

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

### 3. RunnableLambda

#### 1) RunnableLambda

- 사용자 정의 함수 실행 할 수 있는 기능
  - RunnableLambda → 자신만의 함수 정의, 실행 가능
  - 예시: 데이터 전처리, 계산, 외부 API와의 상호 작용 등

#### 2) 사용자 정의 함수 실행하는 방법

- 주의사항
  - RunnableLambda → 사용자 정의 함수를 래핑해서 활용 가능
  - 사용자 정의 함수가 받을 수 있는 인자 = only 1
    - 만약, 여러 인수를 받는 함수로 구현하고 싶다면?
    - 단일 입력을 받아들이고 이를 여러 인수로 풀어내는 래퍼를 작성해야 함

- 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"

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}')

```

- 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.

```

```

from operator import itemgetter

from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableLambda
from langchain_core.output_parsers import StrOutputParser
from langchain_google_genai import ChatGoogleGenerativeAI

from dotenv import load_dotenv
import os
# 3.6s

```

```
# 텍스트의 길이를 반환하는 함수
def length_function(text):
    return len(text)

# 두 텍스트의 길이를 곱하는 함수
def _multiple_length_function(text1, text2):
    return len(text1) * len(text2)

# 2개 인자를 받는 함수로 연결하는 wrapper 함수
# 딕셔너리에서 "text1"과 "text2"의 길이를 곱하는 함수
def multiple_length_function(
    _dict,
):
    return _multiple_length_function(_dict["text1"], _dict["text2"])
```

```
# 프롬프트 템플릿 생성
```

```
prompt = ChatPromptTemplate.from_template("what is {a} + {b}?")
```

```
print(type(prompt))          # <class 'langchain_core.prompts.chat.ChatPromptT
```

```
# 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 = 0
    max_output_tokens=4096,
)
```

- **LLM** 초기화

```
E0000 00:00:1759973579.086644 1654451 alts_credentials.cc:93] ALTS creds ignored
```

```
# 프롬프트와 모델을 연결하여 체인 생성
```

```
chain1 = prompt | gemini_lc
```

```
# 체인 구성
```

```
chain = (
    {
        "a": itemgetter("input_1") | RunnableLambda(length_function),
        "b": {"text1": itemgetter("input_1"), "text2": itemgetter("input_2")}
            | RunnableLambda(multiple_length_function),
    }
)
```

```

    | prompt
    | gemini_lc
    | StrOutputParser()
)

```

- **chain** 실행 → 결과 확인하기

# 주어진 인자들로 체인 실행하기

```
chain.invoke({"input_1": "bar", "input_2": "gah"})
```

- 주어진 인자들로 체인 실행하기 (1.3s)

```
'3 + 9 = 12'
```

### ✓ 3) **RunnableConfig** 인자로 활용

- **RunnableLambda** → (선택적으로) **RunnableConfig** 수용 가능
  - 콜백, 태그, 기타 구성 정보 → 중첩된 실행에 전달 가능
- **gemini** 모델은 응답에 토큰 정보를 포함
  - `get_openai_callback()` = **OpenAI** 전용 방법
  - **geimini** 방법
    - a. `usage_metadata` 직접 사용해보기
    - b. **Custom Callback Handler** 사용하기

```

from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnableLambda, RunnableConfig
import json

```

- **a** 시도 - **usage\_metadata()**

# a. 직접 호출 + `usage_metadata`

```

def parse_or_fix_with_tracking(text: str, config: RunnableConfig):
    fixing_prompt = ChatPromptTemplate.from_template(
        "Fix the following text:\n\ntext\n{input}\n\nError: {error}"
        " Don't narrate, just respond with the fixed data."
    )

```

```

# 토큰 사용량 누적
total_input_tokens = 0
total_output_tokens = 0

# 최대 3번 시도
for attempt in range(3):
    try:
        return json.loads(text)

    except Exception as e:
        # 프롬프트 생성
        messages = fixing_prompt.invoke({"input": text, "error": e})

        # ✨ LLM 직접 호출 (AIMessage 반환)
        ai_message = gemini_lc.invoke(messages, config=config)

        # ✨ 토큰 사용량 추출
        usage = ai_message.usage_metadata
        total_input_tokens += usage['input_tokens']
        total_output_tokens += usage['output_tokens']

        print(f"\n--- Attempt {attempt + 1} ---")
        print(f"Input tokens: {usage['input_tokens']}")
        print(f"Output tokens: {usage['output_tokens']}")
        print(f"Total tokens: {usage['total_tokens']}")

        # 수정된 텍스트
        text = ai_message.content

# 최종 토큰 사용량 출력
print(f"\n=== Final Token Usage ===")
print(f"Total Input tokens: {total_input_tokens}")
print(f"Total Output tokens: {total_output_tokens}")
print(f"Total tokens: {total_input_tokens + total_output_tokens}")

# 파싱 실패 시 → "Failed to parse" 문자열 반환
return "Failed to parse"

```

# a. 실행해보기

```

output = RunnableLambda(parse_or_fix_with_tracking).invoke(
    input="{foo:: bar}",
    config={"tags": ["my-tag"]},
)

print(f"\n\n수정한 결과:\n{output}")

```

- a방법 실행 결과 (0.8s)

```

--- Attempt 1 ---
Input tokens: 50
Output tokens: 6
Total tokens: 56

```

수정한 결과:

```
{'foo': 'bar'}
```

- **b** 시도 - ***Custom Callback Handler***

```
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnableLambda, RunnableConfig
from langchain_core.callbacks import BaseCallbackHandler
from typing import Any
import json

# Custom Callback Handler = 토큰 사용량 추적 핸들러

class TokenTrackingHandler(BaseCallbackHandler):
    """Gemini 토큰 사용량 추적 핸들러"""

    def __init__(self):
        self.total_input_tokens = 0
        self.total_output_tokens = 0
        self.total_tokens = 0
        self.call_count = 0

    def on_llm_end(self, response: Any, **kwargs: Any) -> None:
        """LLM 호출 완료 시 호출"""
        # Gemini의 usage_metadata 추출
        if hasattr(response, 'generations') and response.generations:
            generation = response.generations[0][0]
            if hasattr(generation, 'message') and hasattr(generation.message, 'usage_metadata'):
                usage = generation.message.usage_metadata

                self.total_input_tokens += usage.get('input_tokens', 0)
                self.total_output_tokens += usage.get('output_tokens', 0)
                self.total_tokens += usage.get('total_tokens', 0)
                self.call_count += 1

                print(f"\n[Call {self.call_count}] Tokens:")
                print(f"  Input: {usage.get('input_tokens', 0)}")
                print(f"  Output: {usage.get('output_tokens', 0)}")
                print(f"  Total: {usage.get('total_tokens', 0)}")

    def get_summary(self):
        """최종 요약"""
        return {
            "total_input_tokens": self.total_input_tokens,
            "total_output_tokens": self.total_output_tokens,
            "total_tokens": self.total_tokens,
            "total_calls": self.call_count,
        }
```

```

# parse_or_fix 함수

def parse_or_fix(text: str, config: RunnableConfig):
    fixing_chain = (
        ChatPromptTemplate.from_template(
            "Fix the following text:\n\ntext\n{input}\n\nError: {error}"
            " Don't narrate, just respond with the fixed data."
        )
        | gemini_lc
        | StrOutputParser()
    )

    # 최대 3번 시도
    for _ in range(3):
        try:
            return json.loads(text)

        except Exception as e:
            # 수정 체인 호출
            text = fixing_chain.invoke({"input": text, "error": e}, config)

    return "Failed to parse"

```

```

# b. 실행해보기 - Callback Handler 사용하기

token_handler = TokenTrackingHandler()

output = RunnableLambda(parse_or_fix).invoke(
    input="{foo:: bar}",
    config={
        "tags": ["my-tag"],
        "callbacks": [token_handler], # ✨ Custom Handler
    },
)

# 최종 결과
print(f"\n\n수정한 결과:\n{output}")

# 토큰 사용량 요약
print("\n=== Token Usage Summary ===")
summary = token_handler.get_summary()
for key, value in summary.items():
    print(f"{key}: {value}")

```

- b 실행 결과 (1.0s)

```

[Call 1] Tokens:
  Input: 50
  Output: 6
  Total: 56

```

수정한 결과:

```
{'foo': 'bar'}
```

```
=== Token Usage Summary ===
total_input_tokens: 50
total_output_tokens: 6
total_tokens: 56
total_calls: 1
```

- 📊 a 방법 vs b 방법

방법	장점	단점	추천
a (직접 <code>usage_metadata</code> )	간단, 직관적	Chain 내부 호출 추적 어려움	★★★★★
b (Custom Callback)	정교한 추적, 모든 호출 캐치	구현 복잡	★★★★☆

- 📊 OpenAI callback vs gemini callback

- OpenAI 전용 콜백 함수 존재

```
from langchain.callbacks import get_openai_callback # ✅
```

- gemini 전용 콜백 함수 존재하지 않음

```
from langchain.callbacks import get_gemini_callback # ❌ 존재하지 않음
```

- next: 04. LLM 체인 라우팅 (`RunnableLambda`, `RunnableBranch`)