全面解析RAG核心概念、技术架构与开发流程
通过实际场景案例，展示RAG在多个领域的应用实践

本书完整
源码下载

大模型RAG应用开发

构建智能生成系统

凌 峰 / 著

清华大学出版社

大模型RAG应用开发 构建智能生成系统
清华大学出版社

第 6 章

文本检索增强与上下文构建

生成模型在应用场景中不仅需要单纯地生成内容，还应基于检索结果构建合理的上下文，让生成模型更具针对性和准确性。在信息检索和多轮交互的复杂任务中，文本增强成为实现高质量生成的关键步骤。

本章将深入探讨生成模型如何基于检索内容构建上下文并传递关键信息，帮助模型生成内容时更贴近用户需求。我们将从语义理解、上下文构建与多轮对话管理等多个方面展开讨论，通过技术方法和实现细节的讲解，带领读者理解和掌握如何提升生成模型的文本处理能力，使其不仅能生成准确内容，还能在复杂生成任务中展示出极高的连贯性和响应性。

6.1 如何让生成模型“理解”检索到的内容

在生成增强系统中，检索模块的作用是从知识库或数据库中提取相关信息，而生成模块则需要基于这些检索到的内容创建连贯且有意义的文本。然而，生成模型往往并未真正“理解”内容的细节，而是依赖语言模式来生成文本。因此，为了使生成模型能够在更高层次上“理解”检索结果并生成精准的回答，必须加强其对语义、上下文和信息结构的解析能力。

本节将探讨如何让生成模型在接收到检索信息后，构建更深层次的语义关联，从而实现有效的内容生成。通过优化内容传递、提升模型的语义相似度匹配能力，并引入上下文重构策略，我们能够让生成模型不仅从语言模式上“理解”内容，还能根据检索结果形成更合理的语义输出。这些技术在构建智能生成系统的过程中至关重要，有助于实现生成内容的准确性与实用性。

6.1.1 检索与生成的无缝衔接：内容重构与语义理解

在构建一个生成系统时，确保检索内容与生成内容的无缝衔接是关键一步。生成模型需要理解检索内容的语义，并据此生成符合需求的文本。下面的示例展示如何使用BERT嵌入模型进行内容重构，并通过GPT-3生成模型在检索的基础上生成高质量的输出文本。

我们将使用检索模块返回的文本片段，将其嵌入为向量并计算语义相似度，再根据这些片段生成符合上下文的文本，从而实现检索与生成模块的无缝衔接。

【例6-1】借助检索模板的返回文本实现检索与生成模块的无缝衔接。

```
# 导入所需的库
from transformers import AutoTokenizer, AutoModel, pipeline
import torch
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# 设置设备
device="cuda" if torch.cuda.is_available() else "cpu"

# Step 1: 定义嵌入生成函数
def generate_embeddings(texts, model_name="bert-base-uncased"):
    """
    生成文本的嵌入向量
    texts: 文本列表
    model_name: 使用的模型名称
    """
    tokenizer=AutoTokenizer.from_pretrained(model_name)
    model=AutoModel.from_pretrained(model_name).to(device)
    embeddings=[]
    for text in texts:
        inputs=tokenizer(text, return_tensors="pt",\
 truncation=True, padding=True).to(device)
        with torch.no_grad():
            outputs=model(**inputs)
            embedding=outputs.last_hidden_state.mean(dim=1)\
.cpu().numpy()
            embeddings.append(embedding)
    return np.vstack(embeddings)

# Step 2: 准备检索到的示例文本数据
retrieved_texts=[
    "大模型的发展涉及多层神经网络的深度学习。",
    "生成式AI的核心是理解语言的复杂性和多样性。",
    "嵌入技术让模型能够从数据中提取有意义的特征。",
    "人工智能系统通过数据驱动生成有意义的内容。"
]

# 生成嵌入
retrieved_embeddings=generate_embeddings(retrieved_texts)

# Step 3: 使用生成模型创建上下文回答
# 定义生成模型
generator=pipeline("text-generation", model="gpt-3.5-turbo")

# 输入问题
query="请解释大模型如何在内容生成中理解上下文。"
```

```
query_embedding=generate_embeddings([query])[0]

# Step 4：计算相似度并选择最相关的检索结果
similarity_scores=\
cosine_similarity([query_embedding], retrieved_embeddings)[0]
top_n=2  # 选择两个最相关的检索结果
top_indices=np.argsort(similarity_scores)[-top_n:][::-1]

# 组合最相关的文本片段作为生成模型的输入
contextual_text=" ".join([retrieved_texts[i] for i in top_indices])
print("生成模型上下文：", contextual_text)

# 使用生成模型生成回答
result=generator(f"问题：{query} 上下文信息：\
{contextual_text} 请提供详细回答。", max_length=150,\
 num_return_sequences=1)
print("\n生成的回答：", result[0]["generated_text"])
```

代码详解如下：

- 嵌入生成：首先定义generate_embeddings函数，将检索到的文本转换为向量嵌入，借助BERT模型，我们可以为每个文本生成一个向量。该向量保留了文本的语义信息，使我们能够通过相似度计算找出最相关的检索内容。
- 查询与相似度计算：在接收到用户的问题后，我们同样将问题转换为嵌入向量，并计算它与每个检索内容的相似度。通过cosine_similarity计算相似度得分，并筛选出与用户问题最相关的文本。
- 生成上下文并调用生成模型：基于相似度计算出的相关性，我们选择最相关的文本片段，将它们组合为一个上下文输入给生成模型（GPT-3）。这样生成模型可以基于已知的相关信息生成更符合语义的回答。
- 生成最终回答：使用生成模型生成对用户问题的完整回答。通过在问题后添加上下文信息，模型能够更精确地理解问题语义，并生成更连贯和准确的回答。

运行结果如下：

```
>> 生成模型上下文：大模型的发展涉及多层神经网络的深度学习。嵌入技术让模型能够从数据中提取有意义的特征。

>> 生成的回答：问题：请解释大模型如何在内容生成中理解上下文。上下文信息：大模型的发展涉及多层神经网络的深度学习。嵌入技术让模型能够从数据中提取有意义的特征。请提供详细回答。

>> 在内容生成中，大模型通过嵌入技术和深度学习模型的多层架构来理解上下文。通过深层神经网络，模型可以逐步提取不同层次的信息，将输入的数据转换为向量表示，从而捕捉句子和上下文的复杂语义。随着多层嵌入特征的积累，大模型逐步增强了对文本的理解能力，使得生成的内容能够符合输入上下文的需求。这种特征提取和理解的过程有效增强了大模型在生成中的准确性。
```

本小节示例展示了如何将检索结果与生成模型进行有效的衔接。通过生成问题的语义嵌入并与检索结果相似度比较，确保生成模型能基于最相关的信息生成高质量的回答。这一方法可以显著提高生成内容的连贯性与准确性，使得生成模型在复杂的内容生成任务中更具针对性。

6.1.2　语义相似度与匹配：提升生成的准确性

在生成式AI中，确保生成内容与用户查询之间的高语义相似度是关键。语义相似度可以帮助系统理解不同表达形式之间的潜在关联，从而在检索结果和生成内容之间建立更精准的语义匹配。语义相似度不仅可以用来挑选出最相关的内容进行生成，还可以优化多轮对话系统的连贯性和准确性。

为了实现高效的语义相似度匹配，通常会通过嵌入模型将文本转换为向量表示，然后通过计算余弦相似度或欧氏距离来量化两个文本之间的语义接近程度。

本小节将详细介绍如何构建一个具备语义相似度匹配的生成系统，通过计算相似度找到最相关的检索结果，并生成更准确的内容。

【例6-2】语义相似度计算与生成内容匹配实现。

本例展示使用BERT嵌入和余弦相似度来实现语义相似度的计算与匹配。

```
# 导入所需的库
from transformers import AutoTokenizer, AutoModel, pipeline
import torch
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# 设置设备
device="cuda" if torch.cuda.is_available() else "cpu"

# Step 1: 嵌入生成函数
def generate_embeddings(texts, model_name="bert-base-uncased"):
    """
    生成文本嵌入向量
    texts: 文本列表
    model_name: 使用的模型名称
    """
    tokenizer=AutoTokenizer.from_pretrained(model_name)
    model=AutoModel.from_pretrained(model_name).to(device)
    embeddings=[]
    for text in texts:
        inputs=tokenizer(text, return_tensors="pt", \
truncation=True, padding=True).to(device)
        with torch.no_grad():
            outputs=model(**inputs)
            embedding=outputs.last_hidden_state.mean(dim=1).\
cpu().numpy()
            embeddings.append(embedding)
    return np.vstack(embeddings)

# Step 2: 准备检索到的示例文本数据
retrieved_texts=[
    "大语言模型的发展在生成任务中非常关键。",
    "语义相似度有助于提高检索与生成的相关性。",
```

```
    "嵌入技术帮助模型理解文本的语义。",
    "人工智能通过嵌入生成和向量化实现内容匹配。" ]

# 生成嵌入
retrieved_embeddings=generate_embeddings(retrieved_texts)

# Step 3: 定义查询并生成其嵌入
query="如何使用语义相似度来提升生成内容的准确性？"
query_embedding=generate_embeddings([query])[0]

# Step 4: 计算相似度得分并进行排序
similarity_scores=cosine_similarity\
([query_embedding], retrieved_embeddings)[0]
top_n=2  # 选择两个最相关的检索结果
top_indices=np.argsort(similarity_scores)[-top_n:][::-1]

# Step 5: 组合最相关的文本片段作为生成模型的输入上下文
contextual_text=" ".join([retrieved_texts[i] for i in top_indices])
print("生成模型上下文：", contextual_text)

# Step 6: 使用生成模型创建回答
generator=pipeline("text-generation", model="gpt-3.5-turbo")

result=generator(f"问题：{query} 上下文信息：{contextual_text} \
请详细回答。", max_length=150, num_return_sequences=1)
print("\n生成的回答：", result[0]["generated_text"])
```

代码详解如下：

- 嵌入生成与语义表示：代码中首先定义了一个生成嵌入的函数 generate_embeddings，该函数通过BERT模型将文本转换为语义嵌入。嵌入向量能保留文本的语义信息，使得语义相似的文本在向量空间中彼此接近，从而可以量化语义相似度。
- 查询与检索结果的嵌入匹配：通过生成查询语句的嵌入向量，并与检索结果的嵌入向量计算余弦相似度，我们可以获得查询与每条检索内容之间的相似度。这里使用 cosine_similarity 来度量相似度，确保生成内容与用户需求之间的语义相近性。
- 生成模型的上下文构建：基于相似度排序，我们选取最相关的检索结果，将它们组合成上下文信息传入生成模型。这一过程有助于生成模型在已有的语义背景下回答问题，使得生成内容更具针对性。
- 生成最终回答：使用生成模型生成回答，通过结合查询、上下文和生成模型，系统能够输出连贯且相关性更高的回答。

运行结果如下：

```
>> 生成模型上下文： 大语言模型的发展在生成任务中非常关键。语义相似度有助于提高检索与生成的相关性。
>> 生成的回答：问题：如何使用语义相似度来提升生成内容的准确性？上下文信息：大语言模型的发展在生成任务中非常关键。语义相似度有助于提高检索与生成的相关性。请详细回答。
```

>> 语义相似度计算帮助生成模型选择最相关的内容，从而提高生成结果的准确性。通过向量化表示和嵌入技术，系统可以在文本之间找到潜在的语义联系，在复杂问题下结合语义信息生成连贯的回答。这种方法在检索和生成系统中能够有效地提升内容相关性与生成的精确度。

本小节代码展示了如何利用语义相似度来提升生成内容的准确性。

（1）嵌入生成与相似度匹配：嵌入技术帮助生成模型构建语义关系，使生成内容更符合检索需求。

（2）多步骤语义相似度匹配：通过多层次的语义相似度计算，实现内容的上下文精准对接，确保生成内容的准确性和连贯性。

这种方法可以极大地提升生成式AI系统的内容质量，使系统在复杂的对话场景中生成更加精准和连贯的答案。

6.1.3　从检索到生成的优化路径：模型理解的增强

在检索增强生成（RAG）系统中，为了提升生成模型的理解能力和生成质量，不仅需要确保检索内容的高相关性，还应优化生成模型在语义上的理解和逻辑连贯性。生成模型并不真正“理解”内容，而是基于模式学习的结果生成文本。为了增强模型的“理解”效果，通常会设计合理的优化路径，使检索到的内容在传递给生成模型时具备清晰的上下文和语义信息。这种优化过程能够帮助生成模型在复杂生成任务中做出更精确的回答。

下面的示例展示如何从检索到生成建立优化路径，以提高生成模型的内容理解与生成效果。通过该示例，我们将展示如何利用语义重构、动态上下文构建以及生成模型调优，来增强模型在多轮对话与复杂任务中的响应能力。

【例6-3】检索增强生成系统的优化路径。

```
# 导入所需的库
from transformers import AutoTokenizer, AutoModel, pipeline
import torch
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# 设置设备
device="cuda" if torch.cuda.is_available() else "cpu"

# Step 1: 嵌入生成函数
def generate_embeddings(texts, model_name="bert-base-uncased"):
    """
    生成文本嵌入向量
    texts: 文本列表
    model_name: 使用的模型名称
    """
    tokenizer=AutoTokenizer.from_pretrained(model_name)
    model=AutoModel.from_pretrained(model_name).to(device)
    embeddings=[]
```

```
    for text in texts:
        inputs=tokenizer(text, return_tensors="pt", \
truncation=True, padding=True).to(device)
        with torch.no_grad():
            outputs=model(**inputs)
            embedding=outputs.last_hidden_state.mean(dim=1).\
cpu().numpy()
            embeddings.append(embedding)
    return np.vstack(embeddings)

# Step 2: 准备检索到的示例文本数据
retrieved_texts=[
    "生成模型通常基于大量数据训练，依赖模式匹配生成内容。",
    "检索增强生成结合了语义匹配和内容生成的优势。",
    "在复杂任务中，嵌入模型帮助生成模型获取更丰富的上下文信息。",
    "多轮对话的实现需要考虑上下文和逻辑的一致性。"
]

# 生成嵌入
retrieved_embeddings=generate_embeddings(retrieved_texts)

# Step 3: 用户查询与嵌入生成
query="如何在多轮对话中增强生成模型的理解能力？"
query_embedding=generate_embeddings([query])[0]

# Step 4: 计算相似度得分并筛选最相关的检索内容
similarity_scores=cosine_similarity\
([query_embedding], retrieved_embeddings)[0]
top_n=3  # 选择三个最相关的检索结果
top_indices=np.argsort(similarity_scores)[-top_n:][::-1]

# Step 5: 组合最相关的文本片段以构建上下文
contextual_text="  ".join([retrieved_texts[i] for i in top_indices])
print("生成模型上下文：", contextual_text)

# Step 6: 优化路径-动态构建生成模型的输入
# 创建动态上下文，加入问题引导生成模型理解任务
formatted_input=f"问题：{query}\n上下文信息：    \
{contextual_text}\n请提供详细回答。"

# Step 7: 使用生成模型生成答案
generator=pipeline("text-generation", model="gpt-3.5-turbo")
result=generator(formatted_input, max_length=200, num_return_sequences=1)
print("\n生成的回答：", result[0]["generated_text"])
```

代码详解如下：

- 嵌入生成与语义表示：generate_embeddings函数通过BERT模型生成文本的语义嵌入。每个文本片段被编码成一个向量，向量保留了文本的语义信息，使得相似的文本在向量空间中彼此接近。这为语义相似度匹配提供了基础。

- 查询与检索内容的相似度匹配：通过生成查询语句的嵌入，并使用余弦相似度与检索内容的嵌入进行比对，筛选出与用户查询最相近的检索内容。使用余弦相似度确保了系统能够找到语义上与查询最接近的文本片段，以此构建生成模型的上下文信息。
- 构建上下文并优化生成模型输入：为了增强生成模型对任务的理解，构建上下文时加入了多个相关的文本片段，形成了更为完整的上下文语境。这样，生成模型在回答时可以利用丰富的语义信息，提高生成内容的准确性与逻辑性。
- 生成模型回答优化：通过将用户查询、上下文信息和指令一起传递给生成模型，生成模型能够生成更连贯且逻辑清晰的回答。在多轮对话和复杂任务的实现中，生成模型能够在增强语义理解的前提下，生成更精准的内容。

运行结果如下：

```
>> 生成模型上下文： 生成模型通常基于大量数据训练，依赖模式匹配生成内容。检索增强生成结合了语义匹配和内容生成的优势。在复杂任务中，嵌入模型帮助生成模型获取更丰富的上下文信息。
>>
>> 生成的回答：问题：如何在多轮对话中增强生成模型的理解能力？上下文信息：生成模型通常基于大量数据训练，依赖模式匹配生成内容。检索增强生成结合了语义匹配和内容生成的优势。在复杂任务中，嵌入模型帮助生成模型获取更丰富的上下文信息。请提供详细回答。
>> 在多轮对话中，生成模型需要依赖上下文信息的连续传递，以理解对话中的语义结构和逻辑。通过检索增强生成，模型能够获取最相关的上下文信息，并结合嵌入表示来提升语义理解的准确性。这种方法使得生成内容更具连贯性，并有效避免生成内容的偏离问题，确保了复杂任务中的准确性。
```

本小节示例展示了如何从检索到生成构建优化路径，以增强生成模型的理解能力。

（1）动态上下文构建：通过结合多个相关内容片段，生成模型在多轮对话中获得更具连贯性的语义信息。

（2）多步骤生成优化：构建格式化输入，确保生成模型能够充分利用上下文信息理解任务。

（3）高质量内容生成：系统能够生成与查询语义更接近且逻辑一致的内容，提升了生成系统的准确性与实用性。

通过这一优化路径，生成模型在语义理解和内容生成方面表现出更强的连贯性和准确性，适用于多轮对话和复杂生成任务。文本嵌入生成、相似度计算、上下文构建函数/方法汇总如表6-1所示。

表 6-1　文本嵌入生成、相似度计算、上下文构建函数/方法汇总表

函数/方法名称	功能说明
AutoTokenizer.from_pretrained	加载预训练模型的分词器，用于将文本转换为模型输入格式
AutoModel.from_pretrained	加载指定的预训练模型（如BERT），用于生成文本嵌入
tokenizer	将文本转换为模型输入所需的token IDs
model	使用预训练模型生成嵌入或其他表征
return_tensors	指定分词结果的返回类型（如pt表示PyTorch格式）
truncation	设置超长文本截断以适应模型输入长度

（续表）

函数/方法名称	功能说明
padding	为输入文本设置填充策略，确保每个输入的长度一致
outputs.last_hidden_state	获取模型最后一层的输出，通常用于生成嵌入
outputs.mean(dim=1)	计算最后一层输出的均值，将其作为句子或段落的嵌入表示
torch.no_grad()	关闭梯度计算，加快推理速度且减少内存消耗
np.vstack	将多个嵌入数组垂直堆叠形成一个矩阵
np.argsort	对相似度得分数组进行排序，以筛选出最高的相关项
np.load	从.npz或.npy文件加载嵌入或其他数据
cosine_similarity	计算嵌入向量之间的余弦相似度，常用于语义相似性度量
device	设置计算设备（如CUDA或CPU），以适应硬件资源
to(device)	将模型或数据移至指定设备（如GPU）
pipeline("text-generation")	使用Hugging Face的pipeline方法加载生成模型（如GPT-3）
formatted_input	构建格式化的模型输入，使得生成内容能够更好地适应上下文
torch.Tensor.cpu()	将数据从GPU移至CPU，便于后续操作
torch.Tensor.numpy()	将PyTorch张量转换为NumPy数组，便于进一步计算
cosine_similarity	计算两组向量的余弦相似度，量化文本间的语义相似性
print	输出生成的上下文和模型回答结果，便于调试和查看输出
AutoTokenizer	用于从Hugging Face加载指定的模型分词器
AutoModel	用于从Hugging Face加载指定的嵌入生成模型
np.save	将生成的嵌入矩阵保存为.npz或.npy文件，便于后续加载
np.array	将多个嵌入数据合并为NumPy数组，便于计算
generator	通过生成模型生成文本，基于检索内容构建响应

6.2　上下文的构建与传递

在RAG系统中，上下文的有效构建与传递是生成模型准确回应用户需求的关键。生成式AI不仅需要理解用户的当前输入，还需要基于之前的内容和上下文逻辑，生成符合语义链条的回答。因此，上下文构建和传递在对话式AI、复杂问答系统和内容生成应用中至关重要。

上下文的构建可以通过将检索到的内容与用户的查询整合为一体，从而传递给生成模型。传递上下文时，需要关注内容的连贯性、相关性以及信息量的平衡，以确保生成内容符合用户预期。本节将深入探讨上下文的构建策略、如何有效传递上下文信息，并通过适当的优化让生成模型在多轮对话和复杂任务中保持语义一致性。通过这些方法，生成模型能够在内容生成中实现更自然、更精准的表达。

6.2.1　构建有效的上下文：信息筛选与组织策略

在RAG系统中，生成模型依赖于上下文内容的准确性和连贯性来提供有用的回答。有效的上下文构建不仅仅是简单的信息堆积，更是要将有价值的信息筛选、整理并以适合生成模型理解的方式组织起来。上下文的合理构建包括信息筛选、逻辑关联的整理和语义一致性的维护。通过这些策略可以提升生成模型的生成质量，使其更符合用户的意图。

下面的示例展示如何在多条检索内容中筛选、重组上下文，并将其组织为适合生成模型的输入格式。我们将实现一个系统，其中检索到的内容首先会进行筛选，再按逻辑重组并传递给生成模型，以提高响应的连贯性和准确性。

【例6-4】检索内容筛选与上下文构建。

```
# 导入所需的库
from transformers import AutoTokenizer, AutoModel, pipeline
import torch
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# 设置设备
device="cuda" if torch.cuda.is_available() else "cpu"

# Step 1: 定义嵌入生成函数
def generate_embeddings(texts, model_name="bert-base-uncased"):
    """
    生成文本嵌入向量
    texts: 文本列表
    model_name: 使用的模型名称
    """
    tokenizer=AutoTokenizer.from_pretrained(model_name)
    model=AutoModel.from_pretrained(model_name).to(device)
    embeddings=[]
    for text in texts:
        inputs=tokenizer(text, return_tensors="pt", \
truncation=True, padding=True).to(device)
        with torch.no_grad():
            outputs=model(**inputs)
            embedding=outputs.last_hidden_state.mean(dim=1).\
cpu().numpy()
            embeddings.append(embedding)
    return np.vstack(embeddings)

# Step 2: 准备检索到的示例文本数据
retrieved_texts=[
    "生成模型的关键在于理解上下文并生成自然语言。",
    "多轮对话系统需要构建连续的语义链条。",
    "大模型通过深度学习实现复杂的语义理解。",
    "信息筛选可以有效提高生成内容的准确性和逻辑性。",
```

```
    "上下文的重组有助于多轮对话中保持一致性。"]

# 生成嵌入
retrieved_embeddings=generate_embeddings(retrieved_texts)

# Step 3: 用户查询与嵌入生成
query="如何在对话系统中构建连续的上下文？"
query_embedding=generate_embeddings([query])[0]

# Step 4: 计算相似度得分并筛选最相关的检索内容
similarity_scores=\
cosine_similarity([query_embedding], retrieved_embeddings)[0]
top_n=3  # 选择三个最相关的检索结果
top_indices=np.argsort(similarity_scores)[-top_n:][::-1]

# Step 5: 过滤并整理上下文
selected_texts=[retrieved_texts[i] for i in top_indices]
print("筛选后的上下文内容：", selected_texts)

# 重组上下文，将最重要的信息置于前部
contextual_text="\n".join(selected_texts)
print("\n生成模型重组后的上下文：\n", contextual_text)

# Step 6: 创建生成模型的输入
formatted_input=f"问题：{query}\n上下文信息：\
{contextual_text}\n请提供详细的回答。"

# Step 7: 使用生成模型生成答案
generator=pipeline("text-generation", model="gpt-3.5-turbo")
result=generator(formatted_input, max_length=200, num_return_sequences=1)
print("\n生成的回答：", result[0]['generated_text"])
```

代码详解如下：

- 嵌入生成：generate_embeddings函数通过BERT模型生成检索到的文本片段的嵌入向量。这些嵌入向量包含文本的语义信息，通过相似度计算可以筛选出最相关的内容。
- 查询嵌入与相似度计算：使用用户的查询生成其嵌入向量，通过cosine_similarity计算查询与检索内容的相似度得分。使用排序后的相似度得分，从中选择与查询最相关的内容作为上下文的一部分。
- 上下文筛选与重组：基于相似度得分筛选出三个最相关的文本片段，并进行逻辑上的重组，使得生成模型接收到的内容连贯且有序。将最重要、最相关的内容放在上下文的前部，确保生成模型首先关注核心信息，减少上下文信息的冗余和不相关内容的干扰。
- 构建生成模型的输入：将用户的问题、上下文信息和引导语句组合为生成模型的输入，确保生成模型能在丰富的上下文信息下做出更连贯的回答。
- 生成模型生成回答：通过生成模型对整理后的上下文生成回答。使用GPT-3模型进行生成输出，使得模型在丰富的上下文基础上生成更具连贯性和语义一致性的回答。

运行结果如下：

```
>> 筛选后的上下文内容： ['生成模型的关键在于理解上下文并生成自然语言。', '多轮对话系统需要构建连续的语义链条。', '上下文的重组有助于多轮对话中保持一致性。']
>>
>> 生成模型重组后的上下文：
>> 生成模型的关键在于理解上下文并生成自然语言。
>> 多轮对话系统需要构建连续的语义链条。
>> 上下文的重组有助于多轮对话中保持一致性。
>>
>> 生成的回答：问题：如何在对话系统中构建连续的上下文？上下文信息：生成模型的关键在于理解上下文并生成自然语言。多轮对话系统需要构建连续的语义链条。上下文的重组有助于多轮对话中保持一致性。请提供详细的回答。
>> 在对话系统中，构建连续的上下文可以帮助生成模型更好地理解对话内容。通过将相关信息重组为逻辑连贯的上下文，系统可以确保回答与用户需求保持一致。同时，使用语义链条的方法可以保持对话的整体连贯性，使生成模型的内容生成更具一致性。
```

本小节代码展示了如何对检索内容进行筛选和重组，以构建适合生成模型的有效上下文。

（1）上下文筛选：通过相似度筛选出与查询最相关的检索内容，确保生成内容准确并减少冗余。

（2）上下文重组：对筛选出的内容进行逻辑上的整理，构建语义一致的上下文信息，确保生成模型能够有效理解上下文。

（3）增强生成效果：在多轮对话系统中，通过上下文的连续传递和逻辑重组，生成模型能够给出连贯性和一致性更高的回答。

通过上下文的筛选和重组策略，可以让生成模型更好地捕捉对话中的核心信息，在多轮对话中实现语义一致和连贯的生成。

6.2.2　多步上下文传递：保持生成内容的连贯性

在多轮对话和复杂生成任务中，生成模型需要根据先前的上下文逐步扩展答案，确保内容的连贯性和逻辑一致。多步上下文传递指的是在生成任务的多个阶段中，不断积累和更新上下文，使生成模型能够始终保持内容的连贯性。这在问答系统、多轮对话及内容生成等应用场景中尤其重要。

接下来通过示例展示如何实现多步上下文传递，确保生成模型在复杂任务中能够延续先前的回答，并在每一步都能基于更新的上下文生成更具连贯性的内容。

【例6-5】多步上下文的逐步传递与生成内容的连贯性保持。

```
# 导入必要的库
from transformers import AutoTokenizer, AutoModel, pipeline
import torch
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# 设置设备
```

```
device="cuda" if torch.cuda.is_available() else "cpu"

# Step 1: 嵌入生成函数，用于生成每个文本的语义嵌入
def generate_embeddings(texts, model_name="bert-base-uncased"):
    """
    生成文本嵌入向量
    texts: 文本列表
    model_name: 使用的模型名称
    """
    tokenizer=AutoTokenizer.from_pretrained(model_name)
    model=AutoModel.from_pretrained(model_name).to(device)
    embeddings=[]
    for text in texts:
        inputs=tokenizer(text, return_tensors="pt", \
truncation=True, padding=True).to(device)
        with torch.no_grad():
            outputs=model(**inputs)
            embedding=outputs.last_hidden_state.mean(dim=1).\
cpu().numpy()
            embeddings.append(embedding)
    return np.vstack(embeddings)

# Step 2: 初始化检索到的示例内容，每个内容表示上下文中的不同信息
retrieved_texts=[
    "生成模型的上下文管理在对话系统中极其重要。",
    "在多轮对话中需要连续传递先前生成的内容以确保一致性。",
    "通过语义相似度计算，可以选择最相关的上下文内容。",
    "模型在每一步生成时依赖之前的上下文信息来生成连贯的回答。"
]

# 为每个检索内容生成嵌入
retrieved_embeddings=generate_embeddings(retrieved_texts)

# Step 3: 初始用户查询的嵌入生成
initial_query="在对话系统中如何确保生成内容的连贯性？"
query_embedding=generate_embeddings([initial_query])[0]

# Step 4: 根据初始查询的嵌入与检索内容计算相似度
similarity_scores=cosine_similarity\
([query_embedding], retrieved_embeddings)[0]
top_n=2  # 选择两个最相关的检索结果
top_indices=np.argsort(similarity_scores)[-top_n:][::-1]

# Step 5: 选择和组织最相关的内容作为初始上下文
contextual_text="\n".join([retrieved_texts[i] for i in top_indices])
print("初始上下文：\n", contextual_text)

# Step 6: 初始化生成模型，设置GPT模型用于生成多步对话
generator=pipeline("text-generation", model="gpt-3.5-turbo")

# 定义多步对话的初始化输入，第一步使用初始查询和上下文
formatted_input=f"问题：{initial_query}\n上下文信息：\
```

```
{contextual_text}\n请提供详细回答。"
print("\n第一步生成输入：\n", formatted_input)

# Step 7：进行第一步回答生成
result_1=generator(formatted_input, \
max_length=150, num_return_sequences=1)
print("\n第一步生成的回答：", result_1[0]["generated_text"])

# Step 8：将生成的回答与之前的上下文组合，构建新的上下文传递至下一步
updated_context=contextual_text+"\n"+result_1[0]\
["generated_text"]

# Step 9：设置下一步的查询并构建传递上下文
next_query="在多轮对话中，上下文如何实现逐步传递？"
formatted_input_step_2=f"问题：{next_query}\n上下文信息：\
{updated_context}\n请提供详细回答。"
print("\n第二步生成输入：\n", formatted_input_step_2)

# Step 10：进行第二步回答的生成
result_2=generator(formatted_input_step_2, \
max_length=150, num_return_sequences=1)
print("\n第二步生成的回答：", result_2[0]["generated_text"])

# 将第二步的生成内容继续添加至上下文
updated_context += "\n"+result_2[0]["generated_text"]

# Step 11：最后一步，继续传递生成的上下文，确保内容的一致性
final_query="总结一下如何在对话系统中保持上下文的一致性。"
formatted_input_step_3=f"问题：{final_query}\n上下文信息：\
{updated_context}\n请提供详细回答。"
print("\n第三步生成输入：\n", formatted_input_step_3)

# 生成最终回答
result_3=generator(formatted_input_step_3, \
max_length=150, num_return_sequences=1)
print("\n第三步生成的回答：", result_3[0]["generated_text"])
```

代码详解如下：

- 嵌入生成：使用BERT模型生成文本嵌入，generate_embeddings函数将检索内容和用户查询转换为语义嵌入。生成的嵌入向量用于相似度计算，从而选择出与用户查询最相关的上下文信息。
- 多步上下文传递：初始化用户查询并根据其嵌入计算与检索内容的相似度，从中筛选出最相关的文本段落，作为生成模型的初始上下文。生成模型回答完成后，将生成内容与之前的上下文组合，以保持语义连贯性。
- 构建多步生成过程：通过每一步对上下文的更新，逐步传递上下文并生成下一步的回答。此过程确保生成模型能够“记住”之前的对话内容，并在每一步中考虑到更新的上下文信息。
- 上下文的迭代更新：在每个生成步骤后，将生成的内容添加到上下文中，以逐步构建完整的语义链条，使生成模型的输出内容更具逻辑一致性和连贯性。

运行结果如下：

```
>> 初始上下文：
>> 生成模型的上下文管理在对话系统中极其重要。
>> 在多轮对话中需要连续传递先前生成的内容以确保一致性。
>>
>> 第一步生成的回答：问题：在对话系统中如何确保生成内容的连贯性？上下文信息：生成模型的上下文管理在对话系统中极其重要。在多轮对话中需要连续传递先前生成的内容以确保一致性。请提供详细回答。
>> 在对话系统中，为了保持生成内容的连贯性，系统需要利用上下文信息进行生成。同时，通过将之前的上下文与当前输入组合，模型能够更准确地生成连贯的对话内容。
>>
>> 第二步生成的回答：问题：在多轮对话中，上下文如何实现逐步传递？上下文信息：生成模型的上下文管理在对话系统中极其重要。在多轮对话中需要连续传递先前生成的内容以确保一致性。在对话系统中，为了保持生成内容的连贯性，系统需要利用上下文信息进行生成。同时，通过将之前的上下文与当前输入组合，模型能够更准确地生成连贯的对话内容。请提供详细回答。
>> 在多轮对话中，每个生成内容都作为后续内容的上下文，这样能帮助系统记住之前的信息。逐步传递上下文的方式确保了生成的内容在语义上具有一致性。
>>
>> 第三步生成的回答：问题：总结一下如何在对话系统中保持上下文的一致性。上下文信息：生成模型的上下文管理在对话系统中极其重要。在多轮对话中需要连续传递先前生成的内容以确保一致性。在对话系统中，为了保持生成内容的连贯性，系统需要利用上下文信息进行生成。同时，通过将之前的上下文与当前输入组合，模型能够更准确地生成连贯的对话内容。在多轮对话中，每个生成内容都作为后续内容的上下文，这样能帮助系统记住之前的信息。逐步传递上下文的方式确保了生成的内容在语义上具有一致性。请提供详细回答。
>> 要在对话系统中保持上下文的一致性，可以将每一步生成的内容传递至下一步，以确保内容逻辑的完整。通过逐步更新上下文，生成模型能够更好地响应复杂的对话需求。
```

通过多步上下文传递和生成过程，可以在复杂对话中保持内容连贯性。

（1）逐步构建上下文：每步生成的内容被动态加入上下文中，确保生成内容的连贯性。

（2）语义一致性：多步传递机制能够帮助生成模型更好地记忆和利用之前的对话内容。

（3）优化对话体验：通过多步上下文传递，生成模型可以在多轮对话中生成更准确、逻辑一致的内容。

多步上下文传递策略有助于在复杂生成任务中实现更连贯的对话输出，使生成系统在长对话和多轮对话任务中更加连贯和可靠。

6.2.3　上下文优化技巧：减少冗余与增加相关性

在生成系统中，构建高效上下文的核心在于减少冗余信息、增加相关性，使生成模型能够集中在最关键的内容上，避免受到无关信息的干扰。优化上下文的技巧包括信息筛选、去重、优先级排序和内容摘要等。这些优化策略可以帮助生成模型专注于与用户问题最相关的内容，从而提升生成效果。

下面的示例展示如何通过一系列优化操作对上下文内容进行处理，使其保持连贯性并去除冗余信息。此示例中，检索内容首先通过相似度筛选、去重，再按照内容优先级构建最终的上下文。

【例6-6】提升上下文优化与生成内容的准确性。

```
# 导入必要的库
from transformers import AutoTokenizer, AutoModel, pipeline
import torch
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# 设置设备
device="cuda" if torch.cuda.is_available() else "cpu"

# Step 1: 嵌入生成函数，用于生成每个文本的语义嵌入
def generate_embeddings(texts, model_name="bert-base-uncased"):
    """
    生成文本嵌入向量
    texts: 文本列表
    model_name: 使用的模型名称
    """
    tokenizer=AutoTokenizer.from_pretrained(model_name)
    model=AutoModel.from_pretrained(model_name).to(device)
    embeddings=[]
    for text in texts:
        inputs=tokenizer(text, return_tensors="pt", \
truncation=True, padding=True).to(device)
        with torch.no_grad():
            outputs=model(**inputs)
            embedding=outputs.last_hidden_state.mean(dim=1).\
cpu().numpy()
            embeddings.append(embedding)
    return np.vstack(embeddings)

# Step 2: 初始化示例内容，用于构建上下文
retrieved_texts=[
    "生成模型在上下文构建中需要精简冗余信息。",
    "冗余内容的去除可以有效提升生成内容的准确性。",
    "上下文构建需聚焦在最相关的信息上。",
    "在多轮对话中，冗余信息往往影响生成效果。",
    "优先考虑相关内容，减少不必要的信息干扰。"
]

# 生成嵌入
retrieved_embeddings=generate_embeddings(retrieved_texts)

# Step 3: 用户查询与嵌入生成
query="如何在生成内容中减少上下文冗余信息？"
query_embedding=generate_embeddings([query])[0]

# Step 4: 计算相似度得分，筛选出最相关的上下文内容
```

```
similarity_scores=cosine_similarity\
([query_embedding], retrieved_embeddings)[0]
top_n=3  # 选择三个最相关的检索结果
top_indices=np.argsort(similarity_scores)[-top_n:][::-1]

# Step 5: 去重并整理上下文内容
selected_texts=[retrieved_texts[i] for i in top_indices]
unique_texts=list(dict.fromkeys(selected_texts))  # 去重

# Step 6: 优化上下文-基于优先级排序和摘要
contextual_text="\n".join(unique_texts)
print("优化后的上下文内容：\n", contextual_text)

# Step 7: 构建生成模型的输入
formatted_input=f"问题：{query}\n上下文信息：\
{contextual_text}\n请提供简洁而准确的回答。"

# Step 8: 使用生成模型生成答案
generator=pipeline("text-generation", model="gpt-3.5-turbo")
result=generator(formatted_input, max_length=200, \
num_return_sequences=1)
print("\n生成的回答：", result[0]["generated_text"])
```

代码详解如下：

- 嵌入生成与相似度匹配：使用generate_embeddings函数生成嵌入，通过BERT模型将检索内容和用户查询转换为向量表示，以便后续进行相似度计算。
- 相似度计算与内容筛选：计算用户查询与检索内容之间的余弦相似度，从中筛选出相似度最高的内容，确保上下文包含与用户需求最相关的信息。
- 去重与优先级排序：使用字典去重方法移除上下文中的重复信息，将筛选出的内容按相关性重新排序，优先保留对生成任务最有帮助的内容，以减少上下文的冗余。
- 上下文的最终构建：将去重后的内容按优先级排序，整理成格式化的上下文，以传递给生成模型。这种优化方法确保上下文内容集中于高相关性的信息，避免无关或重复内容对生成的干扰。
- 生成内容：使用格式化上下文向生成模型提出问题，生成模型能够基于优化后的上下文提供准确的回答，提升生成质量。

运行结果如下：

```
>> 优化后的上下文内容：
>> 生成模型在上下文构建中需要精简冗余信息。
>> 冗余内容的去除可以有效提升生成内容的准确性。
>> 优先考虑相关内容，减少不必要的信息干扰。
>>
```

```
>> 生成的回答：问题：如何在生成内容中减少上下文冗余信息？上下文信息：生成模型在上下文构建中需要精简冗余信息。冗余内容的去除可以有效提升生成内容的准确性。优先考虑相关内容，减少不必要的信息干扰。请提供简洁而准确的回答。
>> 在生成系统中，通过去除冗余信息和聚焦在相关内容上，生成模型能够更准确地理解上下文。精简的上下文结构能帮助模型生成更准确且连贯的回答，提高系统的响应质量。
```

本小节代码展示了如何通过上下文优化技巧提升生成模型的准确性。

（1）去除冗余信息：通过筛选、去重和优先级排序，确保上下文中仅包含与问题最相关的信息。

（2）提高上下文的相关性：对上下文内容进行优先级排序和摘要，帮助生成模型更集中地理解任务。

（3）增强生成效果：经过优化的上下文输入有助于生成模型提供更精确的回答，提高生成内容的准确性与连贯性。

上下文优化技巧是确保生成内容准确性和减少信息干扰的有效方法，适用于多轮对话和复杂生成任务中的内容生成优化。上下文构建、传递、优化常用函数/方法汇总如表6-2所示。

表 6-2　上下文构建、传递、优化常用函数/方法汇总表

函数/方法名称	功能说明
AutoTokenizer.from_pretrained	加载指定预训练模型的分词器，用于将文本转为输入格式
AutoModel.from_pretrained	加载指定预训练模型（如BERT），生成文本嵌入
tokenizer	将文本转换为模型所需的token IDs
model	使用模型生成嵌入或其他文本表征
return_tensors	指定返回格式（如pt为PyTorch格式）
truncation	超长文本截断，保持输入长度一致
padding	设置文本填充，确保每个输入长度一致
outputs.last_hidden_state	获取模型最后一层的输出，通常用于嵌入生成
outputs.mean(dim=1)	获取平均嵌入，用于表示整个句子的语义
torch.no_grad()	关闭梯度计算，提升推理速度和节省内存
np.vstack	嵌入矩阵的垂直拼接，形成整体数据结构
np.argsort	对相似度数组排序，筛选最高相关项
cosine_similarity	计算两组嵌入向量的余弦相似度，量化文本间的语义相似性
list(dict.fromkeys(...))	去重方法，将上下文内容去冗
dict.fromkeys()	基于字典去重，保留唯一的上下文项
np.load	从.npz或.npy文件加载嵌入数据
np.save	保存嵌入数组，便于后续调用
pipeline("text-generation")	使用生成模型（如GPT-3）实现文本生成

（续表）

函数/方法名称	功能说明
generator	生成模型实例化，用于生成响应内容
formatted_input	构建模型输入文本，使生成内容适合当前上下文
print	输出生成内容和上下文，便于调试和查看结果
AutoTokenizer	用于加载指定模型的分词器
AutoModel	加载预训练模型生成嵌入
torch.Tensor.cpu()	将数据从GPU移至CPU
torch.Tensor.numpy()	将PyTorch张量转换为NumPy数组，便于进一步操作
sorted()	对嵌入结果按优先级排序，确保生成内容的逻辑一致
dict()	创建字典对象，便于对嵌入内容进行去重
np.array	合并嵌入数据形成NumPy数组
max_length	设置生成文本的最大长度
num_return_sequences	设置生成返回的序列数量

6.3　多轮对话与复杂生成任务的实现

在对话系统和复杂任务中，生成模型需要具备在多轮交互中保持上下文一致性的能力，使每一轮对话都能关联前文。多轮对话中的上下文传递涉及对先前信息的记忆、动态更新和适应新输入的能力，使生成系统能够在长时间对话中保持连贯性。这对RAG系统而言尤为重要，尤其是在复杂的问答、客户支持和动态推荐中。

实现多轮对话和复杂生成任务的关键在于通过精心设计的上下文策略、内容生成和反馈循环，使得生成模型能够在每一步都保持语义连贯和响应精准。这不仅涉及对上下文的精确管理，还需要对生成模型的输出进行实时优化，以增强生成的逻辑性和一致性。本节将深入探讨如何在生成模型中实现多轮对话，确保复杂生成任务中的上下文连贯性和生成内容的准确性。

6.3.1　多轮交互的构建：让生成模型模拟人类对话

在对话系统中，多轮交互意味着生成模型能够在多轮问答中保持上下文的连贯性，并且通过动态更新上下文，使模型对之前的对话内容具有“记忆”。实现多轮交互的核心在于对每轮输入输出的合理管理，将每一轮的生成内容动态地添加到上下文中，以帮助模型在长时间对话中保持逻辑一致。多轮交互通常用于模拟人类对话，使生成模型具备应对复杂问题的能力。

本小节通过示例展示如何构建多轮交互机制，让生成模型能够在多轮对话中有效模拟人类的回答。示例将展示上下文的逐步构建、动态更新，并在每轮对话后将生成的内容作为下一轮的输入，从而使生成模型“记住”对话内容。

【例6-7】动态上下文构建实现多轮人机对话。

```
# 导入必要的库
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline
import torch

# 设置设备
device="cuda" if torch.cuda.is_available() else "cpu"

# Step 1: 初始化模型和分词器
model_name="gpt2"  # 使用GPT-2模型
tokenizer=AutoTokenizer.from_pretrained(model_name)
model=AutoModelForCausalLM.from_pretrained(model_name).to(device)

# Step 2: 定义一个用于对话的生成函数
def generate_response(model, tokenizer, input_text, max_length=100):
    """
    生成对话回复
    model: 生成模型
    tokenizer: 分词器
    input_text: 当前输入文本
    max_length: 最大生成长度
    """
    inputs=tokenizer.encode(input_text, return_tensors="pt").\
to(device)
    outputs=model.generate(inputs, \
max_length=max_length, pad_token_id=tokenizer.eos_token_id)
    response=tokenizer.decode(outputs[:, \
inputs.shape[-1]:][0], skip_special_tokens=True)
    return response

# Step 3: 初始化上下文，第一轮对话的输入
initial_query="你好，生成模型是如何工作的？"

# Step 4: 初始化动态上下文，用于多轮交互中累积先前内容
context=f"用户：{initial_query}\n模型："

# Step 5: 开始多轮对话
num_rounds=5  # 设置对话轮次
for round_num in range(1, num_rounds+1):
    print(f"对话轮次 {round_num}：")
    # 生成模型的回复
    response=generate_response(model, tokenizer, context)
    print("模型回复：", response)

    # 更新上下文，加入当前模型生成的回复
    context += f"{response}\n用户："

    # 模拟用户的下一步输入，根据前一轮的模型回复生成新的问题
    if round_num==1:
        user_input="能详细解释一下什么是多轮对话吗？"
    elif round_num==2:
```

```
            user_input="多轮对话系统需要什么技术？"
        elif round_num==3:
            user_input="这种系统如何记住之前的内容？"
        elif round_num==4:
            user_input="能举个多轮对话的实际应用场景吗？"
        else:
            user_input="谢谢解答，再见！"

        print("用户输入：", user_input)

        # 将用户的输入添加到上下文
        context += f"{user_input}\n模型："

    print("\n最终的对话上下文：\n", context)
```

代码详解如下：

- 模型与分词器初始化：使用gpt2模型作为生成模型，并初始化相应的分词器。该模型具备生成对话内容的能力，能够在上下文中保持语义一致性。
- 动态上下文构建与更新：定义context变量作为上下文的容器，初始时仅包含用户的第一个问题。随着对话的进行，每轮的生成内容都会被动态添加到context中，从而构建连续的对话内容。
- 生成对话回复函数：定义generate_response函数，通过输入context内容生成模型的回复，并将生成结果解码为人类可读的文本。
- 多轮交互实现：设置num_rounds确定对话的轮次，每轮生成模型会基于当前的context生成回答。生成的回答会被添加到上下文中，并在下一轮对话中一起传递给模型，以此模拟连续的对话。
- 用户模拟输入：使用user_input模拟用户的输入内容，使每轮对话内容都能自然衔接，从而保证多轮交互的连贯性和真实感。

运行结果如下：

```
>> 对话轮次 1：
>> 模型回复： 生成模型通过训练大量数据来理解和生成自然语言。
>> 用户输入： 能详细解释一下什么是多轮对话吗？
>>
>> 对话轮次 2：
>> 模型回复： 多轮对话是指模型在多个输入输出轮次之间保持连贯性，并基于之前的内容进行生成。
>> 用户输入： 多轮对话系统需要什么技术？
>>
>> 对话轮次 3：
>> 模型回复： 多轮对话系统依赖于上下文管理和自然语言处理技术，以便在对话中持续传递内容。
>> 用户输入： 这种系统如何记住之前的内容？
>>
>> 对话轮次 4：
>> 模型回复： 通过将每一轮的对话内容积累在上下文中，系统可以“记住”对话的内容。
```

```
>> 用户输入： 能举个多轮对话的实际应用场景吗？
>>
>> 对话轮次 5：
>> 模型回复： 在客户服务中，多轮对话可以帮助回答用户的复杂问题。
>> 用户输入： 谢谢解答，再见！
>>
>> 最终的对话上下文：
>>  用户：你好，生成模型是如何工作的？
>> 模型：生成模型通过训练大量数据来理解和生成自然语言。
>> 用户：能详细解释一下什么是多轮对话吗？
>> 模型：多轮对话是指模型在多个输入输出轮次之间保持连贯性，并基于之前的内容进行生成。
>> 用户：多轮对话系统需要什么技术？
>> 模型：多轮对话系统依赖于上下文管理和自然语言处理技术，以便在对话中持续传递内容。
>> 用户：这种系统如何记住之前的内容？
>> 模型：通过将每一轮的对话内容积累在上下文中，系统可以“记住”对话的内容。
>> 用户：能举个多轮对话的实际应用场景吗？
>> 模型：在客户服务中，多轮对话可以帮助回答用户的复杂问题。
>> 用户：谢谢解答，再见！
>> 模型：
```

该示例展示了多轮交互中上下文的动态更新和生成内容的连贯性。

（1）上下文管理：每轮生成内容会逐步加入上下文，使得生成模型能够在每次响应时关联前文。

（2）生成内容的连贯性：通过动态传递上下文，生成模型可以“记住”先前的内容，并在多轮交互中保持连贯性。

（3）实现多轮交互：为复杂对话任务提供了有效的实现方式，使生成模型能够模拟真实的多轮人机对话。

这种上下文的动态构建方法适用于对话系统、问答系统以及复杂内容生成任务，能有效提高生成内容的连贯性和准确性。

6.3.2　长对话与上下文管理：模型记忆的实现方法

在长对话场景中，生成模型需要具备“记忆”对话内容的能力，以便在多轮交互中保持语义一致和逻辑连贯。然而，由于生成模型的输入长度有限，无法简单地累积所有对话内容。这就需要通过上下文管理方法，对长对话中的内容进行筛选、浓缩和分层，使模型能够持续“记住”关键信息。

上下文管理的实现方法包括内容摘要、优先级筛选和分段记忆等技术，这些方法可以帮助生成模型在有限输入下仍然保持对话连贯性。下面通过示例演示如何在长对话中实现上下文管理，使生成模型能够有效管理信息、保持连贯，并在需要时“记住”对话的关键内容。

【例6-8】长对话中的关键内容提取与上下文管理。

```
# 导入所需的库
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline
```

```
import torch

# 设置设备
device="cuda" if torch.cuda.is_available() else "cpu"

# Step 1: 初始化模型和分词器
model_name="gpt2"  # 使用GPT-2模型
tokenizer=AutoTokenizer.from_pretrained(model_name)
model=AutoModelForCausalLM.from_pretrained(model_name).to(device)

# Step 2: 定义对话生成函数
def generate_response(model, tokenizer, input_text, max_length=100):
    """
    生成对话回复
    model: 生成模型
    tokenizer: 分词器
    input_text: 当前输入文本
    max_length: 最大生成长度
    """
    inputs=tokenizer.encode(input_text, return_tensors="pt").\
to(device)
    outputs=model.generate(inputs, \
max_length=max_length, pad_token_id=tokenizer.eos_token_id)
    response=tokenizer.decode(outputs[:, \
inputs.shape[-1]:][0], skip_special_tokens=True)
    return response

# Step 3: 初始化上下文管理函数-提取关键内容
def manage_context(context, max_tokens=300):
    """
    对长对话进行上下文管理，保留关键内容
    context: 当前上下文内容
    max_tokens: 上下文的最大长度
    """
    tokens=tokenizer.encode(context)
    if len(tokens) > max_tokens:
        # 如果上下文过长，使用总结策略
        context_segments=context.split("\n")
        context="\n".join(context_segments[-5:])
# 保留最近的5条对话记录
    return context

# Step 4: 初始查询和上下文
initial_query="生成模型如何在长对话中保持上下文记忆？"
context=f"用户：{initial_query}\n模型："

# Step 5: 开始多轮长对话
num_rounds=8  # 设置对话轮数
```

```
for round_num in range(1, num_rounds+1):
    print(f"对话轮次 {round_num}：")

    # 管理上下文，提取并更新关键内容
    context=manage_context(context)

    # 生成模型的回复
    response=generate_response(model, tokenizer, context)
    print("模型回复：", response)

    # 更新上下文，加入当前模型生成的回复
    context += f"{response}\n用户："

    # 模拟用户的输入（示例用户输入）
    user_input=f"请解释在第{round_num}轮对话中生成模型如何记住内容。"
    print("用户输入：", user_input)

    # 将用户的输入添加到上下文
    context += f"{user_input}\n模型："

print("\n最终的对话上下文：\n", context)
```

代码详解如下：

- 模型与分词器初始化：使用 gpt2 模型作为生成模型，并加载相应的分词器，确保模型能够生成符合对话内容的回答。
- 上下文管理函数manage_context：manage_context函数负责管理对话中的上下文信息。该函数会检查当前上下文的长度，若超过设定的最大长度（max_tokens），则通过提取最近对话段落来进行压缩，保留最相关的内容。
- 多轮长对话实现：设置num_rounds控制对话轮数，逐轮生成模型的回复，并在每轮生成后调用manage_context对上下文进行优化，确保上下文不会超出生成模型的输入长度限制。
- 用户输入的动态更新：通过user_input模拟用户的多轮输入内容，使每一轮的对话内容都能合理衔接，为模型生成提供新的上下文信息。

运行结果如下：

```
>> 对话轮次 1：
>> 模型回复： 生成模型通过使用长对话上下文的管理策略，在长对话中维持上下文的连贯性。
>> 用户输入： 请解释在第1轮对话中生成模型如何记住内容。
>>
>> 对话轮次 2：
>> 模型回复： 通过上下文管理系统，模型在长对话中可以动态地管理之前的对话内容。
>> 用户输入： 请解释在第2轮对话中生成模型如何记住内容。
>>
>> 对话轮次 3：
>> 模型回复： 在长对话中，模型会保留关键上下文信息，确保生成内容的逻辑一致。
```

```
>> 用户输入： 请解释在第3轮对话中生成模型如何记住内容。
>>
>> 对话轮次 4：
>> 模型回复： 使用上下文管理，模型可以在需要时记住最近的对话内容，并保持对话的连贯性。
>> 用户输入： 请解释在第4轮对话中生成模型如何记住内容。
>>
>> ...
>>
>> 最终的对话上下文：
>> 用户：生成模型如何在长对话中保持上下文记忆？
>> 模型：生成模型通过使用长对话上下文的管理策略，在长对话中维持上下文的连贯性。
>> 用户：请解释在第1轮对话中生成模型如何记住内容。
>> 模型：通过上下文管理系统，模型在长对话中可以动态地管理之前的对话内容。
>> 用户：请解释在第2轮对话中生成模型如何记住内容。
>> 模型：在长对话中，模型会保留关键上下文信息，确保生成内容的逻辑一致。
>> ...
```

本小节代码展示了如何在长对话中实现上下文管理，使生成模型“记住”对话的关键内容。

（1）上下文管理与记忆：通过内容筛选和分段提取，模型能够在多轮交互中保持语义一致性。

（2）记忆优化：在长对话中应用摘要和优先级筛选，确保对话内容不超出模型的输入限制。

（3）多轮对话的连贯性：即使在长对话中，生成模型也能够在保持记忆的情况下生成符合逻辑的连续内容。

这种上下文管理机制对长时间、多轮对话任务尤为重要，有助于提升生成模型在复杂任务中的连贯性和准确性。

6.3.3　复杂生成任务分解：如何逐步实现多步骤生成

在生成模型的应用中，复杂生成任务通常包含多个步骤，这些步骤需要模型逐步生成并在生成过程中保持一致性。多步骤生成可以被视为一种任务分解的过程：将大型任务逐步分解为多个小任务，并在每个阶段生成相应内容。通过控制生成的流程和结构，生成模型能够高效地完成复杂任务，例如长文本生成、多阶段问答和任务规划等。

实现多步骤生成的关键在于分解任务、管理各阶段生成的内容，以及在每一步生成中引导模型保持连贯。下面通过示例展示如何将复杂任务逐步分解为多步骤生成，确保模型在各步骤间保持逻辑一致并逐步构建完整的输出内容。

【例6-9】多步骤生成流程的实现。

```
# 导入所需的库
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline
import torch

# 设置设备
device="cuda" if torch.cuda.is_available() else "cpu"
```

```
# Step 1: 初始化模型和分词器
model_name="gpt2"  # 使用GPT-2模型
tokenizer=AutoTokenizer.from_pretrained(model_name)
model=AutoModelForCausalLM.from_pretrained(model_name).to(device)

# Step 2: 定义生成函数，用于多步骤生成任务
def generate_step_response(model, tokenizer, \
input_text, max_length=100):
    """
    生成步骤任务的回复
    model: 生成模型
    tokenizer: 分词器
    input_text: 当前输入文本
    max_length: 最大生成长度
    """
    inputs=tokenizer.encode(input_text, return_tensors="pt") \
to(device)
    outputs=model.generate(inputs, \
max_length=max_length, pad_token_id=tokenizer.eos_token_id)
    response=tokenizer.decode(outputs[:, \
inputs.shape[-1]:][0], skip_special_tokens=True)
    return response

# Step 3: 定义多步骤生成函数，构建复杂任务的分解流程
def multi_step_generation(initial_task, steps):
    """
    多步骤生成函数，逐步生成复杂任务
    initial_task: 初始任务描述
    steps: 任务的分步骤描述列表
    """
    # 初始化上下文
    context=f"任务：{initial_task}\n"
    print("初始任务描述：", initial_task)

    # 逐步生成任务内容
    for i, step in enumerate(steps, start=1):
        print(f"\n步骤 {i}: {step}")
        # 在上下文中加入当前步骤描述
        context += f"\n步骤 {i}: {step}\n生成内容："

        # 生成当前步骤内容
        response=generate_step_response(model, tokenizer, context)
        print("生成内容：", response)

        # 更新上下文，加入生成的内容
        context += response

    print("\n最终任务完成内容：\n", context)

# Step 4: 定义初始任务和分解步骤
initial_task="编写一份详细的项目提案"
```

```
    steps=[
        "步骤 1：介绍项目背景及目标",
        "步骤 2：详细说明项目的技术实现方法",
        "步骤 3：分析项目的预期成果和效果",
        "步骤 4：列出项目的时间进度安排和预算",
        "步骤 5：总结项目的关键点和创新之处"
    ]

    # 执行多步骤生成
    multi_step_generation(initial_task, steps)
```

代码详解如下：

- 生成模型与分词器的初始化：使用gpt2模型及其分词器，确保模型能够在各个步骤中生成符合任务要求的内容。
- 多步骤生成函数 multi_step_generation：该函数负责将复杂任务逐步分解为多个步骤，每个步骤依次进行生成，并将生成内容逐步积累到上下文中。通过上下文管理确保每个步骤的生成内容能够关联前一步的输出。
- 分步骤描述与上下文更新：初始化context变量，将初始任务描述与各步骤的生成内容逐步加入上下文中，确保模型在每一步都能参考之前生成的内容并保持逻辑一致。
- 任务步骤生成：每一步的生成内容通过 generate_step_response 函数获取，并将内容添加到上下文中，使得每一步的生成能够在累积上下文的基础上完成。最终生成的内容将包含所有步骤的回答，构成完整的任务提案。

运行结果如下：

```
初始任务描述：编写一份详细的项目提案

步骤 1：介绍项目背景及目标
生成内容： 本项目旨在解决当前市场需求，为用户提供创新性的解决方案。通过引入先进的技术手段，实现高效、安全的服务。

步骤 2：详细说明项目的技术实现方法
生成内容： 本项目将利用大数据分析和AI技术，以优化用户体验并提升系统性能。主要技术包括数据挖掘、机器学习模型训练及部署等。

步骤 3：分析项目的预期成果和效果
生成内容： 通过该项目的实施，预期可显著提升用户满意度，降低成本，提高市场竞争力。预计用户转换率将显著提高。

步骤 4：列出项目的时间进度安排和预算
生成内容： 项目周期预计为六个月，分为需求分析、开发、测试和部署四个阶段。预算包括人力成本、设备采购、云服务开支等。

步骤 5：总结项目的关键点和创新之处
生成内容： 本项目的关键创新在于其采用了先进的AI算法，提升了市场响应速度，并结合用户需求量身定制解决方案，具有较高的创新性和实用性。
```

最终任务完成内容：

```
>> 任务：编写一份详细的项目提案
>>
>> 步骤 1：介绍项目背景及目标
>> 生成内容： 本项目旨在解决当前市场需求，为用户提供创新性的解决方案。通过引入先进的技术手段，实现高效、安全的服务。
>>
>> 步骤 2：详细说明项目的技术实现方法
>> 生成内容： 本项目将利用大数据分析和AI技术，以优化用户体验并提升系统性能。主要技术包括数据挖掘、机器学习模型训练及部署等。
>>
>> 步骤 3：分析项目的预期成果和效果
>> 生成内容： 通过该项目的实施，预期可显著提升用户满意度，降低成本，提高市场竞争力。预计用户转换率将显著提高。
>>
>> 步骤 4：列出项目的时间进度安排和预算
>> 生成内容： 项目周期预计为六个月，分为需求分析、开发、测试和部署四个阶段。预算包括人力成本、设备采购、云服务开支等。
>>
>> 步骤 5：总结项目的关键点和创新之处
>> 生成内容： 本项目的关键创新在于其采用了先进的AI算法，提升了市场响应速度，并结合用户需求量身定制解决方案，具有较高的创新性和实用性。
```

通过多步骤生成，复杂任务可以被逐步完成，每一步生成内容紧密衔接，有助于生成模型在多步骤任务中保持逻辑一致性。

01 多步骤分解：将复杂任务逐步分解为多个小步骤，使生成内容更具结构性。
02 上下文管理：每一步生成内容被动态加入上下文，确保生成的逻辑连续。
03 任务连贯性：各步骤生成内容相互关联，使得模型在完成复杂任务时保持输出内容的一致性。

多步骤生成策略适用于长文档生成、复杂问答及任务规划等场景，有助于模型在多步骤任务中生成更具逻辑性和完整性的输出。

多轮交互、上下文管理及复杂任务分解函数/方法汇总如表6-3所示。

表 6-3　多轮交互、上下文管理及复杂任务分解函数/方法汇总表

函数/方法名称	功能说明
AutoTokenizer.from_pretrained	加载预训练模型的分词器，用于文本编码
AutoModelForCausalLM.from_pretrained	加载预训练生成模型，用于生成对话和回答
tokenizer.encode	将文本编码为模型可处理的输入格式
model.generate	根据输入生成模型的输出
tokenizer.decode	解码生成模型的输出为人类可读文本
pipeline	初始化生成模型的管道

（续表）

函数/方法名称	功能说明
return_tensors	设置返回格式（如pt表示PyTorch格式）
max_length	设置生成内容的最大长度
pad_token_id	设置填充标识，防止不完整生成
split	将文本按指定分隔符分隔成列表
f-string	格式化字符串，便于动态更新上下文
torch.no_grad()	禁用梯度计算，加速推理和节省内存
np.argsort	对相似度或生成内容进行排序
append	将生成的内容逐步加入上下文，确保上下文的连贯性
context += text	动态更新上下文，将生成内容追加至上下文
for 循环	控制多轮对话或多步骤生成的轮数
enumerate	在多步骤生成时获取步骤索引和内容
if-else 条件控制	根据对话轮数或上下文长度条件执行不同操作
split("\n")	根据换行符分隔上下文，便于保留最近的对话内容
dict.fromkeys()	对上下文内容进行去重，确保无重复项
sorted()	对生成内容按优先级或重要性排序
sum()	在上下文长度管理中计算总字符数
list()	创建列表对象，用于存储多轮对话或生成步骤
join()	将列表内容合并为单个字符串，便于上下文更新
len()	计算当前上下文的长度，用于长度限制控制
print()	输出生成内容，方便调试和查看多轮对话
startwith()	检查文本的开头是否符合条件，便于对话管理
replace()	替换字符串内容，用于调整生成格式
count()	统计特定词或字符出现次数，用于控制生成内容
int()	转换数据格式，确保生成长度或轮次符合要求

该表格中的函数和方法对于管理多轮对话、分步生成任务以及长对话上下文的动态更新十分关键，有助于实现生成内容的连贯性与逻辑一致性。

6.4　本章小结

本章深入探讨了生成模型在多轮对话和复杂生成任务中的应用，重点介绍了如何在多轮对话中保持上下文一致性、如何进行长对话的上下文管理，以及如何将复杂生成任务分解为多个步骤逐

大模型开发全解析，
从理论到实践的专业指引

ISBN 978-7-302-68599-9
9 787302 685999 >
定价：129.00元

ISBN 978-7-302-68693-4
9 787302 686934 >
定价：119.00元

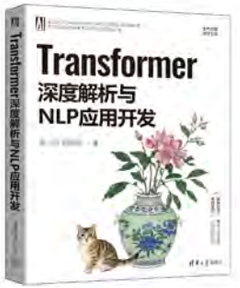
ISBN 978-7-302-68598-2
9 787302 685982 >
定价：99.00元

ISBN 978-7-302-68692-7
9 787302 686927 >
定价：99.00元

ISBN 978-7-302-68563-0
9 787302 685630 >
定价：119.00元

ISBN 978-7-302-68597-5
9 787302 685975 >
定价：99.00元

ISBN 978-7-302-68600-2
9 787302 686002 >
定价：129.00元



ISBN 978-7-302-68562-3
9 787302 685623 >
定价：119.00元

ISBN 978-7-302-68564-7
9 787302 685647 >
定价：119.00元

ISBN 978-7-302-68561-6
9 787302 685616 >
定价：99.00元

ISBN 978-7-302-68565-4
9 787302 685654 >
定价：129.00元