# 大模型应用开发极简入门：基于**DeepSeek**

---

作者： 基于《大模型应用开发极简入门：基于GPT-4和ChatGPT》改编
版本： DeepSeek版本
日期： 2024年

---

## 前言

欢迎阅读《大模型应用开发极简入门：基于DeepSeek》！

本书是基于《大模型应用开发极简入门：基于GPT-4和ChatGPT》改编的DeepSeek版本。由于网络原因无法直接使用OpenAI API，本书将指导您使用DeepSeek API完成相同的开发任务。

本书特点：

- 通俗易懂：用生活化的比喻解释技术概念，让0基础用户也能理解
- 实例丰富：每个章节都包含完整的可执行代码示例
- 循序渐进：从基础概念到实际应用，逐步深入
- 即学即用：所有代码都可以在本地运行，立即看到效果

适合读者：

- 对AI应用开发感兴趣的初学者
- 想学习DeepSeek API的开发者
- 需要将OpenAI应用迁移到DeepSeek的开发者
- 希望了解大模型应用开发的任何人

如何使用本书：

1. 按照章节顺序阅读，每章都有详细的概念解释
2. 运行每章的示例代码，边学边实践
3. 完成每章的练习，巩固所学知识
4. 参考代码示例，构建自己的应用

致谢：

感谢原书《大模型应用开发极简入门：基于GPT-4和ChatGPT》的作者，为本书提供了优秀的框架和思路。

---

## 目录

## 第2章 深入了解DeepSeek的API：打开AI世界的大门

- 2.1 基本概念：API是什么?
- 2.2 DeepSeek API提供的可用模型：选择你的助手
- 2.3 在DeepSeek Playground中使用模型：先试试再买
- 2.4 开始使用DeepSeek Python库：你的第一个AI程序
- 2.5 使用DeepSeek进行文本生成：让AI为你"打字"
- 2.6 使用其他文本补全模型：多轮对话的艺术
- 2.7 考虑因素：使用API的注意事项
- 2.8 其他DeepSeek API和功能：更多可能性
- 2.9 小结：掌握API的使用

## 第3章 使用DeepSeek构建应用程序：从想法到现实

- 3.1 应用程序开发概述：把你的想法变成现实
- 3.2 软件架构设计原则：建造房子的智慧
- 3.3 LLM驱动型应用程序的漏洞：安全防护的重要性
- 3.4 示例项目
    - 3.4.1 简单聊天机器人：你的第一个AI朋友
    - 3.4.2 Web聊天应用（Streamlit）
    - 3.4.3 代码助手
    - 3.4.4 文档总结工具
- 3.5 小结

## 第4章 DeepSeek的高级技巧：让AI更懂你

- 4.1 提示工程：问问题的艺术
    - 4.1.1 提示工程的基本原则
    - 4.1.2 常用提示技巧
- 4.2 微调：让AI成为你的专业助手
    - 4.2.1 微调流程：从数据到模型
    - 4.2.2 微调最佳实践：让培训更有效
- 4.3 小结：掌握高级技巧

## 第5章 使用LangChain框架和插件增强LLM的功能：从手工到自动化

- 5.1 LangChain概述：你的AI应用工具箱
- 5.2 使用LangChain构建应用程序：组装你的第一个应用
    - 5.2.1 基本使用：打开工具箱
    - 5.2.2 链式调用：像流水线一样工作
    - 5.2.3 记忆管理：给AI装一个"记忆系统"
    - 5.2.4 RAG应用：给AI装一个"知识库"
- 5.3 LangChain插件
- 5.4 小结

---

# 第1章 初识DeepSeek：当AI遇见中文世界

## 1.1 LLM概述：从"会说话的机器"说起

### 1.1.1 探索语言模型和NLP的基础

想象一下，你走进一家24小时营业的图书馆，里面有一位博学的图书管理员。这位管理员不仅读过图书馆里的每一本书，还能：

- 当你问"什么是量子物理？"时，他能用通俗易懂的语言解释
- 当你需要写一份工作报告时，他能帮你组织思路、润色文字
- 当你遇到编程难题时，他能提供代码示例和解决方案
- 当你需要翻译外文资料时，他能快速准确地完成

这就是大型语言模型（LLM）给我们的感觉——一个永远在线、知识渊博、耐心细致的智能助手。

什么是大型语言模型（**LLM**）？

大型语言模型就像是一个"超级大脑"，它通过阅读海量的文本数据（包括书籍、文章、代码、对话等），学会了理解和生成人类语言。这个过程就像我们小时候学说话一样——通过不断听大人说话、看故事书，逐渐掌握了语言的规律。

但LLM的"学习"规模是我们无法想象的。它可能读过：

- 整个互联网上的公开文本
- 数百万本电子书
- 数亿行代码
- 无数的对话记录

正因为如此，它才能像一个真正的"语言专家"一样，理解语境、把握语义，甚至进行创造性的表达。

自然语言处理（**NLP**）是什么？

如果把计算机比作一个只会说"机器语言"的外国人，那么NLP就是教它学会"人类语言"的过程。就像教一个外国人学中文，需要：

- 教它认识汉字（分词）
- 教它理解语法（句法分析）
- 教它把握语境（语义理解）
- 教它表达思想（文本生成）

LLM是NLP领域的最新成果，它让计算机不仅能"听懂"我们的话，还能"说"出符合逻辑、富有创造性的内容。

为什么**LLM**这么重要？

传统上，如果我们想让计算机完成一个任务，需要程序员编写详细的指令，就像给机器人写操作手册一样。但有了LLM，我们只需要用自然语言描述需求，就像和朋友聊天一样简单。

举个例子：

- 传统方式：程序员需要写几十行代码，定义变量、写循环、处理异常...
- **LLM**方式：你只需要说"帮我写一个计算平均分的函数"，AI就能生成完整的代码

这就像从"需要专业翻译才能沟通"变成了"直接说中文就能交流"，大大降低了使用AI的门槛。

### 1.1.2 理解Transformer架构：AI的"注意力机制"

什么是**Transformer**?

Transformer可以理解为AI的"注意力系统"。想象你在阅读一本复杂的小说时：

- 你会重点关注主角的名字和关键情节
- 你会记住前面章节的伏笔，在后续章节中寻找呼应
- 你会同时理解多个角色的关系，而不是一个接一个地读

Transformer就是这样工作的。它有一个"注意力机制"，能够：

- 关注重点：就像你读书时会用荧光笔标记重点，Transformer会"标记"文本中的重要部分
- 并行处理：不像传统方法需要逐字逐句处理，Transformer可以同时"看"整个句子，就像你一眼扫过一段文字就能理解大意
- 理解上下文：它能够记住前面的内容，就像你读小说时记得前面章节的情节

为什么**Transformer**这么强大?

让我们用一个生活中的例子来理解：

假设你要理解"小明昨天买的苹果很好吃"这句话：

- 传统方法：像查字典一样，先查"小明"，再查"昨天"，再查"买"... 一个词一个词地理解，效率低
- **Transformer**方法：像人类阅读一样，一眼看到整句话，同时理解"小明"是主语、"苹果"是宾语、"很好吃"是评价，还能理解"昨天"是时间状语

这就是为什么Transformer能够：

1. 效率高：可以并行处理，就像多个人同时阅读不同的章节，然后汇总理解
2. 效果好：能够捕捉长距离的依赖关系，就像你读小说时能记住前面章节的伏笔
3. 可扩展：模型可以做得很大，就像图书馆越大，管理员的知识越丰富

### 1.1.3 解密DeepSeek：从文字到数字的魔法

什么是标记化（**Tokenization**）？

标记化就像把中文翻译成计算机能理解的"密码"。想象你要给一个只会说英语的朋友发中文消息，你需要：

1. 把中文句子拆分成词语
2. 把每个词语翻译成英语
3. 按照英语的语法组织句子

标记化就是类似的过程，只不过是把人类语言转换成计算机语言。

标记化的过程：

让我们用一个简单的例子来说明：

假设你输入"今天天气真好"：

1. 输入文本："今天天气真好"
2. 标记化：模型会把这句话拆分成：["今天", "天气", "真", "好"]
3. 转换为数字：每个词被转换成数字ID，比如：[1234, 5678, 9012, 3456]

这就像给每个词分配一个"身份证号码"，计算机通过这个号码就能找到对应的词。

预测步骤：**AI**如何**"**思考**"**

模型预测下一个词的过程，就像你写文章时的思考过程。

想象你在写"今天天气..."，你的大脑会：

1. 回顾前面的内容："今天天气"
2. 根据语境预测：可能是"很好"、"不错"、"晴朗"、"糟糕"等
3. 选择最合适的词：如果前面是"今天天气"，后面最可能是"很好"或"不错"

模型也是这样工作的：

- 它分析已有的文本："今天天气"
- 根据训练时学到的规律，预测下一个词的概率
- 选择概率最高的词作为输出

实际例子：

当你输入"Python是一种"时，模型会：

- 分析上下文："Python是一种"
- 预测可能的词：编程语言（概率90%）、工具（概率5%）、软件（概率3%）...
- 输出："编程语言"

这就是为什么AI的回答往往很自然，因为它学会了语言的规律。

## 1.2 DeepSeek模型简史：中国AI的探索之路

**DeepSeek**是什么?

DeepSeek是深度求索公司开发的大语言模型系列。如果说ChatGPT是"会说英语的AI"，那么DeepSeek就是"更懂中文的AI"。

就像中餐厅和西餐厅的区别：

- 西餐厅（ChatGPT）：虽然也能做中餐，但可能不够地道
- 中餐厅（DeepSeek）：专门为中国人的口味和习惯设计，更懂中文语境、更理解中国文化

**DeepSeek**的模型系列：三个专业助手

DeepSeek提供了三个"专业助手"，每个都有自己的特长：

1. **DeepSeek Chat -** 全能型助手
    - 就像一位博学的朋友，什么话题都能聊
    - 用途：日常对话、问答、写作、翻译
    - 特点：理解能力强，回答自然流畅，就像和一个知识渊博的朋友聊天
2. **DeepSeek Coder -** 编程专家
    - 就像一位经验丰富的程序员，专门帮你写代码
    - 用途：生成代码、优化代码、调试程序、解释代码
    - 特点：代码理解能力强，生成的代码质量高，就像有一个24小时在线的编程导师
3. **DeepSeek Math -** 数学老师
    - 就像一位耐心的数学老师，帮你解决数学问题

    - 用途：解决数学问题、数学推理、公式推导
    - 特点：数学能力突出，逻辑推理清晰

为什么选择**DeepSeek**?

选择DeepSeek就像选择一家好的餐厅，有几个理由：

1. 更懂中文：就像中餐厅的厨师更懂中国人的口味，DeepSeek对中文的理解更深入，能把握中文的语境、语气、文化背景
2. 代码能力强：如果你需要编程帮助，DeepSeek Coder就像一位经验丰富的程序员，能生成高质量的代码
3. 使用简单：API接口设计得很友好，就像点餐一样简单，不需要复杂的配置
4. 性价比高：相比其他服务，DeepSeek的价格更亲民，就像物美价廉的餐厅
5. 响应快速：就像快餐店出餐快，DeepSeek的响应速度也很快

## 1.3 LLM用例：AI如何改变我们的生活

**LLM**可以做什么?

让我们看看LLM在现实生活中的应用，就像看看智能手机能做什么一样：

### 1. 代码生成与优化：你的编程助手

场景：你是一个刚学Python的初学者，想写一个排序函数，但不知道怎么写。

传统方式：

- 打开搜索引擎
- 查找"Python排序函数"
- 阅读多篇教程
- 理解代码逻辑
- 自己写代码
- 调试错误

使用**LLM**：

- 直接问："用Python实现快速排序"
- AI立即生成完整代码，还带注释

就像从"需要自己查资料学习"变成了"直接问专家"。

### 2. 智能问答系统：24小时在线客服

场景：你半夜遇到问题，想咨询客服，但客服已经下班了。

传统方式：

- 等到第二天
- 或者发邮件等待回复

使用**LLM**：

- 立即得到回答
- 就像有一个24小时在线的客服

就像从"需要等待"变成了"随时可用"。

### 3. 文档总结和翻译：你的阅读助手

场景：你有一篇10000字的英文论文，需要快速了解主要内容。

传统方式：

- 逐字阅读（需要很长时间）
- 或者找人翻译（需要等待和费用）

使用**LLM**：

- 几秒钟生成200字的中文总结
- 或者直接翻译全文

就像从"需要花大量时间"变成了"秒懂"。

### 4. 创意写作助手：你的创作伙伴

场景：你需要写一首诗，但灵感枯竭。

传统方式：

- 苦思冥想
- 或者放弃

使用**LLM**：

- 描述你的需求："写一首关于春天的诗"
- AI立即生成优美的诗歌

就像从"独自创作"变成了"有创作伙伴"。

实际应用案例：

- **GitHub Copilot**：就像编程时的"自动补全"，但更智能，能理解你的意图
- **ChatGPT**：就像有一个知识渊博的朋友，随时可以聊天
- **Notion AI**：就像写作时的"灵感助手"，帮你组织思路、润色文字
- **Be My Eyes**：帮助视障人士"看"世界，就像他们的眼睛

## 1.4 警惕AI幻觉：当AI"记错了"

什么是**AI**幻觉?

AI幻觉是指AI生成看似合理但实际错误的信息。就像人有时会"记错"或"说错"一样，AI也会出错。

想象一下，你问一个朋友："北京是哪个国家的首都？"

- 正常情况下，他会说："中国"
- 但有时候，他可能会说错："日本"（记错了）

AI也是这样，它可能：

- 生成看似合理但实际错误的信息
- 对不确定的事情也给出肯定答案
- 混淆不同来源的信息

为什么会有**AI**幻觉?

让我们用一个生活中的例子来理解:

假设你问一个学生:"第二次世界大战是什么时候结束的?"

- 如果他学过历史,他会说:"1945年"
- 如果他没学过,他可能会猜:"可能是1940年?"(不确定但给出了答案)

AI也是这样:

1. 训练数据不完整:就像学生没学过某些知识,AI可能没见过某些信息
2. 过度自信:就像学生不确定但还是回答了,AI也可能对不确定的信息给出肯定答案
3. 上下文理解错误:就像学生误解了问题,AI也可能误解你的意图

如何避免**AI**幻觉?

就像我们不会完全相信一个人的话一样,我们也不应该完全相信AI:

1. 验证关键信息:对于重要信息,要核实来源,就像查字典确认拼写一样
2. 设置合理的期望:不要期望AI永远正确,就像不要期望人永远不犯错
3. 实施安全检查:在应用中添加验证机制,就像重要文件需要多人审核
4. 提供上下文:给模型更多相关信息,减少误解,就像问问题时提供背景信息

实际建议:

- 适合用**AI**的场景:创意写作、代码生成、文本总结、翻译等,AI很可靠
- ⚠ 需要谨慎的场景:医疗诊断、法律建议、财务决策等,需要人工审核
- 不适合用**AI**的场景:涉及生命安全、重大决策等,不要完全依赖AI

## 1.5 优化**DeepSeek**:让**AI**更懂你

如何提升模型表现?

就像教一个学生,你可以用不同的方法:

### 1. 提示工程(**Prompt Engineering**):问问题的艺术

什么是提示工程?

提示工程就像问问题的艺术。同样的问题,不同的问法会得到不同的答案。

不好的问法:

- "写代码"(太模糊,AI不知道要写什么)

好的问法:

- "用Python写一个计算斐波那契数列的函数,包含类型提示、文档字符串和错误处理"(清晰具体,AI知道要写什么)

就像问路一样：

- "怎么走？"（太模糊）
- "去天安门怎么走？"（清晰具体）

### 2. 微调（Fine-tuning）：专业培训

什么是微调?

微调就像给AI做"专业培训"。如果你想让AI成为"医疗专家"，就用医疗数据训练它；如果想让它成为"法律专家"，就用法律数据训练它。

什么时候需要微调?

- 需要特定领域的专业知识：就像培养专科医生，需要专门的医学知识
- 需要特定的输出格式：就像公司有固定的报告格式，需要AI按照格式输出
- 需要适应特定的对话风格：就像客服需要特定的语气，需要AI学会这种风格

### 3. 插件和工具集成：给AI装"工具"

什么是插件?

插件就像给AI装"工具"，让它能做更多事情。

例子：

- 搜索引擎插件：让AI能搜索最新信息，就像给它装了一个"浏览器"
- 计算器插件：让AI能进行复杂计算，就像给它装了一个"计算器"
- 代码执行插件：让AI能运行代码，就像给它装了一个"编译器"

## 1.6 小结：开启AI之旅

本章要点回顾：

1. LLM基础：我们了解了什么是大型语言模型，它就像一个博学的助手，能理解和生成人类语言
2. Transformer架构：我们理解了AI的"注意力机制"，它让AI能像人类一样理解语言
3. DeepSeek介绍：我们认识了DeepSeek模型系列，它就像三个专业助手，各有特长
4. 应用场景：我们看到了LLM在现实生活中的应用，从代码生成到智能问答，AI正在改变我们的生活
5. 注意事项：我们知道了AI幻觉的存在，学会了如何正确使用AI
6. 优化方法：我们学习了提示工程和微调等方法，让AI更好地为我们服务

下一步：

在下一章，我们将学习如何实际使用DeepSeek API，就像学习如何使用智能手机一样。你将编写你的第一个AI应用，体验AI的强大能力！

思考题：

1. LLM和传统程序有什么区别?
    - 传统程序：需要程序员写详细的指令，就像给机器人写操作手册
    - LLM：只需要用自然语言描述需求，就像和朋友聊天

2. 为什么**Transformer**架构这么重要?
    - 它让AI能像人类一样理解语言，而不是逐字逐句地处理
3. 在你的工作中，哪些场景可以用到**LLM**?
    - 写代码、写文档、回答问题、总结资料... 想想你的日常工作，很多都可以用AI辅助

小故事：

想象一下，你是一个刚入职的新员工，需要学习公司的各种流程。传统方式是你需要：

- 阅读厚厚的员工手册
- 参加培训课程
- 向老员工请教

但有了AI助手，你可以：

- 直接问："公司请假流程是什么？"
- AI立即给出详细步骤
- 就像有一个24小时在线的"老员工"随时为你解答

这就是LLM给我们的改变——让复杂的事情变简单，让专业的知识变易得。

# 第2章 深入了解DeepSeek的API：打开AI世界的大门

## 2.1 基本概念：API是什么?

什么是**API**?

想象一下，你走进一家餐厅：

- 你不需要知道厨房里怎么做菜
- 你只需要看菜单，点菜
- 服务员会把菜端给你

API（Application Programming Interface，应用程序编程接口）就像餐厅的菜单。你不需要知道AI模型内部是怎么工作的，只需要：

- 发送请求（点菜）
- 接收结果（上菜）

**DeepSeek API**是什么?

DeepSeek API就是DeepSeek提供的"菜单"，让你可以通过编程的方式调用AI模型。就像给AI发短信，AI会回复你一样。

为什么使用**API**?

使用API就像使用外卖APP：

- 简单：不需要自己做饭，只需要点餐
- 方便：可以集成到任何应用中
- 强大：可以处理各种任务

**API**的工作流程：

让我们用一个生活中的例子来理解：

```
你想知道明天的天气
  ↓
打开天气APP（你的程序）
  ↓
发送请求到天气服务器（调用API）
  ↓
服务器查询天气数据（AI处理）
  ↓
返回天气信息（返回结果）
  ↓
显示在APP上（显示结果）
```

DeepSeek API也是这样工作的：

```
你的程序 → 发送请求（文本） → DeepSeek API → 处理 → 返回结果（AI生成的文本） → 你的程序
```

## 2.2 DeepSeek API提供的可用模型：选择你的助手

可用的模型：

DeepSeek提供了两个主要的模型，就像两个不同专业的助手：

1. **deepseek-chat -** 全能型助手
    - 就像一位博学的朋友，什么话题都能聊
    - 最适合：日常对话、问答、写作、翻译
    - 特点：理解能力强，回答自然流畅
2. **deepseek-coder -** 编程专家
    - 就像一位经验丰富的程序员
    - 最适合：代码生成、代码优化、调试
    - 特点：代码理解能力强，生成的代码质量高

如何选择模型?

选择模型就像选择助手：

- 需要聊天、问答 → 选择 deepseek-chat（全能型）
- 需要写代码 → 选择 deepseek-coder（专业型）
- 不确定 → 先用 deepseek-chat（更通用）

## 2.3 在DeepSeek Playground中使用模型：先试试再买

什么是**Playground**?

Playground就像"试吃区"，你可以在不写代码的情况下测试AI的效果。就像买车前先试驾一样。

如何使用：

1. 访问DeepSeek官网
2. 找到Playground入口
3. 输入你的问题

4. 查看AI的回答

**Playground vs API**：

- **Playground**：适合测试和学习，不需要编程，就像在餐厅试吃
- **API**：适合集成到应用中，需要编程，就像点外卖

## 2.4 开始使用DeepSeek Python库：你的第一个AI程序

准备工作：就像准备做菜一样

### 步骤1：安装Python库

就像做菜需要准备食材，使用API需要安装库：

```
pip install openai
```

这就像下载一个"工具箱"，里面有你需要的工具。

### 步骤2：配置API密钥

API密钥就像你的"身份证"，证明你有权限使用服务。

创建 config.py 文件（参考 config.py.example）：

```
DEEPSEEK_API_KEY = "your-api-key-here"  # 你的API密钥
DEEPSEEK_BASE_URL = "https://api.deepseek.com"  # API地址
DEEPSEEK_MODEL = "deepseek-chat"  # 使用的模型
```

这就像在手机里保存WiFi密码，以后就不用每次都输入了。

### 步骤3：编写第一个程序

现在让我们写一个最简单的程序，就像学编程时的"Hello World"：

```
import os
import sys
from openai import OpenAI

# 添加项目根目录到路径
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

try:
    from config import DEEPSEEK_API_KEY, DEEPSEEK_BASE_URL, DEEPSEEK_MODEL
except ImportError:
    print("请先复制 config.py.example 为 config.py 并配置API密钥")
    DEEPSEEK_API_KEY = os.getenv("DEEPSEEK_API_KEY", "")
    DEEPSEEK_BASE_URL = "https://api.deepseek.com"
    DEEPSEEK_MODEL = "deepseek-chat"

def basic_chat():
    """基本的聊天对话示例"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
```

```
    print("请设置环境变量 DEEPSEEK_API_KEY 或在 config.py 中配置")
    return

# 初始化客户端
# 这就像打开聊天软件，准备和AI对话
client = OpenAI(
    api_key=DEEPSEEK_API_KEY,  # 你的"身份证"
    base_url=DEEPSEEK_BASE_URL  # AI的"地址"
)

# 准备消息
# messages是一个列表，包含对话历史
# role可以是 "user"（用户）、"assistant"（助手）、"system"（系统）
messages = [
    {"role": "user", "content": "请用Python实现一个快速排序算法，并添加注释"}
]

print("正在调用DeepSeek API...")
print(f"模型: {DEEPSEEK_MODEL}")
print(f"用户消息: {messages[0]['content']}\n")

try:
    # 调用API
    # 这就像给AI发短信，等待回复
    response = client.chat.completions.create(
        model=DEEPSEEK_MODEL,      # 使用的模型
        messages=messages,         # 对话消息
        temperature=0.7,           # 创造性参数（0-1，越高越有创造性）
        max_tokens=1000            # 最大生成token数
    )

    # 获取回复内容
    # 这就像收到AI的回复短信
    print("=" * 50)
    print("DeepSeek回复:")
    print("=" * 50)
    print(response.choices[0].message.content)
    print("=" * 50)

    # 显示使用的token数（用于了解成本）
    # 就像看手机流量使用情况
    print(f"\n使用Token数: {response.usage.total_tokens}")
    print(f"  - 输入: {response.usage.prompt_tokens}")
    print(f"  - 输出: {response.usage.completion_tokens}")

except Exception as e:
    print(f"错误: {e}")
    print("\n提示：")
    print("1. 请检查API密钥是否正确")
    print("2. 请检查网络连接")
    print("3. 请确认API余额是否充足")
```

```
if __name__ == "__main__":
    print("=" * 50)
    print("DeepSeek API 基本使用示例")
    print("=" * 50)
    print()

    # 运行基本聊天示例
    basic_chat()
```

代码解释：

让我们用生活中的例子来理解这段代码：

1. 导入库：from openai import OpenAI
    - 就像导入一个"工具箱"，里面有我们需要的工具
2. 初始化客户端：client = OpenAI(...)
    - 就像打开聊天软件，准备和AI对话
    - api_key：你的"身份证"，证明你有权限
    - base_url：AI的"地址"，告诉程序去哪里找AI
3. 准备消息：messages = [...]
    - 就像写短信内容，准备发给AI
4. 调用**API**：client.chat.completions.create(...)
    - 就像发送短信，等待AI回复
    - temperature：控制AI的创造性，就像控制聊天的风格
    - max_tokens：限制回复的长度，就像限制短信字数
5. 获取结果：response.choices[0].message.content
    - 就像收到AI的回复短信

运行方法：

```
python chapter02/basic_usage.py
```

## 2.5 使用DeepSeek进行文本生成：让AI为你"打字"

### 流式响应：像真人打字一样

有时候AI需要生成很长的文本，如果等全部生成完再显示，就像等一篇文章全部写完才给你看，体验不好。

流式响应可以边生成边显示，就像真人打字一样，你看到文字一个一个地出现。

生活中的例子：

想象你在和朋友视频聊天：

- 非流式：朋友说"我要告诉你一件事"，然后沉默5分钟，突然说完整件事（体验差）
- 流式：朋友边说边想，你看到他的表情和语气，感觉更自然（体验好）

流式响应代码：

```
def streaming_chat():
    """流式响应示例"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    client = OpenAI(
        api_key=DEEPSEEK_API_KEY,
        base_url=DEEPSEEK_BASE_URL
    )

    messages = [
        {"role": "user", "content": "请用一句话介绍Python编程语言的特点"}
    ]

    print("流式响应示例:")
    print(f"用户消息: {messages[0]['content']}\n")
    print("DeepSeek回复（流式）: ", end="", flush=True)

    try:
        # 关键：设置 stream=True 启用流式响应
        # 就像开启"实时显示"模式
        stream = client.chat.completions.create(
            model=DEEPSEEK_MODEL,
            messages=messages,
            stream=True  # 启用流式响应
        )

        # 逐块接收并打印
        # 就像看朋友一个字一个字地打字
        for chunk in stream:
            # chunk.choices[0].delta.content 是新增的内容
            if chunk.choices[0].delta.content is not None:
                print(chunk.choices[0].delta.content, end="", flush=True)
        print("\n")

    except Exception as e:
        print(f"\n错误: {e}")

# 运行流式响应示例
if __name__ == "__main__":
    streaming_chat()
```

流式响应的优势：

- 用户体验更好：不需要等待全部生成，就像看直播比看录播更有趣
- 感觉更自然：像真人打字一样，有"实时感"
- 可以中途停止：如果发现不对可以取消，就像可以打断朋友说话

## 2.6 使用其他文本补全模型：多轮对话的艺术

## 多轮对话：记住我们的聊天

在实际应用中，我们通常需要多轮对话，AI需要记住之前的对话内容。就像和朋友聊天，如果朋友每次都忘记之前说过什么，聊天就没法继续了。

生活中的例子：

想象你和朋友聊天：

- 第一轮：你问"什么是装饰器？"，朋友解释
- 第二轮：你问"能给我一个例子吗？"，朋友知道你在说装饰器，给出例子

如果朋友不记得第一轮的对话，他就不知道"例子"指的是什么。

多轮对话代码：

```python
def multi_turn_conversation():
    """多轮对话示例"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    client = OpenAI(
        api_key=DEEPSEEK_API_KEY,
        base_url=DEEPSEEK_BASE_URL
    )

    # 初始化对话历史
    # system消息用于设定AI的角色和行为
    # 就像告诉AI："你是一个专业的Python编程助手"
    messages = [
        {"role": "system", "content": "你是一个专业的Python编程助手，擅长编写清晰、高效的代码。"}
    ]

    # 第一轮对话
    messages.append({"role": "user", "content": "什么是装饰器？"})
    print("用户: 什么是装饰器？")

    response = client.chat.completions.create(
        model=DEEPSEEK_MODEL,
        messages=messages,
        temperature=0.7
    )

    assistant_reply = response.choices[0].message.content
    print(f"助手: {assistant_reply}\n")

    # 重要：将AI的回复添加到messages中，保持对话历史
    # 就像记住朋友说过的话
    messages.append({"role": "assistant", "content": assistant_reply})

    # 第二轮对话（基于上下文）
    # 注意：这里没有说"装饰器"，但AI知道我们在说装饰器
    messages.append({"role": "user", "content": "能给我一个实际的例子吗？"})
    print("用户: 能给我一个实际的例子吗？")

    response = client.chat.completions.create(
        model=DEEPSEEK_MODEL,
        messages=messages,
        temperature=0.7
    )

    assistant_reply = response.choices[0].message.content
    print(f"助手: {assistant_reply}\n")
```

多轮对话的关键：

1. 保存对话历史：每次调用都要包含之前的对话，就像记住聊天记录

2. 使用**system**消息：设定AI的角色和行为，就像告诉AI它的身份
3. 正确设置**role**：user、assistant、system要区分清楚，就像区分谁在说话

## 参数配置：控制**AI**的"性格"

不同的参数会影响AI的输出效果，就像控制聊天的风格。

**temperature**参数：控制创造性

想象你在和朋友聊天：

- 低**temperature (0.1)**：朋友回答很确定，就像在说"这是标准答案"
- 高**temperature (0.9)**：朋友回答很有创意，就像在说"让我想想，可能有几种方式..."

参数配置代码：

```python
def parameter_demo():
    """参数配置示例"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    client = OpenAI(
        api_key=DEEPSEEK_API_KEY,
        base_url=DEEPSEEK_BASE_URL
    )

    prompt = "写一首关于春天的短诗"

    print("=" * 50)
    print("参数配置示例")
    print("=" * 50)

    # temperature参数：控制创造性
    # 范围：0-1
    # 低temperature (0.1) - 更确定性，输出更稳定
    # 就像让朋友用"标准答案"回答
    print("\n低temperature (0.1) - 更确定性:")
    response = client.chat.completions.create(
        model=DEEPSEEK_MODEL,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.1,    # 低创造性
        max_tokens=200
    )
    print(response.choices[0].message.content)

    # 高temperature (0.9) - 更创造性，输出更多样
    # 就像让朋友"自由发挥"
    print("\n高temperature (0.9) - 更创造性:")
    response = client.chat.completions.create(
        model=DEEPSEEK_MODEL,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.9,    # 高创造性
        max_tokens=200
    )
    print(response.choices[0].message.content)
```

重要参数说明：

| 参数 | 说明 | 推荐值 | 生活比喻 |
| --- | --- | --- | --- |
| temperature | 创造性，0-1，越高越有创造性 | 0.7（对话）、0.2（代码） | 就像控制聊天的"自由度" |
| max_tokens | 最大生成token数 | 根据需求设置 | 就像限制短信字数 |
| top_p | 核采样参数 | 0.9（默认） | 就像控制选择的多样性 |
| frequency_penalty | 频率惩罚，减少重复 | 0-2，通常0 | 就像提醒"不要说重复的话" |

## 代码生成示例：让AI帮你写代码

使用deepseek-coder模型生成代码，就像请一个专业的程序员帮你写代码。

代码生成代码：

```
def code_generation_example():
    """代码生成示例（使用deepseek-coder模型）"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    client = OpenAI(
        api_key=DEEPSEEK_API_KEY,
        base_url=DEEPSEEK_BASE_URL
    )

    code_prompt = """请用Python实现一个函数，功能是：
1. 接收一个整数列表
2. 返回列表中的最大值和最小值
3. 如果列表为空，返回None
请添加详细的注释和类型提示。"""

    print("=" * 50)
    print("代码生成示例")
    print("=" * 50)
    print(f"提示: {code_prompt}\n")

    try:
        # 尝试使用deepseek-coder模型（代码生成专用）
        # 就像请专业的程序员写代码
        response = client.chat.completions.create(
            model="deepseek-coder",  # 代码生成专用模型
            messages=[{"role": "user", "content": code_prompt}],
            temperature=0.2,  # 代码生成使用较低temperature，更稳定
            max_tokens=1000
        )

        print("生成的代码:")
        print("-" * 50)
        print(response.choices[0].message.content)
        print("-" * 50)

    except Exception as e:
        # 如果deepseek-coder不可用，使用deepseek-chat
        print(f"注意: {e}")
        print("使用deepseek-chat模型替代...\n")

        response = client.chat.completions.create(
            model=DEEPSEEK_MODEL,
            messages=[{"role": "user", "content": code_prompt}],
            temperature=0.2,
            max_tokens=1000
        )
```

```
print("生成的代码:")
print("-" * 50)
print(response.choices[0].message.content)
print("-" * 50)
```

代码生成技巧：

1. 使用专用模型：deepseek-coder更适合代码生成，就像专业的事找专业的人
2. 低**temperature**：代码需要准确，不要太有创造性，就像写技术文档要严谨
3. 详细描述需求：描述越详细，生成的代码越好，就像给程序员的需求文档要详细
4. 要求注释：在提示中要求添加注释和类型提示，就像要求代码要有文档

## 2.7 考虑因素：使用**API**的注意事项

使用**API**时需要注意：

就像使用任何服务一样，使用API也有一些注意事项：

### 1. **API**调用频率限制

问题：每个API都有调用频率限制，就像餐厅有营业时间一样。

生活中的例子：

- 如果你点餐太频繁，餐厅可能会说"请稍等，我们正在准备"
- API也是这样，如果调用太频繁，可能会被限制

解决方案：添加重试机制和速率限制，就像"如果忙线，等一会儿再打"

### 2. **Token**使用成本

问题：API按token收费，就像手机按流量收费一样。

生活中的例子：

- 发短信按字数收费，字数越多费用越高
- API也是这样，输入和输出都算token

解决方案：优化提示，减少不必要的token，就像"长话短说"

### 3. 响应时间

问题：API调用需要网络请求，有延迟，就像发短信需要时间才能收到回复。

生活中的例子：

- 发短信给朋友，可能需要几秒钟才能收到回复
- API也是这样，需要网络传输时间

解决方案：使用流式响应，提升用户体验，就像"边打字边显示"

### 4. 错误处理

问题：网络可能出错，API可能返回错误，就像打电话可能占线。

生活中的例子：

- 打电话给朋友，可能遇到"对方正在通话中"
- API调用也可能失败

解决方案：添加try-except处理异常，就像"如果打不通，等会儿再打"

错误处理示例：

```
try:
    response = client.chat.completions.create(
        model=DEEPSEEK_MODEL,
        messages=messages
    )
except Exception as e:
    print(f"API调用失败: {e}")
    # 可以添加重试逻辑
    # 可以记录日志
    # 可以返回默认值
```

## 2.8 其他DeepSeek API和功能：更多可能性

### 流式响应

- 边生成边显示，提升用户体验
- 参考2.5节的示例代码

### 函数调用（Function Calling）

- 让AI能够调用你定义的函数
- 适合需要实时数据的场景
- 例如：查询天气、搜索信息

生活中的例子：

- 就像给AI装"工具"，让它能做更多事情
- 比如让AI"帮我查一下天气"，AI会调用天气API

### 多轮对话

- 保持对话上下文
- 参考2.6节的示例代码

## 2.9 小结：掌握API的使用

本章要点：

1. 了解了API的基本概念，就像理解了"如何点外卖"
2. 学会了如何调用DeepSeek API，就像学会了"如何使用聊天软件"
3. 掌握了基本参数配置，就像学会了"如何设置聊天风格"
4. 学会了多轮对话和流式响应，就像学会了"如何和朋友聊天"
5. 了解了代码生成的方法，就像学会了"如何请AI帮忙写代码"

速查清单：

- [ ] 已获取DeepSeek API密钥（就像有了"身份证"）
- [ ] 已安装openai库：pip install openai（就像有了"工具箱"）
- [ ] 已配置config.py文件（就像保存了"WiFi密码"）
- [ ] 已成功运行basic_usage.py（就像发送了第一条短信）
- [ ] 理解temperature参数的作用（就像理解了"聊天风格"）
- [ ] 理解max_tokens参数的作用（就像理解了"短信字数限制"）
- [ ] 能够进行多轮对话（就像能和朋友连续聊天）
- [ ] 能够使用流式响应（就像能看到朋友实时打字）
- [ ] 能够生成代码（就像能请AI帮忙写代码）

下一步：

在下一章，我们将学习如何构建完整的应用程序，包括聊天机器人、代码助手等实际项目。就像从"学会发短信"到"开发聊天软件"！

练习：

1. 修改**basic_usage.py**，让**AI**用不同的风格回答
    - 就像让朋友用不同的语气说话
    - 试试不同的temperature值
2. 实现一个简单的问答程序，可以连续提问
    - 就像和朋友连续聊天
    - 记住每次的对话历史
3. 尝试生成不同类型的代码（函数、类、脚本）
    - 就像请AI写不同类型的代码
    - 看看不同模型的区别

小故事：

想象一下，你是一个刚学会用智能手机的人：

- 第一天：学会了发短信（基本API调用）
- 第二天：学会了发语音消息（流式响应）
- 第三天：学会了连续聊天（多轮对话）
- 第四天：学会了用各种功能（参数配置）

现在你已经掌握了DeepSeek API的基本使用，就像掌握了智能手机的基本功能。接下来，我们将学习如何用这些"功能"构建真正的"应用"！

# 第3章 使用DeepSeek构建应用程序：从想法到现实

## 3.1 应用程序开发概述：把你的想法变成现实

在前两章，我们就像学会了如何使用工具。现在，是时候用这些工具来建造一些真正有用的东西了。

想象一下，你是一个木工：

- 第1章：你了解了什么是锤子、锯子（了解了AI和LLM）

- 第**2**章：你学会了如何使用这些工具（学会了API调用）
- 第**3**章：现在你要用这些工具建造一个书架、一张桌子（构建应用程序）

什么是应用程序?

应用程序就像你建造的"家具"，它们能完成特定的任务，解决实际的问题。

比如：

- 聊天机器人：就像一个24小时在线的朋友，随时可以和你聊天
- 代码助手：就像一个经验丰富的编程导师，随时可以帮你写代码
- 文档总结工具：就像一个高效的阅读助手，帮你快速理解长文档

为什么要构建应用程序?

让我们用一个生活中的例子来理解：

假设你每天都要做同样的事情——整理文件：

- 传统方式：每天花1小时手动整理（效率低）
- 有了应用程序：点击一下，自动完成（效率高）

构建应用程序的好处：

- 解决实际问题：把AI能力应用到真实场景，就像用工具解决生活中的问题
- 提升效率：自动化重复性工作，就像用洗衣机洗衣服，省时省力
- 学习实践：通过项目学习编程，就像通过做菜学会烹饪
- 展示能力：可以作为作品展示，就像展示你的手工艺品

本章将学习构建：

我们将一起构建四个实用的应用程序，就像学习做四道菜：

1. 命令行聊天机器人：就像做一个简单的家常菜，容易上手
2. **Web**聊天应用：就像做一个精致的菜品，需要更多技巧
3. 代码助手：就像做一个专业菜品，需要专业工具
4. 文档总结工具：就像做一个多功能工具，能做多种事情

## **3.2** 软件架构设计原则：建造房子的智慧

在开始编码之前，我们需要了解一些设计原则。这就像建房子之前要先设计图纸一样，好的设计能让程序更稳定、更易维护。

想象你要建一座房子：

- 没有设计：想到哪建到哪，最后可能结构混乱，难以维护
- 有设计：先画图纸，规划好每个房间的功能，最后房子结构清晰，易于维护

写程序也是这样，好的设计原则能让代码更清晰、更易维护。

### **1.** 模块化设计：像搭积木一样

什么是模块化?

模块化就像搭积木。你把大程序拆分成小模块，每个模块负责一个功能，然后像搭积木一样组合起来。

生活中的例子：

想象你在组装一台电脑：

- 不好的方式：所有零件混在一起，不知道哪个是哪个
- 好的方式：CPU放在一个盒子，内存放在一个盒子，硬盘放在一个盒子，需要时拿出来组装

程序也是这样：

- 不好的设计：所有代码混在一起，难以理解和维护
- 好的设计：分成不同模块，每个模块负责一个功能，需要时调用

好处：

- 代码更清晰：就像房间分类清晰，找东西容易
- 容易维护：就像某个房间出问题，只需要修那个房间
- 可以复用：就像积木可以重复使用，模块也可以在其他项目中使用

示例：

```
# 不好的设计：所有代码混在一起
def chatbot():
    # API调用
    # 用户输入
    # 输出处理
    # 错误处理
    # ... 所有功能混在一起

# 好的设计：分成不同模块
class Chatbot:
    def __init__(self):
        self.client = self._init_client()  # 初始化模块

    def chat(self, user_input):
        response = self._call_api(user_input)  # API调用模块
        return self._format_response(response)  # 格式化模块
```

## 2. 错误处理：为意外情况做准备

为什么需要错误处理?

程序运行时可能出错，就像生活中会遇到意外情况一样。

生活中的例子：

想象你在开车：

- 没有准备：突然下雨，没有雨刷，无法继续行驶
- 有准备：突然下雨，打开雨刷，继续安全行驶

程序也是这样：

- 没有错误处理：网络断开，程序崩溃，用户看到错误信息
- 有错误处理：网络断开，程序显示友好提示，建议用户稍后重试

可能出错的情况：

- 网络连接失败：就像打电话时信号不好

- **API**返回错误：就像餐厅说"这道菜今天没有了"
- 用户输入无效：就像你点菜时说了不存在的菜名

如何处理错误?

使用 try-except 捕获错误，就像准备雨伞应对下雨：

```
try:
    # 可能出错的代码
    # 就像尝试打电话
    response = client.chat.completions.create(...)
except Exception as e:
    # 出错时执行的代码
    # 就像电话打不通时的处理
    print(f"错误: {e}")
    # 可以记录日志、返回默认值等
    # 就像记录"电话打不通"，然后建议"稍后再试"
```

## 3. 配置管理：把重要信息放在安全的地方

什么是配置管理?

配置管理就像把重要信息（如密码、钥匙）放在安全的地方，而不是写在显眼的地方。

生活中的例子：

想象你的银行卡密码：

- 不好的方式：把密码写在纸上，贴在电脑屏幕上（不安全！）
- 好的方式：把密码记在脑子里，或者放在加密的密码管理器中

程序也是这样：

- 不好的做法：把API密钥直接写在代码里，就像把密码写在纸上
- 好的做法：把API密钥放在配置文件中，就像把密码放在密码管理器里

好处：

- 安全：不会泄露密钥，就像密码不会被人看到
- 灵活：可以轻松修改配置，就像可以随时改密码
- 方便：不同环境用不同配置，就像家里和办公室用不同的钥匙

示例：

```
# 不好的做法：硬编码
# 就像把密码写在纸上
api_key = "sk-1234567890"  # 不安全！如果代码泄露，密钥也泄露了

# 好的做法：从配置文件读取
# 就像从密码管理器读取密码
from config import DEEPSEEK_API_KEY  # 密钥在配置文件中，代码里看不到
```

## 4. 日志记录：程序的"日记本"

为什么需要日志?

日志就像程序的"日记本"，记录程序做了什么、遇到了什么问题。就像你写日记记录每天发生的事情一样。

生活中的例子：

想象你在管理一家餐厅：

- 没有记录：客人投诉说"上菜慢"，你不知道是什么时候、哪个服务员、哪道菜出了问题
- 有记录：客人投诉时，你查记录，发现"下午3点，服务员A，菜品B，用时15分钟"，立即知道问题所在

程序也是这样：

- 没有日志：程序出错了，你不知道是什么时候、哪里、为什么出错
- 有日志：程序出错了，你查日志，发现"下午3点，API调用失败，错误：网络超时"，立即知道问题

示例：

```
import logging

# 设置日志级别
# 就像决定日记的详细程度
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# 记录信息
# 就像写日记
logger.info("开始调用API")  # 记录"开始做某件事"
logger.error("API调用失败")  # 记录"某件事出错了"
```

## 3.3 LLM驱动型应用程序的漏洞：安全防护的重要性

什么是漏洞?

漏洞就像房子的安全隐患。应用程序可能存在的安全问题或错误，如果不注意，可能会导致数据泄露、费用超支等问题。

生活中的例子：

想象你的房子：

- 没有防护：门没锁，窗户没关，任何人都能进来（不安全！）
- 有防护：门锁好，窗户关好，有监控系统（安全！）

应用程序也是这样：

- 没有防护：可能被恶意使用，导致费用超支、数据泄露
- 有防护：验证输入、限制访问、监控使用，确保安全

### 1. 输入验证

问题： 用户可能输入恶意内容

解决： 验证和过滤用户输入

```python
def validate_input(user_input):
    """验证用户输入"""
    if not user_input or len(user_input) > 1000:
        return False
    # 可以添加更多验证规则
    return True
```

### 2. 输出过滤

问题： AI可能生成不当内容

解决： 过滤和检查AI输出

```python
def filter_output(content):
    """过滤不当内容"""
    # 可以检查敏感词、不当内容等
    if "敏感词" in content:
        return "内容已过滤"
    return content
```

### 3. 速率限制

问题： 用户可能频繁调用，导致API费用过高

解决： 限制调用频率

```python
import time

class RateLimiter:
    def __init__(self, max_calls=10, period=60):
        self.max_calls = max_calls
        self.period = period
        self.calls = []

    def can_call(self):
        now = time.time()
        # 移除过期的调用记录
        self.calls = [t for t in self.calls if now - t < self.period]

        if len(self.calls) < self.max_calls:
            self.calls.append(now)
            return True
        return False
```

### 4. 成本控制

问题： API调用需要费用

解决： 监控Token使用量，设置上限

```
class CostController:
    def __init__(self, max_tokens_per_day=100000):
        self.max_tokens = max_tokens_per_day
        self.used_tokens = 0

    def check_quota(self, tokens):
        if self.used_tokens + tokens > self.max_tokens:
            return False
        self.used_tokens += tokens
        return True
```

## 3.4 示例项目

### 3.4.1 简单聊天机器人：你的第一个AI朋友

功能： 在命令行中与AI对话

想象一下，你有一个24小时在线的朋友，随时可以和你聊天。这就是聊天机器人给你的感觉。

特点：

- 支持多轮对话：就像和朋友连续聊天，AI会记住之前说过的话
- 可以清空历史：就像开始一个新话题，清空之前的对话
- 可以查看历史记录：就像查看聊天记录，回顾之前的对话

生活中的应用场景：

- 学习助手：就像有一个耐心的老师，随时可以问问题
- 编程助手：就像有一个经验丰富的程序员，随时可以请教
- 写作助手：就像有一个创意伙伴，随时可以讨论想法

完整代码：

```
"""
第3章示例：简单命令行聊天机器人
一个可以在终端中交互的聊天机器人
"""

import os
import sys
from openai import OpenAI

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

try:
    from config import DEEPSEEK_API_KEY, DEEPSEEK_BASE_URL, DEEPSEEK_MODEL
except ImportError:
    DEEPSEEK_API_KEY = os.getenv("DEEPSEEK_API_KEY", "")
    DEEPSEEK_BASE_URL = "https://api.deepseek.com"
    DEEPSEEK_MODEL = "deepseek-chat"

class SimpleChatbot:
    """简单的聊天机器人类"""
```

```
def __init__(self):
    if not DEEPSEEK_API_KEY:
        raise ValueError("未配置API密钥，请设置DEEPSEEK_API_KEY环境变量或在config.py中配置")

    # 初始化API客户端
    self.client = OpenAI(
        api_key=DEEPSEEK_API_KEY,
        base_url=DEEPSEEK_BASE_URL
    )
    self.model = DEEPSEEK_MODEL

    # 初始化对话历史
    # system消息设定AI的角色
    self.messages = [
        {"role": "system", "content": "你是一个友好、专业的AI助手，能够回答各种问题并提供帮助。"}
    ]
    self.conversation_history = []  # 用于显示历史记录

def chat(self, user_input):
    """处理用户输入并返回回复"""
    if not user_input.strip():
        return "请输入有效的问题。"

    # 添加用户消息到对话历史
    self.messages.append({"role": "user", "content": user_input})

    try:
        # 调用API
        response = self.client.chat.completions.create(
            model=self.model,
            messages=self.messages,
            temperature=0.7,
            max_tokens=1000
        )

        # 获取AI回复
        assistant_reply = response.choices[0].message.content

        # 保存对话历史
        self.messages.append({"role": "assistant", "content": assistant_reply})
        self.conversation_history.append({
            "user": user_input,
            "assistant": assistant_reply
        })

        return assistant_reply

    except Exception as e:
        return f"错误: {str(e)}"

def clear_history(self):
```

```
        """清空对话历史"""
        self.messages = [
            {"role": "system", "content": "你是一个友好、专业的AI助手，能够回答各种问题并提供帮助。"}
        ]
        self.conversation_history = []
        print("对话历史已清空")

    def show_history(self):
        """显示对话历史"""
        if not self.conversation_history:
            print("暂无对话历史")
            return

        print("\n" + "=" * 50)
        print("对话历史")
        print("=" * 50)
        for i, item in enumerate(self.conversation_history, 1):
            print(f"\n[对话 {i}]")
            print(f"用户: {item['user']}")
            print(f"助手: {item['assistant']}")
        print("=" * 50 + "\n")

def main():
    """主函数"""
    print("=" * 50)
    print("DeepSeek 聊天机器人")
    print("=" * 50)
    print("输入 'quit' 或 'exit' 退出")
    print("输入 'clear' 清空对话历史")
    print("输入 'history' 查看对话历史")
    print("=" * 50 + "\n")

    try:
        chatbot = SimpleChatbot()
    except ValueError as e:
        print(f"初始化失败: {e}")
        return

    while True:
        try:
            user_input = input("你: ").strip()

            if not user_input:
                continue

            # 处理特殊命令
            if user_input.lower() in ['quit', 'exit', 'q']:
                print("再见！")
                break
            elif user_input.lower() == 'clear':
                chatbot.clear_history()
                continue
```

```
            elif user_input.lower() == 'history':
                chatbot.show_history()
                continue

            # 获取回复
            print("助手: ", end="", flush=True)
            reply = chatbot.chat(user_input)
            print(reply)
            print()

        except KeyboardInterrupt:
            print("\n\n再见！")
            break
        except Exception as e:
            print(f"发生错误: {e}")

if __name__ == "__main__":
    main()
```

运行方法：

```
python chapter03/simple_chatbot.py
```

代码解释：

1. **SimpleChatbot**类：封装了聊天机器人的所有功能
2. **init**方法：初始化API客户端和对话历史
3. **chat**方法：处理用户输入，调用API，返回回复
4. **clear_history**方法：清空对话历史
5. **show_history**方法：显示历史对话
6. **main**函数：主循环，处理用户交互

使用示例：

```
你: 你好
助手: 你好！我是AI助手，有什么可以帮助你的吗?

你: Python中如何读取文件?
助手: 在Python中，可以使用open()函数读取文件...

你: history
[显示对话历史]

你: clear
对话历史已清空

你: quit
再见！
```

### 3.4.2 Web聊天应用（Streamlit）

功能： 在浏览器中与AI对话

特点：

- 图形界面，更友好
- 自动保存对话历史
- 可以清空历史

完整代码：

```
"""
第3章示例：基于Streamlit的Web聊天应用
使用Streamlit构建一个简单的Web聊天界面
"""

import os
import sys
import streamlit as st
from openai import OpenAI

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

try:
    from config import DEEPSEEK_API_KEY, DEEPSEEK_BASE_URL, DEEPSEEK_MODEL
except ImportError:
    DEEPSEEK_API_KEY = os.getenv("DEEPSEEK_API_KEY", "")
    DEEPSEEK_BASE_URL = "https://api.deepseek.com"
    DEEPSEEK_MODEL = "deepseek-chat"

# 页面配置
st.set_page_config(
    page_title="DeepSeek 聊天助手",
    page_icon="    ",
    layout="wide"
)

# 初始化session state（Streamlit的状态管理）
if "messages" not in st.session_state:
    st.session_state.messages = [
        {"role": "system", "content": "你是一个友好、专业的AI助手。"}
    ]

if "client" not in st.session_state:
    if DEEPSEEK_API_KEY:
        st.session_state.client = OpenAI(
            api_key=DEEPSEEK_API_KEY,
            base_url=DEEPSEEK_BASE_URL
        )
    else:
        st.session_state.client = None

def get_response(user_message):
    """获取AI回复"""
    if not st.session_state.client:
```

```
        return "错误：未配置API密钥。请在config.py中设置DEEPSEEK_API_KEY。"

    # 添加用户消息
    st.session_state.messages.append({"role": "user", "content": user_message})

    try:
        # 调用API
        response = st.session_state.client.chat.completions.create(
            model=DEEPSEEK_MODEL,
            messages=st.session_state.messages,
            temperature=0.7,
            max_tokens=1000
        )

        assistant_reply = response.choices[0].message.content
        st.session_state.messages.append({"role": "assistant", "content": assistant_reply})

        return assistant_reply
    except Exception as e:
        return f"错误: {str(e)}"

# 主界面
st.title("      DeepSeek 聊天助手")

# 侧边栏
with st.sidebar:
    st.header("设置")

    if st.button("清空对话历史"):
        st.session_state.messages = [
            {"role": "system", "content": "你是一个友好、专业的AI助手。"}
        ]
        st.rerun()

    st.markdown("---")
    st.markdown("### 使用说明")
    st.markdown("""
    1. 在输入框中输入您的问题
    2. 按Enter或点击发送按钮
    3. AI助手会回复您的问题
    4. 支持多轮对话
    """)

# 显示聊天历史
for message in st.session_state.messages:
    if message["role"] != "system":
        with st.chat_message(message["role"]):
            st.markdown(message["content"])

# 用户输入
if prompt := st.chat_input("请输入您的问题..."):
    # 显示用户消息
```

```
    with st.chat_message("user"):
        st.markdown(prompt)

    # 获取并显示AI回复
    with st.chat_message("assistant"):
        with st.spinner("思考中..."):
            response = get_response(prompt)
            st.markdown(response)

# 运行说明
if not DEEPSEEK_API_KEY:
    st.warning("⚠ 未检测到API密钥。请配置DEEPSEEK_API_KEY环境变量或在config.py中设置。")
```

运行方法：

```
streamlit run chapter03/streamlit_chat.py
```

代码解释：

1. **Streamlit**：一个Python Web框架，可以快速创建Web应用
2. **st.session_state**：保存应用状态（对话历史）
3. **st.chat_message**：显示聊天消息
4. **st.chat_input**：聊天输入框

### 3.4.3 代码助手

功能： 生成、优化、解释和调试代码

特点：

- 生成代码：根据描述生成代码
- 优化代码：优化已有代码
- 解释代码：解释代码功能
- 调试代码：修复代码错误

完整代码：

```
"""
第3章示例：代码助手
一个可以帮助生成、优化和解释代码的助手
"""

import os
import sys
from openai import OpenAI

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

try:
    from config import DEEPSEEK_API_KEY, DEEPSEEK_BASE_URL, DEEPSEEK_MODEL
except ImportError:
    DEEPSEEK_API_KEY = os.getenv("DEEPSEEK_API_KEY", "")
    DEEPSEEK_BASE_URL = "https://api.deepseek.com"
    DEEPSEEK_MODEL = "deepseek-chat"
```

```
class CodeAssistant:
    """代码助手类"""

    def __init__(self):
        if not DEEPSEEK_API_KEY:
            raise ValueError("未配置API密钥")

        self.client = OpenAI(
            api_key=DEEPSEEK_API_KEY,
            base_url=DEEPSEEK_BASE_URL
        )
        self.model = "deepseek-coder"  # 优先使用代码模型
        self.fallback_model = DEEPSEEK_MODEL

    def generate_code(self, description, language="Python"):
        """根据描述生成代码"""
        prompt = f"""请用{language}语言实现以下功能：
{description}

要求：
1. 代码要清晰、易读
2. 添加详细的注释
3. 包含必要的错误处理
4. 如果适用，添加类型提示"""

        try:
            response = self.client.chat.completions.create(
                model=self.model,
                messages=[{"role": "user", "content": prompt}],
                temperature=0.2,  # 代码生成使用较低temperature
                max_tokens=2000
            )
            return response.choices[0].message.content
        except Exception as e:
            # 如果deepseek-coder不可用，使用fallback模型
            print(f"注意: 使用备用模型 ({e})")
            response = self.client.chat.completions.create(
                model=self.fallback_model,
                messages=[{"role": "user", "content": prompt}],
                temperature=0.2,
                max_tokens=2000
            )
            return response.choices[0].message.content

    def optimize_code(self, code):
        """优化代码"""
        prompt = f"""请优化以下代码，使其更高效、更易读：

```python
{code}
```
```

请提供优化后的代码，并说明优化的原因。"""

```
    response = self.client.chat.completions.create(
        model=self.model if hasattr(self, 'model') else self.fallback_model,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.3,
        max_tokens=2000
    )
    return response.choices[0].message.content

def explain_code(self, code):
    """解释代码"""
    prompt = f"""请详细解释以下代码的功能、工作原理和关键点：

{code}
```"""

        response = self.client.chat.completions.create(
            model=self.model if hasattr(self, 'model') else self.fallback_model,
            messages=[{"role": "user", "content": prompt}],
            temperature=0.5,
            max_tokens=1500
        )
        return response.choices[0].message.content

    def debug_code(self, code, error_message):
        """调试代码"""
        prompt = f"""以下代码出现了错误，请帮助修复：

代码：
```python
{code}
```

错误信息：
{error_message}

请提供修复后的代码和解释。"""

    response = self.client.chat.completions.create(
        model=self.model if hasattr(self, 'model') else self.fallback_model,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.2,
        max_tokens=2000
    )
    return response.choices[0].message.content

def main():
    """主函数"""
    print("=" * 50)
    print("DeepSeek 代码助手")
    print("=" * 50)
    print("功能：")
    print("1. 生成代码 - 根据描述生成代码")
```
```

```
print("2. 优化代码 - 优化已有代码")
print("3. 解释代码 - 解释代码功能")
print("4. 调试代码 - 修复代码错误")
print("=" * 50 + "\n")

try:
    assistant = CodeAssistant()
except ValueError as e:
    print(f"初始化失败: {e}")
    return

while True:
    print("\n请选择功能：")
    print("1. 生成代码")
    print("2. 优化代码")
    print("3. 解释代码")
    print("4. 调试代码")
    print("0. 退出")

    choice = input("\n请输入选项: ").strip()

    if choice == "0":
        print("再见！")
        break
    elif choice == "1":
        description = input("请描述需要实现的功能: ")
        language = input("编程语言 (默认Python): ").strip() or "Python"
        print("\n正在生成代码...")
        result = assistant.generate_code(description, language)
        print("\n生成的代码:")
        print("=" * 50)
        print(result)
        print("=" * 50)
    elif choice == "2":
        print("请粘贴需要优化的代码 (输入END结束):")
        code_lines = []
        while True:
            line = input()
            if line.strip() == "END":
                break
            code_lines.append(line)
        code = "\n".join(code_lines)
        print("\n正在优化代码...")
        result = assistant.optimize_code(code)
        print("\n优化结果:")
        print("=" * 50)
        print(result)
        print("=" * 50)
    elif choice == "3":
        print("��粘贴需要解释的代码 (输入END结束):")
        code_lines = []
        while True:
```

```
            line = input()
            if line.strip() == "END":
                break
            code_lines.append(line)
        code = "\n".join(code_lines)
        print("\n正在分析代码...")
        result = assistant.explain_code(code)
        print("\n代码解释:")
        print("=" * 50)
        print(result)
        print("=" * 50)
    elif choice == "4":
        print("请粘贴出错的代码 (输入END结束):")
        code_lines = []
        while True:
            line = input()
            if line.strip() == "END":
                break
            code_lines.append(line)
        code = "\n".join(code_lines)
        error = input("请输入错误信息: ")
        print("\n正在调试代码...")
        result = assistant.debug_code(code, error)
        print("\n调试结果:")
        print("=" * 50)
        print(result)
        print("=" * 50)
    else:
        print("无效选项，请重新选择")
```

if **name** == "**main**":
main()

**运行方法：**

```bash
python chapter03/code_assistant.py
```

### 3.4.4 文档总结工具

功能： 总结文本、提取关键点、翻译、情感分析

完整代码：

```
"""
第3章示例：文档总结工具
可以总结长文本、提取关键信息的工具
"""

import os
import sys
from openai import OpenAI
```

```
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

try:
    from config import DEEPSEEK_API_KEY, DEEPSEEK_BASE_URL, DEEPSEEK_MODEL
except ImportError:
    DEEPSEEK_API_KEY = os.getenv("DEEPSEEK_API_KEY", "")
    DEEPSEEK_BASE_URL = "https://api.deepseek.com"
    DEEPSEEK_MODEL = "deepseek-chat"

class DocumentSummarizer:
    """文档总结工具类"""

    def __init__(self):
        if not DEEPSEEK_API_KEY:
            raise ValueError("未配置API密钥")

        self.client = OpenAI(
            api_key=DEEPSEEK_API_KEY,
            base_url=DEEPSEEK_BASE_URL
        )
        self.model = DEEPSEEK_MODEL

    def summarize(self, text, max_length=200):
        """总结文本"""
        prompt = f"""请总结以下文本，控制在{max_length}字以内，提取关键信息：

{text}

总结："""

        response = self.client.chat.completions.create(
            model=self.model,
            messages=[{"role": "user", "content": prompt}],
            temperature=0.3,
            max_tokens=500
        )
        return response.choices[0].message.content

    def extract_key_points(self, text, num_points=5):
        """提取关键点"""
        prompt = f"""请从以下文本中提取{num_points}个关键点，用简洁的列表形式呈现：

{text}

关键点："""

        response = self.client.chat.completions.create(
            model=self.model,
            messages=[{"role": "user", "content": prompt}],
            temperature=0.3,
            max_tokens=500
```

```
        )
        return response.choices[0].message.content

    def translate(self, text, target_language="英文"):
        """翻译文本"""
        prompt = f"""请将以下文本翻译成{target_language}：

{text}

翻译："""

        response = self.client.chat.completions.create(
            model=self.model,
            messages=[{"role": "user", "content": prompt}],
            temperature=0.3,
            max_tokens=2000
        )
        return response.choices[0].message.content

    def analyze_sentiment(self, text):
        """分析文本情感"""
        prompt = f"""请分析以下文本的情感倾向（正面/中性/负面），并简要说明原因：

{text}

情感分析："""

        response = self.client.chat.completions.create(
            model=self.model,
            messages=[{"role": "user", "content": prompt}],
            temperature=0.3,
            max_tokens=300
        )
        return response.choices[0].message.content

def read_file(file_path):
    """读取文件内容"""
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            return f.read()
    except Exception as e:
        return f"读取文件失败: {e}"

def main():
    """主函数"""
    print("=" * 50)
    print("DeepSeek 文档总结工具")
    print("=" * 50)
    print("功能：")
    print("1. 文本总结")
    print("2. 提取关键点")
    print("3. 文本翻译")
```

```
print("4. 情感分析")
print("=" * 50 + "\n")

try:
    summarizer = DocumentSummarizer()
except ValueError as e:
    print(f"初始化失败: {e}")
    return

while True:
    print("\n请选择功能：")
    print("1. 文本总结")
    print("2. 提取关键点")
    print("3. 文本翻译")
    print("4. 情感分析")
    print("5. 从文件读取文本")
    print("0. 退出")

    choice = input("\n请输入选项: ").strip()

    if choice == "0":
        print("再见！")
        break
    elif choice == "5":
        file_path = input("请输入文件路径: ").strip()
        text = read_file(file_path)
        if text.startswith("读取文件失败"):
            print(text)
            continue
        print(f"\n文件内容 ({len(text)} 字符):")
        print("-" * 50)
        print(text[:500] + ("..." if len(text) > 500 else ""))
        print("-" * 50)
        continue
    else:
        if choice in ["1", "2", "3", "4"]:
            print("\n请输入文本 (输入END结束，或输入FILE:文件路径从文件读取):")
            first_line = input().strip()

            if first_line.startswith("FILE:"):
                file_path = first_line[5:].strip()
                text = read_file(file_path)
                if text.startswith("读取文件失败"):
                    print(text)
                    continue
            else:
                text_lines = [first_line]
                while True:
                    line = input()
                    if line.strip() == "END":
                        break
                    text_lines.append(line)
```

```
            text = "\n".join(text_lines)

        if not text.strip():
            print("文本不能为空")
            continue

        print("\n处理中...")

        if choice == "1":
            max_len = input("最大长度 (默认200字): ").strip()
            max_len = int(max_len) if max_len.isdigit() else 200
            result = summarizer.summarize(text, max_len)
            print("\n总结结果:")
        elif choice == "2":
            num_points = input("关键点数量 (默认5): ").strip()
            num_points = int(num_points) if num_points.isdigit() else 5
            result = summarizer.extract_key_points(text, num_points)
            print("\n关键点:")
        elif choice == "3":
            target = input("目标语言 (默认英文): ").strip() or "英文"
            result = summarizer.translate(text, target)
            print(f"\n翻译结果 ({target}):")
        elif choice == "4":
            result = summarizer.analyze_sentiment(text)
            print("\n情感分析结果:")

        print("=" * 50)
        print(result)
        print("=" * 50)
    else:
        print("无效选项，请重新选择")

if __name__ == "__main__":
    main()
```

运行方法：

```
python chapter03/document_summarizer.py
```

## 3.5 小结

本章要点：

1. 学习了软件架构设计原则
2. 了解了应用程序的常见漏洞和防护
3. 构建了4个实际应用项目
4. 掌握了模块化设计和错误处理

项目总结：

| 项目 | 特点 | 适用场景 |
| --- | --- | --- |
| 聊天机器人 | 简单、易用 | 学习、演示 |

Web聊天应用 图形界面 实际部署
代码助手 专业功能 开发辅助
文档总结工具 文本处理 内容分析

下一步：

在下一章，我们将学习提示工程和微调等高级技巧，进一步提升应用效果！

练习：

1. 改进聊天机器人，添加更多功能
2. 为代码助手添加代码执行功能
3. 优化文档总结工具，支持批量处理

# 第4章 DeepSeek的高级技巧：让AI更懂你

## 4.1 提示工程：问问题的艺术

什么是提示工程?

提示工程就像问问题的艺术。同样的问题，不同的问法会得到完全不同的答案。

想象你在问路：

- 不好的问法："怎么走？"（太模糊，对方不知道你要去哪）
- 好的问法："去天安门怎么走？"（清晰具体，对方知道要指什么路）

提示工程就是这样，通过优化输入提示，让AI输出更好的结果。

为什么需要提示工程?

让我们用一个生活中的例子来理解：

假设你想让朋友帮你写一篇文章：

- 不好的方式："帮我写篇文章"（朋友不知道要写什么、写给谁、什么风格）
- 好的方式："帮我写一篇关于人工智能的科普文章，面向初学者，语言要通俗易懂，大约1000字"（朋友知道要写什么、怎么写）

AI也是这样，提示越清晰，输出越好。

### 4.1.1 提示工程的基本原则

**1.** 明确性：说清楚你的需求

原则： 清晰、具体地描述任务

生活中的例子：

想象你在餐厅点菜：

- 不明确："来点好吃的"（服务员不知道你要什么）
- 明确："来一份宫保鸡丁，微辣，不要花生"（服务员知道要做什么）

代码示例：

```
#      不明确
prompt = "写一个函数"

#      明确
prompt = """用Python写一个函数，功能是：
1. 接收一个整数列表
2. 返回列表中的最大值和最小值
3. 如果列表为空，返回None
4. 包含类型提示和文档字符串"""
```

**2.** 结构化：像写作文一样组织

原则： 使用格式化的提示模板

生活中的例子：

想象你在写一份工作汇报：

- 没有结构：想到哪写到哪，混乱不清
- 有结构：先写背景，再写问题，最后写解决方案，清晰明了

代码示例：

```
prompt = f"""任务：{task}

要求：
1. {requirement1}
2. {requirement2}
3. {requirement3}

输出格式：{format}"""
```

**3.** 上下文：提供背景信息

原则： 提供足够的背景信息

生活中的例子：

想象你在向朋友解释一个问题：

- 没有背景："这个不对"（朋友不知道你在说什么）
- 有背景："我刚才说的那个方案不对，因为..."（朋友知道你在说什么）

代码示例：

```
prompt = f"""你是一位经验丰富的Python开发工程师。

任务：{task}

请以专业开发者的角度，提供最佳实践方案。"""
```

**4.** 示例：让**AI**学习你的风格

原则： 使用few-shot learning提供示例

生活中的例子：

想象你在教朋友做菜：

- 只说理论："做菜要掌握火候"（朋友不知道具体怎么做）
- 给示例："你看，这样炒出来的菜颜色正好，这就是掌握火候"（朋友知道具体怎么做）

代码示例：

```
prompt = f"""以下是几个示例：

示例1:
输入: "今天天气很好"
输出: "positive"

示例2:
输入: "这部电影很糟糕"
输出: "negative"

现在请分析：
输入: "{user_input}"
输出: """
```

### 4.1.2 常用提示技巧

技巧**1**：角色设定——让**AI**扮演特定角色

原理： 让AI扮演特定角色，输出更专业

生活中的例子：

想象你在咨询问题：

- 不设定角色：问"怎么写代码？"，可能得到各种答案
- 设定角色：说"你是一个经验丰富的Python程序员，请告诉我怎么写代码？"，会得到更专业的答案

完整代码示例：

```
"""
角色设定提示示例
"""

import os
import sys
from openai import OpenAI

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

try:
    from config import DEEPSEEK_API_KEY, DEEPSEEK_BASE_URL, DEEPSEEK_MODEL
except ImportError:
    DEEPSEEK_API_KEY = os.getenv("DEEPSEEK_API_KEY", "")
    DEEPSEEK_BASE_URL = "https://api.deepseek.com"
    DEEPSEEK_MODEL = "deepseek-chat"
```

```
def role_based_prompt(task):
    """角色设定提示"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    client = OpenAI(
        api_key=DEEPSEEK_API_KEY,
        base_url=DEEPSEEK_BASE_URL
    )

    # 关键：设定AI的角色
    # 就像告诉AI："你现在是一个专业的Python程序员"
    prompt = f"""你是一位经验丰富的Python开发工程师，擅长编写高质量、可维护的代码。

任务：{task}

请以专业开发者的角度，提供最佳实践方案。"""

    response = client.chat.completions.create(
        model=DEEPSEEK_MODEL,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.7,
        max_tokens=1000
    )

    return response.choices[0].message.content

# 使用示例
if __name__ == "__main__":
    task = "设计一个用户认证系统"
    result = role_based_prompt(task)
    print("任务:", task)
    print("\n结果:")
    print("=" * 50)
    print(result)
    print("=" * 50)
```

运行方法：

```
python chapter04/prompt_engineering.py
```

技巧**2**：思维链——让**AI**展示思考过程

原理： 引导AI逐步思考，得到更准确的答案

生活中的例子：

想象你在解数学题：

- 不展示过程：直接说答案"42"（你不知道怎么算出来的）
- 展示过程：说"首先...然后...最后得到42"（你知道每一步是怎么做的）

完整代码示例：

```
def chain_of_thought(problem):
    """思维链提示"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    client = OpenAI(
        api_key=DEEPSEEK_API_KEY,
        base_url=DEEPSEEK_BASE_URL
    )

    # 关键：要求AI展示思考过程
    # 就像要求朋友"说说你是怎么想的"
    prompt = f"""请解决以下问题，并展示你的思考过程：

问题：{problem}

请按以下步骤思考：
1. 理解问题
2. 分析关键点
3. 制定解决方案
4. 实施并验证

开始："""

    response = client.chat.completions.create(
        model=DEEPSEEK_MODEL,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.7,
        max_tokens=1500
    )

    return response.choices[0].message.content

# 使用示例
if __name__ == "__main__":
    problem = "如何优化一个处理大量数据的Python函数？"
    result = chain_of_thought(problem)
    print("问题:", problem)
    print("\n思考过程:")
    print("=" * 50)
    print(result)
    print("=" * 50)
```

思维链的优势：

- 让AI展示推理过程：就像看朋友解题，知道每一步是怎么做的
- 更容易发现错误：如果某一步错了，你能看出来
- 结果更可靠：有思考过程的结果更可信

技巧3：**Few-shot**学习——通过示例学习

原理： 通过提供示例，让AI学习任务模式

生活中的例子：

想象你在教朋友识别水果：

- 只说概念："苹果是红色的"（朋友可能不知道具体是什么样）
- 给示例："你看，这个就是苹果，红色的，圆形的"（朋友知道具体是什么样）

完整代码示例：

```
def few_shot_learning(task, examples):
    """Few-shot学习提示"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    client = OpenAI(
        api_key=DEEPSEEK_API_KEY,
        base_url=DEEPSEEK_BASE_URL
    )

    # 关键：提供多个示例
    # 就像给AI看几个例子，让它学习
    prompt = f"""以下是几个示例：

{examples}

现在请完成类似的任务：
{task}"""

    response = client.chat.completions.create(
        model=DEEPSEEK_MODEL,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.3,  # Few-shot通常用较低temperature
        max_tokens=1000
    )

    return response.choices[0].message.content

# 使用示例
if __name__ == "__main__":
    examples = """示例1:
输入: "今天天气很好"
输出: "positive"

示例2:
输入: "这部电影很糟糕"
输出: "negative"

示例3:
```

```
输入: "今天是星期一"
输出: "neutral" """

    task = "输入: '这个产品性价比很高'"
    result = few_shot_learning(task, examples)
    print("示例:")
    print(examples)
    print(f"\n任务: {task}")
    print("\n结果:")
    print("=" * 50)
    print(result)
    print("=" * 50)
```

技巧4：结构化输出——指定输出格式

原理： 指定输出格式，方便后续处理

生活中的例子：

想象你在填表格：

- 不指定格式："写一下你的信息"（可能写得很乱）
- 指定格式："姓名：，*年龄：*，电话：___"（格式清晰，容易处理）

完整代码示例：

```
def structured_output(data, format_type="JSON"):
    """结构化输出提示"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    client = OpenAI(
        api_key=DEEPSEEK_API_KEY,
        base_url=DEEPSEEK_BASE_URL
    )

    # 根据格式类型设置不同的指令
    if format_type == "JSON":
        format_instruction = """请以JSON格式输出，包含以下字段：
- summary: 总结
- key_points: 关键点列表
- sentiment: 情感倾向"""
    elif format_type == "XML":
        format_instruction = """请以XML格式输出，使用以下结构：
<result>
  <summary>总结</summary>
  <key_points>
    <point>关键点1</point>
    <point>关键点2</point>
  </key_points>
  <sentiment>情感倾向</sentiment>
</result>"""
```

```
    else:
        format_instruction = "请以清晰的列表格式输出"

    prompt = f"""请分析以下内容：

{data}

{format_instruction}"""

    response = client.chat.completions.create(
        model=DEEPSEEK_MODEL,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.3,
        max_tokens=1000
    )

    return response.choices[0].message.content

# 使用示例
if __name__ == "__main__":
    data = """人工智能正在快速发展，在各个领域都有广泛应用。
机器学习、深度学习等技术不断突破，为人类生活带来便利。
但同时也需要注意AI伦理和安全问题。"""

    result = structured_output(data, "JSON")
    print("输入数据:")
    print(data)
    print("\nJSON格式输出:")
    print("=" * 50)
    print(result)
    print("=" * 50)
```

技巧**5**：约束条件——明确限制和要求

原理： 明确限制和要求，确保输出符合预期

生活中的例子：

想象你在订餐：

- 不说明限制："来份套餐"（可能不符合你的要求）
- 说明限制："来份套餐，不要辣的，不要海鲜"（符合你的要求）

完整代码示例：

```python
def constraint_prompt(task, constraints):
    """带约束条件的提示"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    client = OpenAI(
        api_key=DEEPSEEK_API_KEY,
        base_url=DEEPSEEK_BASE_URL
    )

    # 将约束条件格式化为列表
    constraints_text = "\n".join([f"- {c}" for c in constraints])

    prompt = f"""请完成以下任务，必须遵守以下约束条件：

任务：{task}

约束条件：
{constraints_text}

请确保严格遵守所有约束条件。"""

    response = client.chat.completions.create(
        model=DEEPSEEK_MODEL,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.5,
        max_tokens=1000
    )

    return response.choices[0].message.content

# 使用示例
if __name__ == "__main__":
    task = "写一个Python函数计算斐波那契数列"
    constraints = [
        "使用递归实现",
        "添加类型提示",
        "包含文档字符串",
        "处理边界情况（n<=0）"
    ]

    result = constraint_prompt(task, constraints)
    print("任务:", task)
    print("约束条件:")
    for c in constraints:
        print(f"  - {c}")
    print("\n结果:")
    print("=" * 50)
    print(result)
    print("=" * 50)
```

完整提示工程示例代码：

所有技巧的完整代码都在 chapter04/prompt_engineering.py 中，可以直接运行：

```
python chapter04/prompt_engineering.py
```

## 4.2 微调：让AI成为你的专业助手

什么是微调?

微调就像给AI做"专业培训"。如果你想让AI成为"医疗专家"，就用医疗数据训练它；如果想让它成为"法律专家"，就用法律数据训练它。

生活中的例子：

想象你在培养一个助手：

- 通用培训：教他基本的办公技能（就像基础模型）
- 专业培训：教他你公司的具体业务（就像微调）

微调就是给AI做"专业培训"，让它更懂你的领域。

什么时候需要微调?

1. 需要特定领域的专业知识
    - 例如：医疗、法律、金融等专业领域
    - 就像需要专科医生，而不是全科医生
2. 需要特定的输出格式
    - 例如：特定的JSON结构、特定的代码风格
    - 就像公司有固定的报告格式
3. 需要适应特定的对话风格
    - 例如：客服机器人、教学助手
    - 就像需要特定的语气和风格

微调 **vs** 提示工程：

| 方法 | 优点 | 缺点 | 适用场景 | 生活比喻 |
| --- | --- | --- | --- | --- |
| 提示工程 | 快速、灵活、免费 | 效果有限 | 大多数场景 | 就像临时指导 |
| 微调 | 效果好、专业 | 需要数据、成本高 | 专业领域 | 就像长期培训 |

### 4.2.1 微调流程：从数据到模型

步骤**1**：准备训练数据

数据格式：

微调数据需要是JSONL格式（每行一个JSON对象），就像准备教材一样。

完整代码示例：

```
"""
```

```
微调数据准备示例
"""

import json

def prepare_training_data():
    """准备训练数据示例"""

    # 示例：准备一个代码生成任务的训练数据
    # 就像准备教学案例
    training_data = [
        {
            "messages": [
                {"role": "system", "content": "你是一个Python代码生成助手，专门生成简洁高效的代码。"},
                {"role": "user", "content": "生成一个计算列表平均值的函数"},
                {"role": "assistant", "content": "```python\ndef calculate_average(numbers):\n    \"\"\"计算数字列表的平均值\"\"\"\n    if not numbers:\n        return 0\n    return sum(numbers) / len(numbers)\n```"}
            ]
        },
        {
            "messages": [
                {"role": "system", "content": "你是一个Python代码生成助手，专门生成简洁高效的代码。"},
                {"role": "user", "content": "生成一个检查字符串是否为回文的函数"},
                {"role": "assistant", "content": "```python\ndef is_palindrome(s):\n    \"\"\"检查字符串是否为回文\"\"\"\n    s = s.lower().replace(' ', '')\n    return s == s[::-1]\n```"}
            ]
        }
    ]

    return training_data

def save_training_data(data, filename="training_data.jsonl"):
    """保存训练数据为JSONL格式"""
    with open(filename, 'w', encoding='utf-8') as f:
        for item in data:
            f.write(json.dumps(item, ensure_ascii=False) + '\n')
    print(f"训练数据已保存到 {filename}")
    print(f"共 {len(data)} 条数据")

# 使用示例
if __name__ == "__main__":
    print("=" * 50)
    print("DeepSeek 模型微调指南")
    print("=" * 50)
    print()

    # 准备训练数据
    print("1. 准备训练数据...")
    training_data = prepare_training_data()
    print(f"   生成了 {len(training_data)} 条示例数据")

    # 保存数据
```

```
print("\n2. 保存训练数据...")
save_training_data(training_data)

print("\n" + "=" * 50)
print("微调数据准备完成！")
print("=" * 50)
print("\n注意事项：")
print("1. 实际微调需要DeepSeek API支持微调功能")
print("2. 训练数据需要足够多（建议至少100条）")
print("3. 数据质量比数量更重要")
print("4. 微调前建议先用提示工程尝试解决问题")
print("5. 微调后需要评估模型性能")
```

运行方法：

```
python chapter04/fine_tuning_guide.py
```

### 4.2.2 微调最佳实践：让培训更有效

1. 数据质量 > 数据数量
    - 100条高质量数据 > 1000条低质量数据
    - 就像100个优秀学生 > 1000个普通学生
2. 数据多样性
    - 覆盖不同的场景和问题类型
    - 就像教材要覆盖各个知识点
3. 先尝试提示工程
    - 微调成本高，先试试提示工程能否解决问题
    - 就像先试试简单方法，不行再复杂方法
4. 小规模测试
    - 先用少量数据测试，确认有效再扩大规模
    - 就像先小范围试点，再全面推广
5. 持续评估
    - 定期评估模型效果，及时调整
    - 就像定期检查培训效果

## 4.3 小结：掌握高级技巧

本章要点：

1. 学习了提示工程的基本原则和技巧，就像学会了"问问题的艺术"
2. 掌握了5种常用提示技巧，就像掌握了5种沟通方法
3. 了解了微调的流程和最佳实践，就像了解了"专业培训"的方法
4. 学会了准备微调数据，就像学会了准备教材

提示工程技巧总结：

| 技巧 | 适用场景 | 效果 | 生活比喻 |
| --- | --- | --- | --- |
| 角色设定 | 需要专业输出 | ☆☆☆☆ | 就像找专业顾问 |
| 思维链 | 复杂问题 | ☆☆☆☆☆ | 就像看解题过程 |
| Few-shot | 特定格式 | ☆☆☆☆ | 就像看示例学习 |
| 结构化输出 | 需要解析 | ☆☆☆☆☆ | 就像填表格 |
| 约束条件 | 严格限制 | ☆☆☆☆ | 就像提要求 |

下一步：

在下一章，我们将学习使用LangChain框架，构建更强大的应用！就像从"手工制作"到"使用专业工具"！

练习：

1. 尝试不同的提示技巧，对比效果
    - 就像尝试不同的沟通方式，看哪种更有效
2. 为你的应用设计专门的提示模板
    - 就像为你的工作设计专门的沟通模板
3. 准备一个微调数据集（即使不实际微调）
    - 就像准备教材，即使现在不用，以后可能用到

# 第5章 使用LangChain框架和插件增强LLM的功能：从手工到自动化

## 5.1 LangChain概述：你的AI应用工具箱

什么是**LangChain**?

LangChain就像是一个"AI应用工具箱"，里面装满了各种现成的工具和组件。有了它，你不需要从零开始造轮子，只需要选择合适的工具组装起来。

生活中的例子：

想象你要做一道菜：

- 不用**LangChain**：需要自己种菜、自己磨面、自己调味... 从零开始（耗时耗力）
- 用**LangChain**：超市里有现成的食材、调料、工具，你只需要选择、组合、烹饪（简单高效）

开发LLM应用也是这样：

- 不用**LangChain**：需要自己处理对话历史、自己管理状态、自己集成工具... 从零开始
- 用**LangChain**：框架提供了现成的组件，你只需要选择、组合、使用

为什么使用**LangChain**?

让我们用一个更具体的例子：

假设你要做一个智能客服系统：

- 不用**LangChain**：

    - 需要自己管理对话历史（复杂）
    - 需要自己处理多轮对话（容易出错）
    - 需要自己集成知识库（技术难度高）
- 用**LangChain**：
    - 框架提供了记忆管理组件（简单）
    - 框架提供了对话链组件（可靠）
    - 框架提供了向量数据库集成（容易）

**LangChain**的核心功能：

就像工具箱里的不同工具：

1. 链式调用（**Chains**）：将多个组件串联起来
    - 就像流水线，一个步骤接一个步骤
2. 记忆管理（**Memory**）：管理对话历史
    - 就像给AI装一个"记忆系统"，让它记住之前的对话
3. 工具集成（**Tools**）：集成外部工具和API
    - 就像给AI装"工具"，让它能做更多事情
4. 向量存储（**Vector Stores**）：支持RAG（检索增强生成）
    - 就像给AI装一个"知识库"，让它能基于文档回答问题

**LangChain**的优势：

- 简化开发：提供现成的组件，就像有现成的食材
- 功能强大：支持复杂应用，就像能做各种菜
- 易于扩展：可以自定义组件，就像可以自己调调料
- 社区活跃：有很多示例和插件，就像有很多菜谱可以参考

# 5.2 使用LangChain构建应用程序：组装你的第一个应用

## 5.2.1 基本使用：打开工具箱

安装**LangChain**：

就像买工具箱一样，先要准备好工具：

```
pip install langchain langchain-openai
```

基本使用示例：

现在让我们看看如何使用这个"工具箱"：

```
"""
LangChain基本使用示例
"""

import os
```

```python
import sys
from langchain_openai import ChatOpenAI
from langchain.schema import HumanMessage, SystemMessage

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

try:
    from config import DEEPSEEK_API_KEY, DEEPSEEK_BASE_URL, DEEPSEEK_MODEL
except ImportError:
    DEEPSEEK_API_KEY = os.getenv("DEEPSEEK_API_KEY", "")
    DEEPSEEK_BASE_URL = "https://api.deepseek.com"
    DEEPSEEK_MODEL = "deepseek-chat"

def basic_langchain_usage():
    """LangChain基本使用"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    # 初始化LangChain的ChatOpenAI
    # 注意：虽然叫ChatOpenAI，但可以配置为使用DeepSeek
    llm = ChatOpenAI(
        model=DEEPSEEK_MODEL,
        openai_api_key=DEEPSEEK_API_KEY,
        openai_api_base=DEEPSEEK_BASE_URL,  # 关键：设置DeepSeek的API地址
        temperature=0.7
    )

    # 创建消息
    # SystemMessage：设定AI的角色
    # HumanMessage：用户的消息
    messages = [
        SystemMessage(content="你是一个专业的Python编程助手。"),
        HumanMessage(content="请解释什么是装饰器，并给出一个实际例子。")
    ]

    print("=" * 50)
    print("LangChain + DeepSeek 基本使用")
    print("=" * 50)
    print(f"用户消息: {messages[1].content}\n")

    # 调用模型
    # invoke方法用于单次调用
    response = llm.invoke(messages)

    print("助手回复:")
    print("-" * 50)
    print(response.content)
    print("-" * 50)

def streaming_langchain():
    """LangChain流式响应"""
```

```
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    llm = ChatOpenAI(
        model=DEEPSEEK_MODEL,
        openai_api_key=DEEPSEEK_API_KEY,
        openai_api_base=DEEPSEEK_BASE_URL,
        temperature=0.7,
        streaming=True  # 启用流式响应
    )

    messages = [
        HumanMessage(content="请用一句话介绍Python的特点")
    ]

    print("\n" + "=" * 50)
    print("LangChain 流式响应")
    print("=" * 50)
    print(f"用户消息: {messages[0].content}\n")
    print("助手回复（流式）: ", end="", flush=True)

    # stream方法用于流式响应
    for chunk in llm.stream(messages):
        if chunk.content:
            print(chunk.content, end="", flush=True)
    print("\n")

def batch_processing():
    """批量处理"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    llm = ChatOpenAI(
        model=DEEPSEEK_MODEL,
        openai_api_key=DEEPSEEK_API_KEY,
        openai_api_base=DEEPSEEK_BASE_URL,
        temperature=0.7
    )

    questions = [
        "什么是列表推导式？",
        "什么是生成器？",
        "什么是上下文管理器？"
    ]

    print("\n" + "=" * 50)
    print("LangChain 批量处理")
    print("=" * 50)

    # 准备多个消息列表
```

```
    messages_list = [[HumanMessage(content=q)] for q in questions]

    # batch方法用于批量处理
    responses = llm.batch(messages_list)

    for i, (question, response) in enumerate(zip(questions, responses), 1):
        print(f"\n问题 {i}: {question}")
        print(f"回答: {response.content}")
        print("-" * 50)

if __name__ == "__main__":
    print("=" * 50)
    print("LangChain + DeepSeek 集成示例")
    print("=" * 50)
    print()

    try:
        # 基本使用
        basic_langchain_usage()

        # 流式响应
        streaming_langchain()

        # 批量处理
        batch_processing()

        print("\n" + "=" * 50)
        print("所有示例完成！")
        print("=" * 50)

    except Exception as e:
        print(f"错误: {e}")
        print("\n提示：")
        print("1. 请确保已安装 langchain 和 langchain-openai")
        print("2. 请检查API密钥配置")
        print("3. 请检查网络连接")
```

运行方法：

```
python chapter05/langchain_basic.py
```

代码解释：

1. **ChatOpenAI**：LangChain的LLM封装类
2. **openai_api_base**：关键参数，设置为DeepSeek的API地址
3. **invoke**：单次调用
4. **stream**：流式响应
5. **batch**：批量处理

### 5.2.2 链式调用：像流水线一样工作

什么是链式调用?

链式调用就像工厂的流水线。你将多个步骤串联起来，形成一个完整的流程，就像产品在流水线上一个工序接一个工序。

生活中的例子：

想象你在做一道菜：

1. 切菜 → 2. 炒菜 → 3. 调味 → 4. 装盘

这就是一个"链"，每个步骤的输出是下一个步骤的输入。

示例： 生成代码 → 添加注释 → 优化代码

就像：

1. 生成代码（第一步）
2. 添加注释（第二步，基于第一步的代码）
3. 优化代码（第三步，基于第二步的代码）

完整代码示例：

```
"""
LangChain链式调用示例
"""

import os
import sys
from langchain_openai import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain.chains import LLMChain, SimpleSequentialChain

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

try:
    from config import DEEPSEEK_API_KEY, DEEPSEEK_BASE_URL, DEEPSEEK_MODEL
except ImportError:
    DEEPSEEK_API_KEY = os.getenv("DEEPSEEK_API_KEY", "")
    DEEPSEEK_BASE_URL = "https://api.deepseek.com"
    DEEPSEEK_MODEL = "deepseek-chat"

def simple_chain():
    """简单链式调用"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    llm = ChatOpenAI(
        model=DEEPSEEK_MODEL,
        openai_api_key=DEEPSEEK_API_KEY,
        openai_api_base=DEEPSEEK_BASE_URL,
        temperature=0.7
    )

    # 创建提示模板
```

```
    # {concept} 是占位符，会被实际值替换
    prompt = ChatPromptTemplate.from_template(
        "请用一句话解释以下编程概念：{concept}"
    )

    # 创建链
    # LLMChain：将提示模板和LLM组合
    chain = LLMChain(llm=llm, prompt=prompt)

    print("=" * 50)
    print("LangChain 简单链式调用")
    print("=" * 50)

    concepts = ["装饰器", "生成器", "上下文管理器"]

    for concept in concepts:
        # run方法执行链，传入参数
        result = chain.run(concept)
        print(f"\n概念: {concept}")
        print(f"解释: {result}")

def sequential_chain():
    """顺序链式调用"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    llm = ChatOpenAI(
        model=DEEPSEEK_MODEL,
        openai_api_key=DEEPSEEK_API_KEY,
        openai_api_base=DEEPSEEK_BASE_URL,
        temperature=0.7
    )

    # 第一步：生成代码
    code_prompt = ChatPromptTemplate.from_template(
        "请用Python实现一个{task}的函数，只返回代码，不要其他说明。"
    )
    # output_key指定输出键名
    code_chain = LLMChain(llm=llm, prompt=code_prompt, output_key="code")

    # 第二步：添加注释
    comment_prompt = ChatPromptTemplate.from_template(
        "请为以下Python代码添加详细的中文注释：\n\n{code}"
    )
    comment_chain = LLMChain(llm=llm, prompt=comment_prompt, output_key="commented_code")

    # 组合成顺序链
    # SimpleSequentialChain：按顺序执行多个链
    overall_chain = SimpleSequentialChain(
        chains=[code_chain, comment_chain],
        verbose=True  # 显示执行过程
```

```
    )

    print("\n" + "=" * 50)
    print("LangChain 顺序链式调用")
    print("=" * 50)
    print("任务：生成代码并添加注释\n")

    task = "计算斐波那契数列的第n项"
    result = overall_chain.run(task)

    print("\n最终结果:")
    print("-" * 50)
    print(result)
    print("-" * 50)

if __name__ == "__main__":
    print("=" * 50)
    print("LangChain 链式调用示例")
    print("=" * 50)
    print()

    try:
        # 简单链
        simple_chain()

        # 顺序链
        sequential_chain()

        print("\n" + "=" * 50)
        print("所有示例完成！")
        print("=" * 50)

    except Exception as e:
        print(f"错误: {e}")
        import traceback
        traceback.print_exc()
```

运行方法：

```
python chapter05/langchain_chains.py
```

链式调用的优势：

- 模块化：每个步骤独立，易于维护
- 可复用：链可以在不同场景复用
- 可组合：可以组合多个链形成复杂流程

### 5.2.3 记忆管理：给AI装一个"记忆系统"

什么是记忆管理?

记忆管理就像给AI装一个"记忆系统"，让它能够记住之前的对话。就像人能够记住之前说过的话一样。

生活中的例子：

想象你和朋友聊天：

- 没有记忆：每次说话朋友都忘记之前说过什么，聊天没法继续
- 有记忆：朋友记得之前说过的话，聊天可以连续进行

AI也是这样：

- 没有记忆管理：每次对话都是独立的，AI不知道之前说过什么
- 有记忆管理：AI能记住之前的对话，可以进行连续的多轮对话

**LangChain**提供的记忆类型：

就像不同的记忆方式：

1. **ConversationBufferMemory**：保存完整对话历史
    - 就像记住所有对话细节（占用空间大，但信息完整）
2. **ConversationSummaryMemory**：保存对话摘要（节省token）
    - 就像记住对话的要点（占用空间小，但信息精简）
3. **ConversationBufferWindowMemory**：只保存最近N轮对话
    - 就像只记住最近几段对话（平衡空间和信息）

完整代码示例：

```
"""
LangChain记忆管理示例
"""

import os
import sys
from langchain_openai import ChatOpenAI
from langchain.memory import ConversationBufferMemory, ConversationSummaryMemory
from langchain.chains import ConversationChain
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

try:
    from config import DEEPSEEK_API_KEY, DEEPSEEK_BASE_URL, DEEPSEEK_MODEL
except ImportError:
    DEEPSEEK_API_KEY = os.getenv("DEEPSEEK_API_KEY", "")
    DEEPSEEK_BASE_URL = "https://api.deepseek.com"
    DEEPSEEK_MODEL = "deepseek-chat"

def buffer_memory_demo():
    """缓冲区记忆演示"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return
```

```
    llm = ChatOpenAI(
        model=DEEPSEEK_MODEL,
        openai_api_key=DEEPSEEK_API_KEY,
        openai_api_base=DEEPSEEK_BASE_URL,
        temperature=0.7
    )

    # 创建缓冲区记忆
    # return_messages=True：返回消息对象而不是字符串
    memory = ConversationBufferMemory(return_messages=True)

    # 创建对话链的提示模板
    # MessagesPlaceholder：占位符，用于插入历史消息
    prompt = ChatPromptTemplate.from_messages([
        ("system", "你是一个友好的AI助手。"),
        MessagesPlaceholder(variable_name="history"),  # 历史消息占位符
        ("human", "{input}")  # 当前输入
    ])

    # 创建对话链
    # ConversationChain：自动管理对话历史
    chain = ConversationChain(
        llm=llm,
        memory=memory,
        prompt=prompt,
        verbose=True  # 显示执行过程
    )

    print("=" * 50)
    print("LangChain 缓冲区记忆演示")
    print("=" * 50)

    # 第一轮对话
    print("\n第一轮对话:")
    response1 = chain.predict(input="我的名字是张三，我喜欢编程")
    print(f"用户: 我的名字是张三，我喜欢编程")
    print(f"助手: {response1}")

    # 第二轮对话（测试记忆）
    print("\n第二轮对话（测试是否记住名字）:")
    response2 = chain.predict(input="我刚才说我叫什么名字？")
    print(f"用户: 我刚才说我叫什么名字？")
    print(f"助手: {response2}")

    # 显示记忆内容
    print("\n当前记忆内容:")
    print("-" * 50)
    print(memory.buffer)
    print("-" * 50)

def summary_memory_demo():
```

```
"""摘要记忆演示"""
if not DEEPSEEK_API_KEY:
    print("错误：未配置API密钥")
    return

llm = ChatOpenAI(
    model=DEEPSEEK_MODEL,
    openai_api_key=DEEPSEEK_API_KEY,
    openai_api_base=DEEPSEEK_BASE_URL,
    temperature=0.7
)

# 创建摘要记忆（适合长对话）
# ConversationSummaryMemory：自动生成对话摘要，节省token
memory = ConversationSummaryMemory(
    llm=llm,  # 需要LLM来生成摘要
    return_messages=True
)

prompt = ChatPromptTemplate.from_messages([
    ("system", "你是一个专业的Python编程助手。"),
    MessagesPlaceholder(variable_name="history"),
    ("human", "{input}")
])

chain = ConversationChain(
    llm=llm,
    memory=memory,
    prompt=prompt,
    verbose=False
)

print("\n" + "=" * 50)
print("LangChain 摘要记忆演示")
print("=" * 50)
print("（摘要记忆适合长对话，可以节省token）\n")

# 多轮对话
conversations = [
    "我想学习Python装饰器",
    "能给我一个实际例子吗？",
    "这个例子中，@timer的作用是什么？"
]

for i, user_input in enumerate(conversations, 1):
    print(f"\n对话 {i}:")
    print(f"用户: {user_input}")
    response = chain.predict(input=user_input)
    print(f"助手: {response}")

# 显示记忆摘要
print("\n记忆摘要:")
```

```
    print("-" * 50)
    if hasattr(memory, 'moving_summary_buffer'):
        print(memory.moving_summary_buffer)
    print("-" * 50)

if __name__ == "__main__":
    print("=" * 50)
    print("LangChain 记忆管理示例")
    print("=" * 50)
    print()

    try:
        # 缓冲区记忆
        buffer_memory_demo()

        # 摘要记忆
        summary_memory_demo()

        print("\n" + "=" * 50)
        print("所有示例完成！")
        print("=" * 50)
        print("\n记忆类型说明：")
        print("1. ConversationBufferMemory: 保存完整对话历史")
        print("2. ConversationSummaryMemory: 保存对话摘要，节省token")
        print("3. ConversationBufferWindowMemory: 只保存最近N轮对话")

    except Exception as e:
        print(f"错误: {e}")
        import traceback
        traceback.print_exc()
```

运行方法：

```
python chapter05/langchain_memory.py
```

### 5.2.4 RAG应用：给AI装一个"知识库"

什么是**RAG**?

RAG（Retrieval-Augmented Generation，检索增强生成）就像给AI装一个"知识库"。AI不仅能用自己的知识回答问题，还能从你提供的文档中查找相关信息。

生活中的例子：

想象你在咨询一个专家：

- 不用**RAG**：专家只能用自己的知识回答（可能不够准确或不够新）
- 用**RAG**：专家会先查资料，然后基于资料回答（更准确、更新）

AI也是这样：

- 不用**RAG**：AI只能用自己的训练数据回答
- 用**RAG**：AI会先从你的文档中查找相关信息，然后基于这些信息回答

**RAG**的工作流程：

就像图书馆的工作流程：

1. 文档加载：把书放进图书馆（加载文档）
2. 文本分割：把书分成章节（将文档分割成小块）
3. 生成向量：给每章做索引（将文本转换为向量）
4. 存储向量：把索引存起来（存储到向量数据库）
5. 检索相关文档：根据问题找相关章节（根据问题检索相关文档）
6. 生成回答：基于找到的章节回答问题（基于检索到的文档生成回答）

**RAG**的优势：

- 可以基于特定文档回答问题：就像专家可以基于最新资料回答
- 减少模型幻觉：就像专家基于资料回答，不会"编造"
- 支持知识更新：就像可以随时更新资料库

简化版**RAG**示例：

```
"""
LangChain RAG应用示例（简化版）
注意：完整RAG需要向量数据库，这里提供简化示例
"""

import os
import sys
from langchain_openai import ChatOpenAI
from langchain.prompts import PromptTemplate

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

try:
    from config import DEEPSEEK_API_KEY, DEEPSEEK_BASE_URL, DEEPSEEK_MODEL
except ImportError:
    DEEPSEEK_API_KEY = os.getenv("DEEPSEEK_API_KEY", "")
    DEEPSEEK_BASE_URL = "https://api.deepseek.com"
    DEEPSEEK_MODEL = "deepseek-chat"

def simple_rag_demo():
    """简单的RAG演示（不使用向量数据库）"""
    if not DEEPSEEK_API_KEY:
        print("错误：未配置API密钥")
        return

    llm = ChatOpenAI(
        model=DEEPSEEK_MODEL,
        openai_api_key=DEEPSEEK_API_KEY,
        openai_api_base=DEEPSEEK_BASE_URL,
        temperature=0.7
    )

    # 示例文档（实际应用中从文件或数据库加载）
    documents = [
```

```
    "Python是一种高级编程语言，由Guido van Rossum创建。",
    "Python支持多种编程范式，包括面向对象、函数式和过程式编程。",
    "Python有丰富的标准库和第三方库，如NumPy、Pandas、Django等。",
    "Python的语法简洁清晰，适合初学者学习。",
    "Python广泛应用于Web开发、数据科学、人工智能等领域。"
]

print("=" * 50)
print("LangChain RAG 应用演示（简化版）")
print("=" * 50)
print("\n文档库内容:")
for i, doc in enumerate(documents, 1):
    print(f"{i}. {doc}")

# 简单的关键词匹配检索（实际应使用向量检索）
def simple_retrieve(query, docs, top_k=2):
    """简单的关键词检索"""
    query_lower = query.lower()
    scored_docs = []
    for doc in docs:
        # 计算匹配的关键词数量
        score = sum(1 for word in query_lower.split() if word in doc.lower())
        if score > 0:
            scored_docs.append((doc, score))
    # 按分数排序
    scored_docs.sort(key=lambda x: x[1], reverse=True)
    return [doc for doc, _ in scored_docs[:top_k]]

# 创建RAG提示模板
template = """基于以下上下文信息回答问题。如果上下文中没有相关信息，请说明。

上下文:
{context}

问题：{question}

回答："""

prompt = PromptTemplate(
    template=template,
    input_variables=["context", "question"]
)

# 模拟RAG流程
questions = [
    "Python是谁创建的？",
    "Python有哪些应用领域？"
]

print("\n" + "=" * 50)
print("RAG问答演示")
print("=" * 50)
```

```
    for question in questions:
        print(f"\n问题: {question}")

        # 检索相关文档
        retrieved_docs = simple_retrieve(question, documents)
        context = "\n".join(retrieved_docs)

        print(f"检索到的上下文: {context}")

        # 生成回答
        formatted_prompt = prompt.format(context=context, question=question)
        response = llm.invoke(formatted_prompt)

        print(f"回答: {response.content}")

if __name__ == "__main__":
    print("=" * 50)
    print("LangChain RAG 应用示例")
    print("=" * 50)
    print()

    try:
        # 简单RAG演示
        simple_rag_demo()

        print("\n" + "=" * 50)
        print("RAG应用说明")
        print("=" * 50)
        print("""
RAG（检索增强生成）流程：
1. 文档加载和预处理
2. 文本分割成块
3. 生成嵌入向量
4. 存储到向量数据库
5. 用户查询时检索相关文档
6. 将检索到的文档作为上下文生成回答

优势：
- 可以基于特定文档回答问题
- 减少模型幻觉
- 支持知识更新

注意：完整RAG需要向量数据库（如FAISS、Pinecone等）
        """)

    except Exception as e:
        print(f"错误: {e}")
        import traceback
        traceback.print_exc()
```

运行方法：

```
python chapter05/langchain_rag.py
```

完整**RAG**需要的组件：

1. 文档加载器：从各种来源加载文档
2. 文本分割器：将文档分割成块
3. 嵌入模型：生成文本向量
4. 向量数据库：存储和检索向量
5. 检索器：检索相关文档

## 5.3 LangChain插件

什么是**LangChain**插件?

LangChain支持多种插件和集成，扩展功能。

常用插件：

1. 向量数据库
    - FAISS：本地向量数据库
    - Pinecone：云端向量数据库
    - Chroma：开源向量数据库
2. 文档加载器
    - PDF加载器
    - 网页加载器
    - 数据库加载器
3. 文本分割器
    - 字符分割
    - 标记分割
    - 语义分割
4. 工具集成
    - 搜索引擎
    - 计算器
    - API调用

安装示例：

```
# 安装FAISS向量数据库
pip install faiss-cpu

# 安装PDF加载器
pip install pypdf

# 安装网页加载器
pip install beautifulsoup4
```

## 5.4 小结

本章要点：

1. 了解了LangChain框架的基本概念
2. 学会了LangChain的基本使用
3. 掌握了链式调用的方法
4. 学会了记忆管理
5. 了解了RAG应用的基本原理

**LangChain vs** 直接调用**API**：

| 特性 | 直接调用**API** | **LangChain** |
| --- | --- | --- |
| 简单性 | ☆☆☆☆☆ | ☆☆☆ |
| 功能 | ☆☆☆ | ☆☆☆☆☆ |
| 扩展性 | ☆☆ | ☆☆☆☆☆ |
| 适用场景 | 简单应用 | 复杂应用 |

什么时候用**LangChain**?

- 需要复杂的工作流
- 需要记忆管理
- 需要RAG功能
- 需要集成多个工具

什么时候直接调用**API**?

- 简单的对话应用
- 代码生成工具
- 快速原型开发

下一步：

恭喜！你已经完成了整本书的学习。现在可以：

1. 回顾各章节内容
2. 完成练习项目
3. 构建自己的应用

练习：

1. 使用LangChain构建一个带记忆的聊天机器人
2. 实现一个简单的RAG应用
3. 组合多个链，构建复杂的工作流