

랭체인 기본



1.1 LangChain Expression Language (LCEL)

1.1 LangChain Expression Language (LCEL)



🌸 LangChain Expression Language (LCEL)란?

LCEL은 LangChain에서 체인(**Chain**)과 **Runnable**을 조합해서 파이프라인을 만드는 표준 문법이에요.

쉽게 말해, LangChain의 여러 구성 요소(PromptTemplate, Model, Parser 등)를 ****|** (파이프 연산자)******로 연결해서 하나의 흐름으로 실행할 수 있게 만든 표현식 언어입니다.

1.1 LangChain Expression Language (LCEL)

🔑 핵심 특징

1 파이프라인 스타일 구성

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser

prompt = ChatPromptTemplate.from_template("Translate this to Korean: {text}")
model = ChatOpenAI(model="gpt-4o-mini")
parser = StrOutputParser()

chain = prompt | model | parser
chain.invoke({"text": "Hello, how are you?"})
```

- `|` (파이프 연산자)로 연결하면 입력 → 프롬프트 → 모델 → 파서가 순서대로 실행
- 함수형 프로그래밍 스타일이라 직관적이고 테스트·조합이 쉬움

1.1 LangChain Expression Language (LCEL)

2 모든 구성 요소가 Runnable

- PromptTemplate, Model, Retriever, Parser 등은 모두 Runnable 인터페이스 구현
- .invoke(), .batch(), .stream() 같은 공통 메서드로 실행 가능 → 동기/비동기/스트리밍 쉽게 전환

3 체인 컴포저빌리티(조합 가능성)

- 여러 개의 체인을 묶어 더 큰 체인으로 만들 수 있음
- 입력·출력을 map/dict 형태로 변환하여 병렬 실행도 가능

```
from langchain_core.runnables import RunnableParallel

chain1 = prompt1 | model | parser
chain2 = prompt2 | model | parser

multi_chain = RunnableParallel(first=chain1, second=chain2)
```

1.1 LangChain Expression Language (LCEL)

프로덕션 친화적

- 일관된 인터페이스 덕분에 테스트·캐싱·로깅·모니터링이 쉬움
- 최근 LangChain은 기존 LLMChain 대신 LCEL 방식 사용을 적극 권장

장점 정리

장점	설명
간결한 코드	여러 단계를 한 줄로 연결해 직관적
재사용성	개별 Runnable 블록 재사용 가능
유연한 실행	<code>.invoke()</code> , <code>.stream()</code> , <code>.batch()</code> 모두 지원
조합성	파이프라인, 병렬 실행, 분기처리까지 쉽게 구성
프로덕션 준비	로깅/관측성/캐싱/병렬 처리 내장

1.1 LangChain Expression Language (LCEL)

요약

LCEL은 LangChain의 새로운 표준 방식으로,

"프롬프트 → 모델 → 파서" 같은 데이터 흐름을 간단하게 표현하고 실행하는 문법

이 덕분에 코드를 깔끔하게 유지하면서도
테스트, 확장, 프로덕션 배포가 쉬워집니다.

1.2 LCEL 인터페이스

(Runnable 프로토콜)

1.2 LCEL 인터페이스

사용자 정의 체인을 가능한 쉽게 만들 수 있도록, Runnable 프로토콜을 제공한다. Runnable 프로토콜은 대부분의 컴포넌트에 구현되어 있다. 이는 표준 인터페이스로, 사용자 정의 체인을 정의하고 표준 방식으로 호출하는 것을 쉽게 만듭니다. 표준 인터페이스에는 다음이 포함됩니다.

- [stream](#): 응답의 청크를 스트리밍합니다.
- [invoke](#): 입력에 대해 체인을 호출합니다.
- [batch](#): 입력 목록에 대해 체인을 호출합니다.

1.2 LCEL 인터페이스

- 비동기 메소드

[astream](#) : 비동기적으로 응답의 청크를 스트리밍합니다.

[ainvoke](#) : 비동기적으로 입력에 대해 체인을 호출합니다.

[abatch](#) : 비동기적으로 입력 목록에 대해 체인을 호출합니다.

[astream_log](#) : 최종 응답 뿐만 아니라 발생하는 중간 단계를 스트리밍합니다.

1.2 LCEL 인터페이스

stream: 실시간 출력

`chain.stream` 메서드는 실시간 토큰 스트리밍을 지원합니다.

모델이 응답을 생성할 때 토큰이 만들어지는 즉시 전달되고,
이를 반복(iterate)하며 출력하면 타이핑하듯 한 글자씩 결과가 나타납니다.

- `end=""` → 출력 시 줄바꿈 없이 이어서 표시
- `flush=True` → 출력 버퍼를 바로 비워 즉시 화면에 반영

예시 코드

```
# chain.stream 메서드를 사용하여 "멀티모달" 주제에 대한 스트림 생성
for token in chain.stream({"topic": "멀티모달"}):
    # 토큰을 줄바꿈 없이 이어서 출력하고, 즉시 화면에 반영
    print(token, end="", flush=True)
```

1.2 LCEL 인터페이스

invoke : 호출

`chain.invoke()` 메서드는 한 번의 입력을 받아 해당 입력에 대해 체인을 실행하고 결과를 반환합니다.

- 입력은 보통 딕셔너리 형태(`dict`)로 전달합니다.
- 키는 프롬프트에서 사용하는 변수명, 값은 그 변수에 들어갈 실제 데이터입니다.

예시 코드

```
# chain 객체의 invoke 메서드를 호출하고,  
# 'ChatGPT' 라는 값을 topic 변수에 전달하여 실행  
chain.invoke({"topic": "ChatGPT"})
```

➡ 실행 결과는 모델의 응답(출력 텍스트 또는 파싱된 결과)이 리턴됩니다.

1.2 LCEL 인터페이스

특징

- 단일 입력만 처리 (한 번 호출 = 한 번 응답)
- 동기 방식 실행 → 함수 호출하듯 결과를 바로 얻음
- 여러 개의 입력을 처리하려면 `.batch()` 또는 `.stream()` 을 사용

1.2 LCEL 인터페이스

batch: 배치(단위 실행)

`chain.batch()` 메서드는 여러 입력을 한 번에 처리할 때 사용합니다.

- 입력으로 ****딕셔너리들의 리스트(list)****를 받습니다.
- 각 딕셔너리의 키는 프롬프트 변수명, 값은 해당 변수에 넣을 데이터입니다.
- 모든 입력을 순차 또는 병렬로 실행한 뒤 결과 리스트를 반환합니다.

예시 코드

```
# 여러 개의 topic 값을 한 번에 처리
chain.batch([
    {"topic": "ChatGPT"},
    {"topic": "Instagram"}
])
```

➡ 두 입력이 각각 실행되고, 결과는 리스트로 반환됩니다.

`results[0]`, `results[1]` 형태로 각각 확인할 수 있습니다.

1.2 LCEL 인터페이스

Parallel: 병렬 실행

LangChain Expression Language(LCEL)는 병렬 요청을 지원합니다.
여러 개의 체인을 동시에 실행하고, 결과를 묶어서 반환할 수 있습니다.
이를 위해 `RunnableParallel` 클래스를 사용합니다.

예시 코드

```
from langchain_core.runnables import RunnableParallel
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser

model = ChatOpenAI(model="gpt-4o-mini")
```

1.2 LCEL 인터페이스

```
# 1 {country}의 수도를 물어보는 체인
chain1 = (
    PromptTemplate.from_template("{country}의 수도는 어디야?")
    | model
    | StrOutputParser()
)

# 2 {country}의 면적을 물어보는 체인
chain2 = (
    PromptTemplate.from_template("{country}의 면적은 얼마야?")
    | model
    | StrOutputParser()
)

# 3 두 체인을 병렬로 실행하는 RunnableParallel 객체 생성
combined = RunnableParallel(capital=chain1, area=chain2)

# 4 호출: 두 체인이 동시에 실행되고 결과는 딕셔너리 형태로 반환
result = combined.invoke({"country": "한국"})
print(result)

# 출력 예시: {"capital": "서울", "area": "약 100,210 km²"}
```


1.2 LCEL 인터페이스

특징

- 여러 체인을 동시에 실행하므로 속도가 빠름 (병렬 처리)
- 결과는 지정한 키(capital, area 등)로 묶인 딕셔너리 형태로 반환
- 하나의 입력을 받아 서로 다른 질문을 동시에 던지고, 응답을 합쳐서 활용 가능

1.3 랭체인 메시지 클래스

1.3 랭체인 메시지 클래스

◆ 주요 메시지 클래스

LangChain에서는 다음과 같은 메시지 클래스가 제공된다.

- `SystemMessage`
 - 대화 전반에 대한 규칙, 지침을 담는다.
 - 예: `SystemMessage(content="너는 친절한 수학 선생님이다.")`
- `HumanMessage`
 - 사용자가 입력한 메시지를 나타낸다.
 - 예: `HumanMessage(content="2+2는 얼마인가?")`
- `AIMessage`
 - 모델이 출력한 응답을 나타낸다.
 - 예: `AIMessage(content="답은 4이다.")`
- `FunctionMessage`
 - 도구 호출 또는 함수 실행 결과를 담는다.
 - 예: API 호출 결과를 전달할 때 사용된다.

1.3 랭체인 메시지 클래스

◆ HumanMessage가 필요한 이유

1. 멀티턴 대화 지원

- 문자열 입력만으로는 대화가 길어질수록 역할 구분이 어렵다.
- `HumanMessage`, `AIMessage` 를 사용하면 대화 히스토리를 리스트 형태로 관리할 수 있다.

2. 역할(Role) 명확화

- 모델은 문장이 사람의 질문인지, 시스템 지침인지에 따라 다르게 반응한다.
- 예: `SystemMessage` 에서 "짧게 답하라"라고 지정하면 모든 답변이 짧게 나온다.

3. 에이전트 및 툴 연동 확장성

- 추후 Agent 구현 시 `HumanMessage`, `AIMessage`, `FunctionMessage`를 조합하여 툴 호출이나 계산기 사용 같은 구조적 대화를 구현할 수 있다.

1.3 랭체인 메시지 클래스

예제 소스

```
from langchain_core.messages import HumanMessage, AIMessage, SystemMessage

messages = [
    SystemMessage(content="너는 친절한 조교야."),
    HumanMessage(content="안녕?"),
    AIMessage(content="안녕하세요! 무엇을 도와드릴까요?"),
    HumanMessage(content="LangChain이 뭐야?")
]

response = model.invoke(messages)
print(response.content)
```

1.3 랭체인 메시지 클래스

한줄 요약

단순 문자열은 "한 번 질문하고 끝"에 적합하고,

`HumanMessage` 방식은 대화 히스토리 유지 + 역할 지정 + 메타데이터 + 함수 호출까지 가능해서
실제 서비스나 프로젝트 만들 때 훨씬 유리하다.

1.4 멀티 턴(Muti-turn)

1.4 멀티 턴(Multi-turn)

● 멀티 턴(Multi-Turn) 대화란?

멀티 턴 대화는 사용자와 모델이 여러 차례 주고받으며 맥락(context)을 유지한 대화를 말합니다. 즉, 한 번의 질문(턴) + 답변(턴)으로 끝나는 싱글 턴(single-turn) 대화와 달리, 이전 대화 내용을 기억하고 그 위에 계속 질문/답변을 이어가는 방식입니다.

💬 예시

✅ 싱글 턴

User: 파리의 수도는 어디야?

AI: 프랑스의 수도는 파리입니다.

- 질문 → 답변 → 끝
- 이전 대화를 기억하지 않음

1.4 멀티 턴(Multi-turn)

✓ 멀티 턴

User: 파리의 인구는?

AI: 약 2,100,000명입니다.

User: 거기서 가장 유명한 관광지는?

AI: 에펠탑, 루브르 박물관, 노트르담 대성당 등이 있습니다.

- 두 번째 질문은 *****파리*****라는 이전 맥락을 활용
- AI가 문맥을 유지해서 적절한 답변 가능

1.4 멀티 턴(Multi-turn)



멀티 턴의 핵심 요소

요소	설명
대화 히스토리 저장	이전 사용자 질문과 모델 답변을 함께 기억
맥락 유지	새 질문을 해도 이전 대화와 연결 지어 해석
지시/역할 지속	초반에 설정한 시스템 프롬프트(예: "친절한 과학 선생님")가 계속 유지

1.4 멀티 턴(Multi-turn)

● 멀티 턴 대화 구현 + 실행 예시

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage
from langchain_core.chat_history import InMemoryChatMessageHistory
from langchain_core.runnables.history import RunnableWithMessageHistory

# 1 모델 생성
model = ChatOpenAI(model="gpt-4o-mini")

# 2 세션별 대화 기록 저장소
store = {}

# 3 세션 ID에 따라 대화 기록 가져오기
def get_session_history(session_id: str):
    if session_id not in store:
        store[session_id] = InMemoryChatMessageHistory()
    return store[session_id]

# 4 모델에 대화 기록 기능 추가
with_message_history = RunnableWithMessageHistory(model, get_session_history)
```

1.4 멀티 턴(Multi-turn)

```
# 5 세션 ID 설정
config = {"configurable": {"session_id": "chat-001"}}

# 6 첫 번째 대화
response = with_message_history.invoke(
    [HumanMessage(content="안녕? 나는 홍길동이야!")],
    config=config,
)
print(response.content)

# 7 두 번째 대화 (이전 대화 맥락을 활용)
response = with_message_history.invoke(
    [HumanMessage(content="내 이름이 뭐지?")],
    config=config,
)
print(response.content)
```

실행 결과 예시

안녕하세요, 홍길동님! 만나서 반가워요.
당신의 이름은 홍길동입니다.

1.4 멀티 턴(Multi-turn)

🔑 작동 원리

1. 세션 관리

- `store` 라는 딕셔너리에 `session_id` 를 키로 저장
- 세션마다 별도의 대화 기록이 유지 → 여러 사용자가 동시에 접속해도 대화가 섞이지 않음

2. 대화 히스토리 저장

- `InMemoryChatMessageHistory()` 객체가 이전 메시지(Human/AI)를 메모리에 저장
- 모델 호출 시 자동으로 히스토리를 붙여서 맥락 유지

3. 실행 시 기록 포함

- `RunnableWithMessageHistory` 는 모델 실행 시 해당 세션의 대화 기록을 함께 전달
- 다음 질문을 할 때 이전 대화 내용이 자동으로 입력에 포함 → 멀티 턴 대화 가능

감사합니다