第七章、聊天 Chat

回想一下检索增强生成 (retrieval augmented generation，RAG) 的整体工作流程：

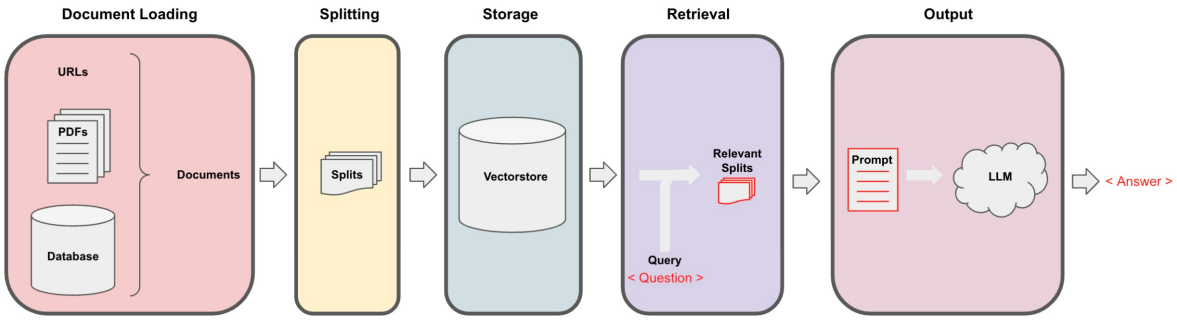

图 4.7.1 RAG

我们已经接近完成一个功能性的聊天机器人了。我们讨论了`文档加载`、`切分`、`存储`和`检索`。我们展示了如何使用`检索 QA`链在 Q+A 中使用`检索`生成输出。

我们的机器人已经可以回答问题了，但还无法处理后续问题，无法进行真正的对话。在本章中，我们将解决这个问题。

我们现在将创建一个问答聊天机器人。它与之前非常相似，但我们将添加聊天历史的功能。这是您之前进行的任何对话或消息。这将使机器人在尝试回答问题时能够考虑到聊天历史的上下文。所以，如果您继续提问，它会知道您想谈论什么。

一、复现之前代码

以下代码是为了 openai LLM 版本备案，直至其被弃用（于 2023 年 9 月）。LLM 响应通常会有所不同，但在使用不同模型版本时，这种差异可能会更明显。

```
import datetime
current_date = datetime.datetime.now().date()
if current_date < datetime.date(2023, 9, 2):
    llm_name = "gpt-3.5-turbo-0301"
else:
    llm_name = "gpt-3.5-turbo"
print(llm_name)
```

```
gpt-3.5-turbo-0301
```

如果您想在`Lang Chain plus`平台上进行实验：

- 前往 langchain plus 平台并注册
- 从您的帐户设置创建 api 密钥
- 在下面的代码中使用此 api 密钥

```
#import os
#os.environ["LANGCHAIN_TRACING_V2"] = "true"
#os.environ["LANGCHAIN_ENDPOINT"] = "https://api.langchain.plus"
#os.environ["LANGCHAIN_API_KEY"] = "..."
```

首先我们加载在前几节课创建的向量数据库，并测试一下：

```
# 加载向量库，其中包含了所有课程材料的 Embedding。
from langchain.vectorstores import Chroma
from langchain.embeddings.openai import OpenAIEmbeddings
import panel as pn  # GUI
# pn.extension()

persist_directory = 'docs/chroma/matplotlib'
embedding = OpenAIEmbeddings()
vectordb = Chroma(persist_directory=persist_directory,
embedding_function=embedding)

question = "这门课的主要内容是什么？"
docs = vectordb.similarity_search(question,k=3)
print(len(docs))
```

```
3
```

接着我们从 OpenAI 的 API 创建一个 LLM：

```
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(model_name=llm_name, temperature=0)
llm.predict("你好")
```

```
'你好！有什么我可以帮助你的吗？'
```

再创建一个基于模板的检索链：

```
# 构建 prompt
from langchain.prompts import PromptTemplate
template = """使用以下上下文来回答最后的问题。如果你不知道答案，就说你不知道，不要试图编造答
案。最多使用三句话。尽量使答案简明扼要。总是在回答的最后说“谢谢你的提问！”。
{context}
问题：{question}
有用的回答："""
QA_CHAIN_PROMPT = PromptTemplate(input_variables=["context",
"question"],template=template,)

# 运行 chain
from langchain.chains import RetrievalQA
question = "这门课的主题是什么？"
qa_chain = RetrievalQA.from_chain_type(llm,
                                       retriever=vectordb.as_retriever(),
                                       return_source_documents=True,
                                       chain_type_kwargs={"prompt":
QA_CHAIN_PROMPT})


result = qa_chain({"query": question})
print(result["result"])
```

```
这门课的主题是 Matplotlib 数据可视化库的初学者指南。
```

二、记忆（Memory）

现在让我们更进一步，添加一些记忆功能。

我们将使用 `ConversationBufferMemory`。它保存聊天消息历史记录的列表，这些历史记录将在回答问题时与问题一起传递给聊天机器人，从而将它们添加到上下文中。

需要注意的是，我们之前讨论的上下文检索等方法，在这里同样可用。

```
from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory(
    memory_key="chat_history", # 与 prompt 的输入变量保持一致。
    return_messages=True # 将以消息列表的形式返回聊天记录，而不是单个字符串
)
```

三、对话检索链（ConversationalRetrievalChain）

对话检索链（ConversationalRetrievalChain）在检索 QA 链的基础上，增加了处理对话历史的能力。

它的工作流程是:

1. 将之前的对话与新问题合并生成一个完整的查询语句。
2. 在向量数据库中搜索该查询的相关文档。
3. 获取结果后,存储所有答案到对话记忆区。
4. 用户可在 UI 中查看完整的对话流程。

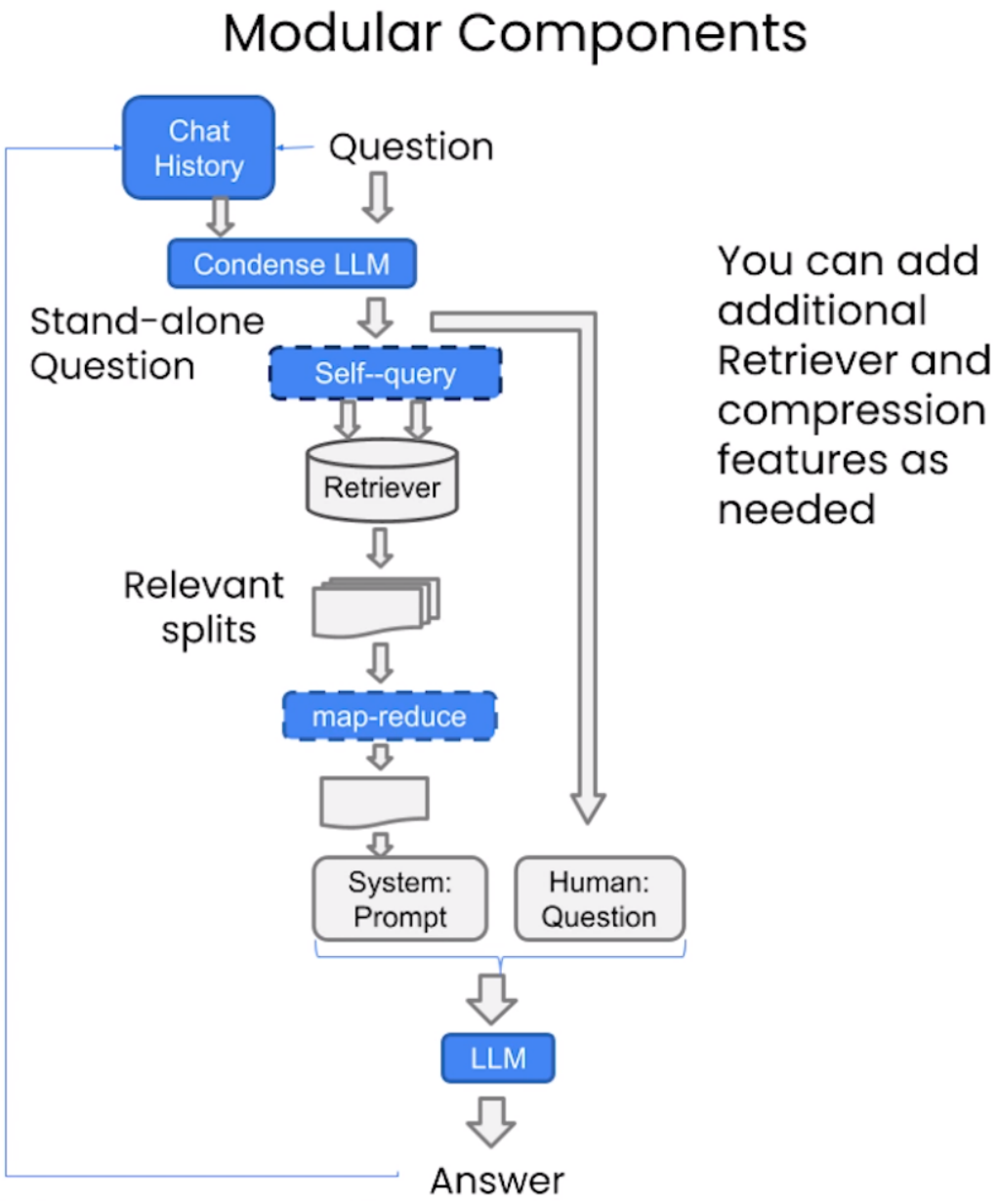

图 4.7.2 对话检索链

这种链式方式将新问题放在之前对话的语境中进行检索，可以处理依赖历史信息的查询。并保留所有信息在对话记忆中，方便追踪。

接下来让我们可以测试这个对话检索链的效果：

首先提出一个无历史的问题“这门课会学习 Python 吗？”，并查看回答。

```
from langchain.chains import ConversationalRetrievalChain
retriever=vectordb.as_retriever()
qa = ConversationalRetrievalChain.from_llm(
    llm,
    retriever=retriever,
    memory=memory
)

question = "这门课会学习 Python 吗？"
result = qa({"question": question})
print(result['answer'])
```

```
是的，这门课程会涉及到 Python 编程语言的使用，特别是在数据可视化方面。因此，学习 Python 是这门课程的前提之一。
```

然后基于答案进行下一个问题“为什么这门课需要这个前提？”：

```
question = "为什么这门课需要这个前提？"
result = qa({"question": question})
print(result['answer'])
```

```
学习Python的前提是需要具备一定的计算机基础知识，包括但不限于计算机操作系统、编程语言基础、数据结构与算法等。此外，对于数据科学领域的学习，还需要具备一定的数学和统计学基础，如线性代数、微积分、概率论与数理统计等。
```

可以看到，虽然 LLM 的回答有些不对劲，但它准确地判断了这个前提的指代内容是学习 Python，也就是我们成功地传递给了它历史信息。这种持续学习和关联前后问题的能力，可大大增强问答系统的连续性和智能水平。

四、定义一个适用于您文档的聊天机器人

通过上述所学内容，我们可以通过以下代码来定义一个适用于私人文档的聊天机器人：

```
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter,
RecursiveCharacterTextSplitter
from langchain.vectorstores import DocArrayInMemorySearch
from langchain.document_loaders import TextLoader
from langchain.chains import RetrievalQA,  ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI
from langchain.document_loaders import TextLoader
from langchain.document_loaders import PyPDFLoader

def load_db(file, chain_type, k):
    """
    该函数用于加载 PDF 文件，切分文档，生成文档的嵌入向量，创建向量数据库，定义检索器，并创建聊天机器人实例。

    参数:
    file (str): 要加载的 PDF 文件路径。
    chain_type (str): 链类型，用于指定聊天机器人的类型。
    k (int): 在检索过程中，返回最相似的 k 个结果。

    返回:
    qa (ConversationalRetrievalChain): 创建的聊天机器人实例。
    """
    # 载入文档
    loader = PyPDFLoader(file)
    documents = loader.load()
    # 切分文档
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
chunk_overlap=150)
    docs = text_splitter.split_documents(documents)
    # 定义 Embeddings
```

```
    embeddings = OpenAIEmbeddings()
    # 根据数据创建向量数据库
    db = DocArrayInMemorySearch.from_documents(docs, embeddings)
    # 定义检索器
    retriever = db.as_retriever(search_type="similarity", search_kwargs={"k": k})
    # 创建 chatbot 链，Memory 由外部管理
    qa = ConversationalRetrievalChain.from_llm(
        llm=ChatOpenAI(model_name=llm_name, temperature=0),
        chain_type=chain_type,
        retriever=retriever,
        return_source_documents=True,
        return_generated_question=True,
    )
    return qa

import panel as pn
import param

# 用于存储聊天记录、回答、数据库查询和回复
class cbfs(param.Parameterized):
    chat_history = param.List([])
    answer = param.String("")
    db_query  = param.String("")
    db_response = param.List([])

    def __init__(self,  **params):
        super(cbfs, self).__init__( **params)
        self.panels = []
        self.loaded_file = "docs/matplotlib/第一回：Matplotlib初相识.pdf"
        self.qa = load_db(self.loaded_file,"stuff", 4)

    # 将文档加载到聊天机器人中
    def call_load_db(self, count):
        """
        count: 数量
        """
        if count == 0 or file_input.value is None:  # 初始化或未指定文件 :
            return pn.pane.Markdown(f"Loaded File: {self.loaded_file}")
        else:
            file_input.save("temp.pdf")  # 本地副本
            self.loaded_file = file_input.filename
            button_load.button_style="outline"
            self.qa = load_db("temp.pdf", "stuff", 4)
            button_load.button_style="solid"
        self.clr_history()
        return pn.pane.Markdown(f"Loaded File: {self.loaded_file}")

    # 处理对话链
    def convchain(self, query):
        """
        query: 用户的查询
        """
        if not query:
            return pn.WidgetBox(pn.Row('User:', pn.pane.Markdown("", width=600)),
scroll=True)
        result = self.qa({"question": query, "chat_history": self.chat_history})
```

```
        self.chat_history.extend([(query, result["answer"])])
        self.db_query = result["generated_question"]
        self.db_response = result["source_documents"]
        self.answer = result['answer']
        self.panels.extend([
            pn.Row('User:', pn.pane.Markdown(query, width=600)),
            pn.Row('ChatBot:', pn.pane.Markdown(self.answer, width=600, style=
{'background-color': '#F6F6F6'}))
        ])
        inp.value = ''  # 清除时清除装载指示器
        return pn.WidgetBox(*self.panels,scroll=True)

    # 获取最后发送到数据库的问题
    @param.depends('db_query ', )
    def get_lquest(self):
        if not self.db_query :
            return pn.Column(
                pn.Row(pn.pane.Markdown(f"Last question to DB:", styles=
{'background-color': '#F6F6F6'})),
                pn.Row(pn.pane.Str("no DB accesses so far"))
            )
        return pn.Column(
            pn.Row(pn.pane.Markdown(f"DB query:", styles={'background-color':
'#F6F6F6'})),
            pn.pane.Str(self.db_query )
        )

    # 获取数据库返回的源文件
    @param.depends('db_response', )
    def get_sources(self):
        if not self.db_response:
            return
        rlist=[pn.Row(pn.pane.Markdown(f"Result of DB lookup:", styles=
{'background-color': '#F6F6F6'}))]
        for doc in self.db_response:
            rlist.append(pn.Row(pn.pane.Str(doc)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)

    # 获取当前聊天记录
    @param.depends('convchain', 'clr_history')
    def get_chats(self):
        if not self.chat_history:
            return pn.WidgetBox(pn.Row(pn.pane.Str("No History Yet")), width=600,
scroll=True)
        rlist=[pn.Row(pn.pane.Markdown(f"Current Chat History variable", styles=
{'background-color': '#F6F6F6'}))]
        for exchange in self.chat_history:
            rlist.append(pn.Row(pn.pane.Str(exchange)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)

    # 清除聊天记录
    def clr_history(self,count=0):
        self.chat_history = []
        return
```

接着可以运行这个聊天机器人：

```
# 初始化聊天机器人
cb = cbfs()

# 定义界面的小部件
file_input = pn.widgets.FileInput(accept='.pdf') # PDF 文件的文件输入小部件
button_load = pn.widgets.Button(name="Load DB", button_type='primary') # 加载数据库的按钮
button_clearhistory = pn.widgets.Button(name="Clear History",
button_type='warning') # 清除聊天记录的按钮
button_clearhistory.on_click(cb.clr_history) # 将清除历史记录功能绑定到按钮上
inp = pn.widgets.TextInput( placeholder='Enter text here…') # 用于用户查询的文本输入小部件

# 将加载数据库和对话的函数绑定到相应的部件上
bound_button_load = pn.bind(cb.call_load_db, button_load.param.clicks)
conversation = pn.bind(cb.convchain, inp)

jpg_pane = pn.pane.Image( './img/convchain.jpg')

# 使用 Panel 定义界面布局
tab1 = pn.Column(
    pn.Row(inp),
    pn.layout.Divider(),
    pn.panel(conversation,  loading_indicator=True, height=300),
    pn.layout.Divider(),
)
tab2= pn.Column(
    pn.panel(cb.get_lquest),
    pn.layout.Divider(),
    pn.panel(cb.get_sources ),
)
tab3= pn.Column(
    pn.panel(cb.get_chats),
    pn.layout.Divider(),
)
tab4=pn.Column(
    pn.Row( file_input, button_load, bound_button_load),
    pn.Row( button_clearhistory, pn.pane.Markdown("Clears chat history. Can use to start a new topic" )),
    pn.layout.Divider(),
    pn.Row(jpg_pane.clone(width=400))
)
# 将所有选项卡合并为一个仪表盘
dashboard = pn.Column(
    pn.Row(pn.pane.Markdown('# ChatWithYourData_Bot')),
    pn.Tabs(('Conversation', tab1), ('Database', tab2), ('Chat History', tab3), ('Configure', tab4))
)
dashboard
```

以下截图展示了该机器人的运行情况：

图 4.7.3 聊天机器人

您可以自由使用并修改上述代码，以添加自定义功能。例如，可以修改 `load_db` 函数和 `convchain` 方法中的配置，尝试不同的存储器模块和检索器模型。

此外，panel 和 Param 这两个库提供了丰富的组件和小工具，可以用来扩展和增强图形用户界面。Panel 可以创建交互式的控制面板，Param 可以声明输入参数并生成控件。组合使用可以构建强大的可配置GUI。

您可以通过创造性地应用这些工具,开发出功能更丰富的对话系统和界面。自定义控件可以实现参数配置、可视化等高级功能。欢迎修改和扩展示例代码,开发出功能更强大、体验更佳的智能对话应用。

五、英文版

1.1 复习

```
# 加载向量库，其中包含了所有课程材料的 Embedding。
from langchain.vectorstores import Chroma
from langchain.embeddings.openai import OpenAIEmbeddings
import panel as pn  # GUI
# pn.extension()

persist_directory = 'docs/chroma/cs229_lectures'
embedding = OpenAIEmbeddings()
vectordb = Chroma(persist_directory=persist_directory,
embedding_function=embedding)

# 对向量库进行基本的相似度搜索
question = "What are major topics for this class?"
docs = vectordb.similarity_search(question,k=3)
print(len(docs))
```

```
3
```

创建一个 LLM

```
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(model_name=llm_name, temperature=0)
llm.predict("Hello world!")
```

```
'Hello there! How can I assist you today?'
```

创建一个基于模板的检索链

```
# 初始化一个 prompt 模板，创建一个检索 QA 链，然后传入一个问题并得到一个结果。
# 构建 prompt
from langchain.prompts import PromptTemplate
template = """Use the following pieces of context to answer the question at the
end. If you don't know the answer, just say that you don't know, don't try to
make up an answer. Use three sentences maximum. Keep the answer as concise as
possible. Always say "thanks for asking!" at the end of the answer.
{context}
Question: {question}
Helpful Answer:"""
QA_CHAIN_PROMPT = PromptTemplate(input_variables=["context",
"question"],template=template,)

# 运行 chain
from langchain.chains import RetrievalQA
question = "Is probability a class topic?"
qa_chain = RetrievalQA.from_chain_type(llm,
                                       retriever=vectordb.as_retriever(),
                                       return_source_documents=True,
                                       chain_type_kwargs={"prompt":
QA_CHAIN_PROMPT})


result = qa_chain({"query": question})
print(result["result"])
```

```
Yes, probability is assumed to be a prerequisite for this class. The instructor
assumes familiarity with basic probability and statistics, and will go over some
of the prerequisites in the discussion sections as a refresher course. Thanks for
asking!
```

2.1 记忆

```
from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory(
    memory_key="chat_history", # 与 prompt 的输入变量保持一致。
    return_messages=True # 将以消息列表的形式返回聊天记录，而不是单个字符串
)
```

3.1 对话检索链

```
from langchain.chains import ConversationalRetrievalChain
```

```
retriever=vectordb.as_retriever()
qa = ConversationalRetrievalChain.from_llm(
    llm,
    retriever=retriever,
    memory=memory
)

question = "Is probability a class topic?"
result = qa({"question": question})
print("Q: ", question)
print("A: ", result['answer'])

question = "why are those prerequesites needed?"
result = qa({"question": question})
print("Q: ", question)
print("A: ", result['answer'])
```

```
Q:  Is probability a class topic?
A:  Yes, probability is a topic that will be assumed to be familiar to students in this class. The instructor assumes that students have familiarity with basic probability and statistics, and that most undergraduate statistics classes will be more than enough.
Q:  why are those prerequesites needed?
A:  The reason for requiring familiarity with basic probability and statistics as prerequisites for this class is that the class assumes that students already know what random variables are, what expectation is, what a variance or a random variable is. The class will not spend much time reviewing these concepts, so students are expected to have a basic understanding of them before taking the class.
```

4.1 定义一个聊天机器人

```
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter,
RecursiveCharacterTextSplitter
from langchain.vectorstores import DocArrayInMemorySearch
from langchain.document_loaders import TextLoader
from langchain.chains import RetrievalQA,  ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI
from langchain.document_loaders import TextLoader
from langchain.document_loaders import PyPDFLoader

def load_db(file, chain_type, k):
    """
    该函数用于加载 PDF 文件，切分文档，生成文档的嵌入向量，创建向量数据库，定义检索器，并创建聊天机器人实例。

    参数:
    file (str): 要加载的 PDF 文件路径。
    chain_type (str): 链类型，用于指定聊天机器人的类型。
    k (int): 在检索过程中，返回最相似的 k 个结果。

    返回:
```

```
    qa (ConversationalRetrievalChain): 创建的聊天机器人实例。
    """
    # 载入文档
    loader = PyPDFLoader(file)
    documents = loader.load()
    # 切分文档
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
chunk_overlap=150)
    docs = text_splitter.split_documents(documents)
    # 定义 Embeddings
    embeddings = OpenAIEmbeddings()
    # 根据数据创建向量数据库
    db = DocArrayInMemorySearch.from_documents(docs, embeddings)
    # 定义检索器
    retriever = db.as_retriever(search_type="similarity", search_kwargs={"k": k})
    # 创建 chatbot 链，Memory 由外部管理
    qa = ConversationalRetrievalChain.from_llm(
        llm=ChatOpenAI(model_name=llm_name, temperature=0),
        chain_type=chain_type,
        retriever=retriever,
        return_source_documents=True,
        return_generated_question=True,
    )
    return qa

import panel as pn
import param

# 用于存储聊天记录、回答、数据库查询和回复
class cbfs(param.Parameterized):
    chat_history = param.List([])
    answer = param.String("")
    db_query  = param.String("")
    db_response = param.List([])

    def __init__(self,  **params):
        super(cbfs, self).__init__( **params)
        self.panels = []
        self.loaded_file = "docs/cs229_lectures/MachineLearning-Lecture01.pdf"
        self.qa = load_db(self.loaded_file,"stuff", 4)

    # 将文档加载到聊天机器人中
    def call_load_db(self, count):
        """
        count: 数量
        """
        if count == 0 or file_input.value is None:  # 初始化或未指定文件 :
            return pn.pane.Markdown(f"Loaded File: {self.loaded_file}")
        else:
            file_input.save("temp.pdf")  # 本地副本
            self.loaded_file = file_input.filename
            button_load.button_style="outline"
            self.qa = load_db("temp.pdf", "stuff", 4)
            button_load.button_style="solid"
        self.clr_history()
        return pn.pane.Markdown(f"Loaded File: {self.loaded_file}")
```

```
    # 处理对话链
    def convchain(self, query):
        """
        query: 用户的查询
        """
        if not query:
            return pn.WidgetBox(pn.Row('User:', pn.pane.Markdown("", width=600)),
scroll=True)
        result = self.qa({"question": query, "chat_history": self.chat_history})
        self.chat_history.extend([(query, result["answer"])])
        self.db_query = result["generated_question"]
        self.db_response = result["source_documents"]
        self.answer = result['answer']
        self.panels.extend([
            pn.Row('User:', pn.pane.Markdown(query, width=600)),
            pn.Row('ChatBot:', pn.pane.Markdown(self.answer, width=600, style=
{'background-color': '#F6F6F6'}))
        ])
        inp.value = ''  # 清除时清除装载指示器
        return pn.WidgetBox(*self.panels,scroll=True)

    # 获取最后发送到数据库的问题
    @param.depends('db_query ', )
    def get_lquest(self):
        if not self.db_query :
            return pn.Column(
                pn.Row(pn.pane.Markdown(f"Last question to DB:", styles=
{'background-color': '#F6F6F6'})),
                pn.Row(pn.pane.Str("no DB accesses so far"))
            )
        return pn.Column(
            pn.Row(pn.pane.Markdown(f"DB query:", styles={'background-color':
'#F6F6F6'})),
            pn.pane.Str(self.db_query )
        )

    # 获取数据库返回的源文件
    @param.depends('db_response', )
    def get_sources(self):
        if not self.db_response:
            return
        rlist=[pn.Row(pn.pane.Markdown(f"Result of DB lookup:", styles=
{'background-color': '#F6F6F6'}))]
        for doc in self.db_response:
            rlist.append(pn.Row(pn.pane.Str(doc)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)

    # 获取当前聊天记录
    @param.depends('convchain', 'clr_history')
    def get_chats(self):
        if not self.chat_history:
            return pn.WidgetBox(pn.Row(pn.pane.Str("No History Yet")), width=600,
scroll=True)
        rlist=[pn.Row(pn.pane.Markdown(f"Current Chat History variable", styles=
{'background-color': '#F6F6F6'}))]
```

```
        for exchange in self.chat_history:
            rlist.append(pn.Row(pn.pane.Str(exchange)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)

    # 清除聊天记录
    def clr_history(self,count=0):
        self.chat_history = []
        return
```

4.2 创建聊天机器人

```
# 初始化聊天机器人
cb = cbfs()

# 定义界面的小部件
file_input = pn.widgets.FileInput(accept='.pdf') # PDF 文件的文件输入小部件
button_load = pn.widgets.Button(name="Load DB", button_type='primary') # 加载数据库的按钮
button_clearhistory = pn.widgets.Button(name="Clear History",
button_type='warning') # 清除聊天记录的按钮
button_clearhistory.on_click(cb.clr_history) # 将清除历史记录功能绑定到按钮上
inp = pn.widgets.TextInput( placeholder='Enter text here…') # 用于用户查询的文本输入小部件

# 将加载数据库和对话的函数绑定到相应的部件上
bound_button_load = pn.bind(cb.call_load_db, button_load.param.clicks)
conversation = pn.bind(cb.convchain, inp)

jpg_pane = pn.pane.Image( './img/convchain.jpg')

# 使用 Panel 定义界面布局
tab1 = pn.Column(
    pn.Row(inp),
    pn.layout.Divider(),
    pn.panel(conversation,  loading_indicator=True, height=300),
    pn.layout.Divider(),
)
tab2= pn.Column(
    pn.panel(cb.get_lquest),
    pn.layout.Divider(),
    pn.panel(cb.get_sources ),
)
tab3= pn.Column(
    pn.panel(cb.get_chats),
    pn.layout.Divider(),
)
tab4=pn.Column(
    pn.Row( file_input, button_load, bound_button_load),
    pn.Row( button_clearhistory, pn.pane.Markdown("Clears chat history. Can use to start a new topic" )),
    pn.layout.Divider(),
    pn.Row(jpg_pane.clone(width=400))
)
# 将所有选项卡合并为一个仪表盘
```

```
dashboard = pn.Column(
    pn.Row(pn.pane.Markdown('# ChatWithYourData_Bot')),
    pn.Tabs(('Conversation', tab1), ('Database', tab2), ('Chat History', tab3),
('Configure', tab4))
)
dashboard
```