



Semantic Kernel

入门手册

AK

介绍



随着大模型兴起，人工智能进入 2.0 时代。开源社区有许多基于大模型应用的框架和解决方案，其中 Semantic Kernel 更适合传统工程以及多语言体系的工程团队使用；LangChain 适合数据科学员进行使用，而 BenToML 适合多模型部署场景。本手册主要介绍 .NET 和 Python 两个版本的 Semantic Kernel，并结合 Azure OpenAI Service 给需要掌握大模型应用开发的各位进行指导。

目录

前言：了解大型语言模型	1
第一章：用 SDK 访问 Azure OpenAI Service	9
第二章：Semantic Kernel 基础	17
第三章：开启大模型的技能之门 - Plugins	22
第四章：Planner 组件 - 让大模型有规划地工作	29
第五章：嵌入式的技巧 - Embedding	32

前言：了解大型语言模型

从上世纪 50 年代开始，人类对人工智能就开始探索。人工智能给人的印象一直是一个划时代的技术，需要有专业技能的数据科学领域从业人员才能使用。但从 2022 年年底开始，人工智能的领域发生了重大改变。OpenAI 发布 GPT-3 模型开始，人工智能不再是单一领域或者说单一场景的解决方案。全新的多模态大型语言模型模型改变了游戏规则，无论你从事什么样的工作，都可以用自然语言和大型语言模型模型进行交流，大型语言模型模型反馈给你的是生成式的内容，这就包括文本，图片，视频等，大大增强了可用性。在进入 Semantic Kernel 的内容前，希望在前言部分和大家说说与大型语言模型模型相关的故事，让各位可以更好地理解大型语言模型模型。

大型语言模型基础

大型语言模型模型（英文：Large Language Model，缩写LLM），也称大型语言模型，是一种人工智能模型，旨在理解和生成人类语言。它们在大量的文本数据上进行训练，可以执行广泛的任務，包括文本总结、翻译、情感分析等等。大型语言模型模型的特点是规模庞大，包含数十亿的参数，帮助它们学习语言数据中的复杂模式。这些模型通常基于深度学习架构以及 Transformer 算法。大型语言模型模型的训练方式是通过自我监督学习，即通过预测序列中的下一个词或标记，为输入的数据生成自己的标签，并给出之前的词。训练过程包括两个主要步骤：预训练和微调。在预训练阶段，模型从一个巨大的、多样化的数据集中学习，通常包含来自不同来源的数十亿词汇，如网站、书籍和文章。在微调阶段，模型在与目标任务或领域相关的更具体、更小的数据集上进一步训练，这有助于模型微调其理解，并适应任务的特殊要求。现在很多公司都在开发大型语言模型，为世人所熟知的就包括 OpenAI 的 GPT-X 和 DALL-E-X 系列，以及 Meta 的 Llama, Google 的 Gemini，以及百度的问心一言等。OpenAI 中的 GPT-4 是现阶段最好的大型语言模型，有非常大的优势。但随着时间推移，也有不少好的行业垂直领域模型诞生。

Transformer 算法

Transformer 算法是一种基于自注意力机制的深度学习模型，它可以用于处理自然语言和其他序列数据。它由编码器和解码器两部分组成，每部分包含多个层，每层又包含多头自注意力和前馈神经网络。Transformer 算法的优点是可以并行处理整个序列，捕捉长距离的依赖关系，而不需要使用循环神经网络或卷积神经网络。Transformer 算法最初是在论文《Attention Is All You Need》¹中提出的，用于机器翻译任务。后来，它被广泛应用于其他自然语言处理任务，如文本摘要、问答、语音识别等。一些著名的基于 Transformer 的模型有 BERT、GPT-3 / 4、T5 等。

本次课程主要围绕 Azure OpenAI Services 的 OpenAI 3/3.5/4 以及 DALL-E-3 展开。至于其他的模型会在日后更多的进阶内容上提及，大家可以关注我的 GitHub Repo。

OpenAI 以及相关模型介绍

虽然 Transformer 的算法来自 Google，但真正让大型语言模型进入公众视野的是 OpenAI。OpenAI 是一个人工智能研究和部署的公司，它的使命是确保人工通用智能（AGI）能够造福全人类。它的愿景是创造一个能够与人类合作和竞争的 AGI，同时遵循人类的价值观和道德。OpenAI 最初是一个非营利性的组织，由一些科技界的知名人士，如埃隆·马斯克、彼得·蒂尔、杨致远等创立于 2015 年¹。它的目标是推进数字智能的发展，使其能够最大程度地惠及人类，而不受金钱利益的束缚。它的研究是开放和透明的，任何人都可以访问和使用 OpenAI 在人工智能领域有着开创性的研究，尤其是在生成模型和安全性方面。它开发了一些强大的大语言模型，如 GPT-3/3.5/4、ChatGPT、DALL-E、Whisper 等，它们可以理解 and 生成文本、图像、声音等多种形式的數據。它也致力于探索 AGI 的潜在风险和影响，以及如何使其与人类的目标和利益保持一致。

GPT 模型

GPT (Generative Pre-trained Transformer, 生成预训练变换器) 是一种基于深度学习的自然语言处理 (NLP) 模型, 由OpenAI开发。GPT系列模型以其能力强大和灵活性而闻名, 在多种语言任务中表现出色。下面是GPT模型的一些关键特点和发展历程:

GPT的核心特点

1. 基于Transformer架构: GPT模型基于Transformer架构, 这是一种特别适合处理序列数据 (如文本) 的深度学习模型。
2. 大规模预训练: GPT通过在大量文本数据上进行预训练来学习语言的通用模式和结构。这包括书籍、网页、新闻文章等各种来源的文本。
3. 微调应用: 预训练后, GPT模型可以针对特定任务进行微调 (fine-tuning), 如问答、文本生成、翻译等。
4. 上下文敏感: GPT模型能够理解和生成与上下文相关的文本, 这使得它在生成连贯和相关内容方面表现突出。

GPT 发展历程

GPT-1: 第一个版本, 展示了大规模未标记数据预训练的潜力, 以及在多种任务上微调的有效性。

GPT-2: 增大了模型规模和训练数据量, 显著提高了文本生成的质量和准确性。GPT-2因其能够生成连贯且有时难以区分于人类编写的文本而引起广泛关注。

GPT-3: 进一步扩大了模型规模, 达到了前所未有的1750亿个参数。GPT-3在多项NLP任务上取得了革命性的表现, 尤其是在可以少量或无需微调的情况下。

GPT-4及以后: 随着技术的不断发展, 后续的GPT模型可能会在模型规模、理解能力、多模态能力等方面继续进步。

GPT 应用领域 GPT 模型在众多领域都有广泛应用, 包括但不限于:

文本生成: 如文章撰写、创意写作、代码生成等。

聊天机器人: 提供流畅的对话体验。

自然语言理解: 如情感分析、文本分类等。

翻译和多语言任务: 自动翻译不同语言。

知识提取和问答: 从大量文本中提取信息, 回答特定问题。

总的来说, GPT模型代表了当前人工智能和自然语言处理领域的一个重要里程碑, 它的强大能力和多样的应用前景持续引领着技术发展的潮流。

变革者 - GPT-3

GPT-3 是一种大型语言模型, 由 OpenAI 开发, 可以理解和生成自然语言。它是目前最大的大型语言模型之一, 拥有 1750 亿个参数, 可以完成文本摘要、机器翻译、对话系统、代码生成等工作。GPT-3 的特点是通过简单的文本提示, 即“少示例学习” (few-shot learning), 来适应不同的任务和领域, 而不需要额外

的微调或标注数据。GPT-3 打开了潘多拉魔盒，改变了行业规则。GPT-3 已经被应用于许多产品和服务中，如 OpenAI API、OpenAI Codex、早期的 GitHub Copilot 等，它们可以让开发者、创作者和学者更容易地使用和学习人工智能。GPT-3 也引发了一些关于人工智能的伦理、社会 and 安全的讨论和思考，如人工智能的偏见、可解释性、责任、影响等。

进化 - GPT-3.5 和 ChatGPT

GPT-3.5 和 ChatGPT 都是基于 GPT-3 架构的大语言模型，它们可以理解和生成自然语言。它们都有 1750 亿个参数，可以在多种语言处理任务上表现出惊人的能力，如文本摘要、机器翻译、对话系统、代码生成等。

GPT-3.5 和 ChatGPT 的主要区别在于它们的范围和目的。GPT-3.5 是一种通用语言模型，可以处理各种语言处理任务。另一方面，ChatGPT 是一个专用模型，专为聊天应用程序设计。它强调了与用户的互动和沟通，可以扮演不同的角色，如猫娘、明星、政治家等。它也可以根据用户的输入，生成图像、音乐、视频等多媒体内容。

GPT-3.5 和 ChatGPT 的另一个区别在于它们的训练数据和训练方式。GPT-3.5 是在 570 GB 的文本数据上进行了预训练，这些数据来自不同的来源，如网站、书籍、文章等。它的训练方式是通过自我监督学习，即通过预测序列中的下一个词或标记，为输入的数据生成自己的标签，并给出之前的词。ChatGPT 则是在 GPT-3.5 的基础上，使用了更多的对话数据，如社交媒体、聊天记录、电影剧本等，进行了进一步的微调。它的训练方式是通过多任务学习，即同时优化多个目标，如语言模型、对话生成、情感分类、图像生成等。

绝对领导者 - GPT-4

GPT-4（第四代生成预训练转换器）是由OpenAI开发的最新一代人工智能语言模型。它是GPT-3的继任者，拥有更先进和精细的功能。下面是 GPT-4 的一些主要特点：

更大的知识库和数据处理能力：GPT-4 可以处理更大量的数据，它的知识库比GPT-3更加广泛和深入。

更高的语言理解和生成能力：GPT-4 在理解和生成自然语言方面有显著提升，能更准确地理解复杂的语言结构和含义。

多模态能力：GPT-4 不仅可以处理文本，还可以理解和生成图像，提供多模态的交互体验。

更好的上下文理解：GPT-4 能更好地理解和维持长篇对话中的上下文，提供更连贯和一致的回答。

提高的安全性和可靠性：OpenAI在 GPT-4 中加强了对不当内容的过滤和控制，以提供更安全可靠的用户体验。

广泛的应用领域：GPT-4 可应用于各种领域，包括但不限于聊天机器人、内容创作、教育辅助、语言翻译、数据分析等。

总的来说，GPT-4 在其前代模型的基础上做出了显著的改进和提升，能够提供更加强大和多样化的功能。GPT-4 在现阶段有绝对的领导地位，也是很多公司大模型的追赶目标。

进击！睁开眼睛看世界 - GPT-4V

GPT-4V 全称是GPT-4 with Vision，它可以理解图片，为用户解析图片并回答图片相关的问题。GPT-4V可以准确理解图像的内容，识别图像中物体、计算物体的数量、提供图片相关的洞察和信息、提取文本等。可以说，GPT-4V 是大型语言模型的皇者，也让大型模型更好地理解世界。GPT-4V 的视觉主要能力和应用方向

物体检测：GPT-4V 能够识别并检测图像中的各种常见物体，例如汽车、动物和家居用品等。其识别能力已在标准图像数据集上进行了评估。

文本识别：此模型具备光学字符识别（OCR）技术，可在图像中发现打印或手写文字，并将其转换为机器可读的文本。这项功能在文档、标志和标题等图像中得到了验证。

人脸识别：GPT-4V 能够找出并识别图像中的人脸。它还具有一定程度的能力，可以通过面部特征来判断性别、年龄和种族属性。该模型的面部分析能力已在 FairFace 和 LFW 等数据集上进行了测试。

验证码解决方案：GPT-4V 在解决基于文本和图像的验证码时展示了视觉推理能力。这表明模型具有高级的解谜技巧。

地理定位：GPT-4V 能够识别风景图片中所呈现的城市或地理位置。这说明模型掌握了关于现实世界的知识，但也意味着存在泄露隐私的风险。

复杂图像：处理复杂的科学图表、医学扫描或具有多个重叠文本组件的图像时，该模型表现较差。它无法把握上下文细节。

DALL·E 模型

DALL·E 是由 OpenAI 开发的一个先进的人工智能程序，专门用于生成图像。它是一个基于 GPT-3 架构的神经网络模型，但与 GPT-3 主要处理文本不同，DALL·E 的专长在于根据文本描述生成相应的图像。这个模型的名称是对著名艺术家萨尔瓦多·达利（Salvador Dalí）和流行动画角色沃利（WALL·E）的致敬。

DALL·E的关键特点 文本到图像的转换：DALL·E能根据用户提供的文本描述生成图像。这些描述可以非常具体或富有创造性，模型会尽力生成符合描述的图像。

创造性和灵活性：DALL·E在生成图像时展现出惊人的创造性，能够组合不同的概念和元素，创建出独特和创新的视觉作品。

多样性和细节：该模型能生成多种风格和类型的图像，并能处理复杂的、具有细节的描述。

应用潜力：DALL·E在艺术创作、广告、设计等多个领域都有广泛的应用潜力。

DALL·E 应用场景包括

艺术创作：艺术家和设计师可以使用DALL·E来探索新的创意和视觉表现。

广告和媒体：生成符合特定主题或概念的图像。

教育和娱乐：用于教学材料的制作或创造娱乐内容。

研究和探索：探索人工智能在视觉艺术领域的可能性。

DALL·E 的出现标志着人工智能在创造性任务中的一个重要进步，显示了 AI在视觉艺术领域的巨大潜力。现在最新的 DALL·E 模型是 DALL·E 3。

Whisper 模型

Whisper 是由OpenAI开发的一种先进的自动语音识别（ASR）模型。这个模型专注于转录语音为文本，且在多种语言和不同的环境中都表现出了卓越的性能。以下是关于Whisper模型的一些关键特点：

特点 多语言支持：Whisper模型能够处理多种不同的语言和方言，使其在全球范围内具有广泛的适用性。

高精度识别：它能准确地识别和转录语音，即使在背景噪音较多的环境中也能保持较高的准确率。

自适应不同语境：Whisper不仅能识别标准的语音输入，还能适应各种口语化和非正式的对话风格。

易于集成和使用：作为一个机器学习模型，Whisper可以集成到各种应用和服务中，提供语音识别功能。

应用

自动字幕和转录：为视频和音频内容自动生成字幕或文本。

语音助手和聊天机器人：提高语音助手和聊天机器人对语音指令的识别能力。

无障碍服务：帮助有听力障碍的人士更好地理解音频内容。

会议和讲座记录：自动记录和转录会议或讲座的内容。

总体来说，Whisper模型代表了自动语音识别领域的一个重要进步，其多语言和高精度识别能力使其在各种应用场景中都极具价值。

微软与 OpenAI



微软和OpenAI之间的合作关系是当代人工智能领域的一个重要发展。自从OpenAI成立以来，微软就一直是其重要的合作伙伴和支持者。以下是他们合作的一些关键方面和影响：

投资和合作

资金支持：微软在OpenAI的早期就进行了重大投资，包括数亿美元的资金。这些投资帮助OpenAI发展其研究项目和技术。

云计算资源：微软向OpenAI提供了其Azure云计算平台的资源，这对于训练和运行大型AI模型，如GPT和DALL·E系列模型，至关重要。

技术合作

共同研发：两家公司在多个AI项目和技术上展开了合作，共同推动人工智能的发展。

产品集成：OpenAI的某些技术，如GPT-3，已被集成到微软的产品和服务中，例如Microsoft Azure和其它企业级解决方案。

战略合作

可持续和安全的AI：双方致力于发展既可持续又安全的AI技术，重视AI伦理和安全性问题。

拓展AI应用：通过合作，两家公司致力于将AI技术应用到更广泛的领域，例如健康保健、教育和环境保护等。

影响

加速AI技术的发展：这种合作促进了人工智能技术的快速发展和创新。

商业应用和服务：微软通过将OpenAI的技术应用于其产品和服务，推动了人工智能在商业领域的广泛应用。

推动AI民主化：这种合作有助于使更多的企业和开发者能够访问和使用先进的AI技术。

总体来说，微软与OpenAI之间的合作是技术创新和商业应用相结合的典范，这种合作对人工智能技术的发展和普及产生了深远的影响。随着双方合作的不断深化，可以预期他们将继续在人工智能领域发挥重要作用。

Azure OpenAI Service

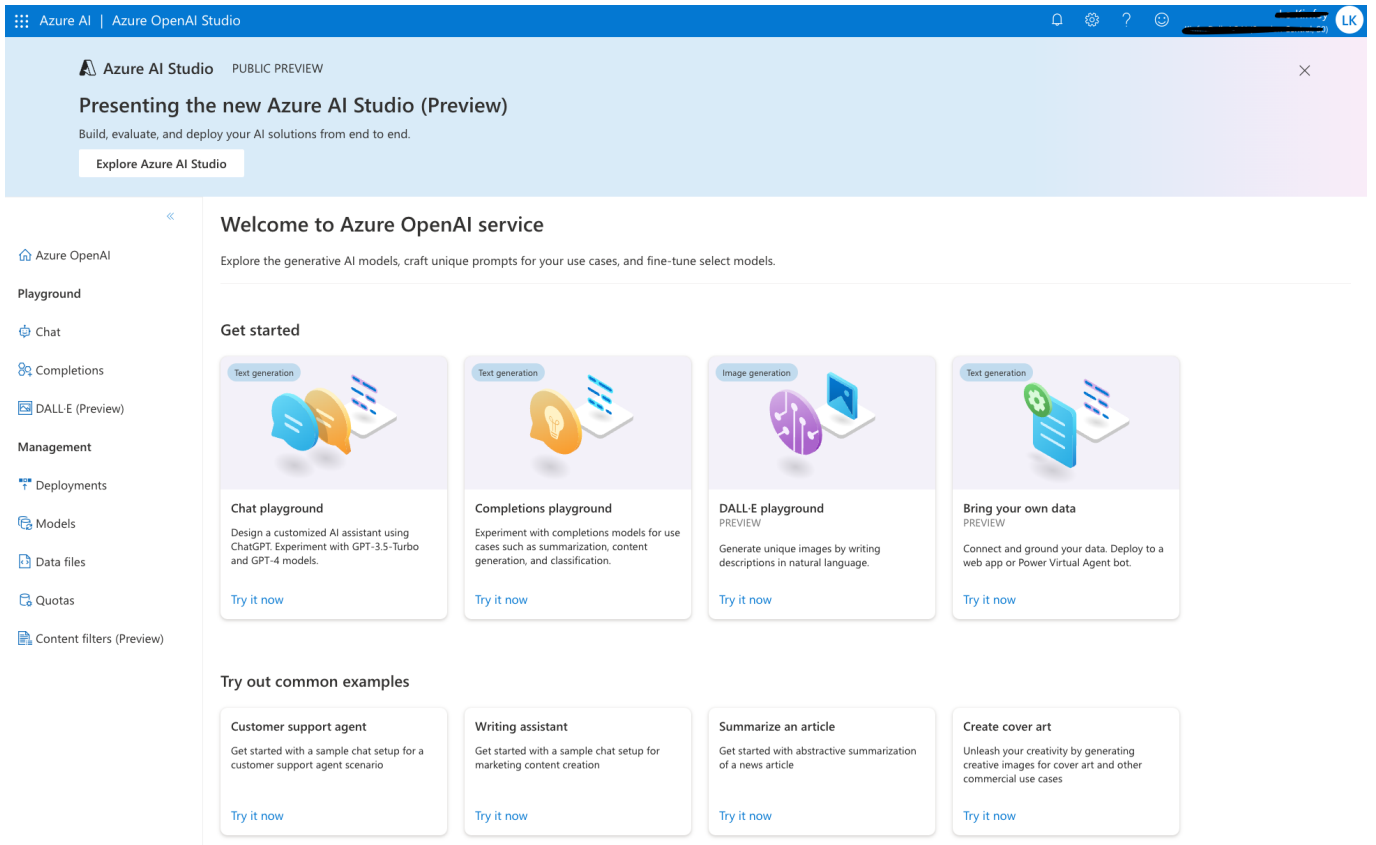
Azure OpenAI Service 是 Microsoft Azure 与领先的人工智能研究组织 OpenAI 之间的合作。 Azure OpenAI Service 是一个基于云的平台，使开发人员和数据科学家能够快速轻松地构建和部署人工智能模型。 借助 Azure OpenAI，用户可以访问各种 AI 工具和技术来创建智能应用程序，包括自然语言处理、计算机视觉和深度学习。 Azure OpenAI Service 旨在加速 AI 应用程序的开发，使用户能够专注于创建为其组织和客户创造价值的创新解决方案。

Azure OpenAI Service 提供对 OpenAI 强大语言模型的 REST API 访问，这些模型包括 GPT-4、GPT-4 Turbo with Vision、GPT-3.5-Turbo 和嵌入模型系列。 此外，新的 GPT-4 和 GPT-3.5-Turbo 模型系列现已正式发布。 这些模型可以轻松适应特定的任务，包括但不限于内容生成、汇总、图像理解、语义搜索和自然语言到代码的转换。 用户可以通过 REST API、Python SDK 或 Azure OpenAI Studio 中基于 Web 的界面访问该服务。

使用 Azure OpenAI Service 你需要有 Azure 账号，然后通过该链接进行申请，等待 1-3 工作日即可使用 Azure OpenAI Service。

Azure OpenAI Studio

我们可以通过 Azure OpenAI Studio 管理我们的模型，以及在 Playground 中测试我们的模型



注意： 所有例子都基于 Azure OpenAI Services

Hugging Face

Hugging Face 是一家专注于自然语言处理（NLP）的人工智能研究公司，以其开源项目和在NLP领域的创新而闻名。公司成立于2016年，其总部位于纽约，但它在全球范围内有影响力和活动。

主要贡献和产品

Transformers库： Hugging Face 最著名的贡献是其开发的“Transformers”库，这是一个广泛使用的Python库，包含了多种预训练的NLP模型，如BERT、GPT、T5等。这个库使得访问和使用这些复杂模型变得更加容易，对促进NLP领域的研究和应用有重大影响。

模型共享和社区： Hugging Face建立了一个强大的社区，促进了研究者和开发者之间的模型和知识共享。通过其平台，任何人都可以上传、分享和使用预训练模型。

研究和合作： Hugging Face在人工智能领域进行积极的研究，与学术界和工业界的众多团队合作。

教育和资源： Hugging Face还提供各种教育资源，包括教程、文档和研究论文，帮助人们更好地了解和使用NLP技术。

影响

技术创新： Hugging Face在推动NLP领域的技术创新方面发挥了重要作用，特别是在预训练模型的发展和应用方面。

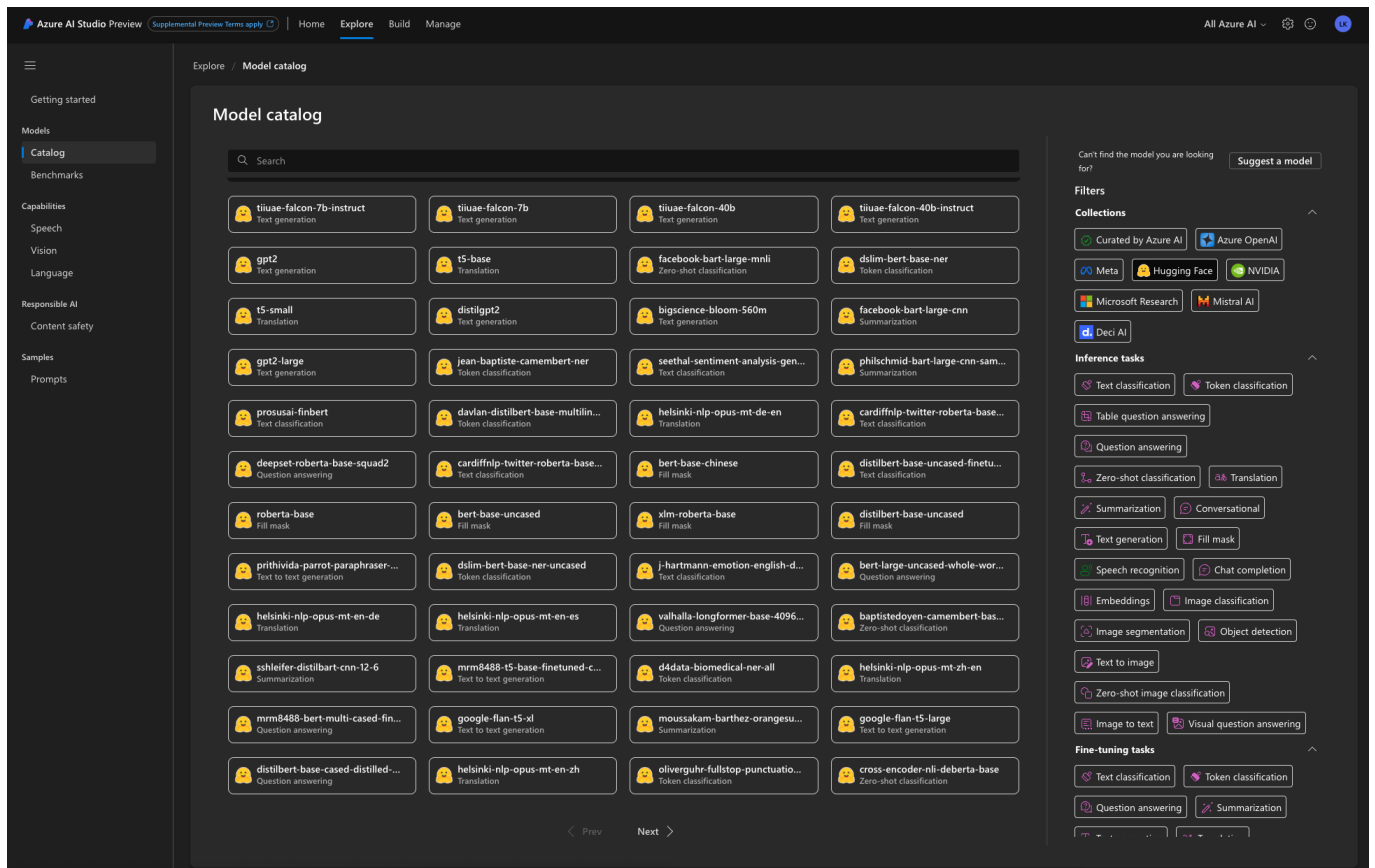
降低技术门槛： 通过提供易于使用的工具和资源，Hugging Face 降低了在NLP领域工作的技术门槛，使得更多的研究者和开发者能够参与到这一领域中。

社区建设：其强大的社区和开源文化促进了知识共享和协作，加速了 NLP 技术的发展和 innovation。

虽然 Hugging Face 最初是作为一个面向消费者的聊天机器人应用开始的，但它迅速转变为专注于提供 NLP 技术和资源的公司。现在，它不仅支持研究和教育，还为企业提供商业解决方案，如定制模型训练、数据分析和机器学习咨询服务。

综上所述，Hugging Face 是 NLP 领域的一个关键参与者，其开源精神和对社区的贡献在促进人工智能技术的民主化和创新方面发挥了重要作用。

Azure AI Studio 也支持 Hugging Face 模型引入，可以让企业更好地结合业务场景使用不同模型解决不同应用场景的问题。



小结

本章为大家介绍了现阶段和大型语言模型相关的知识，特别在 OpenAI, Microsoft, Hugging face 主流大型语言模型平台的相关知识，以及不同模型的应用场景和性能。对于应用场景来说，我们不可能只使用一个模型。在 AI 2.0 时代，我们需要有不同模型的支持，完成更多的智能化应用场景。无论在云端和本地，大型语言模型的应用场景都是未来几年所关注的热点。作为初学者你需要做的是了解不同模型，结合实际场景来完成应用搭建。

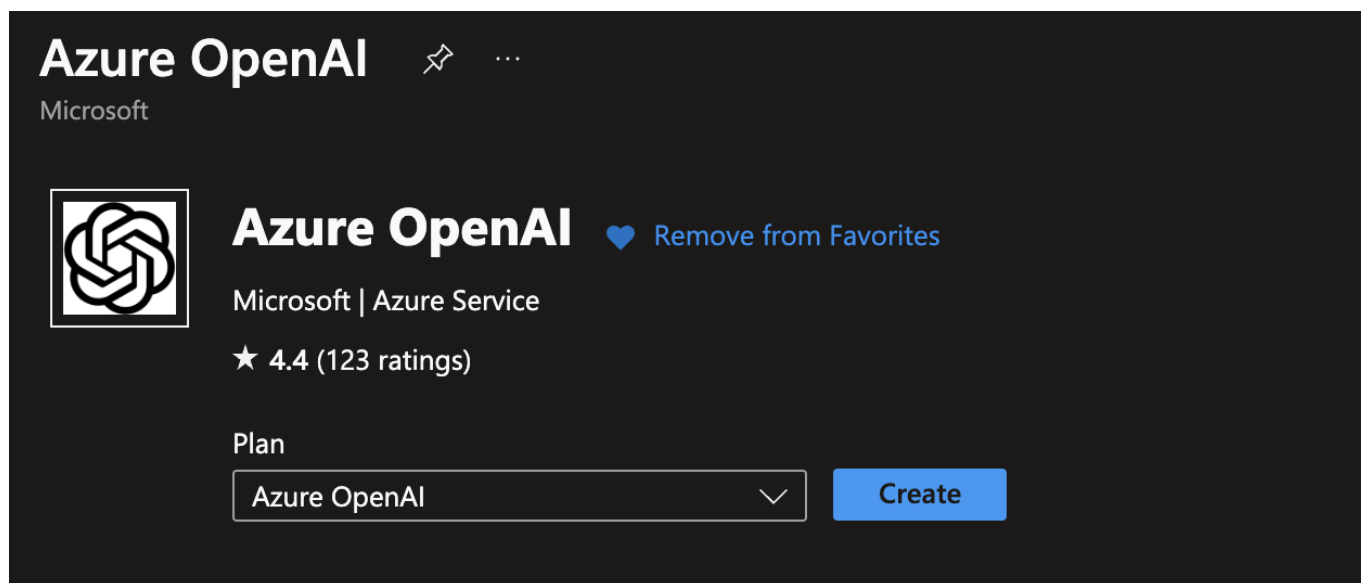
第一章：用 SDK 访问 Azure OpenAI Service

在前言部分，我们了解了大型语言模型的相关知识，下面我想谈谈如何使用大型语义模型，在未进入 Semantic Kernel 之前，我更希望大家看看如何正确通过 SDK 去访问 Azure OpenAI Service 上的 Azure OpenAI 模型。

准备工作：在 Azure OpenAI Studio 部署模型

部署 Azure OpenAI 模型很简单，在申请成功 Azure OpenAI Service 后，通过在 Azure Portal 创建资源进行部署。以下是相关步骤：

1. 在 [Azure Portal](#) 选择 Azure OpenAI 创建资源



选择 'Create' 后，配置好 Azure OpenAI 所在区域，需要注意：因为资源分布不同，不同区域所拥有的 OpenAI 模型不尽相同，在使用前一定要了解清楚。

1 Basics2 Network3 Tags4 Review + submit

Enable new business solutions with OpenAI's language generation capabilities powered by GPT-3 models. These models have been pretrained with trillions of words and can easily adapt to your scenario with a few short examples provided at inference. Apply them to numerous scenarios, from summarization to content and code generation.

[Learn more](#)

Project Details

Subscription * ⓘ

Visual Studio Enterprise Subscription

Resource group * ⓘ

AIGroup

[Create new](#)

Instance Details

Region ⓘ

West US

Name * ⓘ

LukAOAI

Pricing tier * ⓘ

Standard S0

[View full pricing details](#)


Content review policy

To detect and mitigate harmful use of the Azure OpenAI Service, Microsoft logs the content you send to the

Previous

Next

等待片刻，即创建成功



Microsoft.CognitiveServicesOpenAI-20231230-1234567890 | Overview






Deployment


Overview


Inputs

Outputs

Template

 Delete  Cancel  Redeploy  Download  Refresh

 Your deployment is complete



Deployment name : Microsoft.CognitiveServicesOpenAI-20231230-1234567890

Start time : 12/30/2023, 9:02:24 AM

Subscription : Visual Studio Enterprise Subscription

Correlation ID : 7f3b1c1d-1234-5678-9012-345678901234


Resource group : AIGroup

> Deployment details

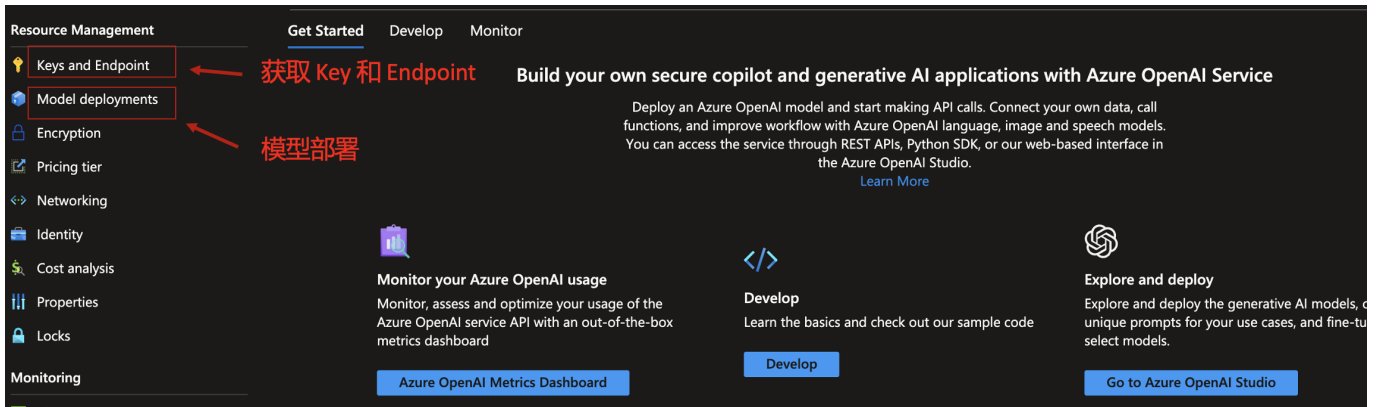
> Next steps

Go to resource

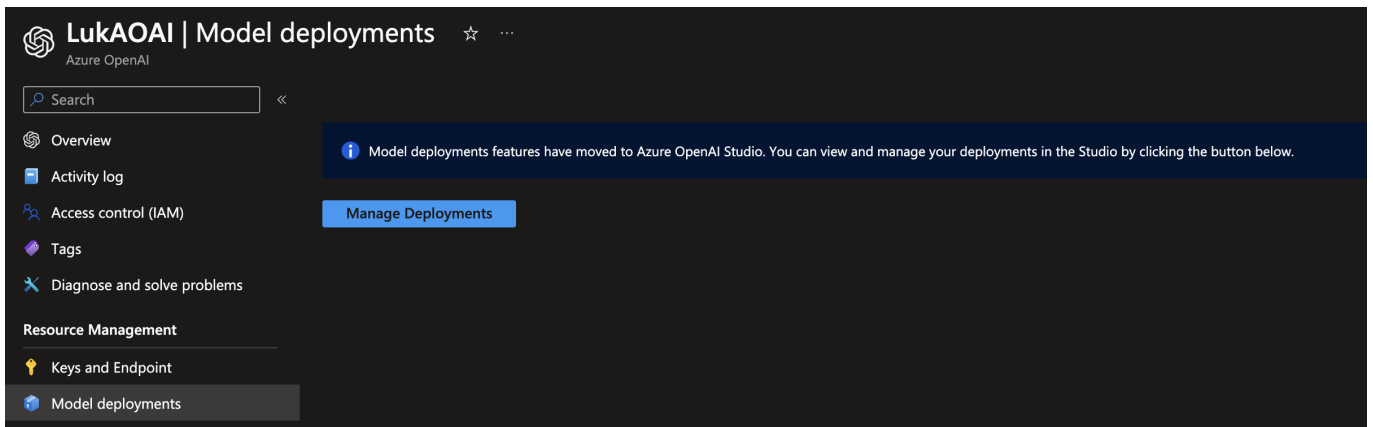
Give feedback

 Tell us about your experience with deployment

2. 进入创建好的资源，你可以部署模型，以及获取 SDK 调用时需要的 Key，以及 Endpoint



3. 进入 'Model Deployment' 选择 'Management Deployment' 进入 Azure OpenAI Studio



4. 在 Azure OpenAI Studio 部署您的模型

[Azure AI Studio](#) > Deployments

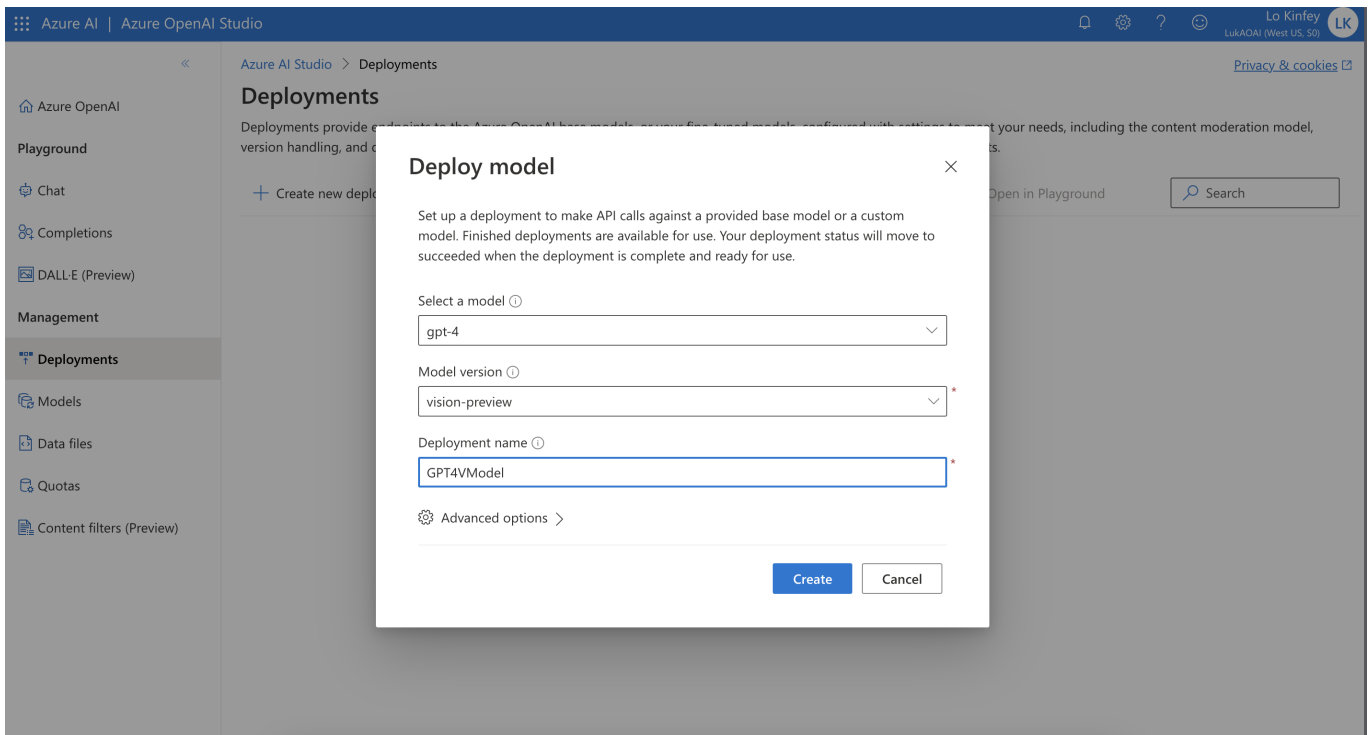
Deployments

Deployments provide endpoints to the Azure OpenAI base models, or your fine-tuned models, configured with settings to meet your needs, including the core version handling, and deployment size. From this page, you can view your deployments, edit them, and create new deployments.



Click here to create deployment

选择你需要的模型



可以看到您的模型列表

[Azure AI Studio](#) > [Deployments](#)

[Privacy & cookies](#)

Deployments

Deployments provide endpoints to the Azure OpenAI base models, or your fine-tuned models, configured with settings to meet your needs, including the content moderation model, version handling, and deployment size. From this page, you can view your deployments, edit them, and create new deployments.

Deployment name	Model name	M...	Deploye...	Capacity	Status	Model dep...	Content Fil...	Rate limit (...)
GPT4Model	gpt-4-32k	0613	Standard	30K TPM	✓ Succeeded	7/5/2024	Default	30000
EmbeddingModel	text-embedding-ada-002	2	Standard	120K TPM	✓ Succeeded	4/3/2025	Default	120000
GPT3	gpt-35-turbo	0613	Standard	120K TPM	✓ Succeeded	6/13/2024	Default	120000
GPT3Model	gpt-35-turbo-16k	0613	Standard	120K TPM	✓ Succeeded	6/13/2024	Default	120000
GPT3TurboModel	gpt-35-turbo	1106	Standard	120K TPM	✓ Succeeded	6/13/2024	Default	120000
GPT4TurboModel	gpt-4	1106-Pre	Standard	10K TPM	✓ Succeeded	3/31/2024	Default	10000

恭喜你，成功部署了模型，接下来就可以用 SDK 对接了。

使用 SDK 链接 Azure OpenAI Service

和 Azure OpenAI Service 对接的 SDK，针对 Python 版本有 OpenAI 发布的 SDK，针对 .NET 也有 Microsoft 发布的 SDK。作为初学者建议都在 Notebook 环境下使用，以便更容易理解执行的关键步骤。

关于 Python SDK

OpenAI 发布的官方 Python SDK, 支持链接 OpenAI 和 Azure OpenAI Service。现在 OpenAI SDK 发布了 1.x 版本，但在市面上很多都在用 0.2x 版本。**本次课程内容都会基于 OpenAI SDK 1.x 版本，并使用 Python 3.10.x。**

安装方式

```
! pip install openai -U
```

关于 .NET SDK

Microsoft 发布基于 Azure OpenAI Service 的 SDK，你可以通过 Nuget 获取最新的包来完成 .NET 生成式 AI 应用。本次课程内容都会基于 **.NET 8** 以及最新的 **Azure.AI.OpenAI SDK** 来展示例子，当然也会使用 **Polyglot Notebook** 作为环境

安装方式

```
#r "nuget: Azure.AI.OpenAI, *-*"
```

在上面我们已经配置好基于 .NET / Python 的 SDK 环境，接下来我们需要创建好链接的类以完成相关的初始化工作

针对 .NET 环境的初识化

```
string endpoint = "Your Azure OpenAI Service Endpoint";  
string key = "Your Azure OpenAI Service Key";  
  
OpenAIClient client = new(new Uri(endpoint), new AzureKeyCredential(key));
```

针对 Python 环境的初始化

```
client = AzureOpenAI(  
    azure_endpoint = 'Your Azure OpenAI Service Endpoint',  
    api_key='Your Azure OpenAI Service Key',  
    api_version="Your Azure OpenAI API version"  
)
```

Azure OpenAI Service 的常用 API 以及 SDK 调用方法

1. 文本补全 API

这是基于 gpt-35-turbo-instruct 模型，对于文本补全是重要的接口

.NET 场景下的文本补全

```

CompletionsOptions completionsOptions = new()
{
    DeploymentName = "gpt-35-turbo-instruct",
    Prompts = { "Can you introduce what is generative AI ?" },
};

Response<Completions> completionsResponse =
client.GetCompletions(completionsOptions);

string completion = completionsResponse.Value.Choices[0].Text;

```

Python 场景下的文本补全

```

start_phrase = 'Can you introduce what is generative AI ?'

response = openai.Completion.create(engine=deployment_name,
prompt=start_phrase, max_tokens=1000)

text = response['choices'][0]['text'].replace('\n', '').replace(' .',
'.').strip()

```

2. Chat API

这是基于 gpt-35-turbo 和 gpt-4 模型，基于聊天场景的接口

.NET 场景下的 Chat

```

var chatCompletionsOptions = new ChatCompletionsOptions()
{
    DeploymentName = "gpt-4",
    Messages =
    {
        new ChatRequestSystemMessage("You are my coding assistant."),
        new ChatRequestUserMessage("Can you tell me how to write python
flask application?"),
    },
    MaxTokens = 10000
};

Response<ChatCompletions> response =

```

```
client.GetChatCompletions(chatCompletionsOptions);
```

Python 场景下的 Chat

```
response = client.chat.completions.create(
    model="gpt-35-turbo", # model = "deployment_name".
    messages=[
        {"role": "system", "content": "You are my coding assistant."},
        {"role": "user", "content": "Can you tell me how to write python
flask application?"}
    ]
)

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

3. 文生图 API

基于 DALL-E 3 模型，文生图的场景

.NET 场景下的文生图

```
Response imageGenerations = await client.GetImageGenerationsAsync(
    new ImageGenerationOptions()
    {
        DeploymentName = "Your Azure OpenAI Service Dall-E 3 model
Deployment Name",
        Prompt = "Chinese New Year picture for the Year of the
Dragon",
        Size = ImageSize.Size1024x1024,
    });
```

Python 场景下的文生图

```
result = client.images.generate(
    model="dalle3",
    prompt="Chinese New Year picture for the Year of the Dragon",
    n=1
)
```

```
json_response = json.loads(result.model_dump_json())
```

4. Embeddings API

基于 text-embedding-ada-002 模型，基于向量转换的实现

.NET 场景下的 Embeddings

```
EmbeddingsOptions embeddingOptions = new()
{
    DeploymentName = "text-embedding-ada-002",
    Input = { "KinfeY is Microsoft Cloud Advocate" },
};

var returnValue = openAIClient.GetEmbeddings(embeddingOptions);

foreach (float item in returnValue.Value.Data[0].Embedding.ToArray())
{
    Console.WriteLine(item);
}
```

Python 场景下的 Embeddings

```
client.embeddings.create(input = ['KinfeY is Microsoft Cloud Advocate'],
    model='text-embedding-ada-002 model').data[0].embedding
```

例子

以下列出了和上述接口相关的例子，请根据您的语言环境进行学习

Python 例子 请[点击访问这里](#)

.NET 例子 请[点击访问这里](#)

小结

我们用最原始，也是最基础的 SDK 与 Azure OpenAI Service 打交道，这也是我们面向生成式人工智能编程的第一步，在没有使用框架下可以更快速地理解不同接口，也为我们进入 Semantic Kernel 打下基础。

第二章：Semantic Kernel 基础

我们已经和大家讲述了 LLMs 以及利用 .NET 和 Python SDK 链接 Azure OpenAI Service 的方法。接下来我们就将进入 Semantic Kernel 的世界。在 2023 年我们有非常多的基于 LLMs 的框架诞生，我们为什么要选择 Semantic Kernel？Semantic Kernel 的优缺点是什么？还有作为传统开发者如何使用 Semantic Kernel？我都会在本章内容中和大家一一细说。

什么是 Semantic Kernel

Semantic Kernel 是一个轻量级的开源框架，通过 Semantic Kernel 您可以快速使用不同编程语言 (C#/Python/Java) 结合 LLMs (OpenAI、Azure OpenAI、Hugging Face 等模型) 构建智能应用。在我们进入生成式人工智能后，人机对话的方式有了很大的改变，我们用自然语言就可以完成与机器的对话，门槛降低了非常多。结合提示工程和大型语言模型，我们可以用更低的成本完成不同的业务。但如何把提示工程以及大模型引入到工程上？我们就需要一个像 Semantic Kernel 的框架作为开启智能大门的基础。在 2023 年 5 月，微软 CTO Kevin Scott 就提出了 Copilot Stack 的概念，人工智能编排就是核心。Semantic Kernel 具备和 LLMs 以及各种提示工程/软件组成的插件组合的能力，因此也被看作 Copilot Stack 的最佳实践。通过 Semantic Kernel，你可以非常方便地构建基于 Copilot Stack 的解决方案，而且对于传统工程，也可以无缝对接。

Semantic Kernel vs LangChain

我们没法不去作出一些客观的比较，毕竟 LangChain 拥有更多的使用群体。无可否认在落地场景上，LangChain 现在比 Semantic Kernel 更多，特别在入门参考例子上。我们来个更为全面的比较：

LangChain 基于 Python 和 Javascript 的开源框架，包含了众多预制组件，开发者可以无需多写提示工程就可以完成智能应用的开发。特别在复杂的应用场景，开发人员可以快速整合多个预定义的组件来综合完成。在开发角度，更适合具备数据科学或则人工智能基础的开发人员。

Semantic Kernel 您可以基于 C#, Python, Java 的开源框架。更大的优势在工程化。毕竟它更像一个编程范式，传统开发人员可以很快掌握该框架进行应用开发，而且可以更好结合自定义的插件和提示工程完成企业的定制化业务智能化工作。

两者有很多共通点，都还在版本迭代，我们需要基于团队结构，技术栈，应用场景作出选择。

Semantic Kernel 的特点

1. **强大的插件** - 你可以通过结合自定义/预定义的插件解决智能业务的问题。让传统的代码和智能插件一起工作灵活地接入到应用场景，简化传统应用向智能化转型的过程。
2. **多模型支持** - 为您的智能应用配置“大脑”，可以是来自 Azure OpenAI Service，也可以是 OpenAI，以及来自 Hugging Face 上的各种离线模型。通过链接器你可以快速接入不同的“大脑”，让您的应用更智能更聪明。
3. **各式各样的链接器** - 链接器除了链接“大脑”外，还可以链接如向量数据库，各种商业软件，不同的业务中间件，让更多的业务场景进入智能成为可能
4. **开发便捷** - 简单易用，开发人员零成本入门

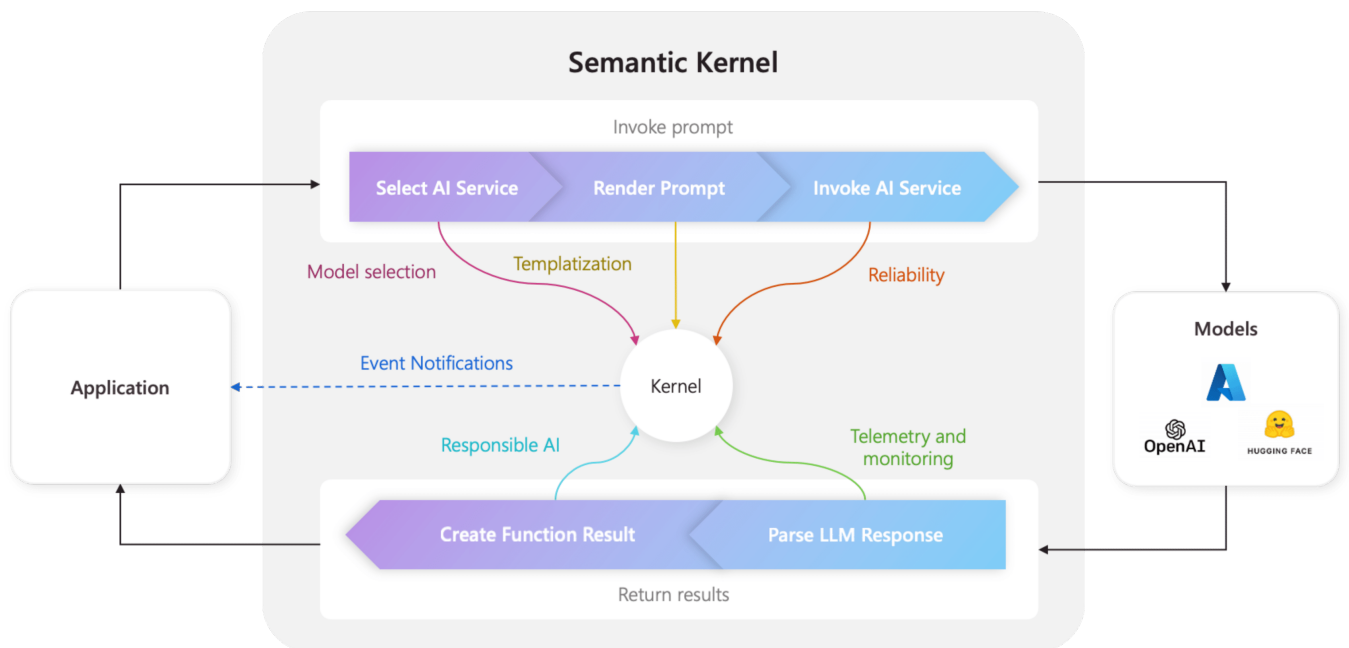
Semantic Kernel 的缺点

毕竟 LLMs 还在不停发展，有很多新模型的加入，也有很多新的功能，以及新的概念引入。Semantic Kernel，LangChain 等开源框架都在努力适应这个新的摩尔定律，但版本的迭代会有不确定的更改。所以在使用的时候，开发者需要多留意对应 GitHub Repo 上的变更日志。

还有 Semantic Kernel 需要兼顾多个编程语言，所以进度也是不一致，也会导致 Semantic Kernel 在不同技术栈人群的选择。

Semantic Kernel 中的 Kernel

如果把 Semantic Kernel 看作是 Copilot Stack 最佳实践，那 Kernel 就是 AI 编排的中心，在官方文档中也有所提及。通过 Kernel 可以和不同插件，服务，日志以及不同的模型链接在一起。所有 Semantic Kernel 的应用都从 Kernel 开始。



用 Semantic Kernel 构建一个简单的翻译项目

说了一些基础知识，我们开始学习，如何把 Semantic Kernel 引入到 .NET 和 Python 工程项目。我们用四步完成一个翻译的实现

第一步：引入 Semantic Kernel 库

.NET 开发者

注意：我们在这里使用的是最新的 **Semantic Kernel 1.0.1** 的版本, 并使用 **Polyglot Notebook** 来完成相关学习

```
#r "nuget: Microsoft.SemanticKernel, *-*"
```

Python 开发者

注意：我们在这里使用的是最新的 **Semantic Kernel 0.4.3.dev0** 的版本, 并使用 **Notebook** 来完成相关学习

```
! pip install semantic-kernel -U
```

第二步：创建 Kernel 对象

这里注意，我们需要把 Azure OpenAI Service 相关的 Endpoint，Key 以及 模型的 Deployment Name 都放在一个固定文件内，方便调用以及设置。在 .NET 环境我设置在 Settings.cs 环境，在 Python 环境我设置在 .env 环境中

.NET 开发者

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Connectors.OpenAI;

Kernel kernel = Kernel.CreateBuilder()
    .AddAzureOpenAIChatCompletion(Settings.AOAIModel ,
    Settings.AOAIEndpoint, Settings.AOAIKey)
    .Build();
```

Python 开发者

```
import semantic_kernel as sk
import semantic_kernel.connectors.ai.open_ai as skaoai

kernel = sk.Kernel()
deployment, api_key, endpoint = sk.azure_openai_settings_from_dot_env()
kernel.add_chat_service("azure_chat_competition_service",
    skaoai.AzureChatCompletion(deployment,endpoint,api_key=api_key,api_version
    = "2023-07-01-preview"))
```

第三步：引入插件

在 Semantic Kernel 中，我们有不同的插件，用户可以使用预定义的插件，也可以使用自定义的插件。想了解更多可以关注下一章的内容，我们会详细讲述插件的使用。该例子，我们使用的是自定义插件，已经在 plugins 目录下了。

.NET 开发者

```
var plugin =  
kernel.CreatePluginFromPromptDirectory(Path.Combine(pluginDirectory,  
"TranslatePlugin"));
```

Python 开发者

```
pluginFunc =  
kernel.import_semantic_skill_from_directory(base_plugin,"TranslatePlugin")
```

第四步：执行

.NET 开发者

```
var transalteContent = await kernel.InvokeAsync( plugin["Basic"],new()  
{["input"] = "你好"});  
  
transalteContent.GetValue();
```

Python 开发者

```
translateFunc = pluginFunc["Basic"]  
  
result = translateFunc("你好，我是你的 AI 编排助手 – Semantic Kernel")
```

具体实现，您可以访问

.NET 例子 请[点击访问这里](#)

Python 例子 请[点击访问这里](#)

小结

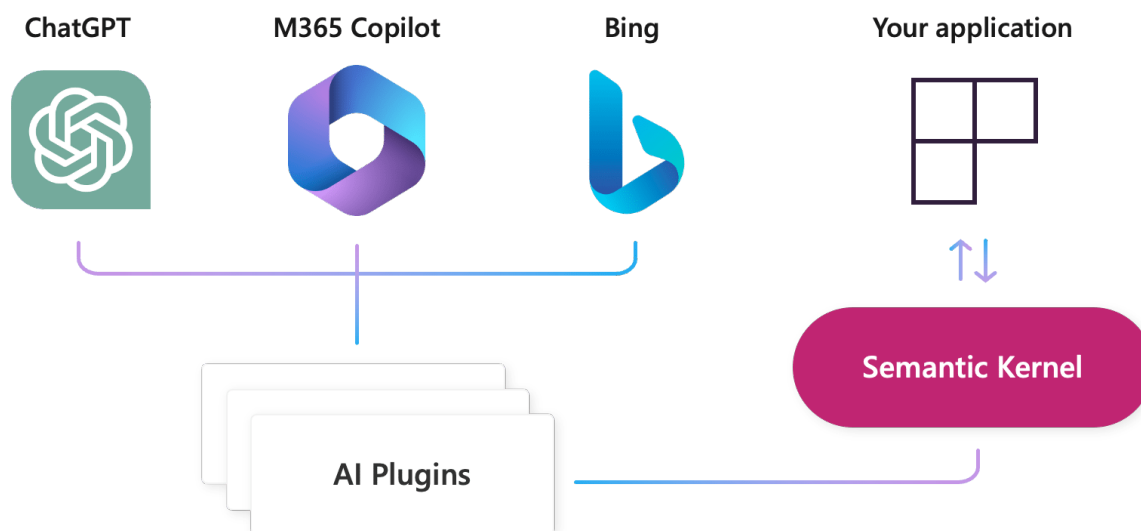
您第一次接触到 Semantic Kernel 的感觉如何呢？您在本章中学习到 Semantic Kernel 的基础知识，以及相关的知识。并了解到 Semantic Kernel 和 LangChain 的对比，以及相关优缺点，对您在工程落地的过程中会有

所帮助。我们也通过 Semantic Kernel 四步完成了一个翻译，当是完成了一个大模型时代下的 hello world，给你的入门带来信心。接下来您可以打开下一章的进阶学习，了解更多的 Semantic Kernel 知识。

第三章：开启大模型的技能之门 - Plugins

在上一章，我们了解了 Semantic Kernel 的基本知识。其中 Semantic Kernel 的一大特点是拥有强大的插件，通过结合自定义/预定义的插件解决智能业务的问题。让传统的代码和智能插件一起工作灵活地接入到应用场景简化传统应用向智能化转型的过程。本章，我们主要介绍如何使用插件。

什么是插件



我们知道 LLMs 本来的数据是有时间限制的，如果要增加实时内容或者企业化的知识是有相当大的缺陷。OpenAI 通过插件将 ChatGPT 连接到第三方应用程序。这些插件使 ChatGPT 能够与开发人员定义的 API 进行交互，从而增强 ChatGPT 的功能并允许有更广泛的操作，如：

1. 检索实时信息，例如，体育赛事比分、股票价格、最新新闻等。
2. 检索知识库信息，例如，公司文档、个人笔记等。
3. 协助用户进行相关操作，例如，预订航班、订餐等。

Semantic Kernel 遵循 OpenAI 的插件的插件规范，可以很方便地接入和导出插件(如基于 Bing, Microsoft 365, OpenAI 的插件)，这样可以让开发人员很简单地调用不同的插件服务。除了兼容 OpenAI 的插件外，Semantic Kernel 内也有属于自己插件定义的方式。不仅可以在规定模版格式上定义 Plugins, 更可以在函数内定义 Plugins.

认知 Plugins

在早期的 Semantic Kernel 预览版本中，Plugins 并定义为 Skills。正如上面提及的，和 OpenAI 看齐。但具体的目标没有变化。你可以理解 Semantic Kernel 的插件就是给开发人员通过完成智能业务需求的各种功能。你可以把 Semantic Kernel 看成是乐高的底座，要完成一个长城的堆砌就需要有各种不同的功能模块。而这些功能模块就是我们所说的插件。

我们可能有基于业务的插件集，如 HRPlugins 下就包含了不同的功能，如：

插件	Intro
----	-------

插件	Intro
Holidays	关于假期内容的插件
Contracts	关于员工合同
Departments	关于组织结构相关的

这些子功能可以根据不同的要求进行有效组合来完成不同的计划任务。例如以下结构：

```
|-plugins
  |-HRPlugins
    |-Holidays
    |-Contract
    |-Departments
  |-EmailsPlugins
    |-HRMail
    |-CustomMail
  |-PeoplesPlugins
    |-Managers
    |-Workers
```

我们向大模型发出一条指令“请向合同到期的管理人员，发送一个 Email”，实际就是从不同的组件找组合，最后就是依据 Contract + Managers + HRMail 来完成相关的工作

Semantic Kernel 插件

通过模版定义插件

我们知道通过提示工程可以和 LLMs 进行对话。对于一个企业或者创业公司，我们在处理业务时，可能不是一个提示工程，可能需要有针对提示工程的合集。我们可以把这些针对业务能力的提示工程集放到 Semantic Kernel 的插件集合内。对于结合提示工程的插件，Semantic Kernel 有固定的模版，提示工程都放在 skprompt.txt 文件内，而相关参数设置都放在 config.json 文件内。最后的文件结构式这样的

```
|-plugins
  |-HRPlugins
    |-Holidays
      |-skprompt.txt
      |-config.json
    |-Contract
      |-skprompt.txt
      |-config.json
    |-Departments
      |-skprompt.txt
      |-config.json
  |-EmailsPlugins
    |-HRMail
```

```

    | -skprompt.txt
    | -config.json
  | -CustomMail
    | -skprompt.txt
    | -config.json
  | -PeoplesPlugins
    | -Managers
      | -skprompt.txt
      | -config.json
  | -Workers
    | -skprompt.txt
    | -config.json

```

我们先来看看 **skprompt.txt** 的定义，这里一般是放置和业务相关的 prompt，可以支持多个参数，每个参数都放置在 {{ \$参数名 }} 内，如以下格式：

```
Translate {{ $input }} into {{ $language }}
```

我们这里的工作是把输入内容翻译成特定的语言，input 和 language 是两个参数，也就是说你可以给这两个参数给出随意的值。

在 **config.json** 中就是配置相关的内容，除了设置和 LLMs 相关的参数外，你也可以设定输入的参数以及相关描述

```

{
  "schema": 1,
  "type": "completion",
  "description": "Translate sentences into a language of your choice",
  "completion": [
    {
      "max_tokens": 2000,
      "temperature": 0.7,
      "top_p": 0.0,
      "presence_penalty": 0.0,
      "frequency_penalty": 0.0,
      "stop_sequences": [
        "[done]"
      ]
    }
  ],
  "input": {
    "parameters": [
      {
        "name": "input",

```

```

        "description": "sentence to translate",
        "defaultValue": ""
    },
    {
        "name": "language",
        "description": "Language to translate to",
        "defaultValue": ""
    }
]
}

```

通过函数定义插件

我们也可以通过函数定义不同的插件，这个有点像 gpt-3.5-turbo 在 6 月 13 日发布的 Function Calling，这是通过增加外部函数，通过调用来增强 OpenAI 模型的能力。正如本章开篇时所提及的。

Function Calling

通过描述式的函数，LLMs 调用这些函数的参数的 JSON 对象。这是一种 GPT 功能与外部工具和 API 连接起来的方法。支持 Azure OpenAI Service 或 OpenAI 上的：

- gpt-4
- gpt-4-1106-preview
- gpt-4-0613
- gpt-3.5-turbo
- gpt-3.5-turbo-1106
- gpt-3.5-turbo-0613

Semantic Kernel 也支持 Function Calling 的方式调用，但一般很少采用，除非你本来有为业务定制好的 Function Calling 方法

Semantic Kernel 定义函数插件

Semantic Kernel 定义函数插件，比起用 Function Calling 更为简单，而且在 Function Calling 诞生之前就有了相关的定义方法，不受模型限制，你可以在早期模型中就用上这种方式对 LLMs 进行知识和数据扩充。所有的自定义函数建议放置在 plugins 的同一文件夹下方便管理。

.NET 应用场景

定义函数扩展，需要加上宏定义

```

[KernelFunction,Description("search weather")]
public string WeatherSearch(string text)
{
    return "Guangzhou, 2 degree,rainy";
}

```

```
}
```

注意： 建议用业务类封装的方式定义好不同的扩展函数，方便调用时更加简单，如

```
using Microsoft.SemanticKernel;
using System.ComponentModel;
using System.Globalization;

public class CompanySearchPlugin
{
    [KernelFunction, Description("search employee infomation")]
    public string EmployeeSearch(string input)
    {
        return "欢迎了解社保相关内容";
    }

    [KernelFunction, Description("search weather")]
    public string WeatherSearch(string text)
    {
        return "Guangzhou, 2 degree, rainy";
    }
}
```

调用方法如下：

```
var companySearchPluginObj = new CompanySearchPlugin();

var companySearchPlugin =
kernel.ImportPluginFromObject(companySearchPluginObj,
"CompanySearchPlugin");

var weatherContent = await kernel.InvokeAsync(
companySearchPlugin["WeatherSearch"], new() { ["text"] = "广州天气" });

weatherContent.GetValue<string>()
```

Python 应用场景

定义函数扩展，需要加上宏定义

```
@sk_function_context_parameter(name="city", description="city string")
def ask_weather_function(self, context: SKContext) -> str:
    return "Guangzhou's weather is 30 celsius degree , and very hot."
```

把自定义函数都放置在 **native_function.py** 中，并通过类来定义，如

```
from semantic_kernel.skill_definition import sk_function,
sk_function_context_parameter
from semantic_kernel import SKContext

class API:
    @sk_function(
        description = "Get news from the web",
        name = "NewsPlugin"
    )
    @sk_function_context_parameter(name="location", description="location
name")
    def get_news_api(self, context: SKContext) -> str:
        return """"Get news from the """" + context["location"] + """".""

    @sk_function(
        description="Search Weather in a city",
        name="WeatherFunction"
    )
    @sk_function_context_parameter(name="city", description="city string")
    def ask_weather_function(self, context: SKContext) -> str:
        return "Guangzhou's weather is 30 celsius degree , and very hot."

    @sk_function(
        description="Search Docs",
        name="DocsFunction"
    )
    @sk_function_context_parameter(name="docs", description="docs string")
    def ask_docs_function(self, context: SKContext) -> str:
        return "ask docs :" + context["docs"]
```

调用方法如下：

```
api_plugin = kernel.import_native_skill_from_directory(base_plugin ,
"APIPlugin")
```

```
context_variables = sk.ContextVariables(variables={
    "location": "China"
})

news_result = await api_plugin["NewsPlugin"].invoke_async(
    variables=context_variables)

print(news_result)
```

内置插件

Semantic Kernel 有非常多的预定义插件，作为解决一般业务的相关能力，当然随着 Semantic Kernel 的成熟，会有更多的内置插件融入进来。

例子

.NET 例子 请[点击访问这里](#)

Python 例子 请[点击访问这里](#)

小结

通过插件你可以基于业务场景完成更多的工作，通过本章的学习，你已经初步掌握插件的定义以及如何使用插件进行工作。希望你在真正的业务工作中，基于业务场景，打造属于企业的插件库

第四章：Planner 组件 - 让大模型有规划地工作

我们在第三章学习了 Semantic Kernel 非常重要的功能 - 插件，通过插件可以完成针对不同领域的工作。LLMs 改变了人机交互的方式，通过自然语言去与大模型对话，让大模型完成工作。但往往我们给出的指令不只是完成单一的工作，如“请根据西雅图的天气给出出差的人群发一个穿衣提醒的邮件”。我们一直希望人工智能和人类作对比，那我们把上述指令用人的思维去思考，就会有如下拆分：

1. 查询西雅图天气
2. 从公司系统查询出差的人以及其联系方式
3. 穿衣提醒邮件模版
4. 发送邮件

LLMs 其实都有同样的思维，在 Semantic Kernel 中就有强大的 Planner 功能来做任务拆分的事情。本章就会和大家讲述相关的内容。

什么是 Planner

Planner 是 Semantic Kernel 一个重要组件，通过它可以接收任务指令，然后与已经在 Kernel 定义好的内置插件或者自定义插件做对应，让任务指令按部就班来工作。就如开篇所提及的内容，实际上针对“请根据西雅图的天气给出出差的人群发一个穿衣提醒的邮件”的指令，会先在 plugins 定义相关的插件，并通过 Kernel 注册后，Semantic Kernel 就协助您去配对步骤。



如何使用 Planner

现在 Planner 还在进化阶段，在最新的 .NET 版本中，我们发现 Semantic Kernel 关于 Planner 组件 Microsoft.SemanticKernel.Planners.Handlebars 的版本与核心的 Microsoft.SemanticKernel 有所不同。有用

用户在质疑是否 Semantic Kernel 的版本号混乱。你可以理解在 2023 年 Semantic Kernel 团队完成了核心部分，至于针对组件化的功能如 Planner，如 Memory，如部分 Connector 等都还在进化。毕竟这些功能和 LLMs 的发展有关。

如果您需要使用 Planner，需要考虑您的业务场景。例如在一些业务流程中加入 Planner，以及工具链中加入 Planner 都是非常有用的，毕竟人类对工作自动化有很多思考

.NET 场景下

需要添加关于 Planner 的相关组件库

```
#r "nuget: Microsoft.SemanticKernel.Planners.Handlebars, *-*"
```

引用库

```
using Microsoft.SemanticKernel.Planning;
```

注意：在采用 HanlerBars 使用时，你需要注意注意这些功能还在演变，在使用时请忽略 SKEXP0060

```
#pragma warning disable SKEXP0060
```

Python 场景下

直接引用库

```
from semantic_kernel.planning.basic_planner import BasicPlanner
```

或

```
from semantic_kernel.planning.sequential_planner import SequentialPlanner
```

注意：这里 Python 的 Planner 和 .NET 的 Planner 设定有所不同，Python 应该和 .NET 同步，所以大家在 Python 使用时，需要有以后变更的准备

改变中的 Planner

在官方的博客中，有所提及 Planner 的变化 <https://devblogs.microsoft.com/semantic-kernel/migrating-from-the-sequential-and-stepwise-planners-to-the-new-handlebars-and-stepwise-planner/> 结合了 Function Calling 的调用重新整理了在预览版中不同的 Planner 整合，大家可以关注一下该内容来了解。

如果你希望了解 Planner 实现的原理，请参考

<https://github.com/microsoft/semantic-kernel/blob/main/dotnet/src/Planners/Planners.Handlebars/Handlebars/CreatePlanPrompt.handlebars>

例子

.NET 例子 请[点击访问这里](#)

Python 例子 请[点击访问这里](#)

小结

Planner 的加入让 Semantic Kernel 的可用性大大提高，特别针对业务和工具类的场景。构建企业级的插件库，对于 Planner 落地也是非常重要，毕竟我们通过插件组合出不同的任务来完成工作。

第五章：嵌入式的技巧 - Embeddings

很多行业希望拥有 LLMs 的能力，希望 LLMs 能解决自己的企业内部问题。这就包括员工相关的内容如入职须知，请假和报销流程，还有福利查询等，企业业务流相关的内容包括相关文档，法规，执行流程等，也有一些面向客户的查询。虽然 LLMs 有强大的知识能力，但是基于行业的数据和知识是没办法获取的。那如何注入这些基于行业的知识内容呢？这也是让 LLMs 迈入企业化重要的一步。本章我们就会和大家讲讲如何注入行业的数据和知识，让 LLMs 变得更专业。也就是我们创建 RAG 应用的基础。

从自然语言中的向量谈起

在自然语言领域，我们知道最细的粒度是词，词组成句，句构成段落，篇章和最后的文档。计算机是不认识词的，所以我们需要对词转换为数学上的表示。这个表示就是向量，也就是 Vector。向量是一个数学上的概念，它是一个有方向的量，有大小和方向。有了向量，我们可以有效地对文本进行向量化，这也是计算机自然语言领域的基础。在自然语言处理领域，我们有很多向量化的方法，比如 One-hot，TF-IDF，Word2Vec，Glove，ELMO，GPT，BERT 等。这些向量化的方法都有各自的优缺点，但是都是基于词的向量化，也就是词向量。词向量是自然语言处理的基础，也是 OpenAI 所有模型的基础。我们分别看看词向量里面的几种常见方法。

One-hot 编码

One-hot 编码，是用 0 和 1 的编码方式来表示词。比如我们有 4 个词，分别是：我，爱，北京，天安门。那么我们可以用 4 个向量来表示这 4 个词，分别是：

```
我 = [1, 0, 0, 0]
爱 = [0, 1, 0, 0]
北京 = [0, 0, 1, 0]
天安门 = [0, 0, 0, 1]
```

在传统的自然语言应用场景中，我们把每个词看成用 One-Hot 向量表示，作为唯一的离散符号。我们的词库中有单词的数量就是向量的维度，如上述的例子总共包含了四个词，所以我们可以用一个四维的向量来表示。在这个向量中，每个词都是唯一的，也就是说每个词都是独立的，没有任何关系。这样的向量我们称为 One-Hot 向量。One-Hot 向量的优点是简单，容易理解，而且每个词都是唯一的，没有任何关系。但是 One-Hot 向量的缺点也很明显，就是向量的维度会随着词的增加而增加。比如我们有 1000 个词，那么我们的向量就是 1000 维的。这样的向量是非常稀疏的，也就是大部分的值都是 0。这样的向量会导致计算机的计算量非常大，而且也不利于计算机的计算。所以 One-Hot 编码的缺点就是向量维度大，计算量大，计算效率低。

TF-IDF 编码

TF-IDF 是一个统计学，通过评估一个词对一个语料的重要程度。TF-IDF 是 Term Frequency - Inverse Document Frequency 的缩写，中文叫做词频-逆文档频率。TF-IDF 的主要思想是：如果某个词在一篇文章中出现的频率高，并且在其他文章中很少出现，那么这个词就是这篇文章的关键词。一般我们习惯把这个概念拆分，分为 TF 和 IDF 两个部分。

TF - 词频

词频 (Term Frequency) 指的是某个词在文章中出现的频率。词频的计算公式如下：

$$TF = \text{某个词在文章中出现的次数} / \text{文章的总词数}$$

TF 有一个问题，就是如果一个词在文章中出现的次数很多，那么这个词的 TF 值就会很大。这样的话，我们就会认为这个词是这篇文章的关键词。但是这样的话，我们会发现，很多词都是这篇文章的关键词，这样的话，我们就无法区分哪些词是这篇文章的关键词了。所以我们需要对 TF 进行一些调整，这个调整就是 IDF。

IDF - 逆文档频率

逆文档频率 (Inverse Document Frequency) 指的是某个词在所有文章中出现的频率。逆文档频率的计算公式如下：

$$IDF = \log(\text{语料库的文档总数} / (\text{包含该词的文档数} + 1))$$

IDF 的计算公式中，分母加 1 是为了避免分母为 0 的情况。IDF 的计算公式中，语料库的文档总数是固定的，所以我们只需要计算包含该词的文档数就可以了。如果一个词在很多文章中都出现，那么这个词的 IDF 值就会很小。如果一个词在很少的文章中出现，那么这个词的 IDF 值就会很大。这样的话，我们就可以通过 TF 和 IDF 的乘积来计算一个词的 TF-IDF 值。TF-IDF 的计算公式如下：

$$TF-IDF = TF * IDF$$

TF-IDF 经常用于文本分类的场景，这也是 TF-IDF 最常用的场景。TF-IDF 的优点是简单，容易理解，而且计算量也不大。TF-IDF 的缺点是没有考虑词的顺序，而且没有考虑词与词之间的关系。所以 TF-IDF 适合用于文本分类的场景，而不适合用于文本生成的场景。

Word2Vec 编码

Word2Vec 我们也叫它做 Word Embeddings，中文叫做词嵌入。Word2Vec 的主要思想是：一个词的语义可以通过它的上下文来确定。Word2Vec 有两种模型，分别是 CBOW 和 Skip-Gram。CBOW 是 Continuous Bag-of-Words 的缩写，中文叫做连续词袋模型。Skip-Gram 是 Skip-Gram Model 的缩写，中文叫做跳字模型。CBOW 模型的思想是通过一个词的上下文来预测这个词。Skip-Gram 模型的思想是通过一个词来预测这个词的上下文。Word2Vec 的优点是可以得到词的语义，而且可以得到词与词之间的关系。对比起 One-Hot 编码和 TF-IDF 编码，Word2Vec 编码的优点是可以得到词的语义，而且可以得到词与词之间的关系。Word2Vec 编码的缺点是计算量大，而且需要大量的语料库。

之前我们提及过，One-Hot 编码的维度是词的个数，而 Word2Vec 编码的维度是可以指定的。一般我们会指定为 100 维或者 300 维。Word2Vec 编码的维度越高，词与词之间的关系就越丰富，但是计算量也就越大。Word2Vec 编码的维度越低，词与词之间的关系就越简单，但是计算量也就越小。Word2Vec 编码的维度一般是 100 维或者 300 维，这样的维度可以满足大部分的应用场景。

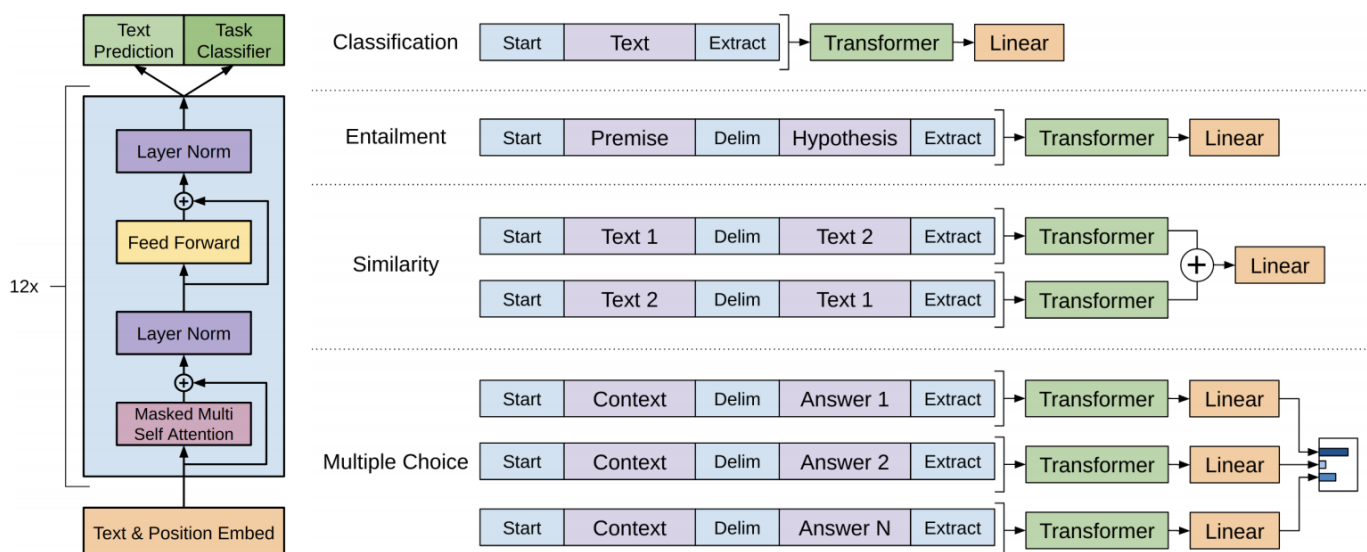
Word2Vec 编码的计算公式非常简单，就是 Word Embeddings。Word Embeddings 是一个词向量，它的维度是可以指定的。Word Embeddings 的维度一般是 100 维或者 300 维，这样的维度可以满足大部分的应用场景。Word Embeddings 的计算公式如下：

$$\text{Word Embeddings} = \text{词的语义} + \text{词与词之间的关系}$$

可以把 Word2Vec 看作是简单化的神经网络。

GPT 模型

GPT 模型的全称是 Generative Pre-Training，中文叫做预训练生成模型。GPT 模型是 OpenAI 在 2018 年提出的，它的主要思想是：一个词的语义可以通过它的上下文来确定。GPT 模型的优点是可以得到词的语义，而且可以得到词与词之间的关系。GPT 模型的缺点是计算量大，而且需要大量的语料库。GPT 模型的结构是一个多层单向的 Transformer 结构，它的结构如下图所示：



训练过程是两个阶段，第一个阶段是预训练，第二个阶段是微调。预训练的语料是维基百科和 BookCorpus，微调的语料是不同的自然语言任务。预训练的目标是通过一个词的上下文来预测这个词，微调的目标是根据不同的自然语言任务，如文本分类、文本生成、问答系统等，对语义模型进行微调，得到不同的模型。

GPT 模型已经经历了 4 个阶段，最为出名的就是 ChatGPT 所使用的 GPT-3.5 以及 GPT 4。GPT 开启了全新的时代，它的出现让我们看到了自然语言处理的无限可能。GPT 模型的优点是可以得到词的语义，而且可以得到词与词之间的关系。GPT 模型的缺点是计算量大，而且需要大量的语料库。很多人希望拥有自己行业对标的 GPT，这也是我们本章需要解决的问题。

BERT 编码

BERT 是 Bidirectional Encoder Representations from Transformers 的缩写，中文叫做双向编码器的 Transformer。BERT 是一个预训练的模型，它的训练语料是维基百科和 BookCorpus。BERT 的主要思想是：一个词的语义可以通过它的上下文来确定。BERT 的优点是可以得到词的语义，而且可以得到词与词之间的关系。对比起 One-Hot 编码、TF-IDF 编码和 Word2Vec 编码，BERT 编码的优点是可以得到词的语义，而且可以得到词与词之间的关系。BERT 编码的缺点是计算量大，而且需要大量的语料库。

Embeddings 向量嵌入技术

我们提及了 One-Hot 编码、TF-IDF 编码、Word2Vec 编码、BERT 编码和 GPT 模型。这些编码和模型都是 Embeddings 嵌入技术的一种。Embeddings 嵌入技术的主要思想是：一个词的语义可以通过它的上下文来确定。Embeddings 嵌入技术的优点是可以得到词的语义，而且可以得到词与词之间的关系。Embeddings 是作为自然语言深度学习的基础，它的出现让我们看到了自然语言处理的无限可能。

对于文本内容的 Embeddings 方法，我们结合上一节，你会发现从 word2vec 技术诞生后，文本内容的 Embeddings 就不断得到加强，从 word2vec 到 GPT 再到 BERT, Embeddings 技术的效果越来越好。Embeddings 技术的本质就是“压缩”，用更少的维度来表示更多的信息。这样的好处是可以节省存储空间，提高计算效率。

在 Azure OpenAI Service 中，Embeddings 技术的应用非常广泛，将文本字符串转换为浮点向量，通过向量之间的距离来衡量文本之间的相似度。不同行业希望加入自己的数据 我们就可以把这些企业级的数据通过 OpenAI Embeddings - text-embedding-ada-002 模型查询出向量，并通过映射进行保存，在使用时将问题也转换为向量，通过相似度的算法对比，找出最接近的 TopN 结果，从而找到与问题相关联的企业内容。

我们可以通过向量数据库将企业数据向量化后保存，结合 text-embedding-ada-002 模型通过向量的相似度进行查询，从而找到与问题相关联的企业内容。现在常用的向量数据库就包括 Qdrant, Milvus, Faiss, Annoy, NMSLIB 等。

Open AI 的 Embeddings 模型

OpenAI 的文本嵌入向量文本字符串的相关性。嵌入通常用于以下场景

- 搜索（结果按与查询字符串的相关性排序）
- 聚类（其中文本字符串按相似性分组）
- 推荐（推荐具有相关文本字符串的项目）
- 异常检测（识别出相关性很小的异常值）
- 多样性测量（分析相似性分布）
- 分类（其中文本字符串按其最相似的标签分类）

嵌入是浮点数的向量（列表）。两个向量之间的距离衡量它们的相关性。小距离表示高相关性，大距离表示低相关性。例如，如果您有一个嵌入为[0.1,0.2,0.3]的字符串“狗”，则该字符串与嵌入为[0.2,0.3,0.4]的字符串“猫”比与嵌入为[0.9,0.8,0.7]的字符串“汽车”更相关。

Semantic Kernel 的 Embeddings

在 Semantic Kernel 中对 Embeddings 的支持非常好，除了支持 text-embedding-ada-002 外，也对向量数据库进行支持。Semantic Kernel 对向量数据库做了抽象，开发人员可以用一致的 API 进行向量数据库的调用。本次案例以 **Qdrant** 为例，为了您顺利运行例子，请先安装好 Docker，并安装好 Qdrant 容器并运行，运行脚本如下：

```
docker pull qdrant/qdrant  
  
docker run -p 6333:6333 qdrant/qdrant
```

使用方式如下：

.NET 场景

添加 Nuget 库

```
#r "nuget: Microsoft.SemanticKernel.Connectors.Qdrant, *-*
```

引用库

```
using Microsoft.SemanticKernel.Memory;  
using Microsoft.SemanticKernel.Connectors.Qdrant;
```

创建实例和 Memory 绑定

```
var textEmbedding = new AzureOpenAITextEmbeddingGenerationService("Your  
Azure OpenAI Service Embedding Models Deployment Name" , "Your Azure  
OpenAI Service Endpoint", "Your Azure OpenAI Service API Key");  
  
var qdrantMemoryBuilder = new MemoryBuilder();  
qdrantMemoryBuilder.WithTextEmbeddingGeneration(textEmbedding);  
qdrantMemoryBuilder.WithQdrantMemoryStore("http://localhost:6333", 1536);  
  
var qdrantBuilder = qdrantMemoryBuilder.Build();
```

注意：Memory 组件还在调整阶段所以你需要注意接口有改变风险，还需要忽略以下信息

```
#pragma warning disable SKEXP0003  
#pragma warning disable SKEXP0011
```

```
#pragma warning disable SKEXP0026
```

Python 场景

引用库

```
from semantic_kernel.connectors.ai.open_ai import AzureChatCompletion,
AzureTextEmbedding
from semantic_kernel.connectors.memory.qdrant import QdrantMemoryStore
```

添加模块支持

```
kernel.add_text_embedding_generation_service(
    "embeddings_services", AzureTextEmbedding("EmbeddingModel",
endpoint, api_key=api_key, api_version = "2023-07-01-preview")
)
```

添加 Memory 链接

```
qdrant_store = QdrantMemoryStore(vector_size=1536,
url="http://localhost", port=6333)
await qdrant_store.create_collection_async('aboutMe')
kernel.register_memory_store(memory_store=qdrant_store)
```

注意： Memory 组件还在调整阶段所以您需要注意接口有改变风险

Semantic Kernel 中保存和搜索您的向量

在 Semantic Kernel 中，通过抽象的方式把不同向量数据的方法进行了统一，您可以很方便地保存和搜索您的向量

.NET 场景

保存向量数据

```
await qdrantBuilder.SaveInformationAsync(conceptCollectionName, id:
```



```
"info1", text: "KinfeY is Microsoft Cloud Advocate");
await qdrantBuilder.SaveInformationAsync(conceptCollectionName, id:
"info2", text: "KinfeY is ex-Microsoft MVP");
await qdrantBuilder.SaveInformationAsync(conceptCollectionName, id:
"info3", text: "KinfeY is AI Expert");
await qdrantBuilder.SaveInformationAsync(conceptCollectionName, id:
"info4", text: "OpenAI is a company that is developing artificial general
intelligence (AGI) with widely distributed economic benefits.");
```

查询向量数据

```
string questionText = "Do you know kinfeY ?";
var searchResults = qdrantBuilder.SearchAsync(conceptCollectionName,
questionText, limit: 3, minRelevanceScore: 0.7);

await foreach (var item in searchResults)
{
    Console.WriteLine(item.Metadata.Text + " : " + item.Relevance);
}
```

Python 场景

保存向量数据

```
await kernel.memory.save_information_async(base_vectordb, id="info1",
text="KinfeY is Microsoft Cloud Advocate")
await kernel.memory.save_information_async(base_vectordb, id="info2",
text="KinfeY is ex-Microsoft MVP")
await kernel.memory.save_information_async(base_vectordb, id="info3",
text="KinfeY is AI Expert")
await kernel.memory.save_information_async(base_vectordb, id="info4",
text="OpenAI is a company that is developing artificial general
intelligence (AGI) with widely distributed economic benefits.")
```

查询向量数据

```
ask = "who is kinfeY?"

memories = await kernel.memory.search_async(
```

```
base_vectordb, ask, limit=3, min_relevance_score=0.8
)

i = 0
for memory in memories:
    i = i + 1
    print(f"Top {i} Result: {memory.text} with score {memory.relevance}")
```

您可以非常简单方便地接入任意的向量数据库来完成相关的操作，也意味着可以非常简单地构建 RAG 应用。

例子

.NET 例子 请[点击访问这里](#)

Python 例子 请[点击访问这里](#)

小结

现在很多的企业数据进入到 LLMs 使用的都是通过 Embeddings 的方式构建 RAG 应用。Semantic Kernel 无论在 Python 还是 .NET 都给了我们非常简单的方式来完成相关的功能，所以对于一些希望在工程增加 RAG 应用的人来说是非常大的帮助。

AK



Microsoft Azure