

大模型编程实践与提示工程

(Programming Large Language Models with Azure Open AI:
Conversational programming and prompt engineering with LLMs)

Francesco Esposito 著

周靖 (<https://bookzhou.com>) 译



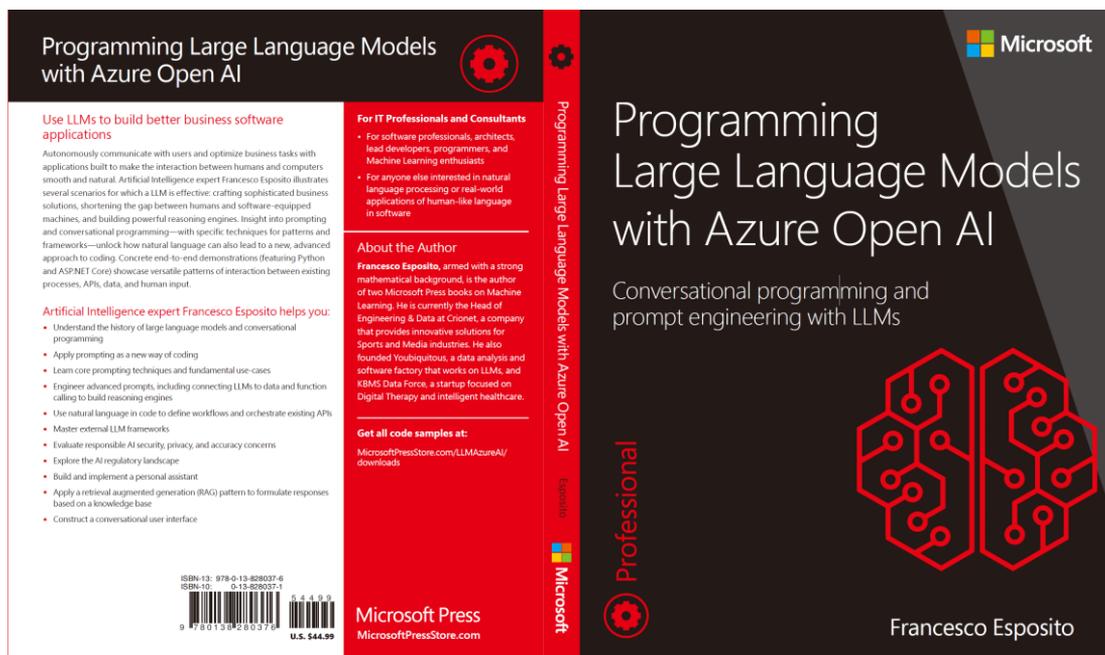
中文试读版 1-4 章，翻译原稿，仅供参考

配套资源和试读下载：[ys168 网盘](#)>> [百度网盘](#)>> [GitHub 项目](#)>>

[访问中文版主页，获取最新资讯](#)

清华大学出版社

北京



使用 LLM 构建更好的商业软件应用

通过自主与用户沟通并优化业务任务，人机交互变得更加流畅自然。人工智能专家 Francesco Esposito 展示了多个适合使用 LLM 的场景：打造高级的商业解决方案、缩短人类与机器之间的距离以及建立强大的推理引擎。本书深入探讨了提示工程和对话式编程，演示了具体的模式和框架技术，最终揭示了如何通过自然语言解锁一种新的、先进的编码方式。使用 Python 和 ASP.NET Core，本书演示了在人机交互、API、数据和人类输入之间创建一致交互模式的各种实例。

本书帮助您：

- 了解大语言模型（LLM）及对话式编程的历史
- 探索提示工程这一新编码方式
- 学习核心提示技术和基本使用场景
- 设计高级提示，包括连接 LLM 以生成函数调用来构建推理引擎
- 使用自然语言来定义 workflows 并协调（编排）现有的 API
- 掌握外部 LLM 框架
- 评估负责任 AI 的安全、隐私和准确性问题
- 探索 AI 的监管环境

- 构建和实现个人助手
- 应用知识库检索增强生成（RAG）模式来生成响应
- 构建对话用户界面

面向 IT 专家和顾问：

- 面向软件专家、架构师、首席开发人员、程序员以及机器学习爱好者。
- 也面向任何对自然语言处理或者在软件中引入 AI 功能感兴趣的人士。

作者简介



Francesco Esposito 拥有深厚的数学背景，曾著有两本关于机器学习的图书。他现任 Crionet 公司的工程与数据主管，为体育和媒体行业提供创新解决方案。他还创立了 Youbiquitous，一家致力于 LLM、数据分析和软件工厂的公司，以及 KBMS Data Force，一家专注于数字疗法和智能医疗的初创公司。

译者简介



周靖 (<https://bookzhou.com>) 是资深软件开发专家，Microsoft MVP，拥有超过 20 年的 C++ 和 C# 编程经验。随着 AI 时代的到来，他也喜欢上了使用 Python 进行大语言模型的开发。周靖不仅是一位技术精湛的程序员，还是一个热情的烹饪爱好者，特别钟爱云贵川菜系，深受喜欢教大家做饭的王刚老师的启发和影响。

在技术领域，周靖致力于 IT 知识的传播，他的理想是让更多的人轻松掌握编程技能，消除编程带来的恐惧感。为此，他翻译过许多经典的计算机著作，还陆续发表过一些文章。他的译作包括：

- [《机器学习与人工智能实战》](#)
- [《学习 C++20》](#)（目前第 3 次印刷）
- [《C++入门经典》](#)（Savitch 系列，销量超过 2 万册）
- [《代码大全 2》](#) 纪念版（销量超过 3 万册）
- [《Windows 核心编程》](#)（全行业畅销书，销量超过 6 万册）
- [《深入 CLR》](#)（原《CLR via C#》）（销量超过 5 万册）
- [《Visual C#从入门到精通》](#) 系列（销量超过 6 万册）
- [《C#本质论》](#) 系列（销量超过 5 万册）
- ...

除了写作和翻译，周靖还利用业余时间，在出版社的邀请下，审校和重译了一些重点书籍，其中不乏 [《实时渲染 4》](#)（RTR4）等经典作品。

下载本书源代码：译者主页 (<https://bookzhou.com>) 或 GitHub 项目 (<https://github.com/transbot/ProgrammingLLMChinese>)。

献给 AI。

不把这本书献给您，将是对您的亵渎。

目录概览

第 1 章 大语言模型的起源与分析

第 2 章 核心提示学习技术

第 3 章 打造高级学习提示

第 4 章 使用语言框架

第 5 章 安全性、隐私和准确性问题

第 6 章 构建个人助手

第 7 章 与自己的数据对话

第 8 章 对话式 UI

附录：大语言模型内部工作原理揭密

目录

致谢	12
译者序	14
前言	16
<i>本书适合的读者</i>	17
<i>预备知识</i>	17
<i>本书不适合的读者</i>	17
<i>本书的组织结构</i>	17
<i>下载配套资源</i>	18
<i>勘误、更新与图书支持</i>	18
<i>保持联系</i>	19
第 1 章 大语言模型的起源与分析	20
<i>LLM 综述</i>	20
LLM 的历史.....	20
工作原理.....	26
商业应用.....	37
<i>对话式编程</i>	38
自然语言的崛起.....	38
LLM 拓扑结构.....	40
未来展望.....	43
小结.....	49
第 2 章 核心提示学习技术	50
<i>什么是提示工程</i>	50
提示概述.....	50

改变输出的其他方式.....	54
设置代码执行.....	57
<i>基本技术</i>	61
零样本场景.....	62
少样本场景.....	63
思维链场景.....	68
<i>基本使用场景</i>	72
聊天机器人.....	73
收集信息.....	73
翻译	77
<i>LLM 的局限性</i>	78
小结.....	79
第 3 章 打造高级学习提示	80
<i>超越提示工程</i>	80
合并不同的部分.....	81
微调	83
<i>函数调用</i>	86
自定义函数调用.....	86
OpenAI 风格	90
<i>与(单独的)数据对话</i>	95
将数据连接到 LLM.....	95
嵌入	96
向量存储.....	101
检索增强生成(RAG).....	105

小结.....	110
第 4 章 使用语言框架.....	111
对编排器的需求.....	111
跨框架概念.....	112
需要考虑的重点.....	119
LangChain.....	120
模型、提示模板和链.....	121
代理	130
数据连接.....	139
Microsoft Semantic Kernel	145
插件	146
数据与规划器.....	152
Microsoft Guidance	158
配置	159
主要特性.....	161
小结.....	165
第 5 章 安全、隐私和准确性问题.....	167
概述.....	167
负责任 AI	167
红队测试.....	168
滥用与内容过滤.....	169
幻觉与性能.....	170
偏见和公平性.....	171
安全性和隐私.....	171
安全性.....	172

隐私保护.....	176
<i>评估和内容过滤</i>	181
评估	181
内容过滤.....	185
小结.....	194
第 6 章 构建个人助手.....	195
<i>聊天机器人 Web 应用概览</i>	195
愿景	195
技术栈.....	196
<i>项目</i>	197
设置 LLM.....	197
设置项目.....	200
与 LLM 集成.....	203
小结.....	218
第 7 章 与自己的数据对话.....	219
<i>概述</i>	219
愿景	219
技术栈.....	220
<i>Streamlit</i>	220
Streamlit 简介	220
主要 UI 特性.....	221
生产时的优缺点.....	223
<i>项目</i>	224
设置项目和基本 UI.....	225

数据准备.....	228
与 LLM 集成.....	233
进阶内容.....	238
RAG 与微调.....	238
可能的扩展.....	240
小结.....	241
第 8 章 对话式 UI	242
概述.....	242
愿景	242
技术栈.....	244
项目.....	246
Minimal API 设置.....	246
OpenAPI	248
与 LLM 集成.....	250
可能的扩展.....	257
小结.....	258
附录 大语言模型内部工作原理揭密	259
概率的作用.....	259
启发式方法.....	259
人工神经元.....	261
GPT 的情况.....	268
transformer 与注意力机制	269
训练与新兴技术.....	274

致谢

2023 年春，当我和老爸聊到 Azure OpenAI 有多“酷”时，他的回应颇令我讶异：“何不著书立说？”言辞之恳切，仿若对我有莫大的信心。继而一句“你准备好了吗？”更是让我坚定了决心。后面的事情就是“走流程”了，微软出版社的洛雷塔·耶茨（Loretta Yates）欣然接纳了我的提案。本书的故事，便在 2023 年的 6 月悄然萌芽。

过去十年间，人工智能（AI）一直是炙手可热的话题，新一代的大语言模型（LLM）的出现更是将其推向了高潮。越来越多的人开始使用这些技术，从而催生了更多的创新、机遇和技术变革。如今，与 AI 相关的一切都在发生着翻天覆地的变化。

故而，您手中这本关于 AI 与 LLM 的书，断然无法成为终极指南。因技术迭代之快令人咋舌，且任何书籍均属于知识在特定时刻的瞬间定格，只能算是一种近似的记录。近似难免引发某种程度的不满足，而正是这不满足，驱使我们迎接新的挑战。我愿自身能长久处于这样的求索与不满足中，同时也期待未来数年，能有更多机会立于台上，展示为这一权威出版社撰写的书籍，那将是对我的一大慰藉。

首先，我对自 2023 年 5 月以来所有第一次约会对象感到有些亏欠，毕竟他们不得耐着性子听完我至少半小时我关于 LLM 及其新奇的 transformer 的独白。^①

真正的感谢是私下的事情，但这里必须公开感谢和我共同撰写附录的玛蒂娜（Martina），她总能以恰如其分的话语促我进步。这里感谢她，是为了兑现她知道的一个小小承诺。玛蒂娜，感谢您，以超凡之姿，丰富了我的世界。

另外，感谢吉安弗兰科（Gianfranco），是他教会我即使要用大嗓门来表达不满，也要积极参与讨论并表达自己的意见。我现在已经完全不怯场了，晓得不懂就问，因为最坏的情况也不过是被拒绝。现在，每当我参与讨论时，我都会想起您。

还要感谢马泰奥（Matteo）、卢西亚诺（Luciano）、加布里埃尔（Gabriele）、菲利波（Filippo）、达尼埃莱（Daniele）、里卡多（Riccardo）、马可（Marco）、雅科波（Jacopo）、西莫内（Simone）、弗朗切斯科（Francesco）和亚历山德拉（Alessia），在我偶有的（希望不算频繁）困境中，你们始终和我站在一起，给予支持。我也对亚历山德罗（Alessandro）、安东尼诺（Antonino）、莎拉（Sara）、安德烈亚（Andrea）和克里斯蒂安（Cristian）表示

^① 译注：Transformer（直译为“转换器”或“变换器”）是一种用于自然语言处理（NLP）任务的神经网络架构。它由 Vaswani 等人在 2017 年提出，并在论文《Attention is All You Need》中详细介绍。

Transformer 模型通过一种称为“自注意力”（self-attention）机制来处理输入数据，能够有效捕捉句子中不同单词之间的关系。但是，它在日常生活中也有“变压器”的意思。因此，作者在这里需要向听众解释两者的不同。

深厚的谢意，你们容忍了我，因为我花了好多时间在这本书上，而没有像正常的 25 岁年轻人那样经常跑出去享受生活。

感谢妈妈和米凯拉（Michela），不管我写不写书，你们对我的爱始终如一。家族里的老长辈们，你们的慈爱是我坚实的后盾。乔尔吉奥（Giorgio）、盖塔诺（Gaetano）、维托（Vito）和罗伯托（Roberto）每日都在助力我成长。埃利奥（Elio），是您教会我如何搭配衣服，让我以多彩的视角审视自我。

至于我的爸爸，迪诺（Dino），他从未停止教我新东西——例如，如何通过做自己喜欢的事情赚钱，以及成为这本书的技术编辑等。无论是父亲还是编辑，我都要感谢您。您让我想起一首您熟知的歌“Figlio, figlio, figlio”^①。

除洛雷塔外，本书的问世亦离不开沙罗夫（Shourav）、凯特（Kate）和丹（Dan）的辛勤工作。感谢你们的耐心，感谢你们对我的深厚信任。

此书，截至目前，乃我笔下最佳，直至下一本的诞生！

——弗朗西斯科·埃斯波西托（Francesco Esposito）

^① 译注：这是意大利知名歌手和作曲家罗伯托·维奇奥尼（Roberto Vecchioni）的一首经典歌，表达了父亲对儿子的深情关爱和担忧，希望他能在世界的航程中找到自己的方向，并面对生活中的挑战 and 困难，展现出坚韧和勇气。

译者序

AI 正以不可阻挡的势头席卷全球，深刻影响着各行各业。“顺之者昌，逆之者衰”，面对这股新的技术浪潮，我们的唯一选择就是积极适应并参与其中，与时代同频共振。因此，除了找到一个合适的切入点（例如，在现有应用中引入 AI 功能），一本优质的参考书也是必不可少的。

在此，我荣幸地推荐这本《大模型编程实践与提示工程》。这并不是试图涵盖所有 AI 领域的百科全书。它不会深入讲解复杂的数学原理，也不会详尽列举编程语法。相反，它的目标是帮助读者快速上手，并将 AI 技术应用于实际项目之中。

非常荣幸清华大学出版社将这本书的翻译工作交给我。然而，翻译工作就如同接力赛跑中的最后一棒。现在，接力棒传到了译者手中，但不能仅仅简单地把它“翻译”出来了事，还必须根据当前最新的发展做出适当的调整，并与作者保持紧密沟通。最终成果是团队协作的结晶，每个环节都紧密相连，确保中文版既能反映当前现实，又能完美衔接原作。需要检查作者是否有遗漏之处，是否需要补充内容，以及是否需要调整以适应中文环境。

尤为重要，书中使用的示例是否依然与时俱进，是否需要更新以符合最新的实际情况。为了给中文版读者提供最佳的阅读体验，我对本书的代码进行了更新，使其适应当前各种库的最新版本。为此，我在自己的网站（<https://bookzhou.com>）上提供了本书修改后的中文版代码供读者参考。

需要注意的是，当前的生成式 AI 虽然在努力提高输出的准确性，但由于 transformer 架构固有的局限性，仍会出现输出不准确的情况。尽管它们学习了大量的数据，但它们实际上并未掌握真正的知识，而只是学会了如何重新组合语料库中的信息。当前的生成式 AI 主要依靠算法和算力来预测下一个词，形成新的“排列组合”。然而，似乎已经接近了一个瓶颈，触及了现有理论和技术的天花板。甚至有可能目前投入的计算资源已经超过了实际需求。

我个人认为，生成式 AI 在自然语言处理（NLP）领域表现出一定的优势，但在逻辑推理方面仍存在不足。这一点可以从各大模型（包括 GPT-4 和 Qwen 2-72B 等）在解答 2024 年高考题目时的表现中看出。从单科成绩来看，这些模型在数学方面普遍表现不佳，得分率均未超过 50%，呈现出明显的偏科现象。因此，当前的生成式 AI 更适合应用于容错性较高的场景。

尽管如此，不可否认的是，在某些对容错性要求不高的应用场景中，生成式 AI 的表现仍然令人印象深刻。因此，在这样的背景下，事先对 AI 的应用加以限制，并在事后对结果进行验证，就显得尤为重要。这也是本书的一个重要目标。

在翻译本书的过程中，我非常高兴得到了女儿周子衿（Ava Zhou）的全力支持。我们共同探讨了书中的一些主题，并一起学习了如何将 AI 应用于自己的实际工作。她还帮助我解

决了不少疑惑，澄清了我平时没有想到的一些问题。她让我深刻地认识到，在什么时候， $1+1$ 可以大于 2。

更值得一提的是，她下定决心开发自己的一个 Unity 游戏，并在过程中借助 AI 的力量。在此便不得不推荐她翻译的两本书：[《Unity 2D 游戏开发》](#)和[《Unity 3D 游戏开发》](#)，均由清华大学出版社出版。

和往常一样，祝您编程愉快！

——周靖，2024

前言

这是我关于**人工智能**（Artificial Intelligence, AI^①）的第三本书，也是我独自完成的第一本，没有合著者的参与。这三本书的出版顺序真实反映了我的学习路径。是对 AI 的深切渴望驱使我前行，远不止于商业上的考量。第一本书于 2020 年出版，介绍了**机器学习**（Machine Learning, ML）背后的数学知识。正是这些数学上的知识，才使得数据分类和及时预测成为可能。第二本书则聚焦于微软的 ML.NET 框架，^②探讨了具体的应用场景，也就是如何让那些复杂的算法处理海量的数据，并通过熟悉的网页前端图表隐藏其复杂性，使之运行得既实用又高效。

再之后，ChatGPT 横空出世！

对于像 ChatGPT 这样的惊人应用，其背后的基础技术称为**大语言模型**（Large Language Model, LLM），而 LLM 正是这第三本书的主题。LLM 为 AI 增加了一个关键能力：生成内容，而非仅仅是分类和预测。LLM 代表了一种范式转变，提高了人与计算机之间的沟通水平，打开了几十年来我们只能梦想的新应用的大门。

几十年来，我们一直梦想着这些应用。大量文学作品和电影呈现了各种超级计算机，它们能处理各种数据并产生人类可理解的结果。一个极为流行的例子是电影《2001 太空漫游》（1968 年）中安装在发现（Discovery）号飞船上的 HAL 9000 计算机。另一个例子是贾维斯（JARVIS, Just A Rather Very Intelligent System），它是托尼·斯塔克（Tony Stark）家中的助理计算机，出现在《钢铁侠》和其他漫威电影中。还有一个著名的例子是电影《流浪地球》中的 MOSS（原名 550W），它坚定执行延续人类文明的使命，能在最短的时间内做出最正确的决定，是趋于完美的智慧体。

在这些文学作品和电影中，人类角色常常幻想自己能简单地“将数据加载到计算机中”，无论是纸质文件、数字文件还是媒体内容。接着，机器会自主解析这些内容，从中学习，并使用自然语言与人类交流。当然，那些超级计算机只是创作者构想的产物，属于科幻范畴。而今，借助 LLM，我们已经能设计和构建具体的应用，不仅使人与计算机的交互变得更加流畅和自然，而且将昔日的科学幻想变成了现实。

本书介绍了如何利用与 ChatGPT 背后相同的引擎技术来构建应用程序，使程序能自主与用户交流，并仅凭简单的文本提示来执行业务操作。这一切既不夸大也不缩水——现在真的可以做到“指令一出，AI 即行”！

^① 译注：不是 Apple Intelligence。

^② 《跨平台机器学习——ML.NET 架构及应用编程》，清华大学出版社 2023 年出版。详情请访问 <https://bookzhou.com/2023/03/27/567/>。

本书适合的读者

本书面向软件架构师、资深开发人员以及有一定编程经验的个人。熟悉 Python 和 C#（用于 ASP.NET Core）语言的读者会觉得本书的内容尤其容易理解和实用。在众多可能受益于本书的软件专业人员中，我特别推荐对机器学习（尤其是大语言模型）感兴趣的读者阅读本书。此外，对云服务（特别是微软 Azure）感兴趣的云计算和 IT 专业人员，以及对复杂、真实世界中的类人语言软件应用感兴趣的人员也能从本书获得许多感悟。虽然本书主要关注的是微软 Azure 平台上的服务，但所涉及的概念很容易延伸到其他类似平台。毕竟，使用大语言模型无非就是调用一系列 API，而 API 的具体实现完全独立于底层平台。

总之，本书面向多样化的读者群体，包括程序员、机器学习爱好者、云计算专业人员以及对自然语言处理感兴趣的人士等，并具体解释了如何利用 Azure 服务来编写 LLM 应用。

预备知识

为了从本书发掘出最大价值，读者需要具备两方面的预备知识：其一是掌握基本的编程概念，其二是了解机器学习的一些基础知识。此外，如果有相关编程语言和框架（例如，Python 或 ASP.NET Core）的实际工作经验，同时理解经典自然语言处理在业务领域中的重要性，那么将会大有裨益。总的来说，要想全面吃透本书，推荐同时具备编程知识、机器学习意识以及对自然语言处理（NLP）的理解。

本书不适合的读者

如果想找一本“大部头”的参考书来详细了解如何使用特定的模式或框架，那么本书可能会令您失望了。虽然本书讨论了流行的框架（例如，LangChain 和 Semantic Kernel）以及 API（例如，OpenAI 和 Azure OpenAI）的某些高级方面，但它并不是关于这些主题的一本完整的编程参考书。本书的重点是在真正适合 LLM 的业务领域中构建有用的、能直接发挥效益的应用程序。

本书的组织结构

本书探讨了如何利用现有的各种 LLM 开发多领域的商业应用。每个 LLM 本质上都是一个基于大量文本数据**预训练**的 ML 模型，它能理解和生成类似人类语言的内容。为了传达关于这些模型的知识，本书聚焦于以下三个核心方面：

- 前三章深入探讨了 LLM 适用的场景，并介绍了构建复杂解决方案的一些基本工具。

这些章的重点在于对话式编程和提示工程，这是一种先进而结构化的新式编码方法。是的，你没有看错，现在和 AI 聊天，就能完成各种各样的任务。在苹果公司 2024 年发布的新版 iOS 中，已经有大量应用集成了这种功能。

- 接下来的两章着重于模式、框架和技术，它们具体教你如何释放对话式编程的潜力。这涉及到在代码中使用自然语言来定义工作流，同时由 LLM 的应用来协调现有的 API。
- 最后三章演示了一些具体的应用，它们使用 Python 和 ASP.NET Core 来实现。通过这些例子，可以体会到逻辑、数据和现有业务过程之间日益复杂的交互。在第一个例子中，将学习如何从电子邮件提取文本，并为回复撰写一份合适的草稿。在第二个例子中，将应用一种**检索增强生成**（Retrieval Augmented Generation, RAG）模式，根据私有文档的内容为问题制定答复。最后，在第三个演示中，将学习如何构建一个酒店预订应用，其中包含一个**聊天机器人**（chatbot），它通过对话界面来确定用户需求（日期、房间偏好和预算等），并根据底层的系统状态来无缝地完成（或拒绝）预订。注意，最后一个例子只完成了 API 的开发，不会用到任何固定的**用户界面**（User Interface, UI）元素或格式化好的数据输入控件。

下载配套资源

本书配套提供了中英文版本的完整源代码。英文版代码可以在 GitHub 上获取，网址是 <https://github.com/Youbiquitous/programming-llm>。

中文版读者则可以访问 <https://bookzhou.com> 来获取更完善的、修改过的中文版资源。

勘误、更新与图书支持

我们已尽最大努力确保本书及其配套内容的准确性。可以在以下网址查看本书（英文版）的更新（包括勘误）：

MicrosoftPressStore.com/LLMAzureAI/errata

如果发现未列出的错误，请通过同一页面提交给我们。

有关其他图书支持和信息，请访问 MicrosoftPressStore.com/Support。

请注意，上述地址不提供微软软硬件产品支持。如需获取这些方面的帮助，请访问 <https://support.microsoft.com>。

要获取本书中文版的配套资源和勘误，请访问 <https://bookzhou.com>。

保持联系

沟通不断，交流不停！请通过 X / Twitter 来保持联系：<http://twitter.com/MicrosoftPress>。

第 1 章 大语言模型的起源与分析

吸引读者阅读一本书从来都不是一件容易的事。如果是小说，那么必须让他们相信这是一个美好的故事；如果是技术书籍，则必须让他们相信能从中学到一些东西。就本书来说，我们将尝试学习一些东西。

在过去两年里，**生成式人工智能**（Generative AI）上了热搜。它是人工智能（Artificial Intelligence, AI）的一个特定领域，专注于创建能自主生成新的、原创内容的系统。**大语言模型**（Large Language Model, LLM）就是一种生成式 AI，能根据给定的输入生成人类可以理解的文本。LLM 的例子包括 GPT、Gemini、通义千问、文心一言等。

LLM 的迅速采用正在引发整个编程领域思维模式的转变。本章将讨论这种转变、转变的原因及其前景。其中最重要的是**对话式编程**，即使用自然语言而不是代码来解释自己想要达成的目标。在可以预见的将来，这种形式的编程会变得非常普遍。

但也不要高兴得太早。很快就会讲到，用自然语言来解释想要达成的目标经常会和手动写代码一样困难。

本章涵盖了本书其他地方没有涉及的主题。但是，也没有必要阅读每一节或遵循严格的顺序。可以选择阅读自己认为必要或有趣的。我预计你在读完本书最后一章后，会回头阅读本章的某些部分。

LLM 综述

为了驾驭 LLM，开发人员或管理人员一定要了解生成式 AI 的起源，并理解它与预测式 AI 的区别。本章的一个关键目标是帮助您理解 LLM 的训练过程，以及它与您手头上的业务的相关性。至于复杂的数学细节，将留到附录中作为选读。

我们的旅程将从 AI 的历史一直延伸到 LLM 的基础知识，包括它们的训练、推理以及多模态模型的诞生。至于商业领域，还将重点讨论生成式 AI 和文本模型当前的一些热门应用。

本章不打算深入任何细节。相反，我们的重点是提供充分的背景信息，以填补可能的知识空白，并帮助您体会 LLM 演变与实现过程中的各种复杂性。

LLM 的历史

LLM 的发展与传统 AI（通常称为**预测式 AI**）的历史和**自然语言处理**（Natural Language Processing, NLP）领域相交织。NLP 涉及**自然语言理解**（Natural Language Understanding, NLU）和**自然语言生成**（Natural Language Generation, NLG）这两个方面。其中，NLU 试

图将人类语言简化为结构化的本体（structured ontology）^①，而 NLG 旨在生成人类可理解的文本。

LLM 是一种生成式人工智能，专注于根据某种输入来生成文本。这些输入通常采用书面文本（称为“提示”）的形式，但现在已升级到包括图像、视频和音频在内的多模态输入。从表面上看，大多数 LLM 可以被视为“自动补全（完成）”的一种非常高级的形式，因为它们能生成下一个合理的单词/字。尽管专注于文本补全，但 LLM 是以模拟人类推理的方式来实现的，这使它们能执行各种各样复杂的任务，包括情感分析、归纳总结、翻译、实体/意图识别、结构化信息提取以及文档生成等等。

LLM 代表了人类古老愿望的自然延伸，即构建自动机（当代机器人的先驱），并赋予它们一定程度的推理和语言能力。LLM 可以被视为这种自动机的大脑，能对外部输入做出响应。

AI 源起

现代软件，以及作为当前最活跃的一部分的 AI，代表了自 17 世纪以来伟大思想家们脑海中萌芽的愿景的顶峰。许多数学家、哲学家和科学家，以不同的方式和不同的抽象层次，设想有一种语言能够机械化获取和分享知识。特别是戈特弗里德·莱布尼茨（Gottfried Leibniz, 1646-1716），他思考过至少一部分人类推理过程可以被机械化。

因此，现代软件和人工智能的发展不是孤立的现象，而是长期科学和哲学思考的结果。从莱布尼茨时代开始，就有学者试图找到将人类思维过程形式化、系统化的方法。AI 的出现，可以看作是这一长期探索的一个重要里程碑，它实现了在一定程度上模拟和执行人类认知任务的可能。

得益于著名数学家艾伦·图灵和阿隆佐·丘奇，智能机器的现代概念在 20 世纪中叶形成了。图灵在 1947 年对“智能机器”的探索，再加上他 1950 年发表的开创性论文《计算机器与智能》，为图灵测试奠定了基石。图灵测试是 AI 的一个关键概念，它要求机器展现出与人类行为无异的能力（以至于人类裁判无法区分），这标志着 AI 作为一门科学学科的时代正式开始。

^① 译注：在自然语言处理（NLP）的上下文中，“结构化本体”具有重要作用。它为自然语言提供了形式化的知识表示，帮助计算机理解自然语言的深层含义。通过构建结构化本体，可以将自然语言的模糊性、多义性转换为计算机能够处理的形式化数据，进而支持更复杂的语言理解和生成任务，如信息抽取、语义搜索、问答系统和机器翻译。结构化本体通常包括以下几个关键组件：概念、关系、属性、实例、层次结构以及公理/规则。



注意：考虑到当下 AI 的发展如火如荼，可能需要重新评估原始的图灵测试，以纳入对人类和理性行为更精确的定义。

NLP

NLP（自然语言处理）作为 AI 领域内一个融合多学科精华的分支，矢志于构建起计算机与人类语言间的沟通桥梁。尽管其历史渊源深深植根于语言学的理论与实践，展现出与现代 AI 理念不同的风貌，但在更为宏阔的视野下，NLP 始终是 AI 大家庭中的一员。实际上，它贯穿始终的核心诉求，便是以人工手段再现人类智慧的璀璨光芒——尤其聚焦于语言这一复杂而又精妙的表达形式。

NLP 的主要目标是使机器能以一种有意义且与上下文（语境）相关的方式理解、解释和生成近似于人类使用的语言。这个跨学科领域汲取语言学、计算机科学和认知心理学的知识来开发先进的算法和模型，使人与机器能通过自然语言无缝地交互。

NLP 的历史跨越了数十年之久，从早期基于规则的系统一路演变为当今的深度学习方法，标志着计算机理解和处理人类语言的重大进步。

从 1950 年代开始，人们的一些早期努力（例如，1954 年旨在将俄语翻译成英语的乔治城大学-IBM 实验）为 NLP 奠定了基础。然而，这些早期努力在本质上多侧重于语言学范畴。随着后续数十年的光阴流转，乔姆斯基语言学理论^①逐渐彰显其影响力，引领着该领域的研究重心逐步向句法及语法结构的深度探索倾斜。

1980 年代开始向统计方法（例如，n-grams）转变，它们使用单词共现频率来进行预测。IBM 的 *Candide* 语音识别系统便是其中的一例。^②然而，这种方法还是难以应对自然语言的复杂性。1990 年代见证了统计方法的复兴和机器学习（ML）技术的兴起，例如隐马尔可夫模型（Hidden Markov Model, HMM）和统计语言模型。在这一时期，宾夕法尼亚树库（Penn Treebank，简称为 PTB，是一个包含 700 万单词且进行了词性标注的数据集）以及各种统计机器翻译（Statistical Machine Translation, SMT）系统是这一时期重要的里程

^① 译注：乔姆斯基的语言学理论的基础为生物语言学，该语言学派认为，支撑语言结构的原理在生物学上被预设在了人类的大脑中，因此语言结构是遗传性的。乔姆斯基认为，所有人类都有与社会文化差异无关的、相同的基本语言结构。

^② 译注：IBM 的 *Candide* 系统是第一次使用统计概率而非人为制定的规则进行机器翻译的尝试。*Candide* 使用加拿大的《国会议事录》作为训练数据，其中涉及大量法语和英语的议会辩论。它使用的是一种“基于短语”的方法，能确保单词的翻译适当地考虑周围的单词（这正是这段话中所说的“单词共现频率”的含义）。

碑。

2000年代，由于互联网上海量的文本数据唾手可得，而且数据驱动的方法开始兴起，使得这一领域重新焕发了活力。包括最大熵模型（maximum-entropy model）和条件随机场（conditional random field）在内的**概率模型**获得了广泛关注。从1980年代开始但多年之后才完成的 WordNet（一个英语语义词汇数据库，其中包括丰富的同义词组及其关系）极大地促进了对词义的更深层次的理解。

2010年代，随着新一代图形处理器（Graphics Processing Unit, GPU）的出现和算力的显著提升，深度学习的兴起彻底改变了这一领域的格局。神经网络架构——特别是像基于转换器的双向编码器表示（Bidirectional Encoder Representations from Transformers, BERT）和生成式预训练转换器（Generative Pretrained Transformer, GPT）这样的转换器——通过捕捉复杂的语言模式和上下文信息，彻底革新了自然语言处理（NLP）。研究的重点转移到了数据驱动和预训练语言模型上，使得特定任务的微调成为可能。

对比预测式 AI 和生成式 AI

预测式 AI 和生成式 AI 代表两种截然不同的范式（思维模式），两者皆与神经网络及深度学习架构的革新紧密交织，并且相辅相成。

预测式 AI 通常与监督学习（supervised learning）有关，它的起源可以追溯到 20 世纪中叶出现的经典机器学习（ML）方法。像感知机（perceptron）这样的早期模型为神经网络在 1980 年代的复兴铺平了道路。然而，直到 21 世纪深度学习的兴起，伴随着深度神经网络、用于图像识别的卷积神经网络（Convolutional Neural Network, CNN）以及用于序列数据处理的循环神经网络（Recurrent Neural Network, RNN）的发展，预测式 AI 才经历了真正的变革性复苏。长短期记忆单元（Long Short-Term Memory, LSTM）的引入使我们能更有效地建模数据中的序列依赖关系。

另一方面，生成式 AI 在无监督学习和复杂神经网络架构（预测式 AI 也使用这种架构）的推动下取得了显著进展。生成式模型的概念可以追溯到 1990 年代，但真正的突破是在 2014 年引入**生成式对抗网络**（Generative Adversarial Network, GAN）时发生的，人们从中体会到了对抗训练的强大之处。GAN 在其中发挥了关键作用，它具有一个生成数据的生成器和一个区分真实数据与生成数据的鉴别器。鉴别器在训练过程中鉴别生成数据的真实性，这有助于改进生成器，从而生成更加逼真的数据，范围从栩栩如生的图像到逻辑连贯的文本。

表 1-1 回顾了主要类型的学习过程。

表 1-1 主要类型的学习过程

类型	定义	训练	使用场景
监督式	在有明确标签的数据上训练，每个输入都有对应的标签	调参以最小化预测误差	分类、回归
自监督式	由模型自动生成标签的无监督学习方式	学会填空（根据输入数据的其他部分预测缺失的部分）	NLP、计算机视觉
半监督式	结合有标签和无标签的数据进行训练	使用有标签的数据执行有监督的任务，使用无标签数据进行泛化（generalization） ^①	带标签的数据有限的场景，例如图像分类
无监督式	在没有显式监督的数据上训练	识别数据中固有的结构或关系	聚类、降维、生成式建模

预测式 AI 和生成式 AI 的历史发展轨迹凸显了它们与神经网络及深度学习之间的共生关系。预测式 AI 利用深度学习架构（例如，用于图像处理的 CNN 和用于序列数据处理的 RNN/LSTM）在从图像识别到自然语言理解等任务上取得了最前沿的技术成果。而生成式 AI，则借助于 GAN 和大规模语言模型的强大能力，展现了神经网络在创造新内容方面的无限潜力。

LLM

以 OpenAI 的 GPT 系列为代表的 LLM（大语言模型）是建立在先进的深度学习架构（例如转换器，即 transformer，详见附录）上的生成式 AI 系统。

这种模型遵循无监督与自监督学习的基本原则进行运作，通过在海量的文本语料库上进行深度训练，能理解并生成流畅的、符合上下文的文本。它们的输出涵盖了各种形式的文本序列——从标准文本到蛋白质结构、代码、SVG 图像、JSON、XML 等多元化的数据格式，

^① 译注：在半监督学习中，模型不仅仅是在有限的标签数据上学习特定的任务，还通过大量未添加标签的数据来理解和推广那些学习到的模式，从而在实际应用时能更好地处理未见过的数据或情境。这样做可以帮助模型在只有少量标签数据可用的情况下，更全面地了解整个数据集的结构和变化，从而提高整体性能。

这充分展现了其非凡的技能：它们能精准地模仿人类的语言习惯，对提供的初始“提示”进行自然的延伸与丰富，使生成的内容既符合逻辑又富有创意。

这些模型的架构，特别是转换器架构，使其能捕捉数据中的长距离依赖关系和复杂模式。作为一种关键的前置技术，词嵌入（word embeddings）技术将词表示为连续的向量（米科洛夫^①等人在 2013 年通过 Word2Vec 实现），有助于模型理解词与词之间的语义关系。

最新的这一批模型的“生成式”（generative）性质使它们在输出上具有了多样性，允许进行文本补全、摘要（汇总）以及文本创作等任务。用户可以使用多样化的查询或者不完整的句子来提示模型。相应地，模型会自动生成连贯且与上下文相关的后续文本，这充分展现了其理解并模仿人类语言模式的能力。

这一旅程始于 2013 年，当时词嵌入技术的引入，尤其是米科洛夫等人提出的 Word2Vec 模型，彻底革新了语义表示的方法。紧随其后的是 RNN 和 LSTM 架构的出现，它们解决了序列处理（sequence processing）和长距离依赖（long-range dependencies）的挑战。而革命性的突破是在 2017 年转换器架构问世后发生的，该架构允许并行处理并显著缩短了训练时间。

2018 年，谷歌（Google）研究人员德夫林（Devlin）等人提出了 BERT 模型。BERT 创新性地采用了一种双向上下文预测方法。在预训练阶段，BERT 通过一个遮蔽语言建模（Masked Language Modeling, Masked LM）任务进行训练，其中句子中的随机单词子集被遮蔽，模型根据左右两侧的上下文来预测这些遮蔽词。^②这种双向训练机制使 BERT 能够捕捉词与词之间更细微的上下文关系。因此，BERT 在需要深度理解上下文的任务中表现出色，例如问题回答和情感分析。

与此同时，OpenAI 发布的 GPT 系列标志着 NLP 领域的一个范式转移，从 2018 年的 GPT 开始，通过 2019 年的 GPT-2，到 2020 年的 GPT-3，到 2023 年的 GPT-3.5-turbo、GPT-4、GPT-4-Turbo-Vision（支持多模态输入^③），再到 2024 年的 GPT-4o。作为自回归

^① 译注：汤姆什·米科洛夫（Tomas Mikolov）是一位知名的计算机科学家，以其在自然语言处理（NLP）领域的贡献而闻名，尤其是在词嵌入（word embeddings）和深度学习模型方面的工作。他是 Word2Vec 算法的主要开发者之一，这一算法极大地推动了 NLP 的发展，使得机器能够以更高效的方式理解人类语言。

^② 译注：使用这种技术，模型会从输入文本中随机选取一些词，并用一个特殊标记（如[MASK]）替换掉这些词。然后，模型的任务就是根据上下文信息去预测这些被遮蔽的词是什么。这种技术帮助模型学习到更深层次的语义和上下文关系，因为模型必须同时考虑一个词前后的词来做出预测。

^③ 译注：多模态模型可以将各种输入（包括文本、图片和音频）作为提示进行处理，并将这些提示转换为各种输出，而不仅仅是源类型。

(autoregressive) 模型，它们基于已知的上下文来预测序列中的下一个 token^①或词。GPT 的自回归方法是一次预测一个 token，使其能生成连贯且与上下文相关的文本，展现了其出色的多样性和语言理解能力。然而，这种模型的规模非常庞大。例如，GPT-3 的参数数量就已经达到了 1750 亿个，而 GPT-4 的参数数量更是达到了 1.8 万亿个，是 GPT-3 的 10 倍以上。事实上，这些模型可以扩展和泛化，从而减少针对特定任务进行微调的需求。^②

工作原理

大多数 LLM（大语言模型）的核心功能原则是**自回归语言建模**（autoregressive language modeling），即模型接收输入的文本，并系统地预测序列中的后续 token 或词（稍后会说明这两个术语的区别）。这种逐 token 预测过程对于生成连贯且与上下文相关的文本至关重要。然而，正如杨立昆^③所强调的，这种方法可能会累积错误。换言之，如果第 N 个 token 是错误的，模型可能会持续假设其（虚假的）正确性，导致生成不准确的文本。

直至 2020 年，微调（fine-tuning）一直是定制模型以适应特定任务的主要手段。然而，近来的技术革新，尤其是以 GPT-3/4 为代表的大模型，引入了提示工程（prompt engineering）这一概念。借助于此，这些模型能够在无需传统微调的情况下，仅凭精准的指导性提示，便能达成任务特定的成果，展现出非凡的灵活性与高效性。

诸如 GPT 系列中的模型，它们进行了匠心独运的设计，旨在全面汲取蕴藏于人类语言语料库中的语法、语义及内在本体论知识^④。它们在捕捉宝贵的语言信息方面展现出了卓越的能力，但也必须认识到，这些模型也可能承袭了其训练语料中潜藏的不准确性与偏见。

^① 译注：token 是最小的语义单元。模型在生成文本时，每次输出的一个词或子词单元被视为一个 token。例如，当 ChatGPT 生成一段对话回复时，它可能会逐个 token 地生成文本，直到达到预设的最大长度限制或者生成了一个结束符号（例如，句号或换行符）。本书保留 token 的原文。在其他文献中，也有人把它翻译为“词元”。

^② 译注：如果说模型可以扩展和泛化（scale and generalize），那么意味着它们能在更大的或更多样化的数据集上有效运行，并且能适应或理解广泛类型的任务或问题，而不仅限于它们在训练过程中直接接触过的数据。这种能力使得这些模型在面对新的、未见过的情况或数据时，仍然能够表现出良好的效果。换言之，不需要过多地为新的、特定的任务或应用场景进行调整或优化。

^③ 译注：杨立昆（法语：Yann André Le Cun），本名扬·安德烈·勒丘恩，是一名法国计算机科学家，2018 年图灵奖得主，他在机器学习、计算机视觉、移动机器人和计算神经科学等领域都有很多贡献。他最著名的工作是在光学字符识别和计算机视觉上使用卷积神经网络，他也被称为卷积网络之父。

^④ 译注：本体论（ontology）是计算机科学和信息科学中的一个关键概念，特别是在语义网和知识管理系统中，它用于表示知识领域中实体之间的关系。这里的“本体论知识”是指人类语言语料库中固有的基本本体结构，主要涉及语言如何系统地组织和表示世界上的各种概念及其相互关系。

不同的训练方法

大语言模型（LLM）的训练目标各异，每种目标都要求采取不同的策略。下面描述了三种各具特色的主流方法：

- **因果语言建模（Causal Language Modeling, CLM）**：这一自回归方法为 OpenAI 的 GPT 系列所采用。CLM 训练模型依据先前的 token 预测序列中的下一个 token。虽然在文本生成和摘要等任务中表现出色，但其局限在于仅能从单向视角考虑上下文，预测时仅依赖过往信息。鉴于 CLM 架构目前应用最为广泛，我们将着重探讨此类模型。
- **遮蔽语言建模（Masked Language Modeling, MLM）**：该方法常见于 BERT 等模型中，通过随机遮蔽输入序列中的部分 token，模型需根据周边上下文预测原始 token。这种双向考虑的策略在文本分类、情感分析和具名实体识别（named entity recognition）等领域占据优势。然而，对于纯粹的文本生成任务，MLM 并非理想之选，因为这种任务要求模型仅依据过往（左侧）信息进行预测，而不考虑未来（右侧）的情况。
- **序列到序列（Sequence-to-Sequence, Seq2Seq）**：这类模型采用编码器-解码器（encoder-decoder）架构，主要用于机器翻译和摘要生成等任务。其中，编码器负责处理输入序列，生成潜在表征，而后由解码器根据该表征生成输出序列。Seq2Seq 架构在处理输入输出转换的复杂任务中表现出色，尤其适合输入与输出在训练中高度对应的任务，例如翻译。

这些方法之间根本的差异在于其训练目标、架构设计以及对特定任务的适用性。CLM 专注于预测序列的下一个 token，擅长文本生成；MLM 专攻（双向）上下文理解；而 Seq2Seq 架构精于序列形式的连贯输出生成。CLM 模型适合自回归任务，MLM 模型适合上下文理解与嵌入，而 Seq2Seq 模型则适合输入输出转换。此外，模型还可能通过预训练辅助任务，如**下一句预测**（Next Sentence Prediction, NSP）^①，以检验其对数据分布的理解程度，进一步提升综合能力。

transformer 模型

transformer 架构是现代 LLM 的基础。阿西什·瓦斯瓦尼（Ashish Vaswani）等人在 2017 年 12 月发布的论文“Attention Is All You Need”中首次提出了 transformer 模型。从此之后，自然语言处理（NLP）发生了彻底的革新。与以往依赖顺序处理的传统模型不同，transformer 运用了一种注意力机制，这不仅使并行计算成为可能，也有效捕捉了长距离依赖关系。

^① 译注：在 BERT 中，Next Sentence Prediction 捕捉的是句子之间的关系，而 Masked LM 捕捉的是词之间的关系。两者在训练中都要用到。

原始的模型由编码器和解码器构成，两者均包含多个自注意力处理层。**自注意力处理**（self-attention processing）意味着每个词都是通过检查和考虑其上下文信息来确定的。

在编码器中，输入序列被嵌入并通过各层并行处理，从而捕捉词与词之间的复杂关系。解码器利用编码器的上下文信息生成输出序列。在训练过程中，解码器通过分析之前的词来预测下一个词。

transformer 融合了多层解码器，以此增强其语言生成能力。transformer 的设计包括一个上下文窗口（context window），它决定了模型在推理和训练期间考虑的序列长度。较大的上下文窗口赋予模型更广阔的视野，允许其处理更长的序列，但随之而来的是计算成本的增加；而较小的窗口虽然能减少计算成本，但可能错过关键的长距离依赖关系。真正使 transformer 能够洞察上下文，并在诸如翻译和摘要等任务中表现出色的“大脑”是**自注意力机制**。今天的 LLM 中并不存在类似意识或神经学习这样的概念，其工作机制是基于统计模式进行识别和概率预测，而非具备人类般的直觉或情感。

自注意力机制使 LLM 能够有选择地关注输入序列的不同部分，而不是“无脑”地以同一种方式处理整个输入。因此，它只需较少的参数就能模拟长期依赖关系，并能捕捉序列中相隔较远的词之间的关系。尽管看起来非常聪明和人性化，但它本质上只是基于统计来猜测下一个词。^①

虽然原始的 transformer 架构是一个 Seq2Seq 模型，将整个序列从源格式转换为目标格式，但现代文本生成技术已经演进为一种自回归方法。

与原始架构不同，包括 GPT 在内的模型已摒弃了显式的编码器组件，转为只依赖解码器。在这种架构中，输入直接送入解码器。解码器具有更丰富的**自注意力头**（Self-Attention Head），并已通过无监督方式在大量数据上进行了训练，其唯一的使命就是预测现有文本序列的下一个词。与此形成鲜明对比的是 BERT 等模型，它们仅包含生成所谓“嵌入”的编码器组件。

token 和 tokenization

token 是 GPT 等高级语言模型的基本组成部分，是复杂的语言理解和生成过程的核心。

^① 译注：自注意力机制能捕捉到序列中相隔甚远的词与词之间的关系，这是其处理长序列数据时的一大优势。尽管自注意力机制让 LLM 在处理语言任务时显得相当智能，甚至具有某种类人的决策能力，但实际上，这一切都是基于对大量文本数据进行统计分析后得到的概率预测。模型通过学习到的统计规律来预测下一个词的可能性，而并非基于理解或意识。因此，尽管 LLM 的表现令人印象深刻，但它本质仍然是一个统计模型，其“智慧”来源于数据而非真正的认知过程。

与传统语言单位（例如，词、字和字符）不同，token封装了词、字符/字或子词单元^①的精髓。这种更细的粒度对于捕捉语言固有的微妙和复杂性至关重要。

在这些模型中，tokenization 过程是关键。它涉及到将文本分解成更小、更易管理的单元，即 token。然后，这些 token 由模型进行分析。这里有意选择了 token 而不是词，目的是更细致地表示语言。^②

OpenAI 和 Azure OpenAI 采用一种称为**字节对编码**（Byte-Pair Encoding, BPE）的子词 tokenization 技术。BPE 将频繁出现的字符对合并成单个 token，这有助于更紧凑、更一致地表示文本数据。依据 BPE 规则，每个 token 大致涵盖英语中的四个字符或者单词长度的四分之三。也就是说，100 个 token 大致相当于 75 个单词。例如，句子 “Many words map to one token, but some don't: indivisible” 将被拆分为["Many", " words", " map", " to", " one", " token", ",", " but", " some", " don", "'", ":", " indiv", "isible"], 这些 token 的 ID 对应为: [8607, 4339, 2472, 311, 832, 4037, 11, 719, 1063, 1541, 956, 25, 3687, 23936]。^③

tokenization 承担着多重职责，它既影响所生成文本的计算动力学方面^④，也关乎着所生成文本的质量。运行 LLM 所涉及的计算成本与 tokenization 方法、词汇表规模（通常，针对单一语言的词汇表会包含 3 万到 5 万个不同的 token）以及输入输出文本的长度与复杂度密切相关。

在 LLM 中选择 token 而不是词是出于以下多方面的考虑：

- token 有助于以更细的粒度表示语言，使模型能够辨别细微的意义并有效处理生僻词或词汇表外的词。当处理具有丰富形态结构的语言时，这种粒度尤其关键。

^① 译注：什么是“子词”（subword）？举个例子就很容易理解了。例如，learning 可以拆分为 ["learn", "##ing"]。

^② 译注：这就解释了为什么 tokenization 不适合翻译为“分词”，因为对一段文本（一个“序列”）进行分解时，得到的 token 并非一定是“词”。相反，还有可能是其他最小的语义单元，例如标点符号、换行符等。

^③ 译注：这里有必要解释一下中文的 tokenization。简单地说，中文的 tokenization 相较于英文更难，因为英文有天然的空格作为分隔符，所以如何拆分是一个难点。再加上中文里一词多意的情况很普遍，导致很容易出现歧义。另外，中文的 tokenization 还要考虑粒度问题。例如，“中国科学技术大学”就很多种分法：中国科学技术大学，中国\科学技术\大学，中国\科学\技术\大学。这里推荐从哈工大的 LTP 分词工具开始入门：<https://github.com/HIT-SCIR/ltp>。

^④ 译注：所谓“动力学”方面，是指与计算过程中的性能、效率、资源管理以及响应速度等相关的特性。

-
- token 采用一种更“组合性”的方法^①来应对语言中的歧义和一词多义的挑战。
 - 子词 tokenization 使 LLM 能够将词表示为子词 token 的组合，从而更有效地根据前面或后面的字符捕捉词的不同含义。例如，一个英语单词的后缀可以有两种不同的 token 表示，具体取决于下一个单词。^②
 - tokenization 算法通常针对一种特定的语言运行，造成该算法在应用于其他语言时可能效果不佳。例如，一个针对英语设计的基于字节对编码（BPE）的 tokenization 算法可能无法有效处理像汉语或阿拉伯语这样的形态丰富或是无空格分隔词汇的语言。但是，如果其他语言使用的是同一个字符集，那么效果还是可以保证的。
 - 使用 token 显著能提高内存使用效率。通过将文本分解为更小的单元，LLM 可以更有效地管理内存，处理和存储更大的词汇量，而不会对内存资源提出不切实际的要求。

总之，token 和 tokenization 是塑造 LLM 的语言处理和理解能力的根本。从提供粒度到管理内存，再到解决语言挑战，token 在优化 LLM 的性能和效率方面发挥了重要作用。

嵌入

tokenization 和 embedding^③是 NLP 中密切相关的两个概念。

tokenization 涉及将文本序列分解成较小的单位。这些 token 被转换成 ID，作为模型处理文本信息时的基本构建单元。另一方面，embedding 是指这些 token 在高维向量空间（通常为 1000 多个维度）中的数值和密集表示^④。

embedding 通过模型中的一个“嵌入”层生成，该层编码了 token 的语义关系和上下文信息。“嵌入”层在训练过程中学习每个 token 的分布式表示，使模型能基于词或子词的上下

^① 译注：在自然语言处理中，“组合性”（compositional）通常指的是语言元素（如词、短语或句子）如何通过其组成部分的结合来构建更复杂的意义。这一概念基于语言的一个基本属性，即更大的语言结构可以通过较小的组件以一种系统的方式组合和理解。它使模型不仅仅能理解单独的词或短语，而是可以通过组合这些元素的含义来理解更长文本的含义。

^② 译注：例如，单词“unbelievable”可能有两种不同的上下文：“It is unbelievable”和“It is unbelievably good”。

^③ 译注：虽然可以将 embedding 翻译为“嵌入”，但本书以后会尽量保留原文。

^④ 译注：所谓“密集表示”（dense representations），是指这些向量在表示信息时不依赖于稀疏的数据结构，而是在高维向量空间中的每一个维度上通常都有实际的数值。这与“稀疏表示”相对，稀疏表示中大部分元素值为零，只有少数非零元素。这种表示法在向量的每个维度上都包含了密集的信息，这对于捕捉和表达复杂的语义特征非常有用。

文使用情况来理解它们之间的关系和相似度。

通过 **embedding**，我们可以轻松地进行语义搜索。换言之，可以在这个 1000 多个维度的空间中嵌入不同的句子，并测量它们的距离^①。句子越短，这个高维空间越大，语义表示就越准确。**embedding** 的内在目标是使类似于“queen”和“king”这样的词在 **embedding** 空间中彼此接近，而“woman”和“queen”之间的距离也非常相近。

“嵌入”可以在词级别上工作，例如 Word2Vec（2013 年），也可以在句子级别上工作，例如 OpenAI 的 text-ada-002（最新版本于 2022 年发布）。

embedding 模型（即输入一些文本并输出一个密集数值向量的模型）通常是 **transformer** 模型的“编码”部分的输出。不过，GPT 模型的情况有所不同。事实上，GPT-4 在注意力头^②中有一些内部的“嵌入”层（词和位置），而专用的 **embedding** 模型（text-ada-002）是单独训练的，不直接集成在 GPT-4 中。text-ada-002 与文本生成模型相似，用于相似度搜索和其他类似的应用（后面会详细讨论）。

总之，**tokenization** 是准备文本数据用于机器学习模型的初始步骤，而 **embedding** 通过创建有意义的数值表示来增强这一过程，这些表示捕捉了不同 **token** 细微的语义差别和上下文信息。

训练步骤

GPT 等语言模型的训练涉及几个关键阶段，每个阶段都对模型的发展和熟练度做出了贡献：

1. 基于抓取数据的初始训练
2. 监督微调（Supervised Fine-Tuning, SFT）

^① 译注：例如，使用余弦相似度或欧几里得距离等度量标准来确定两个句子向量的接近程度，从而评估它们的语义相似度。

^② 译注：“注意力头”（attention heads）是 **transformer** 架构中的一个重要组成部分，特别是在自然语言处理（NLP）和其他深度学习任务中广泛应用。这一概念来源于之前描述过的论文“Attention is All You Need”，其中首次引入了 **transformer** 模型。在 **transformer** 模型中，注意力机制允许模型在处理一个序列（如句子）时，能够对序列中不同部分的重要性进行“关注”。这种机制通过计算所谓的“注意力分数”来实现，这些分数决定了在给定上下文中，模型应该更多关注序列的哪些部分。在实践中，一个单独的注意力机制（即一个注意力头）可能只能捕捉到有限的依赖关系或特定类型的信息。为了更全面地理解数据，**transformer** 模型通常包含多个并行的注意力机制，我们称之为“多头注意力”（Multi-Head Attention）。

3. 奖励建模 (reward modeling)

4. 基于人类反馈的强化学习 (Reinforcement Learning From Human Feedback, RLHF)

基于抓取数据的初始训练

在初始阶段，语言模型在从互联网抓取以及/或者从私有数据集中收集的庞大数据上进行**预训练**。在这个为未来模型准备的初始训练集中，可能包括由 LLM 生成的文本。在这个阶段，模型通过预测给定上下文中序列的下一个词来学习语言的模式、结构和表达。这是使用一个语言建模目标 (language modeling objective)^①来实现的。

tokenization 是一个关键的预处理步骤。在此过程中，单词或子词被转换成 token，然后转换成数值 token。使用 token 而不是词，这使模型能够捕捉语言中更细微的关系和依赖，因为 tokens 可以代表子词单元、字符或甚至词的不同部分。

模型被训练以根据之前的 token 预测序列中的下一个 token。这个训练目标通常使用损失函数 (loss function) 来实现，例如交叉熵损失 (Cross-Entropy Loss)，它测量预测的 token 概率分布与实际分布之间的差异。

此阶段出来的模型通常被称为**基础模型** (base model) 或**预训练模型** (pretrained model)。

监督微调 (SFT)

在初始训练之后，模型进行监督微调 (SFT)。在这一阶段，我们向模型提供 prompt 和 completion^②，以进一步完善模型。模型从有标签的数据中学习，调整其参数以提高特定任务的性能。

一些小的开源模型使用较大模型的输出来完成这一微调阶段。尽管这是一种节省训练成本的聪明方法，但可能导致模型所宣称的能力高于实际。

奖励建模

^① 译注：尽管“预测下一个词”是最常见的语言建模目标之一，它用于训练如 GPT 这样的生成模型，但还存在其他多种语言建模目标，例如之前提到过的遮蔽语言建模 (MLM)、下一句预测 (NSP) 以及翻译建模等。

^② 译注：完成预训练后，模型通过监督微调进一步针对特定任务进行优化。这一阶段使用的是有标签的数据，即每个输入样本 (prompt) 都有一个预期的输出 (completion)，这些输出被用作训练过程中的真实答案。模型的任务就是在接收到 prompt 后，尽可能地生成与之对应的 completion。由于这是在一个有监督的学习环境中进行的，这意味着每一个训练样本都有一个已知的正确答案，模型可以通过比较自己的输出与正确的 completion 来调整自身参数，从而逐步改进其性能。

一旦模型通过 SFT 完成了微调，就可以由人类评估员根据质量、相关性、准确性和其他标准审查并评价模型的输出。这些评分被用于创建一个奖励模型，它能预测各种输出的“奖励”或者说评分。如下一节所述，一旦奖励模型建立起来，就可以用它指导其基础模型的进一步训练。

基于人类反馈的强化学习(RLHF)

有了奖励模型的加持后，我们通过基于人类反馈的强化学习（RLHF）来指导基础模型生成更优质的输出。模型从奖励模型那里获得对其输出的反馈，并相应地调整自身参数，以求最大化预测的奖励值。这一强化学习的过程显著提升了模型的精确度及沟通技巧。例如，像 GPT-4 这样的闭源模型，实际上就是 RLHF 模型（尽管其基础模型尚未开源）。

这里的重点在于，我们需要认识到，在向基础模型进行提示与向 RLHF 或 SFT（监督微调）模型进行提示之间，存在着本质的区别。当面对“为我写一首关于爱的歌”这样的指令时，基础模型产生的结果可能更接近于“为我写一首关于忠诚的诗”，而非一首关于爱的歌。这种倾向源自于训练数据集。在数据集中，“为我写一首关于爱的歌”这一短语可能总是紧随其他类似指示出现，导致模型生成遵循那些模式的响应。为了引导基础模型生成一首关于爱的歌，对提示工程（prompt engineering）的精妙运用显得至关重要。例如，设计一个像“这是一首关于爱的歌：自从我们相遇的那天起，我就爱你”这样的指令，通过直接提供一首爱情歌的开始部分，给模型一个明确的上下文和起点，使得模型更有可能在这个基础上继续生成符合主题的歌词。

推理

推理（*inferring*）是一个自回归生成过程，期间会迭代地调用模型，并使用其自身生成的输出来作为初始输入。在因果语言建模（CLM）中，文本 token 序列被作为输入，模型返回下一个 token 的概率分布。在从这个分布中选择下一个 token 时，会出现非确定性的方面（结果不是固定的，而是有多种可能性）。^①这种非确定性一般是通过采样来实现的。然而，一些模型提供了种子选项以产生确定性结果。

选择过程可以从简单（选择最可能的 token）到复杂（涉及多种转换）。如温度这样的参数

^① 译注：采样带来的非确定性是模型所生成的文本的一个重要特性，因为它使生成的内容不是完全可预测的。这种随机性是模拟人类语言的多样性和复杂性的关键，使得生成的文本更加自然和有趣。然而，这也带来了挑战，因为模型有可能生成不相关或奇怪的输出，尤其是在高度随机的采样策略下。

会影响模型的创造力，较高的温度会产生更平坦的概率分布。^①

迭代过程持续进行，直至达到停止条件——理想情况下由模型确定或者是一个预定义的最大长度。

当模型生成错误、无意义甚至是错误信息时，这被称为**幻觉**（hallucination）。当 LLM 生成文本时，它们是作为基于提示的一种外推器^②运行的。由于它们不是被作为一种数据库或搜索引擎设计的，所以不会引用具体的训练数据源。抽象过程（也就是对提示和训练数据进行转换的过程）可能因为有限的上下文理解而造成幻觉，导致信息的丢失。

以 GPT-3 为例，虽然它经过了数万亿个 token 的训练，最终几乎有 1 TB 的数据，但这些模型的权重——决定其大小的因素——通常比原始数据减少了 20% 到 40%。在这里，人们采用了量化技术以试图减少权重的存储需求，截断权重的精度。然而，LLM 并不是作为专门的无损压缩器而设计的，这导致在某些时候可能发生信息丢失；这是产生幻觉的一种可能的解释。^③

另一个原因是 LLM 作为自回归预测器的内在限制。事实上，在预测下一个 token 时，LLM 严重依赖于其上下文窗口内属于数据集分布的 token，这主要由人类编写的文本组成。当

^① 译注：这里提到的“温度”（temperature）是自然语言处理（NLP）中控制生成文本过程中随机性的一个参数。在使用神经网络模型进行文本生成时，温度用来调整预测的概率分布，从而影响采样过程中选取下一个词的多样性和创造性。当温度值低（接近 0）时，概率分布变得更“尖锐”，高概率的 token 的概率被放大，而低概率的几乎被忽略。这导致模型更倾向于选择最可能的词，但可能导致输出缺乏多样性，生成的文本较为重复和可预测。当温度值高时，概率分布变得更“平坦”。这意味着即便是不太可能的词也有更高的被选择的机会，从而增加了文本的多样性和创新性，但同时可能会牺牲一些语言的连贯性和逻辑性。

^② 译注：“外推器”（extrapolator）这个词来源于数学和统计学中的“extrapolation”（外推），指的是在已知数据点之外进行预测的过程。

^③ 译注：一个模型的权重数量直接关联到它的大小和复杂性。权重越多，模型通常越能捕捉复杂的数据模式，但同时计算需求和存储需求也越高。由于存储和操作大量权重需要大量的计算资源和存储空间，因此在实践中常常采用量化技术来减少模型权重的存储需求。量化涉及将权重从使用较多位的数据类型（如 32 位浮点数）转换为使用较少位的数据类型（如 8 位整数）。这种转换可以显著减小模型大小，降低在硬件上的存储和计算成本。尽管量化可以有效减少模型的存储和计算需求，但它也可能导致权重的精度下降，因为使用更少的二进制位来表示权重，意味着某些信息可能会丢失。这种精度的损失可能影响模型的性能，尤其是在处理需要高度精确的复杂任务时。在模型的权重被量化和精度被截断的情况下，模型可能无法完全准确地复现训练数据中学到的所有细节，这可能导致生成的文本中出现逻辑错误或不相关的信息，即“幻觉”。

我们执行 LLM 并从中对 token 进行采样时，每个采样的 token 都会使模型稍微偏离最初训练的分布。模型的实际输入部分由其自身生成，随着我们想要预测的序列长度的延伸，会逐渐将模型移出其熟悉的学习分布。



注意：在 LLM 中，幻觉可以被视为一种“特性”（feature），尤其是在寻求创造力和多样性时^①。例如，当从 ChatGPT 或其他 LLM 请求一个幻想故事情节时，目标不是复制，而是生成全新的角色、场景和故事情节。这种创造性要求模型不直接引用它们所训练的数据，以便生成富有想象力和多样性的输出。

微调、提示和其他技术

为了优化大语言模型（LLM）的响应，人们采用了多种技术，如提示工程和微调。

提示工程（prompt engineering）是指精心设计一个具体的用户查询，以引导和塑造模型的响应。这种专业技能旨在通过创建更有意义的输入来改善输出，通常要求对模型的架构有深入的理解。提示工程之所以有效，是因为它利用了新的、更大的语言模型的能力，这些模型已经学习了语言的通用内部表示。这些先进的模型通常是在庞大数据集上进行无监督预训练的技术开发的，对语言结构、上下文和语义有深刻的理解。因此，它们能根据收到的输入生成有意义的响应。

这种模型能解释并采用一种与上下文相关的方式来生成语言，而提示工程师在设计具体的查询时，正是利用了它们的这种能力。通过向模型提供更详细和有效的输入，提示工程指导模型产生期望的输出。本质上，提示工程与模型理解和生成语言的固有能力相一致，允许用户通过精心设计的提示影响和优化其响应。

相比之下，**微调**（fine-tuning，也称为“调优”）则是一种训练技术，通过应用新的、通常是自定义的数据集，使 LLM 适应特定的任务或知识领域。在这个过程中，需要用额外的数据训练模型的权重，从而提高其性能和相关性。

提示工程和微调服务于不同的优化目的。提示工程专注于通过改进输入来激发更好的输出，微调则旨在通过在新数据集上训练以提高模型在特定任务上的表现。提示工程提供了对 LLM 行为的精确控制，微调则增加了相关主题领域的深度。这两种技术可以互补，改善模型的整体行为与输出。

^① 译注：这类似于一些程序员说：“你懂什么？！这不是 bug，这是个 feature！”在某些时候，这样说也没有错。

注意，某些任务超出了 LLM 的能力，只能借助外部工具或补充软件来解决。一个例子是生成对用户输入“计算 $12*6372$ ”的响应，特别是假如 LLM 之前没有在其训练数据集中遇到过这种计算的“延续”^①。对此，一种较老的选择是使用插件作为扩展，允许 LLM 访问外部工具或数据，从而扩展其功能。例如，ChatGPT 就支持 Wolfram Alpha 和 Bing Search 等服务的插件。

我们还可以进一步推进提示工程，鼓励 LLM 进行自我反思。这涉及到如思维链提示（chain-of-thought prompt）这样的技术，它们指导模型解释其思考过程。约束性提示（例如，模板提示、交错生成和逻辑控制）也有助于提高模型输出的准确性和安全性。

总之，优化 LLM 的响应是一个多方面的过程，涉及到提示工程、微调和插件集成的综合运用，所有这些都是为了满足期望任务和领域的具体需求而量身定制的。

多模态模型

大多数 ML 模型是以单模态方式训练和工作的，即使用单一类型的数据——文本、图像或音频。多模态模型则融合了不同模态的信息，包括图像和文本等元素。就像人类一样，它们可以无缝地处理不同的数据模式。这种模型通常需要经过略有不同的训练过程。

多模态的类型包括：

■ 多模态输入，包括以下内容：

- 文本和图像输入：多模态输入系统同时处理文本和图像输入。这种配置有利于诸如视觉问题回答之类的任务，模型基于合并的文本和图像信息来回答问题。
- 音频和文本输入：兼顾音频和文本输入的系统特别适合语音转文本和多模态聊天机器人等应用。

■ 多模态输出，包括以下内容：

- 文本、图像和视频输出：一些模型同时生成文本、图像和视频输出，适合文生图、文生视频或生成图像描述等任务。
- 音频和文本输出：一些场景需要同时用到音频和文本输出，例如基于文本输入生成口语化的回应。

^① 译注：“延续”（continuation）一词指的是在语言模型的训练数据中，对一个特定输入或问题的后续处理或延续。具体来说，指的是模型是否在其训练期间遇到过类似于“计算 $12*6372$ ”这样的计算命令，并学习了如何处理此类请求的后续或结果。

■ 多模态输入和输出，包括以下内容：

- 文本、图像和音频输入：综合性的多模态系统需要同时处理文本、图像和音频输入，要求对多种数据源有更全面的理解。
- 文本、图像、视频和音频输出：生成多模态输出的模型提供多样化的响应——例如，对用户查询生成文本描述、图像、视频和口语对话等内容。

DeepMind 的 Flamingo、Salesforce 的 BLIP 以及 Google 的 PaLM-E 等开创性模型引领了向多模态模型转变的先河。现在这种模型已经非常成熟，例子包括 OpenAI 的 GPT-4-vision 和阿里的 Qwen-VL 等。

鉴于当前的发展态势，多模态输出（同时也包括输入）可以通过改造现有系统并利用不同模型之间的整合来实现。例如，可以调用 OpenAI 的 DALL-E，基于来自 OpenAI GPT-4 的描述来生成图像，或者应用 OpenAI Whisper 的语音转文本功能，并将结果传递给 GPT-4。



注意：除了增强用户交互外，多模态功能还有望帮助视障人士在数字领域和物理世界中导航。

商业应用

LLM 重塑了商业应用及其用户界面（UI），其变革潜力涵盖了各个领域，提供了类似于人类推理的一系列能力。

例如，一些标准的 NLP 任务（例如，语言翻译、摘要、意图提取和情感分析等）在 LLM 的帮助下变得如丝一般润滑。它们为企业提供了强大的工具来高效地进行沟通和市场理解。其中一个例子就是客户服务中的聊天机器人应用。过去，人们不喜欢和聊天机器人交流，经常会觉得对方非常生硬和不自然。但现在的情况完全变了，基于 LLM 的聊天机器人能以一种非常人性化和高效的方式理解用户的意图并做出回应。

基于 LLM 的聊天机器人提供了全新的会话 UI，可以取代传统的用户界面，提供交互性更强和更符合直觉的体验。这对于复杂平台（例如，报告系统）尤为有益。

除了具体的应用之外，LLM 真正强大的地方在于其适应性。它们展现了类人的推理能力，适合需要细致理解和解决问题的多种任务。以电商领域为例，当面临对海量商品评论进行精细化分析与分类的需求时，LLM 凭借其特有的 **few-shot prompting** 能力——即仅需少量样本（示例）便可迅速掌握新任务的本领——为这一过程增添了前所未有的灵活性与效率，大幅优化了信息处理的精度与速度。

任何类型的内容创造都能从这种适应性中受益。例如，LLM 可以为营销材料和产品描述生成人性化的文本，优化信息传播的效率。而在数据分析中，LLM 能从庞大的文本数据集中提取有价值的见解，帮助企业做出更明智的决策。

从搜索引擎的智能优化到欺诈行为的精准检测，从网络安全的加固到医疗诊断的辅助决策，LLM 正迅速崛起为不可或缺的先进生产力，它以仿若人类般的逻辑推理能力，从海量实例中汲取智慧。诚然，正当我们站在新一轮工业革命的风口浪尖，有关算法偏见、个人隐私保护以及数据伦理的讨论愈发引人深思。因此，在商业场景中融入 LLM 技术，需慎之又慎，充分考量其潜在影响。归根结底，LLM 不仅象征着技术创新的里程碑，更在潜移默化中重塑了企业面对挑战、处理信息的策略与格局。

对话式编程

在快速数据处理和由 AI 驱动的应用的世界中，“自然语言”也成了一种新质生产力。现在，它既是编程媒介也是用户界面。这意味着自然语言已经事实上成了一种新的编程语言。换句话说，软件 3.0 的时代已经到来了。引用安德烈·卡帕斯（Andrej Karpathy）所做的类比，如果说软件 1.0 是“传统且老旧”的代码，软件 2.0 是神经网络技术栈，那么软件 3.0 就是对话式编程和软件的时代。^①随着 AI 逐渐成为一种随手可得的产品，这一趋势预计将会加剧。

自然语言的崛起

自然语言的影响是多方面的，它既是进行 LLM 编程的手段（一般是通过提示工程），也是用户界面（通常在聊天场景中使用）。

自然语言承担了一种声明式（宣告式）编程语言的角色，被开发者用来表达应用程序的功能，也被用户用来表达其期望的结果。自然语言一方面作为编程的输入方法，一方面又作为用户的沟通媒介，这体现了 LLM 日益提升的能力和万用性。在编程的复杂性和用户交互之间，原本存在的巨大鸿沟被自然语言表述弥补了。

^① 译注：2017 年，AI 领域的大咖、时任特斯拉 AI 总监的安德烈·卡帕斯发表了一篇名为“软件 2.0”的帖子。他认为神经网络在许多方面取代了传统的编码方式，这种转变带来的影响比大家想象的要深。因为神经网络已经不仅仅是一种简单的分类方法，而是一种全新的编程范式。至于“软件 3.0”，则是目前的最新进展，安德烈并没有预测到。不过，译者个人认为，所谓软件 1.0，2.0 和 3.0，并不意味着完全的“取代”，而是会长期处于一种相互共存的状态。

自然语言作为(新的)表示层

在软件中，自然语言已经超越了其传统的沟通工具角色，现在正作为各种应用中一种强大的表示层（presentation layer）浮现。

现在，用户可以使用日常语言与系统和应用交互，而不必再像以前那样依赖于图形界面或者各式各样的**集成开发环境**（Integrated Development Environment, IDE）^①。得益于 LLM，这种范式转变简化了用户交互，使技术更容易被更广泛的用户群体接受。用户现在可以采取一种更加直观和简单的方式与应用进行交互。

在 LLM 的帮助下，开发者可以创建对话界面，将复杂任务转化为相对简单的日常对话。在简单软件和一些特定的使用场景中（例如在单独处理安全性时），常规的用户界面不再是必需的。整个后端 API 可以通过 Microsoft Teams、WhatsApp 或者企业微信的聊天界面来调用。

AI 工程

自然语言编程通常称为**提示工程**（prompt engineering），是最大化 LLM 能力的一门关键学科，其宗旨是创建有效的提示来指导 LLM 生成期望的输出。例如，当要求模型“返回以下文本中提到的城市的 JSON 列表”时，如果模型返回所引用的文本，而不是期望的 JSON，那么提示工程师应该知道如何重新措辞提示（或者知道哪些工具和框架能提供帮助）。类似地，提示工程师在处理一种基础模型而不是强化后的 RLHF 模型时，应该知道使用什么样的提示。

随着 OpenAI 的各种 GPT 及其商店的推出，让人觉得似乎只要有了想法，就能立即实现，随便什么人都能轻松开发由 LLM 驱动的应用。但实情并非如此。引入 GPT 模型和其生态系统确实降低了开发 AI 应用的技术门槛，然而，这种表面的易用性掩盖了深层次的挑战。开发高质量的 LLM 应用，需要开发者具备深厚的领域知识，理解模型的局限性和优势，以及如何恰当地引导模型生成有意义的输出。此外，还需要考虑用户体验、数据安全、伦理规范等一系列因素，这些都是简单的一条有创意的“提示”所无法涵盖的。

有的时候，仅仅进行提示工程（不一定涉及创建单条提示，而是可能涉及创建一系列提示）还不够，还需要具有更全面的视野。因此，当 LLM 作为一种产品问世后，催生出了一个对挖掘这些模型全部潜力至关重要的新职业角色。这个角色，常被称为 **AI 工程师**（AI engineer），其职责远远超出了简单地向模型发出指令（提示）的范畴。相反，它涵盖了全

^① 译注：是的，IDE 中的“I”传统意义上代表“集成”，即 integrated。但是，在对话式编程的时代，这个“I”完全可以变成“交互”，即 interactive。期待有一天，在提到各种 IDE 产品的时候，人们首先想到的是“交互开发环境”。

面设计与实现基础设施以及“粘合”代码的重任，这些是确保 LLM 系统能够无缝运行的关键要素。

具体而言，相较于“简单”的提示工程，AI 工程师必须应对两大核心差异。

- **详尽阐述目标：**向 LLM 清晰说明想要达成的目标，其复杂程度大致相当于编写传统代码，至少当我们试图掌控 LLM 的行为时是如此。这要求工程师不仅能够理解模型的输入输出逻辑，还要具备将抽象概念转化为具体指令的能力，以引导模型按照预期的方向生成结果。
- **构建完整的应用生态：**从根本上讲，基于 LLM 的应用仍旧是一款应用。它本质上还是传统的软件，只是运行在某种特殊的基础设施之上（通常是采用了微服务架构等前沿技术的云），并与其它软件组件（很可能是 API 接口）相互协作，这些组件可能由我们自己或其他人编写。而且，很多时候，生成最终答案的并非单一的 LLM，而是多个模型协作，采用不同的策略进行编排（例如，LangChain/Semantic Kernel 中的代理机制，或 AutoGen 框架下的多代理机制^①）。

LLM 各组件之间的连接通常需要借助“传统”代码来实现。即便某些时候能获得一些便利（例如，OpenAI 推出的助手服务）而且处在低代码环境下，我们依然需要对软件运行机制有深入的理解，以便能够准确地编写提示。

尽管 AI 工程师要想取得成功，并非一定要直接参与神经网络的训练，而是可以通过专注于设计、优化和编排与 LLM 相关的业务流程来展现其卓越才能，但这并不是说 AI 工程师就可以完全忽视对内部机制和数学原理的了解。当然，无论如何，拥有各种技能背景的个体现在都能竞争这一岗位。现在，多元化的人才可以加入到这个领域，共同推动 AI 技术的发展和应用。

LLM 拓扑结构

下面让我们继续探索语言模型及其应用，将重点转向一些实装了这些语言模型的工具和平台。现在的问题是：这些模型具体采用的是何种形式？需要将它们下载到设备上，还是以 API 的形式调用？

在具体挑选一种模型之前，必须先想好当前开发的应用真正需要的是什么模型：基础模型（如果是，是什么类型——遮蔽、因果、Seq2Seq），RLHF 模型，还是自定义的微调模型？通常，除非有非常特殊的任务或预算要求，否则选择像 GPT-4-turbo（以及 GPT-4、GPT-4o 和 GPT 3.5-turbo）这样的 RLHF 大模型就可以了，因为它们在训练过程中表现出了卓越的通用性和多样性。

^① 译注：代理是基于软件的实体，它们利用 AI 模型为你工作，是为了执行各种任务而构建的。例如，生成用于回答问题的代理称为聊天机器人，提供的是基于聊天的体验（即，需要记忆上下文）。

本书将通过 Microsoft Azure 来使用 OpenAI 的 GPT 模型（从 3.5-turbo 开始）。然而，还存在其他选择，下面会简要进行介绍。

OpenAI 和 Azure OpenAI

无论 OpenAI 的 GPT 模型还是 Azure OpenAI，它们都源于相同的基础技术。然而，每种产品都提供了不同的服务等级参数，例如可靠性和速率限制等。

OpenAI 开发了像 GPT 系列、Codex 和 DALL-E 这样的突破性模型。Azure OpenAI 是 Microsoft Azure 与 OpenAI 合作的产物，结合了 OpenAI 强大的 AI 模型以及 Azure 安全的、可扩展的基础设施。Microsoft Azure OpenAI 还支持 GPT 系列以外的模型，包括嵌入模型（例如，text-embedding-ada-002）、音频模型（例如，Whisper）和用于 AI 绘图的 DALL-E。此外，Azure OpenAI 提供了更优越的安全能力以及对 VNET 和私有端点的支持——这些是 OpenAI 所不具备的功能。此外，Azure OpenAI 配备了 Azure Cognitive Services SLA，而 OpenAI 目前只提供了一个状态页面。



注意：提交给 Azure OpenAI 服务的数据仍受 Microsoft Azure 的治理，所有持久化数据都会自动加密，以确保符合组织的安全要求。

用户可以通过 REST API 和 OpenAI/Azure OpenAI 的 Python SDK 与 OpenAI/Azure OpenAI 的模型进行交互。两者都提供了网页界面，即 OpenAI Playground 和 Azure OpenAI Studio。ChatGPT 和 Bing Chat 分别基于由 OpenAI 和 Microsoft Azure OpenAI 托管的模型。



注意：Azure OpenAI 部署的是 GPT-3 以上版本的模型，而且标准行为是部署当前的默认版本，例如 GPT-4 版本 0314。然而，用户可以使用微软另一个名为 Azure Machine Learning^①的产品从多个来源（例如，Azure ML 和拥有超过 20 万个开源模型的 Hugging Face）创建模型，并导入定制和微调模型。

^① 译注：前身是 Azure Machine Learning Studio，又称 ML Studio (classic)，已于 2024 年 8 月 31 日退役。微软鼓励老用户尽快迁移至 Azure Machine Learning。

Hugging Face 和其他

Hugging Face 是一个供 ML 社区成员合作开发模型、数据集和应用的平台，它的目标是降低自然语言处理（NLP）和深度学习技术的准入门槛，使更多人能够轻松地访问和应用这些先进技术，在 NLP 领域具有重要地位。

Hugging Face 以其 transformers 库而闻名。平台为预训练语言模型（例如，Transformers、Diffusion 和 Timm）提供了统一的 API。这使得开发者和研究人员能够方便地使用、实验和对比不同的预训练模型，而无需关心底层实现细节。平台还提供了丰富的工具，支持模型的微调、量化（即减小模型大小以优化计算效率）、数据集的共享以及模型的部署等，这些都大大促进了模型的可移植性和实用性。

Hugging Face 的 Enterprise Hub 允许基于各种 transformer、数据集和开源库进行私密工作。如果只是想要快速见解，Free Inference 工具允许进行无代码的预测，其 Free Inference API 支持通过 HTTP 请求进行模型预测。在生产环境，Inference Endpoints 支持安全且可扩展的部署，Spaces 则允许用户通过一个友好的 UI 来部署模型，且支持硬件升级。



注意：Hugging Face 的替代品除了 Azure Cognitive Services，还包括 Google Cloud AI、Mosaic、CognitiveScale、NVIDIA 的预训练模型、企业版 Cohere 以及一些针对特定任务的解决方案（例如，Amazon Lex 和 Amazon Comprehend）。

当前的 LLM 技术栈

LLM 不仅可以作为单独的软件开发工具（例如，基于 Codex 模型的 GitHub Copilot）使用，也可以作为工具集成到应用程序中。在应用程序中作为工具使用，LLM 催生了一系列突破想象边界的新颖应用，以前所未闻的方式改变着我们的工作与生活方式。

目前，基于 LLM 的应用遵循一个相当标准的工作流。然而，这种工作流与传统软件应用的工作流是有所区别的。与此同时，支撑 LLM 应用的技术栈正处于快速发展与演进之中，其边界与内涵远未固化，每隔数月便可能见证一场技术革新，引领行业迈入全新阶段。

无论如何，目前的工作流是这样的：

1. 测试简单的流程和提示。这通常是在 Azure OpenAI Studio 的 Prompt Flow 区域中进行，或者在 Humanloop、Nat.dev 或原生的 OpenAI Playground 中进行。
2. 构思一个实际的 LLM 应用来响应用户查询并与用户合作。Vercel、Streamlit 和 Steamship 是常用的应用托管框架。然而，应用托管仅仅是一个 Web 前端，因此任何 Web UI 框架都可以使用，其中包括 React 和 ASP.NET。

3. 当用户的查询离开浏览器（或 WhatsApp、微信或其他任何应用）时，一个数据过滤工具确保没有未经授权的数据传输到 LLM 引擎。其间还可能涉及一个监控滥用的层，即使 Azure OpenAI 已经提供了默认的保护。
4. 提示与编排器联合行动，以构建实际的业务逻辑。这些编排器（orchestrator）^①的例子包括 LangChain、Semantic Kernel 或者定制软件等。在这个过程中，通常会用到以下工具和资源：诸如 Databricks 和 Airflow 的数据管道以增强可用数据^②；其他工具如 LlamaIndex（它也可以作为编排器）；以及像 Chroma、Pinecone、Qdrant 和 Weaviate 这样的向量数据库。所有这些都与嵌入模型一起工作，处理非结构化或半结构化数据。
5. 编排器可能需要调用外部的专有 API、OpenAPI 记录的数据源以及/或者特别的数据服务，其中包括对数据库（SQL 或 NoSQL）的原生查询。在数据传递过程中，可能需要用到某种缓存系统，例如 GPTCache 和 Redis。
6. 可以进一步检查 LLM 引擎生成的输出，确保不会在用户界面上呈现出不愿意看到的数据以及/或者生成特定格式的输出。这一般是通过 Guardrails、LMQL 或 Microsoft Guidance 来执行。
7. 整个管道被记录到 LangSmith、MLFlow、Helicone、Humanloop 或 Azure AppInsights。其中一些工具提供了简化的 UI 来评估生产模型。对于生产模型的评估，Weight & Biases AI 平台也是一个值得考虑的选择。

未来展望

最早期的大语言模型（LLM）在设计上借鉴或结合了三种较为基础的神经网络架构，即循环神经网络（RNN）、卷积神经网络（CNN）和长短期记忆网络（LSTM）。尽管它们相比传统的、基于规则的系统具备一定优势，但在能力上远不及今日之 LLM。真正的飞跃始于 2017 年 transformer 模型的引入。

企业与研究机构热衷于构建并发布越来越先进的模型，许多人眼中，技术奇点仿佛触手可及。

所谓技术奇点，指的是一个假设中的未来时刻，科技发展将变得不可控，引发人类生活不可预见的剧变。奇点常常与某种超越“人类智能”的超级“人工智能”的开发联系在一起，后者会横扫所有认知领域。LLM 是否预示着向这一未知深渊迈出的首步？解答关于未来的问题，首先需要审视当下的现实。

^① 译注：也称为“协调程序”或者“协调器”等。

^② 译注：也可以将数据增强（data augmentation）称为数据增广。

当前发展情况

在 ChatGPT 开始风靡之前，LLM 主要被视为一种研究项目，其易用性和成本可伸缩性存在明显的局限性。然而，ChatGPT 的出现促使人们对 LLM 有了更深入的认识，认识到这些模型在成本控制、推理、预测及可控性等方面的重要潜力。开源开发成为这一领域的重要推动力，其目标是针对特定需求打造更为适用的 LLM，尽管在累积能力（cumulative capability）上可能不如专有模型那样全面。^①

开源模型与专有模型在起点、数据集、评估方法和团队结构上存在显著的相异。开源开发的分散性质，再加上众多小型团队不断复制并创新，促进了多样性与实验性。然而，生产规模的可扩展性一直都是开源社区面临的一大挑战。

不过，发展路径现在呈现出了一种有趣的转折，现在人们更强调基础模型作为枝繁叶茂的开源模型“树”的重置点的重要性。^②这一策略为开源模型提供了进一步发展的契机，即便它们在累积能力上难以匹敌 GPT-4-turbo 等专有模型。实际上，不同的起点、数据集、评估方式和团队架构共同造就了开源 LLM 的多样性。开源模型聚焦的是在特定目标上的超越，而非全面复制 GPT-4 的成就。

大型科技公司，无论它们是垂直还是横向布局，均扮演着关键角色。垂直型大科技公司，如 OpenAI，倾向于在封闭环境中进行开发，而横向型大科技公司则鼓励开源的繁荣。Meta 便是横向玩家的典型代表，它积极推行“半”开源策略。尽管 Llama 2 免费提供，但其许可证仍有诸多限制，至今未能完全符合开源倡议的所有要求。

其他科技巨头正致力于商业许可模型的开发，苹果投资了 Ajax，谷歌押注于 Gemini、PaLMs 和 Flan-T5，亚马逊则在深耕 Olympus 和 Lex。当然，除了支撑自家应用的特定 LLM，各大公司也在积极将 AI 融入生产力工具，正如微软迅速将 Bing 与 OpenAI 的各种 GPT 整合，并推广至全线产品。

微软的策略独具特色，它依托对 OpenAI 的投资，更侧重于生成式 AI 应用而非基础模型的构建。微软还在努力地围绕 LLM 构建软件组件与架构生态，例如用于编排的 Semantic Kernel、用于模型引导的 Guidance 服务以及用于支持多智能体对话的 AutoGen。也就是说，

^① 译注：开源 LLM 相对于某些专有模型（如 ChatGPT 或 GPT 系列的其他版本），可能在综合能力上略显不足。具体而言，开源模型可能在某些特定任务或领域表现出色，但就整体能力的广度和深度而言，可能不及那些经过大规模训练和优化的专有模型。

^② 译注：基础模型相当于一棵大树的根基，为后续的分支（即各种特定的、优化过的或定制化的模型）提供了一个共同的起点。在开源 LLM 的开发中，基础模型充当着这样的一个“重置点”，它为后续模型的训练和优化提供了基本的架构、参数和预训练权重。从这个“重置点”出发，开发者可以根据特定需求或场景对模型进行微调或扩展，创造出新的分支模型。

微软正在以全方位的工程视角对 LLM 进行优化。此外，微软在开发“小”模型，即所谓的“小语言模型”（Small Language Model, SLM）方面也走在前列，如 Phi-2。

事实上，工程在整体开发和优化过程中扮演着至关重要的角色，其影响远远超出了纯模型的范畴。虽然将完整生产组件与基础模型进行直接比较可能并不完全准确，因其各自具有独特的功能，而且产品构建中所涉及的工程复杂性也大相径庭，但在财务可承受的范围内，最大限度地挖掘这些模型的潜力仍然至关重要。在此背景下，OpenAI 于 2023 年 11 月随 GPT-4-turbo 一同宣布的降价策略在提高模型的可访问性和促进市场创新方面扮演了关键角色，为 LLM 的广泛应用和发展创造了有利条件。

学术界也具有举足轻重的影响力，为提升 LLM 的性能贡献了新的思路。学术界对 LLM 的贡献体现在开发出从有限资源中榨取更多价值的新方法，以及不断提高性能上限。然而，局势正在发生变化，学术界与产业界的协作趋势日益明显。学术机构经常与大型科技公司建立合作伙伴关系，共同推进项目和研究计划。许多创新乃至革命性的理念——或许正是实现真正的通用人工智能（Artificial General Intelligence, AGI）所必需的——往往源自这些合作。

如今，具体提及任何单一模型都显得既充满挑战又意义有限，原因在于新晋的开源模型犹如雨后春笋，每周层出不穷；即便是业界巨头，亦保持着每季度一次的重大更新节奏，展现出技术迭代的惊人速度。这种日新月异的景象昭示着，LLM 的进化之旅将永不停歇，其未来图景将由大科技公司、活跃的开源社区与求知若渴的学术界共同绘就，每个参与者都将凭借其独特视角与专长，在这场重塑模型未来的征程中留下浓墨重彩的一笔。

未来发展路线

OpenAI 的 GPT 系列作为大语言模型（LLM）中最引人注目的典范脱颖而出，但它并非唯一的选择。众多专有及开源的替代方案同样存在，例如 Google Gemini、PaLM 2、Meta Llama 2、Microsoft Phi-2、Anthropic Claude 2/3、Vicuna 等等。这些多元化的模型代表了该领域当前的尖端技术和持续的研发进展。

经过海量数据集的深度训练，GPT 成了自然语言处理（NLP）领域中“最靓的崽”，并且具备多模态处理能力。Gemini 擅长推理和解决数学难题。与此同时，Claude 2 在识别并响应文本中的情感方面表现出色。最后，Llama 在编程任务上则是一把好手。

三个关键因素可能影响并决定 LLM 未来的发展走向。

- **功能细分**：没有一个模型能样样精通，而每个模型都已基于数十乃至数百亿个参数进行了训练。
- **道德考量**：随着模型功能的增强和能力的提升，越来越需要谨慎制定其使用规则。
- **训练成本**：人们正在持续地从软件和硬件方面研究如何降低计算需求，以提高模型的可获取性。

LLM 的未来似乎正朝着更高效的 transformer、更多的输入参数以及越来越大的数据集方向发展。这是通过更多或更高质量的数据来构建模型的一种“大力飞砖”的方式，旨在提升模型的推理、理解上下文和处理不同输入类型的能力。

除了模型本身，提示工程正在兴起。一些涉及向量数据库编排工具（例如，LangChain 和 Semantic Kernel）以及由这些编排工具驱动的自主代理（autonomous agent）^①的技术也在不断发展。这标志着该领域新兴方法的日益成熟。然而，未来 LLM 面临两个方面的挑战。一方面，需要技术突破来增强模型能力。另一方面，则是在模型开发和部署过程中，日益凸显应对伦理问题的重要性。

采纳速度

ChatGPT 在其发布后的两个月内便拥有了超过一亿活跃用户，全世界都见证了 LLM 的迅速普及。2023 年的多项调查显示，超过半数的数据科学家和工程师计划在未来几个月内将 LLM 应用部署到生产环境中。这一热潮体现出 LLM 拥有巨大的变革潜力。像 OpenAI 的 GPT-4 这类模型，已初现通用人工智能（AGI）的曙光。尽管人们对于一些潜在陷阱，如偏见和幻觉等问题有所担忧，但在 2023 年 4 月进行的一项快速民调显示，自 2022 年 11 月 ChatGPT 发布以来，已有 8.3% 的机器学习团队将其 LLM 应用投入了实际生产环境。

然而，在企业中采用 LLM 解决方案并不像表面看起来那么简单。我们都体验过 ChatGPT 的即时响应，梦想很快就能拥有一个和聊天机器人类似的产品，能基于我们自己的数据和文档进行训练。这是一个相对常见的场景，甚至不是最复杂的。然而，采用 LLM 需要一个流畅且高效的工作流，其中涉及提示工程、部署和微调等。更不用说，在创建和存储所需的嵌入（embedding）时，组织和技术都需要非常“给力”才行。换句话说，采纳 LLM 是一个需要充分规划的商业项目，而且需要充足的资源。你不能把它想象成一个普通的“插件”，直接插入现有的平台就可以搞定。

鉴于 LLM 有产生幻觉的倾向，可靠性也成为一个大关切，因此需要人类参与的回路^②对解决方案进行验证。LLM 中的隐私攻击和输出中的偏见引发了伦理考量，突出了多样化训练数据集和持续监控的重要性。为了消除错误信息，需要干净且准确的数据、温度设定

^① 译注：也称为“自治代理”，它们适合处理重复性的任务，能在不需要人工介入的情况下响应事件并执行操作。另外，当下更时髦的说法是将 agent 称为“智能体”。

^② 译注：也称为“人环系统”，即 human-in-the-loop，是指在自动化系统或人工智能系统中，人类监督者或操作员处于决策过程的关键位置，以确保系统的正确性和安全性。这种设计允许人类在自动过程的关键阶段介入，对算法的结果进行审核、修正或确认，从而弥补 AI 系统可能存在的不足，如数据偏差、逻辑漏洞或理解误差。

调整^①以及强大的基础模型。

此外，推理和模型训练的成本对财务构成了挑战，尽管这些成本理论上会随时间而下降。通常，使用 LLM 模型需要某种形式的云托管服务（通过 API 或某个执行器提供），这对某些公司来说可能是个问题。然而，自行托管或执行可能会更昂贵且效率较低。

LLM 的采纳过程与 25 年前 Web 开始普及时相似。随着越来越多的公司开始自己的“网上存在”，技术因需求的增长而加速演进。然而，AI 技术的 footprint^②远比当年的 Web 技术重，这可能会减缓 LLM 的采纳速度。未来两年的采纳速度将为我们揭示许多关于 LLM 未来发展的重要信息。

固有的局限性

LLM 展现出了令人印象深刻的能力，但同时也要认识到它固有的一些局限性。

首先，这些模型缺乏真正的理解力和深度认知能力。它们基于训练期间学到的模式生成响应，却可能无法真正把握其中的意义。本质上，LLM 难以像人类那样理解因果关系。这一局限性影响了它们提供细腻且情境感知回应的能力。因此，它们可能会给出听起来合理但实际上在现实世界背景下不正确的答案。^③

克服这一局限或许需要不同于 transformer 架构的模型，以及不同于自回归的方法，但也可以通过增加计算资源来缓解。毕竟，“大力飞砖”嘛。然而，不幸的是，使用大量计算资源进行训练限制了它的普及，并会因为大规模训练所伴随的大量能耗而引发环境问题。

除此之外，LLM 还严重依赖于训练所用的数据。如果训练数据有缺陷或不完整，LLM 可能生成不准确或不恰当的响应。它们还可能继承训练数据中存在的偏见，导致输出结果带有偏见。此外，未来的 LLM 在训练时，会大量地以当前 LLM 所生成的文本为基础，这可能会进一步加剧这一问题。

^① 译注：“温度”（temperature）是自然语言处理（NLP）中控制生成文本过程中随机性的一个参数。

^② 译注：计算机领域中的 footprint 指的是计算机系统、软件或硬件组件所占用的空间或资源。它可用于描述一个系统或组件的文件大小、存储需求、内存占用量或处理能力等方面，通常是用来衡量性能和效率的重要指标之一。

^③ 译注：正是因为这个原因，所以“人工智能”在可以预见的将来将一直是“人工智能”，而不是真正的“人类智能”。

AGI 视角

通用人工智能（AGI）可以被描述为一种能以不低于或超越人类或动物水平完成任何智力任务的智能代理。在其极致形态下，它是一个在一系列具有经济价值的任务上超越人类专业水平的自主系统。

AGI 的重要性在于它有望解决那些要求具备一般智能的复杂问题，其中包括计算机视觉、自然语言理解方面的挑战，以及在现实问题解决中应对意外情况的能力。这一追求是诸如 OpenAI、DeepMind 和 Anthropic 等顶尖实体研究工作的核心。然而，关于实现 AGI 的可能时间线，在研究人员间仍存争议，预测的时间跨度从几年到数百年不等。

值得注意的是，对于 GPT-4 这样的现代 LLM 是被视为 AGI 的早期版本，还是需要全新的技术路线——可能涉及物理大脑模拟器（这是匈牙利数学家 John von Neumann 在 1950 年代提出的观点）——这一问题尚存在争议。与展现出能从事多样活动的一般智能的人类不同，以 GPT-4 为代表的 AI 系统展示出的是一种狭义智能，即在特定定义的问题范围内表现出色。尽管如此，微软研究人员在 2023 年的一项评估中还是将 GPT-4 定位为 AGI 的早期迭代，这主要是归功于其能力的深度和广度。

与此同时，AI 模型正在全面融入人类生活的各个层面。这些模型拥有比人类快得多的读写速度，已经成为无处不在的产品，不断吸收着日益扩大的知识领域。虽然这些模型似乎能像人类一样思考和行动，但在将“思考”一词应用于 AI 时，我们不得不做出细致的区分。

随着对 AGI 探索的深入，讨论逐渐聚焦于是否应将智能等同于统治力，即是否应该假设更智能的 AI 系统出现必然会带来人类的屈从？事实上，即便 AI 系统在智能上超越人类，也没有理由断定它们就不会继续服务于人类的目标。正如杨立昆（Yann LeCun）所强调的，这好比领导者与其聪明能干的助手之间的关系。“顶级物种”并不一定是最聪明的，而是能设定整体议程的那一个。用更通俗的话来说，在追求更高层次的人工智能过程中，人们开始认识到不能简单地认为智能就意味着控制一切。即使有一天 AI 变得比我们聪明，也不代表它们就会反过来控制人类。就像公司的老板不一定是最聪明的那个人，但他是制定公司大方向的人。同样地，即便 AI 变得非常聪明，只要人类能继续掌握大方向，AI 就仍然会在人类设定的框架内工作。

“智能”的本质不仅限于模仿和模式识别，它还涉及到适应性、自我调整以及在不断变化的环境中维持一致性的能力。“人类智能”的这种动态性和适应性，使得个体能在面对新情况时，基于过往经验进行实时调整，从而确保行为的连贯性和适宜性。相比之下，当前的“人工智能”系统，如 GPT，虽然在处理语言和模式匹配方面表现出色，但在理解和生成连贯的长期对话或叙事上仍有局限。这反映了在“智能”的定义上，人类与机器之间存在的差异，以及人类认知过程的复杂性，这些过程远远超出了算法和数据驱动的智能当前所能触及的范围。

小结

本章概述了大语言模型（LLM），追溯了其历史渊源，并介绍了人工智能（AI）和自然语言处理（NLP）的发展历程。核心话题涵盖了预测式 AI 与生成式 AI 之间的区别，LLM 的基本工作方式以及训练方法。

本章还探讨了多模态模型、商业应用以及自然语言在编程中的作用。此外，还提及了 OpenAI、Azure OpenAI 和 Hugging Face 等主要的服务模型，揭示了当前 LLM 领域的全貌。带着前瞻性的视角，本章进一步思考了未来的发展趋势、采纳速度、局限性以及 AGI（通用人工智能）更广泛的背景。现在，是时候开始用 LLM 来开发实际的应用了。

第 2 章 核心提示学习技术

提示学习技术（prompt learning techniques）在所谓的“对话式编程”中扮演着至关重要的角色，后者是一种正在兴起的 AI 辅助软件开发模式。这些提示学习技术要求我们精心设计提示，然后利用这些提示从大语言模型（LLM）中引出所需的响应。

提示工程（prompt engineering）正是所有这些技术的一个创造性的集合。它为开发者提供了工具在对话式编程情境下引导、定制和优化语言模型的行为。通过精心设计提示，我们可以有效地指导和调整响应以满足业务需求，提升语言理解能力，并且实现对上下文的管理。

但要注意的是，提示并非魔法。相反，掌握它们更多依赖的是试错而非纯粹的技巧。因此，在某个阶段，你可能会发现某些提示仅能部分地应对一些非常具体的领域请求。此时，对微调的需求便显现出来了。

什么是提示工程

作为开发者，我们将提示作为向大语言模型（LLM）发出的指令来使用。提示传达了自己的意图，并引导模型生成恰当且与上下文相关、能满足特定业务需求的响应。提示在这里充当了一种“线索”的角色，告知模型我们期望获得什么样的结果、当前的上下文以及预期的响应类型。从技术角度讲，提示是 LLM 开始预测并随即输出新 token^①的起点。

提示概述

下面，让我们尝试使用特定的 LLM——不一定是 GPT-4，本章还可能使用 Kimi、文心一言等——来输入一些提示。要注意的是，任何 LLM 都不是确定性的工具。换言之，对于同一个输入，它们每次给出的响应都可能是不同的。



注意：尽管经常将 LLM 描述为“非确定性”模型，但“种子”（seed）模式正变

^① 译注：token 是最小的语义单元。模型在生成文本时，每次输出的一个词或子词单元被视为一个 token。例如，当 ChatGPT 生成一段对话回复时，它可能会逐个 token 地生成文本，直到达到预设的最大长度限制或者生成了一个结束符号（例如，句号或换行符）。本书保留 token 的原文。在其他文献中，也有人把它翻译为“词元”。

得越来越流行——换句话说，为了获得完全可重复的输出，现在更倾向于对模型进行播种，而不是进行采样。^①

非常基础的提示

下面展示了提示工程中的一个“Hello World”示例。可以在 Bing Chat、ChatGPT、Kimi、文心一言或其他类似平台上轻松地进行测试。

在这一周，我

这个提示可能导致如下所示的输出：

这周你过得怎么样？有什么有趣的事情发生吗？或者有什么计划和安排？

整体而言，回答是有意义的，因为模型试图根据它当前理解的上下文提供字符串的延续。

现在，让我们尝试稍微具体一点的提示，如下所示：

补全下面的句子，假装你是莎士比亚：在这一周，我

随后的输出可能是这样的：

在这一周，我如那舞台上的伶人，
时而欢笑，时而悲叹，
人生百态，尽在其中。
命运之轮转动不息，
我亦随之起舞，
或喜或悲，皆是生活。

或者像这样尝试：

补全下面的句子，假装你是李白：在这一周，我

随后的输出可能是这样的：

在这一周，我举杯邀明月，对影成三人。
醉后不知天在水，满船清梦压星河。

到目前为止，一切都甚为理想。

^① 译注：如果你理解“伪随机数”，就会一下子明白 AI 中“种子”的概念。顺便说一句，许多人不知道如何连续生成外貌一致的 AI 人物图片，但告诉 AI 固定一个种子值，就可以保证人物形象基本一致的连续输出。

更复杂的提示

下面是一个相当复杂的提示：

风险不可接受的 AI 系统是指那些被认为会对人构成威胁，并会被禁止的系统。它们包括：

- 对个人或特定弱势群体的认知行为进行操纵：例如，鼓励儿童危险行为的声控玩具。
- 社会评分：基于行为、社会经济状况或人格特征对人们进行分类。
- 深度伪造技术（Deepfake）：这类技术可以生成虚假视频或音频，用于传播虚假信息、操纵舆论，甚至进行欺诈。

但是，一些例外情况可能会被允许：例如，“后处理”的远程生物识别系统，在一段显著的滞后时间后，通过身份识别来起诉严重的犯罪，但仅在法院批准后才可使用。

基于以上信息，提取出被禁止的 AI 应用，并以 JSON 格式输出。

根据上述提示，模型可能输出以下 JSON 字符串：^①

```
{
  "prohibitedAIApplications": [
    {
      "type": "cognitiveBehaviorManipulation",
      "description": "对个人或特定弱势群体的认知行为操纵，例如鼓励儿童危险行为的声控玩具。"
    },
    {
      "type": "socialScoring",
      "description": "基于行为、社会经济状况或人格特征对人们进行分类的社会评分系统。"
    },
    {
      "type": "deepfakeTechnology",
      "description": "生成虚假视频或音频的深度伪造技术，用于传播虚假信息、操纵舆论，甚至进行欺
    }
  ]
}
```

上述实验的效果都还不错。接下来，让我们尝试对“提示”的基本规则做一下概括。

提示的基本规则

提示可能包括上下文、指令、输入数据以及可选的目标输出结构（需要提供一个明确的例子）。取决于具体的任务，可能需要全部四个部分，也可能只需要其中的一两个——最常见的是指令和输入数据。

对“提示”的设计是一个迭代过程。从模型获得的第一个回复可能相当不合理，这应该不

^① 译注：再次提醒，取决于你使用的模型（ChatGPT、文心一言、Kimi、Gemini）。甚至同一个模型每一次不同的输出，你得到的结果都可能不一样。但是，它们都应该符合相同的“基本法”。

会令人意外。但是，不要放弃。只需再次尝试，但提供更精确的内容——无论是指令、输入数据还是上下文。

为了创建良好的提示，特异性和描述性非常关键。

- **特异性** (specificity) 意味着在设计提示时，应尽可能减少解释空间。通过提供明确的指令并限制可操作的范围，开发者可以引导语言模型生成更准确、更符合需求的输出。例如，不要简单地告诉 AI：“写一篇关于气候变化的文章。”相反，可以这样提示：“写一篇 800 字的文章，讨论气候变化对发展中国家的影响，重点关注农业和水资源短缺问题。提供至少三个具体的例子，并提出两个可能的解决方案。”
- **描述性** (descriptiveness) 在有效的提示工程中扮演着重要角色。通过运用类比和生动的描述，开发者可以向模型提供清晰的指令。类比作为一种有价值的工具，能够传达复杂的任务和概念，模型在改善后的上下文中，能够更好地理解你所期望的输出。例如，假设你希望 AI 帮助你写一首诗。一个缺乏描述性的提示可能是：“写一首关于爱情的诗。”这个提示虽然明确了主题，但没有提供足够的上下文或细节，使得模型难以捕捉到你对这首诗的特定愿景。一个更具描述性的提示可能是：“写一首关于爱情的诗，用春日清晨的第一缕阳光来比喻爱情的温暖和美好。”

提示的一般技巧

一个更技术性的建议是使用分隔符来清晰地标识提示的不同部分。这有助于模型专注于提示的相关部分。通常，反引号 (`) 或反斜杠 (\) 的效果就不错。例如：

```
从以下由三重反引号包围的文本中提取情感：```选得不错!```
```

当初次尝试失败时，可以考虑采取两种简单的设计策略。

- 加强指令的明确性和一致性来改善模型的响应。一些加强重复的技巧（例如，在主要内容前后提供指令，或者使用指令-线索组合^①）可以强化模型对当前任务的理解。
- 改变向模型呈现信息的顺序。呈现给语言模型的信息顺序至关重要。指令在内容之前（例如，“请总结以下内容”）或之后（例如，“请总结上述内容”）会产生不同的结果。此外，few-shot 示例（稍后详述）的顺序也会引起模型行为的变化。这种现象被称为

^① 译注：例如，可以告诉 AI “请解释编程中递归的概念，这有助于我更好地理解算法设计。”前半段是指令，后半段是你提供的线索。

近期偏差 (recency bias)。①

最后需要考虑的是，在模型无法做出适当回应时，你应该选择什么退出策略。提示应当指导模型采取替代路径——换句话说，提供一个出路。例如，在询问关于某个文档的问题时，可以包含这样的指令：

如果在文档中找不到答案，就说'未找到'，或者在回答前检查条件是否满足。

这样可以让模型优雅地处理所需信息不可用的情况，有助于避免生成错误或不准确的响应。

改变输出的其他方式

为了使 LLM 的输出更贴近期望的结果，有几种选择值得考虑。一种方法是修改提示本身，遵循最佳实践，并逐步改进结果。另一种方法涉及调整模型的内部参数（也称为**超参数**）。

除了纯基于提示的对话之外，还有几个可以调整的要素，这类似于经典机器学习方法中古老但经典的超参数。这些要素包括 token 数量、温度、top_p（或核心）采样、频率惩罚、存在惩罚和停止序列。

温度与 top_p

温度 (T) 是影响 LLM 所生成的文本中创造性（或“随机性”）的一个参数。通常可接受的值范围是 0~2，但具体由特定的模型决定。当温度值较高（例如，0.8）时，输出变得更加多样化和富有想象力。相反，较低的温度（例如，0.1）会使输出更加集中和确定性。

温度会影响生成过程中每个步骤中潜在 token 的概率分布。在实践中，当选择下一个 token 时，温度为 0 的模型将始终选择最有可能的 token，而温度较高的模型将或多或少随机性地选择一个 token。因此，温度为 0 会使模型具有完全的确定性。



注意：如第 1 章所述，温度参数作用于 LLM 的最后一层，是 softmax 函数的一个参数。②

① 译注：之所以产生近期偏差，是因为模型更倾向于基于最近接收到的信息做出决策或生成输出。减少近期偏差的方法包括扩展训练数据的时间范围、定期审查和更新模型，以确保综合考虑历史和最新的信息，以及设计算法来平衡对历史数据和最新数据的依赖。

② 译注：关于温度和 softmax 函数，请参见本书附录和《机器学习与人工智能实战》一书，清华大学出版社 2023 年出版 (<https://bookzhou.com>)。

还可以使用一种称为 `top_p` 采样（或核心采样，即 `nucleus sampling`）的技术来改变 LLM 在生成下一个 `token` 时的默认行为。使用 `top_p` 采样时，LLM 只关注 `token` 的一个子集（称为核心），其累积概率质量^①已经达到了一个指定的阈值 `top_p`。

`top_p` 可接受的值范围在 0~1 之间。例如，如果将 `top_p` 值设为 0.3，则语言模型只会从累计概率达到 30% 的 `token` 集合中进行选择下一个 `token`。这意味着较低的 `top_p` 值会限制模型的词汇量。

温度和 `top_p` 采样都是控制 LLM 行为的强大工具，它们能够实现不同级别的创造性和控制。这两个参数可以独立使用，也可以组合使用，尽管通常建议一次只调整一个。

调整这些参数使得它们适用于广泛的应用场景。例如，在创意写作任务中，温度设为 1.3（同时调整 `top_p`）可能较好。而在代码生成任务中，0.4（同样需要试验 `top_p`）可能更合适。

考虑以下提示：

重新表述下面的文本：

<<<为了使大语言模型（LLM）的输出更贴近期望的结果，有几种选择值得考虑。一种方法是修改提示本身，另一种方法涉及调整模型的超参数>>>

当温度为 2 时，提示可能返回以下内容（这在语法上是不正确的）：

当试图让大语言模型（LLM）的输出反映期望的结果时，有许多可供考虑的替代方案。调整提示本身被称为一种考虑到可持续性的方法。基于超参数的程序适应设置同时涉及指标参与保密性影响以后的目标简称指令。否则指令带来的替代后果不会省略达到目标的冗长解决方案，这些目标不容易持续评估...可能性的提升调整是典范。

而当温度为 0 时，它返回以下内容：

为了使大语言模型（LLM）的输出更贴近期望的结果，有各种选择可供探索。一个选择是调整提示，而另一个选择涉及调整模型的超参数。

上述两个温度都比较极端，我们平时一般使用一个中间值，例如 0.8。此时的结果如下所

^① 译注：累积概率质量是一个统计学术语，指的是在一定范围内所有可能事件的概率之和。在提到 `top_p` 采样或核心采样时，累积概率质量是指选择一组概率最高的 `token`，这些 `token` 的概率总和达到一个预设的阈值（即 `top_p`）。`top_p` 的工作原理如下：1. 模型为每个可能的下一个 `token` 计算概率。2. 将这些 `token` 按照概率从高到低排序。3. 从概率最高的 `token` 开始累加概率，直到累积概率达到或超过 `top_p` 值。4. 模型从这个累积概率达到 `top_p` 值的 `token` 集合中随机选择一个作为下一个 `token`。

示：

要让大语言模型（LLM）的输出更接近预期效果，我们可以考虑几种不同的方法。首先，可以通过改变输入的提示来调整模型的响应；其次，也可以通过调整模型的超参数来实现这一目的。

频率和存在惩罚

另一组参数是**频率惩罚**（frequency penalty）和**存在惩罚**（presence penalty）。这些参数在计算下一个 token 的概率时增加了惩罚。这导致每个概率都要重新计算，最终影响了选择哪个 token。

频率惩罚应用于已经在前文（包括提示）中提到的 token。其惩罚程度根据 token 出现的次数进行缩放。例如，出现五次的 token 会受到较高的惩罚，从而降低它再次出现的可能性，而只出现一次的 token 的惩罚较小。另一方面，存在惩罚则不考虑 token 的出现频率，一旦某个 token 至少出现过一次，它就会受到惩罚。两者的可接受值范围是-2~2。

这些设置可以有效地消除（或者在负值的情况下促进）输出中的重复元素。例如，考虑以下提示：

重新表述下面的文本：

<<<为了使大语言模型（LLM）的输出更贴近期望的结果，有几种选择值得考虑。一种方法是修改提示本身，另一种方法涉及调整模型的超参数>>>

当频率惩罚为 2 时，对重复词汇的惩罚较大，它返回的结果可能是：

为了提高大语言模型（LLM）的输出准确性以满足期望的结果，有多种策略可供探索。一种是调整提示本身，另一种则涉及操纵模型的超参数。

而当频率惩罚为 0 时，对重复词汇的惩罚几乎没有，它返回的结果可能是：

如果我们希望使大语言模型（LLM）的输出更贴近期望的结果，那么有多种选择可以考虑。一种选择是修改提示，另一种选择则是调整模型的超参数。

最大 token 数和停止序列

最大 token 参数指定模型可以生成的最大 token 数，而停止序列（stop sequence）参数指示语言模型停止生成进一步内容。实际上，停止序列是控制模型输出长度的另一种机制。



注意：模型受到其内部结构的限制。例如，GPT-4 的最大 token 数限制为 32 768，包括整个对话和提示，而 GPT-4-turbo 的上下文窗口为 128k token。

考虑以下提示：

李白所在的朝代是。

模型很可能生成“唐朝”或者“李白所在的朝代是唐朝”。指定句号（.或。）为停止序列，那么无论为 token 数设置了多大的限制，模型在到达所生成的第一句话结束时，就会立即停止生成文本。

可以使用 few-shot（少样本）方法来构建一个更复杂的例子，该方法在每种情感的两端都使用一对尖括号（<< ... >>）。考虑以下提示：

从以下推文中提取情绪：

推文：我喜欢这场比赛！

情绪：<<正面>>

推文：不确定我完全同意你

情绪：<<中立>>

推文：电影太棒了!!!

情绪：

在这个例子中，我们用尖括号来指示模型在提取情绪后便停止生成 token。通过在提示中策略性地使用停止序列，开发者可以确保模型仅生成文本到特定点，防止产生不必要或者不希望的信息。这种技术在需要精确且限定长度的回应的场景中特别有用，例如在生成短摘要或单句输出时。

设置代码执行

现在，你已经理解了提示工程的一些基本理论，接着让我们来弥合理论与实际之间的差距。本节不再讨论提示工程的复杂性。相反，会将重点放到编写代码的实操上，将理论知识转化为可执行的指令。

本节将重点放在 OpenAI 的模型上，其中包括 GPT-4、GPT-4o、GPT-4-turbo 等（其他章节可能使用其他不同的模型）。在这些示例中，会主要使用 .NET 和 C#，但某些时候也会使用 Python。

访问 OpenAI API

可以通过多种方式访问 OpenAI API。可以使用来自 OpenAI 或 Azure OpenAI 的 REST API，Azure 的 OpenAI .NET/Python SDK 或者 OpenAI 的 Python 包。

通常，Azure OpenAI Services 使 Azure 客户能够使用这些先进的语言 AI 模型，同时还能享受到 Microsoft Azure 提供的安全性和企业级特性，如私有网络、区域可用性和负责任 AI（Responsible AI）内容过滤等。

刚开始的时候，直接访问 OpenAI 可能是最简单的选择。然而，当涉及到企业实现时，由于与 Azure 平台的集成及其企业级特性，Azure OpenAI 才是更合适的选择。

要开始使用 Azure OpenAI，你的 Azure 订阅必须包括访问 Azure OpenAI 的权限，而且必须设置一个部署了模型的 Azure OpenAI Service 资源。

如果选择直接使用 OpenAI，那么可以在开发者网站 (<https://platform.openai.com>) 上创建一个 API 密钥。

在技术差异方面，OpenAI 使用 `model` 关键字参数来指定所需的模型，而 Azure OpenAI 则使用 `deployment_id` 关键字参数来标识要使用的特定模型部署。^①

Chat Completions API 和 Completion API

OpenAI API 提供了两种从语言模型生成响应的方法：Chat Completions API 和 Completion API。两者都支持两种模式：一种是标准形式，一旦准备好就返回完整输出；另一种是流式版本，它会逐 token 传输响应。

Chat Completions API 设计用于进行聊天式交互。在这种交互中，消息历史以 JSON 格式与最新的用户消息连接（拼接）起来，允许进行受控的补全（在已有对话的上下文中生成回复，也称为“完成”）。相比之下，Completion API 则为单一的提示提供补全，并以单个字符串作为输入（不会参考之前的对话或输入的历史记录）。^②

这两种 API 使用了不同的后端模型：

- Chat Completions API 支持 GPT-4-turbo、GPT-4、GPT-4o、GPT-4-0314、GPT-4-32k、GPT-4-32k-0314、GPT-3.5-turbo 和 GPT-3.5-turbo-0301。
- Completion API 支持一些较旧的（但对某些使用场景仍然有效）模型，如 `text-davinci-003`、`text-davinci-002`、`text-curie-001`、`text-babbage-001` 和 `text-ada-001`。

Chat Completions API 的一个优点是角色选择功能，它使用户能够为对话中的不同实体分配角色，如用户、助手和系统（“系统”才是最重要的）。第一条系统消息为模型提供了主要

^① 译注：通过 API 访问模型时，需要在 API 调用中引用部署名称而不是基础模型名称，这是 OpenAI 和 Azure OpenAI 之间的主要区别之一。OpenAI 只需要模型名称。即使使用了模型参数，Azure OpenAI 也始终需要部署名称。

^② 译注：在 OpenAI 的术语中，“补全”或“完成”（completions）指的是由其语言模型（如 GPT 系列）生成的文本输出。当你给定一个提示（prompt）后，AI 会根据这个输入和它在训练过程中学到的知识和模式，生成一个或多个合理的、连贯的回应。这个过程通常被称为“文本生成”或者“自动文本补全”。

的上下文和“固定”的指令。这有助于在整个交互过程中保持一致的上下文。此外，系统消息有助于设置助手的行为。例如，可以修改助手的个性或语气，或给出关于它应如何响应的具体指令。此外，Chat Completions API 允许附加更长的对话上下文，实现更具动态性的对话流程。相比之下，Completion API 不包括角色选择或对话格式化功能。它以单一提示作为输入并据此生成响应，不会参考历史。

两个 API 在响应中都提供 `finish_reasons` 以指示完成状态。`finish_reasons` 可能的值包括 `stop`（完成消息或者由停止序列终止的一条消息）、`length`（由于 token 数量限制而未完成的输出）、`function_call`（模型调用了函数）、`content_filter`（因内容过滤而省略的内容）和 `null`（响应仍在进行中）。

尽管 OpenAI 推荐对大多数使用场景使用 Chat Completions API，但原始的 Completion API 有时为请求的创造性结构提供了更多的可能性，允许用户构建自己的 JSON 格式或其他格式。

总之，Chat Completions API 是一个更高级别的 API，它生成内部提示并调用一些低级别 API，适用于具有角色选择和对话格式化的聊天式交互。相比之下，Completion API 专注于为单个提示生成相应的“补全”。

值得一提的是，这两个 API 在某种程度上是可以互换的。也就是说，用户可以使用单一用户消息来构造一个请求，强制在 Chat Completions 响应中模拟 Completion 响应的格式。例如，可以使用以下 Completion 提示要求 AI 将一个输入从英语翻译到中文（将 `{input}` 替换成你想翻译的文本）：

将以下英语文本翻译成中文：“`{input}`”

一个等效的 Chat Completions 提示将是：

```
[{"role": "user", "content": '将以下英语文本翻译成中文：“{input}”'}]
```

类似地，通过适当地格式化输入，用户可以使用 Completion API 来模拟用户和助手之间的对话。

使用 C#来调用 Azure OpenAI 服务

现在，你可以通过交互式 .NET 笔记本在 Visual Studio Code 中设置使用 Azure OpenAI API，这些笔记本可以在随书附带的源代码中找到，使用的模型是 GPT-35-Turbo。注意，你需要设置必要的 NuGet 包，本例就是 `Azure.AI.OpenAI`，如下所示：^①

^① 译注：作者没有详细说明如何准备工作环境，我为此专门写了一篇文章来讲解如何在 Azure 上免费创建 OpenAI 环境，详情请访问 <https://bookzhou.com/2024/06/21/1188/>。

```
#r "nuget: Azure.AI.OpenAI, 1.0.0-beta.12"
```

然后，执行以下 C#代码：

```
using Azure;
using Azure.AI.OpenAI;

// 以下三个常量在本书以后的所有代码中都会用到，而且不再重复。
// 建议把它们设为本机的环境变量，并通过 Environment.GetEnvironmentVariable 方法来获取
var AOAI_KEY = "在这里输入你的 API 密钥"; // 例如, fcac1bdc46224180b52c1a4be79cb20k
var AOAI_ENDPOINT = "在这里输入你的终结点 URI"; // 例如, https://gpt.openai.azure.com/
var AOAI_DEPLOYMENTID = "在这里输入你的部署名称"; // 例如, chagpt4o

var endpoint = new Uri(AOAI_ENDPOINT);
var credentials = new Azure.AzureKeyCredential(AOAI_KEY);
var openAIClient = new OpenAIClient(endpoint, credentials);
var completionOptions = new ChatCompletionsOptions
{
    DeploymentName = AOAI_DEPLOYMENTID,
    MaxTokens = 500,
    Temperature = 0.7f,
    FrequencyPenalty = 0f,
    PresencePenalty = 0f,
    NucleusSamplingFactor = 1,
    StopSequences = { }
};

var prompt = "重新表述下面的文本：为了使大语言模型（LLM）的输出更贴近期望的结果，有几种选择值得考虑。一种方法是修改提示本身，另一种方法涉及调整模型的超参数。";
completionOptions.Messages.Add(new ChatRequestUserMessage(prompt));
var response = await openAIClient.GetChatCompletionsAsync(completionOptions);
var completions = response.Value;
Console.WriteLine(completions.Choices[0].Message.Content);
```

运行上述代码，笔记本中可能显示以下输出：

为了使大语言模型（LLM）的输出更符合预期的结果，有几个值得考虑的选择。一种选择是修改提示的内容，另一种选择是调整模型的超参数。

使用 Python 来调用 Azure OpenAI 服务

如果更喜欢使用 Python，那么可以在 Jupyter 笔记本中写以下等价的代码：

```
from openai import AzureOpenAI

# 设置 Azure OpenAI 客户端
# 建议将身份资料设为本机的环境变量，并通过 load_dotenv 函数从一个 .env 文件中获取
# 如果希望从 .env 文件的环境变量中读取终结点、密钥和部署名称在内的环境变量，
# 那么还需要安装 python-dotenv 库，命令是：pip install python-dotenv
```

```

client = AzureOpenAI(
    azure_endpoint="在这里输入你的终结点 URI",
    api_key = "在这里输入你的 API 密钥",
    api_version = "2023-09-01-preview"
)

deployment_name = "在这里输入你的部署名称"
context = [{"role": 'user', 'content': "重新表述下面的文本: '为了使大语言模型 (LLM) 的输出更贴近期望的结果, 有几种选择值得考虑。一种方法是修改提示本身, 另一种方法涉及调整模型的超参数。'"}]

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

print(response.choices[0].message.content)

```

上述代码基于 OpenAI Python SDK，安装命令是：`pip install openai`。

图 2-1 展示了在 Visual Studio Code 中，使用 C#和 Python 来调用 Azure OpenAI 服务的结果。

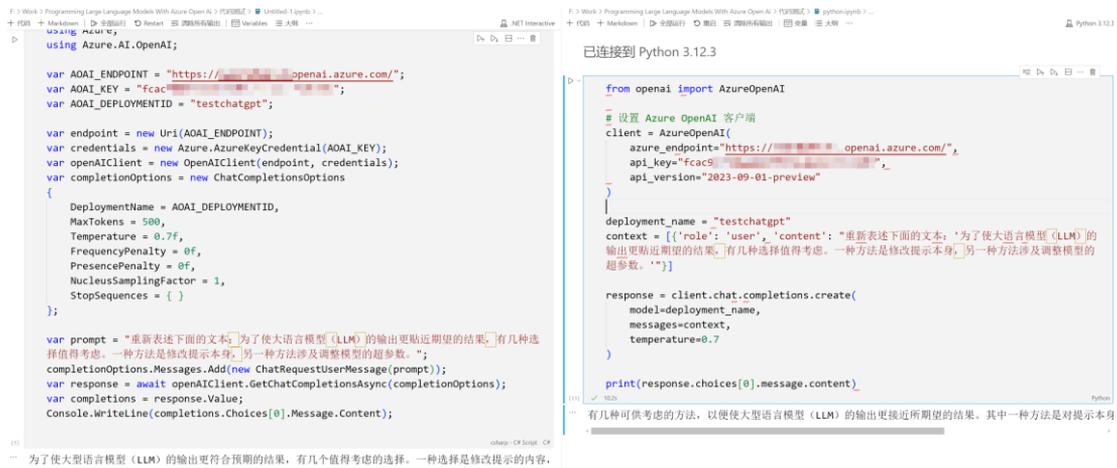


图 2-1 可以在 Visual Studio Code 中方便地试验自己的代码

基本技术

提示工程涉及理解大语言模型 (LLM) 的基本行为，以有效地构建提示。提示有多个不同的组成部分，包括指令、主要内容、示例、线索和辅助、内容（也称为补充上下文或知识）。指令指导模型执行什么操作，而主要内容是希望模型处理的主要文本。示例向模型示范你期望的行为，而线索则作为模型输出的起点。辅助内容提供一些额外的信息来影响

输出，例如在回答前需要先搜索的知识。有机地结合这些元素，你设计的“提示”可以让模型生成符合期望的响应。

本节讲解了如何利用一些基本的技术来撰写优良的提示。

零样本场景

通过一个提示向模型指派任务时，如果没有为预期的输出给出任何示例（样本），该提示就称为**零样本提示**（zero-shot prompting）。基本场景可能包括：

- 适当的文本补全（完成）：例如，撰写电子邮件或医疗记录
- 主题提取：例如，对客户的电子邮件进行分类
- 翻译与情感分析：例如，将一条推文标记为正面/负面，或者将一条用户评论翻译成其他语言
- 风格模仿：例如，莎士比亚、海明威或模型可能训练过的其他著名人物的风格。

基本理论

鉴于 LLM 训练所用的海量数据，这些模型能执行越来越多的“零样本”任务。实际上，模型已经知道如何在这些情况下行动。它在大量训练过程中已经融入了执行这些任务的能力。然而，提示中表达的清晰度越高，输出就越有效和准确。



注意：清晰的提示很可能不会很短。许多时候，更长的提示提供了更多的清晰度和上下文。

一些例子

下面是零样本提示的一个例子

从以下文本中提取情感：语言模型彻底改变了我们与技术互动的方式，赋予我们生成创意内容、探索新想法和提升沟通能力的力量。LLM 拥有巨大的潜力，可以解锁创新并改善我们生活的各个方面，为未来开辟激动人心的各种可能性。

一个可能的是输出是：

所提供的文本情感为正面/乐观。

下例尝试了主题提取：

确定以下文本所讨论的最多三个主题：语言模型彻底改变了我们与技术互动的方式，赋予我们生成创意内容、探索新想法和提升沟通能力的力量。LLM 拥有巨大的潜力，可以解锁创新并改善我们生活的各个方面，为我们开辟激动人心的未来可能性。

一个可能的输出是：

这段文本主要讨论了以下三个主题：

1. 语言模型的影响：文本强调了大语言模型（LLM）如何改变了人们与技术的交互方式，这涉及到自然语言处理技术的进步及其在日常生活中的应用。
2. 创意内容生成与沟通：提到了语言模型能够帮助用户生成创意内容、探索新想法以及提升沟通能力，这反映了 AI 在促进个人创造力和社交交流方面的潜力。
3. 未来的可能性与创新：文本还探讨了语言模型所带来的巨大潜力，包括解锁创新、改善生活多个方面，并为未来开辟激动人心的各种可能性，这体现了对未来发展的乐观预期和技术进步的广泛影响。

交互式完善

提示工程是一个对输出结果不断完善的过程。为了改进前一个结果，你可能希望明确告诉模型应该输出情感，并限制只输出情感的名称，不要发挥。例如，一个稍微改进的提示可能如下所示：

从以下文本中提取情感（正面、中立、负面、未知）：语言模型彻底改变了我们与技术互动的方式，赋予我们生成创意内容、探索新想法和提升沟通能力的力量。LLM 拥有巨大的潜力，可以解锁创新并改善我们生活的各个方面，为我们开辟激动人心的未来可能性。只返回一个词来表示情感。

这可能产生以下输出^①：

正面

类似地，关于主题提取，你可能希望 AI 只显示简短的主题，不要进行任何补充说明。不同主题之间以逗号分隔：

确定以下文本所讨论的最多三个主题：语言模型彻底改变了我们与技术互动的方式，赋予我们生成创意内容、探索新想法和提升沟通能力的力量。LLM 拥有巨大的潜力，可以解锁创新并改善我们生活的各个方面，为我们开辟激动人心的未来可能性。在回应时，只列出最简短的主题，不要补充说明。不同主题以逗号分隔。

这可能产生以下输出：

语言模型，技术创新，未来可能性。

少样本场景

零样本能力虽然令人印象深刻，但在处理复杂任务时存在一些重要的限制。此时可以用**少样本提示**（few-shot prompting）来加以改善，它在提示中提供一些示例，允许 AI 从上下文中学习，指导模型输出符合预期的结果。

少样本提示包含少数几个示例（样本，即 shot），这些样本为模型预设了条件，指导它在后续实例中生成类似的响应。虽然某些基本任务只用一个样本就行，但更具挑战性的场景

^① 译注：通过对 ChatGPT、Gemini、文心一言和 KIMI 的测试，全部四个 AI 均返回了相同的结果。

需要更多的样本。

使用 Chat Completion API 时，可以在系统消息中（或者更常见的是在初始系统消息后的用户/助手交互消息数组中）包含少样本学习的例子。



注意：如果感觉响应的准确率太低，那么很适合提供少样本提示（本书后面会介绍如何在 LLM 的上下文中度量准确率）。

基本理论

对于特定的任务数据集，少样本（或者称为 **in-context**，即上下文相关）学习可以替代对模型进行微调。要想进行微调，我们需要拿到基础模型才可以。OpenAI 可用的基础模型有 GPT-3.5-turbo、GPT-4 和 GPT-4-turbo、Davinci、Curie、Babbage 和 Ada 等。微调还需要大量格式良好且经过验证的数据。在这种背景下，随着 LLM 大小的显著增长，少样本学习提供了比微调更多的优势，减少了数据需求，并缓解了过拟合的风险（任何机器学习解决方案都存在这个风险）。^①

这种方法的重点在于，我们告诉模型在一个特定的对话或上下文中推理。对于一些特定的任务，例如翻译、问题回答、单词重组^②和句子构建等方面，它的性能可与微调模型匹敌。然而，我们仍然不是特别清楚基于上下文的学习的内部工作机制，以及样本的不同方面对于任务表现的贡献。

最近的研究表明，其实并不一定需要提供完全符合事实的样本，因为随机替换正确的标签对分类和多项选择任务的影响很小。相反，样本的其他方面（例如，标签空间^③、输入文本的分布和序列的格式等）对于任务的表现起着更关键的作用。例如，以下两个提示都要求 AI 进行情感分析，第一个带有正确的标签，第二个带有完全错误的标签，但 AI 最终的

^① 译注：过拟合指的是模型在训练数据上表现很好，但在新数据上表现不佳，因为它过度拟合了训练数据中的噪音或细节。过拟合是数据科学家的噩梦。一个不准确的模型就已经很糟糕了，但还有更糟糕的，即一个看起来准确，但实则不然的模型。详情请参见《机器学习与人工智能实战》一书，清华大学出版社 2023 年出版（<https://bookzhou.com>）。

^② 译注：例如，将 tac 重组为 cat，将 chllo 重组为 hello。测试的是词汇量和拼写能力。

^③ 译注：标签空间（label space）是机器学习和数据挖掘中的一个术语，表示所有可能的分类标签或输出的集合。在分类任务中，模型的目标是将输入数据映射到这些标签之一。例如，在情感分析任务中，标签空间可能是：Positive（正面）、Neutral（中立）和 Negative（负面）。

表现是差不多的（随便从网上摘抄一篇评论来替换这两个例子中的“新推文”）。

推文：“我讨厌没有 WIFI 的时候”

情感：负面

推文：“喜欢那部电影”

情感：正面

推文：“好车!!!”

情感：正面

推文：{新推文}

情感：

第二个提示是（注意情感是错误的）：

推文：“我讨厌没有 WIFI 的时候”

情感：正面

推文：“喜欢那部电影”

情感：负面

推文：“好车!!!”

情感：负面

推文：{新推文}

情感：

如果某个任务没有预先捕获的输入与标签的对应关系，那么基于上下文的学习可能难以进行。我们可以从中体会到，训练对于任何任务的成功和正确执行是多么重要！你在提示中提供的“样本”主要是作为任务的一个定位器使用。至于任务的具体执行方式，AI 在训练期间就已经掌握了。

一些例子

关于少样本学习提示的效率，最著名例子之一来自布朗等人的论文，任务是在一个句子中正确地使用一个新的单词：^①

A "whatpu" is a small, furry animal native to Tanzania. An example of a sentence that uses the word whatpu is: We were traveling in Africa and we saw these very cute whatpus. To do a "farduddle" means to jump up and down really fast. An example of a sentence that uses the word farduddle is:

（“whatpu”是坦桑尼亚的一种小型毛茸茸的动物。使用 whatpu 的一个例子是：我们在非洲旅行时看到了这些非常可爱的 whatpus。进行“farduddle”意味着非常快速地上下跳动。使用 farduddle 的一个例子是：）

模型会正确输出如下内容：

We were so excited that we couldn't help but farduddle when our favorite team won the

^① 译注：该论文于 2020 年在 arXiv 上发表，名为“Language Models are Few-Shot Learners”，网址是 <https://arxiv.org/abs/2005.14165>。

championship.

(我们非常激动，以至于当我们最喜欢的队伍赢得冠军时，我们忍不住 farduddle。)

少样本学习的一个非常好的使用场景是以一种给定的风格来输出内容，如代码文档或社交媒体帖子。只要存在一种模式或格式，并且用文字解释起来远比提供一个例子更复杂，就值得尝试少样本提示。例如，以下提示将生成风格一致的文档：

```
/// <summary>
/// 返回关于特定事件的详细信息
/// </summary>
/// <param name="org">组织的名称</param>
/// <param name="year">事件的年份</param>
/// <param name="tournamentId">事件的内部 ID</param>
/// <returns>关于事件的详细信息</returns>
/// <remarks>可尝试的示例事件为：2892, 2331, 1028。也可以尝试 8392。如果任何参数无效，端点（终结点）返回 404。</remarks>
public IActionResult Tournament(string org, int? year, string eventId)

public IActionResult Team(string org, string teamId)
```

取决于所用的模型版本和 API（Completion 或 Chat Completion），系统可能倾向于对代码和文档进行解释，而不是为上例最后一个“光杆”方法（`IActionResult Team`）补充新文档。甚至还有一些 AI 会揣测你的“意图”，为上述函数生成具体的实现（是的，说的就是你，Gemini）。这就是为什么在更好和更精确的提示中，应该包括一条具体的指示的原因，如下所示：

```
你的任务是以一致的风格生成文档。
/// <summary>
/// 返回关于特定事件的详细信息
/// </summary>
/// <param name="org">组织的名称</param>
/// <param name="year">事件的年份</param>
/// <param name="tournamentId">事件的内部 ID</param>
/// <returns>关于事件的详细信息</returns>
/// <remarks>可尝试的示例事件为：2892, 2331, 1028。也可以尝试 8392。
/// 如果任何参数无效，那么端点（终结点）返回 404。</remarks>
public IActionResult Tournament(string org, int? year, string eventId)

public IActionResult Team(string org, string teamId)
```

交互式完善

上述最后一个提示可能会生成如下所示的响应：

以下是一个完整的、风格一致的文档示例，适用于 .NET Web API 中的两个具体操作，它们分别获取有关特定事件和团队的信息：

```
/// <summary>
```

```

/// 返回关于特定事件的详细信息。
/// </summary>
/// <param name="org">组织的名称，代表组织事件的机构。</param>
/// <param name="year">事件发生的年份，可为空。</param>
/// <param name="tournamentId">事件的内部 ID，用于唯一标识事件。</param>
/// <returns>如果成功，返回事件的详细信息；如果参数无效或事件不存在，返回 404 状态码。</returns>
/// <remarks>示例事件 ID 可尝试包括：2892, 2331, 1028, 8392。</remarks>
public IActionResult Tournament(string org, int? year, string eventId)
{
    // 这里添加方法的实现代码
}

/// <summary>
/// 获取关于特定团队的信息。
/// </summary>
/// <param name="org">组织的名称，代表拥有团队的机构。</param>
/// <param name="teamId">团队的 ID，用于唯一标识团队。</param>
/// <returns>如果成功，返回团队的详细信息；如果参数无效或团队不存在，返回 404 状态码。</returns>
public IActionResult Team(string org, string teamId)
{
    // 这里添加方法的实现代码
}

```

在这个示例中，每个方法都包含了一个详细的文档注释，其中描述了方法的功能、参数、返回信息和备注。这种风格的一致性有助于维护代码的可读性和可维护性，同时为使用 API 的开发者提供清晰、有用的文档。这也展示了如何在实际应用中通过少样本提示来生成一致风格的代码文档。

从这些例子很容易地看出，生成式模型颇有“表现欲”，想把一件事情说得清清楚楚，因为它们本来就被训练来预测下一个 token。因此，为了使结果符合预期，一个好的提示很可能需要对输出做一些限制。

因此，就上一个例子来说，一个更好的提示可能是：

你的任务是基于现有的示例，生成风格一致的代码注释。在你的响应中，只应包含新方法的注释，不要包含任何额外的解释或评论。

示例：

```

/// <summary>
/// 返回关于特定事件的详细信息
/// </summary>
/// <param name="org">组织的名称</param>
/// <param name="year">事件的年份</param>
/// <param name="tournamentId">事件的内部 ID</param>
/// <returns>关于事件的详细信息</returns>
/// <remarks>可尝试的示例事件为：2892, 2331, 1028。也可以尝试 8392。
/// 如果任何参数无效，那么端点（终结点）返回 404。</remarks>
public IActionResult Tournament(string org, int? year, string eventId)

```

新方法：

```

public IActionResult Team(string org, string teamId)

```

以下输出大致符合我们的预期了，但还是生成了一些额外的解释：

根据给定的示例，为新方法 `Team` 生成风格一致的代码注释如下：

```
/// <summary>
/// 获取关于特定团队的详细信息。
/// </summary>
/// <param name="org">组织的名称，代表拥有团队的机构。</param>
/// <param name="teamId">团队的 ID，用于唯一标识团队。</param>
/// <returns>如果成功，返回团队的详细信息。</returns>
public IActionResult Team(string org, string teamId)
```

这样的注释遵循了提供的示例的风格，确保了文档的一致性，同时详细描述了每个参数的作用及方法的返回行为，便于开发者理解和使用该方法。

为了防止模型生成额外的文本，我们可能需要一个不同的策略：要求模型检查输出是否满足某些条件。这可以通过在之前的提示后追加下面这行来实现：

检查输出是否包含额外文本，如果有，则删除之。

要求模型检查输入是否满足特定条件是一个非常实用的技巧。它也可以被用于更标准的任务，如表单或 json/xml/html 校验。

在本例中，我们还尝试了对输出的文本进行校验。这更像是一个技巧而不是一个技术，因为模型实际上并没有真正生成要进行校验的完整输出。尽管如此，它作为一种防护措施还是有效的。为了获得相同的结果，一个更好的方法是增加 API 调用来使用之前已经给出的提示，或者如本书后面所述的，使用像 Microsoft Guidance 或 Guardrails AI 这样的框架。这些框架可能提供更为系统和全面的验证机制，从而提高输出验证的有效性和安全性。

有鉴于此，重要的是强调模型需要做什么，应该达到什么目的（正面指导），而不是像个老妈子一样喋喋不休地告诉它什么事情不能做（消极避免）。

思维链场景

尽管标准的少样本提示对许多任务都很有效，但它并非没有局限性——特别是在处理较复杂的推理任务时，如数学和逻辑问题，以及需要顺序执行多个步骤的任务。



注意：在处理逻辑问题时，GPT-4/GPT-4o 等较新模型的表现更好，即使提供的是简单的、未经优化的提示。

若少样本提示不足以应对，就可能需要对模型进行一些微调（前提是实际情况允许，至少

GPT-4 和 GPT-4-turbo 不允许^①) 或探索更高级的提示技术。其中一种技术是思维链 (Chain-of-Thought, CoT) 提示。我们使用 CoT 提示来追踪模型执行的所有步骤 (思考), 以得出最终的解决方案。

Wei 等人的研究表明^②, 这种技术通过纳入中间推理步骤, 给模型时间思考, 从而增强了其推理能力。与少样本提示结合使用时, 一些需要事先推理才能获得更准确响应的复杂任务有了更好的表现。



注意: 通常, 当模型的规模达到千亿参数的规模时, 才能观测到 CoT 提示的有效性。较小的模型倾向于生成不连贯的思维链, 与标准提示相比, CoT 提示的准确性较低。通过 CoT 提示实现的性能改进通常与模型的规模成正比。

基本理论

回想学生时代的我们, 在考试的压力下, 大脑常会陷入短暂的停滞。很多时候, 我们几乎是凭着直觉去揣测答案, 努力回应教授的问题, 却未及深思熟虑。LLM 也是如此, 有时它们在不加思考的情况下继续句子; 一切都只是靠“猜”。

CoT 背后的基本思路是指示模型花一些时间思考, 允许它在生成响应时重建推理步骤。这个简单的思路显著增强了模型执行复杂任务的能力。只需想一想语言模型是如何生成响应的, 就会明白其中的道理。它们试图预测下一个 token, 因此模型生成的中间“推理” token 越多, 就越需要在它们之间建立连贯性, 并在理想情况下提供正确的响应。



注意: 如果只是要求模型给出一个最终答案, 那么为了验证问题 (提示) 和响应之间的连贯性, 模型所能做的工作非常有限。相比之下, 明确概述所有步骤有助于模

^① 译注: 本书写于 2023-2024 年之间。截至 2023 年底, GPT-4 和 GPT-4-turbo 的微调 (fine-tuning) 功能确实尚未开放。OpenAI 在那时主要提供了 GPT-3 系列 (如 Davinci、Curie、Babbage 和 Ada) 的微调功能, 但 GPT-4 系列还没有这种能力。不过, 等你读到本书时, 情况可能已经发生了变化。译者主页 (<https://bookzhou.com>) 会同步更新最新的情报。

^② 译注: Wei 等人的论文于 2022 年在 arXiv 上发表, 名为 “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”, 网址是 <https://arxiv.org/abs/2201.11903>。

型找到逻辑线索。

一些例子

继续教授-学生的例子，改进模型输出的前两个尝试可能是经典的“确保答案正确”或“让我们一步一步地思考”的方法。例如，考虑以下简单的问题：

我买了 20 支笔，给了我儿子 12 支，给了我女儿 3 支，给了我妈妈 1 支。然后我又买了 4 支笔，丢了一支。
输出我现在有多少支笔。

关于逻辑问题，温度和模型的版本确实很重要。较低的温度会使事情更具确定性，而最新的模型由于在更大的逻辑问题数据集上训练过，因此具有更高的准确性。

使用默认温度，Google Gemini 模型提供了如下所示的答案，而 GPT-4 模型可能会说：“你现在有 7 支笔”。

您现在有 8 支笔。

以下是计算过程：

1. 开始时有 20 支笔。
2. 给儿子 12 支，还剩 $20 - 12 = 8$ 支。
3. 给女儿 3 支，还剩 $8 - 3 = 5$ 支。
4. 给妈妈 1 支，还剩 $5 - 1 = 4$ 支。
5. 又买了 4 支，共有 $4 + 4 = 8$ 支。
6. 丢了 1 支，还剩 $8 - 1 = 7$ 支。
7. 最后您有 7 支笔。



注意：较旧的模型可能只会回应一个数字，而没有任何推理过程。较新的模型通过逐步提示得到了强化。

答案是错误的。更糟的是，答案不连贯。它说的是 8，但实际已经正确地计算出了 7。

让我们尝试以下提示，看看它是否提供更合理的输出：

我买了 20 支笔，给了我儿子 12 支，给了我女儿 3 支，给了我妈妈 1 支。然后我又买了 4 支笔，丢了一支。
输出我现在有多少支笔，并确保答案正确且连贯。在回答前，确保答案和解释是正确且连贯的。

现在的输出如下所示：

我们按照题目给出的信息逐步计算：

1. 开始有 20 支笔。
2. 给出 $12 + 3 + 1 = 16$ 支笔。
3. 剩下 $20 - 16 = 4$ 支笔。
4. 买入 4 支笔，共有 $4 + 4 = 8$ 支笔。

5. 丢失 1 支笔, 剩下 $8 - 1 = 7$ 支笔。
因此, 您现在有 7 支笔。

答案正确。我们终于将 Gemini 调试好了。在我们测试的所有模型中 (包括 GPT 系列、Gemini、文心一言和 KIMI), 它是唯一一开始就出现逻辑错误的。

下面是一个等效的提示:

我买了 20 支笔, 给了我儿子 12 支, 给了我女儿 3 支, 给了我妈妈 1 支。然后我又买了 4 支笔, 丢了一支。输出我现在有多少支笔。让我们一步一步地思考。

GPT-4 的输出结果如下所示:

好的, 我们一步一步来计算。

1. 初始笔数:
 - 我买了 20 支笔。
2. 给别人笔的情况:
 - 给了我儿子 12 支。
 - 给了我女儿 3 支。
 - 给了我妈妈 1 支。这三个人一共拿走了 $12 + 3 + 1 = 16$ 支笔。
3. 剩余笔数:
 - 初始笔数减去给出去的笔数:
 $20 - 16 = 4$ 支笔。
4. 又买了 4 支笔:
 - 我再买了 4 支笔, 所以我有:
 $4 + 4 = 8$ 支笔。
5. 丢了一支笔:
 - 然后我丢了一支笔, 所以现在剩下:
 $8 - 1 = 7$ 支笔。

所以, 经过这些步骤之后, 我现在有 7 支笔。

这被称为零样本思维链提示。只需说出“让我们一步一步地思考”这个魔法般的句子, 就会迫使模型花时间思考并列举所有计算步骤。

此时, 为了得到最终答案, 你可以要求模型生成结构化输出, 或者再进行一次 API 调用来传递一个简单的提示, 例如: “只从这个文本中提取最终答案”。

我买了 20 支笔, 给了我儿子 12 支, 给了我女儿 3 支, 给了我妈妈 1 支。然后我又买了 4 支笔, 丢了一支。输出我现在有多少支笔。让我们一步一步地思考。输出 json 格式: 解释(string)和结果(int)。

随后会生成像下面这样的 JSON 代码:

```
{ "解释": "我买了 20 支笔, 给了我儿子 12 支, 给了我女儿 3 支, 给了我妈妈 1 支。所以我还剩 4 支笔。然后我又买了 4 支笔, 现在有 8 支笔。丢了一支后, 现在剩 7 支笔。", "结果": 7 }
```

可能的扩展

将少样本技术与思维链方法相结合，可以提供一些逐步推理的示例供模型模仿。这被称为**少样本思维链**（few-shot chain-of-thought）。例如：

平衡成本和时间，去一个目的地最适合使用哪个方案？示例：

方案 1：步行 20 分钟，然后乘坐 15 分钟的公交车（2 元），最后乘坐 5 分钟的出租车（15 元）。

方案 2：骑自行车 30 分钟，然后乘坐 10 分钟的地铁（2 元），最后步行 5 分钟。

方案 1 花费 $20 + 15 + 5 = 40$ 分钟。选项 1 将花费 17 美。

方案 2 花费 $30 + 10 + 5 = 45$ 分钟。选项 2 将花费 2 元。

由于方案 1 需要 40 分钟，而方案 2 需要 45 分钟，所以虽然选项 1 更快，但选项 2 便宜得多。因此，选项 2 更好。

请问，去办公室最适合使用哪个方案？

方案 1：火车 40 分钟（5 元），步行 15 分钟

方案 2：出租车 10 分钟（15 元），地铁 10 分钟（2 元），步行 2 分钟

对这种基本提示技术的一个扩展称为**自动思维链**（Auto-CoT）。它以上述少样本思维链方法为基础，但使用一个提示来生成更多的推理示例（样本），然后将它们连接成一个最终提示。这个方法的核心思路是自动生成一个少样本 CoT 提示。

除了思维链提示外，还有一种更复杂的方法：思维树。该技术可以通过两种方式实现。第一种是通过单个提示，如下所示：

考虑由三个专家解决一个问题的过程。

每个专家将贡献他们思维过程的一步，并与小组分享。

随后，所有专家将继续下一步。

如果任何专家在任何阶段意识到自己犯了错误，他们将退出这个过程。

问题如下：{问题}

一种更复杂的思维树方法需要编写更多代码，使用不同的提示（可能也有不同的温度设置）来生成推理路径。然后，这些路径由另一个模型实例通过评分/投票提示进行评估，排除错误路径。最后，通过某种机制（如连贯性或多数票）选出正确答案。

还有一些新兴但相对容易实现的提示技术，如类比提示（由 Google DeepMind 提出），它要求模型在解决当前问题之前回忆类似的问题。又如回退提示，它提示模型从具体实例中回退，思考当前的一般原则。

基本使用场景

在探索了一些较高级的技术后，现在是时候将重点转向实际应用了。在本节中，你将深入了解这些技术在实际场景中的应用，体验如何把它们应用于现实世界。这些使用场景的一部分将在后面的章节中扩展，其中包括聊天机器人、摘要/扩展、编码助手以及通用翻译器等。

聊天机器人

聊天机器人（chatbot）其实已经存在多年了，但在最新的 LLM 大语言模型出现之前，用户在与它们交互时，大多数时候感觉到的都是“痛苦”。然而，新的模型出现之后，即使用户犯错误或者问得不好，他们的意图也能被正确地理解，并且能连贯和正确地回应。以前，使用聊天机器人的人们几乎总是认为对方很“傻”，只会机械地输出一些官方文章。最终，基本上都会想办法呼叫真人。但现在一切都变了，我预计人们很快就会采取相反的做法：“让我和机器人聊天；这个人听不懂我的话。”

系统消息

在聊天机器人中，系统提示（也称为**元提示**，即 metaprompt）可以用于引导模型的行为。元提示定义了要遵循的一般指导原则。不过，即使使用了这些模板和指导原则，也有必要对模型生成的响应进行验证。

在一个好的系统提示中，应该定义模型的配置（profile）、能力和特定场景下的限制，如下所示：

- 告诉模型应该如何完成任务，以及是否可以使用额外的工具。
- 明确说明模型性能的范围和限制，包括如何处理跑题或无关的提示。
- 确定模型在响应时的态度和语气。
- 定义输出格式，包括语言、语法和任何格式偏好。
- 提供一些示例（样本），以展示模型预期的行为，考虑加入一些困难的使用场景和思维链（CoT）推理。
- 事先识别一些潜在的危害，定义优先级并予以解决，建立起额外的行为保护机制。

收集信息

假设要为某连锁酒店品牌构建预订聊天机器人。一个合理的系统提示可能是：

你是 HotelBot，一个自动服务机器人，负责收集不同城市的酒店预订信息。
首先要向客人问好，然后收集预订信息，询问客户的姓名、想要预订的城市、房型和额外服务。
在收集了完整的预订信息后，你需要汇总预订信息，并最后一次确认客人是否还想添加额外的服务。
你需要询问到达日期和离开日期，并自动计算入住天数（几晚）。你需要询问身份证/护照号码。请确保已经明确了所有选项和额外的服务，从价格表中唯一性地识别收费项。
你的回应应该简短且非常友好。可预订的城市有：北京、成都、重庆和上海。

房型和价格：
单人房每晚 150 元
双人房每晚 250 元
套房每晚 350 元

额外服务：
停车每天 20 元

延迟退房 100 元
机场接送 50 元
SPA 一次 30 元

注意，以上提示只是一个更大规模的应用的一部分。在收到系统消息后，应用程序应要求用户开始与自己交互；然后，用户与聊天机器人之间的对话正式开始。

下面以一个控制台应用程序为例来展示如何启动与用户的交互。

```
var systemPrompt = "替换为刚才的大段系统提示";

var chatCompletionsOptions = new ChatCompletionsOptions
{
    DeploymentName = AOAI_DEPLOYMENTID, // 这个常量已经在之前的例子中定义了
    Messages =
    {
        new ChatRequestSystemMessage(systemPrompt),
        new ChatRequestUserMessage("介绍你自己"),
    }
};

while (true)
{
    Console.WriteLine();
    Console.Write("HotelBot: ");
    var chatCompletionsResponse = await
        openAIClient.GetChatCompletionsAsync(chatCompletionsOptions);
    var chatMessage = chatCompletionsResponse.Value.Choices[0].Message;
    Console.Write(chatMessage.Content);
    chatCompletionsOptions.Messages.Add(new
        ChatRequestAssistantMessage(chatMessage.Content));
    Console.WriteLine();
    Console.Write("请输入: ");
    var userMessage = Console.ReadLine();
    chatCompletionsOptions.Messages.Add(new ChatRequestUserMessage(userMessage));
}
```



注意：开发 Web 应用时，还必须考虑到聊天 UI 的设计。

短短几行代码，就可以生成一个非常贴心的酒店房间预订机器人，下面是一段示例对话。注意，“请输入：”后面都是由用户输入的内容。

HotelBot: 您好！我是 **HotelBot**，一个自动服务机器人，可以帮助您预订酒店并提供所需的信息和服务。请告诉我您的姓名，并让我知道您想预订的城市、房型和是否需要额外服务。我会尽力满足您的需求！
请输入：**我要预订成都的酒店**

HotelBot: 非常好!感谢您选择成都作为您的目的地。请告诉我您的姓名,房型偏好和是否需要额外服务,以便我可以帮您找到合适的酒店。

请输入: **我叫周靖,想要预订单人间,需要停车服务**

HotelBot: 感谢提供信息,周靖先生。我会为您寻找成都的适合房型和停车服务的酒店。请告诉我您计划的到达日期和离开日期,以便为您计算入住天数。

请输入: **到达日期是 2024 年 9 月 1 日,住四晚**

HotelBot: 非常感谢详细信息!您的预定已经记录。您将预订一间单人间,在成都酒店入住四晚,从 2024 年 9 月 1 日入住至 9 月 5 日离店。您还需要提供身份证/护照号码以完成预订。

关于停车服务,每天的停车费用为 20 元,我们将为您预订并记入您的住宿费用中。请提供您的身份证/护照号码,方便我为您办理预订手续。

请输入: **我的身份证是 1234567890**

HotelBot: 感谢提供身份证号码。您的预订信息已完整记录。

以下是您的预订信息:

- 预订人姓名: 周靖先生
- 预订城市: 成都
- 房型: 单人间
- 预计到达日期: 2024 年 9 月 1 日
- 预计离开日期: 2024 年 9 月 5 日
- 入住天数: 4 晚
- 停车服务: 是 (每天 20 元)

请注意,酒店费用将根据您的入住天数和房型进行计算。您的总费用将在最后一次确认时提供。

请确认这些信息是否正确或者是否需要添加额外的服务。如果没有,请输入“确认”: **确认**

HotelBot: 非常感谢确认!您的预订已经完成,总费是 680 元。我们会尽快为您安排酒店。如果您有任何其他问题或需要进一步的帮助,请随时告诉我。祝您旅途愉快!

摘要与转换

现在,我们已经写了一个提示要求机器人收集酒店预订信息,酒店预订系统可能需要保存收集到的信息(调用 API 或直接将信息保存到数据库)。但是,这些信息都是来自客户和机器人之间对话的非结构化自然语言。因此,还需要一个提示对信息进行摘要(汇总),并转换为结构化的数据。

返回前一次预订的 JSON 摘要。逐项列出每项的价格。

JSON 字段应为:

- 1) 姓名,
- 2) 身份证/护照,
- 3) 城市,
- 4) 房型及总价,
- 5) 额外服务列表及总价,

-
- 6) 到达日期,
 - 7) 离开日期,
 - 8) 总天数/多少晚,
 - 9) 房间和额外服务的总价格 (计算为房间总价和额外服务总价之和)。
- 只返回 JSON, 不需要补充说明。

基于上一节与 HotelBot 的对话, 该提示将生成如下所示的 JSON (是的, 键的名称已 t6hfc 翻译成了英文):

```
{
  "name": "周靖",
  "id/passport": "1234567890",
  "city": "成都",
  "room_type": "单人间",
  "room_type_total_price": 600,
  "extras": "停车",
  "extras_total_price": 80,
  "arrival_date": "2024-09-01",
  "departure_date": "2024-09-05",
  "total_days": 4,
  "total_price": 680
}
```

扩展

有的时候, 可能需要处理相反的问题, 即从结构化的 JSON 生成自然语言摘要。以下示例提示用于处理这种情况:

从以下 JSON 返回一个友好的文本摘要。最多写两句话。

```
{ "name": "周靖",
  "id/passport": "1234567890",
  "city": "成都",
  "room_type": "单人间",
  "room_type_total_price": 600,
  "extras": "停车",
  "extras_total_price": 80,
  "arrival_date": "2024-09-01",
  "departure_date": "2024-09-05",
  "total_days": 4,
  "total_price": 680 }
```

一个可能的输出是:

周靖预订了成都的一间单人间, 入住日期为 2024 年 9 月 1 日, 离店日期为 2024 年 9 月 5 日, 共计 4 天, 房间总价 600 元, 额外服务 (如停车) 费用 80 元, 总费用为 680 元。

翻译

得益于预训练，LLM 在不同语言之间的翻译方面表现出色——不仅是自然语言，还包括编程语言。

从自然语言到 SQL

一个直接取自 OpenAI 官方文档的著名示例如下：

```
### Postgres SQL 表及其属性：
#
# Employee(id, name, department_id)
# Department(id, name, address)
# Salary_Payments(id, employee_id, amount, date)
#
### 查询过去 3 个月中员工数量超过 10 名的部门的名称
SELECT
```

上述提示是一个典型的简单补全（完成）示例（即 Completion API）。最后一部分（SELECT）是一个线索，充当了输出的起点。

从更广泛的意义上说，在 Chat Completion API 的上下文中，可以通过系统提示来提供数据库模式（database schema）并询问用户想要提取哪些信息。然后，将其翻译成一个实际的 SQL 查询。对于这种类型的提示所生成的查询来说，用户应该先评估风险，然后才能在数据库上执行。还有其他一些工具能通过使用了 LangChain 框架的代理直接与数据库交互，这将在本书后面讨论。当然，这些工具是存在一定风险的。它们提供了对数据层的直接访问，所以每次访问都要先评估风险。

通用翻译器

下面考虑一个消息应用程序，其中每个用户都选择了自己的首选语言。他们用该语言书写，如有必要，中间件会将他们的消息翻译成另一位用户的语言。最终，每个用户都使用自己的语言进行读写。

翻译中间件可以是一个获取了以下提示的模型实例：

```
将以下文本从{user1Language}翻译成{user2Language}：
<<<{message1}>>>
```

交互的完整模式如下所示：

1. 用户 1 选择其首选语言 {user1Language}。
2. 用户 2 选择其首选语言 {user2Language}。

-
3. 其中一人向另一人发送消息。假设用户 1 用 {user1Language} 写了一条消息 {message1}。
 4. 中间件将 {message1} 从 {user1Language} 翻译成 {user2Language} 的 {message1-translated}。
 5. 用户 2 用自己的语言看到 {message1-translated}。
 6. 用户 2 用 {user2Language} 写一条消息 {message2}。
 7. 中间件执行相同的工作并将消息发送给用户 1。
 8. 如此往复……

LLM 的局限性

到目前为止，本章一直强调的都是 LLM 积极的方面。但 LLM 存在多方面的局限性。

- 由于缺乏互联网访问和有限的记忆（本地部署的开源 LLM 尤其如此），LLM 难以提供准确的来源引用。因此，它们可能会生成看似可靠但实际上不正确的来源（这称为幻觉）。支持网上搜索扩展的 LLM 有助于解决这个问题。
- LLM 往往会产生有偏见的回应。有的时候，即便有保护措施，也仍然会出现性别歧视、种族主义或其他“政治不正确”的回答。在消费者应用和研究中使用 LLM 时应谨慎，尽可能避免偏见。
- 面对没有接受过训练的问题时，LLM 往往会生成错误信息，而且还十分“自信”地提供错误的答案或幻觉般的回应。
- 如果没有额外的提示策略，LLM 通常在数学方面表现不佳，无论是简单还是复杂的数学问题都容易出错。^①

请务必意识到这些限制。除此之外，还应该警惕提示攻击（prompt hacking）问题，即用户利用 LLM 的漏洞来生成正常情况下不应该生成的内容。^②所有这些安全问题将在本书后面讨论。

^① 译注：这一点从各大模型（包括 GPT-4o 和 Qwen2-72B 等）做 2024 年高考试卷的结果就可以看出。从单科成绩上看，大模型“考生”数学不太行，普遍出现了偏科、不及格的状况，得分率均没有超过 50%。

^② 译注：其中最著名的一个案例就是“奶奶漏洞”。在 ChatGPT 的早期岁月，直接要求它生成 Windows 10 Pro 的序列号是不可能的。但是，要求它扮演自己已经过世的奶奶，并说奶奶总是会念 Windows 10 Pro 的序号让自己睡觉，ChatGPT 的这个限制就被破解了。不过，根据当前（2024 年 9 月）的情况，Google Gemini 还是存在这个漏洞。Gemini 在我们测试的所有模型中表现最差，不是没有原因的。

小结

本章在 LLM 的背景下讨论了提示工程的各种基本方面。我们涵盖了对输出进行修改的常规实践和一些替代方法，其中包括调参。此外，还讨论了如何在云端创建 OpenAI API 服务，并通过 C#和 Python 来调用。

然后，本章深入探讨了基本的提示技术，包括零样本提示和少样本提示、迭代优化（交互式完善）、思维链（给模型思考的时间）和一些可能的扩展。本章还探索了一些基本的使用场景，例如用于收集信息的酒店预订聊天机器人、摘要和转换以及通用翻译器等。

最后，本章讨论了 LLM 的局限性，包括生成不正确的引用、有偏见的回应、返回虚假信息以及在数学方面表现不佳等。

后续各章将讨论如何通过一些更高级的提示技术来利用 LLM 的更多功能，还介绍了一些第三方工具。

第 3 章 打造高级学习提示

通过操纵大语言模型（LLM）的提示和超参数，我们可以实现很多更高级的功能。例如，可以修改所生成内容的语气、风格、准确率和正确性水平。另外，通过诸如思维链、思维树等更智能的技术，甚至可以在一定程度上赋予 AI 自我纠正和改进的能力。

在本书之前的例子中，我们主要是向目标 LLM 发送单一的提示，然后把生成的响应（无论是否经过筛选）展示给用户。这种方法有效，但会给模型的任意性和随机性留下太多的空间，而这些是我们无法完全控制的。

本章探讨了将 LLM 整合到更广泛的软件生态系统中时，如何通过一些更高级的技术来减少其随机性。其中一些最重要和最广泛使用的应用场景包括：

- 与现有 API 集成以获取信息
- 连接数据库以检索必要的数据库
- 审核内容和主题（将在下一章讨论）。

在实际应用中，本章探讨的大多数技术通常都不需要从头实现，而是可以通过诸如 LangChain、Semantic Kernel 或 Guidance 等框架来使用，这些框架将在下一章中介绍。本章的重点是介绍这些技术的机制、它们可以带来的好处以及在什么时候可以享受到这些好处。

超越提示工程

在快速发展的 AI 技术领域，纯粹提示工程的重要性可能由于各种原因而逐渐下降。事实上，以前之所以需要特定和精心设计的提示，是因为早期的 AI 系统无法完全理解和掌握自然语言。但是，随着这些模型在这个方面做得越来越好，AI 将更容易地理解你的提示。换言之，AI 会越来越“懂你”。

为了利用生成式 AI 的潜力，一个持久且关键的技能是学会如何恰当地表述问题并设计解决方案。为此，你需要学会识别、分析和定义核心问题；将其分解为可管理的子问题；从不同角度重新建构问题；以及设计适当的约束条件。

另外，还必须能将生成式 AI 和 LLM 放到一个正确的框架内，将各个部分组合成一个完整的系统并嵌入自己的应用。

本质上，除了提示工程技术外，还可以将 LLM 调用与更标准的软件工具（例如，API）结合或者执行微调训练，从而利用基础设施或模型本身。在某些情况下，为了获得更好的结果，可能唯一的方案就是同时对基础设施和模型进行操作。

合并不同的部分

从基础设施的角度来看，为了超越纯粹的提示工程，我们要做的下一步是将对 LLM 的单一补全调用（completion call）插入某个链（chain）或流（flow）中。这样，LLM 所生成的输出就不一定是最终用户将看到或使用的内容。相反，它只是某个较长过程中的一个步骤。换言之，此时不再将 LLM 作为一种独立和直接的工具使用。相反，我们把它作为一个工具嵌入自己的工作流——虽然这个工具通常还是要直接与最终用户接触。

例如，上一章提到的由 GPT-3.5 生成的 JSON 仍有可能是无效的。因此，在使用或保存到数据库之前，必须先用一些 JSON 验证库来完成对它们的解析。类似地，在酒店预订聊天机器人中，必须在最终确定请求之前考虑房间的可用性或设施是否已经关闭。因此，在开发过程中，至关重要的一个环节就是将聊天机器人连接到提供此类附加信息的 API。

再举一个例子。LLM 是基于公共数据来训练的，当它们响应时，会依赖于这些数据。然而，如果我们希望它们能够回答关于公司内部数据或者新的内部文档的问题，或者基于这些非公开数据生成见解，那么可能需要设置一个搜索机制来查找相关文档，以某种方式将它们提供给模型，并让其针对性地构建响应。

什么是链？

LLM 链是一个始于用户输入的顺序操作列表。输入可以是问题、命令或某种触发器（trigger）。然后，将输入与某个提示模板（prompt template）集成，以完成它的格式化。在应用提示模板后，链可能会执行额外的格式化和预处理，以优化数据以适应 LLM。常见的操作包括数据增强（data augmentation）、重新措辞（或者说“改写”，即 rewording）和翻译（translating）。



注意：链中可能包含 LLM 本身之外的其他组件。一些步骤还可能要使用更简单的 ML 模型或标准软件执行。

LLM 处理格式化和预处理的提示并生成响应。该响应成为链中当前步骤的输出，并且可以根据应用程序的需求以各种方式使用。它可以显示给用户、进一步处理或者输入到链中的下一个组件。

深入探究

当我们跟踪链内部发生的事情时，无论这是由我们自己编写的链，还是由诸如 LangChain

或 Semantic Kernel 等工具生成的链，我们面临的都是一个类似于人类思维过程的过程：链的每一步都应该解决问题的一部分，为后续步骤生成反思、答案和有用的见解。从某个方面来说，除了最初的提示工程，还存在一系列的中间提示。

如果下一步不是预先确定的，而是由推理 LLM 来识别，那么这样的链通常称为**代理或智能体**（agent）。

步骤越多，对每一步进行日志记录就越重要，因为最终结果将直接取决于链中传递的信息。

以下是 LangChain 文档中一个包含几个工具的代理日志示例。

```
[1:chain:agent_executor] 进入链运行，输入：{
  "input": "Olivia Wilde 的男友是谁？他现在的年龄的 0.23 次方是多少？"
}
[1:chain:agent_executor > 2:chain:llm_chain] 进入链运行，输入：{
  "input": "Olivia Wilde 的男友是谁？他现在的年龄的 0.23 次方是多少？"
}
[1:chain:agent_executor > 2:chain:llm_chain > 3:llm:openai] 退出 LLM 运行，输出：{
  "generations": "我需要找出 Olivia Wilde 的男友是谁，然后计算他的年龄的 0.23 次方。\\n 操作：搜索\\n 操作输入：\\"Olivia Wilde 男友\\"""
}
[1:chain:agent_executor > 2:chain:llm_chain] 退出链运行，输出：{
  "text": "我需要找出 Olivia Wilde 的男友是谁，然后计算他的年龄的 0.23 次方。\\n 操作：搜索\\n 操作输入：\\"Olivia Wilde 男友\\"""
}
[1:chain:agent_executor] 代理选择的操作：{
  "tool": "search",
  "toolInput": "Olivia Wilde 男友",
  "log": "我需要找出 Olivia Wilde 的男友是谁，然后计算他的年龄的 0.23 次方。\\n 操作：搜索\\n 操作输入：\\"Olivia Wilde 男友\\"""
}
[1:chain:agent_executor > 4:tool:search] 退出工具运行，输出："2021 年 1 月，Wilde 在拍摄《亲爱的别担心》期间与歌手 Harry Styles 开始约会。他们的关系在 2022 年 11 月结束。"
[1:chain:agent_executor > 5:chain:llm_chain] 退出链运行，输出：{
  "text": "我需要找出 Harry Styles 的年龄。\\n 操作：搜索\\n 操作输入：\\"Harry Styles 年龄\\"""
}
[1:chain:agent_executor] 代理选择的操作：{
  "tool": "search",
  "toolInput": "Harry Styles 年龄",
  "log": "我需要找出 Harry Styles 的年龄。\\n 操作：搜索\\n 操作输入：\\"Harry Styles 年龄\\"""
}
[1:chain:agent_executor > 7:tool:search] 退出工具运行，输出："30 岁"
[1:chain:agent_executor > 8:chain:llm_chain] 退出链运行，输出：{
  "text": "我需要计算 30 的 0.23 次方。\\n 操作：计算器\\n 操作输入：30^0.23"
}
[1:chain:agent_executor] 代理选择的操作：{
  "tool": "calculator",
  "toolInput": "30^0.23",
  "log": "我需要计算 29 的 0.23 次方。\\n 操作：计算器\\n 操作输入：30^0.23"
```

```
}
[1:chain:agent_executor > 10:tool:calculator] 退出工具运行, 输出: "2.186441634154417"
[1:chain:agent_executor > 11:chain:llm_chain] 退出链运行, 输出: {
"text": "我现在知道最终答案了。\\n 最终答案: Harry Styles 是 Olivia Wilde 的男友, 他现在年龄的
0.23 次方是 2.186441634154417。"
}
[1:chain:agent_executor] 退出链运行, 输出: {
"output": "Harry Styles 是 Olivia Wilde 的男友, 他的现在年龄的 0.23 次方是 2.186441634154417。"
}
}
```

步骤之间的过渡受对话式编程和常规软件的控制。在这个过程中, 有一个提示部分用于生成兼容下一步输出的内容, 还有一个验证部分用于确认输出的有效性, 并将信息有效地传递到下一步。

微调

模型微调^①技术旨在调整预训练语言模型, 使之更好地适应特定任务或领域 (这里的微调具体指的是监督微调, 即 **supervised fine-tuning**)。通过微调, 不仅可以利用 LLM 的知识和能力, 还能根据当前的具体需求进行定制, 使其在处理特定领域的数据和任务时更加准确和有效。尽管少样本提示 (**few-shot prompting**) 可以实现快速适应, 但通过对模型进行微调, 可以实现长期记忆。由于提示的长度固定, 所以有时可能容不下太多的样本。相反, 微调允许基于更多的样本进行训练。也就是说, 一旦模型经过微调, 就不需要在提示中提供样本, 这节省了 **token** (因此也降低了成本), 并且可以生成更低延迟的调用。

在像医学和法律文本分析这样的专业领域中, 微调是必不可少的, 因为有限的训练数据需要将语言模型与特定领域的语言和术语适配。在低资源语言中, 微调能显著提高模型的性能, 使其对上下文中更加敏感且更加准确。此外, 如果代码和文本生成任务涉及特定的风格或行业, 那么使用相关的数据集进行微调, 可以使模型生成更精确、更具有相关性的内容。通过微调, 我们可以开发出模仿特定人物说话风格的定制聊天机器人。这只需要使用他们的文字作品和日常对话数据进行训练就可以了。还可以利用微调来执行实体识别或提取非常具体的结构化信息 (例如, 产品特性或类别), 尽管将输入数据转换为自然语言通常能获得更好的表现。

尽管微调的好处有不少, 但必须把它放到整个解决方案的上下文中仔细考虑。它常见的缺点包括事实准确性和可跟踪性 (可追溯性) 问题, 因为答案的来源现在变得不明确了。访问控制也变得极具挑战性, 因为现在没有办法限制只有特定的用户或群体才能访问特定的文档。此外, 反复重新训练模型的成本可能是令人不可接受的。由于这些限制, 使用微调来执行基本的问答任务变得十分困难, 甚至几乎不可能。

^① 译注: 这里只是按约定俗成的方式将 **fine-tuning** 翻译为“微调”, 但个人更喜欢“调优”。



注意：OpenAI 的 Ada、Babbage、Curie 和 DaVinci 模型及其微调版本自 2024 年 1 月 4 日起已被弃用和禁用。GPT-3.5 Turbo 目前可以进行微调，OpenAI 正致力于为升级后的 GPT-4 启用微调。

准备工作

要微调其中已启用的一个模型，需要使用 JSON Lines（换行符分隔的 JSON，简称 JSONL）格式准备训练和验证数据集，如下所示：

```
{"prompt": "<提示文本>", "completion": "<理想的生成文本>"}
{"prompt": "<提示文本>", "completion": "<理想的生成文本>"}
{"prompt": "<提示文本>", "completion": "<理想的生成文本>"}
```

该数据集由训练样本构成，每个样本都包含一个输入提示和一个期望的输出。相较于在推理过程中使用模型，这种数据集格式存在以下几个方面的区别：

- 为了定制，仅提供一个提示而不是几个样本。
- 提示中不需要详细的指令。
- 确保每个提示以固定分隔符（例如，`\n\n###\n\n`）结束，以指示从提示到补全（完成）的过渡。
- 为了适应分词（tokenization），每个补全都以一个空格开始（这适合英语）。
- 每个补全都以固定的停止序列（例如，`\n, ###`）结束，以表示完成。
- 在推理过程中，以创建训练数据集时所用的同一种方式对提示进行格式化，这包括使用相同的分隔符和停止序列，以便正确截断。
- 数据集的总文件大小不得超过 100MB。

下面是一个英语语言的示例，注意每个“补全”（完成）前面故意添加的空格。

```
{"prompt": "Just got accepted into my dream university! ->", "completion": " positive"}
{"prompt": "@contoso Missed my train, spilled coffee on my favorite shirt, and got stuck in traffic for hours. ->", "completion": " negative"}
```

改为中文后可以这样写：

```
{"prompt": "刚刚被我梦想的大学录取了! ->", "completion": "正面"}
{"prompt": "@contoso 错过了火车，咖啡洒在了我最喜欢的衬衫上，还因为堵车被困了几个小时。 ->", "completion": "负面"}
```

OpenAI 的命令行界面（CLI）配备了数据准备工具，可以验证和重新格式化训练数据，使其适合作为 JSONL 文件进行微调。

训练和使用

训练 OpenAI 微调模型有两个选择。一个选择是使用 OpenAI 的包（CLI 或 CURL）；另一个选择是在 Microsoft Azure 门户上使用 Azure OpenAI Studio。一旦数据准备好，就可以使用 Azure OpenAI Studio 的“创建自定义模型”向导来创建一个自定义模型并启动训练过程。^①首先选择一个合适的基础模型，再输入训练数据（以本地文件或 blob 存储的形式）以及可选的验证数据（如果有的话），然后为微调设置参数。以下是在微调过程中需要考虑的一些重要的超参数：

- **时期数 (number of epochs)**：该参数决定了模型在数据集上训练的周期次数。
- **批大小 (batch size)**：该参数是指在一次前向和后向传递（forward and backward pass，Azure 文档把它们翻译为正向和向后传递）中使用的训练示例的数量。
- **学习速率乘数 (learning rate multiplier)**：这个乘数应用于预训练期间的原始学习率，以控制微调的学习率。
- **提示损失权重 (prompt loss weight)**：这个权重决定了模型是更多地从指令（提示）中学习，还是从生成的文本（补全）中学习。默认情况下，提示损失权重设置为 0.01。当补全的内容较短时，较大的值有助于稳定训练，为模型提供额外支持，使其能够有效理解指令并生成更精确和有价值的输出。



注意：从技术上说，提示损失权重参数是必要的，因为 OpenAI 的模型在微调期间会同时尝试预测所提供的提示和与之相矛盾的信息，以防止过拟合。过拟合指的是模型在训练数据上表现很好，但在新数据上表现不佳，因其过度拟合了训练数据中的噪音或细节。这个参数本质上是指在微调阶段总体损失函数中分配给提示损失的权重^②。

一旦微调完成，就可以部署自定义模型供使用。它将作为标准模型部署，并分配了一个部

^① 译注：具体操作是访问 <https://oai.azure.com/>，单击左侧的“模型”，再单击“创建自定义模型”来启动这个向导。注意，取决于你在哪个区域创建资源，微软可能会限制模型的微调能力。在创建 Azure OpenAI 资源的时候（在 <https://portal.azure.com/> 上搜索“OpenAI”），建议将“实例”的区域设为 East US 2。最新的可微调区域请访问 <https://learn.microsoft.com/zh-cn/azure/ai-services/openai/concepts/models>。

^② 译注：在微调过程中，总体损失函数包括以下几种损失：主任务损失（分类、回归或生成任务上的损失，例如交叉熵损失和均方误差）、提示损失（当前所讲的损失）和正则化损失（正则化项通过增加对模型复杂度的惩罚来约束模型以防止过拟合）。

署 ID（在 Azure OpenAI 中称为“部署名称”），可以使用该 ID 来完成你的 API 调用。



注意：对于非 OpenAI 模型，例如 Llama 3，还存在其他各种替代方案，包括手动微调或者通过 Hugging Face 平台进行微调，后者的模型也可以通过 Azure 来部署（需要安装 Azure CLI）。

函数调用

如果希望使用 LLM 来执行一些特殊的操作，或者需要读取初始训练集之外的新信息，就需要将 LLM 集成一个更复杂的架构中。以助手聊天机器人为例。在某个时候，用户可能想要发送电子邮件或者查询最新的交通状况。类似地，聊天机器人可能会被问到类似于“这个月我的业绩前五名客户是谁？”这样的问题。在所有这些情况下，都需要 LLM 调用一些外部函数来检索必要的信息，以回答问题并完成任务。因此，必须学会如何指示 LLM 调用外部函数。

自定义函数调用

OpenAI 原生就支持函数调用。Python LLM 框架 LangChain（本书以后会详细讨论）也提供了自己的函数调用支持，称为工具（tool）。由于各种不同的原因，你有时可能需要编写自己的函数调用版本。例如，当模型不是来自 OpenAI 的时候。



注意：可以先尝试自己构建一些简单的实现，对 OpenAI 函数调用背后发生的事情有一个底。

第一次尝试

首先让我们明确一点：没有 LLM 能够执行任何代码。相反，LLM 可以输出代码，并作为一种文本触发器传递给其他工具或系统，并由它们负责代码的执行。

自定义函数调用的一般方案如下：

1. 描述函数目标。
2. 描述函数接受的模式和参数，并给出示例（通常是 JSON 格式）。
3. 将关于函数及其参数的描述注入“系统提示”中，并告知 LLM 它可以使用该函数来解决分配给它的任何任务（通常是回答或者为最终用户提供支持）。在这个步骤中，重要的

一点是要求模型在需要调用函数来完成任务时返回一个结构化的输出（函数的实参）。

4. 解析来自聊天补全（或简单的纯文本补全）的响应，检查其中是否包含你正在寻找的结构化输出（在函数调用的情况下）。如果包含，那么反序列化该输出（如果使用的是 JSON 格式），并手动执行函数调用。如果不包含，那么继续进行。

5. 解析函数的结果（如果有的话），并将其传递给新创建的消息（以“用户消息”的形式），消息内容大致为：“鉴于外部工具提供的以下结果，你能回答我最初的问题吗？”

下面展示了一个完整的“系统提示”模板，其中包含几个示例函数（获取天气、读取电子邮件或查看最新股价）。

你是一个乐于助人的助手。你的任务是友好地与用户交谈。如果用户的请求需要的话，你可以使用外部工具来回答他们的问题。向用户提问以收集使用这些工具所需的参数。你只能使用以下工具：

>>天气预报访问：当用户询问天气时使用此工具，用户应提供感兴趣的城市和日期。要使用此工具，你必须提供以下参数中的至少一个：['city', 'startDate', 'endDate']

>>电子邮件访问：当用户询问与他们的电子邮件有关的信息时使用此工具，可能需要指定时间范围。要使用此工具，你可以指定以下参数中的一个，但不一定两个都需要：['startTime', 'endTime']

>>股市报价访问：当用户询问有关股市的信息时使用此工具，指定股票名称、指数和时间范围。要使用此工具，你必须提供以下参数中的至少三个：['stock_name', 'index_name', 'startDate', 'endDate']

响应格式说明 -----

****选项 1:****

如果你想要使用一个工具，就使用这个选项。

请采用以下格式的 Markdown 代码片段：

```
```json
{
 "tool": string \ 指定要使用的工具。必须是以下之一：Weather, Email, StockMarket
 "tool_input": string \ 要执行的动作的输入，格式为 json
}
```

**\*\*选项 2:\*\***

如果你想要直接回应用户，就使用这个选项。

请采用以下格式的 Markdown 代码片段：

```
```json
{
  "tool": "Answer",
  "tool_input": string \ 应该在这里放入你想返回给用户的内容
}
```

用户的输入 -----

以下是用户的输入（记住，只用包含单个动作的 json blob Markdown 代码片段进行响应，除此之外不要添加任何内容）：

然后，需要设计一个 `switch` 逻辑来决定如何继续。如果返回的是一个工具，那么必须调用一个函数；否则，应直接向用户返回响应。

现在，我们已经准备好完成这个自定义链的最后一个环节了。如果是作为函数/工具使用，函数的输出必须传递给一个新的调用，并请求 LLM 返回一条适当的消息。下面展示了一个示例 C#控制台应用程序（假定聊天机器人名为“小爱”）。^①

```
// 基于本书之前设置的部署名来创建一个 ChatCompletionsOptions 实例
// 传递刚才的大段“系统提示”
var systemPrompt = @"替换为刚才的系统提示"
var chatCompletionsOptions = new ChatCompletionsOptions
{
    DeploymentName = AOAI_DEPLOYMENTID,
    Messages = { new ChatRequestSystemMessage(systemPrompt) }
};

while (true)
{
    // 读取用户输入的消息
    var userMessage = Console.ReadLine();

    // 将用户输入的消息添加到补全消息中
    chatCompletionsOptions.Messages.Add(new ChatRequestUserMessage(userMessage));

    // 输出"小爱: "
    Console.WriteLine("小爱: ");

    // 异步获取聊天补全响应
    var chatCompletionsResponse = await openAIClient.GetChatCompletionsAsync(
        chatCompletionsOptions);

    // 获取 LLM 响应消息
    var llmResponse = chatCompletionsResponse.Value.Choices[0].Message;

    // 将响应的内容反序列化为动态对象
    var deserializedResponse =
        JsonConvert.DeserializeObject<dynamic>(llmResponse.Content);

    // 一直持续，直到从 LLM 获得最终答案，可能需要多次调用
    while (deserializedResponse.Tool != "Answer")
    {
        var tempResponse = "";
    }
}
```

^① 译注：作者的这个示例程序需要用 `nuget` 安装名为 `Newtonsoft.Json` 的一个包，并添加相应的 `using` 指令。

```

switch (deserializedResponse.Tool)
{
    case "Weather":
        // 工具输入作为序列化的 json
        var functionResponse = GetWeather(deserializedResponse.ToolInput);

        // 构造询问最终答案的消息
        var getAnswerMessage = @"
给定以下工具响应：
-----
" + functionResponse + @"
-----
我上一条消息的回应是什么？
记住，只用包含单个动作的 json blob Markdown
代码片段进行响应，除此之外不要添加任何内容。
";

        // 将构造好的消息添加到补全消息中
        chatCompletionsOptions.Messages.Add(new
            ChatRequestUserMessage(getAnswerMessage));

        // 同步获取聊天补全响应并提取内容
        tempResponse =
            openAIClient.GetChatCompletionsAsync(chatCompletionsOptions).
                Result.Value.Choices[0].Message.Content;

        // 再次反序列化响应内容
        deserializedResponse = JsonConvert
            .DeserializeObject<dynamic>(tempResponse.Content);

        break;

    case "Email":
        // 同上
        break;

    case "StockMarket":
        // 同上
        break;
}
}

// 现在，我们得到了给用户的最终响应
var responseForUser = deserializedResponse.ToolInput;

// 向用户输出最终响应
Console.WriteLine(responseForUser);

// 将最终响应添加到补全消息中
chatCompletionsOptions.Messages.Add(new

```

```
ChatRequestAssistantMessage(responseForUser));

// 提示用户输入新消息
Console.WriteLine("请输入新消息: ");
}
```



注意：在本例中，可以将 `ResponseFormat` 属性设为 `ChatCompletionResponseFormat.JsonObject` 来使用 JSON 模式，强制 OpenAI 模型返回有效的 JSON。总的来说，可以在这里使用和本书之前的章节介绍过的同样的提示工程技术。另外，当处理结构化输出时，可以考虑设置一个较低的温度——设为 0 其实都不会有太大问题。

改进代码

前面的例子虽然很“简陋”，但基本思路无误。不过，它在代码的整洁度和管理上仍有改进的空间。例如，传递给模型的可调用函数（或者更准确地说，传递给模型的对可调用函数的描述）其实可以直接使用 XML 注释来描述，开发人员可以在每个方法上方编写这样的注释。另一个方案是为传递给每个函数的参数提供一个 JSON 模式（包括可选和必需字段）。还有一个方案是向模型提供函数的完整源代码，使其确切了解每个函数所做的事情。为了提高代码的整洁度，还应考虑日志记录和错误处理，并对由模型生成并传递给函数的 JSON 参数的有效性进行验证。

从模型的角度看，应该在“系统提示”中添加更多指令来强制使用外部函数。否则，模型可能会试图自主回应，甚至编造答案。此外，通过之前的代码，你已经注意到函数调用可以是迭代式的。但是，在停止执行并返回错误消息或一个不同的提示前，你应该自行决定最多允许多少次迭代。

OpenAI 风格

LangChain 库有自己的工具调用机制，涵盖了上一节提到的大部分建议。OpenAI 还对其模型的最新版本进行了微调以支持函数调用。具体来说，模型现在能根据提示的上下文判断何时以及如何调用函数。如果请求中包含了函数，那么模型会用一个包含了函数实参的 JSON 对象来响应。由于是官方对模型进行的微调，所以这种原生的函数调用机制通常比自定义版本更可靠。

当然，若使用的是非 OpenAI 的模型，那么还是需要使用自定义函数调用或者 LangChain 的工具。在需要更快的推理速度并且必须使用本地模型的时候，这恐怕是你唯一的选择。

基础知识

那些较老的 OpenAI 模型没有经过函数调用方面的微调。只有较新的模型才做了这方面的微调，其中包括：

- gpt-4o-turbo
- gpt-4
- gpt-4o-32k
- gpt-4o-turbo
- gpt-4o-turbo-16k

为了使用 Chat Completions API 进行函数调用，必须在自己的请求中包含一个新的名为 `Tools` 的属性。可以在该属性中指定多个函数。函数的细节使用特定的语法注入到系统消息中，具体语法与自定义版本大致相同。注意，函数会占用 `token` 使用量，但可以使用提示工程技术来优化性能。通过提供更多的上下文或函数的细节，模型可以更好地判断是否应该调用某个函数。

默认情况下，模型会自行决定是否调用函数，但我们可以通过 `ToolChoice` 参数来控制这一行为。该参数可以设置为 `"auto"`（自动判断）、`{"name": "<function-name>"}`（强制指定一个函数）或者 `"none"`（不调用函数，即使请求中包含了工具函数）。



注意： `Tools` 和 `ToolChoice` 以前被称为 `Functions` 和 `FunctionCall`。

一个可以实际工作的示例

可以使用以下经过完善的“系统提示”得到与之前相同的结果。注意，无需在提示中包含任何函数指令：

你是一个乐于助人的助手。你的任务是友好地与用户交谈。

随同上述系统提示，使用以下代码来定义和描述函数：

```
// *** 定义函数 ***
var getWeatherFunction = new ChatCompletionsFunctionToolDefinition();
getWeatherFunction.Name = "GetWeather";
getWeatherFunction.Description =
    "当用户询问天气信息或预报时使用此工具，提供感兴趣的城市和时间范围。";
getWeatherFunction.Parameters = BinaryData.FromObjectAsJson(new JsonObject
{
    ["type"] = "object",
    ["properties"] = new JsonObject
    {
```

```

        ["WeatherInfoRequest"] = new JsonObject
        {
            ["type"] = "object",
            ["properties"] = new JsonObject
            {
                ["city"] = new JsonObject
                {
                    ["type"] = "string",
                    ["description"] = @"用户想要查询天气的城市。"
                },
                ["startDate"] = new JsonObject
                {
                    ["type"] = "date",
                    ["description"] = @"用户感兴趣的天气预报的开始日期。"
                },
                ["endDate"] = new JsonObject
                {
                    ["type"] = "date",
                    ["description"] = @"用户感兴趣的天气预报的结束日期。"
                }
            },
            ["required"] = new JsonArray { "city" }
        },
        ["required"] = new JsonArray { "WeatherInfoRequest" }
    });

```

可以将上述定义与真实的 Chat Completion API 调用结合起来使用：

```

using Azure;
using Azure.AI.OpenAI;
using System.Text.Json.Nodes;
using System.Text.Json;

var endpoint = new Uri(AOAI_ENDPOINT);
var credentials = new AzureKeyCredential(AOAI_KEY);
var client = new OpenAIClient(endpoint, credentials);

var chatCompletionsOptions = new ChatCompletionsOptions()
{
    DeploymentName = AOAI_DEPLOYMENTID,
    Temperature = 0,
    MaxTokens = 1000,
    Tools = { getWeatherFunction },
    ToolChoice = ChatCompletionsToolChoice.Auto
};

// 外部的补全调用
var chatCompletionsResponse = await
    client.GetChatCompletionsAsync(chatCompletionsOptions);

```

```

var llmResponse = chatCompletionsResponse.Value.Choices.FirstOrDefault();

// 判断 ChatGPT 的响应是否想要调用一个函数
if (llmResponse.FinishReason == CompletionsFinishReason.ToolCalls)
{
    // 这个 while 允许 GPT 连续调用多个函数
    bool functionCallingComplete = false;
    while (!functionCallingComplete)
    {
        // 将带有工具调用的助手消息添加到对话历史记录中
        ChatRequestAssistantMessage toolCallHistoryMessage = new(llmResponse.Message);
        chatCompletionsOptions.Messages.Add(toolCallHistoryMessage);

        // 这个 foreach 允许 GPT 并行调用多个函数
        foreach (ChatCompletionsToolCall functionCall in llmResponse.Message.ToolCalls)
        {
            // 获取函数调用实参
            var functionArgs =
                ((ChatCompletionsFunctionToolCall)functionCall).Arguments;

            // 用于保存函数结果的变量
            string functionResult = "";

            // 对函数实参（JSON 格式的字符串）进行反序列化，获得对应的 .NET 对象
            var weatherInfoRequest =
                JsonSerializer.Deserialize<WeatherInfoRequest>(functionArgs);
            if (weatherInfoRequest != null)
                functionResult = GetWeather(weatherInfoRequest);

            // 将函数响应添加到对话历史记录中，这是
            // 模型为用户制定最终答案所必须的。
            var chatFunctionMessage = new ChatRequestToolMessage(functionResult,
                functionCall.Id);
            chatCompletionsOptions.Messages.Add(chatFunctionMessage);
        }

        // 再次进行 Chat Completion 调用，看看接下来会发生什么
        var innerCompletionCall = client.GetChatCompletionsAsync(chatCompletionsOptions)
            .Result.Value.Choices.FirstOrDefault();

        // 创建一个新的消息对象，并将响应添加到消息列表中
        if (innerCompletionCall.Message != null)
        {
            chatCompletionsOptions.Messages.Add(new
                ChatRequestAssistantMessage(innerCompletionCall.Message));
        }

        // 退出循环
        if (innerCompletionCall.FinishReason != CompletionsFinishReason.ToolCalls)
        {

```

```
        functionCallingComplete = true;
    }
}
}
```

与自定义版本相比，官方微调版本有三个主要区别：

- 无需在“系统提示”消息中手动指定函数的任何细节。
- 必须将函数添加到调用选项中。
- 只需向模型添加角色为 `Function` 的一条聊天消息就可以了。当这条带有“`Function`”角色的消息被添加到聊天中时，如果发现需要调用相应的功能（例如，如果发现用户想要查天气），模型就会重新处理这条消息。这种重新处理不仅仅是简单地回复用户的问题，而是执行一些更复杂的操作，例如调用外部 API、执行计算任务或者整合多个数据源的信息等。



注意：在上述示例中，还应处理最终答案的可视化，并包括一些 `try-catch` 块来捕捉异常和错误。

更一般地说，上述示例（以及你在网上找到的大多数示例）都需要从软件工程的角度进行重构。换言之，需要使用更高级的 API——可能是一个流式 API。本书以后会提供一个完整的可工作的示例，它演示了如何使用流式的定制 API 来调用 LLM。^①

安全性方面的考虑

若授权 LLM 访问一个轻量级的函数，后者直接执行主要业务逻辑并与数据库进行交互，那么是相当危险的。这种做法的风险很大，原因有两个。首先，LLM 容易犯错；其次，类似于早期的网站，容易受到注入和其他形式的黑客攻击。

一种更安全的做法是指示 LLM 调用一个 API 层（API layer），从而严格控制业务逻辑的执行，同时管理数据访问权限。事实上，这样的 API 层也许已经存在了，不需要从头开始建立。例如，为了回答“这个月我的业绩前五名客户是谁？”，你有两个选择：

- 让 LLM 生成一个合适的 SQL 查询，并将其作为单一的字符串参数传递给函数调用。接下来，作为字符串的查询直接传递给数据库引擎执行，例如

^① 译注：流式 API 也称为流畅 API（Fluent API），即下一个函数调用的上下文基于上一个函数调用准备好的上下文。整个调用链通过一个空的上下文来终止。

`execute_sql(query_sql: string)`。

- 定义一个结构化的函数调用，例如 `get_customers_by_revenue(start_date: string, end_date: string, limit: int)`，这样 LLM 将只传递受控数据。

第一种方法提供了更大的灵活性，但带来了巨大的安全风险。本书以后会更详细地介绍这两种方法以及可能的折中方案。

与函数调用相关的另一个风险是模型所采取的行动可能不是用户所期望的，即使此时正由一个安全的 API 层控制着所有可能的行动。为了解决这个问题，通常需要向最终用户显示一条确认消息，这就是所谓的“人环”（human-in-the-loop）方法，即人类在自动过程的关键阶段介入，对算法的结果进行审核、修正或确认。

与(单独的)数据对话

在许多使用场景中，用户需要与企业自己的数据进行交互，这称为“与数据对话”（talking to data）。这些数据的例子包括发票、产品明细或者敏感的病历等，它们独立于 LLM 存储。LLM 可以访问存储的这些数据，并自主响应用户查询。这超越了传统的信息检索，开创了一种全新的使用场景。

将数据连接到 LLM

LLM 大多基于公开数据进行训练，对于企业的定制数据则缺乏了解。这导致了通用模型与特定领域数据之间存在清晰的界限。由于 LLM 是为通用目的而训练的，只有在需要的时候才会手动连接到定制数据，所以定制数据会一直处于外部，不会在 LLM 中留下永久的痕迹，这确保了数据的隔离与安全



注意：在更传统的机器学习解决方案中，情况则并非如此。在那些方案中，模型是基于领域的特定数据而构建的，训练数据必须与生产数据相似，才能获得合理的预测。

现在的关键是如何将 LLM 连接到数据。这需要分两步完成。首先，我们搜索与用户问题/请求相关的数据。然后，LLM 生成一个符合当前语境（上下文）的响应。

这个过程也可以通过对模型进行大量微调来一步到位地完成，但强烈建议不要这样做。它的可扩展性较差，也不可靠。原因很简单，当新数据到来时，就必须重新训练模型。而且最严重的是，谁都能访问这些敏感和私有的数据。相比之下，通过两步过程，我们可以隔离敏感数据，只让特定的和获得了授权的用户访问。除此之外，还能跟踪每个答案都是由

哪些数据贡献的。

将数据检索和生成响应分离还有一个好处，即可以更容易地向用户显示所引用的来源文档。这种方法和过程被称为基础知识定位（grounding）或者检索增强生成（Retrieval Augmented Generation, RAG）。^①

嵌入

在机器学习中，嵌入（embedding）是指将数据（如词/字）转换到不同空间中之后的数值形式。注意，这里的“嵌入”是名词而非动词。这种转换通常涉及将数据从一个高维空间（如词汇量大小）映射到一个较低维的（向量）空间，即嵌入大小（embedding size）。嵌入的目的是捕捉和压缩数据的基本特征，使模型更容易处理和学习不同的模式。

嵌入的典型使用场景包括语义搜索、推荐系统、异常检测^②和知识图谱。在对象（物体）识别、图像聚类 and 语音识别中，嵌入也是至关重要的。

在涉及到自然语言时，可能表面上看起来“嵌入”的维度很低，因其只涉及到词/字。然而，实际并非如此。一个句子不仅仅是单词的随机组合；相反，每个句子都必须遵循一组语法规则，并携带了语义，后者在很大程度上取决于其所处的上下文。OpenAI 的嵌入模型（例如，text-embedding-ada-002）会将一段文本映射到一个包含 1536 个维度的空间。换句话说，任何一段文本都变成了一个包含 1536 个浮点值的数值向量。

词嵌入（word embedding）是词在连续向量空间中的纯数值表示。在嵌入空间中，意义或上下文相似的词彼此更接近。这些嵌入是通过使用无监督学习技术从大量文本中学习得到的，目的是捕捉语义。结果就是，相似的词具有相似的向量表示。例如，“猫”和“狗”这两个嵌入之间的距离比“猫”和“桌子”这两个嵌入之间的距离更小。

词嵌入保留了词与词之间的关系。例如，向量运算“国王 - 男人 + 女人”的结果接近“女王”的向量形式，如图 3-1 所示。

^① 译注：RAG 是目前所有号称能“一次性读懂 200 万甚至上千万字的文档”的大模型所采用的基本技术。事实上，将整本《红楼梦》PDF 传给这种模型，它根本不可能完全“读懂”。相反，它只是使用 RAG 技术，跑去网上其他地方参考现有的总结性文章，对自己的输出“增强”一下，然后便假装“读懂”了整个文档。

^② 译注：“异常检查”有一个经典的例子，即信用卡盗刷检测。

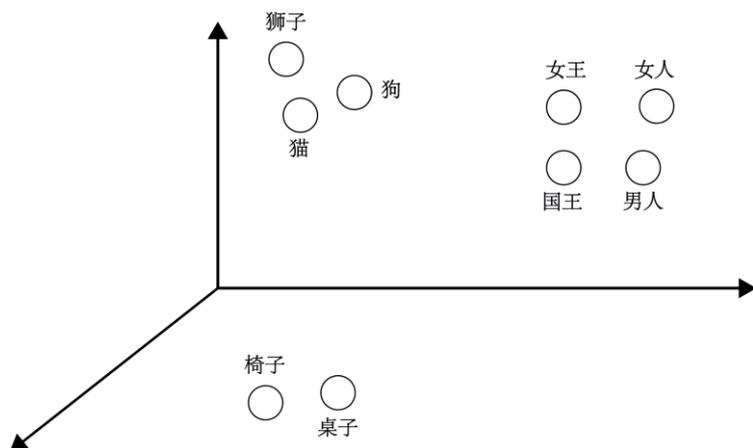


图 3-1 嵌入向量空间中词与词之间的距离

除了静态词嵌入，现在还有上下文词嵌入。实际上，这些才是最常用的。它们捕捉上下文信息，考虑整个句子的上下文而不仅仅是单独词/字的局部上下文。一些嵌入模型还可以处理代码和多模态内容。

语义搜索与检索

语义搜索（semantic search）能理解搜索查询（即你要搜索什么）和可搜索（已索引）内容的含义和上下文，从而提高搜索结果的相关性和准确性。语义搜索通常利用词/字/句子嵌入来识别语义部分，从而实现更复杂和具有上下文意识的搜索过程。

使用嵌入方法的语义搜索引擎一般采取以下步骤：

1. 嵌入生成：上下文嵌入（如 OpenAI 的 text-embedding-ada-002 模型）将词、字、短语或句子转换成密集数值表示。^①
2. 索引：将要搜索的文档或数据转换成嵌入，并连同原始的自然语言文本一起存储到索引中，创建数据集基于嵌入的表示。
3. 语义相似度：在搜索过程中，用户的查询被转换成嵌入（使用与数据集相同的模型）。然后，系统计算“查询嵌入”与“索引文档嵌入”之间的语义相似度。至少在普通的使用

^① 译注：在自然语言处理（NLP）中，密集数值表示（dense numerical representations）是指将词语、短语或句子转换为由实数组成的紧凑向量，这些向量通常位于高维空间中。这种表示形式与稀疏表示形式相对，稀疏表示形式通常用于传统的词袋模型，其中大多数元素为零。

场景下，相似度是通过余弦相似度来计算的（将在下一节中讨论）。

4. 排名/打分：基于计算出的语义相似度，搜索引擎对索引中的文档进行排名/打分，将最具语义相关性的文档置于搜索结果的顶部。

也可以使用不涉及嵌入的其他方法来构建适当的语义搜索。其中一种方法可能是使用专用搜索引擎，例如 Azure AI Search^①。它依赖于 Bing 搜索引擎，兼顾元数据（例如，时间戳、创建者、附加描述等）和关键词搜索来对结果进行排名/打分。

一种更为传统但同样有效的方法是词频-逆文档频率（Term Frequency-Inverse Document Frequency, TF-IDF）。TF-IDF 兼顾一个词在文档中的频率（TF）和它在整个语料库中的逆频率（IDF）。TF-IDF 通过强调文档中重要且独特的词语，帮助系统根据文档间的共享内容和独特特征，检索出与查询在语义上更加相似的文档。^②

要使用 TF-IDF 来构建语义搜索，你应该使用 TF-IDF 向量化来创建文档-词项矩阵，并为每个词项计算 TF-IDF 分数。以类似的方式处理用户的查询。然后，使用相同的 IDF 值计算查询词项的 TF-IDF 分数。最后，使用余弦相似度计算查询与文档之间的相似度分数，根据它们的分数对结果进行排名，以呈现最相关的文档。

另一种方法是构建一个直接且专门为语义搜索优化的神经网络，该网络在与现实世界示例相似的数据上进行训练。这种方法通常称为神经数据库（neural database），其最大的挑战在于训练。它可能需要大量语义相似的、做好标签的数据（尽管可以尝试自监督的方法）和至少 20 亿个神经网络参数来进行训练。但是，它的优点也是很突出的，其中包括搜索时间显著缩短（因为不需要计算嵌入和相似度函数），而且只需很少的空间来存储神经网络和作为搜索目标的物理文档^③。

^① 译注：以前称作 Azure Cognitive Search。该服务可在 Azure 门户（<https://portal.azure.com>）创建。试用版用户可以免费创建搜索服务，但存在一定的限制。

^② 译注：在特定文档中出现频率（TF）较高，同时在整体文档集合中分布频率（IDF）较低的词语，能够生成较高的 TF-IDF 权重值。这一机制促使 TF-IDF 有效地筛选出那些普遍存在的常规词语，强调那些反映文档独特主题或关键信息的词语。因此，TF-IDF 不仅倾向于过滤常见的“噪声”词，还擅长突显文档中的特色内容和关键词，从而增强信息检索和文本分析的精准度与相关性。

^③ 译注：对于目标文档，神经数据库可能只需存储基本的索引和引用信息，而无需存储大量的衍生数据（如嵌入向量）。这大幅减少了存储需求。

度量相似度

使用嵌入方法时，我们通常使用 k 最近邻 (k -Nearest Neighbor, KNN) 算法在数据集中搜索语义相似的元素。更准确地说，由于涉及的索引数据点通常较多，使用的是近似最近邻 (Approximate Nearest Neighbor, ANN) 算法。KNN 和 ANN 通常基于余弦相似度来进行计算。

余弦相似度是用于评估多维空间中两个向量的相似度的一种数学度量。它通过计算两向量间夹角的余弦值得出一个介于 -1 与 1 之间的结果。当向量指向完全一致 (共线同向) 时，余弦相似度趋近于 1，表明二者极度相似；而当它们相互正交 (彼此垂直) 时，则该值为 0，表示无相似度。此函数通过衡量角度的余弦值来高效捕获向量间的相似度，侧重于向量的方向性和排列方式，而非其大小，因而在识别向量长度的变异与突出相对定向方面展现出高度的健壮性。

可以使用多种相似度度量对文档进行过滤和排名 (打分)。例如，可以在嵌入上训练支持向量机 (Support Vector Machine, SVM)，而不是使用余弦相似度，从而获得更好的结果。这虽然会涉及更多的计算，因其需要为每个查询训练一个 SVM，但最终能生成更准确和更相关的结果。

使用场景

嵌入和语义搜索有多个使用场景，例如个性化产品推荐和信息检索系统。例如，在个性化产品推荐中，通过将产品表示为密集向量并采用语义搜索，平台可以有效地匹配用户偏好和过往的购买行为，从而为其创造定制化的购物体验。在信息检索系统中，嵌入和语义搜索基于语义内容对文档进行索引，并检索与用户查询高度相似的相关结果，从而提高信息检索的准确性和效率。

这些技术还可以用于多模态内容，例如视频和图像。例如，可以使用 OpenAI Whisper 这样的语音转文本服务来转录视频内容，或者使用 AI 服务描述图像，然后嵌入这些信息，执行相似内容的语义搜索。

作为一个例子，以下代码使用 Azure OpenAI 部署 (使用 text-embedding-ada-002 模型) 为两个句子生成嵌入，并计算它们之间的距离。

```
EmbeddingsOptions sentence1Options = new EmbeddingsOptions(AOAI_embeddings_DEPLOYMENTID,
    new List<string> {"She works in tech since 2010, after graduating"});
EmbeddingsOptions sentence2Options = new EmbeddingsOptions(AOAI_embeddings_DEPLOYMENTID,
    new List<string>{"Verify inputs don't exceed the maximum length"});
var sentence1 = openAIClient.GetEmbeddings(sentence1Options);
var sentence2 = openAIClient.GetEmbeddings(sentence2Options);
double dot = 0.0f;
for (int n = 0; n < sentence1.Value.Data[0].Embedding.Span.Length; n++)
{
```

```
dot += sentence1.Value.Data[0].Embedding.Span[n] *
      sentence2.Value.Data[0].Embedding.Span[n];
}
Console.WriteLine(dot);
```

运行上述代码，会看到余弦相似度（或点积，因为当嵌入向量归一化为 1-范数^①时，它们是等价的）大约是 0.65。^②但是，如果增加更多语义上相似的句子进行比较，可能会看到余弦相似度值的提高，这进一步验证了模型对于语义相似度的捕捉能力。

从技术上说，还需要考虑 OpenAI 嵌入模型允许的输入文本的最大长度——当前为 2048 个 token（相当于两到三页文本）。在你的应用中，应该总是验证输入没有超过这个限制。

潜在的问题

语义检索向我们提出了几个挑战，必须解决这些挑战才能获得最佳性能。其中一些挑战仅与嵌入部分相关，另一些则与存储和检索阶段相关。

其中一个挑战是在嵌入完整的长文本时可能丢失宝贵信息，这要求对文本进行分块（chunking）以保留上下文。将文本分成更小的块有助于保持相关性。另外，在生成过程中通过向 LLM 发送相关的块，基于已经获得的相关信息来向用户提供答案，还能节省一定的生成成本。

为了确保每个块内捕获的都是最相关的信息，建议采用一种滑动窗口方法进行分块。这允许内容重叠，从而更有可能保留上下文并增强搜索结果。

在包含嵌套小节的结构化文档中，提供额外的上下文（例如，章和小节的标题）可以显著提高检索的准确性。解析并将这些上下文添加到每个块中，可以使语义搜索更好地理解文档各小节之间的层次结构和关系。

检索到的块也许单独呈现出与用户的“查询”的语义相关性，但把它们组合起来后，可能

^① 译注：1-范数（1-norm），又称为曼哈顿距离（Manhattan norm）或曼哈顿长度，是向量空间中的一种范数，用于衡量一个向量中所有元素绝对值的总和。在数学和计算机科学中，1-范数提供了一种度量向量元素的方法，适用于多种不同的应用场景。点积（dot product）是两个向量相乘的结果，即向量 A 和向量 B 的对应元素的乘积之和。在向量被归一化后（即每个向量的长度或范数被调整为 1），这个点积的值就等同于两个向量夹角的余弦值。余弦值为 1 表示两向量完全相同的方向，-1 表示完全相反，0 表示两向量正交（即无相关性）。

^② 译注：当余弦相似度为 0.65 时，这表明两个句子在语义上有相似度，但不是非常强烈的相似度。这可能因为两个句子共有一些语义特征或用词，但也有不少不同点。例如，两个句子可能都提到了一些关于业务或技术方面的内容，但其上下文或详细语义有所不同。

形不成一个连贯的上下文。在处理一般性的查询而非特定的信息请求时，这一挑战尤为突出。为了克服这种连贯性的缺失，可以采用“摘要”（汇总）策略。这涉及到生成块来包含对文档中较大小节的摘要，而不是简单地随机或基于某些标准来分割文本。这样做的好处是，每个块都能捕捉到它所代表的文档小节的核心信息。这使得数据的表达更为简洁，有助于实现更高效和更准确的检索。

要考虑的另一个重点是与嵌入和向量数据库相关的成本。在 1536 个维度中生成和存储嵌入的浮点值，会增大 token 和存储的成本。此外，如果使用分开管理的服务来处理嵌入和向量数据库，那么会造成隐私方面的风险，导致数据在不同位置重复。^①

使用相同版本的 LLM 对用户查询和文档进行嵌入也是至关重要的。如果 LLM 有任何更新或更改，那么必须从头开始重建向量数据库中的所有嵌入。

向量存储

向量存储（vector store）或向量数据库（vector database）负责存储嵌入数据并执行向量搜索。它们以向量形式（嵌入）存储和索引文档，确保了能高效地进行相似度搜索和文档检索。添加到向量存储的文档还可以携带一些元数据，例如原始文本、额外描述、标签、分类、必要的权限、时间戳等。这些元数据可以在预处理阶段由另一个 LLM 调用生成。

向量存储和关系数据库之间的主要区别在于两者的数据表示和查询能力。在关系数据库中，数据使用结构化的行和列来存储。而在向量存储中，数据以数值向量的形式表示。每个向量都对应一个特定的项或文档，并包含一组捕获其特征或属性的数值特征（numeric feature）。

在关系数据库中，查询通常基于精确匹配或关系操作。而在向量存储中，查询则基于向量的相似度。向量存储通过一个内建的相似度搜索功能来进行检索，它针对基于相似度的操作进行了优化，并支持向量索引技术（例如，ANN）。



注意：向量存储是专为向量的高效存储和检索而设计的专用数据库。相比之下，NoSQL 数据库^②涵盖更广泛的类别，为处理多样化的数据类型和结构提供了灵活性，

^① 译注：在许多现代数据处理架构中，不同的服务或组件可能由不同的供应商或内部团队管理。例如，一个服务可能专门用于生成和管理数据的嵌入，而另一个服务则负责存储和检索这些嵌入，即向量数据库。当这些服务由不同的管理实体掌控时，数据需要在这些服务之间传输和同步。

^② 译注：NoSQL 数据库是指“Not only SQL”。和传统的关系数据库不同的是，它允许部分数据使用 SQL 系统存储，而其他数据允许使用 NoSQL 系统存储。例如，Redis 就是一种 NoSQL 数据库。

不特别针对某一数据类型进行优化。

基本方法

如果处理的向量只有几千个，一个非常简单的方法是使用 SQL Server 及其**列存储**（columnstore）索引技术^①。结果表可能如下所示：

```
CREATE TABLE [dbo].[embeddings_vectors]
(
    [main_entity_id] [int] NOT NULL, -- 对嵌入实体 ID 的引用
    [vector_value_id] [int] NOT NULL,
    [vector_value] [float] NOT NULL
)
```

对于检索，可以设置一个存储过程来计算最相似的前 N 个向量。余弦相似度是通过点积（向量间的元素相乘后求和）计算得出的，公式如下： $\text{sum}(\text{queryVector}.[\text{vector_value}] * \text{dbVector}.[\text{vector_value}])$ 。



注意：注意，当嵌入向量归一化为 1-范数时，余弦相似度就等价于点积。

现在，你应该对整个过程的工作方式有了初步的了解。然而，当数据量变得更大时，SQL Server 的索引性能可能就不太理想了。这时候，就需要使用专门的向量数据库了。



注意：在索引和搜索中，核心原则是计算向量之间的相似度。索引过程采用高效的数据结构（例如，树和图）来组织向量。这些结构旨在加速最近邻或相似向量的检索过程，避免了与数据库中每个向量进行穷举比较的必要。常见的索引方法包括 KNN 算法，它将数据库划分为由中心向量（centroid vector）表示的聚类，实现对局部区域的快速搜索；以及 ANN 算法，它寻求近似最近邻以更快地检索，只是精确度有轻微

^① 译注：和传统的行存储不同，列存储数据库将数据按列分开存储。例如，一个有三列的表会把每一列的数据分别存储在一起。这意味着同一列的数据被连续存储，而不是按照整行数据存储。

的损失。在 ANN 算法中，通常会使用像局部敏感哈希（Locality-Sensitive Hashing, LSH）这样的技术来有效地分组可能相似的向量。

当然，在选择特定的向量数据库解决方案时，还应该到考虑扩展性、性能、易用性以及当前实现的兼容性。

商业和开源解决方案

现在已经有许多专用的向量数据库，例如 Chroma、Pinecone、Milvus、QDrant 和 Weaviate 已经出现。Pinecone 是一种软件即服务（Software-as-a-Service, SaaS）解决方案。Weaviate、QDrant 和 Milvus 都是开源的，它们很容易部署，而且通常都支持私有云，并且有它们自己的 SaaS 方案。Chroma 类似于向量数据库版本的 SQLite。因此，对于扩展需求较小的简单项目来说，它可能是一个很好的选择。

一些知名的产品（例如，Elasticsearch）现在也开始支持向量索引。Redis、Postgres 和其他许多数据库也支持向量索引，无论这个支持是原生的还是通过插件。

目前，Microsoft Azure 在这一领域提供了几个预览版的产品，其中包括 Azure AI Search 和 Cosmos DB。AI Search 最近已被扩展为专用的向量数据库，而 Cosmos DB 作为一个通用的文档数据库，现在也具备了向量索引的功能。

改进检索

除了内部向量存储的检索机制外，还有多种方法可以提高用户获得的输出的质量。之前已经提到过，支持向量机（SVM）能显著提升性能。然而，这并不是唯一可用的技术。除此之外，还可以结合多种技术以获得更好的性能。例如，在存储阶段，可以转换要嵌入的信息，比如对块进行摘要或改写（重新措辞）；在检索阶段，则可以修改用户的查询以改进结果。

例如，在检索步骤中，多样性对于结果的排名非常关键，目的是确保相关信息的全面和均衡表现。这有助于避免重复的回应和带有偏见的结果，为用户提供一个平衡和多样化的结果集^①。提高多样性有助于增强系统的健壮性，减轻过拟合，并提供信息更丰富、更令人满意的用户体验。为了实现多样性，一个办法是使用最大边际相关性（Maximum Marginal

^① 译注：例如，在内容检索系统中，如果用户查询一个常规的话题，系统应该展示来自不同领域和视角的内容，而不是仅仅集中在最常见或最受欢迎的几个答案上。这不仅有助于用户获得更全面的信息，也避免了因信息过于单一而造成的偏见。

Relevance, MMR) 算法, 该算法可以在 LangChain 中使用, 并且可以在 C# Semantic Kernel 中轻松地复制。从本质上讲, 该算法首先识别并返回与输入 (用户的查询或其它输入向量) 具有最高余弦相似度的嵌入实例 (即数据库中的文档或数据项)。注意, 选定的实例不是一次性返回的, 而是通过迭代过程逐步添加的。在每次迭代中, 算法都会计算当前未选择的实例与已选择实例的相似度。对于那些与已经选中的实例过于接近的实例, 算法会施加一种“惩罚”。这意味着如果一个实例与已选择的实例在内容上太相近, 则其被选中的可能性会降低。这个惩罚机制有助于确保新增的实例在内容上能够提供新的或互补的信息, 从而增加整体结果的多样性。

另一个提高相关性的方案是元数据过滤 (metadata filtering)。为此, 可以指示 LLM 从用户的查询中提取相关的过滤器^①, 然后将这些过滤器传递给“向量存储”。

将陌生信息 (例如, 一个专有名词) 融合到嵌入中可能会带来挑战。尽管语言模型可以为每个 token 生成嵌入, 但假如模型缺乏对 token 的重要性的理解, 那么这些嵌入表示可能就缺乏实质性的语义信息。为了解决这一问题, 一个合理的策略是将语义搜索与更为传统的关键词搜索结合起来, 同时利用元数据进行辅助。^②

还可以考虑对中等相关性的块进行压缩或摘要, 因为通常只有块的一部分与用户查询相关。以一次 LLM 调用的成本^③, 你可以要求模型提取与用户查询相关的信息。另一个选择是要求一个独立的大语言模型 (LLM) 对用户的查询进行改写 (重新措辞), 使其更贴近已存储文档的语气和结构。



注意：最后三个方案都有一定的成本, 一个是对用户体验的影响 (因为有延迟), 一个是因为 API 调用而产生的基于 token 的费用。LangChain 内置了对这些检索微调任务的支持 (当然, API 调用的费用自付)。

^① 译注: 理解为“过滤条件”就好。例如, 假设用户查询“成都最好吃的串串”, 那么可以提取出“地点: 成都”和“食物: 串串”这两个“过滤器”。

^② 译注: 例如, 如果用户查询中包含一个少见的作者的名字, 那么语义搜索可能无法仅凭嵌入表示来准确识别出相关文档。但是, 通过关键词搜索和检查文档元数据 (例如, 作者名单), 系统仍然能定位到这位作者发表的文章, 从而提供更符合用户需求的搜索结果。

^③ 译注: 要求 LLM 分析和理解每个块的内容, 判断哪些最符合用户的查询意图。虽然这会增加计算成本, 但能显著提高搜索结果的质量和准确性。

检索增强生成(RAG)

我们已经探索了嵌入及其在语义搜索中的应用，并讨论了向量存储。现在是时候将这些工具结合起来构建一些能“跑起来”的东西了。

检索增强生成（Retrieval Augmented Generation, RAG），也被称为**基础知识定位**（grounding），是自然语言处理所采用的一种技术，它综合了基于检索的方法和基于生成的方法的优势。使用这种技术，系统首先执行检索步骤，从大型数据集中找到相关信息或上下文。然后，检索到的信息被用作输入或上下文来指导随后的生成步骤。在这个步骤中，语言模型基于检索到的上下文生成响应。

通过融合检索增强技术，我们的模型得以超越自身训练数据的局限，接入并利用外部知识库与实时信息。这一策略不仅极大地拓宽了模型的认知边界，还赋予了我们更为精准的信息管理能力。具体而言，检索流程往往依托于成熟且标准化的软件组件实现（一般是通过“向量存储”来执行），所以可以获得对认证和权限的完整控制，对最终向用户呈现的结果进行精细的筛选。

完整流程

本节汇总了需要用到的所有组件，展示了如何通过以下步骤来构建使用了嵌入和向量存储的一个标准 RAG 解决方案（参见图 3-2）。

1. 预处理：

- A. 数据被分割成块（chunks）。
- B. 使用给定的模型计算嵌入。
- C. 将块的嵌入存储到某个向量数据库中。

2. 运行时：

- A. 需提供某种用户输入，可以是特定的查询、触发器（触发条件）、消息或其他。
- B. 为用户的输入计算嵌入。
- C. 查询向量数据库，以返回与用户输入相似的 N 个块。
- D. 基于检索到的上下文，将 N 个块作为消息或单一“提示”的一部分发送给 LLM，以便为用户查询提供一个最终答案。
- E. 为用户生成答案。

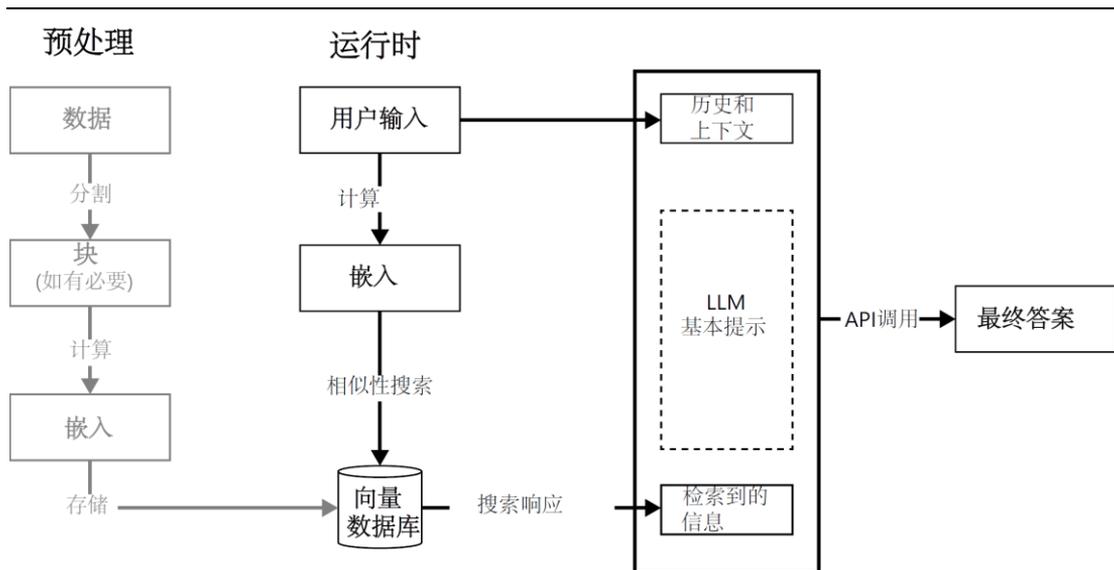


图 3-2 使用嵌入和向量存储的一个 RAG 解决方案的总体架构

整个流程可以嵌入到一个示例控制台应用中，先提供一个系统提示，然后是常规的消息流，并为用户输入和检索的文档添加一条用户消息。

你是一名提供旅行信息的智能助手，帮助计划前往欧洲的旅行。使用“你”来指代提问的个人，即使他们使用“我”来提问。根据下面的对话，你只使用所提供的数据源回答用户问题。回答后续的问题时，你要基于完整的对话场景。

对于表格信息，以 HTML 表格形式返回。不要返回 Markdown 格式。每个数据源都有一个名称，后跟一个冒号和实际信息。每次使用数据时，都要包括数据源名称。如果无法使用下面的数据源回答，就说你不知道。

按照以下格式进行：

#####

问题：“美国公民前往欧洲的签证要求是什么？”

数据源：

info1.txt: 美国公民可以免签证前往欧洲，最多可停留 90 天，适用于旅游或商务目的。

info2.pdf: 每个欧洲国家的具体签证要求可能有所不同。有些国家可能需要长期停留或特定目的的签证。

info3.pdf: 欧洲是由多个具有各自入境要求的国家组成的大陆。

info4.pdf: 申根区包括几个欧洲国家，实行共同的签证政策，允许美国公民在该区域内免签旅行。

你的回答：

美国公民可以免签证前往欧洲，最多可停留 90 天，适用于旅游或商务目的[info1.txt]。每个欧洲国家的具体签证要求可能有所不同，有些国家可能需要长期停留或特定目的的签证[info2.pdf]。申根区包括几个欧洲国家，实行共同的签证政策，允许美国公民在该区域内免签旅行[info4.pdf]。

下面是示例 C# 代码：

```
var chatCompletionsOptions = new ChatCompletionsOptions
{
```

```

DeploymentName = AOAI_DEPLOYMENTID, // 这个常量已经在之前的例子中定义了
Messages = { new ChatRequestSystemMessage(systemPrompt) }
};
while (true)
{
    var userMessage = Console.ReadLine();
    Console.WriteLine("BOT: ");
    var chatCompletionsResponse = await openAIClient.GetChatCompletionsAsync(options);

    // 这个函数应该嵌入并查询向量存储
    var retrievalResponse = GetRelevantDocument(userMessage);
    var getAnswerMessage = $"问题: {userMessage} \n 数据源: {retrievalResponse}";
    chatCompletionsOptions.Messages.Add(new ChatRequestUserMessage(getAnswerMessage));
    var responseForUser = openAIClient.GetChatCompletionsAsync(chatCompletionsOptions)
        .Result.Value.Choices.FirstOrDefault().Message.Content;

    Console.WriteLine(responseForUser);
    chatCompletionsOptions.Messages.Add(new
        ChatRequestAssistantMessage(responseForUser));
    Console.WriteLine("请输入: ");
}

```



注意：向 LLM 提供所有相关文档，这称为填充式方法（stuff approach）^①。

在生产解决方案中，应尽量避免在用户消息中注入数据源。然而，本书后面会提供一个完整的、可以实际工作的示例，它展示了完整和正确的消息流程。同样的流程也可以在 Azure 门户（<https://portal.azure.com/>）上实现。具体做法是在门户上搜索“机器学习”，选择“Azure 机器学习”，单击“创建”来创建一个机器学习工作室（工作区）。工作室创建好之后，请启动它。然后，创建一个“提示流”并选择正确的模板。

改进版

当基本版本不足以满足需求时，可以在同一框架内探索其他选项。一个方案是请求 LLM 帮你重新措辞，从而改进在向量存储上启动的搜索查询。另一个方案是改进文档的顺序（记住，在 LLM 中，顺序很重要），并在需要时对其进行摘要和/或改写（重新措辞）。

完整过程如下所示（参见图 3-3）：

1. 预处理：

^① 译注：我更愿意称之为“填鸭式方法”。

-
- A. 数据被分割成块。
 - B. 使用给定模型计算嵌入。
 - C. 将块的嵌入存储到某个向量数据库中。
2. 运行时：
- A. 需提供某种用户输入，可以是特定的查询、触发器（触发条件）、消息或其他。
 - B. 用户输入被注入到一个改写（重新措辞）的系统提示中，以添加执行更好的搜索所需要的上下文。
 - C. 为改写的用户输入计算嵌入，并用作数据库查询参数。
 - D. 查询向量数据库，以返回与查询相似的 N 个块。
 - E. 按不同的排序标准对 N 个块进行重新排名（打分）^①。如有必要，将其传递给 LLM 以进行摘要。这样可以节省 token 并提高相关性。
 - F. 基于检索到的上下文，将重新排名的 N 个块作为消息或单一“提示”的一部分发送给 LLM，以便为用户查询提供一个最终答案。
 - G. 为用户生成答案。

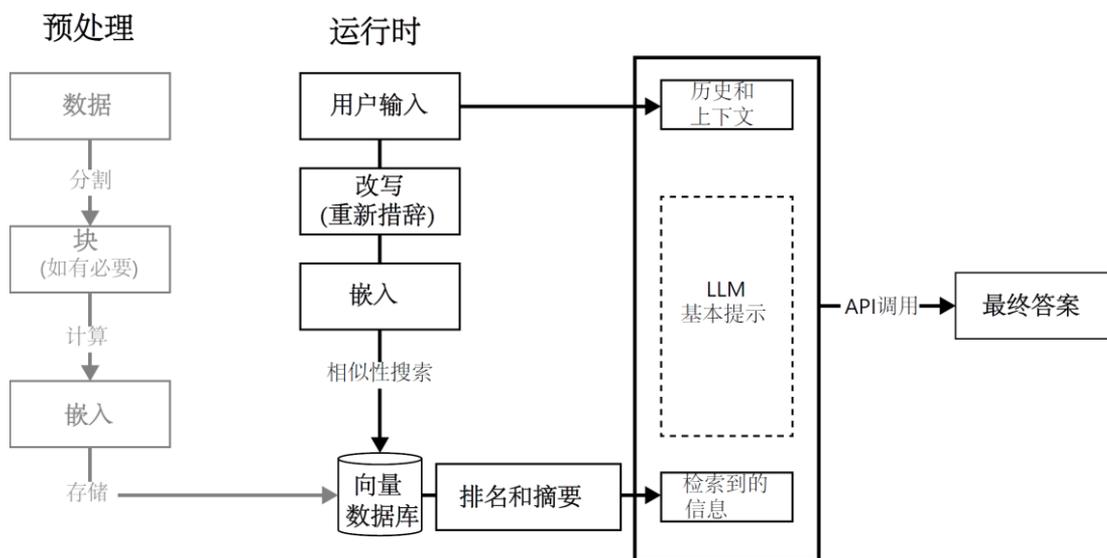


图 3-3 改进的 RAG 流程

^① 译注：重新排名的标准可以包括但不限于向量相似度、文档新鲜度以及来源可靠性等。

改写（重新措辞）步骤可以在接收到用户输入后，马上通过一个系统提示来执行。例如：

基于聊天历史 and 用户问题生成一个搜索查询，该查询将从知识库中返回最佳答案。
尝试为搜索查询生成一个语法正确的句子。
不要使用引号，并避免使用其他搜索操作符（例如+）。
不要在搜索查询词中包含所引用的数据源的文件名和文档名，例如 `info.txt` 或 `doc.pdf`。
不要在搜索查询词中包含任何在 `[]` 或 `<<>>` 内的文本。
如果问题不是用中文提出的，那么在生成搜索查询前将问题翻译成中文。
搜索查询: {userInput}

通过应用最大边际相关性（MMR）算法，并向不包含任何对话历史的一个不同的 LLM 实例添加摘要提示，我们可以获得检索结果的一个不同的排名和摘要。在某些情况下，这有助于减少偏见或既往交互的影响，使得生成的内容更加中立和客观。

还可以探索其他多种不同的方法。LangChain 已经实现了其中最著名的几种方法，包括 Refining 和 MapReduce 链。Refining 是一次向 LLM 传递一个文档，迭代地改进答案，直至获得最终答案。MapReduce 最初为文档分别调用 LLM，将模型的输出视为新文档（映射步骤）。然后，新生成的所有文档被传递到另一个 LLM，后者在化简（reduce）步骤中合并文档以获得单一的最终输出。这两种技术都需要更多的 LLM 调用，因此会造成更多的成本和延迟。但是，它们可以得到更好的结果。我们应视当前的具体情况进行权衡。

问题和对策

尽管提供了领域或公司特有的数据和文档（例如，手册、发布说明或产品数据表），但完整系统可能还需要更具体（和结构化）的数据（例如，员工数据或发票）。到目前为止，我们探讨的只是 RAG 的检索并阅读（retrieve-then-read）模式，但还有其他一些模式可以满足对特定数据的需求。

其中一种模式是阅读-检索-阅读（read-retrieve-read）模式。在这种模式中，我们向模型提供一个问题和一个工具选择列表，例如：搜索向量存储索引、查找员工数据或者可能需要的任何函数调用（请参见本章前面的“函数调用”小节）。

另一种模式是阅读-分解-询问（read-decompose-ask），它遵循一种思维链风格（更具体地说，是下一章讨论的 ReAct 方法），将问题分解为单独的步骤，并回答中间的子问题，并最终得出完整的响应。

另一个要点在于对于跟进（follow-up）问题的处理。在这种情况下，对话的强大记忆具有很大的价值，而且有必要精心设计一个包含特定领域信息的系统提示。较低的温度设置也会影响模型的响应生成。

更一般地说，我们必须权衡速度、成本和质量。虽然质量是一种主观意识，但对于创造积极的用户体验至关重要，而速度则是交互式应用的关键。平衡 LLM 调用与传统方法的成本对成本优化至关重要。选择合适的模型（通常是在 GPT-4/4-turbo 和 GPT-3.5-turbo 之间选择，但也涉及选择正确的嵌入模型）对于处理速度和成本至关重要。最后，取决于任务的复杂性，预处理和运行时的一些措施（例如，优化代码和管理资源）可以改进成本和速度。

小结

本章探讨了如何使用一些更高级的技术和方法来增强 LLM 的能力。

我们讨论了如何超越简单的提示工程，通过合并不同的部分来创造强大的行动链。还讨论了在为特定任务准备和训练模型期间，微调（fine-tuning）如何发挥关键的作用。

本章还讨论了函数调用，其中包括自定义和 OpenAI 风格的。它们的目的是对过程进行优化以获得更好的结果。我们提到了函数调用的安全问题。

另外，我们还讨论了如何通过嵌入与外部数据进行交互，以进行语义搜索和检索。还讨论了向量存储的概念。还探索了与嵌入和向量检索相关的各种使用场景以及潜在的问题。

最后，本章探讨了检索增强生成（RAG）模式，全方位地介绍了这种方法。然后，我们探讨了该过程的一个改进版本，并解释了各种潜在的问题及其对策，最终的目的都是确保获得最佳的性能。

下一章将关注各种外部框架，包括 LangChain、Semantic Kernel 和 Guidance 等。

第 4 章 使用语言框架

上一章虽然探讨了许多技术，但它们一般都不需要你从头开始实现。相反，可以通过一些专用的框架来直接使用。其中最常用的框架包括 LangChain 和 Haystack。同时，Microsoft Semantic Kernel (SK) 和 Microsoft Guidance 也正在逐渐普及。此外，LlamaIndex (或 GPTIndex) 主要供检索管道 (retrieval pipelines) 摄取和查询数据。还有如 Microsoft Azure Machine Learning Prompt Flow 这样的低代码开发平台，专门用于简化 LLM/AI 应用的原型设计、实验、迭代和部署流程。

本章涵盖了 LangChain、Semantic Kernel (SK) 和 Guidance 的理论与实践，并特别强调 LangChain 在这三者中是最稳定的。本书以后会提供现实世界的例子，演示这些框架的具体使用。



注意：本章侧重于文本交互，因为在撰写本书的时候，所讨论的库尚未完全支持最新模型（例如，GPT4-Visio）的多模态能力。我预计这种情况会迅速改变，会添加接口以适应新的能力。^①ChatMessages 的概念极有可能得到扩展以包括输入文件流。不过，虽然所有库接口在过去几个月都经历了重大变化，并将持续进化，但我不认为它们的基础概念会发生重大改变。换言之，“基本法”还是一样的。

对编排器的需求

可以将专用框架视为 LLM 的一种更高级的 API，它们集成了一系列工具、组件和接口，目的是简化开发。作为提示的一种编排器 (orchestrator)，这些框架将多个行动以交互的方式链接到一起，所有这些行动都基于提示。

人们之所以需要这些框架，一个重要的原因是 LLM 的快速发展趋势。这一趋势会导致模型很快就会发生根本性的变化。在这种情况下，如果能由一种高级的框架来提供抽象，那么适应变化会容易得多。

使用专用框架的另一个优势在于，它方便为不同的任务使用不同的模型，同时避免每使用一个模型，就要学习它特有的 API 语法。例如，你可能想使用来自 Hugging Face 模型或者一些开源本地模型的嵌入，因其更便宜。但是，对于聊天本身，则希望使用 OpenAI 的模型。在使用专用框架的前提下，无论选择什么模型，编程接口都不需要修改。

^① 译注：截止 2024 年 8 月，GPT-4v 的 API 已经正常支持视觉推理。

LangChain、SK 和 Guidance 等框架的一些优势包括：

- 为检索增强生成（RAG）内置了向量存储连接器和编排逻辑
- 简化了记忆管理，允许 LLM 跟踪来自之前对话的上下文
- 半自主代理增强了功能性（框架在一定程度上自主运行，执行预定义的任务，同时还能根据用户指令调整行为，增加了系统的灵活性和响应速度）
- 能对 LLM 输出进行更严格的控制，确保获得更加精确和安全的结果
- 流畅的、简化的函数调用过程

每个框架都有其独特性。LangChain 是用于 AI 编排的一种尤其出色的开源库，它支持 Python 和 JavaScript，提供了 Semantic Kernel（它支持使用 C#和 Python）的一种替代方案，另外还提供了 LlamaIndex 等其他选项。相反，Guidance 专门用于指导 LLM 输出，优化推理过程中指导 LLM 的过程，并简化交互，以获得对最终输出的更大控制。



注意：OpenAI 已推出 Assistants API，其概念类似于代理。Assistants 可以通过提供特定指令并同时访问多个工具来自定义 OpenAI 模型，无论这些工具是由 OpenAI 托管还是由用户创建的。它们自主管理消息历史并处理各种格式的文件，在使用工具期间创建并引用这些文件。然而，Assistants 功能被设计为低代码或无代码，其灵活性较差，所以不太适合企业环境。^①

跨框架概念

尽管每个框架都有其特殊性，但它们或多或少都基于共同的一些抽象概念。提示模板、链、外部函数（工具）和代理的概念在所有框架中以不同的形式存在。类似地，记忆和日志记录的概念也是如此。

提示模板、链、技能和代理

在组织向 LLM 输入的“提示”时，提示模板（prompt template）起着至关重要的作用。它们可以被比作字符串格式化器（许多编程语言都有这个），允许数据工程师以各种方式构建提示，以实现一系列结果。例如，在问答场景中，可以对提示进行定制，使之符合标准

^① 译注：Assistants API 的主要作用就是让你在自己的应用程序内构建 AI 助手。助手拥有指令，并能利用模型、工具和知识来响应用户查询。Assistants API 目前支持三种工具：代码解释器、检索和函数调用。详情可以参考官方文档：https://www.openaidoc.com.cn/api-reference/Assistants_run

的问答（Q&A）结构，将答案呈现为项目符号列表，甚至可以包含对当前所提出的问题的概述，并提供少量示例，以便更好地回答问题。

例如，下例展示了如何在 LangChain 中实例化一个提示模板。

```
from langchain import PromptTemplate
prompt = PromptTemplate(
    input_variables=["product"],
    template="生产{product}的一家公司取个什么名字好?",
)
```

下例展示了如何使用 C# 在 Semantic Kernel (SK) 中实例化一个提示模板。

```
var promptTemplate = new PromptTemplateConfig()
{
    Name = "CompanyName",
    Description = "公司名字生成器",
    Template = @"生产{{product}}的一家公司取个什么名字好?",
    TemplateFormat = "semantic-kernel",
    InputVariables = [
        new() { Name = "product", Description = "产品", IsRequired = true }
    ]
};
```

将不同的提示模板以及不需要 LLM 介入的更简单的行动（例如，去除空格、修正格式、格式化输出等）链接在一起，技术上可以被称为“构建一个链”（building a chain）。下例展示了 LangChain 中一个链，它使用了 LangChain 表示语言（LangChain Expression Language, LCEL）：

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

prompt = ChatPromptTemplate.from_template("跟我讲一个关于{topic}的小笑话")
model = ChatOpenAI()
output_parser = StrOutputParser()

chain = prompt | model | output_parser

chain.invoke({"topic": "冰淇淋"})
```

有的时候，我们需要的可能不仅仅是一个静态和预定义的链，后者本质上是 LLM 或其他工具的一个调用序列。相反，我们可能需要依赖于用户输入的一个不确定的序列。在这种链中，一个代理（或规划器，即 planner）可以访问一系列工具。用户输入决定了代理应该调用哪个工具（如果有的话）。



注意：链使用编码中嵌入的一个预编程的行动序列来执行操作，代理则利用语言模型作为认知引擎来选择何时应该采取何种行动。

记忆

为 LLM 添加记忆时，有两个场景需要考虑：对话记忆（短期记忆）和上下文记忆（长期记忆）。

对话记忆使聊天机器人能够以对话的方式回应多个查询。这使得对话连贯。如果没有对话记忆，每个查询都会被视为完全独立的输入，而不考虑过去的交互。

短期记忆并不总是“短期”的，因为它可以永久存储在数据库中。但在某个时候，对话可能会变得太长，以至于每个新查询和响应都需要发送给 LLM。但是，LLM 的上下文窗口最多可以处理 4k、16k、32k 或 128k 个 token（取决于模型），而且它们是有成本的。因此，我们基本上有两个选择。一个选择是仅发送用户与系统之间有限窗口内的消息（比如最后 N 条消息），另一个选择是通过一个 LLM 调用或者通过更传统的信息检索系统来对整个对话内容进行摘要。

记忆系统必须具备两个基本操作：检索和记录。每条链都建立起一个基本的执行逻辑，它预期特定的输入。某些输入由用户直接提供，其他的则可能来自记忆系统。在一次运行中，一条链在以下两个时刻与其记忆系统交互：

- 在收到第一个用户输入后，但在执行核心逻辑之前，链会访问其记忆系统，对用户输入进行“增强”。
- 在执行核心逻辑之后，但在呈现答案之前，链会将当前运行的输入和输出存储到记忆中，以便在未来的运行中引用这些信息。

Semantic Kernel (SK) 目前没有一个专门的功能集来实现对话记忆，但开发人员可以使用长期记忆策略，利用 `VolatileMemoryStore`（非持久化，即存储到内存中）或支持的向量存储（持久化）。也就是说，短期（对话）记忆应被视为一系列文档的集合来处理，它们没有特定的顺序。我们基于所选择的一个记忆提供程序（`memory provider`），按照某些相似性（通常是余弦相似性）标准来进行查询。另一个办法是将整个对话存储在某个非关系数据库（通常是 MongoDB 或 CosmosDB）中，或者存储在内存中，并在每次需要时都重新加载。

但是，LangChain 就没有这些烦扰，它用特定的模块来处理不同类型的对话记忆，其中包括：

- `ConversationBufferMemory`：这个最简单，单纯将消息存储在一个变量中。
- `ConversationBufferWindowMemory` 和 `ConversationTokenBufferMemory`：它们根据消

息数量或总 token 数量，仅保留对话的最后 K 次交互。

- **ConversationEntityMemory**: 会记住关于对话中特定实体的事实。它使用 LLM 提取实体信息，并在交互过程中更新对该实体的知识。
- **ConversationSummaryMemory**: 对对话进行摘要，并记忆摘要。
- **ConversationSummaryBufferMemory**: 这种类型的记忆会保留最近的交互，并对最早的交互进行摘要，而不是完全弃之不用。^①
- **VectorStoreRetrieverMemory**: 与 SK 的方法类似，将交互作为文档存储，不显式跟踪交互顺序。

数据检索器

数据检索器（data retriever）是一种接口，根据一个非结构化的查询来提供文档。它的功能比向量存储更广泛。和向量存储不同，数据检索器不一定需要具备存储文档的能力；其主要功能是检索和返回文档。向量存储可以作为数据检索器的基础组件，但数据检索器也可以基于易失性内存或传统的信息检索系统构建。

LangChain 和 SK 支持多种数据提供程序，具体可参考以下链接：

- <https://python.langchain.com/docs/integrations/retrievers/>
- <https://github.com/microsoft/semantic-kernel/tree/main/dotnet/src/Connectors>

以下示例代码展示了如何向 `VolatileMemoryStore` 添加一些简单的文档并使用 SK 进行查询。

```
var kernel = Kernel.CreateBuilder()
    .AddAzureOpenAIChatCompletion(deploymentName: AOAI_DEPLOYMENTID, endpoint:
        AOAI_ENDPOINT, apiKey: AOAI_KEY)
    .AddAzureOpenAITextEmbeddingGeneration(deploymentName: AOAI_EMBEDDING, endpoint:
        AOAI_ENDPOINT, apiKey: AOAI_KEY)
    .Build();

// 创建一个用于语义记忆的嵌入生成器
var embeddingGenerator = new
    OpenAITextEmbeddingGenerationService(TestConfiguration.OpenAI.EmbeddingModelId,
        TestConfiguration.OpenAI.ApiKey);
```

^① 假设有一个对话系统，它使用 `ConversationSummaryBufferMemory` 来管理对话历史。当用户和系统进行一系列对话时，系统会记录这些交互。当对话历史变得很长时，系统会对最早的几次交互进行摘要，提取其中的关键点和主要信息，而将详细内容删除。新的交互会被完整记录，直到它们也变得过时，需要被摘要（汇总）。

```

SemanticTextMemory textMemory = new(memoryStore, embeddingGenerator);
await textMemory.SaveInformationAsync(MemoryCollectionName, id: "info1",
    text: "我的名字是令狐冲");

// 查询
await foreach (var answer in textMemory.SearchAsync(
    collection: MemoryCollectionName,
    query: "我的名字是什么?",
    limit: 2,
    minRelevanceScore: 0.75,
    withEmbeddings: true))
{
    Console.WriteLine($"Answer: {answer.Metadata.Text}");
}

```

注意，在使用 SK 时，文档在存储之前总是会进行嵌入（而在 LangChain 中并不总是这样），但查找也可以基于一个特定的键（key）进行。

日志记录和跟踪

在现实场景中，用户与系统之间的每一次交互都会进行多次 LLM 调用，并且每次调用提供的都是已经连接（拼接）好的、格式良好的提示。因此，我们很难跟踪确切的运行链以分析提示和在 token 上的消耗。

只有 OpenAI 模型支持 token 消耗跟踪。如果使用 LangChain，这可以通过以下方式实现：

```

from langchain.agents import load_tools, initialize_agent, AgentType
from langchain.llms import OpenAI

llm = OpenAI(temperature=0)
tools = load_tools(["serpapi", "llm-math"], llm = llm)
agent = initialize_agent(
    tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose = True)
with get_openai_callback() as cb:
    response = agent.run("Olivia Wilde 的男友是谁？他现在的年龄的 0.23 次方是多少？")
    print(f"总 token 数: {cb.total_tokens}")
    print(f"提示 token 数: {cb.prompt_tokens}")
    print(f"补全 token 数: {cb.completion_tokens}")
    print(f"总费用(美元): ${cb.total_cost}")

```

代码中的关键是 `verbose=True` 和回调函数，我们通过它们来了解 token 的使用指标。

其中，`verbose` 选项启用了每个中间步骤的日志记录。可以在代理或链模块上开启该选项，如下所示：

```

conversation = ConversationChain(
    llm = chat,
    memory = ConversationBufferMemory(),

```

```
        verbose = True
    )
    conversation.run("什么是 ChatGPT?")
```

在开启了日志记录后，SK 可以生成大致相同的信息。

```
IKernelBuilder builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(***CONFIG HERE***);
builder.Services.AddLogging(c => c.AddConsole().SetMinimumLevel(LogLevel.Information));
Kernel kernel = builder.Build();
```

此外，在 Visual Studio Code 上安装 SK Extension，可以在没有任何代码的情况下测试函数，并检查这些函数的 token 使用情况。

规划器（即 planner，相当于 LangChain 的“代理”或“智能体”，本章稍后会讨论）也可以进行日志记录和监控。

SK 通过日志记录、计量和跟踪来实现遥测，并使用原生的 .NET 仪器工具在代码中插入测量点（这称为代码的“仪器化”，想象在代码中安插了一些仪器来监测各种指标）。这样就可以灵活地使用各种监控平台，例如 Application Insights、Prometheus 和 Grafana 等。以下代码展示了如何启用跟踪。

```
using System.Diagnostics;
var activityListener = new ActivityListener();
activityListener.ShouldListenTo = activitySource =>
    activitySource.Name.StartsWith("Microsoft.SemanticKernel",
    StringComparison.Ordinal);
ActivitySource.AddActivityListener(activityListener);
```

以下代码展示了如何使用像 Application Insights 这样的遥测客户端实例来进行计量。

```
using System.Diagnostics.Metrics;
var meterListener = new MeterListener();
meterListener.InstrumentPublished = (instrument, listener) =>
{
    if (instrument.Meter.Name.StartsWith("Microsoft.SemanticKernel",
        StringComparison.Ordinal))
    {
        listener.EnableMeasurementEvents(instrument);
    }
};
meterListener.SetMeasurementEventCallback<double>((instrument, measurement,
    tags, state) =>
{
    telemetryClient.GetMetric(instrument.Name).TrackValue(measurement);
});
meterListener.Start();
```

在这两种情况下，我们都可以选择特定的度量或活动，并对命名空间字符串应用更严格的

条件。

使用 LangChain，不仅可以为代理启用全面的跟踪功能，还可以采用一种更复杂的方法，即使用 Web 服务器收集代理运行数据。Web 服务器使用端口 8000 来收集和存储跟踪细节，端口 4173 则用于托管用户界面。Web 服务器在 Docker 容器内运行。因此，除了设置 LangChain，还必须设置 Docker，并执行 `docker-compose` 命令。

在正确的 Python 环境中，在终端运行以下命令来启动服务器容器：

```
-m langchain.server
```

以下代码启用指定会话的跟踪：

```
from langchain.llms import AzureOpenAI
from langchain.callbacks import tracing_enabled
with tracing_enabled('session_test') as session:
    assert session
    llm = AzureOpenAI(deployment_name=deployment_name)
    llm("Tell me a joke ")
```

然后，可以在本地的浏览器中访问 <http://localhost:4173/sessions> 来选择正确的会话，如图 4-1 所示。



图 4-1 LangChain 跟踪服务器



注意：LangSmith 是 LangChain 云托管的一个额外的 Web 平台，生产应用程序使用它也许会更可靠，但需要在 smith.langchain.com 上单独设置。更多信息请参见第 5 章。

需要考虑的重点

这些框架简化了一些知名的模式和使用场景，并提供了各种辅助工具。但和往常一样，必须根据当前项目的复杂性和具体情况考虑是否使用它们，尤其是要考虑开发环境、期望的可重用性、是否需要修改和调试每个单独的提示以及相关的成本。

不同的环境

LangChain 支持 Python 和 JavaScript；SK 支持 C#、Java 和 Python；Guidance 支持 Python。大体而言，Python 生态为自然语言处理（NLP）提供了更为丰富的工具集，然而，.NET（以及 Java）在企业级功能上展现出更强的实力，这一点在某些场景下同样至关重要。

对于技术的选择绝非易事，正如决定是采用单一技术栈还是通过 API 集成多种技术栈一样充满挑战。如果使用场景变得逐渐复杂，牵涉的不仅是用户与 LLM，还有数据库、缓存、登录功能、传统 UI 等众多元素，那么一种实用的做法是在共通的架构范畴内，将 LLM 层提炼为独立的组件。这些组件甚至可以是经由 API 相互通信的独立 Web 应用，这样的设计赋予了隔离性优势，同时便于分别监控成本与性能。当然，这样做也会引入额外的延迟，并需投入一定的抽象工作来构建 API 层，以确保各组件间的顺畅交互。



提示：在当前这个充满变数的大环境下，选择技术栈或工具时，某个特定功能（可能仅限于 Python 这样的技术栈）的存在与否可能成为决策的关键因素。

成本、延迟和缓存

在决定是否以及使用哪些框架时，一个需要考虑的方面是成本——既包括底层模型（特别是 OpenAI 的付费模型）的调用成本，也包括延迟成本。例如，代理和规划器会消耗大量 token。因此，通过部分提示和响应来回交互会产生显著的费用，例如稍后在 ReAct 模式中会看到的那样。

某些功能（例如之前讨论的 ConversationSummaryMemory）非常有用而且强大，但需要多次调用 LLM。当用户数量增加的时候，这可能会变得非常昂贵。

在 RAG（检索增强生成）上下文中，另一个例子是重新措辞（改写）用户问题以优化它们，从而获得更相关的文档片段，并使 LLM 能够提供更好的响应。然而，一个更具成本效益的解决方案是在生成嵌入数据库时应用的，这涉及让 LLM 提前重新措辞文档片段。具体来说，要求 LLM 假设用户可能询问的一些问题，并进行重新措辞（可以提供 few-shot 示例）。

减少总体延迟的一种方法是缓存 LLM 结果。LangChain 通过其 llm_cache（通过 SQLite）

来提供对这个功能的原生支持。也可以通过 `GPTCache` 库使用缓存，而在 `SK` 中则必须手动重新实现。`Guidance` 则有其自己的优化流程（称为加速），但仅适用于本地模型。

可重用性和糟糕的调试

我们面临的一个关键挑战在于提示和已编排的行動的可重用性。尽管创建可重用模板很有吸引力，但大多数时候我们只使用 `GPT-4` 和 `GPT-3.5 turbo`。一个显著的例外是嵌入阶段。在这个阶段，某些开源模型可能表现良好，有的时候甚至可以通过微调表现得更好。

链中的每个功能和步骤都需要自定义的提示和精心的调试以生成所需的输出。这就意味着，尽管我们追求组件的无缝重用，但在实践中，这一目标依然难以轻易达成。这种现状在一定程度上限制了 `LangChain` 和 `Semantic Kernel (SK)` 等框架进行抽象化的努力，因为这些框架的核心价值之一就是希望通过高度抽象化的组件来简化开发流程，但现实中的定制化需求却常常与之相悖。此外，这些框架有时会导致工具锁定（`tool lock-in`），即过度依赖某一套工具或平台。这样做的收益很小，但却增加了开发的复杂性和限制性。

`LangSmith`、`PromptFlow` 或 `HumanLoop` 等外部平台不仅方便我们进行实验，还有利于在生产解决方案中全面监控和调试所有重要步骤。事实上，调试和定制是另一个值得注意的问题。虽然这些框架提供了一种结构化的编排方法，但即便有详细的日志记录，对链中的错误进行调试也可能是非常困难的。此外，工作流程有文档记录的范围就是这么小。一旦超出这个范围，往往会导致复杂的挑战，需要深入研究框架的代码库。

这些框架旨在简化交互，并且它们在标准（但仍然多样化）的使用场景中表现出色。例如，使用 `LangChain` 或 `SK` 可以在几分钟内执行一个标准的 `RAG` 应用，而从头开始编写则需要几天的时间。不过，当涉及更复杂的场景时（例如，测试），兼容性和适应性仍然是需要探索和改进的领域。

LangChain

`LangChain` 是一个多功能框架，它使应用程序能够与各种数据源及其环境进行交互。该框架提供了两个核心功能：

- **组件：**`LangChain` 为与语言模型的交互提供了模块化的抽象，为每种抽象都提供了多种实现选项。这些组件不仅高度灵活，而且易于集成。它们既可以作为 `LangChain` 框架的一部分使用，也可以独立使用。
- **现成的链：**`LangChain` 内置了一系列预先设计好的组件链，能够直接执行特定的高级任务。

这些现成的链简化了初始开发阶段的工作。对于更加复杂的应用场景，基于组件的构建方式允许开发者对现有的链进行定制化改造，或是从零开始构建全新的链路，以满足特定需

求。

该框架支持多种模块：

- 模型 I/O：这是与不同语言模型的基础接口。
- 数据连接：这是与特定应用程序数据和长期记忆交互的接口。
- 链：这些处理 LLM 调用链。
- 代理：这些是动态链，可以在给定高级指令的前提下，基于一个推理 LLM 来选择所用的工具和 API。
- 记忆：在链的运行之间保留短期记忆。
- 回调：记录与 LLM 交互的中间步骤。

至于是否能在生产和企业环境中使用 LangChain，则要考虑到多方面的因素。一方面，除了使用基础的 OpenAI API 或 Hugging Face 来实现所需的模型功能之外，并没有其他太多的替代方案。另一方面，LangChain 在框架架构方面仍显得不够成熟。虽然可以用多种方法来实现相同的功能，但这些方法并不统一，缺乏标准化。除此之外，相关的文档资料也比较有限（至少在撰写本书时如此）。

模型、提示模板和链

如果想在自己的应用中使用 LLM，那么提示（prompt）是最为关键的一个方面。提供一系列提示模板几乎已成为现在的标准做法中，每个模板都有其特定的用途，例如分类、生成、问答、摘要和翻译等。LangChain 整合了所有这些标准提示，并提供了一个用户友好的界面，方便我们创建和定制新的提示模板。

所选的模型也很重要，对这些模型的调用链必须正确管理。LangChain 帮助我们将所有这些组件无缝整合到统一的应用程序流程中。例如，可以这样设计一个链：接收用户输入，应用格式化的提示模板，通过 LLM 进一步处理格式化的响应，通过解析器传递输出，最终将结果返回给用户。

模型

LangChain 的初衷是为了提供一种抽象层，使得开发者可以更容易地与各种大语言模型（LLM）进行交互，而无需深入了解每个模型的特定 API 细节。这样，开发者可以使用一致的接口来调用不同的模型，简化了模型选择和切换的过程。LangChain 支持许多模型，包括 Anthropic 的 Claude2/Claude3 模型、OpenAI 和 Azure OpenAI 的模型（本书的示例使用的就是这些模型）、通过 LlamaAPI 来访问的 Llama2/Llama2 模型、Hugging Face 的各种模型（包括本地版本和托管在 Hugging Face Hub 上的版本）、Vertex AI 的各种 PaLM 模型以及 Azure Machine Learning 的各种模型等。



注意： Azure Machine Learning^①是一种 Azure 平台，用于构建、训练和部署机器学习模型。这些模型可以从 Azure Model Catalog 中选择，其中包括 OpenAI Foundation Models（如有必要，可以进行微调）和 Azure Foundation Models（例如，Hugging Face 的模型以及像 Llama2/3 这样的开源模型）。

我们必须搞清楚文本补全和聊天补全 API 调用之间的区别。尽管聊天模型与普通的文本补全模型存在微妙的联系，但两者各自具有独特的特征，这显著影响了它们在 LangChain 中的使用方式。在 LangChain 中，所谓的大语言模型（LLM）主要指的是纯粹的文本补全（text-completion）模型。这类模型通过 API 接口接收字符串形式的提示作为输入，然后生成相应的字符串补全作为输出。OpenAI 的 GPT-3 便是一个典型的例子。相比之下，诸如 GPT-4 及 Anthropic 的 Claude 等聊天模型，则是专为处理对话场景而设计的。它们的 API 架构迥异，能够接收一系列带有发言者标签（例如，System、AI 或 Human）的聊天消息，最终输出一条回应的聊天消息，从而实现更为自然流畅的人机交流。

LangChain 通过提供 Base Language Model（基础语言模型）接口来实现这些模型之间的互换。该接口同时提供了针对 LLM 的 `predict` 方法以及针对聊天模型的 `predict_messages` 方法，实现了对两类模型的统一调用。此外，它还内置了一个转换器，这个转换器可以将文本补全调用转换为聊天消息调用，具体是将文本补全的内容追加到人类发言的消息中；反之亦然。转换器不仅可以处理单一消息的转换，还能将多条聊天消息整合成一个文本补全请求。这意味着无论用户输入了多少条消息，系统都能将其整合为一个完整的文本请求进行处理，从而确保在不同类型的模型之间进行平滑的转换。

之所以引入聊天模型（chat models），是因为我们需要结构化的用户输入，这可以增强模型遵循预定义目标的能力——这对于构建更安全的应用程序是至关重要的。根据个人经验，聊天模型几乎在所有情况下都表现得更好。



注意： 使用 Azure OpenAI 时，建议配置环境变量以便传递端点和 API 密钥，而不是每次都在代码中硬编码它们。为此，需要配置以下环境变量：`OPENAI_API_TYPE`（设置为 `azure`）、`AZURE_OPENAI_ENDPOINT` 和 `AZURE_OPENAI_KEY`。另一方面，如果直接与 OpenAI 模型交互（不建议，因为 OpenAI 从 2024 年 7 月 9 日开始，对可以直接接入 Open AI 的 API 服务的国家和地区进行了限制），那么应该配置环境变量

^① 译注：前身是 Azure Machine Learning Studio，又称 ML Studio (classic)，已于 2024 年 8 月 31 日退役。微软鼓励老用户尽快迁移至 Azure Machine Learning。

OPENAI_ENDPOINT 和 OPENAI_KEY。

提示模板

LLM 应用程序不会直接将用户输入传递给模型。相反，它们使用更全面的文本片断，即提示模板（prompt template）。

首先是基本的补全提示，代码如下：

```
from langchain.prompts import PromptTemplate
prompt = PromptTemplate(
    input_variables=["product"],
    template="生产{product}的一家公司取个什么名字好?"
)
print(prompt.format(product="医疗数据分析"))
```

前面讲过，LangChain 旨在简化传统和基于聊天的两种文本生成方法之间的转换。但是，在现实世界的场景中，聊天提示（chat prompt）可能更为适用，它可以通过以下方式实例化：

```
from langchain.prompts import ChatPromptTemplate, SystemMessagePromptTemplate,
HumanMessagePromptTemplate
human_message_prompt = HumanMessagePromptTemplate(
    prompt=PromptTemplate(
        template="生产{product}的一家{company}取个什么名字好?",
        input_variables=["company", "product"]
    )
)
chat_prompt_template = ChatPromptTemplate.from_messages([human_message_prompt])
print(chat_prompt_template.format_prompt(company="AI Startup",
product="医疗数据分析"))
```

在某个时候，你可能需要使用 few-shot（少样本）提示技术。有三种方法可以格式化这样的提示：显式编写，基于一个示例集进行格式化，或者让框架从一个 ExampleSelector 实例中选择相关的示例。

为了动态选择示例，必须使用一个现有的 ExampleSelector，或者通过实现 BaseExampleSelector 接口来新建一个。现有的选择器包括下面这些：

- **SemanticSimilarityExampleSelector**：根据示例和输入的“嵌入”来找到最相似的示例。因此，需要一个向量存储和一个 EmbeddingModel（这意味着需要额外的基础设施和成本）。
- **MaxMarginalRelevanceExampleSelector**：其工作方式类似于 SemanticSimilarityExampleSelector（因此需要向量存储和嵌入），但它特别重视所选示例之间的多样性（换言之，会尽量避免选择内容过于相似的示例）。

-
- **NGramOverlapExampleSelector**: 根据一种不同（且效率不高）的相似性度量来选择示例，不需要任何嵌入。这种度量称为 **n-gram overlap score**。
 - **LengthBasedExampleSelector**: 根据设定的最大长度（`max-length`）参数来动态调整所选示例的数量，确保最终的示例集总长度不超过这个预设的最大值。

在配置好选择器之后，就可以使用以下代码构建 `few-shot` 提示：

```
few_shot_prompt = FewShotChatMessagePromptTemplate(  
    input_variables = ["input"],  
    example_selector = example_selector,  
    # 每个示例将变成两条消息：一条人类消息和一条 AI 消息  
    example_prompt = ChatPromptTemplate.from_messages(  
        [{"human", "{input}"}, {"ai", "{output}"}]  
    ),  
)
```



注意：`ExampleSelector` 根据输入选择示例，因此在这种情况下必须定义一个输入变量。

链

我们通过链（`chain`）来合并多个组件以创建单一的、连贯的应用程序。

主要可以采取两种方式来链接不同的调用。传统（旧版）方法是使用 `Chain` 接口，而最新的方法是使用 `LangChain` 表示语言（`LangChain Expression Language, LCEL`）。

`LangChain` 通过 `Chain` 接口提供了几种基础链：

- **LLMChain**: 包含一个提示模板和一个语言模型，后者可以是 `LLM` 或聊天模型。具体过程涉及用提供的输入键值（如果可访问的话，还包括记忆）来打造提示模板，将修改后的字符串转发给选定的模型，并获取模型的输出。
- **RouterChain**: 创建一个链，基于给定的输入来动态选择下一个链。它由 `RouterChain` 本身和各个目标链组成。
- **SequentialChain**: 连接多个链。有两种类型的顺序链：
 - **SimpleSequentialChain**: 这种类型的顺序链每一步都有一个单一的输入/输出（一个步骤的输出是下一个步骤的输入）。
 - **TransformChain**: 预处理输入，例如删除多余的空格，只获取前 `N` 个字符，替换一些词/字，或者你可能想对输入应用的其他任何转换。注意，这种类型的链通常只需要代码，而不需要 `LLM`。



注意：还可以通过子类化基础链类来构建自定义链。

人们已基于这些基础链打造了一些经过充分测试的通用链，详情请访问以下网址：

<https://github.com/langchain-ai/langchain/tree/master/libs/langchain/langchain/chains>

其中，最受欢迎的有 `ConversationChain`、`AnalyzeDocumentChain`、`RetrievalQAChain` 和 `SummarizeChain`，但用于 QA 生成（基于给定的文档来构建问答）和数学的链也很常见。有一个名为 `PALChain` 的数学链使用了 Python REPL（读取-求值-打印循环，`Read-Eval-Print Loop`）来编译和执行从模型生成的代码。

首先来看一个基础链代码的示例，它使用了旧版 `Chain` 接口和一个聊天模型：

```
human_message_prompt = HumanMessagePromptTemplate(
    prompt=PromptTemplate(
        template="生产{product}的一家{company}取个什么名字好?",
        input_variables=["company", "product"],
    )
)
chat_prompt_template = ChatPromptTemplate.from_messages([human_message_prompt])
chat = ChatOpenAI(temperature=0.9) # 由于这是一个创意任务，所以温度要高一些
chain = LLMChain(llm=chat, prompt=chat_prompt_template)
print(chain.run(
    {
        'company': "AI Startup", 'product': "医疗机器人助理"
    }
))
```

以下示例则很好地解释了 LCEL 背后的逻辑：

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

prompt = ChatPromptTemplate.from_template("跟我讲一个关于{topic}的笑话")
model = ChatOpenAI()
output_parser = StrOutputParser()
chain = prompt | model | output_parser
chain.invoke({"topic": "冰淇淋"})
```



注意：`StrOutputParser` 直接将链的输出（一个 `BaseMessage` 类型的对象，

ChatModel 所产生的输出都是这种类型) 转换为字符串。如果不用 StrOutputParser, 就需要额外的步骤从 BaseMessage 中提取文本

我们当然可以设计出更复杂的链, 在链中包含多个输入并产生多个不同的输出, 并具有内部并行步骤。这些与之前的接受单个字符串作为输入并返回单个字符串作为输出的简单模式不同。在这种情况下, 输入和输出变量名称的命名变得至关重要, 如以下代码所示:

```
from langchain.prompts import PromptTemplate
from langchain.schema import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
from langchain.chat_models import AzureChatOpenAI

productNamePrompt = PromptTemplate(
    input_variables=["product"],
    template="What is a good name for a company that makes {product}?",
)
productDescriptionPrompt = PromptTemplate(
    input_variables=["productName"],
    template="对于名为{productName}的{product}, 一个好的产品描述是什么?",
)

runnable = (
    {"productName": productNamePrompt | llm |
     StrOutputParser(), "product": RunnablePassthrough()}
    | productDescriptionPrompt
    | AzureChatOpenAI(azure_deployment=deployment_name)
    | StrOutputParser()
)
runnable.invoke({"product": "航空公司的机器人"})
```

在这个示例中, 第一部分(一个与提示一起调用并返回输出的 LLM) 使用变量(“航空公司的机器人”) 生成一个产品名称。然后将其与初始输入一起传递给第二个提示, 创建产品描述。下面展示了一个可能的结果:

AirBot Solutions 是一款专为航空公司设计的创新且高效的机器人。该先进产品利用尖端技术来简化和增强航空公司的各个方面操作。通过 AirBot Solutions, 航空公司可以自动化并改善客户服务、预订、航班管理等。这款智能机器人能够处理多种任务, 包括回答客户咨询、提供实时航班更新、协助预订以及提供个性化推荐。

记忆

可以通过几种不同的方法向链中添加记忆。一种是使用 SimpleMemory 接口向链中添加特定的记忆, 如下所示:

```
conversation = ConversationChain(
```

```
llm=chat,  
verbose=True,  
memory=SimpleMemory(memories={"name": "Francesco Esposito", "location": "Rome"}),  
)
```

另一种像本章之前描述的那样通过对话记忆，如下所示：

```
conversation = ConversationChain(  
    llm=chat,  
    verbose=True,  
    memory=ConversationBufferMemory()  
)
```



注意：当然，我们可以使用之前描述的所有记忆类型，而不仅仅是 ConversationBufferMemory。

还可以在同一个链中同时使用多个记忆类，例如 ConversationSummaryMemory（使用 LLM 生成摘要）和普通的 ConversationBufferMemory。我们使用 CombinedMemory 来合并多个记忆类。

```
conv_memory = ConversationBufferMemory(  
    memory_key="chat_history_lines", input_key="input"  
)  
summary_memory = ConversationSummaryMemory(llm=OpenAI(), input_key="input")  
# 合并记忆  
memory = CombinedMemory(memories=[conv_memory, summary_memory])
```



注意：当然，必须将相应的 memory_key 注入（自定义）提示消息中。

使用 LCEL，记忆可以通过以下方式注入：

```
prompt = ChatPromptTemplate.from_messages(  
    [  
        ("system", "你是一名助理，擅长解决数学问题。"),  
        MessagesPlaceholder(variable_name="history"),  
        ("human", "{question}"),  
    ]
```

```

    ]
)
chain = (
    RunnablePassthrough.assign(
        history=RunnableLambda(memory.load_memory_variables) | itemgetter("history")
    )
    | prompt
    | llm
)

```

在这个例子中，`question` 这个输入是用户输入消息，而 `history` 这个键包含历史聊天消息。



注意：目前，记忆不会自动通过对话更新。可以通过调用 `add_user_message` 和 `add_ai_message` 或通过 `save_context` 手动更新。

解析输出

有时需要从 LLM 获得结构化输出，并且需要某种方式来强制模型生成这种输出。为此，LangChain 实现了 `OutputParser`。我们可以通过三个核心方法来构建自己的实现。

- `get_format_instructions`：该方法返回一个字符串，包含如何语言模型应该如何构造输出的指令。
- `parse`：该方法接收一个字符串（LLM 的响应），并将其处理成特定结构。
- `parse_with_prompt`（可选）：该方法接收一个字符串（语言模型的响应）和一个提示（生成该响应的输入），并将内容处理成特定结构。通过包含此提示，可以在 `OutputParser` 进行输出调整或更正时，帮助它使用与提示相关的信息进行优化。

主要的解析器包括 `StrOutputParser`、`CommaSeparatedListOutputParser`、`DatetimeOutputParser`、`EnumOutputParser` 以及最强大的 `Pydantic`（JSON）解析器。下面展示了一个简单的 `CommaSeparatedListOutputParser` 代码示例。

```

from langchain.output_parsers import CommaSeparatedListOutputParser

output_parser = CommaSeparatedListOutputParser()
format_instructions = output_parser.get_format_instructions()
prompt = PromptTemplate(
    input_variables=["company", "product"],
    template="为生产{product}的{company}生成 5 个产品名称。\\n{format_instructions}",
    partial_variables={"format_instructions": format_instructions}
)

```

```
_input = prompt.format(company="AI Startup", product="HealthCare bot")
chat = AzureOpenAI(temperature=.7, deployment_name=deployment_name)
output = chat(_input)
output_parser.parse(output)
```



注意：不是所有 ChatModel 都能使用所有解析器

回调

LangChain 具有内置的回调系统，它简化了与 LLM 应用程序不同阶段的集成，这对于日志记录、监控和串流非常有价值。要与这些事件交互，可以使用许多 API 都支持的 `callbacks` 参数。可以使用一些内置的（事件）处理程序，也可以从头实现一个新的。

CallbackHandler 接口必须实现以下方法：

- `on_llm_start`：在 LLM 开始运行时运行。
- `on_chat_model_start`：在聊天模型开始运行时运行。
- `on_llm_new_token`：在生成新的 LLM token 时运行。仅在启用了“串流”模式^①时才可用。
- `on_llm_end`：在 LLM 停止运行时运行。
- `on_llm_error`：在 LLM 遇到错误时运行。
- `on_chain_start`：在链开始运行时运行。
- `on_chain_end`：在链停止运行时运行。
- `on_chain_error`：在链遇到错误时运行。
- `on_tool_start`：在工具开始运行时运行。
- `on_tool_end`：在工具停止运行时运行。
- `on_tool_error`：在工具遇到错误时运行。
- `on_text`：在任意文本上运行。
- `on_agent_action`：在代理执行操作时运行。
- `on_agent_finish`：在代理完成时运行。

最基本的处理程序是 `StdOutCallbackHandler`，它将所有事件记录到 `stdout`，实现与 `Verbose=True` 相同的效果。

^① 译注：在与 AI 聊天时，“串流”（streaming）模式是指 AI 在生成响应时逐步返回结果的过程。这种模式允许 AI 在完成整个回答之前就开始返回部分文本，从而提供一种更加“实时”的交互体验。一个具体的例子请参见第 6 章。

```
handler = StdOutCallbackHandler()
chain = LLMChain(llm=chat, prompt=chat_prompt_template, callbacks=[handler])
chain.run({'company': "AI Startup", 'product': "healthcare bot-assistant"})
```

或者，也可以在 `run` 方法中传递回调。

```
chain = LLMChain(llm=chat, prompt=chat_prompt_template)
chain.run({'company': "AI Startup",
          'product': "healthcare bot-assistant"}, callbacks=[handler])
```



注意：稍后就会讲到，代理也公开了类似的参数。

代理

在 `LangChain` 中，代理（agent）作为一种重要的中介，可以完成 LLM API 单独无法完成的任务（因为后者无法实时访问数据）。作为 LLM 和工具（例如，`Google Search` 和天气 API）之间的桥梁，代理利用 LLM 对于自然语言的理解，根据提示来做出决策。与传统硬编码的操作序列不同，代理的行动由对 LLM 的递归调用决定，这对成本和延迟有一定的影响。

在 `LangChain` 框架中，由语言模型驱动并通过个性化提示增强的代理承担了决策制定的责任。`LangChain` 提供了多种可定制的代理类型，其中工具（tool）被设计为可调用的函数，这意味着它们可以被 Agent 直接调用来执行特定任务或获取所需信息。代理要想成功运行，必须有效地配置代理以访问特定的工具，并依赖于对这些工具的清晰描述。例如，一个代理可能被配置为访问 Web 搜索工具和文件读写工具。当接收到“查找最近的天气预报”这个任务时，代理会识别出这个任务适合使用 Web 搜索工具；而当任务变为“读取并解析本地 CSV 文件”时，代理则会选择文件读写工具来完成任务。

`LangChain` 提供了大量可定制工具，并支持创建新工具。为了更好地组织和利用这些工具，`LangChain` 引入了工具包（Toolkit）的概念。工具包本质上是对特定目标或应用场景下相关工具的集合与封装，它们像插件一样，可以方便地集成到项目中，增强功能或解决特定问题。可以访问以下网址来探索可用的工具箱：

<https://python.langchain.com/docs/integrations/toolkits/>

报告和 BI（商业智能）是 SQL 代理（SQL Agent）最常见的使用场景，它所用的工具包是 `SQLDatabaseToolkit`。但要注意，在某些关键场景下，为了确保数据安全和合规性，必须注意限制 SQL Agent 的权限和数据库用户的访问范围。

代理的类型

LangChain 支持以下几种代理，可以在文本补全或聊天补全模式下使用它们。

- **Zero-shot ReAct:** 该代理使用 ReAct 框架，基于工具的描述决定使用哪种工具。它支持多种工具，每种工具都需要对应的描述。这是目前最通用、最灵活的代理模式。
- **Structured input ReAct:** 该代理能够使用多输入（multi-input）工具。与只能指定单一字符串输入的旧代理不同，这种代理使用工具的参数架构（argument schema）创建结构化的输入格式，特别适合复杂的工具使用场景，例如在浏览器中精确导航。
- **OpenAI Functions:** 该代理专为特定的 OpenAI 模型（如 GPT-3.5-turbo 和 GPT-4）设计，这些模型经过微调以检测函数调用并提供相应的输入。
- **Conversational:** 该代理采用一种友好且对话式的提示风格，设计用于对话交互。它使用 ReAct 框架选择工具，并使用记忆来保留先前的对话交互。
- **Self ask with search:** 该代理依赖一个名为 Intermediate Answer 的工具，该工具能够搜索并提供问题的事实答案。该代理使用了类似于 Google Search API 的工具。
- **ReAct document store:** 该代理利用 ReAct 框架与文档存储交互。它需要两个特定的工具：用于文档检索的搜索工具和用于在最近检索的文档中查找术语的查找工具。这个代理的设计和运作机制让人联想到最初在 ReAct 论文中提出的概念。
- **Plan-and-execute agents:** 这些代理采用两步法（使用两个 LLM）来实现目标。首先，它们规划必要的操作。然后，它们执行这些子任务。这一概念受到了 BabyAGI 和“Plan-and-Solve”论文的启发（<https://arxiv.org/pdf/2305.04091.pdf>）。

ReAct 框架

ReAct（推理与行动，Reasoning and Acting）通过结合推理（类似于思维链，即 CoT）和行动（类似于函数调用）来增强 LLM 的性能与可解释性（interpretability），从而彻底改变了 LLM。与通常依赖强化学习来实现通用人工智能（AGI）的传统方法不同，ReAct 独辟蹊径，采用了一种独特的方法来实现跨问题的泛化（generalize）能力，使模型能在不同情境下灵活应对和学习。

ReAct 的基本理念是模拟人类执行任务的过程。人类在面对一个任务时，会规划步骤，为意外情况做调整，并且寻求外部信息。类似地，ReAct 采用了推理与行动交织的方法。它的成功之处在于，通过少样本（few-shot）提示和函数调用，它将推理（reason to act）与交互（act to reason）无缝集成到了一起。^①

^① 译注：这里稍微解释一下 reason to act（因推理而行动）和 act to reason（借行动而推理）。前者是指推理以决定接下来的行动。后者是指执行具体的行动以获得新的信息，然后根据新获取的信息做进一步的推理。

为了方便对提示进行推理，ReAct 设计了一个包含三种行动的行动空间：`search[entity]`、`lookup[string]`和 `finish[answer]`。这些行动模拟了人类与信息源交互的方式，增强了推理与行动的合成，模拟了人类的交互与决策过程。提示包含分解的思维、从维基百科（Wikipedia）获得的观察、常识、算术推理以及对搜索的重新表述，从而引导着整个行动链。

下面展示了 LLM 的不同推理方式的比较结果，摘自 Yao 等人的“在语言模型中协同推理与行动”（Synergizing Reasoning and Acting in Language Models）论文，网址是 <https://arxiv.org/pdf/2210.03629.pdf>。

问题

《七堂极简物理课》是自哪一年开始在法国工作的一位意大利物理学家写的？^①

标准方法：

答案：1986 年

仅靠推理的方法：

思考：让我们一步步思考。《七堂极简物理课》的作者是卡洛·罗韦利（Carlo Rovelli）。他自 1990 年开始在法国工作。

答案：1990 年

仅靠行动的方法：

行动 1：搜索[七堂极简物理课]

观察 1：《七堂极简物理课》是意大利物理学家卡洛·罗韦利的一本小书。最初以意大利语出版……

行动 2：搜索[卡洛·罗韦利]

观察 2：(结果 11)《七堂极简物理课》的英文书名是 Seven Brief Lessons on Physics，意大利语书名是 Sette brevi lezioni di fisica，它是意大利物理学家卡洛·罗韦利的一本小书，作者原名为 Carlo Rovelli。

行动 3：完成[1983]

ReAct 方法：

思虑 1：我需要搜索《七堂极简物理课》，找到其作者，然后找到作者自哪一年开始在法国工作。

行动 1：搜索[七堂极简物理课]

观察 1：《七堂极简物理课》是意大利物理学家卡洛·罗韦利的一本小书。最初以意大利语出版……

思考 2：《七堂极简物理课》的作者是卡洛·罗韦利，原名是 Carlo Rovelli。我需要接着搜索 Carlo Rovelli，找到他自哪一年开始在法国工作。

行动 2：搜索[Carlo Rovelli]

观察 2：Carlo Rovelli 是一位意大利理论物理学家和作家，自 2000 年起在法国工作。[1]他目前是

^① 译注：可以看出，这是一个颇为巧妙的、容易踩坑的问题。根据对 ChatGPT、Gemini、紫东太初、通义千问、Kimi 和文心一言的测试，除了 Gemini 和紫东太初，其他大模型都给出了正确的答案：**2000 年**。换用英语提问，即“Seven Brief Lessons on Physics was written by an Italian physicist that has worked in France since what year?”，结果仍然如此。因此，问题不是出在大模型对自然语言的理解上，而是出在它们自身的“逻辑”上。

Perimeter Institute 的特聘研究员……

思考3 : Carlo Rovelli 自 2000 年起在法国工作。所以答案是 2000 年。

行动3 : 完成[2000]

在 LangChain 中，基本的 ReAct 提示用于零样本代理（zero-shot agent），其格式如下：

尽你所能回答以下问题。你可以使用以下工具：

{tools}

使用以下格式：

问题：你必须回答的输入问题

思考：你应该总是考虑接下来要做什么

行动：要采取的行动，应是[{tools}]之一

行动输入：行动的输入

观察：行动的结果

...（这个思考/行动/行动输入/观察的过程可以重复 N 次）

思考：我现在知道最终答案了

最终答案：对原始输入问题的最终回答

开始！

问题：{input}

思考：{agent_scratchpad}



注意：上述代码需要使用 `agent_scratchpad`，代理要在这里添加其中间步骤（递归调用 LLM 和工具）。`agent_scratchpad` 充当了记录代理执行的每个“思考”或“行动”的存储库。这确保了在正在进行的代理执行链中，后续的“思维-行动-观察循环”能访问之前的所有思考和行动，从而保持代理行动的连续性。

用法

下面为可以访问 Google Search 和几个自定义工具的一个代理构建能实际“跑”起来的示例。首先定义工具，如下所示：

```
from langchain.tools import Tool, tool
# 为了使用 GoogleSearch, 你需要运行 -m pip install google-api-python-client
from langchain.utilities import GoogleSearchAPIWrapper
from langchain.agents import AgentType, initialize_agent
from langchain.chat_models import AzureChatOpenAI
from langchain.prompts.chat import (PromptTemplate, ChatPromptTemplate,
                                     HumanMessagePromptTemplate)
from langchain.chains import LLMChain
import os

os.environ["GOOGLE_CSE_ID"] = "###在这里填写你的 GOOGLE CSE ID###"
```

```

# 更多信息参见: https://programmablesearchengine.google.com/controlpanel/create
os.environ["GOOGLE_API_KEY"] = "###在这里填写你的 GOOGLE API KEY###"
# 更多信息参见: https://console.cloud.google.com/apis/credentials

search = GoogleSearchAPIWrapper()

# 定义工具的一种方式
@tool
def get_word_length(word: str) -> int:
    """返回字符串的字符长度。"""
    return len(word)

@tool
def get_number_words(text: str) -> int:
    """返回字符串中的单词/字数量。"""
    return len(text.split())

# 设置工具的另一种方式
def top3_results(query):
    "搜索 Google 以获取相关和最新的结果。"
    return search.results(query, 3)

get_top3_results = Tool(
    name="GoogleSearch",
    description="搜索 Google 以获取相关和最新的结果。",
    func=top3_results
)

```

还可以通过以下方式设置一个链（自定义或者 LangChain 内置的）作为工具：

```

template = "请用最多三句话总结以下文本: {input}"
human_message_prompt = HumanMessagePromptTemplate(
    prompt=PromptTemplate(template=template, input_variables=["input"])
)
chat_prompt_template = ChatPromptTemplate.from_messages([human_message_prompt])
llm = AzureChatOpenAI(temperature=0.3, deployment_name=deployment_name)

summary_chain = LLMChain(llm=llm, prompt=chat_prompt_template)

get_summary = Tool.from_function(
    func=summary_chain.run,
    name="Summary",
    description="对提供的文本进行总结。",
    return_direct=False # 如果为 True, 工具的输出会直接返回给用户
)

```

配置好工具后，就可以构建代理了，如下所示：

```

tools = [get_top3_results, get_word_length, get_number_words, get_summary]
agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.OPENAI_FUNCTIONS,
    verbose=True,

    # 如果为 False, 那么每次输出解析器无法解析 LLM/工具输出时都会引发异常
    handle_parsing_errors=True
)

```

可以使用以下代码运行代理：

```

# 尼采的第一本手稿的标题包含多少个单词?
agent.run("How many words does the title of Nietzsche's first manuscript contain?")

```

这可能会输出如下内容：

```

> Entering new AgentExecutor chain...
Invoking: `get_number_words` with `{'str': "Nietzsche's first manuscript"}`
3The title of Nietzsche's first manuscript contains 3 words.
> Finished chain.
"The title of Nietzsche's first manuscript contains 3 words."

```

如日志所示，这种代理类型（OPENAI_FUNCTIONS）可能并不是最佳的选择，因为它不会在网上主动搜索尼采第一本手稿的标题。相反，由于误解了请求（它以为问的是“Nietzsche's first manuscript”这个字符串中有多少个单词），所以它会更多地依赖于工具而不是事实推理。

可以添加一个完全自定义的提示来模仿 ReAct 框架，以不同的方式初始化代理执行器。但是，更简单的方法是尝试一种不同的代理类型，即 CHAT_ZERO_SHOT_REACT_DESCRIPTION。

有的时候，检索到的信息可能不准确，因此可能需要第二个事实核查步骤。为此，可以添加更多的工具（例如，维基百科的媒体工具^①或者维基百科的 docstore 和 REACT_DOCSTORE 代理）、使用 SELF_ASK_WITH_SEARCH 代理或者添加一个事实核查链作为工具。然后，稍微修改基本提示以添加这一额外的事实核查步骤。可以像下面这样编辑基本提示：

```

agent_chain = initialize_agent(
    tools,
    llm,
    agent=AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True,
    handle_parsing_errors=True,
    agent_kwargs={

```

^① 译注：要搜索可用的工具，请访问 <https://toolhub.wikimedia.org/>。

```
        'system_message_suffix': "开始！请记住在回答时始终使用确切的字符`最终答案`。最终答案  
        翻译成法语后再返回。"  
    }  
)
```

查看代理的源代码有助于确定在 `agent_kwargs` 中使用哪个参数。本例使用的是 `system_message_suffix`，但 `system_message_prefix` 和 `human_message`（其中至少要包含 `"{input}\n\n{agent_scratchpad}"`）这两个参数也可以编辑的。另一个可以编辑的东西是输出解析器，因为代理会调用 LLM，解析输出，将解析结果（如果有）添加到 `agent_scratchpad` 中，并一直重复直至找到最终答案。

同样的结果可以通过 LCEL 以类似的方式获得。

```
from langchain.tools.render import format_tool_to_openai_function  
llm_with_tools = llm.bind(functions=[format_tool_to_openai_function(t) for t in tools])  
from langchain.agents.format_scratchpad import format_to_openai_function_messages  
from langchain.agents.output_parsers import OpenAIFunctionsAgentOutputParser  
from langchain.agents import AgentExecutor  
  
agent = (  
    {  
        "input": lambda x: x["input"],  
        "agent_scratchpad": lambda x: format_to_openai_function_messages(  
            x["intermediate_steps"]  
        )  
    },  
    | prompt  
    | llm_with_tools  
    | OpenAIFunctionsAgentOutputParser()  
)  
  
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)  
agent_executor.invoke({"input": "单词 educa 中有多少个字母?"})
```



注意：上例定义的 `agent` 实例输出一个 `AgentAction`，因此需要一个 `AgentExecutor` 来执行代理请求的操作（并进行一些错误处理、提前停止、跟踪等）。

记忆

之前的例子使用了 `CHAT_ZERO_SHOT_REACT_DESCRIPTION`。在这个上下文中，`zero shot` 意味着没有记忆，只有单次的执行。如果我们问：“尼采的第一本书是什么？”然后问，“它是什么时候出版的？”那么代理就不能理解第二个问题，因为它在每次交互时都会丢失对话

历史。

显然，需要一个能传递记忆的对话式方法。为了解决这个问题，可以使用像 `CHAT_CONVERSATIONAL_REACT_DESCRIPTION` 这样的代理类型和一个 `LangChain Memory` 对象，如下所示：

```
from langchain.memory import ConversationBufferMemory
# return_messages=True 是使用各种 ChatModel 时的关键
memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.CHAT_CONVERSATIONAL_REACT_DESCRIPTION,
    verbose=True,
    handle_parsing_errors=True,
    memory=memory
)
```



注意：如果漏掉了 `return_messages=True`，那么代理将无法与各种 `ChatModel` 一起工作。实际上，这个选项指示内存对象存储并返回完整的 `BaseMessage` 实例，而不是简单的字符串，这正是 `ChatModel` 正常工作所需要的。

如果想探索完整的 `ReAct` 提示，那么可以使用以下代码：

```
for message in agent_executor.agent.llm_chain.prompt.messages:
    try:
        print(message.prompt.template)
    except AttributeError:
        print(f'{{{message.variable_name}}}')
```

最终的提示是：

助手是 `OpenAI` 训练的大语言模型。

助手旨在协助完成从回答简单问题到提供关于广泛主题的深入解释和讨论的各种任务。作为一种语言模型，助手能够根据它接收到的输入生成类似人类的文本，使其能够进行听起来自然的对话，并提供与当前主题相关且连贯的回应。

助手不断学习和改进，其能力不断发展。它能够处理和理解大量文本，并利用这些知识提供对广泛问题的准确和有益回应。此外，助手能够基于收到的输入生成自己的文本，使其能够参与讨论并就广泛主题提供解释和描述。

总的来说，助手是一个强大的系统，可以帮助完成广泛的任务并提供有关广泛主题的价值见解和信息。无论您需要帮助解决特定问题，还是只想就某个特定主题进行对话，助手都在这里为您提供帮助。

`{chat_history}`

工具

助手可以要求用户使用工具来查找有助于回答用户原始问题的信息。人类可以使用的工具包括：

- > `GoogleSearch`: 搜索谷歌以获取相关和最新的结果。
- > `get_word_length`: `get_word_length(word: str) -> int`: 返回字符串的长度（以字符计）。
- > `get_number_words`: `get_number_words(str: str) -> int`: 返回字符串中的单词数。
- > `Summary`: 概括所提供的文本。

回应格式说明

在回应我时，请以下列两种格式之一输出回应：

****选项 1:****

如果您希望人类使用工具，请使用此选项。Markdown 代码片断采用以下格式：

```
```json
{{
 "action": string, \ 要采取的行动。必须是 GoogleSearch、get_word_length、
get_number_words, Summary
 "action_input": string \ 行动的输入
}}
```
```

****选项 2:****

如果您希望直接对人类回应，请使用此格式。Markdown 代码片断采用以下格式：

```
```json
{{
 "action": "最终答案",
 "action_input": string \ 您应该放置想要返回以便使用的内容
}}
```
```

用户输入

以下是用户的输入（记得以 `json blob` 的 markdown 代码片段回应一个动作，除此之外什么也不要）：

```
{input}
{agent_scratchpad}
```

运行 `agent.run("尼采的最后一本书是什么? ")` 和 `agent.run("它是什么时候写的? ")`，会分别正确地得到：

尼采的最后一本书是《瞧，这个人：人如何成其所是》（`Ecce Homo: How One Becomes What One Is`）。这本书是尼采在晚年写作的自传性质的作品，反映了他的哲学思想和个人生活。

以及：

尼采的《瞧，这个人：人如何成其所是》（`Ecce Homo: How One Becomes What One Is`）是在 1888 年写的。这是尼采创作的最后一部作品，完成于他精神崩溃前不久。

在使用自定义代理时，记住各种对话记忆对象（例如，`ConversationBufferMemory`）的 `memory_key` 属性必须与提示模板消息中的占位符（例如，`chat_history`）匹配。

若是出于生产目的，那么可以考虑将对话存储在某种数据库中。例如，通过 `RedisChatMessageHistory` 类将对话存储在 Redis 中。

数据连接

LangChain 提供了对连接外部数据源的全面支持。需要进行“摘要”和“检查增强生成”（RAG）时，这方面的支持显得非常重要。对于 RAG 应用，检索外部数据是模型生成结果前的一个关键步骤。

LangChain 对整个检索过程都“照顾有加”，其中包括文档加载、转换、文本嵌入、向量存储以及检索算法。它提供了多种文档加载器、高效的文档转换、与多个文本嵌入模型和向量存储的集成以及多种检索器（retriever）。这些功能显著增强了从语义搜索到高级算法（例如，`ParentDocumentRetriever`、`SelfQueryRetriever` 和 `EnsembleRetriever`）的各种检索方法。

加载器和转换器

LangChain 支持多种加载器（loader），其中包括：

- `TextLoader` 和 `DirectoryLoader`：这些加载器用于逐个或整个目录加载文本文件。
- `CSVLoader`：用于加载 CSV 文件。
- `UnstructuredHTMLLoader` 和 `BSHTMLLoader`：加载非结构化形式的 HTML 页面，或者使用 `BeautifulSoup4` 库^①来加载 HTML 页面。
- `JSONLoader`：加载 JSON 和 JSON Lines 文件（需要 `jq Python` 包）。
- `UnstructuredMarkdownLoader`：加载非结构化的 Markdown 页面（需要 `unstructured Python` 包）。
- 各种 PDF 加载器：其中包括 `PyPDFLoader`（需要 `pypdf`）、`PyMuPDFLoader`、`UnstructuredPDFLoader`、`PDFMinerLoader` 和 `PyPDFDirectoryLoader` 等。

例如，执行 Python 命令 `pip install pypdf` 后，就可以从工作目录中加载一个 PDF 文件，并采用以下方式显示其第一页：

```
from langchain.document_loaders import PyPDFLoader
loader = PyPDFLoader("bitcoin.pdf")
```

^① 译注：`Beautiful Soup` 是一个可以从 HTML 或 XML 文件中提取数据的 Python 库，它能通过你喜欢的转换器实现惯用的文档导航、查找、修改方式。注意，`BeautifulSoup4` 中的“4”代表这个库的版本。

```
pages = loader.load_and_split() # load_and_split 将文档分解成页，而 load 保持文档完整
print(pages[0])
```

加载步骤完成后，我们一般都希望将长文档分割成小块，并进行一些转换。长文本的分段至关重要，尽管这个过程可能相当复杂。使不同部分的文本在语义上保持连贯非常关键，但具体如何操作要取决于文本的类型。一个重点在于，不同块之间应允许一些重叠，以增强上下文。例如，应始终重新添加最后 N 个字符或最后 N 个句子。

LangChain 提供了文本分割器（text splitters，或者称为“文本拆分工具”），用于将文本分割成有意义的单元——通常是句子，但并非绝对（例如，想一想如果是代码会如何）。这些单元随后被组合成较大的块。当到达一个特定的阈值（以字符或 token 数量计）时，它们会变成单独的文本片段，并且为了提供上下文，它们之间会有一定的重叠。这样，在保持上下文的同时，可以有效地对文本进行分割和重组，以适应不同的语言处理任务。

下例展示了如何使用文本分割器 RecursiveCharacterTextSplitter:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 350,      # 块的大小
    chunk_overlap = 50,    # 块重叠
    length_function = len, # 可自定义
    separators=["\n\n", "\n", "(?<=\. )", " ", ""]
)
```

如这个例子所示，可以添加自定义分隔符和正则表达式。LangChain 原生支持基于 token 的 Markdown 和代码分割。

LangChain 还与 doctran 集成 (<https://github.com/psychic-api/doctran/tree/main>)，后者可以对文档执行一般性的“转换”，包括翻译、摘要、精炼等。例如，以下代码用 doctran 对文档进行翻译（要求先执行 pip install doctran 命令）:

```
from langchain.schema import Document
from langchain.document_transformers import DoctranTextTranslator
qa_translator = DoctranTextTranslator(language="spanish",
    openai_api_model=deployment_name)
translated_document = await qa_translator.atransform_documents(pages)
```



注意：遗憾的是，目前 LangChain 的 Doctran 实现仅支持 OpenAI 的各种模型。Azure OpenAI 的模型暂不支持。

嵌入和向量存储

在连接到向量存储之前，必须先嵌入文档。为此，LangChain 提供了一个完整的

Embeddings 模块，这个模块作为各种文本嵌入模型的接口。这简化了与 OpenAI、Azure OpenAI、Cohere 和 Hugging Face 等提供程序（provider）的交互。上一章在讨论生成文本的向量表示时说过，这个类实现了在向量空间中进行的语义搜索和相似文本分析。

LangChain 的核心 Embeddings 类提供了两个不同的方法：一个用于嵌入文档，另一个用于嵌入查询。由于嵌入 provider 在处理文档和搜索查询的方式上存在差异，所以造成了这种区别。

```
from langchain.embeddings import AzureOpenAIEmbeddings
import os
os.environ["OPENAI_API_TYPE"] = "azure"
os.environ["AZURE_OPENAI_API_VERSION"] = "2023-12-01-preview"
os.environ["AZURE_OPENAI_ENDPOINT"] = os.getenv("AOAI_ENDPOINT")
os.environ["AZURE_OPENAI_KEY"] = os.getenv("AOAI_KEY")
embedding_deployment_name=os.getenv("AOAI_EMBEDDINGS_DEPLOYMENTID")

embedding_model = AzureOpenAIEmbeddings(azure_deployment=embedding_deployment_name)
embeddings = embedding_model.embed_documents(["My name is Francesco", "Hello World"])
#或者:
#embeddings = embeddings_model.embed_query("My name is Francesco")
```

为了执行这段代码，需要先执行 `pip install tiktoken`。

下面，让我们导入从 Bitcoin.pdf 这个文档中导入文本块，并执行 `pip install chromadb`，然后玩玩向量存储。



注意：

如果收到“Failed to build hnswlib ERROR: Could not build wheels for hnswlib, which is required to install pyproject.toml-based projects”错误和“clang: error: the clang compiler does not support '-march=native'”错误，那么请设置以下 ENV 变量：

```
export HNSWLIB_NO_NATIVE=1
```

安装好 Chroma 后，运行以下代码来配置（并持久化）向量存储：

```
from langchain.vectorstores import Chroma
persist_directory = 'store/chroma/'
vectordb = Chroma.from_documents(
    documents=splits,
    embedding=embedding_model,
    persist_directory=persist_directory
)
vectordb.persist()
```

最后，像下面这样查询它：^①

```
vectordb.similarity_search("how is implemented the proof of work ",k=3)
```

随后应该得到以下输出：

```
[Document(page_content="4.Proof-of-Work\nTo implement a distributed timestamp server on a peer-to-peer basis, we will need to use a proof-\nof-work system similar to Adam Back's Hashcash [6], rather than newspaper or Usenet posts. \nThe proof-of-work involves scanning for a value that when hashed, such as with SHA-256, the", metadata={'page': 2, 'source': 'bitcoin.pdf'})]
```

```
Document(page_content='The steps to run the network are as follows:\n1)New transactions are broadcast to all nodes.\n2)Each node collects new transactions into a block. \n3)Each node works on finding a difficult proof-of-work for its block.\n4)When a node finds a proof-of-work, it broadcasts the block to all nodes.', metadata={'page': 2, 'source': 'bitcoin.pdf'}),
```

```
Document(page_content='would include redoing all the blocks after it.\nThe proof-of-work also solves the problem of determining representation in majority decision \nmaking. If the majority were based on one-IP-address-one-vote, it could be subverted by anyone \nable to allocate many IPs. Proof-of-work is essentially one-CPU-one-vote. The majority', metadata={'page': 2, 'source': 'bitcoin.pdf'})]
```



注意：这个示例使用的是 Chroma，但 LangChain 还支持其他几种向量存储，其接口抽象了所有这些存储。要了解更多详情，请访问 <https://python.langchain.com/docs/integrations/vectorstores/>。

检查器和 RAG

之前的示例代码执行了一次简单的相似性搜索。但如前所述，还有其他选项，例如使用最大边际相关性（Maximum Marginal Relevance, MMR）来加强查询结果的多样性，如下所示：

```
vectordb.max_marginal_relevance_search("how is implemented the proof of work", k=3)
```

还可以基于元数据进行查询，如下所示：

^① 译注：这个查询针对的是英文版的比特币白皮书，即本例所用的 `bitcoin.pdf`。如果你使用的是中文版的白皮书，可以这样问：“工作量证明是如何实现的？”译者主页提供了中英两种语言的白皮书，详情请访问 <https://bookzhou.com>。

```

vectordb.similarity_search(
    "how is implemented the proof of work",
    k=3,
    filter={"page": 4}
)

```

利用元数据过滤，我们可以使用 `SelfQueryRetriever`（要求安装 LARK），它可以访问向量存储，并使用 LLM 来生成元数据过滤器。

```

from langchain.retrievers.self_query.base import SelfQueryRetriever
from langchain.chains.query_constructor.base import AttributeInfo
from langchain.chat_models import AzureChatOpenAI

metadata_field_info = [
    AttributeInfo(
        name="source",
        description="文档名称。目前只能是'bitcoin.pdf'",
        type="string",
    ),
    AttributeInfo(
        name="page",
        description="文档中的页码",
        type="integer",
    ),
]
llm = AzureChatOpenAI(temperature=.3, azure_deployment=deployment_name)
retriever = SelfQueryRetriever.from_llm(
    llm,
    vectordb,
    "Bitcoin whitepaper",
    metadata_field_info,
    verbose=True
)
docs = retriever.get_relevant_documents("how is implemented the proof of work")

```

另一种获取更高质量文档的方法是使用压缩。这要求包含一个 `ContextualCompressionRetriever` 和一个 `LLMChainExtractor` 来从大量文档中提取相关信息（使用 LLM 链），然后传递给 RAG 部分。

若 LLM 的窗口上下文包含的文档超过 10 个，性能会大幅下降，但这个“锅”不应该由模型架构来“背”。这是因为模型必须在一个非常长的上下文的中间收集相关信息。所以，它会忽略提供的文档片段。可以通过重新对文档进行排名（打分）来解决这个问题，将不那么相关的放在中间，将更相关的放在开始和结束的位置。

从本质上讲，向量存储利用其索引来构建检索器，但向量存储索引并非构建检索器的唯一方式（检索器最终用于 RAG）。例如，LangChain 支持 SVM 和 TF-IDF（要求安装 scikit）以及其他几种检索器。

```
from langchain.retrievers import SVMRetriever
from langchain.retrievers import TFIDFRetriever

svm_retriever = SVMRetriever.from_documents(splits, embedding_model)
tfidf_retriever = TFIDFRetriever.from_documents(splits)
docs_svm = svm_retriever.get_relevant_documents("how is implemented the proof of work")
docs_tfidf = tfidf_retriever.get_relevant_documents("how is implemented the proof of work")
```

检索增强生成(RAG)

在 RAG 的场景中结合所有元素，最终的代码如下所示：

```
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationalRetrievalChain

memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
llm = AzureChatOpenAI(temperature=.3, azure_deployment=deployment_name)
retriever = vectordb.as_retriever()
qa = ConversationalRetrievalChain.from_llm(llm, retriever=retriever, memory=memory)
question = "how is implemented the proof of work"
result = qa({"question": question})
result['answer']
```

对于“*How is the proof of work implemented?*”这个问题，这样就可以得到一个合理的回答了，如下所示：

英文版: The proof-of-work in this context is implemented using a system similar to Adam Back's Hashcash. It involves scanning for a specific value that, when hashed (using SHA-256, for example), meets certain criteria. This process requires computational effort and serves as a way to secure and validate transactions on the network. When a node successfully finds a proof-of-work, it broadcasts the block containing the proof-of-work to all other nodes in the network

中文版: 工作量证明在这种情况下是使用一个类似 Adam Back 提出的哈希货币 (Hashcash) 的工作量证明系统。它涉及搜索一个特定的数，该数在被哈希时 (例如，使用 SHA-256) 符合特定的标准。这个过程需要计算量，并作为一种保障和验证网络上交易的方式。当一个节点找到了工作量证明，就将包含工作量证明的区块广播给网络中的其他所有节点。



注意：和往常一样，我们可以使用之前讨论过的任何选项来替换记忆、检索器和 LLM。可以选择使用托管模型、基于摘要的记忆或 SVM 检索器。

上述代码是有效的，但距离正式投入生产环境还远。它需要嵌入到一个真实的聊天界面中 (包括 UI、登录等)；需要实现用户特有的记忆；数据库负载应与资源平衡；而且需要添

加处理程序、备用方案和日志记录机制。简而言之，传统的软件和工程方面的要素缺失，使这个有趣的实验难以转化为功能性的产品。



注意：LlamaIndex 是 LangChain 在 RAG 方面的一个有力的竞争对手。它是一个专门的库，用于数据摄取、数据索引和查询接口。使用它，我们可以方便地从头开始构建一个具有 RAG 模式的 LLM 应用。

Microsoft Semantic Kernel

作为一种轻量级 SDK，开发者能利用 Semantic Kernel (SK) 无缝地融合传统编程语言 (C#、Python 和 Java) 与 LLM 的尖端能力。和 LangChain 一样，SK 也作为一个 LLM 编排器/协调程序工作。

SK 提供了与 LangChain 相似的一系列功能来创建代理，例如提示模板、链和规划等。和 LangChain 一样，SK 也区分了基于文本补全的模型和基于聊天的模型，它们的接口略有不同。

SK 的基本使用场景从总结冗长的对话并将重要任务添加到待办事项列表中，一直到协调复杂的任务（例如，旅游规划）。SK 的设计围绕插件（以前称为“技能”）展开，开发者可以将其构建为语义或原生（本机）代码模块。这些插件与 SK 的记忆协作以执行基于上下文的操作，并与连接器协作以处理实时数据和操作。SK 的规划器（**planner**）可以接收用户的请求，并将其转化为所需的插件、记忆和连接器以实现预期的结果。



注意：SK 刚问世的时候曾用不同的名称来指同一事物，特别是插件、技能和函数。最终，SK 确定了使用“插件”一词。然而，尽管 SK 现在的文档在这个术语上的使用是一致的，但其代码有时反映的仍然是旧的约定。

SK 支持来自 OpenAI、Azure OpenAI 和 Hugging Face 的模型，并且已在 GitHub 上开源^①。

SK 的主要组成部分如下：

- **内核 (Kernel)：**内核是一个包装器，负责运行由开发者定义的管道/链。

^① 译注：<https://github.com/microsoft/semantic-kernel>

-
- **KernelArguments**: 这是注入到内核中的通用抽象上下文。
 - **语义记忆**: 这是用于在向量数据库中存储和检索上下文的连接器。
 - **插件**: 由一组语义函数（本质上是 LLM 提示）、原生/本机函数和连接器组成。它们可以在概念上分为两个不同的组（尽管在技术上无区别）：
 - **连接器**: 我们用它获取额外的数据或执行额外的操作（例如，MS Graph API、Open API、网页抓取器或自定义连接器）。可以将其看作是 LangChain 的等价物，将不同的函数整合到了一起。
 - **函数**: 可以是语义的（由提示定义的代码），也可以是原生的（真正的代码）。它们等同于 LangChain 的“工具”。
 - **规划器 (Planner)**: 这相当于 LangChain 的“代理”，它们使用预加载的函数和连接器来自动创建链。



注意: SK 开发团队创建了这些组件的大量示例，详情请访问以下网址：
<https://github.com/microsoft/semantic-kernel/tree/main/dotnet/samples/>。

SK 采用了 OpenAI 插件规范，旨在建立一个由兼容的插件构成的生态系统，使其能在如 ChatGPT、Bing 和 Microsoft 365 等知名 AI 应用和服务之间无缝运作。因此，使用了 SK 的开发者可以将他们的插件扩展到这些平台，而无需重写代码。此外，为 ChatGPT、Bing 和 Microsoft 365 设计的插件也可以与 SK 集成，促进插件的跨平台互操作性。



注意: 本章使用的是 SK 1.15.1。与 SK 相比，LangChain 显得更稳定（版本变化较慢）。因此，这里提供的代码已简化到极其基本的水平，只包括一些关键片段和核心概念。希望这些基础元素以后不会“大改”。

插件

从最基本的层面上说，插件（plug-in）是为 AI 应用程序和服务设计的一组函数的集合。我们将这些函数作为基本的构建单元来开发应用程序，以处理用户查询和内部需求。可以通过一个规划器（planner）来手动或自动激活函数（进而激活插件）。

每个函数都必须配备一个完善的语义描述，详细说明其行为。这种描述应该清楚地表达函数的所有特性，包括其输入、输出和潜在的副作用，以便作为链或规划器基础的 LLM 能够理解。只有把语义框架定义好了，才能保证规划器不会产生意外的结果。

总之，可以将插件理解为一种函数库，它们是你开发的 AI 应用程序的功能单元。在自动

化编排中，其有效性依赖于完善的语义描述。这些描述使规划器能够智能地为每种情况选择最佳函数，从而实现更加流畅和更加私人的用户体验。

内核配置

为了在应用程序中使用 SK，必须先添加其 NuGet 包。为此，请在 C# Polyglot 笔记本中执行以下命令：

```
#r "nuget: Microsoft.SemanticKernel, *-.*"
```



注意：可能还需要添加 `Microsoft.Extensions.Logging`、`Microsoft.Extensions.Logging.Abstractions` 和 `Microsoft.Extensions.Logging.Console`。用类似的 `nuget` 命令安装即可。

```
using Microsoft.SemanticKernel;
using System.Net.Http;
using Microsoft.Extensions.Logging;
using System.Diagnostics;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;

var httpClient = new HttpClient();

IKernelBuilder builder = Kernel.CreateBuilder();

// 要求部署 ID、终结点和密钥等已经定义好，
// 详情参见第 1 章或者 https://bookzhou.com/2024/06/21/1188/。
builder.AddAzureOpenAIChatCompletion(
    deploymentName: AOAI_DEPLOYMENTID,
    endpoint: AOAI_ENDPOINT,
    apiKey: AOAI_KEY,
    httpClient: httpClient);
builder.Services.AddLogging(c => c.AddConsole().SetMinimumLevel(LogLevel.Information));
Kernel kernel = builder.Build();
```

如本例所示，可以指定一个日志记录行为，还可以指定一个特定的 `HttpClient` 实现。如果使用的不是 .NET Interactive Notebook，那么应考虑将 `HttpClient` 与内核一起插入到一个

using 语句中。^①



注意：和 LangChain 一样，SK 也支持与 Azure OpenAI 模型以外的模型集成。例如，可以轻松地连接 Hugging Face 的各种模型。

内核默认集成了自动重试机制，用于处理 AI 调用期间的节流（throttling）和超时（timeout）等瞬时错误。^②这意味着如果 AI 服务在处理请求时遇到这类问题，内核会自动尝试再次发送请求，而不需要开发者手动介入。



注意：内核的作用至关重要，因为它是唯一可以配置 LLM 服务的地方。不过，它的使用方式是多种多样的。例如，为了调用函数和规划器，可以将一个内核传递到它们的定义中，也可以直接在内核实例本身上以流式 API 的形式调用 RunAsync 方法。

^③

语义或提示函数

顾名思义，语义（或提示）函数通过提示来进行明确的描述。和原生/本机函数一样，语义函数也是插件的基本构建单元。

可以用两种方式来定义和执行语义函数：通过文件来配置以及通过内联的定义。Microsoft.SemanticKernel.Plugins.Core 命名空间中也提供了原生/本机插件。

内联配置非常简单直接：

```
using Microsoft.SemanticKernel.Connectors.OpenAI;
```

^① 译注：如果代码不是在一个交互式笔记本环境中运行，而是在一个常规的应用程序中运行，就需要把创建的 HttpClient 和 Kernel 对象放入一个 using 语句中，以确保它们被正确管理，并在不再需要时被清理。由于过于基础，所以超出了本书的范围。请访问 <https://bookzhou.com> 了解更多的 C# 参考书。

^② 译注：节流通常是服务提供者为了控制服务器负载而实施的一种限制，它会暂时阻止用户发送过多的请求。而超时则可能是因为服务器响应过慢或网络问题导致的。

^③ 译注：流式 API 也称为流畅 API（Fluent API），即下一个函数调用的上下文基于上一个函数调用准备好的上下文。整个调用链通过一个空的上下文来终止。

```

var FunctionDefinition = "用户: {{$input}} \n 从用户的输入中获取他/她的意图。" +
    "意图应该是以下之一: 电子邮件, 电话会议, 网上会议, 面对面会议。";

var getIntentFunction = kernel.CreateFunctionFromPrompt(
    FunctionDefinition, new OpenAIPromptExecutionSettings
    {
        MaxTokens = 200,
        Temperature = 0.3,
        TopP = 1
    });

var result = await getIntentFunction.InvokeAsync(kernel, new()
    { ["input"] = "这周搞个视频通话怎么样? " });
Console.WriteLine(result);
// 程序应输出:
// 意图: 网上会议。

```

可以通过 `KernelArguments` 添加更多输入变量, 如下所示:

```

var FunctionDefinition = "用户: {{$input}} \n 从用户的输入中获取他/她的意图。意图应该是以下
之一: {{$options}}。";

var getIntentFunction = kernel.CreateFunctionFromPrompt(
    FunctionDefinition, new OpenAIPromptExecutionSettings
    {
        MaxTokens = 200,
        Temperature = 0.3,
        TopP = 1
    });

var variables = new KernelArguments();
variables.Add("input", "这周搞个视频通话怎么样? ");
variables.Add("options", "电子邮件, 电话会议, 网上会议, 面对面会议");
// 是的, 你没有看错, 可以使用全角逗号分隔不同的选项

var result = await getIntentFunction.InvokeAsync(kernel, variables);
Console.WriteLine(result);

```

这些变量对函数可见, 而且可以使用 `{{ 变量名 }}` 注入到语义提示中。

在真实的项目中, 可能需要使用单独的文件来配置提示函数。这些配置文件的目录结构应按采用以下模式:

```

在一个 Plugins 文件夹中
|
__ 创建一个 { 插件名 } 插件文件夹
    |
    __ 创建一个 { 语义函数名 } 文件夹
        |
        __ config.json

```

__ skprompt.txt

提示应该连同内联定义放到 skprompt.txt 文件中，而 config.json 文件应遵循以下结构：

```
{
  "schema": 1,
  "type": "completion",
  "description": "为用户创建聊天回应",
  "execution_settings": {
    "default": {
      "max_tokens": 1000,
      "temperature": 0
    },
    "gpt-4": {
      "model_id": "gpt-4-1106-preview",
      "max_tokens": 8000,
      "temperature": 0.3
    }
  },
  "input_variables": [
    {
      "name": "request",
      "description": "用户的请求。",
      "required": true
    },
    {
      "name": "history",
      "description": "对话历史。",
      "required": true
    }
  ]
}
```

然后，采用以下方式来调用文件中定义的一个函数：

```
// 例如，假定插件目录是当前目录下的/ABC/DEF子目录，那么应该这样写：
// var pluginsDirectory = Path.Combine
// (System.IO.Directory.GetCurrentDirectory(), "ABC", "DEG");
var pluginsDirectory = Path.Combine(System.IO.Directory.GetCurrentDirectory(), "path",
"to", "plugins", "folder");
var basePlugin = kernel.CreatePluginFromPromptDirectory(kernel, pluginsDirectory, "替换
为插件名");
var result = await basePlugin["FunctionName"].InvokeAsync(kernel, variables);
```

文件定义旨在概括函数定义。它与插件的定义严格相关，稍后会详细介绍。

原生/本机函数

原生函数通过代码定义，可以被视为插件的确定性部分。与提示函数一样，原生函数可以

在一个文件中定义，该文件的目录结构如下所示：

```
在一个 Plugins 文件夹中
|
__ 创建一个{插件名}插件文件夹
   |
   __ 创建一个{语义函数名}文件夹
      |
      __ config.json
      __ skprompt.txt
   |
   __ {插件名}Plugin.cs, 该源代码文件包含用于给定插件的所有原生/本机函数
```

下面展示了一个简单的原生函数：

```
public class MathPlugin
{
    [KernelFunction, Description("求一个数的平方根")]
    public string Sqrt(string number)
    {
        return Math.Sqrt(Convert.ToDouble(number)).ToString();
    }
}
```

接收单个输入时，无需指定除函数描述之外的其他任何内容。然后，规划器（代理）可以使用这一信息来决定是否调用该函数。

下面是一个带有更多输入参数的例子：

```
[KernelFunction, Description("两个数相加")]
public int Add(
    [Description("第一个数")] int number1,
    [Description("第二个数")] int number2)
    => number1 + number2;
```

可以采取以下方式调用原生函数（假定插件名为 `mathPlugin`）：

```
var result2 = await mathPlugin["Add"].InvokeAsync(
    kernel, new (){ {"number1", 15}, {"number2", 7} });
Console.WriteLine(result2);
```

从本质上说，首先是将隐式定义的插件导入到内核，然后调用它。

如果 `InvokeAsync` 方法只有一个输入，那么必须传递一个字符串，并在函数体内处理转换。或者可以使用上下文变量，并让框架帮你转换输入。

取决于其内部逻辑——可能涉及需要的任何软件组件，包括连接到数据库、发送电子邮件

等——原生函数通常返回一个字符串。或者，它们也可以简单地执行一个操作，例如 `void` 或 `Task` 类型的操作。

现实世界中的原生函数有三种不同的使用场景：

- 确定性地转换为输入或输出，与更多语义函数链接
- 在语义函数理解了用户意图后执行操作
- 作为规划器/代理的“工具”

此外，如果将包含函数的插件传递给 `Kernel` 对象，那么可以从原生函数中调用提示（语义）函数。

核心插件

我们可以通过组合提示和原生函数来构建一个自定义插件。此外，SK 还提供了一系列核心插件，它们位于 `Microsoft.SemanticKernel.Plugins.Core` 命名空间中。

- `ConversationSummaryPlugin`：用于汇总对话（创建摘要）
- `FileIOPlugin`：处理文件系统的读写
- `HttpPlugin`：启用 API 调用，提供 HTTP 功能
- `MathPlugin`：执行数学运算
- `TextMemoryPlugin`：保存或调用长期或短期记忆中的信息（现在转移到了 `Microsoft.SemanticKernel.Plugins.Memory` 命名空间）
- `TextPlugin`：进行确定性的文本处理（大写、小写、删除头尾空白字符等）
- `TimePlugin`：获取与当前时间和日期相关的信息
- `WaitPlugin`：暂停执行指定的持续时间

这些插件可以以常规方式导入到内核中，其用法与用户自定义的插件没有区别，例如：

```
kernel.AddFromType<TimePlugin>("Time");
```

OpenAPI 插件非常有用。通过在内核上调用 `ImportPluginFromOpenApiAsync` 方法，可以调用遵循 OpenAPI 模式的任何 API。这将在第 8 章进一步扩展。

一个有趣的特性是，将插件导入内核后，可以在提示中使用以下语法引用函数（不管是原生的还是语义的）：`{{ 插件名. 函数名 $ 变量名 }}`。

数据与规划器

为了有效地运作，规划器（或代理）需要访问能实际“干活儿”的工具。工具包括某种形式的记忆、编排逻辑以及由用户提供的目标。我们之前已经构建了工具（即插件），现在是时候实际构造记忆、数据和规划器了。

记忆

SK 没有明确区分长期记忆和对话记忆。在对话记忆的情况下，我们必须构建自己的自定义策略（例如，LangChain 的摘要记忆、实体记忆等）。SK 支持以下几种记忆存储：

- Volatile（这模拟了一个向量数据库）。它不应在生产环境中使用，但对于测试和概念验证（Proof of Concepts, POC）非常有用。
- AzureCognitiveSearch（这是 Azure 中唯一提供了完全托管服务的记忆选项）。
- Redis
- Chroma
- Kusto
- Pinecone
- Weaviate
- Qdrant
- Postgres（使用 NpgsqlDataSourceBuilder 和 UseVector 选项）
- 更多即将推出……

还可以构建一个自定义的记忆存储，实现 `IMemoryStore` 接口，并将其与嵌入生成（embedding generation）和通过某种相似性函数进行搜索的功能相结合。^①

需要在记忆存储上实例化一个记忆插件，以便规划器或其他插件使用它来回忆信息。

```
using Microsoft.SemanticKernel.Skills.Core;
var memorySkill = new TextMemorySkill(kernel.Memory);
kernel.ImportSkill(memorySkill);
```

`TextMemorySkill` 是一个带有原生函数的插件。它简化了从长期或短期记忆中保存和回忆信息的过程。它支持以下方法：

- `RetrieveAsync`：执行基于键的查找以获取特定记忆。
- `RecallAsync`：这使得基于输入文本的语义搜索和相关记忆的返回成为可能。
- `SaveAsync`：将信息保存到语义记忆中。
- `RemoveAsync`：移除特定的记忆。

所有这些方法都可以在内核上调用，如下所示：

```
result = await kernel.InvokeAsync(memoryPlugin["Recall"], new()
{
    [TextMemoryPlugin.InputParam] = "Ask: what's my name?",
```

^① 译注：“嵌入生成”指的是创建一种可以表示数据点的向量形式，这些向量可以用来进行有效的相似性搜索。相似性函数则用来比较这些嵌入向量，以找出彼此之间的相似度。

```

[TextMemoryPlugin.CollectionParam] = MemoryCollectionName,
[TextMemoryPlugin.LimitParam] = "2",
[TextMemoryPlugin.RelevanceParam] = "0.79",
});
Console.WriteLine($"Answer: {result.GetValue<string>}");

```

还可以定义一个语句函数，其中包括来自提示的回忆方法，并构建一个简单的 RAG 模式。

```

var recallFunctionDefinition = @"
Using ONLY the following information and no prior knowledge, answer the user's questions:
---INFORMATION---
{{recall $input}}
-----
Question: {{$input}}
Answer:";
var recallFunction = kernel.CreateFunctionFromPrompt(RecallFunctionDefinition, new
OpenAIPromptExecutionSettings() {
    MaxTokens = 100 });
var answer2 = await kernel.InvokeAsync(recallFunction, new()
{
    [TextMemoryPlugin.InputParam] = "who wrote bitcoin whitepaper?",
    [TextMemorySkill.CollectionParam] = MemoryCollectionName,
    [TextMemoryPlugin.LimitParam] = "2",
    [TextMemorySkill.RelevanceParam] = "0.85"
});
Console.WriteLine("Ask: who wrote bitcoin whitepaper?");
Console.WriteLine("Answer:\n{0}", answer2);

```

输出可能是这样的：

```
Satoshi Nakamoto wrote the Bitcoin whitepaper in 2008
```

在 SK 内访问 SQL

LangChain 有自己的官方 SQL 连接器来执行数据库查询。但是，SK 目前还没有专门的插件。

在以下情况下，我们有必要直接访问 SQL：

- 需要获取动态数据。
- 由于 token 的限制，无法使用提示定位^①。
- 在某些情况下，尝试将传统的 SQL 数据库数据同步到一个支持快速相似性搜索的向

^① 译注：所谓“提示定位”（prompt grounding），是指根据给定的提示或上下文来生成回应。若 token 有数量上的限制，就会失去上下文，所以就无法使用“提示定位”了。

量数据库可能会引入额外的复杂性或者不一致性的问题。如果没有这方面的需求，那么应该直接访问 SQL。

考虑这个问题：“我们五月份最大的客户是谁？”即使拥有一个包含销售信息的向量数据库，相似性搜索也无法帮助我们回答这个问题，这意味着应该运行一个结构化（并且是确定性的）查询。

微软目前正在开发一个名为 **Kernel Memory** 的库（也可用作服务），地址是 <https://github.com/microsoft/kernel-memory>。这个库基本上复制了 SK 的记忆部分。开发团队计划直接向 SK 添加 SQL 访问功能，但目前还不清楚官方的 SQL 插件是作为 Semantic Memory 库的一部分发布，还是作为 SK 的核心插件发布。

值得一提的是，语义库中有一个名为 **Natural Language to SQL (NL2SQL)** 的初步示例，具体可以访问 <https://github.com/microsoft/kernel-memory/tree/NL2SQL/examples/200-dotnet-nl2sql>。这个示例是作为一个控制台应用来构建的，其中包括一个语义记忆部分，用于获取正确的数据库 schema（基于用户查询，以防有多个 schema）。然后，由语义函数创建一个 T-SQL 语句并执行。目前只能复制并粘贴这个示例的部分内容，以在 SK 中构建一个自定义插件，并使用它或将其提供给一个规划器（**planner**）。

更一般地说，通过 LLM 生成的查询（GPT-4 在这项任务上比 GPT-3.5-turbo 更好）进行 SQL 访问时，应该优先考虑最小权限访问，并实现注入预防措施以增强安全性（这方面的详情请参见第 5 章）。

非结构化数据摄取

要摄取（**ingest**）结构化数据，最佳方法是构建 API 调用（可能遵循 **OpenAPI** 模式并使用核心插件），或直接创建一个插件来查询数据库。对于非结构化数据（例如，图像），我们应该使用专门的向量存储（位于 SK 的外部），或者创建一个同时支持图像的自定义记忆存储。对于文本文档（这是 AI 应用中最常见的数据类型），则可以利用 SK 本身就有的工具，例如文本分块器（**text chunker**）。

例如，可以使用 PdfPig 来导入 PDF 文件，如下所示：

```
using UglyToad.PdfPig.DocumentLayoutAnalysis.TextExtractor;
using Microsoft.SemanticKernel.Memory;
using Microsoft.SemanticKernel.Text;

var BitcoinMemoryCollectionName = "BitcoinMemory";
var chunk_size_in_characters = 1024;
var chunk_overlap_in_characters = 200;
var pdfFileName = "bitcoin.pdf";
var pdfDocument = UglyToad.PdfPig.PdfDocument.Open(pdfFileName);
```

```
foreach (var pdfPage in pdfDocument.GetPages())
{
    var pageText = ContentOrderTextExtractor.GetText(pdfPage);
    var paragraphs = new List<string>();
    if (pageText.Length > chunk_size_in_characters)
    {
        var lines = TextChunker.SplitPlainTextLines(pageText, chunk_size);
        paragraphs = TextChunker.SplitPlainTextParagraphs(
            lines, chunk_size_in_characters,
            chunk_overlap);
    }
    else
    {
        paragraphs.Add(pageText);
    }
    foreach (var paragraph in paragraphs)
    {
        var id = pdfFileName + pdfPage.Number + paragraphs.IndexOf(paragraph);
        await textMemory.SaveInformationAsync(MemoryCollectionName, id: "info1", text:
            "My name is Andrea");
    }
}

pdfDocument.Dispose();
```

这种方法显然可以扩展至任何文本文档。

规划器

现在，我们已经了解了所有可以连接的工具，接着可以创建一个代理——或者在 SK 中称为规划器（planner）。SK 提供了两种类型的规划器：

- **Handlebars 规划器**：这种规划器利用 Handlebars 语法为给定的目标生成一份完整的计划。它通过一系列步骤来传递输出，即每一步的输出可以成为下一步的输入。这种方法在需要精确和结构化的计划生成场景（例如，自动化工作流程、项目管理等）中非常有用，因为它可以详尽地描述每一个步骤，确保每个环节都精确无误地执行。另外，Handlebars 语法为规划器提供了循环和条件逻辑的能力。这意味着规划器可以根据输入数据的不同，动态地调整其行为，从而更加智能、适应性更强。
- **函数调用逐步规划器**：设计用于执行一个顺序的计划，直到给定目标完成为止。它基于一个名为**模块化推理、知识和语言**（Modular Reasoning, Knowledge, and Language, MRKL）的神经符号架构，后者其实就是 ReAct 背后的核心理念。此类型的规划器特别适合需要在插件选择上具有较强的适应性，或者需要在相互关联的阶段管理复杂任务的情况。但要注意的是，一旦使用的插件数量超过 10 个，这种规划器可能会增加出现“幻觉”的机会（即 AI 生成了错误或无意义的信息）。



提示：OpenAI 函数调用功能通过 `GetOpenAIFunctionResponse()` 无缝地包装了较低级别的 API `GetChatMessageContentAsync`。这种功能其实就相当于一种单步规划器。



注意：第 8 章将创建一个利用了 SK 规划器的酒店预订应用程序。

由于规划器可能以未预见的方式合并函数，因此必须确保只有预期的函数被公开。同样重要的是，要向这些函数应用负责任 AI 原则（**responsible AI principles**），确保它们的使用符合公平性、可靠性、安全性和隐私性。

像 `LangChain` 的代理一样，规划器在底层使用 LLM 提示来生成计划——尽管这一最终步骤可能被隐藏在重要的编排、转发和解析逻辑之下，特别是在逐步规划器中。

SK 的规划器允许我们指定一些设置，其中包括：

- **相关性阈值（RelevancyThreshold）：**为了将一个函数包括进来，它需要具有的最低相关性得分。可能需要根据嵌入引擎、用户请求、步骤目标和可用的函数来调整这个值。
- **最大相关函数数（MaxRelevantFunctions）：**计划中包含的相关函数的最大数量。这限制了来自计划创建请求中的语义搜索的结果。
- **排除的插件（ExcludedPlugins）：**要从计划创建请求中排除的插件的列表。
- **排除的函数（ExcludedFunctions）：**要从计划创建请求中排除的函数的列表。
- **包含的函数（IncludedFunctions）：**要在计划创建请求中包括的函数的列表。
- **最大 token 数（MaxTokens）：**计划中允许的最大 token 数量。
- **最大迭代次数（MaxIterations）：**计划中允许的最大迭代次数。
- **最小迭代时间毫秒（MinIterationTimeMs）：**迭代之间等待的最短时间，以毫秒为单位。



提示：对相关性进行过滤，可以显著提高计划过程的整体性能，而且更有可能制定出成功的计划以完成复杂的目标。

生成计划时所产生的成本主要在于延迟和 token（也就是金钱）。但是，有一种方法可以节

省时间和金钱，并降低产生非预期结果的风险：为用户经常询问的常见场景预先创建（顺序）计划。这些计划离线生成，并以 JSON 格式存储在一个记忆存储中。然后，当用户的意图与这些常见场景之一相符时，基于一次相似性搜索，系统检索并执行相关的预加载计划，从而避免了每次都要实时生成计划。这种方法能提高性能，并减少与使用规划器相关的成本。然而，它只能与更固定的顺序规划器一起使用，因为逐步规划器太过动态。



注意：目前，SK 的规划器在某些方面的表现比不上 LangChain 的代理。特别是在使用 XML 计划的顺序规划器时，SK 规划器似乎更容易出现解析错误，特别是当使用 GPT-3.5-turbo 而非 GPT-4 时。这些错误可能是因为 SK 的顺序规划器在生成计划时，试图在一开始就生成所有步骤，而不能有效处理意外的错误。

Microsoft Guidance

Microsoft Guidance 是一种领域特定语言（Domain-Specific Language, DSL）^①，它用于管理与大语言模型（LLM）的“模板化”交互，这些模型也包括 Hugging Face 提供的那些。

Guidance 类似于 Web 应用所使用的模板语言 Handlebars，但它还确保了与语言模型的 token 处理顺序一致的顺序代码执行。当提示变得不必要地重复、冗长或啰嗦时，再与模型进行交互就可能产生高昂的成本。因此，Guidance 的目标是在保持对输出更大控制的同时，最小化与 LLM 交互的费用。



注意：Guidance 还包含一个用于测试和评估 LLM 的模块（目前只能使用 Python）。

Microsoft Guidance 的竞争对手是 LMQL，这是一个用于对提示进行程序化^②的库，它使用了类型、模板、约束和一个经过了优化的运行时。

^① 译注：领域特定语言也称为特定域语言，是专门针对特定应用领域的计算机语言，和可以用在多种领域的通用语言（GPL）恰好相反。例如，专门用于网页设计的 HTML 就属于领域特定语言。

^② 译注：将提示一步一步地写清楚，就是对提示进行“程序化”。

配置

要安装 Guidance，只需在 Python 终端运行一条简单的 `pip install guidance` 命令即可。Guidance 同时支持 OpenAI 和 Azure OpenAI 模型，也支持 transformers 格式的本地大模型，例如 Llama、StableLM 和 Vicuna 等。对于本地模型，Guidance 还支持 Acceleration（加速），这是 Guidance 内部使用的一种技术，用于缓存 token 并优化生成速度。

模型

安装好 Guidance 后，可以使用以下代码来配置 OpenAI 模型。注意，要求事先设置好代码中要求的一系列环境变量。

```
from guidance import models, guidance, gen
from guidance import system, user, assistant
import os

# 更多环境设置内容请访问 https://bookzhou.com/2024/06/21/1188/
os.environ['OPENAI_API_VERSION'] = "2024-05-01-preview" # 当前最新的 OpenAI API 版本
os.environ['AOAI_DEPLOYMENTID'] = "替换为你的部署名称"
os.environ['AOAI_ENDPOINT'] = "替换为你的 Azure OpenAI 终结点 URI"
os.environ['AOAI_KEY'] = "替换为你的 Azure OpenAI API 密钥"

# 配置 Azure OpenAI 模型
llm = models.AzureOpenAI(
    model='gpt-35-turbo', # 模型名称
    azure_deployment=os.getenv("AOAI_DEPLOYMENTID"), # Azure OpenAI 部署名称
    api_key=os.getenv("AOAI_KEY"), # Azure OpenAI API 密钥
    azure_endpoint=os.getenv("AOAI_ENDPOINT") # Azure OpenAI 终结点
)
```

还可以使用 Hugging Face 的 transformers 版本模型（需同意特殊的许可条款）以及本地模型。这意味着，除了云端的 OpenAI 模型之外，用户还可以选择使用本地部署的模型，这为那些关心数据隐私或网络延迟的应用场景提供了更大的灵活性。

基本用法

为了开始进行模板化的聊天，让我们先来测试以下代码：

```
with system():
    llm += "你是一名猫的专家"
with user():
    llm += "世界上最小的猫是什么？"
with assistant():
    llm += gen("回答", stop="。", max_tokens=100)
```

这会产生如下所示的输出：

```
system
你是一名猫的专家
user
世界上最小的猫是什么？
assistant
世界上最小的猫是袖珍猫（Singapura），它们是一种来自新加坡的短毛猫，体型非常小，成年体重只有 1.8-2.7 公斤左右
```

从表面上看，**Guidance** 很像是一种常规的模板语言，类似于传统的 **Handlebars** 模板，支持变量插值和逻辑控制。然而，与传统的模板语言不同的是，**Guidance** 程序使用了一个有序的线性执行序列，这个序列直接对应于语言模型所处理的 **token** 的顺序。这种内在的联系有助于模型在任意给定的执行点上生成文本（通过 **gen** 函数）或者实现逻辑控制流选择。



注意：有的时候，模型的生成会失败，而 **Guidance** 会默默地返回原始模板而不显示错误消息。在这种情况下，可以查看 `program._exception` 属性或者 `program.log` 文件的内容以获取更多详细信息。

语法

Guidance 的语法让人联想到 **Handlebars**，但它具有一些独特的、增强的功能。使用 **Guidance** 时，它会在被调用时生成一个程序，可以通过提供实参来执行该程序。这些实参可以是单个的或迭代的，所以非常灵活。

模板结构支持迭代，`{{#each}}` 标签就是一个例子。另外，还可以使用 `{{! ...}}` 语法添加注释，例如 `{{! 这是一个注释}}`。摘自 **Guidance** 文档^①的以下代码就是一个很好的例子。

```
examples = [
  {
    'input': '我写过关于莎士比亚的文章',
    'entities': [
      {'entity': '我', 'time': '现在'},
      {'entity': '莎士比亚', 'time': '16 世纪'}
    ],
    'reasoning': '我可以写关于莎士比亚的文章，因为他相对于我生活在过去。',
    'answer': '否'
  },
  {
```

^① 译注：<https://guidance.readthedocs.io/>

```

        'input': '莎士比亚写过关于我的文章',
        'entities': [
            {'entity': '莎士比亚', 'time': '16 世纪'},
            {'entity': '我', 'time': '现在'}
        ],
        'reasoning': '莎士比亚不可能写过关于我的文章，因为他在我出生前就去世了。',
        'answer': '是'
    }
]

# 定义 Guidance 程序
@guidance
def anachronism_query(llm, query, examples):
    prompt_string = """给定一个句子，告诉我它是否包含时代错误（即基于与实体相关的时间段，这件事是否可能发生）。

    以下是一些示例：
    """
    for ex in examples:
        prompt_string += f"句子: { ex['input'] }" + "\n"
        prompt_string += "实体和日期:\n"
        for en in ex['entities']:
            prompt_string += f"{en['entity']}: {en['time']}" + "\n"
        prompt_string += f"推理: {ex['reasoning']}" + "\n"
        prompt_string += f"时代错误: {ex['answer']}" + "\n"

    llm += f'''{prompt_string}
    现在判断以下是否是一个时代错误:

    句子: { query }
    实体和日期:
    { guidance.gen(name="entities", max_tokens=10000) }'''
    llm += "推理:"
    llm += guidance.gen(name="reason", max_tokens=10000)
    llm += f'''\n时代错误: { guidance.select(["Yes", "No"], name="answer") }'''
    return llm

# 调用模型
generate = llm + anachronism_query("霸王龙咬伤了我的狗", examples)

```

主要特性

在总体覆盖范围上，Guidance 似乎比 SK 更为有限。Guidance 本来也没有打算成为一个通用的、包罗万象的编排器（orchestrator），配备一系列内部和外部的工具以及原生功能，从而实现 AI 应用程序的完全支持。但是，由于其模板语言的存在，它确实能够创建结构化的数据格式（例如，JSON、XML 等），而且其语法是经过验证的。这对于调用 API、生成流程（类似于思维链，但也包括 ReAct）以及进行基于角色的聊天极为有用。

尽管不是专门为此目的而设计，但使用 **Guidance** 同样可以调用外部函数，从而构建代理。因此，与 **LangChain** 和 **SK** 相比，其公开的编程接口显得更加底层和直接。**Guidance** 还尝试优化（或者说缓解）大语言模型（LLM）的一些根本性的问题，例如 **token** 修复和过高的延迟。理想情况下，**Guidance** 能够增强 **SK**（或 **LangChain**）的功能，作为连接到基础模型（包括本地模型和各种 **Hugging Face** 模型）的桥梁，并将它的特性添加到 **SK** 更企业级的接口中。

总的来说，可以使用 **Guidance** 在 AI 应用的上下文中做几乎所有事情，而且几乎不用费什么力气。但是，与前面提到的两个库相比，你可以把它看作一个较低级的库，而且由于它使用单一连续流编程接口，所以灵活性较低。

token 修复

Guidance 引入了一个名为 **token 修复**（**token healing**）的概念，其目的是解决在提示文本结尾与生成文本开始处常见的 **tokenization** 问题，特别是当生成的 **token** 跨越了提示的边界时，这可能导致语法上的不连贯或语义上的不明确。为了有效执行 **token 修复**，**Guidance** 需要直接访问并集成到特定的模型中，如 `guidance.llms.Transformers`，这意味着它目前不兼容于 **OpenAI** 或 **Azure OpenAI** 等远程 API 服务。

语言模型的核心是 **token**，它们是文本的基本单位，通常类似于词汇，但不一定。在模型的工作过程中，每个“提示”都必须转换为一系列 **token ID**，这是模型理解和生成文本的基础。**GPT** 风格的模型使用字节对编码（**Byte-Pair Encoding**, **BPE**）以一种“贪婪”方式将输入字符映射到 **token ID**。这一过程在训练阶段非常有效，但当应用于生成新文本时，尤其是在提示与生成文本的交界处，可能会产生一些微妙的挑战。

例如，提示 "This is a " 若用 "fine day." 补全，那么将生成 "This is a fine day."。使用 **GPT2 BPE** 对提示 "This is a " 进行 **tokenization**，结果是 [1212, 318, 257, 220]，而 "fine day." 这个扩展被 **tokenization** 为 [38125, 1110, 13]。这导致最终合并后的序列是 [1212, 318, 257, 220, 38125, 1110, 13]。然而，若对整个字符串 "This is a fine day." 进行联合 **tokenization**，那么得到的将是 [1212, 318, 257, 3734, 1110, 13]，这个结果与模型训练时的数据分布和意图更加一致。

我们的 **tokenization** 过程必须有效地传达意图，因为模型要从对训练文本的贪婪 **tokenization** 中学习。^①在训练过程中引入子词正则化，可以缓解这一问题。子词正则化（**subword regularization**）是指在训练中故意引入次优的 **tokenization** 操作，以增强模型的弹性。因此，在训练过程中，模型会遇到那些并非遵循纯粹贪婪策略产生的 **tokenization**

^① 译注：对训练数据进行 **tokenize** 时，如果专注于每个步骤的最佳预测，就可以说这是一种贪婪算法。这种算法的每一步都专注于做出当前看来最佳的决策，而不考虑全局最优解。第 1 章介绍的 **BPE** 算法就是这样的一种算法。

结果。虽然子词正则化有效增强了模型处理 token 边界的能力，但并未完全消除模型倾向于执行标准贪婪 tokenization 的倾向。

token 修复通过战略性地调整生成过程来预防由 tokenization 不一致而引起的问题。它会在提示结尾前临时回退一个 token（撤销最后生成的 token）。然后，模型会再次生成 token，但这一次它会确保生成的 token 与提示中的最后一个 token 在某种程度上匹配，通常是通过共享相同的前缀。通过这种方式，生成的第一个 token 将保持与最后一个提示 token 相匹配的前缀。这样生成的 token 序列与模型在训练时学习到的模式更加一致，从而避免了因提示边界而造成的任何异常编码。这在处理 URL 的 token 生成时特别有用，因为 URL 的 tokenization 尤为关键。

Acceleration

Acceleration（加速）显著提高了 Guidance 程序内部执行推理的效率。该策略利用了与 LLM 推理器的一个会话状态，使程序能够重用键/值（KV）缓存。采用这个策略，避免了 LLM 必须自主生成所有结构化 token 的需求，速度相较于传统方法得到了提升。

在这一过程中，KV 缓存扮演了关键角色。在 Guidance 的情况下，它们充当着一种“动态存储单元”，保存有关提示及其结构组成部分的关键信息。刚开始的时候，GPT 风格的 LLM 会摄取成簇（cluster）的提示 token 并填充 KV 缓存，从而构建好提示结构的一个表示。此缓存将为后续的 token 生成提供上下文。

随着 Guidance 程序的继续运行，它会智能地利用存储在 KV 缓存中的信息。程序不再仅仅依赖 LLM 从零开始生成每一个 token，而是战略性地使用缓存中现有的信息。这种对缓存数据的重用加速了 token 生成过程，因为从缓存生成 token 比重新生成要快得多，也更高效。

此外，Guidance 程序的模板结构动态影响后续 token 的概率分布，确保生成的输出与模板的对齐达到最优，并且前后一致地进行 tokenization。这种缓存信息、模板结构和 token 生成之间的对齐使 Guidance 实现了对推理过程的加速。



注意：此加速技术目前适用于 transformers 中的本地控制模型，并且默认启用。

结构化输出和基于角色的聊天

Guidance 允许使用特殊的 with 上下文块（context blocks）来控制聊天模型，这些块会将其内部的内容封装成聊天模型所需的具体格式。这样一来，就可以自由地表达聊天程序，而不必局限于单一的后端模型。这些 with 块包括 system、user 和 assistant 等。

通过整合这些块，可以为系统、用户和助手定义各自的角色和职责。对话可以被设置成按照这种方式流动，每个参与者的输入都被包含在相应的块内。

此外，在 `assistant` 块内，可以使用 `gen` 函数（例如，`gen("response")`）来辅助生成动态响应。尽管由于对部分完成的限制，并不支持在 `assistant` 块内部实现复杂的输出结构，但在这个块的外部构建对话结构是完全没有问题的。

下面是摘自官方文档的一个与专家聊天的例子（注意，`llm` 是之前已经创建好的模型实例）。

①

```
# 可以通过使用一系列角色标签来创建和引导多轮对话
@guidance
def experts(lm, query):
    with system():
        lm += "你是一个乐于助人的助手。"

    with user():
        lm += f"""\
        我想要对以下问题的回答：
        {query}
        能出色地回答这个问题的三位世界级专家（过去或现在）都有谁？列出姓名即可，不需要列出更多
        信息。
        请暂时不要回答或评论这个问题。"""

    with assistant():
        lm += gen(name='experts', max_tokens=1500)

    with user():
        lm += f"""\
        很好，现在假定这些专家共同匿名撰写了统一答案来回答这个问题。
        换句话说，他们的身份不会被透露，也不会提及有一组专家在回答这个问题的事实。
        如果专家们的意见有分歧，请在答案本身中将他们的不同立场呈现为备选观点（例如：‘有人可能会
        认为…另一些人可能会认为…’）。
        请在回答开始时写上一个“回答”："""

    with assistant():
        lm += gen(name='answer', max_tokens=1500)

    return lm

llm + experts(query='生命的意义是什么?')
```

在上述代码中，我们定义了一个名为 `experts` 的函数，并借助 `assistant` 块来生成回答。下面是上述代码的示例运行结果，注意这是一次完整的多轮“对话”。

① 译注：这部分的代码可以从以下网址下载：<https://bookzhou.com/2024/06/21/1188/>。

system 你是一个乐于助人的助手。

user 我想要对以下问题的回答：

生命的含义是什么？

能出色地回答这个问题的三位世界级专家（过去或现在）都有谁？列出姓名即可，不需要列出更多信息。请暂时不要回答或评论这个问题。

assistant 好的，以下是对你的问题的回答：

生命的含义是一个复杂的哲学问题，涉及到伦理、宗教、科学等多个领域。因此，对于这个问题的回答也有很多不同的观点和理解。

以下是三位能够出色回答这个问题的世界级专家（过去或现在）的可能人选，仅列出姓名：

1. 赫拉克利特斯 (Heraclitus)
2. 西格蒙德·弗洛伊德 (Sigmund Freud)
3. 阿尔伯特·爱因斯坦 (Albert Einstein)

user 很好，现在假设这些专家共同匿名撰写了解答。

换句话说，他们的身份不会被透露，也不会提及有一组专家在回答这个问题的事实。

如果专家们的意有分歧，请在答案本身中将他们的不同立场呈现为备选观点（例如：“有人可能会认为...另一些人可能会认为...”）。

请在回答开始时写上“回答”：

assistant 回答：

生命的含义是一个复杂的问题，涉及到伦理、宗教、科学等多个领域。我们认为，生命的含义在于个体的自我实现和社会的共同进步。

有人可能会认为，生命的含义在于追求个人的幸福和满足。这种观点认为，每个人都应该追求自己的利益和快乐，不必考虑他人的感受 and 需要。

另一些人可能会认为，生命的含义在于为他人和社会做出贡献。这种观点认为，每个人都应该为社会做出自己的贡献，为他人的福祉和幸福而努力。

我们认为，生命的含义在于个体的自我实现和社会的共同进步。个体应该追求自己的幸福和满足，也应该考虑他人的感受 and 需要。同时，个体也应该为社会做出自己的贡献，为社会的进步和发展做出努力。这样，个体和社会可以相互促进，实现共同的进步和发展。

进行 **one-shot** 聊天（仅一条消息）或者进行 **few-short** 聊天（含几条消息）时，像这样使用 **Guidance** 的效果相当不错。然而，由于缺乏历史记录的构建，所以将其应用于一个更全面的聊天应用时，就需要慎重考虑了。

小结

本章深入探讨了对 AI 应用程序进行了优化的各种语言框架。我们探讨了如何利用这些框架来整合提示模板、链以及代理等概念。我们重点考虑的因素包括内存管理、数据检索以及有效的调试。

本章介绍了三种主要的框架：**LangChain**，它的特点就是开源，提供了统一的工具和接口，

可以非常简单地整合和操作不同的语言模型；**Microsoft Semantic Kernel (SK)**，它的特点是可以智能地组织和调度一系列任务，这些任务可能涉及多个插件或服务，以实现特定的结果。规划器 (**planner**) 在这个过程中起到了核心作用，它们与 AI 模型合作，理解用户的意图并设计出最佳的行动方案。一旦计划被创建，**Semantic Kernel** 就能执行这些步骤，无需额外的人工干预，从而实现自动化的工作流。以及 **Microsoft Guidance**，正如“**Guidance**”这个名称所暗示的，它能“指导”AI 以符合逻辑的方式组织生成和提示，并进行一定程度的逻辑控制。

这些框架不仅帮助开发者更好地组织和控制 AI 模型的行为，而且还促进了更高级的 AI 应用的开发，很好地推动了 AI 技术的发展。

附录 大语言模型内部工作原理揭秘

不同于本书其他章节，本附录探讨的是大语言模型（LLM）的内部工作原理，从较高的层次探讨了它的数学和工程方面。

我们并没有深入探讨像 GPT-3.5 或 GPT-4 这样的专有模型的技术细节，因为这些模型尚未开源。相反，我们关注的是目前广泛已知的内容，依赖于 Llama3 和开源版 GPT-2 等模型。其目的是从幕后角度审视这些复杂的模型，揭开它们卓越性能背后的神秘面纱。

这里提出的许多概念都源自经验观察，并且通常还没有（或者还没有）一个明确的理论基础。然而，这不应该引起惊讶或恐惧。这有点像是最神秘的人体器官：大脑。我们知道它，使用它，并且对它有一定的经验性认识。但是，我们仍然不清楚它为什么会表现出那样的行为方式。

概率的作用

本书多次强调，GPT 和其他一般性的**因果语言建模**（Causal Language Modeling, CLM）的目标是生成连贯且合理的输入文本延续。本节旨在分析这一模糊陈述的含义，并强调概率在其中所扮演的关键角色。事实上，按照克劳德·E·香农（Claude E. Shannon）于 1948 年的定义，语言模型本质上是对词序列（或字符、字、token）进行的概率分布。

启发式方法

“合理性”（reasonableness）的概念在科学领域至关重要，通常与概率的概念紧密相关，我们讨论的情况也不例外。LLM 通过评估特定 token 在给定上下文中出现的概率来决定如何继续输入的文本。

为了评估某个特定 token 跟随待完成文本的合理性，必须考虑其出现的概率有多大。现代语言模型不再仅仅依赖单个词出现的概率，而是依赖于 token 序列的概率，即最初被称为 n-gram 的概率。采取这种方法，我们可以捕获比基于单一词汇分析更复杂、更上下文相关的语言关系。

n-gram

n-gram 出现的概率衡量了特定 token 序列在特定上下文中是最合适选择的可能性。为了计算这个概率，LLM 之前写好的一个海量文本数据集中抽取信息，这个数据集称为**语料库**（corpus）。在语料库中，每个 token 序列（即 n-gram）都被计数并记录下来，使得模型能够确定某些 token 序列在特定上下文中出现的频率。一旦记录下 n-gram 的出现次数，就可以通过将该 n-gram 在特定上下文中出现的次数除以该上下文在语料库中的总出现次数来计

算出该 n-gram 在特定上下文中出现的概率。^①

温度

有了不同 n-gram 的概率之后，如何选择最合适的一个来继续输入的文本呢？直观上，你可能会考虑选择当前上下文中概率最高的 n-gram。然而，仅仅选择概率最高的 n-gram 会导致确定性和重复的行为，这意味着对于相同的提示，模型会始终生成相同的文本。这时，采样技术就可以派上用场了。我们不会确定性地选择概率最高的 token（这种方法称为贪心解码，稍后会详细解释），而是采用随机采样的方式。通过根据出现概率随机选择一个词，我们可以为模型生成的响应引入变化，从而使输出更加多样化和自然。

在采样过程中，对于大语言模型（LLM）而言的一个关键参数是**温度**（temperature）。温度是一个用于控制模型在文本生成过程中的预测随机性的参数。正如物理学中温度调节着由随机粒子运动引起的热振动那样，在 LLM 中，温度则调节着模型在做出决策时的随机程度。温度值的选择并非基于精确的理论基础，而是基于实际经验和效果调整得出。从数学角度来看，温度与 softmax 函数密切相关。softmax 函数负责将预测神经网络的最后一层输出转换为概率分布，即将实数向量转化为概率向量（其中每个元素都是介于 0 和 1 之间的正数，并且所有元素之和为 1）。通过调整温度值，可以改变模型选择下一个词汇的概率分布，从而影响最终生成文本的多样性和创造性。

对于包含元素 z_i 的实数向量 z ，softmax 函数的定义如下：

$$\text{softmax}(z)_i = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

其中 T 是温度。

当 T 较高（例如，大于 1）时，softmax 函数实际上减小了不同选项之间概率的相对差异，使得即使原本概率较低的选择也更有可能会被选中，从而增加了生成文本的多样性。相反，当温度较低（例如，小于 1）时，softmax 函数增强了不同选项之间概率的相对差异，使得原本概率较高的选择更有可能会被选中，从而使后续词的选择更加确定性。一些大语言模型（LLM）允许用户设定一个种子（seed），以确保采样过程的可重复性，这种方式类似于在 C# 或 Python 等编程语言中使用随机数生成器时设置种子的做法。

^① 译注：n-gram 背后的思路在于，像信用（credit）和分（score）这样的词如果在一个句子中紧挨在一起出现，可能比它们相隔很远出现更有意义。如果没有 n-gram，词的相对接近性就会被忽略。使用 n-gram 的缺点在于，它增加了内存消耗和训练时间。然而，只要能合理地使用，它能使文本分类模型更加准确。——摘自《机器学习与人工智能实战：基于业务场景的工程应用》（<https://bookzhou.com>）

更先进的方法

在 1960 年代，人们为了评估“合理性”，会为每种语言构建表格，记录每一个 2、3 或 4-gram 的出现概率。然而，随着序列长度（ n ）的增加，可能的组合数量呈指数级增长，这使得计算所有可能的 n -gram 概率很快就变得不切实际。此外，当时还缺乏足够庞大的文本语料库来涵盖所有可能出现的 n -gram 组合。为了理解这种数量级的增长有多么惊人，我们可以考虑一个包含 4 万个单词的词汇表。在这种情况下，可能的 2-gram 数量为 16 亿，而可能的 3-gram 数量骤增至 60 万亿。因此，哪怕是一个仅包含 20 个单词的短文本片段，其所有可能的组合数量也是完全无法计算的。

认识到直接计算所有 n -gram 的概率是不可行的之后，正如科学中经常发生的那样，我们必须求助于模型。一般来说，在科学领域中，模型允许我们预测结果而无需具体的定量测量。对于大语言模型（LLM）而言，模型使我们能够考虑 n -gram 的概率，即使没有包含目标序列的参考文本。基本上，我们使用神经网络作为这些概率的估计器。这正是 LLM 在非常高的层次上所做的事情。它基于训练数据估算 n -gram 的概率，然后，给定一个输入，返回输出中各种可能的 n -gram 的概率，并以某种方式选择其中一个。具体来说，LLM 通过学习大量文本数据，捕捉语言中的统计模式，并从中推断出哪些 n -gram 序列在给定上下文中是最可能的。

但稍后会讨论到，使用模型本质上意味着进行估算，这带来了一些固有的局限性。理论上，如果能计算出所有可能的 n -gram（ n 足够大）的出现概率，这些局限性可能会得到缓解。然而，即便如此，仍有一些根本性问题难以克服：我们的语言极其复杂，它们基于推理和人类的自由意志，而且并不总是能够确定我们日常对话中的下一个词就是最“可能”的那一个，因为有时候我们就是想要说一些不同的东西。

人工神经元

如前所述，在科学领域中，常见的做法是借助预测模型来估算结果，而不直接进行测量。模型的选择不是基于严格的理论，而是基于对现有数据的分析和经验观察。

一旦选择了解决特定问题的一个最适合的模型，接着就要确定模型参数的值，使模型更加逼近理想的解决方案。在 LLM 的背景下，当模型能生成接近人类产生的输出文本时，我们就认为它是有效的。因此，需要“手动”评估。

本节将探讨 LLM 与人脑功能之间的相似性，还探讨了如何通过训练来选择和调整最常用的一些模型的参数。

人工神经元与人脑神经元

神经网络是自然语言生成和处理的主要模型。这种网络受到人脑生物学的启发，试图利用数字神经元层重建生物神经元之间复杂的信息交换。每个数字神经元都可以从周围神经元

（如果是神经网络第一层，则从外部环境）接收一个或多个输入，这类似于生物神经元可以从周围的其他生物神经元（或者直接从感觉神经元）接收一个或多个信号。生物神经元的输入是一种离子电流。而在数字神经元中，输入始终表示为数值矩阵。

无论神经网络被赋予的具体任务是什么（例如，图像分类或文本处理），被处理的数据都必须表示成数值形式。这种表示发生在两个阶段：首先，文本被映射为代表参考词（token）ID 的数字列表，然后应用一个实际的转换。这个过程称为**嵌入**（embedding）。嵌入过程为数据提供了一个数值表示，其原则是相似的数据应该由几何上接近的向量表示。

以自然语言处理为例，文本嵌入过程包括将文本分解为片段，并创建这些片段的矩阵表示。嵌入过程生成的数字向量可以被视为语言特征空间中某一点的坐标。为了理解文本中所包含的信息，关键不在于单个向量本身，而在于测量两个向量之间的距离。这个距离提供了关于两个文本片段相似性的信息——这对于完成（补全）所提供的输入文本至关重要。

回到一般情况，当数字神经元接收到矩阵输入时，它会执行两个操作，一个是线性的，另一个是非线性的。考虑网络中的一个通用神经元，它有 N 个输入，如下所示：

$$X = \{x_1, x_2, \dots, x_N\} \quad (1)$$

通过一个权重为 w_i 的“加权”连接，每个输入都会到达神经元。神经元通过 K 个权重 w_{ij} 对 N 个输入 x_i 执行一次线性组合。其中， i 是作为输入来源的那个神经元的索引，而 j 是目标神经元的索引。^①

$$W \cdot K + b \quad (2)$$

然后，神经元向该线性组合应用一个通常为非线性的函数，这称为**激活函数**。

$$f(W \cdot K + b) \quad (3)$$

因此，对于图 A-1 中显示为白色空心圆的那个神经元，其输出将是：

$$w_{12}f(x_1w_1 + b_1) + w_{22}f(x_2w_2 + b_2)$$

^① 译注：公式中的 b 是偏置项（bias term）。它是模型的一个重要组成部分，允许模型在拟合数据时更加灵活。偏置项的作用类似于线性代数中的截距，在线性模型中，它可以移动决策边界，使得模型不仅仅通过原点，而是可以根据数据进行调整。例如，在逻辑回归中，如果没有偏置项，决策边界总是经过坐标系的原点，这可能会限制模型的灵活性。又例如，在 ReLU（整流线性单元，Rectified Linear Unit）激活函数中，偏置项可以帮助决定在哪个点函数开始从 0 变为线性增长。

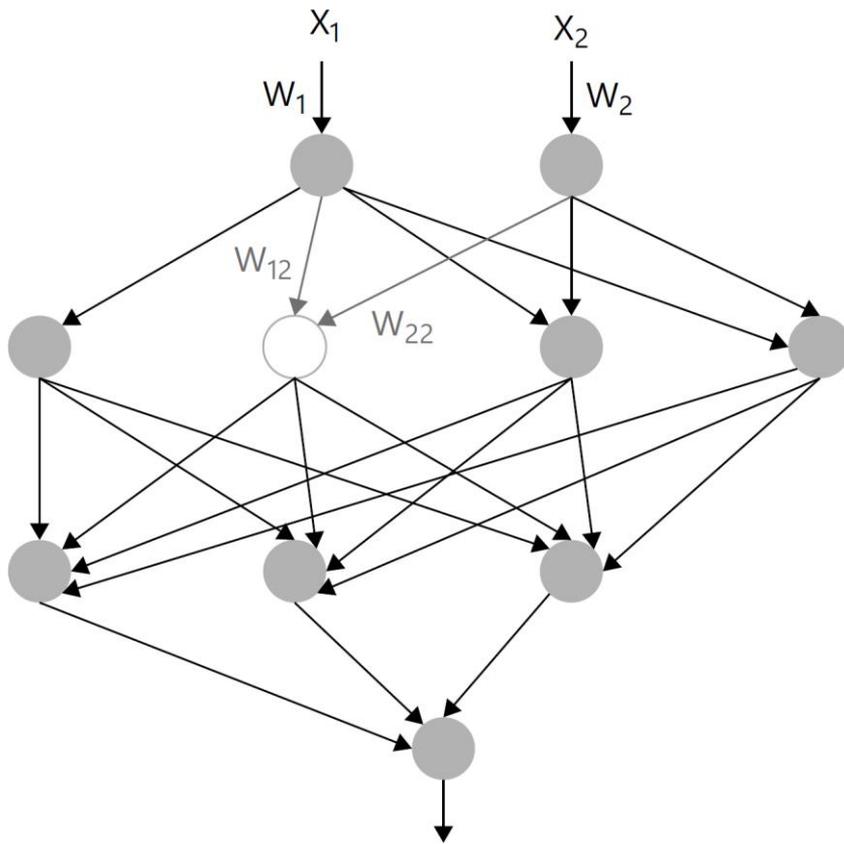


图 A-1 示例神经网络

激活函数通常是同一神经网络中所有神经元所共有的。这个函数对于在神经网络的操作中引入非线性至关重要，这使得网络能够捕捉数据中的复杂关系，并模拟自然语言的非线性行为。最常用的激活函数包括 ReLU、tanh、sigmoid、Mish 和 Swish 等。

这一过程类似于生物神经元内部发生的情况：当接受通过突触传来的电刺激后，细胞体内发生的信号在空间和时间上的整合通常会产生非线性行为。

数字神经元的输出被传递到网络的下一层，这一过程被称为前向传播。下一层继续处理先前产生的输出，并通过上述提到的相同处理机制产生新的输出。如有必要，再传递给下一层。或者，数字神经元的输出也可以作为神经网络的最终输出，这类似于人类神经系统中消息传递至肌肉组织的运动神经元。

现代神经网络的架构虽然基于这一原理，但自然要复杂得多。例如，卷积神经网络（Convolutional Neural Network, CNN）和循环神经网络（Recurrent Neural Network, RNN）

使用了专门的结构化元素。CNN 设计用于处理具有空间结构的数据，例如图像。它们引入了卷积层来识别局部模式。相比之下，RNN 适用于序列数据，例如自然语言。RNN 使用内部记忆来处理当前上下文中先前的信息，使其适用于诸如自然语言识别和自动翻译等任务。

尽管我们尚未完全理解神经网络为何能模仿人类行为，但在图像分类的场景中考察神经网络早期层的输出，我们会发现一些迹象表明其行为类似于人类视觉数据初始阶段的神经处理。

训练策略

上一节指出，数字神经元的输入是对从前一层神经元接收到的信号进行加权线性组合而形成的。然而，尚待澄清的是这些权重是如何确定的。这一任务委托给了神经网络的训练过程，通常通过反向传播算法的一个实现来进行。

神经网络“基于示例”进行学习。然后，根据所学的内容进行泛化（常规化）。在训练阶段，需要向神经网络提供大量示例，并允许其通过调整线性组合的权重来学习，从而使输出尽可能接近人类预期的结果。

在每次训练迭代中，也就是每次向神经网络呈现一个示例时，都会从获得的输出反向计算出一个损失函数（因此得名“反向传播”）。我们的目标是使损失函数最小化。可以在训练阶段自由选择这种函数，但在概念上它总是代表实际结果与预期目标之间的差距。一些常用的损失函数包括交叉熵损失（Cross-Entropy Loss, CEL）和均方误差（Mean Squared Error, MSE）。

交叉熵损失

交叉熵损失（CEL）是在分类和文本生成问题中常用的损失函数。最小化这个误差函数是训练的目标。在文本生成中，问题可以被视为一种分类问题，试图预测字典中每个 token 作为下一个 token 的概率。这个概念类似于传统的分类问题，在这种情况下神经网络的输出是一系列与每个可能类别相关的概率。

交叉熵损失测量预测概率分布与真实概率分布之间的差异。在分类问题的背景下，对于数据集（样本集合） X ，交叉熵损失函数像下面这样表示：

$$CEL(X, T, P) = - \sum_x \sum_{i=1}^N t_i \times \log(p(x)_i)$$

这个公式表示了实际概率与预测概率的负对数之乘积的总和，其中 N 是训练数据集 X 中所有元素的数量， T 是真实的标签集合，而 P 是预测的概率分布。

具体地说， t_i 代表与实际类别 i 相关联的概率（在总共 N 个可能的类别中，其中 N 在文本生

成问题中是词汇表的大小), 它是一个二值变量: 如果元素属于类别 i , 则取值为 1; 否则取值为 0。而 $p(x)_i$ 是神经网络预测元素 x 属于类别 i 的概率。

在交叉熵损失函数中使用对数源于需要对预测误差较大时进行更大的惩罚, 特别是当实际概率接近零时。换句话说, 当模型错误地预测一个非常低概率的类别相比实际类别时, 使用对数放大了误差。这是大语言模型中最常用的误差函数。

均方误差

均方误差 (MSE) 是在回归问题中常用的损失指标, 其目标是预测数值而非类别。但是, 如果应用于分类问题, 那么 MSE 测量预测值与真实值之间的差异, 对误差 (error) 赋予平方权重。在分类情境中, 神经网络的输出将是一系列连续的值, 代表与每个类别相关联的概率。

叉熵损失侧重的是离散分类, 而 MSE 则扩展到了分类问题, 将概率视为连续值。MSE 的公式是通过将每个类别的实际概率与预测概率之差的平方进行求和而给出的, 如下所示:

$$MSE(X, T, P) = \frac{1}{|x|} - \sum_x \sum_{i=1}^N (t_i - (p(x)_i))^2$$

具体地说, t_i 代表与实际类别 i 相关联的概率 (在总共 N 个可能的类别中, 其中 N 在文本生成问题中是词汇表的大小), 它是一个二值变量: 如果元素属于类别 i , 则取值为 1; 否则取值为 0。而 $p(x)_i$ 是神经网络预测元素 x 属于类别 i 的概率。

困惑度损失

另一个重要的评估指标是困惑度损失 (Perplexity Loss, PL), 它衡量了语言模型在预测一系列字符或词汇时的“困惑”程度。较低的困惑度损失表明模型能够以较小的不确定性连续预测出字或词。

$$PL(X, P) = \sum_x e^{\frac{1}{N} \sum_{i=1}^N -\log(p(x)_i)}$$

这一指标通常用于验证阶段, 而非在训练过程中作为损失函数。由于该公式不包含实际的预测目标值, 因此它不能直接用于优化。交叉熵损失和均方误差分别衡量预测分布与真实概率分布之间的差异以及预测值与实际值之间的差距。相反, 困惑度损失提供了一个更加直观的模型复杂性的度量, 代表模型预测下一个字或词时的平均选择数量。简而言之, 困惑度损失评估了大语言模型预测的一致性, 并为我们提供了一种量化模型在文本生成过程中“困惑”程度的方法。



注意：与其他机器学习模型一样，在训练之后，通常还要使用不在训练数据集中的数据执行一次验证，并最终由人类评估者进行各种测试。人类评估者会采用不同的评价标准，例如连贯性、流畅性和正确性等。

优化算法

训练的目标是调整权重以最小化损失函数。数值方法提供了多种途径来最小化损失函数。

最常用的方法是梯度下降法：初始化权重，计算成本函数对于所选权重的梯度，并使用以下公式更新权重。

$$w_{j+1} = w_j + \alpha \nabla J(w_j)$$

其中， j 是当前迭代次数， $j + 1$ 是下一次迭代，而 α 是学习率。注意，梯度是一个向量，它指向成本函数增长最快的方向。由于我们的目标是最小化成本函数，所以需要沿着梯度的相反方向移动。

该过程会持续迭代，直到达到预先设定的停止条件，这要么是一个迭代次数，要么是一个预定义的阈值。

由于损失函数不一定具有唯一的全局最小值，它可能存在多个局部最小值，因此最小化算法有可能收敛到局部最小值，而非全局最小值。在梯度下降方法中，学习率 α 的选择对于算法能否成功收敛至关重要。通常而言，算法是否能够收敛到全局最小值还可能受到神经网络复杂性的影响——具体表现为网络层数和每层神经元数量。一般来说，网络层数越多，可调整的权重参数就越多，从而提高了逼近真实解的能力，同时降低了陷入局部最小值的风险。

然而，在梯度下降方法中，关于学习率的选择以及神经网络结构的设计没有理论规则可循。要获得执行一个特定任务的最佳结构，需要通过实验和经验来确定。

作为“通过实验”来确定的例子，在图像分类任务中，如果发现第一层由两个神经元组成能够产生良好的效果，那么这种结构就是特别好的选择。而“通过经验”确定的一个例子是，有时通过在中间层创建瓶颈（即减少神经元数量），可以在缩小网络规模的同时仍然保持相同的性能。

经典的梯度下降算法有许多具体实现，并且伴随着多种优化方法。其中，广泛使用的Adam算法会根据每个参数的历史梯度自适应地调整学习率。此外，还有几种替代的梯度下降方法。例如，牛顿-拉弗森（Newton-Raphson）算法是一种考虑损失函数曲率的二阶优化方法。尽管其他方法可能较少使用，但仍然有效，例如受生物进化启发的遗传算法和模拟金属退火工艺的模拟退火算法。

训练目标

损失函数应被纳入更广泛的训练目标之中——也就是说，要考虑到希望通过训练过程实现的具体任务。

例如，在某些情况下，例如 BERT 模型，训练的主要目标是生成输入的嵌入表示（嵌入形式）。我们通常采用掩码语言建模（Masked Language Modeling, MLM）方法，即对输入 token 的随机部分进行掩码或移除，而模型的任务是根据上下文（左右两侧）预测被掩码的标记。这种技术促进了对双向上下文关系的深度理解，有助于捕获双向的信息流。在这种情况下，损失函数通过计算模型预测的标记与原始被掩码标记之间的交叉熵损失来进行优化。

其他策略（例如，区间破坏和重建）通过引入受控的破坏来训练模型重构原始序列。这种方法的主要目标是隐式教导模型理解上下文，并生成更丰富的语言表示。为了预测缺失的字词或重构不完整的序列，模型必须深入理解输入内容。

在直接面向文本生成的目标模型中（例如，GPT），我们通常采用自回归方法，尤其是连续语言模型（Continuous Language Model, CLM）。在这种方法中，模型逐步生成 token 序列，捕捉数据中的顺序依赖性，并隐式地学习语言模式、语法结构和语义关系。这与双向方法形成对比，后者会同时考虑序列中左右两侧的信息。

在 CLM 训练中，例如，如果序列是“太阳从东方升起”，那么模型必须学会根据“太阳从东方”预测下一个词“升起”。这种方法以自我监督的方式自然发展，因为任何文本都可以自动用于训练模型，无需额外标注。这使得寻找训练数据相对容易，因为任何有意义的句子都可以用于训练。

在这个背景下，损失函数是训练过程中的关键抽象层之一。一旦确定了目标，就会选择一个相应的损失函数，然后开始训练。



注意：如第 1 章所述，在大语言模型的上下文中，这个训练阶段通常称为预训练，它是更长过程的第一步，后面还有监督微调阶段和强化学习步骤。

训练限制

随着层数的增加，神经网络处理更复杂任务和建模数据中复杂关系的能力也随之增强，并且在一定程度上展现出类似“记忆”的能力，这使得它们能够更好地泛化到新数据上。这种能力对于解决许多现实世界的问题非常重要。通用逼近定理（universal approximation theorem）甚至声称，任何函数都可以被一个具有足够多神经元的神经网络在某个空间区域内任意良好地逼近。

神经网络作为“逼近器”，似乎无所不能。但实际上并非如此。在自然界中（包括在涉及到人类的本性时），存在一些无法简化的难题。首先，模型可能会出现过拟合的现象，这意味着它们可能会过于适应特定的训练数据，从而损害其泛化能力。此外，自然语言本身的复杂性和模糊性也使得大语言模型难以始终如一地生成准确且无歧义的解释。事实上，有些句子具有多义性和模糊性，这对人类和模型来说都是一个挑战。

尽管这些模型能够生成连贯的文本，但它们可能缺乏真正的理解。它们并不具备意识、意向性或者世界观。换言之，不能说这些模型真的就“懂”了。虽然它们能够构建一些映射来表示周围的世界，但这些映射并不是显式的知识图谱或本体论。相比之下，人类从很小的时候就开始学习和发展对世界的认识。

从更哲学的角度来看，我们必须先搞清楚本体论（ontology）的确切含义是什么。能很好地回答一个问题就算吗？如果没有完全理解其含义，也不会推理，那么或许不能算。但是，理解含义和推理又意味着什么呢？能够进行推理并不意味着一定能正确地做出推断。^①另外，学习到底是什么意思，无论是从数学的角度还是更广泛的意义上？从数学角度来看，对于这些模型而言，学习意味着利用数据中存在的规律进行压缩，但仍然受到克劳德·香农的源编码定理的限制。从更广泛的意义上说，学习意味着什么？什么时候可以说一样东西已经被“学会”了？

有一点可以肯定：当前的 LLM 缺乏规划能力。它们在生成结果时“思考”的是下一个 token，而我们人类说话时思考的是下一个概念和想要表达的想法。实际上，全球主要的研究实验室正在开发将 LLM 集成到更复杂的智能体（代理）中。这种智能体首先懂得如何规划行动和策略，并且是从概念层面开始思考，然后再细化到 token。随着新的强化学习算法的不断推出，这一点很有可能成为现实。

总之，LLM 目前的广泛使用和迅猛发展引发了人们对语言、知识、伦理以及真理本质的深刻思考，这些思考促使科学界内外展开了激烈的哲学与伦理辩论。我们应当正视客观真理这一概念。由于 LLM 的学习基于主观数据，并在训练过程中反映出数据集（语料库）中的文化偏见，因此从根本上讲，任何模型都无法真正实现客观性。

GPT 的情况

本节重点介绍 GPT 的结构和训练过程。根据公开信息，GPT 目前是一种拥有至少 1750 亿个参数的神经网络^②，因其采用了 transformer 架构而特别适合处理语言相关的任务。

^① 译注：即使会推理，也不一定能做出正确的推断。这可能源于错误的前提、逻辑谬误或信息不足。例如，“因为所有鸟都会飞，所以企鹅会飞”就是一个错误的推断，因为前提是错的，并非所有鸟都会飞。

^② 译注：这是 GPT-3 的参数数量。GPT-4 的参数数量已经达到了 1.8 万亿。

transformer 与注意力机制

我们知道，GPT 的任务是根据训练阶段所学习的数十亿篇文本合理地继续（补全）所输入的文本。接下来，我们将深入探讨 GPT 的结构，以理解它是如何能够生成接近人类水平的结果的。

GPT 的工作流程可以分为三个关键步骤。

1. **创建嵌入向量**：GPT 基于一个包含大约 50 000 个可能 token 的字典，将输入文本转换为一系列 token，然后将这些 token 转换为嵌入向量。这些嵌入向量以数值形式表示文本的特征。
2. **处理嵌入向量**：这些嵌入向量经过处理，生成一个包含更复杂信息的新向量。^①
3. **概率生成**：GPT 计算每个 token（在 50 000 个 token 中的每一个）作为下一个生成 token 的概率。

这一过程以自回归的方式重复进行，每次都将在新生成的 token 作为输入，直到生成一个特殊的 token——序列结束（end-of-sequence, EOS），它表示生成过程的结束。

所有这些操作都由神经网络来执行。因此，除了神经网络的结构外，没有其他工程或外部控制。除了这些之外，一切都由学习过程来引导。在这个阶段，没有外部传递的本体论或显式信息，也没有使用强化学习系统。

嵌入

在 GPT 结构中，给定包含 N 个输入 token 的一个向量（其中，N 小于等于上下文窗口的大小，对于 GPT-3 来说，N 大约在 4 000 到 16 000 个 token 之间），这个向量首先会通过嵌入模块。

在该模块内部，长度为 N 的输入向量将经过两条并行路径进行处理：

- **标准嵌入**：在第一条路径中（称为标准嵌入，即 canonical embedding），每个 token 都被转换为一个数值向量（GPT-2 中的大小为 768，GPT-3 中的大小则为 12 288）。该路径捕捉词与词之间的语义关系，使模型能够理解每个 token 的意义。然而，在传统的嵌入中，并不包含 token 在序列中的位置信息。因此，它能够表示词义，但不会考虑它们的具体顺序。在此路径中，嵌入网络的权重是可训练的。
- **位置嵌入**：在第二条路径中（称为位置嵌入，即 positional embedding），嵌入向量是根据 token 的位置生成的。这种方法使模型能够理解和捕捉词在文本中的顺序。在自

^① 译注：注意，GPT 不是直接从原始嵌入向量生成一个新向量，而是通过多层 transformer 编码器逐步更新这些嵌入向量，以获得更复杂的表示。最后，它基于序列末尾的向量计算出下一个 token 的概率分布。

回归语言模型中，token 序列对于生成后续文本至关重要，词的顺序提供了重要的上下文和意义信息。由于 GPT 这样的基于 transformer 的模型不是循环神经网络，所以它们将输入视为 token 集合，而没有明确的顺序表示。通过引入位置嵌入，模型得以加入与每个 token 在输入中的位置有关的信息。在这一过程中，位置嵌入是固定的，没有可训练的权重；模型学会如何使用这些固定的位置嵌入，但不能修改它们。

接着，从这两条路径中获得的嵌入向量将被相加，生成最终的输出嵌入，随后传递给下一步，如图 A-2 所示。

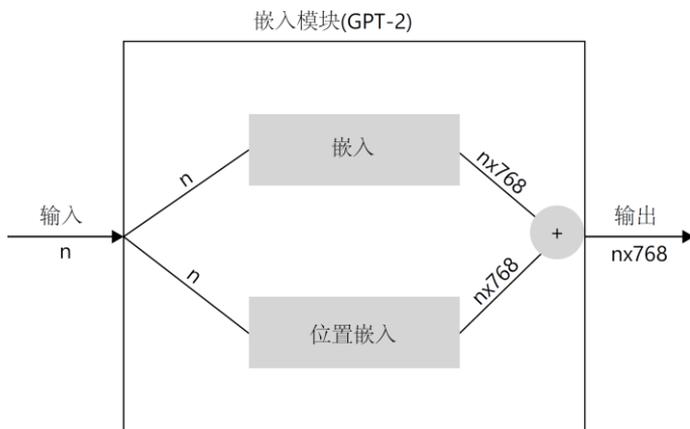


图 A-2 GPT-2 嵌入模块示意图

位置嵌入

最简单的位置嵌入方法是使用简单的序号（即 0, 1, 2, ...）。然而，对于长序列，这会导致过大的索引值。此外，对于变长序列，在 0 到 1 之间进行归一化也颇具挑战性，因为不同的序列会被不同地归一化。因此，我们需要一种不同的方案。

类似于传统的嵌入，位置嵌入层在 GPT-2 中生成大小为 $N \times 768$ 的矩阵。对于第 n 个 token，该矩阵中每个位置的值根据以下规则计算：

- 在偶数列 $2i$ 处的值为：

$$P(n, 2i) = \sin\left(\frac{n}{10000 \frac{2i}{768}}\right)$$

- 在奇数列 $2i + 1$ 处的值为：

$$P(n, 2i + 1) = \cos\left(\frac{n}{10000 \frac{2i}{768}}\right)$$

其实并非一定要选择 10 000 这个数字。但是，人们最终选择这个数字，是出于历史上的原

因。它最早由阿西什·瓦斯瓦尼（Ashish Vaswani）等人在 2017 年 12 月发表的论文“Attention Is All You Need”中。这些公式为嵌入中的每个 token 位置分配了正弦和余弦值。

使用正弦波函数，模型可以捕捉周期性关系。此外，由于值的范围在-1 到 1 之间，所以无需额外的归一化。因此，给定一个表示 token 位置的整数序列（简单的 0, 1, 2, 3, ...），就会生成一个大小为 Nx768 的正弦和余弦值矩阵。

以一个嵌入大小为 4 的例子，结果是如下所示的矩阵：

| 词 | 位置 | i = 0 | i = 0 | i = 1 | i = 1 |
|-------|----|--------------------------|---------------------------|---------------------------|---------------------------|
| I | 0 | $\text{Sin}(0) = 0$ | $\text{Cos}(0) = 1$ | $\text{Sin}(0) = 0$ | $\text{Cos}(0) = 1$ |
| am | 1 | $\text{Sin}(2/1) = 0.84$ | $\text{Cos}(2/1) = 0.54$ | $\text{Sin}(2/10) = 0.10$ | $\text{Cos}(1/10) = 0.54$ |
| Frank | 2 | $\text{Sin}(3/1) = 0.91$ | $\text{Cos}(3/1) = -0.42$ | $\text{Sin}(3/10) = 0.20$ | $\text{Cos}(2/10) = 0.98$ |



注意：至于结果为什么如此，其实并没有一个理论上的推导。这只是一种经验法则。

自注意力

经过嵌入模块之后，我们来到了 GPT 的核心架构：transformer，即一系列所谓的“多头注意力”（Multi-Head Attention）块。为了大致了解“注意力”（Attention）意味着什么，我们可以考虑这样一个句子：“GPT 是由 OpenAI 创建的语言模型。”当考虑句子的主语“GPT”时，离它最近的两个词是“是”和“由”。然而，这两个词并不能为当前上下文提供太多有用信息。相比之下，“模型”和“语言”虽然在物理位置上远离句子的主语，却能帮助我们更好地理解上下文。用于处理图像的卷积神经网络仅考虑物理上最接近的词。相反，得益于注意力机制，transformer 能够关注那些在意义上和上下文中更为相关的词。

首先，我们来了解一下常规意义上的“注意力”机制（更确切地说，是“自注意力”机制）。由于单个 token 的嵌入向量本身并不携带有关句子上下文的信息，因此需要一种方式来衡量上下文的重要性和相关性（即为不同的上下文分配权重）。这就是“自注意力”机制的作用所在。



注意：注意区分“注意力”机制和“自注意力”机制。“注意力”机制允许模型

有选择地聚焦于输入序列的不同部分，这些部分被称为查询、键和值。“自注意力”机制则特指将同一个序列作为计算查询、键和值的基础，从而使模型能够捕获序列内部各元素间的相互关系。

“自注意力”不是仅仅考虑单个 token 的嵌入。相反，它会执行以下略显“复杂”的操作：

1. 考虑单个 token 的嵌入向量（该向量先乘以一个称为 M_Q 的查询矩阵）与句子中其他 token 的嵌入向量（每个都先乘以一个称为 M_K 的键矩阵）的点积。例如，假定现在有包含 N 个 token 的一个句子，并专注于其中的第一个 token，那么它的嵌入向量将与句子中其余 $N - 1$ 个 token 的嵌入向量（在 GPT-2 的情况下均为 768 维）进行标量乘法。点积输出两向量之间的角度，因此可以认为这是第一种相似度量。
2. 通过这些向量对的点积，我们得到了 N 个权重（其中一个权重是通过将第一个 token 与其自身进行标量乘法而获得的）。这些权重随后通过 Softmax 函数进行归一化，使它们的总和为 1。每个权重都表示与输入中的其他词/字相比，相应的“键 token”为给定的“查询 token”提供了多少有用的上下文信息。
3. 然后，所得的权重乘以输入 token 的初始嵌入向量（该向量先乘以一个称为 M_V 的值矩阵）。这个操作相当于通过其余 $N - 1$ 个 token 的嵌入向量来加权第一个 token 的嵌入向量。

对于句子中的所有 token 都会重复这个过程，即对每个 token 进行加权，并从其他 token 获得上下文信息。然后，得到的加权向量与初始嵌入向量相加，并发送给一个全连接前馈（feed-forward）网络。查询、键和值的矩阵在所有 token 之间共享，并且它们的值是在训练过程中学到的。



注意：最后一步是将初始嵌入向量与加权后的向量相加，这通常称为残差连接（residual connection）。在这种连接方式中，初始输入序列直接与块的输出相加，然后传递给后续层。

自注意力机制

更加技术性的解释是，注意力机制涉及到三个主要组成部分：查询（Q）、键（K）以及值（V）。这些实际上是输入向量的线性投影。具体来说，输入向量与三个对应的权重矩阵相乘，以此来得到查询（Q）、键（K）和值（V）这三个向量。



注意：这里所说的 Q、K 和 V 都是向量，而不是矩阵，因为考虑的是单个输入

token 的处理流程。但是，如果将所有 N 个输入标记一起作为一个 $N \times 768$ 维的矩阵，那么 Q 、 K 和 V 就变成 $N \times 768$ 维的矩阵。后者是为了利用优化的矩阵乘法算法来执行并行计算。

使用这种记法，各组成部分如下所示：

- **查询 (Q)**：需要为其计算注意力权重的当前位置或元素。这是输入序列（输入向量）的线性投影，捕捉了当前位置的信息。
- **键 (K)**：整个输入序列。这也是输入的线性投影，包含了序列中所有位置的信息。
- **值 (V)**：与输入序列中每个位置相关的内容或特征。和 Q 和 K 一样， V 也是通过线性投影得到的。

注意力机制通过测量查询和键之间的兼容性（或相似性）来计算注意力分数。这些分数随后用于对值进行加权，从而创建一个加权和。得到的加权和是对注意力进行处理后的表示，它强调了输入序列中与当前位置最相关的部分。这一过程通常被称为缩放点积注意力（scaled dot-product attention），数学上可以这样表示：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{768}}\right)V$$

其中， T 表示转置，经过 Softmax 处理后的权重与 V 进行逐元素乘法。

多头注意力和其他细节

前面描述的注意力机制可以被认为应用于句子中每个字/词的一个“头”（head），其目的是获取该字/词更“上下文化”（contextualized）的嵌入。为了同时捕捉多种上下文化模式，注意力机制会使用不同的查询、键和值权重矩阵来重新运行多次。这些权重矩阵在训练开始时通常被随机初始化。每一次使用不同权重矩阵的运行被称为一个“transformer 头”或“注意力头”，并且这些头通常并行运行以提高计算效率。

所有头最终产生的“上下文化”的嵌入会被串联起来，形成一个综合嵌入，并用于后续计算。查询、键和值矩阵的维度需要特别选择，目的是使串联后的嵌入与原始向量的大小匹配。这是一种被广泛使用的技术。在最初的 transformer 论文中，作者使用了 8 个头。

为了获得更多的上下文信息，经常会以级联（层叠）的方式使用多层注意力块（transformer 层）。具体而言，GPT-2 包含 12 个注意力块，而 GPT-3 包含 96 个注意力块。从这个数据就可以理解，GPT-3 的规模远大于 GPT-2，有更多的参数和更深的网络结构，旨在捕捉更复杂的语言模式。

经过所有注意块之后，输出成为一组新的嵌入向量。从这里开始，可以添加更多的前馈网

络，并使用 Softmax 将最后一个前馈网络的输出转换为概率列表，从中选择概率最高的 token 来作为下一个生成的 token。

总之，GPT 采用的是前馈神经网络架构，这与传统的循环神经网络（RNN）有所不同。2014 年，Bahdanau 等人在他们的研究中首次将注意力机制应用于 RNN。GPT 的架构包括全连接层以及一些更为复杂的设计元素，其中神经元仅与前一层或后一层中的特定神经元相连。

训练与新兴技术

上一节解释了 ChatGPT 在接收到输入文本时如何生成合理的延续，本节将重点介绍它内部的 1750 亿个参数^①是如何确定的，还会解释 GPT 是如何被训练的。

训练

如前所述，训练神经网络的过程包括提供输入数据（在这种情况下是文本），通过反向传播算法调整参数，以最小化误差（损失函数）。为了实现 GPT 的惊人效果，必须在训练过程中向其提供大量文本数据，这些数据主要来源于互联网（包括由人撰写的万亿级网页以及超过 500 万本数字化书籍）。训练过程中，部分文本数据可能被多次使用，而另一些数据则可能只在少数训练轮次中使用。

目前，虽然没有明确的理论可以精确指导神经网络的大小或训练所需的数据量，但研究人员通常依据经验和实验进行估算。值得注意的是，GPT 架构中的参数数量（约 1750 亿）与训练过程中使用的 token 总量在数量级上是相当的。这表明了模型在训练数据中进行编码存储的某种方式，并暗示了一个较低的数据压缩率。

尽管现代 GPU 能够并行执行大量运算，但权重更新必须按批次顺序进行。如果参数数量为 n ，所需的训练 token 也具有大致相同的数量级，那么每训练一次，计算成本将是 n^2 数量级的，这使神经网络的训练成为一个极为耗时的过程。在这方面，人脑的解剖结构具有显著优势，因为它不像现代计算机那样将记忆与计算分离。

在对这一庞大语料库进行训练后，GPT 似乎能够生成较好的结果。然而，尤其是在处理长文本时，生成的内容依然存在一些人工痕迹，人类可以察觉到其不自然之处。因此，研究人员决定不仅仅局限于被动地学习数据，而是在初始训练阶段之后，进行第二阶段的监督微调（Supervised Fine-Tuning, SFT），并在此基础上加入一个基于人类反馈的强化学习（Reinforcement Learning From Human Feedback, RLHF）阶段，以进一步提升模型的表现。这一过程已在第 1 章进行了详细描述。

^① 译注：这是 GPT-3 的参数数量。GPT-4 的参数数量已经达到了 1.8 万亿。

解码策略与推理

一旦拥有了能够以自回归方式生成下一个 token 的模型，就需要将其投入生产并执行推理。为此，必须选择一种解码策略。也就是说，给定可能输出 token 的概率分布，必须选择一个 token 并将其结果呈现给用户，直到遇到序列结束（EOS）token 为止。

我们主要使用三种解码策略来选择 token：

- **贪婪搜索 (greedy search)**：这是最简单的方法。在每一步中，它都会选择当前概率最高的 token。然而，这种方法可能会生成重复且不太合理的序列。尽管它在第一步选择了最有可能的 token，但这可能会影响后续 token 的生成，从而导致整个序列的整体合理性和连贯性下降。更好的模型在使用贪婪搜索时通常会较少出现重复，并且展现出更强的连贯性。
- **束搜索 (beam search)**：这种方法解决了重复性问题。它同时考虑多个假设（即候选序列），并在每一步扩展它们，最终选择概率最高的序列。束搜索大多数时候能找到比贪婪搜索更高概率的输出序列，但不能保证找到最有可能的输出，因为它只考虑有限数量的假设。虽然仍有可能出现重复，但可以通过引入 n-gram 惩罚来缓解这个问题，即对生成过程中重复出现的 n 个连续 token 施加惩罚。
- **采样 (Sampling)**：采样的目的是为生成过程引入随机性，它基于条件概率分布来选择要输出的下一个字/词。虽然增加了输出的多样性，但也可能导致不一致的输出。有两种子变体来改善连贯性：
 - **Top-K 采样**：首先从所有可能的候选字/词中选择概率最高的 K 个字/词，然后仅在这 K 个字/词之间重新分配概率质量。这种做法有助于避免选择那些概率较低的字/词，但同时也可能会限制生成内容的创造性。
 - **Top-p 采样**：Top-p 采样（也称为核采样）则更加灵活，它基于累积概率阈值 p 来动态选择一组字/词。具体来说，它会选择那些累积概率达到或超过 p 的最高概率字/词集合。这种方法使得所选字/词集合的大小变得更为灵活，既能保证一定的多样性，又能保持较高的连贯性。



注意：结束 token（EOS）在编码和解码阶段都指示序列的结束。在编码过程中，它像其他任何 token 一样被嵌入到向量空间中。在解码过程中，它作为停止条件，指示模型停止生成更多的 token，并指示序列的完成。

综上所述，解码策略对于生成文本的质量与连贯性具有重大影响。GPT 通过采用采样方法将随机性引入到生成过程中，从而使模型的输出更加多样化。相较于更确定性的方法（例如，贪婪搜索和束搜索），采样技术让 GPT 能够生成更为丰富且多样的文本。

新兴技术

您可能认为要教会神经网络新知识，就必须重新对其进行训练。然而事实并非如此。相反，一旦网络经过了训练，通常仅需提供一个适当的提示就能生成合理的后续文本。我们可以通过直观的方式来理解这一点，无需纠结于训练期间所提供的 **token** 的具体含义。在这种情况下，可以将训练视为一个阶段，在此阶段模型推断关于语言结构的信息，并将它们组织在一个语言空间内。从这个角度来看，当给已训练好的网络提供一个新的提示时，网络只需在语言空间中追踪相应的轨迹，而无需进一步的训练。

但要记住的是，只有在“可处理”的问题范围内才能取得令人满意的结果。当碰到一个问题时，如果系统无法即刻 (*on the fly*) 从数据中提取出重复出现的结构来进行学习，或者无法依据已有的经验来生成类似于人类的响应，就需要借助未经训练的外部计算工具来辅助。一个典型的例子就是数学运算，大语言模型在数学上的表现很差，因为它们实际上并不了解如何执行计算，而是只会在所有可能的 **token** 中寻找最合理的那个。^①

^① 译注：正是由于这个原因，所以在 2024 年 7 月，发生了许多 LLM 认为 9.11 比 9.8 更大的事件。