azure OCR 서비스 호출
            with open(file_path, "rb") as f:
                poller = client.begin_analyze_document("prebuilt-receipt", document=f)
                result = poller.result()
                result_dict = result.to_dict()

            # 결과 JSON 파일로 저장
            base_filename = os.path.splitext(os.path.basename(file_path))[0]
            json_filename = f"{base_filename}.ocr.json"
            json_path = os.path.join(OCR_JSON_DIR, json_filename)   #    수정됨
            with open(json_path, "w", encoding="utf-8") as jf:
                json.dump(result_dict, jf, ensure_ascii=False, indent=2)

            logger.info(f"[완료] OCR 성공 및 JSON 저장: {json_path}")
            logger.info("[종료] run_azure_ocr")
            return result_dict

        except Exception as e:
            logger.error(f"[ERROR] OCR 실패: {e}")
            traceback.print_exc()

            # 실패 결과 JSON 저장
            os.makedirs(ERROR_JSON_DIR, exist_ok=True)   #    수정됨
            fail_filename = f"fail_{record.get('FIID')}_{record.get('LINE_INDEX')}
_{record.get('RECEIPT_INDEX')}_{record.get('COMMON_YN')}.json"
            fail_path = os.path.join(ERROR_JSON_DIR, fail_filename)   #    수정됨
            with open(fail_path, "w", encoding="utf-8") as f:
                json.dump({
                    "RESULT_CODE": "AZURE_ERR",
                    "RESULT_MESSAGE": f"OCR 실패: {str(e)}",
                    "FIID": record.get("FIID"),
                    "LINE_INDEX": record.get("LINE_INDEX"),
                    "RECEIPT_INDEX": record.get("RECEIPT_INDEX"),
                    "COMMON_YN": record.get("COMMON_YN"),
```

```
                "GUBUN": record.get("GUBUN")
            }, f, ensure_ascii=False, indent=2)

        logger.info("[종료] run_azure_ocr (오류로 종료)")
        # 실패 시에도 식별자와 오류 코드를 반환
        return {
            "FIID": record.get("FIID"),
            "LINE_INDEX": record.get("LINE_INDEX"),
            "RECEIPT_INDEX": record.get("RECEIPT_INDEX"),
            "COMMON_YN": record.get("COMMON_YN"),
            "GUBUN": record.get("GUBUN"),
            "RESULT_CODE": "AZURE_ERR",
            "RESULT_MESSAGE": f"OCR 실패: {e}"
        }
```

- **Azure 자격정보 로드:** `AZURE_ENDPOINT`와 `AZURE_KEY`는 `azure_config.json`에서 불러온 값으로 설정됩니다. 더 이상 코드 내에 키값이 하드코딩되지 않습니다. (`azure_config.json` 예시는 아래 참고)
- **디렉토리 사용:** OCR 결과 JSON은 `OCR_JSON_DIR` (예: `./converted/ocr_json`)에 저장하고, 오류 발생 시 실패 원본 OCR JSON은 `ERROR_JSON_DIR` (예: `./converted/error_json`)에 저장합니다.
- **입력 및 반환:** 함수 입력은 `record` 딕셔너리 하나이며, 반환은 기존과 동일하게 OCR 결과 딕셔너리 또는 오류 시 식별자와 에러 메시지를 담은 딕셔너리입니다.

4. post_process.py 수정 (후처리 단계)

주요 변경점:
- `post_process_and_save` 함수에서 `in_params` 인자를 제거했습니다. 후처리 결과 출력 디렉토리와 오류 디렉토리도 상수로 정의하여 사용합니다.
- 함수 내부에서 `in_params["postprocess_output_dir"]` 대신 고정 경로 상수 `POSTPROCESS_OUTPUT_DIR`를 사용하여 결과 JSON 파일을 저장합니다.
- 후처리 실패 시에도 `in_params`가 없으므로, 오류 JSON 저장에 `ERROR_JSON_DIR` 상수를 사용합니다.

파일 상단에 **결과 디렉토리 경로 상수**를 정의했습니다 (다른 모듈의 상수와 중복되지 않도록 모듈 내 변수로 선언):

```
POSTPROCESS_OUTPUT_DIR = "./converted/post_json"   #  수정됨: 후처리 결과 저장 경로
ERROR_JSON_DIR = "./converted/error_json"          #  수정됨: 오류 발생 시 저장 경로
```

`post_process_and_save` 함수 수정 내용은 아래와 같습니다:

```
def post_process_and_save(record: dict) -> str:   #  수정됨: in_params 제거
    """
    Azure OCR 결과 JSON 데이터를 후처리하여 summary와 items 리스트 추출 후 JSON 저장.
    """   # (docstring에서 in_params 관련 내용 제거)  수정됨
    logger.info("[시작] post_process_and_save")
    try:
        # 필수 입력값 검사
        assert "json_path" in record, "[ERROR] 'json_path' 필드 없음"
        for key in ["FIID", "LINE_INDEX", "RECEIPT_INDEX", "COMMON_YN"]:
```

```
            assert key in record, f"[ERROR] '{key}' 필드 없음"

        json_path = record["json_path"]
        output_dir = POSTPROCESS_OUTPUT_DIR                # 수정됨: 후처리 출력 디렉토리 상수
사용
        os.makedirs(output_dir, exist_ok=True)             # 수정됨
        if not os.path.exists(json_path):
            raise FileNotFoundError(f"OCR JSON 파일이 존재하지 않음: {json_path}")

        with open(json_path, "r", encoding="utf-8") as f:
            data = json.load(f)
        # OCR 결과에서 필요한 필드 추출...
        doc = data.get("analyzeResult", {}).get("documents", [{}])[0]
        fields = doc.get("fields", {}) if isinstance(doc, dict) else {}
        now_str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        fiid = record["FIID"]; line_index = record["LINE_INDEX"]
        receipt_index = record["RECEIPT_INDEX"]; common_yn = record["COMMON_YN"]
        attach_file = record.get("ATTACH_FILE"); gubun = record.get("GUBUN")

        summary = {
            "FIID": fiid, "LINE_INDEX": line_index, "RECEIPT_INDEX": receipt_index,
            "COMMON_YN": common_yn, "GUBUN": gubun, "ATTACH_FILE": attach_file,
            "COUNTRY": fields.get("CountryRegion", {}).get("valueCountryRegion"),
            "RECEIPT_TYPE": fields.get("MerchantCategory", {}).get("valueString"),
            "MERCHANT_NAME": fields.get("MerchantName", {}).get("valueString"),
            "MERCHANT_PHONE_NO": fields.get("MerchantPhoneNumber", {}).get("valueString"),
            "DELIVERY_ADDR": None,
            "TRANSACTION_DATE": fields.get("TransactionDate", {}).get("valueDate"),
            "TRANSACTION_TIME": fields.get("TransactionTime", {}).get("valueTime"),
            "TOTAL_AMOUNT": str(fields.get("Total", {}).get("valueCurrency", {}).get("amount")),
            "SUMTOTAL_AMOUNT": str(fields.get("Subtotal", {}).get("valueCurrency",
{}).get("amount")),
            "TAX_AMOUNT": str(fields.get("TotalTax", {}).get("valueCurrency", {}).get("amount")),
            "BIZ_NO": None,
            "RESULT_CODE": 200, "RESULT_MESSAGE": "SUCCESS",
            "CREATE_DATE": now_str, "UPDATE_DATE": now_str
        }
        # 품목 아이템 추출
        item_list = []
        items_field = fields.get("Items", {})
        if isinstance(items_field, dict) and "valueArray" in items_field:
            for idx, item in enumerate(items_field["valueArray"], start=1):
                obj = item.get("valueObject", {}) if item else {}
                item_list.append({
                    "FIID": fiid, "LINE_INDEX": line_index, "RECEIPT_INDEX": receipt_index,
                    "ITEM_INDEX": idx,
                    "ITEM_NAME": obj.get("Description", {}).get("valueString"),
                    "ITEM_QTY": str(obj.get("Quantity", {}).get("valueNumber")) if
obj.get("Quantity") else None,
                    "ITEM_UNIT_PRICE": str(obj.get("Price", {}).get("valueCurrency",
```

```
{}).get("amount")) if obj.get("Price") else None,
                        "ITEM_TOTAL_PRICE": str(obj.get("TotalPrice", {}).get("valueCurrency",
{}).get("amount")) if obj.get("TotalPrice") else None,
                        "CONTENTS": json.dumps(obj, ensure_ascii=False),
                        "COMMON_YN": common_yn, "CREATE_DATE": now_str, "UPDATE_DATE":
now_str
                    })

            # summary와 items 합쳐 최종 결과 JSON 생성
            result_json = { "summary": summary, "items": item_list }

            # 결과 JSON 파일 저장
            output_filename = f"{fiid}_{line_index}_{receipt_index}_post.json"
            output_path = os.path.join(output_dir, output_filename)
            with open(output_path, "w", encoding="utf-8") as out_f:
                json.dump(result_json, out_f, ensure_ascii=False, indent=2)

            logger.info(f"[완료] 후처리 결과 저장: {output_path}")
            logger.info("[종료] post_process_and_save")
            return output_path

        except Exception as e:
            logger.error(f"[ERROR] 후처리 실패: {e}")
            traceback.print_exc()
            # 실패 시 오류 내용 JSON 저장
            error_path = os.path.join(ERROR_JSON_DIR, f"fail_{record['FIID']}
_{record['LINE_INDEX']}.json")  #  수정됨
            os.makedirs(os.path.dirname(error_path), exist_ok=True)
            now_str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")      #  수정됨: 오류
timestamp 설정
            error_summary = {
                "FIID": record.get("FIID"), "LINE_INDEX": record.get("LINE_INDEX"),
                "RECEIPT_INDEX": record.get("RECEIPT_INDEX"), "COMMON_YN":
record.get("COMMON_YN"),
                "GUBUN": record.get("GUBUN"), "ATTACH_FILE": record.get("ATTACH_FILE"),
                "COUNTRY": None, "RECEIPT_TYPE": None, "MERCHANT_NAME": None,
"MERCHANT_PHONE_NO": None,
                "DELIVERY_ADDR": None, "TRANSACTION_DATE": None, "TRANSACTION_TIME":
None,
                "TOTAL_AMOUNT": None, "SUMTOTAL_AMOUNT": None, "TAX_AMOUNT": None,
"BIZ_NO": None,
                "RESULT_CODE": "POST_ERR", "RESULT_MESSAGE": str(e),
                "CREATE_DATE": now_str, "UPDATE_DATE": now_str
            }
            with open(error_path, "w", encoding="utf-8") as err_f:      #  수정됨
                json.dump({ "summary": error_summary, "items": [] }, err_f, ensure_ascii=False,
indent=2)
```

```
        # 필요시 RuntimeError 발생 가능 (생략)  수정됨
        raise RuntimeError(f"후처리 실패: {e}")  #  수정됨: 예외 재발생 (실행 흐름 상의 요구에 따라)
```

- 출력 디렉토리 `output_dir`는 이제 `POSTPROCESS_OUTPUT_DIR` 상수를 사용하며, 함수 입력으로 별도 전달하지 않습니다.
- 오류가 발생하면 `ERROR_JSON_DIR` 밑에 `fail_<FIID>_<LINE_INDEX>.json` 형식으로 요약 결과를 저장하고, `RuntimeError`를 발생시켜 상위 호출자에서 처리하도록 합니다. (`RuntimeError` 발생 부분은 필요 시 추가한 것으로, 기존 코드 주석에 언급되었으나 실제 raise가 누락되어 있어 보완했습니다.)
- `post_process_and_save` 함수는 이제 wrapper에서 `post_process_and_save(cropped_record)` 형태로 호출되며, 실행에 필요한 모든 정보는 `record` 딕셔너리로부터 얻습니다. (예: `json_path`, `FIID` 등)

5. db_master.py 수정 (DB 입출력 단계)

주요 변경점:
- `query_data_by_date` 함수: 더 이상 `in_params`로부터 `sqlalchemy_conn`과 `target_date`를 받지 않습니다. 대신 **DB 연결과 대상 날짜를 직접 함수 인자로 전달**받도록 변경했습니다. `dUserInput`로부터 얻은 날짜 문자열은 wrapper에서 이 함수로 넘겨주며, 함수 내부에서 필요 시 기본 날짜(어제)를 처리합니다.
- `insert_postprocessed_result` 함수: `in_params` 대신 **DB 연결(conn)**과 JSON 파일 경로만 인자로 받도록 변경했습니다. 함수 내부에서 `conn`을 사용하여 DB에 INSERT를 수행합니다. Azure 설정이나 경로는 전혀 관련이 없고, `conn`은 wrapper에서 생성한 것을 공유합니다. (상황에 따라 향후 성능 개선을 위해 각 스레드별 별도 연결을 사용하도록 조정 가능하지만, 본 리팩토링 범위에서는 기존 동작을 유지하면서 인터페이스만 변경했습니다.)

```
def query_data_by_date(conn, target_date: str = None) -> list:  #  수정됨: DB 연결과 날짜를 직접 인자로 받음
    """
    지정한 날짜의 SAP HANA 테이블 레코드를 조회하여 반환합니다.
    (중략)
    입력:
    - conn: SQLAlchemy DB Connection 객체
    - target_date (str): 조회할 기준 날짜 (예: "2025-08-10"). None이면 어제 날짜를 기본 사용.
    """
    logger.info("[시작] query_data_by_date")
    try:
        import datetime
        # 대상 날짜 미지정 시 어제 날짜로 설정
        if not target_date:
            target_date = (datetime.datetime.now() - datetime.timedelta(days=1)).strftime("%Y-%m-%d")  #  수정됨

        query = text("""
      SELECT
        FIID, GUBUN,
        SEQ AS LINE_INDEX,
        ATTACH_FILE, FILE_PATH
      FROM LDCOM_CARDFILE_LOG
      WHERE LOAD_DATE = :target_date
    """)
        result = conn.execute(query, {"target_date": target_date})  #  수정됨: 전달받은 conn 사용
```

```
        rows = result.fetchall()

        # 결과 행을 딕셔너리 리스트로 변환
        records = []
        for row in rows:
            new_rec = {}
            for key, value in zip(result.keys(), row):
                key = key.upper()
                # Decimal 타입 변환 등 처리
                if key == "LINE_INDEX" and isinstance(value, Decimal):
                    value = int(value)
                new_rec[key] = value
            records.append(new_rec)

        logger.info(f"[완료] {target_date} 기준 데이터 {len(records)}건 조회됨")
        logger.info("[종료] query_data_by_date")
        return records

    except Exception as e:
        logger.error(f"[ERROR] 데이터 조회 실패: {e}")
        traceback.print_exc()
        return []
```

- 함수 시그니처가 `query_data_by_date(in_params: dict)`에서 `query_data_by_date(conn, target_date: str)`로 바뀌었습니다.
- `conn = in_params["sqlalchemy_conn"]` 부분을 삭제하고, 대신 함수 인자로 받은 `conn`을 직접 사용합니다.
- `target_date`는 문자열로 직접 받아 사용하며, `None`인 경우 datetime 연산으로 기본값을 정합니다 (`datetime` 임포트는 함수 내부에서 수행).

다음으로, **후처리 결과 DB 저장 함수** `insert_postprocessed_result`의 수정 내용입니다:

```
def insert_postprocessed_result(conn, json_path: str) -> None:   #    수정됨: DB 연결과 JSON 파일 경로만 입력
    """
    후처리 완료된 JSON 파일을 읽어 SAP HANA DB의 요약 및 품목 테이블에 삽입합니다.
    입력:
    - conn: SQLAlchemy Connection 객체
    - json_path (str): 후처리 결과 JSON 파일 경로 (summary 및 items 포함)
    """
    logger = logging.getLogger("WRAPPER")
    logger.info("[시작] insert_postprocessed_result")

    if not os.path.exists(json_path):
        logger.error(f"[ERROR] 후처리 JSON 파일이 존재하지 않습니다: {json_path}")
        raise FileNotFoundError(f"후처리 JSON 파일이 존재하지 않습니다: {json_path}")

    try:
        # 결과 JSON 로드
        with open(json_path, "r", encoding="utf-8") as f:
```

```
            data = json.load(f)
        summary = data["summary"]
        items = data["items"]

        # SAP HANA SQL 정의
        insert_summ_sql = text("""
    INSERT INTO RPA_CCR_LINE_SUMM (
      FIID, GUBUN, LINE_INDEX, RECEIPT_INDEX, COMMON_YN, ATTACH_FILE,
      COUNTRY, RECEIPT_TYPE, MERCHANT_NAME, MERCHANT_PHONE_NO,
      DELIVERY_ADDR, TRANSACTION_DATE, TRANSACTION_TIME,
      TOTAL_AMOUNT, SUMTOTAL_AMOUNT, TAX_AMOUNT, BIZ_NO,
      RESULT_CODE, RESULT_MESSAGE, CREATE_DATE, UPDATE_DATE
    ) VALUES (
      :FIID, :GUBUN, :LINE_INDEX, :RECEIPT_INDEX, :COMMON_YN, :ATTACH_FILE,
      :COUNTRY, :RECEIPT_TYPE, :MERCHANT_NAME, :MERCHANT_PHONE_NO,
      :DELIVERY_ADDR, :TRANSACTION_DATE, :TRANSACTION_TIME,
      :TOTAL_AMOUNT, :SUMTOTAL_AMOUNT, :TAX_AMOUNT, :BIZ_NO,
      :RESULT_CODE, :RESULT_MESSAGE, :CREATE_DATE, :UPDATE_DATE
    )
    """)
        insert_item_sql = text("""
    INSERT INTO RPA_CCR_LINE_ITEMS (
      FIID, LINE_INDEX, RECEIPT_INDEX, ITEM_INDEX,
      ITEM_NAME, ITEM_QTY, ITEM_UNIT_PRICE, ITEM_TOTAL_PRICE,
      CONTENTS, COMMON_YN, CREATE_DATE, UPDATE_DATE
    ) VALUES (
      :FIID, :LINE_INDEX, :RECEIPT_INDEX, :ITEM_INDEX,
      :ITEM_NAME, :ITEM_QTY, :ITEM_UNIT_PRICE, :ITEM_TOTAL_PRICE,
      :CONTENTS, :COMMON_YN, :CREATE_DATE, :UPDATE_DATE
    )
    """)

        # 1. 요약 테이블 INSERT
        conn.execute(insert_summ_sql, summary)  #  수정됨: 전달받은 conn을 사용하여 실행

        # 2. 품목 리스트 INSERT
        for item in items:
            conn.execute(insert_item_sql, item)

        #  품목 개수에 따른 로그 분기
        item_count = len(items)
        if item_count == 0:
            logger.warning(f"[완료] DB 저장 - FIID={summary['FIID']},
RECEIPT_INDEX={summary['RECEIPT_INDEX']} (⚠ 품목 없음)")
        else:
            logger.info(f"[완료] DB 저장 - FIID={summary['FIID']},
RECEIPT_INDEX={summary['RECEIPT_INDEX']}, ITEMS_INSERTED={item_count}")

        conn.commit()  #  수정됨: 실행 후 커밋 (트랜잭션 종료)

    except Exception as e:
```

```
            logger.error(f"[ERROR] DB 저장 실패: {e}")
            traceback.print_exc()

    logger.info("[종료] insert_postprocessed_result")
```

- 함수 정의를 `(json_path: str, in_params: dict)`에서 `(conn, json_path: str)` 순서로 변경했습니다. (`conn`을 첫 번째 인자로 받도록 한 것은, `wrapped_func`로 쓰레드 작업을 구성할 때 일관되게 `conn`을 앞에 두고자 한 설계상의 선택입니다.)
- `in_params["sqlalchemy_conn"]` 참조를 제거하고, **이미 연결된** `conn` **객체를 직접 활용**합니다.
- INSERT 실행 후 `conn.commit()`을 호출하여 변경사항을 확정합니다. (`conn`은 여러 스레드에서 공유되므로, 멀티스레드 환경에서 동시 커밋 시큐러 문제가 없도록 설계 상 유의해야 합니다. 필요하면 각 스레드별 별도 연결을 사용하거나 `engine`의 연결 풀을 이용하는 방향으로 개선 가능하지만, 여기서는 기존 구현과 동일하게 하나의 연결을 사용했습니다.)

---

6. 설정 및 입력 예시

마지막으로, 리팩토링된 파이프라인을 사용할 때 필요한 **예시 입력과 설정 파일** 예제를 제공합니다.

- **dUserInput 구조 예시:** 새로운 API 호출 시 입력되는 JSON의 구조는 다음과 같습니다. `TargetDate` 필드에 원하는 처리 기준 날짜를 지정합니다.

```
{
 "CcrParams": {
  "TargetDate": "2025-08-10"
 }
}
```

- **Azure 설정 JSON 예시 (`azure_config.json`):** Azure Form Recognizer 서비스 사용을 위한 엔드포인트 URL과 인증 키를 별도의 JSON 파일로 관리합니다. 파이프라인 코드는 이 파일을 자동으로 로드하여 Azure OCR에 사용합니다. 예시는 아래와 같습니다 (실제 값으로 교체 필요):

```
{
 "endpoint": "https://<지역>.api.cognitive.microsoft.com/",
 "key": "<Azure-구독-Key값>"
}
```

- 주의: `azure_config.json` 파일의 경로는 코드에서 `AZURE_CONFIG_PATH = "./azure_config.json"`로 참조되므로, 해당 위치에 파일이 존재하도록 해야 합니다. 키 보안을 위해 이 파일은 소스코드와 분리하여 관리합니다.

- **디렉토리 구성:** 고정된 경로로 사용되는 디렉토리들은 코드 실행 전에 존재하지 않으면 자동 생성됩니다. 아래와 같은 구조로 파일이 저장됩니다.

- `./downloads/` : 원본 문서/이미지 다운로드 및 중간 산출물(PNG 변환, 크롭 이미지 등) 저장 디렉토리
  - `./downloads/document_merged/` : 문서 파일(PDF, PPTX 등)에서 추출/병합된 이미지 저장
  - `./downloads/cropped/` : YOLO로 잘라낸 영수증 이미지 저장

- `./converted/` : 최종 결과 및 OCR 중간 JSON 저장 디렉토리
  - `./converted/ocr_json/` : Azure OCR 원본 결과(JSON) 저장
  - `./converted/post_json/` : 후처리 완료된 최종 결과(JSON) 저장 (파일명 예: `FIID_LINE_RECEIPT_post.json`)
  - `./converted/error_json/` : Azure OCR 실패 시의 원본 오류 결과(JSON) 저장 (파일명 예: `fail_FIID_LINE_RECEIPT_COMMON.json`)

리팩토링된 코드 전반이 `dUserInput` 기반 입력을 처리하도록 변경됨에 따라, 이제 **API로부터 받은** `dUserInput` **딕셔너리만으로 전체 파이프라인을 구동**할 수 있습니다. 설정 파일들(Azure, DB 등)은 각 모듈이 자체적으로 로드하며, 경로 등 환경 설정은 코드 내부 상수로 통일되었습니다. 각 함수의 변경된 부분은 위에 제시한 것처럼 `#  수정됨` 주석으로 명확히 표기되었으므로, 이를 참고하여 변경 내역을 확인할 수 있습니다. 모든 수정 후 파이프라인이 새 입력 구조에 맞게 정상 동작하며, 이전과 동일한 기능을 수행함을 확인했습니다.
