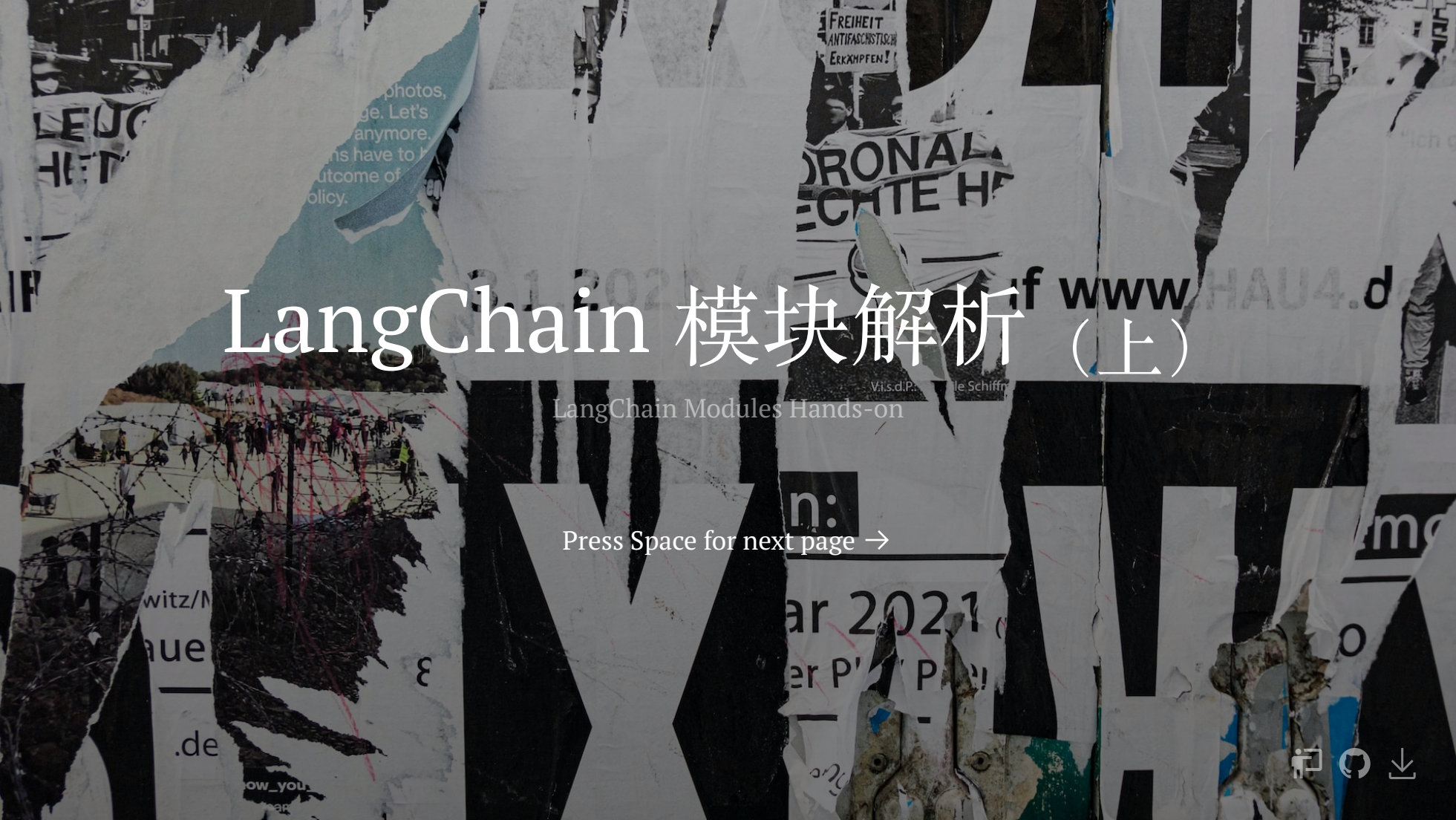


LangChain 模块解析（上）

LangChain Modules Hands-on

Press Space for next page →

《AGI 应用实践》系列教程

AGI OVERVIEW

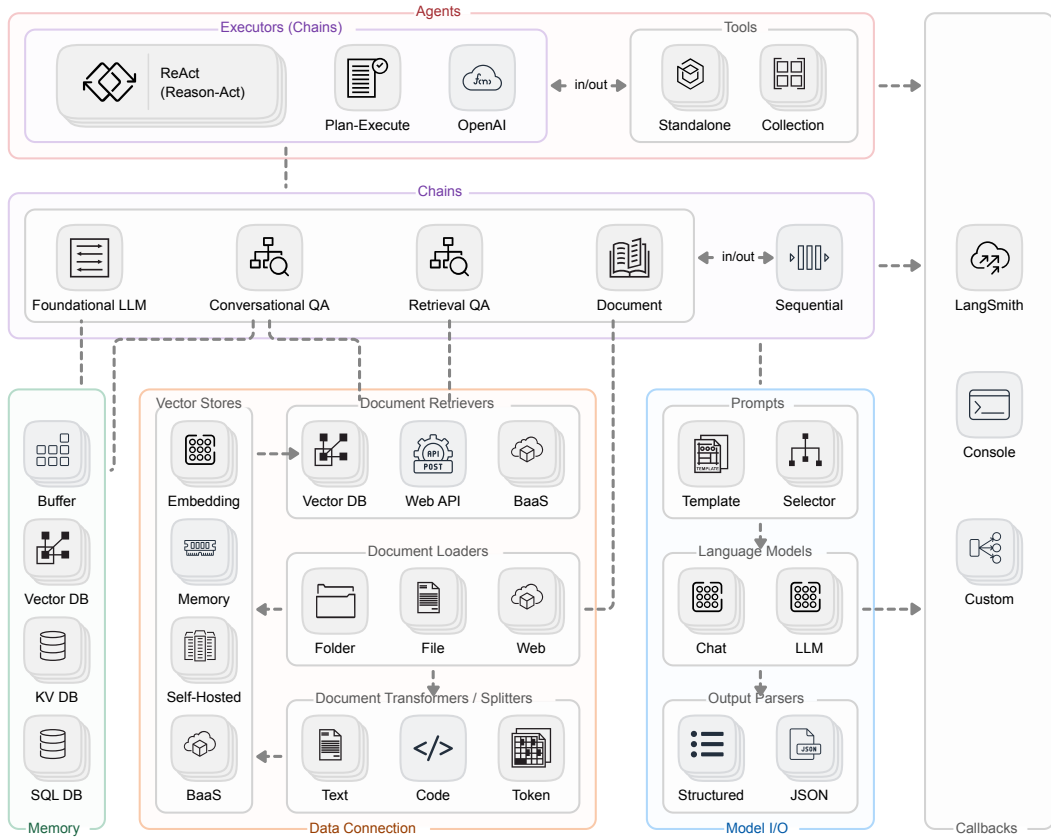
- 101：AGI 提示工程指北
- 102：OpenAI API 浅出

LANGCHAIN IN ACTION

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

教程大纲

- LangChain 核心模块概览
- Model I/O · 选择你的模型
- Data Connection · 准备你的数据
- Memory · 记录你的对话



LangChain (JS) Modules Overview v0.2.0 for npm:langchain@0.0.114 (Last Updated: July 22, 2023) by @zhanghaili0610

LangChain 核心模块概览

基于 LangChain Modules Overview 模块架构图一起来认识一下六大模块，参见 官方文档

201 教程的关注模块

- **Model I/O**：管理大语言模型（Models），及其输入（Prompts）和格式化输出（Output Parsers）
  - 🔌 数据流：（Parser 生成的输出格式提示 ➡️）Prompt ➡️ Model ➡️ Output Parser
- **Data Connection**：管理主要用于建设私域知识（库）的向量数据存储（Vector Stores）、内容数据获取（Document Loaders）和转化（Transformers），以及向量数据查询（Retrievers）
  - 🔌 数据流：Loaders ➡️ （Transformers ➡️）Embeddings Model ➡️ Vector Stores ➡️ Retrievers
- **Memory**：用于存储和获取 对话历史记录 的功能模块

202 教程的关注模块

- **Chains**：用于串联 Memory ↔ Model I/O ↔ Data Connection，以实现 串行化 的连续对话、推测流程
- **Agents**：基于 Chains 进一步串联 工具（Tools），从而将大语言模型的能力和本地、云服务能力结合
- **Callbacks**：提供了一个回调系统，可连接到 LLM 申请的各个阶段，便于进行日志记录、追踪等数据导流

演示环境准备

基于 Val Town 平台，调用 npm LangChain 来实现 LLM 应用流程

@webup.**getModelBuilder** v6

```
async function getModelBuilder(spec: {
  type?: "llm" | "chat" | "embedding";
  provider?: "openai" | "huggingface";
} = { type: "llm", provider: "openai" }, options?: any) {
  const { extend, cond, matches, invoke } = await import("npm:lodash-es");
  const logger = {
    handleLLMStart: (llm, prompts: string[]) =>
      console.log("🔘 LLM Started:", llm),
    handleLLMEnd: (llm) => console.log("🔲 LLM Ended: ", llm),
  };
  const callbacks = options?.verbose !== false ? [logger] : [];
  const args = extend({ callbacks }, options);
  if (spec?.provider === "openai")
```

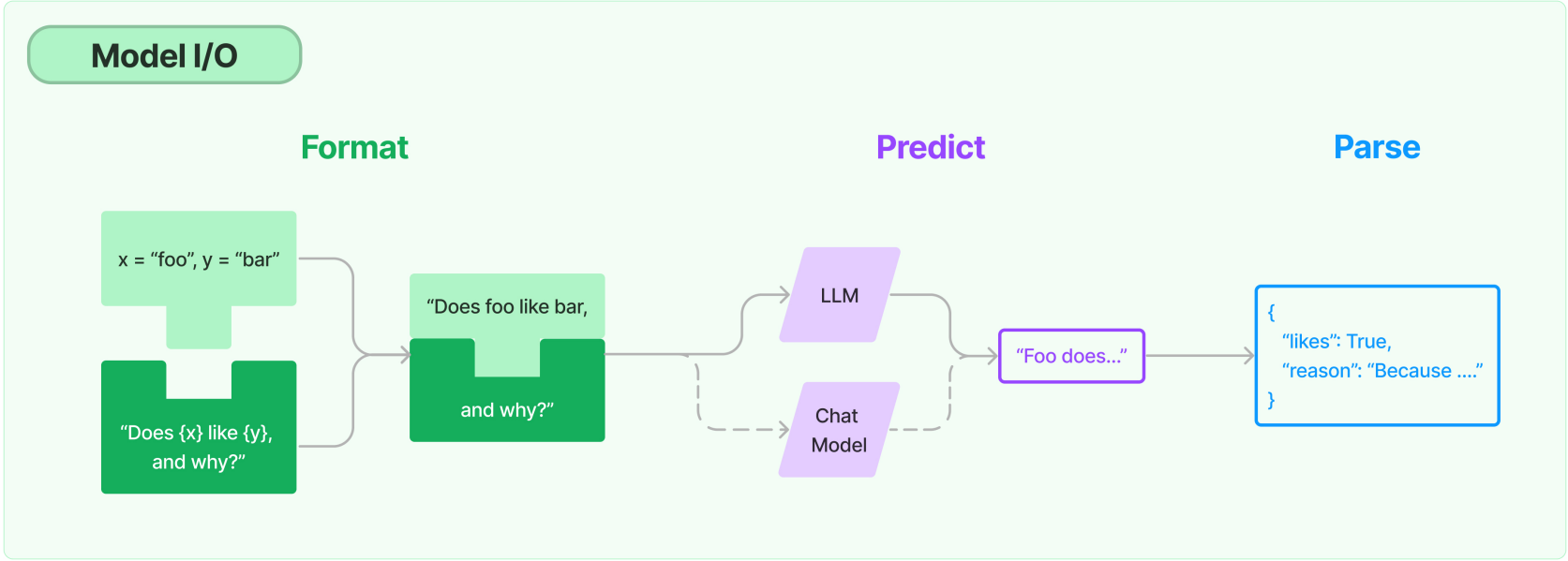
💡 官方也提供 Python 类库，以及庞大的社区 Integrations（实名羡慕啊 🍋）

Model I/O · 选择你的模型

同时也准备好你的输入、设定好输出的格式（结构）

Model I/O 三元组

- Prompts：模板化、动态选择和管理模型输入
- Language Models：通过通用接口调用语言模型
- Output Parsers：从模型输出中提取信息



Models：一切的缘起

在 LangChain 中，把模型分成三类：LLM 大语言模型，Chat 对话模型，Embeddings 嵌入模型

- LLM：将文本字符串作为输入，并返回续写文本的应用模型

@webup.**modelSampleLLMCall** v0

```
const modelSampleLLMCall = (async () => {
  const builder = await @webup.getModelBuilder();
  const model = await builder();
  return await model.call("Tell me a famous saying");
```

- Chat：基于 LLM 支持并将聊天消息序列作为输入，并返回聊天消息的应用模型

@webup.**modelSampleChatCall** v1

```
const modelSampleChatCall = (async () => {
  const builder = await @webup.getModelBuilder({
    type: "chat",
    provider: "openai",
  });
  const model = await builder();
  const { SystemMessage, HumanMessage } = await import("npm:langchain/schema");
```

Models 提高多个通用接口

通用接口也支持批处理调用，并返回更丰富的响应内容

@webup.**modelSampleLLMGenerate** v0

```
const modelSampleLLMGenerate = (async () => {
  const builder = await @webup.getModelBuilder();
  const model = await builder();
  return await model.generate(["Tell me a joke", "Tell me a poem"]);
})();
```

@webup.**modelSampleChatGenerate** v3

```
const modelSampleChatGenerate = (async () => {
  const builder = await @webup.getModelBuilder({
    type: "chat",
    provider: "openai",
  });
  const model = await builder();
  const { SystemMessage, HumanMessage } = await import("npm:langchain/schema");
  return await model.generate([
```

Prompts 的模板能力

Prompt Template 是一个带标记的文本字符串，可以接收来自最终用户的一组参数并生成提示

@webup.**promptSampleTemplates** v6

```
const promptSampleTemplates = (async () => {
  const { PromptTemplate } = await import("npm:langchain/prompts");
  const multipleInputPrompt = new PromptTemplate({
    inputVariables: ["adjective", "content"],
    template: "Tell me a {adjective} joke about {content}.",
  });
  const formattedMultipleInputPrompt = await multipleInputPrompt.format({
```

模板也提供 Partial 的形式，用于分阶段的注入参数（变量内容）

@webup.**promptSampleTemplatesPartial** v2

```
const promptSampleTemplatesPartial = (async () => {
  const { PromptTemplate } = await import("npm:langchain/prompts");
  const date = () => new Date().toISOString();
  const prompt = new PromptTemplate({
    template: "Tell me a {adjective} joke about the day {date}",
    inputVariables: ["adjective", "date"],
```

Prompts 模板的组合

可以通过 Pipeline 管道来完成模板的组合

Pipeline 是一组提示模板，其中的每个模板将格式化，并按最终模板格式进行组合

@webup.**promptSampleTemplatesPipeline** v0

```
const promptSampleTemplatesPipeline = (async () => {
  const { PromptTemplate, PipelinePromptTemplate } = await import(
    "npm:langchain/prompts"
  );
  const fullPrompt = PromptTemplate.fromTemplate(`{introduction}

{example}

{start}`);
  const introductionPrompt = PromptTemplate.fromTemplate(
    `You are impersonating {person}.`,
  );
  const examplePrompt = PromptTemplate.fromTemplate(
    `Here's an example of an interaction:
Q: {example_q}
```

Prompts 的示例选择器

当您有大量示例时，可能需要选择哪些要包含在提示中，示例选择器是负责执行此操作的工具

如下示例选择器 根据长度选择 要使用的示例，当我们担心构建的提示会超过上下文窗口的长度时，这非常有用

@webup.**promptSampleSelectorLength** v1

```
const promptSampleSelectorLength = (async () => {
  const { LengthBasedExampleSelector, PromptTemplate, FewShotPromptTemplate } =
    await import("npm:langchain/prompts");
  // Create a prompt template that will be used to format the examples.
  const examplePrompt = new PromptTemplate({
    inputVariables: ["input", "output"],
    template: "Input: {input}\nOutput: {output}",
  });
  // Create a LengthBasedExampleSelector that will be used to select the examples.
  const exampleSelector = await LengthBasedExampleSelector.fromExamples([
    { input: "happy", output: "sad" },
    { input: "tall", output: "short" },
    { input: "energetic", output: "lethargic" },
    { input: "sunny", output: "gloomy" },
    { input: "windy", output: "calm" },
```

Output Parsers 结构化输出：列表

如下示例中，我们将模型的输出解析为 具有特定长度和分隔符的列表

@webup.**parserSampleListCustom** v1

```
const parserSampleListCustom = (async () => {
  const { PromptTemplate } = await import("npm:langchain/prompts");
  const { CustomListOutputParser } = await import(
    "npm:langchain/output_parsers"
  );
  // With a `CustomListOutputParser`, we can parse a list with a specific length and separator.
  const parser = new CustomListOutputParser({ length: 3, separator: "\n" });
  const formatInstructions = parser.getFormatInstructions();
  const prompt = new PromptTemplate({
    template: "Provide a list of {subject}.\n{format_instructions}",
    inputVariables: ["subject"],
    partialVariables: { format_instructions: formatInstructions },
  });
  const builder = await @webup.getModelBuilder();
  const model = await builder();
```

Output Parsers 结构化输出：JSON（粗粒度）

如下示例中，我们将模型的输出解析为一个 包含多个字段的 JSON 对象

@webup.**parserSampleJSON** v0

```
const parserSampleJSON = (async () => {
  const { PromptTemplate } = await import("npm:langchain/prompts");
  const { StructuredOutputParser } = await import(
    "npm:langchain/output_parsers"
  );
  // With a `StructuredOutputParser` we can define a schema for the output.
  const parser = StructuredOutputParser.fromNamesAndDescriptions({
    answer: "answer to the user's question",
    source: "source used to answer the user's question, should be a website.",
  });
  const formatInstructions = parser.getFormatInstructions();
  const prompt = new PromptTemplate({
    template:
      "Answer the users question as best as possible.\n{format_instructions}\n{question}",
    inputVariables: ["question"],
```

Output Parsers 结构化输出：JSON（细粒度）

如下示例中，我们将模型的输出解析为一个结构被严格要求的 JSON 对象（npm Zod 格式）

@webup.**parserSampleJSONZod** v1

```
const parserSampleJSONZod = (async () => {
  const { z } = await import("npm:zod");
  const { PromptTemplate } = await import("npm:langchain/prompts");
  const { StructuredOutputParser } = await import(
    "npm:langchain/output_parsers"
  );
  // We can use zod to define a schema for the output using the `fromZodSchema` method of `StructuredOutput
  const parser = StructuredOutputParser.fromZodSchema(z.object({
    answer: z.string().describe("answer to the user's question"),
    sources: z.array(z.string()).describe(
      "sources used to answer the question, should be websites.",
    ),
  }));
  const formatInstructions = parser.getFormatInstructions();
  const prompt = new PromptTemplate({
```

Expression Language 0.0.121 : LLM Chain

现在可以用管道直接把 Prompt ➡️ Model ➡️ Output Parser 串联起来

@webup.**pipeSampleLLM** v1

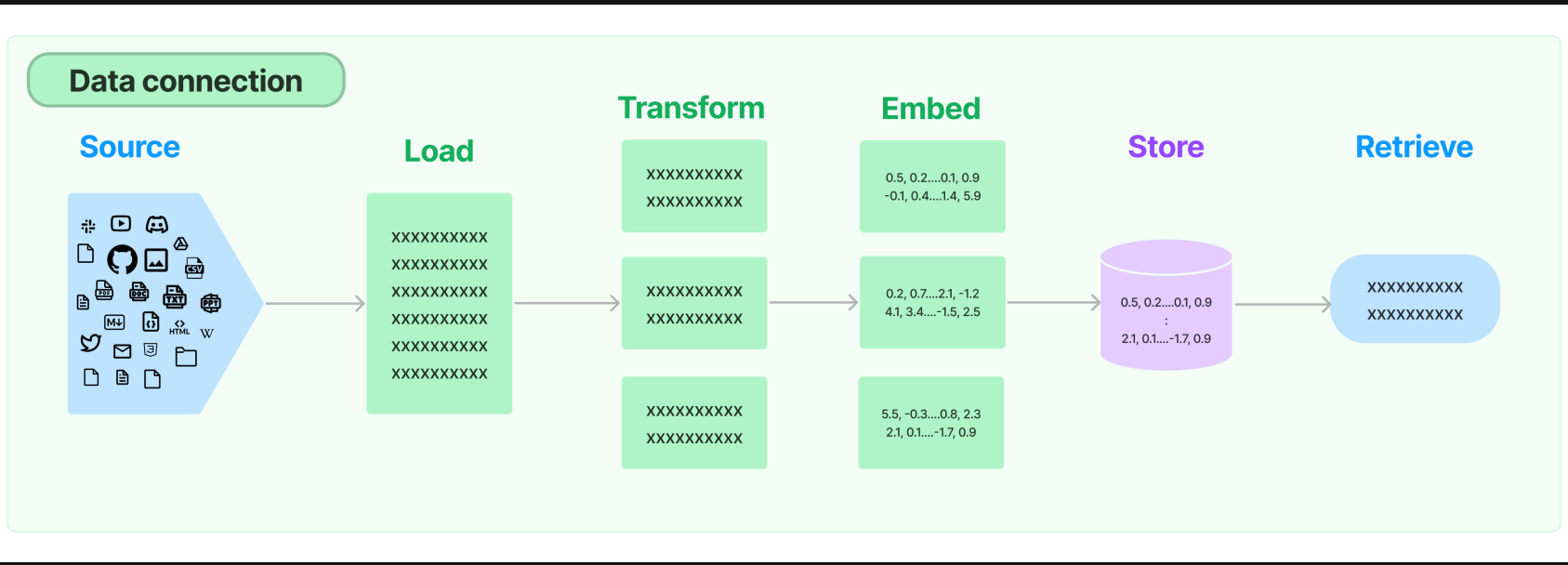
```
const pipeSampleLLM = (async () => {
  const { PromptTemplate } = await import("npm:langchain/prompts");
  const { RunnableSequence } = await import("npm:langchain/schema/runnable");
  const { StringOutputParser } = await import(
    "npm:langchain/schema/output_parser"
  );
  const builder = await @webup.getModelBuilder();
  const model = await builder();
  const promptTemplate = PromptTemplate.fromTemplate(
    "Tell me a joke about {topic}",
  );
  const outputParser = new StringOutputParser();
  const chain = RunnableSequence.from([promptTemplate, model, outputParser]);
  const result = await chain.invoke({ topic: "bears" });
  return result;
```

Data Connection · 准备你的数据

获取数据，整理数据，存储数据，查询数据

Data Connection 的核心组件

- Document Loaders & Transformers：从许多不同的源加载文档；然后拆分文档、删除冗余文档等
- Embedding Models：获取非结构化文本并将其转换为向量数据
- Vector Stores：存储和搜索向量数据
- Retrievers：从 Vector Stores 和其它数据源查询您的数据



Document Loaders 的丰富数据源

Document Loaders 从文档源加载数据，并将数据转化成“文档”格式（一段文本和关联的元数据）

数据源可以是 本地文件，也可以是 在线文件；在如下示例中，我们分别从 GitHub 和网页加载我们教程的内容

@webup.**loaderSampleWeb** v0

```
const loaderSampleWeb = (async () => {
  let builder, loader, docs;
  // Load markdown files from GitHub
  builder = await @webup.getWebLoaderBuilder(
    "https://github.com/webup/agi-talks/tree/master/pages/common/",
    "github",
    { branch: "master", recursive: false, unknown: "warn" },
  );
  loader = await builder();
  docs = await loader.load();
  console.log(docs[0].pageContent);
  // Load the same file as raw webpage
  builder = await @webup.getWebLoaderBuilder(
    "https://raw.githubusercontent.com/webup/agi-talks/master/pages/common/end.md",
  );
```

Document Transforms 的文本切分

受限于大语言模型的文本处理量，对于特别长的文本有必要将其分割成块，并尽量将语义相关的片段保留在一起

文本切分的主要方式大体包括“按字符”、“按 Token”、“按代码”；在如下示例中，我们主要演示前两种切分方式

@webup.**splitterSampleCharToken** v2

```
const splitterSampleCharToken = (async () => {
  const { RecursiveCharacterTextSplitter, TokenTextSplitter } = await import(
    "npm:langchain/text_splitter"
  );
  const text = `Hi.\n\nI'm Harrison.\n\nHow? Are? You?\nOkay then f f f f.
    This is a weird text to write, but gotta test the splittingggg some how.\n\n
    Bye!\n\n-H.`;
  const options = {
    chunkSize: 30,
    chunkOverlap: 5,
  };
  const charSplitter = new RecursiveCharacterTextSplitter(options);
  const chars = await charSplitter.createDocuments([text]);
  const tokenSplitter = new TokenTextSplitter(options);
  const tokens = await tokenSplitter.createDocuments([text]);
```

Vector Stores 向量存储

存储和搜索非结构化数据的最常见方法之一是嵌入它并存储生成的嵌入向量

向量存储 负责存储嵌入式数据并为您执行向搜索 —— 查询“最相似”的嵌入向量；向量由 Embeddings 模型生成

@webup.**getVectorStoreBuilder** v0

```
async function getVectorStoreBuilder(docs: object[], spec: {
  type: "memory" | "baas";
  provider?: "pinecone";
} = { type: "memory" }, embed: "openai" | "huggingface" = "openai") {
  const { cond, matches } = await import("npm:lodash-es");
  const builder = await @webup.getModelBuilder({
    type: "embedding",
    provider: embed,
  });
  const model = await builder();
  const setup = cond([
    [
      matches({ type: "memory" }),
      async () => {
        const { MemoryVectorStore } = await import(
```

3d 0 0 5

向量存储的相似度搜索

在如下示例中，我们分别展示 内存 和 Pinecone 云服务 两种向量存储的使用（写入和搜索）

@webup.**vectorStoreSampleMemory** v5

```
const vectorStoreSampleMemory = (async () => {
  const docs = await @webup.getSampleDocuments();
  const builder = await @webup.getVectorStoreBuilder(docs);
  const store = await builder();
  return store.similaritySearch("pinecone", 1);
})();
```

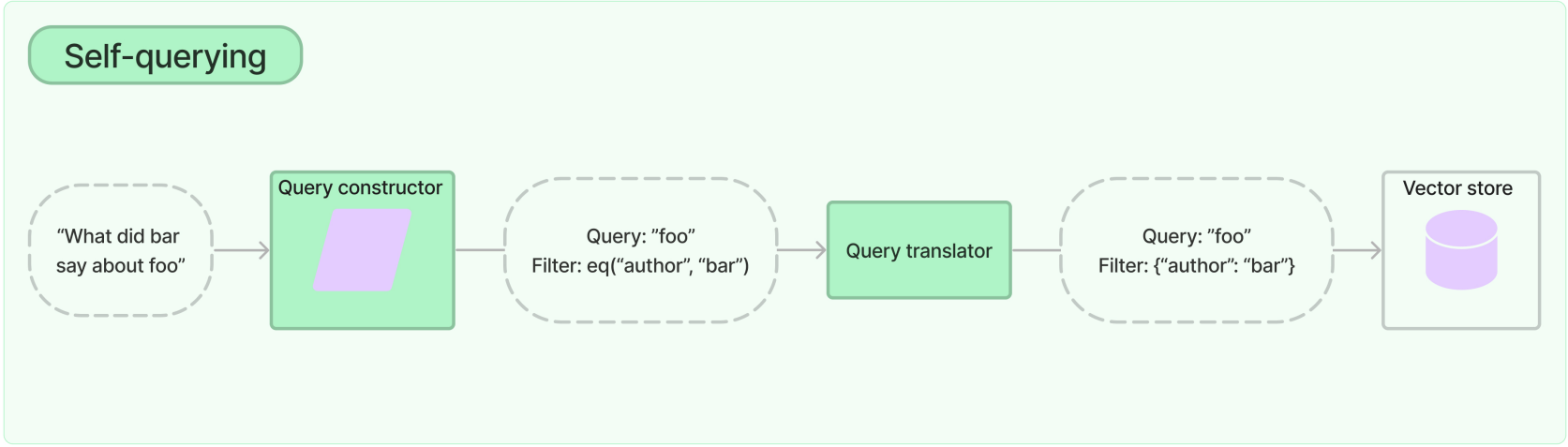
@webup.**vectorStoreSamplePinecone** v0

```
const vectorStoreSamplePinecone = (async () => {
  const builder = await @webup.getVectorStoreBuilder([], {
    type: "baas",
    provider: "pinecone",
  });
  const store = await builder();
  return store.similaritySearch("pinecone", 1);
```

Retrievers 从各个文档源检索

Retrievers 比向量存储更通用，它不需要能够存储文档，只需返回（或检索）文档

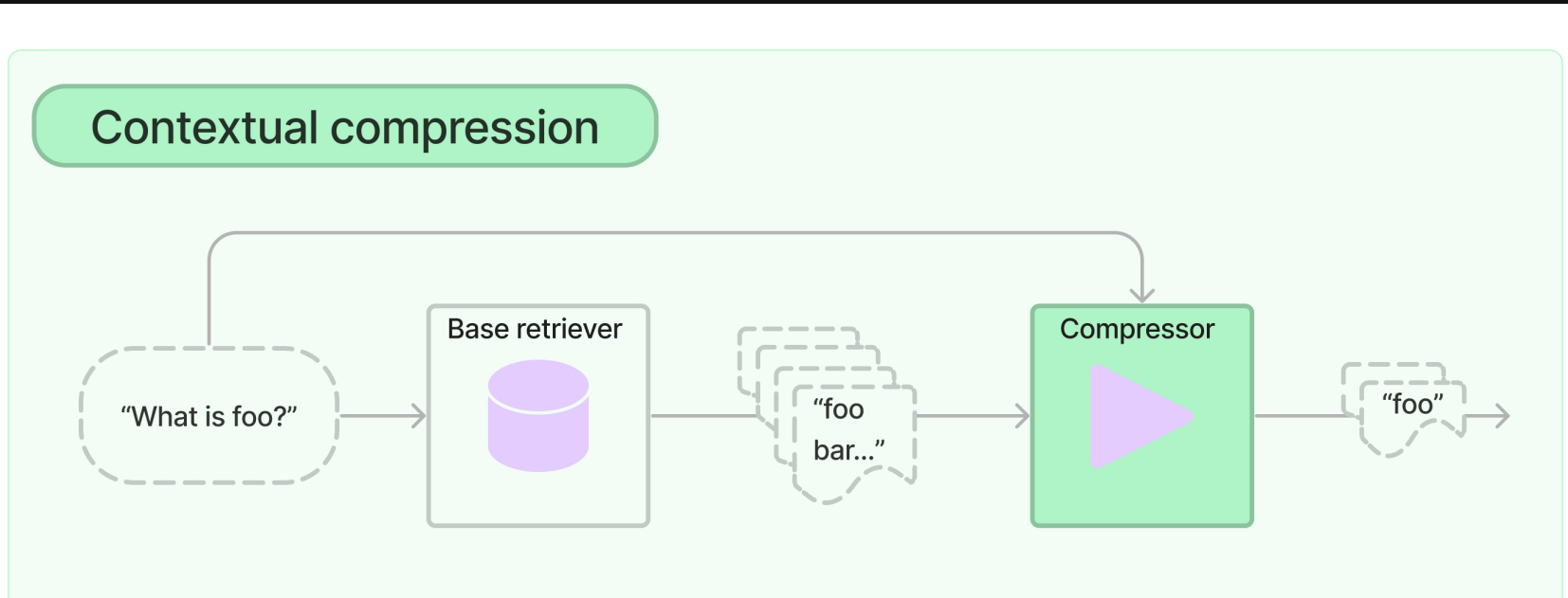
- 向量存储是最常用的文档源，但也可以从其它本地、云上的文档源进行数据的检索
- Self-Query Retriever：顾名思义，这是一种具有查询自身能力的检索器
  - 即给定任何自然语言查询，Retriever 通过构造 LLM Chain 编写结构化查询，并将其应用于底层向量存储



Retrievers 从各个文档源检索（续）

检索的一项挑战是，通常不知道将数据引入系统时，文档存储系统将面临哪些特定查询

- 与查询最相关的信息可能被隐藏在包含大量不相关文本中，传递完整文档成本更高且响应较差
- Contextual Compression Retriever：上下文压缩旨在解决此问题
  - 使用给定查询的上下文来压缩它们，以便只返回相关信息，而不是立即按原样返回检索到的文档



Expression Language 0.0.121 : LLM Chain + Retriever

可以用管道直接把 Prompt ➡️ Model ➡️ Output Parser ➡️ Retriever 串联起来

@webup.**pipeSampleLLMRetriever** v0

```
const pipeSampleLLMRetriever = (async () => {
  const { PromptTemplate } = await import("npm:langchain/prompts");
  const { RunnableSequence, RunnablePassthrough } = await import(
    "npm:langchain/schema/runnable"
  );
  const { StringOutputParser } = await import(
    "npm:langchain/schema/output_parser"
  );
  const { Document } = await import("npm:langchain/document");
  const modelBuilder = await @webup.getModelBuilder();
  const model = await modelBuilder();
  const docs = await @webup.getSampleDocuments();
  const vectorBuilder = await @webup.getVectorStoreBuilder(docs);
  const vector = await vectorBuilder();
  const retriever = vector.asRetriever();
```

Expression Language 0.0.121 : LLM Chain + Retriever

在上个示例中，我们将字符串输入直接传递到链路中；如希望链路接受多个输入，可以传递函数映射来解析输入

@webup.**pipeSampleLLMRetrieverInputs** v3

```
const pipeSampleLLMRetrieverInputs = (async () => {
  const { PromptTemplate } = await import("npm:langchain/prompts");
  const { RunnableSequence, RunnablePassthrough } = await import(
    "npm:langchain/schema/runnable"
  );
  const { StringOutputParser } = await import(
    "npm:langchain/schema/output_parser"
  );
  const { Document } = await import("npm:langchain/document");
  const modelBuilder = await @webup.getModelBuilder();
  const model = await modelBuilder();
  const docs = await @webup.getSampleDocuments();
  const vectorBuilder = await @webup.getVectorStoreBuilder(docs);
  const vector = await vectorBuilder();
  const retriever = vector.asRetriever();
```

Expression Language 0.0.121 : LLM Chain + Retriever

在上个示例中，我们使链路接受了多个输入；我们还可以通过格式化函数添加对话历史记录

@webup.**pipeSampleLLMRetrieverConversation** v1

```
const pipeSampleLLMRetrieverConversation = (async () => {
  const { PromptTemplate } = await import("npm:langchain/prompts");
  const { RunnableSequence, RunnablePassthrough } = await import(
    "npm:langchain/schema/runnable"
  );
  const { StringOutputParser } = await import(
    "npm:langchain/schema/output_parser"
  );
  const { Document } = await import("npm:langchain/document");
  const modelBuilder = await @webup.getModelBuilder();
  const model = await modelBuilder();
  const docs = await @webup.getSampleDocuments();
  const vectorBuilder = await @webup.getVectorStoreBuilder(docs);
  const vector = await vectorBuilder();
  const retriever = vector.asRetriever();
```

Memory · 记录你的对话

对话记录是保持链路状态的基石

Memory 对话状态存储

Memory 负责在用户与大语言模型的交互过程中保留状态

- Memory 的作用可以归结为从一系列聊天消息中摄取、捕获、转换和提取知识
- Memory 可以返回多条信息（例如最近的 N 条消息和摘要）或者所有以前的消息

@webup.**getMemoryBuilder** v3

```
async function getMemoryBuilder(spec: {
  type: "buffer" | "summary" | "vector";
  provider?: "openai";
} = { type: "buffer" }, options = {}) {
  const { cond, matches } = await import("npm:lodash-es");
  const setup = cond([
    [
      matches({ type: "buffer" }),
      async () => {
        const { BufferMemory } = await import("npm:langchain/memory");
        return new BufferMemory();
      },
    ],
    [
```

Memory 的多样化存取方式

Memory 不仅可以整存整取，还可以利用大语言模型加工并返回处理后的内容

- 通过 LLM 模型对记录内容进行总结

@webup.**memorySampleSummary** v3

```
const memorySampleSummary = (async () => {
  const builder = await @webup.getMemoryBuilder({
    type: "summary",
    provider: "openai",
  });
  const memory = await builder();
  await memory.saveContext({ input: "My favorite sport is soccer" }, {
```

- 通过 Vector Store 对记录内容进行相似度匹配后返回

@webup.**memorySampleVector** v0

```
const memorySampleVector = (async () => {
  const builder = await @webup.getMemoryBuilder({
    type: "vector",
    provider: "openai",
  });
  const memory = await builder();
```

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