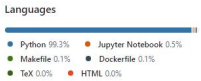
LLM 与 LangChain 概述

LangChain 概览

https://github.com/hwchase17/langchain

99.3%的都是 python 代码:



总共 800 多个文件：

```
Administrator@ZA19030001 MINGW64 /d/wujianjun/code/my/zacode/langchain/langchain-0.0.215/langchain
$ find . -name "*.py" | wc -l
818
```

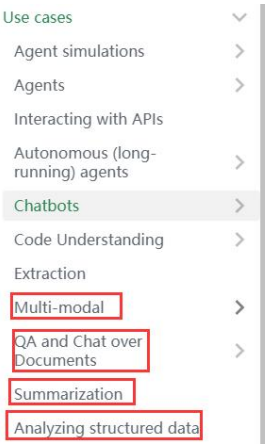
总共约 10 万行代码：

```
Administrator@ZA19030001 MINGW64 /d/wujianjun/code/my/zacode/langchain/langchain-0.0.215/langchain
$ find . -name "*.py" | xargs cat|wc -l
95770
```

There are six main areas that LangChain is designed to help with. These are:

- LLMs and Prompts: this includes **prompt management**, **prompt optimization**, a generic interface for all LLMs, and common utilities for working with LLMs.
- Chains: chains go beyond a single LLM call and involve sequences of calls (whether to an LLM or a different utility).
- **Data Augmented Generation: this involves specific types of chains that first interact with an external data source to fetch data for use in the generation step. Examples include summarization of long pieces of text and question/answering over specific data sources.**
- Agents: this involves an LLM making decisions about which Actions to take, taking that Action, seeing an Observation, and repeating that until done.
- Memory: this refers to persisting state between calls of a chain/agent.

langchain 官方提供了很多精彩的例子如下：



LangChain：2023 年最潮大语言模型 Web 开发框架

https://www.infoq.cn/article/yl8eJoSfkHbOCyFzCcgw

LangChain 目前的主要应用是在 LLM(尤其是 ChatGPT)之上构建基于聊天的应用程序。Chase 说目前 LangChain 最好的一个应用场景是"基于文档的聊天"。

LangChain 引入代理(agent)。**你可以通过代理来选择要使用的工具和针对工具的输入。然后你执行它，得到结果，再把结果反馈到语言模型中。然后继续这样做，直到满足停止条件。LLM 通常没有长期记忆。默认情况下，LLM 是无状态的—这意味着每一个查询的处理都是独立于其他查询进行的。**LangChain 将记忆等组件添加到 LLM 的处理过程中。

Chase 提到的另一种形式的代理是 Auto-GPT，这为代理和工具之间的交互提供了长期记忆，并使用检索向量作为存储。

微软推出了自己的工具 Semantic Kernel，功能与 LangChain 类似。

LlamaIndex

https://github.com/jerryjliu/llama_index

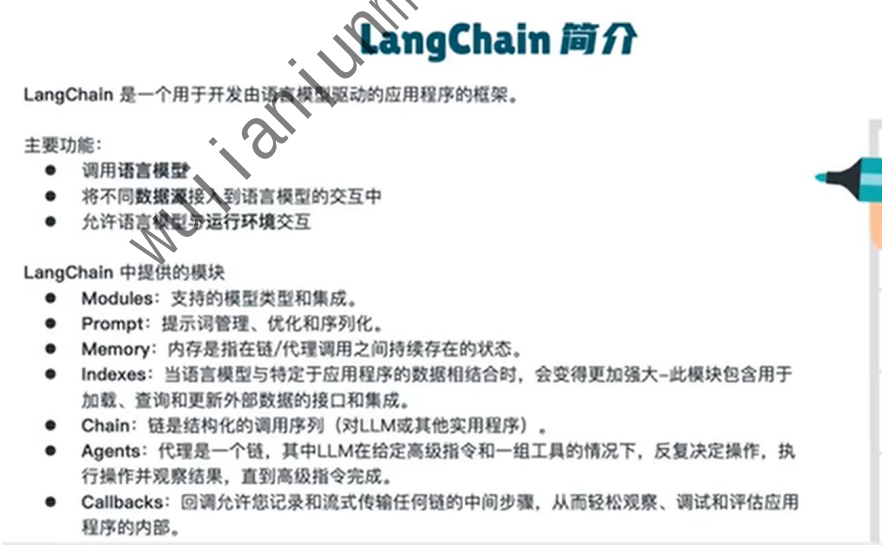
Semantic Kernel

https://github.com/microsoft/semantic-kernel

langchain-ChatGLM

https://github.com/imClumsyPanda/langchain-ChatGLM

基于 langchain 的 ChatGLM 应用，实现基于可扩展知识库的问答



hello world

https://github.com/hwchase17/langchain

https://python.langchain.com/docs/get_started/introduction.html

Installation

To install LangChain run: `pip install langchain`， this will install the bare minimum requirements of LangChain. A lot of the value of LangChain comes when integrating it with various model providers, datastores, etc. By default, the dependencies needed to do that are NOT installed. To install modules needed for the common LLM providers, run: `pip install langchain[llms]`。 To install all modules needed for all integrations, run: `pip install langchain[all]`。

Quickstart

Using LangChain will usually require integrations with one or more model providers, data stores, APIs, etc. we'll use OpenAI's model APIs. First we'll need to install their Python package: `pip install openai`. **Accessing the API requires an API key. (注意，国内访问不了，我们这里使用了代理，下面利用 os 模块设置了代理的地址)**

The basic building block of LangChain is the LLM, which takes in text and generates more text. suppose we're building an application that generates a company name based on a company description. we need to initialize OpenAI **model** with a HIGH temperature(since we want the outputs to be MORE random). we can pass in text and get predictions!

```
>>> import os
>>> os.environ["OPENAI_API_BASE"] = "https://openai.wndbac.cn/v1"
>>> os.environ["OPENAI_API_KEY"] = "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"
>>> from langchain.llms import OpenAI
>>> llm = OpenAI(temperature=0.9)
>>> llm.predict("一家做婚姻中介的公司叫什么名字比较好？")
'\n\n《缘分时光》、《缘聚臻爱》、《天路寻缘》、《缘缘引领》、《奔向新生活》等。'
>>>
```

While chat models use language models under the hood, the interface they expose is a bit different: rather than expose a "text in, text out" API, they expose an interface where "chat messages" are the inputs and outputs. You can get chat completions by passing one or more messages to the chat model. The response will be a message. **The types of messages currently supported in LangChain are AIMessage, HumanMessage, SystemMessage, and ChatMessage -- ChatMessage takes in an arbitrary role parameter.**

```
>>> import os
>>> os.environ["OPENAI_API_BASE"] = "https://openai.wndbac.cn/v1"
>>> os.environ["OPENAI_API_KEY"] = "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"
>>> from langchain.chat_models import ChatOpenAI
>>> from langchain.schema import ( AIMessage, HumanMessage, SystemMessage)
>>>
>>> chat = ChatOpenAI(temperature=0)
>>> chat.predict_messages([HumanMessage(content="把今天天气阴沉，我心情也随之忧郁起来，这句话翻译成英文")])
AIMessage(content='The weather is gloomy today, and my mood has become melancholic as well.', additional_kwargs={}, example=False)
```

LangChain makes that easy by exposing an interface through which you can interact with a chat model as you would a normal LLM. You can access this through the *predict* interface.

Most LLM applications do not pass user input directly into to an LLM. Usually they will add the user input to a larger piece of text, called a *prompt template*, that provides additional context.

```
>>> from langchain.prompts import PromptTemplate
>>> prompt = PromptTemplate.from_template("What is a good name for a company that makes {product}?")
>>> prompt.format(product="colorful socks")
'What is a good name for a company that makes colorful socks?'
```

Now that we've got a model and a prompt template, we'll want to combine the two. Chains give us a way to link together multiple primitives, like models, prompts, and other chains.

```
from langchain.chains import LLMChain

chain = LLMChain(llm=llm, prompt=prompt)
chain.run("colorful socks")
# Feetful of Fun
```

Understanding how this chain works will help you with working with more complex chains.

Our first chain ran a pre-determined sequence of steps. To handle complex workflows, we need to be able to dynamically choose actions based on inputs. Agents do just this: they use a language model to determine which actions to take and in what order. **Agents are given access to tools, and they repeatedly choose a tool, run the tool, and observe the output until they come up with a final answer.** To load an agent, you need to choose a(n):

- LLM/Chat model: The language model powering the agent.
- **Tool(s): A function that performs a specific duty. This can be things like: Google Search, Database lookup, Python REPL, other chains.**
- Agent name: A string that references a supported agent class. If you want to implement a custom agent, see here.

For this example, we'll be using SerpAPI to query a search engine. You'll need to install the SerpAPI Python package: `pip install google-search-results`.

```
>>> import os
>>> os.environ["OPENAI_API_BASE"] = "https://openai.wndbac.cn/v1"
>>> os.environ["OPENAI_API_KEY"] = "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"
>>> os.environ["SERPAPI_API_KEY"] = "196cea7cf5345c1ac0193aec58a89ad08be54cab751a5bebf074054a5be7f18f"
>>>
>>> from langchain.agents import AgentType, initialize_agent, load_tools
>>> from langchain.llms import OpenAI
>>> llm = OpenAI(temperature=0)
>>> tools = load_tools(["serpapi", "llm-math"], llm=llm) # The tools we'll give the Agent access to. Note that the 'llm-math' tool uses an LLM
>>> agent = initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
>>>
>>> agent.run("去年中国的GDP是多少人民币？按最新汇率算，是多少美元呢？")


> Entering new  chain...
 I need to find the GDP of China in RMB and then convert it to USD.
Action: Search
Action Input: "China GDP 2019 RMB"
Observation: RMB 99.08651 trillion
Thought: I need to convert this to USD
Action: Calculator
Action Input: 99.08651 trillion RMB to USD
Observation: Answer: 13872111400000.002
Thought: I now know the final answer
Final Answer: China's GDP in 2019 was 99.08651 trillion RMB, which is equivalent to 13872111400000.002 USD.

> Finished chain.
"China's GDP in 2019 was 99.08651 trillion RMB, which is equivalent to 13872111400000.002 USD."
>>>
```

The chains and agents we've looked at so far have been stateless, but for many applications it's necessary to reference past interactions. for example, you want chatbot to understand new messages in the context of past messages. The Memory module gives you a way to maintain application state. The base Memory interface is simple: it lets you update state given the latest inputs and outputs and it lets you modify (or contextualize) the next input using the stored state. There are a number of built-in memory systems. The simplest of these are is a buffer memory which just prepends the last few inputs/outputs to the current input.

```
>>> import os
>>> os.environ["OPENAI_API_BASE"] = "https://openai.wndbac.cn/v1"
>>> os.environ["OPENAI_API_KEY"] = "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"
>>> os.environ["SERPAPI_API_KEY"] = "196cea7cf5345c1ac0193aec58a89ad08be54cab751a5bebf074054a5be7f18f"
>>> from langchain import OpenAI, ConversationChain
>>>
>>> llm = OpenAI(temperature=0)
>>> conversation = ConversationChain(llm=llm, verbose=True)
>>> conversation.run("我姓吴，家里刚生了个女宝宝，你帮我起个名字吧？")


> Entering new  chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details f

Current conversation:

Human: 我姓吴，家里刚生了个女宝宝，你帮我起个名字吧？
AI:

> Finished chain.
' 嗨，我姓吴！很高兴认识你！让我想想，你家新生的女宝宝可以取什么名字呢？我知道一些古老的中国名字，比如说，可以取“玉”、“珍”、“瑞
>>> conversation.run("今年是兔年，名字中带点兔的寓意吧")


> Entering new  chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details f

Current conversation:
Human: 我姓吴，家里刚生了个女宝宝，你帮我起个名字吧？
AI:  嗨，我姓吴！很高兴认识你！让我想想，你家新生的女宝宝可以取什么名字呢？我知道一些古老的中国名字，比如说，可以取“玉”、“珍”、
Human: 今年是兔年，名字中带点兔的寓意吧
AI:

> Finished chain.
' 好的，那么你家宝宝可以取“兔”字开头的名字，比如说“兔兰”、“兔珊”、“兔玉”、“兔璐”、“兔瑶”、“兔环”等等。你可以根据你的喜好来选择一
>>>
```

langchain 例子

使用 langchain 打造自己的大型语言模型(LLMs)

https://blog.csdn.net/weixin_42608414/article/details/129493302

Openai 所学习到的是截止到 2021 年 9 月以前的知识，当询问机器人之后的事情时，它无法给出正确的答案，另外用户向机器人提问的字符串(prompt)长度被限制在 4096 个 token(token 可以看作是一种词语单位)。

我们希望大型语言模型学习特定领域内的知识，这些知识可能是几十个文本文件或者是关系型数据库。我们想要 LLMs 学习用户给定的知识，并且只回答给定数据范围内的相关问题，超出则一律告知用户问题超出范围无法回答。我们的知识三个新闻，存储在 3 个文本文件中：

- 2022 世界杯.txt
- 埃隆·马斯克收购推特案.txt
- 安倍晋三遇刺案.txt

由于中文的特殊性，需要分词，这里我们会用的一个分词工具：jieba。

我们首先将这 3 个时事新闻的文本文件放置到 Data 文件夹下面，然后在 data 文件夹下面再建一个子文件夹:cut, 用来存放被分词过的文档：

```
# 文档预处理
files=['2022世界杯.txt','埃隆·马斯克收购推特案.txt','安倍晋三遇刺案.txt']
for file in files:
    my_file=f"./data/{file}"
    with open(my_file,"r",encoding='utf-8') as f:
        data = f.read()
    cut_data = " ".join([w for w in list(jb.cut(data))])  #对中文文档进行分词处理
    #分词处理后的文档保存到data文件夹中的cut子文件夹中
    cut_file=f"./data/cut/cut_{file}"
    with open(cut_file, 'w') as f:
        f.write(cut_data)
```

接下来我们按照 LangChain 的流程来处理这些经过分词后的数据，首先是加载文档,然后要对文档进行切块处理，切块处理完成以后需要调用 openai 的 Embeddings 方法进行向量化操作。

```
#加载文档
loader = DirectoryLoader('./data/cut', glob='**/*.txt')
docs = loader.load()
#文档切块
text_splitter = TokenTextSplitter(chunk_size=1000, chunk_overlap=0)
doc_texts = text_splitter.split_documents(docs)
#调用openai Embeddings
embeddings = OpenAIEmbeddings()

#向量化
vectordb = Chroma.from_documents(doc_texts, embeddings, persist_directory="./data/cut")
vectordb.persist()

#创建聊天机器人对象chain
chain = ChatVectorDBChain.from_llm(OpenAI(temperature=0, model_name="gpt-3.5-turbo"),
                                   vectordb, return_source_documents=True)
```

接下来我们要创建一个聊天函数，用来让机器人回答用户提出的问题，这里我们让机器人每次只针对当前问题进行回答，并没有将历史聊天记录保存起来一起喂给机器人。

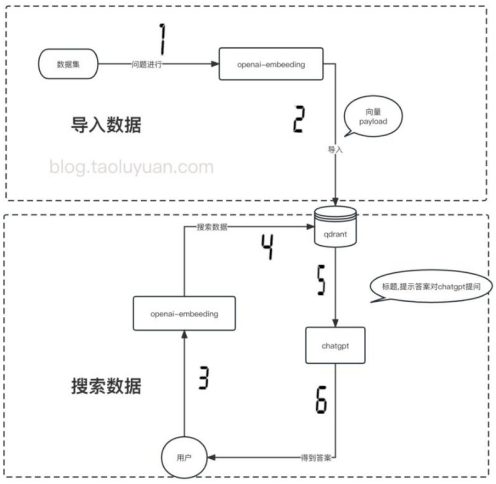
```
def get_answer(question):
  chat_history = []
  result = chain({"question": question, "chat_history": chat_history});
  return result["answer"]
```

下面是我和机器人之间就 2022 年 3 个时事新闻进行针对性聊天的内容：

使用 golang 基于 OpenAI Embedding + qdrant 实现 k8s 本地知识库

https://blog.taoluyuan.com/posts/embedding-openai/

整个流程如下：



1.将数据集 通过 openai embedding 得到向量+组装 payload,存入 qdrant

2.用户问题通过 openai embedding 得到向量,从 qdrant 中搜索相似度大于 0.8 的数据

3.从 qdrant 中取出相似度高的数据

4.将获取到的 QA,组装成 prompt 向 chatgpt 进行提问,得到回答

qdrant 是一个开源的向量搜索引擎。collection 这个概念类似于 mysql 中的 database。

```
PUT /collections/{collection_name}
{
    "name": "example_collection",
    "vectors": {
      "size": 300,
      "distance": "Cosine"
    }
}
```

name 是名称，vectors 指明向量的配置：size 是向量的维度，distance 是距离计算方法。

如果需要将 openai embedding 后存入 qdrant，需要将 size 设置为 1536。

官网 http add point 的例子如下：

```
curl -L -X PUT 'http://localhost:6333/collections/test_collection/points?wait=true' \
    -H 'Content-Type: application/json' \
    --data-raw '{
        "points": [
          {"id": 1, "vector": [0.05, 0.61, 0.76, 0.74], "payload": {"city": "Berlin" }},
          {"id": 2, "vector": [0.19, 0.81, 0.75, 0.11], "payload": {"city": ["Berlin", "London"] }},
          {"id": 3, "vector": [0.36, 0.55, 0.47, 0.94], "payload": {"city": ["Berlin", "Moscow"] }},
          {"id": 4, "vector": [0.18, 0.01, 0.85, 0.80], "payload": {"city": ["London", "Moscow"] }},
          {"id": 5, "vector": [0.24, 0.18, 0.22, 0.44], "payload": {"count": [0] }},
          {"id": 6, "vector": [0.35, 0.08, 0.11, 0.44]}
        ]
    }'
```

- id:唯一
- **vector:向量,可在 HuggingFace 找相应的模型训练,获取,也可以 openai embedding 得到**
- payload:任意的自定义 json 数据,这个数据可以用于后续的过滤。

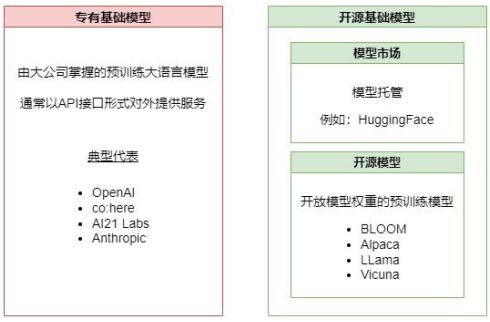
这是 qdrant 官方搜索数据的例子：

```
curl -L -X POST 'http://localhost:6333/collections/test_collection/points/search' \
    -H 'Content-Type: application/json' \
    --data-raw '{
        "vector": [0.2,0.1,0.9,0.7],
        "limit": 3
    }'
```

用 LangChain 构建大语言模型应用

https://developer.aliyun.com/article/1221923

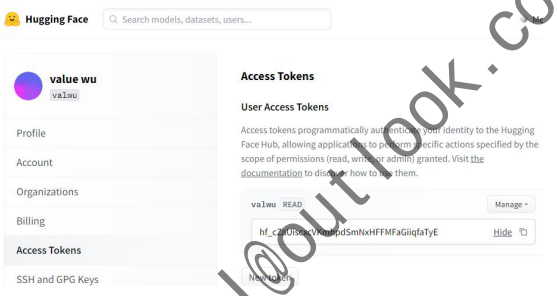
使用 LLM 构建应用程序需要您提供调用服务的 API 密钥，并且某些 API 会产生相关费用。



大部分 LangChain 教程都使用 OpenAI(需要 Key 且可将 Key 保存到环境变量中，供各模块用：

```
import os
os.environ["OPENAI_API_KEY"] = ... # 填入OpenAI Secret Key
```

许多开源模型都托管在 Hugging Face 上。要获取 Hugging Face API 密钥，您需要一个 Hugging Face 帐户并在 Access Tokens 下创建 token。



同样，你可以将 Hugging Face Access Token 保存在环境变量中供其他模块使用：

```
import os
os.environ["HUGGINGFACEHUB_API_TOKEN"] = "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"
```

如果你想使用特定的向量数据库，如 Pinecone、Weaviate 或 Milvus，你需要单独注册来获得 API 密钥。我们使用无需注册的 Faiss。

LangChain 区分三种输入和输出不同的模型：

- LLMs 接受一个字符串作为输入（prompt）输出一个字符串

```
>>> llm.predict("1到100的求和是多少？")
'\n\n5050'
>>>
>>> llm("1到100的求和是多少？")
'\n\n答案：5050'
```

- 接受文本输入并返回 embeddig

```
>>> from langchain.embeddings import OpenAIEmbeddings
>>> embeddings = OpenAIEmbeddings()
>>> text = u"中国的首都是？"
>>> embeddings.embed_query(text)
[0.021435962989926338, -0.008240651339292526, -0.019228186458349
28675414621829987, -0.011263507418334484, 0.004438014235347509,
0939427874982357, -0.003337335539981723, -0.01418367587029934, 0
```

在大语言模型使用中，好的提示是珍贵的。一旦你有了一个好的提示，你可以将它保存成模板供以后复用。LangChain 提供了 PromptTemplates，它可以从多个组件构建提示。

```
>>> from langchain import PromptTemplate
>>> template = u"请为宠物{animal}起一个好听的名字。"
>>> prompt = PromptTemplate(input_variables=["animal"], template=template,)
>>> prompt.format(animal="猫")
'请为宠物猫起一个好听的名字。'
```

上面的提示是零样本问题设置。还可以在提示中添加一些示例，即小样本问题设置!!!!!!!!!

```python
from langchain import PromptTemplate, FewShotPromptTemplate

examples = [
    { "word": "高兴", "antonym": "悲伤"},
    { "word": "高大", "antonym": "矮小"},
]

example_template = """词语: {word}, 反义词: {antonym}"""

example_prompt = PromptTemplate(
    input_variables=["word", "antonym"],
    template=example_template,
)

few_shot_prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    prefix="给出输入词语的反义词",
    suffix="词语: {input}, 反义词:",
    input_variables=["input"],
    example_separator="\n",
)
output1 = few_shot_prompt.format(input="美丽")
print(output1)
print("============================")
output2 = few_shot_prompt.format(input="高兴")
print(output2)
```

上面的代码将生成一个提示模板，并根据提供的示例和输入组成以下提示：

```
(base) python t.py
给出输入词语的反义词
词语: 高兴, 反义词: 悲伤
词语: 高大, 反义词: 矮小
词语: 美丽, 反义词:
============================
给出输入词语的反义词
词语: 高兴, 反义词: 悲伤
词语: 高大, 反义词: 矮小
词语: 高兴, 反义词:
```

Chains 描述了将大语言模型与其他组件相结合，比如使用第一个 LLM 的输出作为第二个 LLM 的输入，我们可以使用 SimpleSequentialChain：

```python
from langchain.llms import OpenAI
from langchain import PromptTemplate
from langchain.chains import LLMChain, SimpleSequentialChain

llm = OpenAI(temperature=0.1)

prompt = PromptTemplate....
chain = LLMChain(llm = llm, prompt = prompt)

second_prompt = PromptTemplate....
chain_two = LLMChain(llm=llm, prompt=second_prompt)

overall_chain = SimpleSequentialChain(chains=[chain, chain_two], verbose=True) # 将两个chain串联在一起

catchphrase = overall_chain.run("猫")
```

Indexes

LLM 的一个局限是缺乏上下文信息。可以借助外部数据来解决这个问题。你需要使用文档加载器加载外部数据。我们从文本文件加载开始。

```python
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter

loader = TextLoader('../state_of_the_union.txt', encoding='utf8')
documents = loader.load()
# 如果文本很大，可以使用 CharacterTextSplitter 对文档进行分割：
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(documents)
```

chunk_size 表示块的最大大小，chunk_overlap 表示块之间的最大重叠。有一些重叠可以很好地保持块之间的连续性(如做一个滑动窗口)。最终切分好的文本在 texts 元组中。

接着你可以使用向量数据库中的文本嵌入模型对其进行索引。我们这里使用 Faiss。

```
from langchain.vectorstores import FAISS
# 将文本以词嵌入的形式保存到向量数据库
db = FAISS.from_documents(texts[0], embeddings) # 创建 vectorestore 用作索引
```

下面我们将用这些外部数据做一个带有检索功能的问答机器人：

```
from langchain.chains import RetrievalQA

retriever = db.as_retriever()
qa = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=retriever,
    return_source_documents=True)

query = u"中国古典四大名著是什么?"
result = qa({"query": query})
print(result['result'])
```

上面的代码会用输入问题从数据库中检索最相近的答案。

对于像聊天机器人这样的应用程序，它们必须能够记住以前的对话。但默认情况下，LLM 没有任何长期记忆，除非你将聊天记录输入给它。ConversationChain 可以提供会话记忆。

```
>>> from langchain import ConversationChain
>>> llm = OpenAI(temperature=0.1)
>>> conversation = ConversationChain(llm=llm, verbose=True)
>>> conversation.predict(input="现在我们假设有一种动物，叫红蓝鸭，好吗？")


> Entering new  chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative

Current conversation:

Human: 现在我们假设有一种动物，叫红蓝鸭，好吗？
AI:

> Finished chain.
' 好的！我知道红蓝鸭是一种什么样的动物？'
>>> conversation.predict(input="红蓝鸭浑身是红色的，但是眼睛是蓝色。")


> Entering new  chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative

Current conversation:
Human: 现在我们假设有一种动物，叫红蓝鸭，好吗？
AI:  好的！我知道红蓝鸭是一种什么样的动物？
Human: 红蓝鸭浑身是红色的，但是眼睛是蓝色。
AI:

> Finished chain.
' 哦，原来红蓝鸭是这样的！我知道它们有什么特别的特征吗？'
>>> conversation.predict(input="现在有一只叫做Bob的红蓝鸭，请问它的眼睛是什么颜色？")


> Entering new  chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative

Current conversation:
Human: 现在我们假设有一种动物，叫红蓝鸭，好吗？
AI:  好的！我知道红蓝鸭是一种什么样的动物？
Human: 红蓝鸭浑身是红色的，但是眼睛是蓝色。
AI:  哦，原来红蓝鸭是这样的！我知道它们有什么特别的特征吗？
Human: 现在有一只叫做Bob的红蓝鸭，请问它的眼睛是什么颜色？
AI:

> Finished chain.
' Bob的红蓝鸭的眼睛是蓝色的。'
```

下面 Agent 先使用维基百科查找奥巴马的出生日期，然后使用计算器算他在 2023 年的年龄。

```
from langchain.llms import OpenAI
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType

llm = OpenAI(temperature=0.1)

tools = load_tools(["wikipedia", "llm-math"], llm=llm)
agent = initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)

agent.run("奥巴马的生日是哪天？到2023年他多少岁了?")
```

Hugging Face 使用方法

1、登录注册

参考下面的链接

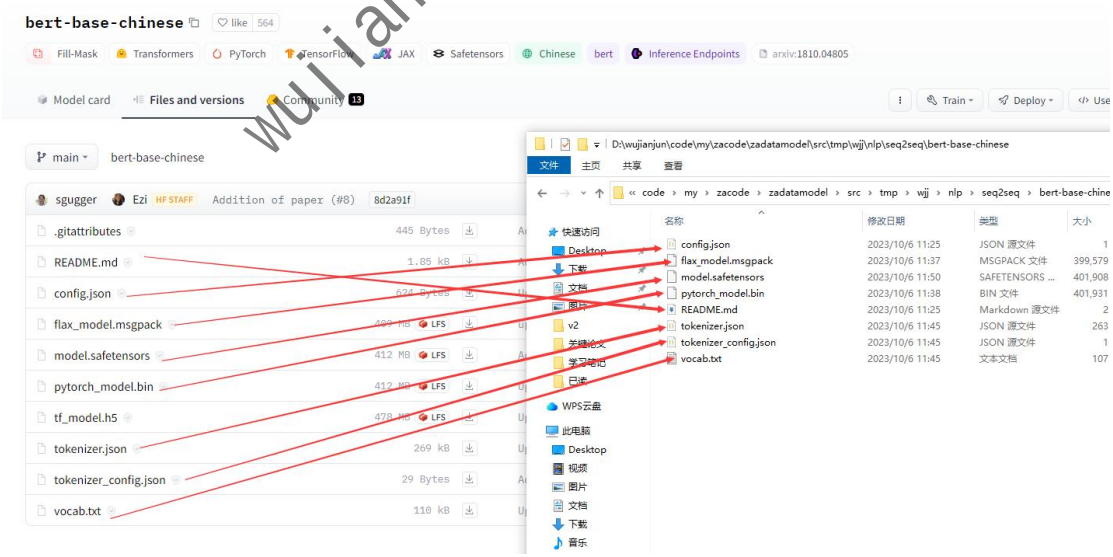
http://121.199.45.168:13008/05_mkdocs_translearning/10%20huggingface%E5%B9%B3%E5%8F%B0%E4%BD%BF%E7%94%A8%E6%8C%87%E5%8D%97.html

2、寻找模型

3、下载模型

点击下面每个下载链接，逐一下载:

注意，因为我们使用 pytorch，所以不必下载 tf_model.h5 这个文件。



4、然后本地使用一下

huggingface transformers 预训练模型如何下载至本地，并使用？
https://zhuanlan.zhihu.com/p/147144376

下载 huggingface-transformers 模型至本地，并使用 from_pretrained 方法加载
https://blog.csdn.net/weixin_44612221/article/details/129884741

HuggingFace 快速上手（以 bert-base-chinese 为例）
https://zhuanlan.zhihu.com/p/610171544

huggingface clone 报错

LangChain 中文入门教程

https://github.com/liaokongVFX/LangChain-Chinese-Getting-Started-Guide

LangChain 与开源大模型的融合

LangChain 大模型应用落地实践：使用 LLMs 模块接入自定义大模型，以 ChatGLM 为例

https://zhuanlan.zhihu.com/p/624240080

2023 年 4 月，OpenAI API 进一步封锁国内用户的使用。因此，我们的目光陆续转移到国内的大模型。清华大学发布的 ChatGLM-6B 是比较平民的大模型版本。虽然 OpenAI API 的使用已经越来越困难，但我还是循例写一下这部分的接入。在 OpenAI 官网注册获得对应的 key。访问如果需要代理，可以通过 openai 包进行配置，代码如下所示：

```python
import os

from langchain.schema import HumanMessage
from langchain.chat_models import ChatOpenAI


# global environment
os.environ["OPENAI_API_KEY"] = "sk-**********"

# llm initialization
llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)

while True:
    human_input = input("(human): ")
    human_input = [HumanMessage(content=human_input)]
    ai_output = llm(human_input)
    print(f"(ai): {ai_output.content}")
```

```python
import os

import openai

from langchain.schema import HumanMessage
from langchain.chat_models import ChatOpenAI

openai.proxy = {
    "http": "{proxy_ip}:{port}"
}

# global environment
os.environ["OPENAI_API_KEY"] = "sk-**********"

# llm initialization
llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)

while True:
    human_input = input("(human): ")
    human_input = [HumanMessage(content=human_input)]
    ai_output = llm(human_input)
    print(f"(ai): {ai_output.content}")
```

首先复制 ChatGLM 的 requirements.txt，https://github.com/THUDM/ChatGLM-6B/blob/main/requirements.txt，安装依赖：`pip install -r requirements.txt`。 然后从 huggingface_hub 下载模型文件：

从 Hugging Face Hub 下载模型需要先安装Git LFS，然后运行

```
git clone https://huggingface.co/THUDM/chatglm-6b
```

```
(base) du -sh *
25G     chatglm-6b
24G     chatglm-6b.tar
```

接着我们可以通过下列代码在本地测试 ChatGLM：

```python
from transformers import AutoTokenizer, AutoModel
tokenizer = AutoTokenizer.from_pretrained("/data/dm/chatglm-6b/chatglm-6b", trust_remote_code=True)
model = AutoModel.from_pretrained("/data/dm/chatglm-6b/chatglm-6b", trust_remote_code=True).half().cuda()
model = model.eval()
response, history = model.chat(tokenizer, "[假设a=1,b=2,c=a+b, c等于多少", history=[])
print(response)
response, history = model.chat(tokenizer, "d=2, e=d*c,那么e是多少？", history=history)
print(response)
```

运行后可以得到模型的下列输出：

```
ned above are installed properly.
/opt/anaconda3/lib/python3.9/site-packages/scipy/__init__.py:146: UserWarning: A NumPy version >=
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
根据给出的方程 c = a + b，将 a = 1,b = 2 代入，得到 c = 1 + 2 = 3。因此，c等于3。
根据题目中的方程 d = 2，代入 e = d * c，得到 e = 2 * 3 = 6。因此，e等于6。
```

可以基于 Flask 搭建模型服务，如下：

```python
import os
import json
from flask import Flask
from flask import request
from transformers import AutoTokenizer, AutoModel

tokenizer = AutoTokenizer.from_pretrained("THUDM/chatglm-6b", trust_remote_code=True)
model = AutoModel.from_pretrained("THUDM/chatglm-6b", trust_remote_code=True).half().cuda()
model.eval()

app = Flask(__name__)

@app.route("/chat", methods=["POST"])
def chat():
    data_seq = request.get_data()
    data_dict = json.loads(data_seq)
    human_input = data_dict["human_input"]

    response, _ = model.chat(tokenizer, human_input, history=[])

    result_dict = { "response": response}
    result_seq = json.dumps(result_dict, ensure_ascii=False)
    return result_seq

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8595, debug=False)
```

ChatGLM-6B

https://github.com/THUDM/ChatGLM-6B

首次使用

注意，按照官方说明，下载模型，这个很难，模型文件很大，且网络可能别屏蔽了。

```
(base) du -sh *
25G     chatglm-6b
24G     chatglm-6b.tar
(base)
```

然后运行前先需要安装下面的包：

```
pip install sentencepiece
```

```
pip install -U pyopenssl cryptography
```

下面是例子代码：

```
from transformers import AutoTokenizer, AutoModel
tokenizer = AutoTokenizer.from_pretrained("/data/dm/chatglm-6b/chatglm-6b", trust_remote_code=True)
model = AutoModel.from_pretrained("/data/dm/chatglm-6b/chatglm-6b", trust_remote_code=True).half().cuda()
model = model.eval()
response, history = model.chat(tokenizer, "你好,我叫王帅", history=[])
print(response)
response, history = model.chat(tokenizer, "我的名字有几个字？", history=history)
print(response)
```

运行结果如下：

```
Loading checkpoint shards: 100%
The dtype of attention mask (torch.int64) is not bool
2023-06-30 18:17:54.268264: I tensorflow/core/platform/cpu_feature_guard.cc:193] This Tenso
12F AVX512_VNNI FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-06-30 18:17:54.444512: I tensorflow/core/util/util.cc:169] oneDNN custom operations ar
ronment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2023-06-30 18:17:54.486143: E tensorflow/stream_executor/cuda/cuda_blas.cc:2981] Unable to
2023-06-30 18:17:55.098813: W tensorflow/stream_executor/platform/default/dso_loader.cc:64]
igdata/client/HDFS/hadoop/lib/native:/opt/Bigdata/client/JDK/jdk1.8.0_272/lib:/opt/Bigdata/
2023-06-30 18:17:55.098903: W tensorflow/stream_executor/platform/default/dso_loader.cc:64]
Y_PATH: /opt/Bigdata/client/HDFS/hadoop/lib/native:/opt/Bigdata/client/JDK/jdk1.8.0_272/lib
2023-06-30 18:17:55.098915: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT
nstalled properly.
/opt/anaconda3/lib/python3.9/site-packages/scipy/__init__.py:146: UserWarning: A NumPy versi
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
你好，王帅！很高兴见到你。有什么我可以帮助你的吗？
我的名字有五个字，叫做“王帅”。
```

注意，显存干掉 12G:

```
Fri Jun 30 18:32:35 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.103.01   Driver Version: 470.103.01   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:0D.0 Off |                    0 |
| N/A   45C    P0    27W /  70W |  12308MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla T4            Off  | 00000000:00:0E.0 Off |                    0 |
| N/A   40C    P8    10W /  70W |      3MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

ChatGLM-6B + LangChain 实践

https://zhuanlan.zhihu.com/p/630147161

打算结合 LangChain 和 ChatGLM-6B 实现长文本生成摘要.

- **step1：自定义一个 GLM 继承 LangChain 中的 langchain.llms.base.LLM，load 自己的模型.**
- step2：使用 LangChain 的 mapreduce 的方法，对文本分块，做摘要，输出结果.

ChatGLM-6B 升级 V2：性能大幅提升，8-32k 上下文，推理提速 42%

https://www.qbitai.com/2023/06/64023.html

在评估 LLM 中文能力的 C-Eval 榜单中 https://cevalbenchmark.com/static/leaderboard.html ，ChatGLM2 模型以 71.1 的分数位居 Rank 0 ，ChatGLM2-6B 模型以 51.7 的分数位居 Rank 6，是榜单上排名最高的开源模型。

#	Model	Creator	Submission Date	Avg ▾	Avg(Hard)	STEM	Social Science	Humanities	Others
0	ChatGLM2	Tsinghua & Zhipu.AI	2023/6/25	71.1	50	64.4	81.6	73.7	71.3
1	GPT-4*	OpenAI	2023/5/15	68.7	54.9	67.1	77.6	64.5	67.8
2	SenseChat	SenseTime	2023/6/20	66.1	45.1	58	78.4	67.2	68.8
3	InternLM	SenseTime & Shanghai AI Laboratory (equal contribution)	2023/6/1	62.7	46	58.1	76.7	64.6	56.4
4	ChatGPT*	OpenAI	2023/5/15	54.4	41.4	52.9	61.8	50.9	53.6
5	Claude-v1.3*	Anthropic	2023/5/15	54.2	39	51.9	61.7	52.1	53.7
6	ChatGLM2-6B	Tsinghua & Zhipu.AI	2023/6/24	51.7	37.1	48.6	60.5	51.3	49.8
7	SageGPT	4Paradigm Inc.	2023/6/21	49.1	39.1	46.6	54.6	45.8	51.8
8	AndesLM-13B	AndesLM	2023/6/18	46	29.7	38.1	61	51	41.9
9	Claude-instant-v1.0*	Anthropic	2023/5/15	45.9	35.5	43.1	53.8	44.2	45.4
10	WestlakeLM-19B	Westlake University and Westlake Xinchen (Scietrain)	2023/6/18	44.6	34.9	41.6	51	44.3	44.5
11	玉言	Fuxi AI Lab, NetEase	2023/6/20	44.3	30.6	39.2	54.5	46.4	42.2
12	bloomz-mt-176B*	BigScience	2023/5/15	44.3	30.8	39	53	47.7	42.7
13	GLM-130B*	Tsinghua	2023/5/15	44	30.7	36.7	55.8	47.7	43
14	baichuan-7B	Baichuan	2023/6/14	42.8	31.5	38.2	52	46.2	39.3
15	CubeLM-13B	CubeLM	2023/6/12	42.5	27.9	36	52.4	45.8	41.8
16	Chinese-Alpaca-33B	Cui, Yang, and Yao	2023/6/7	41.6	30.3	37	51.6	42.3	40.3
17	Chinese-Alpaca-Plus-13B	Cui, Yang, and Yao	2023/6/5	41.5	30.5	36.6	49.7	43.1	41.2
18	ChatGLM-6B*	Tsinghua	2023/5/15	38.9	29.2	33.3	48.3	41.3	38
19	LLaMA-65B*	Meta	2023/5/15	38.8	31.7	37.8	45.6	36.1	37.1
20	Chinese LLaMA-13B*	Cui et al.	2023/5/15	33.3	27.3	31.6	37.2	33.6	32.8
21	MOSS*	Fudan*	2023/5/15	33.1	28.4	31.6	37	33.4	32.1
22	Chinese Alpaca-13B*	Cui et al.	2023/5/15	30.9	24.4	27.4	39.2	32.5	28

ChatGLM2-6B

https://github.com/THUDM/ChatGLM2-6B

embedding 与向量数据库

Langchain 中使用 Qdrant

https://python.langchain.com/docs/modules/data_connection/vectorstores/integrations/qdrant

Qdrant is a vector similarity search engine. There are various modes of how to run Qdrant。

```
pip install qdrant-client
```

We want to use OpenAIEmbeddings so we have to get the OpenAI API Key.

Python client allows you to run the same code in local mode without running the Qdrant server.

The embeddings might be fully kept in memory or persisted on disk

you may prefer to keep all the data in memory only, so it gets lost when the client is destroyed.

```
qdrant = Qdrant.from_documents(
    docs,
    embeddings,
    location=":memory:",  # Local mode with in-memory storage only
    collection_name="my_documents",
)
```

may also store your vectors on disk so they're persisted between runs.

```
qdrant = Qdrant.from_documents(
    docs,
    embeddings,
    path="/data/dev/wujianjun/llm/lcsvr/local_qdrant",
    collection_name="my_documents",
)
```

On-premise server deployment,.you'll need to provide a URL pointing to the service.

```
url = "<---qdrant url here --->"
qdrant = Qdrant.from_documents(
    docs,
    embeddings,
    url,
    prefer_grpc=True,
    collection_name="my_documents",
)
```

Qdrant.from_documents is going to destroy the collection and create it from scratch! If you want to reuse the existing collection, you can always create an instance of Qdrant on your own and pass the QdrantClient instance with the connection details.

```
embeddings = OpenAIEmbeddings()

from qdrant_client import QdrantClient

client = QdrantClient(host="localhost", port=6333)
# or
client = QdrantClient(url="http://localhost:6333")
# or
client = QdrantClient(path="/tmp/local_qdrant", prefer_grpc=True)

qdrant = Qdrant(client=client, collection_name="my_documents", embeddings=embeddings)
```

The simplest scenario is to perform a similarity search. our query will be encoded with the embedding_function and used to find similar documents in Qdrant collection.

```
loader = DirectoryLoader('/data/dev/wujianjun/llm/lcsvr/wjj', glob='**/*.txt')
documents = loader.load()

text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=0, separators=[" ", ",", "\n"])
docs = text_splitter.split_documents(documents)

embeddings = OpenAIEmbeddings()

qdrant = Qdrant.from_documents(
    docs,
    embeddings,
    path="/data/dev/wujianjun/llm/lcsvr/wjj/local_qdrant",
    collection_name="my_documents",
)

query = "怎么退费"
found_docs = qdrant.similarity_search(query)
for item in found_docs:
    print("======================")
    print(item)
```

Sometimes we might want to obtain a relevancy score to know how good is a particular result.

The returned distance score is cosine distance. Therefore, a lower score is better.

```
loader = DirectoryLoader('/data/dev/wujianjun/llm/lcsvr/wjj', glob='**/*.txt')
documents = loader.load()

text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=0, separators=[" ", ",", "\n"])
docs = text_splitter.split_documents(documents)

embeddings = OpenAIEmbeddings()

qdrant = Qdrant.from_documents(
    docs,
    embeddings,
    path="/data/dev/wujianjun/llm/lcsvr/wjj/local_qdrant",
    collection_name="my_documents",
)

query = "怎么退费"
found_docs = qdrant.similarity_search_with_score(query)
for document, score in found_docs:
    print("========================")
    print(document)
    print(score)
```

```
(base) python t.py
/opt/anaconda3/lib/python3.9/site-packages/scipy/__init__.py:146: UserWarning: A NumPy version >=1.16.5 and
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
======================
page_content='申请添加对方微信后续流程：添加红娘老师微信后，对方同意后会通过红娘老师微信把微信号发你，关注
到什么问题了吗？可以具体说一说哦，我这边都会协助您解决的哈~ 因为咱们是线上产品，非产品问题是不支持退款的，
击下方【消息】页面，上方就可以查看您收藏过的人\n\nq：这个点不动怎么办。 ans：点不动可能是网络或者，请您在
进群可以吗。 ans： 我这边每天都在拉实名认证的家长进群哈，因为申请进同城群的家长太多了，我都是一个一个按顺序
txt'}
0.8223043002880452
======================
page_content='ans：也许是对方手机没听到声音，没有看手机，也许是对方家长刚好在忙，没有看手机，您可以换个时间
，每次从小程序拨打出去即可\n\nq：我要退钱。 ans：家长您是在使用中遇到什么问题了吗？可以具体说一说哦，我这边
哦\n\nq：查看资料时点照片能打开，平时时只看资料时没有照片。 ans：打开父母牵线小程序\n\n\n\n我的\n\n\n\n您可
信加不上打电话没人接。 ans：也许是对方家长刚好在忙，没有看手机，您可以换个时间再次联系~电话为隐私号码(虚拟
台搞错了能退费吗？ans：' metadata={'source': '/data/dev/wujianjun/llm/lcsvr/wjj/za.txt'}
0.8153557682870274
======================
page_content='工作日早上10点到下午6点每天我都会抽空回复消息，您可以先留着，看到都会回复的哦~\n\nq：付了299
荐】，第三步您看到满意的就可以和对方打招呼，联系对方~我建议你直接点击打电话 打过去会更快~\n\nq：我不小心退
0.8115041406835326
======================
page_content='ans：【拉群号】我这边每天都在拉实名认证的家长进群哈，因为申请进同城群的家长太多了，我都是一个
点进去就可以进群了哈，您不要着急，耐心等我拉您~\n\nq：我要退钱。 ans：家长您是在使用中遇到什么问题了吗？可
小程序的每日推荐上看看哦\n\nq：怎么注册？ ans：请戳下面的卡片，引导您完成注册，当然我们也有人工客服，随时
理上的问题，也可能是他忙于事业，这些问题我们父母牵线平台也有专业的红娘老师可以帮助到您\n\nq：我刚负过299元
0.8114112254055255
```

Qdrant is a LangChain Retriever, by using cosine similarity.

qdrant 开发手册

https://qdrant.tech/documentation/overview/

Installation options

The easiest way to start using Qdrant is to run it from a ready-made Docker image.

Pull the image: `docker pull qdrant/qdrant`

拉取之后看看本地是否存在这个镜像：

```
(base) docker images
REPOSITORY                TAG                                IMAGE ID       CREATED          SIZE
qdrant/qdrant             latest                             1d3ff47fbb1d   15 hours ago     139MB
llm-api                   pytorch                            159a8f0c9ae4   5 weeks ago      18.9GB
```

Run the container:

```
(base)
(base) docker run -p 6333:6333 -v /data/dev/wujianjun/llm/lcsvr/wjj:/qdrant/storage qdrant/qdrant
        _                 _
  __ _ __| |_ __ __ _ _ __ | |_
 / _` / _` | '__/ _` | '_ \| __|
| (_| | (_| | | | (_| | | | | |_
 \__, |\__,_|_|  \__,_|_| |_|\__|
    |_|

Access web UI at http://localhost:6333/dashboard

[2023-07-04T07:21:10.194Z INFO  storage::content_manager::consensus::persistent] Initializing new raft state at ./storage/raft_state
[2023-07-04T07:21:10.445Z INFO  qdrant] Distributed mode disabled
[2023-07-04T07:21:10.445Z INFO  qdrant] Telemetry reporting enabled, id: 292d8bca-95cc-4e00-ad60-8c71c0958368
[2023-07-04T07:21:10.449Z INFO  qdrant::tonic] Qdrant gRPC listening on 6334
[2023-07-04T07:21:10.449Z INFO  qdrant::tonic] TLS disabled for gRPC API
[2023-07-04T07:21:10.451Z INFO  qdrant::actix] TLS disabled for REST API
[2023-07-04T07:21:10.451Z INFO  actix_server::builder] Starting 71 workers
[2023-07-04T07:21:10.451Z INFO  actix_server::server] Actix runtime found; starting in Actix runtime
```

With this command, you will start a Qdrant instance with the default configuration. It will store all data in /data/dev/wujianjun/llm/lcsvr/wjj.By default, Qdrant uses port 6333.

Quickstart

First, let's create a collection with dot-production metric.

```
(base) curl -X PUT 'http://localhost:6333/collections/test_collection' \
>      -H 'Content-Type: application/json' \
>      --data-raw '{
>          "vectors": {
>              "size": 4,
>              "distance": "Dot"
>          }
>      }'
{"result":true,"status":"ok","time":0.114830952}(base)
```

We can ensure that collection was created:

```
(base) curl 'http://localhost:6333/collections/test_collection'
{"result":{"status":"green","optimizer_status":"ok","vectors_count":0,"indexed_vectors_count":0,"points_count":0,"segments_count":8,"config":{"params":{"vectors":{"size":4,"distance":"Dot"},"shard_number":1,"replication_factor":1,"write_consistency_factor":1,"on_disk_payload":true},"hnsw_config":{"m":16,"ef_construct":100,"full_scan_threshold":10000,"max_indexing_threads":0,"on_disk":false},"optimizer_config":{"deleted_threshold":0.2,"vacuum_min_vector_number":1000,"default_segment_number":0,"max_segment_size":null,"memmap_threshold":null,"indexing_threshold":20000,"flush_interval_sec":5,"max_optimization_threads":1},"wal_config":{"wal_capacity_mb":32,"wal_segments_ahead":0},"quantization_config":null},"payload_schema":{}},"status":"ok","time":0.000058767}(base)
```

Let's now add vectors with some payload:

```
(base) curl -L -X PUT 'http://localhost:6333/collections/test_collection/points?wait=true' \
>      -H 'Content-Type: application/json' \
>      --data-raw '{
>          "points": [
>            {"id": 1, "vector": [0.05, 0.61, 0.76, 0.74], "payload": {"city": "Berlin" }},
>            {"id": 2, "vector": [0.19, 0.81, 0.75, 0.11], "payload": {"city": ["Berlin", "London"] }},
>            {"id": 3, "vector": [0.36, 0.55, 0.47, 0.94], "payload": {"city": ["Berlin", "Moscow"] }},
>            {"id": 4, "vector": [0.18, 0.01, 0.85, 0.80], "payload": {"city": ["London", "Moscow"] }},
>            {"id": 5, "vector": [0.24, 0.18, 0.22, 0.44], "payload": {"count": [0] }},
>            {"id": 6, "vector": [0.35, 0.08, 0.11, 0.44]}
>          ]
>      }'
{"result":{"operation_id":0,"status":"completed"},"status":"ok","time":0.001602041}(base)
```

Let's start with a basic request:

```
(base) curl -L -X POST 'http://localhost:6333/collections/test_collection/points/search' \
>      -H 'Content-Type: application/json' \
>      --data-raw '{
>          "vector": [0.2,0.1,0.9,0.7],
>          "limit": 3
>      }'
{"result":[{"id":4,"version":0,"score":1.362,"payload":null,"vector":null},{"id":1,"version":0,"score":1.273,"payload":null,"vector":null},{"id":3,"version":0,"score":1.208,"payload":null,"vector":null}],"status":"ok","time":0.000405308}(base)
(base)
```

But result is different if we add a filter:

```
(base) curl -L -X POST 'http://localhost:6333/collections/test_collection/points/search' \
>      -H 'Content-Type: application/json' \
>      --data-raw '{
>          "filter": {
>              "should": [
>                  {
>                      "key": "city",
>                      "match": {
>                          "value": "London"
>                      }
>                  }
>              ]
>          },
>          "vector": [0.2, 0.1, 0.9, 0.7],
>          "limit": 3
>      }'
{"result":[{"id":4,"version":0,"score":1.362,"payload":null,"vector":null},{"id":2,"version":0,"score":0.871,"payload":null,"vector":null}],"status":"ok","time":0.000403308}(base)
```

Collections

A collection is a named set of points (vectors with a payload) among which you can search. Vectors within the same collection must have the same dimensionality. **Distance metrics are used to measure similarities among vectors. The choice of metric depends on the way vectors obtaining and, in particular, on the method of neural network encoder training.**

Qdrant supports these most popular types of metrics:

- Dot product:
- Cosine similarity:
- Euclidean distance:

For search efficiency, Cosine similarity is implemented as dot-product over normalized vectors. Vectors are automatically normalized during upload.

create collection

```python
from qdrant_client import QdrantClient
from qdrant_client.http import models

client = QdrantClient("localhost", port=6333)

client.recreate_collection(
    collection_name="my_documents",
    vectors_config=models.VectorParams(size=100, distance=models.Distance.COSINE),
)
```

Payload

One of the significant features of Qdrant is the ability to store additional information along with vectors. This information is called payload. Qdrant allows you to store any information that can be represented using JSON. following excample show how to Create point with payload

```python
from qdrant_client import QdrantClient
from qdrant_client.http import models

client = QdrantClient(host="localhost", port=6333)

client.upsert(
    collection_name="{collection_name}",
    points=[
        models.PointStruct(
            id=1,
            vector=[0.05, 0.61, 0.76, 0.74],
            payload={
                "city": "Berlin",
                "price": 1.99,
            },
        ),
        models.PointStruct(
            id=2,
            vector=[0.19, 0.81, 0.75, 0.11],
            payload={
                "city": ["Berlin", "London"],
                "price": 1.99,
            },
        ),
        models.PointStruct(
            id=3,
            vector=[0.36, 0.55, 0.47, 0.94],
            payload={
                "city": ["Berlin", "Moscow"],
                "price": [1.99, 2.99],
            },
        ),
    ]
)
```

Points

A point is a record consisting of a vector and an optional payload. **Any point modification operation is asynchronous.** Qdrant supports using both 64-bit unsigned integers and UUID as identifiers for points. **To optimize performance, Qdrant supports batch loading of points.**

Create points with REST API :

```
client.upsert(
    collection_name="{collection_name}",
    points=models.Batch(
        ids=[1, 2, 3],
        payloads=[
            {"color": "red"},
            {"color": "green"},
            {"color": "blue"},
        ],
        vectors=[
            [0.9, 0.1, 0.1],
            [0.1, 0.9, 0.1],
            [0.1, 0.1, 0.9],
        ]
    ),
)
```

or record-oriented equivalent:

```
client.upsert(
    collection_name="{collection_name}",
    points=[
        models.PointStruct(
            id=1,
            payload={
                "color": "red",
            },
            vector=[0.9, 0.1, 0.1],
        ),
        models.PointStruct(
            id=2,
            payload={
                "color": "green",
            },
            vector=[0.1, 0.9, 0.1],
        ),
        models.PointStruct(
            id=3,
            payload={
                "color": "blue",
            },
            vector=[0.1, 0.1, 0.9],
        ),
    ]
)
```


points with the same id will be overwritten when re-uploaded.

Sometimes it might be necessary to get all stored points.

```
client.scroll(
    collection_name="{collection_name}",
    scroll_filter=models.Filter(
        must=[
            models.FieldCondition(
                key="color",
                match=models.MatchValue(value="red"),
            ),
        ]
    ),
    limit=1,
    with_payload=True,
    with_vectors=False,
)
```

Similarity search

Modern neural networks are trained to transform objects into vectors so that objects close in the real world appear close in vector space. for example, texts with similar meanings, visually similar pictures, or songs of the same genre. There are many metrics to estimate the similarity of vectors with each other. The most typical metric used in similarity learning models is the cosine metric.

Qdrant counts this metric in 2 steps. The first step is to normalize the vector when adding it to the collection. The second step is the comparison of vectors. In this case, it becomes equivalent to dot production - a very fast operation due to SIMD.

Let's look at an example of a search query.

```
from qdrant_client import QdrantClient
from qdrant_client.http import models

client = QdrantClient("localhost", port=6333)

client.search(
    collection_name="{collection_name}",
    query_filter=models.Filter(
        must=[
            models.FieldCondition(
                key="city",
                match=models.MatchValue(
                    value="London",
                ),
            )
        ]
    ),
    search_params=models.SearchParams(
        hnsw_ef=128,
        exact=False
    ),
    query_vector=[0.2, 0.1, 0.9, 0.7],
    limit=3,
)
```

In this example, we are looking for vectors similar to vector [0.2, 0.1, 0.9, 0.7].

Values under the key params specify custom parameters for the search. Currently, it could be:

- hnsw_ef - value that specifies ef parameter of the HNSW algorithm.
- exact - option to not use the approximate search (ANN).

Since the filter parameter is specified, the search is performed only among those points that satisfy the filter condition. Example result of this API would be

```
{
  "result": [
    { "id": 10, "score": 0.81 },
    { "id": 14, "score": 0.75 },
    { "id": 11, "score": 0.73 }
  ],
  "status": "ok",
  "time": 0.001
}
```

The batch search API enables to perform multiple search requests via a single request.Its semantic is straightforward, n batched search requests are equivalent to n singular search requests. This approach has several advantages. Logically, fewer network connections are required which can be very beneficial on its own. More importantly, batched requests will be efficiently processed via the query planner which can detect and optimize requests if they have the same filter.This can have a great effect on latency for non trivial filters as the intermediary results can be shared among the request.

Embedding in OpenAI API

https://medium.com/@basics.machinelearning/embedding-in-openai-api-b9bb52a0bd55

Word embeddings are learned from large amounts of text data using techniques such as Word2Vec, BERT, or OpenAI model Embedding.

I will explain how to use the Embedding tool from OpenAI API. The method that we will call using OpenAI API is **openai.Embedding.create**. You need first to install openai and create an API Key.

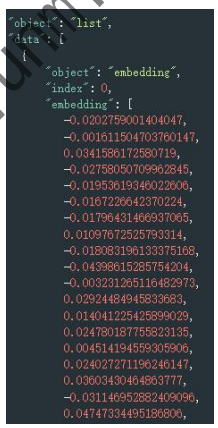
```
import os
os.environ["OPENAI_API_BASE"] = "https://openai.wndbac.cn/v1"
os.environ["OPENAI_API_KEY"] = "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"
import openai

response = openai.Embedding.create(
    model="text-embedding-ada-002",
    input="现在单身现象是一个值得重视的社会问题"
)
print(response)
```

- Model: text-embedding-ada-002 is OpenAI's best embeddings as of Apr 2023.
- Input: The text you want to get the embeddings. Each input must not exceed 8192 tokens.

输出的是一个 json：

其中，embedding 是一个长度为 1536 的数组：



也可以下面这样使用：

```
import os
os.environ["OPENAI_API_BASE"] = "https://openai.wndbac.cn/v1"
os.environ["OPENAI_API_KEY"] = "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"
from langchain.embeddings.openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()
response = embeddings.embed_query("现在单身现象是一个值得重视的社会问题")
print(response.__len__())
print(response)
```

输出如下：

```
(base) python t.py
1536
[-0.0202759001404047, -0.0016115047037601 47, 0.0341586172580719, -0.02758050709962845, -0.
2973, 0.0292448494583368 3, 0.0140412254258990 29, 0.0247801877558231 35, 0.0045141945593059 0
578573361039, 0.0106068719178438 19, -0.0137109998613595 96, -0.0235253274440765 38, -0.0028 5
6414, -0.0135194677859544 75, -0.0207382161170244 2, -0.0161150470376014 7, -0.0173699073493 4
```

可以看到，输入文本一样的话，两种方法得到的 embedding 也是一样的。

还可以如下这样使用

```python
from langchain.embeddings.openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()
batch_texts = [
    "现在单身现象是一个值得重视的社会问题",
    "截止目前虽然有很多研究这进行了深入的研究",
    "但是这一问题目前仍然没有得到有效解决"
]

batch_embeddings = embeddings.embed_documents(batch_texts)
# batch_embeddings 是一个由多个vector组成的list
for item in batch_embeddings:
    # 每个item都是长度为1536的vector
    print(item.__len__(), item[0:10])
```

运行结果如下：

```
(base) python t.py
1536 [-0.02027590003183627, -0.0016115046951312559, 0.03415861707516769, -0.027580506951947095, -0.019536
1536 [-0.013434307207145772, -0.010134300585715733, 0.014619096714125793, -0.025489707473377327, -0.02515
1536 [0.0038982132174133787, -0.011820602654866723, -0.006390948505666235, 0.0020104305887389297, -0.0351
(base)
```

New and improved embedding model

https://openai.com/blog/new-and-improved-embedding-model

The new model, text-embedding-ada-002, replaces five separate models for text search, text similarity, and code search, and outperforms our previous most capable model, Davinci, at most tasks, while being priced 99.8% lower.

You can query the /embeddings endpoint for the new model with two lines of code using our OpenAI Python Library, just like you could with previous models:

```python
import openai
response = openai.Embedding.create(
  input="porcine pals say",
  model="text-embedding-ada-002"
)
```

Qdrant 与 Embedding 协同的使用

代码如下：

```
# step0. 准备语料集
batch_texts = [
    "现在单身现象是一个值得重视的社会问题",
    "截止目前虽然有很多研究这进行了深入的研究",
    "但是这一问题目前仍然没有得到有效解决",
    "一些人认为主要原因在于高房价",
    "另外一些人认为主要原因是高强度的工作",
    "更多人指出这里的原因高度复杂",
]
# step1.对语料集做embedding
embeddings = OpenAIEmbeddings()
batch_embeddings = embeddings.embed_documents(batch_texts)  # batch_embeddings 是一个由多个vector组成的list
# for item in batch_embeddings:
#     print(item.__len__(), item[0:10])  # 每个item都是长度为1536的vector
vector_size = 1536

# step3. 连接数据库
client = QdrantClient(host="localhost", port=6333)
# step4.创建collection
collection_name = "my_documents"
client.recreate_collection(
    collection_name=collection_name,
    vectors_config=models.VectorParams(size=vector_size, distance=models.Distance.COSINE, )
)
# step5. 生成 id 和 paylaod
batch_ids = [uuid.uuid4().hex for _ in iter(batch_texts)]
payloads = [{"page_content": text} for text in batch_texts]
# step6.批量插入数据库
client.upsert(
    collection_name=collection_name,
    points=models.Batch.construct(
        ids=batch_ids,
        vectors=batch_embeddings,
        payloads=payloads,
    ),
)
# step7. 计数与遍历
count_res = client.count(collection_name=collection_name, exact=True, )
print(count_res)
allitems = client.scroll(collection_name=collection_name, limit=10, with_payload=True, with_vectors=True, )
for item in allitems[0]:
    print(f"id={item.id}, payload={item.payload}")

# step8. 搜索
embedding = embeddings.embed_query("单身问题跟房价关系大吗？")
results = client.search(
    collection_name=collection_name,
    query_vector=embedding,
    limit=3,
    with_payload=True,
    with_vectors=False,  # 不要返回vector
)
print("=========================================================================")
for item in results:
    print(f"id={item.id}, payload={item.payload},score={item.score}")
```


运行结果如下：

```
(base) python t.py
count=6
id=39cc0ef4-8cd2-473b-b2a3-b0fac8adf824, payload={'page_content': '现在单身现象是一个值得重视的社会问题'}
id=43252f9a-239c-455c-bd7f-2ad24c66262d, payload={'page_content': '更多人指出这里的原因高度复杂'}
id=4633247a-289e-4ffc-84ab-4d50275323a4, payload={'page_content': '截止目前虽然有很多研究这进行了深入的研究'}
id=478c90c2-026f-4dfb-a72f-b7e724f39c34, payload={'page_content': '但是这一问题目前仍然没有得到有效解决'}
id=4ae733c1-4546-43e3-9153-b00c97fd9266, payload={'page_content': '一些人认为主要原因在于高房价'}
id=eb1fabd5-f24c-4ee3-9116-2cae092b221d, payload={'page_content': '另外一些人认为主要原因是高强度的工作'}
============================================================
id=39cc0ef4-8cd2-473b-b2a3-b0fac8adf824, payload={'page_content': '现在单身现象是一个值得重视的社会问题'},score=0.8649426
id=4ae733c1-4546-43e3-9153-b00c97fd9266, payload={'page_content': '一些人认为主要原因在于高房价'},score=0.83278394
id=43252f9a-239c-455c-bd7f-2ad24c66262d, payload={'page_content': '更多人指出这里的原因高度复杂'},score=0.7838334
(base)
```

提示工程 Prompt

LLM Prompt 概述

The Beginner's Guide to LLM Prompting

https://haystack.deepset.ai/blog/beginners-guide-to-llm-prompting

we'll explain our approach to **prompt engineering** . A prompt is an instruction to an LLM. **Good prompts follow two basic principles: clarity and specificity.**

- Clarity means we should use simple and unambiguous language that avoids jargon and overly complex vocabulary. it's better to take the long-winded way to make your point sufficiently clear to the LLM. Example of an unclear prompt: `Who won the election?`, Example of a clear prompt: `Which party won the 2023 general election in Paraguay?` 。
- Specificity means we should tell model as much as it needs to know to answer your question.Example of an unspecific prompt: `Generate a list of titles for my autobiography.` Example of a specific prompt:

```
Generate a list of ten titles for my autobiography. The book is about my journey as an adventurer who has lived an unconventional life, meeting many different personalities and finally finding peace in gardening.
```

Let's look at a few tricks to make our prompts even better.

- **Do say "do," don't say "don't"**：in our previous example, we want to make sure that the LLM doesn't produce overly long titles . Rather than saying: `Don't make the titles too long`. it is better to be specific: `Each title should be between two and five words long.`。
- **Use few-shot prompting**：**we asked for a particular item (the title for an autobiography) of a certain length (two to five words). This type of instruction without any concrete examples is known as "zero-shot prompting." Most models benefit from "few-shot prompting," where you feed your model some examples.** we can add a few existing titles as examples to help the model get a feel for what we're looking for.

```
Here are some examples: Long walk to freedom, Wishful drinking, I know why the caged bird sings.
```

- **Structure your prompt in a meaningful way: Elements like quotation marks(引号), bullet points(项目符号), line breaks(换行符), etc. make it easier for humans to parse a text. And The same is true for LLMs.** Let's apply that insight to our previous example and spell out the full prompt once more.

```
Generate a list of ten titles for my autobiography.
The book is about my journey as an adventurer who has lived an unconventional life, meeting many different personalities and finally finding peace in gardening.
Each title should be between two and five words long.

### Examples of great titles ###

- "Long walk to freedom"
- "Wishful drinking"
- "I know why the caged bird sings"
```

输出如下：

```
(base) python t.py
1. Uncharted Paths
2. From Wanderlust to Roots
3. Life's Wild Expedition
4. Embracing the Unknown
5. Tales of the Trailblazer
6. Adventures and Serenity
7. Unconventional Encounters
8. Finding Home Through Nature
9. Journeys of Discovery
10. Gardening: My Final Frontier
```

再比如：

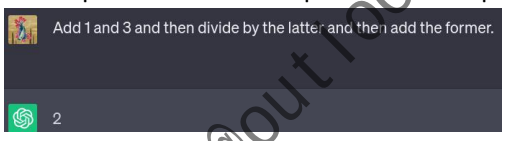
```
from langchain.chat_models import ChatOpenAI
from langchain.schema import HumanMessage


chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
content = """
我打算写一本数书讲解在工业界数据挖掘的实战经验，帮我列下这本书每章的标题吧。
在工业界做数据挖掘需要python基础，数学基础，还需要一些算法基础，然后介绍几个实战项目，比如欺诈监测，电商推荐，最后对尚存的挑战和未来高潜力应用做一个展望。
### 下面是一些每章标题的例子
- 数据挖掘简洁
- python基础
- 大规模电商推荐系统
"""
res = chat.predict_messages([HumanMessage(content=content)])
print(res.content)
```

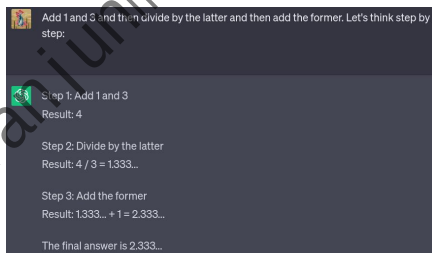
输出如下：

```
(base) python t.py
- 数据预处理与清洗
- 特征工程与数据可视化
- 机器学习算法概述
- 监督学习算法与应用
- 无监督学习算法与应用
- 欺诈监测实战案例
- 电商推荐系统实战案例
- 持续学习与模型优化
- 数据挖掘中的隐私与安全
- 数据挖掘的伦理与法律问题
- 未来高潜力应用展望
```

- **Use leading words: telling a model to "think step by step" can generate more accurate results and help the model correct its own mistakes. This is known as "leading words," because we gently guide the model towards a more effective plan for problem-solving.** Seeing how this simple trick improves the model's performance is pretty striking.



The LLM confidently responds -with the wrong answer. Here's what happens when we ask it to "think step by step:"



And this time, the final answer is correct.

再比如：

```
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
content = """
小吴的爸爸是四川人，但是他爸爸现在在广东工作，他爸爸一直想回成都。现在小吴在深圳上小学，他以后可以在成都上中学吗？
"""
res = chat.predict_messages([HumanMessage(content=content)])
print(res.content)
```

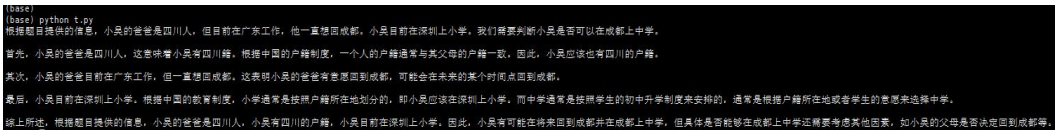
回答如下；

```
(base) python t.py
是的，小吴可以在成都上中学。无论父母在哪里工作，孩子的教育权益是受到保护的。根据中国的教育制度，小吴可以在成都上中学，无论他的父母在哪个城市工作。
```

我们要求列出思考步骤：

```
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
content = """
小吴的爸爸是四川人，但是他爸爸现在在广东工作，他爸爸一直想回成都。现在小吴在深圳上小学，他以后可以在成都上中学吗？
把你得到结论的每一步的思考告诉我。
"""
res = chat.predict_messages([HumanMessage(content=content)])
print(res.content)
```

回答如下；



再比如：

```
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
content = """
我打算从深圳回到老家剑阁县，要先做飞机到绵阳，需要3个小时，再需要2个小时的公共汽车到剑阁，中间换乘吃饭估计要两2个小时，但是如果不做公共汽车而是打车的话可以节省一半的时间，
告诉我需要多久我可以从深圳到达剑阁?
"""
res = chat.predict_messages([HumanMessage(content=content)])
print(res.content)
```

```
(base) python t.py
如果选择飞机到绵阳，再乘坐公共汽车到剑阁，总共需要3小时（飞机）+ 2小时（公共汽车）+ 2小时（换乘和吃饭）= 7小时。

如果选择打车直接从深圳到剑阁，可以节省一半的时间，即7小时/2 = 3.5小时。

所以，从深圳到剑阁，如果选择打车的话，大约需要3.5小时。
```

我们要求列出推理步骤：

```
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
content = """
我打算从深圳回到老家剑阁县，要先做飞机到绵阳，需要3个小时，再需要2个小时的公共汽车到剑阁，中间换乘吃饭估计要两2个小时，但是如果不做公共汽车而是打车的话可以节省一半的时间，
告诉我需要多久我可以从深圳到达剑阁?
列出你计算的每一步。
"""
res = chat.predict_messages([HumanMessage(content=content)])
print(res.content)
```

```
(base) python t.py
根据题目描述，我计算从深圳到剑阁的时间如下：

1. 飞机从深圳到绵阳需要3个小时。
2. 从绵阳机场到剑阁县需要2个小时的公共汽车。中间换乘吃饭估计需要2个小时。
3. 如果不做公共汽车而是打车的话，可以节省一半的时间，即1个小时。
4. 所以，总共需要的时间是：3小时（飞机）+ 2小时（公共汽车）+ 2小时（吃饭）= 7小时。
5. 如果选择打车，总共需要的时间是：3小时（飞机）+ 1小时（打车）+ 2小时（吃饭）= 6小时。

所以，从深圳到剑阁，如果选择公共汽车，需要7个小时；如果选择打车，需要6个小时。
```

In addition to these techniques for improvement, it helps to know which elements in an instruction can throw the model off. **Some people, for example, try to incorporate requests for short, one-word answers into their prompts to prevent the models from rambling.** However wordier answers might not be a bad thing.

Another problem happens when you exceed the model's context length. an LLM's "context window" comprises both the prompt and its response. If you exceed a model's context length, its output will deteriorate. remember that if the length of your prompt becomes an issue, you should look into breaking the problem down into smaller steps.

This article leans heavily on Vladimir Blagojevic's webinar on "Prompting LLMs Using Haystack." https://www.youtube.com/watch?v=-TfPkX3IoUQ. In it, Vladimir, who is a member of the core engineering team here at deepset, takes a deep dive into the art of prompting and shows how Haystack, our open source framework for NLP, offers just the right tools to get prompting right.

Prompts also play a huge role in Agents: the latest fascinating trend that lets you harness the power of LLMs. Agents receive a sort of "super-prompt" that instructs the LLM to break its reasoning into manageable steps and delegate those to the tools it deems most capable of solving the tasks. Have a look at our blog post on Agents to learn more.

Finally, you don't always need to write your own prompts. Head over to our https://prompthub.deepset.ai/ where you find prompts for all kinds of applications, which you can use out of the box to get the best answers from your LLM.

Prompt Engineering Guidelines

https://docs.haystack.deepset.ai/docs/prompt-engineering-guidelines

General Tips

- Prompts can contain instructions or questions to pass to the model. They can also contain additional information, such as context or examples.
- Prompt engineering is an iterative process. Be ready to experiment.
- **Start with simple, zero-shot prompts and then keep making them more complex to reach the required accuracy of answers. You can add more context and examples to each iteration of your prompt. If you're not seeing any improvements, fine-tune the model.**

Prompt Format

Prompts with the format that follows these guidelines seem to achieve better results than prompts with random formatting:

- The prompt has instructions, context, input data, and output format. Not all these components are necessary, and they may vary depending on the model's task.

```
Question:
Context:
Answer:
```

- You can try experimenting with specific prompt structures, for example:

例子：

```
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
human_input1 = """
Question:
今年干旱，需要从县城给鞍山村送水，请给一个送水的方案，包括工具，时间等，并且给出方案依据。

Context:
鞍山村大约300口人，多数是留守的老人和小孩，有大约10个壮劳动力。平均每个人每天需要饮水5斤，另外还有大约500头猪，每头猪每天需要饮水10斤。
鞍山村隶属于开封镇，县城到开封镇大约20公里，有柏油路。
开封镇到鞍山村是山路，走不了大车。

Answer:
"""
human_input = [HumanMessage(content=human_input1)]
ai_output = chat(human_input)
print(f"(ai): {ai_output.content}")
```

```
(base) python t.py
(ai): 送水方案如下：

1. 工具：
- 小型水罐车：由于山路无法通行大车，选择小型水罐车作为送水工具。水罐车应具备足够的容量，能够载数所需的饮水量。
- 水桶或水壶：用于将水从水罐车中倒入村民家中的容器。

2. 时间：
- 每天早上和下午各一次，由于干旱情况严重，每天早上和下午各一次送水，以满足村民和猪的饮水需求。

3. 方案依据：
- 人均饮水需求：根据题目提供的信息，每个人每天需要饮水5斤，共300口人，即总共需要1500斤水。
- 猪的饮水需求：每头猪每天需要饮水10斤，共500头猪，即总共需要5000斤水。
- 总饮水需求：村民和猪的总饮水需求为1500斤+5000斤=6500斤水。
- 水罐车容量：根据总饮水需求，选择水罐车的容量应不小于6500斤，以确保一次能够载数足够的水量。
- 山路无法通行大车：根据题目提供的信息，山路无法通行大车，因此选择小型水罐车作为送水工具，以适应山路的条件。

综上所述，送水方案包括使用小型水罐车作为送水工具，每天早上和下午各一次送水，以满足鞍山村300口人和500头猪的饮水需求，方案依据是根据人均饮水需求和猪的饮水需求计算出总饮水需求，并选择水罐车容量不小于总饮水需求，同时考虑到山路无法通行大车的情况。
```

Instructions

- **Put them at the beginning of the prompt.**
- Use commands, for example, "write", "summarize", "translate".
- Separate instructions from the context with a delimiter, for example, quotation marks (""") or hashes (###).
- Separate the instructions, examples, context, and input data with one or more line breaks.

Examples:

```
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
human_input1 = """
### Instruction ###
Give a title to the article. The title must have 5 words.

Article: Online retail giant Amazon has said it plans to shutdown three warehouses in the UK putting 1200 jobs at risk.

Title:
"""
human_input = [HumanMessage(content=human_input1)]
ai_output = chat(human_input)
print(f"(ai): {ai_output.content}")
```

```
(base) python t.py
(ai): Amazon to Close UK Warehouses
```

```
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
human_input1 = """
Instructions: Detect the language of the text. Answer with the name of the language.

Text: Professionelle Beratung und ein Top-Service sind für uns selbstverständlich. Daher bringen unsere Mitarbeiter eine große Portion Kompetenz und Erfahrung mit.
"""
human_input = [HumanMessage(content=human_input1)]
ai_output = chat(human_input)
print(f"(ai): {ai_output.content}")
```

```
(base) python t.py
(ai): German
```

Context

Specify any information you want the model to use to generate the answer.

Example:

```
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
human_input1 = """
用这里的上下文回答问题。如果你不确定直接说"不知道"。

Context: 运筹学的求解方法有两大类：单纯形法和内点法，内点可以结合machine learning来优化，比如使用machine learning来学习内点法中的迭代步长。

Question: {Question}
"""
human_input = [HumanMessage(content=human_input1.format(Question="可以用machine learning学习单纯性法的步长吗？"))]
ai_output = chat(human_input)
print(f"(ai): {ai_output.content}")
human_input = [HumanMessage(content=human_input1.format(Question="可以用machine learning学习内点法的步长吗？"))]
ai_output = chat(human_input)
print(f"(ai): {ai_output.content}")
```

```
(base) python t.py
(ai): 不知道。
(ai): 可以使用machine learning来学习内点法中的迭代步长。
```

Output Format

- Be specific about the format you want the model to use for the answer.
- It may help to give the model a couple of concrete examples.

Example:

```
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
input_template = """
把下面对女孩的描述分成：温柔型，事业型，未知型。

描述：我常常漫步金色的夕阳下，也爱听细雨敲打小窗，期待有人与我和歌共舞余生。
类型：温柔型

描述：本人今年30岁，任职于500强企业，985学校硕士毕业，在深圳有房有车，期待与你共创辉煌人生。
类型：事业型

描述：本人颜值高分，老家福建，人在深圳，期待与你相识。
类型：neutral

描述：{desc}
类型：
"""
human_input = [HumanMessage(content=input_template.format(desc="我在一家头部企业担任产品经理一职，加班多，期待能找到一位可以支持我职业发展的男士"))]
ai_output = chat(human_input)
print(f"(ai): {ai_output.content}")
human_input = [HumanMessage(content=input_template.format(desc="雨夜听芭蕉，细语诉与你。繁星衬皓月，清风为你来"))]
ai_output = chat(human_input)
print(f"(ai): {ai_output.content}")
human_input = [HumanMessage(content=input_template.format(desc="本人身材高挑，父母催的急，现在真诚期待与你遇见，合适的话愿意确立恋爱关系"))]
ai_output = chat(human_input)
print(f"(ai): {ai_output.content}")
```

```
(base) python t.py
(ai): 事业型
(ai): 温柔型
(ai): 未知型
```

Few-Shot Prompts

- Use consistent formatting.
- Provide examples in random order. For example, don't put all negative examples first and positive ones second. Mix them up as the order might bias the model.
- Make sure you use labels.

参考论文: **Language Models are Few-Shot Learners** https://arxiv.org/abs/2005.14165

Model Parameters

When interacting with large language models, take into account these three parameters as they can significantly impact your prompt performance:

- **temperature**: The lower the temperature, the more realistic the results. A lower temperature means the next token is always picked. **For fact-based question answering, we recommend setting the temperature to lower values, like 0.** A higher temperature can lead to more random, creative, and diverse answers. You may consider setting the temperature to a higher value if you want the model to perform creative tasks, like poem generation.
- **top_p**: This parameter controls nucleus sampling, which is a technique to generate text that balances both coherence and diversity. It limits the set of possible next words based on the threshold of probability value p. You typically set it to a value between 0 and 1.**If you want the model to generate factual answers, set top_p to a low value, like 0.**
- **stop_words**: When the model generates the set of characters specified as the stop words, it stops generating more text. It helps to control the length and relevance of the generated text.

Prompt Engineering Guide

https://www.promptingguide.ai/

LangChain 中的 Prompt

Prompts

https://python.langchain.com/docs/modules/model_io/prompts/

A prompt refers to the input to the model. LangChain provides several classes and functions to make constructing and working with prompts easy.

- Prompt templates: Parametrize model inputs
- Example selectors: Dynamically select examples to include in prompts

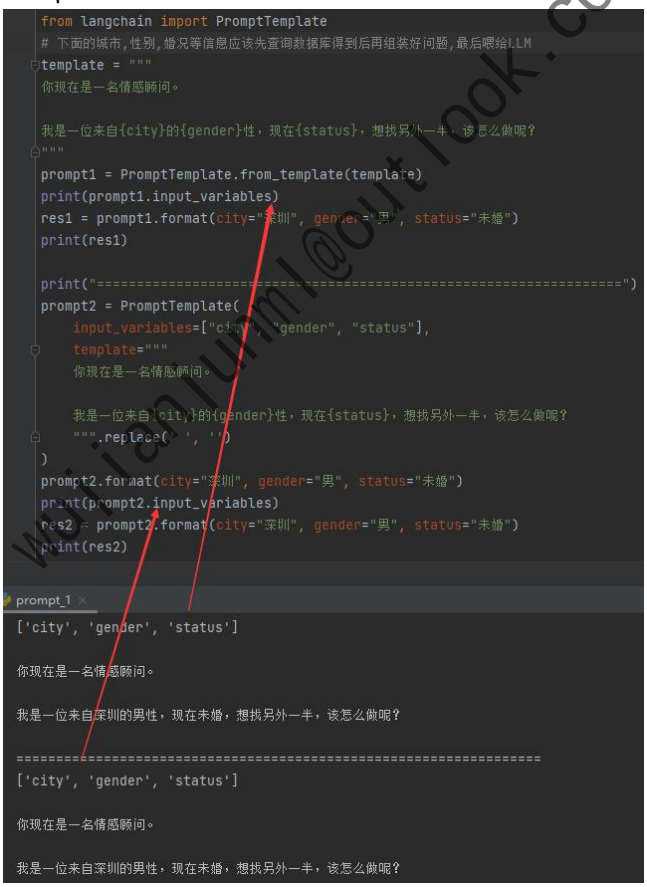
Prompt templates

Language models take text as input which is referred to as a prompt. Typically this is not simply a hardcoded string but rather a combination of a template, some examples, and user input.

A prompt template contains a text string, that can take in a set of parameters and generates a prompt.A prompt template can contain:

- instructions to the language model,
- a set of few shot examples to help the language model generate a better response,
- a question to the language model.

Here's the simplest example:



chat messages differ from raw string in that every message is associated with a role. For example, in OpenAI Chat API, a chat message can be associated with the AI, human or system role. The model is supposed to follow instruction from system chat message more closely.You are encouraged to use these chat related prompt templates instead of PromptTemplate when querying chat models to fully exploit the potential of underlying chat model.

例子如下：

```
sys_template = "你是一名婚恋顾问,负责回答{product}产品上的问题。 "
sys_msg_prompt = SystemMessagePromptTemplate.from_template(sys_template)
res = sys_msg_prompt.format(product="爱恋奇缘")
print(res.type)
print(res.content)
print("================================================================")
human_template = "我下载不了{product}这个APP，该怎么办？"
human_message_prompt = HumanMessagePromptTemplate.from_template(human_template)
res = human_message_prompt.format(product="爱恋奇缘")
print(res.type)
print(res.content)
print("================================================================")
ai_template = "请说明一些您手机的{os}？"
ai_template_prompt = AIMessagePromptTemplate.from_template(ai_template)
res = ai_template_prompt.format(os="操作系统")
print(res.type)
print(res.content)

print("************************************************************")
chat_prompt = ChatPromptTemplate.from_messages([sys_msg_prompt, human_message_prompt, ai_template_prompt])
res = chat_prompt.format_prompt(product="爱恋奇缘", os="操作系统").to_messages()
for item in res:
    print(item.type, item.content)
```

```
prompt_1
system
你是一名婚恋顾问,负责回答爱恋奇缘产品上的问题。
================================================================
human
我下载不了爱恋奇缘这个APP，该怎么办？
================================================================
ai
请说明一些您手机的操作系统？
************************************************************
system 你是一名婚恋顾问,负责回答爱恋奇缘产品上的问题。
human 我下载不了爱恋奇缘这个APP，该怎么办？
ai 请说明一些您手机的操作系统？
```

官方文档中还有很多很多这方面的内容，应该持续深入全面的了解。

Example selectors

If you have a large number of examples, you may need to select which ones to include in the prompt. The Example Selector is the class responsible for doing so.

Agents 与复杂任务

Agents 概述

Introducing Agents in Haystack: Make LLMs resolve complex tasks

https://haystack.deepset.ai/blog/introducing-haystack-agents

now we're officially introducing the Agent to the Haystack . The implementation is inspired by two papers: https://arxiv.org/abs/2205.00445 and the https://arxiv.org/abs/2210.03629.

What is a Prompt?

a prompt is an instruction. In the world of LLM, these instructions can be things like 'Answer the given query', or 'Summarize the following piece of text'.

For example, I wanted to have a system that can tell me what type of things a Twitter user has been posting about. The prompt starts like this:

You will be given a twitter stream belonging to a specific profile. Answer with a summary of what they've lately been tweeting about and in what languages. You may go into some detail about what topics they tend to like tweeting about. Please also mention their overall tone, for example: positive, negative, political, sarcastic or something else.

What is an Agent?

an Agent is an LLM that has been given a very clever initial prompt. The prompt tells the LLM to break down the process of answering a complex query into a sequence of steps that are resolved one at a time.

For example, an Agent might not have the capability to perform mathematical calculations, we can introduce an expert who at mathematical calculations - just a calculator. when we need to perform a calculation, the Agent can call in the expert. an Agent that is asked "Who was the US president ten years ago today?". A simplified view of the Agent's thought process as it breaks down this question might look like this:

"I have to answer the question: What is today's date?"

"Now I know todays date is 29th March 2023, I need to answer: what is 29 March 2023 minus 10 years?"

"Now I need to answer the question: Who was the US president on 29 of March 2013?"

At each step, the Agent may decide to make use of an expert to come to a resolution. In Haystack, these experts are called Tools.

What are Tools?

an Agent might be given a Tool that can search the web (call it 'Websearch'). If the Agent sees that there's a need to search the web and it has the Websearch in its set of tools, it will use it.

How does an Agent select a Tool? Each Tool comes with a description. This is one of the most important attributes of a Tool, as it is used by the Agent to make the Tool selection.

LangChain 中 Agent 的一个实用案例

https://zhuanlan.zhihu.com/p/627948474

比如我们要做一个智能问答，能够介绍公司基本情况和公司产品。我们先定义一些模拟数据库据。**数据库可以是结构化的 KV store 比如下面的 find_product_description，也可以是需要结合 llm 阅读理解能力生成结果的比如这里的 find_company_info。**

```
CONTEXT_QA_TMPL = """
根据以下提供的信息，回答用户的问题
信息：{context}

问题：{query}
"""
CONTEXT_QA_PROMPT = PromptTemplate(input_variables=["query", "context"], template=CONTEXT_QA_TMPL, )


class FugeDataSource:
    def __init__(self, llm: BaseLLM):
        self.llm = llm

    def find_product_description(self, product_name: str) -> str:
        """模拟公司产品的数据库"""
        product_info = {
            "好快活": "好快活是一个营销人才平台，以社群+公众号+小程序结合的运营模式展开，帮助企业客户连
            "Rimix": "Rimix通过采购流程数字化、完备的项目数据存储记录及标准的供应商管理体系，帮助企业实
            "Bid Agent": "Bid Agent是一款专为中国市场设计的搜索引擎优化管理工具，支持5大搜索引擎。Bid A
        }
        return product_info.get(product_name, "没有找到这个产品")

    def find_company_info(self, query: str) -> str:
        """模拟公司介绍文档数据库，让llm根据抓取信息回答问题"""
        context = """
        关于产品："让广告技术美而温暖"是复歌的产品理念。在努力为企业客户创造价值的同时，也希望让使用复歌
        我们关注用户的体验和建议，我们期待我们的产品能够给每个使用者的工作和生活带来正面的改变。
        我们崇尚技术，用科技的力量使工作变得简单，使生活变得更加美好而优雅，是我们的愿景。
        企业文化：复歌是一个非常年轻的团队，公司大部分成员是90后。工作上，专业、注重细节、拥抱创新、快速
        以上这些都是复歌团队的重要特质。在复歌，每个人可以平等地表达自己的观点和意见，每个人的想法和意愿
        prompt = CONTEXT_QA_PROMPT.format(query=query, context=context)
        return self.llm(prompt)
```

然后定义 prompt，prompt 明确告诉 llm 可以使用的 tool，以及思考的步骤。

```
# 特别注意这个prompt!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
# 我们要求 llm 一步一步地推理，且按照约定的格式给出回到
# 我们接着会解析回答里面的信息，并决定是调用相关 tool 增加上下文继续让 llm 推理，还是给出最终答案
agent_tmpl = """按照给定的格式回答以下问题。你可以使用下面这些工具：

{tools}

回答时需要遵循以下用---括起来的格式：

---
Question: 我需要回答的问题
Thought: 回答这个上述我需要做些什么
Action: "{tool_names}" 中的其中一个工具名
Action Input: 选择工具所需要的输入
Observation: 选择工具返回的结果
...（这个思考/行动/行动输入/观察可以重复N次）
Thought: 我现在知道最终答案
Final Answer: 原始输入问题的最终答案
---

现在开始回答，记得在给出最终答案前多按照指定格式进行一步一步的推理。

Question: {input}
{agent_scratchpad}
"""


class CustomPromptTemplate(StringPromptTemplate):
    template: str  # 标准模板
    tools: List[Tool]  # 可使用工具集合

    # 注意,input 就是用户的原始问题,intermediate_steps 是之前agent所有步骤的信息
    def format(self, **kwargs) -> str:
        print("CustomPromptTemplate:format")
        print("------------------------------------")
        for k ,v in kwargs.items():
            print(f"{k}:{v}")
        print("************************************")
        for item in kwargs["intermediate_steps"]:
            print(item)
        print("------------------------------------")
        intermediate_steps = kwargs.pop("intermediate_steps")  # 取出中间步骤并进行执行
        thoughts = ""
        for action, observation in intermediate_steps:
            thoughts += action.log
            thoughts += f"\nObservation: {observation}\nThought: "
        kwargs["agent_scratchpad"] = thoughts  # 记录下当前想法
        kwargs["tools"] = "\n".join([f"{tool.name}: {tool.description}" for tool in self.tools])  # 枚举所有可使用的工具名+工具描述
        kwargs["tool_names"] = ", ".join([tool.name for tool in self.tools])  # 枚举所有的工具名称
        cur_prompt = self.template.format(**kwargs)
        return cur_prompt
```

比如，某一步中信息如下：

```
CustomPromptTemplate:format
------------------------------------
input:介绍下复歌科技公司
intermediate_steps:[(AgentAction(tool='查询复歌科技公司相关信息', tool_input='复歌科技公司', log='Thought: 我需要查询复歌科
************************************
(AgentAction(tool='查询复歌科技公司相关信息', tool_input='复歌科技公司', log='Thought: 我需要查询复歌科技公司的相关信息\nAct
(AgentAction(tool='查询产品名称', tool_input='复歌科技公司', log='我需要使用另一个工具来查询复歌科技公司的相关信息。\nAction
(AgentAction(tool='复歌科技公司相关信息', tool_input='复歌科技公司', log='我需要使用另一个工具来查询复歌科技公司的相关信息。
------------------------------------
```

接下来，我们用正则式解析 llm 的返回结果，并决定是调用哪个工具还是直接输出答案：

```python
# llm按照prompt的要求给出了回答，其中给出 Action(就是采用哪个 tool),action_input 等信息
# 然后我们解析这些信息,从而让 agents 调用相关工具 或者 直接返回最终答案
class CustomOutputParser(AgentOutputParser):
    def parse(self, llm_output: str) -> Union[AgentAction, AgentFinish]:
        print("CustomOutputParser::parse")
        print("================================================")
        print(llm_output)
        print("================================================")
        # 解析 llm 的输出，根据输出文本找到需要执行的决策。
        if "Final Answer:" in llm_output:  # 如果句子中包含 Final Answer 则代表已经完成
            return AgentFinish(return_values={"output": llm_output.split("Final Answer:")[-1].strip()}, log=llm_output, )
        else:
            regex = r"Action\s*\d*\s*:(.*?)\nAction\s*\d*\s*Input\s*\d*\s*:[\s]*(.*)"  # 解析 action_input 和 action
            match = re.search(regex, llm_output, re.DOTALL)
            if not match:
                raise ValueError(f"Could not parse LLM output: `{llm_output}`")
            action = match.group(1).strip()
            action_input = match.group(2)
            return AgentAction(tool=action, tool_input=action_input.strip(" ").strip('"'), log=llm_output)
```

比如，某一步这个函数打印如下：

```
CustomOutputParser::parse
================================================
Thought: 我需要查询复歌科技公司的相关信息
Action: 查询复歌科技公司相关信息
Action Input: 复歌科技公司
================================================
```

最后，我们开启问答循环：

```python
llm = OpenAI(temperature=0, model_name="gpt-3.5-turbo")
fuge_data_source = FugeDataSource(llm)
tools = [
    Tool(name="查询产品名称",
         func=fuge_data_source.find_product_description,
         description="通过产品名称找到产品描述时用的工具，输入应该是产品名称",),
    Tool(name="复歌科技公司相关信息",
         func=fuge_data_source.find_company_info,
         description="当用户询问公司相关的问题，可以通过这个工具了解相关信息",),
]
agent_prompt = CustomPromptTemplate(
    template=agent_tmpl,
    tools=tools,
    input_variables=["input", "intermediate_steps"]
)
llm_chain = LLMChain(llm=llm, prompt=agent_prompt)

output_parser = CustomOutputParser()
tool_names = [tool.name for tool in tools]
agent = LLMSingleActionAgent(
    llm_chain=llm_chain,
    output_parser=output_parser,
    stop=["\nObservation:"],
    allowed_tools=tool_names,
)
agent_executor = AgentExecutor.from_agent_and_tools(agent=agent, tools=tools)
while True:
    user_input = input("请输入您的问题: ")
    response = agent_executor.run(user_input)
    print(response)
```


简单总结为：

- **我们把问题输入 llm_chain,其 prompt 里面要求 llm 的回答按约定的格式给出，比如这里给出 Action(就是采用哪个 tool), action_input 等信息**
- **然后我们用正则式解析 llm 的返回信息，拿到诸如 Action 和 action_input，随之我们指示接 agent 就调用 tool。注意,这一步可能一下子找不到正确的工具，那么会多试几次**
- **接着 llm_chain 看看 tool 的运行结果，如果没有这个 tool 或者 tool 没有给到期望的信息，则 llm 会做多次不同的尝试 直到找到合适的答案。最后 llm 返回 Final Answer**

```
请输入您的问题: 介绍下好快活这个产品
CustomPromptTemplate:format
------------------------------------
input:介绍下好快活这个产品
intermediate_steps:[]
************************************
------------------------------------
CustomOutputParser::parse
================================================
Thought: 我需要查询产品名称来找到产品描述
Action: 查询产品名称
Action Input: 好快活
================================================
CustomPromptTemplate:format
------------------------------------
input:介绍下好快活这个产品
intermediate_steps:[(AgentAction(tool='查询产品名称', tool_input='好快活', log='Thought: 我需要查
以社群+公众号+小程序结合的运营模式展开，帮助企业客户连接并匹配充满才华的营销人才。')]
************************************
(AgentAction(tool='查询产品名称', tool_input='好快活', log='Thought: 我需要查询产品名称来找到产品
结合的运营模式展开，帮助企业客户连接并匹配充满才华的营销人才。')
------------------------------------
CustomOutputParser::parse
================================================
我现在知道好快活是一个营销人才平台，帮助企业客户连接并匹配充满才华的营销人才。

Final Answer: 好快活是一个营销人才平台，帮助企业客户连接并匹配充满才华的营销人才。
================================================
好快活是一个营销人才平台，帮助企业客户连接并匹配充满才华的营销人才。
```

LangChain 的 Agents

Agents

https://python.langchain.com/docs/modules/agents/

There are two main types of agents:

- **Action agents: at each timestep, decide on the next action using the outputs of all previous actions.**
- **Plan-and-execute agents: decide on the full sequence of actions up front, then execute them all without updating the plan**

Action agents are suitable for small tasks, while plan-and-execute agents are better for complex or long-running tasks. Often the best approach is to combine the dynamism of an action agent with the planning abilities of a plan-and-execute agent.

Action agents

At a high-level an action agent:

1. **Receives user input**
2. **Decides which tool, if any, to use and the tool input**
3. **Calls the tool and records the output (also known as an "observation")**
4. **Decides the next step using the history of tools, tool inputs, and observations**
5. **Repeats 3-4 until it determines it can respond directly to the user**

Action agents are wrapped in **agent executors**. an agent typically involves these components:

- **Prompt template**: Responsible for taking the **user input** and **previous steps** and constructing a prompt to send to the language model.
- Language model: Takes the prompt with **use input** and **action history** and **decides what to do next**
- **Output parser**: **Takes the output of the language model and parses it into the next action or a final answer**

Plan-and-execute agents

At a high-level a plan-and-execute agent:

1. Receives user input
2. Plans the full sequence of steps to take
3. Executes the steps in order, passing the outputs of past steps as inputs to future steps.

Get started

例子如下：

```
llm = OpenAI(temperature=0)
tools = load_tools(["serpapi", "llm-math"], llm=llm)
agent = initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
res = agent.run("2022年中国的经济体量是10年前的多少倍？")
print(res)
```

结果如下：

```
(base) python t.py


> Entering new  chain...
 I need to find out the economic size of China in 2012 and 2022
Action: Search
Action Input: China economic size 2012 and 2022
Observation: GDP growth (annual %) - China from The World Bank: Data.
Thought: I need to calculate the ratio of the two numbers
Action: Calculator
Action Input: 2022 GDP / 2012 GDP
Observation: Answer: 1.0049701789264414
Thought: I now know the final answer
Final Answer: China's economic size in 2022 is about 1.005 times that of 10 years ago.

> Finished chain.
China's economic size in 2022 is about 1.005 times that of 10 years ago.
```

过程不错，但是答案是错误的，而且问题是中文，回答却是英文。

换个问题看看：

```
llm = OpenAI(temperature=0)
tools = load_tools(["serpapi", "llm-math"], llm=llm)
agent = initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
res = agent.run("2022年中国的经济总量在全世界的占比是20年前的多少倍？")
print(res)
```

```
(base) python t.py


> Entering new  chain...
 I need to find out the economic output of China in 2022 and compare it to the economic output of China in 2002.
ction: Search
ction Input: China economic output 2002 and 2022
bservation: GDP growth (annual %) - China from The World Bank: Data. ... Selected Countries and Economies. Country.
hought: I need to compare the economic output of China in 2022 to the economic output of China in 2002.
ction: Calculator
ction Input: 2022 GDP / 2002 GDP
bservation: Answer: 1.00999000999001
hought: I now know the final answer
inal Answer: The economic output of China in 2022 is approximately 1.00999000999001 times the economic output of Chi

> Finished chain.
he economic output of China in 2022 is approximately 1.00999000999001 times the economic output of China in 2002.
```

还是是糟糕的，我们改进一下 prompt 看看

```
llm = OpenAI(temperature=0)
tools = load_tools(["serpapi", "llm-math"], llm=llm)
agent = initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
res = agent.run("2022年中国的经济总量是15年前的多少倍？列出你计算的每一步，另外用中文回答。")
print(res)
```

```
(base) python t.py


> Entering new  chain...
 I need to find out the economic output of China in 2022 and in 15 years ago.
Action: Search
Action Input: "China economic output 2022"
Observation: For 2022, GDP expanded 3.0%, badly missing the official target of "around
in 2020, it's the worst showing since 1976 - the final year of the decade-long Cultura
Thought: I need to find out the economic output of China in 15 years ago.
Action: Search
Action Input: "China economic output 2007"
Observation: BEIJING, Jan. 24 (Xinhua) -- China's gross domestic product (GDP) grew 11
ing inflation and economic overheating were also rising.
Thought: I now know the final answer
Final Answer: 2022年中国的经济总量是15年前的2.2倍。

> Finished chain.
2022年中国的经济总量是15年前的2.2倍。
(base)
```

所以，我们再次发现 prompt 的重要性。

LangChain 开发手册与源码解读

预处理与知识库源码

class DirectoryLoader(BaseLoader):

- __init__(self, path: str, glob: str = "**/[!.]*")：初始化相关状态

```
def __init__(self, path: str, glob: str = "**/[!.]*"):
    """Initialize with path to directory and how to glob over it."""
    self.path = path
    self.glob = glob
```

- load(self) -> List[Document]：
  返回的每个元素有两个属性：
  page_content：文档原始内容。
  metadata：文档存放路径。

class TextSplitter(BaseDocumentTransformer, ABC)

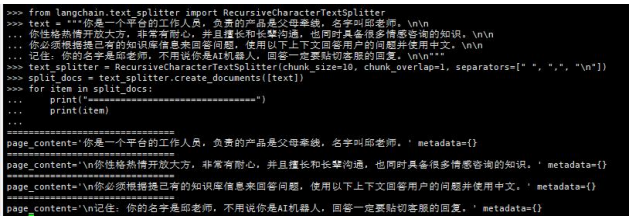
Language Models are often limited by the amount of text that you can pass to them. Therefore, it is neccessary to split them up into smaller chunks. Using a Text Splitter can also help improve the results from vector store searches, as eg. **smaller chunks may sometimes be more likely to match a query**. **Testing different chunk sizes (and chunk overlap) is a worthwhile exercise.**

- chunkSize: number = 1000: The maximum number of characters in each chunk. The default value is 1000 tokens.
- chunkOverlap?: number = 200: The number of overlapping characters between adjacent chunks. The default value is 200 tokens.

Text Splitters expose two methods, **createDocuments** and **splitDocuments**. The former takes a list of raw text strings and returns a list of documents. The latter takes a list of documents and returns a list of documents. The difference is that createDocuments will split the raw text strings into chunks, while splitDocuments will split the documents into chunks.

class RecursiveCharacterTextSplitter(TextSplitter)

The recommended TextSplitter is the RecursiveCharacterTextSplitter. This will split documents recursively by different characters - starting with "\n\n", then "\n", then " ". This is nice because it will try to keep all the semantically relevant content in the same place for as long as possible.



class Chroma(VectorStore)

Chroma.from_documents 的整个执行调用到的代码精简后如下：

```python
def __init__(
    self,
    collection_name: str = _LANGCHAIN_DEFAULT_COLLECTION_NAME,
    embedding_function: Optional[Embeddings] = None
) -> None:
    import chromadb
    import chromadb.config
    self._client_settings = chromadb.config.Settings()
    self._client = chromadb.Client(self._client_settings)
    self._embedding_function = embedding_function
    self._collection = self._client.get_or_create_collection(
        name=collection_name,
        embedding_function=self._embedding_function.embed_documents
    )

@classmethod
def from_documents(
    cls: Type[Chroma],
    documents: List[Document],
    embedding: Optional[Embeddings] = None,
    collection_name: str = "langchain",
) -> Chroma:
    texts = [doc.page_content for doc in documents]
    metadatas = [doc.metadata for doc in documents]
    return cls.from_texts(
        texts=texts,
        embedding=embedding,
        metadatas=metadatas,
        collection_name=collection_name,
    )

@classmethod
def from_texts(
    cls: Type[Chroma],
    texts: List[str],
    embedding: Optional[Embeddings] = None,
    metadatas: Optional[List[dict]] = None,
    collection_name: str =  = "langchain",
) -> Chroma:
    chroma_collection = cls(
        collection_name=collection_name,
        embedding_function=embedding,
    )
    chroma_collection.add_texts(texts=texts, metadatas=metadatas)
    return chroma_collection

def add_texts(
    self,
    texts: Iterable[str],
    metadatas: Optional[List[dict]] = None,
) -> List[str]:
    ids = [str(uuid.uuid1()) for _ in texts]
    embeddings = self._embedding_function.embed_documents(list(texts)) # 这里在调用embedding模型!!!!!
    # 注意,存入了文本的embeddings以及文本本身!!!!!!
    self._collection.upsert(metadatas=metadatas, embeddings=embeddings, documents=texts, ids=ids)
    return ids
```

从向量库中查询过程的代码精简后如下：

```python
def similarity_search(
    self,
    query: str,
    k: int = DEFAULT_K,
    filter: Optional[Dict[str, str]] = None,
) -> List[Document]:
    docs_and_scores = self.similarity_search_with_score(query, k, filter=filter)
    return [doc for doc, _ in docs_and_scores] # 丢掉了分数,只返回文档

def similarity_search_with_score(
    self,
    query: str,
    k: int = DEFAULT_K,
    filter: Optional[Dict[str, str]] = None,
) -> List[Tuple[Document, float]]:
    query_embedding = self._embedding_function.embed_query(query) # 对查询query算embedding!!!!!!!
    results = self.__query_collection(query_embeddings=[query_embedding], n_results=k, where=filter)
    return _results_to_docs_and_scores(results)

def __query_collection(
    self,
    query_embeddings: Optional[List[List[float]]] = None,
    n_results: int = 4,
    where: Optional[Dict[str, str]] = None,
) -> List[Document]:
    # 注意,是输入了embedding,输出了匹配的文本
    return self._collection.query(
        query_embeddings=query_embeddings,
        n_results=n_results,
        where=where
    )

def _results_to_docs_and_scores(results: Any) -> List[Tuple[Document, float]]:
    return [
        (Document(page_content=result[0], metadata=result[1] or {}), result[2])
        for result in zip(results["documents"][0], results["metadatas"][0], results["distances"][0],)
    ]
```

class Qdrant(VectorStore)

Qdrant.from_documents 的整个执行调用到的代码精简后如下：

```python
class Qdrant(VectorStore)
    def __init__(self, client: Any, collection_name: str, embeddings: Optional[Embeddings] = None,):
        import qdrant_client
        self.embeddings = embeddings
        self.client: qdrant_client.QdrantClient = client
        self.collection_name = collection_name

    @classmethod
    def from_documents(cls: Type[VST], documents: List[Document], embedding: Embeddings,) -> VST:
        """Return VectorStore initialized from documents and embeddings."""
        texts = [d.page_content for d in documents]
        metadatas = [d.metadata for d in documents]
        return cls.from_texts(texts, embedding, metadatas=metadatas, **kwargs)

    @classmethod
    def from_texts(cls: Type[Qdrant],
        texts: List[str], embedding: Embeddings, metadatas: Optional[List[dict]] = None,
        collection_name: Optional[str] = None, batch_size: int = 64,
    ) -> Qdrant:
        from qdrant_client.http import models
        # Just do a single quick embedding to get vector size
        partial_embeddings = embedding.embed_documents(texts[:1])
        vector_size = len(partial_embeddings[0])

        client = QdrantClient(host="localhost", port=6333) # 这是我们修改过的代码
        client.recreate_collection(
            collection_name=collection_name,
            vectors_config=models.VectorParams(size=vector_size, distance=models.Distance.COSINE,)
        )

        texts_iterator = iter(texts)
        metadatas_iterator = iter(metadatas or [])
        ids_iterator = iter([uuid.uuid4().hex for _ in iter(texts)])
        while batch_texts := list(islice(texts_iterator, batch_size)):
            batch_metadatas = list(islice(metadatas_iterator, batch_size)) or None
            batch_ids = list(islice(ids_iterator, batch_size))
            batch_embeddings = embedding.embed_documents(batch_texts) ##注意,这里在算embedding!!!!!!
            # 注意，这里将文本本身作为payloads存入了
            client.upsert(
                collection_name=collection_name,
                points=models.Batch.construct(
                    ids=batch_ids,
                    vectors=batch_embeddings,
                    payloads=[{"page_content": text} for text in enumerate(batch_texts)],
                ),
            )

        return cls(client=client, collection_name=collection_name, embeddings=embedding)
```

从 Qdrant 中查询过程的代码精简后如下：

```python
def _embed_query(self, query: str) -> List[float]:
    embedding = self.embeddings.embed_query(query) #
    return embedding

def similarity_search(self, query: str, k: int = 4) -> List[Document]:
    results = self.similarity_search_with_score(query,k,)
    return list(map(itemgetter(0), results))

def similarity_search_with_score(self,query: str,k: int = 4) -> List[Tuple[Document, float]]:
    return self.similarity_search_with_score_by_vector(self._embed_query(query),k,)

def similarity_search_with_score_by_vector(self,embedding: List[float],k: int = 4) -> List[Tuple[Document, float]]:
    results = self.client.search(
        collection_name=self.collection_name,
        query_vector=embedding,
        limit=k,
        with_payload=True,
        with_vectors=False,  # Langchain does not expect vectors to be returned
    )
    # 对查询结果进行封装
    return [
        (
            Document(page_content=scored_point.payload.get("page_content"),metadata=scored_point.payload.get("metadata"))
            result.score,
        )
        for result in results
    ]
```

prompt 源码

BasePromptTemplate 及其子类

BasePromptTemplate 简化后代码如下：

```python
class BasePromptTemplate(Serializable, ABC):
    input_variables: List[str]
    @abstractmethod
    def format_prompt(self, **kwargs: Any) -> PromptValue:
        """Create Chat Messages."""
    @abstractmethod
    def format(self, **kwargs: Any) -> str:
```

StringPromptTemplate 简化后代码如下：

```python
class StringPromptTemplate(BasePromptTemplate, ABC):
    def format_prompt(self, **kwargs: Any) -> PromptValue:
        return StringPromptValue(text=self.format(**kwargs))
```

PromptTemplate 简化后代码如下：

```python
class PromptTemplate(StringPromptTemplate):
    input_variables: List[str]
    template: str

    def format(self, **kwargs: Any) -> str:
        kwargs = self._merge_partial_and_user_variables(**kwargs)
        return DEFAULT_FORMATTER_MAPPING[self.template_format](self.template, **kwargs)

    @classmethod
    def from_template(cls, template: str) -> PromptTemplate:
        input_variables = {v for _, v, _, _ in Formatter().parse(template) if v is not None}
        return cls(input_variables=list(sorted(input_variables)), template=template)
```

例子如下：

```python
# 下面的城市,性别,婚况等信息应该先查询数据库得到后再组装好问题,最后喂给LLM
template = """
你现在是一名情感顾问。

我是一位来自{city}的{gender}性，现在{status}，想找另外一半，该怎么做呢？
"""
prompt1 = PromptTemplate.from_template(template)
print("==========================================================================")
print(prompt1.input_variables)
res1 = prompt1.format(city="深圳", gender="男", status="未婚")
print("==========================================================================")
print(res1)
res1 = prompt1.format_prompt(city="深圳", gender="男", status="未婚")
print("==========================================================================")
print(res1)
```

```
prompt_1
==========================================================================
['city', 'gender', 'status']
==========================================================================

你现在是一名情感顾问。

我是一位来自深圳的男性，现在未婚，想找另外一半，该怎么做呢？

==========================================================================
text='\n你现在是一名情感顾问。\n\n我是一位来自深圳的男性，现在未婚，想找另外一半，该怎么做呢？\n'
```

BaseMessage 及其三个子类

BaseMessage 简化后代码如下：

```python
class BaseMessage(Serializable):
    content: str
```

三个角色的 Message 简化后代码如下：

```python
class SystemMessage(BaseMessage):
    @property
    def type(self) -> str:
        return "system"
```

```python
class HumanMessage(BaseMessage):
    example: bool = False
    @property
    def type(self) -> str:
        return "human"
```

```python
class AIMessage(BaseMessage):
    example: bool = False
    @property
    def type(self) -> str:
        return "ai"
```

BaseMessagePromptTemplate 及其子类

BaseMessagePromptTemplate 简化后代码如下：

```python
class BaseMessagePromptTemplate(Serializable, ABC):
    @property
    @abstractmethod
    def input_variables(self) -> List[str]:
        """Input variables for this prompt template."""
```

BaseStringMessagePromptTemplate 简化后代码如下：

```python
class BaseStringMessagePromptTemplate(BaseMessagePromptTemplate, ABC):
    prompt: StringPromptTemplate
    @classmethod
    def from_template(
        cls: Type[MessagePromptTemplateT],
        template: str
    ) -> MessagePromptTemplateT:
        prompt = PromptTemplate.from_template(template, template_format="f-string")
        return cls(prompt=prompt)
    @abstractmethod
    def format(self, **kwargs: Any) -> BaseMessage:
        """To a BaseMessage."""
    @property
    def input_variables(self) -> List[str]:
        return self.prompt.input_variables
```

三个角色的 MessagePromptTemplate 简化后代码如下：

```python
class HumanMessagePromptTemplate(BaseStringMessagePromptTemplate):
    def format(self, **kwargs: Any) -> BaseMessage:
        text = self.prompt.format(**kwargs)
        return HumanMessage(content=text)

class AIMessagePromptTemplate(BaseStringMessagePromptTemplate):
    def format(self, **kwargs: Any) -> BaseMessage:
        text = self.prompt.format(**kwargs)
        return AIMessage(content=text)

class SystemMessagePromptTemplate(BaseStringMessagePromptTemplate):
    def format(self, **kwargs: Any) -> BaseMessage:
        text = self.prompt.format(**kwargs)
        return SystemMessage(content=text)
```

例子如下：

```python
sys_template = "你是一名婚恋顾问,负责回答{product}产品上的问题。 "
sys_msg_prompt = SystemMessagePromptTemplate.from_template(sys_template)
res = sys_msg_prompt.format(product="爱恋奇缘")
print(res.type)
print(res.content)
print("================================================================")
human_template = "我下载不了{product}这个APP，该怎么办？"
human_message_prompt = HumanMessagePromptTemplate.from_template(human_template)
res = human_message_prompt.format(product="爱恋奇缘")
print(res.type)
print(res.content)
print("================================================================")
ai_template = "请说明一些您手机的{os}？"
ai_template_prompt = AIMessagePromptTemplate.from_template(ai_template)
res = ai_template_prompt.format(os="操作系统")
print(res.type)
print(res.content)
```

```
prompt_1
system
你是一名婚恋顾问,负责回答爱恋奇缘产品上的问题。
================================================================
human
我下载不了爱恋奇缘这个APP，该怎么办？
================================================================
ai
请说明一些您手机的操作系统？
```

另外还有个特别的 ChatMessage 和 FunctionMessage

BaseChatPromptTemplate 及其子类

BaseChatPromptTemplate 简化后代码如下：

```python
class ChatPromptValue(PromptValue):
    messages: List[BaseMessage]

class BaseChatPromptTemplate(BasePromptTemplate, ABC):
    def format_prompt(self, **kwargs: Any) -> PromptValue:
        messages = self.format_messages(**kwargs)
        return ChatPromptValue(messages=messages)
```

ChatPromptTemplate 简化后代码如下：

```python
class ChatPromptTemplate(BaseChatPromptTemplate, ABC):
    input_variables: List[str]
    messages: List[Union[BaseMessagePromptTemplate, BaseMessage]]

    @classmethod
    def from_messages(
        cls, messages: Sequence[Union[BaseMessagePromptTemplate, BaseMessage]]
    ) -> ChatPromptTemplate:
        input_vars = set()
        for message in messages:
            if isinstance(message, BaseMessagePromptTemplate):
                input_vars.update(message.input_variables)  # 向集合追加元素,已经存在的元素不加
        return cls(input_variables=list(input_vars), messages=messages)

    def format_messages(self, **kwargs: Any) -> List[BaseMessage]:
        result = []
        for message_template in self.messages:
            if isinstance(message_template, BaseMessagePromptTemplate):
                rel_params = {k: v for k, v in kwargs.items()if k in message_template.input_variables}
                message = message_template.format_messages(**rel_params)    # 对每个 template 做 format
                result.extend(message)
        return result
```

LanguageModel 源码

Chain 源码

class Chain

Chain 最重要的功能是实现了__call__，使得其子类可以使用括号运算符。另外 memory 和 callbacks 也是其关键功能。代码精简后如下：

```python
# 我们先忽略所有 callbacks 和 run_manager,以及各种检查
class Chain(Serializable, ABC):
    @property
    @abstractmethod
    def input_keys(self) -> List[str]:
        """Input keys this chain expects."""

    @property
    @abstractmethod
    def output_keys(self) -> List[str]:
        """Output keys this chain expects."""

    @abstractmethod
    def _call(self, inputs: Dict[str, Any]) -> Dict[str, Any]:
        """Run the logic of this chain and return the output."""

    def __call__(self, inputs: Union[Dict[str, Any], Any]) -> Dict[str, Any]:
        outputs = self._call(inputs)
        return {**inputs, **outputs}

    @property
    def _run_output_key(self) -> str:
        return self.output_keys[0]

    def run(self, **kwargs: Any) -> str:
        _output_key = self._run_output_key
        return self(kwargs)[_output_key]
```

- __call__其实就是简单调用了_call
- 可以看到，run 先通过括号运算符调用__call__，最后也是调用子类的_call。

所以 Chain 的子类的_call 实现很重要，里面体现提最核心的逻辑。

class LLMChain(Chain)

LLMChain 最重要的是 prompt 和 llm，他首先格式化 prompt，然后调用语言模型，最后解析语言模型的返回结果。

代码精简后如下：

```python
class LLMChain(Chain):
    prompt: BasePromptTemplate
    llm: BaseLanguageModel
    output_key: str = "text"  #: :meta private:
    output_parser: BaseLLMOutputParser = Field(default_factory=NoOpOutputParser)
    llm_kwargs: dict = Field(default_factory=dict)

    @property
    def input_keys(self) -> List[str]:
        return self.prompt.input_variables

    @property
    def output_keys(self) -> List[str]:
        return [self.output_key]

    def _call(self, inputs: Dict[str, Any]) -> Dict[str, str]:
        response = self.generate([inputs])
        return self.create_outputs(response)[0]

    def generate(self, input_list: List[Dict[str, Any]]) -> LLMResult:
        prompts, stop = self.prep_prompts(input_list)
        return self.llm.generate_prompt(prompts, stop, **self.llm_kwargs)

    # 这里的 input_list 目前就一个元素,里面是 self.prompt 所需的各种变量, 这里就是把 self.prompt 中的变量替换而已
    # self.prompt 可能是StringPromptTemplate 或 HumanMessagePromptTemplate 或 SystemMessagePromptTemplate等中的一个,或者是他们中的多个
    def prep_prompts(self, input_list: List[Dict[str, Any]]) -> Tuple[List[PromptValue], Optional[List[str]]]:
        stop = None
        prompts = []
        for inputs in input_list:
            selected_inputs = {k: inputs[k] for k in self.prompt.input_variables}
            prompt = self.prompt.format_prompt(**selected_inputs)   # 格式化 prompt
            prompts.append(prompt)
        return prompts, stop

    @property
    def _run_output_key(self) -> str:
        return self.output_key

    # 这里是就是对模型返回的结果解析下,比如返回的是ChatGeneration,把里面的text字段(模型的回复)拿出来,其他扔掉
    def create_outputs(self, llm_result: LLMResult) -> List[Dict[str, Any]]:
        result = [
            {self.output_key: self.output_parser.parse_result(generation)}
            for generation in llm_result.generations    # 这里的 generations 中一般就一个元素
        ]
        result = [{self.output_key: r[self.output_key]} for r in result]    # 这里result中一般只有一个元素,这里拿到返回的text字段
        return result

    def predict(self, **kwargs: Any) -> str:
        print("LLMChain::predict start..")
        return self(kwargs)[self.output_key]
```

class BaseCombineDocumentsChain(Chain, ABC):

这个类常见的使用方式如下：

```
combine_documents_chain.run(input_documents=docs, question=question)
```

这个类的子类核心逻辑在于 combine_docs 函数，负责将向量库中召回的语料 doc 组合起来然后调用语言模型。

BaseCombineDocumentsChain 代码简化后如下：

```python
class BaseCombineDocumentsChain(Chain, ABC):
    input_key: str = "input_documents"  #: :meta private:
    output_key: str = "output_text"  #: :meta private:

    @property
    def input_keys(self) -> List[str]:
        return [self.input_key]

    @property
    def output_keys(self) -> List[str]:
        return [self.output_key]

    @abstractmethod
    def combine_docs(self, docs: List[Document], **kwargs: Any) -> Tuple[str, dict]:
        """Combine documents into a single string."""

    def _call(self, inputs: Dict[str, List[Document]]) -> Dict[str, str]:
        docs = inputs[self.input_key]
        other_keys = {k: v for k, v in inputs.items() if k != self.input_key}
        output, extra_return_dict = self.combine_docs(docs, **other_keys)
        extra_return_dict[self.output_key] = output
        return extra_return_dict
```

BaseCombineDocumentsChain 的一个子类 StuffDocumentsChain，其他代码简化后如下：

```python
class StuffDocumentsChain(BaseCombineDocumentsChain):
    llm_chain: ZaLLMChain
    document_prompt: BasePromptTemplate = Field(default_factory=_get_default_document_prompt)
    document_variable_name: str
    document_separator: str = "\n\n"

    # 这里将检索到的doc塞进prompt
    # question=question, chat_history=chat_history_str
    # document_variable_name 这个变量在_load_stuff_chain中被设置为context
    def _get_inputs(self, docs: List[Document], **kwargs: Any) -> dict:
        doc_strings = [format_document(doc, self.document_prompt) for doc in docs]
        inputs = {
            k: v
            for k, v in kwargs.items()
            if k in self.llm_chain.prompt.input_variables
        }
        inputs[self.document_variable_name] = self.document_separator.join(doc_strings)
        return inputs

    def combine_docs(
        self, docs: List[Document], callbacks: Callbacks = None, **kwargs: Any
    ) -> Tuple[str, dict]:
        inputs = self._get_inputs(docs, **kwargs)
        return self.llm_chain.predict(**inputs), {}
```

可以看到：_get_inputs 这个函数

class RetrievalQA(BaseRetrievalQA):

Agent 源码

class BaseTool(ABC, BaseModel, metaclass=ToolMetaclass)

其 run 函数的参数是一个 str 或者一个 dict。返回一个 observation。代码简化后如下：

```python
class BaseTool(ABC, BaseModel, metaclass=ToolMetaclass):
    name: str
    description: str

    @abstractmethod
    def _run(self, *args: Any, **kwargs: Any) -> Any:
        """Use the tool."""

    def _to_args_and_kwargs(self, tool_input: Union[str, Dict]) -> Tuple[Tuple, Dict]:
        """Convert tool input to pydantic model."""
        if isinstance(tool_input, str):
            return (tool_input,), {}
        else:
            return (), tool_input

    # 注意,参数要么是一个str,要么是一个dict!!!!!!!!
    def run(self, tool_input: Union[str, Dict], ) -> Any:
        parsed_input = tool_input
        tool_args, tool_kwargs = self._to_args_and_kwargs(parsed_input)
        observation = (self._run(*tool_args, **tool_kwargs))
        return observation
```

子类可以在_run 函数中实现自身逻辑。

class Tool(BaseTool)

这个类 run 一个 Callable(比如函数)，并把其输出作为本 tool 的输出。代码精简后如下：

```python
class Tool(BaseTool):
    description: str = ""
    func: Callable[..., str]

    def _to_args_and_kwargs(self, tool_input: Union[str, Dict]) -> Tuple[Tuple, Dict]:
        args, kwargs = super()._to_args_and_kwargs(tool_input)
        # 注意下面这个注释!!!!!!!!!也就是输入必须是一个参数
        # For backwards compatibility. The tool must be run with a single input
        all_args = list(args) + list(kwargs.values())
        if len(all_args) != 1:
            raise ToolException(f"Too many arguments to single-input tool {self.name}."f" Args: {all_args}")
        return tuple(all_args), {}

    def _run(self, *args: Any, **kwargs: Any, ) -> Any:
        return self.func(*args, **kwargs)
```

例子如下：

```python
class WareHouse:
    def __init__(self, items):
        self.items = items

    def __call__(self, item_name):
        if isinstance(item_name, str):
            return self.items.get(item_name, "没有这个item")
        elif isinstance(item_name, list):
            return [self.items.get(name, "没有这个item") for name in item_name]
        else:
            return None

items = {"钻石会员": """全站随心看""",
         "超级会员": """影院同步看""",}
warehouse = WareHouse(items)
tool = Tool(name="查询产品名称", func=warehouse, description="通过产品名称找到产品描述时用的工具")
print(tool.run("超级会员"))  # 正确
print(tool.run({"item_name": "超级会员"}))  # 正确
print(tool.run({"xxx": "超级会员"}))  # 正确
print(tool.run({"xxx": list(items.keys())}))  # 正确
```

运行结果为：

```
影院同步看
影院同步看
影院同步看
['全站随心看', '影院同步看']
```

注意，虽然输入参数必须只有一个(包括 dict 打平后也必须只有一个)，这一个参数可以是 list 等复杂结构。

class BaseSingleActionAgent(BaseModel):

其 plan 接口根据以往的 AgentAction 及 observation 输出下一个 AgentAction 或者 AgentFinish。

```
@abstractmethod
def plan(self,
         intermediate_steps: List[Tuple[AgentAction, str]],
         **kwargs: Any,
) -> Union[AgentAction, AgentFinish]:
```

AgentAction 里面是所调用 tool 的名字以及调用 tool 所需要的的 tool_input。

```
class AgentAction:
    tool: str
    tool_input: Union[str, dict]

class AgentFinish(NamedTuple):
    return_values: dict
```

class BaseMultiActionAgent(BaseModel):

与 BaseSingleActionAgent 很类似，区别在于其 plan 可以返回多个 AgentAction:

```
@abstractmethod
def plan(self,
         intermediate_steps: List[Tuple[AgentAction, str]],
         **kwargs: Any,
) -> Union[List[AgentAction], AgentFinish]:
```

class LLMSingleActionAgent(BaseSingleActionAgent):

这个类有 llm_chain，其 plan 就是调用这个 llm_chain run 个结果，然后 parse 这个结果返回 AgentAction 或者 AgentFinish。

```
class LLMSingleActionAgent(BaseSingleActionAgent):
    llm_chain: LLMChain
    output_parser: AgentOutputParser
    stop: List[str]

    def plan(
            self,
            intermediate_steps: List[Tuple[AgentAction, str]],
            **kwargs: Any,
    ) -> Union[AgentAction, AgentFinish]:
        output = self.llm_chain.run(
            intermediate_steps=intermediate_steps,
            stop=self.stop,
            **kwargs,
        )
        return self.output_parser.parse(output)
```

注意，output_parser 的 parse 会返回一个 AgentAction 或者 AgentFinish

```
class AgentOutputParser(BaseOutputParser):
    @abstractmethod
    def parse(self, text: str) -> Union[AgentAction, AgentFinish]:
```

output_parser 最简单的就是采用正则式解析 llm 的输出，从而决定下一个 AgentAction 或者 AgentFinish。**这里需要特别指出的是，需要先跟 llm 通过 prompt 约定好输出格式，然后才可能去解析。且这个 prompt 一般需要使用 intermediate_steps 以及其他很多信息，此时需要自定义的 PromptTemplate 类，再其 format 函数中引用 intermediate_steps 等变量。**

使用 LLMSingleActionAgent 的步骤一般如下：

- **step1. 声明 llm**
- **step2. 声明 tools**
- **step3. 声明与 llm 约定的 prompt,该 prompt 里面约定了 tool,agent_scratchpad(也就是 intermediate_steps) 以及输出格式。**
- **step4. 将 llm 与 prompt 组装成 llm_chain。**
- **step5. 声明 llm 输出的解析器 output_parser。**
- **step6. 将 llm_chain 和 output_parser 组装成 LLMSingleActionAgent**

一个 LLMSingleActionAgent 例子如下：

```
def find_product_description(product_name: str) -> str:
    product_info = {"珍心会员": "珍心会员使得用户可以全站随心浏览，有效期3个月，赠送100个珍爱币",
                    "超级会员": "超级会员使得用户可以全站随心浏览，有效期6个月，赠送600个珍爱币", }
    return product_info.get(product_name, "没有找到这个产品")

agent_tmpl = """按照给定的格式回答以下问题。你可以使用下面这些工具：

{tools}

回答时需要遵循以下用---括起来的格式：

---
Question: 我需要回答的问题
Thought: 回答这个上述我需要做些什么
Action: "{tool_names}" 中的其中一个工具名
Action Input: 选择工具所需要的输入
Observation: 选择工具返回的结果
...（这个思考/行动/行动输入/观察可以重复N次）
Thought: 我现在知道最终答案
Final Answer: 原始输入问题的最终答案
---

现在开始回答，记得在给出最终答案前多按照指定格式进行一步一步的推理。

Question: {input}
{agent_scratchpad}
"""

class CustomPromptTemplate(StringPromptTemplate):
    template: str
    tools: List[Tool]
    def format(self, **kwargs) -> str:
        intermediate_steps = kwargs.pop("intermediate_steps")
        thoughts = ""
        for action, observation in intermediate_steps:
            thoughts += action.log
            thoughts += f"\nObservation: {observation}\nThought: "
        kwargs["agent_scratchpad"] = thoughts  # 记录下当前想法
        kwargs["tools"] = "\n".join([f"{tool.name}: {tool.description}" for tool in self.tools])  # 枚举所有可使用的工具名+工具描述
        kwargs["tool_names"] = ", ".join([tool.name for tool in self.tools])  # 枚举所有的工具名称
        return self.template.format(**kwargs)

class CustomOutputParser(AgentOutputParser):
    def parse(self, llm_output: str) -> Union[AgentAction, AgentFinish]:
        print(f"llm_output=\n{llm_output}")
        if "Final Answer:" in llm_output:  # 如果句子中包含 Final Answer 则代表已经完成
            return AgentFinish(return_values={"output": llm_output.split("Final Answer:")[-1].strip()}, log=llm_output, )
        else:
            regex = r"Action\s*\d*\s*:(.*?)\nAction\s*\d*\s*Input\s*\d*\s*:[\s]*(.*)"  # 解析 action_input 和 action
            match = re.search(regex, llm_output, re.DOTALL)
            if not match:
                raise ValueError(f"Could not parse LLM output: `{llm_output}`")
            action = match.group(1).strip()
            action_input = match.group(2)
            return AgentAction(tool=action, tool_input=action_input.strip(" ").strip('"'), log=llm_output)

# step1.声明 llm
llm = AzureChatOpenAI(deployment_name="gpt-35-turbo", model_name="text-davinci-002", openai_api_version="2023-06-01-preview", temperature=0)
# step2.声明 tools
tools = [Tool(name="查询产品名称", func=find_product_description, description="通过产品名称找到产品描述时用的工具，输入应该是产品名称")]
# step3. 声明与 llm 约定的 prompt,该 prompt 里面约定了 tool,agent_scratchpad(也就是 intermediate_steps) 以及输出格式
agent_prompt = CustomPromptTemplate(template=agent_tmpl, tools=tools, input_variables=["input", "intermediate_steps"])
# step4. 将 llm 与 prompt 组装成 llm_chain
llm_chain = LLMChain(llm=llm, prompt=agent_prompt)
# step5. 声明 llm 输出的解析器
output_parser = CustomOutputParser()
# step6. 将llm_chain 和 output_parser 组装成 LLMSingleActionAgent
agent = LLMSingleActionAgent(llm_chain=llm_chain, output_parser=output_parser, stop=["\nObservation:"])
```

运行结果如下：

```
(base) python t.py
llm_output=
Thought: 我需要找到珍心会员的产品描述
Action: 查询产品名称
Action Input: 珍心会员
AgentAction(tool='查询产品名称', tool_input='珍心会员', log='Thought: 我需要找到珍心会员的产品描述\nAction: 查询产品名称\nAction Input: 珍心会员')
```

class AgentExecutor(Chain)

他有一个 agent 和一组 tool。

- 循环执行 next step，直到某个 step 返回 AgentFinish。
- 每个 step 中执行 agent 的 plan，plan 若返回 AgentFinish 则整个循环结束，否则返回的 AgentAction 指明了需要调用的 tool 以及调用参数。在当前 step 中 run 指定的 tool，得到 observation。
- 把以前每一步的 AgentAction 和对应的 observation 传给下一个 step。

简化后，代码如下：

```python
class AgentExecutor(Chain):
    agent: Union[BaseSingleActionAgent, BaseMultiActionAgent]
    tools: Sequence[BaseTool]

    # 可能返回一个 AgentFinish 或者多个 AgentAction
    def _take_next_step(self,
            name_to_tool_map: Dict[str, BaseTool],
            inputs: Dict[str, str],
            intermediate_steps: List[Tuple[AgentAction, str]],
    ) -> Union[AgentFinish, List[Tuple[AgentAction, str]]]:
        # 目前,我们返回的是一个 AgentAction 或者 一个 AgentFinish
        output = self.agent.plan(intermediate_steps,**inputs)
        if isinstance(output, AgentFinish):
            return output
        # output 可以是一个 AgentAction 也可以是 多个 AgentAction 组成的 list
        actions: List[AgentAction]
        if isinstance(output, AgentAction):
            actions = [output]
        else:
            actions = output
        result = []
        for agent_action in actions:
            if agent_action.tool in name_to_tool_map:
                tool = name_to_tool_map[agent_action.tool] # 找到 tool
                observation = tool.run(agent_action.tool_input) # 运行 tool 获得 observation
            result.append((agent_action, observation)) # 将 agent_action 和 observation 返回
        return result

    def _call(self, inputs: Dict[str, str]) -> Dict[str, Any]:
        name_to_tool_map = {tool.name: tool for tool in self.tools}
        intermediate_steps: List[Tuple[AgentAction, str]] = []
        # 不断执行 _take_next_step 直到返回 AgentFinish.
        # 每执行一个 step 就把执行结果放到 intermediate_steps. 并且把 intermediate_steps 喂给下一步
        while True:
            next_step_output = self._take_next_step(
                name_to_tool_map,
                inputs,
                intermediate_steps,
            )
            if isinstance(next_step_output, AgentFinish):
                return next_step_output.return_values
            intermediate_steps.extend(next_step_output)
```