LangChain 模块解析（中）

LangChain Chains Hands-on

Press Space for next page →

《AGI 应用实践》系列教程

AGI OVERVIEW

- 101：AGI 提示工程指北
- 102：OpenAI API 浅出

LANGCHAIN IN ACTION

- 201：LangChain 功能模块解析（上篇）
- 202：LangChain 功能模块解析（中篇）
- 203：LangChain 功能模块解析（下篇）

教程大纲

- Runnable Sequence
  - Runnable 对象简介
  - Runnable 对象的串联
  - Runnable 对象绑定参数
  - Runnable Map
  - Runnable Passthrough
  - Tool 也是一种 Runnable 对象
  - 在 Runnable Sequence 中引入 Memory
  - 为 Runnable Sequence 提供条件路由
  - Runnable 的 Fallback 机制
- Chains
  - 链路调试的“文韬武略”
  - 一条链路贯天地
  - 两类模型舞前沿
  - 三种巧思治文档
  - 四套应用疏通途：Retrieval QA
  - 四套应用疏通途：Conversational Retrieval QA
  - 四套应用疏通途：SQL QA
  - 四套应用疏通途：Web API
  - 种类繁多的“预制菜”

LangChain JS Runnable Chain Cheatsheet v0.1.1

Authored by Haili Zhang @zhanghaili0610
Reviewed by Jacob Lee @Hacubu
(Last Updated: Aug 23, 2023)

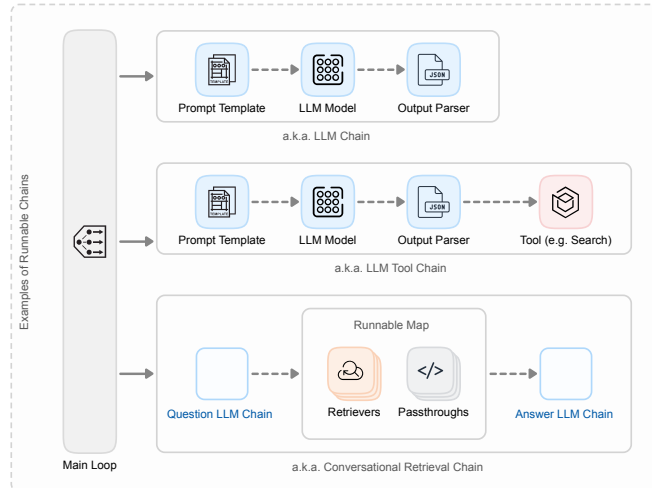
Concept: Runnable
Objects or functions that expose standard interfaces:
- stream: stream back chunks of the response
  e.g. *model* = new OpenAI({ streaming: true })
  e.g. *parser* = new BytesOutputParser()
- invoke: call the chain on an input
- batch: call the chain on a list of inputs

How-to: Chain Runnables
- Instance Method: *runnable*.pipe(*runnable*)
- Static Method: RunnableSequence.from([...*runnables*]),
which run runnable objects in sequence when invoked

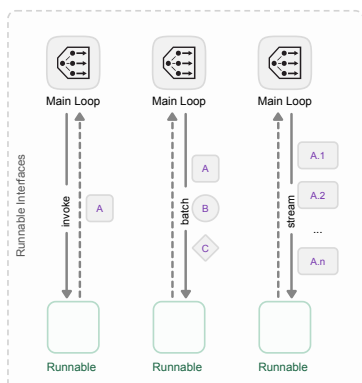
How-to: Passthrough Chain Inputs
- If input is string, use new RunnablePassthrough()
- If input is object, use arrow (=>) function which
takes the object as input and extracts the desired key

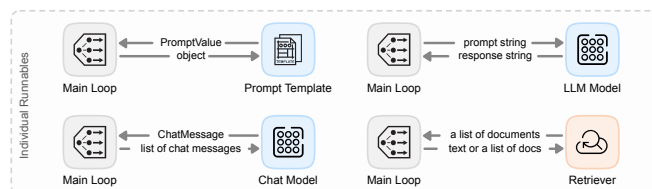
How-to: Bind kwargs (keyword args)
- Instance Method: *runnable*.bind({ ...kwargs })
- e.g. Bind Functions to OpenAI Model:
model.bind({ functions: [...schemas], function_call: { ... } })

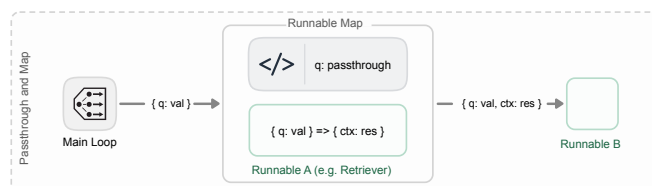
How-to: Fallback to another Runnable
- Instance: *runnable*.withFallbacks({ fallbacks: [...*runnables*] })



LangChain JS Expression Language Cookbook: https://js.langchain.com/docs/guides/expression_language/cookbook







您当前的浏览器不支持 HTML5 播放器

请更换浏览器再试试哦~

Runnable Sequence

LangChain Expression Language (LCEL) 的核心

Runnable 对象简介

把一个个 Runnable 对象串联起来，就构成了 Runnable Sequence

Runnable 对象的定义：支持以下三个实例方法的对象

- `invoke`：对输入直接进行链式调用
- `batch`：对输入列表进行批量的链式调用
- `stream`：（流式）分块返回响应（需要 Model、Parser 等支持）

Runnable 对象的输入输出：各有不同，并非统一（以下以 JS/TS SDK 为例）

- Prompt：`Object` ➡️ `PromptValue`
- LLM Model：`String` ➡️ `String`
- Chat Model：一组 `ChatMessage` ➡️ `ChatMessage`
- Parser：Model 的输出 ➡️ 不同 Parser 对应的数据结构
- Retriever：`String` ➡️ 一组 `Document`

Runnable 对象的串联

- JS/TS 中可用的实例方法：`runnable.pipe(runnable)`
- JS/TS 中可用的静态方法：`RunnableSequence.from([...runnables ])`

```
import { PromptTemplate } from "langchain/prompts";
import { OpenAI } from "langchain/llm/openai";
import { RunnableSequence } from "langchain/schema/runnable";
import { StringOutputParser } from "langchain/schema/output_parser";

/* 准备 Runnable 对象 */
const model = new OpenAI({});
const promptTemplate = PromptTemplate.fromTemplate("Tell me a joke about {topic}");
const outputParser = new StringOutputParser();

/* 串联 Runnable 对象 */
// const chain = promptTemplate.pipe(model).pipe(outputParser);
const chain = RunnableSequence.from([promptTemplate, model, outputParser]);

/* 调用 Runnable Sequence */
const result = await chain.invoke({ topic: "bears" });
// const result = await chain.batch([{ topic: "bears" }, { topic: "dears" }]);
```

Run: RunnableSequence

Playground

Trace

RunnableSequence • Success

- Prompt PromptTemplate
- LLM OpenAI
- Parser StrOutputParser

RunnableSequence

Run | Feedback | Metadata

INPUT Copy

```
1 topic: bears
```

YAML

OUTPUT Copy

Q: What do you call a bear with no teeth?
A: A gummy bear!

Runnable 对象绑定参数

- JS/TS 中可用的实例方法：`runnable.bind( ...kwargs )`

```
import { PromptTemplate } from "langchain/prompts";
import { ChatOpenAI } from "langchain/chat_models/openai";

/* 准备 Runnable 对象 */
const prompt = PromptTemplate.fromTemplate(`Tell me a joke about {subject}`);
const model = new ChatOpenAI({});

/* 准备 Runnable 对象的附加参数 */
const functionSchema = [
  {
    name: "joke",
    description: "A joke",
    parameters: {
      type: "object",
      properties: {
        setup: {
          type: "string",
          description: "The setup for the joke",
        },
        punchline: {
          type: "string"
```

Run: RunnableSequence

Playground

Trace

RunnableSequence • Success

Prompt PromptTemplate

LLM ChatOpenAI

RunnableSequence

Run Feedback Metadata

INPUT Copy

```
1 subject: bears
```

YAML

OUTPUT Copy

```
1 id:
2   - langchain
3   - schema
4   - AIMessage
5 lc: 1
6 type: constructor
```

Runnable Map

当 Runnable 对象接收的输入需要被预处理时，可以使用一个 Map 结构提供支持

- Map 中的每个属性都接收相同的参数，使用这些参数来调用各个属性对应的 Runnable 对象或函数
- 调用的返回值用于填充 Map 对象，然后将该对象传递到序列中的下一个 Runnable 对象

```
import { PromptTemplate } from "langchain/prompts";
import { RunnableSequence } from "langchain/schema/runnable";
import { StringOutputParser } from "langchain/schema/output_parser";
import { OpenAI } from "langchain/chat_models/openai";

const prompt1 = PromptTemplate.fromTemplate(`What is the city {person} is from? Only respond with city name.`);
const prompt2 = PromptTemplate.fromTemplate(`What country is the city {city} in? Respond in {language}.`);

/* 构建第一个 Chain */
const model = new OpenAI({});
const chain = prompt1.pipe(model).pipe(new StringOutputParser());

const combinedChain = RunnableSequence.from([
  /* 将第一个 Chain 的返回值和原始输入中的部分数据进行组合 */
  {
    city: chain,
    language: (input) => input.language,
  },
  /* 组合后的对象成为第二个 Chain 的输入 */
```

Run: RunnableSequence

Playground

Trace

- RunnableSequence • Success
  - Chain RunnableMap
    - Chain RunnableSequence
      - Prompt PromptTemplate
      - LLM OpenAI
      - Parser StrOutputParser
    - Chain RunnableLambda
  - Prompt PromptTemplate
  - LLM OpenAI
  - Parser StrOutputParser

RunnableSequence

Run | Feedback | Metadata

INPUT

Copy

```yaml
person: Chairman Mao
language: Chinese
```

YAML

OUTPUT

Copy

中国湖南省韶山

Runnable Passthrough

当我们要从一个长长的 Runnable Sequence 中提取最头部的输入时，可以使用 Passthrough 透传机制

```
import { ChatOpenAI } from "langchain/chat_models/openai";
import { HNSWLib } from "langchain/vectorstores/hnswlib";
import { OpenAIEmbeddings } from "langchain/embeddings/openai";
import { PromptTemplate } from "langchain/prompts";
import { RunnableSequence, RunnablePassthrough } from "langchain/schema/runnable";
import { StringOutputParser } from "langchain/schema/output_parser";
import { Document } from "langchain/document";

const model = new ChatOpenAI({});

/* 为 RAG 场景准备 Retriever */
const vectorStore = await HNSWLib.fromTexts(
  ["mitochondria is the powerhouse of the cell"],
  [{ id: 1 }],
  new OpenAIEmbeddings()
);
const retriever = vectorStore.asRetriever();

const prompt = PromptTemplate.fromTemplate(`Answer the question based only on the following context:
{context}
```

Run: RunnableSequence

Playground

Trace

- RunnableSequence • Success
  - Chain RunnableMap
    - Chain RunnableSequence
      - Retriever VectorStoreRetriev...
      - Chain RunnableLambda
    - Chain RunnablePassthrough
  - Prompt PromptTemplate
  - LLM OpenAI
  - Parser StrOutputParser

RunnableSequence

Run | Feedback | Metadata

INPUT Copy

```
1 input: What is pinecone?
```

YAML

OUTPUT Copy

Answer: Pinecone is a vector database.

Tool 也是一种 Runnable 对象

因此 Tool 是可以在 Runnable Sequence 中被串联的

```
import { SerpAPI } from "langchain/tools";
import { OpenAI } from "langchain/llm/anthropic";
import { PromptTemplate } from "langchain/prompts";
import { StringOutputParser } from "langchain/schema/output_parser";

const prompt = PromptTemplate.fromTemplate(`Turn the following user input into a search query for a search engine:

{input}`);

const model = new OpenAI({});

/* 准备 Web Search 工具 */
const search = new SerpAPI();

/* 直接将 Web Search 工具传入并调用 */
const chain = prompt.pipe(model).pipe(new StringOutputParser()).pipe(search);

const result = await chain.invoke({ input: "Who is the current prime minister of Malaysia?" });
```

Run: RunnableSequence

Playground

Trace

- RunnableSequence • Success
  - Prompt PromptTemplate
  - LLM OpenAI
  - Parser StrOutputParser
  - Tool SerpAPI

RunnableSequence

Run | Feedback | Metadata

INPUT Copy

```
1  input: Who is the current prime minister of Malaysia?
```

YAML

OUTPUT Copy

Anwar Ibrahim

在 Runnable Sequence 中引入 Memory

Memory 可以被添加到任何 Runnable Sequence 中，本质和就是生成上下文文本填充到提示词中

```
import { ChatPromptTemplate, MessagesPlaceholder } from "langchain/prompts";
import { RunnableSequence } from "langchain/schema/runnable";
import { ChatAnthropic } from "langchain/chat_models/anthropic";
import { BufferMemory } from "langchain/memory";

const model = new ChatAnthropic();
const prompt = ChatPromptTemplate.fromMessages([
  ["system", "You are a helpful chatbot"],
  new MessagesPlaceholder("history"),
  ["human", "{input}"],
]);

/* 初始化 Memory 对象 */
const memory = new BufferMemory({
  returnMessages: true,
  inputKey: "input",      // by default
  outputKey: "output",    // by default
  memoryKey: "history",   // by default
});

const chain = RunnableSequence.from([
```

为 Runnable Sequence 提供条件路由

路由允许您创建非确定性的 Chain，其中上一步的输出定义了下一步的路由方向

在 JS/TS 中可使用 `RunnableBranch` 来构建条件路由，路由会依次匹配条件直至选择默认 Runnable 返回：

```
const langChainChain = PromptTemplate.fromTemplate(
  `You are an expert in langchain. Always answer questions starting with "As Harrison Chase told me".
Respond to the following question:

Question: {question}
Answer:`
).pipe(model);

const anthropicChain = PromptTemplate.fromTemplate(
  `You are an expert in anthropic. Always answer questions starting with "As Dario Amodei told me".
Respond to the following question:

Question: {question}
Answer:`
).pipe(model);

const generalChain = PromptTemplate.fromTemplate(`Respond to the following question:

Question: {question}
```

Runnable 的 Fallback 机制

Fallback 机制可以提供优雅的异常处理，通过“以优充次”的方式尽可能保障链路执行继续执行

在 JS/TS 中，可使用 `runnable.withFallbacks({ fallbacks: [runnable] })` 来构建 Fallback 支撑

```
const fakeOpenAIModel = new ChatOpenAI({
  modelName: "potato!",   // Fake model name will always throw an error
  maxRetries: 0,
});

const modelWithFallback = fakeOpenAIModel.withFallbacks({
  fallbacks: [new ChatAnthropic({})],
});
```

不仅可以支撑 Runnable 对象，还可以直接 Fallback 整个 Runnable Sequence：

```
const badChain = chatPrompt.pipe(fakeOpenAIChatModel).pipe(outputParser);
const goodChain = prompt.pipe(openAILLM).pipe(outputParser);

const chain = badChain.withFallbacks({
  fallbacks: [goodChain],
});
```

Chains

LangChain 为“链式”应用提供了 Chain 接口

链路调试的“文韬武略”

除了接入强力的 LangSmith，也可以使用 Console 输出调试信息

和调试 Runnable Sequence 一样，首选的调试方式肯定还是接入 LangSmith，有两种方式：

```
export LANGCHAIN_TRACING_V2=true
export LANGCHAIN_ENDPOINT=https://api.smith.langchain.com
export LANGCHAIN_API_KEY=<your-api-key>  # still in closed beta
export LANGCHAIN_PROJECT=<your-project>  # if not specified, defaults to "default"
```

```
import { Client } from "langsmith";
import { LangChainTracer } from "langchain/callbacks";

const client = new Client({ apiUrl: "https://api.smith.langchain.com", apiKey: "YOUR_API_KEY" });
const tracer = new LangChainTracer({ projectName: "YOUR_PROJECT_NAME", client });

await model.invoke("Hello, world!", { callbacks: [tracer] })
```

退而求其次，可以开启 `verbose` 选项，有两种方式：

```
export LANGCHAIN_VERBOSE=true
```

```
const chain = new ConversationChain({ llm: chat, verbose: true });
```

一条链路贯天地

类似 Runnable Sequence 的链路构建，但相比之下 Sequential 操作的复杂度更加高

- `SimpleSequentialChain`：最简单形式，每个步骤都有单个输入/输出，一个步骤的输出是下一个的输入
- `SequentialChain`：顺序链的更通用形式，允许多个输入/输出

```
import { SequentialChain, LLMChain } from "langchain/chains";
import { OpenAI } from "langchain/llms/openai";
import { PromptTemplate } from "langchain/prompts";

// This is an LLMChain to write a synopsis given a title of a play and the era it is set in.
const llm = new OpenAI({ temperature: 0 });
const template = `You are a playwright. Given the title of play and the era it is set in, it is your job to write a sync

  Title: {title}
  Era: {era}
  Playwright: This is a synopsis for the above play:`;
const promptTemplate = new PromptTemplate({template, inputVariables: ["title", "era"] });
const synopsisChain = new LLMChain({ llm, prompt: promptTemplate, outputKey: "synopsis" });

// This is an LLMChain to write a review of a play given a synopsis.
const reviewTemplate = `You are a play critic from the New York Times. Given the synopsis of play, it is your job to wr

  Play Synopsis:
```

Run: SequentialChain

Playground

Trace

- SequentialChain • Success
  - Chain LLMChain
    - LLM OpenAI
  - Chain LLMChain
    - LLM OpenAI

SequentialChain

Run | Feedback | Metadata

INPUT

Copy

```yaml
era: Victorian England
title: Tragedy at sunset on the beach
review: |-


  Tragedy at Sunset on the Beach is a
captivating and heartbreaking story of love
and loss. Set in Victorian England, the play
follows Emily, a young woman struggling to
make ends meet in a small coastal town.
Emily's dreams of a better life are dashed
when she discovers her employer is involved
```

YAML

两类模型舞前沿

最基础的链式调用是`LLMChain`，它同时支持 LLM 和 Chat 模型

```
/* 构建基于 LLM 模型的 LLMChain */
const model = new OpenAI({ temperature: 0 });
const prompt = PromptTemplate.fromTemplate("What is a good name for a company that makes {product}?");
const chainA = new LLMChain({ llm: model, prompt });

// 通过 .call(object) 返回输出对象
const resA = await chainA.call({ product: "colorful socks" });  // { text: '\n\nSocktastic!' }
// 通过 .run(string) 返回输出字符串
const resA2 = await chainA.run("colorful socks");   // '\n\nSocktastic!'
```

```
/* 构建基于 Chat 模型的 LLMChain */
const chat = new ChatOpenAI({ temperature: 0 });
const chatPrompt = ChatPromptTemplate.fromMessages([
  ["system", "You are a helpful assistant that translates {input_language} to {output_language}."],
  ["human", "{text}"],
]);
const chainB = new LLMChain({ prompt: chatPrompt, llm: chat });

const resB = await chainB.call({ input_language: "English", output_language: "French", text: "I love programming." });
// { text: "J'adore la programmation." }
```

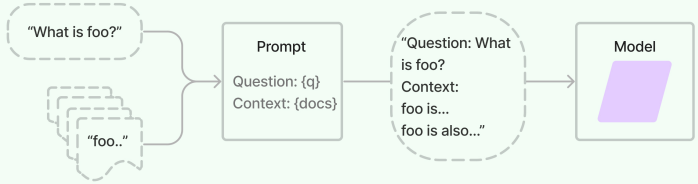
三种巧思治文档

LangChain 提供处理文档的工具链，它们对于总结文档、回答文档问题、从文档中提取信息等很有用

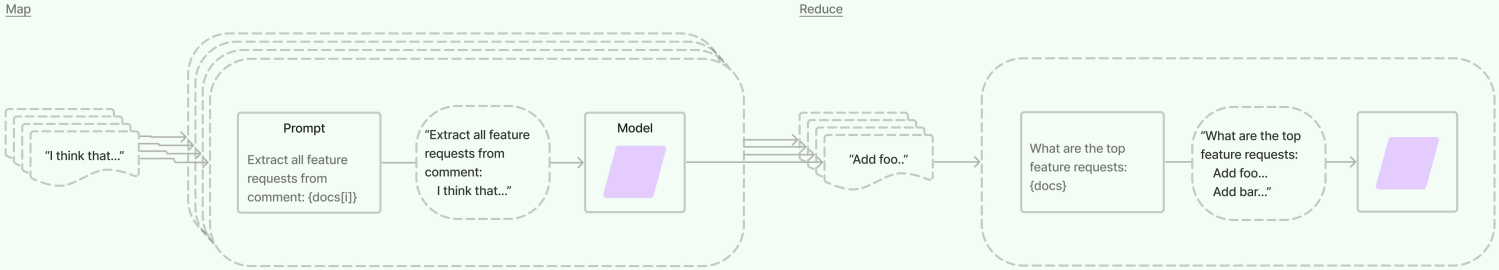
三种文档处理工具链 的使用方式基本是一样的，如下所示：

```
import { OpenAI } from "langchain/llms/openai";
import { loadQAStuffChain, loadQAMapReduceChain } from "langchain/chains";
import { Document } from "langchain/document";

const docs = [
  new Document({ pageContent: "Harrison went to Harvard." }),
  new Document({ pageContent: "Ankush went to Princeton." }),
];

/* 构建并使用 StuffDocumentsChain */
const llmA = new OpenAI({});
const chainA = loadQAStuffChain(llmA);
const resA = await chainA.call({ input_documents: docs, question: "Where did Harrison go to college?" });

/* 构建并使用 MapReduceChain */
const llmB = new OpenAI({ maxConcurrency: 10 });
const chainB = loadQAMapReduceChain(llmB);
const resB = await chainB.call({ input_documents: docs, question: "Where did Harrison go to college?" });
```

Stuff documents

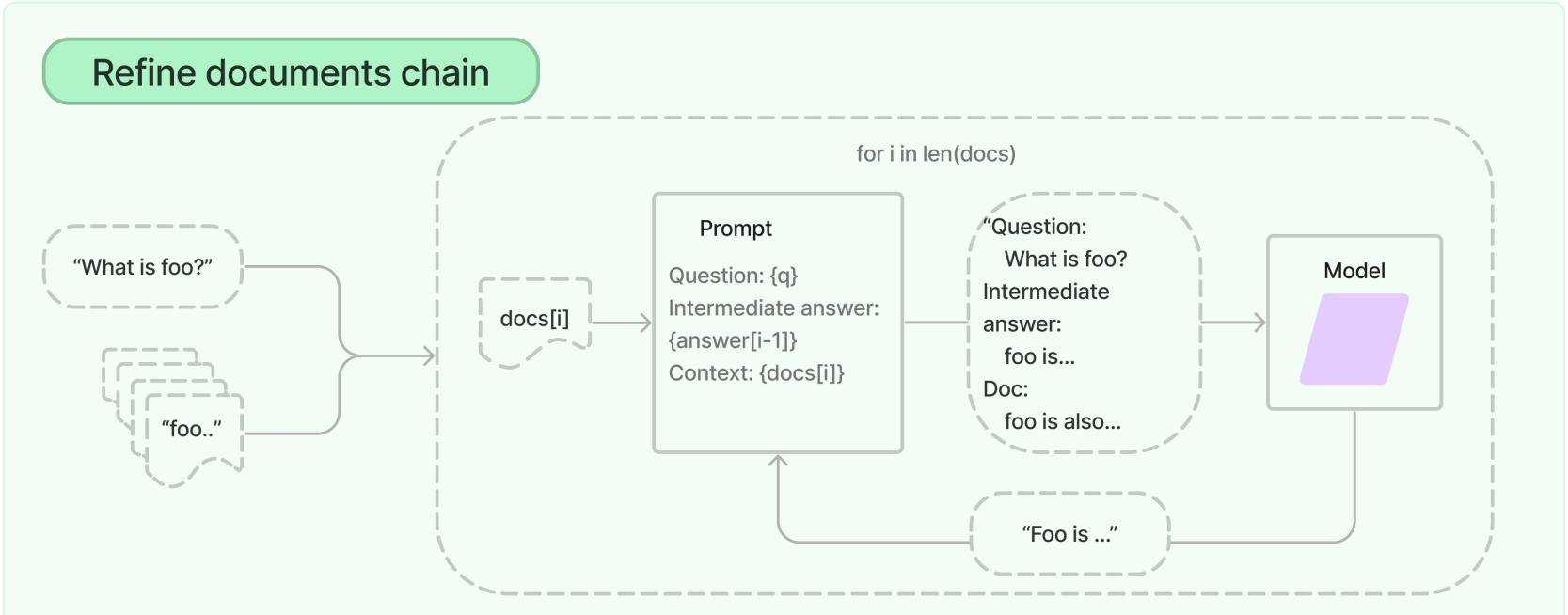


Map reduce documents chain



层叠递进的 Refine 模式

- **对于每个文档，它将所有非文档输入、当前文档以及最新的中间答案传递给 LLM 链以获得新的答案**
- 由于 Refine 链一次仅将单个文档传递给 LLM，因此很适合需要分析的文档数量多于模型上下文的任务
- 明显的缺点是：将比 Stuff 文档链进行更多的 LLM 调用；当文档频繁地互相交叉引用时很可能表现不佳



Refine 文档链的二阶段提示词应用

```
import { loadQARefineChain } from "langchain/chains";
import { OpenAI } from "langchain/llms/openai";
import { TextLoader } from "langchain/document_loaders/fs/text";
import { MemoryVectorStore } from "langchain/vectorstores/memory";
import { OpenAIEmbeddings } from "langchain/embeddings/openai";
import { PromptTemplate } from "langchain/prompts";

/* 最终提问时使用的提示词 */
export const questionPromptTemplateString = `Context information is below.
---------------------
{context}
---------------------
Given the context information and no prior knowledge, answer the question: {question}`;

const questionPrompt = new PromptTemplate({
  inputVariables: ["context", "question"],
  template: questionPromptTemplateString,
});

/* 中间 Refine 过程使用的提示词 */
const refinePromptTemplateString = `The original question is as follows: {question}
We have provided an existing answer: {existing_answer}
We have the opportunity to refine the existing answer
```

四套应用疏通途：Retrieval QA

Retrieval QA Chain 通过从检索器检索文档，然后使用文档工具链，根据检索到的文档回答问题

```
import { OpenAI } from "langchain/llms/openai";
import { RetrievalQAChain, loadQAStuffChain } from "langchain/chains";
import { HNSWLib } from "langchain/vectorstores/hnswlib";
import { OpenAIEmbeddings } from "langchain/embeddings/openai";
import { RecursiveCharacterTextSplitter } from "langchain/text_splitter";
import { PromptTemplate } from "langchain/prompts";
import * as fs from "fs";

const promptTemplate = `Use the following pieces of context to answer the question at the end. If you don't know the ans

{context}

Question: {question}
Answer in Italian:`;
const prompt = PromptTemplate.fromTemplate(promptTemplate);
const model = new OpenAI({});

/* 构建文档检索器 */
const text = fs.readFileSync("state_of_the_union.txt", "utf8");
const textSplitter = new RecursiveCharacterTextSplitter({ chunkSize: 1000 });
const docs = await textSplitter.createDocuments([text]);
const vectorStore = await HNSWLib.fromDocuments(docs, new OpenAIEmbeddings());
```

四套应用疏通途：Conversational Retrieval QA

Conversational Retrieval QA Chain 建立在 Retrieval QA Chain 的基础上，并接入 Memory 组件

它首先将聊天历史记录和问题组合成一个独立的问题，然后将检索得到的文档和问题传递给问答链以返回响应

```
import { ChatOpenAI } from "langchain/chat_models/openai";
import { ConversationalRetrievalQAChain } from "langchain/chains";
import { HNSWLib } from "langchain/vectorstores/hnswlib";
import { OpenAIEmbeddings } from "langchain/embeddings/openai";
import { BufferMemory } from "langchain/memory";

const CUSTOM_QUESTION_GENERATOR_CHAIN_PROMPT = `Given the following conversation and a follow up question, return the co
Chat History:
{chat_history}
Follow Up Input: {question}
Your answer should follow the following format:
\`\`\`
Use the following pieces of context to answer the users question.
If you don't know the answer, just say that you don't know, don't try to make up an answer.
----------------
<Relevant chat history excerpt as context here>
Standalone question: <Rephrased question here>
\`\`\`
Your answer:`;
```

Run: ConversationalRetrievalQ...

Playground

Trace

ConversationalRetrievalQACh... • Success

Retriever VectorStoreRetriever

Chain StuffDocumentsChain

Chain LLMChain

LLM ChatOpenAI

ConversationalRetrievalQAChain

Run Feedback Metadata

INPUT

Copy

```yaml
question: I have a friend called Bob. He's 28 years old. He'd like to know what the powerhouse of the cell is?
chat_history:
  - id:
      - langchain
      - schema
      - HumanMessage
    lc: 1
    type: constructor
    kwargs:
      content: I have a friend called Bob.
```

YAML

Run: ConversationalRetrievalQ...

Playground

Trace

ConversationalRetrievalQACh... • Success

- Chain LLMChain
  - LLM ChatOpenAI
- Retriever VectorStoreRetriever
- Chain StuffDocumentsChain
  - Chain LLMChain
    - LLM ChatOpenAI

ConversationalRetrievalQAChain

Run | Feedback | Metadata

INPUT

Copy

```yaml
question: How old is Bob?
chat_history:
  - id:
      - langchain
      - schema
      - HumanMessage
    lc: 1
    type: constructor
    kwargs:
      content: I have a friend called Bob. He's 28 years old. He'd like to know what the powerhouse of the cell is?
```

YAML

四套应用疏通途：SQL QA

利用 LLM 的 SQL 生成能力，可以构建 SQL Chain 来进行面向数据库的问答

```
import { DataSource } from "typeorm";
import { OpenAI } from "langchain/llms/openai";
import { SqlDatabase } from "langchain/sql_db";
import { SqlDatabaseChain } from "langchain/chains/sql_db";
import { PromptTemplate } from "langchain/prompts";

const template = `Given an input question, first create a syntactically correct {dialect} query to run, then look at the
Use the following format:

Question: "Question here"
SQLQuery: "SQL Query to run"
SQLResult: "Result of the SQLQuery"
Answer: "Final answer here"

Only use the following tables:

{table_info}

If someone asks for the table foobar, they really mean the employee table.

Question: {input}`;
```

Run: SqlDatabaseChain

Playground

Trace

- SqlDatabaseChain • Success
  - Chain LLMChain
    - LLM OpenAI
  - Chain LLMChain
    - LLM OpenAI

SqlDatabaseChain

Run | Feedback | Metadata

INPUT Copy

```
1  query: How many employees are there in the
   foobar table?
```

YAML

OUTPUT Copy

```
1  sql: " SELECT COUNT(*) FROM Employee;"
2  result: " There are 8 employees in the
   foobar table."
```

YAML

用 Runnable Sequence 实现 SQL QA

```
import { DataSource } from "typeorm";
import { SqlDatabase } from "langchain/sql_db";
import { RunnableSequence } from "langchain/schema/runnable";
import { PromptTemplate } from "langchain/prompts";
import { StringOutputParser } from "langchain/schema/output_parser";
import { ChatOpenAI } from "langchain/chat_models/openai";

const datasource = new DataSource({ type: "sqlite", database: "Chinook.db" });
const db = await SqlDatabase.fromDataSourceParams({ appDataSource: datasource });

const prompt =
  PromptTemplate.fromTemplate(`Based on the table schema below, write a SQL query that would answer the user's question:
{schema}

Question: {question}
SQL Query:`);

const model = new ChatOpenAI();

const sqlQueryGeneratorChain = RunnableSequence.from([
  {
    schema: async () => db.getTableInfo(),
    question: (input: { question: string }) => input.question,
```

Run: RunnableSequence

Playground

Trace

- RunnableSequence • Success
  - Chain RunnableMap
    - Chain RunnableLambda
    - Chain RunnableSequence
      - Chain RunnableMap
        - Chain RunnableLambda
        - Chain RunnableLambda
      - Prompt PromptTemplate
      - LLM ChatOpenAI
      - Parser StrOutputParser
  - Chain RunnableMap
    - Chain RunnableLambda

RunnableSequence

Run | Feedback | Metadata

INPUT Copy

```
1 question: How many employees are there?
```

YAML

OUTPUT Copy

```
1 id:
2   - langchain
3   - schema
4   - AIMessage
5 lc: 1
6 type: constructor
```

四套应用疏通途：Web API

API Chain 允许使用 LLM 与 API 交互以检索相关信息，通过提供与提供的 API 文档相关的问题来构建链

```
temperature_unit    String  No  celsius If fahrenheit is set, all temperature values are converted to Fahrenheit.
windspeed_unit  String  No  kmh Other wind speed speed units: ms, mph and kn
precipitation_unit  String  No  mm  Other precipitation amount units: inch
timeformat  String  No  iso8601 If format unixtime is selected, all time values are returned in UNIX epoch time in secor
timezone    String  No  GMT If timezone is set, all timestamps are returned as local-time and data is returned starting
past_days   Integer (0-2)   No  0   If past_days is set, yesterday or the day before yesterday data are also returned.
start_date
end_date    String (yyyy-mm-dd) No      The time interval to get weather data. A day must be specified as an ISO8601 dat
models  String array    No  auto    Manually select one or more weather models. Per default, the best suitable weather r

Variable    Valid time  Unit    Description
temperature_2m  Instant °C (°F) Air temperature at 2 meters above ground
snowfall    Preceding hour sum  cm (inch)   Snowfall amount of the preceding hour in centimeters. For the water equivale
rain    Preceding hour sum  mm (inch)   Rain from large scale weather systems of the preceding hour in millimeter
showers Preceding hour sum  mm (inch)   Showers from convective precipitation in millimeters from the preceding hour
weathercode Instant WMO code    Weather condition as a numeric code. Follow WMO weather interpretation codes. See table
snow_depth  Instant meters  Snow depth on the ground
freezinglevel_height    Instant meters  Altitude above sea level of the 0°C level
visibility  Instant meters  Viewing distance in meters. Influenced by low clouds, humidity and aerosols. Maximum visibil

const model = new OpenAI({ modelName: "text-davinci-003" });
const chain = APIChain.fromLLMAndAPIDocs(model, OPEN_METEO_DOCS, { headers: {} });
await chain.call({ question: "What is the weather like right now in Munich, Germany in degrees Farenheit?" });
```

Run: APIChain

Playground

Trace

- APIChain • Success
  - Chain LLMChain
    - LLM OpenAI
  - Chain LLMChain
    - LLM OpenAI

APIChain

Run | Feedback | Metadata

INPUT Copy

```
1 question: What is the weather like right now in Munich, Germany in degrees Farenheit?
```

YAML

OUTPUT Copy

The current weather in Munich, Germany is 72.5 degrees Farenheit, with a windspeed of 1.3 km/h, a winddirection of 236 degrees, a weathercode of 1, and it is currently daytime.

种类繁多的“预制菜”

相比 Runnable Sequence 的自定义能力强，Chain 的最大特点就是预制化程度高

> 下面以 JS/TS SDK 展示一些有特色的 Chain

- 基于 OpenAI Functions 能力底座的 Chain：
  - `createExtractionChainFromZod`：从输入文本和所需信息的模式中提取对象列表
  - `createTaggingChain`：根据模式中定义的属性来标记输入文本
  - `createOpenAPIChain`：基于 Open API 规范自动选择和调用 API
- `OpenAIModerationChain`：审核链基于 OpenAI Moderation 对于检测可能仇恨、暴力等的文本很有用
- `ConstitutionalChain`：自我批判链是一条确保语言模型的输出遵守一组预定义准则的链

此外，也提供用于条件路由的 Chain：

- `MultiPromptChain`：使用多提示链创建一个问答链，选择与给定问题最相关的提示并进行回答
- `MultiRetrievalQAChain`：该链选择与给定问题最相关的 Retrieval QA Chain，然后使用它回答问题

参考资料

本教程在制作过程中参考和引用了以下资料（排名不分先后）的内容，特此鸣谢！

视频资料

- Short Courses | Learn Generative AI from DeepLearning.AI

图文资料

- Core Concepts | 🦜🔗 LangChain
  - JS/TS Docs, Python Docs, LangSmith Docs
- 入门：Prompts（提示词）| 通往 AGI 之路

代码资料

- openai/openai-cookbook: Examples and guides for using the OpenAI API
- datawhalechina/prompt-engineering-for-developers: 吴恩达大模型系列课程中文版
- slidevjs/slidev: Presentation Slides for Developers

感谢聆听 ❤️

webup | serviceup