Ch13. LangChain Expression Language (LCEL)

LangChain Expression Language(LCEL)는 LangChain에서 복잡한 LLM 애플리케이션을 보다 구조적으로 설계하기 위해 도입된 선언적 파이프라인 인터페이스다. 기존의 LangChain 체인이 "어떤 순서로 무엇을 실행할 것인가"에 집중했다면, LCEL은 "입력이 어떤 경로를 따라 흘러가며 어떻게 변형되는가"를 중심에 둔다. 이 차이는 단순한 문법상의 변화가 아니라, LLM 시스템을 설계하는 사고방식 자체를 바꾼다.

LCEL에서는 프롬프트, 모델, 검색기, 후처리 로직, 메모리까지 모두 동일한 인터페이스인 Runnable로 취급된다. 이 덕분에 서로 다른 성격의 구성 요소를 일관된 방식으로 연결할 수 있고, 파이프 연산자(|) 하나로 전체 실행 흐름을 자연스럽게 표현할 수 있다. 코드 자체가 곧 실행 그래프가 되기 때문에, 복잡한 RAG나 멀티 모델 시스템도 읽기 쉬운 구조로 유지된다.

```
from langchain_openai
import ChatOpenAIfrom langchain_core.output_parsers
import StrOutputParserfrom langchain_core.promptsimport Cha
tPromptTemplate

prompt = ChatPromptTemplate.from_template(
    "Answer the following question:\n{question}"
)

model = ChatOpenAI()
parser = StrOutputParser()

chain = prompt | model | parser
```

LCEL의 설계 철학

LCEL의 가장 큰 특징은 선언적 구성이다. 개발자는 "어떻게(if, for, 상태 관리)"를 직접 제어하기보다는, 데이터가 어떤 단계를 거쳐 어디로 흘러가는지를 기술한다. 이로 인해 코드의 목적과 구조가 명확해지고, 수정이나 확장이 훨씬 수월해진다.

또 하나 중요한 점은 모든 것이 Runnable이라는 사실이다. 프롬프트도, LLM도, 사용자 정의 함수도 모두 동일한 방식으로 취급되기 때문에, 병렬 처리나 분기, 메모리 결합 같은 고급 기능이 일관된 인터페이스 위에서 자연스럽게 동작한다. LCEL은 단순히 체인을 편하게 만드는 도구가 아니라, LLM 애플리케이션을 하나의 데이터 플로우 시스템으로 바라보게 만드는 언어에 가깝다.

Runnable이라는 공통 단위

Runnable은 LCEL의 최소 실행 단위다. Runnable은 입력을 받아 출력을 반환할 수 있는 모든 객체를 의미하며, invoke, stream, batch 같은 공통 메서드를 가진다. 중요한 점은 Runnable이 "무엇을 하는지"보다 "어떻게 연결될 수 있는지"에 초점을 둔 인터페이스라는 것이다.

RunnableSequence는 여러 Runnable을 순차적으로 연결한 형태이고, RunnableParallel은 하나의 입력을 여러 Runnable에 동시에 전달해 병렬 실행을 수행한다. RunnableLambda는 사용자 정의 Python 함수를 Runnable로 감싸는 역할을 하며, RunnableBranch는 입력에 따라 실행 경로를 분기한다. 이 모든 구성 요소가 동일한 규칙으로 결합될 수 있기 때문에, LCEL 파이프라인은 매우 높은 조합성을 가진다.

RunnablePassthrough: 데이터 흐름을 유지하는 핵심 도구

RunnablePassthrough는 처음 보면 아무 일도 하지 않는 것처럼 보인다. 입력을 받아 그대로 반환하기 때문이다. 하지만 LCEL을 실제로 사용하다 보면, 이 클래스가 데이터 흐름을 설계하는 데 얼마나 중요한 역할을 하는지 금방 알게 된다.

```
from langchain_core.runnablesimport RunnablePassthrough

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

RunnablePassthrough는 입력을 손상시키지 않고 다음 단계로 전달하고 싶을 때 사용된다. 특히 병렬 구조나 RAG 파이프라인에서 "질문은 그대로 두고, 일부 데이터만 가공하고 싶을 때" 필수적인 역할을 한다. assign 메서드를 함께 사용하면, 기존 입력을 유지한 채 새로운 키를 추가할 수도 있다. 이 패턴은 입력을 덮어쓰지 않고 점진적으로 확장하는 LCEL 특유의 데이터 설계 방식을 잘 보여준다.

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

```
).invoke({"num":2})
```

RunnableParallel과 결합하면 그 효과는 더욱 분명해진다. 하나의 입력을 여러 경로로 동시에 흘려보내면서, 어떤 경로에서는 원본을 그대로 유지하고, 어떤 경로에서는 값을 계산하거나 변형할 수 있다. 이때 RunnablePassthrough는 "기준선이 되는 입력"을 유지하는 역할을 맡는다.

```
from langchain_core.runnablesimport RunnableParallel

runnable = RunnableParallel(
    passed=RunnablePassthrough(),
    extra=RunnablePassthrough.assign(mult=lambda x: x["nu
m"] *3),
    modified=lambda x: x["num"] +1,
)

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

RAG 파이프라인에서의 RunnablePassthrough 활용

검색 기반 생성(RAG) 구조에서 RunnablePassthrough의 역할은 특히 중요하다. 질문(question)은 그대로 유지되어야 하고, 컨텍스트(context)만 검색기를 거쳐 가공되어야 하는 경우가 대부분이기 때문이다.

```
retrieval_chain = (
    {
        "context": retriever | format_docs,"question": Runn
ablePassthrough(),
    }
    | prompt
    | model
    | StrOutputParser()
)
```

LCEL에서는 이 구조를 매우 자연스럽게 표현할 수 있다. 입력이 들어오면, 한쪽에서는 retriever를 통해 문서를 검색해 컨텍스트를 만들고, 다른 한쪽에서는 RunnablePassthrough를 통해 질문을 그대로 전달한다. 이후 이 두 값이 하나의 프롬프트

로 결합되어 모델에 전달된다. 이 구조는 “질문은 보존하고, 근거만 보강한다”는 RAG의 본질을 코드 구조 자체로 드러낸다.

Runnable 실행 구조를 그래프로 이해하기

LCEL로 체인을 구성하다 보면, 실제로 어떤 순서와 경로로 실행되는지 시각적으로 확인하고 싶어질 때가 많다. 이를 위해 제공되는 것이 get_graph 메서드다. 이 메서드는 체인을 실행 그래프로 변환하여 각 Runnable이 노드로, 데이터 흐름이 엣지로 표현된 구조를 반환한다.

```
graph = retrieval_chain.get_graph()
graph.print_ascii()
```

ASCII 형태로 출력된 그래프는 다소 투박해 보일 수 있지만, 병렬 입력, 패스스루, 프롬프트, 모델, 출력 파서까지의 전체 흐름을 한눈에 파악하는 데 큰 도움을 준다. 특히 복잡한 RAG나 분기 구조를 디버깅할 때, “어디서 어떤 데이터가 합쳐지는지”를 확인하는 데 매우 유용하다.

또한 get_prompts 메서드를 통해 체인 내부에서 실제로 사용되는 프롬프트를 추출할 수 있다. 이는 LLM 출력 품질을 개선하거나, 의도치 않은 프롬프트 구성을 점검할 때 중요한 도구다.

```
retrieval_chain.get_prompts()
```

RunnableLambda: 사용자 정의 로직을 LCEL에 녹여내기

RunnableLambda는 개발자가 직접 작성한 Python 함수를 LCEL 파이프라인에 포함시키기 위한 수단이다. 이를 통해 데이터 전처리, 계산, 외부 API 호출, 후처리 로직 등을 자연스럽게 체인 안에 넣을 수 있다.

```
from langchain_core.runnables import RunnableLambda

def length_function(text):
    return len(text)

chain = RunnableLambda(length_function)
chain.invoke("hello")
```

중요한 제약은 RunnableLambda가 단일 입력만 받을 수 있다는 점이다. 여러 인자를 처리하고 싶다면, 입력을 dict 형태로 구성하고 내부에서 이를 분해하는 방식으로 설계해야 한다. 이 제약은 오히려 데이터 구조를 명확하게 설계하도록 유도한다.

```
def multiple_length_function(inputs):
    return len(inputs["text1"]) + len(inputs["text2"])

chain = (
    {
        "text1": RunnablePassthrough(),
        "text2": RunnablePassthrough(),
    }
    | RunnableLambda(multiple_length_function)
)
```

RunnableLambda는 RunnableConfig를 함께 받을 수도 있다. 이를 통해 콜백, 태그, 메타데이터 같은 실행 정보를 전달할 수 있으며, LangSmith 추적과 결합하면 매우 세밀한 실행 분석이 가능해진다.

```
chain.invoke(
    "{foo:: bar}",
    config={
        "tags": ["debug"],
        "callbacks": [cb],
    },
)
```

조건 분기와 라우팅: RunnableLambda와 RunnableBranch

LCEL에서 조건에 따라 실행 경로를 바꾸는 방법은 크게 두 가지다. 하나는 RunnableLambda를 사용해 “다음에 실행할 Runnable 자체를 반환하는 방식”이고, 다른 하나는 RunnableBranch를 사용하는 방식이다.

```
def route(info):
    if "math" in info["topic"]:
        return math_chain
```

```
    elif "science" in info["topic"]:
        return science_chain
    else:
        return general_chain

full_chain = (
    RunnablePassthrough()
    | RunnableLambda(route)
)
```

RunnableBranch는 보다 선언적인 분기 구조를 제공한다. 조건과 Runnable 쌍을 나열하고, 첫 번째로 참이 되는 조건에 해당하는 Runnable을 실행한다.

```
from langchain_core.runnablesimport RunnableBranch

branch = RunnableBranch(
    (lambda x:"math"in x["topic"], math_chain),
    (lambda x:"science"in x["topic"], science_chain),
    general_chain,
)
```

RunnableParallel과 병렬 처리의 실제 의미

RunnableParallel은 하나의 입력을 여러 Runnable에 동시에 전달하고, 그 결과를 하나의 dict로 모아준다. LCEL의 병렬 처리는 단순한 편의 기능이 아니라, LLM 호출 비용과 응답 시간을 줄이기 위한 핵심 설계 요소다.

```
from operator import itemgetter

chain = {
    "context": itemgetter("question") | retriever,
    "question": itemgetter("question"),
    "language": itemgetter("language"),
}
```

여러 질문을 서로 다른 프롬프트로 동시에 처리하거나, 하나의 입력에서 여러 속성을 병렬로 계산할 때 RunnableParallel은 매우 효과적이다. 실제로 개별 체인을 순차 실행하는 것과

병렬 실행하는 것의 전체 실행 시간이 거의 동일하다는 점은, 병렬화가 곧 성능 최적화라는 사실을 잘 보여준다.

런타임 구성: configurable_fields와 configurable_alternatives

LCEL의 강력함은 실행 시점에 체인의 내부 구성을 바꿀 수 있다는 점에서 완성된다. configurable_fields를 사용하면 모델 이름, temperature 같은 속성을 런타임에 동적으로 변경할 수 있다.

```
from langchain_core.runnables import ConfigurableField

model = ChatOpenAI().configurable_fields(
    model_name=ConfigurableField(id="gpt_version")
)

model.invoke(
    "hello",
    config={
        "configurable": {"gpt_version": "gpt-4o-mini"}
    },
)
```

configurable_alternatives는 한 단계 더 나아가, Runnable 자체를 대체할 수 있게 해준다.

```
from langchain_anthropic import ChatAnthropic

llm = ChatAnthropic(model="claude-3-opus").configurable_alt
ernatives(
    ConfigurableField(id="llm"),
    default_key="anthropic",
    openai=ChatOpenAI(),
    gpt4o=ChatOpenAI(model="gpt-4o"),
)

llm.with_config(configurable={"llm": "openai"})
```

```
prompt = ChatPromptTemplate.from_template(
    "What is the capital of {country}?"
).configurable_alternatives(
    ConfigurableField(id="prompt"),
    capital=ChatPromptTemplate.from_template(
        "Capital of {country}?"
    ),
    area=ChatPromptTemplate.from_template(
        "Area of {country}?"
    ),
)
```

@chain 데코레이터로 함수 기반 체인 구성

```
from langchain_core.runnables import chain

@chain
def custom_chain(text):
    return text.upper()
```

RunnableWithMessageHistory: 메모리를 파이프라인에 결합하기

RunnableWithMessageHistory는 기존 Runnable 위에 메시지 기록을 덧붙이는 방식으로 동작한다.

```
from langchain_core.runnables.history
import RunnableWithMessageHistoryfrom langchain_core.chat_h
istory
import InMemoryChatMessageHistory

store = {}defget_session_history(session_id):if session_idn
otin store:
        store[session_id] = InMemoryChatMessageHistory()ret
```

```
urn store[session_id]

with_history = RunnableWithMessageHistory(
    chain,
    get_session_history,
    input_messages_key="input",
    history_messages_key="history",
)

with_history.invoke(
    {"input":"Hello"},
    config={"configurable": {"session_id":"abc123"}},
)
```

```
from langchain_community.chat_message_histories
import RedisChatMessageHistorydefget_session_history(sessio
n_id):
    return RedisChatMessageHistory(
            session_id=session_id,
        url=REDIS_URL,
  )
```

제네레이터 기반 스트리밍 파서

```
from typingimport Iterator,List
def split_into_list(input: Iterator[str]) -> Iterator[List
[str]]:
    buffer =""for chunkininput:
        buffer += chunkif","in buffer:
            parts = buffer.split(",")for pin parts[:-1]:yie
ld [p.strip()]
            buffer = parts[-1]

list_chain = str_chain | split_into_list
```

Runtime Argument 바인딩과 Tool 연동

```
model.bind(stop="SOLUTION")
```

```
model.bind(
    function_call={"name":"solver"},
    functions=[schema],
)
```

Fallback 전략: 실패를 설계의 일부로

```
llm = openai_llm.with_fallbacks([anthropic_llm])
```

```
safe_chain = bad_chain.with_fallbacks([
    fallback_chain_1,
    fallback_chain_2,
])
```

마무리: LCEL은 문법이 아니라 관점이다

LCEL은 단순히 LangChain의 새로운 체인 문법이 아니다.

LLM 애플리케이션을 “프롬프트 중심 코드”가 아니라 “데이터 흐름 기반 시스템”으로 바라보게 만드는 관점의 전환이다.

프롬프트는 리소스가 되고

모델은 교체 가능한 컴포넌트가 되며

체인은 실행 그래프가 되고

실패는 fallback 경로가 된다.