

LangChain及RAG原理

[LangChain](#)

[LangChain框架的组成](#)

[LangChain 工具库所包含的相关包](#)

[安装](#)

[LCEL构建LLM应用](#)

[实现步骤：](#)

[1. 创建语言模型](#)

[2. 使用 PromptTemplates 和 OutputParsers](#)

[3. LangChain Expression Language \(LCEL\)](#)

[4. 使用 LangServe 部署应用程序](#)

[palyground](#)

[LangChain聊天机器人](#)

[具体实现流程：](#)

[1. 创建语言模型](#)

[2. 使用消息历史记录](#)

[3. 使用 PromptTemplates](#)

[4. 管理对话历史记录](#)

[流](#)

[检索增强生成\(RAG\)概述](#)

[什么是检索增强生成？](#)

[RAG实现细节](#)

[1. 定制系统提示](#)

[2. 提供知识来源](#)

[3. 合并内容提出问题](#)

[检索步骤：从知识库中获取正确的信息](#)

[Embedding](#)

[使用嵌入找到最好的知识片段](#)

[索引自定义的知识库](#)

[过程整体回顾](#)

LangChain

LangChain 是一个开发大语言模型应用的框架。它能够为应用程序带来：

- 内容组件：建立原始内容到语言模型建的关联（通过结构化的提示和少量的举例就可以得到满意的回复）
- 推理：通过语言模型进行推理（给出基于当前问题的解答和行动策略）

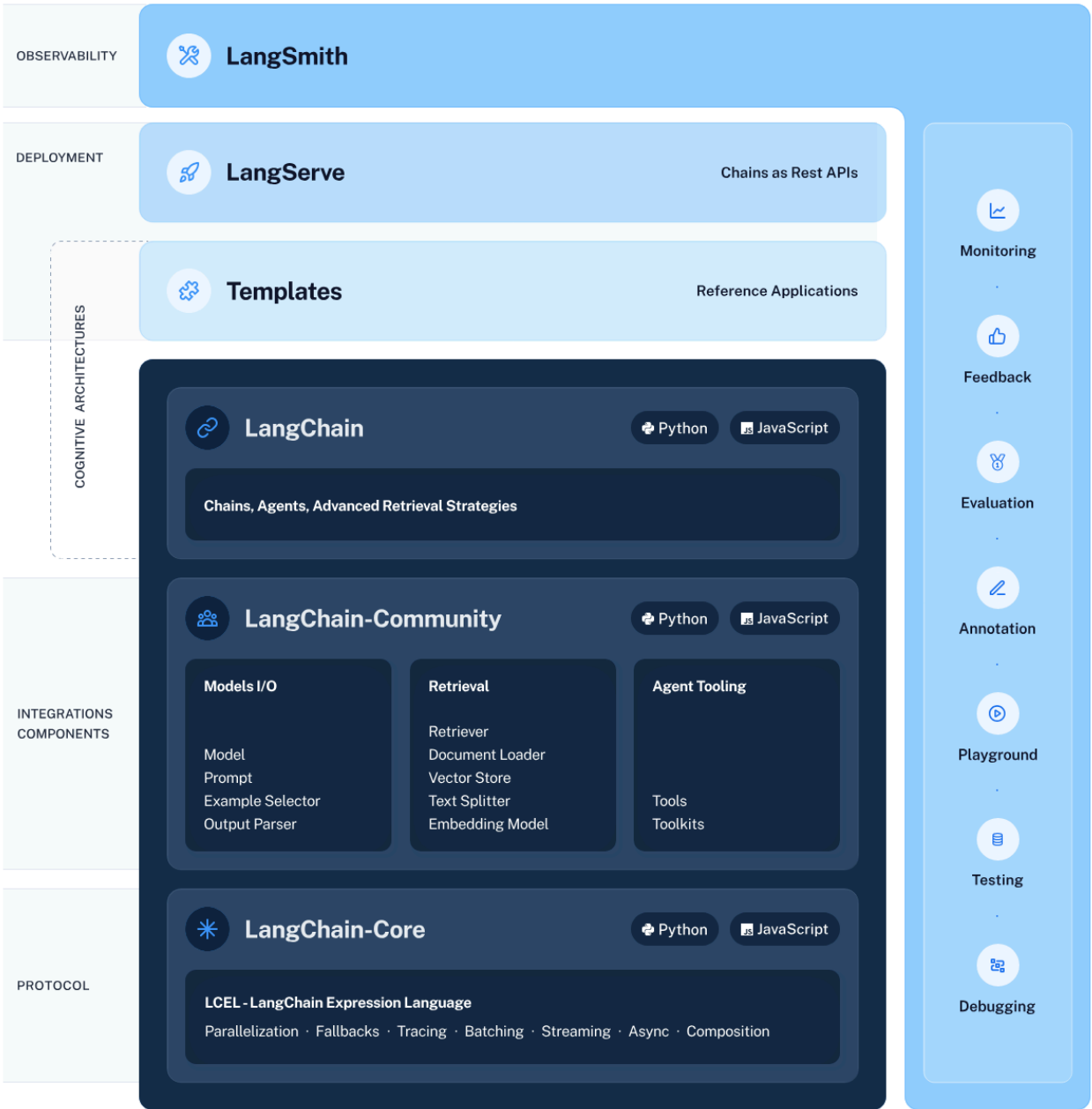
LangChain框架的组成

LangChain Libraries：基于Python和JavaScript的库。包含各种组件（component）和接口（interface）集成；由组件构建的链（chain）和代理（agent）的基本运行环境；链（chain）和代理（agent）的现有功能实现。

LangChain Templates：一组易于部署的参考架构，适用于各种任务。

LangServer：将LangChain链部署为REST API的库。

LangSmith：一个开发平台，允许我们调试、测试、评估和监控构建在任何LLM框架上的链，与LangChain无缝集成。



这些产品能够为我们简化应用程序的开发周期：

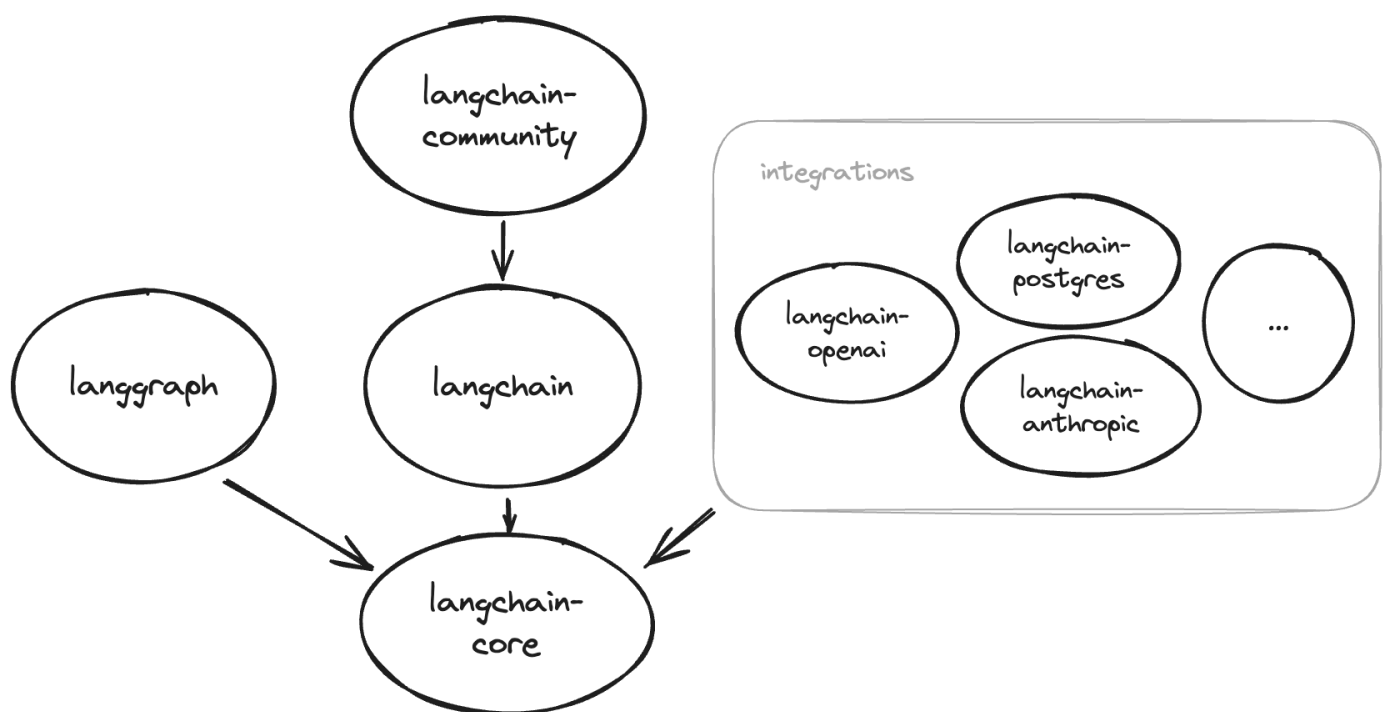
开发：适用LangChain/LangChain.js编写自定义的应用程序。使用模版作为参考，可以立即投入到应用领域。

生产：适用LangSmith检查、测试和监控自定义的链，一遍不断优化和放心进行部署。

部署：适用LangServe可以将任意的链转换为API。

LangChain 工具库所包含的相关包

LangChain 生态系统中的所有包都依赖于 `langchain-core` 它包含 基础功能及其它包使用的类和抽象。



安装指定的包时，包的依赖项会自动安装。所以当我们安装 `langchain` 包时，其底层所依赖的 `langchain-core` 包会自动安装。

`langchain-core` 基础的抽象和 LangChain 表达式语言。

`langchain` 链，代理以及构成应用程序认知架构的检索策略。

`langchain-community` 第三方集成

`langgraph` 通过将建模步骤描述为图中的边和节点，可以更方便的使用 LLM构建稳定、有状态、多参与者的应用程序。

`langserve` 将LangChain链部署为REST API（网络服务）。

`LangSmith` 一个开发人员平台，可以调试、测试、评估和监控LLM应用程序。

安装

初期，根据基本需求。我们可以先安装两个包 `langchain` 和 `langchain-openai`

```
pip install --upgrade langchain langchain-openai -i https://pypi.tuna.tsinghua.edu.cn/simple
```

LCEL构建LLM应用

我们通过LangChain构建一个基于OpenAI LLM的应用程序示例来学习LCEL。

实现步骤：

- `.env`

1. 创建语言模型

LangChain支持很多种不同的语言模型，这里我们先使用OpenAI。

```
import os
from dotenv import load_dotenv, find_dotenv
from langchain_openai import ChatOpenAI
```

```
# 加载.env文件条目并注册系统环境变量 "OPENAI_API_KEY"
load_dotenv(find_dotenv())

model = ChatOpenAI(model="gpt-4o-mini")
```

语言模型创建完成后，我们可以通过调用 `invoke()` 方法来和模型进行通讯。

- **直接调用**

```
model.invoke('翻译：Was this page helpful?')
```

- **通过HumanMessage调用**

```
from langchain_core.messages import HumanMessage

message = HumanMessage(content="翻译:Was this page helpful?")

model.invoke([message])
```

- **SystemMessage结合HumanMessage调用**

```
from langchain_core.messages import HumanMessage, SystemMessage

messages = [
    SystemMessage(content="将下面的内容从英文翻译成中文"), # 应用程序逻辑
    HumanMessage(content="Was this page helpful?"),
]

model.invoke(messages)
```

2. 使用 PromptTemplates 和 OutputParsers

- **PromptTemplates**

提示模板是LangChain中的一个概念，主要是为了把应用程序逻辑和用户输入转换为准备传递给语言模型的消息列表。它接收原始用户输入并返回准备传递到语言模型的数据（prompt）

```
from langchain_core.prompts import ChatPromptTemplate

# 创建提示模板对象
prompt_template = ChatPromptTemplate.from_messages(
    [("system", "将下面的内容从英文翻译成{language}"), ("user", "{text}")]
)
# 调用模板对象
prompt_template.invoke({"language": "意大利语", "text": "Was this page helpful?"})
```

- **OutputParsers**

模型调用后，返回的是一个 `AIMessage` 对象，里面包含了响应的字符串以及有关该响应的其它元数据。通常，我们可能只想处理响应中的字符串内容。所以我们可以通过使用简单的输出解析器来解析此响应。

```
from langchain_core.output_parsers import StrOutputParser

# 简单输出解析器
parser = StrOutputParser()
result = model.invoke(messages)
# 提取响应中的文本内容
parser.invoke(result)
```

3. LangChain Expression Language (LCEL)

LangChain表达式语言（LCEL）是一种声明式的链的组合方式。

LCEL 从出现的第一天起就被设计为支持生产环境。

从最简单的“提示 + LLM”链到最复杂的链，都无需更改代码。

```
chain = prompt_template | model | parser

chain.invoke({"language": "italian", "text": "hi"})
```

4. 使用 LangServe 部署应用程序

LangServe帮助开发者将LangChain链部署为**REST API**。

REST API 是一种让不同的软件或者设备能够方便、高效地在网络上进行交流和获取数据的一种方式。

REST API 主要是依靠一些常见的网络操作来完成工作的，例如 **GET**（获取信息）、**POST**（发送新的信息）、**PUT**（更新已有的信息）和 **DELETE**（删除信息）。

如果我们把网络世界里的各种信息和功能当作是一个大商场里的不同店铺和服务。

REST API 就像是连接我们和这些店铺、服务的**通道**，或者说是“**服务员**”。当我们打开天气应用想查看今天的天气，手机就会通过 **GET** 这个操作向服务器发送请求说：“嘿，我想要今天的天气信息。”然后服务器就会通过 **REST API** 把天气信息给您传回来。

```
#!/usr/bin/env python
from typing import List

from fastapi import FastAPI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI
from langserve import add_routes
from dotenv import load_dotenv, find_dotenv

load_dotenv(find_dotenv())

# 1. 创建 prompt template
system_template = "将下面的内容从英文翻译成{language}:"
prompt_template = ChatPromptTemplate.from_messages([
    ('system', system_template),
    ('user', '{text}')
])

# 2. 创建 model
model = ChatOpenAI()

# 3. 创建 parser
parser = StrOutputParser()

# 4. 创建 chain
chain = prompt_template | model | parser
```

```
# 4. App 定义
app = FastAPI(
    title="LangChain Server",
    version="1.0",
    description="A simple API server using LangChain's Runnable interfaces",
)

# 5. 添加 chain 路由

add_routes(
    app,
    chain,
    path="/chain",
)

if __name__ == "__main__":
    import uvicorn

    uvicorn.run(app, host="localhost", port=6006)
```

通过调用 `langserve.add_routes`，为 chain 提供服务器端调用的路由。

FastAPI 是一个现代、快速（高性能）的 Web 框架，用于使用 Python 构建 API。

如果要创建一个简单的 API 来获取用户信息，使用 FastAPI 可以像下面这样写：

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/users/{user_id}")
def get_user(user_id: int):
    return {"user_id": user_id}
```

在代码中，使用 `@app.get` 装饰器定义了一个 GET 请求的处理函数，当用户访问 `/users/{user_id}` 路径时，会返回对应的用户 ID 信息。


Uvicorn 是一个快速的 ASGI（Asynchronous Server Gateway Interface）服务器，用于构建异步 Web 服务。它基于 `asyncio` 库，支持高性能的异步请求处理，适用于各种类型的 Web 应用程序。

Uvicorn 可与各种 ASGI 应用程序框架（如 **FastAPI**、**Django** 等）配合使用，帮助开发者轻松构建异步 API 服务、处理大量并发请求等，以提高系统的性能和吞吐量。

palyground

每个 LangServe 服务都带有一个简单的内置 UI，用于配置和调用应用程序，并提供流式输出和对中间步骤的可见性。

我们可以通过 <http://localhost:6006/chain/playground/> 进行测试！输入与对应的语言和内容，就可以查看到服务反馈的结果。

 **LangServe Playground**

Try it

Inputs

Reset

LANGUAGE*
法语

TEXT*
你好

Output

Bonjour

Intermediate steps 3 >

Share

Start

如果遇到程序运行报错：

TypeError: ForwardRef._evaluate() missing 1 required keyword-only argument: 'recursive_guard'

可以通过升级pydantic包来解决问题

```
pip install --upgrade pydantic
```

LangChain聊天机器人

聊天历史 的概念指的是LangChain中的一个类，它可以用来包装一个任意的链（chain）。这个聊天历史将跟踪底层链的输入和输出，将它们作为消息附加到消息数据库中。然后，未来的交互将加载这些消息并将它们作为输入的一部分传递到链中。

具体实现流程：

- 创建语言模型
- 使用消息历史记录
- 使用 PromptTemplates
- 管理对话历史记录

1. 创建语言模型

LangChain及RAG原理

7

```
import os
from dotenv import load_dotenv, find_dotenv
from langchain_openai import ChatOpenAI

# 加载.env文件条目并注册系统环境变量 "OPENAI_API_KEY"
load_dotenv(find_dotenv())

model = ChatOpenAI(model="gpt-4o-mini")
```

直接调用模型，将消息列表传递给 `invoke()` 。

```
from langchain_core.messages import HumanMessage

model.invoke([HumanMessage(content="嗨！ 我是Bob")])
```

输出：

```
AIMessage(content='你好Bob！ 今天我能为做些什么？ ', response_metadata={'token_usage': {'completion_token': 10, 'prompt_token': 10, 'total_token': 20}, 'model': 'gpt-4o-mini', 'finish_reason': 'stop', 'logprobs': None})
```

由于模型本身没有任何状态概念。如果后面继续提问：

```
model.invoke([HumanMessage(content="我叫什么名字?")])
```

输出：

```
AIMessage(content="很抱歉，除非你提供给我，否则我无权访问个人信息。今天我能为做些什么？ ", response_metadata={'token_usage': {'completion_token': 10, 'prompt_token': 10, 'total_token': 20}, 'model': 'gpt-4o-mini', 'finish_reason': 'stop', 'logprobs': None})
```

我们可以看到，模型无法正确回答问题。 这导致了糟糕的聊天机器人体验！

2. 使用消息历史记录

使用消息历史记录类来包装我们的模型并使其有状态。 这样可以跟踪模型的输入和输出，并将它们通过某种方式进行存储。 然后，着后面的交互中加载这些消息，并将它们作为输入的一部分传递到链中。

因为我们将使用 `langchain-community` 中的模块来存储消息历史记录。

```
pip install langchain_community
```

之后，我们可以导入相关类并设置我们的链，该链包装模型并添加此消息历史记录。这里的一个关键部分就是 `get_session_history` 函数。它接收一个 `session_id` 并返回 `Message History` 对象。 `session_id` 用于区分独立的会话，并且在调用新链时应作为配置的一部分传入。

```
from langchain_core.chat_history import (
    BaseChatMessageHistory,
    InMemoryChatMessageHistory,
)
from langchain_core.runnables.history import RunnableWithMessageHistory

store = {}

def get_session_history(session_id: str) → BaseChatMessageHistory:
    if session_id not in store:
        store[session_id] = InMemoryChatMessageHistory()
    return store[session_id]

with_message_history = RunnableWithMessageHistory(model, get_session_history)
```

`BaseChatMessageHistory` 是一个抽象父类，用于描述存储聊天消息历史的的相关操作函数。

`InMemoryChatMessageHistory` 在内存中实现聊天消息历史记录类。

`RunnableWithMessageHistory` 为另一个可运行的对象管理聊天消息的历史记录类。`RunnableWithMessageHistory` 会包裹（warp）LangChain对象 并为其管理聊天消息历史，负责读取和更新该历史。默认情况下，每个chain 都有一个 session_id 以对应一系列聊天历史消息。

现在我们创建一个配置信息，每次都传递给 runnable。配置中包含的 session_id 信息不是聊天消息，但可以识别当前的对话。

```
config = {"configurable": {"session_id": "abc2"}}
response = with_message_history.invoke(
    [HumanMessage(content="你好！我是Bob")],
    config=config,
)

response.content
```

输出

嗨！Bob！今天我能为做些什么？

```
response = with_message_history.invoke(
    [HumanMessage(content="我叫什么名字?")],
    config=config,
)

response.content
```

输出

你的名字是Bob。我今天能帮你做些什么，Bob？

现在聊天机器人记住了关于连续对话之间的沟通。如果我们更改配置，使用不同的session_id，那么会发现它重新开始了对话。

```
config = {"configurable": {"session_id": "abc3"}}

response = with_message_history.invoke(
    [HumanMessage(content="我叫什么名字？ ")],
    config=config,
)

response.content
```

输出

对不起，我不能确定你的名字，因为我是一个人工智能助理，没有获得这些信息。

这就是聊天机器人与许多用户进行对话的关键所在！我们只是在模型调用的外部添加了一个简单的持久层就实现了这个功能。我们还可以通过添加提示模板来实现更复杂的交互和个性化定制。

3. 使用 PromptTemplates

提示模板有助于将原始用户信息转换为LLM可以使用的格式。在这种情况下，原始用户输入只是一条消息，我们将其传递给LLM。现在让我们把它变得更复杂一点。首先，添加一个带有一些自定义指令的系统消息（但仍将消息作为输入）。接下来，除了消息之外，我们将添加更多输入。

首先，让我们添加一条系统消息。为此，我们将创建一个 `ChatPromptTemplate`。我们将利用 `MessagesPlacehold` 传递所有消息。

```

from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "你是一个得力的助手。尽你所能使用回答所有问题。",
        ),
        MessagesPlaceholder(variable_name="messages"),
    ]
)

chain = prompt | model

```

`MessagesPlaceholder` 消息列表的提示模板类。可用于传入消息列表作为占位符。

传入带有消息键的dict，其中包含消息列表。

```

response = chain.invoke({"messages": [HumanMessage(content="你好！ 我是Bob")]}))

response.content

```

我们现在可以将 chain 包装在消息历史对象中

```

with_message_history = RunnableWithMessageHistory(chain, get_session_history)

config = {"configurable": {"session_id": "abc5"}}
response = with_message_history.invoke(
    [HumanMessage(content="你好！ 我是Jim")],
    config=config,
)

response.content

```

```

response = with_message_history.invoke(
    [HumanMessage(content="我叫什么名字? ")],
    config=config,
)

response.content

```

现在让我们将这个更复杂的链包装在一个消息历史类中。这次，因为输入中有多个键，我们需要指定正确的键来保存聊天记录。

```

prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "你是一个乐于助人的助手。尽你所能用{language}来回答所有问题。",
        ),
        MessagesPlaceholder(variable_name="messages"),
    ]
)

chain = prompt | model

```

```
with_message_history = RunnableWithMessageHistory(
    chain,
    get_session_history,
    input_messages_key="messages",
)
```

我们现在可以调用带有历史记录的 chain 并传入我们选择的语言。

```
config = {"configurable": {"session_id": "abc11"}}
response = with_message_history.invoke(
    {"messages": [HumanMessage(content="你好！ 我是Bob")], "language": "西班牙语"},
    config=config,
)

response.content
```

4. 管理对话历史记录

构建聊天机器人时要了解的一个重要概念是如何管理对话历史记录。如果不加以管理，消息列表将无限增长，并可能溢出 LLM 的上下文窗口。因此，添加一个步骤来限制您传入的消息的大小非常重要。

LangChain 有一些内置的辅助函数，用于管理消息列表。我们使用 `trim_messages` 来帮助程序减少向模型发送的消息数。修剪器允许我们指定要保留的token数量，以及其他参数，例如是否要始终保留系统消息以及是否允许部分消息：

```
from langchain_core.messages import SystemMessage, trim_messages

trimmer = trim_messages(
    max_tokens=65,
    strategy="last",
    token_counter=model,
    include_system=True,
    allow_partial=False,
    start_on="human",
)

messages = [
    SystemMessage(content="你是一个优秀的助手"),
    HumanMessage(content="你好！ 我是Bob"),
    AIMessage(content="你好!"),
    HumanMessage(content="我喜欢香草冰激凌"),
    AIMessage(content="赞！ "),
    HumanMessage(content="2 + 2等于多少"),
    AIMessage(content="4"),
    HumanMessage(content="谢谢"),
    AIMessage(content="不客气!"),
    HumanMessage(content="要开心呦?"),
    AIMessage(content="是的!"),
]

trimmer.invoke(messages)
```

在将输入传递给提示符之前调用修剪器

```
from operator import itemgetter

from langchain_core.runnables import RunnablePassthrough
```

```
chain = (
    RunnablePassthrough.assign(messages=itemgetter("messages") | trimmer). # Runnable系列类 生成对象都可以
    | prompt
    | model
)

response = chain.invoke(
    {
        "messages": messages + [HumanMessage(content="我叫什么名字?")],
        "language": "English",
    }
)
response.content
```

`RunnablePassthrough` 确保数据可以在调用时直接传递给后面的目标。 它描述的是一种程序或系统的运行机制，即能够直接传递未改变的输入或者带有新增键值的输入。

流

现在我们有一个功能聊天机器人。然而，聊天机器人应用程序的一个真正重要的用户体验考虑因素是流。LLM 有时可能需要一段时间才能响应，因此，为了改善用户体验，大多数应用程序所做的一件事就是在生成每个 token 时就通过流返回。这使用户能够查看生成内容的进度。

实际上，做到这一点非常容易！

所有链都公开了一个方法 `.stream` ，就算是使用消息历史记录链也不例外。我们可以简单地使用这种方法来返回流式响应。

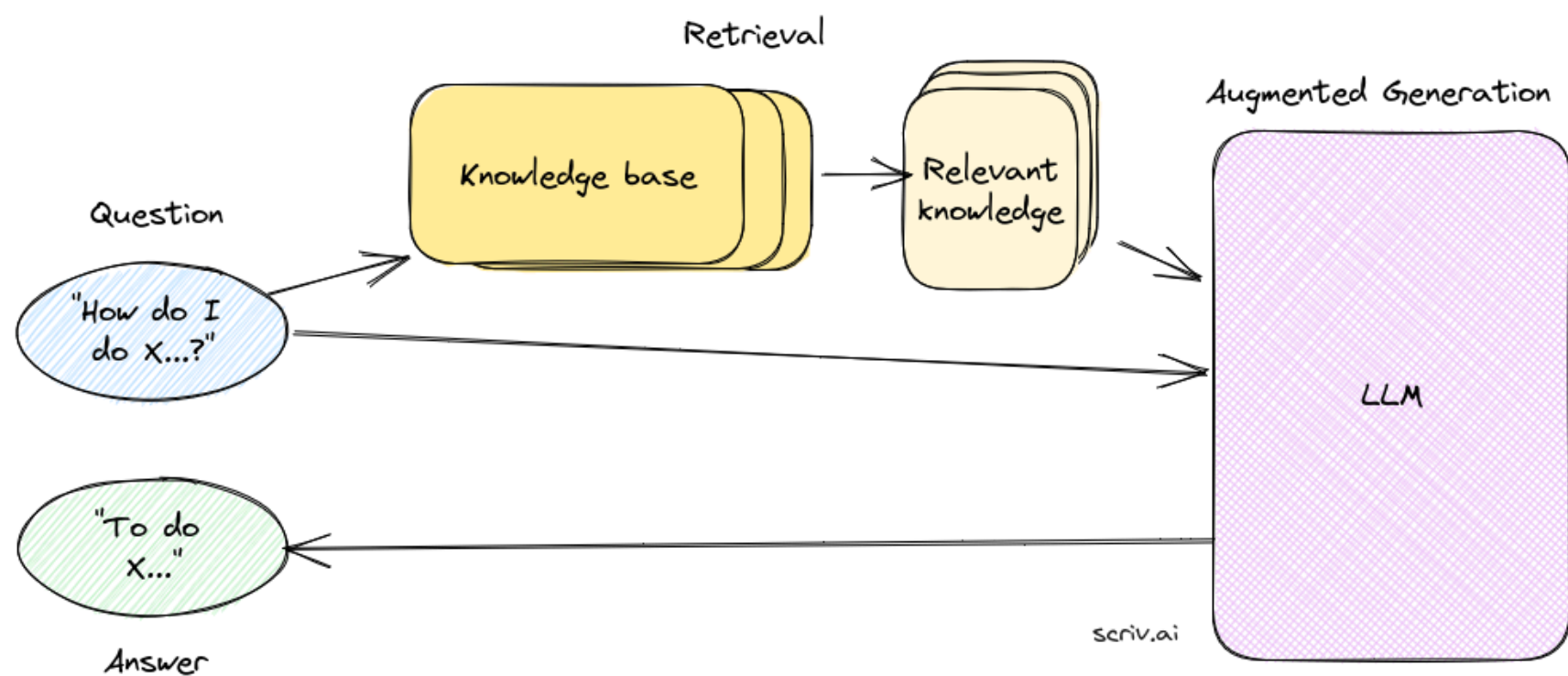
```
config = {"configurable": {"session_id": "abc15"}}
for r in with_message_history.stream(
    {
        "messages": [HumanMessage(content="你好!我是todd。给我讲个笑话")],
        "language": "中文",
    },
    config=config,
):
    print(r.content, end="|")
```

检索增强生成(RAG)概述

什么是检索增强生成？

检索增强生成（retrieval Augmented Generation）就是在应用大语言模型过程中，使用从其它途径 **检索而得到** 的额外信息作为模型的补充输入。**把这些内容输入到大型语言模型 (LLM)（例如 ChatGPT）的过程。**之后，LLM可以使用该信息来**增强其生成的**响应内容。

增强生成的步骤：



我们将用户和LLM交互的整个过程看作是一个链（chain）。

- 首先从用户发出提问开始：“我该如何完成<某件事情>？”
- 接下来会先进行检索，从已有知识库中搜索对于该用户问题的所有相关内容知识（这是 RAG 过程中相当重要的一个环节）。
- 接下来我们会将这些从知识库中获取到的相关信息与用户问题一起发送到大语言模型（LLM）
- 最后 LLM（ChatGPT）“读取”所提供的信息并回答问题。

RAG实现细节

LLM（例如ChatGPT）只是一个语言模型，我们怎样才能让LLM来完成这样的工作呢？

本质上所有的这一切都是 **由提供给LLM的定制提示和信息来实现的。**

1. 定制系统提示

定制系统提示是为了 **给予语言模型更具体的指导**。对于ChatGPT，系统提示类似于 **“你是一个实用性的助手”** 这样的描述。在这种情况下，我们希望它能做一些更具体的事情。由于它是一个语言模型，我们可以 **告诉它我们想要它做什么**。

以下是一个简短的系统提示示例，为LLM提供了更详细的说明：

你是一个知识机器人。你将获得从知识库中提取的部分信息（标记文档）和一个问题。请使用知识库中的信息来回答问题。

我们可以看作是在给LLM赋予一个角色，**“hi！人工智能，我会给你一些可以阅读的东西。请阅读它，然后来回答我的问题？谢谢。”**

因为人工智能非常擅长遵循我们的指令，所以它会遵从提示而转换成为一个特定的助手。

2. 提供知识来源

我们需要为人工智能提供阅读材料。由于最新的人工智能模型很擅长 **理解并解决问题**，我们可以通过提供一些 **带有结构和格式的知识信息** 来帮助它来完成任务。

可用于将文档传递给 LLM 的格式：

----- DOCUMENT 1 -----

This document describes the blah blah blah...

----- DOCUMENT 2 -----

This document is another example of using x, y and z...


----- DOCUMENT 3 -----

[more documents here...]


我们必须严格按照上面的这种格式来提供知识信息吗？答案是不一定。请记住！我们所提供的只是一种便于让模型读取和加载的文本格式。

也就是说，可以是我们所熟悉的或任意一种带有规范的文本格式都可以，例如 JSON、YAML、HTML。

这样做的目的只是让我们的提示信息变得更加明确和清晰。

 当然，在一些高级应用场景中，使用一种固定的格式还是非常有必要的。例如，我们需要让LLM 给出知识的引用来源时。

当我们确定了某种格式化的知识文档后，只需将其作为普通聊天消息发送给LLM。

 请记住，在系统提示中我们告诉它我们要给它一些文档

这就是我们在这里所需要做的一切了。

3. 合并内容提出问题

我们准备好了系统提示和带有格式的知识“文档”信息，我们只需将用户的问题与这些内容合并到一起发送给LLM就可以了。

```
openai_response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=[
        {
            "role": "system",
            "content": get_system_prompt(), # 先给出系统提示
        },
        {
            "role": "system",
            "content": get_sources_prompt(), # 然后是格式化的知识文档
        },
        {
            "role": "user",
            "content": user_question, # 想要获得解答的问题
        },
    ],
)
```

根据 自定义系统提示、两条消息，我们就能够获得特定于当前问题内容的解答！

检索步骤：从知识库中获取正确的信息

检索是一种搜索操作——我们希望根据用户的输入查找最相关的信息。

就像搜索一样，有两个主要部分：

1. **索引：**将知识库变成可以搜索/查询的内容。

2. **查询**：通过搜索词从中提取最相关的知识。

这里注意的是，任何 *可用于查询的处理过程* 都可以称之为检索（Retrieval）。类似于接收用户输入后返回相应结果的操作。

那也就是说，我们可以直接寻找一些域问题匹配的相应文档发送给LLM，也可以把搜索引擎的查询结果发送给LLM（微软开发的AI智能机器人Bing就是这个原理）。

当今大多数 RAG 系统都依赖于一种称为 **语义检索（semantic search）** 的技术，它使用的是人工智能的另一项核心技术：**嵌入（Embedding）**。

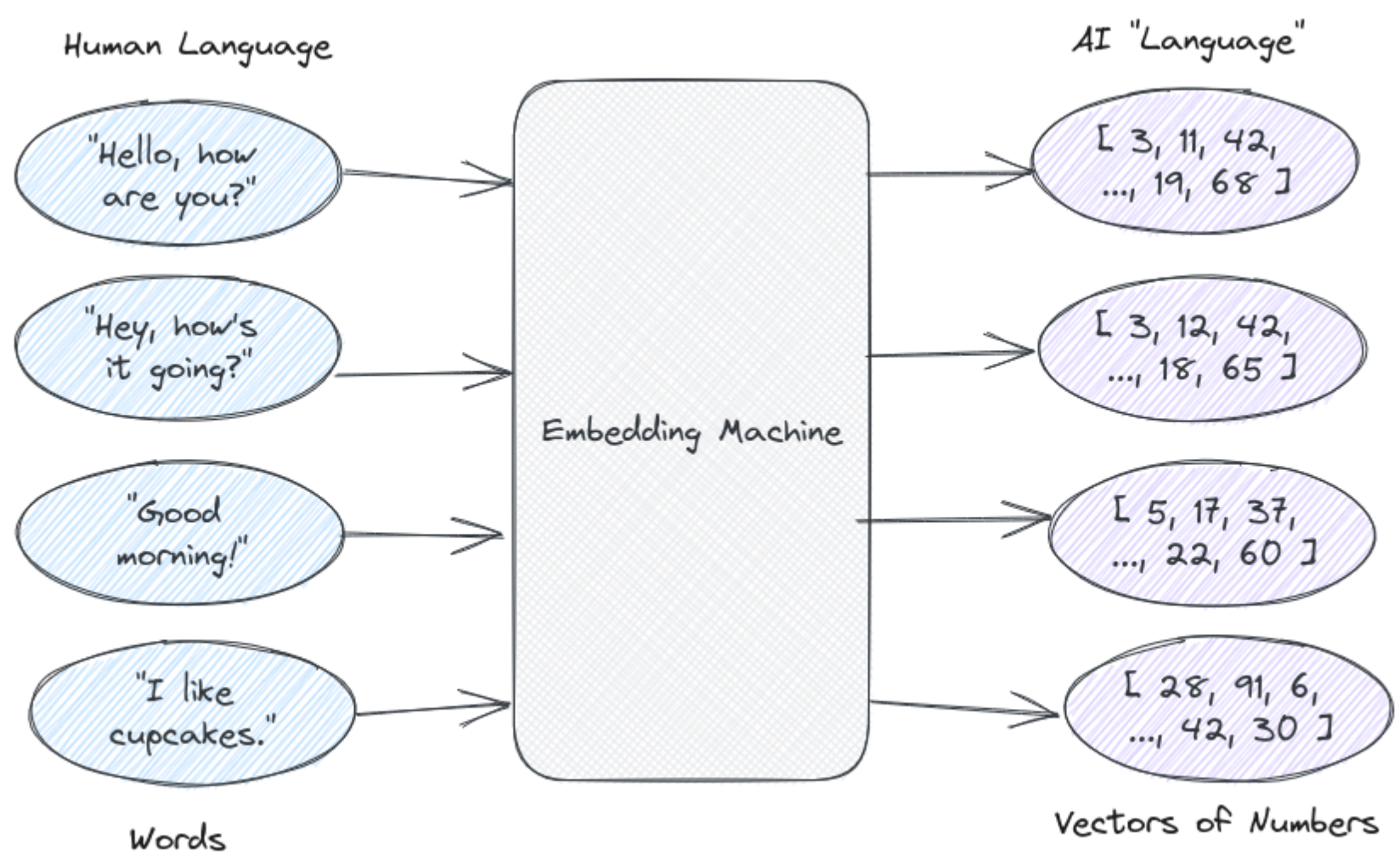
Embedding

如果我们尝试询问一个人，**某段文本或概念该如何解释**？他很可能会根据自己的认知和理解，用自己的表达方式说出一些抽象且与概念相关的描述。

就好像我们经常说的：“我明白你的意思了”。在我们大脑深处的某个地方，有一个复杂的结构，它知道“儿童”和“孩子”本质上是相同的，“红色”和“绿色”都是颜色，“高兴”、“快乐”和“兴高采烈”都代表着相同的情绪，但程度不同。我们无法解释它是如何工作的，我们只是**明白了**。

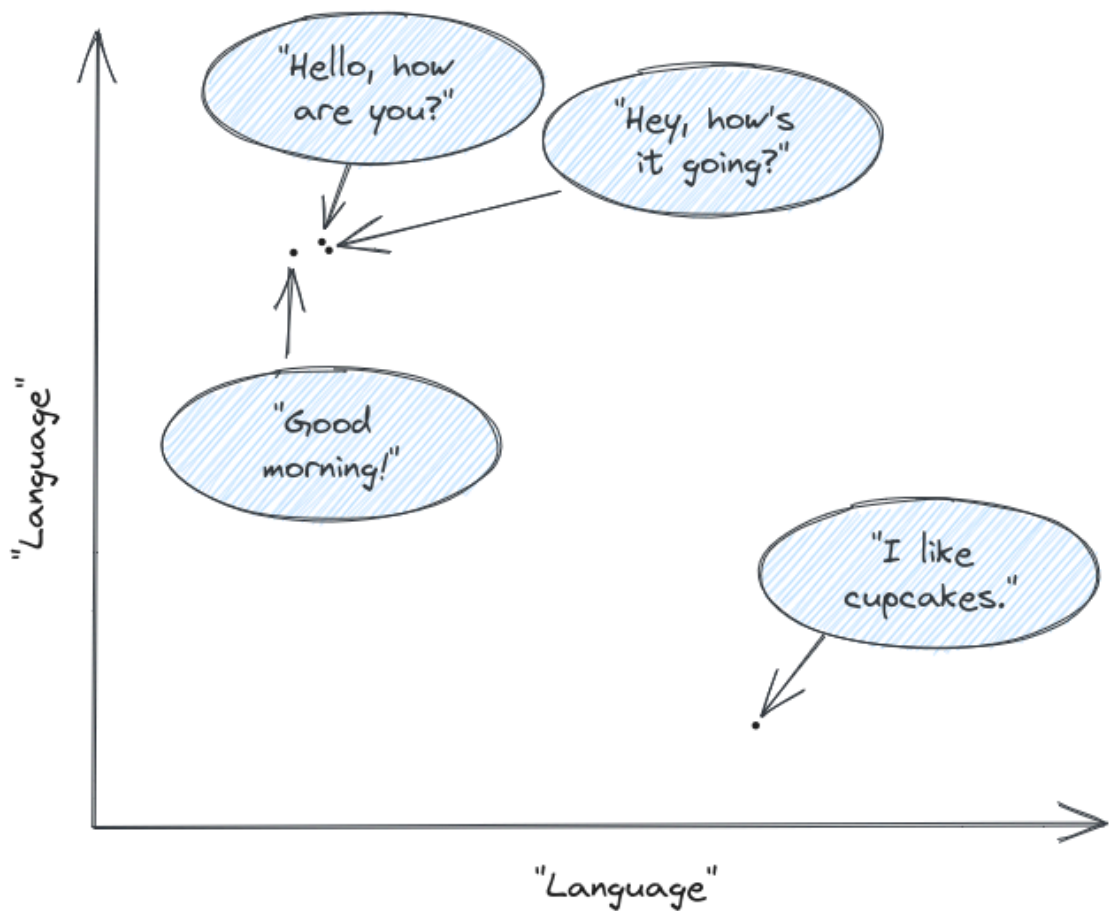
大语言模型对语言也有类似的复杂理解，只不过，由于工作环境是是**计算机**，所以这种理解不存在于大脑中，而是由**数字**组成。在LLM的世界中，任何人类语言都可以表示为数字向量（列表）。这个数字向量就是一个**嵌入（Embedding）**。

LLM 技术的一个关键部分就是从人类**文字语言**到人工智能**数字语言**的**翻译器**。我们将这个翻译器称为“嵌入器（embedding machine）”，尽管在幕后它只是一个 API 调用。人类语言输入，人工智能数字输出。



这些数字向量的组合不是人类可以解释的，它们只对人工智能“有意义”。但是，我们会发现，*相似的单词最终会得到相似的数字组*。因为在幕后，人工智能使用这些数字来“阅读”和“说话”。因此，这些数字在人工智能语言中包含了某种神奇的理解力——即使我们不理解它。嵌入器就是我们的翻译器。

现在，既然我们有了这些神奇的人工智能数字，我们就可以绘制它们了。上述示例的简化图可能看起来像这样——其中x、y轴分别对应着人类/人工智能语言的一些抽象表示：



一旦我们绘制了它们，我们就可以看到，在这个假设的语言空间中，两点彼此越接近，它们所表达的意思就越相似。“你好吗？”和“嘿，怎么样？”实际上是互相叠加的。另一种问候语“早上好”与这些问候语相距不远。“我喜欢纸杯蛋糕”位于一个与其他地方完全不同的点上。

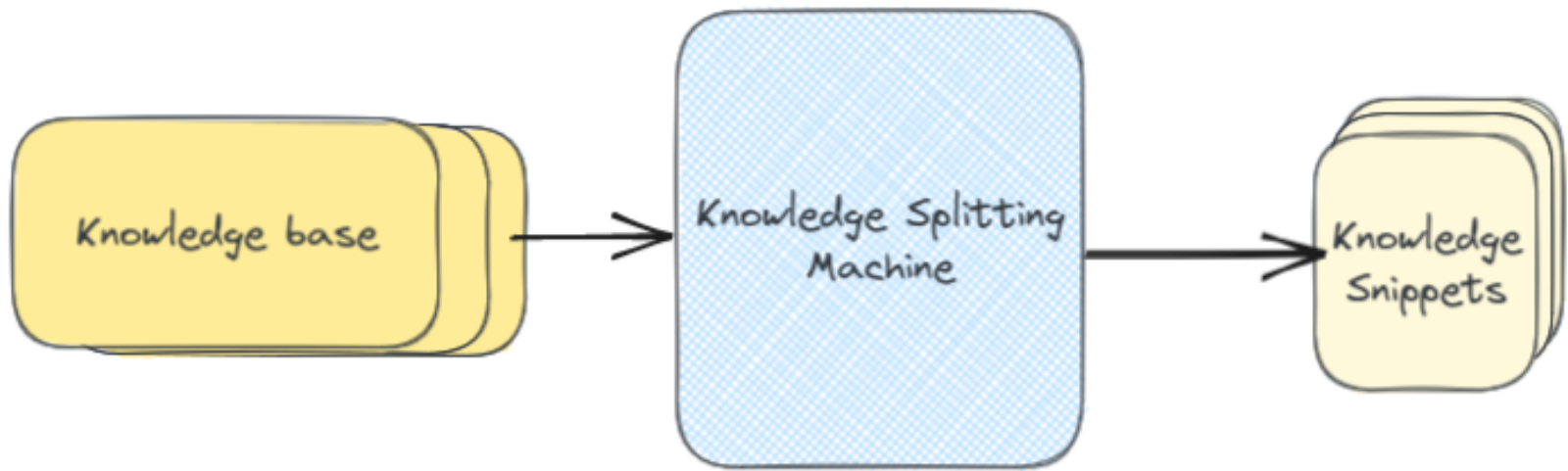
当然，我们不会只在二维坐标图上表示整个人类语言，但理论是相同的。实际上，Embedding 有更多的坐标（OpenAI 当前使用的模型有 1,536 个）。但我们仍然可以进行基本的数学计算来确定两个 Embedding（关联的两段文本）彼此之间的接近程度。

这些嵌入（Embedding）技术和确定“近似度”的计算就是 语义搜索 背后的核心本质，也是支持检索操作的基本步骤。

使用嵌入找到最好的知识片段

一旦我们了解了嵌入搜索的工作原理，我们就可以构建检索步骤的高级图像。

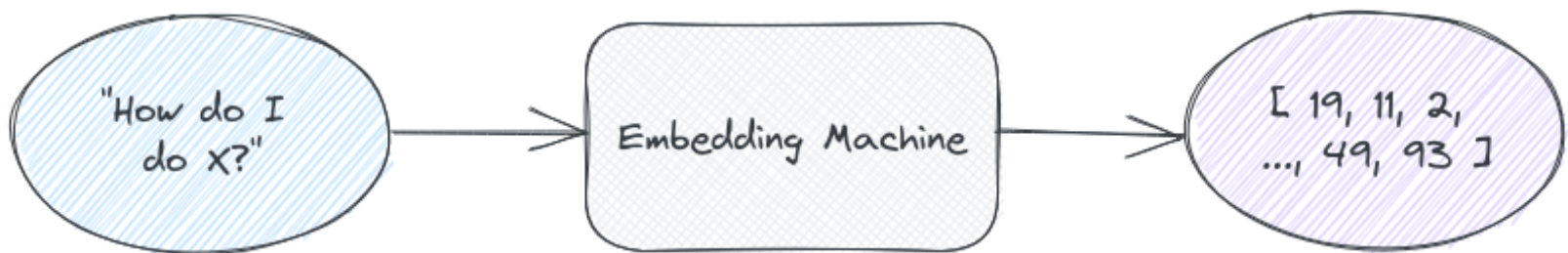
在索引方面，首先我们必须将知识库分解为文本块（chunk）。这个过程本身就是一个完整的优化问题，我们接下来将介绍它，但现在假设我们知道如何做。



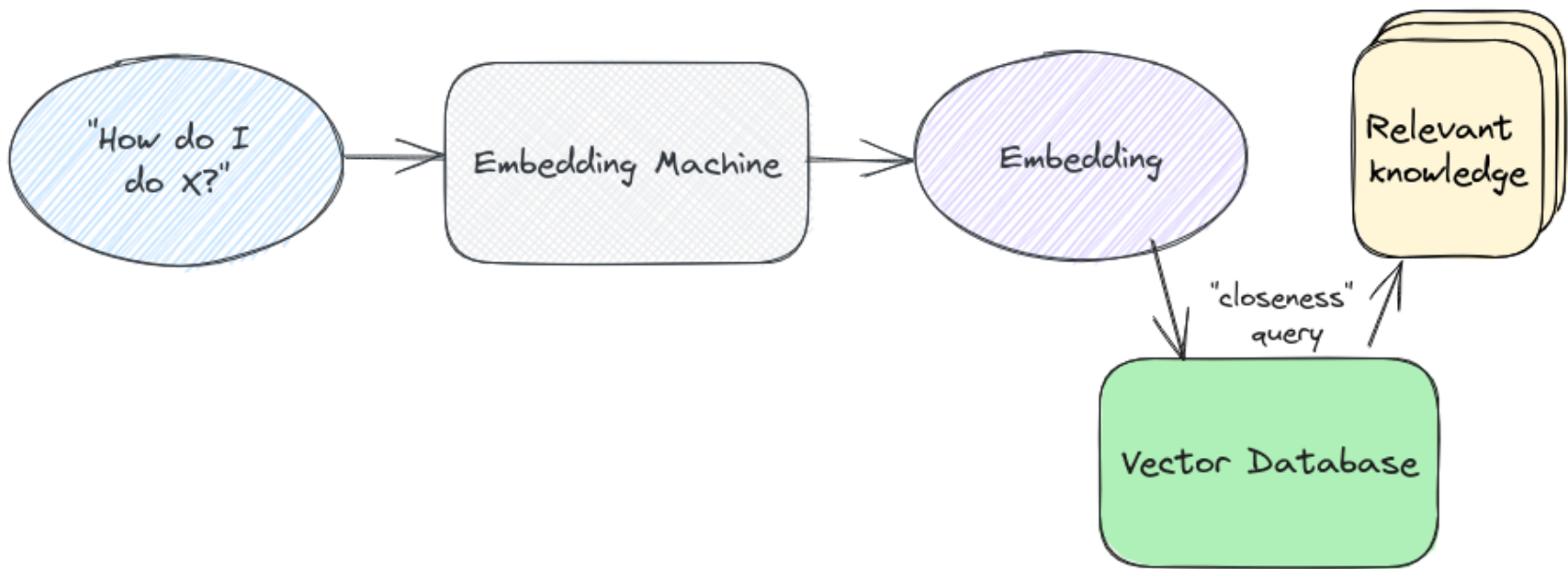
完成此操作后，我们将每个知识片段通过嵌入机器（实际上是 OpenAI API 或类似的机器模型）传递，并返回该文本的 Embedding 表示。然后，我们保存该片段以及 向量数据库 中的嵌入——该数据库针对数字向量进行了优化。

现在我们有了一个数据库，其中嵌入了我们所有的内容。从概念上讲，我们可以将其视为我们整个知识库在“语言”图上的图：

一旦我们有了这个图，在查询方面，我们会执行类似的过程。首先我们获得用户输入的嵌入：



然后我们将其绘制在相同的向量空间中并找到最接近的片段（在本例中为 1 和 2）：
神奇的嵌入器认为这些是与所提出的问题最相关的答案，这些也是我们提取发送给LLM的片段！
在实践中，这个“最近的点是什么”问题是通过查询我们的向量数据库来完成的。所以实际的过程看起来更像是这样的：



查询本身涉及一些复杂的数学——通常使用称为余弦距离的方式来实现计算，当然还有其它的计算方法。从实践的角度来看，很大程度上这些算法都可以封装到第三方库或数据库系统里面。所以我们可以直接通过方法调用来实现以上的这些操作。

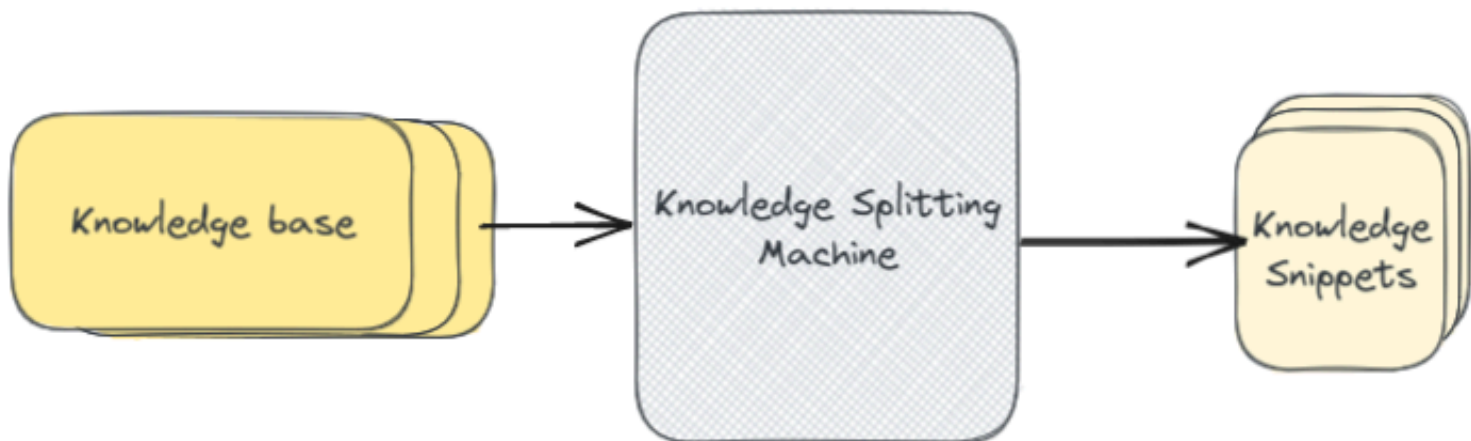
```
# 在 LangChain 示例中，一行代码就已经涵盖了上面提到的所有操作。这个函数的调用隐藏了很多内部的复杂性！

index.query("What should I work on?")
```

索引自定义的知识库

在了解了如何使用 Embedding 技术来实现知识库中最相关部分的查找功能后，将所有内容传递给LLM，就可以获得到增强后的答案。

最后，我们将介绍一些如何从自定义的知识库来创建初始索引。换句话说，这张图中的“知识分裂机”：



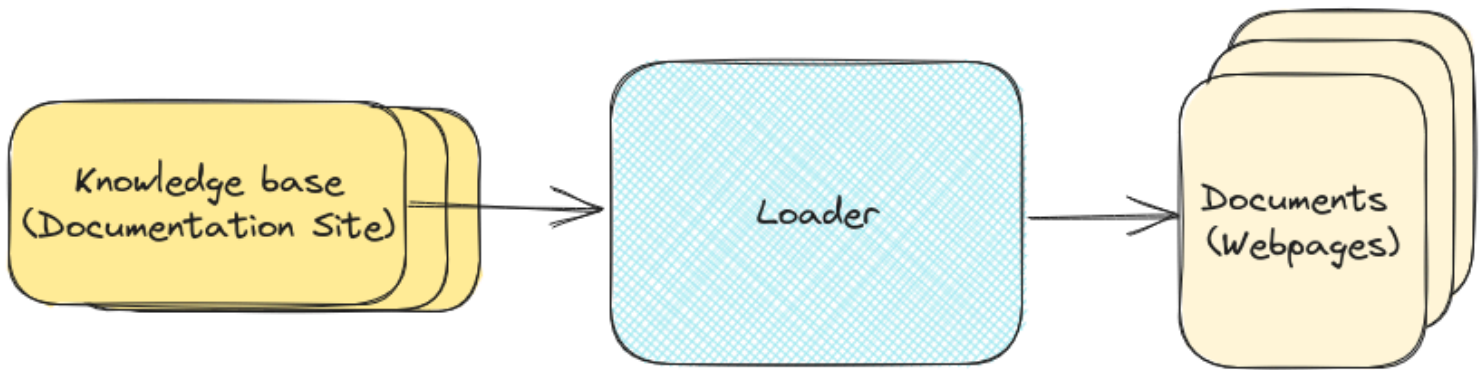
索引自定义的知识库通常是整个过程中最重要也是最难的部分。

究其根本原因是在于，它需要的是更多的艺术而不是科学，并且涉及大量的试验和排错过程。

总体而言，索引过程可归结为两个高级步骤。

- 1. **加载**：从通常存储的位置获取知识库的内容。
- 2. **分割**：将知识分割成适合嵌入搜索的片段。

让我们以自定义的用例为例。构建一个聊天机器人来回答有关于 **SaaS 样板产品 SaaS Pegasus 的** 问题。添加自定义知识库中的第一件事是 **文档站点**。加载器是一个基础设施，它可以访问我们自定义的文档，找出可用的页面，然后拉取每个页面。加载程序完成后，它将输出单独的文档- 网站上的每个页面都有一个文档。



装载机内部实际上完成了很多工作！例如需要抓取所有页面，抓取每个页面的内容，然后将 HTML 格式化为可用的文本。用于其它文本内容（例如 PDF 或 Google Drive）的加载器也是由类似的多个工作流程的组成的。这其中还包括并行化、错误处理等等问题，都需要解决。

这是一个几乎无限复杂的主题，但出于本文的目的，我们将主要将其转移到一个库中。所以现在，我们可以把这个过程的处理看成是一个神奇的盒子，里面有一个“知识库”，然后出来的是单独的“文档”。



内置的加载器是 LangChain 最有用的部分之一。它们提供了一**系列的内置加载器**，可用于从 Microsoft Word 文档到整个 Notion 站点的任何内容中提取内容。

LangChain加载器的接口与上面描述的完全相同。输入一个“知识库”，出来一个“文档”列表。

加载器完成的文档的提取和解析后，我们将获得与文档节点中每个页面相对应的文档集合。此外，理想情况下，此时额外的无关标记已被删除，仅保留底层结构和文本。

现在，我们可以将这些整个网页传递到我们的嵌入器，并将它们用作我们的知识片段。但是，每一页可能涵盖很多内容！而且，页面中的内容越多，该页面的嵌入就越“不具体”。这意味着我们的“接近度”搜索算法可能效果不太好。

更有可能的情况是，用户提出问题的主题只与 页面内的某些文本 相匹配。这就是需要 **拆分（split）** 的地方。通过拆分，我们可以把任何一个文档拆分为更适合搜索的小块、可嵌入的块。

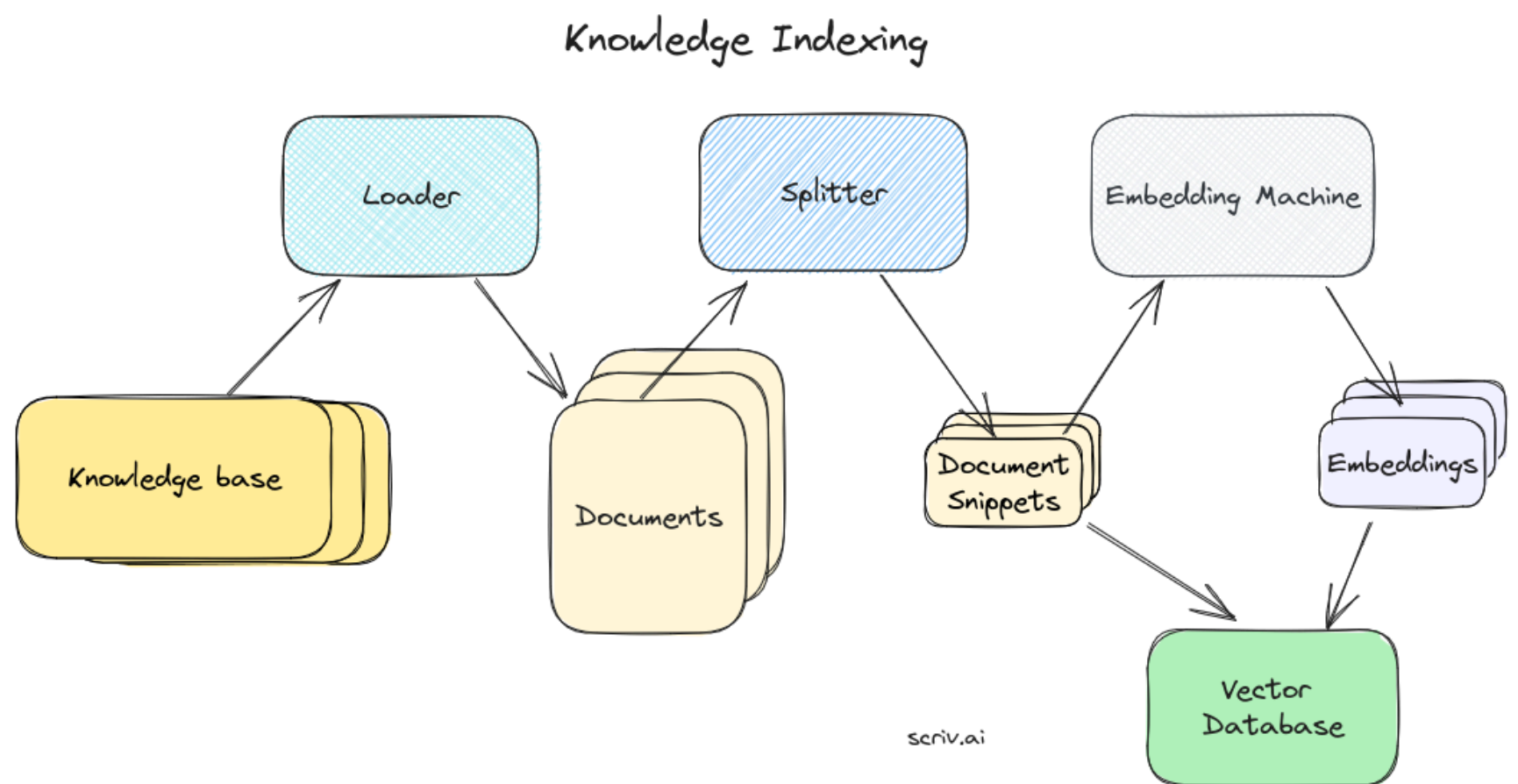
分割文档是一门完整的艺术，包括平均片段的大小（太大，它们不能很好地匹配查询，太小，它们没有足够的有用上下文来生成答案），如何拆分内容（通常按标题，如果有的话），等等。但是，当然，也可以从默认值开始使用和完善我们的数据。



在LangChain中，拆分器属于一个更大的类别，称为“**文档转换器**”。除了提供各种分割文档的策略之外，他们还提供删除冗余内容、翻译、添加元数据等工具。我们在这里只关注拆分器，因为它们实现了绝大多数文档转换的工作。

一旦我们有了文档片段，我们就将它们保存到我们的向量数据库中，如上所述，我们终于完成了！

这是对知识库进行索引的完整图片。



在LangChain中，整个索引过程都封装在这两行代码中。首先，我们初始化网站加载器并告诉它我们要使用什么内容：

```
loader = WebBaseLoader("http://www.paulgraham.com/greatwork.html")
```

然后我们从加载器构建整个索引并将其保存到我们的向量数据库中：

```
index = VectorstoreIndexCreator().from_loaders([loader])
```

加载、分割、嵌入和保存都在后台实现

过程整体回顾

RAG 流水线整体过程

