


LangChain 개요

LangChain은 대규모 언어 모델(LLM)을 활용한 애플리케이션 개발에 특화된 오픈소스 프레임워크입니다.



LangChain이란?



정의

LangChain은 대규모 언어 모델(LLM)을 활용한 애플리케이션 개발에 특화된 오픈소스 프레임워크입니다.



중요성

LangChain은 챗봇, 질문-답변 시스템, 콘텐츠 생성, 요약기 등 다양한 LLM 기반 애플리케이션을 보다 효율적으로 개발할 수 있도록 도와주며, AI 개발 시간을 단축하고 생산성을 향상시킵니다.

LangChain 패키지 그룹

LangChain은 다양한 컴포넌트들을 모듈식으로 결합하여 LLM 애플리케이션을 재사용 가능하고 유연하게 설계할 수 있습니다.



langchain-core

다양한 컴포넌트의 기본 추상화와 인터페이스만 제공

주요 컴포넌트: LLM 인터페이스, 프롬프트 템플릿, 체인, 메모리



langchain-community

서드파티 통합을 위한 패키지로, 다양한 외부 도구, API, 모델과의 연결을 제공합니다.

주요 컴포넌트: 문서 로더, 임베딩 모델, 벡터 스토어, Retriever, 도구 통합



langchain

주요 패키지로, langchain-core + 일반적인 구현을 포함

주요 컴포넌트: 체인, 에이전트, 메모리 시스템, 문서 변환기, RAG 파이프라인



partners

특정 partner(제공자)별 통합을 위한 경량 패키지로, 특정 LLM 제공자나 도구와의 최적화된 연결을 제공합니다.

예시: langchain-openai, langchain-anthropic, langchain-google-vertexai,
langchain-huggingface,

langchain-text-splitters

텍스트를 청크라고 분리는 단위로 분할하는 Text splitter



langchain-experimental

연구, 시험 목적의 코드나 알려진 취약점(CVE)을 포함하는 코드

LangChain의 핵심 구성 요소



Models (모델)

다양한 언어 모델(LLM)과의 인터페이스를 제공합니다. OpenAI의 GPT, Hugging Face 모델 등 다양한 모델을 연결하고 쿼리할 수 있도록 추상화된 API를 제공합니다.



Prompts (프롬프트)

언어 모델에 입력할 텍스트를 구성하는데 사용됩니다. Prompt Template, Example Selectors 등을 포함하여 모델이 특정 작업에 대해 정확하고 일관된 응답을 생성하도록 돕습니다.



Chains (체인)

여러 구성 요소를 연결하여 복잡한 작업을 수행하는 핵심 메커니즘입니다. 사용자의 쿼리부터 모델의 출력에 이르기까지 자동화된 작업을 정의하며, 다양한 용도로 활용됩니다.



Indexes (인덱스)

대량의 텍스트 데이터를 효율적으로 처리하고 검색하는 데 사용됩니다. Document Loaders, Text Splitters, Retriever, Vectorstore 등을 포함하여 외부 문서나 데이터베이스의 정보를 LLM이 활용할 수 있도록 구조화합니다.



Memory (메모리)

대화의 이전 내용을 저장하고 관찰하여 LLM이 과거 상호 작용에서 얻은 정보를 바탕으로 응답을 수정할 수 있도록 합니다. 간단한 대화 기록부터 복잡한 메모리 구조까지 다양한 형태를 지원합니다.



Agents (에이전트)

주어진 목표를 달성하기 위해 자동으로 도구를 선택하고 사용하는 지능형 시스템입니다. 언어 모델이 어떤 조치를 취할지 결정하고, 해당 조치를 실행하며, 관찰하고, 필요한 경우 반복하는 과정을 포함합니다.

이러한 구성 요소들은 서로 유기적으로 결합하여 복잡한 AI 워크플로우를 구축합니다.

LangChain의 장단점



장점



다양한 LLM 통합의 유연성

OpenAI의 GPT-4나 Hugging Face 모델과 같은 다양한 대규모 언어 모델과 쉽게 통합할 수 있어, 복잡하고 다양한 애플리케이션을 구축할 수 있습니다. 모델을 확장하고 조정하는 높은 확장성을 제공합니다.



높은 커스터마이징 가능성

프롬프팅, 컨텍스트 관리, 파인튜닝 등 다양한 기능을 통해 사용자가 애플리케이션의 동작을 세밀하게 조정할 수 있는 높은 유연성을 제공합니다. 다양한 산업 분야에서 맞춤형 솔루션을 구현하는 데 중요합니다.



활발한 오픈소스 커뮤니티 지원

LangChain은 활발한 오픈소스 커뮤니티의 지원을 받으며 지속적인 업데이트와 개발이 이루어지고 있습니다. 개발자들은 커뮤니티를 통해 경험을 공유하고 문제 해결에 도움을 받을 수 있습니다.



단점



복잡한 애플리케이션에서의 성능 최적화 필요성

매우 복잡한 애플리케이션을 구축하거나 대규모 데이터 소스를 처리할 때 성능 저하가 발생할 수 있습니다. 추가적인 하드웨어 리소스나 코드 최적화가 필요할 수 있습니다.



초심자를 위한 학습 곡선

LangChain을 처음 사용하는 개발자에게는 프레임워크가 다소 복잡하게 느껴질 수 있습니다. 다양한 기능과 옵션을 효과적으로 활용하기 위해서는 깊이 있는 이해와 일정한 학습 시간이 요구됩니다.



모든 유스케이스에 적합하지 않음

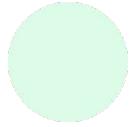
LangChain은 특정 사용 사례에 적합하도록 설계되었지만, 모든 상황에 완벽하게 맞는 것은 아닙니다. 특정 산업에서 요구하는 고도로 특화된 기능이 기본적으로 제공되지 않을 수 있습니다.



LangChain의 Model

- ✓ 여러 공급자의 모델을 일관된 인터페이스로 사용할 수 있도록 추상화

주요 Model 제공사



OpenAI

GPT-3.5, GPT-4o 등 다양한 채팅 전용 LLM을 제공하며, 높은 성능과 폭넓은 활용성을 자랑합니다.



Anthropic

Claude 모델 시리즈를 통해 AI 안전성과 연구에 중점을 둔 LLM을 제공합니다.



Google

Gemini 등 Google의 LLM을 LangChain과 연동하여 사용할 수 있습니다.



Azure

Microsoft Azure OpenAI 서비스를 통해 OpenAI 모델을 활용할 수 있습니다.



AWS Bedrock

Amazon Bedrock을 통해 다양한 파운데이션 모델을 LangChain과 통합하여 사용할 수 있습니다.



기타 제공사

- ✓ Hugging Face: 오픈소스 모델 통합
- ✓ Ollama: 로컬 환경에서 오픈소스 모델 실행
- ✓ Cohere: 기업용 AI 솔루션
- ✓ Upstage: 국내 스타트업 LLM 및 Document AI

Model의 주요 유형



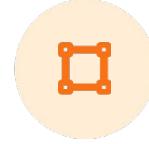
LLMs (대규모 언어 모델)

- ✓ 텍스트 문자열을 입력받아 텍스트 문자열을 출력하는 가장 기본적인 언어 모델 유형
- ✓ 콘텐츠 작성, 언어 번역, 텍스트 요약, 질문 답변 등 단일 작업에 활용



Chat Models (채팅 모델)

- ✓ 메시지 목록(시스템 메시지, 사용자 메시지, AI 메시지)을 입력으로 받아 다음 대화 메시지를 출력
- ✓ 대화의 맥락과 흐름을 유지하며 상호작용하는 애플리케이션에 적합
- ✓ 최근 모델들은 대부분 채팅 모델 형태로 제공되며, 메시지에 고유한 역할을 할당



Text Embedding Models (텍스트 임베딩 모델)

- ✓ 텍스트 데이터를 숫자 벡터(임베딩)로 변환하는 역할을 함
- ✓ 텍스트 간의 의미적 유사성을 비교하거나 효율적인 검색을 수행할 수 있도록 지원
- ✓ RAG(검색 증강 생성) 시스템에서 핵심 구성 요소로, 외부 데이터 소스에서 관련 정보를 검색

기본 호출 방법 (Invoke)

⚡ invoke() 메서드

- ✓ 동기 방식의 기본적인 호출 방법
- ✓ 단일 프롬프트에 대한 모델의 응답을 즉시 반환
- ✓ 간단한 질의응답이나 단일 작업 처리에 적합

💡 핵심 개념

invoke()는 LangChain에서 모델과의 상호작용을 시작하는 가장 직관적인 방법으로, 요청을 보내고 응답을 받을 때까지 기다리는 블로킹 메서드입니다.

</> 코드 예시

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

model = ChatOpenAI(
    model="gpt-4o-mini",      # 모델명
    temperature=0,           # 응답 일관성
    openai_api_key=os.environ["OPEN_API_KEY"])

messages = [
    SystemMessage("당신은 여행 전문가입니다."),
    HumanMessage("한국의 대표적인 관광지 3군데를 추천해주세요"),
    AIMessage("안녕하세요! 한국의 대표적인 관광지로는 다음 세 곳을 추천드립니다. ..."),
    HumanMessage("그 중에서 가족 단위 여행객에게 가장 적합한 곳은 어디인가요?")]

# AIMessage
ai_message = model.invoke(messages)
print(ai_message)
print(ai_message.content)
```

스트리밍과 배치 처리



스트리밍 (Stream)

모델의 응답을 실시간으로 토큰 단위로 받아오는 방식으로, 챗봇과 같은 즉각적인 응답이 필요한 애플리케이션에서 사용자 경험(UX)을 향상시킵니다.

```
model.stream({"topic": "한국의 유명 아이돌"})
```

실시간 토큰 단위 처리 →

사용자에게 즉각적 응답



배치 처리 (Batch)

여러 입력(prompt)을 한 번에 묶어서 일괄 처리할 수 있도록 설계되었습니다.

```
chain.batch(["아이스크림", "스파게티", "떡볶이"])
```

여러 요청 한 번에 처리 →

비용 감소 & 효율성 향상

특징:

- 여러 프롬프트를 한 번의 호출로 처리

특징:

- 모델의 응답을 토큰 단위로 실시간 처리
- 대화형 애플리케이션에서 사용자 경험 향상
- 스트리밍 방식으로 메모리 효율적 처리

비동기 호출과 고급 기능

LangChain은 동시성 처리가 필요한 애플리케이션에서 성능을 최적화하기 위해 비동기 메서드를 제공합니다.

ainvoke()

비동기 방식으로 단일 프롬프트에 대한 모델 응답을 받습니다.

```
response = await chain.invoke("아이스크림")
```

astream()

비동기 방식으로 모델 응답을 토큰 단위로 실시간으로 받습니다.

```
async for chunk in chain.astream("주제"):
    print(chunk, end="", flush=True)
```

abatch()

비동기 방식으로 여러 프롬프트를 한 번의 호출로 처리합니다.

```
responses = await chain.abatch(["피자", "커피", "책"])
```

비동기 메서드의 장점

- ✓ 여러 작업을 동시에 실행하여 성능 최적화
- ✓ 리소스 소비를 줄이고 효율적인 처리
- ✓ 응답 대기 시간을 최소화
- ✓ 스트리밍 및 배치 처리에 적합



Langchain의 PromptTemplate

효율적인 LLM 애플리케이션 개발을 위한 프롬프트 템플릿 활용법

</>템플릿 생성

✿✿변수 치환

◆◆LLM 상호작용

PromptTemplate이란?

PromptTemplate 개요

LangChain에서 프롬프트를 템플릿화하여 동적으로 내용을 변경할 수 있게 해주는 핵심 컴포넌트입니다.

주요 특징

- 프롬프트 템플릿화를 통해 재사용성 향상
- 동적 데이터 삽입 가능 (예: {dish})
- 사용자 입력을 안전하게 처리

레시피 생성 AI 앱

사용자 입력: "카레"



PromptTemplate

템플릿 정의

당신은 전문 요리사입니다.
다음 요리의 레시피를 만들어 주세요.

요리명: {dish}

조건:

- 재료와 분량을 상세히 작성
 - 조리 순서와 시간, 온도를 구체적으로 표시
 - 요리 팁과 변형 방법 포함
- 최종 레시피는 단계별로 명확하게 작성해주세요.



완성된 프롬프트

당신은 전문 요리사입니다.
다음 요리의 레시피를 만들어 주세요.

요리명: 카레

조건:

- 재료와 분량을 상세히 작성
 - 조리 순서와 시간, 온도를 구체적으로 표시
 - 요리 팁과 변형 방법 포함
- 최종 레시피는 단계별로 명확하게 작성해주세요.

ChatPromptTemplate

ChatPromptTemplate

대화형 모델, 즉 챗봇과 같은 애플리케이션에 특화된 프롬프트 템플릿입니다.

시스템, 인간, AI 메시지 템플릿을 사용하여 대화의 맥락을 명확하게 정의하고, LLM이 보다 자연스럽고 일관된 응답을 생성하도록 유도합니다.

 ChatPromptTemplate은 대화형 LLM 애플리케이션 개발에 있어 필수적인 구성 요소로, 대화의 흐름과 일관성을 유지하는 데 큰 역할을 합니다.

대화 흐름

ChatPromptTemplate은 세 가지 메시지 템플릿을 결합하여 대화의 맥락을 정의합니다

⚙️ 시스템 메시지

👤 인간 메시지

🤖 AI 메시지

시스템 메시지 템플릿

LLM의 전반적인 역할, 행동, 지시사항을 정의합니다.

“ 예: “당신은 친절한 AI 비서입니다.” ”

인간 메시지 템플릿

사용자의 질문이나 발언을 나타냅니다. 동적인 입력 변수를 포함할 수 있습니다.

“ 예: “오늘 날씨는 어떻게 되나요?” ”

AI 메시지 템플릿

LLM이 이전에 생성했던 응답을 나타냅니다. 대화의 흐름을 유지하는데 도움이 됩니다.

“ 예: “오늘은 화날씨입니다. 우산을 챙기세요.” ”

MessagePlaceholder의 개념과 필요성

정의 및 목적

MessagePlaceholder는 LangChain의 채팅 프롬프트 템플릿 (ChatPromptTemplate) 내에서 여러 메시지 객체로 구성된 메시지 목록을 프롬프트의 특정 위치에 동적으로 주입할 수 있습니다.

MessagePlaceholder 기본 구조

```
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant."),
    MessagesPlaceholder(variable_name="chat_history"),
    ("human", "{input}")
])
```

MessagePlaceholder의 해결책

MessagePlaceholder는 이러한 문제를 해결하며, 대화 기록(Chat History)과 같은 동적으로 변하는 메시지 목록을 프롬프트의 원하는 위치에 쉽게 삽입할 수 있도록 합니다. 이를 통해 개발자는 대화의 맥락을 효과적으로 유지하고, 모델이 이전 상호작용을 기반으로 정보에 입각한 결정을 내리도록 도웁니다.

```
prompt_value = prompt.invoke(
    {
        "chat_history": [
            HumanMessage(content="안녕하세요. 저는 앤리스입니다."),
            AIMessage("안녕하세요. 앤리스님! 어떻게 도와드릴까요?")
        ],
        "input": "제 이름을 아시나요?"
    }
)
```

PromptTemplate의 장점과 결론



프롬프트 재사용성

동일한 구조의 프롬프트를 여러 번 작성할 필요 없이, 템플릿 하나로 다양한 상황에 맞는 프롬프트를 생성할 수 있습니다.



유지보수성

프롬프트 내용 변경 시 템플릿만 수정하면 되므로, 전체 코드베이스를 일일이 찾아 수정하는 번거로움을 줄일 수 있습니다.

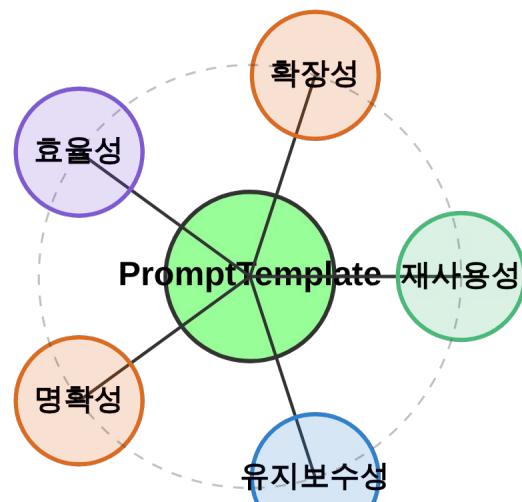


명확성

템플릿을 통해 프롬프트의 의도와 구조가 명확하게 드러나므로, 개발자와 LLM 모두 프롬프트를 더 정확하게 이해하고 처리할 수 있습니다.

★ 결론

PromptTemplate은 Langchain 기반 LLM 애플리케이션의 **효율성**, **확장성**, 및 **견고성**을 확보하는데 필수적인 도구이며, LLM과의 상호작용을 더욱 정교하고 효과적으로 만들어줍니다. 복잡한 LLM 애플리케이션을 개발할 때 PromptTemplate을 적절히 활용하면, 코드의 가독성과 유지보수성을 높이고, 보다 일관되고 신뢰할 수 있는 결과를 얻을 수 있습니다.



LangChain OutputParser

대규모 언어 모델(LLM)의 출력을 구조화된 형식으로 변환하는 핵심 컴포넌트

OutputParser의 필요성

대규모 언어 모델(LLM)의 출력을 개발자가 원하는 구조화된 형식으로 변환하여 애플리케이션에서 쉽게 활용할 수 있도록 돕는 핵심 컴포넌트입니다.

OutputParser 미사용 시

"송강호는 2019년에 개봉한 영화 기생충에서 주연을 맡았습니다. 그의 대표 작품으로는 살인의 추억, 광해, 왕이 된 남자 등도 있습니다."

⚠️ 비정형 텍스트: 애플리케이션에서 직접 활용하기 어려움



OutputParser

</> OutputParser 사용 시

```
{  
  "이름": "송강호",  
  "출연작": ["기생충", "살인의 추억", "광해, 왕이 된 남자",  
             "반도", "엽기적인 그녀"]  
}
```

✓ 구조화된 데이터: 애플리케이션에서 쉽게 활용 가능

- LLM은 기본적으로 free-form text를 생성하기 때문에 직접 활용하기 어려움
- 개발자는 OutputParser를 통해 LLM의 출력을 원하는 형식으로 쉽게 변환할 수 있음

주요 OutputParser 종류

LangChain은 다양한 형식으로 LLM 출력을 변환하고 처리할 수 있도록 여러 종류의 OutputParser를 제공합니다.

A StrOutputParser

가장 기본적인 출력 파서로, LLM의 응답을 어떠한 변형 없이 그대로 문자열로 반환합니다.

원시 텍스트 간단한 테스트

 주로 LLM의 원시 텍스트 출력을 확인하거나, 구조화된 데이터가 필요하지 않은 간단한 텍스트 응답 처리에 사용됩니다.

</> JsonOutputParser

LLM의 출력을 JSON 형식으로 파싱하여 Python 딕셔너리로 변환하는 데 사용됩니다.

키-값 형태 JSON 스키마

 모델의 출력이 이미 JSON 구조를 가지고 있거나, JSON 형식으로 출력을 유도할 때 매우 유용합니다.

PydanticOutputParser

Pydantic 모델을 기반으로 LLM에게 구조화된 출력 형식을 강제합니다.

명시적 프롬프트를 만들어야 함 (`get_format_instructions()` 사용)

LLM 응답을 받은 후, `invoke()` 로 Python 객체(Pydantic 모델)로 변환

데이터 검증 타입 안정성

 복잡한 데이터 구조를 다루거나, 데이터 유효성 검사 및 타입 안정성이 필요한 경우에 주로 사용됩니다.

CommaSeparatedListOutputParser

LLM의 출력을 쉼표로 구분된 리스트 형태로 변환할 필요가 있을 때 사용됩니다.

목록 형태 자동 변환

 여러 항목을 목록 형태로 받아야 할 때 유용하며, 사용자가 입력한 데이터를 쉼표로 구분하여 명확하고 간결한 목록 형태로 제공합니다.

💡 OutputParser의 핵심 가치

LangChain OutputParser는 LLM(거대 언어 모델)의 비정형 텍스트 출력을 개발자가 원하는 구조화된 데이터 형식으로 변환하는 [핵심적인 도구](#)입니다. 이는 LLM 기반 애플리케이션 개발 시 데이터 처리의 효율성과 안정성을 크게 향상시킵니다.



다양한 출력 형식 지원

OutputParser는 단순 문자열 반환부터 JSON, Pydantic 모델, 쉼표로 구분된 리스트 등 다양한 형식으로 출력을 변환하며, 데이터 유효성 검증 및 후속 처리의 용이성을 제공합니다.



LCEL과의 조합

LangChain Expression Language(LCEL)를 활용하여 Prompt, Model과 함께 체인으로 구성함으로써, LLM의 잠재력을 최대한 발휘하고 복잡한 애플리케이션을 효과적으로 구축할 수 있습니다.



애플리케이션 통합 용이

OutputParser를 통해 LLM의 출력을 애플리케이션에서 직접 활용 가능한 형태로 가공하면, 모델과 애플리케이션 간의 인터페이스 역할을 하여 통합 과정을 간소화합니다.



개발 생산성 향상

OutputParser는 LLM 기반 애플리케이션에서 데이터의 일관성과 활용성을 높여주며, 개발자의 코드 복잡성을 줄이고 생산성을 향상시킵니다.

★ OutputParser는 LLM의 출력을 실제 시스템에서 활용 가능한 형태로 가공하는 데 필수적인 역할을 수행합니다.

LangChain with_structured_output

기존 방식

복잡한 파싱 로직 & 오류 처리

with_structured_output

간단하고 신뢰able한 구조화된 출력

새로운 패러다임

모델의 네이티브 기능 활용

with_structured_output 소개

LangChain의 `with_structured_output` 메서드는 LLM의 출력을 신뢰성 높고 효율적으로 구조화하는 강력한 해결책입니다.

💡 핵심 개념

`with_structured_output`은 모델이 특정 스키마와 일치하는 출력을 반환하도록 강제하며, 내부적으로 모델의 [네이티브 기능](#)을 활용합니다.

🛠 도구/함수 호출

- ✓ 최신 LLM은 특정 기능을 호출하는 방식으로 구조화된 데이터를 생성할 수 있습니다
- ✓ `with_structured_output`은 이 기능을 활용하여 정의된 스키마에 따라 모델이 응답하도록 지시

📦 JSON 모드

- ✓ OpenAI, Mistral, Ollama 등 일부 모델은 JSON 모드를 지원
- ✓ JSON 모드는 출력이 항상 유효한 JSON 형식으로 제한되도록 합니다



LLM



`with_structured_output`
(스키마 적용)



구조화된 출력
(Pydantic, TypedDict, JSON)

핵심 장점

LangChain의 `with_structured_output` 메서드는 LLM 기반 애플리케이션 개발에 여러 가지 중요한 이점을 제공합니다.



코드 간소화

복잡한 설정 없이 간단하게 스키마 기반의 구조화된 출력을 얻을 수 있어 개발 과정을 크게 단순화합니다.



높은 신뢰성

모델의 네이티브 기능을 활용하여 스키마를 정확하게 따르도록 출력을 강제하므로, 출력의 신뢰성이 매우 높습니다.



자동 파싱 및 유효성 검사

적절한 출력 파서를 자동으로 가져오고 스키마를 모델에 맞게 포맷팅하는 작업을 처리하여, 개발자가 수동으로 파싱 로직을 구현할 필요가 없습니다.



모델 호환성

OpenAI, Grok, Gemini 등 다양한 모델에서 `with_structured_output`을 지원하며, 모델 목록은 지속적으로 업데이트됩니다.



유연한 스키마 정의

Pydantic 클래스, TypedDict, JSON 스키마 등 다양한 방식으로 출력 스키마를 정의할 수 있어 프로젝트의 요구사항에 맞춰 유연하게 적용할 수 있습니다.



`with_structured_output`를 사용하면 보다 간결하고 안정적인 코드로 구조화된 출력을 얻을 수 있습니다

Pydantic 모델 활용

Pydantic `BaseModel`을 활용하는 것은 `with_structured_output`을 사용하는 가장 강력하고 권장되는 방법 중 하나입니다. 모델의 출력 스키마를 명확하게 정의하고 자동으로 유효성을 검사하는 이점을 제공합니다.

</> Pydantic 모델 정의

```
from pydantic import BaseModel, Field

class Receipe(BaseModel):
    ingredients: list[str] = Field(description="요리 재료")
    steps: list[str] = Field(description="요리 순서")
```

모델 적용:

```
chain = prompt | model.with_structured_output(Receipe)

receipe = chain.invoke({"dish": "김치찌개"})

pprint.pprint(receipe)
print(receipe.ingredients)
print(receipe.steps)
```



Pydantic 모델의 이점

- 자동 유효성 검사로 안정적인 데이터 처리
- 명확한 필드 정의와 설명으로 코드 가독성 향상
- 필수 필드 누락이나 타입 오류 시 자동으로 오류 발생



</>

LangChain Expression Language

(LCEL)



LCEL



LCEL의 기본 개념

i LCEL이란?

LangChain Expression Language(LCEL)는 LangChain 애플리케션에서 복잡한 체인을 구성하고 조립하는 데 사용되는 강력한 도구로, 선언적이고 조합 가능한 방식으로 구성 요소를 연결하여 LLM 기반 애플리케이션의 개발을 간소화합니다.

💬 프롬프트 (Prompt)

사용자 입력과 컨텍스트를 기반으로 LLM에 전달될 최종 프롬프트 문자열을 생성합니다.

🧠 모델 (Model)

실제 언어 모델(LLM) 또는 채팅 모델(ChatModel)을 나타내며, 프롬프트 입력을 받아 응답을 생성합니다.

📋 출력 파서 (Output Parser)

모델이 생성한 원시 문자열 출력을 구조화된 형식으로 변환합니다 (JSON, 리스트, 특정 객체).

എ পাইপ যন্ত্র (|)

পাইপ যন্ত্র হল Unix পাইপ ও উভাবে একটি কম্পোনেন্ট যা একটি কম্পোনেন্টের উত্তর পথের মাধ্যমে অন্য কম্পোনেন্টের প্রয়োগ করে।

prompt

model

output_parser

LCEL 주요 구성 요소



프롬프트 (Prompt)

사용자 입력과 컨텍스트를 기반으로 LLM에 전달될 최종 프롬프트 문자열을 생성합니다.

예시: ChatPromptTemplate.from_template("당신은 친절한 AI 어시스턴트입니다. 질문: {question}")



모델 (Model)

실제 언어 모델(LLM) 또는 채팅 모델(ChatModel)을 나타내며, 프롬프트 입력을 받아 응답을 생성합니다.

예시: ChatOpenAI(model="gpt-4")



출력 파서

모델이 생성한 원시 문자열 출력을 구조화된 형식(예: JSON, 리스트, 특정 객체)으로 변환합니다.

예시: StrOutputParser(),
JsonOutputParser()



RunnableLambda

사용자 정의 함수나 람다 표현식을 LCEL 체인의 일부로 통합할 수 있게 합니다.

예시: RunnableLambda(lambda x:
x["question"].upper())



RunnableParallel

여러 Runnable를 병렬로 실행하여 그 결과를 하나의 딕셔너리로 묶어 반환하는 구조입니다.

예를 들면:

- 같은 입력을 가지고 여러 가지 결과를 동시에 계산하고 싶은 경우
- 서로 다른 모델이나 서로 다른 파서를 동시에 실행하고 싶은 경우

예시:

RunnableParallel(question=itemgetter("question"),
context=itemgetter("context"))



RunnableBranch

조건부 로직을 구현하여 특정 조건에 따라 서로 다른 실행 경로를 선택할 수 있게 합니다.

예시: RunnableBranch((lambda x: "foo" in x["topic"],
foo_chain), bar_chain)

파이프 연산자로 체인 구성하기

</> 파이프 연산자 (|)란?

파이프 연산자는 한 컴포넌트의 출력을 다음 컴포넌트의 입력으로 연결하는 기능입니다.



기본 코드 예시

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

# 파이프 연산자(|)로 체인 구성
chain = ChatPromptTemplate.from_template("질문: {question}") | ChatOpenAI() | StrOutputParser()
```

💡 다양한 Runnable 연결하기

RunnableParallel

+

RunnableBranch

```
from langchain_core.runnables import RunnableParallel, RunnableBranch

# 병렬 실행 - 파이프(|)로 연결
parallel = RunnableParallel(
    summary=prompt1 | model,
    translation=prompt2 | model
)
# 조건부 분기 - 파이프(|)로 연결
branch = RunnableBranch(
    (lambda x: len(x) > 100, prompt1 | model),
    prompt2 | model
)
# 다양한 Runnable을 파이프(|)로 연결
chain = user_input | branch | parallel
```

Runnable 실행 방법

다양한 실행 방법

LCEL 체인은 Runnable 인터페이스로 다양한 방식의 실행이 가능합니다.

.invoke() - 단일 입력 실행

단일 입력에 대해 동기식으로 체인을 실행

```
response = chain.invoke({"question": "LCEL이란?"})
```

.batch() - 배치 처리

여러 입력을 한 번에 일괄 처리

```
responses = chain.batch([{"question": "질문1"}, {"q": "질문2"}])
```

.stream() - 스트리밍 응답

생성되는 즉시 응답을 순차적으로 반환

```
for chunk in chain.stream({"question": "질문"}): print(chunk)
```

.ainvoke() - 비동기 실행

비동기 방식으로 체인을 실행

```
result = await chain.ainvoke({"question": "질문"})
```

</> 체인 구성 및 실행 예시

```
# 체인 구성
prompt = ChatPromptTemplate.from_template("질문: {question}")
model = ChatOpenAI()
chain = prompt | model | StrOutputParser()

# 실행 예시
chain.invoke({"question": "질문"})
chain.batch([{"question": "질문1"}, {"question": "질문2"}])
```

추가 실행 메서드

.astream()
비동기 스트리밍 출력

.abatch()
비동기 일괄 처리

.with_config()
런타임 설정 적용

LangChain Chat History와 Memory 기능

대화형 AI 애플리케이션의 핵심 요소

Chat History

안녕하세요!

이전 대화를 기억하고 있습니다.

맥락을 이해합니다.



Memory



맥락을 관리하고 요약



LLM의 근본적 한계와 해결 필요성

⚠️ LLM의 상태 비저장 특성

- ✖️ 이전 대화 내용을 **기억하지 못함**
- ✖️ 매번 새로운 질문으로 인식하여 **답변하는 한계**
- ✖️ 사용자와의 자연스러운 **연속적 대화**에 걸림돌

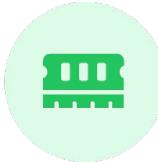
🔍 대화형 AI 애플리케이션에서의 문제

챗봇과 같은 대화형 AI 애플리케이션은 자연스럽고 연속적인 대화가 필요하지만, LLM의 상태 비저장 특성은 이에 부적합합니다.

💡 LangChain의 해결책



Chat History



Memory

LangChain은 **Chat History** 와 **Memory** 기능을 통해 이전 대화의 맥락을 유지하고 유기적인 상호작용을 가능하게 합니다.

Chat History 개념과 역할

⌚ Chat History의 정의

Chat History는 대화 메시지를 물리적인 공간에 저장하고 관리하는 핵심적인 역할을 수행합니다.

이는 LLM이 이전 대화의 맥락을 이해하고 연속적인 대화를 이어갈 수 있도록 하는 **기반이 됩니다**.

⚙️ Chat History의 역할

- ✓ 메시지를 구분하여 저장 함으로써 대화의 흐름을 유지
- ✓ Memory 기능과 연동되어 **프롬프트의 내용을 구성** 하는 데 활용
- ✓ LLM에 전달될 **메시지 구조를 정의** 함

📍 메시지 구조

LangChain은 대화의 각 참여자(시스템, 사용자, AI)의 메시지를 명확히 구분하기 위해 특정 메시지 유형을 사용합니다.



System
시스템 메시지



Human
사용자 메시지



AI
AI 메시지

LangChain ChatMessageHistory 유형

LangChain은 다양한 **저장소**를 활용하여 대화 기록을 **효과적으로 관리** 할 수 있는 여러 ChatMessageHistory 구현체를 제공합니다.



SQLChatMessageHistory

SQL 데이터베이스에 대화 기록을 저장하는 방식입니다.

- ✓ 구조화된 데이터 관리와 복잡한 쿼리 가능
- ✓ 대규모 대화 기록에 적합



RedisChatMessageHistory

Redis에 채팅 기록을 저장하는 고성능 방식입니다.

- ✓ 초고속 메모리 기반 데이터 액세스
- ✓ 실시간 애플리케이션에 이상적



FileChatMessageHistory

파일 시스템에 대화 기록을 저장하는 간단한 방식입니다.

- ✓ 설정이 간단하고 외부 의존성 없음
- ✓ 소규모 프로젝트나 프로토타이핑에 적합



MongoDBChatMessageHistory

MongoDB에 대화 기록을 저장하는 유연한 방식입니다.

- ✓ 문서 기반 유연한 스키마
- ✓ 대화 데이터의 복잡한 구조도 쉽게 저장



CloudflareD1ChatMessageHistory

Cloudflare D1 서비스에 대화 기록을 저장하는 방식입니다.

- ✓ 글로벌 엣지 네트워크에서 동작
- ✓ 확장성과 성능이 뛰어나며 서비스 환경에 적합

Memory 기능의 역할과 중요성

Memory는 저장된 대화 기록을 가공하여 대규모 언어 모델(LLM)의 프롬프트에 효과적으로 전달하는 관리자 역할을 수행합니다.



Chat History
대화 메시지 저장



Memory
메모리 관리 및 처리



LLM
맥락을 이해한 답변 생성



맥락(contextual) 대화

Memory는 LLM이 이전 대화의 맥락을 이해하고 자연스러운 대화를 이어갈 수 있도록 돕습니다.



적극적 메모리 관리

저장된 대화 기록을 적절히 관리하여 LLM의 토큰 제한을 초과하지 않도록 합니다.



다양한 메모리 유형

LangChain은 다양한 상황과 요구사항에 맞춰 여러 가지 Memory 유형을 제공합니다.

ConversationBufferMemory

모든 대화 기록을 그대로 저장하는 기본 메모리 유형

ⓘ 개념 및 작동 방식

ConversationBufferMemory는 LangChain에서 제공하는 가장 기본적인 메모리 유형으로, 이전 대화의 모든 기록을 그대로 저장합니다.

⚠ 장단점

장점

- ✓ 모든 맥락 유지
- ✓ 구현 용이성

단점

- ⚠ 토큰 제한 문제

ConversationBufferWindowMemory

최근 k개의 대화만 기억하는 윈도우 방식 메모리

작동 방식

- ✓ **k** 값으로 유지할 대화 개수 지정
- ✓ 새 대화 추가 시 **오래된 대화부터 삭제**
- ✓ 프롬프트의 크기를 일정하게 유지

장점

- > 토큰 제한 관리
- > API 비용 절감
- > 응답 속도 향상

단점

- > 오래된 대화 손실
- > 맥락 부족 문제

윈도우 메모리 작동 방식

오래됨 오래됨 최근1 최근2 최근3

k=3 설정 시 3개의 최근 대화만 유지

</> 코드 예시

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferWindowMemory

llm = ChatOpenAI(temperature=0.0)

# 최근 1개 대화만 기억
memory = ConversationBufferWindowMemory(k=1)
```

ConversationSummaryMemory

LLM을 활용한 고급 메모리 관리 방식

개념 정의

ConversationSummaryMemory는 대화 진행 중 이전 대화 내용을 LLM으로 요약하고, 그 요약 내용을 메모리에 저장하는 방식입니다.

작동 방식

단계 1: 대화 발생 시 LLM이 이전 대화 기록 요약

단계 2: 요약된 내용을 메모리에 저장

단계 3: 다음 대화에 요약본을 포함하여 전달

장점

- ✓ 긴 대화 맥 context 유지: 핵심 정보를 보존
- ✓ 토큰 효율성: 요약본 사용으로 토큰 수 감소

코드 예시

```
from langchain.chat_models import ChatOpenAI
from langchain.memory import ConversationSummaryMemory
from langchain.chains import ConversationChain

summary_model = ChatOpenAI(
    model="gpt-4o-mini",
    temperature=0.0,
    openai_api_key=os.environ["OPEN_API_KEY"]
)

summary_prompt = ChatPromptTemplate.from_template(
    """다음 대화를 한국어로 간결하고 이해하기 쉽게 요약하세요.
    요약 시 아래를 반드시 지켜주세요:
    1. 중요한 정보와 맥락을 포함할 것.
    2. 불필요한 반복이나 사소한 내용은 제거할 것.
    3. 가능한 한 자연스러운 한국어 문장으로 작성할 것.
    대화 내용:
    {summary}
    새로운 메시지:
    {new_lines}
    위 내용을 바탕으로 최신 대화까지 포함하여 한국어로 요약하세요."""
)

summary_memory = ConversationSummaryMemory(
    llm=summary_model,
    # 메모리에 저장될 key 이름, 프롬프트에서 {summary}로 접근 가능
    memory_key="summary",
    # LM 출력 반환 형태 : False면 문자열 요약 반환, True면 AIMessage 객체로 반환
    return_messages=False,
    # 요약 생성용 커스텀 프롬프트
    prompt=summary_prompt
)
```

RunnableWithMessageHistory 활용

LCEL 기반의 최신 구현에서 대화 기록 자동 관리

Message-based memory 만 자동 저장하며, SummaryMemory는
수동으로 관리해야 합니다.

요약 메모리나 대화 히스토리를 기반으로 LLM에게 컨텍스트를 제공합니다.

세션별로 히스토리 관리가 가능합니다.

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_community.chat_message_histories import ChatMessageHistory
```

1. LLM 및 프롬프트 정의

```
model = ChatOpenAI(
    model="gpt-4o-mini",
    temperature=0.3,
    openai_api_key=os.environ["OPEN_API_KEY"]
)
```

```
chat_prompt = ChatPromptTemplate.from_messages([
    ("system",
        "당신은 친절한 AI 비서입니다 .\n"
        "요약된 대화와 최근 히스토리를 참고하여 답변하세요 ."),
    ("system", "요약된 대화:{summary}"),
    MessagesPlaceholder(variable_name="chat_history"),
    ("human", "{input}")
])
```

2. DB 기반 Chat History 로딩

```
def get_history(session_id: str):
    return SQLChatMessageHistory(
        connection_string=f"sqlite:///{{DB_FILE}}",
        session_id=session_id,
        table_name="message_store"
    )
```

3. RunnableWithMessageHistory로 체인에 기록 기능 추가

```
chain = chat_prompt | model | StrOutputParser()

chat_runnable = RunnableWithMessageHistory(
    runnable=chain,
    get_session_history=get_history,
    history_messages_key="chat_history", # MessagesPlaceholder
    input_messages_key="input",          # prompt의 {input}
    max_history=3
)
```

RunnableWithMessageHistory 활용

LCEL 기반의 최신 구현에서 대화 기록 자동 관리

1 프롬프트 구성

ChatPromptTemplate으로 시스템 메시지와
MessagesPlaceholder를 포함한 프롬프트 정의

2 세션 기록 함수 정의

session_id 기반 ChatMessageHistory 객체를 반환하는 함수 정의

3 체인 생성

정의된 프롬프트와 LLM을 연결하여 체인 생성

4 RunnableWithMessageHistory 생성

체인과 세션 기록 함수를 RunnableWithMessageHistory에 전달하여
대화 기록 관리 기능 통합

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_community.chat_message_histories import ChatMessageHistory
```

1. LLM 및 프롬프트 정의

```
model = ChatOpenAI(  
    model="gpt-4o-mini",  
    temperature=0.3,  
    openai_api_key=os.environ["OPEN_API_KEY"]  
)
```

```
chat_prompt = ChatPromptTemplate.from_messages([  
    ("system",  
        "당신은 친절한 AI 비서입니다 .\n",  
        "요약된 대화와 최근 히스토리를 참고하여 답변하세요 ."),  
    ("system", "요약된 대화:\n{summary}"),  
    MessagesPlaceholder(variable_name="chat_history"),  
    ("human", "{input}")  
])
```

2. DB 기반 Chat History 로딩

```
def get_history(session_id: str):  
    return SQLChatMessageHistory(  
        connection_string=f"sqlite:///{{DB_FILE}}",  
        session_id=session_id,  
        table_name="message_store"  
)
```

3. RunnableWithMessageHistory로 체인에 기록 기능 추가

```
chain = chat_prompt | model | StrOutputParser()
```

```
chat_runnable = RunnableWithMessageHistory(  
    runnable=chain,  
    get_session_history=get_history,  
    history_messages_key="chat_history", # MessagesPlaceholder  
    input_messages_key="input", # prompt의 {input}  
    max_history=3  
)
```

결론 및 활용 방향

★ Chat History와 Memory의 핵심 가치

LangChain의 Chat History와 Memory 기능은 LLM의 **상태 비저장(stateless)** 한계를 극복하여, 자연스럽고 맥락을 유지하는 대화형 AI 애플리케이션을 구현하는 데 필수적입니다.



Chat History

- ✓ 대화 메시지를 물리적으로 저장
- ✓ 인메모리부터 Redis, SQLite까지 다양한 저장소 옵션



Memory 유형

- ✓ ConversationBufferMemory: 모든 대화 기록 유지
- ✓ ConversationBufferWindowMemory: 최근 k개 대화만 기억
- ✓ ConversationSummaryMemory: LLM을 활용한 요약 관리



선택 가이드

- ✓ 애플리케이션 특성에 따라 적합한 Memory 유형 선택
- ✓ 데이터 지속성 요구사항 고려
- ✓ 토큰 및 비용 효율성 평가

LangChain의 Chat History와 Memory 기능을 적절히 활용하면, **LLM 기반 애플리케이션이 단순한 질문-응답을 넘어 실제 사람과의 대화처럼 자연스러운 상호작용을 가능하게 합니다.**