
- 출처: LangChain 공식 문서 또는 해당 교재명
- 원본 URL: https://smith.langchain.com/hub/teddynote/summary-stuff-documents

---

`CH13.` `LangChain Expression Language` *(LCEL)*

- `LangChain Expression Language` *(LCEL)*
  - `LangChain` 라이브러리에서 제공하는 선언적 방식의 인터페이스
  - 복잡한 `LLM` (`Large Language Model`) 애플리케이션을 구축하고 실행하기 위한 도구
  - `LLM`, `프롬프트`, `검색기`, `메모리` 등 `다양한 컴포넌트`를 `조합`하여 강력하고 유연한 `AI 시스템`을 만들 수 있게 해줌

- `주요 특징`
  - `선언적 구문`: 복잡한 로직 간결 → 읽기 쉬운 방식으로 표현 가능
  - `모듈성`: 다양한 컴포넌트 → 쉽게 조합, 재사용 가능
  - `유연성`: 다양한 유형의 LLM 애플리케이션 구축 가능
  - `확장성`: 사용자 정의 컴포넌트 쉽게 통합 가능
  - `최적화`: 실행 시 자동으로 최적화 수행

- `기본 구성 요소`
  - `Runnable`: 모든 `LCEL` 컴포넌트의 기본 클래스
  - `Chain`: 여러 `Runnable`을 순차적으로 실행
  - `RunnableMap`: 여러 `Runnable`을 병렬로 실행
  - `RunnableSequence`: `Runnable`의 시퀀스 정의
  - `RunnableLambda`: 사용자 정의 함수 = `Runnable`로 래핑

- `사용 예시`
  - `LCEL`을 사용한 간단한 예시
    - *프롬프트, 모델, 출력 파서를 파이프라인으로 연결하여 간단한 체인으로 만들고 있음*

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain.schema.output_parser import StrOutputParser

prompt = ChatPromptTemplate.from_template("다음의 주제에 대해서 설명해줘: {topic}")
model = ChatOpenAI()
output_parser = StrOutputParser()

chain = prompt | model | output_parser

result = chain.invoke({"topic": "LangChain"})
print(result)
```

- `고급 기능`
  - `병렬 처리`: `Runnable` → 여러 작업을 동시에 실행 가능

- **조건부 실행** : `RunnableBranch` → 조건에 따라 다른 경로로 실행 가능
- **재시도 및 폴백** : 실패 시 자동으로 재시도 or 대체 경로로 실행 가능
- **스트리밍** : 대규모 데이터를 효율적으로 처리 가능

- **장단점**
  - **장점**
    - **코드 가독성** : 복잡한 로직 → 명확, 간결하게 표현 가능
    - **유지보수성** : 모듈화된 구조 → 유지보수 용이
    - **성능** : 자동 최적화 → 효율적 실행 가능
    - **확장성** : 새로운 컴포넌트 → 쉽게 추가 및 통합 가능

*

- **단점**
  - **학습 곡선** : 새로운 패러다임에 익숙해지는 데 시간이 필요할 수 있음
  - **디버깅** : 복잡한 체인의 경우 디버깅이 어려울 수 있음
  - **성능 오버헤드** : 매우 간단한 작업의 경우 오버헤드가 발생할 수 있음

- **활용 사례**
  - `LCEL` 은 다음과 같은 다양한 `LLM` 애플리케이션 구축에 활용 가능
    - 대화형 AI 시스템
    - 문서 요약 및 분석 도구
    - 질의응답 시스템
    - 데이터 추출 및 변환 파이프라인
    - 다국어 번역 서비스

*

- `LLM` 애플리케이션 개발을 위한 강력하고 유연한 도구
  - 선언적 구문 + 모듈성 → 복잡한 AI 시스템을 효율적으로 구축 → LangChain 생태계의 핵심 요소
  - 지속적 발전 → `LCEL` - AI 개발자들에게 더욱 중요한 도구가 될 것

---

1. `RunnablePassthrough`

1) `RunnablePassthrough`

- `RunnalbePassthrough`
  - 데이터 전달하는 역할
  - **`invoke()` 메서드 → 입력 데이터를 그대로 반환**

- 유용하게 활용되는 경우
  - 데이터를 변환 or 수정할 필요가 없는 경우
  - 파이프라인의 특정 단계를 뛰어건너야 하는 경우
  - 디버깅 or 테스트 목적으로 데이터 흐름을 모니터링 해야 하는 경우

- **`Runnable` 인터페이스 구현 → 다른 `Runnable` 객체와 함께 파이프라인에서 사용 가능**

- **환경설정**

```
# API 키를 환경변수로 관리하기 위한 설정 파일
from dotenv import load_dotenv

# API 키 정보 로드
load_dotenv()                           # True
```

```
from langsmith import Client
from langsmith import traceable

import os

# LangSmith 환경 변수 확인

print("\n--- LangSmith 환경 변수 확인 ---")
langchain_tracing_v2 = os.getenv('LANGCHAIN_TRACING_V2')
langchain_project = os.getenv('LANGCHAIN_PROJECT')
langchain_api_key_status = "설정됨" if os.getenv('LANGCHAIN_API_KEY') else "설정되지 않음" # API 키 값은 직접 출력하지 않음

if langchain_tracing_v2 == "true" and os.getenv('LANGCHAIN_API_KEY') and langchain_project:
    print(f"✅ LangSmith 추적 활성화됨 (LANGCHAIN_TRACING_V2='{langchain_tracing_v2}')")
    print(f"✅ LangSmith 프로젝트: '{langchain_project}'")
    print(f"✅ LangSmith API Key: {langchain_api_key_status}")
    print("  -> 이제 LangSmith 대시보드에서 이 프로젝트를 확인해 보세요.")
else:
    print("❌ LangSmith 추적이 완전히 활성화되지 않았습니다. 다음을 확인하세요:")
    if langchain_tracing_v2 != "true":
        print(f"  - LANGCHAIN_TRACING_V2가 'true'로 설정되어 있지 않습니다 (현재: '{langchain_tracing_v2}').")
    if not os.getenv('LANGCHAIN_API_KEY'):
        print("  - LANGCHAIN_API_KEY가 설정되어 있지 않습니다.")
    if not langchain_project:
        print("  - LANGCHAIN_PROJECT가 설정되어 있지 않습니다.")
```

- 셀 출력

```
--- LangSmith 환경 변수 확인 ---
✅ LangSmith 추적 활성화됨 (LANGCHAIN_TRACING_V2='true')
✅ LangSmith 프로젝트: 'LangChain-prantice'
✅ LangSmith API Key: 설정됨
-> 이제 LangSmith 대시보드에서 이 프로젝트를 확인해 보세요.
```

---

2) 데이터 전달하기

- **`RunnablePassthrough`** = **입력을 변경하지 않고 그대로 전달** = **추가 키를 더해서 전달 가능**
  - 일반적으로 **`RunnableParallel`** 과 함께 사용 → 데이터를 맵의 새로운 키에 할당하는 데 활용
  - **`RunnablePassthrough()`** 단독으로 호출 → **단순히 입력을 그대로 반환**

- **`RunnablePassthrough.assign(...)`** = **assign 함수에 전달된 추가 인자 더하기**
  - **`Runnable`** 클래스 사용 → **병렬로 실행 가능한 작업 정의**
  - **`passed`** 속성: `RunnablePassthrough` 인스턴스 할당 → 입력을 그대로 반환하도록 설정
  - **`extra`** 속성: `RunnablePassthrough.assign(...)` 메서드 사용 → 입력의 `num` 값에 `3`을 곱한 결과를 `mult`키에 할당하는 작업 정의
  - **`modified`** 속성: 람다 함수 사용 → 입력의 `num` 값에 `1`을 더한 결과를 더하는 작업 정의
  - **`runnable.invoke()`** 메서드 호출 → **`{ "num" : 1 }`** 입력 → **병렬 작업 실행**

```
from langchain_core.runnables import RunnableParallel, RunnablePassthrough

runnable = RunnableParallel(
    passed=RunnablePassthrough(),                                   # 전달된 입력을 그대로 반환하는 Runnable 설정
    extra=RunnablePassthrough.assign(mult=lambda x: x["num"] * 3),  # 입력의 "num" 값에 3을 곱한 결과를 반환하는 Runnable 설정
    modified=lambda x: x["num"] + 1,                                # 입력의 "num" 값에 1을 더한 결과를 반환하는 Runnable 설정
)

# {"num": 1}을 입력으로 Runnable 실행하기
runnable.invoke({"num": 1})
```

- 셀 출력 (`0.3s`)

```
{'passed': {'num': 1}, 'extra': {'num': 1, 'mult': 3}, 'modified': 2}
```

- `passed` = **`RunnablePassthrough()`** 와 함께 호출 → **단순히 `{ 'num' : 1 }` 을 전달함**
- `extra` = **`RunnablePassthrough.assign(mult=lambda x: x["num"] * 3)`** 와 함께 호출 → **단순히 `{ 'num' : 1 }` 을 전달함**
- `combined` = **`RunnablePassthrough.assign(mult=lambda x: x["num"] * 3)`** 와 함께 호출 → **단순히 `{ 'num' : 1 }` 을 전달함**

```
r = RunnablePassthrough.assign(mult=lambda x: x["num"] * 3)

r.invoke({"num": 1})
```

- 셀 출력

```
{'num': 1, 'mult': 3}
```

---

3) `검색기 예제`

- **`RunnablePassthrough` 사용하는 예제**

```
from langchain_community.vectorstores import FAISS
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_huggingface import HuggingFaceEmbeddings

from dotenv import load_dotenv
import os
import warnings                                    # 5.4s
```

```
# 임베딩(Embedding) 생성

from langchain_huggingface import HuggingFaceEmbeddings
import warnings                                                    # 경고 무시

warnings.filterwarnings("ignore")                                  # HuggingFace Embeddings 사용

embeddings = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2",
    model_kwargs={'device': 'cpu'},    \
    encode_kwargs={'normalize_embeddings': True})

print("✅ hugging-face 임베딩 모델 로딩 완료!")
```

- ✅ hugging-face 임베딩 모델 로딩 완료! (`8.8s`)

```
# API 키 확인
if not os.getenv("GOOGLE_API_KEY"):
    os.environ["GOOGLE_API_KEY"] = input("Enter your Google API key: ")

# LLM 초기화
gemini_lc = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash-lite",
    temperature=0,                                          # temperature = 0으로 설정
    max_output_tokens=4096,
)
```

- `gemini-2.5-flash-lite`로 `LLM` 생성하기

```
E0000 00:00:1759825336.333701 2593718 alts_credentials.cc:93] ALTS creds ignored. Not running on GCP and untruste
```

```
# 텍스트로부터 FAISS 벡터 저장소를 생성함

vectorstore = FAISS.from_texts(
    [
        "A는 랭체인 주식회사에서 근무를 하였습니다.",
        "B는 A와 같은 회사에서 근무하였습니다.",
        "A의 직업은 개발자입니다.",
        "B의 직업은 디자이너입니다.",
    ],
    embedding=embeddings
)
```

```
# 벡터 저장소를 검색기로 사용하기

retriever = vectorstore.as_retriever()
```

```
# 템플릿 정의하기

template = """Answer the question based only on the following context:
{context}

Question: {question}
"""
```

```
# 템플릿으로부터 채팅 프롬프트 생성하기

prompt = ChatPromptTemplate.from_template(template)
```

```
print(type(prompt))                # <class 'langchain.prompts.chat.ChatPromptTemplate'>
```

```
# LLM 모델 초기화

gemini_lc = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash-lite",
    temperature=0,                                          # temperature = 0으로 설정
    max_output_tokens=4096,
)
```

```
# 문서를 포맷팅하는 함수
def format_docs(docs):
    return "\n".join([doc.page_content for doc in docs])
```

```
# 검색 체인 구성

retrieval_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | gemini_lc
    | StrOutputParser()
)
```

```
# 검색 체인을 실행하여 질문에 대한 답변을 얻기

retrieval_chain.invoke("A의 직업은 무엇입니까?")
```

- `query_1` ("A의 직업은 무엇입니까?") - (`1.2s`)

```
'개발자'
```

```
# 검색 체인을 실행하여 질문에 대한 답변을 얻기

retrieval_chain.invoke("B의 직업은 무엇입니까?")
```

- `query_2` ("B의 직업은 무엇입니까?") - (`0.7s`)

```
'B의 직업은 디자이너입니다.'
```

```
# 검색 체인을 실행하여 질문에 대한 답변을 얻기

retrieval_chain.invoke("A의 직장은 어디입니까?")
```

- `query_3` ("A의 직장은 어디입니까?") - (`0.7s`)

```
'랭체인 주식회사'
```

```
# 검색 체인을 실행하여 질문에 대한 답변을 얻기

retrieval_chain.invoke("B의 직장은 어디입니까?")
```

- `query_4` ("B의 직장은 어디입니까?") - (`0.8s`)

```
'랭체인 주식회사'
```

- next: ***`02. Runnable 구조(그래프) 검토`***
