1. Langchain-OutputParser输出解析器

1.1 OutputParser输出解析器是什么?

Langchain中的OutputParser是用于解析LLM（大语言模型）返回的文本结果的组件。

因为大模型通常会返回非结构化的自然语言，而在实际应用中，我们需要从这些文本中提取出特定的信息并进行后续处理。

> “非结构化的自然语言”时，我们指的是由人以自由形式编写的文字，这些文字没有遵循任何特定的数据格式或规则。这种类型的文本信息包含了人类日常交流中使用的所有复杂性和变化性，包括但不限于语法、语义、上下文含义等。由于缺乏预定义的格式，这类文本对于计算机处理来说往往比较困难。

因此，OutputParser的作用就是将模型的输出解析成可编程的结构化数据，如列表、字典或特定的Python对象。

假设有一个大语言模型被用来回答用户的问题：“北京的天气怎么样？”模型可能返回以下的回答：

- “北京今天天气晴朗，气温适中，建议外出活动。”
- “目前北京的天气非常好，阳光明媚，温度大约在20度左右。”

这两个回答虽然提供了相似的信息，但是表达方式不同，而且没有遵循固定的格式。如果应用程序需要从中提取具体的天气数据（比如温度、天气状况），就需要一个OutputParser来解析这段文本，并将其转换为结构化的格式，例如JSON对象：

```
{
  "city": "北京",
  "weather": "晴朗",
  "temperature": "20度",
  "suggestion": "建议外出活动"
}
```

通过OutputParser，开发者可以方便地将LLM生成的内容转换为更适合编程的形式，使得与大模型的交互更加高效且可靠。

1.2 Langchain使用OutputParser

1.2.1 列表解析器

列表解析器用于将LLM输出的文本解析为Python列表格式。

比如，当LLM返回一个包含项目的自然语言段落时，我们可以通过列表解析器将其转换为Python的list类型，以便于后续处理。

```
from langchain_openai import ChatOpenAI
# 列表解析器
from langchain.output_parsers import
CommaSeparatedListOutputParser
from langchain_core.prompts import PromptTemplate

chat_model = ChatOpenAI(
    openai_api_key="EMPTY",
    base_url="http://127.0.0.1:10222/v1",
    model="Qwen2___5-7B-Instruct"
)

# 解析大模型的输出为逗号分隔的列表Comma Separated List
parser = CommaSeparatedListOutputParser()
prompt = PromptTemplate(
    template="回答用户查询。\n{format_instructions}\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions":
parser.get_format_instructions()},  # 获取解析器提示词
)
print(f"parser.get_format_instructions():
{parser.get_format_instructions()}")
formatted_prompt = prompt.format(query="列出5种水果。")

print(f"formatted_prompt: {formatted_prompt}")

# 生成响应
response = chat_model.invoke(formatted_prompt).content
print(response)
# 解析响应
output = parser.parse(response)

# 打印输出结果
```

```
print(output)
```

`partial_variables` 是一个字典，其中键是模板中变量的名称，值是这些变量的固定值。这些变量在模板渲染时会被自动替换，而不需要在每次调用 `format` 方法时传递它们。

> formatted_prompt发现，其实是修改了prompt给到大模型，让大模型能够理解prompt并输出。
>
> 跳转到 `CommaSeparatedListOutputParser` 下的 `parse()` 发现就是将用逗号输出的结果使用split解析成列表。

```
156     def parse(self, text: str) -> list[str]:
157         """Parse the output of an LLM call.
158 
159         Args:
160             text: The output of an LLM call.
161 
162         Returns:
163             A list of strings.
164         """
165         return [part.strip() for part in text.split(",")]
```

1.2.2 datetime解析器

datetime解析器用于解析包含日期和时间的LLM输出。它可以将LLM返回的时间字符串转化为Python的datetime对象，从而便于进一步的时间计算和操作。

```
from langchain_openai import ChatOpenAI
# 时间解析器
from langchain.output_parsers import DatetimeOutputParser
from langchain_core.prompts import PromptTemplate

chat_model = ChatOpenAI(
    openai_api_key="EMPTY",
    base_url="http://127.0.0.1:10222/v1",
    model="Qwen2___5-7B-Instruct"
)

# 将LLM调用的输出解析为日期时间。
parser = DatetimeOutputParser()
prompt = PromptTemplate(
    template="回答用户查询。\n{format_instructions}\n{query}\n",
```

```
    input_variables=["query"],
    partial_variables={"format_instructions":
parser.get_format_instructions()},  # 获取解析器提示词
)
print(f"parser.get_format_instructions():
{parser.get_format_instructions()}")
formatted_prompt = prompt.format(query="北京奥运会是什么时候开幕的？")

print(f"formatted_prompt: {formatted_prompt}")

# 生成响应
response = chat_model.invoke(formatted_prompt).content
print("------------------------")
print(response)
# 解析响应
output = parser.parse(response)
print("------------------------")
# 打印输出结果
print(output)
```

> formatted_prompt发现，其实是修改了prompt给到大模型，让大模型能够理解prompt并输出。
>
> 跳转到 `DatetimeOutputParser` 下的 `parse()` 发现就是 `datetime.strptime`。

1.2.3 枚举解析器

枚举解析器用于将LLM的输出解析为预定义的枚举值。它通常用于限定LLM输出的范围，并确保返回的结果符合特定的枚举选项。

```
from langchain_openai import ChatOpenAI
# 枚举解析器
from langchain.output_parsers import EnumOutputParser
from langchain_core.prompts import PromptTemplate
from enum import Enum


class Color(Enum):
    RED = "red"
    BLUE = "blue"
    YELLOW = "yellow"
```

```
chat_model = ChatOpenAI(
    openai_api_key="sk-
talrfpdubittuoctscpxqyuhotkkqgcmuxrxxlmmhkqpzlxd",
    base_url="https://api.siliconflow.cn/v1",
    model="Qwen/Qwen2.5-7B-Instruct"
)

# 将LLM调用的输出解析为日期时间。
parser = EnumOutputParser(enum=Color)
prompt = PromptTemplate(
    # template="回答用户查询。\n{format_instructions}\n{query}\n",
    template="回答用户查询。请直接输出颜色名称，如 'red', 'blue', 或
'yellow'。\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions":
parser.get_format_instructions()},  # 获取解析器提示词
)
print(f"parser.get_format_instructions():
{parser.get_format_instructions()}")
formatted_prompt = prompt.format(query="香蕉是什么颜色的？")

print(f"formatted_prompt: {formatted_prompt}")

# 生成响应
response = chat_model.invoke(formatted_prompt).content
print("------------------------")
print(response)
# 解析响应
output = parser.parse(response)
print("------------------------")
# 打印输出结果
print(output)
```

1.2.4 结构化输出解析器

结构化输出解析器用于解析具有特定结构的LLM输出，如JSON、CSV等。它可以预定义输出的结构，并确保LLM的输出能够符合该结构化格式。

```
from langchain_openai import ChatOpenAI
from langchain.output_parsers import StructuredOutputParser,
ResponseSchema
```

```
from langchain_core.prompts import PromptTemplate

# 初始化ChatOpenAI实例
chat_model = ChatOpenAI(
    openai_api_key="sk-talrfpdubittuoctscpxqyuhotkkqgcmuxrxxlmmhkqpzlxd",  # 替换为你的API密钥
    base_url="https://api.siliconflow.cn/v1",
    model="Qwen/Qwen2.5-7B-Instruct"
)

# 定义响应模式
response_schemas = [
    ResponseSchema(name="event_name", description="事件名称"),
    ResponseSchema(name="date", description="日期，格式为YYYY-MM-DD"),
]

# 创建结构化输出解析器
parser = StructuredOutputParser(response_schemas=response_schemas)

# 创建提示模板
prompt = PromptTemplate(
    template="回答用户查询。\n{format_instructions}\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions": parser.get_format_instructions()}
)

# 输出格式说明
print(f"parser.get_format_instructions(): {parser.get_format_instructions()}")

# 格式化提示
formatted_prompt = prompt.format(query="北京奥运会是什么时候开幕的？")

print("==============================")
# 输出格式化后的提示
print(f"formatted_prompt: {formatted_prompt}")
print("==============================")
# 调用模型生成响应
response = chat_model.invoke(formatted_prompt).content
```

```
print("------------------------")
print(response)

# 解析模型响应
output = parser.parse(response)
print("------------------------")
# 输出解析结果
print(output)
```

1.2.5 Pydantic（JSON）解析器

Pydantic解析器基于Pydantic库，用于将LLM的输出解析为JSON格式并转换为Pydantic模型。这种解析器特别适合需要将输出转化为特定的Python数据模型的场景，如处理API响应、数据库记录等。

假设有一个语言模型，该模型生成的输出是一个 JSON 字符串，表示用户的个人信息。使用 `PydanticOutputParser` 来解析这个 JSON 字符串并验证其格式。

当模型输出为：`'{"id": 1, "name": "Alice", "email": "alice@example.com"}'` 时，被解析后：`id=1 name='Alice' email='alice@example.com' is_active=True`。

注意：Pydantic解析可能会出错

```
from langchain_openai import ChatOpenAI
from langchain.output_parsers import PydanticOutputParser
from langchain_core.prompts import PromptTemplate
from pydantic import BaseModel, Field

# 定义输出模型
class EventDetails(BaseModel):
    event_name: str = Field(description="事件名称")
    date: str = Field(description="日期，格式为YYYY-MM-DD")

# 初始化ChatOpenAI实例
chat_model = ChatOpenAI(
    openai_api_key="sk-
talrfpdubittuoctscpxqyuhotkkqgcmuxrxxlmmhkqpzlxd",
    base_url="https://api.siliconflow.cn/v1",
    model="Qwen/Qwen2.5-7B-Instruct"
)
```

```
# 创建Pydantic输出解析器
parser = PydanticOutputParser(pydantic_object=EventDetails)

# 创建提示模板
prompt = PromptTemplate(
    template="回答用户查询。\n{format_instructions}\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions":
parser.get_format_instructions()}
)

# 输出格式说明
# print(f"parser.get_format_instructions():
{parser.get_format_instructions()}")

# 格式化提示
formatted_prompt = prompt.format(query="北京奥运会是什么时候开幕的？")

# 输出格式化后的提示
print(f"formatted_prompt: {formatted_prompt}")

# 调用模型生成响应
response = chat_model.invoke(formatted_prompt).content
print("------------------------")
print(response)

# 解析模型响应
try:
    output = parser.parse(response)
    print("------------------------")
    # 输出解析结果
    print(output)
except Exception as e:
    print(f"解析错误: {e}")
```


1.2.6 自动修复解析器

自动修复解析器用于在LLM输出不完全符合预期格式时，自动尝试修复输出，并返回可解析的结果。

- **核心功能**：尝试修复错误输出，使其符合预期格式，而不是完全重新生成。

- **工作机制**：
  1. 接收模型的初始输出。
  2. 如果输出格式不正确或不完全符合要求，使用一个“修复工具”（通常是另一个语言模型或自定义函数）对输出进行修正。
  3. 修复工具尝试尽可能从错误输出中提取有效信息并生成符合要求的格式。
- **适用场景**：
  - 输出有一定的错误或不完整，但可以通过后处理来纠正。
  - 希望尽量避免完全重新生成，以节省生成时间或避免丢失有效信息。
- **优点**：
  - 更适合处理轻微错误（如格式问题、小的字段缺失等）。
  - 修复过程通常比重新生成更快。

```
from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field

# 使用Pydantic创建一个数据格式
class Person(BaseModel):
    name: str = Field(description="姓名")
    age: int = Field(description="年龄")


# 定义一个格式不正确的输出:Python期望属性名称被双引号包围，但在给定的JSON字符串中是单引号。
misformatted = "{'name': 'John', 'age': 30}"

# 创建一个用于解析输出的Pydantic解析器
parser = PydanticOutputParser(pydantic_object=Person)

# parser.parse(misformatted) 会产生异常json.decoder.JSONDecodeError: Expecting property name enclosed in double quotes: line 1 column 2 (char 1)

# 从langchain库导入所需的模块
from langchain.output_parsers import OutputFixingParser

# 使用OutputFixingParser创建一个新的解析器，该解析器能够纠正格式不正确的输出
new_parser = OutputFixingParser.from_llm(parser=parser, llm=local_llm)

# 使用新的解析器解析不正确的输出
```

```
result = new_parser.parse(misformatted)
print(result)
```

`fixing_parser.parse` 函数为：

```
def parse(self, completion: str) -> T:
    retries = 0
    while retries <= self.max_retries:
        try:
            # 使用基础解析器 self.parser 尝试解析 completion。如果解析成功，直接返回解析结果。如果解析失败，捕获 OutputParserException 异常。
            return self.parser.parse(completion)
        except OutputParserException as e:
            # 如果当前重试次数已经达到最大重试次数 self.max_retries，则重新抛出捕获的异常 e。
            if retries == self.max_retries:
                raise e
            else:
                # 否则，增加重试计数器 retries。
                retries += 1
                # 如果 self.legacy 为 True 并且 self.retry_chain 有 run 方法，调用 run 方法尝试修复输出。
                if self.legacy and hasattr(self.retry_chain, "run"):
                    completion = self.retry_chain.run(

 instructions=self.parser.get_format_instructions(),
                        completion=completion,
                        error=repr(e),
                    )
                # 调用 invoke 方法尝试修复输出。如果 self.parser 没有 get_format_instructions 方法，调用 invoke 方法时不传递 instructions 参数。
                else:
                    try:
                        completion = self.retry_chain.invoke(
                            dict(

 instructions=self.parser.get_format_instructions(),
                                completion=completion,
                                error=repr(e),
                            )
```

```
                    )
                except (NotImplementedError, AttributeError):
                    # Case: self.parser does not have
get_format_instructions
                    completion = self.retry_chain.invoke(
                        dict(
                            completion=completion,
                            error=repr(e),
                        )
                    )
        raise OutputParserException("Failed to parse")
```

主要逻辑是：

1. 尝试使用基础解析器解析模型输出。
2. 如果解析失败，记录错误并尝试修复输出。
3. 如果修复后仍然解析失败，继续重试，直到达到最大重试次数。
4. 如果所有重试都失败，最终抛出解析失败的异常。

1.2.7 重试解析器(需要用强一点的模型)

重试解析器用于在第一次解析失败时，自动重新尝试解析LLM输出。它可以配置重新解析的次数，并在每次失败时根据不同的策略调整解析方式。

- **核心功能**：当生成模型返回的输出不符合预期格式时，自动重新尝试生成新的输出。
- **工作机制**：
  1. 接收模型的初始输出。
  2. 如果初始输出解析失败（例如，不符合JSON格式或预期的schema），则触发重新生成。
  3. 重新生成过程会传递错误信息到生成模型，让它改正之前的错误并生成符合预期的输出。
- **适用场景**：
  - 当模型的输出需要严格遵循特定格式（如JSON、特定字段结构）且希望利用模型本身的能力来修正错误时。
  - 对错误的上下文信息敏感，例如希望模型知道**为什么之前的输出不符合要求**。
- **优点**：
  - 能动态利用错误提示来提高生成的准确性。

- 支持多次重试。

```
from langchain_openai import ChatOpenAI
from langchain.output_parsers import PydanticOutputParser,
OutputFixingParser, RetryWithErrorOutputParser
from langchain_core.prompts import PromptTemplate
from pydantic import BaseModel, Field

# 创建 OpenAI 聊天模型
chat_model = ChatOpenAI(
    openai_api_key="sk-
talrfpdubittuoctscpxqyuhotkkqgcmuxrxxlmmhkqpzlxd",
    base_url="https://api.siliconflow.cn/v1",
    model="Qwen/Qwen2.5-7B-Instruct"
)

# 定义一个 Action 类
class Action(BaseModel):
    action: str = Field(description="要采取的操作")
    action_input: str = Field(description="操作的输入")


# 定义 PydanticOutputParser
parser = PydanticOutputParser(pydantic_object=Action)

prompt = PromptTemplate(
    template="回答用户查询。\n{format_instructions}\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions":
parser.get_format_instructions()},
)

prompt_value = prompt.format_prompt(query="黑悟空是谁？")
# 定义一个错误格式的字符串： 提供action字段,没有提供action_input字段，与
Action数据格式的预期不符，解析会失败
bad_response = '{"action": "search"}'

# 创建一个 OutputFixingParser，用于尝试修复错误格式的响应
fix_parser = OutputFixingParser.from_llm(parser=parser,
llm=chat_model, max_retries=1)

# 尝试修复并解析
```

```
try:
    parse_result = fix_parser.parse(bad_response)
    print('OutputFixingParser的parse结果:', parse_result)
except Exception as e:
    print('OutputFixingParser解析时发生错误:', str(e))


# 创建 RetryWithErrorOutputParser 包装基本解析器，并设置重试次数
# 功能：在解析失败时，重新运行模型并传递错误信息，以期模型能够生成更符合预期的输出。
# 适用场景：适用于需要重新生成整个输出的情况，特别是在初始输出完全不正确或无法修复的情况下。
retry_parser = RetryWithErrorOutputParser.from_llm(
    parser=parser,
    llm=chat_model,  # 将 LLM 指定为修复和重试的模型
    max_retries=1  # 设置最大重试次数
)
# 使用 retry_parser 进行重试解析
try:
    retry_parse_result =
retry_parser.parse_with_prompt(bad_response, prompt_value)
    print('RetryWithErrorOutputParser的parse结果:',
retry_parse_result)
except Exception as e:
    print('RetryWithErrorOutputParser解析时发生错误:', str(e))
```

这段代码用于比较 `OutputFixingParser` 和 `RetryWithErrorOutputParser` 在处理错误输入时的表现，并测试两者在修复不符合预期格式的响应时的能力：

1. 对比目标

- `OutputFixingParser` 的目标是修复现有的错误响应。
  - 它会尝试用语言模型（LLM）基于提供的错误响应补充或修改内容，使其符合预期格式。
  - 适用于部分错误的修复，例如缺少字段、数据格式不符等场景。
- `RetryWithErrorOutputParser` 的目标是重试生成一个全新的响应。
  - 它不是修复，而是重新生成整个输出，同时将原始的响应内容和错误信息作为提示传递给 LLM。
  - 适用于初始响应完全错误、难以直接修复的情况。

通过代码，你可以清晰地对比两者在以下方面的差异：

- **修复机制**（修改 vs 重新生成）。
- **对错误输入的容错能力**。
- **适用场景**（轻微错误 vs 严重错误）。

2. 测试输入设计合理

- 使用 `bad_response = '{"action": "search"}'` 作为测试输入，这个输入：
  - 不符合 `Pydantic` 定义的 `Action` 模型格式（缺少 `action_input` 字段）。
  - 是典型的解析失败案例，能够很好地验证两种解析器的表现。
  - 对于 `OutputFixingParser` 来说，能测试它是否能够通过 LLM 补全缺失字段。
  - 对于 `RetryWithErrorOutputParser` 来说，能测试它是否能基于错误信息重新生成一个正确响应。

3. 功能验证全面

- `OutputFixingParser`：
  - 检查其是否能在解析失败时修复响应。
  - 验证其是否能生成符合 `Pydantic` 模型格式的修复结果。
- `RetryWithErrorOutputParser`：
  - 检查其是否能基于错误信息和提示重试生成一个全新响应。
  - 验证其对解析失败的处理能力，以及能否有效利用错误上下文提示。

主要区别

特性	RetryWithErrorOutputParser	OutputFixingParser
修复方式	提供了错误信息给修复链（LLM），并将原始错误信息与 `completion` 一起传递给修复链	通过修复链尝试修复输出，可能不提供原始错误信息，仅重新生成输出
错误信息传递	将原始错误信息（`error=repr(e)`）传递给修复链，理论上帮助模型理解如何修复	仅在解析失败时尝试修复输出，不一定传递错误信息
修复链的输入	修复链接收 `completion`、`prompt` 和 `error` 作为输入，提供更多上下文信息	修复链只使用 `completion`，重点在于修复输出
主要差异	强调将错误信息反馈给模型，让模型了解失败原因，以便更精确地修复问题	更侧重于根据已有的 `completion` 尝试修复，错误信息的使用不如前者明确
适用场景	适合需要提供详细错误上下文以帮助修复的场景，如模型可能需要理解错误的原因	更适合用于轻微修复，如格式错误、少量缺失等，不需要提供复杂的错误上下文
重试链的功能实现	在修复过程中传递更多信息（`completion` 和 `error`），并且可能使用提示模板来重新生成修复输出	基于基础解析器重试，并通过修复链尝试解决格式问题
方法名称	使用 `parse_with_prompt` 和 `aparse_with_prompt`，明确使用了 `prompt` 作为输入的一部分	使用 `parse` 和 `aparse`，没有明确的 `prompt` 输入