2025.11.07

LangChain

정보컴퓨터공학부 김명석

LangChain

CONTENTS

LangChain

LCEL(LangChain Expression Language)

오케스트레이션 솔루션, LangChain이 최적화된 방식으로 체인의 런타임 실행을 처리할 수 있게 해준다.

- 최적화된 병렬 실행
- observability와 debuggability를 제공
- 스트리밍 단순화
- 보장된 비동기 지원

```
from langchain_core.runnables import RunnableSequence
chain = RunnableSequence([runnable1, runnable2])

final_output = chain.invoke(some_input)
```

```
from langchain_core.runnables import RunnableParallel
chain = RunnableParallel({
    "key1": runnable1,
    "key2": runnable2,
})

final_output = chain.invoke(some_input)
```

```
rag_chain = (
    {
        "context": retriever | format_docs,
        "question": RunnablePassthrough()
    }
    | rag_prompt
    | llm
    | StrOutputParser()
)
```

LangChain

Runnable

LangChain 컴포넌트 작업의 기반이며, 언어 모델, Output Parser, Retrievers 등에서 구현

- **Invoked**: 단일 입력이 출력으로 변환된다.
- **Batched**: 여러 입력이 효율적으로 출력으로 변환된다.
- **Streamed**: 출력이 생성되는 대로 스트리밍된다.
- **Inspected**: Runnable의 입력, 출력 및 설정에 대한 스키마 정보를 액세스할 수 있다.
- **Composed**: LCEL을 사용하여 여러 Runnable을 함께 구성하여 복잡한 파이프라인을 만들 수 있다.

+ 여러 입력을 병렬로 처리할 수 있는 내장 API를 제공한다.

+ 모든 Runnable은 입력 및 출력 타입으로 특징지어지며, Runnable 자체에 의해 정의된다.

LangChain

__init__.py

```python
if TYPE_CHECKING:
    from langchain_core.runnables.base import (
        Runnable,
        RunnableBinding,
        RunnableGenerator,
        RunnableLambda,
        RunnableMap,
        RunnableParallel,
        RunnableSequence,
        RunnableSerializable,
        chain,
    )
    from langchain_core.runnables.branch import RunnableBranch
    from langchain_core.runnables.config import (
        RunnableConfig,
        ensure_config,
        get_config_list,
        patch_config,
        run_in_executor,
    )
    from langchain_core.runnables.fallbacks import RunnableWithFallbacks
    from langchain_core.runnables.history import RunnableWithMessageHistory
    from langchain_core.runnables.passthrough import (
        RunnableAssign,
        RunnablePassthrough,
        RunnablePick,
    )
    from langchain_core.runnables.router import RouterInput, RouterRunnable
    from langchain_core.runnables.utils import (
        AddableDict,
        ConfigurableField,
        ConfigurableFieldMultiOption,
        ConfigurableFieldSingleOption,
        ConfigurableFieldSpec,
        aadd,
        add,
    )
```

```python
__all__ = (
    "AddableDict",
    "ConfigurableField",
    "ConfigurableFieldMultiOption",
    "ConfigurableFieldSingleOption",
    "ConfigurableFieldSpec",
    "RouterInput",
    "RouterRunnable",
    "Runnable",
    "RunnableAssign",
    "RunnableBinding",
    "RunnableBranch",
    "RunnableConfig",
    "RunnableGenerator",
    "RunnableLambda",
    "RunnableMap",
    "RunnableParallel",
    "RunnablePassthrough",
    "RunnablePick",
    "RunnableSequence",
    "RunnableSerializable",
    "RunnableWithFallbacks",
    "RunnableWithMessageHistory",
    "aadd",
    "add",
    "chain",
    "ensure_config",
    "get_config_list",
    "patch_config",
    "run_in_executor",
)
```

```python
_dynamic_imports = {
    "chain": "base",
    "Runnable": "base",
    "RunnableBinding": "base",
    "RunnableGenerator": "base",
    "RunnableLambda": "base",
    "RunnableMap": "base",
    "RunnableParallel": "base",
    "RunnableSequence": "base",
    "RunnableSerializable": "base",
    "RunnableBranch": "branch",
    "RunnableConfig": "config",
    "ensure_config": "config",
    "get_config_list": "config",
    "patch_config": "config",
    "run_in_executor": "config",
    "RunnableWithFallbacks": "fallbacks",
    "RunnableWithMessageHistory": "history",
    "RunnableAssign": "passthrough",
    "RunnablePassthrough": "passthrough",
    "RunnablePick": "passthrough",
    "RouterInput": "router",
    "RouterRunnable": "router",
    "AddableDict": "utils",
    "ConfigurableField": "utils",
    "ConfigurableFieldMultiOption": "utils",
    "ConfigurableFieldSingleOption": "utils",
    "ConfigurableFieldSpec": "utils",
    "aadd": "utils",
    "add": "utils",
}
```

```python
def __getattr__(attr_name: str) -> object:
    module_name = _dynamic_imports.get(attr_name)
    result = import_attr(attr_name, module_name, __spec__.parent)
    globals()[attr_name] = result
    return result


def __dir__() -> list[str]:
    return list(__all__)
```

Facade Pattern

단일 인터페이스를 제공

Lazy Loading
→ Static Type Checking

컴파일 타임에
최적화를 하기 위한 방법과
자동 완성이나 타입 오류 검사

Public API Control

__all__과 __dir__ 정의

LangChain

Runnable

```python
class Runnable(ABC, Generic[Input, Output]):
    """A unit of work that can be invoked, batched, streamed, transformed and composed.
    ...
    """

    @abstractmethod
    def invoke( # line 725
        self,
        input: Input,
        config: Optional[RunnableConfig] = None,
        **kwargs: Any,
    ) -> Output:
        """Transform a single input into an output."""

    async def ainvoke( # line 849
        self,
        input: Input,
        config: Optional[RunnableConfig] = None,
        **kwargs: Any,
    ) -> Output:
        """Transform a single input into an output."""
        return await run_in_executor(config, self.invoke, input, config, **kwargs)

    def stream( # line 1136
        self,
        input: Input,
        config: Optional[RunnableConfig] = None,
        **kwargs: Optional[Any],
    ) -> Iterator[Output]:
        """Default implementation of ``stream``, which calls ``invoke``."""
        yield self.invoke(input, config, **kwargs)

    def __or__( # line 614
        self,
        other: Union[
            Runnable[Any, Other],
            ...
        ],
    ) -> RunnableSerializable[Input, Other]:
        """Runnable "or" operator."""
        return RunnableSequence(self, coerce_to_runnable(other))
```

- **ABC, Generic**

  모든 객체가 동일한 메서드를 갖도록 강제

  Duck Typing을 클래스 수준에서 보장

- **@abstractmethod**

  반드시 invoke 메서드를 직접 구현

- **yield**

  현재 값을 caller에게 양보하고, 그 자리에서 실행을 멈추고 일시 중지

- **run_in_executor**

- **연산자 오버로딩**

- **RunnableSerializable**

  실행할 수 있고(Runnable), 동시에 저장할 수도 있는(Serializable)

LangChain

RunnableSequence

```python
class RunnableSequence(RunnableSerializable[Input, Output]):
    ...
    def invoke(
        self, input: Input, config: Optional[RunnableConfig] = None, **kwargs: Any
    ) -> Output:
        ...
        input_ = input # 1. 최초 입력

        try:
            for i, step in enumerate(self.steps): # 2. 모든 단계를 순회
                ...
                # 3. 각 단계의 invoke를 순서대로 호출
                input_ = context.run(step.invoke, input_, config)
        except:
            pass
        return cast("Output", input_) # 4. 마지막 결과 반환
```

- 최초 입력을 input_ 변수에 저장
- self.steps(즉, [prompt, llm, parser])를 for 루프로 순회
- prompt.invoke(input_)를 호출하고, 그 결과를 input_에 덮어씀
- 다음 루프에서 llm.invoke(input_) (즉, prompt의 결과)를 호출하고, 그 결과를 다시 input_에 덮어씀
- 모든 루프가 끝나면 마지막 input_를 반환

LangChain

RunnableParallel

```python
class RunnableParallel(RunnableSerializable[Input, dict[str, Any]]):
    ...
    def invoke(
        self, input: Input, config: Optional[RunnableConfig] = None, **kwargs: Any
    ) -> dict[str, Any]:
        ...
        def _invoke_step(
            step: Runnable[Input, Any], input_: Input, config: RunnableConfig, key: str
        ) -> Any:
            ...
            return context.run(step.invoke, input_, child_config)

        try:
            steps = dict(self.steps__)
            # 1. 스레드 풀 실행기
            with get_executor_for_config(config) as executor:
                # 2. 모든 스텝을 병렬로 제출
                futures = [
                    executor.submit(_invoke_step, step, input, config, key)
                    for key, step in steps.items()
                ]
                # 3. 모든 결과를 dict로 취합
                output = {key: future.result() for key, future in zip(steps, futures)}
        except:
            pass
        return output
```

- RAG에서 **{"context": retriever, "question": passthrough}**처럼 dict를 쓰면, RunnableParallel 객체가 생성

- **get_executor_for_config**로 ThreadPoolExecutor를 가져옴
- **executor.submit**을 사용해 dict 안의 모든 Runnable(retriever, passthrough)의 invoke를 동시에 다른 스레드로 실행
- 모든 스레드가 완료되면(**future.result()**), 그 결과들을 다시 dict로 묶어 반환

- {}은 시스템적으로 모든 하위 작업을 개별 스레드에서 동시에 실행하고 결과를 취합하는 장치

LangChain

RunnableLambda

```python
class RunnableLambda(Runnable[Input, Output]):
    ...
    def invoke(
        self,
        input: Input,
        config: Optional[RunnableConfig] = None,
        **kwargs: Optional[Any],
    ) -> Output:
        if hasattr(self, "func"):
            # 1. 실제 실행은 _call_with_config 헬퍼에게 위임
            return self._call_with_config(
                self._invoke, # 2. _invoke 헬퍼가 실제 함수(self.func)를 호출
                input,
                ensure_config(config),
                **kwargs,
            )
        ...

    def _invoke(
        self,
        input_: Input,
        run_manager: CallbackManagerForChainRun,
        config: RunnableConfig,
        **kwargs: Any,
    ) -> Output:
        ...
        # 3. 우리가 정의한 실제 함수(self.func)가 여기서 호출됨
        output = call_func_with_variable_args(
            self.func, input_, config, run_manager, **kwargs
        )
        return cast("Output", output)
```

1. **invoke**가 호출되면, Runnable의 **_call_with_config**라는 헬퍼 함수를 호출
2. 이 헬퍼 함수가 **_invoke**를 호출하고,
3. **_invoke**가 드디어 self.func(우리가 만든 함수)를 실행

LangChain

_call_with_config

Aspect-Oriented Programming

핵심 비즈니스 로직과 로깅 같은 공통 부가 기능을 분리하여 코드의 모듈
성과 재사용성을 높이는 프로그래밍 패러다임

LangSmith는 바로 이 콜백을 수신하여 실행 과정을 시각화

ChatOpenAI 같은 모든 기본 Runnable의 invoke는 해당 헬퍼를 사용하도록 권장

```python
class Runnable(ABC, Generic[Input, Output]):
    ...
    def _call_with_config(
        self,
        func: ..., # 1. 실행할 실제 로직 (e.g., _invoke)
        input_: Input,
        config: Optional[RunnableConfig],
        # ...
    ) -> Output:
        ...
        # 2. "실행 시작" 콜백 (LangSmith)
        run_manager = callback_manager.on_chain_start(
            serialized,
            input_,
            name=config.get("run_name") or self.get_name(),
            # ...
        )
        try:
            # 3. 실제 로직(func) 실행
            output = cast(
                "Output",
                context.run(
                    call_func_with_variable_args,
                    func,
                    input_,
                    config,
                    run_manager,
                    **kwargs,
                ),
            )
        except BaseException as e:
            # 4. "실행 실패" 콜백
            run_manager.on_chain_error(e)
            raise
        else:
            # 5. "실행 완료" 콜백
            run_manager.on_chain_end(output)
            return output
```