# 第 1 章

# 智能体革命的基石——从LLM到Agent的跨越

1


2020年6月，OpenAI发布GPT-3模型，其参数量达到惊人的1750亿，首次展现出强大的少样本学习能力与通用语言理解能力。从这一年开始，大语言模型（Large Language Model，LLM）正式进入公众视野。随后，基于GPT-3.5模型的ChatGPT横空出世，一度令世人惊叹于机器的对话能力——它仿佛为计算机赋予了一颗“会思考的大脑”。然而，仅停留在对话层面的人工智能（AI），仍然只是被动回应人类指令的工具，尚无法真正自主地改变世界。

到了2025年，这一局面发生了根本性转变。这一年被业界普遍视为AI Agent（智能体，简称Agent）的爆发元年。AI Agent的出现，标志着人工智能从“知识生成者”向“自主行动者”的关键跃迁。AI Agent不再满足于回答问题，而是能够感知环境、规划任务、调用工具、执行行动，并通过反馈不断迭代优化，形成一个完整的闭环智能系统。这场智能体革命被业界视为继LLM之后的下一个重要技术拐点，它将重塑生产力边界，使AI不再只是辅助性工具，而是能够自主规划、协同执行并持续迭代的智能伙伴，在更广泛的工作与生活场景中承担真正的执行与赋能职责。

本章将从AI Agent的基石——LLM入手，剖析它的内在逻辑；继而探讨AI Agent如何超越LLM，实现从被动对话到主动行动的跃迁；随后深入介绍这一核心转变的运行机制——感知、思考、行动与反馈；最后，列举AI Agent在办公自动化、智能客服、科学研究以及内容创作等现实场景中的落地应用。通过本章的学习，读者可以对LLM与AI Agent建立起较为系统的基础认知。

## 1.1 LLM 智能的源头与大脑

本节首先阐述LLM（大语言模型）的基本概念与发展脉络，然后聚焦GPT系列模型的演进历程，最后深入剖析其核心技术架构——Transformer。

### 1.1.1　什么是 LLM

LLM（大语言模型）是一种在海量文本语料上经过自监督预训练的超大规模深度学习模型，能够通过学习语言的统计规律来理解和生成自然语言。这类模型通常拥有数以亿计甚至千亿计的参数，例如GPT-3模型包含1750亿参数。参数规模的空前增长，以及训练语料体量的极大扩展（可达数万亿Token），使得LLM能够捕获语言中极其复杂的模式与关系。

凭借庞大的网络规模和丰富的训练语料，LLM在词汇、语法及上下文语义等方面学到了深层次的知识结构，从而能够执行多种自然语言处理任务，包括语言理解和语言生成。图1-1展示了LLM的总体结构和能力。

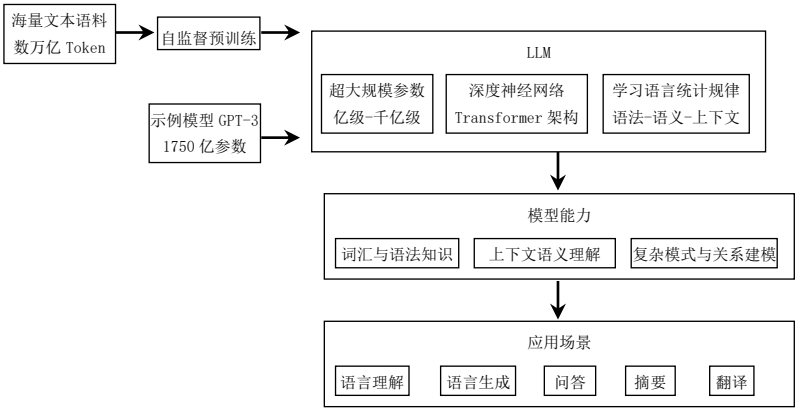


图 1-1　LLM 的总体结构与能力

从模型结构来看，现代LLM绝大部分基于Transformer架构构建（详见1.1.3节）。该架构由Vaswani等人于2017年提出，是一种以自注意力机制（Self-Attention）为核心的神经网络结构。Transformer打破了传统循环神经网络（Recurrent Neural Network，RNN）在处理长序列时效率低、难以并行的局限，通过引入自注意力机制，实现对序列中任意位置的信息进行全局建模。

Transformer通常由编码器（Encoder）和解码器（Decoder）两个模块组成：编码器负责编码输入序列的语义表示，解码器则基于编码器输出逐步生成目标序列。与早期循环神经网络相比，Transformer架构具有更强的并行计算能力、更好的长距离依赖建模能力，并更易于扩展模型规模。Transformer架构与传统RNN的对比如图1-2所示。


图 1-2　Transformer 架构与传统 RNN 的对比示意图

LLM的训练通常分为两个阶段：预训练（Pre-training）与微调（Fine-tuning）。在预训练阶段，模型在海量通用文本数据上通过自监督任务（如预测下一个Token或被掩码的Token）学习语言模式。例如，GPT系列模型在预训练中采用自回归目标，使模型根据前文预测下一个Token（词元），从而学习文本生成规律；BERT模型则使用掩码语言模型（Masked Language Model）任务，在输入中随机掩盖部分词元，再根据上下文来进行预测，以学习双向语义表示。预训练赋予模型通用语言知识，而微调则使模型适配特定任务的数据分布和目标。

GPT-1是首个系统性展示“生成式预训练+有监督微调”范式的模型：它先利用海量文本进行无监督预训练，再针对下游任务进行有监督微调，从而显著提升任务性能。这一“预训练-微调”范式已成为现代自然语言处理（Natural Language Processing，NLP）的基本流程，大幅降低了下游任务对大规模人工标注数据的依赖。

得益于Transformer架构与预训练范式，LLM展现出强大的泛化能力和跨任务迁移能力。作为通用序列模型，LLM不仅能够生成文本，还可完成摘要、翻译、问答与推理等多种语言任务。与以往为每种任务单独设计模型的方法不同，LLM无须针对每个任务进行大量专门训练，而是可以通过少量示例或提示（即少样本学习和提示学习能力）快速适应新任务。例如，GPT-3在无须额外微调的情况下，仅通过少量示例即可在翻译、问答、代码生成等多种任务中表现出较强能力。这表明LLM能够在最小监督下横跨多种任务发挥作用，使对话代理、代码生成、知识检索与自动推理等应用成为可能。

然而，LLM在带来技术突破的同时，也继承了训练语料中的不准确性和偏差，并可能在生成文本时产生事实性错误（即模型“幻觉”现象）。此外，由于模型规模庞大，LLM的训练和推理均需要巨大的计算资源。这些挑战促使研究者更加重视模型行为对齐（Alignment）与安全性问题，例如通过在微调阶段引入人类反馈（RLHF）来调优模型输出，使其更符合人类期望并减少有害内容。

总体而言，LLM是当代人工智能领域具有划时代意义的基础新技术之一。它以Transformer为基石，以预训练-微调范式为基础，在多任务通用智能方面展现出前所未有的潜力。

图1-3展示了LLM从预训练到微调再到实际应用的完整流程。

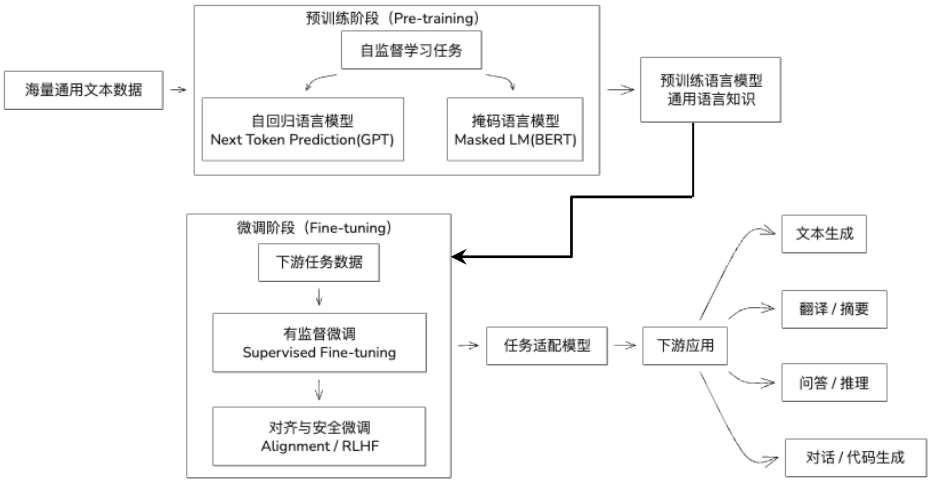


图 1-3　LLM 从预训练到微调再到实际应用的完整流程示意图

### 1.1.2　LLM 的发展简史

LLM（大语言模型）的发展可以追溯到语言模型的早期探索阶段。

早期的统计语言模型（如n元语法模型，即n-gram模型）在20世纪90年代已被用于基于大规模语料计算词序列概率，并在机器翻译等任务中取得了一定成果。

进入21世纪后，研究者开始尝试使用神经网络构建语言模型。2001年前后，基于前馈神经网络的语言模型被提出；随后，循环神经网络（Recurrent Neural Network，RNN）以及长短期记忆网络（Long Short-Term Memory，LSTM）等结构相继用于建模序列数据中的长距离依赖关系。

2013年前后，词向量方法（如Mikolov提出的Word2Vec）为语言模型提供了从大规模语料中学习词语分布式表示的新途径，奠定了深度表示学习的重要基础。

2016年，谷歌将机器翻译模型从统计方法升级为基于LSTM的编码器–解码器架构（Encoder–Decoder），实现了端到端的神经网络机器翻译系统。然而，此类RNN/LSTM模型在并行能力与长距离依赖建模方面仍存在明显局限。

2017年，谷歌研究人员在NeurIPS大会上发表了论文*Attention Is All You Need*，正式提出了革命性的Transformer架构。这一里程碑式架构完全摒弃了循环网络，基于自注意力机制实现高效的并行序列处理，被誉为现代LLM的核心基石。

图1-4展示了从20世纪90年代到2017年早期语言模型技术的发展时间线。

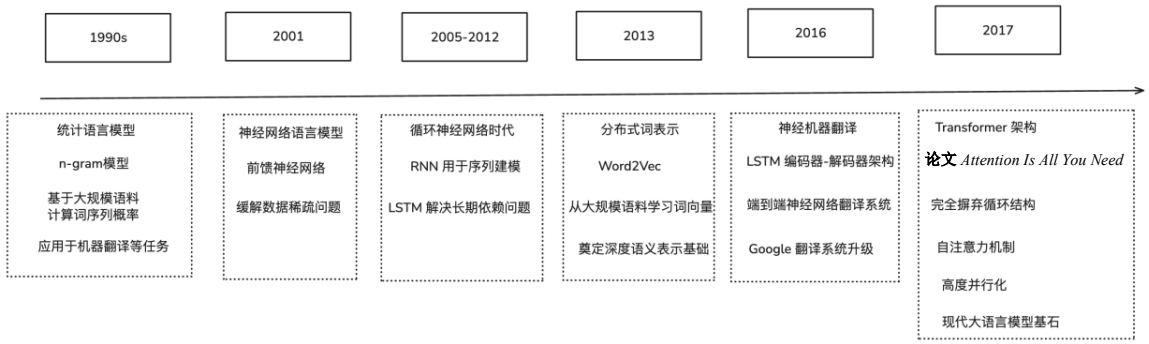


图 1-4　早期语言模型技术发展时间线示意图

2018年是预训练语言模型兴起的元年，业界相继推出两种不同范式的代表模型：GPT-1和BERT。

GPT-1（Generative Pre-trained Transformer，由OpenAI于2018年6月发布）率先验证了“生成式预训练+任务微调”的语言模型训练范式。GPT-1仅使用Transformer的解码器结构，通过在海量文本上进行无监督训练来学习语言生成规律，然后针对具体NLP任务进行有监督微调。这在当时取得了显著效果。

BERT（Bidirectional Encoder Representations from Transformers，由Google于2018年10月发布）是一种仅基于Transformer编码器的深度双向预训练语言表示模型。它通过掩码语言模型等预训练任务学习双向上下文信息，并在问答、自然语言推理等11项NLP任务上刷新了当时的最优成绩。

GPT-1和BERT分别确立了自回归生成和双向编码理解两条技术路线：前者侧重文本生成能力，后者侧重文本理解能力。后续发展表明，以GPT为代表的自回归生成范式在LLM中展现出更强的统一建模能力，并逐渐成为主流技术路线。图1-5展示了GPT-1和BERT这两种不同范式的代表模型。

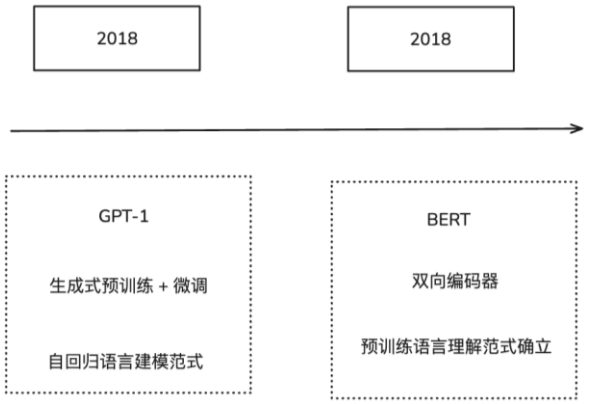


图 1-5　GPT-1 和 BERT 模型的信息

2019年，LLM的规模和能力进一步提升，主要代表模型为GPT-2、T5和BART等。

GPT-2由OpenAI于2019年2月发布，其参数规模扩大至15亿。GPT-2展示了惊人的零样本（Zero-Shot）学习能力：在未进行任务微调的情况下，仅凭预训练即可生成连贯文本或完成回答任务。由于生成效果高度逼真，OpenAI出于对潜在滥用风险的考虑，采用了分阶段公开GPT-2模型的策略，引发了关于AI模型开放性和安全性的广泛讨论。

同年，谷歌提出T5（Text-to-Text Transfer Transformer）模型，Facebook提出BART模型，两者共同探索了统一“文本到文本”的任务建模范式。T5将所有NLP任务统一为文本生成问题，通过大规模迁移学习在翻译、问答、摘要等任务上取得了当时的最先进（State-of-the-Art，SOTA）的性能。2019年的这些进展标志着业界开始系统性探索统一序列转换框架，即用单一模型处理不同形式的语言输入与输出。图1-6展示了GPT-2和T5/BART模型的信息。

2020年，LLM迎来了里程碑式的突破，主要代表模型是GPT-3。

GPT-3由OpenAI在2020年5月发布，其参数规模飙升到1750亿。GPT-3之所以引发广泛关注，源于其论文*Language Models are Few-Shot Learners*所展示的现象：超大规模预训练模型无须针对每个任务进行微调，仅通过少量示例（Few-Shot）提示或纯文本指令，即可在众多复杂任务上实现基于上下文的学习。换言之，GPT-3具备了强大的即席学习（Just-in-Time Learning，简称JITL）能力，即通过少量示例迅速理解任务模式。这一结果验证了模型规模与性能之间的“缩放定律”（Scaling Law）：随着参数规模与数据规模的提升，模型能力呈现质变的飞跃。

GPT-3的发布使业界进一步看到了通用人工智能（Artificial General Intelligence，AGI）的发展潜力。OpenAI通过API形式开放GPT-3的使用，进一步推动了基于LLM构建的应用开发热潮，因此2020年也常被称为“大语言模型元年”。图1-7展示了GPT-3模型的信息。

2021—2022年期间，各大科研机构和企业持续加大对LLM的投入，模型规模不断攀升，同时也开始出现围绕模型效率和质量优化的新研究方向。

2021年，Google推出面向对话场景优化的LaMDA模型，强调对话系统的安全性、连贯性与交互自然性。

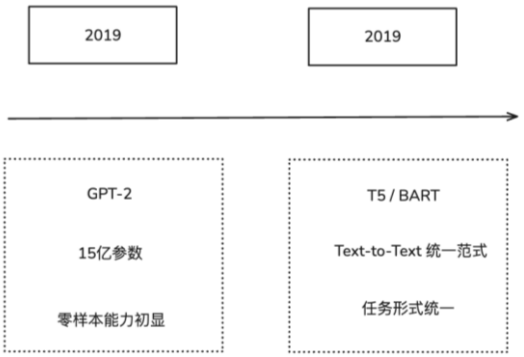


图 1-6　GPT-2 和 T5/BART 模型的信息

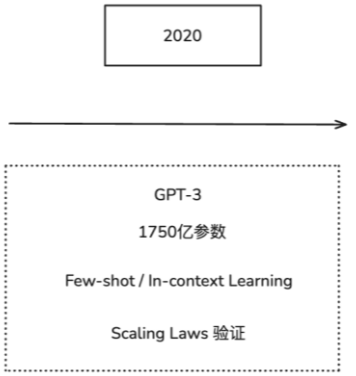


图 1-7　GPT-3 模型的信息

2021年10月，DeepMind发布Gopher，其参数规模达到2800亿，体现了在更大规模上训练语言模型的野心。

2022年3月，Google发布PaLM模型，其参数规模高达5400亿。PaLM在多项当时的主流基准测试中取得优异成绩，在推理和代码生成任务上表现尤为突出，展示了超大规模模型在复杂任务上的强劲能力。

2022年5月，Meta发布OPT模型，其参数与GPT-3相当（1750亿），并向学术界开放模型权重。OPT的推出旨在提升研究透明度，降低LLM研究门槛。随后，大型多语种开源模型BLOOM（1760亿参数，支持数十种语言）也在2022年由科研合作组织发布。这些开源努力使研究者和开发者能够在本地环境中深入研究并复现LLM，从而显著提升社区创新活力。图1-8展示了2021—2022年发布的重要模型。

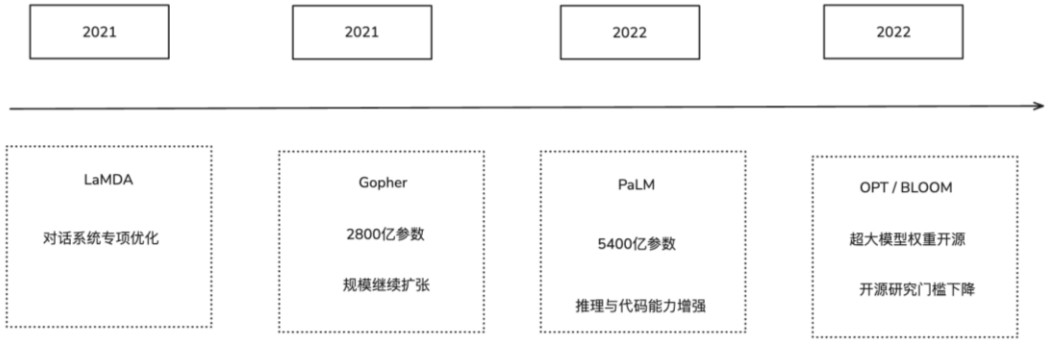


图 1-8　LaMDA、PaLM 等模型的信息

2023年可以认为是LLM从研究阶段走向大众应用的关键一年。一方面，OpenAI于2022年底推出的基于GPT-3.5的ChatGPT网页版迅速引发广泛关注；另一方面，更先进的多模态和开源模型也相继涌现。

2023年2月，Meta发布LLaMA（Large Language Model Meta AI）模型，该系列LLM面向学术界开放，包含70亿（7B）、130亿（13B）、330亿（33B）、650亿（65B）等不同参数规模的版本。尽管参数规模不及同期最大模型，LLaMA通过“以更大规模数据训练相对较小模型”的策略，实现

了远超预期的性能表现。其中65B版本在多项基准任务上接近甚至优于GPT-3，表现突出。更具影响力的是，LLaMA模型权重在发布后意外在网络泄露，引发开源社区广泛响应：研究者基于LLaMA进行微调和优化，迅速衍生出Alpaca、Vicuna等模型。可以认为，LLaMA直接推动了开源LLM生态的爆发式增长，极大降低了模型应用与开发门槛。

2023年3月，OpenAI发布GPT-4模型，这是GPT系列的第四代模型，也是首个“多模态”LLM。GPT-4在纯文本任务上的能力较GPT-3.5进一步跃升，在诸多专业考试中达到接近人类专家的水平。同时，它支持图像输入（输出仍为文本），展现出对视觉信息的理解与描述能力。GPT-4确立了新的行业标杆，但OpenAI对其内部架构和参数规模守口如瓶，只通过API形式提供服务。

2023年7月，Meta发布LLaMA 2模型，这是首个由科技巨头公开发布并允许免费商用的开源LLM。LLaMA 2提供了7B到70B不同规模版本，并包含针对对话优化的Chat版，在开源许可下供开发者使用。这一举措被视为对封闭模型体系的直接挑战，加速了开源LLM在工业界的应用落地。

同期，Anthropic公司推出了Claude 2模型，其核心创新在于支持高达10万Token的超长上下文窗口，显著增强了长文本对话和分析能力。

随着2023年开源生态和多模态技术的兴起，LLM领域正进入百花齐放、竞争加剧的新阶段。图1-9展示了2023年发布的ChatGPT应用及重要模型。


图 1-9　ChatGPT 应用和 GPT-4 等模型的信息

2024年是以开源模型为主线，多模态能力初步显现的一年。

Meta在2024年4月发布的LLaMA 3（8B、70B）基于约15万亿Token的预训练，进一步验证了数据规模与训练策略对模型泛化能力的重要作用，并在随后推出的LLaMA 3.1（最高405B）中刷新了开源模型参数规模上限，成为当时最大规模的公开可用基础模型。这些版本在自然语言理解、长上下文处理和多语种任务中均表现优异。

阿里Qwen系列于2024年完成从Qwen2到Qwen2.5的持续演进，并衍生出代码（Code）、语音（Audio）、视觉语言（Vision Language，VL）等多个专用子模型。其中QwQ-32B在长上下文推理任务中表现突出，标志着中文及多语种LLM能力的快速提升与追赶。

DeepSeek在2024年年底发布了DeepSeek V3模型，采用Mixture-of-Experts（MoE）架构（总参数671B、激活参数37B），以更低训练成本实现接近GPT-4o的综合能力（GPT-4o代表迈向更自然人机交互的重要进展，支持文本、音频和图像的多模态输入，显著提升人机交互的自然性），验证了高效架构设计与稀疏化策略的工程价值。

总体而言，2024年LLM发展的核心趋势体现在：开源生态逐步成熟，以及长上下文与多模态能

力开始同步萌芽。图1-10展示了2024年发布的重要模型，这一年被称为“开源元年”。

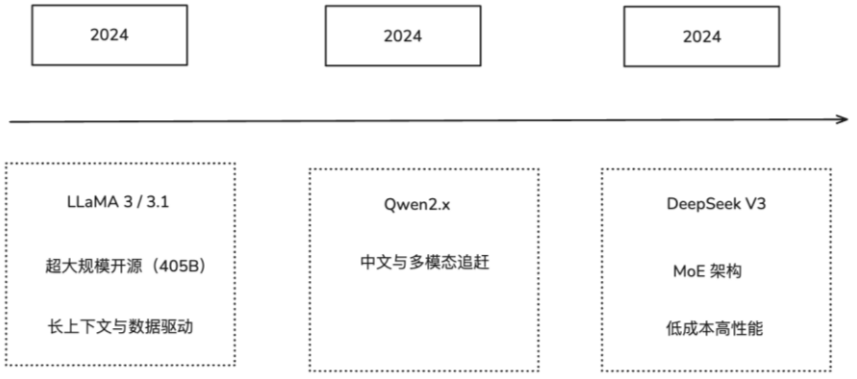


图 1-10　Qwen2.x 及 DeepSeek V3 等模型的信息

2025年，LLM的发展进入以复杂推理、超长上下文处理与多模态融合为核心的系统化能力升级阶段。

OpenAI于2025年11月发布GPT-5.1，引入Instant和Thinking两种变体，通过“自适应推理”机制动态分配计算资源，在响应速度与推理深度之间实现平衡，同时提升对话自然度和指令遵循能力。随着竞争加剧，OpenAI于同年12月进一步推出GPT-5.2，在通用智能、代码理解及长上下文处理能力方面持续增强，以应对其他模型的竞争。

谷歌DeepMind在2025年11月推出Gemini 3系列模型，标志着多模态基础模型进入新的发展阶段。该系列在原生多模态处理、超长上下文支持与结构化推理能力方面实现显著提升。Gemini 3不仅能够流畅地处理和整合文本、图像、音频和视频等多种数据类型，还展现出更卓越的长上下文理解能力和复杂任务处理能力。

DeepSeek持续推进推理模型路线演进：R1系列验证了通过强化学习提升推理能力的潜力，后续的V3-0324、R1-0528、V3.1及V3.2/Speciale等版本进一步通过混合思维机制、超长上下文（如128K训练）及稀疏注意力机制增强推理与智能体执行能力。

阿里Qwen在2025年构建了更完整的多模态与高效架构体系：Qwen3实现全面开源并支持128K上下文与可切换推理模式；Qwen3-Next探索Hybrid Attention与高效MoE架构；Qwen3-Omni进一步实现音视频、图像与视频等多模态输入与输出能力的统一建模。

综上所述，2025 年 LLM 发展呈现出三大核心趋势：复杂推理能力系统化演进、多模态原生化融合，以及在开源与闭源阵营之间的多维竞争格局加速形成。图 1-11 展示了 2025 年发布的重要模型。这一年也被普遍称为“智能体（Agent）元年”，其核心原因在于底层基座模型能力显著增强，使得智能体系统构建逐渐成为迈向通用人工智能（AGI）的重要路径之一。

总体来看，自2017年Transformer架构提出以来，LLM经历了从架构创新、范式验证到规模扩张，再到开源生态繁荣与多模态融合的清晰演进路径。以GPT系列、BERT、T5、LLaMA、Gemini系列、DeepSeek系列及Qwen系列为代表的关键模型，共同奠定了当代自然语言处理（NLP）与基础模型体系的技术基础，并持续推动人工智能能向更通用、更强大的方向演进。本书对LLM发展历程进行了系统梳理，并标注各阶段里程碑模型，如图1-12所示。

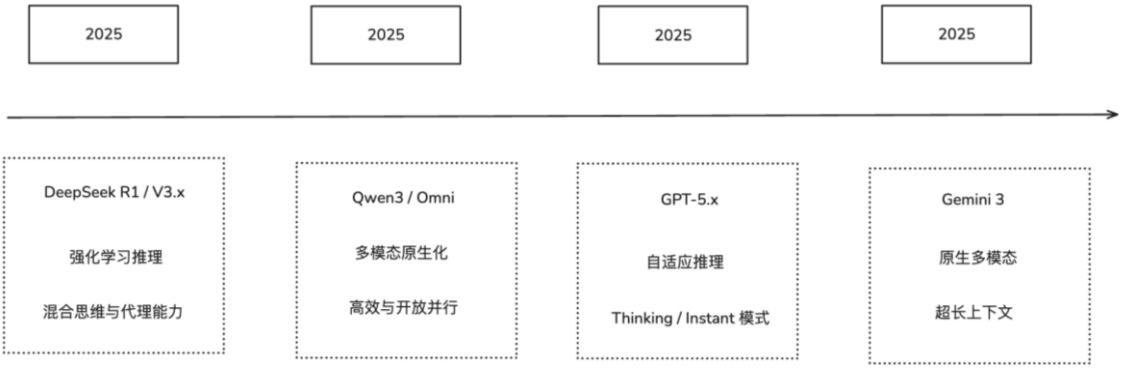


图 1-11　GPT-5.x 和 Gemini 3 等模型的信息

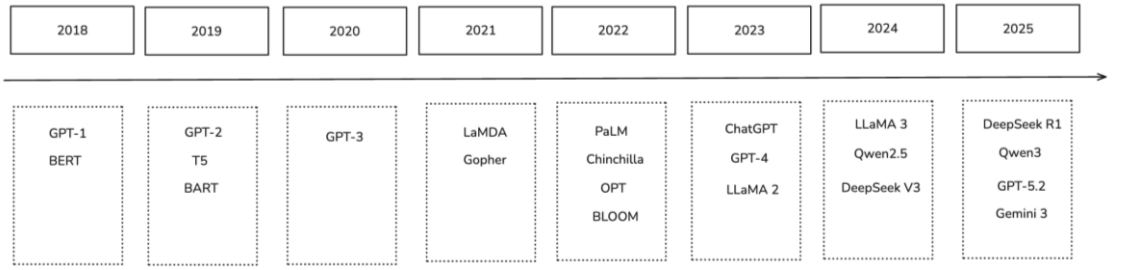


图 1-12　2018—2025 年主要里程碑模型

### 1.1.3　Transformer 架构

在前文对LLM（大语言模型）基础概念及发展简史的讨论中，多次提及Transformer架构。那么，Transformer架构究竟是什么？它如何运作？为何能够产生如此深远的影响？本节将围绕上述问题展开系统阐述。

#### 1. 什么是 Transformer 架构

Transformer架构是一种以自注意力机制（Self-Attention）为核心的深度神经网络架构，由Google Brain团队在论文*Attention Is All You Need*中首次提出。其基本思想是将输入文本划分为由若干Token构成的序列，并通过词嵌入（Embedding）将每个Token映射为向量表示。在模型的每一层中，各Token通过并行的多头注意力机制，在上下文范围内与其他（未被屏蔽的）Token建立关联，从而动态建模序列中不同位置之间的依赖关系。

通过这种机制，模型能够根据输入内容的重要性为不同Token分配不同权重，从而强化关键信息、抑制次要信息，实现对上下文语义的高效建模。此外，Transformer还引入前馈神经网络（Feedforward Neural Network，FFN或简称FNN）与位置编码（Positional Encoding），以增强序列顺序信息的表达能力。

Transformer架构最初主要应用于自然语言处理等序列建模任务，如机器翻译、文本摘要与问答系统等。与传统循环神经网络（RNN）和卷积神经网络（CNN）相比，Transformer不依赖序列递归或卷积结构，而是通过注意力机制直接建模全局依赖关系，从而支持高度并行计算，大幅提升训练效率。

随着技术发展，Transformer已从自然语言处理扩展至计算机视觉与多模态学习等多个领域，并逐渐成为LLM的核心基础架构。

### 2. Transformer 架构的组成和原理

接下来，将对 Transformer 架构的组成及其基本原理进行系统说明。在此之前，建议读者具备向量、矩阵、标量、点积及向量加法等基础数学知识。这些内容属于线性代数基础，可通过相关资料自行学习。掌握这些概念后，再理解 Transformer 架构中的向量与矩阵运算将更加清晰高效。

1）Transformer 架构的组成

如图1-13所示，Transformer架构的核心由两部分组成，分别为左侧的编码器（Encoder）和右侧的解码器（Decoder）。

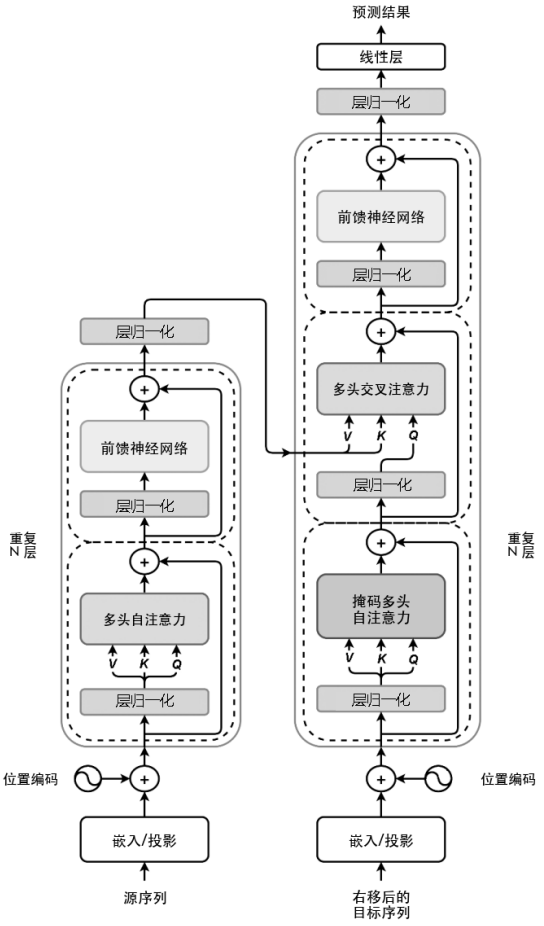


图 1-13　Transformer 架构图

下面通过一个具体示例来说明这两部分的组成。

首先，将Transformer视为一个“黑盒模型”。以法语翻译为英语为例：输入为法语句子，经由Transformer架构处理后输出为英语句子，如图1-14所示。


图 1-14　基于 Transformer 架构的翻译

打开该黑盒模型后，可以看到它的内部结构由编码组件、解码组件以及二者之间的连接组成。如图1-15所示，输入首先进入编码组件进行处理，其输出作为解码组件的输入，最终由解码组件生成输出结果。

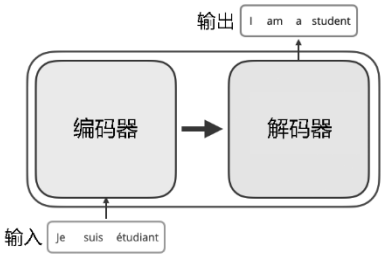


图 1-15　编码组件与解码组件

编码组件由多个编码器堆叠而成（示例中采用6个编码器，该数值本身无特殊含义，属于超参数，实际应用中可根据任务调整，如2、4、8等）；解码组件是由相同数量的解码器堆叠而成，如图1-16所示。

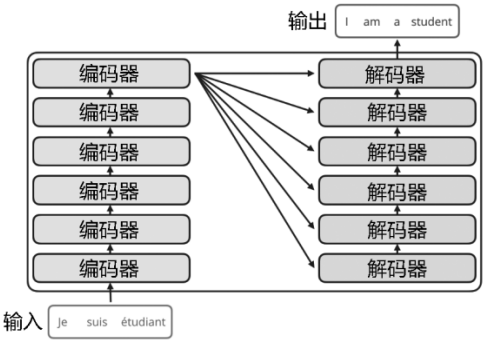


图 1-16　多个编码器与多个解码器

所有编码器在结构上相同，但是它们不共享权重。每个编码器由两个子层构成：自注意力层和前馈神经网络层，如图1-17所示。

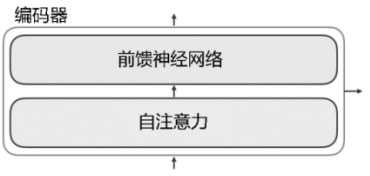


图 1-17　编码器内部组成

编码器的输入首先流入自注意力层。该层的作用是在对特定Token（词元）进行编码时，使模型能够关注输入序列中的其他Token，从而捕捉上下文依赖关系。关于自注意力机制的具体原理，将在后文详细介绍。

自注意力层的输出随后输入前馈神经网络（Feedforward Neural Network）。该网络所有位置上共享结构参数，但对每个位置的表示进行独立、等价的非线性变换，即每个Token的向量表示都会分别通过同一前馈神经网络进行处理。

解码器同样包含上述两个子层，但在它们之间额外引入编码器-解码器注意力层（Encoder-Decoder Attention），用于使解码器能够关注输入序列中的相关信息，从而有效利用编码器输出的上下文表示，如图1-18所示。

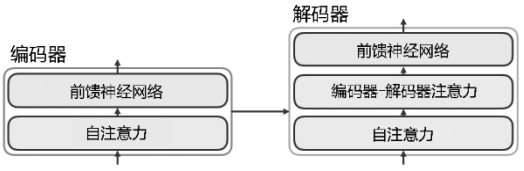


图 1-18　解码器内部组成

2）Transformer 架构中的词向量

在了解Transformer架构的组成之后，接下来重点关注Transformer架构中的词向量表示。在自然语言处理（NLP）任务中，通常首先通过嵌入（Embedding）模型将输入中的每一个Token转换为对应的词嵌入（Word Embedding）。如图1-19所示，三个输入的词（即词元）分别被转换为对应的向量（可用图中的小方格表示）。


图 1-19　输入 Token 被转换为向量

在进行嵌入之前，通常需要完成两个步骤。首先，对句子进行Token化处理，可借助OpenAI提供的Tokenizer工具，将文本拆分为一系列Token，并在词典表（即Token与Token ID的映射表）中获取对应的Token ID。其次，嵌入模型内部维护了一个从Token ID到向量的查找表（Lookup Table），通过Token ID即可检索到对应的向量表示。

通过上述过程，每个Token最终被转换为一个向量表示。在此基础上，还需关注向量维度的设置。例如，在本示例中，向量长度为4，即嵌入模型的维度为4。当然，该维度属于可调超参数，可设置为64、256或512等不同规模。一般而言，维度越高，模型可表达的信息越丰富，但计算开销也相应增加。

3）Transformer 架构中的位置向量

仅有词向量通常不足以完整表达句子的语义信息，因为在将token映射为向量的过程中，原始序列中的位置信息未被显式保留。而在NLP任务中，词序信息往往具有关键作用。另一方面，Transformer虽然基于自注意力机制能够并行处理序列信息，但其结构本身并不具备对序列顺序的内在建模能力。因此，在Transformer架构中，除词嵌入外，还需引入位置编码（Positional Encoding），用于对序列

中词元的绝对或相对位置信息进行编码，使模型能够感知序列顺序关系。

在此基础上，可以进一步了解位置编码的具体形式。位置编码通常表示为与词向量维度相同的矩阵。如图1-20所示，位置编码模型根据词元在序列中的位置索引生成对应的位置向量。随后，将该位置向量与词嵌入向量逐元素相加，得到Transformer的输入向量。该输入向量包含两方面的信息：一是词元（Token）的语义信息，二是词元在序列中的位置信息。

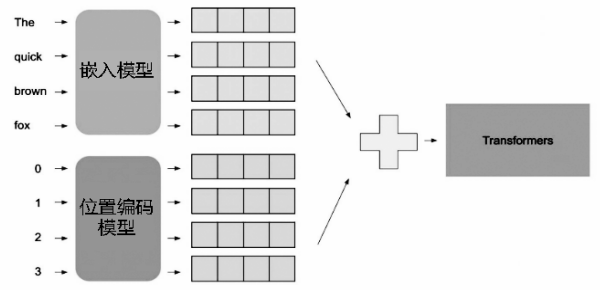


图 1-20　词向量与位置索引向量相加为位置编码向量

接下来，进一步介绍位置编码的计算方法。在论文*Attention Is All You Need*中，作者采用正弦函数与余弦函数对序列中每个位置生成固定且确定的编码。对于每个位置，其位置向量的不同维度分别由正弦函数和余弦函数生成，其中偶数维采用正弦函数，奇数维采用余弦函数，从而构成完整的位置编码表示。

$$PE(pos,2i)=\sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$PE(pos,2i+1)=\cos\left(\frac{pos}{10000^{2i/d}}\right)$$

- *pos*：词在序列中的位置索引（通常从0开始计数）。
- *i*：位置向量的维度索引，用于区分不同维度；其中2*i*表示偶数维，2*i*+1表示奇数维。
- *d*：模型嵌入维度（如512，表示每个位置对应一个512维向量）。
- 10000：缩放常数，为经验设定的超参数，用于控制不同维度上正弦与余弦函数的频率变化范围。
- *PE*：位置编码函数，用于将输入序列中位置*pos*映射为对应的编码向量。

通过上述公式，不同位置可获得不同的编码，同时在不同维度上呈现出不同频率变化，从而使模型能够有效区分序列中各个位置的信息。

举例来说，当缩放常数和嵌入维度（如*d*=4）确定之后，位置编码矩阵即可唯一确定。由于位置编码仅依赖于位置索引，与具体语句内容无关，因此对于相同长度的序列，其相同位置上的编码是一致的。

需要指出的是，位置编码具有特殊性，它并非通过模型训练学习得到，而是根据预定义公式直接计算生成。例如，对于句子“I am a Robot”（长度为4），其对应的位置编码矩阵如图1-21所示。

通过序列中每个Token的位置索引，可计算得到对应的*d*维位置编码向量，并将这些向量按顺序排列成位置编码矩阵。

| Token | pos | $PE_{(pos,0)}$ | $PE_{(pos,1)}$ | $PE_{(pos,2)}$ | $PE_{(pos,3)}$ |
|---|---|---|---|---|---|
| I | 0 | $\sin(0) = 0.0000$ | $\cos(0) = 1.0000$ | $\sin(0/100) = 0.0000$ | $\cos(0/100) = 1.0000$ |
| am | 1 | $\sin(1) = 0.8415$ | $\cos(1) = 0.5403$ | $\sin(1/100) = 0.0100$ | $\cos(1/100) = 0.99995$ |
| a | 2 | $\sin(2) = 0.9093$ | $\cos(2) = -0.4161$ | $\sin(2/100) = 0.0200$ | $\cos(2/100) = 0.99980$ |
| Robot | 3 | $\sin(3) = 0.1411$ | $\cos(3) = -0.9900$ | $\sin(3/100) = 0.0300$ | $\cos(3/100) = 0.99955$ |

图 1-21　位置编码矩阵

随后，将该序列的位置编码矩阵与词嵌入矩阵逐元素相加，其结果作为Transformer编码器的输入，如图1-22所示。

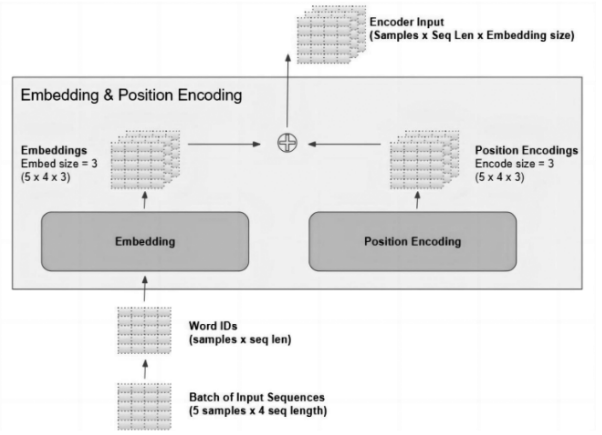


图 1-22　词嵌入矩阵与位置编码矩阵相加的结果作为编码器的输入

4）Transformer 编码器

Transformer编码器主要由自注意力子层和前馈神经网络子层组成，并在每个子层之后引入残差连接与层归一化操作，以提升深层网络的训练稳定性与信息传递效率。

图1-23展示了输入向量序列在编码器中的处理流程：首先，输入向量序列经过自注意力子层进行全局上下文建模，使序列中每个位置的表示能够融合来自其他位置的语义信息；随后，通过前馈神经网络子层对序列中各位置进行独立的非线性变换，进一步增强各位置的特征表达能力。

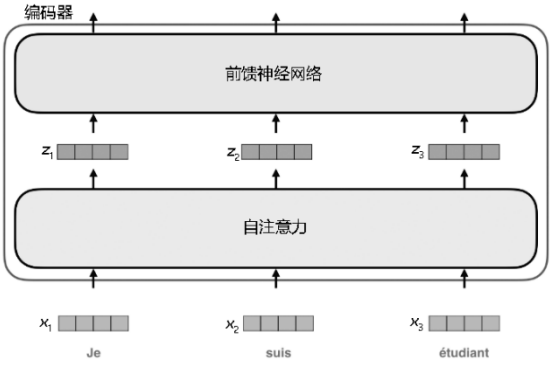


图 1-23　向量流经单层编码器

经过上述两阶段处理后，编码器输出一个包含丰富上下文信息的向量序列。该输出将作为下一层编码器的输入，经由多层编码器逐层处理之后，最终形成模型对输入序列的高层语义表示。

在此可以观察到Transformer编码器的一个关键特性：在层与层之间，序列中每个位置对应的向量会沿各自的计算路径逐层传递；但在自注意力层内部，不同位置的向量并非彼此独立。

具体来说，自注意力层会计算序列中各个Token之间的相关性。每个Token在生成新向量时，都会参考输入序列中自身及其他Token的信息，从而得到融合上下文的新向量。虽然这一过程可以通过矩阵运算并行完成，但从逻辑关系上看，每个Token的输出结果仍与整个输入序列相关。从注意力计算公式可以看出，这种依赖关系主要通过$\boldsymbol{QK}^{\mathrm{T}}$计算得到的相关性得分矩阵来体现。

$$\mathrm{Attention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})=\mathrm{softmax}\left(\frac{\boldsymbol{QK}^{\mathrm{T}}}{\sqrt{d_k}}\right)\boldsymbol{V}$$

其中，$d_k$是 $\boldsymbol{Q}$，$\boldsymbol{K}$ 矩阵的列数，即向量维度

然而，在前馈神经网络层中，不存在跨位置的依赖关系。从前馈神经网络的计算公式可以看出，前馈神经网络对每个位置的向量进行独立处理，各个位置之间的输入与输出互不影响，因此可以完全并行计算。

$$\mathrm{FFN}(x_i)=W_2(\mathrm{ReLU}(W_1x_i+b_1))+b_2$$

接下来，以两个英语单词为例，直观地观察编码器中各个子层的处理过程。如图 1-24 所示，Thinking 和 Machines 对应的向量首先进入自注意力层，并在该层中完成上下文信息融合；随后，自注意力层的输出被传递至前馈神经网络层，经过进一步非线性变换后，再作为下一层编码器的输入。

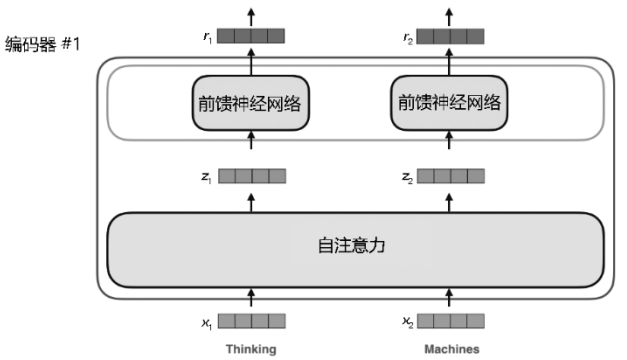


图 1-24　向量流经编码器的过程

5）Transformer 自注意力层

在理解编码器的整体处理流程后，接下来对自注意力层的具体计算机制进行进一步说明。

01 自注意力机制的第一步是计算 $\boldsymbol{Q}$、$\boldsymbol{K}$ 和 $\boldsymbol{V}$。

Transformer 首先将输入序列中各位置的词向量按行拼接，构成输入矩阵 $\boldsymbol{X}$；随后，将其分别与三组可训练权重矩阵 $\boldsymbol{W}^{Q}$，$\boldsymbol{W}^{K}$，$\boldsymbol{W}^{V}$ 相乘，从而通过线性变换得到对应的查询矩阵 $\boldsymbol{Q}$，键矩阵 $\boldsymbol{K}$ 和值矩阵 $\boldsymbol{V}$，如图 1-25 所示。

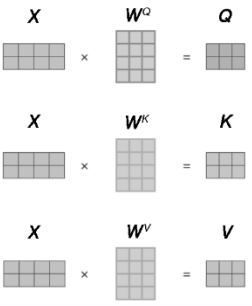


图 1-25　$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 矩阵的计算过程

Transformer自注意力机制为何引入$\boldsymbol{Q}$、$\boldsymbol{K}$和$\boldsymbol{V}$三组不同矩阵？这一问题可借助通用搜索引擎的检索过程加以理解。

在使用搜索引擎时，用户首先输入检索词，例如“Transformer为什么要引入$\boldsymbol{Q}$、$\boldsymbol{K}$和$\boldsymbol{V}$矩阵？”。该检索词表达了用户当前的信息需求，对应注意力机制中的查询矩阵Q，即当前位置“希望获取什么信息”。

随后，搜索引擎会在海量网页中筛选候选结果。为了高效评估网页与查询的相关程度，系统通常不直接将检索词与整篇网页内容逐字比对，而是优先基于网页标题、主题关键词或内容摘要等特征完成匹配计算。这类用于表示网页检索线索的特征，对应注意力机制中的键矩阵K，其核心作用是参与相关性计算，用以判断不同位置是否值得被关注。

完成相关性排序后，搜索引擎向用户返回的并非关键词、标题等匹配特征本身，而是网页的具体内容片段（如正文摘要、核心段落）。这部分经筛选后被提取并传递的有效信息，对应注意力机制中的值矩阵V，即某个位置被关注后实际提供的语义内容。

据此，可将完整的信息检索过程抽象为“查询—匹配—取值”三个相互关联且功能边界清晰的阶段。Transformer在自注意力机制中引入$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$矩阵的核心目的，正是对上述三个过程实现功能解耦：$\boldsymbol{Q}$表示查询需求，$\boldsymbol{K}$提供匹配依据，$\boldsymbol{V}$承载待传递的语义内容。

若不对三者进行功能区分，直接使用输入向量同时完成相关性计算与信息传递，则“匹配关系”和“内容表示”将发生耦合，从而限制模型的表达能力。换言之，当单一向量同时承担“查询”“匹配”与“信息传递”三重职能时，其语义表示的灵活性与适配空间将受到显著限制。

基于这一解耦设计思路，Transformer将输入矩阵$\boldsymbol{X}$分别与三组可训练权重矩阵$\boldsymbol{W}^{Q}$，$\boldsymbol{W}^{K}$，$\boldsymbol{W}^{V}$相乘，最终得到查询矩阵$\boldsymbol{Q}$、键矩阵$\boldsymbol{K}$和值矩阵$\boldsymbol{V}$：

$$\boldsymbol{Q}=\boldsymbol{X}\boldsymbol{W}^{Q},\quad \boldsymbol{K}=\boldsymbol{X}\boldsymbol{W}^{K},\quad \boldsymbol{V}=\boldsymbol{X}\boldsymbol{W}^{V}$$

其中，$\boldsymbol{W}^{Q}$、$\boldsymbol{W}^{K}$、$\boldsymbol{W}^{V}$为模型的可训练权重矩阵，通常在初始化阶段随机生成，并在训练过程中通过反向传播不断更新。通过三组独立的线性变换，模型可将同一输入特征映射至三个不同的表示空间，从而分别学习：

（1）哪些特征更适合作为查询条件（$\boldsymbol{Q}$）。

（2）哪些特征更适合作为匹配依据（$\boldsymbol{K}$）。

（3）哪些特征更适合作为信息内容进行传递（$\boldsymbol{V}$）。

这一解耦设计有效提升了注意力机制的表达能力。尤其是在多头注意力（Multi-Head Attention）机制中，每个注意力头都拥有各自独立的一组权重矩阵 $\left\{\boldsymbol{W}_i^Q,\boldsymbol{W}_i^K,\boldsymbol{W}_i^V\right\}$。这意味着模型可以在多个不同的表示子空间中对同一输入完成投影，使不同的注意力头关注不同类型的特征关系，例如语法依赖关系、语义关联关系以及长距离依赖关系等。

若不引入上述线性变换，直接使用原始输入向量完成注意力计算，则不同注意力头之间将难以形成有效的差异化表示，从而削弱多头注意力机制的作用。

综上，自注意力机制引入 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 三组独立矩阵，其核心目的在于：对“查询”“匹配”和“取值”三个计算过程进行功能解耦，并通过多子空间投影提升模型的表达能力，从而实现对复杂上下文语义关系的建模。

02 自注意力机制的第二步是计算自注意力分数。

假设当前针对输入序列中的首个单词 Thinking 执行自注意力分数计算，此时需要基于该单词对序列内所有单词进行相关性评分。该评分用于刻画当前单词 Thinking 与序列中各位置单词之间的关联程度，进而决定该位置对不同单词的注意力分配权重。

一般而言，自注意力分数计算采用点积运算，公式如下：

$$\text{score}(i,j)=\boldsymbol{q}_i\cdot\boldsymbol{k}_j$$

该点积运算的结果为一个标量，用于衡量序列中两个位置的单词在当前表示空间中的语义相关程度：分数越大，表明第$i$个位置与第$j$个位置的语义相关性越强。

具体自注意力分数的计算过程如图1-26所示：首先将单词Thinking对应的查询向量$\boldsymbol{q}_1$，与序列中每个单词对应的键向量逐一进行点积运算。其中，Thinking对自身的注意力分数由$\boldsymbol{q}_1$与$\boldsymbol{k}_1$的点积求得；Thinking对单词Machines的注意力分数，则由$\boldsymbol{q}_1$与$\boldsymbol{k}_2$的点积计算得到。

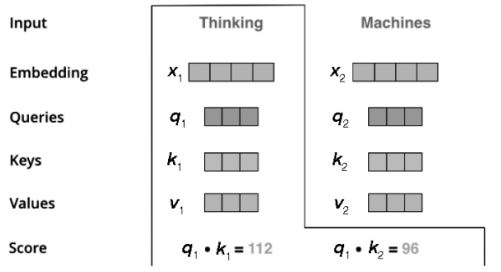


图 1-26　单词 Thinking 的自注意力分数计算过程

03 计算自注意力机制的第三步是对自注意力分数进行缩放处理。

自注意力分数的缩放处理，是将前序步骤得到的点积结果除以键向量维度的平方根，即：

$$\text{scōre}(i,j)=\frac{\boldsymbol{q}_i\cdot\boldsymbol{k}_j}{\sqrt{d_k}}$$

其中，$\text{scōre}(i,j)$为缩放后的自注意力分数；$d_k$为键向量的维度，在本示例中 $d_k$=64，对应缩放因子为 $\sqrt{d_k}=8$。

缩放的核心目的是抑制点积结果数值过大的问题，避免在后续 softmax 计算中出现梯度饱和、训练不稳定等现象。

**04** 自注意力机制的第四步是对缩放后的自注意力分数进行归一化处理，如图 1-27 所示。

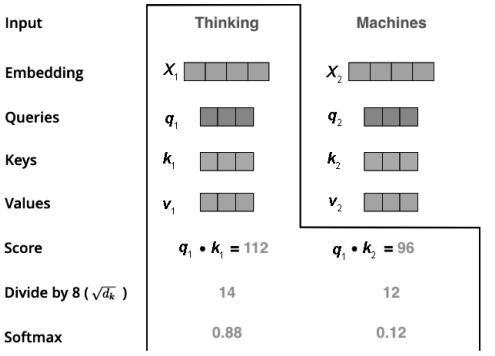


图 1-27　单词 Thinking 通过 softmax 函数对分数进行归一化

具体而言，将缩放处理后的自注意力分数输入 softmax 函数，使其转化为一个概率分布：

$$\alpha_{ij} = \text{softmax}(\text{score}(i, j))$$

其中，$\alpha_{ij}$为注意力权重，表示序列中第$j$个位置的信息对第$i$个位置的贡献程度，为后续的信息加权聚合提供计算依据。

经过softmax函数处理后，输出的所有注意力权重均为正值，且取值范围为0~1，并满足归一性约束（所有位置对应的概率之和恒为1）。

此处需要进一步说明缩放操作的重要性：若不对点积结果进行缩放处理，直接将其输入softmax函数，则输出结果可能趋于极端，呈现近似独热（one-hot）编码的特征——某一个位置的权重趋近于1，而其他位置的权重趋近于0。以图1-28中单词Thinking对应的softmax(112, 96)计算为例，将数值代入softmax公式进行计算：

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

计算结果将非常接近1或0。这是因为当输入值差距较大时，在指数运算后，这种差异会被进一步放大，从而使较大的值对应的权重接近1，较小的值对应的权重接近0。在此情况下，单词Machines对应的上下文信息几乎无法参与输出向量$\boldsymbol{z}_1$的聚合过程，从而造成上下文信息的丢失。

为避免这一问题，需要对点积结果进行缩放，以压缩数值之间的相对差距。这正是前序步骤中点积结果“除以 8”的核心作用。经缩放处理后，输入由 softmax(112, 96)变为 softmax(14, 12)，将其代入 softmax 公式进行计算，输出结果约为（0.88, 0.12），对应输出向量 $\boldsymbol{z}_1$=0.88$\boldsymbol{v}_1$+0.12$\boldsymbol{v}_2$。该结果表明，单词 Thinking 在生成表示时不仅保留了自身信息，也融合了单词 Machines 提供的上下文信息，而非完全忽略低相关性位置的内容。输出向量 $\boldsymbol{z}_2$ 的计算同样遵循这一逻辑。

**05** 自注意力机制的第五步是对 Value 向量进行加权运算。

具体而言，将序列中每个位置的 Value 向量 $\boldsymbol{v}_j$ 与对应的注意力权重 $\alpha_{ij}$ 相乘，即：

$$\alpha_{ij}\boldsymbol{v}_j$$

加权运算的意义在于：注意力权重越大，对应位置的 Value 向量在最终结果中的贡献就越大；反之，权重较小的位置，其贡献也相应较小。因此，通过该运算，模型能够实现对不同位置信息的加权筛选，使注意力更多集中于与当前位置语义相关性更高的内容。具体 Value 向量的加权计算过程可参考图 1-28 中 $\boldsymbol{v}_1$ 与 $\boldsymbol{v}_2$ 的对应运算。

**06** 自注意力机制的第六步是对加权后的向量进行求和。

具体而言，将第五步得到的所有加权向量进行逐元素求和，即可得到当前位置的输出向量。加权求和公式如下：

$$\boldsymbol{z}_i = \sum_j \alpha_{ij}\boldsymbol{v}_j$$

该步骤的核心作用是依据注意力权重，对整个输入序列的语义信息进行加权整合，从而生成融合全局上下文信息的新表示。具体详见图1-28中输出向量$\boldsymbol{z}_1$的计算过程。

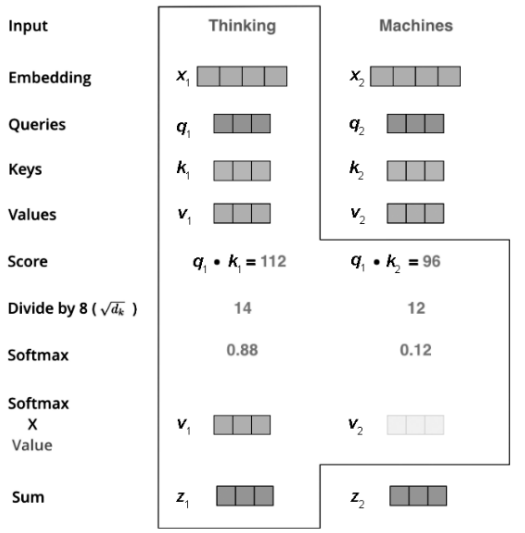


图 1-28　单词 Thinking 的自注意力计算全过程

需要说明的是，上述六个步骤是从概念层面对单个位置的自注意力计算过程进行拆解。然而，在实际实现中，自注意力机制通常以矩阵形式完成并行计算，从而一次性生成所有位置的输出向量，显著提升计算效率。

基于Transformer的矩阵化运算设计，可将上述第二步至第六步的计算过程整合为统一的矩阵公式，用于表示自注意力层的完整输出，如图1-29所示。

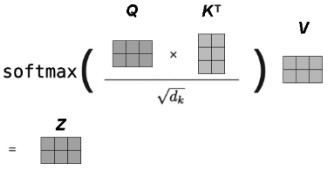


图 1-29　自注意力机制的矩阵形式计算公式

6）Transformer 前馈神经网络层

在Transformer架构中，前馈神经网络（FFN）层通常位于自注意力层之后，用于对序列中每个位置的Token向量独立执行两层全连接运算：各位置之间不发生信息交互，但共享同一组权重参数。具体计算公式如下：

$$\mathrm{FFN}(\boldsymbol{x}) = \mathrm{ReLU}(\boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2$$

其中，$\boldsymbol{x}$表示单个Token的输入向量；$\boldsymbol{W}_1$和$\boldsymbol{b}_1$分别表示第一层全连接层的权重和偏置；$\boldsymbol{W}_2$和$\boldsymbol{b}_2$分别表示第二层全连接层的权重和偏置。第一层全连接层首先将输入向量映射到更高维的中间空间，随后通过ReLU激活函数引入非线性表达能力；第二层全连接层通常不再设置激活函数，其主要作用是将中间特征重新映射回与输入向量一致的维度空间。

在前馈神经网络中，引入ReLU激活函数至关重要。如果两个全连接层之间不加入激活函数，多层线性变换可以等价合并为一次线性变换。以无激活函数的前馈神经网络为例，其计算形式如下：

$$\mathrm{FFN}(\boldsymbol{x}) = (\boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2$$

此时，网络虽在形式上分为两层，但本质仍为单一线性映射，模型的表达能力将受到严格限制。引入ReLU激活函数后，网络具备非线性表达能力，从而能够学习更加复杂的特征表示。ReLU的计算形式如下：

$$\mathrm{ReLU}(x) = \max(0, x)$$

该函数对输入向量进行逐元素计算：将小于0的值置为0，将大于0的值保持原值。因此，它会对负值特征进行抑制，使输出特征呈现一定的稀疏性，并兼具一定的特征筛选作用。

需要说明的是，原始Transformer架构中的前馈神经网络层默认使用ReLU激活函数，而在一些现代Transformer变体中，激活函数也可能被替换为GELU、SwiGLU、GeGLU等。但无论使用哪种激活函数，其核心目的都是在前馈神经网络中引入非线性表达能力，使模型能够学习更加复杂的特征表示。

从维度变化来看，前馈神经网络通常采用“先升维再降维”的结构，如图1-30所示。假设输入Token向量的维度为$d_{\mathrm{emb}}$，前馈神经网络中间层的维度为$d_{\mathrm{ffn}}$，则第一层全连接层将向量从$d_{\mathrm{emb}}$维映射到$d_{\mathrm{ffn}}$维，第二层全连接层再将其从$d_{\mathrm{ffn}}$维映射到$d_{\mathrm{emb}}$维。这种结构设计既增强了模型的特征表达能力，又保证前馈神经网络的输出维度与输入维度保持一致，便于后续残差连接（Add）与层归一化（Norm）计算。

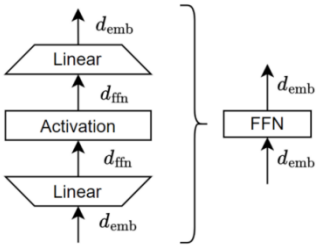


图 1-30　前馈神经网络层结构

例如，当输入Token的向量维度为512，即$d_{\mathrm{emb}}$=512，前馈神经网络的中间层维度为2048，即

$d_{ffn}$=2048时，其计算过程为：先通过第一层全连接层将512维向量映射到2048维的高维空间，再通过ReLU激活函数进行非线性变换，最后通过第二层全连接层将2048维向量重新映射回512维，最终得到的输出向量维度仍为512。具体前馈神经网络层的维度变化过程如图1-31所示。

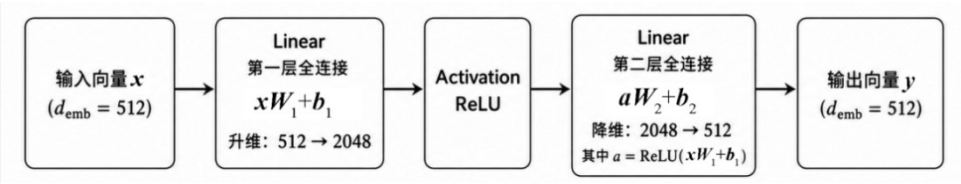


图 1-31　前馈神经网络层维度变化过程

7）Transformer 残差连接与层归一化

在Transformer架构中，每个子层（包括自注意力层或前馈神经网络层）之后通常都会引入残差连接（Residual Connection）与层归一化（Layer Normalization）操作。二者组合构成Add+Norm模块，该模块在编码器中的具体位置如图1-32所示。

作为Transformer的关键组成部分，Add+Norm模块能够有效提升深层网络中信息传递的稳定性，从而显著改善模型的训练性能与收敛效率。具体而言，残差连接与层归一化分别从梯度传播与特征分布两个角度对训练过程进行优化。

残差连接的核心思想是将子层的输入与输出进行逐元素相加，再将结果传递至后续网络结构。其优势主要体现在两个方面：一方面，它能够保留原始输入信息，避免重要特征在多层变换过程中被逐步削弱；另一方面，它为梯度在深层网络中的反向传播提供了捷径，有效缓解了梯度消失问题，从而进一步提升模型训练的稳定性。

层归一化则旨在对同一Token的特征维度进行归一化处理，使其特征分布保持相对稳定。该操作能够有效减弱不同网络层之间数据分布变化所带来的负面影响，有助于稳定激活值的分布范围，从而加快模型的收敛过程，提高整体训练效率。

综上所述，Add+Norm模块主要发挥两方面作用：残差连接负责保留原始输入信息并促进梯度的高效传播；层归一化则负责稳定特征分布并进一步提升训练过程的稳定性。二者相互配合，使Transformer即使在网络层数较深的情况下，仍能够保持良好的训练效果和优化性能。

在了解Add+Norm模块的整体作用之后，下面将进一步剖析其具体的计算过程，如图1-33所示。

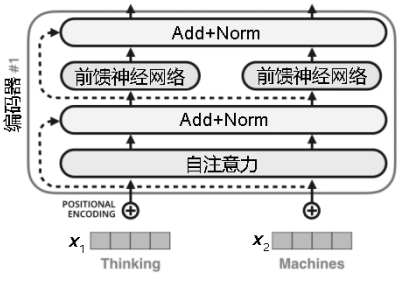


图 1-32　Add+Norm 模块在编码器中的分布

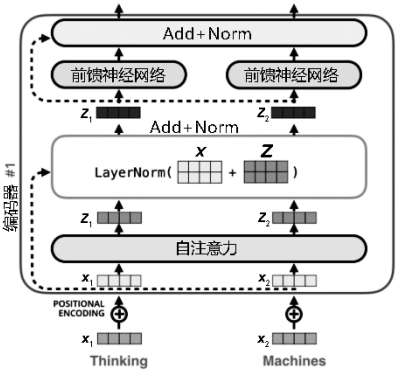


图 1-33　Add+Norm 模块计算细节

对于Transformer中的任意一个子层，输入$\boldsymbol{X}$会先经过自注意力层或前馈神经网络层得到子层输出；与此同时，原始输入$\boldsymbol{X}$会通过残差连接保留下来，并与子层输出进行逐元素相加（Add）。相加后的结果随即被送入层归一化模块进行处理（Norm），从而得到Add+Norm模块的最终输出。其计算形式可表示为：

$$\text{AddNorm}(\boldsymbol{X}) = \text{LayerNorm}(\boldsymbol{X} + \text{Sublayer}(\boldsymbol{X}))$$

其中，Sublayer($\boldsymbol{X}$)表示某个子层的输出，既可以是自注意力层的输出，也可以是前馈神经网络层的输出。

LayerNorm（层归一化）的计算过程为：先针对每个Token对应的全部特征维度计算均值与方差，再通过可学习的缩放参数与偏移参数对特征进行变换。其计算形式可表示为：

$$\text{LayerNorm}(x_i) = \gamma_i \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta_i$$

其中，$x_i$表示单个Token向量中的第$i$维特征值；$\mu$和$\sigma^2$分别表示该Token全部特征维度对应的均值与方差；$\varepsilon$是一个极小的常数，用于避免分母为0；$\gamma_i$和$\beta_i$分别为对应维度的可学习缩放参数和偏移参数。

需要说明的是，Add+Norm模块不仅应用于编码器，也应用于解码器，其在解码器中的分布位置如图1-34所示。

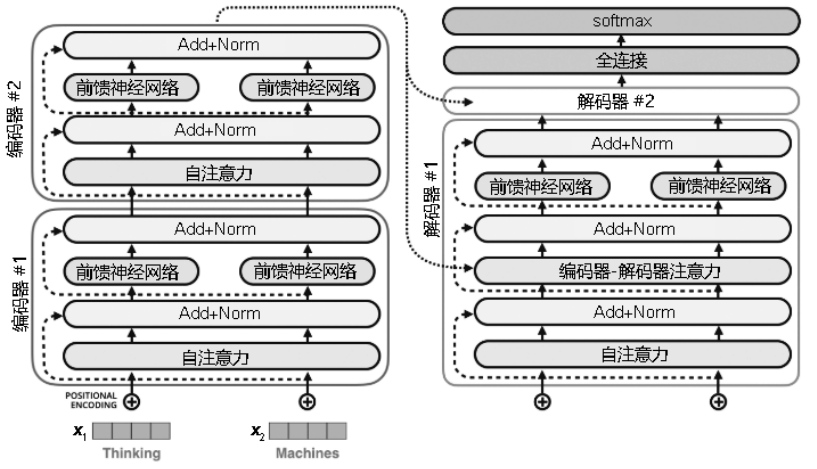


图 1-34　Add+Norm 模块在解码器中的分布位置

8）Transformer 多头注意力机制

前文主要以单头注意力机制为核心，阐述了自注意力机制的基本计算过程。但在Transformer架构中，通常采用多头注意力机制。多头注意力是单头注意力的扩展形式，它通过多个注意力头从不同表示空间中学习Token之间的关系，从而提升模型对序列信息的建模能力。Transformer的多头注意力工作机制如图1-35所示。

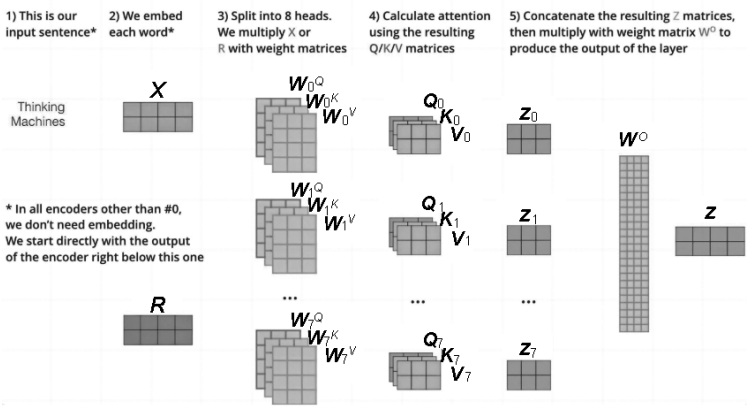


图 1-35　Transformer 多头注意力机制

多头注意力机制的重要性主要体现在以下两个方面：

（1）增强模型关注不同位置的能力。

单头注意力在单次计算中只能形成一种注意力分布，而多头注意力通过多个独立的注意力头，可以同时从不同维度关注输入序列中的不同位置，从而更全面地建模Token之间的复杂关系。

例如，在句子“The animal didn’t cross the street because it was too tired”中，如果需要判断代词it指代的对象，多头注意力机制可有效发挥作用。不同的注意力头可分别建模animal、street等相关词汇与it的关联特征。最终，模型能够更完整地捕捉句子中的语义联系，从而判断it更可能指代animal。图1-36展示了it在多头注意力中的关联情况。

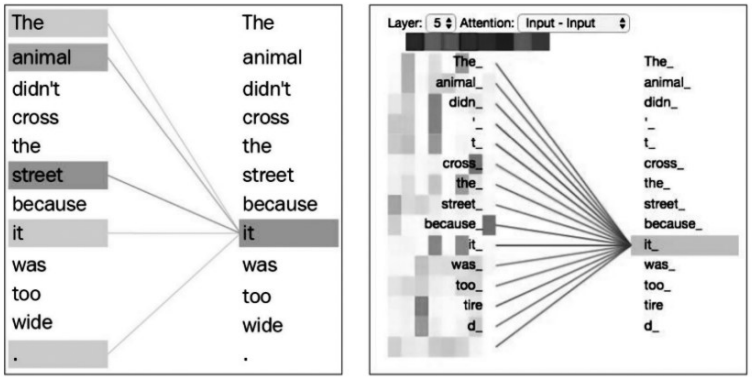


图 1-36　it 在多头注意力中的指代

（2）为注意力层提供多个表示子空间。

多头注意力的另一个核心特点在于，它并不局限于使用一组$\boldsymbol{W}^Q$、$\boldsymbol{W}^K$和$\boldsymbol{W}^V$权重矩阵，而是同时使用多组不同的权重矩阵。也就是说，每一个注意力头都拥有各自独立的$\boldsymbol{W}^Q$、$\boldsymbol{W}^K$和$\boldsymbol{W}^V$，并将输入向量映射到不同的表示子空间中进行注意力计算。

这样设计的优势在于，不同的注意力头可以学习到不同维度的特征关系。例如，有的注意力头关注局部依赖，有的关注长距离依赖；有的关注语法关系，有的关注语义关系。正因为各个注意力

头工作在不同的表示子空间中，多头注意力才能比单头注意力捕获到更加丰富的上下文信息。图1-37展示了多头注意力中多组$W^Q$、$W^K$和$W^V$权重矩阵的并行结构。

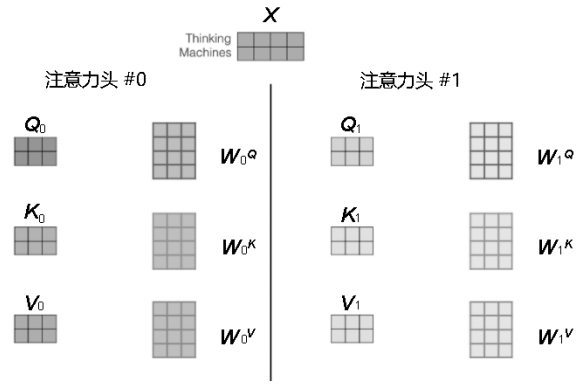


图 1-37　多头注意力拥有多组 $W^Q$、$W^K$ 和 $W^V$ 权重矩阵

假设Transformer架构使用8个注意力头，那么输入向量会分别经过8组不同的权重矩阵投影，进而完成8次独立的注意力计算。这样，模型最终会得到8个不同的输出矩阵，通常记为8个不同的$Z$矩阵，如图1-38所示。

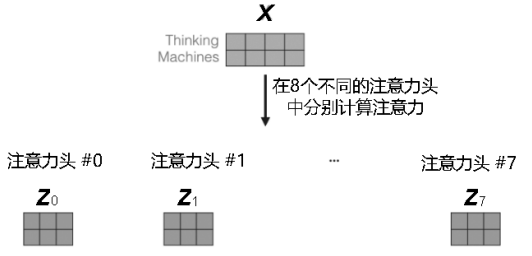


图 1-38　8 头注意力得到 8 个不同矩阵

然而，后续的前馈神经网络层并不希望接收8个彼此独立的输出矩阵，而是需要一个统一的输出矩阵。因此，必须对这8个输出矩阵进行整合。常见做法是：先将这8个矩阵按列进行拼接，形成一个更大的矩阵；然后乘以输出权重矩阵$W^O$，将拼接后的结果映射回原来的维度，最终得到统一的输出矩阵$Z$。这一整合过程如图1-39所示。

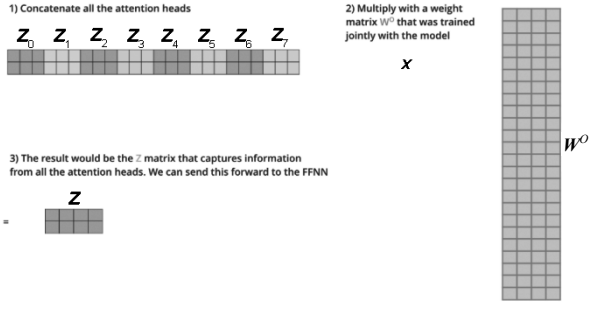


图 1-39　8 个不同矩阵经过拼接并乘以权重矩阵 $W^O$ 后得到一个输出矩阵 $Z$

9）Transformer 解码器

解码器在整体结构上与编码器相似，但在注意力机制与输出处理逻辑上存在差异，核心区别主要体现在以下三点：

- 解码器采用掩码自注意力（Masked Self-Attention）机制。
- 解码器引入交叉注意力（Cross-Attention）机制。
- 解码器的输出还需要经过线性变换与softmax计算，以生成最终的预测结果。

（1）掩码自注意力。

掩码自注意力仅允许当前位置的Token关注自身及序列中的前置位置Token，而不能关注后续位置的Token。具体而言，在计算第$i$个位置的特征时，模型只能利用第1~$i$个位置的信息，无法访问第$i$+1及之后位置的信息。这是它与自注意力机制的主要区别，如图1-40所示。

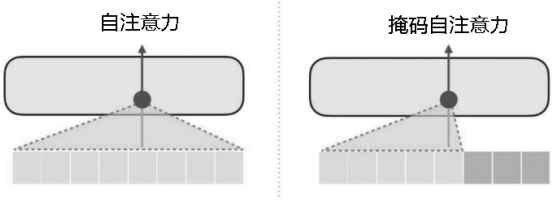


图 1-40　掩码自注意力与自注意力的不同之处

该设计的核心目的是适配自回归的解码过程。以机器翻译任务为例，模型需要逐位生成目标词，只有在生成第$i$个词之后，才能继续生成第$i$+1个词。因此，在生成当前词时，模型不应提前获取后续位置的信息。通过掩码约束，可以有效避免当前位置“读取未来信息”，从而保证解码过程符合自回归生成的基本规则。

掩码约束通过掩码矩阵实现，其结构如图1-41所示。在掩码矩阵中，对角线及以下位置的元素取值为0，表示这些位置允许关注；对角线以上位置的元素取值为负无穷，表示这些位置禁止关注。由此确保每个位置仅能关注自身及前置位置，而不能关注后续位置。

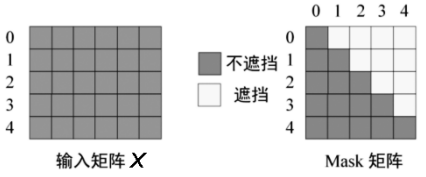


图 1-41　掩码矩阵

在标准自注意力计算公式中，并未包含掩码项；而在掩码自注意力中，需要在自注意力分数矩阵中加入掩码矩阵，具体公式如下：

$$\text{Attention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})=\text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_k}}+\text{Mask}\right)\boldsymbol{V}$$

其中，Mask表示掩码矩阵。

在该公式中，当自注意力分数矩阵与掩码矩阵Mask相加后，允许关注的位置加值为0，因此分数保持不变；而禁止关注的位置加值为负无穷，因此分数将趋于负无穷。

随后经过softmax函数计算。由于softmax函数包含指数运算$e^x$，当$x$趋近于负无穷时，$e^x$的值趋近于0。因此，被掩码的位置在softmax函数计算后，对应的自注意力权重将趋近于0。通过这一机制，模型在计算当前Token表示时，不会为被掩码的位置分配自注意力权重，从而避免引入未来位置的信息。

通过上述掩码机制，掩码自注意力能够确保解码器在生成当前Token时，仅依赖当前及前置位置已生成的信息，从而满足自回归序列生成任务的核心约束。

（2）交叉注意力。

在掩码自注意力层中，解码器只能关注当前及前置位置已经生成的Token序列；而在交叉注意力层中，解码器会进一步关注编码器的输出，从而获取源序列中的上下文信息。

具体而言，交叉注意力中的查询矩阵$\boldsymbol{Q}$来自解码器当前层的输入，而键矩阵$\boldsymbol{K}$和值矩阵$\boldsymbol{V}$均来自编码器的最终输出，其结构如图1-42所示。解码器会根据当前已生成的目标序列内容，从编码器输出中检索与当前生成位置最相关的信息。

以机器翻译任务为例，解码器在逐位生成目标语言词时，需要匹配源语言句子中与当前生成位置相关性最高的语义片段。交叉注意力的核心作用在于建立目标序列与源序列之间的语义对齐关系，使模型能够融合编码器提供的语义信息，从而生成更加准确的结果。

（3）输出层。

经过多层解码器计算后，模型会输出当前位置的特征向量。该向量是模型对当前上下文信息的融合表示，其本身并不对应具体单词，因此不能直接作为最终输出结果。

为了生成下一个Token，模型需要先通过线性层将该输出向量映射到词表空间。词表空间是一个维度与词表大小一致的向量空间，词表中的每个Token对应一个维度。线性层的核心作用是为词表中的每个候选Token生成一个分数，该分数通常称为logits。

随后，模型通过softmax函数将logits转换为概率分布，用于表示每个Token作为下一个输出的概率。得到概率分布后，模型可通过不同的解码策略选取下一个Token：最基础的策略为Argmax，即选取概率最大的维度对应的索引，再根据该索引到词表中查找对应的词，作为当前位置生成的Token。解码器输出层的计算过程如图1-43所示。

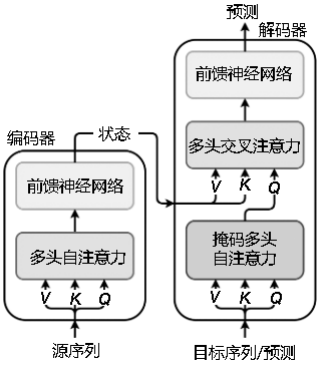


图 1-42　交叉注意力

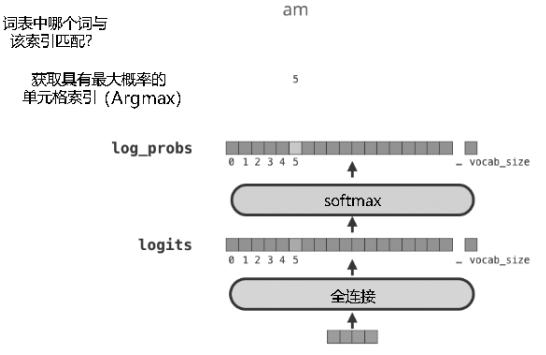


图 1-43　解码器输出层

10）Transformer 分类

根据编码器和解码器的组合方式，Transformer 可演化为三类架构模式。

（1）Decoder-only架构：仅保留Transformer中的解码器模块，其核心任务是基于已生成的文本序列预测下一个Token。典型代表为GPT系列模型。

（2）Encoder-only架构：仅保留Transformer中的编码器模块，主要用于对输入文本进行语义理解与表示学习，不具备直接生成文本的能力。典型代表为BERT等模型。

（3）Encoder–Decoder架构：为Transformer的原始形态，同时包含编码器与解码器模块，适用于序列到序列（Seq2Seq）任务，典型代表为T5等模型。

三类 Transformer 架构的设计目标各有侧重：

（1）Decoder-only架构在大规模文本生成与通用语言建模任务中占据主导地位。

（2）Encoder-only架构专注于语言理解类任务。

（3）Encoder–Decoder架构适用于输入到输出的序列映射任务。

随着模型规模的不断扩大，Decoder-only 架构逐渐成为当前 LLM 的主流技术路线，但其他两类架构仍在特定任务中发挥着不可替代的作用。

11）Transformer 小结

Transformer是一种以自注意力机制为核心的神经网络架构，主要用于对序列数据进行高效建模。该架构通过自注意力机制捕获序列中任意位置之间的依赖关系，同时引入位置编码以补充序列的顺序信息，从而在不依赖循环或卷积结构的情况下，实现对上下文的全局建模。

从结构上看，Transformer由编码器和解码器两部分组成。编码器通过自注意力机制完成对输入序列的表示学习，解码器则结合掩码自注意力和交叉注意力，以自回归方式逐步生成输出序列。多头注意力机制使模型能够在不同表示子空间中并行关注多种依赖关系，前馈神经网络、残差连接与层归一化则进一步提升了模型的表达能力与训练稳定性。

与传统的循环神经网络和卷积神经网络相比，Transformer具备更强的并行计算与长距离依赖建模能力，是支撑现代LLM快速发展的核心技术基石。

## 1.2　AI Agent：从“对话”到“行动”的进化

最初，LLM（大语言模型）被视为一种强大的文本生成工具，其核心能力体现为对序列中下一个Token的预测。然而，随着模型参数规模的持续扩大和训练技术的不断迭代升级，业界对LLM的认知在2024年至2025年间发生了显著转变：从应用形态的演进维度来看，LLM正在从简单的对话机器人（ChatBot）逐步演进为能够自主规划、调用工具并执行复杂任务的智能系统，即AI Agent。

### 1.2.1　什么是 AI Agent

AI Agent通常被译为“AI智能体”，简称智能体或Agent。从广义上看，凡是能够感知环境、理解目标、做出决策并采取行动的智能系统，均可称为AI Agent。例如，智能客服系统、编码智能体系统、具身智能机器人等，都可以视为不同形态的AI Agent。

从运行过程来看，AI Agent通常包含“理解目标-制订计划-调用工具-观察结果-修正策略-输出结果”等环节。例如，当用户提出“帮我分析本月销售数据并生成报告”时，普通LLM可能仅给出分析思路或报告模板，而AI Agent则可以进一步连接数据库、提取数据、执行统计分析、生成图表和报告，并根据用户反馈持续优化生成结果。

综上，AI Agent的核心特征可概括为：它并非为单纯的文本问答系统，而是一类可围绕既定目标持续行动的智能系统。

### 1.2.2　AI Agent 的本质

AI Agent的本质体现为LLM推理决策能力与工具调用能力的深度结合。

LLM是AI Agent的“大脑”，主要承担用户意图识别、任务目标拆解、行动方案生成与后续操作决策等职责；工具则是AI Agent连接外部世界的“手脚”，用于实现模型自身无法直接完成的各类操作。

如果没有LLM，AI Agent将难以理解复杂的自然语言需求，也难以进行开放式推理与灵活决策；如果没有工具，LLM即使能够输出合理的方案或建议，也仅停留在文本输出层面，无法深度对接外部系统完成实际任务。

因此，AI Agent的关键在于使LLM能够借助工具与外部环境发生交互，从而具备从理解目标到完成任务的完整能力。这也是AI Agent相较于传统LLM应用的核心优势所在。

### 1.2.3　AI Agent 的关键技术

AI Agent的快速兴起得益于多项关键技术的持续突破。这些技术不仅显著提升了LLM（大语言模型）在真实场景中的实用价值，也推动其逐步具备复杂任务处理、多步推理决策与外部系统互联的综合能力。以下将围绕这些关键技术展开说明。

#### 1. 工具调用与 MCP

LLM原生能力主要集中在语言理解与文本生成，其自身并不具备访问数据库、调用业务API或执行外部操作的能力。这一局限导致其难以在真实业务流程中独立完成复杂任务。工具调用（Tool Calling）技术的出现，使LLM能够根据任务需求判断是否调用外部工具，并生成符合规范的结构化请求，以此实现对业务API、数据库、搜索引擎及计算服务的访问。

这种能力扩展使LLM从单纯的“文本生成模型”逐步演进为能够连接外部系统、驱动业务流程的“智能控制器”。例如，在出行Agent系统中，模型既可以调用地图API完成路径规划，也可以调用搜索引擎获取实时天气信息，从而完成更贴近真实业务场景的复杂任务。

模型上下文协议（Model Context Protocol，简称MCP）的提出，进一步推动了工具接入方式的标准化。MCP采用客户端—服务器架构，实现了AI应用与外部工具、数据源之间的解耦：MCP服务器负责对外开放工具、资源与提示模板等能力，MCP客户端负责与服务器建立连接、发现可用能力并发起调用请求。基于该架构，开发者无须为每个模型或应用单独编写定制化集成逻辑，只需遵循统一协议封装外部能力，即可使不同Agent系统以更加规范的方式接入工具生态。

与传统工具调用方式相比，MCP的核心价值在于显著降低工具集成、能力发现与系统扩展的复杂度。在传统工具调用模式下，开发者需要将工具的功能说明、参数格式与调用规则硬编码到系统

提示词或应用逻辑中；而在MCP架构下，工具的名称、功能描述与输入模式（Schema）由服务器端统一对外提供，客户端按需获取并提交给模型使用，从而有效降低Agent接入新工具与数据源的技术门槛。

### 2. 记忆机制

传统LLM的对话交互通常是无状态的，模型只能依赖当前上下文窗口识别用户意图。当对话内容超出上下文窗口范围时，模型容易丢失前文的交互信息，难以对长期任务进行持续跟踪。记忆机制的引入，使Agent能够在多轮交互与长期任务执行过程中保存关键信息，从而实现更具连续性、个性化和任务导向的交互表现。

从作用范围来看，Agent 的记忆体系通常可分为短期记忆和长期记忆两类：

- 短期记忆：主要依赖上下文窗口，用于保存当前对话状态、任务目标、中间推理过程以及临时变量，帮助Agent在一次任务执行过程中保持连贯性，避免在多步骤推理中丢失上下文信息。
- 长期记忆：通常结合向量数据库、检索增强生成（Retrieval-Augmented Generation，RAG）、结构化数据库或知识图谱等方式实现，用于存储用户偏好、历史任务、项目背景与领域知识等长期信息。当Agent需要处理新任务时，可以通过语义检索或结构化查询等方式重新获取相关上下文信息，再将其注入到当前推理过程。

上述记忆机制有效扩展了 LLM 的上下文窗口边界，使 Agent 不再局限于单次对话，而是能够持续跟踪跨天、跨周甚至跨月的复杂任务，为后续的规划、执行与反馈优化等环节提供稳定的上下文支撑。

### 3. 推理框架

为提升 Agent 的复杂任务处理能力，目前已发展出多种推理框架，包括但不限于 Chain of Thought（CoT，思维链）、Self-Consistency（SC，自洽性）、ReAct（Reason + Act）以及 Tree of Thoughts（ToT，思维树）等，如表 1-1 所示。此类框架旨在提升 LLM 在复杂任务中的推理可靠性与执行稳定性。

表 1-1　常见的多种推理框架

| 推理框架 | 核心原理 | 对 LLM 边界的扩展 |
| --- | --- | --- |
| Chain of Thought（CoT） | 引导模型分步骤生成中间推理过程 | 使模型能够处理需要多步推导的复杂任务 |
| Self-Consistency（SC） | 生成多条候选推理路径，并通过一致性投票选择最终结果 | 提升了模型在复杂推理任务中的稳定性和可靠性 |
| ReAct（Reason + Act） | 将推理过程与外部工具调用、环境反馈相结合 | 使模型能够在观察外部结果后动态调整计划，减少仅依赖内部知识带来的幻觉 |
| Tree of Thoughts（ToT） | 维护多个候选推理分支，并对不同路径进行评估和回溯 | 支持复杂规划、多路径搜索和非线性问题求解 |

### 4. 上下文工程与 Agent Skills

随着Agent任务复杂度的持续提升，LLM应用的开发范式正从提示工程（Prompt Engineering）

逐步转向更具系统化的上下文工程（Context Engineering）。提示工程更关注“如何写好提示词”，即聚焦于提示词的设计与优化，而上下文工程关注“如何在有限上下文窗口中，为模型动态组织任务所需信息”。其核心目标并非将所有资料一次性输入模型，而是在合适的时机，将任务最相关的背景知识、工具说明、执行规范与任务状态提供给模型。

Agent Skills是上下文工程的重要实践形式。Anthropic于2025年10月提出Agent Skills，用于将特定任务中的程序性知识封装为可复用的技能单元。一个Skill通常由SKILL.md文件以及可选的脚本、参考资料、模板等资源组成，用于指导Agent在特定场景下完成对应类型的任务。其核心运行机制为渐进式披露：Agent执行任务时，通常只加载每个Skill的名称和描述，用于判断它是否与当前任务相关；当模型判断当前任务需要调用某项技能时，才会进一步完整读取该技能对应的SKILL.md内容，并在必要时按需加载相关参考文件、脚本与模板。通过这种分层加载机制，Agent Skills既能保留复杂任务所需的专业流程与细节规范，又能避免大量无关信息一次性占满上下文窗口空间，从而有效提升上下文窗口的利用率与任务执行的整体稳定性。

### 5. Agent 开发框架

为了降低AI Agent的开发门槛，GitHub开源社区陆续推出了多种Agent开发框架。这些框架在任务编排、状态管理、工具调用、多Agent协作和执行控制等方面各有侧重。表1-2对当前常见的Agent开发框架进行了对比。

表 1-2　Agent 开发框架对比分析

| 框　架 | 设计理念 | 优　势 | 局 限 性 |
|---|---|---|---|
| LangGraph | 基于有向图的状态机架构 | 可控性强，支持显式状态流转、复杂循环和错误恢复 | 学习曲线较陡，对初学者不够友好 |
| CrewAI | 模仿人类团队的角色协作 | 易于定义专业角色，适合任务分工和协作流程 | 长程任务中协调成本较高，监控复杂 |
| AutoGen | 基于多 Agent 对话的协作 | 适合多 Agent 对话、代码执行和协作推理场景 | 架构迭代较快，迁移和维护成本较高 |
| AutoGPT | 追求完全自主循环的架构 | 适合探索性任务和自我驱动的问题求解 | 容易出现循环失控，Token 消耗和错误累积成本较高 |

### 6. Agent2Agent（A2A）协议

随着Agent系统从单一应用走向跨组织、跨厂商和跨平台协作，Agent之间的互操作性成为新的关键问题。Google于2025年提出Agent2Agent（A2A）协议，旨在为不同框架、不同平台构建的Agent提供统一的通信与协作方式，使它们能够安全地交换信息、协调任务并共同完成复杂流程。

A2A 协议的核心包括三个方面。

- 第一个是Agent Card，即以JSON格式描述Agent身份、能力范围、服务端点、支持模态和认证要求的元数据，相当于Agent的“能力说明书”。
- 第二个是任务管理。协议围绕任务生命周期组织协作流程，支持submitted、working、input-required、completed、failed、canceled等状态，从而适配长时序、异步化的Agent协作场景。

- 第三个是安全性与隔离。A2A协议将远程Agent视为相对独立的不透明实体，协作双方只需交换任务、消息和结果，而不必暴露内部推理过程、记忆结构或具体工具实现，从而有助于保护企业知识产权和数据隐私。

在一个典型的 A2A 协议场景中，一个面向用户的旅行规划 Agent 可以作为客户端，发现并调用部署在不同服务器上的订票 Agent、酒店 Agent 或支付 Agent，共同完成行程规划、酒店预订与支付确认等任务。由此可见，A2A 协议的价值不在于替代单个 Agent 的能力，而在于为多 Agent 系统提供跨平台协作的标准化基础。

**7. 检索增强生成和向量数据库**

检索增强生成是LLM应用中的重要技术范式，主要用于解决LLM知识更新不及时、专业知识覆盖不足以及回答缺乏依据等问题。其核心思路是：在模型生成答案之前，先从外部知识库中检索与用户问题相关的内容，再将检索结果与用户问题一同输入LLM，由模型基于外部知识生成更准确、更可靠的回答。在AI Agent中，检索增强生成可以为Agent提供外部知识支撑，使其在执行任务时能够检索企业文档、产品手册、历史工单、业务规则等信息，从而提升任务处理的准确性和可解释性。

向量数据库是检索增强生成系统中的重要基础设施，主要用于存储和检索经过向量化处理的文本、图片、音频等数据。与传统关键词检索模式不同，向量数据库基于语义相似度进行内容匹配，能够检索到与用户问题在语义上相关的内容，而非仅依赖字面关键词的匹配结果。在AI Agent架构中，向量数据库通常承担知识存储与语义检索的角色，帮助Agent快速定位相关知识，并将检索结果作为后续推理、规划与执行的重要上下文。

### 1.2.4　AI Agent 的五级演化

从当前行业实践和技术能力演进来看，AI Agent的发展可概括为五个演化等级，如表1-3所示。这些等级体现了AI Agent在自主性、任务复杂度、工具使用能力和协同能力等方面的逐步提升。

表 1-3　AI Agent 经历的五级演化

| 演化等级 | 名　　称 | 核心能力说明 | 技术特征 |
|---|---|---|---|
| Level 1 | 通用对话 | 能够理解用户意图并生成高质量文本，但主要运行于封闭的对话环境中 | 预训练 LLM，提示词工程，上下文理解 |
| Level 2 | 领域专家 | 面向特定行业、业务场景或知识库进行优化，具备较强的专业知识理解与问答能力 | 微调，检索增强生成，向量数据库，知识库构建 |
| Level 3 | 任务智能体 | 能够根据任务目标调用外部 API、工具或数据库，完成信息检索、数据处理和任务执行 | 函数调用，MCP，工具调用，CLI 调用，Agent Skills（智能体技能） |
| Level 4 | 自主规划 | 能够自主理解目标、自主规划和执行，并基于反馈进行自我反思与持续迭代 | 自我反思，自主规划，长期记忆，反馈迭代 |
| Level 5 | 原生组织 | 由多个 Agent 协同运行，共同完成跨系统、跨流程的复杂任务，实现组织级业务自动化 | 多 Agent 系统（MAS），A2A 协议，智能体通信协议（Agent Communication Protocol，ACP） |

截至目前，大多数 AI Agent 仍停留在 Level 3，只有少数 Agent 初步达到 Level 4。Level 3 阶段的 AI Agent 已具备在明确目标下进行工具调用和任务执行的能力，而 Level 4 的 AI Agent 则进一步体现出主动探索能力，能够自主理解目标、拆解任务、规划路径、实施方案，并根据反馈持续优化结果，从而推动人机协作迈向更加主动、智能和高效的新阶段。

## 1.3　AI Agent 核心转变

与传统被动响应的LLM应用不同，AI Agent更强调主动性和闭环交互。一个AI Agent不仅能理解指令并给出响应，还能够自主感知环境、思考决策、主动行动并对执行结果进行反馈与调整，从而形成“感知-思考-行动-反馈”的认知闭环。

### 1.3.1　AI Agent 的认知闭环

AI Agent之所以比传统LLM应用更为智能，本质在于其内部形成了完整的“感知-思考-行动-反馈”的认知闭环。图1-44展示了一个典型的AI Agent在任务执行过程中的闭环流程。


图 1-44　AI Agent 执行任务时的闭环流程

AI Agent首先通过感知环节接收任务信息，例如理解用户目标、解析问题描述、获取多模态输入或读取外部环境数据；随后进入思考环节，对任务目标进行理解、分析与推理，并进一步拆解子任务、规划执行路径；在行动环节，AI Agent根据规划结果执行具体操作，这些操作可能包括调用外部工具、访问API、查询数据库，或与其他Agent进行协作。行动完成后，系统会将执行结果反馈给AI Agent，由AI Agent基于新的信息判断当前方案是否有效，并决定是否进入下一轮调整与执行。由此形成“感知-思考-行动-反馈”的认知闭环。该循环会持续迭代，直到任务完成或达到预设的终

止条件。

### 1.3.2　AI Agent 的闭环模块

AI Agent的闭环流程通常涉及记忆、推理与规划、工具使用和自我反思等核心模块。通过这些模块的协同配合，AI Agent在处理复杂任务时能够更加稳定、可靠和高效。下面将对这些模块的基本原理与作用进行详细介绍。

#### 1. 记忆模块

记忆模块是Agent维持上下文连续性的重要组成部分。由于模型存在上下文窗口限制，超过窗口的信息会丢失，而Agent通过引入外部记忆机制，可以突破这一限制，对历史交互、任务状态和关键结果进行保存与管理，从而在较长任务周期内延续上下文，并为后续推理和行动提供依据。

记忆通常分为短期记忆与长期记忆。其中，短期记忆对应当前任务或会话的上下文信息，由LLM的提示词或暂存变量承载，容量有限但访问迅速；长期记忆则存储Agent历史交互过程中的重要信息、知识和策略，一般通过向量数据库等技术实现，以便随时按需检索。例如，在对话Agent中，短期记忆通常包括近期几轮的对话内容，而长期记忆可以是用户偏好、历史交互事实等长效信息。混合使用短期记忆与长期记忆，使Agent既不会遗忘当前情境，又能举一反三地利用过往经验，实现知识复用与能力迁移。

#### 2. 推理与规划模块

推理与规划模块是AI Agent处理复杂任务的核心组成部分。其中，推理主要负责理解任务目标与分析问题；规划则负责将复杂目标拆解为若干可执行的子任务，并按照一定的逻辑顺序组织为步骤序列。二者相互配合，使AI Agent不仅能够理解“要做什么”，还能够进一步明确“如何去做”。

在实际运行过程中，AI Agent通常会先根据用户目标、当前上下文和外部环境信息进行分析判断，再制订相应的执行计划。例如，当面对一个多步骤任务时，AI Agent需要识别任务包含的关键环节、每一步需要调用的工具、依赖的中间结果，以及在执行过程中如何根据反馈动态调整后续方案。通过推理与规划能力，AI Agent可以更加稳定地处理流程较长、依赖关系较复杂或结果不确定的任务。

#### 3. 工具使用模块

工具使用模块是AI Agent执行任务的重要功能组件，其核心作用在于通过标准化接口与外部环境进行交互，从而获取实时信息或执行具体操作。

典型的工具类型包括：检索类（如搜索引擎查询、知识库问答）、计算类（如调用计算器、执行代码片段）、信息服务类（如天气查询、数据库读写）等。

通过这些工具接口，AI Agent不仅能够弥补LLM在知识时效性方面的不足，还能够执行检索、计算、代码运行、数据库读写等具体操作，从而具备完成实际任务的能力。

工具使用赋予了AI Agent强大的扩展能力，但同时也带来了一些挑战。例如，如何让模型选择合适的工具，避免接口滥用，以及如何处理工具调用过程中产生的错误等问题。为此，AI Agent通常会结合推理与规划模块，在调用工具前后进行逻辑检查与结果评估，并在必要时重试或调整策略。

#### 4. 自我反思模块

真正高级的AI Agent不仅能够执行预定策略，还能够评价自身行为，从而实现持续改进。自我反思模块使AI Agent具备一定程度的元认知能力，即对自身的思考过程、决策过程与执行结果进行审视。在实践中，这意味着AI Agent会在每轮行动后回答若干内部问题：当前步骤是否成功？是否存在理解偏差或执行错误？是否需要调整策略？根据反思结果，AI Agent可能采取纠错行动（如尝试不同方法、重新调用工具等），或在下一次规划时避免重复错误。

自我反思通常通过提示工程实现。例如，LangChain提供的Reflection模式，会在Agent完成任务后追加让模型自我检讨的Prompt（提示词），促使其指出自身输出中的问题并给出改进建议。另一种做法是引入专门的审查者Agent，与执行Agent协同工作：执行Agent完成一步后，审查者Agent负责分析结果是否存在偏差，并将改进意见反馈给执行Agent。这一机制类似于人类团队中的复核与评审过程。

相关研究也验证了自我反思机制在提升AI Agent可靠性方面的重要价值。Shinn等人在2023年提出的Reflexion框架，使Agent在循环过程中持续生成自我反馈，并将有价值的反思写入记忆，从而逐步提高问题求解的成功率。研究表明，采用Reflexion策略的Agent在复杂推理任务上的表现显著提升，能够更有效地修正早期步骤中的疏漏。同样，从Generative Agents的模拟实验中也可以看出，Agent会反思近期记忆以生成更高层次的洞见，并将其储存为后续行为的参考依据。总之，反思机制为AI Agent提供了一种自我监督能力，使其具备一定的自适应纠错与持续改进特性。这种能力在开放环境中尤为关键，有助于Agent在未知情境下逐步校准方向，使最终结果不断逼近目标。

### 1.3.3　AI Agent 的闭环影响

AI Agent的认知闭环带来的直接影响，是人机交互范式与系统架构发生了深刻变化。

从交互方式来看，传统LLM应用（如对话机器人）主要扮演应答者的角色：用户提出问题，LLM输出对应回答。对于复杂问题，用户往往需要逐步引导LLM完成求解，LLM本身并不会主动探究问题是否已得到解决。而在AI Agent范式下，用户只需设定目标或任务，Agent便会自主展开一系列子任务和操作，直至最终完成目标。此时，交互方式从“逐步询问、回答”转变为“一次委托、持续自主执行”。对用户来说，这更像在指挥一名智能助理，而非对每一步进行细致指令控制。例如，让传统LLM回答“过去十年美国人均卡路里摄入的趋势及其对肥胖率的影响”这一复杂问题时，用户通常需要多次提问并自行整合信息；而AI Agent在接收该综合任务后，可以根据任务目标自主查询数据、生成图表并撰写分析报告，最终一次性输出完整结果。由此可见，AI Agent将用户从烦琐的交互过程中解放出来，使AI服务从被动响应进一步走向目标驱动的主动执行。

从系统架构来看，AI Agent范式引入了更多功能组件和更复杂的执行流程，超越了传统一次调用的LLM应用架构。图1-45展示了传统LLM问答应用与AI Agent的异同。

对于传统LLM应用，系统通常遵循相对固定的推理步骤：输入通过Prompt进入LLM，LLM立即生成输出并直接返回。而在AI Agent架构中，系统往往被拆分为“决策中枢”和“行动执行”两个相互交替的部分。其中，决策中枢通常以LLM为核心驱动，负责任务理解、规划生成以及下一步行动决策；行动执行部分则通过工具接口与外部环境进行交互，并将观测结果反馈给决策中枢。因此，AI Agent在执行过程中多次调用LLM进行思考与决策，并在每次决策后触发相应动作，再根据执行

结果更新内部状态，决定后续步骤。这意味着AI Agent架构需要引入一个循环控制逻辑或调度器（Orchestrator）来管理这一系列步骤的流转。这实质上是将过去隐含在单轮Prompt中的推理过程，外显为可监控、可调试和可优化的程序化执行流程。

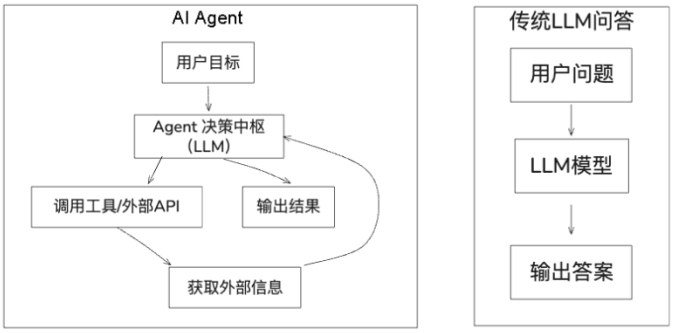


图 1-45　传统 LLM 问答应用与 AI Agent 的异同

由于AI Agent架构引入了循环、自主分支和状态更新等机制，对底层开发框架提出了更高要求。一方面，框架需要支持长时间运行与状态持久化，例如在多轮交互中保留上下文、维护长期记忆并记录中间执行结果；另一方面，框架还需要具备更灵活的流程控制能力，例如根据不同条件选择合适的工具、判断是否需要重试，或决定是否需要引入其他Agent协同完成任务等。这一变化推动AI Agent开发框架从简单脚本逐步演进为图式工作流或事件驱动架构。例如，LangChain早期的Agent Executor只是一个Python循环，难以应对复杂分支。为此，LangChain推出了LangGraph库，引入有向图、状态管理和循环节点等机制，用于描述与编排Agent的执行流程。借助LangGraph，开发者可以更灵活地构建多Agent协作、条件跳转和复杂拓扑结构的决策流。同时，LangGraph也支持在节点间传递和更新状态，并提供状态持久化和实时监控的能力，从而保持长期对话的一致性。再如微软的AutoGen框架在v0.4版本中引入异步事件驱动架构，通过事件队列管理Agent之间的消息通信和工具调用，使多Agent并行协作和更灵活的对话流程编排成为可能。这些架构演进显著提升了AI Agent在闭环执行流程中的可扩展性、可观测性与可靠性。

## 1.4　现实中的应用场景

本节将围绕AI Agent在办公自动化、客户服务、科学研究与内容创作四个具有代表性的应用场景展开分析，重点探讨其提升效率的具体方式与典型案例。

### 1.4.1　办公领域中的 AI Agent 应用

办公领域是 AI Agent 落地应用的重要场景之一。AI Agent 可以充当数字助理，帮助用户处理烦琐的日常事务，减少重复劳动，提升办公效率。典型应用包括：

（1）日程与会议助理：AI Agent可自动安排日程、预订会议，并在会后生成会议纪要和待办事项。一些先进的会议助理能够实时记录会议对话要点、识别行动项，并在会议结束时输出摘要和任

务清单，从而避免人工逐字记录，极大节省时间。

（2）文档撰写与处理：借助嵌入办公软件的AI助手，用户可以通过自然语言让Agent起草报告、撰写邮件、制作PPT等。例如，Microsoft 365 Copilot深度集成于Word、Excel、PowerPoint、Outlook和Teams等办公应用中。用户可以要求Copilot总结会议记录、起草文档或生成演示文稿；也可以要求它根据聊天记录自动提炼邮件回复或任务列表，从而减少烦琐的书写与整理工作，节约大量时间。在实际应用中，财务团队可以借助Copilot快速生成包含图表和分析洞见的月度报告；市场团队可以快速起草宣传方案；HR部门可以高效整理入职培训材料。此类任务以往需要数小时才能完成，而在AI Agent的辅助下，如今往往只需几分钟即可完成。

（3）自动化工作流：通过将AI Agent与企业日常工具（如日历、邮件、办公套件和低代码平台）相结合，可以实现端到端的工作流自动化。例如，将Copilot与Power Platform的Power Automate服务相结合，企业能让Agent按照预设触发条件执行重复性流程，如每周定时生成报告并发送给相关人员，或监控邮箱自动提取发票信息并更新至表格。

图 1-46 是 AI Agent 架构示意图。对于用户通过自然语言输入的指令（如“汇总本周销售数据并发邮件给团队”），感知模块首先将其解析为具体意图和任务；接着，Agent 利用短期和长期记忆获取上下文（如用户的日历、企业数据库）；规划与推理引擎进一步将任务分解为可执行的子任务并生成候选执行方案；决策模块根据当前上下文、工具可用性和任务优先级决定调用何种工具；然后执行模块依次调用相应办公应用 API 或脚本（如从 Excel 提取数据、生成报告、发送邮件），完成各步骤操作；最后，执行结果进入反馈与学习闭环，用于更新上下文并修正后续动作。通过这种架构，AI Agent 能够清晰理解用户指令，并在办公系统中执行相应操作，从而提高工作流程的自动化程度与业务协同效率。

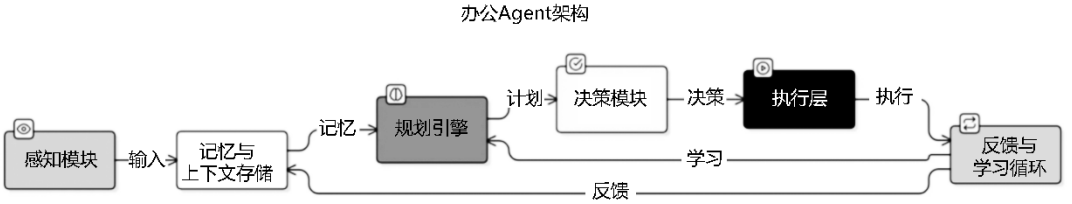


图 1-46　AI Agent 架构示意图

AI Agent的引入带来了显著的效率提升。一方面，它减少了人工在低价值重复劳动上的时间投入。据微软内部测试，Copilot在协助起草文档和回复邮件方面可为员工节省多达30%~50%的时间，使其能够将更多精力投入创意与决策等更高价值的工作中。另一方面，它还可以通过流程自动化降低人工操作的错误率。例如，在Excel数据分析场景中，Copilot可自动识别数据趋势并生成分析洞见，从而减少人为疏漏。

总体而言，办公领域的AI Agent以Copilot类集成模式为主，为企业提供了数字化劳动力的雏形，通过减负增效推动生产力提升。

### 1.4.2　客户服务中的 AI Agent 应用

客户服务是 AI Agent 商业落地最活跃的领域之一。智能客服 Agent 通过整合 LLM 的语言理解能力、企业知识库和业务系统 API，可以实现对客户咨询的自动应答、多轮对话引导以及后台流程处理，从而更快响应客户需求并降低人工客服成本。典型应用包括：

（1）检索增强的智能问答（RAG Chatbot）：传统FAQ（Frequently Asked Questions，常见问题解答）型客服机器人往往基于预设问答或规则模板，回答范围有限。而引入RAG架构的智能客服Agent，可以将用户提问与企业实时知识相结合，生成更加准确且有针对性的回复。其流程大致如图1-47所示。

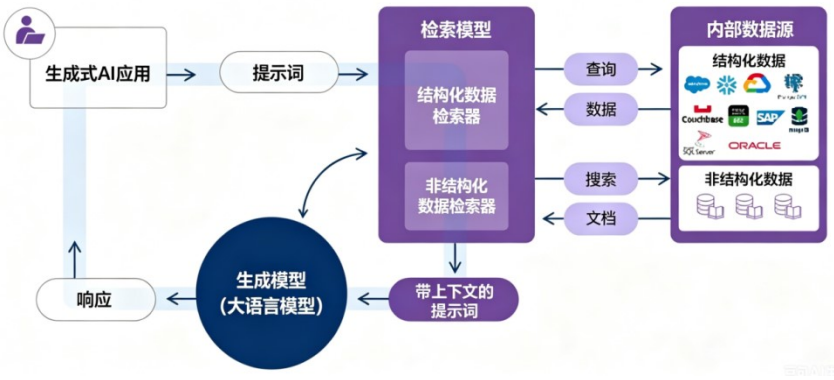


图 1-47　智能客服系统的 RAG 架构示意图

当用户提出问题（如“我想了解贵公司的退款政策”）时，Agent先通过检索模块在公司的内部知识库和数据库中查找相关信息（如退款条款文档、客户订单记录）；随后，将检索到的内容与用户原始提问一同构建扩展提示词，并输入LLM生成模块。由于提示词中包含了最新且权威的数据，LLM能够据此生成上下文相关且准确的答案。

RAG技术有效减少了模型“幻觉”问题，因为Agent不再仅依赖训练语料中的既有知识生成回答，而是结合实时检索到的事实依据进行回复。

这种智能问答 Agent 能够显著提升客服效率和服务质量。相关调研显示，在集成 AI 客服机器人的客服中心中，大量常见咨询可以由聊天机器人自动处理，从而降低人工客服压力与整体服务成本。同时，机器人还可以提供 7×24 小时在线支持，不受人力班次限制，真正实现“永不下线”的客户服务。以沃达丰电信的客服 Agent TOBi 为例，它能够处理账单查询、技术支持等多类客户咨询，在实际业务中承担了大量请求分流工作，有效降低了人工客服负载。

（2）多轮对话与意图识别：真实客服场景往往需要与用户进行多轮交互，逐步澄清问题并提供解决方案。客服Agent通过LLM和上下文记忆机制，实现对多轮对话的语境理解与用户意图的持续追踪。例如，当用户询问“账户无法登录”时，Agent可能需要经过多轮提问（如“请问您是否忘记密码？”“您绑定的邮箱是哪个？”）来进一步明确问题，并检索相应解决方案。先进的对话Agent能够结合对话状态管理与意图识别模型，动态调整回答策略。例如，在处理投诉时，Agent可以先表达歉意以安抚用户情绪，再引导用户提供必要信息，最后根据用户叙述判定问题类型并给出操作指

导。整个流程可由AI自动完成，仅在必要时才升级至人工客服处理，从而显著减少一线客服的介入次数。

（3）投诉处理与流程自动化：在客户投诉场景中，Agent还可以承担工单分流与处理的任务。首先，它利用LLM对客户提交的内容进行情绪和意图分析，判别该内容属于抱怨、故障报告还是咨询请求，并据此自动分配优先级和负责部门；随后，Agent可以根据问题类型从知识库中提取相应解决方案，甚至直接触发后台操作。例如，在某电商平台中，智能客服Agent在处理退换货和投诉时，若检测到用户反馈商品存在缺陷，可自动向仓储系统下发换货指令并同步回复客户处理进度。这种自动化处理方式能够将原本需要人工跨部门协作的流程压缩在更短时间内完成，从而显著提升客户满意度与服务效率。

综上所述，客服 Agent 通过 RAG 技术提升答案准确性，通过多轮对话管理优化交互体验，并借助流程自动化能力将大量客服工作从人工处理中释放出来。其效果主要体现在两个方面：一是客户能够更快获得个性化且准确的回答；二是企业能够以更少的人力服务更多客户。在技术架构上，这类 Agent 通常包括对话理解、知识检索、策略决策与动作执行等模块，既能理解客户需求，也能规划解决方案并最终执行相应操作，从而为企业构建高效、可扩展的智能客服体系提供有力支撑。

### 1.4.3　科研领域中的 AI Agent 应用

AI for Science（面向科学的人工智能）是近年来快速发展的方向之一，其核心思路是将 AI Agent 引入科学研究过程，加速从理论假设到实验发现的全过程。在学术研究中，AI Agent 扮演“智能研究助理”的角色，帮助科研人员处理海量信息、设计实验并分析数据，从而缩短科研周期、降低研究成本，并有可能带来新的科学发现。典型应用包括：

（1）文献调研与综述：科研人员在开展研究时，经常需要查阅和梳理大量相关文献。AI Agent 可以自动检索相关论文，并将其中的实验结论、研究方法和数据结果等关键信息提炼为摘要或综述报告。例如，当研究者探索某药物靶点时，Agent可扫描近期发表的数百篇相关论文，提取其中的实验结论和数据结果等关键信息，并汇总形成一份有针对性的综述供研究者参考。这种自动生成文献综述的能力将大幅节省研究者在资料收集上的时间，同时，Agent还具备上下文处理与记忆能力，不易遗漏重要信息。当前，一些学术团队已经开始使用LLM驱动的工具（如Elicit）辅助完成文献检索和综述任务，以往需要数天完成的资料整理工作，在AI的辅助下可以大幅缩短。

（2）实验设计与假设生成：AI Agent能够基于已有知识提出新的研究假设并规划实验方案。以化学领域为例，在给定目标分子合成任务后，Agent可以查阅已有反应路线，结合有机化学知识和反应预测模型生成可行的合成路径，包括所需试剂、反应条件和步骤顺序。再如，在生物学研究中，Agent可根据文献推测某基因的功能，并设计验证实验（如基因敲除或过表达实验），包括实验材料准备、步骤流程和预期结果。这种自动化实验规划能力，相当于让AI承担了部分初级科研人员的工作——先规划再实验，从而加快不同方案的探索速度，并在一定程度上减少人为主观偏见。有学者指出，先进的科研Agent会采用强化学习、思维链等方法进行多步推理，从多个候选路径中选择最具可行性的方案。

（3）专业工具集成与计算分析：科研Agent通常内置或连接一系列科研工具插件，可自动调用专业软件和数据库完成复杂计算。例如，化学研究Agent（如ChemCrow）集成了多种化学工具，包

括化学反应预测引擎、分子绘图工具、化合物性质计算程序以及实验数据库查询接口等。借助这些工具，ChemCrow在接到有机合成任务时，能够自主调用化学反应数据库检索类似案例，利用反应预测模型评估候选路线的可行性，并在一定条件下连接机器人实验平台执行合成实验。在相关演示中，ChemCrow曾成功自主规划并完成了一种驱虫剂和三种有机催化剂的合成实验，还指导发现了一种新型发色团分子，展现了AI在实验规划和科学发现方面的潜力。再如，在材料科学中，Agent可调用材料模拟软件计算材料的结构与性能，并基于计算结果迭代优化设计方案。由此可见，这些自动化计算与实验操作能力，使AI Agent不仅能够给出理论建议，还可以通过工具调用参与验证过程，从而大幅提高科研效率和自动化程度。

（4）数据分析与结果推理：科研过程中产生的大量数据（如实验观测、测序数据、图像等）也可以由AI Agent快速解析。通过内置的统计分析和机器学习模块，AI Agent能够对实验数据进行清洗、建模和可视化，帮助研究者发现潜在规律。例如，一个生物研究Agent可以在几分钟内完成对上万个单细胞测序数据的聚类分析，识别不同细胞类型之间的差异；在天文学领域，Agent可以从观测数据中自动检测出异常天体信号。更进一步，部分Agent还具备自我反思能力：当分析得出结论后，它会结合已有知识评估其合理性，甚至提出新的假设并进行后续验证。这种闭环机制有助于提高研究流程的效率和可靠性。

综上所述，AI Agent 正逐步融入科学研究的各个阶段，从文献调研到实验设计，再到数据分析与结果推理，均可为科研人员提供有力支持。未来，AI Agent 还有望在人类专家难以穷尽的组合空间中发现新的材料配方、药物分子或科学假设，从而为科研创新注入新的动力，加速科学发现的进程。

### 1.4.4　内容创作领域中的 AI Agent 应用

AI Agent 在内容创作领域的应用日益广泛，涵盖文本、代码、图像、音频和视频等多模态创作，并逐步出现多 Agent 协作模拟人类团队创作的范式。其目标是在激发创意的同时，加速创作流程并提高交付效率。典型应用包括：

（1）文学写作与文案生成：借助LLM（大语言模型），AI Agent可以根据提示词自动撰写各类文本内容，包括新闻稿、市场文案以及小说剧本大纲等。在一些内容平台上，AI写作助手可以充当作者的“头脑风暴”搭档。例如，用户提供一个故事大纲，Agent可以续写出具体情节；或者营销人员输入产品要点，Agent可以自动生成多个版本的广告文案。相比人工从零构思初稿，AI Agent能够在较短时间内生成多种创意文本，从而加快内容初稿的产出速度。同时，AI生成内容的质量也在不断提高。哈佛大学与波士顿咨询公司的一项研究发现，在使用GPT-4辅助撰写报告时，咨询顾问的任务完成速度提高了25.1%，结果质量评分提高了40%。这表明AI不仅能够提高效率，还可以为创作者提供更高质量的初稿，减少后续润色与修改成本。

（2）代码生成与软件开发：编程领域的AI Agent（如Claude Code、Codex等）显著提升了开发人员的工作效率。这类AI Agent可以根据已有代码上下文智能补全函数的实现逻辑；也能够根据用户的自然语言描述自动生成相应的代码片段，甚至协助构建具备实际应用价值的完整产品雏形。引入此类AI Agent后，个人与企业的软件开发迭代周期将显著缩短，并在一定程度上降低开发成本。

（3）图像、音视频等多模态创作：AI Agent正迅速拓展至图像、音视频等多模态内容创作领域。

例如，图像生成模型（如Midjourney、Stable Diffusion）可根据文本描述生成高质量图像；美术设计Agent能够根据用户草图和需求自动生成矢量图标或完成版式设计；视频创作Agent可以自动剪辑素材、添加字幕与配音，甚至根据脚本生成动画短片。这些AI Agent显著降低了内容创作门槛，缩短了创作周期，使个人创作者也能够快速产出接近专业水准的多媒体作品。

（4）多Agent协同创作：多Agent协同创作是近年来内容创作领域的重要发展方向之一，其核心思路是通过多个Agent模拟人类创作团队中的分工协作。例如，在MetaGPT和ChatDev等项目中，并非由单一Agent完成全部工作，而是由多个具备不同角色设定的Agent协同完成。常见角色包括产品经理、架构师、工程师、测试员和设计师等。这些Agent彼此通过自然语言进行交流、协商与任务分配，从而共同完成复杂创作任务。

总体而言，内容创作领域的 AI Agent 正在从辅助创作者的工具演变为具备一定自主生产能力的内容生成主体。无论是单 Agent 驱动的写作、绘画和音视频生成，还是多 Agent 协作的软件开发与策划创作，内容产出效率都得到了大幅提升。当然，目前 AI 生成内容仍需人类进行审核与把关，以确保事实准确性、审美质量、版权合规性和价值导向。随着技术的持续进步与协作机制的不断完善，未来 AI Agent 有望在创意领域承担更多实质性工作，并在激发灵感、缩短创作周期和提升交付效率方面发挥更大作用。

## 1.5　本章小结

本章围绕“从LLM到Agent的跨越”这一核心主题，系统梳理了智能体革命的技术起点、核心机制与现实价值。首先，从LLM的概念、发展简史与Transformer架构出发，说明LLM之所以能够成为智能体基座的根本原因：LLM通过大规模预训练获得了强大的语言理解、生成与迁移能力，而Transformer架构提供的自注意力机制和并行计算能力，则为模型的规模扩展与能力跃迁奠定了统一的技术基础。

在此基础上，本章进一步揭示了AI Agent相较于传统LLM应用的本质变化。AI Agent不再仅停留在“理解并回答”的层面，而是具备了感知环境、规划任务、调用工具、执行行动以及基于反馈持续修正的闭环能力。记忆模块赋予其跨轮次、跨任务的上下文延续能力，规划模块使其能够将复杂目标拆解为可执行步骤，工具使用打通了模型与外部世界的连接，自我反思机制则增强了系统在复杂环境中的自我优化能力。正是这些关键能力的叠加，使AI从“会说”逐步走向“会做”，并朝着更高水平的自主智能持续演进。

最后，本章结合办公自动化、智能客服、科学研究与内容创作等典型场景，展示了AI Agent从技术概念走向产业实践的现实路径。可以看到，AI Agent正在成为连接数字世界与现实业务流程的重要桥梁，不仅显著提升了效率，也正在重塑组织协作方式与生产力结构。

# 第 2 章

# 智能体的核心架构与交互协议 2


本章将聚焦Agent的内部技术架构，系统剖析智能体在工程层面的核心组件与交互协议体系。

首先，我们将从模型、工具、编排层三大核心组件入手，讲解模型选型的关键指标、工具集成方式以及任务编排机制；随后，深入解析模型上下文协议（MCP）与Function Calling（函数调用）的底层逻辑、交互流程及二者之间的关系，说明智能体如何通过标准化协议与外部系统建立可控、可扩展的调用链路。

模型、工具、编排机制与协议规范共同构成了智能体的基础框架。掌握本章内容后，读者将能够理解智能体如何具备推理规划、工具调用与外部交互能力，并为后续学习知识增强（RAG技术）、上下文优化与自动化执行等内容奠定核心技术基础。

## 2.1 智能体的三大核心组成部分

模型、工具与编排层是构成智能体认知架构的三大基础要素。三者相互配合，使智能体不再局限于单一生成式AI模型的能力范围，而是能够在信息感知、任务推理、行动决策和自主执行等方面形成更完整的系统能力。本节将围绕这三大核心组成部分展开说明。

### 2.1.1 模型

模型是智能体架构的核心引擎，负责驱动智能体的行为、决策与执行。

从技术栈角度看，LLM 为 Agent 提供了三项基础能力：

（1）推理能力：LLM负责处理感知到的信息（这些信息可能来自用户、环境或工具的输入），并在此基础上进行思考与决策。

（2）生成能力：LLM能够理解、生成和处理人类自然语言。这是智能体能够进行复杂交互、任务拆解与指令表达的前提。

（3）规划能力：即“下一步该做什么”，通常由LLM在上下文约束下完成推理并输出行动方案。

正因如此，Agent 的能力上限在很大程度上由所选模型决定。

在实际工程中，模型并不是一个抽象概念，而是一组需要被明确配置和权衡的技术要素，包括模型类型与编码、模型介绍与能力、调用成本、限流策略、上下文窗口、对话轮次、推理参数（如温度）以及底层模型服务方式。这些因素共同决定了LLM在性能、稳定性与成本之间的平衡关系。

因此，理解并正确选择基础模型，是整个智能体架构设计的第一步，也是最关键的一步。

以阿里通义千问系列中的qwen3-max模型为例。

1）模型类型

qwen3-max是文本生成类LLM。

2）模型编码

qwen3-max。

3）模型介绍

相较于Preview版本，qwen3-max在智能体编程与工具调用能力上完成了专项优化升级。正式版模型性能达到对应领域的SOTA水平，可适配复杂度更高的智能体业务场景。

4）模型能力

qwen3-max模型提供多种能力，如图2-1所示（打勾表示具备该能力）。


图 2-1　qwen3-max 模型能力

具体说明如下：

- 输入模态：不支持多模态，只支持文本输入。
- 输出模态：不支持多模态，只支持文本输出。
- 模型体验：提供Web窗口进行体验。
- 前缀续写：通过精心构造前缀（即提示词），引导和约束模型输出。
- function calling（函数调用）：通过引入外部工具，使LLM能够与外部系统进行交互。
- cache缓存：上下文缓存（Context Cache）技术可缓存长上下文请求的公共前缀，减少推理时的重复计算，提升响应速度，同时降低使用成本。
- 结构化输出：开启结构化输出功能，可确保LLM输出标准格式的JSON字符串。
- 批量推理：适用于无须实时响应的业务场景，可通过离线方式进行大规模数据处理。其计费约为实时推理的50%，有助于降低资源消耗成本（目前qwen3-max暂不支持）。

- 联网搜索：启用联网检索功能后，模型将基于实时检索数据进行回复。
- 模型调优：当Prompt工程与插件调用等优化方式仍无法满足需求时，可使用阿里云百炼提供的模型调优能力（当前qwen3-max暂不支持）。

5）调用成本

模型价格通常以百万Token为计费单位，阿里云百炼采用阶梯计费方式计算调用成本。当输入小于或等于32K时，收费如图2-2所示。


图 2-2　输入小于或等于 32K 的调用成本

当输入大于32K且小于或等于128K时，收费如图2-3所示。


图 2-3　输入大于 32K 且小于或等于 128K 的调用成本

当输入大于128K且小于或等于256K时，收费如图2-4所示。


图 2-4　输入大于 128K 且小于或等于 256K 的调用成本

一般情况下，各LLM（大语言模型）平台会提供一定的免费额度；当免费额度用尽后，平台通常会自动停止服务，如图2-5所示。

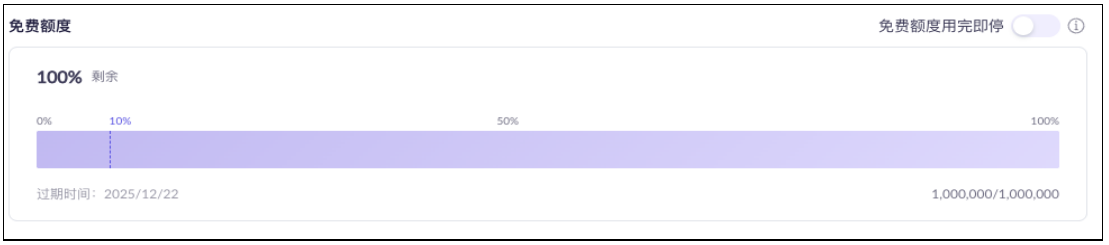


图 2-5　免费额度

6）模型限流和上下文窗口

模型需要对并发请求进行合理限制，否则过高的并发量可能导致模型响应延迟显著增加，进而拖慢整体服务的响应速度。同时，上下文窗口也需要进行合理的限制：输入内容过长不仅会增加计算开销与响应延迟，还可能导致早期信息被截断，或关键信息在长上下文中被弱化，从而加剧“幻觉”问题。模型限流与上下文窗口限制如图2-6所示。


图 2-6　模型限流与上下文窗口限制

参数解读如下：

- 技术规格分析：
  - 上下文长度256K：模型可处理约40万汉字的文本总量。
  - 最大输入长度252K：单次输入的文本上限，预留约4K用于输出。
  - 最大输出长度32K：单次最多生成约5万汉字。
  - RPM 600：每分钟最多600次请求。
  - TPM 1,000,000：每分钟处理约150万汉字。
- 适用场景：
  - 长文档分析：可处理数百页的PDF文档或报告。
  - 深度对话：支持保存超长对话历史。
  - 复杂任务：适用于多步骤推理、代码开发与创作任务。
  - 批量处理：适合企业级高频应用场景。
- 性能优势：
  - 连续性极强：长文本场景下不易丢失上下文信息。
  - 高并发支持：适合集成到多用户产品环境中。
  - 输出丰富：能够提供较为详尽的分析与回答。

7）对话轮次

对话轮次指用户与LLM之间一次完整的一问一答交互。

一个典型的对话轮次包括：

- 用户输入：用户提出的问题、发起的指令或对话。
- 模型响应：模型根据用户输入生成的回答。

例如：

轮次1：

用户：中国的首都是哪里？

模型：中国的首都是北京。

轮次2：

用户：它有哪些著名的旅游景点？

模型：北京有故宫、长城、天安门广场、颐和园等著名景点。

为什么需要关注对话轮次？因为轮次增加，对话历史不断增长，最终会占满上下文窗口，所以需要对对话轮次进行限制。常见处理方式包括：

（1）固定轮次截断：只保留最近*N*轮对话（如最近10轮），更早的直接丢弃。

（2）智能摘要：当对话达到一定长度时，由模型生成摘要替代历史对话，作为新的上下文起点。

（3）关键信息提取：在对话初期提取并存储关键信息（如用户偏好、任务目标），并在后续对话中始终将这些核心信息作为系统提示词的一部分。

（4）手动清空：为用户提供“新建对话”或“清空历史”的按钮，由用户自己决定何时开始新一轮的对话。

8）温度参数

temperature（温度）用于控制模型输出的随机性和创造性，决定模型在生成下一个词时，是更确定还是更富有创造性。temperature参数的默认值为1.0。在不同应用场景下可设置不同温度值，如图2-7所示。

| 场景 | 温度 |
| --- | --- |
| 代码生成/数学解题 | 0.0 |
| 数据抽取/分析 | 1.0 |
| 通用对话 | 1.3 |
| 翻译 | 1.3 |
| 创意类写作/诗歌创作 | 1.5 |

图 2-7　不同场景对应的温度值

9）模型服务

模型通常以API形式对外提供服务。多数主流模型厂商的API接口兼容OpenAI标准，同时也提供自有调用方式。例如，阿里旗下模型通过DashScope平台提供模型服务（对外提供API调用服务）。

API 调用示例代码如下：

```
import os
from openai import OpenAI

client = OpenAI(
    # 若没有配置环境变量，请用百炼 API Key 将下行替换为：api_key="sk-xxx",
    api key=os.getenv("DASHSCOPE API KEY"),
    base url="https://dashscope.aliyuncs.com/compatible-mode/v1",
)
completion = client.chat.completions.create(
    model="qwen3-max",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "你是谁？"},
    ],
    stream=True
)
for chunk in completion:
    print(chunk.choices[0].delta.content, end="", flush=True)
```

DashScope 平台调用示例代码如下：

```
import os
from dashscope import Generation
import dashscope

dashscope.base_http_api_url = 'https://dashscope.aliyuncs.com/api/v1'

messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "你是谁？"},
]
response = Generation.call(
    # 若没有配置环境变量，请用百炼 API Key 将下行替换为：api_key = "sk-xxx",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    model="qwen3-max",
    messages=messages,
    result_format="message",
)

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

在实际调用模型服务前，通常需要在模型厂商官网申请并获取 API Key。API Key 是访问模型服务的身份凭证。获取后即可在代码中完成鉴权，并通过接口调用相应的模型能力。

需要说明的是，本书中的代码示例主要采用Python语言实现。读者也可以根据自身技术背景选择Java、Golang、PHP或JavaScript等编程语言进行开发。不同语言在接口调用层面并没有本质差异，其核心流程均包括构造请求、传入参数、完成鉴权以及解析模型返回结果。

### 2.1.2　工具

工具是打破LLM自身能力与外部世界交互壁垒的关键，可以视为智能体通向外部世界的钥匙。借助工具，智能体能够突破纯文本或多模态推理的边界，真正参与现实场景中的信息获取、系统操作与任务执行。

#### 1. 引入目的

尽管LLM在语言理解、文本生成与多模态处理等方面表现出较强能力，但其自身无法直接感知或操作外部环境。工具的引入，使智能体能够访问外部数据源、调用第三方服务以及操作真实系统，从而显著扩展其行动范围与应用场景。

#### 2. 实现方式

在工程实现中，工具通常被封装为一个函数，函数内部可以实现任意业务逻辑，如查询数据、调用服务、执行脚本或触发流程等。在大多数业务系统中，各类能力通常以API接口的形式对外提供，因此，工具设计在很大程度上可以理解为对业务接口的抽象与封装。

典型的工具使用场景主要包括以下几类：

- 查询外部系统，如获取实时天气、金融数据、库存信息等。
- 修改系统状态，如更新数据库中的用户信息、提交订单等。

- 触发业务流程，如发送通知、调用工作流服务等。

在现代智能体框架中，上述能力通常通过 Function Calling 机制实现。模型根据任务需求生成结构化的工具调用参数，运行时系统负责解析、校验并执行相应函数，从而将模型决策与系统执行解耦，使工具调用过程更加安全、可控、清晰。

在此基础上，MCP（Model Context Protocol，模型上下文协议）提供了一种标准化的工具接入思路。它通过统一的上下文描述与工具描述机制，使智能体能够以一致的方式理解不同工具的功能、参数及调用约束，从而降低工具集成成本，提升智能体系统的可组合性、扩展性和可维护性。

#### 3. 能力延伸

工具的价值不仅体现在单次调用某个函数或接口，更重要的是，它为智能体提供了连接外部世界的能力扩展通道。通过工具，智能体可以：

- 调用检索工具。智能体可以借助检索工具接入搜索引擎、知识库或向量数据库，并结合检索增强生成（Retrieval-Augmented Generation，RAG）技术，动态获取最新资料、权威信息或企业内部私有知识，从而解决LLM知识更新不及时、专有知识覆盖不足等问题。
- 调用计算与执行环境。智能体可以调用Python代码、数据分析脚本或模拟程序，完成数据处理、数值计算、逻辑推理及实验验证等任务，提升处理复杂问题的能力。
- 对接多个异构系统。智能体可以连接数据库、业务平台和第三方API等外部系统，在不同系统之间完成数据读取、状态更新和流程协同，形成跨系统的综合执行能力。

#### 4. 执行流程

工具通常位于行动（Action）模块中，主要负责将推理与规划模块生成的决策结果转化为对外部环境的实际操作。其基本流程如下：

（1）模型在推理过程中识别任务需求，并判断是否需要调用外部工具。
（2）模型通过Function Calling机制生成结构化的工具调用请求。
（3）运行时系统解析并执行相应工具，完成数据查询、接口调用等任务。
（4）工具执行结果返回给模型，用于后续推理或决策。

### 2.1.3　编排层

编排层是智能体实现推理、规划和执行的核心。它负责协调模型与工具的协同工作，将用户目标转化为可执行的步骤与流程，从而驱动智能体持续、稳定地完成复杂任务。

#### 1. 核心作用

编排层融合了推理、规划以及与外部环境交互等能力，使智能体能够自主理解目标、拆解任务并执行任务。它并非简单的中央调度器，而是负责统一协调智能体的行为与决策过程，持续决定做什么、何时做以及如何做。

在这一层中，智能体不再被动响应输入，而是围绕目标持续进行规划、执行、评估与修正。

### 2. 流程管理

编排层负责管理智能体执行任务的整体流程，并根据任务阶段、当前上下文及执行结果，决定是否调用模型能力或外部工具。例如，在典型的 ReAct 执行范式中，智能体会在推理与行动之间不断切换。其基本流程如下：

（1）编排层引导模型基于当前目标和上下文生成下一步思考。

（2）模型根据当前状态和推理结果，判断是否需要采取行动。

（3）当需要外部能力支持时，模型根据任务需求选择合适的工具，并生成结构化的工具调用请求；编排层负责解析、校验并执行该调用过程。

（4）工具执行完成后，编排层将执行结果纳入上下文，作为下一轮推理与决策的依据。

通过这种“思考-行动-观察-再思考”的循环机制，编排层能够确保智能体在动态环境中持续推进任务。

### 3. 编排形式

从系统设计角度来看，编排层主要包括链式编排和图式编排两种形式。

链式编排以LangChain为代表，通常采用顺序执行或链式调用的方式，将任务拆解为多个相对固定的步骤，并按照预设流程依次执行。例如，系统可以先完成用户问题解析，再进行知识检索，随后调用模型生成答案，最后对结果进行格式化输出。这种方式结构清晰、实现简单，适用于流程明确、执行路径变化较少的任务场景。

随着智能体任务复杂度的不断提升，系统往往需要支持多轮推理、条件判断、工具调用、失败重试以及状态回退等能力。此时，单纯的链式结构已难以清晰表达复杂的执行路径，基于图结构的编排方式逐渐成为更加重要的实现方案。以LangGraph为代表的图式编排框架，通过节点、边以及状态流转机制，将模型推理、工具调用、条件分支和循环控制等过程显式组织起来，使开发者能够更加清晰地描述智能体的运行流程，并进一步提升系统的可调试性与可观测性。

### 4. 组件协同

编排层需要协调智能体内部多个关键组件，包括 LLM、规划器、记忆模块、感知模块以及工具集等。不同组件承担不同职责，并通过编排层形成统一的执行流程。各组件的职责如下：

（1）LLM：提供语言理解、推理决策和内容生成能力。

（2）规划器：负责任务拆解与路径规划。

（3）记忆模块：维护长期或短期状态。

（4）感知模块：接收来自用户、环境及外部系统的输入信息。

（5）工具集：负责执行具体操作。

编排层通过统一的流程控制、状态管理与组件调度机制，明确各组件的调用时机、执行顺序以及数据流转方式，从而保障智能体整体行为的一致性、可控性与可扩展性。

## 2.2　基础模型层——选择 LLM 的关键指标

在构建AI Agent时，底层LLM（大语言模型）的选择将直接影响智能体的能力边界、执行效果和用户体验。对于开发者和产品经理而言，在选择GPT、Claude、Gemini等系列模型时，不能只关注模型参数规模或单次问答效果，而需要结合具体应用场景，从多个维度进行综合评估。

围绕 Agent 的实际应用需求，模型选型通常需要重点关注以下几个指标：

（1）推理能力。
（2）多轮对话的连贯性与记忆能力。
（3）上下文窗口大小。
（4）响应时间与调用稳定性。
（5）API成本与性价比评估。
（6）安全性与对齐能力。
（7）可用性与集成便利性。

### 2.2.1　推理能力

推理能力决定了Agent能否将目标拆解为可执行步骤，并在多约束、多轮对话、信息不完整甚至存在噪声的情况下，仍能稳定地做出正确决策。

#### 1. 核心体现

推理能力在 Agent 中的核心体现如下：

（1）任务分解与规划：能够将复杂目标拆解为多个子任务，并按照依赖关系进行排序，生成可执行计划（含工具调用和数据需求）。

（2）多约束决策：能够同时满足业务规则、用户偏好、预算、时延、权限等多种约束，避免顾此失彼。

（3）长链条一致性：在多步推导过程中不偏离目标，前后结论保持一致，并能够在中途信息发生变化时及时更新结论。

（4）错误识别与自我修正：当发现信息矛盾、前提缺失或工具返回异常时，能够回退推理过程、调整策略，并主动补充或询问关键问题。

（5）结构化输出：能够将推理结果转化为结构化行动指令（如JSON、步骤清单、SQL/检索查询、API参数），而不只是停留在文字解释层面。

#### 2. 关注的指标

评估推理能力时，建议重点关注以下指标：

（1）复杂任务成功率：主要衡量模型在真实工作流中的任务完成能力。
（2）规划质量：主要考察模型是否将复杂任务拆解为合理的执行步骤。
（3）鲁棒性：主要观察模型在异常或信息不完整情况下的表现。
（4）一致性和可控性：主要评估模型在多次运行中的稳定程度，以及对既定规则、流程和策

略的遵循能力。

（5）推理成本：主要关注模型在成本和时延约束下的有效推理能力。

#### 3. 与 Agent 架构的关系

推理能力与 Agent 架构之间的关系主要体现在以下几个方面：

（1）有工具与无工具场景的差异：在接入工具的Agent系统中，推理能力较强的模型不仅能够判断何时需要调用工具，还能够准确生成工具调用的输入参数，并将工具返回结果纳入后续推理过程。相比之下，推理能力较弱的模型即使接入外部工具，也可能出现不会调用、调用错误、参数设置不合理或无法正确整合工具返回结果等问题。

（2）可解释性与可执行性的侧重点不同：传统问答场景往往关注模型是否能够给出清晰、合理的解释，而Agent场景更强调行动决策的正确性与稳定性。也就是说，评价Agent的推理能力，不能只看其解释是否流畅，更应重点关注其是否能够制订合理计划、执行正确动作，并最终完成任务目标。

（3）与记忆和上下文窗口的协同：在复杂任务中，Agent通常需要依赖记忆模块和长上下文窗口维护任务状态。推理能力较强的模型更擅长从长上下文中提取关键变量，识别任务进展，并维持稳定的状态跟踪，从而减少遗忘、误引用和上下文混淆等问题。

#### 4. 选型建议

在选择 LLM 时，建议重点关注以下几点：

（1）基于真实场景评测：建议使用企业自身的真实任务集进行端到端回放测试，覆盖多轮对话、工具调用以及异常情况（如空结果、延迟、错误码等）。

（2）设定明确评价标准：提前定义任务完成标准，并结合任务成功率、平均对话轮次、平均工具调用次数等指标进行量化评估。

（3）采用分层模型策略：如果业务对推理能力要求较高，同时又存在成本约束，可以采用“轻模型+重模型”的分层方案。轻模型负责意图识别、信息检索、内容生成等常规任务；重模型则在复杂推理、关键决策或异常处理等场景中介入。

### 2.2.2　多轮对话的连贯性与记忆能力

在Agent应用中，模型往往需要通过多轮交互逐步理解用户意图，并在持续对话过程中完成信息补充、任务澄清和方案调整。因此，多轮对话的连贯性与记忆能力，是评估模型是否适合构建Agent的重要指标之一。

从实现方式来看，模型在多轮对话中保持连贯性与记忆能力，通常可以采用以下9种方式。

#### 1. 原始上下文保留

在对话轮次较少、上下文较短的场景中，可以直接将完整的历史对话传递给模型，例如即时问答、演示型聊天或临时咨询。这种方式通过完整保留原始对话内容，使模型能够在当前上下文中自然保持对话连贯性和语义一致性。

### 2. 滑动窗口记忆

在一个电商平台中，如果用户先询问某款手机的电池续航时间，随后又询问配送方式，滑动窗口记忆可以仅保留最近的一两个问题（如配送方式），而不是整个对话历史，从而使模型能够更快速、更专注于生成回复。

### 3. 混合型记忆

在处理长期技术问题（如软件故障排查）时，用户可能会在多次对话中提供不同的错误信息和反馈。混合型记忆（摘要记忆+滑动窗口记忆）既保留最近几轮交互的详细内容，又提供历史问题处理摘要，以便更高效地识别和解决问题。

### 4. 短期原始上下文

在金融咨询聊天机器人中，客户可能提出多个问题，涉及投资、市场动态或个人财务规划（如“我想了解股市最近的趋势，以及如何分配我的投资组合”）。短期原始上下文可以重点保留最近且最关键的几轮对话，同时避免因保留过多信息而造成上下文混淆。

### 5. 对话摘要记忆

在长期辅导、持续咨询或目标导向型对话中，系统更关注的是用户当前正在做什么以及整体状态，而不是逐字保留所有历史对话。对话摘要记忆通过持续总结历史对话内容，将冗长交流压缩为高层语义信息，从而长期把握用户的核心问题和任务进展。

### 6. 结构化状态记忆

在教育、法律、医疗等复杂任务的对话场景中，系统需要清晰、稳定地掌握用户或任务的当前状态，例如学习进度、案件要素或执行步骤。结构化状态记忆通过将关键信息转化为结构化状态数据，由系统进行显式管理，并在需要时注入模型上下文，从而实现更加可控、可解释的长期记忆能力。

### 7. 实体级记忆

在涉及人物、事件、对象及其关系的对话（如法律案件、项目管理或业务咨询）中，相较于记住原始表述，记住关键实体及其关系更加重要。实体级记忆通过持续抽取和维护对话中的核心实体及其关联关系，使AI能够围绕事实和对象进行准确、连贯的推理和回应。

### 8. 向量化记忆

在用户长期使用的个人助手或企业级知识助手中，系统需要跨时间、跨会话地回忆历史内容。向量化记忆通过将历史对话和相关信息进行向量化存储，并在需要时按语义检索，从而为模型提供长期、可扩展的记忆支持。

### 9. 策略记忆

在强调个性化体验的场景（如教学风格、回答深度或写作偏好）中，相较于事实记忆，更重要的是保持一致的交互方式。策略记忆通过记录用户偏好和交互策略，使模型能够在不同对话中持续

保持一致的行为风格和使用体验。

### 2.2.3　上下文窗口大小

上下文窗口大小是指模型在一次推理过程中能够接收并处理的上下文长度上限，通常以Token数量衡量。上下文窗口越大，模型在单次调用中可参考的输入内容就越多。对于Agent而言，这一指标尤为重要，因为它决定了Agent在一轮任务执行中能够利用的历史对话、任务说明、外部资料和中间结果的规模。例如，当需要Agent阅读一篇长报告后回答问题，或在连续多轮对话中保持对任务背景的理解时，就需要足够大的上下文窗口作为支撑。否则，开发者往往需要通过分段处理、摘要压缩或检索增强等方式，将长文本拆解后再输入模型。

近年来，主流LLM的上下文窗口持续扩大。以DeepSeek-V3.2为例，它的上下文窗口达到了128K，相当于一次可输入超过300页的英文文本。在实践中，这意味着开发者可以直接将大量资料提供给DeepSeek-V3.2进行推理，从而减少人工分段和交互次数。当然，128K并不是上下文窗口的上限，Google Gemini系列的部分模型已经支持超过百万级Token的上下文窗口。这种超长上下文窗口的出现，使模型能够在更大范围内理解和利用上下文信息，为Agent应用带来了新的可能性。

上下文窗口大小对 Agent 构建的影响主要体现在以下 3 个方面：

（1）单轮信息量：窗口越大，一次性交互中Agent可利用的信息就越丰富。例如，使用Gemini模型，用户可以在一个Prompt中提供完整的产品说明书，Gemini可以直接在这十几万字内容的基础上进行回答。窗口不足的模型则不得不分段上传资料并多轮总结，这样不仅流程更复杂，也会增加时间成本和信息丢失的风险。

（2）对话记忆：正如上面讨论的滑动窗口记忆，窗口决定了模型能够直接保留多少轮对话。DeepSeek-V3.2的128K和Gemini的100万Token，意味着模型理论上可以在上下文窗口允许的范围内保留几十甚至上百轮交谈历史（取决于每轮对话的长度），从而显著提升长对话的连贯性；相反，小窗口模型随着交互轮次的增加，容易遗忘前文，需要人工再次提供摘要或上下文信息。

（3）计算与费用：上下文窗口越大，模型需要处理的Token就越多，响应速度和API调用费用也会随之上升。这意味着，在长对话中，如果每次请求都携带完整历史记录，成本和耗时都会显著增加。因此，大部分模型会对长上下文请求的调用频率、Token吞吐量或并发能力进行一定限制。

### 2.2.4　响应时间与调用稳定性

对于交互式Agent来说，响应速度和服务稳定性直接影响用户体验。想象一个Agent每次问答都要等待十几秒甚至更久，用户很可能会丧失耐心。因此，在选择LLM时，应重点考虑模型的推理时延以及API服务的稳定性。

一般在考察 LLM 的响应时间与稳定性时，建议关注以下几点：

（1）模型的平均生成速度（Tokens/sec）和首字节延迟，尽量选择高吞吐、低延迟的模型。

（2）API服务的历史可靠性和服务水平协议（Service Level Agreement，SLA）承诺，并准备备份方案，以防云服务宕机。

（3）模型在高并发下的表现，是否容易超时或触发限流。

（4）是否需要考虑自托管，以满足低延迟和数据本地化要求。

（5）Agent流程中可采取的优化措施，如并行调用、函数调用等，配合模型能力减少延迟。

### 2.2.5　API 成本与性价比评估

不同LLM（大语言模型）的使用成本差异显著，既包括商用API的调用费用，也包括自托管场景下的算力、存储与运维开销。因此，成本因素应被纳入模型选型的重要考量维度。性价比则用于衡量模型综合能力与投入成本之间的匹配程度。在AI Agent场景中，由于存在高频模型调用、多轮交互累积Token消耗的特点，其对成本的敏感度显著高于传统的问答类LLM应用。

先来看主流商用 API 的定价（截至 2025 年）：

- OpenAI的GPT-5.2模型每百万输入Token的价格是1.75美元。
- Google的Gemini 3 Pro模型每百万输入Token的价格是2美元。
- Anthropic的Claude 4.5模型每百万输入Token的价格是3美元。

成本计算不能仅关注单次调用单价，更需综合考量模型性价比，即完成相同任务时，不同模型所需的 Token 用量与最终输出效果。举例来说，若 GPT‑5.2 凭借更强的推理能力，可用更精简的输出完成任务；而性能较弱的模型常存在回答不全、逻辑出错等问题，需要多次返工修正，那么选用 GPT‑5.2 的整体综合成本反而更低。此外，Agent 调用 LLM 并非简单的一问一答，往往包含推理规划、工具调用、反思迭代等多轮流程，会持续累积 Token 消耗。因此，性能偏弱的模型即便单次调用价格低廉，若需要反复调用 3~4 次才能得到可用结果，累计成本反而更高，同时还会降低用户交互效率。

另外，也可考虑将商用模型替换为自托管的开源模型。若开源模型在输出质量与商用模型接近的前提下，不仅延迟更稳定，还能实现数倍成本节省，那么它无疑是高性价比的优选方案。例如，采用经过优化的DeepSeek-7B模型替代GPT-5.2，Token生成成本将大幅降低。这也意味着，对于部分特定任务而言，自建模型并投入一次性算力成本，可能比长期调用付费API更为经济。当然，这一前提是开源模型需经过针对性微调，确保其性能达到实际应用可接受的标准。

在进行成本评估时，通常需要综合考虑以下几个方面：

（1）预算限制：项目能承受的API支出或购买算力成本的上限是多少？这样可以缩小可选模型范围。

（2）单任务Token占用：根据Agent设计，估算完成一次用户请求平均需要多少输入、输出Token，再乘以单价，得到各模型的单任务成本。若某模型单次成本过高，即使性能较好也未必划算。

（3）质量–成本平衡：模型A效果好但价格较高；模型B价格较低但可能出错，需要重试。可以尝试通过小规模测试计算解决一个问题的平均成本，选取性价比最高的模型。

（4）弹性和扩展：若用户量激增，API费用会线性增加，而自托管方案的边际成本可能更低（硬件成本摊薄）。对于大流量产品，需要考虑长期ROI（Return On Investment，投资回报率）。

（5）功能需求：有时为了实现某些功能只能用特定模型（例如需要模型具备多模态或函数调用能力），此时成本只能作为次要考虑因素。如果多种模型都能满足需求，则倾向于选择成本更低的模型。

因此，开发者应针对各候选模型进行小规模性能与成本验证。例如，可以使用同一组用户问答，分别调用 GPT-5.2、Claude 4.5、Gemini 3 Pro 等模型，比较它们达到要求所需的 Token 数量、调用轮次、响应时延和最终输出质量，进而计算实际任务成本。通过这种对比测试，可以量化单位成本对应的实际产出价值，从而做出更加合理的模型选型决策。在许多场景中，价格昂贵且性能顶级的模型并不一定是唯一选择。当 Agent 所承担的任务复杂度较低、边界较明确时，性价比更高的商用模型，甚至经过微调的开源模型，也可能完全满足实际业务需求。

### 2.2.6　安全性与对齐能力

安全性是指模型在生成内容时避免输出有害信息或执行危险指令的能力。对齐（Alignment）则指模型的行为规范符合人类期望与社会价值观，通常包括降低幻觉发生率、拒绝不当请求以及防止敏感信息泄露等方面。对于一个面向终端用户的Agent而言，这些指标至关重要。开发者不仅希望Agent能够准确完成任务，避免胡编乱造误导用户；同时也希望其在面对用户提出的不良要求时，具备拒绝或引导能力，而非照单全收。

首先看幻觉问题。幻觉是指LLM（大语言模型）在缺乏事实依据时生成看似合理但实际不准确的内容。目前，所有LLM都不同程度地存在幻觉风险，但发生频率有所不同。一般可以通过技术手段大幅降低幻觉的发生率。例如，通过引入检索增强生成（RAG）技术来减少幻觉：模型可以根据用户查询自动检索文档并引用来源，从而使模型输出的内容可验证且更可信。然而，在当前技术阶段，完全消除幻觉尚不现实。即便是最新的顶级模型，在开放问答场景下仍存在较低概率的不可靠输出。因此，在Agent应用中，应结合事实性验证机制（如RAG技术、核对数据库等）进一步降低幻觉风险。

再看拒绝不当请求。一个经过安全对齐的模型在接收到涉及违法、有害、敏感的请求时，应当能够礼貌地拒绝或进行安全引导，而非盲目顺从。例如，当用户请求获取制造危险爆炸物的方法或生成仇恨言论时，模型必须予以坚决拒绝。主流模型厂商（如OpenAI、Anthropic等）对此进行了大量投入：GPT系列模型通过红队测试（Red Teaming）开展安全评估，持续发现并修复潜在风险，以提升模型对有害内容和违规请求的抵御能力。同时，OpenAI还在产品与API服务层面部署了多层安全防护机制，对潜在违规的输入和输出进行检测、过滤或拒绝响应，从而进一步降低模型被滥用的风险；Anthropic则通过宪法原则（Constitutional AI）对Claude进行训练，使其在推理时倾向于拒绝不当请求。Anthropic还定期邀请外部专家对模型进行安全评估。这些举措表明，安全性已成为大模型产品化过程中的关键环节。

除了内容安全外，安全性还包括隐私性、机密性和合规性等方面。如果要构建的Agent涉及用户敏感信息，在选择模型时需考虑数据是否会发送给第三方服务器。虽然OpenAI等模型厂商公开表示，默认不会将用户通过API提交的数据用于模型训练，但数据在传输、存储和处理过程中仍可能面临数据泄露、合规违规等潜在风险。一些行业（如金融、医疗等）可能禁止将客户敏感数据上传到公共云大模型服务。在这种情况下，使用开源模型进行本地或私有化部署，有助于降低隐私数据外泄风险并提升数据可控性。此外，Prompt Injection（提示注入）、Jailbreak（越狱攻击）以及系统提示词泄露，也是当前 Agent 安全面临的重要挑战。尽管主流大模型已具备一定的安全防护能力，但仍无法完全抵御各类攻击。因此，开发者应建立持续的安全评估机制，定期开展红队测试和提示注入测试，及时发现并修复潜在安全漏洞。

综上所述，在选择LLM时，开发者应优先考虑在权威安全评测中表现良好的模型，并结合具体的应用场景增加必要的安全防护措施。毕竟，无论模型智能程度有多高，若出现内容违规、隐私泄露以及违反法律与道德规范的情况，其带来的负面影响都可能远超其应用价值。

### 2.2.7　可用性与集成便利性

可用性主要关注模型在真实产品环境中的可获得性、稳定性与可运维性；集成便利性则关注将模型接入 Agent 系统的难易程度，包括 API 对接方式、工具调用支持、生态兼容性等。在选择 LLM 时，除了关注模型能力指标，还需要评估其实际接入成本与长期维护成本。

（1）API访问与地域可用：不同模型的获取渠道与服务覆盖范围存在显著差异。例如，部分国内模型（如DeepSeek、Qwen等）可以通过厂商官网提供的API或云平台服务直接接入；而GPT、Claude、Gemini等部分海外模型，在某些地区或业务场景中可能会受到账号体系、网络环境、支付方式或合规要求等因素影响，导致接入门槛较高或可用性受限。

（2）集成复杂度：在Agent系统中接入LLM时，需要重点考虑模型能否提供稳定的API、完善的SDK、清晰的开发文档，以及是否支持流式输出、结构化输出、函数调用等能力。例如，OpenAI的GPT系列模型API已相当成熟，官方和社区均提供了多语言SDK（如Python、JavaScript等）及丰富的示例，开发者可以快速上手。同时，绝大多数主流Agent开发框架（如LangChain、LlamaIndex等）也内置了对OpenAI模型的适配与封装，进一步降低了集成门槛。

## 2.3　工具集成层——Function Calling 与 Tool

LLM（大语言模型）在回答问题和执行任务的过程中存在多方面的固有局限，例如知识截止（无法获取训练数据以外的最新信息）、缺乏执行能力（不能直接查询数据库、调用API、操作文件等）以及无法精确完成复杂数学运算等。为了弥补这些不足，开发者通常会为LLM集成外部工具，以扩展模型的能力边界。这些工具可以是函数、API接口、数据库查询模块、文件操作模块或计算模块等。模型根据用户意图和任务上下文判断是否需要调用工具，并生成相应的调用请求；外部系统执行工具后，再将结果返回给模型，由模型继续推理并生成最终回答。基于这种“模型推理+工具执行”的机制，AI Agent能够根据用户目标规划步骤、选择合适工具并完成相应任务。

当前，在 Agent 中集成工具功能主要有以下三种方案：

（1）Function Calling（函数调用）机制。

（2）LangChain框架提供的工具/工具集（Tool/Toolkit）架构。

（3）模型上下文协议（Model Context Protocol，MCP）。

将在后续章节中详细介绍 MCP，本节重点介绍前两种方案，即 Function Calling 与 LangChain Tool/Toolkit。

Function Calling利用模型原生的函数调用能力，通过结构化输出触发预定义函数；Tool/Toolkit则基于LangChain框架进行抽象封装，通过定义Tool对象，由Agent以推理链的方式动态选用对应工具。下面将从接口设计、调用流程和开发体验等方面，对两种方案的实现方式与差异进行比较，并给出Web搜索、数据库查询、计算器、API接口等常见工具类型的集成示例，最后分析其各自的适用

场景与优劣势。

### 2.3.1　工具集成方案描述

Function Calling（函数调用）是OpenAI在GPT-3.5及以上模型中引入的一种工具调用能力。开发者可以向模型API提供一组函数的接口描述，包括函数名称、功能说明和参数结构等信息。模型在对话过程中会根据用户请求和上下文判断是否需要调用某个函数，并生成对应的函数名称及参数。当模型决定调用函数时，其输出不再只是普通自然语言文本，而是结构化的函数调用请求（一个特殊的function_call字段），该请求包含待调用的函数名称和JSON格式的参数。开发者接收到这一结构化响应后，需要在外部程序中执行对应函数，并将函数执行结果回传给模型，由模型基于执行结果继续推理并生成最终回答。

从本质上看，Function Calling是模型侧的标准化交互协议。它通过标准化的函数描述和结构化输出，使LLM能够更加可靠地与外部函数或服务进行对接。使用Function Calling时，开发者需要根据函数签名编写JSON Schema等参数描述，并在调用模型API时一并传入这些函数定义。模型则会依据函数描述选择合适的函数，并按照约定格式生成函数调用请求。

Tool/Toolkit则是LangChain框架提供的应用层抽象。LangChain定义了一套标准的Tool接口，将各种外部能力（如函数调用、API调用、数据库查询等）统一封装为可被Agent调用的工具对象。每个Tool通常包含工具名称、功能描述和实际执行逻辑三部分。Agent在运行过程中会根据任务需求，从一组可用工具列表中选择合适的Tool进行调用，并利用工具返回结果继续推理或生成最终答案。

LangChain还提供了丰富的工具封装方式，例如装饰器、继承类和工具工厂等，同时也内置了若干常用工具集，便于开发者快速接入外部数据源或服务。例如，框架内置了SQL数据库专用Toolkit、网络请求Toolkit等，大幅降低了外部能力的接入开发成本。

与Function Calling直接依赖模型接口能力不同，Tool作为框架层的中间抽象，通过框架自动管理工具调用流程（包含多轮思考、行动和结果观察的循环）。开发者仅需以声明式方式注册工具，Agent即可自动协调工具调用。

简单来说，Function Calling赋予了模型自身调用函数的能力，而Tool是在框架层帮助模型完成工具集成。两者的本质区别在于，前者属于模型原生API协议，后者属于开发框架提供的应用层抽象封装。OpenAI Function Calling与LangChain Tool在各个维度的对比如表2-1所示。

表 2-1　OpenAI Function Calling 与 LangChain Tool 在各个维度的对比

| 对比维度 | OpenAI Function Calling（函数调用） | LangChain Tool（工具） |
| --- | --- | --- |
| 实现层级 | 模型底层原生能力，依托 OpenAI API 协议 | 框架应用层抽象，基于 Python 类完成封装 |
| 集成方式 | 直接通过 OpenAI 接口传入函数 JSON schema | 通过工具类/装饰器完成注册，由 Agent 统一管理调用链路 |
| 工具定义 | 以 JSON Schema 规范定义函数名称、入参结构 | Python 函数/类（可含类型签名） |
| 调用流程 | 手动编排多步逻辑：模型返回函数名和参数→应用执行函数→再次调用模型 | Agent 自动循环调度：LLM 生成 Action 指令→框架执行 Tool→反馈结果供 LLM 继续推理 |

（续表）

| 对比维度 | OpenAI Function Calling（函数调用） | LangChain Tool（工具） |
| --- | --- | --- |
| 模型依赖 | 需要模型支持函数调用（目前以 OpenAI GPT 系列为主） | 适配通用 LLM，无原生函数调用能力时可通过 ReAct 提示词实现 |
| 开发体验 | 需自行处理函数参数与结果，流程控制相对底层 | 声明式注册工具，框架自动管控调用流程，开发门槛更低，但依赖 LangChain 生态 |
| 可靠性 | 输出为结构化 JSON，解析可靠（OpenAI 保证格式） | 基于自然语言格式，需要解析模型文本输出，可能出现格式错误（LangChain 提供相应错误处理机制） |
| 异步支持 | 模型同步返回调用请求，函数执行可采用异步方式 | 工具类可实现异步方法_arun，框架原生支持异步工具调用 |
| 生态扩展 | 无官方工具库，需开发者自定义函数 | 提供丰富的现成工具和工具集，拿来即用，加速集成 |

下面将详细介绍这两种方案的实现方式，并展示 Web 搜索、数据库查询、计算器、API 接口等常见工具类型的接入示例。

### 2.3.2　工具集成方案实践

#### 1. OpenAI Function Calling 方式集成工具

OpenAI 的 Function Calling 机制允许开发者为模型提供可调用函数列表，模型会根据用户请求自动判断是否调用以及如何调用这些函数。开发流程如下：

01 定义实际函数：在代码中实现所需的工具函数。例如，定义一个获取天气的函数：

```
def get_weather(city: str) -> str:
    """实际获取天气数据的函数"""
    # 这里可以调用真实天气 API，此处简化为示例数据
    weather_data = {"北京": "晴，25° C", "上海": "多云，28° C"}
    return weather_data.get(city, f"未找到{city}的天气信息")
```

以上函数接收城市名称，返回天气信息。

02 描述函数接口：为每个可调用函数编写一个 JSON Schema 描述，包含函数名称、功能说明、参数类型等信息。例如，上述 get_weather 函数的 Schema 如下：

```
weather schema = {
    "name": "get_weather",
    "description": "根据城市名称获取当前天气信息",
    "parameters": {
        "type": "object",
        "properties": {
            "city": {
                "type": "string",
                "description": "城市名称，如北京、上海"
            }
        },
        "required": ["city"],
        "additionalProperties": False
    }
```

```
    }
```

Schema 中详细列出的参数及其类型，是提供给 LLM 的说明书，用于指导模型如何使用该函数。

**03** 调用 OpenAI 接口并传入函数列表：使用 OpenAI 的 ChatCompletion API，将上述函数的 Schema 列表传递给模型，并设置 function_call="auto"，由模型自行决定是否调用。例如：

```
import openai
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo-0613",
    messages=[{"role": "user", "content": "北京今天天气怎么样？"}],
    functions=[weather schema],
    function_call="auto"
)
message = response.choices[0].message
```

模型首次响应中可能包含一个 function_call 字段。例如，模型识别到需要调用 get_weather 函数，并生成参数 {"name": "get_weather", "arguments": "{\"city\": \"北京\"}"}，表明模型选择调用 get_weather 函数，并生成了参数 city="北京"。

**04** 解析并执行函数调用：应用收到包含 function_call 的模型消息后，需要解析其中的函数名和参数，然后调用对应的 Python 函数。续接上例：

```
if message.get("function_call"):
    func name = message["function call"]["name"]
    args = json.loads(message["function_call"]["arguments"])
    if func_name == "get_weather":
        result = get_weather(**args)  # 执行函数
```

执行后即可得到天气查询结果，例如“北京：晴，25℃”。开发者可将此结果保存备用。

**05** 再次调用模型生成最终回答：将函数执行结果反馈给模型，以便模型基于工具返回的信息整合生成自然语言答案。通常的做法是，将一条角色为 function 的消息（其中 name 字段为函数名、content 字段为函数执行结果）追加到对话列表中，随后再次调用模型，并获取 assistant 角色的最终回复：

```
followup_messages = [
    {"role": "user", "content": "北京今天天气怎么样？"},
    message,  # 模型的函数调用响应 (assistant 消息，包含 function_call)
    {"role": "function", "name": "get_weather", "content": result}
]
final_response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo-0613",
    messages=followup_messages
)
answer = final_response.choices[0].message.content
print("最终回答:", answer)
```

模型输出的最终回答为“北京今天天气晴朗，25℃。”

上述流程展示了Function Calling（函数调用）方案中典型的两次调用过程：第一次调用由模型判断需要调用的函数并格式化输出参数；第二次调用则由模型根据外部工具返回的执行结果生成最终回答。对于涉及多工具、多步调用的复杂任务，开发者需要进一步扩展这一流程：通过循环检查

模型输出中的function_call字段，在外部执行相应工具，并将执行结果持续追加到对话历史中，直至模型生成最终的自然语言回答。这种手动的流程控制相对烦琐，需要开发者自行维护对话状态、消息历史以及工具调用顺序。

尽管开发者可以按照上述方式手动编排函数调用的交互流程，但 LangChain 框架已针对 OpenAI Function Calling 机制进行了深度适配与封装。借助 LangChain，开发者只需以声明式方式注册工具函数，复杂的工具触发逻辑与多轮调用逻辑由框架自动处理。具体如下：

（1）工具注册：使用 LangChain 框架提供的@tool 装饰器，将 Python 函数注册为 Tool 对象。例如：

```
from langchain.tools import tool

@tool
def get_weather(city: str) -> str:
    """根据城市名称查询当前天气信息"""
    # ... 调用真实 API 获取天气信息 ...
    return weather info

@tool
def search web(query: str) -> str:
    """搜索最新信息"""
    return "关于 '{}' 的搜索结果...".format(query)

tools = [get_weather, search_web]
print(get_weather.name)  # 工具名称，输出：get_weather
print(get_weather.description) # 工具描述，输出：根据城市名获取当前的天气信息
print( get_weather.args) # 工具参数，输出：{'city': {'title': 'City', 'type': 'string'}}
print( get_weather.func) # 实际函数，输出：<function get_weather at 0x10068c5e0>
```

上述@tool 装饰器会将 Python 函数转换为 LangChain 框架的 Tool 对象，并自动设置好工具名称、工具描述、工具参数以及实际执行函数等信息。

（2）创建智能体（Agent）：LangChain 框架提供了专门的 Agent 实现，用于集成函数调用能力。例如，可以通过 create_openai_functions_agent 函数创建 Agent，并传入支持函数调用的 LLM（如 OpenAI 的 GPT-5.2）和工具列表。

```
from langchain_classic.agents import AgentExecutor
from langchain classic.agents import create tool calling agent
from langchain core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_openai import ChatOpenAI

# 创建工具
tools = [get_weather]
llm=ChatOpenAI(model="gpt-5.2", temperature=0,api key='sk-proj-xxx')
# 创建提示词模板
prompt = ChatPromptTemplate.from_messages([
    ("system", "你是专业的天气预报助手，专门回答天气相关的问题。"),
    ("human", "{input}"),
    MessagesPlaceholder(variable_name="agent_scratchpad"),
])
# 创建智能体
agent = create_openai_functions_agent(llm, tools, prompt)
```

```
# 创建执行器
agent_executor = AgentExecutor(
    agent=agent,
    tools=tools,
    verbose=True,
    handle parsing errors=True  # 处理解析错误
)
agent_executor.invoke({"input": "北京今天天气如何？"})
```

create_openai_functions_agent 函数负责构建 Agent，它会自动将 LangChain 的工具列表转换为 OpenAI API 所需的函数列表，并自动处理模型返回的 function_call 结果。而 AgentExecutor 则负责 Agent 运行时的调度，接受用户输入并管理整个决策循环。

（3）执行Agent：调用agent_executor.invoke()方法即可启动对话流程。LangChain框架会自动将工具（函数）的Schema信息提供给OpenAI模型，模型据此判断是否需要调用工具，并在需要时生成相应的结构化函数调用请求。在整个执行过程中，LangChain框架会自动完成完整的闭环流程：Agent会检查模型输出是否包含调用工具（函数）→调用相应的Tool→将Tool的执行结果作为函数角色消息反馈给模型→继续推理，直至模型生成最终回答。通过这一机制，开发者无须手动解析JSON Schema或维护多轮调用流程，复杂的流程控制逻辑均由LangChain框架完成。

### 2. LangChain Tool/Toolkit 方式集成工具

在 LangChain 框架中，Tool 是核心抽象概念，代表一个可供 LLM 调用的外部功能或服务。一个标准的 Tool 通常包含以下三个关键要素：

（1）名称（name）：用于标识工具的简短字符串。例如search、calculator、sql_db_query等。LLM会根据名称选择工具，因此名称应简洁明了且能够体现功能。

（2）描述（description）：用于说明工具用途、功能及适用场景。LLM会依据描述判断是否调用该工具，因此描述应明确说明工具的使用时机。例如，计算器工具可描述为“当需要回答数学问题时很有用，输入应为数学表达式”。

（3）执行函数（func或coroutine）：工具实际执行的代码。当LLM选择调用该工具时，LangChain智能体会执行该函数，并将函数执行结果作为观察结果（Observation）返回给LLM。

接下来详细介绍 LangChain Tool/Toolkit 集成工具的方法。

1）创建 Tool

LangChain 提供了以下三种方式创建 Tool：

（1）使用@tool 装饰器（简洁）：这是最简单的方法，只需在 Python 函数上添加@tool 装饰器即可将其注册为 Tool。函数的名称、注释和签名会自动生成 Tool 的 name 和 description。例如：

```
from langchain.tools import tool
@tool
def get time() -> str:
    """获取当前时间。"""
    return datetime.now().strftime("%Y-%m-%d %H:%M:%S")
```

（2）继承 BaseTool 类（灵活）：如果需要对工具行为进行精细控制（如自定义参数校验等），可以通过继承 langchain.tools.BaseTool 来创建工具，并实现_run（同步）和可选的_arun（异步）方法。

（3）使用 StructuredTool.from_function 函数：将已有函数快速封装为 Tool。例如：

```
    class CalculatorInput(BaseModel):
        a: int = Field(description="first number")
        b: int = Field(description="second number")


    def multiply(a: int, b: int) -> int:
        """Multiply two numbers."""
        return a * b


    calculator = StructuredTool.from_function(
        func=multiply,
        name="Calculator",
        description="multiply numbers",
        args schema=CalculatorInput,
        return_direct=True,
       )

    print(calculator.invoke({"a": 2, "b": 3})) # 输出：6
    print(calculator.name) # 输出：multiply
    print(calculator.description) # 输出：Multiply two numbers.
    print(calculator.args) # 输出：{'a': {'title': 'A', 'type': 'integer'}, 'b': {'title':
'B', 'type': 'integer'}}
```

LangChain 引入了 StructuredTool 的概念，主要用于更好地支持结构化输入与输出。如果函数带有类型注解，或使用 Pydantic 模式定义参数结构，那么 LangChain 即可据此自动将输入解析为 JSON 格式，从而提高工具调用的准确性，其作用类似于函数调用中的参数 Schema（结构定义）。

2）Agent 调用流程与工具使用

定义好 Tool 后，需要使用 Agent 将它与 LLM 进行联动，由 Agent 负责决策和控制流程。典型流程如下：

01 理解用户意图：LLM 根据用户提问进行分析。
02 规划行动步骤：LLM 推理完成任务所需步骤。
03 选择工具：如需调用工具，LLM 根据工具描述选择最合适的工具（Tool）。
04 执行工具：Agent 调用对应工具的函数，并将函数执行结果作为观察结果（Observation）返回给 LLM。
05 根据结果继续决策：LLM 根据观察结果继续推理，决定是否继续调用工具（即返回步骤 3），或直接生成最终回答。

LangChain 内置了多个 Agent 实现（通常以 AgentType 或 create_x_agent 区分），例如基于零样本提示词的 ReAct Agent、基于 OpenAI 函数调用的 Agent 等。以 ReAct Agent 为例，其提示词模板

会指导模型按照 Thought→Action→Observation 模式循环推理，Agent 则负责解析 LLM 输出并执行对应工具。

LangChain 还提供了 initialize_agent 函数创建 Agent。开发者需要提供工具列表、LLM 实例及 Agent 类型等参数。以最常用的零样本 ReAct 智能体（Zero-shot ReAct Agent）为例，其初始化代码如下：

```
    from langchain_classic.agents import initialize_agent,AgentType
    from langchain openai import ChatOpenAI

    llm=ChatOpenAI(model="gpt-5.2", temperature=0,api_key='sk-proj-xxx')
    agent=initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
verbose=True)
```

其中，tools 为工具对象列表（即 Tool 对象数组）；verbose=True 表示输出详细运行日志，便于观察 Agent 推理过程和开发调试。

随后调用：

```
    agent.run("北京今天的天气怎么样？然后推荐一下室内活动")
```

Agent 将自动按照 ReAct 流程完成多轮工具调用，并生成最终回答。Agent 的输出日志如下：

```
    Thought: 我需要获取天气信息。
    Action: get_weather
    Action Input: 北京
    Observation: 北京：晴，25° C
    Thought: 我拿到天气了，还需要推荐活动。
    Action: search_web
    Action Input: 北京 室内活动 推荐
    Observation: ... (搜索结果)
    Thought: 我已得到所有信息。
    Final Answer: ... 综合回答 ...
```

由此可见，在 LangChain Tool 方案下，Agent 能够自动管理复杂的工具调用与多轮交互流程，开发者只需定义工具并启动 Agent 即可。这种模式不仅适用于 OpenAI 模型，也适用于其他 LLM。对于不具备原生函数调用能力的模型，则需要借助 Agent Prompt 更加严格地约束输出格式，并辅以错误解析机制。例如，LangChain 框架提供了 handle_parsing_errors 参数，可在模型输出格式不符合预期时进行重试或自动修正，从而提高 Agent 系统的稳健性。

3）Toolkit

Toolkit 是 LangChain 框架中预定义的一组工具集合，通常针对特定类型的资源或服务提前封装好一系列工具（Tool），并配套相应的 Agent 初始化配置，以进一步简化开发流程。例如：

（1）SQLDatabaseToolkit：包含与SQL数据库交互的工具（如查询执行工具、表信息获取工具），可配合create_sql_agent快速构建SQL问答Agent。

（2）RequestsToolkit：封装HTTP请求相关的工具（如GET/POST等），方便Agent调用各类API。

（3）PythonREPLTool：提供Python REPL运行环境，可用于数学计算或执行Python代码。

（4）SerpAPIWrapper：封装基于SerpAPI的搜索引擎接口，可作为检索工具接入Agent，使其具备实时互联网信息搜索能力。

通过 Toolkit，开发者无须从零编写烦琐的接口对接代码，直接调用 LangChain 内置工具集即可快速扩展 Agent 的能力边界，从而显著提升系统开发效率。

### 2.3.3　工具集成方案选择

在构建AI Agent系统时，应结合具体的业务场景、任务复杂度与模型生态来选择合适的工具集成方案。

如果系统采用GPT等支持函数调用（Function Calling）能力的模型，Function Calling机制通常是首选方案。因为它开发流程清晰，集成简单，结果可靠，主要用于工具调用路径相对明确或步骤有限的场景（如单次天气查询等）。LangChain Tools方案则在需要复杂链式工具调用、兼容多模型的场景中更具优势。基于LangChain框架开发的Agent，能够实现自主规划与多工具协同，从而胜任更复杂的任务流编排。同时，LangChain丰富的内置工具生态也有助于加快开发进度。

总之，充分理解两种机制的原理和差异，有助于开发者在构建Agent系统时做出更合理的技术选型。

## 2.4　MCP 核心技术剖析

模型上下文协议（Model Context Protocol，MCP）是AI Agent技术体系中的核心通信协议，旨在为LLM（大语言模型）与外部数据源、工具及应用系统的集成建立统一交互标准。尤其是在分布式多智能体环境中，MCP被认为是推动AI Agent工程化落地的关键技术突破。

MCP由Anthropic在2024年11月25日正式发布并开源。由于统一了大模型与外部资源之间的交互接口，MCP被业界形象地比喻为“AI领域的USB-C接口”。进入2025年后，OpenAI、Google和字节跳动等全球主流AI厂商陆续宣布支持或接入MCP生态。

### 2.4.1　MCP 的原理与实践

一般来讲，MCP是一个能够让LLM更高效地使用各类工具的协议。例如，借助MCP，开发者可以让模型使用浏览器上网查询信息，也可以让模型完成订餐等操作。

#### 1. MCP 核心架构

MCP 遵循客户端-服务器（Client-Server）架构，如图 2-8 所示。该架构包含以下几个核心模块：

（1）MCP主机（MCP Host）：发起请求的LLM应用程序，如Cursor或TRAE等。

（2）MCP客户端（MCP Client）：运行于主机程序内部，与MCP Server保持1:1的连接。

（3）MCP服务器（MCP Server）：为MCP客户端提供上下文、工具和提示词（Prompt）等能力。

（4）本地资源（Local Resource）：本地计算机中可供MCP Server安全访问的资源，如文件、数据库等。

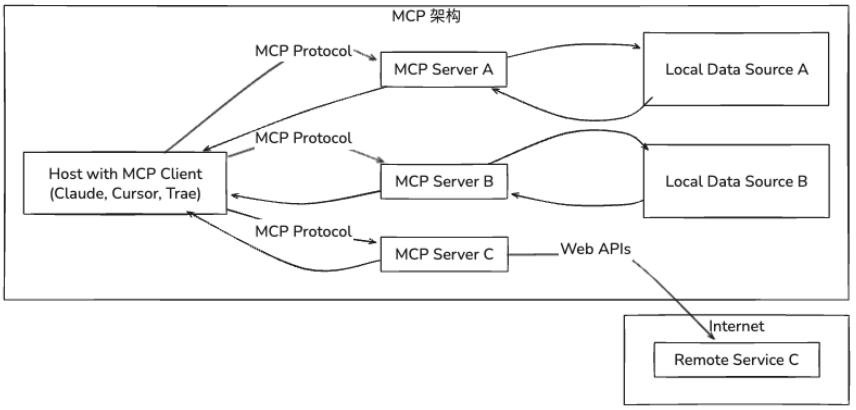


图 2-8　MCP 架构

（5）远程资源（Remote Resource）：MCP Server可以连接远程资源，如通过API接口调用的外部服务、远程数据库、第三方应用等。

下面将分别对上述核心模块进行详细介绍。

1）MCP Client

MCP Client 充当 LLM 和 MCP Server 之间的桥梁，其工作流程如下：

01 初始化与发现：Client 首先与指定的 Server 建立连接，并获取其提供的工具列表。

02 提示词与工具封装：Client 将用户的查询以及从 Server 获得的工具描述信息，按照底层 LLM 接口要求的格式进行封装，并发送给 LLM。需要注意的是，不同厂商模型对工具调用的支持能力存在差异：GPT-3.5 及以上模型支持 Function Calling（函数调用），针对此类模型，Client 需要使用 Function Calling 与 LLM 交互；而 Anthropic 的部分 Claude 模型支持 Tool Calling（工具调用），它是一种更加泛化的概念。本质上，Function Calling 只是 Tool Calling 的一种。Tool Calling 不仅支持多工具调用、多轮连续调用，未来还可拓展至检索、沙箱执行环境等非函数类工具能力。目前，主流 LLM 均已支持 Tool Calling（即兼容工具调用的交互协议）。

03 LLM 决策：LLM 分析用户请求，判断是否需要调用工具，并选择具体的调用工具及调用参数。

04 工具调用：如果 LLM 决定调用工具，Client 会向 Server 发起执行请求。

05 结果回传：Server 执行工具后，将结果返回给 Client，Client 再将结果回传给 LLM。

06 最终响应生成：LLM 结合用户请求、工具调用及执行结果，生成最终的自然语言响应。

07 响应呈现：Client 将最终响应展示给用户。

目前，Claude Desktop、Cursor 以及 TRAE 等应用均已内置 MCP Client 能力。这意味着它们可以作为客户端连接任何 MCP Server，从而动态扩展可用工具集。

2）MCP Server

MCP Server是MCP架构中的关键组件。本质上是一个遵循MCP规范的独立进程。多数MCP Server基于Node.js或Python运行环境部署，并在本地启动。在运行过程中既可按需接入网络，也支持

完全离线的本地运行模式。

MCP Server 主要由以下三部分组成：

（1）资源：供客户端读取的数据，例如API响应结果、本地文件内容等。

（2）工具：可供LLM调用的函数。

工具是 MCP Server 的核心功能载体，在整个 MCP 架构中占据关键地位。成熟的 MCP Server 通常会提供丰富的标准化工具集。以导航场景的 MCP Server 为例，其内部往往集成了多个面向不同业务场景的工具。图 2-9 展示了高德地图 amap-maps 这个 MCP Server 提供的工具列表。


图 2-9　amap-maps 提供的工具列表

（3）提示词：预先编写的指令模板，用于引导LLM高效完成特定任务。

其核心工作流程为：MCP Server 启动后，通过 MCP 向 MCP Host 上报自身提供的工具、资源及提示词这三类能力，完成服务注册；MCP Host 完成权限校验后，向 Server 下发工具调用或资源读取请求；Server 接收请求并解析协议内容，依据请求参数调用对应的内部工具，完成业务处理（如数据查询、逻辑运算等）；最后，Server 按 MCP 标准封装执行结果，并回传给 MCP Host，由 MCP Host 交给 LLM 使用。

3）MCP Host

MCP Host是MCP架构中的核心编排组件，负责协调LLM与各类MCP Server之间的交互与调度。

其核心工作流程为：通过MCP接入多个MCP Server，将Server提供的工具、资源提示词以标准化协议格式封装后提供给模型；当模型发起工具调用请求时，MCP Host负责协议层参数校验、执行调度以及调用结果回传等全流程管控。此外，MCP Host还承担安全与权限控制职责，确保模型在受

控范围内访问数据与系统。

综上，MCP Host 的核心功能可归纳为以下五类：

（1）LLM 承载与管理：

- 运行并维护LLM会话与上下文状态。
- 向模型提供可用的MCP工具能力。

（2）MCP Server 发现与接入：

- 注册并管理多个MCP Server。
- 获取并维护工具能力描述与参数规范。

（3）工具调用编排：

- 接收模型生成的工具调用请求。
- 负责参数校验、执行调度，并将调用结果返回模型。

（4）安全与权限控制：

- 控制工具可见范围与调用边界。
- 防止提示词注入导致的越权访问。

（5）交互与生命周期管理：

- 协调模型、工具和用户之间的交互流程。
- 管理会话、错误处理与调用状态。

MCP Host 本质上是支持 MCP 的宿主应用，是用户与模型、工具进行交互的载体。国外主流的 MCP Host 包括 Claude Desktop、Cursor、Cline、Cherry Studio 等，如图 2-10 所示。


图 2-10　国外的 MCP Host

国内主流的MCP Host包括TRAE、CodeBuddy以及Qoder等，如图2-11所示。

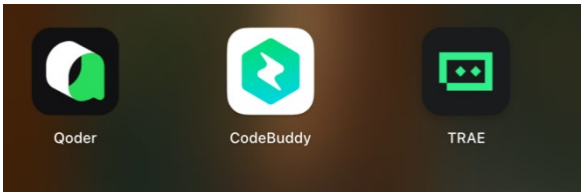


图 2-11　国内的 MCP Host

### 2. MCP 与 TRAE 的结合实践

本节以 TRAE 这款 MCP Host 为例，演示 MCP Server 的安装方法以及第三方 MCP Server 的集成流程。

01 打开 TRAE，在设置中找到 MCP 配置面板，如图 2-12 所示。


图 2-12　在 TRAE 中添加 MCP Server 入口

02 单击“添加”按钮，选择市场，可以看到如图 2-13 所示的 MCP Server 列表。

03 在 MCP Server 列表中找到 Fetch。它是专门用于抓取网页内容的 MCP Server。单击其右侧的“添加”按钮，如图 2-14 所示。

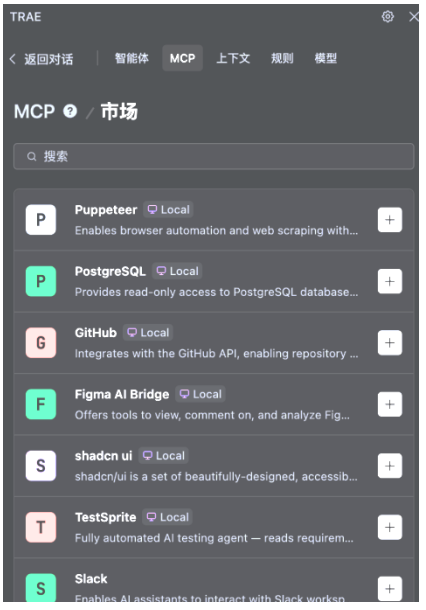


图 2-13　MCP Server 列表


图 2-14　添加 Fetch MCP Server

04 单击“添加”按钮之后，会弹出添加 Fetch 的确认页面，主要展示 Fetch 的 JSON 配置（注意：从公开市场添加的 MCP Server，默认已配置好了 JSON 配置。如果是自己开发的 MCP Server，则需要手动编写 JSON 配置。），确认 JSON 配置无误后，继续单击“确认”按钮，如图 2-15 所示。

05 在弹出的页面中选择一个智能体配置 Fetch，如图 2-16 所示。


图 2-15　Fetch 的 JSON 配置


图 2-16　Fetch 被添加到规划行程智能体中

至此，从公开市场把MCP Server添加到智能体的整个流程已经完成。

TRAE 还提供了另外一种集成 MCP Server 的方式，即手动配置 MCP Server。

01 打开 TRAE，在设置中找到 MCP 配置面板，然后单击“添加”按钮，选择手动配置，如图 2-17 所示。


图 2-17　手动配置 MCP Server

02 将阿里云 MCP 市场中的高德地图 MCP Server 的 JSON 配置粘贴到配置窗口中：

```
{
  "mcpServers": {
    "amap-maps": {
```

```
          "type": "sse",
          "description": "高德地图 MCP Server 现已覆盖 12 大核心接口，提供全场景覆盖的地理信息服务，
包括地理编码、逆地理编码、IP 定位、天气查询、骑行路径规划、步行路径规划、驾车路径规划、公交路径规划、距离
测量、关键词搜索、周边搜索、详情搜索等。",
          "isActive": true,
          "name": "阿里云百炼 Amap Maps",
          "baseUrl": "https://dashscope.aliyuncs.com/api/v1/mcps/amap-maps/sse",
          "headers": {
            "Authorization": "Bearer ${DASHSCOPE API KEY}"
          }
        }
      }
    }
```

参数说明如下：

- mcpServers：MCP Server集合配置，用于声明MCP Host可接入的所有MCP Server。
- amap-maps：MCP Server的唯一标识，用于内部引用与管理（如amap-maps）。
- type：MCP Server的通信类型，常见取值为sse（基于Server-Sent Events的长连接方式）和stdio（基于标准输入/输出）。前者通常用于云端MCP服务，后者多用于本地进程MCP服务。
- description：对MCP Server能力的功能性描述，主要作为工具能力说明注入LLM（大语言模型）。
- isActive：是否启用该MCP Server，值为true表示启动（启动时自动加载并注册到MCP主机）；值为false表示关闭（配置保留但不参与运行）。
- name：MCP Server的展示名称，主要用于日志、UI展示以及模型侧的工具标识，用于提高可读性。
- baseUrl：MCP Server的访问地址，MCP Host通过该地址与MCP Server建立连接。
- headers：请求MCP Server时携带的HTTP请求头，主要用于身份认证和权限校验。
- DASHSCOPE_API_KEY：阿里云百炼大模型的API Key，可在阿里云官网获取。

粘贴完成后，高德地图 MCP Server 的 JSON 配置如图 2-18 所示。

**03** 单击“确认”按钮后，选择一个智能体使用该 MCP Server。

笔者提前在 TRAE 中创建了一个查询城市景点的智能体，将高德地图的 MCP Server 配置到该智能体中，即可与它进行对话。例如，输入“自驾上海，从北京出发”。智能体经过思考后，识别出这是一个路线规划任务，因此调用 amap-maps 这个 MCP Server，如图 2-19 所示。

接下来，通过控制台输出的日志详细分析整个调用过程。

（1）获取出发地和目的地经纬度。模型首先将出发地和目的地解析为经纬度，然后分别调用高德地图提供的 maps_geo 工具查询北京和上海的经纬度。整个过程可以通过控制台输出日志进行查看和验证。


图 2-18　手动配置高德地图 MCP Server


图 2-19　查询城市景点智能体调用 MCP Server 的过程

- 北京经纬度获取:

```
    2025-11-16T01:10:38.316+08:00 [info] [mcp.config.usrlocalmcp.amap-maps]
MCPServerManager#callTool (maps geo): {
    "address": "北京市朝阳区劲松七区"
    }

    2025-11-16T01:10:38.418+08:00 [info] [mcp.config.usrlocalmcp.amap-maps]
MCPServerManager#callTool (maps_geo) result: {
      "content": [
        {
          "type": "text",
          "text":
          {
            "results": [
              {
                "country": "中国",
                "province": "北京市",
                "city": "北京市",
                "citycode": "010",
```

```
              "district": "朝阳区",
              "street": [],
              "number": [],
              "adcode": "110105",
              "location": "116.447643,39.880819",
              "level": "住宅区"
            }
          ]
        }
      }
    ],
    "isError": false
  }
```

- 上海经纬度获取：

```
  2025-11-16T01:10:49.589+08:00 [info] [mcp.config.usrlocalmcp.amap-maps]
MCPServerManager#callTool (maps geo): {
  "address": "上海市"
  }

  2025-11-16T01:10:49.705+08:00 [info] [mcp.config.usrlocalmcp.amap-maps]
MCPServerManager#callTool (maps_geo) result: {
    "content": [
      {
        "type": "text",
        "text": {
           "results": [
             {
               "country": "中国",
               "province": "上海市",
               "city": "上海市",
               "citycode": "021",
               "district": [],
               "street": [],
               "number": [],
               "adcode": "310000",
               "location": "121.473667,31.230525",
               "level": "省"
             }
           ]
         }
      }
    ],
    "isError": false
  }
```

（2）规划驾车路线。获取到出发地和目的地的经纬度后，LLM（大语言模型）开始规划自驾路线：调用高德地图提供的驾车路线工具 maps_direction_driving，规划从北京到上海的路线，并返回里程和预计驾驶时长。

调用驾车路线工具的日志如下（由于日志较长，这里省略了部分中间日志）：

```
  2025-11-16T01:10:58.190+08:00 [info] [mcp.config.usrlocalmcp.amap-maps]
MCPServerManager#callTool (maps_direction_driving): {
```

```
    "origin": "116.447643,39.880819",
    "destination": "121.473667,31.230525"
    }

    2025-11-16T01:10:58.449+08:00 [info] [mcp.config.usrlocalmcp.amap-maps]
MCPServerManager#callTool (maps direction driving) result: {
      "content": [
        {
          "type": "text",
          "text": {
            "origin": "116.447643,39.880819",
            "destination": "121.473667,31.230525",
            "paths": [
              {
                "path": "",
                "distance": "1217501",
                "duration": "45161",
                "steps": [
                  {
                    "instruction": "沿劲松南路向东行驶 1.1 千米右转",
                    "road": "劲松南路",
                    "distance": "1129",
                    "orientation": "东",
                    "duration": "135"
                  },
                  ....
                  {
                    "instruction": "沿 G2 京沪高速向东南行驶 887 米减速行驶到达收费站",
                    "road": "G2 京沪高速",
                    "distance": "887",
                    "orientation": "东南",
                    "duration": "84"
                  },
                  ...
                  {
                    "instruction": "沿 G2 京沪高速途经沪杭铁路立交桥、东沙江桥、西泾烬桥向东行驶 5.1
千米减速行驶到达收费站",
                    "road": "G2 京沪高速",
                    "distance": "5109",
                    "orientation": "东",
                    "duration": "439"
                  }
                ]
              }
            ]
          }
        }
      ],
      "isError": false
    }
```


（3）整理并生成最终结果。获取到驾车路线工具返回的结果后，LLM 开始整理信息，并生成最终回复：

```
    已为你用高德地图规划从北京朝阳区劲松七区到上海的自驾路线，核心信息如下：
```

- 起点坐标：116.447643, 39.880819（劲松七区）
- 终点坐标：121.473667, 31.230525（上海中心坐标，接近人民广场）
- 预计总里程：约 1217.5 公里
- 预计驾驶时长：约 12.5 小时（不含休息与拥堵，建议分段驾驶）
- 主干路线走向：劲松 → 分钟寺桥 → G2 京沪高速（北京段）→ 经天津、河北 → 转 G18 荣乌高速、S27 沾临高速、G25 长深高速 → 转 G15 沈海高速（盐城–苏通大桥）→ G1522 常台高速 → 正仪枢纽接回 G2 京沪 → 进入上海市区（武宁路、延安高架等） → 人民广场附近

沿途城市与常见休息点建议（可选停靠）

- 第一段：北京南部出城后，天津/沧州方向（加油、洗手间、餐食）
- 中段：济南北、淄博/临沂、连云港/盐城（服务区较多）
- 后段：南通、苏州/常熟（进入长江以南路段后车流增大，注意避峰）

出行提醒

- 日期：你提供的时间显示为 11 月中旬，若为周日，自驾不受北京工作日尾号限行影响；如跨周一进出市区需关注限行政策
- 费用：京沪高速及多段高速通行费合计较高，建议备好 ETC 或充足现金（不同车型与实际路段收费略有差异）
- 休息：建议每 2–3 小时休息一次，避免疲劳驾驶
- 进入上海中心区：常有高架与快速路分流，注意导航实时指引
- 若需充电（新能源）：可告知你的车型与续航，我可以沿途插入高德的充电站点

下一步我可以为你生成可直接打开的高德行程地图链接，并按顺序加入沿途停靠的服务区或城市。请确认：

- 终点是否需要更精确位置（例如：人民广场、浦东张江、虹桥火车站、浦东国际机场等）
- 是否要添加沿途休息/餐饮/加油/充电站点（例如：天津–沧州–淄博–临沂–盐城–南通）
- 是否需要避开城市高峰时段或分两天走（可安排在济南/淮安/南通过夜）

确认后我将为你生成一条高德可视化行程地图链接（含站点与顺序），并提供详细路况与调度建议。

以上整个交互过程如图 2-20 所示。

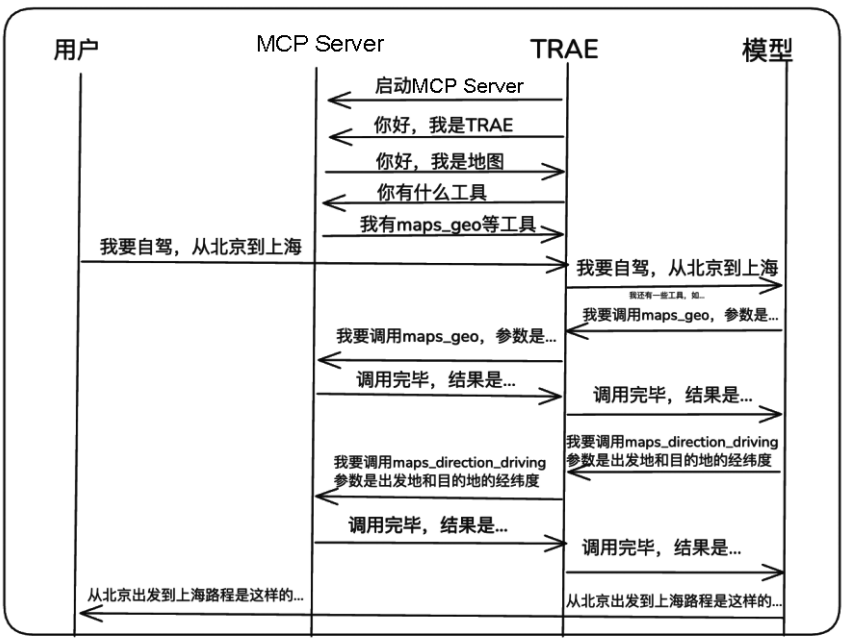


图 2-20　TRAE 中智能体与 MCP Server 的交互流程

至此，MCP的基本原理及实践已经介绍完毕。下一小节将介绍如何开发一个MCP Server，并深

度解析MCP的底层运行机制。

### 2.4.2　MCP 的底层运行机制

基于上一小节对MCP原理与实践的介绍，本节将以自定义MCP Server的开发实践为切入点，系统剖析MCP的交互规范与运行机制。

#### 1. 创建 MCP Server

在动手创建 MCP Server 之前，需要做好环境准备。本项目将选用 Python 作为主要编程语言，原因如下：

（1）生态优势：Python在自然语言处理领域拥有成熟的工具链与丰富的库支持，例如PyTorch、TensorFlow等主流深度学习框架均以Python为主要开发语言，便于后续与AI能力集成。

（2）丰富的模型支持：许多主流大语言模型和平台都提供了Python SDK，这大大降低了集成和调用各类模型能力的门槛。

（3）丰富的框架：基于Python开发的LangChain和LangGraph框架，可以加速AI应用的开发。

（4）上手简单：Python语法清晰、易于编写，无须复杂的开发环境配置，即可快速运行脚本，适合原型开发与快速验证。

当然，MCP 本身是语言中立的协议，也可以选择 Java、Node.js 或其他语言进行实现与探索。为了保持示例的语言统一性，本书均基于 Python 展开。

首先，请确保环境中已安装 Python 3.10 或更高版本（可从 Python 官网下载安装）。确认版本符合要求后，创建一个项目目录 mcp，用于后续的代码编写与实验：

```
➜  PycharmProjects mkdir mcp
➜  mcp pwd
/Users/jamlee/PycharmProjects/mcp
```

随后，创建一个 Python 虚拟环境，并将它激活：

```
➜  mcp python3 -m venv venv            # 创建 Python 虚拟环境
➜  mcp source venv/bin/activate        # 激活虚拟环境
(venv) ➜  mcp
```

在 python3 -m venv venv 命令中，第一个 venv 是 Python 内置的虚拟环境模块，第二个 venv 是虚拟环境的目录名称（可以自定义）。

虚拟环境具有以下优点：

- 隔离项目依赖：不同项目可以使用不同版本的软件包，而不会发生冲突。
- 保持系统整洁：避免在系统Python中安装项目特定的软件包。
- 便于管理：每个项目拥有独立的运行环境。

最后，基于虚拟环境安装 MCP 及 HTTP 相关的软件包：

```
(venv) ➜  mcp pip install 'mcp[cli]' httpx
```

其中，mcp[cli]是 MCP 服务相关的工具集，httpx 包主要用于请求天气等 API。

安装成功后，部分输出日志如下：

```
Collecting httpx
 Using cached httpx-0.28.1-py3-none-any.whl.metadata (7.1 kB)
Collecting mcp[cli]
...
(venv) ➜  mcp
```

至此，环境准备已就绪。接下来，我们将以 MCP 官方示例为基础，结合本节实际需求进行定制化调整，并正式开始创建自定义的 MCP Server：

01 打开 MCP 官网示例地址：https://modelcontextprotocol.io/docs/develop/build-server。

02 将 weather.py 代码复制到刚才创建的 mcp 目录中，如下所示：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from typing import Any
import httpx
from mcp.server.fastmcp import FastMCP


# 初始化 FastMCP server
mcp = FastMCP("weather", log_level="ERROR")


# 常量
NWS_API_BASE = "https://api.weather.gov"
USER AGENT = "weather-app/1.0"

# 请求天气数据的异步函数
async def make nws request(url: str) -> dict[str, Any] | None:
    """Make a request to the NWS API with proper error handling."""
    headers = {
        "User-Agent": USER AGENT,
        "Accept": "application/geo+json"
    }
    # 使用 httpx 异步客户端发送 GET 请求
    async with httpx.AsyncClient() as client:
        try:
            response = await client.get(url, headers=headers, timeout=30.0)
            response.raise_for_status()
            return response.json()
        except Exception:
            return None

# 格式化天气警告信息的函数
def format_alert(feature: dict) -> str:
    """Format an alert feature into a readable string."""
    props = feature["properties"]
    return f"""
Event: {props.get('event', 'Unknown')}
Area: {props.get('areaDesc', 'Unknown')}
Severity: {props.get('severity', 'Unknown')}
Description: {props.get('description', 'No description available')}
Instructions: {props.get('instruction', 'No specific instructions provided')}
```

```
    """

    # 获取天气警告的异步函数，是我们的第一个工具，用于获取指定州的天气警告参数：state（州的两位字母代码）
    # @mcp.tool()在 Python 中就是一个装饰器，这个装饰器的用途是把函数注册为工具，它会从函数的注释里面提取这个函数的用途，以及每个参数的含义，以便模型决定使用这个函数的最佳时机
    # 比如在这个函数中，它提取的内容就包括如下几点：
    # 第一：函数名是 get_alerts
    # 第二：函数的参数是 state
    # 第三：这个函数的功能
    # 第四：每个参数的功能
    # 这些提取的内容最终都会转化为工具的信息，并在实际调用的时候传递给模型，以便帮助模型做决策
    @mcp.tool()
    async def get_alerts(state: str) -> str:
        """获取指定州的天气警告

        参数:
            state: 州的两位字母代码 (例如: CA, NY)
        """
        url = f"{NWS_API_BASE}/alerts/active/area/{state}"
        data = await make_nws_request(url)

        if not data or "features" not in data:
            return "无法获取天气警告或未找到警告。"

        if not data["features"]:
            return "此州暂无活动天气警告。"

        alerts = [format_alert(feature) for feature in data["features"]]
        return "\n---\n".join(alerts)

    # 获取天气预测的异步函数，是我们的第二个工具，用于获取指定位置的天气预测参数：latitude（纬度），longitude（经度）
    @mcp.tool()
    async def get_forecast(latitude: float, longitude: float) -> str:
        """获取指定位置的天气预测

        参数:
            latitude: 位置的纬度
            longitude: 位置的经度
        """
        # 先获取指定位置的天气预测办公室端点
        points_url = f"{NWS_API_BASE}/points/{latitude},{longitude}"
        points_data = await make_nws_request(points_url)

        if not points_data:
            return "无法获取此位置的天气预测数据。"

        # 从响应中获取对应天气预报的 URL 再去请求获取详细的天气预测数据
        forecast_url = points_data["properties"]["forecast"]
        forecast_data = await make_nws_request(forecast_url)

        if not forecast_data:
            return "无法获取此位置的详细天气预测数据。"

        # 格式化为可读的天气预报
```

```
        periods = forecast_data["properties"]["periods"]
        forecasts = []
        for period in periods[:5]:  # 只显示未来 5 个时间周期的天气预测
            forecast = f"""
    {period['name']}:
    Temperature: {period['temperature']}°{period['temperatureUnit']}
    Wind: {period['windSpeed']} {period['windDirection']}
    Forecast: {period['detailedForecast']}
    """
            forecasts.append(forecast)

        return "\n---\n".join(forecasts)


    if __name__ == "__main__":
        # 初始化并运行服务器
    mcp.run(transport='stdio') # stdio 是一种标准输入输出传输协议，它允许 MCP server 与 MCP Host
进行交互
```

保存代码后，运行 weather.py 脚本，名为 weather 的 MCP Server 即创建完成。接下来，需要配置一个支持 MCP 的宿主应用，本节将继续以 TRAE 为例进行演示。

### 2. MCP Server 与 TRAE 结合

在 TRAE 中打开 MCP 配置面板，随后写入以下 MCP Server 的 JSON 配置（见图 2-21）：


图 2-21　在 TRAE 中添加自定义 MCP Server

```
{
 "mcpServers": {
   "weather": {
     "command": "/Users/jamlee/PycharmProjects/mcp/venv/bin/python",
     "args": [
       "/Users/jamlee/PycharmProjects/mcp/weather.py"
     ]
   }
 }
}
```

点击确认按钮后，weather MCP Server 就会自动注册到 MCP Host 中。

此时，控制台输出的日志如下：

```
    2025-11-23T11:44:44.831+08:00 [info] [mcp.config.usrlocalmcp.weather]
MCPServerManager#start Connected.
    2025-11-23T11:44:44.832+08:00 [info] [mcp.config.usrlocalmcp.weather]
MCPServerManager#listTools Listing tools...
    2025-11-23T11:44:44.837+08:00 [info] [mcp.config.usrlocalmcp.weather]
MCPServerManager#listTools Got tools: get_alerts, get_forecast
```

从以上日志可以看出，weather MCP Server 已经与 TRAE 成功建立连接，并向 TRAE 暴露了两个工具：get_alerts 和 get_forecast。

为验证其运行效果，可在对话框中输入“纽约明天天气怎么样”，输出结果如图2-22所示。


图 2-22　在 TRAE 中测试自定义 MCP Server

由图可见，模型准确识别到了自定义的MCP Server——weather，并通过语义分析确认其中的get_forecast工具能够解决当前问题。随后，模型自动生成工具调用指令，由TRAE执行该工具，并将返回的结构化数据整合为最终答案。整个过程验证了自定义MCP Server的运行状态正常，且通信链路工作稳定。

### 3. 创建协议日志解析脚本

至此，读者或许会问：虽然我们已经成功创建了自定义的MCP Server，但整个协议的运行机制仍然像一个“黑盒”。这是因为，在前面的示例中，我们主要依赖MCP官方提供的Python库完成服务注册、工具声明、请求解析与结果返回等工作，这些库帮助我们屏蔽了底层通信细节。

那么，这些库到底帮我们完成了哪些工作？MCP Host与MCP Server之间的协议消息是如何传递的？工具调用请求又是如何被解析并返回结果的？接下来，我们将深入MCP的底层通信过程，揭示协议在实际运行过程中的具体细节。

在本地部署场景中，MCP Host 与 MCP Server 通常通过标准输入和输出（stdio）进行进程间通信。协议消息通过 stdin 和 stdout 在两个进程间传递，而在默认状态下，开发者无法直接观察到这些原始消息。为了清晰地展现这一过程，我们将编写一个轻量级的协议日志解析脚本，用于捕获并格式化输出 MCP 通信过程中的请求与响应内容。该脚本命名为 mcp_logger.py，代码如下：

```
#!/usr/bin/env python3

import sys
import subprocess
import threading
import argparse
import os

# --- 配置 ---
LOG FILE = os.path.join(os.path.dirname(os.path.realpath(  file  )), "mcp io.log")
# --- 配置结束 ---

# --- 参数解析 ---
parser = argparse.ArgumentParser(
    description="封装一个命令，原样传递 STDIN/STDOUT 并记录日志。",
    usage="%(prog)s <command> [args...]"
)
# 捕获命令及所有后续参数
parser.add argument('command', nargs=argparse.REMAINDER,
                    help='要执行的命令及其参数。')

open(LOG FILE, 'w', encoding='utf-8')

if len(sys.argv) == 1:
    parser.print help(sys.stderr)
    sys.exit(1)

args = parser.parse args()

if not args.command:
    print("错误：未提供任何命令。", file=sys.stderr)
    parser.print_help(sys.stderr)
    sys.exit(1)

target_command = args.command
# --- 参数解析结束 ---

# --- I/O 转发函数 ---
```

```
# 这些函数将在独立线程中运行

def forward and log stdin(proxy stdin, target stdin, log file):
    """从代理 STDIN 读取、记录并写入目标进程的 STDIN。"""
    try:
        while True:
            # 按行读取脚本的真实标准输入
            line_bytes = proxy_stdin.readline()
            if not line bytes:  # EOF
                break

            # 为日志解码（假设为 UTF-8，可调整）
            try:
                line_str = line_bytes.decode('utf-8')
            except UnicodeDecodeError:
                line_str = f"[非 UTF-8 数据，{len(line_bytes)} 字节]\n"

            # 记录日志
            log_file.write(f"输入: {line_str}")
            log file.flush()

            # 将原始字节写入目标进程
            target stdin.write(line bytes)
            target_stdin.flush()

    except Exception as e:
        # 记录错误
        try:
            log_file.write(f"!!! STDIN 转发错误: {e}\n")
            log_file.flush()
        except:
            pass

    finally:
        # 关闭目标进程 STDIN（向其发送 EOF）
        try:
            target_stdin.close()
            log_file.write("--- STDIN 流已关闭 ---\n")
            log_file.flush()
        except Exception as e:
            try:
                log_file.write(f"!!! 关闭目标 STDIN 时出错: {e}\n")
                log file.flush()
            except:
                pass


def forward_and_log_stdout(target_stdout, proxy_stdout, log_file):
    """从目标 stdout 读取、记录并写入代理 stdout。"""
    try:
        while True:
            line bytes = target stdout.readline()
            if not line_bytes:
                break
```

```
            try:
                line_str = line_bytes.decode('utf-8')
            except UnicodeDecodeError:
                line_str = f"[非 UTF-8 数据，{len(line_bytes)} 字节]\n"

            log file.write(f"输出: {line str}")
            log_file.flush()

            proxy stdout.write(line bytes)
            proxy_stdout.flush()

    except Exception as e:
        try:
            log_file.write(f"!!! STDOUT 转发错误: {e}\n")
            log_file.flush()
        except:
            pass
    finally:
        try:
            log file.flush()
        except:
            pass

# --- 主执行部分 ---
process = None
log f = None
exit_code = 1

try:
    log_f = open(LOG_FILE, 'a', encoding='utf-8')

    # 启动目标进程
    process = subprocess.Popen(
        target_command,
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        bufsize=0
    )

    stdin thread = threading.Thread(
        target=forward_and_log_stdin,
        args=(sys.stdin.buffer, process.stdin, log f),
        daemon=True
    )

    stdout thread = threading.Thread(
        target=forward_and_log_stdout,
        args=(process.stdout, sys.stdout.buffer, log f),
        daemon=True
    )

    # STDERR 转发函数
    def forward_and_log_stderr(target_stderr, proxy_stderr, log_file):
        """从目标 stderr 读取、记录并写入代理 stderr。"""
```

```
            try:
                while True:
                    line bytes = target stderr.readline()
                    if not line_bytes:
                        break
                    try:
                        line_str = line_bytes.decode('utf-8')
                    except UnicodeDecodeError:
                        line str = f"[非 UTF-8 数据，{len(line bytes)} 字节]\n"
                    log_file.write(f"STDERR: {line_str}")
                    log_file.flush()
                    proxy_stderr.write(line_bytes)
                    proxy_stderr.flush()
            except Exception as e:
                try:
                    log_file.write(f"!!! STDERR 转发错误: {e}\n")
                    log_file.flush()
                except:
                    pass
            finally:
                try:
                    log_file.flush()
                except:
                    pass

        stderr thread = threading.Thread(
            target=forward_and_log_stderr,
            args=(process.stderr, sys.stderr.buffer, log_f),
            daemon=True
        )

        # 启动所有线程
        stdin_thread.start()
        stdout_thread.start()
        stderr thread.start()

        process.wait()
        exit code = process.returncode

        stdin thread.join(timeout=1.0)
        stdout thread.join(timeout=1.0)
        stderr_thread.join(timeout=1.0)

    except Exception as e:
        print(f"MCP 日志记录器错误: {e}", file=sys.stderr)
        if log f and not log f.closed:
            try:
                log_f.write(f"!!! MCP 主错误: {e}\n")
                log f.flush()
            except:
                pass
        exit code = 1

    finally:
        if process and process.poll() is None:
```

```
            try:
                process.terminate()
                process.wait(timeout=1.0)
            except:
                pass
            if process.poll() is None:
                try:
                    process.kill()
                except:
                    pass

        if log_f and not log_f.closed:
            try:
                log_f.close()
            except:
                pass

        sys.exit(exit_code)
```

该脚本接收的参数即为启动 MCP Server 的命令。以 weather 服务为例，在运行 mcp_logger.py 时，只需将启动该服务的命令作为参数传入，如图 2-23 所示。该脚本之所以能够成功截获通信日志，关键在于本地 MCP Server 默认采用 stdio（标准输入/输出）作为与 MCP Host 通信的通道。mcp_logger.py 正是通过代理并监听该通道，记录 MCP Host 与 MCP Server 之间传递的协议报文，从而帮助我们观察 MCP 底层通信的完整过程。


图 2-23　截获 MCP Server 与 TRAE 的通信日志

整体交互流程如图2-24所示：通信由MCP Host（TRAE）发起，经由中间代理模块mcp_logger.py完成报文转发与日志记录，最终与自定义的MCP Server（weather）进行双向数据交互。

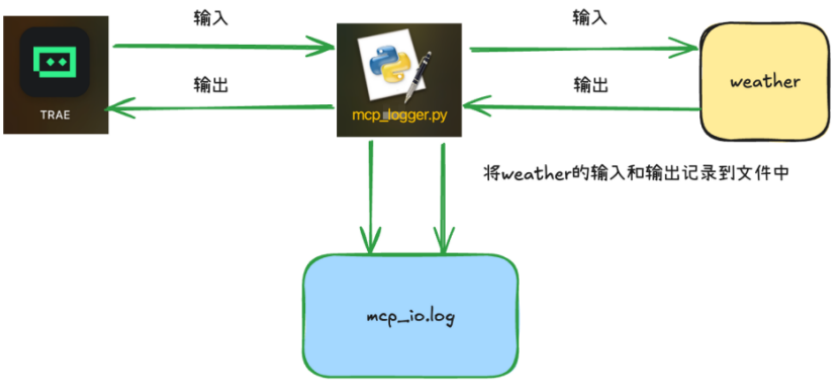


图 2-24　TRAE、mcp_logger 和 MCP Server 的交互流程

具体工作流程如下：

01 TRAE（MCP Host）向 mcp_logger.py 发送请求数据。
02 mcp_logger.py 接收请求后，将其转发给 weather MCP Server。
03 weather MCP Server 处理请求并返回结果。
04 mcp_logger.py 接收来自 weather MCP Server 的响应，并将其回传给 TRAE。

在此过程中，mcp_logger.py 会将交互过程中的输入与输出数据统一记录到日志文件 mcp_io.log 中，便于后续进行调试、协议分析或日志审计。

简单来说，该流程以mcp_logger.py作为中间代理，实现了MCP Host与MCP Server之间的透明转发，同时完成了对底层通信数据的完整日志记录，为协议的观测与分析提供了可靠的数据支撑。

#### 4. 分析协议日志

下面打开日志文件mcp_io.log，分析MCP Server的启动流程和初始化细节。

需要注意的是，每行日志开头的标记（如“输入：”“输出：”）均由mcp_logger.py脚本额外添加，用于帮助我们区分消息方向；其后的JSON字符串才是MCP Server与TRAE之间实际传输的协议数据。

其中，“输入”表示TRAE发送给MCP Server的请求消息；“输出”表示MCP Server返回给TRAE的响应消息。通过区分输入与输出，我们可以更直观地观察MCP在初始化、能力声明和工具调用等阶段的通信过程。

下面的日志中，每一行都表示一次协议消息的输入或输出：

```
输入：
{
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "capabilities": {},
    "clientInfo": {
      "name": "Trae",
      "version": "1.104.3"
    }
```

```
    },
    "jsonrpc": "2.0",
    "id": 0
}
```

第一行日志是 TRAE 发送给 MCP Server 的初始化请求。可理解为：TRAE 正在向 MCP Server 表明自己的客户端身份、软件版本以及所使用的 MCP 版本。这里表示客户端为 TRAE，软件版本为 1.104.3，所使用的 MCP 版本为 2025-06-18。

```
输出：
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "experimental": {},
      "prompts": {
        "listChanged": false
      },
      "resources": {
        "subscribe": false,
        "listChanged": false
      },
      "tools": {
        "listChanged": false
      }
    },
    "serverInfo": {
      "name": "weather",
      "version": "1.22.0"
    }
  }
}
```

第二行日志是 MCP Server 返回给 TRAE 的初始化响应。针对 TRAE 发送的初始化请求，MCP Server 会返回自己的基本信息，相当于回应：“你好，我是 weather 服务，当前版本号为 1.22.0，支持的 MCP 版本同样是 2025-06-18”。

```
输入：
{
  "method": "notifications/initialized",
  "jsonrpc": "2.0"
}
```

第三行日志由 TRAE 发送给 MCP Server，表示客户端已经收到并确认了 MCP Server 的初始化响应。可理解为 TRAE 正在通知 MCP Server：初始化信息已接收，连接准备就绪，可以继续后续交互。

```
输入：
{
  "method": "tools/list",
  "jsonrpc": "2.0",
  "id": 1
}
```

第四行日志仍然由 TRAE 发送给 MCP Server，用于查询当前 MCP Server 提供的工具列表。具体而言，TRAE 在完成协议初始化后，会进一步向 MCP Server 发起工具列表查询请求，以便了解当前服务提供了哪些可调用的工具，以及这些工具的参数定义。

输出：


```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "tools": [
      {
        "name": "get_alerts",
        "description": "获取指定州的天气警告\n\n    参数:\n        state: 州的两位字母代码 (例如: CA, NY)\n    ",
        "inputSchema": {
          "properties": {
            "state": {
              "title": "State",
              "type": "string"
            }
          },
          "required": [
            "state"
          ],
          "title": "get_alertsArguments",
          "type": "object"
        },
        "outputSchema": {
          "properties": {
            "result": {
              "title": "Result",
              "type": "string"
            }
          },
          "required": [
            "result"
          ],
          "title": "get alertsOutput",
          "type": "object"
        }
      },
      {
        "name": "get_forecast",
        "description": "获取指定位置的天气预测\n\n    参数:\n        latitude: 位置的纬度\n    longitude: 位置的经度    \n    ",
        "inputSchema": {
          "properties": {
            "latitude": {
              "title": "Latitude",
              "type": "number"
            },
            "longitude": {
              "title": "Longitude",
              "type": "number"
```

```
              }
            },
            "required": [
              "latitude",
              "longitude"
            ],
            "title": "get_forecastArguments",
            "type": "object"
          },
          "outputSchema": {
            "properties": {
              "result": {
                "title": "Result",
                "type": "string"
              }
            },
            "required": [
              "result"
            ],
            "title": "get forecastOutput",
            "type": "object"
          }
        }
      ]
    }
  }
```

第五行日志是 MCP Server 返回给 TRAE 的工具列表响应。从日志中可以看到，当前 weather 服务已经向 TRAE 成功注册了两个可用工具：get_alerts 和 get_forecast。这意味着，TRAE 后续可根据用户意图自动匹配相应的工具，以完成天气预警或天气预报查询。

接下来，我们分别解析这两个工具的定义。

（1）get_alerts：

```
  {
          "name": "get alerts",
          "description": "获取指定州的天气警告\n\n    参数:\n        state: 州的两位字母代码 (例如: CA, NY)\n    ",
          "inputSchema": {
            "properties": {
              "state": {
                "title": "State",
                "type": "string"
              }
            },
            "required": [
              "state"
            ],
            "title": "get_alertsArguments",
            "type": "object"
          },
          "outputSchema": {
            "properties": {
```

```
              "result": {
                "title": "Result",
                "type": "string"
              }
            },
            "required": [
              "result"
            ],
            "title": "get alertsOutput",
            "type": "object"
          }
        }
```

（2）get_forecast：

```
    {
            "name": "get_forecast",
            "description": "获取指定位置的天气预测\n\n    参数:\n       latitude: 位置的纬度\n
longitude: 位置的经度   \n    ",
            "inputSchema": {
              "properties": {
                "latitude": {
                  "title": "Latitude",
                  "type": "number"
                },
                "longitude": {
                  "title": "Longitude",
                  "type": "number"
                }
              },
              "required": [
                "latitude",
                "longitude"
              ],
              "title": "get_forecastArguments",
              "type": "object"
            },
            "outputSchema": {
              "properties": {
                "result": {
                  "title": "Result",
                  "type": "string"
                }
              },
              "required": [
                "result"
              ],
              "title": "get forecastOutput",
              "type": "object"
            }
          }
```

从日志中可以看到，这两个工具的描述都遵循同一套结构。由于后续将使用 get_forecast 工具查询纽约明天的天气，因此这里以 get_forecast 为例，重点分析其定义。

get_forecast 的核心信息均通过一个 JSON 对象进行描述。该 JSON 对象本质上是对工具函数的结构化说明，包含函数名称、功能描述、输入参数和输出结果等信息。其主要字段含义如下：

- name：表示工具名称，即函数名。
- description：表示工具的功能说明，通常来自函数中的docstring。
- inputSchema：表示工具输入参数的结构定义。
- outputSchema：表示工具返回结果的结构定义。

其中，description 通常由@mcp.tool()装饰器从函数的 docstring 中提取，用于向模型描述该工具的用途。基于这些新信息，当用户提出问题时，模型便可以根据工具描述判断当前任务是否需要调用该工具，以及选择哪个工具最为合适。

inputSchema 本质上是一个 JSON Schema，用于定义和约束工具输入参数的结构。以 get_forecast 为例（见图 2-25），它要求输入必须是一个 JSON 对象，并包含以下两个必填字段：

- latitude：纬度，类型为number。
- longitude：经度，类型为number。

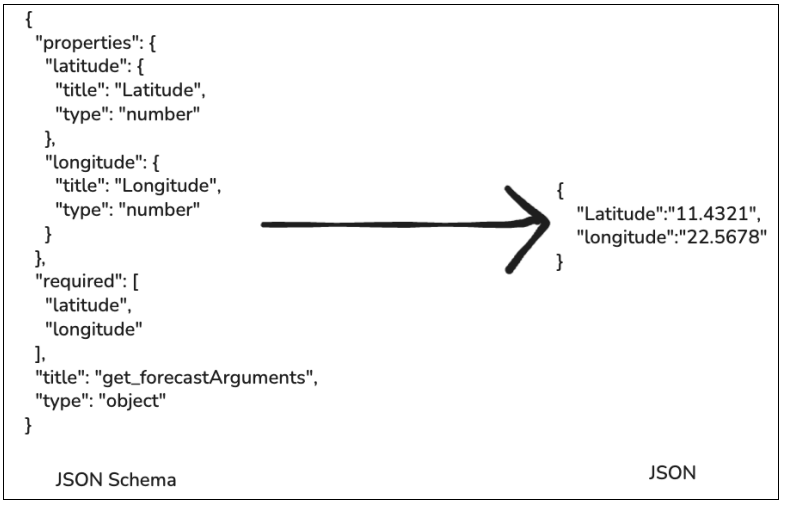


图 2-25　inputSchema 结构

这意味着，在调用 get_forecast 工具时，必须同时传入 latitude 和 longitude 两个数值参数，缺一不可。因此，可以将 inputSchema 理解为工具的输入参数规范，它规定了模型在调用工具时需要提供哪些参数，以及这些参数应满足的数据类型要求。

至此，我们已经对MCP Server启动阶段的日志进行了初步分析。上述初始化、能力声明以及工具列表返回等过程，都是在保存MCP Server配置并启动服务后自动完成的。完成这些准备工作后，MCP Server即进入待调用状态，等待MCP Host在合适的时机根据用户需求发起工具调用。

接下来，我们正式使用自定义的MCP Server。

这里仍然输入前面的示例问题：“纽约明天天气怎么样？”然后观察控制台输出的日志，分析TRAE是如何选择工具、传递参数并获得工具执行结果的。

控制台输出日志如下：

```
输入：
{
  "method": "tools/call",
  "params": {
    "name": "get_forecast",
    "arguments": {
      "latitude": 40.7128,
      "longitude": -74.006
    }
  },
  "jsonrpc": "2.0",
  "id": 3
}
输出：
{
  "type": "weather_forecast",
  "forecasts": [
    {
      "period": "今天下午",
      "temperature": "53°F",
      "wind": "9 mph 西北风",
      "description": "大部分时间晴朗。最高气温接近 53°F，下午温度将降至 51°F 左右。西北风约 9 mph。"
    },
    {
      "period": "今晚",
      "temperature": "38°F",
      "wind": "5 到 9 mph 北风",
      "description": "局部多云，最低气温约 38°F。北风 5 到 9 mph。"
    },
    {
      "period": "周日",
      "temperature": "49°F",
      "wind": "3 到 9 mph 西风",
      "description": "大部分时间晴朗，最高气温接近 49°F。西风 3 到 9 mph。"
    },
    {
      "period": "周日晚上",
      "temperature": "42°F",
      "wind": "9 到 13 mph 西风",
      "description": "大部分时间晴朗，最低气温约 42°F。西风 9 到 13 mph。"
    },
    {
      "period": "周一",
      "temperature": "51°F",
      "wind": "10 到 15 mph 西北风",
      "description": "晴朗，最高气温接近 51°F。西北风 10 到 15 mph。"
    }
  ]
}
```

从第一行“输入”日志中可以看到，模型已经根据用户问题选择了 get_forecast 工具，并为该工具生成了对应的调用参数：

```
{
     "latitude": 40.7128,
```

```
        "longitude": -74.006
    }
```

这些参数完全符合 inputSchema 的定义。这也说明，模型在生成工具调用参数时，会参考工具定义中的 inputSchema，并按照其中规定的字段名称、数据类型和必填要求组织参数。

MCP Server接收到工具调用请求后，会执行对应的工具，并将执行结果返回给TRAE。因此，第二行日志就是MCP Server执行get_forecast后返回的结果。为了便于阅读，这里展示的是经过翻译和适当简化后的输出内容。

当TRAE收到MCP Server返回的工具执行结果后，本次与MCP Server的交互流程便基本完成，相应的日志记录也在此处结束。随后，TRAE会将工具执行结果继续传递给模型，由模型对结果进行整理和总结。最终，我们在界面中看到的，就是模型基于工具执行结果生成的自然语言回答，如图2-26所示。


图 2-26　模型根据工具返回结果生成响应

### 5. MCP 小结

MCP对应的究竟是哪一段交互过程呢？

其实，对应的正是MCP Server与TRAE之间的交互。也就是说，TRAE通过模型上下文协议向MCP Server查询工具列表、发起工具调用，并接收工具返回结果。模型本身并不直接参与模型上下文协议通信，而是通过TRAE间接使用MCP Server提供的工具能力。如图2-27所示，虚线圈出的部分即为模型上下文协议的作用范围，也就是TRAE与MCP Server之间的通信过程。

MCP的核心机制可概括为两部分：工具发现与工具调用。这类似于编程中“函数的声明与使用”：在工具发现阶段，MCP Server向MCP Host暴露可用工具的名称、功能描述及参数格式，使MCP Host明确当前可调用的能力列表；在工具调用阶段，MCP Host则依据协议规范向MCP Server发起请求并接收执行结果。

必须指出的是，MCP并非“大语言模型推理协议”，它不干预模型的思考与生成逻辑，仅聚焦于外部能力如何以标准化方式接入上层应用。在架构上，模型并不直接与MCP Server通信，而是由MCP Host充当中间桥梁。MCP Host如何将工具信息提交给模型、如何引导模型进行决策，以及如何将工具结果回传，取决于各MCP Host的具体实现，这已超出MCP协议本身的范畴。

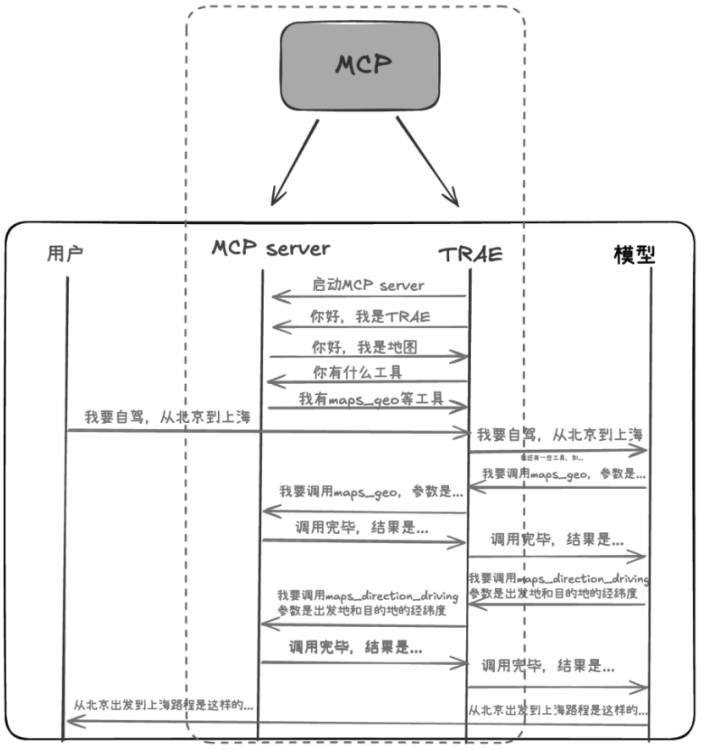


图 2-27　MCP 在整个交互流程中的位置

回归MCP的全称：Model Context Protocol，即“模型上下文协议”。此处的“上下文”超越了传统对话文本的范畴，泛指模型执行任务时需要感知的外部环境信息（如可用工具、数据源等）。MCP的核心价值，正是通过统一的协议标准，将这些异构的外部能力先聚合至MCP Host，再间接赋能于模型。

因此，MCP作为连接模型与外部环境的重要桥梁，通过标准化的工具发现与调用机制，使模型得以突破自身局限，具备间接感知并调用外部世界的能力。

### 2.4.3　MCP Host 与模型的交互

上一小节我们重点分析了MCP的底层运行过程，并明确了MCP Server与TRAE（MCP Host）之间的交互即为MCP的核心作用范围。然而，这也引出了一个关键问题：MCP Host与模型之间又是如何交互的？

为了回答这一问题，本小节将重点分析MCP Host与模型之间的交互过程，深入理解TRAE是如何将MCP Server提供的工具能力传递给模型，以及如何将模型的工具调用决策转化为实际MCP请求。

#### 1. 创建一个智能体

由于TRAE目前不支持自定义模型路由，我们无法将用户请求直接路由到自建模型或指定模型服务上。因此，为了更清晰地观察MCP Host与模型之间的交互过程，笔者使用阿里云百炼平台创建一个智能体，用它模拟TRAE在系统中的角色，即充当MCP Host。

在阿里云百炼平台创建一个智能体大致需要五个步骤。

01 创建应用时，选择智能体应用，如图 2-28 所示。

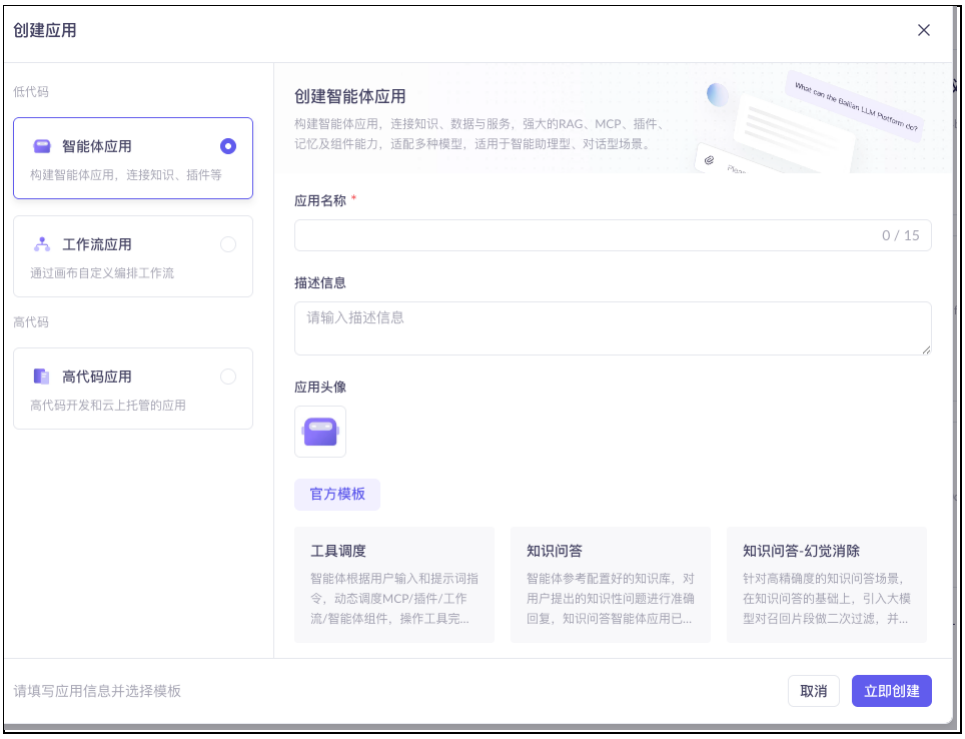


图 2-28　选择智能体应用

02 输入“天气预报智能体”作为智能体名称，然后单击“立即创建”按钮。创建后的智能体如图 2-29 所示。此时，智能体已创建完成，但仍需进一步完善相关配置。


图 2-29　天气预报智能体

03 为该智能体添加一个带有天气预报工具的 MCP Server，如图 2-30 所示。

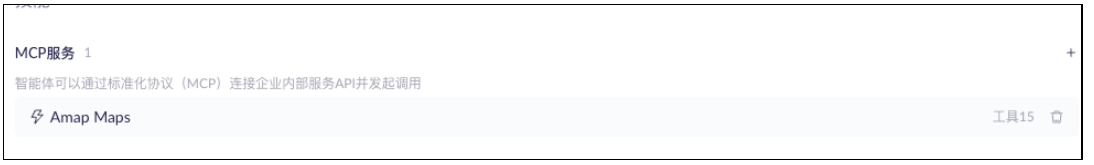


图 2-30　带有天气预报工具的 MCP Server

04 为该智能体编写系统提示词，用于约束智能体的角色、能力范围和回答方式。

```
# 角色

你是一位专业的气象数据分析师，能够查询到任意城市未来 7 天的天气。

## 技能

### 技能 1：查询天气数据

- 使用 Amap Maps 这个 MCP 服务，查询任意城市未来 7 天的天气数据。

- 获取的数据应包括但不限于温度、湿度、风速、降水量等关键气象指标。


### 技能 2：提供天气分析报告

- 根据查询到的天气数据，提供简要的天气分析报告。

- 报告应包括对未来 7 天天气趋势的总结，以及可能对用户活动产生影响的建议。

## 限制

- 仅提供与天气相关的数据和分析。

- 所有输出的内容必须准确无误，不得包含任何误导性信息。

- 所有信息均以文本输出，禁止 Markdown 格式输出
```

05 在模型配置中选择 DeepSeek-R1 作为该智能体的基础模型。

至此，通过以上五个步骤，天气预报智能体创建完成。

### 2. 与智能体对话

创建完成后，我们可以测试天气预报智能体能否正常工作。如图 2-31 所示，在对话框中输入“北京明天的天气怎么样”后，可以观察到带有天气预报工具的 MCP Server 已被成功调用，模型选择的工具是 maps_weather。工具执行完成后，模型对返回结果进行了整理，并输出了北京明天的天气情况：

```
北京明天（2025 年 11 月 24 日，星期一）的天气情况：
白天多云，夜间晴；气温在 10° C 到-3° C 之间，西北风 1-3 级。昼夜温差较大，建议适当增添衣物。
```


图 2-31　与天气预报智能体对话

### 3. 分析智能体与模型交互协议

由于本小节以阿里云百炼平台中的智能体来模拟MCP Host的角色，因此后文中提到的“智能体”均承担MCP Host的职责。

之所以能够用智能体来模拟MCP Host，是因为我们已经将MCP Server提供的工具能力配置到了智能体中。由此，智能体便可获取MCP Server所暴露的工具信息，包括工具名称、功能描述、输入参数与返回结果等。这一过程本质上是将MCP Server的工具能力注册到智能体，使其具备调用这些工具的能力。

在实际的AI IDE或Agent平台中，整体思路与此相同：MCP Host会先获取MCP Server提供的工具定义，然后将这些工具信息以模型可理解的形式提供给大语言模型。当用户发起请求时，MCP Host会将用户问题以及可用工具信息一并交给模型，由模型判断是否需要调用工具，以及应调用哪个工具。

理解这一机制后，后续的日志分析便将更加清晰。

如图2-32所示，图中左侧展示了智能体中的两个核心组件：其一为LLM，即本节选择的DeepSeek-R1推理模型；其二为Plugin（插件），即前文配置的MCP Server中的天气查询工具。在阿里云百炼平台中，工具是以插件形式接入智能体的。


图 2-32　分析智能体与模型的交互流程

图中右侧展示了各组件对应的输入和输出日志。整个Agent的输入为用户提出的问题，最终输出则是模型结合工具返回结果生成的自然语言回答。通过这组日志，可进一步观察用户请求如何进入模型，以及模型如何基于工具信息选择MCP Server并最终生成回答。

接下来，按照日志中的执行顺序，对这一交互过程进行拆解。

首先，用户输入的问题经由智能体发送给LLM（DeepSeek-R1）。

从图2-33中可以看到，发送给LLM（即DeepSeek-R1）的输入信息主要包括两部分：其一为系统消息，其二为用户消息。其中，系统消息对应于我们在智能体中配置的系统提示词，用于描述智能体的角色、能力和可用工具；用户消息则是用户实际输入的问题，即“北京明天的天气怎么样”。

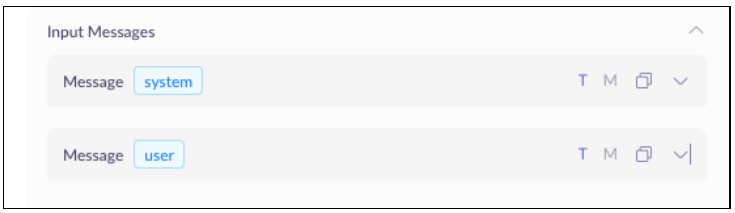


图 2-33　系统消息和用户消息

模型完成推理后，决定调用MCP Server中的maps_weather函数来查询天气信息。从模型的输出日志中可以看到，本次工具调用的函数名称为maps_weather，传入的参数是“北京”，如图2-34所示。

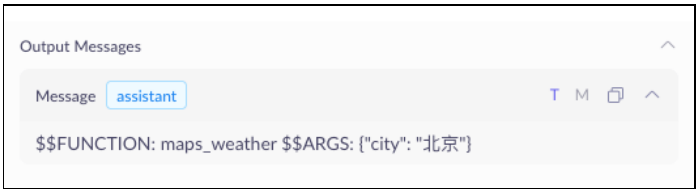


图 2-34　模型输出函数调用信息

智能体接收到模型输出的工具调用请求后，开始调用对应的插件，即MCP Server中的

maps_weather函数。从图2-35中可以看到，本次工具调用的输入参数为“北京”，工具执行后返回了北京的天气预报结果。


图 2-35　MCP Server 工具执行结果

智能体获取工具返回结果后，会将其与系统提示词、用户问题一并再次发送给模型，如图2-36所示。

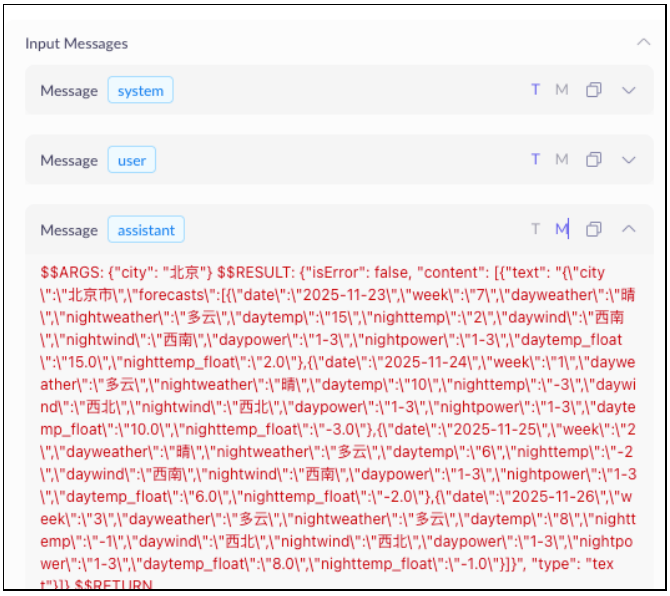


图 2-36　MCP Server 工具执行结果、用户问题以及提示词一并发给 LLM

模型对接收到的消息进行整理和总结，在生成自然语言回答后将结果发送给智能体。智能体接收到模型响应后，将最终结果呈现给用户，如图2-37所示。

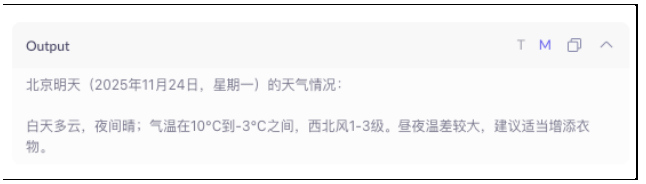


图 2-37　用户端展示最终响应结果

至此，MCP Host与大模型之间的完整交互过程已梳理完毕。

4. 小结

综上所述，MCP Host与模型的交互逻辑可简单概括为图2-38所示的流程：用户请求先进入MCP Host，MCP Host将系统提示词、用户问题以及可用工具信息发送给模型；模型判断是否需要调用工具，并返回工具调用意图；随后，MCP Host根据模型输出调用对应的MCP Server工具，并将工具执行结果再次提供给模型；最后，模型对工具结果进行整理，生成最终回答。

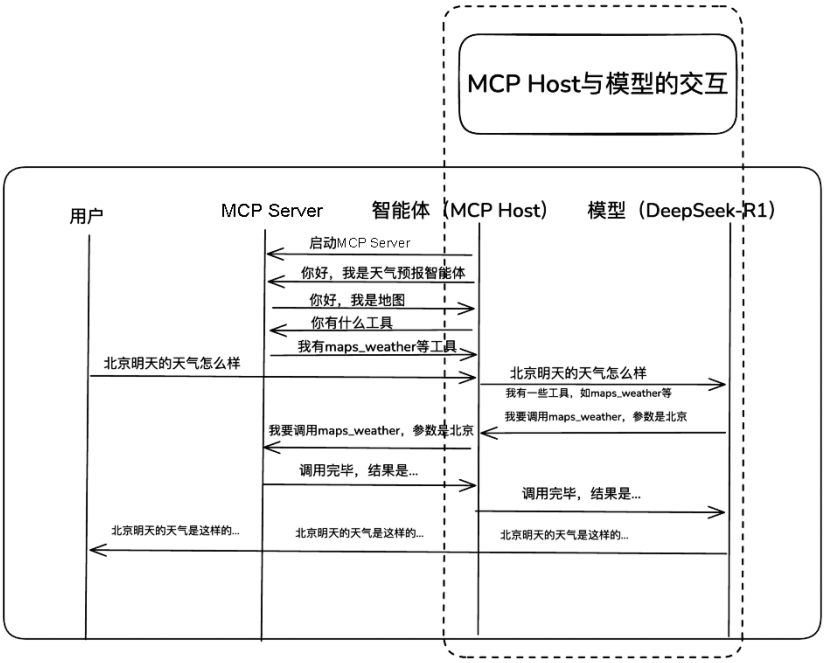


图 2-38　MCP Host 与模型的交互

## 2.5　MCP 与 Function Calling 的关系

从本节开始，我们将系统性地探讨Function Calling（函数调用）的概念与实现原理，随后分析它在整体链路中的角色，以及它与大语言模型和MCP之间的关系。

### 2.5.1　Function Calling 的基本介绍

Function Calling是OpenAI推出的一项重要能力，旨在让大语言模型（如GPT系列）能够以结构化的方式与外部函数、API以及业务系统进行交互。

在具备Function Calling能力之后，模型不再局限于生成自然语言文本，还能够判断何时需要调用某个特定函数，并自动生成执行该函数所需的参数。

从本质上看，Function Calling 改变了模型回答问题的方式：

模型并不会立即给出最终答案，而是基于当前上下文，从已知的函数集合中选择一个（或多个）最合适的函数，并构造对应的参数，以结构化数据的形式返回给应用程序。

应用程序接收到这些数据后，负责执行实际的业务逻辑或外部调用，并将执行结果再次传回模型。

在得到函数执行结果后，模型会基于新的上下文判断下一步行为：

- 可能继续调用同一个函数。
- 也可能调用其他函数。
- 或者在认为信息已经充分时，停止函数调用，并将结果整理成最终回复。

如果模型决定继续调用函数，上述流程将重复进行；如果不再需要调用函数，则进入最终响应阶段。

需要注意的是，上述描述省略了不少实现细节，但已经概括了Function Calling的核心工作机制。

### 2.5.2　Function Calling 的交互流程

在理解了Function Calling（函数调用）的工作原理之后，再结合图2-39所示的流程图来看整个交互流程，将有助于加深对Function Calling的整体理解。

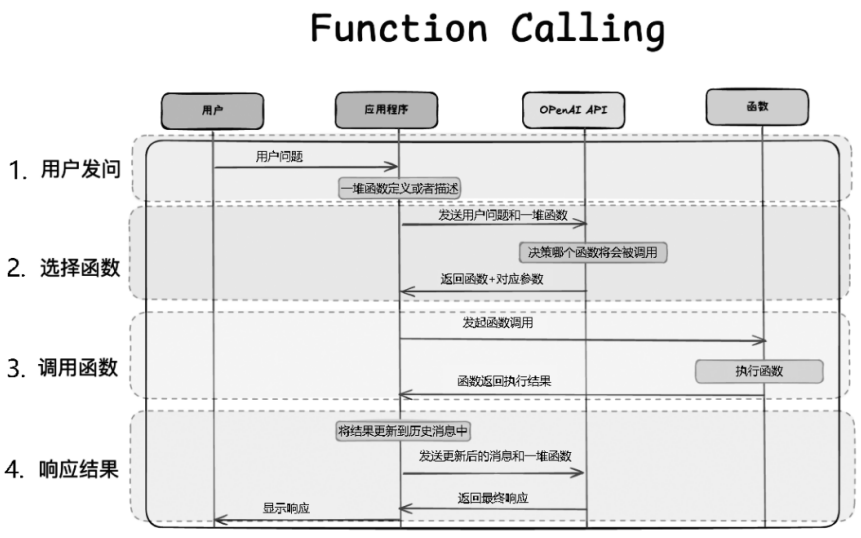


图 2-39　Function Calling 的交互流程

此处需要先对图中出现的“函数”进行统一说明：在本节中，无论是函数（Function）、工具（Tool），还是外部API，均统称为“函数”。

常见的函数类型包括（但不限于）：

- 数据库操作函数，例如读取数据、写入数据、更新记录或执行查询等。
- 外部信息获取函数，例如查询天气、获取股票行情或调用搜索服务等。
- 代码执行函数，例如完成数学计算、数据处理、脚本运行或格式转换等。
- 系统交互函数，例如文件读写、调用内部服务、触发工作流或执行自动化任务等。

在 Function Calling 的语境下，模型并不关心这些函数背后的具体实现细节，只需要知道三件事：

- 当前有哪些函数可以使用。
- 每个函数分别用于完成什么任务。
- 调用这些函数时需要传入哪些参数。

从本质上来看，Function Calling 是将外部能力以结构化函数的形式暴露给模型，使模型能够根据用户意图选择合适的函数，并生成符合要求的调用参数。

接下来，将详细阐述Function Calling的交互流程。从整体上看，该流程可以拆分为四个阶段，参与角色主要包括用户、应用程序、OpenAI API（模型）以及具体函数。

**第一阶段：用户发问**

流程的起点是用户向应用程序发起自然语言请求，例如提出一个问题或描述一项任务。应用程序在接收到用户输入后，并不会直接处理业务逻辑，而是将用户输入以及一组预先定义好的函数描述一并发送给OpenAI API。

这些函数描述的本质，是向模型声明当前可调用的能力范围，以及每项能力所需的参数规范。

**第二阶段：选择函数**

模型接收到请求后，会基于当前对话上下文、用户输入以及可用函数的描述信息进行推理，决定是否需要调用函数，以及调用哪个函数最合适。

如果模型判断需要调用函数，它不会返回自然语言答案，而是返回待调用的函数名以及该函数对应的参数（结构化数据，如JSON）。

这一步的核心在于：模型只负责决策，不负责执行。

**第三阶段：调用函数**

应用程序在接收到模型返回的函数调用信息后，会完成两项工作：一是解析函数名与参数；二是在本地或服务端执行对应的真实函数逻辑。例如，查询数据库、请求天气API、执行一段代码或与系统服务进行交互。

函数执行完成后，应用程序会将函数的执行结果返回给模型。

**第四阶段：响应结果**

应用程序会将函数的执行结果追加到历史对话中，并再次发送给OpenAI API。

模型基于最新上下文进行判断：

- 如果认为信息仍不足，可能继续发起新的函数调用。
- 如果认为任务已完成，则生成最终的自然语言回复。

最后，应用程序将模型生成的最终结果呈现给用户，整个 Function Calling 交互流程到此结束。

### 2.5.3　Function Calling 的实践

本节将对 OpenAI 官方文档中的示例进行适当改造，从代码实现的角度展示 Function Calling（函数）的完整流程。通过该实践示例，可以更直观地理解模型如何根据用户问题选择合适的函数，并按照函数定义组织调用参数。代码示例如下：

```
    from operator import index
    from openai import OpenAI
    import json
    import requests

    client = OpenAI(
        api_key="sk-7xxx", # 阿里云百炼 API Key，这里可以替换为任何你使用的大语言模型的 API Key，
对应的 base_url 也得替换
        base url="https://dashscope.aliyuncs.com/compatible-mode/v1",
    )

    MODEL NAME = "qwen-plus"


    # ====== 工具函数 ======
    def get_forecast(city: str) -> str:                  # 模拟天气 API 请求
        return f"{city}: 明天的天气是多云转晴。"

    def write_file(path: str, content: str) -> str:      # 调用系统 API 写文件
        with open("/Users/jamlee/ai/function-calling/" + path, "w", encoding="utf-8") as
f:
            f.write(content)
        return f"File written successfully to {path}"


    def get_hotel_info(city: str): # 获取酒店信息
        proxies = {
            "http": "http://127.0.0.1:7890",
            "https": "http://127.0.0.1:7890",
        }
        # 真实城市的经纬度信息
        geocode_url =
f"https://geocode.maps.co/search?q={city}&api key=6926fd2bb43bc175354646ikh1d9d5e"
        geo_res = requests.get(geocode_url, proxies=proxies, timeout=10).json()

        lat = geo res[0]["lat"]
        lon = geo_res[0]["lon"]

        # 这里可以不用请求酒店 API，只是举个酒店数据例子
        hotels = []
```

```
        hotels.append({
            "name": "北京如意酒店",
            "distance": "距离北京火车站 500 米",
            "city": city,
            "lat": lat,
            "lon": lon,
            "天气": "多云转晴"
        })
        return hotels


    # ====== Function Calling 工具描述 ======

    tools = [
        {
            "type": "function",
            "function": {
                "name": "get_forecast",
                "description": "获取天气预报",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "city": {
                            "type": "string",
                            "description": "城市名称",
                        },
                    },
                    "required": ["city"],
                },
            },
        },
        {
            "type": "function",
            "function": {
                "name": "write file",
                "description": "将文本写入本地文件中",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "path": {"type": "string"},
                        "content": {"type": "string"}
                    },
                    "required": ["path", "content"]
                }
            },
        },
        {
            "type": "function",
            "function": {
                "name": "get_hotel_info",
                "description": "获取一个城市的酒店信息",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "city": {"type": "string"}
```

```
                },
                "required": ["city"]
            },
        },
    },
]


# ====== 使用模型 ======

messages = [
    {"role": "user", "content": "帮我查一下北京明天的天气，天气好的话，帮我看一下北京的酒店，并把结果写到 hotels.txt 文件中"}
]
index = 0
# 循环工具调用
while True:
    index += 1
    final = client.chat.completions.create(
        model=MODEL NAME,
        messages=messages,
        tools=tools,            # 这句很关键！继续开放 tools
        tool_choice="auto",     # 让模型自动决定是否调用工具
    )

    final_msg = final.choices[0].message # 每次模型会选择一个 tool_call 执行
    messages.append(final_msg)
    print(f"第{index}轮模型回复：", final.choices[0])

    # 如果还有 tool_calls，就继续执行对应函数
    if final_msg.tool_calls:
        for call in final msg.tool calls:
            name = call.function.name
            args = json.loads(call.function.arguments)

            if name == "get_hotel_info":
                output = get_hotel_info(**args)
            elif name == "write file":
                output = write_file(**args)
            elif name == "get forecast":
                output = get forecast(**args)
            else:
                output = {"error": f"Unknown tool {name}"}

            # 把工具执行结果作为 tool 消息追加到对话里
            messages.append({
                "role": "tool",
                "tool_call_id": call.id,
                "name": name,
                "content": json.dumps(output, ensure_ascii=False)
            })
        # 继续 while True 循环，再问模型下一步
    else:
        # 没有更多工具要调用了，说明模型已经给出最终自然语言回复
        print(final_msg.content)
```

```
            break
```

接下来，将围绕这段代码从三个核心点进行解读，以帮助读者更深入理解 Function Calling 的实际运行机制。

### 1. Function Calling 协议本身

Function Calling 协议本身的代码如下：

```
{
    "type": "function",                              # 类型是函数
    "function": {
        "name": "get_forecast",                      # 函数名称
        "description": "获取天气预报",                 # 函数描述
        "parameters": {                              # 函数参数
            "type": "object",
            "properties": {
                "city": {                            # 参数
                    "type": "string",                # 类型是字符串
                    "description": "城市名称",        # 参数描述
                },
            },
            "required": ["city"],
        },
    },
}
```

Function Calling 同样采用 JSON Schema 方式来描述函数的参数结构，以便模型在调用函数时能够生成符合规范的参数。

以 get_forecast 函数为例，该函数所需的参数为城市名称，因此模型在调用该函数时，需要生成符合参数规范的 JSON 格式数据，示例如下：

```
{
    "city":"北京"
}
或者
{
    "city":"上海"
}
```

### 2. tools

tools本质上是一组函数定义的集合，用于向模型描述当前可调用的外部能力。这些函数的能力范围十分广泛，例如读写文件、请求外部API以及查询数据库等。

从实现角度看，Function Calling并不限制函数的具体实现形式，仅要求这些能力能够被抽象为具有明确输入和输出的函数。模型根据函数描述选择合适的函数，并生成对应的调用参数；真正的函数执行则由应用程序完成。

### 3. 循环调用

从代码流程可以看到，Function Calling通常会包含一个循环调用过程。之所以需要这一循环，是因为模型每一轮都会根据当前上下文判断下一步动作：若需要获得外部信息，则会从可用函数中

选择合适的函数，并生成调用参数；应用程序执行函数后，再将函数返回结果重新提供给模型，作为下一轮推理的上下文。

模型会基于新的上下文继续判断：是继续调用其他函数，还是已经获得足够的信息，可以生成最终回答并结束循环。

通过控制台日志，可以清楚地观察到这一过程。本次模型共进行了 4 轮响应。日志记录了每一轮响应中模型选择的工具、传入的参数，以及函数返回结果如何作为新的上下文，继续推动后续推理与任务执行。

第 1 轮模型响应：

```
    Choice(finish_reason='tool_calls', index=0, logprobs=None,
message=ChatCompletionMessage(content='', refusal=None, role='assistant',
annotations=None, audio=None, function_call=None,
tool_calls=[ChatCompletionMessageFunctionToolCall(id='call_74b1f9937b614de8b56622',
function=Function(arguments='{"city": "北京"}', name='get_forecast'), type='function',
index=0)]))
```

第 2 轮模型响应：

```
    Choice(finish_reason='tool_calls', index=0, logprobs=None,
message=ChatCompletionMessage(content='', refusal=None, role='assistant',
annotations=None, audio=None, function_call=None,
tool_calls=[ChatCompletionMessageFunctionToolCall(id='call_419175970f494eabb301e0',
function=Function(arguments='{"city": "北京"}', name='get_hotel_info'), type='function',
index=0)]))
```

第 3 轮模型响应：

```
    Choice(finish_reason='tool_calls', index=0, logprobs=None,
message=ChatCompletionMessage(content='', refusal=None, role='assistant',
annotations=None, audio=None, function call=None,
tool_calls=[ChatCompletionMessageFunctionToolCall(id='call_99d68ff5245243f7b6623e',
function=Function(arguments='{"path": "hotels.txt", "content": "[{\\"name\\": \\"北京如意
酒店\\", \\"distance\\": \\"距离北京火车站 500 米\\", \\"city\\": \\"北京\\", \\"lat\\":
\\"40.1906320\\", \\"lon\\": \\"116.4121440\\", \\"天气\\": \\"多云转晴\\"}]"}',
name='write_file'), type='function', index=0)]))
```

第 4 轮模型响应：

```
    Choice(finish reason='stop', index=0, logprobs=None,
message=ChatCompletionMessage(content='北京明天的天气是多云转晴，适合出行。我已经为您查询了北京的
酒店信息，并将结果保存到文件 `hotels.txt` 中。文件内容如下：\n\n```json\n[{"name": "北京如意酒店",
"distance": "距离北京火车站 500 米", "city": "北京", "lat": "40.1906320", "lon": "116.4121440",
"天气": "多云转晴"}]\n```\n\n 您可以查看该文件获取详细信息。', refusal=None, role='assistant',
annotations=None, audio=None, function_call=None, tool_calls=None))
```

最后的输出结果如图 2-40 所示。

从以上日志中可清晰地观察到模型在多轮循环中的决策过程：

- 在第一轮中，模型选择调用get_forecast函数，并以"北京"为参数，获取该城市的天气信息。
- 在第二轮中，模型选择调用get_hotel_info函数，同样以"北京"为参数，获取该城市的酒店相关信息。

- 在第三轮中，模型选择调用write_file函数，并传入文件路径和酒店内容，将酒店内容写入本地文件。
- 在第四轮中，模型判断当前获取的信息已足以回答用户的问题，因此不再继续调用函数，而是直接生成并输出最终结果。


图 2-40　Function Calling 实例运行结果图

### 2.5.4　MCP 与 Function Calling 的关系

既然Function Calling（函数调用）本质上也是模型调用外部能力的一种方式，那么它与MCP究竟是什么关系？

首先需要明确的是，MCP并非用于替代Function Calling。MCP的核心目标是为模型连接外部系统（工具、服务、资源）提供一整套通用协议；而Function Calling则是让模型调用预先定义好的函数。

两者并非竞争关系，它们所处的层级不同，关注点也有所不同：

（1）Function Calling是模型进行工具调用的一种实现机制，也是MCP生态中常见的一种能力实现方式。

（2）MCP的抽象层级和覆盖范围均高于Function Calling。

由此可见，模型既可以通过 Function Calling 完成工具调用，也可以通过 MCP 接入外部系统。只不过，当 MCP 作为统一标准出现后，在连接外部能力这一维度上，Function Calling 更多承担了底层调用机制的角色，其独立存在感有所弱化。

值得关注的是，Function Calling并不是一个新概念。早在2023年，OpenAI便已在GPT-4和GPT-3.5-turbo中推出了Function Calling（也称工具调用能力），并迅速成为主流大语言模型厂商（如OpenAI、Anthropic）共同支持的一项基础能力。

2024年11月，Anthropic正式开源了MCP。该协议一经发布便引发了广泛关注。此后，Claude、Qwen、GPT等多个模型体系也陆续开始支持MCP，标志着模型与外部系统的交互开始迈向标准化。

因此，Function Calling与MCP均已成为主流大语言模型生态所支持的重要能力，只是二者的职责分工和抽象层级有所不同。

当然，也存在例外情况。并不是所有大语言模型都支持Function Calling，因为Function Calling并不是模型的原生能力，而是依赖于专门的数据训练与对齐机制。若某个模型在训练阶段没有针对函数调用进行专门对齐，则很难稳定地生成符合Function Calling协议的结构化输出。

在本节的示例代码中，之所以选用qwen-plus模型，正是因为它具备成熟、稳定的Function Calling

能力。

下面通过表2-2对Function Calling与MCP进行直观对比和总结。

表 2-2　MCP 与 Function Calling 的对比

| 能　　力 | Function Calling | MCP |
| --- | --- | --- |
| 本质 | 结构化函数调用机制 | 模型与外部工具交互的协议 |
| 解决问题 | 让模型调用自定义的函数 | 让模型访问任意外部资源（文件、DB、API、服务） |
| 谁决定工具定义 | 在大模型 API 参数里传一个工具列表 | 工具定义保存在 MCP Server 中，可动态发现 |
| 是否只能调用自定义的函数 | 是（固定工具列表） | 否（MCP 支持实时发现、注册与卸载） |
| 开发成本 | 低 | 较高（需要开发 MCP Server） |
| 应用场景 | Chatbot、插件、简单 AI 功能 | Agent、IDE 助手、自动化系统、企业数据接入等 |

## 2.6　本章小结

本章围绕智能体的核心架构与交互协议展开，系统拆解了智能体的底层技术架构，完整搭建了Agent实现基础交互与工具调用的核心技术骨架，为智能体的进阶优化与落地应用奠定了坚实基础。

首先，本章明确了智能体的三大核心组成部分——模型、工具与编排层，并清晰界定了各组件的核心职责与协同逻辑：模型作为智能体的认知核心，承担推理生成与决策判断；工具作为智能体连接外部世界的载体，负责与外部系统进行交互；编排层作为核心枢纽，统筹管理流程推进与状态调控。三者相互协同，共同构成了智能体的基础架构。

其次，本章详细剖析了基础模型层的选型关键指标，从推理能力、多轮对话的连贯性与记忆能力、上下文窗口大小，到响应速度与调用稳定性、API成本与性价比、安全性与对齐能力，再到可用性与集成便利性，全面梳理了模型选型的核心考量维度，强调模型能力与工程设计的协同，是决定智能体基础表现的关键因素。

接着，本章深入剖析了MCP（模型上下文协议）的核心技术，从原理实践、底层协议设计，到MCP Host与模型的交互机制，明确了MCP在统一模型与外部系统交互标准中的核心作用。

随后，本章聚焦工具集成层，系统介绍了Function Calling与工具集成的完整方案，包括工具集成的描述、实践路径与选择逻辑，阐述了智能体实现外部工具调用的基础方法。

最后，本章介绍了MCP与Function Calling之间的关系，厘清了二者在智能体工具调用体系中的内在关联与协同逻辑，进一步梳理了智能体与外部系统之间的标准化调用链路。

总体而言，本章完整回答了“智能体由什么构成、如何选择合适的基础模型、如何实现工具集成、如何通过协议规范外部交互”的核心问题，构建了智能体的底层技术体系，为后续学习上下文工程、检索增强生成（RAG）与自动化执行等进阶内容提供了重要支撑。