LangChain3 模型调用和输出解析

调用模型

调用大语言模型有两种方式，一种是通过HuggingFace的transformers库。首先导入分词器和预训练的模型，然后将用户问题转成pytorch张量。使用generate方法生成回复，使用分词器的decode方法将数字转回文本。

```
# 导入必要的库
from transformers import AutoTokenizer, AutoModelForCausalLM
# 加载预训练模型的分词器
tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-2-7b-chat-hf")
# 加载预训练的模型
# 使用 device_map 参数将模型自动加载到可用的硬件设备上，例如GPU
model = AutoModelForCausalLM.from_pretrained(
          "meta-llama/Llama-2-7b-chat-hf",
          device_map = 'auto')
# 定义一个提示，希望模型基于此提示生成故事
prompt = "请给我讲个玫瑰的爱情故事?"
# 使用分词器将提示转化为模型可以理解的格式，并将其移动到GPU上
inputs = tokenizer(prompt, return_tensors="pt").to("cuda")
# 使用模型生成文本，设置最大生成令牌数为2000
outputs = model.generate(inputs["input_ids"], max_new_tokens=2000)
# 将生成的令牌解码成文本，并跳过任何特殊的令牌，例如[CLS], [SEP]等
response = tokenizer.decode(outputs[0], skip_special_tokens=True)
# 打印生成的响应
print(response)
```

另外一种方法是通过langchain，其中又有HuggingFace Hub和HuggingFace Pipeline两种。HuggingFace Hub不推荐，因为有一些模型不支持通过HuggingFace Hub调用。

```
# 导入HuggingFace API Token
import os
os.environ['HUGGINGFACEHUB_API_TOKEN'] = '你的HuggingFace API Token'
# 导入必要的库
from langchain import PromptTemplate, HuggingFaceHub, LLMChain
# 初始化HF LLM
llm = HuggingFaceHub(
    repo_id="google/flan-t5-small",
)
# 创建简单的question-answering提示模板
```

```
template = """Question: {question}
              Answer: """
# 创建Prompt
prompt = PromptTemplate(template=template, input_variables=["question"])
# 调用LLM Chain --- 我们以后会详细讲LLM Chain
llm_chain = LLMChain(
    prompt=prompt,
    llm=llm
)
# 准备问题
question = "Rose is which type of flower?"
# 调用模型并返回结果
print(llm_chain.run(question))
```

HuggingFace Pipeline允许指明任务类型、模型精度等参数。

序号	参数	含义
1	"text-generation"	这个参数告诉 Pipeline 我们希望执行的任务是文本生成。Transformers 库支持多种 NLP 任务， 包括但不限于文本分类（"text-classification"）、问答（"question-answering"）、摘要（"summarization"）等。
2	model	指定了我们想要加载的预训练模型的名称或路径。
3	torch_dtype=torch.float16	这个参数是用来设置计算的数据类型的。在这里，我们设置为 torch.float16（也被称为半精度浮点数）， 这样可以减少内存的使用并提高计算速度，但可能会牺牲一些数值的精度。 这对于大模型和在 GPU 上运行的情况尤其有用。
4	device_map="auto"	这个参数用来设置运行模型的设备。"auto" 意味着 Pipeline 会自动选择可用的设备。 如果有 GPU 可用，它会优先选择 GPU，否则就使用 CPU。
5	max_length = 1000	这个参数用来设定生成文本的最大长度。如果模型生成的文本超过这个长度，那么超出部分的文本将被截断。 这个参数可以防止模型生成过长的输出，以避免消耗资源。

```
# 指定预训练模型的名称
model = "meta-llama/Llama-2-7b-chat-hf"
# 创建一个文本生成的管道
import transformers
import torch
pipeline = transformers.pipeline(
    "text-generation",
    model=model,
    torch_dtype=torch.float16,
    device_map="auto",
    max_length = 1000
)
# 创建HuggingFacePipeline实例
from langchain import HuggingFacePipeline
llm = HuggingFacePipeline(pipeline = pipeline,
                          model_kwargs = {'temperature':0})
# 定义输入模板，该模板用于生成花束的描述
```

```
template = """
                为以下的花束生成一个详细且吸引人的描述：
                花束的详细信息：
                ```{flower_details}```
            """
# 使用模板创建提示
from langchain import PromptTemplate,  LLMChain
prompt = PromptTemplate(template=template,
                        input_variables=["flower_details"])
# 创建LLMChain实例
from langchain import PromptTemplate
llm_chain = LLMChain(prompt=prompt, llm=llm)
# 需要生成描述的花束的详细信息
flower_details = "12支红玫瑰，搭配白色满天星和绿叶，包装在浪漫的红色纸中。"
# 打印生成的花束描述
print(llm_chain.run(flower_details))
```

输出解析

输出解析器将模型输出的文本转化成结构化信息。输出解析器类通常需要实现两个核心方法：

- get_format_instructions：这个方法需要返回一个字符串，用于指导如何格式化语言模型的输出，告诉它应该如何组织并构建它的回答。
- parse：这个方法接收一个字符串（也就是语言模型的输出）并将其解析为特定的数据结构或格式。这一步通常用于确保模型的输出符合我们的预期，并且能够以我们需要的形式进行后续处理。

还有一个可选的方法。

- parse_with_prompt：这个方法接收一个字符串（也就是语言模型的输出）和一个提示（用于生成这个输出的提示），并将其解析为特定的数据结构。这样，你可以根据原始提示来修正或重新解析模型的输出，确保输出的信息更加准确和贴合要求。

Langchain定义了多种输出解析器：

- 列表、日期、枚举解析器
- 结构化输出解析器StructuredOutputParser：生成一定特定结构的复杂回答。
- JSON解析器：将模型的输出转化为符合特定格式的JSON对象。
- 自动修复解析器：修复某些常见的模型输出格式错误、语法错误。
- 重试解析器： 在模型的初次输出不符合预期时，尝试修复或重新生成新的输出，针对输入输出不完整的问题。

Json解析器

使用负责数据格式验证的Pydantic库来创建带有类型注解的类FlowerDescription，它可以自动验证输入数据，确保输入数据符合你指定的类型和其他验证条件。具有以下特点：
```

1. 数据验证：当你向Pydantic类赋值时，它会自动进行数据验证。例如，如果你创建了一个字段需要是整数，但试图向它赋予一个字符串，Pydantic会引发异常。
2. 数据转换：Pydantic不仅进行数据验证，还可以进行数据转换。例如，如果你有一个需要整数的字段，但你提供了一个可以转换为整数的字符串，如 `"42"`，Pydantic会自动将这个字符串转换为整数42。
3. 易于使用：创建一个Pydantic类就像定义一个普通的Python类一样简单。只需要使用Python的类型注解功能，即可在类定义中指定每个字段的类型。
4. JSON支持：Pydantic类可以很容易地从JSON数据创建，并可以将类的数据转换为JSON格式。

```python
from pydantic import BaseModel, Field
class FlowerDescription(BaseModel):
    flower_type: str = Field(description="鲜花的种类")
    price: int = Field(description="鲜花的价格")
    description: str = Field(description="鲜花的描述文案")
    reason: str = Field(description="为什么要这样写这个文案")
```

使用PydanticOutputParser创建了输出解析器，该解析器将用于解析模型的输出，以确保其符合FlowerDescription的格式。

```python
# ------Part 3
# 创建输出解析器
from langchain.output_parsers import PydanticOutputParser
output_parser = PydanticOutputParser(pydantic_object=FlowerDescription)

# 获取输出格式指示
format_instructions = output_parser.get_format_instructions()
# 打印提示
print("输出格式：",format_instructions)
# ------Part 4
# 创建提示模板
from langchain import PromptTemplate
prompt_template = """您是一位专业的鲜花店文案撰写员。
对于售价为 {price} 元的 {flower} ，您能提供一个吸引人的简短中文描述吗？
{format_instructions}"""

# 根据模板创建提示，同时在提示中加入输出解析器的说明
prompt = PromptTemplate.from_template(prompt_template,
        partial_variables={"format_instructions": format_instructions})

# 打印提示
print("提示：", prompt)
```

```
flowers = ["玫瑰", "百合", "康乃馨"]
prices = ["50", "30", "20"]
# ------Part 5
for flower, price in zip(flowers, prices):
    # 根据提示准备模型的输入
    input = prompt.format(flower=flower, price=price)
    # 打印提示
    print("提示: ", input)

    # 获取模型的输出
    output = model(input)

    # 解析模型的输出
    parsed_output = output_parser.parse(output)
    parsed_output_dict = parsed_output.dict()  # 将Pydantic格式转换为字典

```

自动修复解析器

错误的代码会生成引发OutputParserException错误

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

# 使用Pydantic创建一个数据格式，表示花
class Flower(BaseModel):
    name: str = Field(description="name of a flower")
    colors: List[str] = Field(description="the colors of this flower")
# 定义一个用于获取某种花的颜色列表的查询
flower_query = "Generate the charaters for a random flower."

# 定义一个格式不正确的输出
misformatted = "{'name': '康乃馨', 'colors': ['粉红色','白色','红色','紫色','黄色']}"

# 创建一个用于解析输出的Pydantic解析器，此处希望解析为Flower格式
parser = PydanticOutputParser(pydantic_object=Flower)
# 使用Pydantic解析器解析不正确的输出
parser.parse(misformatted)
```

在OutputFixingParser内部，调用了原有的PydanticOutputParser，如果成功，就返回；如果失败，它会将格式错误的输出以及格式化的指令传递给大模型，并要求LLM进行相关的修复。

```
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=ChatOpenAI())

# 使用新的解析器解析不正确的输出
result = new_parser.parse(misformatted) # 错误被自动修正
print(result) # 打印解析后的输出结果
```

重试解析器

由于bad_response只提供了action字段，而没有提供action_input字段，这与Action数据格式的预期不符，所以解析会失败。

```
# 定义一个模板字符串，这个模板将用于生成提问
template = """Based on the user question, provide an Action and Action Input
for what step should be taken.
{format_instructions}
Question: {query}
Response:"""

# 定义一个Pydantic数据格式，它描述了一个"行动"类及其属性
from pydantic import BaseModel, Field
class Action(BaseModel):
    action: str = Field(description="action to take")
    action_input: str = Field(description="input to the action")

# 使用Pydantic格式Action来初始化一个输出解析器
from langchain.output_parsers import PydanticOutputParser
parser = PydanticOutputParser(pydantic_object=Action)

# 定义一个提示模板，它将用于向模型提问
from langchain.prompts import PromptTemplate
prompt = PromptTemplate(
    template="Answer the user query.\n{format_instructions}\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions":
parser.get_format_instructions()},
)
prompt_value = prompt.format_prompt(query="What are the colors of Orchid?")

# 定义一个错误格式的字符串
```

```
bad_response = '{"action": "search"}'
parser.parse(bad_response) # 如果直接解析，它会引发一个错误

```

根据传入的原始提示，还原了action_input字段的内容。

```
# 初始化RetryWithErrorOutputParser，它会尝试再次提问来得到一个正确的输出
from langchain.output_parsers import RetryWithErrorOutputParser
from langchain.llms import OpenAI
retry_parser = RetryWithErrorOutputParser.from_llm(
    parser=parser, llm=OpenAI(temperature=0)
)
parse_result = retry_parser.parse_with_prompt(bad_response, prompt_value)
print('RetryWithErrorOutputParser的parse结果:',parse_result)

```