

## Chapter 5 : 태스크 간 의존성 정의하기

### #복잡한 패턴 구현

- 조건부 태스크, 분기, 조인 등등...

### #테스트 의존성 패턴 유형

(1) **선형체인**(Linear chain) : 연속적으로 실행되는 작업

- 이전 태스크 결과 -> 다음 태스크 "입력 값" 으로 사용 (선행작업 완료 필수)
- 오른쪽 비트 시프트 연산자(>>) 사용해 태스크 간 의존성 만듦
- 태스크 의존성 명시적 설정의 장점 : 순서가 직관적으로 명확히 정의됨

(2-1) **팬아웃**(fan-out) : 한 태스크 - 여러 개 다운스트림 태스크 연결

(2-2) **팬인**(fan-in) : 여러 업스트림 태스크-한 태스크가 연결

(ex) --> 실행방향



- start : dummy 태스크
- 위 DAG작업은 (1)판매/날씨 데이터를 병렬로 수집 후, (2)정제하고, (3)데이터 세트로 결합 후, 이를 활용해 (4)머신러닝 모델 학습 후 (5)배포하는 작업 스케줄
- 판매/날씨 수집(fetch\_sales, fetch\_weather)의 경우 독립적으로 실행됨
- 위 작업들은 선형적으로 진행됨

### #브랜치하기

#### [1] 태스크 내 브랜치

- 기존 시스템 데이터가 아닌, 새로운 시스템(or 새로 가공된 데이터) 필요한 경우  
기존 소스를 두 개의 **개별 코드로 분리** 가능
- DAG 자체 구조 수정하지 않고, DAG에 약간의 유연성 허용한 것  
➔ (한계) 유사한 태스크로 구성된 작업인 경우만 작동  
\*완전 다른 태스크 체인 필요한 경우 -> 2개 개별 태스크 세트로 분할 권면  
(즉! 공통점 多 -> 단일 태스크 분개 / 아닐경우 -> 개별 태스크로 분할)

➔ Airflow 뷰(웹상) 통해 태스크 내 브랜치 확인 (x) -> 코드 검사 or 로그 검사 필요

(ex) (1) clean\_~ : 정제 태스크 안 브랜치

```
def _clean_sales(**context):
    if context[ "execution_date" ] < ERP_CHANGE_DATE:
        _clean_sales_old(**context)
    else
        _clean_sales_new(**context)
    ~~~

clean_sales_data=PythonOperator(
    task_id="clean_sales",
    python_callable=_clean_sales,
)
```

(ex) (2) fetch\_~ : 수집 태스크 안 브랜치

```
def _fetch_sales(**context):
    if context[ "execution_date" ] < ERP_CHANGE_DATE:
        _fetch_sales_old(**context)
    else
        _fetch_sales_new(**context)
    ~~~
```

## [2] DAG 내 브랜치

- 각 시스템(작업) 별 태스크 개발 후, **DAG가 수집작업 실행 선택** 하도록 하는 것

➔ 개별 태스크 : 적절한 Operator 활용해 태스크 생성

(ex) fetch\_sales\_old=PythonOperator(~~~~)

clean\_sales\_old=PythonOperator(~~~~)

fetch\_sales\_new=PythonOperator(~~~~)

clean\_sales\_new=PythonOperator(~~~~)

fetch\_sales\_old >> clean\_sales\_old // 의존성 추가

fetch\_sales\_new >> clean\_sales\_new

- **BranchPythonOperator** 인수 사용 : 파이썬 콜러블 인수 中 하나

➔ 위 인수 활용해, "다운스트림 태스크 세트" 중 선택

(ex) def \_pick\_erp\_system(\*\*context):

```
    if context[ "execution_date" ] < ERP_CHANGE_DATE:
        return "fetch_sales_old"
    else
        return "fetch_sales_new"
```

```
pick_erp_system=BranchPythonOperator(

    task_id="pick_erp_system",

    python_callable=_pick_erp_system,

)
```

```
pick_erp_system >> [fetch_sales_old, fetch_sales_new]
```

- **BranchPythonOperator**에 전달된 "콜러블 인수"는 작업 결과 "다운스트림 태스크"의 ID 반환
  - ➔ "반환된ID"는 브랜치 태스크 완료 후, 실행할 "다운스트림 태스크" 결정
  - ➔ "반환된ID 리스트"를 반환한 경우, Airflow는 참조된 모든 태스크 실행

(ex) 후행 작업 연결

```
start_task >> pick_erp_system // start태스크에 브랜치 연결
```

```
[clean_sales_old, clean_sales_new] >> join_datasets // 브랜치를 join_datasets태스크에
연결
```

- 위 같이 연결 시, 병렬로 정제 태스크 수행 중 하나만 작업 완료할 경우, join\_dates 태스크 실행 (x)

## #트리거

- 트리거 규칙 : 태스크 실행 시기는 "트리거 규칙"에 의해 제어됨
  - (이슈사항 : 트리거 규칙 없을시, 작업 건너 뛴 경우도 발생가능)
- ➔ **trigger\_rule** 인수 이용해 개별 태스크에 대해 "트리거 규칙" 정의함
- ➔ **all\_success** 설정 : 모든 상위(parents) 태스크 성공해야 해당 태스크 실행 가능
  - (트리거 규칙 없을시, 작업 건너 뛴 경우도 발생가능)
- ➔ **none\_failed** 설정 : "업스트림\_태스크" 中 하나 건너뛰더라도, 계속 트리거 진행
  - 즉, 상위항목이 실행 완료(실패가 없을 시) 즉시작업 실행

(특징)

- (1) **성공 or 일부 스킵** : 이전 태스크들이 모두 성공 or 일부가 스킵 되었더라도 실패만 하지 않았다면, 현재 태스크 실행
- (2) **실패한 태스크가 없는 것만 확인** : 이전 태스크들 중 하나라도 실패한 경우, 현재 태스크는 실행되지 않음

**all\_success**와는 달리, 모든 부모 태스크가 성공할 필요는 없음.

단지 실패하지 않으면 됨

(ex) join\_datasets=**PythonOperator**(

~~,

**trigger\_rule="none\_failed",**

)

➤ “**더미조인 태스크**” 추가하는 경우

- DAG에 서로 다른 브랜치 결합하는 경우, “더미 태스크” 추가해 브랜치 조건 명확히 함
  - ➔ 이를 통해, join\_datasets 태스크에 대한 트리거 규칙 변경 필요 없어짐
  - ➔ 브랜치를 좀 더 독립적으로 유지

(ex) from airflow.operators.dummy import DummyOperator

join\_branch=**DummyOperator**(

task\_id="join\_erp\_branch",

trigger\_rule="none\_failed"

)

[clean\_sales\_old, clean\_sales\_new] >> join\_branch

join\_branch >> join\_datasets

## #조건부 태스크

- 특정 태스크 건너 뛴 방법 제공

(ex) 데이터 변경사항이 있어, 특정 버전만 배포 원할 경우

-> 최근 실행한 DAG에 대해서만 모델 배포하도록 DAG변경

### [1] 태스크 내 조건구현

- PythonOperator 사용해 배포 구현 + 배포 함수 내 DAG 실행날짜 명시적 확인

(ex) def \_deploy(\*\*context):

```
    if context[ "execution_date" ] == ....:
```

```
        deploy_model()
```

```
deploy=PythonOperator(
```

```
    task_id="deploy_model",
```

```
    python_callable=_deploy,
```

```
)
```

- 위 처럼 배포 태스크 내 구현하면, 동작은 되지만 로직 혼용과 PythonOperator 외 다른 기본제공 Operator 사용 (x) + Airflow UI 상 태스크 결과 추적에 혼란

### [2] 조건부 태스크 만들기

- 배포 태스크 자체를 조건부 화
  - ➔ 조건 실패할 경우, 모든 다운스트림 태스크 작업 건너뛰는 태스크를 DAG에 추가
  - ➔ 필요의 경우, [AirflowSkipException](#) 함수 실행 (참고 : 100~101p)

(ex) def \_latest\_only(\*\*context):

```
    로직
```

```
    latest_only=PythonOperator(
```

```
        task_id="latest_only",
```

```
        python_callable=_latest_only
```

```
    )
```

```
    latest_only >> deploy_model
```

### [3] 내장 Operator 사용하기

- [LatestOnlyOperator](#) 클래스 사용 : 가장 최근 실행한 DAG만 실행
  - ➔ 이 Operator는 PythonOperator 기반 동일한 작업을 가능하게 함
  - ➔ 조건부 배포를 구현하는 데, 복잡한 로직 필요(x)

(ex) from airflow.operators.latest\_only import LatestOnlyOperator

```
latest_only=LatestOnlyOperator(
    task_id="Latest_Only",
    dag=dag
)

Join_datasets >> train_model >> deploy_model

latest_only >> deploy_model
```

#### [참고] 실패의 영향

- **전파** : trigger\_rule="all\_success" (default 설정) 시, 업스트림 태스크 결과  
-> 다운스트림 태스크에도 영향

#### [참고] Airflow에서 지원하는 다양한 트리거 규칙 (참고 : 105p)

- **all\_done** : 결과 상관없이, 의존성 완료되는 즉시 실행되는 태스크 정의 가능
- 트리거 규칙 활용으로, DAG에 더 복잡한 동작 가능

#### #태스크 간 데이터 공유(1) : XCom 사용

- **XCom**(cross-communication) 사용 : 태스크 간 작은 데이터 공유 가능
  - ➔ 배포 시, 배포 모델의 버전 식별자 -> deploy\_model 태스크에 전달해야 함
  - ➔ XCom 이용해 train\_model 및 deploy\_model 간 식별자 공유

#### [1] xcom\_push 메서드

: train\_model 태스크는 다른 태스크가 XCom값 사용할 수 있도록 XCom 모델 식별자 값 보냄  
(xcom\_push 메서드 사용 -> 값 게시)

(ex) xcom\_push 사용해 명시적으로 XCom 값 게시

```
def _train_model(**context):

    model_id=str(uuid.uuid4())

    context[ "task_instance" ].xcom_push(key="model_id", value=model_id)

train_model=PythonOperator(
```

```

task_id="train_model",

python_callable=_train_model

)

```

- **xcom\_push**에 대한 호출 : 해당 태스크(train\_model)와 해당 DAG 및 실행날짜에 대한 XCom 값으로 "model\_id 값"을 등록 가능
- Airflow UI 에서, **Admin > XCom** 항목에서 XCom 값 확인 가능

## [2] xcom\_pull 메서드

: 다른 태스크에서 XCom 값 확인

```
def _deploy_model(**context):
```

```

    model_id=context[ "task_instance" ].xcom_pull(

        task_id="train_model", key="model_id"

    )

```

```
    print(f"Deploying model {model_id}")
```

```

deploy_model=PythonOperator(

    task_id="deploy_model",

    python_callable=_deploy_model

)

```

- Airflow가 train\_model 태스크의 model\_id와 일치하는 XCom 값 가져오도록 지시
- xcom\_pull 통해 XCom 값 가져올 때, dag\_id 및 실행날짜 정의 가능 (XCom의 default 값 : DAG와 실행날짜로 설정됨)

## [활용] XCom 값 활용 로직 (예시)

```
(ex) def _deploy_model(templates_dict, **context):
```

```

    model_id=templates_dict[ "model_id" ]

    print(f "Deploying model {model_id}")

```

```
deploy_model=PythonOperator(
```

```

task_id="deploy_model",

python_callable=_deploy_model,

templates_dict={

    "model_id": "{{ task_instance.xcom_pull(

        task_ids='train_model', key='model_id')}}"

},

)

```

#### [활용] return 사용해 XCom 값 게시

(ex) def \_train\_model(\*\*context):

```

    model_id=str(uuid.uuid4())

    return model_id

```

- return\_value 기준으로 XCom 등록됨 (Airflow UI 상)

#### #XCom 사용시 고려사항 (참고 : 109p)

- (1) 숨겨진 의존성 주의 : XCom 값 공유로 인해 복잡성 증가 -> 권장(x)
- (2) Operator의 원자성 무너뜨리는 패턴이 될 가능성 있음
- (3) 기술적 한계 : 모든 값 직렬화를 지원해야 함

(참고 : XCom은 Airflow의 "메타스토어"에 저장되며, 크기 제한 있음 -> 큰 데이터 세트 저장시 사용 안함)

- XCom 단점 : 태스크 간 의존성 확인 못함

#### #커스텀 XCom 백엔드 사용 (참고 : 110p)

- 좀 더 유연한 XCom 사용위해, 백엔드 지정 옵션 추가 가능

#### #태스크 간 데이터 공유(2) : Taskflow API 사용

- Taskflow API를 통해 태스크 의존성 정의 가능
- Airflow 2는 (태스크 의존성 정의를 위한) Taskflow API 제공



- PythonOperator / XCom 사용 보다 **코드 단순**

(ex) Taskflow API 사용해 train\_model / deploy\_model 정의

```
from airflow.decorators import task
```

로직

```
with DAG(~~) as dag:
```

로직

**@task**

```
def train_model():
```

```
    model_id=str(uuid.uuid4())
```

```
    return model_id                // 단순히 ID return
```

**@task**

```
def deploy_model(model_id: str):
```

```
    print(f "Deploying model {model_id}")
```

```
// Taskflow 태스크 간 의존성 설정
```

```
model_id=train_model()
```

```
deploy_model(model_id)
```

- 두 태스크(train\_model, deploy\_model) 간 의존성 포함된 DAG 생성됨
- 장점 : 태스크 간 의존성 명시적인 값(ex: model\_id)으로 전달 -> 의존성 확인 명확

(DAG 단순화 good)

- 단점 : PythonOperator로 구현된 "파이썬 태스크"로 사용 제한
  - ➔ 다른 Operator관련 태스크의 경우, 일반 API 사용해 태스크 의존성 정의해야 함

(참고 : 두가지 스타일 혼용해 사용해도 되지만, 코드 복잡성 증가함) (참고 : 114p)